ЕВГЕНИЙ КОРНИЛОВ

bhv

ПРОГРАММИРОВАНИЕ ШАХМАТ И ДРУГИХ ЛОГИЧЕСКИХ ИГР

МЕТОДИКИ И АЛГОРИТМЫ ПРОГРАММИРОВАНИЯ ШАХМАТНЫХ ПРОГРАММ

ПРОВЕРЕННЫЕ ПРИЕМЫ ПРОГРАММИРОВАНИЯ ЛОГИЧЕСКИХ ИГР

ПРИМЕРЫ НА ЯЗЫКАХ C++ И PASCAL

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

+CD

Евгений Корнилов

ПРОГРАММИРОВАНИЕ ШАХМАТ И ДРУГИХ ЛОГИЧЕСКИХ ИГР

Санкт-Петербург
«БХВ-Петербург»
2005

УДК 681.3.068
ББК 32.973
К67

Корнилов Е. Н.

К67 Программирование шахмат и других логических игр. — СПб.: БХВ-Петербург, 2005. — 272 с.: ил.

ISBN 5-94157-497-5

Рассмотрено программирование логических игр методом перебора на примере шахмат. Описываются стандартные методики создания шахматной программы, а также приемы, позволяющие разрабатывать более эффективные компьютерные логические игры. Представлены примеры использования рассмотренных методов при программировании других логических игр ("крестики-нолики", "уголки", шашки). Приведено большое количество исходных кодов программ на языках C++ и Pascal и полезных практических советов. На компакт-диске содержатся наиболее известные открытые коды шахматных программ, а также исходные тексты программ, написанных автором.

Для программистов

УДК 681.3.068
ББК 32.973

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24 07 00 Подписано в печать 09 12 04
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл печ л. 21,93.
Тираж 3000 экз. Заказ № 711
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр , 29

Санитарно-эпидемиологическое заключение на продукцию
№ 77 99.02.953.Д 006421.11 04 от 11.11 2004 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 5-94157-497-5

Содержание

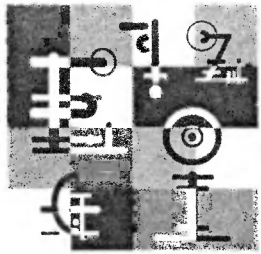
Введение

"Неприлично рассуждать о шахматном
программировании, не имея
работающей программы"
Кен Томпсон

Машина играет в шахматы — разве это не удивительно! Шахматы считаются самой интеллектуальной игрой, и вдруг в нее играет машина. Когда-то подобное казалось невероятным, но сейчас к этому привыкли. Такое впечатление, что изобретен искусственный интеллект. На самом деле, это, конечно, не так. Машина играет настолько умно, насколько ее научит человек. Есть простые игры, типа "Крестиков-ноликов", с ограниченным количеством ходов. В них машину ничему не нужно учить, она легко просчитывает ситуацию до конца и играет идеально. В шахматах количество ходов намного больше, и дерево перебора огромно. До конца просчитать ситуацию, кроме некоторых редких случаев, просто немыслимо. Как же машина играет и выбирает лучший ход? Об этом и пытается рассказать эта книга. Ее условно можно поделить на две части. В первой дается история развития шахматных программ и алгоритмов, далее описаны современные стандартные методики оптимизации перебора, приведена сравнительно несложная шахматная программа. Вторая часть рассказывает о более изощренной технике выборочного поиска и отсечении статической оценкой. Эта часть более трудна для понимания, особенно для не знакомых с этой техникой. Там нет готовых рецептов, ясны лишь направления, куда нужно двигаться. Все современные программы в той или иной мере используют эту технику. Даже сугубые консерваторы, не верящие ни в какой forward prunnig (отсечение ветвей без их предварительного рассмотрения), тем не менее, используют эту же технику в своих форсированных вариантах, которые считают только взятия. Автор надеется, что книга будет полезна всем интересующимся данной тематикой. От читателя не требуется никакой специальной подготовки, только знание какого-либо алгоритмического языка. Книга может быть полезна и более продвинутым в данной области. Они могут найти в ней некоторые идеи или просто альтернативный взгляд на

некоторые вещи. Хочется подчеркнуть, что все описываемые методики известны (некоторые уже давно) и (в той или иной мере) используются во многих программах, даже чемпионов среди компьютерных программ. Автора очень интересует обмен мнениями по данной проблеме. Замечания или просто обсуждение того или иного алгоритма, а также иной взгляд на решение затронутых проблем можно прислать по адресам: **e_k@sbor.net**, **kevgeniy@yandex.ru**.

Приятного чтения!



ГЛАВА 1

Общие сведения

История развития шахматных программ

Когда же в первый раз машина заиграла в шахматы? На этот вопрос трудно ответить, наверное, в 1769 году. Венгерский инженер барон Вольфганг фон Компелен создал машину, способную играть в шахматы (рис. 1.1). Она была предназначена для развлечения королевы Марии-Терезии. Машина, действительно, неплохо играла — внутри ее сидел сильный шахматист, который и делал ходы. История примечательная и, в некоторой степени, актуальна и в наши дни. Что программист вложит в машину, то она и играет. Гарри Каспаров, например, понимает этот тезис буквально. Он утверждает, что в его втором, решающем матче с DeepBlue (первый он выиграл), начиная со второй партии, в игру машины вмешивался человек и компенсировал неспособность машины понимать позиционную игру. Оставим этот вопрос на совести корпорации IBM и Гарри Каспарова. Я лично нисколько не сомневаюсь, что Каспаров может выиграть еще один матч с DeepBlue (если этот матч возможен), если, конечно, ему повезет.

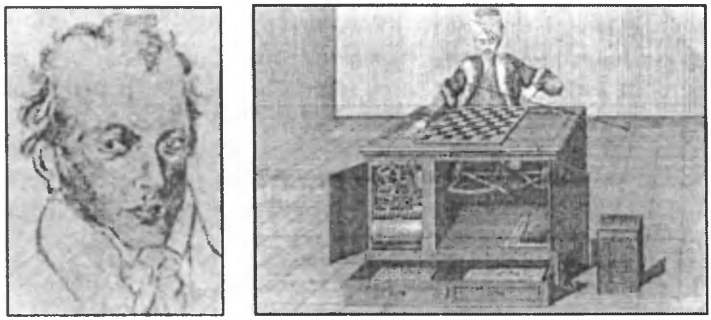


Рис. 1.1. Вольфганг фон Компелен и "играющий турка"

В 1951 году Алан Тьюринг (рис. 1.2) написал алгоритм, при помощи которого машина могла бы играть в шахматы. Только в роли машины выступал сам изобретатель. Этот нонсенс даже получил название — "бумажная машина Тьюринга". Автору требовалось более получаса, чтобы сделать один ход. Алгоритм был довольно условный, и сохранилась даже запись партии, где "бумажная машина" Тьюринга проиграла одному из его коллег.

Рис. 1.2. Алан Тьюринг (1912—1954)

Нужно отметить, что это была не единственная попытка изобрести алгоритм шахматной игры. Особенно примечательна история Ботвинника. Сам факт, что Ботвинник, будучи чемпионом мира и автором многочисленных работ по теории шахмат, воспитавшим целую плеяду чемпионов мира, верил, что существует алгоритм этой игры, заслуживает внимания. Но итог печален. Ботвинник как программист более 20 лет занимался созданием шахматной программы. Его программа так никогда и не заиграла. Кто-то даже сказал по этому поводу: "Ботвинник только думает, что он знает, как он думает". Алгоритм, возможно, существует, но он чрезвычайно сложен и, может быть, так никогда и не будет найден. Шахматы — не простая игра. В ней на любое правило всегда найдется исключение.

Примерно в это же время, в 1951 году, математик Клод Шеннон пишет свою первую статью о программировании шахмат. Он писал: "Хотя, возможно, это и не имеет никакого практического значения, сам вопрос является, теоретически, интересным, и будем надеяться, что решение этой проблемы послужит толчком для решения других проблем аналогичной природы и большого значения". Шеннон отмечает теоретическое существование лучшего хода в шахматах и практическую невозможность найти его. Он описывает две стратегии поиска лучшего хода, обе основываются на эвристической функции оценки конечных точек:

- тип А — перебор всех возможных ходов на фиксированную глубину, с вызовом в конце оценочной функции (т. к. невозможно осуществить перебор до конца);

- тип B — выполняет только выборочное расширение определенных строк, используя накопленные шахматные знания, чтобы подрезать неинтересные ветви. Например, программа Тьюринга только расширяла строки, включая взятия.

Все это выглядит несколько грубо и схематично с сегодняшних позиций, но принцип двух подходов (или их синтеза) сохраняется и в настоящее время. Шеннон писал, что тип B, несомненно, лучший, потому что больше похож на мышление человека. В чистом виде эти методы сегодня не используются, но, в зависимости от предпочтений автора конкретной программы, делается уклон в сторону одного или другого. Следующий шаг в развитии шахматных программ был сделан в ядерной лаборатории Лос-Аламоса в 1952 году, на компьютере MANIAC1 (11 кГц — тактовая частота, 600 слов память !!!). Для него была написана шахматная программа игры на доске 6×6, без участия слонов. Машина считала 4 полухода и тратила на это 12 минут. Известно, что этот компьютер сыграл одну партию против сильного шахматиста, она длилась 10 часов и закончилась победой шахматиста. Еще одна партия была сыграна против недавно научившейся играть в шахматы девушки. Машина победила на 23 ходу. Сейчас это выглядит смешно, но для своего времени это было целое достижение.

1957 год — на IBM704 (42 кГц, 7 Кбайт — память) был реализован тип B программы на полной доске, с участием всех фигур. Машина считала 4 полухода за 8 минут. Уровень игры — любительский.

1962 год — Ньюэл, Саймон и Шоу открывают алгоритм, получивший название alpha-beta. Он давал результат не хуже, чем полный перебор, не исследуя все варианты. В нем не требовалось никаких специальных шахматных знаний, и он мог быть применен для решения любой переборной задачи. Суть алгоритма, вкратце, в том, что в каждой строке игры, для белых и для черных отслеживались их максимальные результаты, и если в некоторой точке черные уже получили результат, сравнявшийся с максимумом белых, достигнутым до этого, то дальше перебирать нет смысла. Когда перебор вернется в точку, где был достигнут максимум белых, результат, все равно, будет отвергнут, т. к. у белых в этой точке есть уже не худший ход. Подробно эта техника будет описана далее. Наиболее известная работа по этой теме — статья Кнута и Мура в 1976 году. Считается, что этот алгоритм ранее открыл советский ученый Брудно. Хочется добавить, что основная заслуга этих людей в том, что они были первыми. Этот алгоритм великое множество раз открывался заново людьми, не подозревавшими о его существовании. Он очевиден. С его помощью можно считать значительно быстрее, но все еще недостаточно быстро. Тем не менее в основе всех современных шахматных программ лежит одна из усовершенствованных версий данного алгоритма. Примерно до 1973 года все шахматные программы были типа B. Они главным образом основывались на генераторах правдоподобных перемещений, отсекая статической оценкой малоправдоподобные. С появлени-

ем более мощных процессоров программисты стали переключаться на тип А. Первыми ласточками были Teach и Chess4. Это были программы "грубой силы". Как только они достигали глубины 5 полуходов средней стадии игры, они начинали побеждать в соревнованиях с программами типа В.

1975 год — Роберт Хьят начинает разрабатывать CrayBlitz, который долгое время был самой быстрой шахматной программой и в период с 1983 по 1989 гг. — мировым чемпионом среди шахматных программ. Он искал примерно 40—50 тыс. позиций в секунду (в 1983 году), что для своего времени было большим достижением.

1977 год — Томпсон и Кондон из лаборатории Bell создают первый специализированный шахматный компьютер. Дело в том, что скорость просчета зависит, помимо всего прочего, и от того, насколько программа задерживается в каждой позиции. Наиболее разорительными по времени являются генераторы ходов, функция оценки позиции, детекторы шахов и атаки и пр. Возникла простая идея, не дожидаясь увеличения мощности процессоров, реализовать эти части программы аппаратно. Иногда вещи, которые довольно сложно реализовать программно, очень просто решаются на уровне регистров процессора. Лучшие компьютеры того времени могли исследовать до 5 тысяч позиций в секунду, машина же Кена Томпсона, которую назвали Belle, обрабатывала 180 тысяч в секунду. Belle мог продумывать позицию на 8—9 полуходов вперед, что ставило его на уровень мастера. Он побеждал на многих компьютерных шахматных турнирах в период с 1980 по 1983 год. Несмотря на то, что специализированное железо на порядок быстрее, чем обычная машина, программа CrayBlitz на сверхсовременной тогда машине CrayXMPs (стоимостью 60 миллионов долларов) все равно играла лучше. В России подобные попытки были предприняты даже немного раньше: в конце 1960-х годов одна из отечественных машин М-20 была немного модернизирована, в нее были вставлены специализированные шахматные команды. В середине 1980-х годов Ганс Берлинер, компьютерный ученый университета Карнеги-Меллон, усовершенствовал компьютер Кена Томпсона. Был создан HiTech с параллельно действовавшими 64 шахматными чипами нового поколения. CrayBlitz, однако, все равно у него выиграл на всемирном чемпионате среди компьютеров в 1986 году. Вскоре после этого ученики Берлинера, студенты Хэй, Кемпбелл и др., разработали свой собственный специализированный шахматный компьютер Chip Test (он считал только материал), а позже и DeepThought (предшественник DeepBlue). DeepThought содержал 250 специализированных чипов, два процессора, был способен просчитать 750 тыс. позиций в секунду и вести перебор на 10 полуходов. Среди его достижений была победа над гроссмейстером Джудит Полгар. Однако Каспаров выиграл у экспериментальной шестипроцессорной версии DeepThought, которая искала более 2 млн позиций в секунду. Позже эта же группа, уже в расширенном составе, разработала DeepBlue. Таким образом, DeepBlue, скандально выигравший после модернизации второй матч у Каспарова (рис. 1.3), начинался как студенческая разработка.

Интересно было группе способных студентов пробовать свои силы, да и тема диплома была отличная. После создания DeepThought группу разработчиков заметил маркетинговый департамент фирмы IBM и обратился к ней с предложением. Результатом стали DeepBlue и DeepBlue2. Таким образом, DeepBlue — это результат более чем десятилетней работы очень способной группы студентов, при поддержке гроссмейстеров и миллионов IBM. Эта машина, помимо специализированного компьютерного железа, использовала распараллеливание alpha-beta перебора.

Рис. 1.3. Гарри Каспаров в решающем матче с DeepBlue

Оценочная функция DeepBlue не использует ни баз данных, известных из истории вариантов (кроме баз данных эндшпилей), ни генетических алгоритмов, ни нейронных сетей. Вместо этого, DeepBlue оценивает аппаратно приблизительно 6 тыс. специфических для шахмат показателей. Как производилась настройка этого множества показателей? В основном, вручную. Гроссмейстер Джон Бенджамен играл с машиной, пока не обнаруживал, что она неверно оценивает ту или иную позицию. После этого весь коллектив обсуждал проблему и изменял некоторые коэффициенты в оценочной функции. За большие деньги можно заниматься и не таким делом!

Немного об архитектуре DeepBlue. Основу машины представляет собой сервер RS6000SP. В сервере имеется 32 процессора, один объявлен главным. К каждому процессору подключено до 16 специализированных шахматных чипов — итого 512 на всю машину. Первые 4 полухода перебираются главным процессором. После этого он передает работу остальным 31, и они перебирают еще 4 полухода. Программа написана на обычном языке C, и использует алгоритм NegaScout. Предусмотрены выборочные продления некоторых вариантов, так что 4 полухода могут значительно растянуться. Некоторые критерии для продлений в DeepBlue:

- если значение оценочной функции для данного хода значительно выше, чем для остальных;

- по угрозе;
- по влиянию;
- проходные пешки;
- некоторые варианты, выходящие за границы alpha-beta (видимо, это так называемая сингулярная эвристика продлений, или ожидание второй отсечки за beta).

Система этих продлений описана несколько туманно. Такое впечатление, что продлевается все мыслимое и немыслимое. После основных процессоров работу перехватывают специализированные шахматные чипы. В них используются разработанные Кеном Томпсоном аппаратные средства для генерации перемещений, и осуществляется их сортировка. Аппаратно реализован детектор шахов (он работает один такт), контроль X-rays, вилок, перегрузок фигур и пр. Коэффициенты оценочной функции загружаются с диска, что дает возможность вести их настройку. Весь комплекс обрабатывает миллиард или более позиций в секунду.

Но вернемся к шахматным программам. Ричард Лэнг, пишущий исключительно на ассемблере (включая графическую часть!), сделал очень сильную программу выборочного поиска Genius. Любителям шахмат эта программа известна по шахматным компьютерам "Мефисто" с мощным специализированным процессором, в память которых зашит Genius. После проигрыша Каспарова в матче из двух партий (блиц) в 1995 году большинство ведущих гроссмейстеров потребовало запретить участие компьютеров в турнирах людей. Genius и сегодня участвует в мировых компьютерных первенствах и стабильно держит 5—6 место. Из особенностей Genius известно, что он просматривает шахи (особенно если у противника есть единственный ответ) на очень большую глубину.

Нужно отметить Hiarcs Марка Униаке, играющего в сильные позиционные шахматы. Hiarcs долгое время был мировым чемпионом и сегодня входит в тройку лучших программ.

Из отечественных программ нужно отметить "Каиссу" (Битман, Арлазоров, Адельсон-Вельский). Это была революционная для своего времени программа (1970-е годы). Она осуществляла с помощью итеративных углублений перебор на 7 полуходов плюс довольно сложные форсированные варианты. Использовались фильтры ходов. На обдумывание одного хода уходило примерно полтора часа. В матче с американцами программа выиграла буквально в самом начале партии, найдя очень глубоко форсированный мат. Американцы не могли поверить, что программа могла считать так глубоко, но им показали, что вся комбинация состоит из форсированных ходов: взятий и шахов. К сожалению, развития эта победа не получила, и самые сильные программы стали появляться за рубежом. Исключение составляет "Мираж" (1992, 1998, Рыбинкин-Шпеер). Он по сей день остается чемпионом России. Чемпионаты, правда, уже не проводятся. Это очень интересная

программа выборочного поиска (подробностей я, к сожалению, не знаю). Известно, что "Мираж" считает не все, но достаточно глубоко. Он участвовал в мировых первенствах, но первого места (как и последнего) не занимал. Играет он очень интересно, но стабильно сильную игру не показывает. Известны его разгромные победы над Fritz и Hiarcs. В настоящее время Сергей Марков, автор SmarThink, утверждает, что его программа играет сильнее, чем "Мираж". Я лично нисколько не сомневаюсь, что некоторые варианты она просчитывает лучше, но надо еще доказать, что она стабильно сильнее играет. "Мираж" все-таки очень удачная программа.

Из развития шахматных алгоритмов хочется отметить появление эвристики пустого хода (NullMove). Идея состоит в том, что в большинстве случаев, если мы пропускаем ход (передаем очередь хода противнику и перебираем на меньшую глубину, на два или более полухода), то теряем, а не приобретаем. Если после такого пропуска наша оценка все еще больше beta, то перебор можно прервать — это неинтересная часть дерева игры. Точное время появления этой эвристики неизвестно. В 1986 году на мировом первенстве в Германии Дон Бил участвовал с программой, которая использовала технику недействительного перемещения в его статическом поиске. В течение турнира Франц Морш заинтересовался этой идеей и спустя несколько лет осуществил ее в программе Fritz. Ускорение получалось очень значительное, а результат очень точным. С середины 1990-х годов недействительное перемещение стало использоваться практически во всех шахматных программах, (правда, подход к его применению очень различный).

Следующее усовершенствование было сделано Эрнст Хайнц в его программе DarkThough и называлось Futility pruning и Razoring. Это выборочное отсечение определенных пограничных ветвей дерева перебора, без их предварительного исследования. Отсекаемая позиция должна удовлетворять определенным условиям. Данные методы не вносят практически никакой погрешности, но есть люди, не признающие их. Они никак не могут понять, что считать надо очень глубоко, а рост дерева выражается экспоненциальной зависимостью и "засадит" любой мыслимый и немыслимый процессор.

Некоторые приемы программирования

Поклонники языков C и Паскаль могут пропустить этот раздел. В нем рассказывается о некоторых тривиальных приемах программирования, которые, тем не менее, полезно знать или освежить в памяти перед чтением книги. Это, во-первых, рекурсия.

Рекурсия

Кто получил специальное образование (повезло с преподавателем) или сам занимался этим вопросом, тот, несомненно, знает преимущества рекурсив-

ных алгоритмов. В обычных учебниках им отводится до смешного мало места или не отводится вообще. Популярно следующее замечание: "Любая задача, решенная с помощью рекурсии, может быть решена и обычным способом". Это так, если моделировать стек программы. Некоторые задачи, тем не менее, практически невозможно решить без рекурсии. Объем кода неимоверно возрастает, и без всякой пользы.

Рекурсия — это вызов из функции самой себя. Это допустимо даже в Бейсике. Функция вызывает сама себя, при этом все переменные (стековые), кроме статических, задаются заново. Если у нас есть некоторая функция `Foo` и переменная `dummy`, то `dummy` всегда новая при следующем вызове функции.

Приведем пример "пустой" рекурсии. Функция вызывает сама себя и не совершает никакой полезной работы. Единственное, нужно ограничить количество рекурсивных вызовов некоторым реальным числом:

```
void Foo(int callNumber)
{
if(callNumber <= 0) return;
Foo(callNumber-1);
}
```

Пример первого вызова функции:

```
Foo(10);
```

Функция `Foo` вызывает себя 10 раз, и программа завершается. Значение `callNumber` уменьшается при каждом вызове, и если равно нулю, дальнейших рекурсивных вызовов нет. Переменная `callNumber` обозначает глубину рекурсии и обычно называется `depth`. Ключевое слово **`void`** обозначает, что функция не возвращает никакого значения. В языке C нет специально зарезервированного слова `function`. Оно подразумевается. В C++ мы можем не писать **`void`** (его можно опустить и в некоторых версиях языка C). Тогда будет подразумеваться, что функция возвращает целочисленный тип со знаком — **`int`**. Размерность типа **`int`** равна разрядности регистра аккумулятора (`EAX` или `AX`) процессора. Для 16-разрядных программ это 16, для 32-разрядных — 32 бита. Напишем:

```
Foo(int depth)
  {
if(depth > 0) Foo(depth-1);
return 0;
  }
```

Можно, без привычки, не сразу и догадаться, что перед нами сигнатура функции. Возвращаемое значение 0, в данном случае, неинформативно. Некоторые компиляторы позволяют опускать его. Это ненужные тонкости. Желательно, если функция не возвращает значения, писать явно **`void`**.

Это же относится и к списку аргументов функции. Если аргументов нет, то в C нужно обязательно указать:

```
void Foo(void);
```

В C++ слово `void` можно опустить и просто написать:

```
void Foo();
```

Давайте заставим нашу рекурсивную функцию совершить какую-нибудь полезную работу. Например, вычислить возведение числа в степень.

```
int Foo(int value, int n)
  {
if(n == 0) return 1;
return value * Foo(value,n-1);
  }
```

Описанная функция не вычисляет отрицательные степени. Это как раз неудачный пример рекурсии, и задача может быть легко решена без нее:

```
int Foo(int value, int n)
  {
int tmp = 1;
for(;n > 0; n--)
tmp = tmp * value;
return tmp;
  }
```

В некоторых случаях рекурсия, тем не менее, оказывается очень полезной. Вот пример.

Ханойские башни

Имеется 3 стержня: A, B и C. На стержень A нанизано n дисков так, что диск с меньшим диаметром всегда лежит поверх диска с большим. Говоря другими словами, на стержне A пирамида из дисков, сужающаяся вверх. Требуется переместить все диски на стержень B. Стержень C можно использовать в качестве вспомогательного. Основное условие состоит в том, что диск с большим диаметром не должен находиться поверх диска с меньшим.

Как это сделать? Предположим, что мы умеем перекладывать пирамиду из n−1 дисков. Как, мы не знаем, но умеем. Тогда нам нужно переместить первые n−1 дисков с A на C. Затем перенести самое большое кольцо с A на B. Оно окажется внизу. Затем n−1 дисков нужно переместить с C на B. Готово. Мы только задали закон развития алгоритма. Как он будет выполнять-

ся в действительности — очень любопытно посмотреть по распечатке результата. Если дисков много, задача получается очень головоломной.

```
#include <stdio.h>
void Ring(int n, char a, char b, char c)
{
if(n <= 0) return;
Ring(n-1,a,c,b);
printf("диск %d   %c -> %c \n",   n,a,b);
Ring(n-1,c,b,a);
}

//основная программа
int main(void)
{
Ring(3,'a','b','c');
return 0; //код завершения программы
}
```

Мы вызываем функцию `Ring`, чтобы переложить только 3 кольца. Если увеличить это число, то процедура перекладывания будет очень замысловатой.

Директивой `#include` в программу включен заголовочный файл `<stdio.h>`. Он содержит стандартные библиотеки ввода/вывода языка С, и описание функции `printf` находится именно в нем. Текст программы поступит сначала на вход препроцессора языка С. Он просмотрит данный файл и вставит программный код файла stdio.h точно в то место, где он объявлен директивой `#include`. Препроцессору все равно, какой файл вставить в это место. Было бы указано его название, и был бы он на диске в доступном каталоге.

Когда код программы поступит на вход компилятора, он будет значительно больше приведенного, но это нас не волнует, в данном случае. Компилятор при трансляции кода делает один проход, поэтому он должен встретить предварительное описание функции `printf` перед тем, как вызов данной функции встретится в программе. Иначе компилятор не будет знать, что это за функция, и каков у нее список аргументов. Наша небольшая программа содержится в файле с расширением с. При трансляции кода компилятором получится объектный файл с расширением obj. Он должен быть связан компоновщиком (например, `link`) с другими объектными файлами, если они есть в проекте. В результате мы получим выполняемый файл с расширением exe, который и является исполняемой программой. В нашем случае компоновщик должен будет найти библиотечный объектный файл (name.obj или name.lib) и взять оттуда сам код функции `printf`. В файле stdio.h содержится только ее предварительное (forward) описание. Мы можем и не включать файл stdio.h, а написать в начале программы:

```
extern printf(...);
```

Сигнатуру функции `printf` можно посмотреть в вашем файле stdio.h. Это обычный текстовый файл. Сигнатура функции `printf` непростая, и лучше оставить все это на совести разработчиков компилятора.

В проекте может быть несколько файлов с расширением c. Компилятор для каждого создаст свой объектный файл. Единственное, что нужно помнить, что это функция `main` является точкой начала исполнения программы, и она должна быть одна в каком-либо файле. В каком — безразлично.

Зачем нужно много файлов? Это очень удобно. Некоторые файлы могут содержать пользовательские библиотечные функции, и их можно включать в разные проекты. Как вызвать из одного файла функцию, содержащуюся в другом? Для этого нужно воспользоваться директивой **extern**. Если мы хотим ограничить область видимости некоторой функции одним файлом (или переменной), мы должны перед ее сигнатурой написать ключевое слово **static**. В нашем примере функция `Ring` и основная функция `main` были в одном файле. Давайте поместим функцию `Ring` в отдельный файл. Тогда основной файл программы будет содержать только функцию `main`. Перед использованием функции `Ring` нужно написать:

```
extern Ring(int n,char a, char b, char c);
```

Когда компилятор встретит это описание, он ничего не будет знать об устройстве функции `Ring`, потому что каждый файл компилируется отдельно. Он будет только знать сигнатуру данной функции, чтобы осуществить ее вызов. При этом компилятору не важны названия аргументов, важен их тип и порядок. Мы могли бы написать:

```
extern Rint(int,char,char,char);
```

Компоновщик после компиляции установит необходимые связи. Если он не найдет функцию `Ring` ни в одном файле проекта, он сообщит об ошибке компоновки. Директива **extern** может использоваться и для объявления внешних переменных. Мы можем написать в одном файле:

```
extern int dummy;
```

А сама переменная `dummy` будет находиться в другом файле.

```
int dummy = 0;
```

Каждый раз использовать директиву **extern** не всегда удобно. У нас может быть большой проект и много файлов. Мы можем собрать все внешние объявления (**extern**) в одном или нескольких файлах с расширением h и включить их директивой `#include`. Давайте создадим для нашего случая простой файл (*.h) и включим туда описание внешней функции `Ring`.

```
//файл MyHeader.h
#idndef MY_HEADER
```

```
#define MY_HEADER
extern Ring(int,char,char,char);
#endif
```

Файл main.c тогда будет выглядеть так:

```
#include "MyHeader.h"
int main(void)
{
Ring(3,'a','b','c');
return 0;
}
```

Сама функция `Ring` будет находиться в третьем файле с расширением c, название которого совершенно не важно. Обратите внимание, что в файле MyHeader.h содержится только описание функции `Ring`. В каком она файле — не указано. В одном из файлов проекта. Директиву `extern` можно опустить для описания внешней функции в файле (*.h). Это подразумевается. В языке C значком # начинаются директивы препроцессора. Это не компилятор, это отдельная программа, которая совершает предварительную обработку текста программы. Компилятор не встретит в программе уже ни одной директивы, начинающейся значком #. В нашем файле MyHeader.h есть несколько директив препроцессора. Их можно опустить, и тогда файл будет выглядеть:

```
//////////////////////////////////
void Foo(int,char,char,char);
//////////////////////////////////
```

Зачем они нужны? Дело в том, что данный файл может быть включен в текст программы несколько раз. Как это может быть? Допустим, у нас несколько заголовочных файлов (*.h). В один файл может быть включен другой заголовочный файл, а потом оба, по ошибке, включены в третий заголовочный файл. Это происходит довольно часто. Как этого избежать? Для этого мы используем директиву препроцессора, проверяющую условие. Она аналогична оператору `if` языка программирования, только пишется `#if`, например:

- директива `#ifdef` — если определена;
- директива `#ifndef` — если не определена;
- директива `#define` вводит константу (символическое имя препроцессора).

В нашем случае мы написали:

```
#define MY_HEADER
```

MY_HEADER — это вымышленное символическое имя препроцессора. Если файл MyHeader.h был включен препроцессором хоть один раз при трансляции кода для конкретного файла, то это имя препроцессор запомнил. Если встретится повторное включение файла MyHeader.h, то препроцессор проверит условие:

```
#ifndef MY_HEADER
```

Так как оно ложно, весь текст до директивы `#endif` будет пропущен, и повторного включения файла MyHeader.h не произойдет. Мы должны понимать, что препроцессору все равно, что включать и сколько раз. Если мы напишем 10 раз `#include "MyHeader.h"` и не примем мер, то препроцессор включит этот файл 10 раз в место, указанное директивой `#include`. Ошибку выдаст компилятор. Он 10 раз встретит предварительное описание функции `Ring` и откажется дальше компилировать. Если разобраться, в этом нет ничего сложного. Это обычные макросы. Нам может быть неудобно много раз выписывать один и тот же код, и мы можем его оформить в виде макроса. Потом в тексте программы, вместо того чтобы нудно выписывать одно и то же, мы можем просто написать имя макроса, и препроцессор расширит его нужным кодом. Например, у нас есть массив из 10-ти чисел, и нам нужно его много раз обнулять или присваивать какие-либо значения. Мы можем написать один раз макрос, а потом использовать в программе имя макроса, например:

```
//инициализация массива data[10]
#define INIT_ARRAY \
data[0] = 1; data[1] = 2;\
data[3] = 4;.... data[9] = 512;
```

Содержимое файла (*.h) вставляется точно так же препроцессором, и мы должны написать в этом файле то, что хотим видеть в этом месте программы. Как некоторый рекурсивный казус, мы можем включить в файл MyHeader.h в самом конце сам этот файл `#include "MyHeader.h"`. Что произойдет? Конечно, препроцессор должен заметить ошибку, иначе он будет расширять файл MeHeader.h самим собой, пока хватит памяти, или впадет в бесконечный цикл.

Задача о ферзях

Найти способ расстановки 8 ферзей на доске 8 × 8 так, чтобы они не били друг друга. Данная задача решается методом перебора с помощью рекурсии. Прежде чем поставить на доску очередного ферзя, нужно проверить, не бьет ли его какой другой ферзь. Если мы будем каждый раз сканировать все возможные перемещения всех ферзей, то это достаточно неэффективно. Можно ускорить проверку, если обратить внимание на один факт. Ферзь, как

известно, ходит во всех направлениях, по всем диагоналям, вертикалям и горизонталям. С вертикалями и горизонталями все понятно: если был хоть один ферзь с координатой X или Y такой, как у нашего ферзя, то данное поле бьется. Как быть с диагоналями? Если диагональ восходящая (/), то сумма X и Y для всех клеток диагонали одна. Если нисходящая (\), то разность X — Y для всех клеток одинакова. Мы можем иметь некоторые множества, обнуленные в начале вычисления:

1. Множество X.
2. Y.
3. X+Y.
4. X−Y.

Чтобы узнать, бьется ли данное поле каким-либо ферзем, мы должны проверить клетки массивов (множеств) 1, 2, 3, 4. Для массива 1 это будет клетка с индексом X, для массива 2 — Y, для 3 — (X+Y), для 4 — (X−Y+8). Все эти клетки должны быть нулевыми. Если хоть одна клетка 1, то поле бьется. Если мы поставили ферзя в какую-то клетку, то в соответствующие ячейки множеств должны записать 1. Если отменили ход — должны восстановить старое значение ячеек. Массив 4 имеет добавочный индекс 8 потому, что X−Y не может быть меньше 0, а в языке C индексация массивов от 0. Если мы поставили последнего ферзя, то конец перебора и вывод результата. Так как мы имеем дело с частным вырожденным случаем, то вместо 4 множеств можно использовать 3, а координату X увеличивать пошагово. Это гарантирует нам, что не будет двух ферзей с одинаковой координатой X. Старые значения множеств можно также не сохранять и не восстанавливать, т. к. если мы поставили ферзя на клетку, то это гарантия того, что в соответствующих ячейках массивов были нули. Вы можете попробовать написать эту программу сами. Это довольно занятно. Можно попытаться получить все возможные размещения ферзей.

Я же хочу обратить внимание на некоторый аспект этой задачи применительно к программированию шахмат. Допустим, нам нужно определить, бьется ли поле (X,Y) фигурами противника. Как это сделать? Существует множество способов, но применительно к данному примеру мы рассмотрим один. Допустим, у нас есть 4 массива со значениями в ячейках, соответствующих дальнобойным фигурам противника. В реальной программе это может делаться пошагово. Поставили фигуру (сделан ход) — увеличили соответствующие ячейки на единицу, убрали — уменьшили. Если в первом массиве у нас не 0 в ячейке с индексом X, возможно, в этой вертикали есть ладья или ферзь. Мы должны просканировать два луча в обоих направлениях до первой фигуры. Если в третьем массиве в ячейке с индексом (X+Y) не 0, то, возможно, на восходящей (/) диагонали нас бьет ферзь или слон противника. Нужно тоже просканировать два луча. Приращение при построении линии известно сразу. Для пешек нужно просто проверить две

соседние клетки по диагонали. Несколько сложнее с королем и конем. Проверять соседние 8 клеток, в надежде найти там короля противника, и еще 8, в надежде найти коня, как-то не хочется. Но об этом несколько позднее.

Локальные функции

Локальной или вложенной называется процедура или функция, описанная внутри другой процедуры (функции). Она может иметь свои локальные переменные, а может и обращаться к переменным функции, внутри которой она описана. Вызывать ее можно только из функции, внутри которой она описана, т. е. область видимости ограничена родительской функцией.

Основным недостатком языка С при программировании шахмат является именно отсутствие в нем локальных функций. Вот пример локальной функции (процедуры) на паскале:

```
procedure Foo(arg1,arg2:integer);
procedure SubFoo(arg3,arg4:integer);
begin
         {...}
end;
begin
     {...}
end;
```

В функции `Foo` можно вызвать процедуру `SubFoo`, которая может обращаться к переменным `arg1`, `arg2` вызывающей ее функции `Foo`. Почему в С нет локальных функций? Трудно сказать. В языке С первоначально много чего не было. Видимо, основная причина заключается в том, что это считается несистемной вещью, т. к., чтобы обратиться к переменным родительской функции, `SubFoo` должна извлечь из стека регистр `BP(EBP)`, а проще говоря, указатель на эти переменные. Тем не менее это очень удобный прием. В некоторых версиях компиляторов С можно эмулировать локальную функцию на ассемблере (если она не имеет своих аргументов):

```
void Foo(int arg1, int arg2)
  {
SubFoo: //метка
         {...}
_asm ret; //возврат из подпрограммы
{...} //инструкции функции Foo
_asm call SubFoo; //вызов вложенной подпрограммы
}//function
```

Этот прием, тем не менее, является довольно опасным. Нужно очень четко представлять, какие регистры нужно сохранить в локальной процедуре, и, кроме того, разработчики некоторых компиляторов очень активизируются в данных случаях, и нет гарантии, что это будет работать, раз не предусмотрено синтаксисом языка. Гораздо проще оформить переменные (стековые) функции Foo в виде структуры, а в SubFoo передавать указатель на эту структуру.

```
typedef struct{
int arg1,arg2;
}Targ.*Parg;
void SubFoo(Parg arg)
   {
     {...}
   }
vois Foo(Parg arg)
   {
     {...}
SubFoo(arg);
   }
```

В некоторых случаях такой прием бывает чрезвычайно удобен. Функция Foo может быть очень большой и содержать множественные вызовы SubFoo, которая, в свою очередь, вызывает вложенные функции следующего уровня. Передавать каждый раз огромный список параметров очень накладно, и, кроме того, локальные функции должны иметь возможность изменять эти переменные. К сожалению, данный прием не улучшает читабельность программы. Когда-нибудь в C появятся локальные функции (как и нормальный строковый тип данных).

Игра "Уголки"

Для иллюстрации всего вышесказанного приведем небольшой пример. На игровом поле 8 × 8 находятся 12 черных шашек (в левом верхнем углу) и 12 белых (в правом нижнем). Нужно занять исходные позиции противника. Черные должны стремиться занять правый нижний угол, а белые — левый верхний. Шашки ходят во всех направлениях на одну клетку (кроме диагоналей). Шашка может прыгать через другие шашки в любых направлениях (кроме диагоналей).

Для решения этой задачи используется перебор. Ход противника не учитывается. Если говорить про моделирование поведения противника, то мы моделируем "болвана". Тем не менее, т. к. взятий нет, это неплохо работает на практике. Используется некоторый массив, где в каждой клетке нарисовано, куда фигуре лучше пойти. Просчет мы ведем только для черных. Таким об-

разом, числа массива минимальные в верхнем левом углу и максимальные в правом нижнем. Теоретически, массив оценок для черных должен быть следующий:

```
1, 2, 3, 4, 5, 6, 7, 8,
9,10,11,12,13,14,15,16,
17,18,19,20,21,22,23,24,
25,26,27,28,29,30,31,32,
33,34,35,36,37,38,39,40,
41,42,43,44,45,46,47,48,
49,50,51,52,53,54,55,56,
57,58,59,60,61,62,63,64
```

Я использовал другие числа, чтобы программа более явно стремилась в правый нижний угол. Дело в том, что мы не учитываем ход противника (для него массив инвертирован), и у нас нет времени на вычисления. В "Уголках" программа должна ходить практически сразу. Если она будет думать по 30 секунд над тривиальной задачей и вычислять позиционные преимущества, это будет выглядеть смешно. Мы считаем на 3 хода (только своих). Если бы мы учитывали ход противника, то задача ничуть не уступала бы шахматам. Потребовалась бы большая глубина перебора (чтобы было хотя бы 3 своих полухода, нужно суммарно иметь 6: 3 своих и 3 противника). Пришлось бы прибегать к сложным программным решениям, а для такой простой программы это выглядело бы нелепо. На практике играла бы программа не лучше (может, и хуже). Данный прием пропуска хода используется и в шахматах, но несколько по-другому. Суть в том, что если мы делаем подряд два хода, это засчитывается как наш возможный максимум. Если противник сделает ход, то результат может быть только меньше для нас. Здесь мы считаем на 3 хода и, следовательно, пропускаем 2 хода противника (полухода). Получается, что мы играем не с "болваном", а с "суперболваном". Программа ходит так, как если бы противник 2 раза не походил. Но т. к. игра построена в основном на "зевках", это работает.

Есть еще один интересный момент в программе. Это определение "прыгающих" ходов шашки. Шашка начинает движение из своей стартовой точки и может выписывать довольно замысловатые траектории. При нахождении возможного пути мы пользуемся следующим правилом: шашка может сделать любой легальный ход, за исключением тех клеток, на которых она уже побывала. Это только при взятии перемещений шашки в одном узле. Для запоминания полей, на которых шашка уже побывала, мы используем множество, в котором запоминаем индекс клетки массива, где побывала шашка. В языке С типа данных "множество" нет, и приходится конструировать его самим. Это делается чрезвычайно просто. Решение довольно любопытное и может применяться в других программах. Наше множество представлено массивом 16 байт. В каждом байте — 8 бит, следовательно, в мно-

жестве может быть запомнено 16 × 8 = 128 элементов (значений от 0 до 127). Если элемент (например, 10) есть, в множестве соответствующий бит установлен в 1. Если нет — в 0. Для определения бита, соответствующего данному числу, мы должны найти номер байта и номер бита в этом байте. Это делается очень просто:

```
byteNumber = N / 8;    //результат деления
bitNumber  = N % 8;    //остаток от деления
```

Игровое поле в уголках 8 × 8. Мы используем несколько больший массив: (10,10). Края массива (лишние поля) заполнены флагом выхода за пределы поля. Таким образом, не нужны проверки выхода за пределы. Мы всегда наращиваем координату на одну клетку, и если в ней флаг выхода за пределы игрового поля, то это значит, что мы вышли за пределы поля 8 × 8.

Листинг 1.1. Игра "Уголки"

```
CopyRight ©  2004 by Kornilov Evgeny e_k@sbor.net
//////////  DEFINITION ////////////////
#define B    1  //черная шашка
#define W    2  //белая
#define F    3  //флаг выхода за пределы доски
//стартовая позиция
int pos[10*10] =
{
F,F,F,F,F,F,F,F,F,F,
F,B,B,B,B,0,0,0,0,F,
F,B,B,B,B,0,0,0,0,F,
F,B,B,B,B,0,0,0,0,F,
F,0,0,0,0,0,0,0,0,F,
F,0,0,0,0,0,0,0,0,F,
F,0,0,0,0,W,W,W,W,F,
F,0,0,0,0,W,W,W,W,F,
F,0,0,0,0,W,W,W,W,F,
F,F,F,F,F,F,F,F,F,F
};
//куда лучше ходить
int StTab[10*10] =
{
0, 0,0,0,0,0,0,0,0,0,
0, 1,2,3,4,5,6,7,8,0,
0, 9,10,11,12,13,14,15,16,0,
0, 10,17,18,19,20,21,22,23,0,
0, 11,18,24,25,26,27,28,29,0,
```

```
0, 12,19,25,30,31,32,33,34,0,
0, 13,20,26,31,401,402,403,404,0,
0, 14,21,27,32,402,405,406,407,0,
0, 15,22,28,33,403,406,408,409,0,
0, 0,0,0,0,0,0,0,0,0
};
//стартовый список фигур машины (черных)
int PieceTab[12] = {11,12,13,14,
21,22,23,24,
31,32,33,34};
//номера конечных позиций
int EndTab[12] = {65,66,67,68,
75,76,77,78,
85,86,87,88};
//приращения при движении шашки
int DelTab[4] = {-1,1,-10,10};
//описатель перемещения
typedef struct{
int source,dest;
}TMove,*PMove;
////////////////тип – множество//////////
typedef union{
unsigned char data[16];
long __data[4];
}TSet,*PSet;
//обнуляет множество
#define Clear(set)\
(set).data[0]=0; (set).data[1]=0; (set).data[2] = 0; (set).data[3]=0;
//включает элемент n в множество set
#define Include(set,n)\
(set).data[(n) >> 3]   |=   (1 << ((n) & 7));
//возвращает не 0, если элемент n в множестве set
#define In(set,n)\
( (set).data[ (n) >> 3 ]    &   (1 << ((n) & 7)) )
////////////////////////////////////////////
//переменные при просчете
typedef struct{
int score,ply,max;
TMove bestMove;
TSet set;
}Targ,*Parg;
//максимальное значение при просчете
#define INFINITY 30000
```

```
//максимальная глубина просчета
#define MAX_PLY 3
//проверка на конец
int CheckEnd(void)
{
int n;
for(n = 0; n < 12; n++)
if(pos[ EndTab[n] ]  !=  B) return 0;
return 1;
}//function
//делает перемещение
#define MakeMove\
pos[move.source] = 0;\
pos[move.dest] = B;\
PieceTab[index] = move.dest;
//отменяет ход
#define UnMakeMove\
pos[move.dest] = 0;\
pos[move.source] = B;\
PieceTab[index] = move.source;
///////  FORWARD //////////
void SimpleMove(Parg,int);
void Move(Parg,int);
void CallSearch(Parg,PMove);
void Search(Parg);
//простой ход на клетку
void SimpleMove(Parg arg,int index)
{
int n;
TMove move;
move.source = PieceTab[index];
for(n = 0; n < 4; n++)
   {
move.dest = move.source + DelTab[n];
if(pos[move.dest] == 0)
       {
MakeMove
CallSearch(arg,&move);
UnMakeMove
       }
   }
}//function
//прыгает через шашку
void Move(Parg arg, int index)
```

```
{
int n,tmp;
TMove move;
for(n = 0; n < 4; n++)
   {
move.source = PieceTab[index];
move.dest = move.source + DelTab[n];
tmp = pos[move.dest];
if(tmp == 0  ||  tmp == F) continue;
move.dest += DelTab[n];
tmp = pos[move.dest];
if(tmp != 0) continue;
if(In(arg->set, move.dest)) continue;
//нашли — куда походить
Include(arg->set, move.dest);
MakeMove
CallSearch(arg,&move);
Move(arg,index);
UnMakeMove
}//for
}//function
//вызов просчета
void CallSearch(Parg arg, PMove move)
{
Targ nextArg;
nextArg.ply = arg->ply+1;
nextArg.score = arg->score  +
(StTab[move->dest] — StTab[move->source]);
Search(&nextArg);
//если нашли лучший ход
if(nextArg.max > arg->max)
    {
arg->max = nextArg.max;
arg->bestMove.source = move->source;
arg->bestMove.dest   = move->dest;
    }
}//function
//основная функция просчета
void Search(Parg arg)
{
int n;
if(arg->ply >= MAX_PLY)
     {
arg->max = arg->score;
```

```
return;
     }
if(CheckEnd())
     {
arg->max = INFINITY - arg->ply;
return;
     }
arg->max = -INFINITY;
//все перемещения на одну клетку
for(n = 0; n < 12; n++)
SimpleMove(arg,n);
//все перемещения через шашки
for(n = 0; n < 12; n++)
     {
//очистим множество
Clear(arg->set);
//занесем стартовую координату
Include(arg->set,PieceTab[n]);
Move(arg, n);
     }
}//function
//  пример первого вызова
/********
Targ arg;
int n;
memset(&arg,0,sizeof(Targ));
Search(&arg);
//изменение в позиции и в списке фигур
for(n = 0; n < 12; n++)
if(PieceTab[n] == arg.bestMove.source)
    {
PieceTab[n] = arg.bestMove.dest;
pos[arg.bestMove.source] = 0;
pos[arg.bestMove.dest] = B;
break;
    }
*********/
```

Грубое усилие и избирательность

Когда в 70-е годы прошлого века состоялся второй матч между русской "Каиссой" и американской Chess, то создатели американской программы сделали эпохальное заявление: "Мы собираемся заниматься выборочным

поиском, потому что полный перебор — это тупик". Такое или подобное заявление делали многие шахматные программисты на определенном этапе своей деятельности. И только очень немногие сделали приличную программу выборочного поиска. Среди них Ричард Лэнг с его программой Genius. В чем же трудность? Почему так мало удачных программ выборочного поиска? На этот вопрос можно ответить, но меня смущают некоторые вещи. Я недавно читал книгу по 3D-движкам и понял содержание только первых нескольких глав, да и то потому, что сам в свое время построил простейший 3D-движок. Остальная часть книги совершенно непонятна. Темы не раскрыты, а только затронуты, тон менторский, автор — явный зазнайка. В примерах программ к книге у него вращался (очень медленно) трехмерный зеленый бублик, и это приводилось в качестве достижения трехмерного моделирования. Если бы все читали только эту книгу, то не было бы ни Doom, ни Ducke, ни других хороших игрушек. Для кого была написана та книга? Видимо, для тех, кто и так все знает, а только хочет нечто освежить в памяти. Я все боюсь уподобиться этому автору. Для меня в alpha-beta алгоритме и его свойствах нет ничего сложного, потому что я долго занимался этим вопросом. Как дела обстоят у читателя? Да простят меня профессионалы, я буду много времени уделять тривиальным вещам и много раз повторяться. Я надеюсь, что при каждом повторе высветится та или иная грань алгоритма.

Почему только alpha-beta? Дело в том, что никакой альтернативы этому алгоритму не найдено, и это как искусственный интеллект в миниатюре, если мы считаем до конца или до такой большой глубины, что погрешностью можно пренебречь. Что значит считать до конца? В любой игре есть некоторый набор правил и есть условие, когда игра закончена. Вот и нужно считать до этого условия. Оно может никогда и не встретиться из-за повторов позиций. Что такое повторы? Это когда позиция повторяется в строке игры через несколько ходов. В шахматах, если позиция повторилась три раза в строке (проигранной), то оппонент может потребовать ничью. Например, есть некоторая строка:

```
a1b1 a8b8 b1a1 b8a8
```

Здесь белая ладья пошла с a1 на b1, а черная — с a8 на b8. Это 2 полухода. В следующих двух полуходах видно, что белая и черная ладья вернулись на свои места. Получился повтор позиции через 4 полухода. Если такая ситуация повторится 3 раза, то через 12 полуходов может быть предъявлена оценка `draw`. Оценка `draw` при просчете в большинстве случаев принимается равной нулю (ничья). Некоторые программисты делают оценку `draw` равной небольшой отрицательной величине, чтобы программа при просчете не зацикливалась из-за небольшого минуса в стратегической оценке. Что такое стратегическая оценка?

Оценка в шахматах может быть стратегической и материальной. Любой человек, учащийся играть в шахматы, должен представлять себе ценности фи-

гур. Это — материальная оценка. Она может изменяться в зависимости от стадии игры и конкретного набора фигур на доске, но, для простоты, ее можно считать статической, т. е. неизменной.

Вот примерные величины материальной оценки:

Название фигуры	Материальная оценка
пешка	1
слон	3
конь	3
ладья	5
ферзь	9
король	1000

Король "весит" больше, чем все остальные фигуры, вместе взятые, потому что когда съели короля — игра прекращается, и программа должна это понимать. Можно принять вес короля равным 0, а когда его съели, то просчет прекращается, и возвращается некоторая максимальная величина (обычно называемая `INFINITY`) минус глубина просчета. Почему вычитается глубина просчета? Дело в том, что программа считает только на некоторую глубину. Если белого короля съели на 10 полуходе, и это неизбежно (белые не могут ничего предпринять против), то оценка будет `INFINITY-10`. Если при этом же просчете есть другой способ поставить мат белым на 12 полуходе или глубже, то оценка будет `INFINITY-12`. Если мы во всех случаях будем возвращать просто `INFINITY`, то программа может зациклиться. В одном случае она будет выбирать вариант, ведущий к выигрышу на 12 полуходе, в другом — на 10.

Кроме материальной, есть еще и позиционная оценка. Относительно способов формирования позиционной оценки нет однозначного мнения (как и по большинству аспектов шахматного программирования), и мы пока не будем на ней останавливаться подробно. Нужно лишь помнить, что максимальное позиционное преимущество не переходит в материальное.

Если мы ведем перебор или поиск, и съеден король, то, как было сказано, просчет досрочно прекращается, и возвращается точная оценка.

Если мы в строке поиска встретили повтор позиции 3 раза, то просчет тоже прекращается, и, для простоты, можно возвратить 0.

Это явные два случая, когда программа сосчитала до конца. Как быть с остальными? Простейший вариант: превышена глубина перебора — вернем статическую оценку.

Есть ситуации, когда перебор можно досрочно прекратить и вернуть оценку без внесения дополнительной погрешности при вычислении. Этот алгоритм называется alpha-beta и является основным методом оптимизации полного перебора. Мы в некотором узле имеем набор позиций, полученных в результате ходов из данной позиции. Для получения лучшего хода мы должны перебрать все эти позиции и получить их оценки с точки зрения противника. Оценки берутся со знаком минус, т. к. это оценки с точки зрения противника. Для получения оценки с точки зрения противника мы рекурсивно вызываем функцию просчета для противника и передаем ей, в качестве аргумента, позицию, измененную после нашего хода. Функция противника тоже получает набор позиций противника и пытается их оценить, в свою очередь, рекурсивно вызывая функцию просчета для нашего цвета фигур, но уже после своего хода. Этот процесс продолжается рекурсивно, пока не исчерпана глубина просчета (мы не можем рекурсивно углубляться до бесконечности) или пока не выполнится некоторое условие досрочного прекращения перебора, которое мы обсуждаем. Что это за условие?

Дело в том, что, перебирая свой набор позиций, мы в некоторой переменной запоминаем максимальный результат. Это нужно, чтобы функция, обсчитывающая данную позицию, запомнила и вернула оценку лучшего хода. Эта переменная изначально равна (-INFINITY), и оценка любого хода улучшает ее. Теперь внимание! Допустим, эта переменная называется alpha, и мы уже нашли некоторый ход, оценка которого равна, например, 1. Мы не прекращаем перебор в данном узле (позиции) и вызываем рекурсивно поиск со следующим перемещением. Если в следующем узле (или в любом другом узле, который обсчитывается с точки зрения противника) противник уже получил результат 1 или больше (для нас — меньше), то в этом более глубоком узле перебор можно досрочно прекратить и вернуть 1 (для противника со знаком минус). Дело в том, что противник уже достиг нашего максимума, или alpha, и дальше будет только увеличивать оценку (для нас со знаком минус: только уменьшать). Когда поиск вернется в наш узел, мы эту оценку все равно не запишем, т. к. мы записываем только оценки, увеличивающие alpha, а эта оценка будет равна alpha или меньше. Здесь мы видим, что перебор можно досрочно прекратить, если alpha (результат) в узле достиг максимума противника в этой строке. Этот максимум называется beta. Наш alpha — это минус beta противника, достигнутый до этого. Переменные alpha и beta передаются в стек рекурсивно (как аргументы функции). Таким образом, они всегда новые в конкретной строке (копии оригинала). Это позволяет легко отслеживать наш максимум и максимум противника (alpha и beta).

Вот пример:

```
int AlphaBeta(int alpha,int beta,TPos pos, int ply)
   {
//если исчерпан лимит глубины - вернем статическую оценку позиции
if(ply >= MAX_PLY) return Evaluate();
```

```
//получим все позиции-перемещения
Generate();

//перебираем список (:
while(moveList)
      {
tmp = -AlphaBeta(-beta,-alpha,nextPos,ply+1);
if(tmp > alpha) alpha = tmp;
if(alpha >= beta) return beta; //отсечение
}//while
//если дошли до этой точки, то не было отсечения
return alpha;
}//function
```

Первый вызов:

```
AlphaBeta(-INFINITY,INFINITY,pos,0);
```

Это несколько сложно для понимания, но нужно не пожалеть времени и разобраться. Разберем подробно данную функцию и принятые в ней обозначения:

- `Generate` — определяет набор позиций, полученный в результате возможных ходов из данной позиции; наборы позиций в реальных программах не используются, это сделано только для простоты;
- `alpha` — наш максимум;
- `beta` — максимум противника. При рекурсивном вызове эти переменные меняются местами и заносятся со знаком минус. Таким образом, в каждом узле у нас корректные alpha и beta;
- `pos` — позиция со сделанным в ней последним ходом противника. Если начало игры, то хода противника в эту позицию нет;
- `ply` — глубина просчета от 0. Если глубина просчета превысила некоторое значение (`MAX_PLY` может быть равно, например, 6, чтобы программа не "умерла"), то просчет досрочно прекращается;
- `Evaluate` — функция, возвращающая статическую оценку позиции. В самом простейшем случае она считает только материал.

Вот пример простейшей функции `Evaluate`:

```
//цвета
#define BLACK 128
#define WHITE 64
//коды фигур
#define KING    1
#define QUEEN   2
```

```
#define ROOK    3
#define BISHOP  4
#define KNIGHT  5
#define PAWN    6

#define wPAWN   1
#define wKNIGHT 3
#define wBISHOP 3
#define wROOK   5
#define wQUEEN  9
#define wKING   100

//массив перекодировки кода фигуры в ее вес
int _wf[7] = {0,wKING,wQUEEN,wROOK,wBISHOP,wKNIGHT,wPAWN};
//позиция
typedef unsigned char    TPos[64];
int Evaluate(TPos pos, int player)
{
int score = 0, n, square;
for(n = 0; n < 64; n++)
   {
square = pos[n];
if(square == 0) continue;
if((square & (BLACK | WHITE)) == player)
score += _wf[square & 7];
else
score -= _wf[square & 7];
}//for
return score;
}//function
```

Клетка доски у нас представлена типом `unsigned char`. Это переменная размером один байт без знака. Байт содержит 8 бит, каждый из которых может принимать значение 1 или 0. Если значение некоторого бита 1 (все остальные 0), то это соответствует следующему десятичному числу:

Номер установленного бита	7	6	5	4	3	2	1	0
Десятичное значение	128	64	32	16	8	4	2	1
Маска выделения кода	0	0	0	0	0	1	1	1
Маска выделения цвета	1	1	0	0	0	0	0	0

Коды фигур занимают первые 3 бита, и поэтому когда мы делаем square & 7, то выделяем фигуру. Это можно записать как макрос с параметрами:

```
#define PIECE(square)  ((square) & 7)
```

Макрос с параметрами подобен аналогичной функции, только выполняется быстрее. Функция будет следующей:

```
int Piece(int square)
{
return square & 7;
}
```

Из-за такой тривиальной функции нет смысла загружать процессор, а мы должны помнить, что скорость выполнения программы — это критичное условие в данном случае. Современные процессоры очень хорошо считают, но миллионы лишних вызовов функций — это как раз тот случай, что может отобрать "излишки" скорости.

Для выделения цвета из клетки доски (суммарного кода фигуры) мы тоже можем написать макрос:

```
#define COLOR(square) ((square) & (BLACK | WHITE))
```

Операция | — это поразрядное ИЛИ, операция & — это поразрядное И, операция | оставляет в результате те биты, которые хоть в одном из операндов были установлены в 1. Например, двоичная запись:

```
01010   первый операнд
ИЛИ
10100   второй операнд

  ------
11110   результат
  //////////////////////
01010
И
00011

  --------
00010
```

Операция поразрядного И (&) используется как маска, чтобы выделить первые 3 бита, обозначающие код фигуры. Десятичное 7 — это двоичное `00000111`. Если у нас в клетке доски черный король, это будет `BLACK + KING = 129`, или `BLACK | KING = 129`, или в двоичной записи `10000001`. Применив маску для выделения кода фигуры `00000111`, получим `00000001`.

Получили только код короля. Кто знает хитрости двоичной арифметики, да простит мне это отступление. Позиция у нас представлена массивом 0..63. Можно использовать и двумерный массив, это не принципиально. Это только в дальнейшем раздует некоторые структуры данных.

```
8*8 = 64;
```

Наш массив 0..63 — это одномерный массив, имитирующий двумерный. Если использовать хитрости двоичной арифметики, то, имея один индекс N для массива 0..63, можно легко получить координаты X и Y для массива `[8][8]`.

```
x = n & 7;
y = n >> 3;
```

Операция >> — это битовый сдвиг вправо, в данном случае на 3. Сдвиг вправо на 1 эквивалентен делению на 2, соответственно, влево на 1 — умножению на 2. Сдвиг вправо на 3 эквивалентен `n / 8`. Если записать приведенное выше иначе, то, чтобы получить координату X и Y по N, надо:

```
x = n % 8;  //остаток от деления  (x := n mod 8)
y = n / 8;  //деление              (y := n div 8)
```

Функция `Evaluate` у нас сканирует весь массив игры в поисках фигур. Это расточительно по времени, но пока мы не будем на этом заострять внимание. Если позиция представлена одномерным массивом, то в реальных программах организация его несколько сложнее для определения выхода за пределы доски. Для двумерного массива 8 × 8 общий случай определения выхода может быть записан так:

```
if((unsigned)x < 8   &&   (unsigned)y < 8))
  {
//клетка X,Y в пределах доски 8*8
  }
```

Если функция `Evaluate` считает материал с точки зрения белых, она суммирует вес всех белых фигур, а потом из них вычитает вес всех черных фигур. Если оценка берется для черных, то наоборот. Оценка, таким образом, получается инвертированная. Если у белых лишняя пешка, то это +1 для белых и −1 для черных.

Этот алгоритм называется NegaMax. В функции `AlphaBeta`, если переменная `ply` превысила максимальную глубину просчета, то перебор прекращается, и возвращается статическая оценка позиции `Evaluate` для конкретного цвета фигур.

Это типичный пример грубого усилия по Клоду Шеннону — перебор всех вариантов в некотором фиксированном горизонте. Сейчас никто такого грубого усилия не использует, а в конце перебора вместо функции `Evaluate` вызывается некоторый упрощенный вариант поиска, считающий только форсированные ходы. Что такое форсированные ходы?

Это ходы, требующие немедленного ответа или дающие максимальное приращение в оценке. Ходы, требующие немедленного ответа, — это, в простейшем случае, шахи. Ходы, дающие максимальное приращение в оценке — это взятия (превращения пешек тоже дают материальное приращение

оценки). Самый простейший форсированный (статический) поиск считает только взятия. Сторонники грубой парадигмы (полного перебора или грубого усилия) почему-то присвоили себе этот вид статического поиска и считают, что если он вызывает минимальные накладные расходы (взятий мало), то это и не выборочный поиск вовсе.

Таким образом, мы видим, что грубое усилие в чистом виде сегодня не практикуется, есть только спор о сложности форсированного поиска. Чем больше ходов учитывает форсированный поиск, тем он точнее, но тем меньше глубина полного перебора (alpha-beta). Сторонники грубой парадигмы считают, что нужно перебрать максимально глубоко полным перебором, а затем вызвать максимально простой статический поиск. Сторонники выборочного поиска вставляют между конечным поиском взятий и полным перебором еще один промежуточный вариант поиска, который является усложненным статическим поиском, учитывающим, кроме взятий, еще и шахи, а на меньшей глубине — и другие "сильные" перемещения, например атаки, всяческие угрозы и т. д.

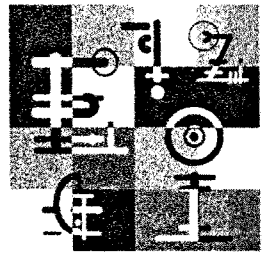
Что лучше? Трудно сказать. В настоящее время появился нулевой ход, как усредненный метод выборочного поиска, но некоторые шахматные программисты не признают его или признают частично, в качестве некоторого дополнительного условия. Итак: и грубое усилие, и избирательность в настоящее время не сильно отличаются друг от друга. Речь идет о конкретных настройках для конкретной программы. Основная идея выборочного поиска — это отсечение некоторых ветвей дерева перебора без их просчета на полную глубину. Человеку понятно, что просчитывать некоторые ветви просто абсурдно, но как научить этому машину? Машина признает только точные оценки. Она может всегда "подрезать" интересную ветвь, и наоборот, считать всякий абсурд. Сторонники грубого усилия, исходя из того, что на каждое правило отсечения ветви дерева перебора может найтись исключение, считают, что любые Forward pruning — неприемлемо. Такие взгляды легко иметь, когда не пишешь реальную программу. Даже мощности DeepBlue мало для грубого усилия. Можно подождать еще лет 30, и такие процессоры, применительно к шахматам, наверняка появятся. Это напоминает ситуацию с 3D-играми. Сейчас существуют мощные процессоры и графические ускорители, позволяющие делать игры с высокой степенью детализации, но если взять частный случай — Doom-образные миры, то такие программы работали и на 386, а самые простые — и на 286 процессоре с 16 МГц тактовой частоты. Если бы создатели игры Doom ждали, пока не решится задача построения 3D-миров в целом, и не появятся суперпроцессоры, мы не имели бы этой и многих других игр, поразивших воображение в свое время. Применительно для программирования шахмат, можно сказать, что совершенно не обязательно ждать появления процессоров, с помощью которых задача (конкретная) для доски 8 на 8 и данного набора фигур и правил будет решена силовым методом. Ведь мы имеем дело не с аб-

страктной переборной задачей, а с конкретной. Если учесть определенную шахматную специфику, то можно построить хорошую программу на не очень мощном процессоре, и она не будет просить 100 Мбайт под хеш-таблицу. Все это требует, однако, большого усилия, и нужно много думать. Нет, к сожалению, готовых рецептов. Есть некоторые приемы, до последнего времени (а частично, и сейчас), не разглашаемые шахматными программистами-профессионалами. Кроме приемов, многое зависит и от конкретной реализации. В любом случае, если современная программа и не использует некоего хитроумного способа выборочного поиска, она применяет нулевой ход для подрезки ветвей дерева, а это тоже, в сущности, выборочный поиск.

Итак, на каком методе остановиться? На современных машинах можно осуществить грубое усилие на 8 полуходов, плюс облегченный форсированный поиск, плюс расширение некоторых строк (как минимум, шахи). Это даст неплохую программу. Отдельные ветви она будет просчитывать значительно глубже за счет выборочных продлений. Выборочные продления — это когда в пределах полного перебора глубина не сокращается при определенных ходах. Простейший случай — шахи. Чтобы написать такую программу, нужно сосредоточиться на эффективности кода, хеш-таблице и прочих методах ускорения перебора и улучшения порядка ходов. Это требует немалого усилия, и нужно разобраться в теории и много экспериментировать. При достаточном мастерстве можно сосчитать и немного глубже.

Для выборочного поиска нужно больше задумываться о специфике конкретной игры, какие ветви необходимо считать далее. Это требует достаточно высокого уровня понимания игры. Хочется отметить, что в любой хорошей программе есть своя изюминка, и не всегда решения, пригодные для одной схемы, будут работать в другой.

Русская программа "Каисса" в первом матче с Chess нашла форсированный мат на много ходов вперед. Американцы недоумевали, как может программа считать так глубоко? Им показали, что все ходы, ведущие к мату, были форсированными — взятия и шахи. Ко второму матчу американцы были подготовлены значительно лучше. Они использовали достижения противника и сделали одну из лучших программ для своего времени. Подобная ситуация, в принципе, сохраняется и теперь. Кто нашел новое интересное решение, тот и написал сильнейшую программу.



ГЛАВА 2

Основы программирования

MiniMax и NegaMax

Стратегия поиска лучшего хода была описана еще Клодом Шенноном. Впрочем, она очевидна, и любой человек, знакомый с программированием и рекурсией, может вывести ее самостоятельно. Коротко о рекурсии. Рекурсия — это когда функция вызывает сама себя, и каждый раз при входе в функцию в стеке программы создается новая копия блока переменных. Это не относится к статическим переменным, общим для всех вызовов функции, просто, если они описаны в теле функции, их область видимости ограничена данной функцией. Чем же так примечательна рекурсия? Она позволяет описать некоторые процессы, просто определив законы их развития, а как конкретно будет развиваться алгоритм, не так уж и важно. Главное, мы задаем его параметры развития, и совершенно очевидно, что он работает. Формальные доказательства нас в данном случае не интересуют. Некоторые задачи, легко решаемые с помощью рекурсии, часто невозможно или сложно решить другими способами. Это же относится к поиску лучшего хода. Допустим, у нас есть некоторая воображаемая игра, где на доске (размеры которой не важны), ходят белые и черные фигуры. Есть правила игры, и фигуры им подчиняются. Эта игра может быть шахматами, шашками, уголками и т. д. Алгоритма лучшего хода мы не знаем. Как же нам найти если не лучший, то хоть осмысленный ход? Предположим, у нас уже есть функция, дающая оценку каждому ходу. Устройства этой функции мы не знаем, но можем ей пользоваться и оценивать каждый ход. Как же нам для некоторой позиции в игре найти лучший ход? Очевидно, надо получить все перемещения, сделать оценку каждого из них и найти перемещение с максимальной оценкой. Рассмотрим некоторый код в листинге:

```
PMove  SearchBestMove(void)
{
   int score      = -INFINITY;    //минимально возможное значение
```

```
    PMove bestMove = 0;            //лучший ход
    PMove move     = GenerateAllMoves();    //получим все перемещения
    while(move)
    {
        MakeMove(move);                  //сделаем ход
        int tmp = EvaluatePosition();    //оценка позиции
        UnMakeMove(move);                //восстановим исходную позицию
        if(tmp > score)
        {
          score = tmp;
          bestMove = move;
        }
        move = move->next;
    }
    return bestMove;
}
```

Данная тривиальная функция вернет лучший ход. Единственным неясным моментом остается функция `EvaluatePosition`. Как же она оценивает позицию, если неизвестен алгоритм данной игры?

Давайте теперь перепишем функцию так, чтобы она возвращала оценку лучшего хода, а не сам ход. Это не вызовет затруднений. Единственное, введем переменную `color`, обозначающую цвет фигур данной позиции, в которой мы ищем лучший ход и его оценку. Функция `EvaluatePosition` будет возвращать оценку относительно цвета конкретных фигур. Чем же отличаются оценки относительно белых и относительно черных для одной и той же позиции? Только знаком. Если в какой-то позиции у белых есть преимущество, то оценка относительно черных для этой позиции будет такой же, но со знаком минус. Оценка описывается следующим кодом:

```
int  SearchBestMove(int color)
{
    int score      = -INFINITY;  //минимально возможное значение
    PMove move     = GenerateAllMoves();    //получим все перемещения
    int opColor    = (color == BLACK?WHITE:BLACK); //цвет фигур противника
    while(move)
    {
        MakeMove(move);
        int tmp = -EvaluatePosition(opColor);  //оценка позиции
        UnMakeMove(move);
        if(tmp > score) score = tmp;
        move = move->next;
    }
    return score;
}
```

Обратите внимание, что мы определили цвет фигур противника и использовали результат функции EvaluatePosition со знаком минус. Что же это за неведомая функция, которая оценивает нашу позицию? Эта и есть наша функция SearchBestMove. Происходит рекурсия. Функция вызывает сама себя для оценки позиции противника. Нужно только ввести переменную Depth, которая будет ограничивать глубину рекурсии, говоря другими словами, определять, на сколько полуходов мы считаем. При каждом вызове Depth уменьшается на 1. Когда Depth равно нулю, рекурсия прекращается, и мы вызываем действительно оценочную функцию. Простейшая оценочная функция для шахмат считает фигуры на доске. Условимся о весе фигур:

- пешка — 1;
- конь и слон — 3;
- ладья — 5;
- ферзь — 9;
- король — INFINITY (максимально возможная оценка).

Если съели короля — это выигрыш. Расчет досрочно прекращается. Сейчас мы не будем заострять на этом внимание. Что должна сделать наша оценочная функция, чтобы получить оценку для белых? Она должна суммировать вес всех белых фигур и вычесть из полученного результата сумму всех черных фигур. Если фигур поровну, функция вернет 0. Если у белых есть преимущество в одну пешку, вернет 1. Если у черных лишний конь, вернет −3. Эта оценка приближенная, но чем глубже мы считали, тем ближе эта оценка к истинной. Вот наша переписанная функция:

```
int  SearchBestMove(int Depth,  int color)
{
   int score      = -INFINITY;  //минимально  возможное значение
   int opColor    = (color == BLACK?WHITE:BLACK); //цвет фигур противника
   PMove move;
   if(Depth == 0) return Evaluate(); //оценка конечной точки
   move = GenerateAllMoves();     //получим все перемещения
   while(move)
   {
        MakeMove(move);
        int tmp = -SearchBestMove(Depth-1,opColor);  //оценка позиции
        UnMakeMove(move);
        if(tmp > score) score = tmp;
        move = move->next;
   }
   return score;
}
```

Функция `Evaluate` просто подсчитывает материал в конце перебора. Если мы вызовем такую функцию в шахматах с глубиной 4 (`Depth = 4`), то она не даст нам уже поставить программе известный мат в 3 хода.

Этот алгоритм осуществляет полный перебор всех вариантов. Число рассмотренных позиций будет оцениваться как W в степени D, где W — примерное количество ходов в одной позиции, D — глубина просчета.

Для шахмат среднее количество возможных перемещений из одной позиции примерно равно 40. Это значит, что, считая на глубину 4, мы должны перебрать 40 × 40 × 40 × 40 = 2560 тыс. позиций, на глубину 5 — 10 240 тыс. и т. д. Если мы будем продолжать дальше, то получим число, превышающее количество атомов во Вселенной. Дерево перебора, которое строит этот алгоритм, растет экспоненциально. В этом и заключается основная проблема программирования логических игр методом перебора. Считать надо очень глубоко, а все варианты глубоко просмотреть невозможно. На сегодняшний день на самых мощных процессорах при самом оптимальном коде можно считать на глубину 6 в реально оцениваемый промежуток времени. Данный алгоритм называется NegaMax. Он использует инвертированную оценку. Есть еще один вариант этого алгоритма — MiniMax, в нем оценка с одним знаком, но существуют две функции: одна максимизирует результат, вторая минимизирует. Оба алгоритма делают одно и то же, но NegaMax более компактен и удобен для понимания. Все дальнейшие усовершенствования используют NegaMax в качестве своей основы.

Alpha-beta

Сейчас мы рассмотрим основной алгоритм оптимизации перебора. Он называется alpha-beta отсечениями, или просто alpha-beta. Суть его в том, что для получения оценки такой же точности, как и при полном переборе, совершенно не обязательно просматривать все варианты. Для определения отсекаемых вариантов не требуется знать особенностей данной игры.

В рекурсивной функции мы должны ввести две новые переменные — максимум для белых и максимум для черных. При начальном вызове обе эти величины равны минимально возможному значению: (`-INFINITY`). Если в какой-то позиции, например, для черных, мы получили результат, превышающий максимум для черных, достигнутый до этого, мы увеличиваем это значение. Теперь рассмотрим некоторую строку игры. Представим, что где-то в глубине дерева перебора мы обнаружили, что максимумы белых и черных сравнялись. Надо помнить, что это инвертированные величины, и сравнивать их нужно со знаком минус:

```
if( -maxWhite == maxBlack) {...}
```

Допустим, мы просчитываем позицию для белых. Если мы продолжим перебирать в данной позиции, то максимум для белых может еще увеличить-

ся, а может остаться прежним, но он уже сравнялся с максимумом черных. Это значит, что когда программа вернется в точку (рекурсивно), в которой максимум для черных, результат будет отвергнут, т. к. он не превышает максимума для черных в этой позиции. Также это значит, что в данной позиции для белых мы можем прекратить перебор и вернуть полученный результат досрочно. Дальше считать нет смысла. Рассмотрим программный код, иллюстрирующий сказанное:

```
int Search(int color, int Depth, int maxWhite, int maxBlack)
{
    if(Depth == 0) return Evaluate(color); //оценка конечной точки
    int score   = -INFINITY;
    int opColor = (color == BLACK?WHITE:BLACK);
    PMove move = GenerateAllMoves();

    while(move)
    {
      MakeMove(move);
      int tmp = -Search(opColor,Depth-1,maxWhite,maxBlack);
      UnMakeMove(move);

      if(tmp > score)
      {
          if(color == WHITE)
          {
               if(score > maxWhite) maxWhite = score;
               if(-maxWhite <= maxBlack) return maxWhite;
          }else{
               if(score > maxBlack) maxBlack = score;
               if(-maxBlack <= maxWhite) return maxBlack;
          }
      }

      move = move->next;
    }
    return score;
}
```

Пример первого вызова:

```
Search(WHITE, 4, -INFINITY, -INFINITY);
```

Переменные `maxWhite` и `maxBlack` получили название alpha и beta, а сама функция описывается без избыточного кода:

```
int AlphaBeta(int color, int Depth, int alpha, int beta)
{
    if(Depth == 0) return Evaluate(color);
    PMove move = GenerateAllMoves();

    while(move  &&  alpha < beta)
    {
      MakeMove(move);
      int tmp = -AlphaBeta(color==BLACK?WHITE:BLACK,  Depth-1, -beta,
                           -alpha);
      UnMakeMove(move);
      if(tmp > alpha) alpha = tmp;
      move = move->next;
    }

    return alpha;
}
```

Пример первого вызова:

```
AlphaBeta(WHITE, 6, -INFINITY, INFINITY);
```

При первом вызове функция вызывается с максимальным окном. При рекурсивных вызовах переменные alpha и beta меняются местами с инверсией знака. Как видите, все очень просто, но требует некоторого осмысления. Alpha-beta алгоритм, при наилучшем порядке ходов, построит значительно меньшее дерево перебора. Это примерно равно корню квадратному из числа позиций, просматриваемых при полном переборе. Нужно учесть, что при наихудшем порядке ходов, т. е. когда отсечку за beta вызывает последний ход, alpha-beta просмотрит столько же позиций, что и NegaMax. Это говорит о том, что alpha-beta очень чувствителен к порядку ходов. Скорость просчета еще сильно зависит на практике от возможного диапазона оценок. Если мы в шахматах считаем только материал, то оценка всем ходам, кроме взятий, равна нулю. Это значит, что отсечек будет очень много, особенно если взятия мы будем рассматривать первыми. Это очень схематично, но на практике верно. Если же мы, кроме материальной, учитываем еще и позиционную оценку, нам намного труднее получить дерево, близкое к минимальному. Примерная оценка, с учетом позиционной составляющей:

- пешка — 1000
- слон и конь — 3000
- ладья — 5000
- ферзь — 9000
- король — 90 000

Позиционная оценка — от −500 до 500.

Для получения более быстрых отсечек мы должны ходы, дающие большие приращения (материальной или просто позиционной оценки), рассматривать первыми. Это, повторяю, очень схематично, и подробнее об упорядочивании перемещений будет сказано позже.

Alpha-beta алгоритм является основой всех современных шахматных программ.

Alpha-beta с амортизацией отказов

Мы рассмотрели alpha-beta алгоритм. Он всегда возвращает значение из alpha-beta диапазона. Если мы, например, вызвали функцию поиска из корня дерева с параметрами alpha = −1, beta = 1, то, каким бы ни был истинный результат, наша функция вернет значение из заданного промежутка alpha и beta. Можно модернизировать функцию таким образом, чтобы возвращаемое значение не ограничивалось этим промежутком. Представим, что мы в некоторой точке дерева. Функция перебрала все перемещения, но так и не смогла улучшить alpha. В таком случае можно вернуть не alpha, а максимальное полученное значение для этого узла. Это никоим образом не скажется на конечном результате. Точность просчета будет такой же. Alpha-beta перебор с возвратом максимального полученного результата каждой функции называется alpha-beta с амортизацией отказов. Приведем его код:

```
int AlphaBeta(int color, int Depth, int alpha, int beta)
{
   if(Depth == 0) return Evaluate(color);
   int score = -INFINITY;
   PMove move = GenerateAllMoves(color);

   while(move)
   {
     MakeMove(move);
     int tmp = -AlphaBeta(color==WHITE?BLACK:WHITE, Depth-1, -beta,
                          -alpha);
     UnMakeMove(move);
     if(tmp > score) score = tmp;
     if(score > alpha) alpha = score;
     if(alpha >= beta) return alpha;
     move = move->next;
   }
   return score;
}
```

В практических реализациях используется, как правило, эта схема. Скорость работы обоих вариантов alpha-beta примерно одинакова. В шахматах мы,

например, можем легко определить пат с заданной схемой. Если мы видим, что сгенерированы только перемещения короля, а по максимальной оценке видно, что он бьется на следующем полуходе, то вместо полученной оценки функция должна возвратить ноль — ничья. Вариант alpha-beta с амортизацией отказов используется также в некоторых алгоритмах, например, NegaScout.

Перебор с нулевым окном

В этом разделе мы рассмотрим несколько отвлеченную тему. Она будет необходима для понимания особенностей работы некоторых алгоритмов. Это — перебор с нулевым окном. Используется вариант alpha-beta алгоритма с амортизацией отказов. Что такое нулевое окно? Мы вызывали alpha-beta поиск с окном от `-INFINITY` до `INFINITY`. Результат получается такой же точности, что и при полном переборе. Если мы присвоим alpha некоторое значение, например 0, то beta при нулевом окне будет alpha+1. Поиск при нулевом окне намного быстрее, чем при полном окне. Это связано с тем, что больше будет отсечек. Допустим, мы вызвали alpha-beta функцию поиска с нулевым окном и получили некоторый результат:

```
score = AlphaBeta(alpha,alpha+1);
```

Что он означает?

- Если `score` меньше или равно `alpha`, то результат просчета при полном окне, т. е. истинный, будет от `-INFINITY` до `score`, включительно. Говоря другими словами, `score` — это наш возможный максимум.
- Если `score` больше `alpha`, то истинный результат будет от `score`, включительно, до `INFINITY`. `score` — это наш минимум.

Эти вещи несколько трудны для понимания, но их необходимо усвоить. Они используются в NegaScout и других алгоритмах.

Почему, если функция при нулевом окне вернула результат, меньший или равный alpha, это наш максимум? Я затрудняюсь объяснить. Нужно размышлять, чтобы понять. Она, может, и могла вернуть результат еще меньше, но произошла отсечка, и если мы произведем перебор даже с полным окном, результат не будет более `score`.

Principal Variation Search

Поиск с основным вариантом имеет название Principal Variation Search (PVS). Мы делаем предположение, что нам известны лучшие ходы в некоторых узлах. Тогда, если мы нашли такой ход, мы его рассматриваем с полным alpha-beta окном, как при стандартном alpha-beta алгоритме. Ожидается, что ход потенциально хороший и может повысить значение alpha в узле.

Остальные ходы мы условно относим к плохой категории и ожидаем, что они не улучшат значение alpha. Если ход не улучшает alpha, то зачем его рассматривать с полным окном? Мы можем взять данный ход с окном:

```
newAlpha = alpha;
newBeta  = alpha+1;
```

Ожидается, что результат расчета будет меньше или равен alpha. Так и получается на практике для большинства позиций. Если же результат оказался больше alpha, нужно этот ход пересчитать с полным окном от alpha до beta:

```
//поиск с основным вариантом
int PrincipalVariation(int alpha, int beta, int depth)
{
   int tmp;
    if(depth <= 0) return Evaluate();
    //если нашли хороший ход
    if( findPV)
    {
       //делаем перемещение
       MakeMove;
       //ищем с полным окном
       tmp = -PrincipalVariation(-beta,-alpha,depth-1);
       //восстанавливаем позицию
       UnMakeMove;
       if( tmp > alpha) alpha = tmp;
    }//endif
    //получим список перемещений
    //из нашей позиции
    Generate();
    while(moveList   &&   alpha < beta)
    {
       //сделали ход
       MakeMove;
       //просчет с нулевым окном
       tmp = -PrincipalVariation(-(alpha+1),-alpha,depth-1);
       //если эвристика не сработала, и ход улучшает
       // alpha, то пересчитаем с полным окном
       if(tmp > alpha   &&  tmp < beta)
         tmp = -PrincipalVariation(-beta,-alpha,depth-1);
       //восстановили позицию
       UnMakeMove;
       if(tmp > alpha) alpha = tmp;
    }//while
    return alpha;
}
```

Основные обозначения:

- `alpha` — наш максимальный результат, достигнутый в строке просчета до этого;
- `beta` — максимальный результат противника;
- `depth` — глубина просчета;
- `Evaluate` — функция, возвращающая статическую оценку позиции;
- `Generate` — определяет все перемещения из данной позиции;
- `MakeMove` — делает ход;
- `UnMakeMove` — восстанавливает позицию;
- `findPV` — ищет заведомо хороший ход.

Как программа ищет потенциально хороший ход? Способ состоит в извлечении главной строки изменения после каждой итерации. Например: просчитали мы на 5 полуходов и извлекли главную строку изменения. Главная строка изменения, или главное изменение (PV) — это наилучшая игра для обеих сторон, по мнению программы. Когда мы будем при следующей итерации перебирать на 6 полуходов, узлы PV будут рассмотрены с полным окном, а остальные мы попытаемся сначала отсечь за alpha с нулевым окном. Это старый, испытанный способ. В настоящее время применяют хеш-таблицы и другие методы для улучшения порядка ходов.

Для понимания работы этого алгоритма мы должны учитывать, что перебор с нулевым окном выполняется быстрее, чем с полным alpha-beta окном. Причина здесь в том, что если окно alpha-beta минимальное, это вызывает гораздо больше отсечек. Principal Variation считает быстрее на практике, чем стандартный метод alpha-beta. Ускорение может достигать 50% и более, особенно в тех случаях, когда главное изменение не сильно отличается от PV предыдущей итерации. Principal Variation является простым и эффективным способом ускорения счета, хотя ускорение даже в два раза не является принципиальным. Это даст нам возможность сосчитать быстрее на глубину N, но мы не сможем сосчитать на глубину N+1, при том же контроле времени.

Можно усовершенствовать приведенный пример с извлечением строки главного изменения и просмотра ее первой при следующей итерации. Если ход PV, рассматриваем с полным окном, если нет — пытаемся отсечь за alpha.

```
//описатель перемещения
typedef struct tagMove{
     int source,  //откуда пошла фигура
         dest;    //куда
     struct tagMove *next; //следующий элемент списка
}TMove, *PMove;
```

```
//строка перемещений
//если source = dest, то перемещение пустое
//строка всегда заканчивается пустым описателем
#define MAX_LINE 100
typedef TMove TLine[MAX_LINE];
//копирует описатель перемещения
#define COPY_MOVE(move1,move2)\
(move2).source = (move1).source;\
(move2).dest   = (move1).dest;
//обнуляет описатель перемещения
#define RESET_MOVE(move)\
(move).source = (move).dest = 0;
//возвращает 1, если описатель перемещения пустой
#define IS_ZERRO_MOVE(move)\
( (move).source == (move).dest)
//записывает ход + строка в line
void SaveLine(PMove move, PMove tmpLine, PMove line)
{
    int n;
          //копируем наш ход + строка
          COPY_MOVE(*move,line[0]);
          n = 0;
          while( n < MAX_LINE-2   &&
                 !ZERRO_MOVE(tmpLine[n]))
          {
              COPY_MOVE(tmpLine[n],line[n+1]);
              n++;
          }
          //запишем пустой описатель
          //в конец строки
          RESET_MOVE(line[n+1]);
}
////////////////// VAR ///////////////////
//максимальная глубина просчета
int maxPly;
//главное изменение от корня дерева перебора
TLine  bestLine;
//// search /////////////////
  int PrincipalVariation(int alpha, int beta,
                int ply,    //глубина от 0
                PMove line,//возврат строки
                bool PV     //главное изменение?
               )
```

```
{
   TLine tmpLine;
   PMove move;
    if(ply >= maxPly) return Evaluate();
    //если превышен лимит времени
    if(TimeUp) return ...;//любое число
    //попробуем ход PV
    if(PV  &&  !ZERRO_MOVE(bestLine[ply]))
    {
       move = &bestLine[ply];
       MakeMove;
       RESET_MOVE(tmpLine[0]);
       tmp  = -PrincipalVariation(-beta,-alpha,ply+1,
                                  tmpLine, true);
       UnMakeMove;
       if(TimeUp) return ...;
       if(tmp > alpha)
       {
          alpha = tmp;
          if(alpha < beta)
              SaveLine(move,tmpLine,line);
       }
    }
    //начало списка перемещений
    move = Generate();
    while(move  &&  alpha < beta)
    {
       MakeMove;
       RESET_MOVE(tmpLine[0]); //строка пуста
       tmp = -PrincipalVariation(-(alpha+1),-alpha,
                        ply+1,
                        tmpLine,
                        false);
       if(tmp > alpha  &&  tmp < beta)
       {
           //повторно сбросим строку
           RESET_MOVE(tmpLine[0]);
           tmp = -PrincipalVariation(-beta,-alpha,
                        ply+1,
                        tmpLine,
                        false);
       }
       UnMakeMove;
```

```
        if(TimeUp) return ...;
        if(tmp > alpha)
        {
          alpha = tmp;
          //запишем строку
          if(alpha < beta)
              SaveLine(move,tmpLine,line);
        }//end if
        move = move->next;
      }//while
      return alpha;
  }
#define MAX_PLY 16
//итеративные углубления
  int Iterative Deepening(
                          PMove bestMove //лучший ход
                         )
  {
     int tmp,
         score;
     //сбросим главную строку изменения
     RESET_MOVE(bestLine[0]);
     for(maxPly = 1; maxPly <= MAX_PLY; maxPly++)
     {
        tmp = PrincipalVariation(-INFINITY,INFINITY,0,
                                 bestLine,true);
        if(TimeUp) break;
        score = tmp;
     }
     //запишем лучший ход
     COPY_MOVE(bestLine[0],*bestMove);
     return score;
  }
```

В дереве перебора может быть масса отрывочного PV, который никогда не достигнет корня дерева. Оценка в потенциальном узле PV больше alpha, но меньше beta. Существует вариант Principal Variation, который просматривает узел с полным окном, пока не найден хоть один ход PV:

```
int PrincipalVariation (int alpha, int beta, int depth)
{
   int oldAlpha;
   if(depth <= 0) return Evaluate();
   oldAlpha = alpha; //запомним alpha
```

```
    Generate();
    while(moveList &&  alpha < beta)
    {
         if(alpha == oldAlpha)
         {
            //не было узла PV
            tmp = - PrincipalVariation (-beta,-alpha,depth-1);
         }else{
            tmp = - PrincipalVariation (-(alpha+1),-alpha,depth-1);
            if(tmp > alpha  &&  tmp < beta)
               tmp = - PrincipalVariation (-beta,-alpha,depth-1);
         }
         if(tmp > alpha) alpha = tmp;
     }//while
    return alpha;
}
```

Если мы осуществляем полный перебор или грубое усилие, то подобные тонкости совершенно не играют роли, и можно просто всегда первый ход рассматривать с полным окном. Если же у нас есть некоторый вариант выборочного статического поиска, то уточнение PV не от корня дерева не прибавляет ума программе.

NegaScout

NegaScout — самый популярный на сегодня алгоритм грубого усилия. Он чрезвычайно прост и дает некоторое (до 50%) ускорение, не внося никакой дополнительной погрешности в вычисление. Он очень хорошо сочетается с современными атрибутами шахматных программ — хеш-таблицами. NegaScout не предусматривает извлечения строки главного изменения или других хитростей. Основная его идея следующая: прежде чем исследовать любой узел, мы его пытаемся сначала отсечь за перебором с нулевым окном (от alpha до alpha+1). Если результат счета улучшает alpha, тогда перебираем снова. Учитывается также один тонкий момент. Если мы перебрали с нулевым окном и получили результат больше, чем alpha, то это — наш минимум, и мы можем смело повторить перебор с пределами не от alpha до beta, а от полученной оценки до beta. Само собой, если этот наш минимум уже сравнялся с beta, то уточнять его не нужно. Можно сразу вернуть beta. Это очень похоже на Principal Variation, за исключением повышения alpha при повторном переборе. Первый ход в каждом узле рассматривается с полным окном. Какой это ход? Это зависит от генератора перемещений. Это может быть ход из хеш-таблицы, хорошее взятие или любой другой ход. Алгоритмом это не определяется. Просто этот ход первый.

```
int NegaScout(int alpha, int beta, int depth)
{
    int   tmp;
    PMove move;
    if(depth <= 0) return Evaluate();
    //начало списка ходов
    move = Generate();
    //первый ход с полным окном
    //////////  first move search /////////
    MakeMove;
    tmp = -NegaScout(-beta,-alpha,depth-1);
    UnMakeMove;
    if(tmp > alpha) alpha = tmp;
    move = move->next;
    //////////  end first move search //////
    while(move   &&   alpha < beta)
    {
       MakeMove;
       tmp = -NegaScout(-(alpha+1),-alpha,depth-1);
       if(tmp > alpha   &&   tmp < beta)
       {
          tmp = -NegaScout(-beta,-tmp,depth-1);
       }
       UnMakeMove;
       if(tmp > alpha) alpha = tmp;
       move = move->next;
    }//while
    return alpha;
}
```

Привожу другое описание:

```
int NegaScout(position p; int alpha, beta)
{
   int a,b,t,i;
   determin successors p_1,...., p_w of p;
   if(w == 0) return Evaluate(p);
   a = alpha;
   b = beta;
   for(i = 1; i <= w; i++)
   {
     t = -NegaScout(p_i,-b,-a);
     if(t > a  &&  t < beta  &&  i > 1  && d < maxDepth-1)
```

```
        a = -NegaScout(p_i,-beta,-t);
      a = max(a,t);
      if(a == beta) return a;
      b = a + 1;
    }
    return a;
}
```

Предлагаемый вариант ждет сходимости `a == beta`. На практике это довольно опасно. Перебор продолжается, пока alpha меньше beta. Ключевой идеей NegaScout является попытка отсечь узел перебором с нулевым окном, прежде чем рассматривать его с полным alpha-beta диапазоном. Так как мы не отслеживаем явно Principal Variation и осуществляем грубое усилие, то можно и не перебирать первый ход с полным окном. В некоторых случаях будет чуть медленнее, в некоторых — чуть быстрее, а в среднем — то же самое.

```
int Search(int alpha, int beta, int depth)
{
    int score = -INFINITY,
        tmp;
    if(depth <= 0) return Evaluate();
    Generate();
    while(moveList   &&   alpha < beta)
    {
       MakeMove;
       tmp = -Search(-(alpha+1),-alpha,depth-1);
       if(tmp > alpha   &&   tmp < beta)
       {
          tmp = -Search(-beta,-tmp,depth-1);
       }
       UnMakeMove;
       if(tmp > score) score = tmp;
       if(score > alpha) alpha = score;
    }//while
    return score;
}
```

Здесь есть два тонких момента. Это использование результата отсечки при повторном переборе:

```
if(tmp > alpha   &&   tmp < beta)
{
   tmp = -NegaScout(-beta,-tmp,depth-1);
}
```

Если бы мы не использовали отсечку, надо было бы написать:

```
if(tmp > alpha    &&    tmp < beta)
{
   tmp = -NegaScout(-beta,-alpha,depth-1);
}
```

Также функция возвращает не alpha, а результат отсечки score. Он может быть и меньше alpha. Это фрагмент alpha-beta алгоритма с амортизацией отказов. На практике от этих тонкостей толка никакого. Они только усиливают нестабильность поиска. Хотя теоретически нельзя отрицать возможность ускорения в исключительном случае. Без этих тонкостей программный код выглядел бы следующим образом:

```
int Search(int alpha, int beta, int depth)
{
    int tmp;
    if(depth <= 0) return Evaluate();
    Generate();
    while(moveList    &&    alpha < beta)
    {
       MakeMove;
       tmp = -Search(-(alpha+1),-alpha,depth-1);
       if(tmp > alpha    &&    tmp < beta)
       {
          tmp = -NegaScout(-beta,-alpha,depth-1);
       }
       UnMakeMove;
       if(tmp > alpha) alpha = tmp;
    }//while
    return alpha;
}
```

Все эти манипуляции с alpha-beta окном мы проводили в преузле, если так можно выразиться. Если смотреть из следующего узла, то попытки отсечь за alpha в преузле выглядели бы как попытки отсечь за beta в нашем узле. Это немного неудобно, но суть дела от этого не меняется.

```
int Search(int alpha, int beta, int depth)
{
   int score = -INFINITY,
       tmp,
       newAlpha,
       searchNumber = 0;
   PMove move,first;
```

```
        if(depth <= 0) return Evaluate();
        newAlpha = beta - 1;
        //начало списка перемещений
        move = first = Generate();
loop:
        while(moveList)
        {
          MakeMove;
          tmp = -Search(-beta,-newAlpha);
          UnMakeMove;
          if(tmp > score) score = tmp;
          if(score > newAlpha) newAlpha = score;
          if(newAlpha >= beta) return newAlpha;
        }//while
        searchNumber++;
        //до этой точки программа доходит редко
        if(score > alpha  &&  searchNumber == 1)
        {
           beta = score;
           score = -INFINITY;
           newAlpha = alpha;
           move = first;
           goto loop;
        }
        return score;
}//function
```

Эту задачу можно рассматривать как своего рода упражнение для ума. Попробуйте ответить: чем этот алгоритм отличается от NegaScout? Чего я только ни слышал по этому поводу.

Генерация и сортировка перемещений

Рассмотрим схематично генерацию перемещений. Схем множество, мы рассмотрим самую простую, для иллюстрации основных принципов.

Списки фигур

Чтобы генерировать перемещения для некоторой позиции, нужно, прежде всего, найти свои фигуры. Каждый раз сканировать весь массив доски неэффективно. Кроме того, фигуры лучше брать в определенной последовательности, например:

1) король;

2) ферзь;

3) ладьи;

4) слоны и кони;

5) пешки.

Это нужно, во-первых, для сортировки взятий (об этом позже). Во-вторых, когда перемещения наиболее ценных и сильных фигур сгенерированы сначала, расчет может получиться более эффективным. Например, когда король под шахом, и перемещения короля идут первыми, этого в большинстве случаев достаточно, чтобы вызвать отсечку за beta. Также ферзь, как наиболее ценная и сильная фигура, скорее найдет большее приращение материальной и стратегической оценки, чем пешка. Для решения этих задач можно использовать списки фигур. Каждый элемент списка представляет собой часть простого связного списка с указателем на следующий элемент и некоторой служебной информацией, касающейся конкретной фигуры. Вот простейший элемент списка:

```
typedef struct __TItem{
   int x,y;                    //координаты фигуры
   struct __TItem* next;    //следующий элемент списка
}TItem,*PItem;
```

Списки инициализируются один раз перед началом счета и постоянно обновляются при переборе. Если некоторая фигура переместилась, это должно быть отражено в элементе списка, если фигура съедена, элемент списка должен быть временно удален. Для удаления элементов списков можно ввести вспомогательный массив, где в элементе, соответствующем фигуре, содержится ссылка на указатель элемента списка для данной фигуры. Например:

```
unsigned char   pos[8][8];      //доска с фигурами
PItem*  ptrPos[8][8];   //ссылки на указатели
PItem   whiteList;      //список белых фигур
PItem   blackList;      //список черных фигур
```

Используя ссылки на указатели, можно легко удалить нужный элемент из связного списка и восстановить его затем. Например, белого ферзя съели в клетке [4:4].

```
PItem* p     = ptrPos[4][4];
ptrPos[4][4] = 0;
PItem saveP  = (*p);
(*p) = (*p)->next;
{ .... }
```

Если нужно восстановить элемент списка:

```
saveP->next = (*p);
(*p) = saveP;
ptrPos[4][4] = p;
```

Это, в общем-то, тривиальные операции со связными списками. Рассмотрим инициализацию связных списков:

```
#define KING   1
#define QUEEN  2
#define ROOK   3
#define BISHOP 4
#define KNIGHT 5
#define PAWN   6
#define BLACK  64
#define WHITE  128
#define FIGURE(f)  ((f) & 7)                 //возвращает код фигуры
#define COLOR(f)   ((f) & (BLACK | WHITE))   //возвращает цвет фигуры

//вспомогательные списки белых по коду фигуры
PItem  wList[7] = {0,0,0,0,0,0,0};
PItem* pWList[7] = {&wList[0],&wList[1],&wList[2],&wList[3],
                    &wList[4],&wList[5],&wList[6]};
//черных
PItem  bList[7] = {0,0,0,0,0,0,0};
PItem* pBList[7] = {&bList[0],&bList[1],&bList[2],&bList[3],
                    &bList[4],&bList[5],&bList[6]};
static TItem data[40]; //буфер памяти под элементы списков
int count = 0;
for(int y = 0; y < 8; y++) for(int x = 0; x < 8; x++)
{
    int f = pos[y][x];
    if(!f) continue;
    PItem p = data[count];
    count++;
    p->x = x;
    p->y = y;
    p->next = 0;
    if(COLOR(f) == WHITE)
    {
       *pWList[FIGURE(f)] = p;
        pWList[FIGURE(f)] = &p->next;
    }else{
       *pBList[FIGURE(f)] = p;
        pBList[FIGURE(f)] = &p->next;
    }
}
//соединяем концы подсписков с началом предыдущих
for(int n = 5; n >= 1; n--)
```

```
{
  *pWList[n] = wList[n+1];
  *pBList[n] = bList[n+1];
}
whiteList = wList[KING];
blackList = bList[KING];
```

В данном коде используются ссылки на указатели, чтобы соединить концы подсписков с началом предыдущих. Подсписки ведутся по коду фигуры. Например, для коней в подсписке может быть два элемента. Конец подсписка коней будет соединен с началом подсписка пешек, конец подсписка слонов — с началом подсписка коней и т. д. Глобальные списки для белых и черных указывают на первые элементы, т. е. на королей. Осталась еще инициализация вспомогательного массива ссылок на указатели для удаления фигур из списков при взятии:

```
memset(ptrPos,0,sizeof(ptrPos));
PItem* pp = &whiteList;
while(*pp)
{
  ptrPos[(*pp)->y][(*pp)->x] = pp;
  pp = &(*pp)->next;
}
pp = &blackList;
while(*pp)
{
  ptrPos[(*pp)->y][(*pp)->x] = pp;
  pp = &(*pp)->next;
}
```

Данная задача может быть решена иначе. Фигуры из списков можно и не удалять, а хранить дополнительную информацию, позволяющую определить, что фигуры нет. Тут надо учесть, что если элемент списка не удален, а фигура покинула поле, то может произойти повторная генерация перемещений. Списки инициализировать можно и значительно проще, если учесть, что это делается один раз, и никакая особая скорость здесь не нужна:

```
whiteList = blackList = 0;
for(int n = PAWN; n >= KING; n--)
{
   for(int y = 0; y < 8; y++) for(int x = 0; x < 8; x++)
   {
      int f = pos[y][x];
```

```
      if(FIGURE(f) != n) continue;
      PItem p = data[count++];
      p->x = x;
      p->y = y;
      if(COLOR(f) == WHITE)
      {
         p->next = whiteList;
         whiteList = p;
      }else{
         p->next = blackList;
         blackList = p;
      }

   }
}
```

Списки фигур являются очень эффективным средством для оптимизации генерации перемещений. Схему со списками можно упростить до предела. Инициализированные списки фигур хранятся в массиве `SquareData`. Максимальное их количество 32, т. к. фигур у противников по 16. Массив игры для счета содержит не фигуры, а номера описателей фигур в массиве `SquareData` или указатели на них. Описатель перемещения получается совсем простым:

```
TMove = record
    N1,N2:byte; {откуда, куда пошла фигура}
end;
{массив игры}
pos:array[0..16*16-1] of byte;
```

Поле 8 × 8 находится в центре массива, а края заполнены флагом переполнения в виде десятичного числа 32. Каждый байт массива игры содержит вариант:

- 0 — нет фигуры;
- 2 — флаг выхода за пределы поля;
- 0..31 — номер элемента списка фигур.

Остаются свободными два старших бита в байте. Их можно использовать для выбора цвета: 64 — черная фигура, 128 — белая.

Элемент списка фигур содержит следующую информацию:

```
TSquare = record
   N:integer;       {координата фигуры}
   F:integer;       {сама фигура}
```

```
    enable:boolean; {если false, фигура съедена}
    isMove:boolean; {фигура перемещалась хоть один раз}
end;
```

Чтобы перебрать все свои фигуры, нужно просто просмотреть список своих фигур. После инициализации фигуры белых в массиве `SquareData` будут располагаться так: King, Queen, Rook, Rook, Bishop, Bishop, Knight, Knight, Pawn. Фигуры можно просмотреть хоть по убыванию весов, хоть по возрастанию.

Данная схема представления данных является одной из возможных схем. Выбор представления данных в большой степени дело вкуса программиста. Некоторую трудность при такой схеме составит написание функций `MakeMove` и `UnMakeMove`. Описатель перемещения содержит только поля, откуда и куда пошла фигура. Нужно отметить, что эти поля и сама фигура содержат всю необходимую информацию для определения типа хода, а функция `MakeMove` не является критичной по времени функцией программы, и несколько лишних операций сравнения не имеют значения. Вся остальная организация программы при таком описателе перемещения существенно упростится. Массив игры — это просто строка проигранных перемещений, классический дебют из библиотеки — тоже просто строка, и т. д. Объем кода программы значительно сокращается.

Если взяли какую-то фигуру, на доске в ее клетке запишем 0, а в элементе списка фигур сбросим флаг `enable`, чтобы для парных фигур не было повторной генерации перемещений.

Функция `MakeMove`, когда делается ход, может заполнить некоторую вспомогательную структуру, содержащую всю информацию для восстановления позиции с помощью функции `UnMakeMove`. Единственная сложность обстоит с флагом `isMove`, перемещалась ли фигура. Но если мы имеем строку игры, эту трудность можно обойти. Хочется отметить, что представление данных в шахматах является довольно сложной задачей, и до сих пор единственно верного решения не существует.

Сортировка взятий (MVV/LVA)

Это очень важная тема для шахмат. Взятия — это ходы, дающие наиболее значительное приращение оценки. Для ускорения alpha-beta поиска они должны рассматриваться первыми. Во многих позициях не обязательно генерировать все перемещения, достаточно рассмотреть взятия, и они уже вызовут отсечку за beta. В шахматах есть еще статический поиск, где взятия рассматриваются до конца. Эта тема будет рассмотрена позже, пока лишь мы должны знать, что для статического поиска порядок ходов имеет наиважнейшее значение, иначе это вызовет взрыв дерева поиска, и программа будет считать недопустимо долго. Мы уже рассматривали alpha-beta и пом-

ним, что этот алгоритм работает принципиально быстрее при наилучшем порядке ходов. Если N — число позиций, которые программа должна рассмотреть при полном переборе (N равно W в степени D, где W — примерное количество ходов в одной позиции, D — глубина просчета), то alpha-beta с наилучшим упорядочиванием ходов рассмотрит $\sqrt{N}$ позиций. Скорость счета зависит от мощности процессора, порядка ходов, задержки программы в каждой позиции. Задержка обуславливается многими факторами, главный из которых — генератор ходов. Если бы мы написали программу, которая не задерживалась бы в одной позиции (т. е. генератор ходов был бы совмещенным с функцией счета), мы повысили бы количество рассматриваемых в секунду позиций. Может, нет смысла сортировать перемещения? Давайте рассмотрим следующий пример. Допустим, мы считаем на 8 полуходов, и среднее количество ходов в одной позиции равно 40. Допустим, наша программа с сортировкой ходов рассматривает 100 тыс., а без сортировки — 2 млн позиций в секунду.

При наилучшем порядке ходов медленная программа с сортировкой будет считать достаточно быстро, т. к. размер дерева перебора качественно меньше. Заметьте, что alpha-beta с наихудшим порядком ходов переберет столько же позиций, сколько и полный переборщик. Это значит, что программа в реальном времени не сможет сосчитать на 8 полуходов при наихудшем порядке перемещений. Чтобы сосчитать за 65 секунд на 8 полуходов, она должна в секунду рассматривать примерно 100 млрд позиций в секунду. Даже DeepBlue не обладает подобной мощью. Но что такое 8 полуходов? Считать надо значительно дальше. Хотя задержка в каждой позиции и влияет на скорость, тем не менее порядок ходов намного важнее.

Упорядочивая перемещения, можно просчитать только до некоторой глубины, а если учесть, что идеальный порядок ходов недостижим, полный перебор на 10 полуходов выглядит вообще довольно проблематичным. Чтобы считать на такую глубину, нужно применять методы подрезки дерева перебора. В шахматах, чтобы программа играла на мастерском уровне, нужна глубина перебора 10—12 полуходов, как минимум. Дальше считается не все (об этом позднее). Нужно достичь невозможного.

Итак, мы убедились, что сортировка перемещений важна, особенно сортировка взятий. В форсированных вариантах сортировка взятий имеет принципиальное значение. Как же осуществляется сортировка взятий? Первыми должны идти наиболее ценные взятия. Например, сначала взятия ферзя, потом ладьи, потом слона и коня, потом пешек. Если взяли короля — конец просчета. Причем, замечено, что потенциально лучше те взятия, когда наименее ценная фигура пытается захватить наиболее ценную. Например для взятий ферзя: сначала будут рассмотрены пешка—ферзь, потом слон(конь)—ферзь, ладья—ферзь, ферзь—ферзь, король—ферзь. Этот алгоритм называется MVV/LVA (наиболее ценная жертва — наименее ценный

нападающий). Как же его осуществить на практике? У нас уже есть инициализированные списки фигур по убыванию веса фигур, причем код фигуры косвенным образом характеризует вес фигуры. Сначала в списке идет король (1), потом — ферзь (2), потом — ладьи (3), потом — кони (4), потом — слоны (5), потом — пешки (6). В генераторе ходов используется массив связных списков по коду фигуры. Взятия вставляются в начало каждого списка. В списке пешек, например, первым будет вставлено взятие король—пешка (наихудшее). Потом его вытеснит ферзь—пешка, и т. д. После проверки всех взятий просматриваются все списки, начиная с наибольших взятий. Вот код, иллюстрирующий сказанное:

```
//списки взятий
PMove eatList[7] = {0,0,0,0,0,0,0};
//буфер для описателей взятий
TMove data[300];
PMove ptr = &data[0]; //первый свободный описатель
//возвращает 1, если выход за пределы доски
#define OutBoard(x,y) ((unsigned)(x) >= 8 || (unsigned)(y) >= 8)
//просматриваем список наших фигур
PItem item = whiteList; //наши фигуры белые
while(item)
{
   int f = pos[item->y][item->x]; //фигура
   switch(FIGURE(f))
   {
     case QUEEN:{
          static int addXQueen[8] = {1,-1,1,-1,1,-1,0,0};
          static int addYQueen[8] = {1,-1,-1,1,0,0,1,-1};
          for(n = 0; n < 8; n++)
          {
            fx = item->x + addXQueen[n];
            fy = item->y + addYQueen[n];
            while(!OutBoard(fx,fy))
            {
              eatF = pos[fy][fx];
              if(!eatF)
              {
                 fx += addXQueen[n];
                 fy += addYQueen[n];
                 continue;
              }
              if(COLOR(eatF) != BLACK) break;
              //не съели ли короля?
              if(FIGURE(eatF) == KING) return INFINITY-ply;
```

```
                //заполним описатель
                ptr->x = fx;
                { ... } //все поля описателя перемещения
                //вставим в начало списка
                ptr->next = eatList[FIGURE(eatF)];
                eatList[FIGURE(eatF)] = ptr;
                ptr++; //следующий свободный описатель
                break;
              }//while
            }//for
      }break;
      case BISHOP:{
       //...
      }break;
      case ... //все фигуры
    }//switch

    item = item->next;//следующая фигура
}//while

//просмотрим все взятия, начиная с наибольших
for(n = QUEEN; n <= PAWN; n++)
{
  {...} //необходимая инициализация переменных
  MakeMove(...);
  score = -AlphaBeta(-beta,-alpha,...);
  UnMakeMove(...);
  {...} //обработка результата просчета
}
```

Данный код требует некоторого пояснения. Массив `eatList` содержит начала списков взятий. Номер списка в массиве соответствует коду взятой фигуры. Массив `data` используется в качестве буфера памяти для организации связных списков. Это стековая переменная, и собственно элементы списков содержатся в буфере. В данном примере генерация взятий объединена с функцией счета, нет отдельной функции генерации перемещений. Буфер `data`, таким образом, располагается в стеке функции счета alpha-beta. После поиска взятий и помещения их в списки в отсортированном порядке эти списки просматриваются, начиная с наибольших взятий, и сразу рекурсивно вызывается функция `AlphaBeta` для расчета данного перемещения. Генерацию взятий можно выполнить и в виде отдельной функции. Тогда полученные списки надо соединить, начало одного с концом другого. Макрос `OutBoard` возвращает 1, если есть выход за пределы доски. Для уменьшения

операций сравнения знаковый тип приводится к беззнаковому. Здесь небольшая хитрость двоичной арифметики: если −1 со знаком привести к беззнаковому, то получится число, явно большее, чем 7 — максимальный допустимый индекс. В качестве примера приводятся только перемещения ферзя. Массивы `addXQueen` и `addYQueen` содержат приращения при движении ферзя по лучу с номером n. Это, конечно, самый простой способ получения перемещений.

В реальной программе лучше ввести еще один список, куда будут заноситься взятия последней ходившей фигуры противника. Этот список должен рассматриваться первым. Во многих позициях (особенно — острых, тактических позициях) достаточно взять эту фигуру, и дальше можно не считать. Также превращение пешки можно приравнять к взятиям, т. к. оно изменяет материальную оценку. В форсированных вариантах, в глубине горизонта, применяют иногда другой алгоритм — статистической оценки *(см. главу 4)*. Он ищет только победившие или равные взятия. Например, пешка—ферзь и ферзь—ферзь будут рассмотрены, а ферзь—пешка — нет. Это дает только очень приблизительную оценку позиции горизонта. Нужно учесть, что alpha в форсированных вариантах повышается текущей оценкой. Подробнее об этом будет сказано позже.

В профессиональных программах при генерации взятий все силы брошены на немедленное получение перемещения, чтобы программа не ждала, пока генерируются все взятия. Это требует изощренного кода в функции `MakeMove`, делающей перемещение в следующую позицию. Она делает некоторые вычисления, чтобы было ясно сразу, кто может бить данную фигуру. Например, взятие последней ходившей фигуры во многих случаях сразу дает отсечку за beta. Если у нас при ходе скорректированы вспомогательные структуры данных, мы можем сразу ответить на вопрос: бьет ли кто-нибудь эту фигуру, и попробовать это перемещение первым. Если мы в форсированном поиске взятий, то взятия некоторых фигур можно и не брать, если приращение оценки не улучшает alpha. Хочется отметить, что программа в погоне за профессионализмом может стать невероятно сложной в неважных частях. Самое главное — это порядок ходов. Это дает принципиальное ускорение. Судите сами: за то же время можно просчитать в два раза глубже, при наилучшем порядке.

Сортировка простых перемещений

Простые перемещения тоже нужно сортировать. Если у простых перемещений нет приращения стратегической оценки (в случае, если мы считаем только материал), то все они одинаковы, с точки зрения alpha-beta, и сортировать их незачем. Как же осуществляется сортировка простых перемещений? У нас есть функция `Evaluate`, которая статически оценивает позицию. Вызывать ее для каждого перемещения в генераторе ходов, чтобы по-

лучить величину приращения оценки при этом перемещении, слишком большая роскошь. Тут возможны два пути:

- Если в программе материальная и позиционная оценка вычисляются пошагово (сделали ход — подсчитали изменение), то все очень просто. Мы легко можем узнать приращение стратегической оценки для данного хода.
- Если оценка позиции в программе такова, что не может быть вычислена пошагово, нужно определять примерное приращение стратегической оценки для данного хода.

Можно делать три списка (по аналогии с сортировкой взятий). В первый список заносятся худшие ходы (приращение оценки меньше 0), во второй — средние (незначительное приращение), в третий — лучшие. Этого, как правило, оказывается достаточно, чтобы ускорить счет в несколько раз. Конец списка соединяется с началом предыдущего и получается один список. Для отслеживания конца списка можно использовать ссылку на указатель. Вставка осуществляется всегда в конец списка, а ссылка на указатель передвигается при каждой вставке и всегда указывает на конец. Начало списка — простой указатель. Если генератор ходов объединен с функцией счета, то список лучших ходов можно не вести, а тотчас же выполнять счет, как только нашли лучший ход. Шахи являются, бесспорно, самыми лучшими из простых ходов, но специально их выявлять в генераторе ходов слишком накладно. При использовании хеш-таблиц лучшие ходы все равно будут запоминаться, и это не принципиально. Для сортировки простых перемещений используются также различные эвристики: эвристика убийцы (Killer heuristic), эвристика истории лучших ходов (History heuristic), таблица перестановок, итеративные углубления. Это все будет описано позже. Пока мы лишь рассмотрели сортировку простых перемещений по значению, что, несмотря на простоту, качественно ускоряет счет. Для сравнения: NegaScout разгоняет максимум на 50%, а простая сортировка перемещений — в несколько раз.

Существует еще один способ построения программы. Получили хорошее перемещение — сразу же обсчитываем и не ждем генерации всех перемещений. Такая программа содержит множество вложенных функций. Со списками фигур такую схему построить нетрудно. Скажем, для взятий мы начинаем перебирать наш список с наименьших фигур. Если фигура может бить последнюю ходившую фигуру противника (для этого используется прием, аналогичный детектору шахов), то выстраивается линия до нее (для дальнобойных фигур). Если ход корректен, то его сразу обсчитываем, потом пробуем лучший ход соседа по узлу, потом — остальные взятия. Мы просматриваем список фигур противника, начиная с наибольших, и пытаемся бить нашей наименьшей фигурой. Нашли перемещение — сразу же в следующую позицию. Простые перемещения тоже можно брать со смыслом.

Если расстояние между нашим ферзем и королем противника опасно сократилось или проходная пешка — обсчитываем сразу. Если плохое перемещение — добавляем в список и рассмотрим в конце. Генерация перемещений должна быть разумной и в то же время быстрой.

Испытанный метод сортировки перемещений

Существует старый и простой способ сортировки, популярный на протяжении многих лет. Перемещения генерируются в некотором буфере. У каждого описателя перемещения есть поле, назовем его `sortValue`. Это поле содержит значение данного перемещения для сортировки. Применяют аналог пузырьковой сортировки: весь массив не сортируют сразу, а подкачивают вверх только одно самое лучшее перемещение. Функция подкачки получила даже традиционное название `Pick`.

Вот пример кода:

```
Const
 MaxPly = 100;

Type
 TMove = record
    from,      {откуда пошла фигура}
    to,        {куда}
    piece,     {код фигуры}
    kind:byte;{тип хода}
    sortValue:LongInt; {поле сортировки}
 end;

var
 Tree:array[0..MaxPly-1] of TMove; {буфер для перемещений}
 TreeCnt:array[0..MaxPly-1] of integer; {начало-конец списка перемещений}

procedure Pick(N1,N2:integer);
var
 maxV:TScore;
 maxN,N:integer;
 tmp:TMove;
begin
  maxV := Tree[N1].mSortValue; maxN := N1;
  N := N1 + 1;
  while N < N2 do begin
    with Tree[N] do
      if SortValue > maxV then begin
         maxV := mSortValue;
```

```
        maxN := N;
      end;
    N := N + 1;
  end;
  if maxN <> N1 then begin
    tmp := Tree[N1]; Tree[N1] := Tree[maxN]; Tree[maxN] := tmp;
  end;
end;
function Search(alpha,beta,ply,depth,side,xside:integer):integer;
var
 mCnt,tmp:integer;
begin
  if (depth < 0) or (ply >= MaxPly) then begin
     Search := Evaluate(side);
     exit;
  end;
  TreeCnt[ply+1] := TreeCnt[ply];
  GenerateAllMoves(side,ply));
  mCnt := TreeCnt[ply];
  while (mCnt < TreeCnt[ply+1]) and (alpha < beta) do begin
     Pick(mCnt,TreePnt[ply+1]);
     tmp := -Search(-beta,-alpha,depth-1,ply+1,xside,side);
     if tmp > alpha then alpha := tmp;
     mCnt := mCnt + 1;
  end;{while}
  Search := alpha;
end;
procedure IterativDeep;
var
 depth:integer;
begin
   TreeCnt[0] := 0;
   depth := 1;
   while (depth < MaxPly) and not TimeUp do
   begin
     Search(-INFINITY,INFINITY,0,depth,cWhite,cBlack);
     depth := depth + 1;
   end;
end;
```

Перемещения откладываются в буфере `Tree`. Массив `TreeCnt` содержит индекс первого пустого элемента для данного `ply`. Если мы, например, сгенерировали перемещения для глубины 0 (`ply = 0`), то индекс первого элемента будет `TreeCnt[0]`, а индекс последнего (`TreeCnt[1]-1`). Функция `Pick` для

скорости даже не меняет элементы местами в массиве, как при пузырьковой сортировке. Она только запоминает индекс элемента с наибольшим значением и, если это не первый элемент, меняет его местами с первым. При alpha-beta алгоритме очень часто не нужны все перемещения, и совершенно не обязательно сортировать весь буфер. Лучший ход может сразу вызвать отсечку за beta, и расчет этого узла прекратится. В качестве поля для сортировки для взятий может использоваться вес фигуры минус код взявшей фигуры, для соблюдения порядка "наиболее ценная жертва — наименее ценный нападающий". Перед взятиями идет ход из хеша. После взятий может быть один или два лучших хода для соседнего узла. Далее простые перемещения могут сортироваться по значению (приращение оценки при ходе), или может использоваться значение из таблицы истории. Для самого быстрого варианта взятия должны генерироваться сначала отдельной функцией, а тихие перемещения потом. Если поле сортировки нулевое (например, для таблицы истории), то оставшиеся в буфере перемещения можно брать без сортировки.

0 × 88

Это старое и хорошо известное представление позиции. Поле представлено массивом не 8 × 8, а 8 × 16. Это дает возможность контролировать выход за пределы доски всего одной операцией сравнения:

```
if( (N & 0x88)  != 0)
{
     //вышли за пределы поля
 }
```

Если клетки за пределами доски обозначить F, то массив будет выглядеть:

```
TSquare pos[8*16] =
{
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
0,0,0,0,0,0,0,0,F,F,F,F,F,F,F,F
};
```

Преимуществом данного представления является то, что размер есть степень числа 2, и если нужно перевести координату N в X,Y массива 8 × 8

(стандартного представления), то это можно сделать очень быстро на двоичном уровне:

```
Y = N >> 4;          // N / 16
X = N & 15;          // N % 16
```

Приращения при движении фигуры можно представить обычной строкой с 0 в конце, например:

- приращения при движении ферзя и короля

```
int delQueen[] = {-1,1,-16,16,-17,-15,17,15, 0};
```

- при движении ладьи

```
int delRook[] = {-1,1,-16,16, 0};
```

Мы можем объединить генерацию перемещений для нескольких фигур сразу.

Недостатком является сложная структура массива атаки. Кроме того, представление 16 × 16 вообще не нуждается в контроле выхода за пределы поля.

BitBoard

BitBoard — это изощренная битовая техника генерации перемещений и оценки позиции. Она использует следующее понятие: массив игры в шахматах представлен 64 клетками. Взяв 64-битовое число без знака, можно получить некоторое битовое представление позиции, если биты, соответствующие фигурам позиции, установить в 1. Например, все белые пешки в битовой нотации можно представить одним числом:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
```

Первый ход e2—e4 будет выглядеть так:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 1 0 1 1 1
0 0 0 0 0 0 0 0
```

Исходный массив игры выглядит как следующий:

```
0  8  0 0 0 0 0 0 0 0
1  7  0 0 0 0 0 0 0 0
2  6  0 0 0 0 0 0 0 0
3  5  0 0 0 0 0 0 0 0
4  4  0 0 0 0 0 0 0 0
5  3  0 0 0 0 0 0 0 0
6  2  0 0 0 0 0 0 0 0
7  1  0 0 0 0 0 0 0 0
      a b c d e f g h
      0 1 2 3 4 5 6 7
```

Перед началом счета у нас есть инициализированные следующие структуры данных:

- списки фигур для белых и черных;
- 64-битовые числа-маски:
  - `AllBlack` — все черные фигуры;
  - `AllWhite` — все белые фигуры;
  - `BlackPawns` — черные пешки;
  - `WhitePawns` — белые пешки;
- массив вычисленных заранее битовых масок для возможных перемещений коня и короля для каждой клетки доски;
- четыре массива для каждой стороны, содержащие данные, сколько на каждой горизонтали, вертикали, восходящей и нисходящей диагоналях находится фигур.

Самым критичным по времени является генерация взятий. Предположим, нам нужно получить все взятия белых пешек. Маска белых пешек следующая:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0
1 1 1 0 0 1 1 1
0 0 0 0 0 0 0 0
```

Пешки бьют по диагонали. Сдвинув это число влево на 9, мы получим все поля, битые пешками налево. Самый левый столбец (`x = 0`) при таком сдвиге может стать самым правым (`x = 7`). Для этого мы должны к результату применить поразрядное логическое И со следующей маской:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 1
```

Назовем эту маску `ROut`. Аналогичная маска для левой стороны будет `LOut`. Тогда, чтобы получить поля, битые белыми пешками, нужно:

```
( (WhitePawn << 9)  &  (~ROut))   |
( (WhitePawn << 7)  &  (~LOut))
```

Применив эту маску с операцией И со всеми черными фигурами, мы получим все взятия белых пешек:

```
PawnCap =   (
             ( (WhitePawn << 9)  &  (~ROut))   |
             ( (WhitePawn << 7)  &  (~LOut))
           )  &  AllBlack;
```

Допустим, это число:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Оно может содержать несколько взятий. Как их извлечь? Это целая область двоичной казуистики. Рассмотрим самый простой способ:

```
X  &  -X
```

Эта операция возвращает самый младший (самый правый) бит числа x. Для иллюстрации этого попробуем вычислить вручную:

```
0010 1000   исходное число
1101 0111   NOT
1101 1000   +1
---------   И с исходным числом
0000 1000   результат
```

Чтобы получить двоичное представление отрицательного числа, нужно взять инвертированное число (1 меняется на 0) и добавить к нему 1.

Чтобы извлечь из `PawnCap` младший бит (одно взятие), нужно

```
move = PawnCap & -PawnCap;
```

Мы получим число:

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Если нужно убрать полученное перемещение из исходной маски, можно

```
PawnCap = PawnCap & (~move);
```

Может возникнуть необходимость не только в маске целевой клетки, но и в ее индексе. Извлечение его усложняет задачу на порядок. Самый простой способ — использовать массив перекодировки от 0 до 65 535. В каждой клетке массива задано, какой индекс соответствует числу с данным единственным установленным битом. Исходная маска 64-разрядная. Это число следует декодировать в четыре приема, т. к. в 64-битовом числе содержится четыре 16-битовых.

Значение клеток массива перекодировки:

Номер бита	Индекс массива	Значение
0	1	15
1	2	14
2	4	13
3	8	12
4	16	11
5	32	10
6	64	9
7	128	8
8	256	7
9	512	6
10	1024	5
11	2048	4
12	4096	3
13	8192	2
14	16384	1
15	32768	0

```
#define BITBOARD unsigned __int64
#define MASK 65535
int Index0_63(BITBOARD move)
{
   BITBOARD bank;
      bank = (move >> 48) & MASK;
      if(bank) return hlat[bank];
      bank = (move >> 32) & MASK;
      if(bank) return hlat[bank] + 16;
      bank = (move >> 16) & MASK;
      if(bank) return hlat[bank] + 32;
      bank = move & MASK;
      if(bank) return hlat[bank] + 48;
      _ASSERT(0);
      return ...;
}
```

Просто, не правда ли?!

Если нам нужно получить взятия короля или коня, мы просто делаем операцию поразрядного И маски перемещений этой фигуры для конкретной клетки и маски всех фигур противника. Для получения перемещений дальнобойных фигур используются повернутые (в прямом и переносном смысле) битовые доски. В данном примере используется упрощенный прием. У нас перед началом вычисления инициализированы 4 массива для каждого цвета фигур. В них содержится информация, сколько фигур в каждой вертикали, горизонтали и диагоналях. Если диагональ восходящая, то сумма координат всех клеток одинакова. Если нисходящая — разность XY одна для всех клеток диагонали. Массивы корректируются пошагово. Например, пошла белая пешка e2–e4. Ее целевые координаты X = 4, Y = 4. В первом массиве в клетке с индексом X увеличим значение на 1. Во втором массиве — в клетке с индексом Y. Остальные два массива нужно корректировать в клетках (X+Y) и (X–Y+8). Соответствующим образом нужно корректировать массивы клеток, откуда переместилась фигура. Далее, например, берем взятия королевы, сканируем вертикаль, только если в первом массиве в клетке X не равно 0. Все остальные лучи берем, если в соответствующих массивах в клетках Y, X+Y, XY+8 не нули. Это просто и достаточно эффективно.

Вращенный BitBoard

Первым описание этого приема дал Эрнст Хайнц. При программировании генератора ходов для шахмат наибольшую сложность представляет собой получение перемещений дальнобойных фигур, особенно их взятий. Рекомендуется не строить линии (например, для слона), а сразу знать, есть ли

в конце линии фигура противника. Для этого требуется много предварительных вычислений. Используют четыре битовые плоскости: нормальную, транспонированную (длина и ширина поменялись местами) и повернутые на 45° по и против часовой стрелки.

```
type squares = (
 a8,b8,c8,d8,e8,f8,g8,h8,
 a7,b7,c7,d7,e7,f7,g7,h7,
 a6,b6,c6,d6,e6,f6,g6,h6,
 a5,b5,c5,d5,e5,f5,g5,h5,
 a4,b4,c4,d4,e4,f4,g4,h4,
 a3,b3,c3,d3,e3,f3,g3,h3,
 a2,b2,c2,d2,e2,f2,g2,h2,
 a1,b1,c1,d1,e1,f1,g1,h1
);

const
__normal:array[0..63] of squares =
(
 a8,b8,c8,d8,e8,f8,g8,h8,
 a7,b7,c7,d7,e7,f7,g7,h7,
 a6,b6,c6,d6,e6,f6,g6,h6,
 a5,b5,c5,d5,e5,f5,g5,h5,
 a4,b4,c4,d4,e4,f4,g4,h4,
 a3,b3,c3,d3,e3,f3,g3,h3,
 a2,b2,c2,d2,e2,f2,g2,h2,
 a1,b1,c1,d1,e1,f1,g1,h1
);

__flipped:array[0..63] of squares =
(
 a8,a7,a6,a5,a4,a3,a2,a1,
 b8,b7,b6,b5,b4,b3,b2,b1,
 c8,c7,c6,c5,c4,c3,c2,c1,
 d8,d7,d6,d5,d4,d3,d2,d1,
 e8,e7,e6,e5,e4,e3,e2,e1,
 f8,f7,f6,f5,f4,f3,f2,f1,
 g8,g7,g6,g5,g4,g3,g2,g1,
 h8,h7,h6,h5,h4,h3,h2,h1
);

__a1h8:array[0..63] of squares =
(
 a8,  b1,c2,d3,e4,f5,g6,h7,
 a7,b8,  c1,d2,e3,f4,g5,h6,
```

```
 a6,b7,c8,  d1,e2,f3,g4,h5,
 a5,b6,c7,d8,  e1,f2,g3,h4,
 a4,b5,c6,d7,e8,  f1,g2,h3,
 a3,b4,c5,d6,e7,f8,  g1,h2,
 a2,b3,c4,d5,e6,f7,g8,  h1,
 a1,b2,c3,d4,e5,f6,g7,h8
);

__a8h1:array[0..63] of squares =
(
 a8,b7,c6,d5,e4,f3,g2,h1,
 a7,b6,c5,d4,e3,f2,g1, h8,
 a6,b5,c4,d3,e2,f1, g8,h7,
 a5,b4,c3,d2,e1, f8,g7,h6,
 a4,b3,c2,d1, e8,f7,g6,h5,
 a3,b2,c1, d8,e7,f6,g5,h4,
 a2,b1, c8,d7,e6,f5,g4,h3,
 a1, b8,c7,d6,e5,f4,g3,h2
);
__a1h8mask:TArray64 =
(
 1,  2,2,2,2,2,2,2,
 1,1,  2,2,2,2,2,2,
 1,1,1,  2,2,2,2,2,
 1,1,1,1,  2,2,2,2,
 1,1,1,1,1,  2,2,2,
 1,1,1,1,1,1,  2,2,
 1,1,1,1,1,1,1,  2,
 1,1,1,1,1,1,1,1
);
__a8h1mask:TArray64 =
(
 1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1, 2,
 1,1,1,1,1,1, 2,2,
 1,1,1,1,1, 2,2,2,
 1,1,1,1, 2,2,2,2,
 1,1,1, 2,2,2,2,2,
 1,1, 2,2,2,2,2,2,
 1, 2,2,2,2,2,2,2
);
```

При перемещении фигуры функция `MakeMove` корректирует все четыре битовые плоскости. Плоскости a1h8 и a8h1 используются для диагональных ли-

ний дальнобойных фигур. Они преобразуют диагонали в горизонтальную последовательность битов. Нужно только вычислить индекс фигуры в развернутой плоскости и применить маску для этой линии. Для вычисления индекса используется массив перекодировки. Рассмотрим схематичный пример для иллюстрации принципа. Для битового представления позиции используем 64-битовый тип без знака:

- в GNU C — unsigned long long;
- в Visual C — unsigned __int64;
- в Delphi — int64.

Примечание

Delphi не имеет 64-разрядного типа без знака, но из-за дружественного компилятора Delphi тип int64 вполне можно использовать.

Мы выбираем тип без знака по двум причинам. Во-первых, если установлен единственный знаковый бит (с номером 63 — самый старший), то число должно быть не 0.

```
typedef unsigned long long I64;
I64 foo;
foo = (I64)1 << 63;
if( foo)
{
 ...
}
```

Некоторые компиляторы могут неоднозначно толковать эту ситуацию. Во-вторых, при типе без знака любой компилятор использует логический сдвиг, а не арифметический.

Пример:

```
typedef unsigned long long I64;
I64 foo = ((I64)1 << 63) >> 63;
```

Мы сдвинули 1 влево на 63, а потом результат сдвинули вправо на 63. Если тип будет со знаком:

```
typedef long long I64;
I64 foo = ((I64)1 << 63) >> 63;
```

В результате мы получим не 1, а 0 × FFFFFFFFFFFFFFFF. Вместо логического сдвига компилятор применил арифметический, и старший знаковый бит не сдвигается, а расширяется. Компилятор Delphi корректно обрабатывает оба случая, и его вполне можно использовать для BitBoard.

Есть еще один неприятный момент в программировании 64-разрядного BitBoard. Дело в том, что хотя тип данных и 64-разрядный, но процессоры,

преимущественно, остаются 32-разрядными, и может понадобиться много инструкций для обработки некоторых команд. Так для Intel-процессоров нет команды прямого сдвига пары регистров. Компилятор может довольно эффективно сдвигать 64-битовое число на фиксированное количество битов, но сдвиг на непостоянное значение, в зависимости от компилятора и полноты обрабатываемой ситуации, для простой инструкции может вылиться в дополнительную операцию сравнения (GNU C) или даже в подпрограмму и несколько операций сравнения (Delphi). Поэтому нужно избегать сдвига на непостоянное значение.

Для осуществления BitBoard следует познакомиться с некоторыми выражениями:

- (b AND −b) оставляет самый младший бит;
- (b AND (b−1)) сбрасывает самый младший бит;
- (b XOR (b−1)) устанавливает в 1 все нулевые биты справа и оставляет самый правый ненулевой бит.

Пример кода для BitBoard:

```
Type
 I64 = int64;
 TBitBoard = record
    case byte of
    1:(bc64:I64);
    2:(bc32:array[0..1] of LongWord);
    3:(bc16:array[0..3] of word);
    4:(bc8:array[0..7] of byte)
 end;
 TBitRec = record
    toIndex,fromIndex,maskY:array[0..63] of byte;
    board:TBitBoard;
 end;

Var
 ByteFirstBitNumber:array[0..128] of byte;
 ByteRotate:array[0..255] of byte;
 Normal,Flipped,A1H8,A8H1:TBitRec;

procedure BitBoardInit;
type squares = (
 a8,b8,c8,d8,e8,f8,g8,h8,
 a7,b7,c7,d7,e7,f7,g7,h7,
 a6,b6,c6,d6,e6,f6,g6,h6,
 a5,b5,c5,d5,e5,f5,g5,h5,
```

```
 a4,b4,c4,d4,e4,f4,g4,h4,
 a3,b3,c3,d3,e3,f3,g3,h3,
 a2,b2,c2,d2,e2,f2,g2,h2,
 a1,b1,c1,d1,e1,f1,g1,h1
);
TArray64 = array[0..63] of byte;
const
__normal:array[0..63] of squares =
(
 a8,b8,c8,d8,e8,f8,g8,h8,
 a7,b7,c7,d7,e7,f7,g7,h7,
 a6,b6,c6,d6,e6,f6,g6,h6,
 a5,b5,c5,d5,e5,f5,g5,h5,
 a4,b4,c4,d4,e4,f4,g4,h4,
 a3,b3,c3,d3,e3,f3,g3,h3,
 a2,b2,c2,d2,e2,f2,g2,h2,
 a1,b1,c1,d1,e1,f1,g1,h1
);

__flipped:array[0..63] of squares =
(
 a8,a7,a6,a5,a4,a3,a2,a1,
 b8,b7,b6,b5,b4,b3,b2,b1,
 c8,c7,c6,c5,c4,c3,c2,c1,
 d8,d7,d6,d5,d4,d3,d2,d1,
 e8,e7,e6,e5,e4,e3,e2,e1,
 f8,f7,f6,f5,f4,f3,f2,f1,
 g8,g7,g6,g5,g4,g3,g2,g1,
 h8,h7,h6,h5,h4,h3,h2,h1
);

__a1h8:array[0..63] of squares =
(
 a8,  b1,c2,d3,e4,f5,g6,h7,
 a7,b8,  c1,d2,e3,f4,g5,h6,
 a6,b7,c8,  d1,e2,f3,g4,h5,
 a5,b6,c7,d8,  e1,f2,g3,h4,
 a4,b5,c6,d7,e8,  f1,g2,h3,
 a3,b4,c5,d6,e7,f8,  g1,h2,
 a2,b3,c4,d5,e6,f7,g8,  h1,
 a1,b2,c3,d4,e5,f6,g7,h8
);
```

```
__a8h1:array[0..63] of squares =
(
 a8,b7,c6,d5,e4,f3,g2,h1,
 a7,b6,c5,d4,e3,f2,g1, h8,
 a6,b5,c4,d3,e2,f1, g8,h7,
 a5,b4,c3,d2,e1, f8,g7,h6,
 a4,b3,c2,d1, e8,f7,g6,h5,
 a3,b2,c1, d8,e7,f6,g5,h4,
 a2,b1, c8,d7,e6,f5,g4,h3,
 a1, b8,c7,d6,e5,f4,g3,h2
);
__a1h8mask:TArray64 =
(
 1,  2,2,2,2,2,2,2,
 1,1,  2,2,2,2,2,2,
 1,1,1,  2,2,2,2,2,
 1,1,1,1,  2,2,2,2,
 1,1,1,1,1,  2,2,2,
 1,1,1,1,1,1,  2,2,
 1,1,1,1,1,1,1,  2,
 1,1,1,1,1,1,1,1
);
__a8h1mask:TArray64 =
(
 1,1,1,1,1,1,1,1,
 1,1,1,1,1,1,1, 2,
 1,1,1,1,1,1, 2,2,
 1,1,1,1,1, 2,2,2,
 1,1,1,1, 2,2,2,2,
 1,1,1, 2,2,2,2,2,
 1,1, 2,2,2,2,2,2,
 1, 2,2,2,2,2,2,2
);
 function RotByte(b:byte):byte;
 var
  N:integer;
  ret:byte;
 begin
   ret := 0;
   for N := 0 to 7 do
    if (b and (1 shl N)) <> 0 then
       ret := ret or (1 shl (7-N));
   RotByte := ret;
 end;
```

```
function Mask(var data:TArray64; index:integer):byte;
var
 X,Y,value:integer;
 ret:byte;
begin
  ret := 0; Y := index div 8; value := data[index];
  for X := 0 to 7 do begin
    if data[Y*8+X] = value then
      ret := ret or (1 shl X);
  end;
  Mask := ret;
end;
procedure FillMaskKingAndKnight;
 const
  addXKnight:array[0..7] of integer = (1,1,-1,-1,2,-2,2,-2);
  addYKnight:array[0..7] of integer = (2,-2,2,-2,1,1,-1,-1);
  addXKing:array[0..7] of integer = (1,-1,1,-1,1,-1,0,0);
  addYKing:array[0..7] of integer = (1,-1,-1,1,0,0,1,-1);
 var
  X,Y,N,newX,newY:integer;
begin
  for Y := 0 to 7 do
   for X := 0 to 7 do begin
     MaskKnight[Y*8+X].bc64 := 0;
     MaskKing[Y*8+X].bc64 := 0;
     for N := 0 to 7 do
     begin
        newX := X + addXKnight[N]; newY := Y + addYKnight[N];
        if word(newX or newY) < 8 then
          with MaskKnight[Y*8+X] do
            bc8[newY] := bc8[newY] or (byte(1) shl newX);
        newX := X + addXKing[N]; newY := Y + addYKing[N];
        if word(newX or newY) < 8 then
          with MaskKing[Y*8+X] do
            bc8[newY] := bc8[newY] or (byte(1) shl newX);
     end;
   end;
end;{function}

var
 N:integer;
 b:byte;
begin {main}
```

```
  for b := 0 to 255 do
     ByteRotate[b] := RotByte(b);

  for N := 0 to 7 do
    ByteFirstBitNumber[1 shl N] := N;
  with normal do begin
   for N := 0 to 63 do begin
     toIndex[N] := byte(__normal[N]);
     fromIndex[N] := byte(__normal[N]);
     maskY[N] := $FF;
   end;
   board.bc64 := 0;
  end;
  with Flipped do begin
   for N := 0 to 63 do begin
     fromIndex[N] := byte(__flipped[N]);
     toIndex[byte(__flipped[N])] := N;
     maskY[N] := $FF;
   end;
   board.bc64 := 0;
  end;
  with A1H8 do begin
   for N := 0 to 63 do begin
     fromIndex[N] := byte(__a1h8[N]);
     toIndex[byte(__a1h8[N])] := N;
     maskY[N] := Mask(__a1h8mask,N);
   end;
   board.bc64 := 0;
  end;
  with A8H1 do begin
   for N := 0 to 63 do begin
     fromIndex[N] := byte(__a8h1[N]);
     toIndex[byte(__a8h1[N])] := N;
     maskY[N] := Mask(__a8h1mask,N);
   end;
   board.bc64 := 0;
  end;
  FillMaskKingAndKnight;
end;
function BitNumber(var b:TBitBoard):integer;
begin
  with b do begin
    if bc32[0] <> 0 then begin
```

```
        if bc16[0] <> 0 then begin
          if bc8[0] <> 0 then BitNumber := ByteFirstBitNumber[bc8[0]]
          else BitNumber := ByteFirstBitNumber[bc8[1]] + 8;
        end else begin
          if bc8[2] <> 0 then BitNumber := ByteFirstBitNumber[bc8[2]] +
          16
          else BitNumber := ByteFirstBitNumber[bc8[3]] + 24;
        end;
    end else begin
        if bc16[2] <> 0 then begin
          if bc8[4] <> 0 then BitNumber := ByteFirstBitNumber[bc8[4]] +
          32
          else BitNumber := ByteFirstBitNumber[bc8[5]] + 40;
        end else begin
          if bc8[6] <> 0 then BitNumber := ByteFirstBitNumber[bc8[6]] +
          48
          else BitNumber := ByteFirstBitNumber[bc8[7]] + 56;
        end;
    end;
  end;
end;{function}
procedure GetMoves(var rec:TBitRec; squareN:integer; var
moves:TBitBoard);
var
 X,Y,index,N:integer;
 line,my,leftBits,rightBits:byte;
begin
   with rec do begin
      index := toIndex[squareN];
      X := index and 7; Y := index shr 3;
      my := byte(1) shl X;
      line := board.bc8[Y] and  not my  and  maskY[index];
      rightBits := line and (my xor (my-1));
      leftBits := line and not rightBits;
      if leftBits <> 0 then begin
        N := fromIndex[ ByteFirstBitNumber[ leftBits and -leftBits] + (Y
        shl 3)];
        with moves do
         bc8[N shr 3] := bc8[N shr 3] or (byte(1) shl (N and 7));
      end;
      if rightBits <> 0 then begin
        rightBits := ByteRotate[rightBits];
        N := fromIndex[ (7-ByteFirstBitNumber[ rightBits and -rightBits])
        + (Y shl 3)];
```

```
        with moves do
          bc8[N shr 3] := bc8[N shr 3] or (byte(1) shl (N and 7));
      end;
   end;
end;
procedure BishopAttack(squareN:integer; var moves:TBitBoard);
begin
   moves.bc64 := 0;
   GetMoves(A1H8,squareN,moves);
   GetMoves(A8H1,squareN,moves);
end;
procedure RookAttack(squareN:integer; var moves:TBitBoard);
begin
   moves.bc64 := 0;
   GetMoves(Normal,squareN,moves);
   GetMoves(Flipped,squareN,moves);
end;
```

Если мы получим маску для всех фигур, атакованных слоном, то, применив эту маску с операцией поразрядного AND к фигурам противника, мы получим все фигуры противника, которые бьет данный слон:

```
BishopAttack(from,moves);
moves := moves and All[xside]; {выделили фигуры противника}
while moves <> 0 do
begin
  item = moves and -moves; {первая атакованная фигура}
  to := BitNumber(TBitBoard(item));
  LinkMove(from,to);
  moves := moves and (moves-1); //очистили младший бит
end;
```

Можно получить также все простые перемещения. Если у нас есть маска для фигур слева на линии и фигур справа на линии, то нетрудно получить простые перемещения. Они лежат посередине между этими масками. Вот фрагмент функции GetMoves с простыми перемещениями дальнобойных фигур:

```
index := toIndex[fSource];
X := index and 7; Y := index shr 3;
my := byte(1) shl X;
line := board.bc8[Y] and  not my  and  maskY[index];
rightBits := line and (my xor (my-1));
leftBits := line and not rightBits;
rightBits := ByteRotate[rightBits];
```

```
leftMask := (leftBits xor (leftBits-1)) and not(leftBits and -
leftBits);
rightMask := (rightBits xor (rightBits-1)) and not(rightBits and -
rightBits);
moves := leftMask and ByteRotate[rightMask] and not my and
maskY[index];
```

Данный код только иллюстрирует принцип и не оптимизирован. Для нормальной и транспонированной плоскости не нужны битовые маски. Лишние промежуточные вызовы функций тоже непозволительная роскошь и т. д.

Для BitBoard совершенно не обязательны списки фигур. Мы можем хранить одну битовую маску для пешек и для королей. Общая маска для белых и для черных выглядит следующим образом:

```
var
 Pieces:array[pKing..pPawn] of TBitBoard;
 All:array[cWhite..cBlack] of TBitBoard;
```

Атака дальнобойных фигур

Давней мечтой шахматных программистов является генерация взятий дальнобойных фигур, не строя линии. Генерация взятий является самым критичным по времени выполнения фрагментом программы. Дело в том, что большинство позиций можно отсечь только взятиями, и даже не обязательно иметь весь список перемещений. Кроме того, взятия в форсированном поиске рассматриваются до конца. Генерация перемещений на глубине 20 намного больше влияет на скорость выполнения программы, чем на глубине 5. Для ускорения получения взятий существует много методик. Мы рассмотрим только один старый и очень простой прием. Идея следующая: основная масса перемещений после хода некоторой стороны не изменяется, за исключением некоторых ходов, связанных с последней ходившей фигурой. Например, если пешка пересекла линию ферзя, то ферзь уже не может бить по этой линии. Остальная масса перемещений осталась без изменений. Изменились только ходы пешки и ферзя, которому она перекрыла линию. Мы сейчас, для простоты, не будем рассматривать, как можно корректировать перемещения всех фигур при ходе, а рассмотрим только дальнобойные фигуры. Введем некоторый массив, где будут представлены все фигуры на доске, независимо от цвета и веса. Их можно представить как ферзи без учета цвета. Даже если это пешка, мы ее будем рассматривать как ферзя, потому что она так же может перекрыть линию дальнобойной фигуре. Ферзь, как известно, объединяет в себе ходы всех дальнобойных фигур. Каждая клетка нашего нововведенного массива будет иметь 8 ссылок на ближайшие фигуры по всем лучам. Наш массив будет иметь размерность не 8 × 8, а 10 × 10. По краям находятся лишние описатели. Даже если по ли-

нии фигуры нет, ссылка все равно будет указывать на некоторый элемент, а именно на ближайший за пределами доски. Если нам нужно убрать некоторую фигуру, мы просто должны восстановить все перекрестные ссылки. Фигура справа, например, будет ссылаться на фигуру слева. Чтобы поставить фигуру в новое место, мы должны просканировать 8 лучей во всех направлениях до первой фигуры или до края доски и навести перекрестные ссылки. В действительности, нам нужно сканировать не 8 лучей, а 4. Когда мы нашли фигуру справа, она уже ссылается на фигуру слева, и ее искать и строить линию не нужно. Таким образом, при каждом ходе мы строим дополнительно только 4 луча, но можем выбирать взятия дальнобойных фигур, не строя линии. Если это ладья, то нужно просто проверить ссылки по вертикали и горизонтали, не указывают ли они на фигуры противника. Обратите внимание, что если одна фигура бьет другую, то все остается без изменений, и никаких лучей строить не нужно. Когда на глубине идет поиск одних взятий, это позволяет очень быстро вычислять результат размена дальнобойных фигур. При некоторой изощренной технике можно брать взятия с фронта. Мы просто просматриваем список фигур противника, начиная с самых ценных, и сразу видно, какие фигуры их бьют. Далее приведен пример кода, дающего возможность брать взятия дальнобойных фигур, не строя линий:

```
/*
              *** Массив атаки ***

        v1       up      v2
          *      |      +
            +    |    +
    left      +  | +      right
          --------------
             +   |  *
           +     |    *
        v21     down   v11
*/
#include <string.h>
typedef unsigned char byte;
//фигуры
typedef enum{EMPTY,KING,QUEEN,ROOK,BISHOP,KNIGHT,PAWN,OUT};
//цвета
typedef enum{CWHITE,CBLACK,COUT};
//клетка основного массива игры
typedef struct{
  byte piece,           //фигура
       color,           //ее цвет
```

```
      pieceTabIndex,    //индекс в списке фигур
      isMove;           //перемещалась ли фигура
}TSquare,*PSquare;
/*
  массив игры
  поле 8×8 находится посередине
  по краям piece = OUT
  приращения при построении линии
      -11   -10   -9
        +    |   +
          +  | +
  -1    --------------- 1
          +  | +
        +    |   +
      11     10   9
*/
typedef TSquare TBoard[10*10];
/////////////// список фигур ////////////
int PieceTab[32+1];
//элемент массива атаки
typedef struct{
         byte   left,
                right,
                up,
                down,
                v1,
                v11,
                v2,
                v21;
}TVec,*PVec;
/*
    массив атаки

*/
typedef TVec TVecPos[10*10];
/////////////// GLOBAL VAR ////////////
static TVecPos _pos;
/*************************************************************
     удаляет элемент с индексом start из массива атаки

*************************************************************/
void RemoveVec(int start)
{
    PVec item;
```

```
    item = &_pos[start];
    _pos[item->right].left = item->left;
    _pos[item->left].right = item->right;
    _pos[item->up].down = item->down;
    _pos[item->down].up = item->up;
    _pos[item->v1].v11 = item->v11;
    _pos[item->v11].v1 = item->v1;
    _pos[item->v2].v21 = item->v21;
    _pos[item->v21].v2 = item->v2;
}//function
/***********************************************
  вставляет элемент с индексом N на старое место

***********************************************/
void BackVec(int start, PVec item)
{
   _pos[item->right].left = start;
   _pos[item->left].right = start;
   _pos[item->down].up = start;
   _pos[item->up].down = start;
   _pos[item->v2].v21 = start;
   _pos[item->v21].v2 = start;
   _pos[item->v1].v11 = start;
   _pos[item->v11].v1 = start;
   _pos[start] = (*item);
}//function
/*****************************************
   вставляет элемент с индексом start в _pos
 *****************************************/
void InsertVec(int start, TBoard gamePos)
{
   TVec item;
   int N;
   //сканируем все лучи и связываем ссылки
   ///// горизонтальные лучи //////////
   N = start+1;
   while(gamePos[N].piece == 0) N++;
   item.right = N;
   item.left = _pos[N].left;
   /////////// вертикальные лучи ///////////////
   N = start-10;
   while(gamePos[N].piece == 0) N-=10;
   item.up = N;
   item.down = _pos[N].down;
```

```
   /////////////// восходящая диагональ /////////
   /*
         +
       +
     O
   */
   N = start-9;
   while(gamePos[N].piece == 0) N-=9;
   item.v2 = N;
   item.v21 = _pos[N].v21;
   /////////////// нисходящая диагональ ////////////
   /*
         *
           *
             *
               O
   */
   N = start-11;
   while(gamePos[N].piece == 0) N-=11;
   item.v1 = N;
   item.v11 = _pos[N].v11;
   BackVec(start,&item);
}//function
//инициализирует массив атаки по позиции
void InitAttack(TBoard gamePos)
{
/*
 0  1  2  3  4  5  6  7  8  9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
*/
  int n,k;
  memset(_pos,0,sizeof(TVecPos));
{////////////  восходящие (/) диагонали
  static int sourceV[] = {10,20,30,40,50,60,70,80,90,
                          91,92,93,94,95,96,97,98,    100};
```

```
  static int destV[] =   { 1,2, 3, 4, 5, 6, 7, 8, 9,
                           19,29,39,49,59,69,79,89,    100};
  for(n = 0; sourceV[n] != 100; n++)
  {
     _pos[ sourceV[n] ].v2 = destV[n];
     _pos[ destV[n] ].v21 = sourceV[n];
  }
}/////////////  end ////////////////////
{/////////////  нисходящие (\) диагонали /////////
  static int sourceN[] = {80,70,60,50,40,30,20,10,0,
                            1, 2, 3, 4, 5, 6, 7, 8,    100};
  static int destN[]   = {91,92,93,94,95,96,97,98,99,
                           89,79,69,59,49,39,29,19,    100};
  for(n = 0; sourceN[n] != 100; n++)
  {
     _pos[ sourceN[n] ].v11 = destN[n];
     _pos[ destN[n] ].v1  = sourceN[n];
  }
}////////// end ///////
  //// вертикали /////
  for(n = 1; n <= 8; n++)
  {
     _pos[n].down = n + 90; _pos[n+90].up = n;
  }
  //// горизонтали //////
  for(n = 10; n <= 80; n+=10)
  {
     _pos[n].right = n + 9; _pos[n+9].left = n;
  }
  //вставим фигуры
  for(n = 0; n < 100; n++)
    if(gamePos[n].piece != EMPTY  &&
       gamePos[n].piece != OUT)
       InsertVec(n,gamePos);
}//function
/*
    обнуляет основной массив игры
    и устанавливает флаги по краям
*/
void ClearMainPos(TBoard board)
{
  int x,y;
  memset(board,0,sizeof(TBoard));
```

```
  for(y = 0; y < 10; y++) for(x = 0; x < 10; x++)
    if(y == 0 || y == 9 || x == 0 || x == 9)
    {
       PSquare p;
       p = &board[y*10 + x];
       p->piece = OUT;
       p->color = COUT;
    }
}
TBoard board;
int main(void)
{
   TVec item;
    ClearMainPos(board);
    InitAttack(board);
    InsertVec(44,board);
    item = _pos[44];
    RemoveVec(44);
    BackVec(44,&item);
    return 0;
}
```

Правила игры

Если кто забыл, напомним основные правила игры.

Начальная позиция изображена на рис. 2.1.

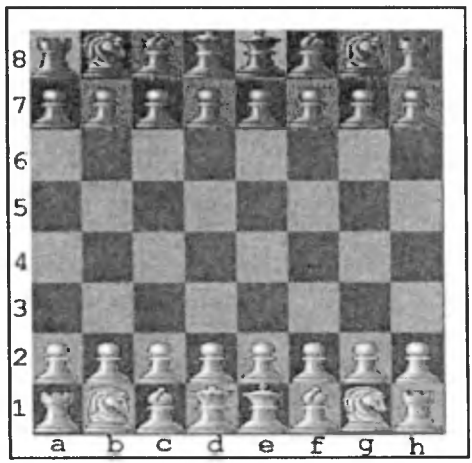


Рис. 2.1. Начальная позиция

Пешка ходит вперед на одну клетку. Бьет вправо и влево по диагонали на одну клетку. Если пешка совершает свой первый ход, и клетка перед ней свободна, она может пойти через клетку. Если пешка пошла через клетку и пересекла поле, битое пешкой противника, пешка противника может взять ее, при этом она становится в поле, которое бьет, т. е. хотя пешка пошла через клетку, ее взяли так, как будто она пошла на одну клетку. Если пешка дошла до конца доски, она может превратиться в любую фигуру, кроме короля. Вот примеры ходов пешки на рис. 2.2 и через клетку на рис. 2.3.

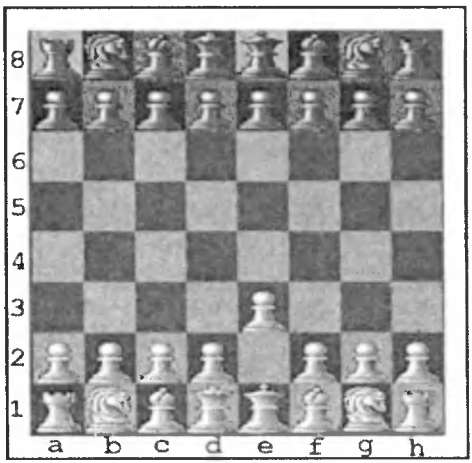


Рис. 2.2. Простой ход пешки e2–e3

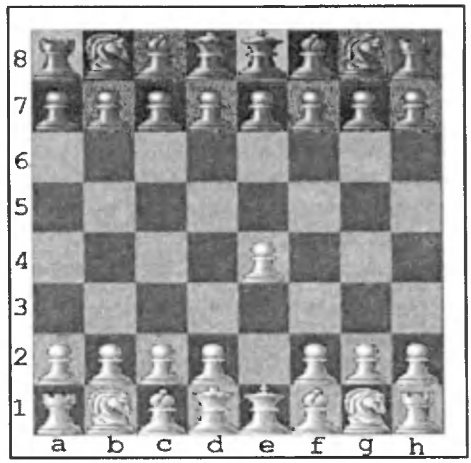


Рис. 2.3. Первый ход пешки через клетку e2–e4

Поля, битые пешкой, показаны на рис. 2.4.

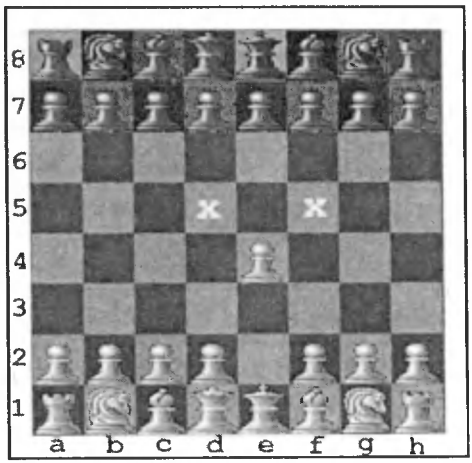


Рис. 2.4. Поля, битые пешкой

На рис. 2.5 белая пешка встанет на f6.

Король ходит на одну клетку во всех направлениях (рис. 2.6).

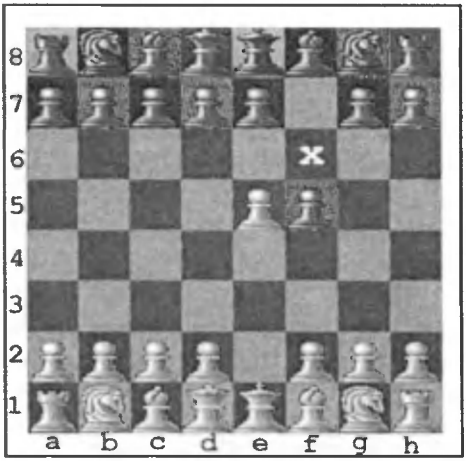


Рис. 2.5. Взятие пешкой через битое поле

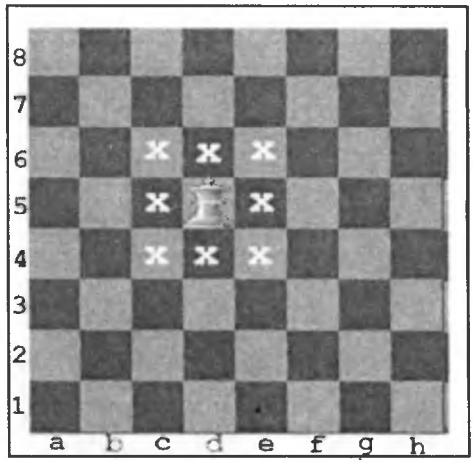


Рис. 2.6. Ходы короля

Если король и ладья не перемещались с начала игры, они могут сделать рокировку. При этом король под шахом рокировку не делает и не должен пересекать и становиться на битое поле. Примеры рокировок показаны на рис. 2.7 и 2.8.

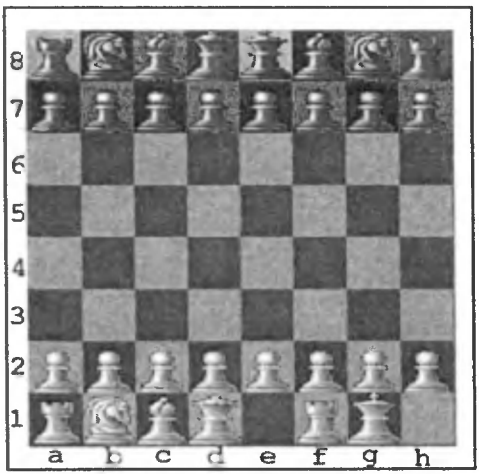


Рис. 2.7. Король переместился e1–g1, ладья —h1–f1

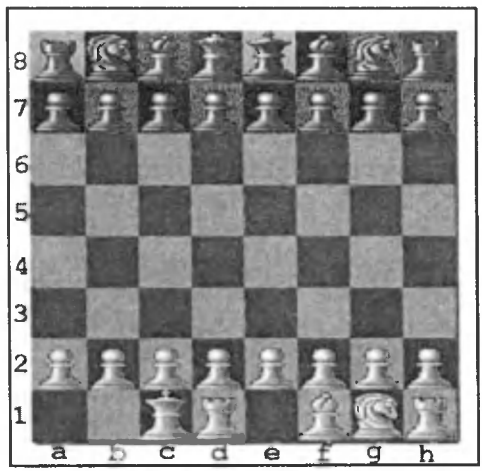


Рис. 2.8. Король переместился e1–c1, ладья —a1–d1

Конь ходит буквой Г (рис. 2.9).

Ходы остальных фигур достаточно просты. Слон ходит по диагонали на любое количество клеток (рис. 2.10).

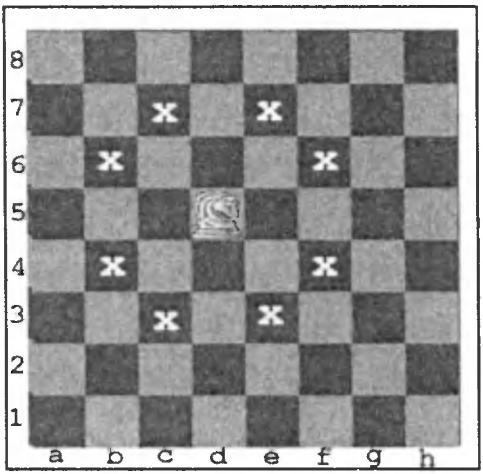


Рис. 2.9. Ходы коня

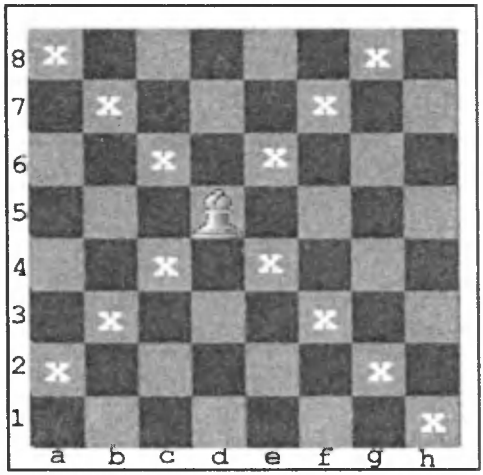


Рис. 2.10. Ходы слона

Ладья ходит по вертикали и горизонтали на любое количество клеток (рис. 2.11).

Королева (ферзь) — самая сильная фигура. Она ходит во всех направлениях, и мат, как правило, ставится при ее помощи (рис. 2.12).

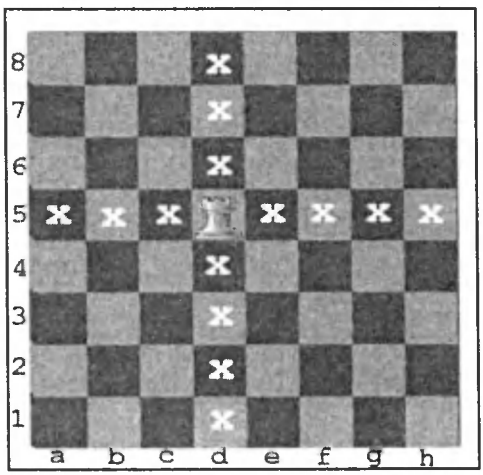


Рис. 2.11. Ходы ладьи

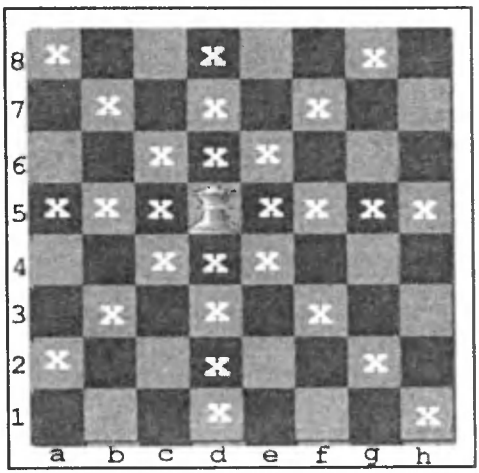


Рис. 2.12. Ходы ферзя

Целесообразно упомянуть еще ряд правил:

- Все фигуры, кроме пешек, бьют точно так же, как ходят.
- Цель игры — поставить мат противнику. Мат — это когда король под боем, и пойти ему некуда, он везде попадает под бой.
- Пат — это когда ходить может только король, а пойти ему некуда, он везде попадает под бой. Король при пате не под шахом. Пат — это ничья.
- Ничейным считается результат, когда какая-либо позиция повторяется три раза.
- Ничейным может быть некоторое бесперспективное окончание, когда нет средств, чтобы поставить мат, или противники соглашаются на ничью.

Оценка позиции

Оценка позиции является очень непростым этапом в построении шахматной программы. Существует множество факторов, учитываемых шахматистами. Как заставить программу понимать их? Как обеспечить их гармоничное взаимодействие в зависимости от фазы игры? Эти вопросы до конца не решены. В данном разделе рассматриваются два подхода к этой проблеме.

Типичная оценочная функция

Рассмотрим типичную оценку в шахматной программе. В настоящее время существует множество открытых исходных кодов шахматных программ. Большинство из них использует подход, аналогичный следующему.

Пешки

Материальная оценка пешки примерно 100. Стратегическая оценка возрастает при продвижении пешки. Ладейные пешки получают меньший стимул для продвижения вперед, центральные пешки — больший. Для получения предварительной оценки используется массив, подобный следующему:

```
int BlackPawn[64] =
{0, 0, 0, 0, 0, 0, 0, 0,
 4, 4, 4, 0, 0, 4, 4, 4,
 6, 8, 2, 10,10,2, 8, 6,
 6, 8, 12,16,16,12,8, 6,
 8, 12,16,24,24,16,12,8,
 12,16,24,32,32,24,16,12,
 12,16,24,32,32,24,16,12,
 0, 0, 0, 0, 0, 0, 0, 0};
```

Пешки в примере двигаются сверху вниз. Видно, что первый ход королевской пешкой через клетку даст прирост стратегической оценки 16, а первый ход ладейной пешкой — только 2. Все это довольно условно, и конкретные величины в массиве могут отличаться в разных программах.

Проходные пешки обрабатываются особым образом. У них гораздо больше стимул для продвижения вперед, особенно ближе к концу игры. Примерная оценка для проходных пешек:

```
int PassedPawn[8] = {0, 5, 10, 15, 20, 30, 140, 0};
```

Данная оценка может изменяться, в зависимости от стадии игры и других факторов. Ладейные пешки получают меньший стимул для продвижения, чем остальные. Ближе к концу игры стремление пешек в ферзи увеличивается.

Король

Для короля основной приоритет в начале игры — остаться на своей горизонтали:

```
int  King1[64] =
{0,    0,  -4,  -10, -10, -4,    0,   0,
 -4,   -4, -8,  -12, -12, -8,   -4,  -4,
 -12, -16, -20, -20, -20, -20, -16, -12,
 -16, -20, -24, -24, -24, -24, -20, -16,
 -16, -20, -24, -24, -24, -24, -20, -16,
 -12, -16, -20, -20, -20, -20, -16, -12,
 -4,   -4, -8,  -12, -12, -8,   -4,  -4,
 0,    0,  -4,  -10, -10, -4,    0,   0};
```

Как видно, королю даже дан приоритет занять угол доски, в надежде, что он сделает рокировку.

В конце игры у короля — приоритет центра:

```
int King2[64] =
{0,  6,  12, 18, 18, 12, 6,  0,
 6,  12, 18, 24, 24, 18, 12, 6,
 12, 18, 24, 30, 30, 24, 18, 12,
 18, 24, 30, 36, 36, 30, 24, 18,
 18, 24, 30, 36, 36, 30, 24, 18,
 12, 18, 24, 30, 30, 24, 18, 12,
 6,  12, 18, 24, 24, 18, 12, 6,
 0,  6,  12, 18, 18, 12, 6,  0};
```

Ферзь

Материальная оценка королевы примерно 900. Основной стимул для коро левы — быть ближе к королю противника. Расстояние до короля противни

ка, умноженное на некоторый коэффициент, принимается за стратегическую оценку для королевы.

Ладьи

Материальная оценка ладьи примерно 500.

Конь

Материальная оценка коня — примерно 300. Основной стимул для коня — занять центр доски.

```
 int knight[64] =
{0, 4,  8,  10, 10, 8,  4,  0,
 4, 8,  16, 20, 20, 16, 8,  4,
 8, 16, 24, 28, 28, 24, 16, 8,
 10,20, 28, 32, 32, 28, 20, 10,
 10,20, 28, 32, 32, 28, 20, 10,
 8, 16, 24, 28, 28, 24, 16, 8,
 4, 8,  16, 20, 20, 16, 8,  4,
 0, 4,  8,  10, 10, 8,  4,  0};
```

Слон

Материальная оценка для слона — примерно 300. Основной стимул для слона — занять главные диагонали.

```
int bishop[64] =
{14, 14, 14, 14, 14, 14, 14, 14,
 14, 22, 18, 18, 18, 18, 22, 14,
 14, 18, 22, 22, 22, 22, 18, 14,
 14, 18, 22, 22, 22, 22, 18, 14,
 14, 18, 22, 22, 22, 22, 18, 14,
 14, 18, 22, 22, 22, 22, 18, 14,
 14, 22, 18, 18, 18, 18, 22, 14,
 14, 14, 14, 14, 14, 14, 14, 14};
```

Все значения в данных массивах являются приблизительными. Материальная оценка фигур может быть несколько больше, чем приведенная, и может зависеть от наличия других фигур. (Например, пара слонов является плюсом.) Приведенные оценки, в принципе, могут вычисляться пошагово, но они недостаточно характеризуют конкретную позицию.

Дополнительные факторы

Рассмотрим важные дополнительные факторы:

- Структура пешек. Этот момент требует особого рассмотрения. У белых и черных может быть поровну пешек, но из-за слабости структуры одна из

сторон может проиграть. Некоторые учитываемые факторы: сдвоенные пешки получают штраф, сдвоенные заблокированные пешки — еще больший штраф. Штрафом наказываются также изолированные и отсталые (не защищенные) пешки.

- Атака X-ray, когда одна дальнобойная фигура стоит позади другой, получает некоторую премию.
- Для ладей вводится дополнительно некоторая премия для занятия открытой вертикали и, начиная с середины игры, — некоторый штраф, в зависимости от расстояния до короля противника.
- Два слона имеют некоторое преимущество, по сравнению с двумя другими легкими фигурами.
- Мобильность: фигуры получают некоторую премию или штраф в зависимости от того, сколько полей они могут атаковать.
- Безопасность короля — это особая тема. Нужно с помощью штрафов попытаться сохранить пешечный экран короля (пока у противника есть ферзь). Можно оценивать некоторым косвенным образом давление на короля противника. Чем больше мы обращаем внимание на безопасность короля, тем лучше играет наша программа. Можно ввести некоторый штраф в зависимости от числа шахов в строке игры.
- Некоторыми коэффициентами может регулироваться активность в атаке, стремление пешек в ферзи и т. д.

Все это довольно непросто настроить. Надо очень хорошо понимать шахматную игру, чтобы добиться приемлемой позиционной оценки.

Простая оценочная функция

Если вы не чувствуете в себе сил, чтобы настроить все сложные позиционные факторы, можно воспользоваться простой позиционной оценкой. Пока на доске существует хоть одна пешка, для всех фигур используется следующая позиционная оценка для верхних (черных):

```
int upSt[8][8] =
{
{ 1, 2, 3, 4, 5, 6, 7, 8},
{ 9,10,11,12,13,14,15,16},
{17,18,19,20,21,22,23,24},
{25,26,27,28,29,30,31,32},
{33,34,35,36,37,38,39,40},
{41,42,43,44,45,46,47,48},
{49,50,51,52,53,54,55,56},
{57,58,59,60,61,62,63,64}
};
```

для нижних: (белых):

```
int botSt[8][8] =
{
{64,63,62,61,60,59,58,57},
{56,55,54,53,52,51,50,49},
{48,47,46,45,44,43,42,41},
{40,39,38,37,36,35,34,33},
{32,31,30,29,28,27,26,25},
{24,23,22,21,20,19,18,17},
{16,15,14,13,12,11,10, 9},
{ 8, 7, 6, 5, 4, 3, 2, 1}
};
```

Когда нет пешек, для всех фигур обеих сторон используется одна общая оценка с приоритетом центра:

```
int endGSt[8][8] =
{
{ 1,2,3,4,  4, 3, 2, 1},
{ 9,10,11,12, 12,11,10, 9},
{17,18,19,20, 20,19,18,17},
{25,26,27,28, 28,27,26,25},
{25,26,27,28, 28,27,26,25},
{17,18,19,20, 20,19,18,17},
{ 9,10,11,12, 12,19,18,17},
{ 1, 2, 3, 4,  4, 3, 2, 1}
};
```

Если кто-то думает, что это шутка, то напрасно. Начиная с глубины перебора 8, программа начинает понимать позиционную игру. Получение более точной оценки для некоторых позиционных факторов регулируется гибкой системой выборочных продлений. Жестких, статических приоритетов нет. Все они вычисляемые. Программа должна расширять строки с шахами и до некоторой степени (как правило, в пределах полного перебора) строки с взятиями. После дебюта можно совсем немного расширять строки, в которых расстояние между ферзем противника и нашим королем опасно сокращается. Расширяются ходы, когда пешка идет на предпоследнюю горизонталь. Отдельным образом можно обрабатывать проходные пешки. Вся оценка вычисляется пошагово, т. е. самой оценочной функции нет, и на вычисление практически не тратится времени. Все силы должны быть брошены на то, чтобы просчитать как можно глубже. Кроме того, надо добавить материальную и позиционную оценку для взятой фигуры противника.

Данный вариант предпочтительнее для тех, кто хочет попробовать свои силы в написании первой шахматной программы или является сильным про-

граммистом и может сделать эффективную, глубоко считающую программу. За счет различных систем выборочных продлений можно научить программу вычислять практически все позиционные факторы, а не прописывать жесткие рамки. Здесь есть масса возможностей для экспериментирования. Особенно интересные варианты можно получить, когда есть список легальных ходов (не теряем материал), а позиционная составляющая вычисляется с помощью статического поиска (неперспективные варианты подрезаются статической оценкой).

Примерная оценка материала для данного варианта:

- пешка — 1000
- слон и конь — 3000
- ладья — 5000
- ферзь — 9000

Приращение стратегической оценки никогда не сможет сравняться с материальной оценкой. Если нужно научить программу определять жертву фигур, это делается другими способами.

Эффект горизонта

Шахматы — сложная игра. Перебор на некоторую фиксированную глубину для спокойных, тихих перемещений, может быть, и достаточен. В напряженных же строках игры программа может получить неверный результат и повести себя непредсказуемо, например, начать выбрасывать фигуры. Это явление получило название эффекта горизонта. Простейшее его проявление — программа не может до конца рассмотреть длинный размен, ей кажется, что она выигрывает, а на самом деле она теряет. Существует еще множество проявлений эффекта горизонта. Например, если программа перебирает на 8 полуходов, то в некоторой строке игры вполне может оказаться несколько взятий и шахов, и программа в конце перебора получит неизвестно что. Вариантов тактики на доске может быть сколько угодно, и даже в серьезной, профессиональной программе может обнаружиться эффект горизонта. Например, наш ферзь в ловушке. Человек посмотрит на такую позицию, и ему ясно практически сразу, что ферзь пропал. Программа же смотрит не так. Она находит длинную тактическую строку, в которой, например, есть размен фигур, шах, несколько угроз или просто атак, и она уже не видит потерю ферзя, данный ход вытеснен за горизонт видимости программы. Существует великое множество всяких тактических последовательностей, которые приводят к эффекту горизонта. Как же бороться с этим эффектом? Лучше всего перебрать ходов на 16 или на 20, но это невозможно, и приходится прибегать ко всяческим хитростям. В конце основного поиска (полного перебора) вставляют упрощенный поиск, который рас-

сматривает только форсированные ходы. В самом простейшем случае это взятия. Считается правилом хорошего тона рассматривать взятия до конца. Таким образом, программа как бы считает до конца, но не все. В более сложных вариантах форсированного поиска до некоторой глубины могут просматриваться шахи и даже некоторые другие ходы (например, атаки). Более подробно об организации форсированных вариантов будет сказано позже. Далее, в основном дереве перебора глубину в напряженных строках можно сделать более гибкой, т. е. чтобы программа просматривала такие строки глубже, чем остальные. Это называется выборочными продлениями. Типичные случаи продлений — при шахе, при размене, при авансе пешек (пешка пошла на предпоследнюю горизонталь). Продления могут быть, если оценка данного хода существенно отличается от остальных (это, как правило, взятия). Если программа рассматривает взятия до конца и имеет некоторые выборочные продления, то эффект горизонта в значительной степени уже сглажен. Но все не так просто. Тактических последовательностей может быть много. Если мы начнем продлевать все такие варианты, это приведет к взрыву дерева поиска и программной аварии. Я уже упомянул о варианте, где ферзь попадал в ловушку. Полный переборщик, чтобы увидеть это, должен считать на 12 полуходов, что нереально. Если расширять строки, то нужно расширять и атаки, что приведет к резкому замедлению поиска, и программе для нормальной игры не будет хватать простых перемещений в полном дереве перебора. Это напоминает ситуацию "нос вытащишь — хвост увязнет, хвост вытащишь — нос увяз". Программа не видит других важных строк, которые не попадают под упрощенные критерии сильных перемещений.

В шахматах бывает такая ситуация: жертва фигуры на переднем плане и угроза королю на заднем плане. Человек, как правило, сразу видит такие ситуации и их потенциальную опасность. Иное дело, машина. Машина, чтобы увидеть, должна сосчитать. Тут как раз в полной мере проявляется эффект горизонта. Программа прерывает просчет в такой опасной ситуации, что дальше некуда, но она этого не понимает. Вернемся к жертве фигуры. Жертва в большинстве случаев оправдывается угрозой королю. Программа не понимает, что в глубине дерева перебора она потеряет значительно больше, а может, и проиграет. Рассмотрим простой пример на рис. 2.13.

В этой позиции ход черных. Они могут взять конем ладью на a1. Полный переборщик не видит, что его король под угрозой. Он видит в пределах своего дерева перебора, что он выигрывает ладью, а королем успевает уйти. Если он и чувствует угрозу, то это оценка стратегического порядка, она меньше материальной. Выигрыш ладьи все перекрывает. Черные берут ладью на a1, белые идут ферзем на f8, и пошло... Чтобы прочувствовать данную позицию, программа должна считать на 10—12 полуходов. Необходимо сказать, что данная позиция проста. Нужно просто продлевать шахи, и программа все прекрасно увидит. Для данного примера достаточно даже шести полуходов полного перебора с продлением шахов и взятий до конца

в форсированных вариантах. Но не всегда шахов бывает достаточно. Не всегда все так просто. Угроза королю может быть глубокой. Нет шахов или их мало. Но давление на короля, тем не менее, усиливается. Вот пример (рис. 2.14).

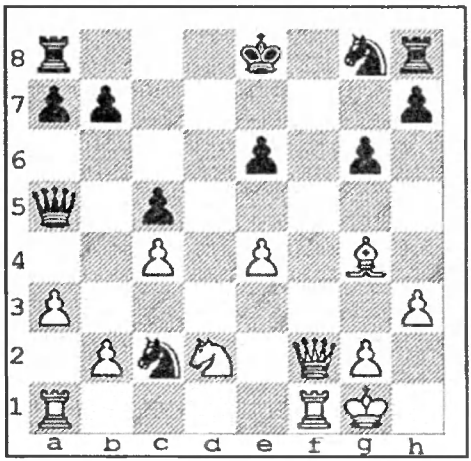


Рис. 2.13. Эффект горизонта

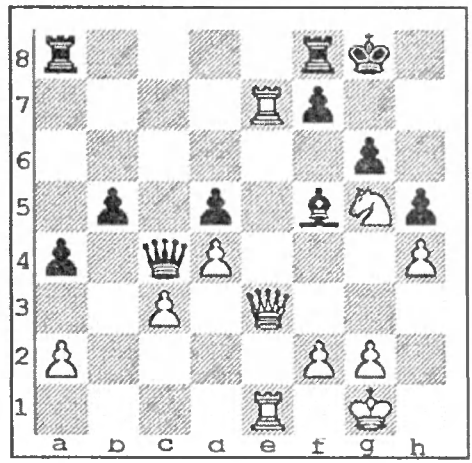


Рис. 2.14. Глубокая угроза королю

В этой позиции ход тоже черных. Они могут взять пешку ферзем на a2. Какая, казалось бы, может быть опасность? Только ферзь уходит от защиты своего короля, а вокруг него чувствуется какая-то потенциальная угроза. Белые следующим ходом идут ферзем на e5. И что? В чем опасность? Опасность заключается в ходе белых конем g5—f7 с взятием пешки и, по существу, жертвой коня. Коня черные могут взять, но вокруг них начнет разворачиваться опасная матовая ситуация. В любом случае, подобных ситуаций лучше избегать, и пешка не стоит всего того, что за этим последует. На какой глубине здесь начинаются реальные шахи? Как программе это увидеть? Чтобы решить эту задачу, полному переборщику потребуется неизвестно сколько уровней. Полный переборщик на 10 полуходов, расширяющий шахи, с нулевым ходом и взятиями до конца, не видит в глубине дерева неизбежной потери материала и берет пешку. Что же делать? Если мы внимательно посмотрим на ключевые строки этой позиции, то увидим, что программа должна взятия рассматривать глубже. Это, прежде всего, ход черных c4a2, потом длинный размен по полю f7 (примерно 4 взятия), и программа увидит ситуацию правильно и оценит опасность. Для данного варианта достаточно полного перебора на 8 полуходов, расширять взятия до какой-то глубины и шахи значительно глубже. Говоря другими словами, для данного примера ключевыми являются строки с взятиями.

Эти два примера являются типичнейшими для шахмат. Кроме взятий и шахов, их еще объединяет потенциальная угроза ферзя королю противника. Такие ситуации программа должна просчитывать глубже, чтобы понимать

жертву материала, иначе простой выигрыш пешки у нее перевесит все стратегические преимущества.

Форсированные варианты

По окончании полного перебора поиск может быть продолжен в форсированном варианте, который, в сущности, является конечной стадией выборочного поиска. Основное его назначение — дать примерную оценку позиции горизонта. Какие ходы являются форсированными? В самом простейшем случае это взятия. Взятия в шахматах дают самый большой прирост оценки (существенно больше, чем стратегическая оценка), и поэтому рассмотрение взятий в глубине поиска сглаживает до некоторой степени эффект горизонта. Взятия являются также ходами, требующими немедленной реакции. В шахматах бывают очень длинные размены, которые программа должна рассматривать до конца. Если программа считает на 8 полуходов и рассматривает все размены в конце, уровень ее игры будет значительно выше. Кроме взятий, и другие ходы могут являться форсированными. Например, шахи. Шах является, в сущности, видом атаки. Атака может быть на пешку, на ферзя, а в случае шаха — на короля. Шах является особым ходом в шахматах. Игра при шахе как бы замирает, и сама ситуация требует более глубокого просчета. Шахи в очень сильной степени влияют на эффект горизонта. Если форсированные варианты не рассматривают шахи, их оценка является уж очень приблизительной. В некоторых случаях и этого бывает достаточно. Если программа считает глубоко и имеет развитую систему выборочных продлений полного перебора, то на некоторой глубине ей может оказаться достаточно самой приблизительной оценки горизонта. Существуют два основных подхода к этой проблеме:

1. Максимально глубокий поиск с полным перебором и упрощенные форсированные варианты. В этом случае программа очень хорошо видит в пределах горизонта и выборочных продлений, но плохо — на глубине. Если она сосчитала на 6 полуходов, и не было ни одного случая расширения поиска (шах, размен и т. д.), то она выходит в форсированный вариант и имеет очень приблизительную оценку (только размен фигур). Для острых позиций, когда рассматриваемый вариант не касается короля, этого может оказаться и достаточно, но такая программа будет постоянно попадать в тупик при сильной позиционной игре. Ее неспособность глубоко считать частично компенсируется оценочной функцией, которая может заставить программу делать ходы, кажущиеся наполненными каким-то смыслом. Особенно это заметно, когда нет длинных разменов, и идет дистанционная позиционная игра. Программа запросто может погнаться за пешкой и оголить короля, т. к. оценка выигрыша пешки, как правило, больше максимального позиционного преимущества. Компенсация недостаточно глубокого просчета за счет оценочной функции срабатывает только против слабых игроков.

2. Сложный форсированный вариант, требующий много времени процессора и, как следствие, не обеспечивающий такой глубокий полный перебор. Это другая крайность. В форсированном варианте могут рассматриваться не только взятия—шахи, но еще осуществляется некоторый выборочный поиск на первой его стадии. На первых полуходах форсированного варианта могут рассматриваться приращения позиционной оценки, статической оценкой не подрезаются взятия, атаки на простые фигуры и т. д. Такой форсированный вариант дает более точную оценку, но сокращается полный перебор. Программа может превосходно вести себя в острых, тактических позициях и ошибаться в спокойных. Может просто не хватить глубины перебора, чтобы видеть потенциальную угрозу ходов, не подпадающих под формальные определения сильных перемещений.

Все упирается в основную проблему программирования игр методом перебора. Считать надо очень глубоко, а очень глубоко считать невозможно. Есть различные подходы к решению этой проблемы. Основная задача состоит в выделении лучших ходов и более глубокого их просчета при помощи выборочных продлений или в форсированных вариантах. Разберем простейший форсированный вариант, рассматривающий только взятия до конца:

```
int Quies(int alpha, int beta)
{
  int val = Evaluate(); //статическая оценка текущей позиции
  if(val > alpha) alpha = val;
  PMove move = GenerateCaptures(); //все взятия
  while( move && alpha < beta)
  {
     MakeMove();
     val = -Queis(-beta,-alpha);
     UnMakeMove();
     if(val > alpha)alpha = val;
  }

  return alpha;
}
```

Рассмотрим приведенный код подробнее. В сущности, это обычный alpha-beta поиск, только вместо всех перемещений генерируются взятия, и alpha повышается текущей оценкой. Глубина поиска не ограничена. Взятия ограничены количеством фигур на доске, и они будут рассмотрены до конца. Так как мы предусматриваем только взятия и повышаем alpha текущей оценкой, такой поиск будет выполняться значительно быстрее, чем полный широтный поиск. Почему мы повышаем alpha? Дело в том, что рассматри-

ваются не все перемещения, а только взятия. Если взятие ведет к ухудшению оценки, программа не делает его. Вместо остальных перемещений используется текущая оценка данной позиции. Мы исходим из предположения, что программа нашла бы ход, не ухудшающий текущую оценку, если бы все перемещения были предусмотрены. Если рассматривать данный узел с точки зрения предыдущего узла, из которого был сделан ход противника, то в нем оценкой после перемещения занижается максимум (beta). Например, взяли коня — наш максимум ограничивается текущей оценкой плюс оценка хода (вес коня). Таким образом, если смотреть из преузла, максимум ограничивается взятой фигурой, и программа не выясняет все комбинации, а только возможность взятия данного коня. Говоря другими словами, форсированный поиск взятий рассматривает только перемещения, которые захватывают. Взятие коня может повлечь за собой некоторую комбинацию, приводящую к еще большему выигрышу материала. Программа этого не увидит. Она исходит из предположения, что раз черные взяли коня, и ход белых, они всегда найдут ответ, не приводящий к дальнейшей потере материала. Это приблизительно, но на большой глубине нет другого выхода. Или мы не считаем ничего, или считаем упрощенно.

Взятия в генераторе перемещений должны быть обязательно отсортированы. Например, по принципу "наиболее ценная жертва — наименее ценный нападающий" (MVV/LVA). Без сортировки взятий форсированный вариант существенно замедлит выполнение программы и приведет к более сокращенному полному широтному поиску, что недопустимо. Если форсированный вариант рассматривает только взятия, то основное его назначение — дать приблизительную оценку позиции горизонта, и выполняться он должен очень быстро. Может возникнуть искушение не считать все взятия, а давать еще более приблизительную оценку позиции горизонта. Например, если текущая оценка меньше alpha на ладью, то взятие пешки вряд ли имеет смысл. Существует алгоритм SEE, рассматривающий только победившие или равные взятия. Например, сочетание ферзь—пешка не будет рассмотрено. Очень может оказаться, что пешка не защищена, и такой алгоритм дает ошибку. Основная ставка делается на то, что форсированный вариант в любом случае ошибается, а если мы сосчитали уже достаточно глубоко, можно ограничиться совсем приблизительной оценкой позиции горизонта. Здесь есть масса возможностей для эксперимента. Мое мнение, что к SEE надо подходить очень осторожно. Отвергать, как несущественное, взятие пешки можно на достаточно большой глубине, а взятие ладьи — тем более. В целом, SEE вполне работоспособный алгоритм.

В рассмотренном примере форсированный вариант подсчитывает только взятия. В этой главе мы не будем останавливаться на нюансах выборочного поиска и отсечения статической оценкой. Рассмотрим только форсированный вариант с шахами. Более углубленный просмотр шахов увеличивает достоверность форсированного поиска, но может резко замедлить время

выполнения программы. Просмотр шахов заключается в следующем. Если обнаружено, что король под шахом, то alpha не повышается текущей оценкой. Вызывается полный широтный поиск с глубиной 1, ответы на шах будут рассмотрены все. Это очень усеченный вариант статического поиска, и вообще, выборочный поиск делается немного иначе. Но если мы хотим рассматривать дальше только шахи, можно воспользоваться и этой схемой. Вот пример кода:

```
int Quies(int alpha, int beta)
{
  if(inCheck) return AlphaBeta(1,alpha,beta); //полный поиск
  int val = Evaluate(); //статическая оценка текущей позиции
  if(val > alpha) alpha = val;
  PMove move = GenerateCaptures(); //все взятия
  while( move && alpha < beta)
  {
     MakeMove();
     val = -Queis(-beta,-alpha);
     UnMakeMove();
     if(val > alpha)alpha = val;
  }

  return alpha;
}
```

Если король под шахом, вызывается функция полного широтного поиска. Несмотря на простоту примера, в нем видна одна тонкость: не повышая alpha под шахом, мы делаем предположение, что результат расчета может увеличить его, и не хотим мешать этому. Для всех ходов, кроме шаха, ограничиваем максимум результатом сделанного хода. Минимум неизвестен, он выявится в результате просчета. Для шахов не ограничиваем максимум. Таким образом в миниатюре демонстрируется основной принцип выборочного поиска. Лучшие ходы надо просчитывать глубже. Так как отсечение выполняется статической оценкой, результат не точен, просто мы позволяем лучшим ходам подрасти. Какие ходы являются лучшими — отдельный вопрос. На данной глубине просчета мы полагаем, что лучшими являются шахи, и не ограничиваем их максимум. Повышение alpha в нашем узле — это то же самое, что и понижение beta после сделанного хода в преузле. Продление шахов в статическом поиске резко улучшает чутье программы на безопасность короля и может в некоторых случаях найти глубокий форсированный мат. Также результат разменов получается более точный. Обратите еще раз внимание, что у нас продлеваются только шахи. Взятия не продлеваются. Наша функция считает взятия как ходы, дающие наибольший прирост оценки. Можно обсчитывать и все ходы, но счет резко замедлится. В случае

продления только шахов, счет тоже значительно замедляется. Мы усиливаем форсированные варианты за счет сокращения полного широтного поиска, что недопустимо. Какой же выход? Есть один испытанный прием. В форсированном варианте глубина просмотра шахов делается более гибкой. Изначально она равна, допустим, 2. Это значит, что на первых двух полуходах форсированного варианта программа обнаружит все шахи. При каждом шахе глубина может увеличиваться на 1 (до разумного предела, например, до 30 полуходов). Если на шах есть единственный ответ, то глубину просмотра можно увеличить на 2. Это позволяет программе найти глубокие тактические выпады, анализируя минимальное количество позиций. Такая схема требует изощренного программирования. С возрастанием мощности процессоров возрастает глубина полного перебора. В пределах полного перебора шахи тоже могут расширяться, но несколько другим образом. Не будем забывать, что полный перебор экспоненциален, а основное назначение форсированного варианта — приблизительная оценка позиции горизонта.

Какой из двух вариантов лучше использовать, с шахами или без них? Это дело вкуса и сильно зависит от конкретной реализации. В пределах полного перебора при шахах не сокращается глубина до некоторого предела. Это принцип расширения полного широтного поиска, и об этом будет сказано позднее.

Выборочные продления

Краткая характеристика расширений

В полном широтном поиске, который мы рассматривали, глубина поиска задавалась при первом вызове рекурсивной функции и уменьшалась в каждом узле на 1. Таким образом, мы осуществляли полный перебор на некоторую фиксированную глубину. Мы рассмотрели также устройство форсированного варианта, который вызывался при `Depth` = 0 и сглаживал эффект горизонта. Форсированный вариант просчитывал дальше наиболее сильные перемещения и давал приблизительную оценку позиции горизонта. Допустим, что при минимальном размере дерева перебора, т. е. при оптимальном порядке перемещений, для углубления счета на глубину N+1 нам необходимо рассмотреть в 20 раз больше позиций, чем при счете на глубину N. Это довольно приблизительно, но нам для рассуждений нужно отталкиваться от каких-то величин. Существует мнение, что увеличение глубины счета на один полуход повышает силу игры программы на разряд. Может, не стоит ломать голову над разными ухищрениями, а просто подождать, пока процессоры станут мощнее в 20 раз. Мы сможем считать на 9 полуходов, потом еще в 20 раз — мы сосчитаем на 10 полуходов. Бывают тактические последовательности, которые программа должна просчитывать значительно глубже. Например, если на доске существует форсированная угроза королю,

то может потребоваться считать на 14 полуходов. Гроссмейстеры просчитывают некоторые варианты на 20 и более полуходов. Вряд ли мощность процессоров когда-либо возрастет до такого уровня, да и программы желательно писать сейчас, а не через 10 лет. Человек, когда просчитывает некоторые варианты, руководствуется множеством критериев, зависящих от его опыта. Он может просчитать ситуацию очень глубоко, проанализировав при этом минимальное количество позиций. Несмотря на то, что моделировать мышление шахматиста мы не можем, существуют некоторые строки игры, просчет которых можно продолжить по совершенно формальным признакам. Возникла простая идея — не сокращать глубину просмотра при некоторых ходах. Мы можем считать на 6 полуходов, но некоторые варианты будут рассмотрены значительно глубже. При каких же ходах не сокращается глубина счета? Если мы говорим о шахматах, то это, прежде всего, шах. При шахе игра как бы замирает и требует от противника немедленного ответа. Замедление счета при этом тоже минимальное, т. к. число легальных ходов под шахом ограничено, и узел будет обсчитан очень быстро. Продление всего лишь шахов резко увеличивает силу игры программы. Считая на 6 полуходов, она способна обнаружить глубокие форсированные маты и другие комбинации, основанные на угрозе королю и приводящие к выигрышу материала. Пример функции счета, продлевающей шахи:

```
#define MAX_PLY 20
int AlphaBeta(int color, int Depth, int ply, int alpha, int beta)
{
     if(Depth == 0 || ply > MAX_PLY) return Quies(color,alpha,beta);

     if(InCheck()) Depth++; //если шах, глубину не уменьшаем
     Depth--;

     PMove move = GenerateAllMoves();
     while(move && alpha < beta)
     {
        MakeMove(move);
        int tmp = -AlphaBeta(color==BLACK?WHITE:BLACK, Depth, ply+1
                             -beta,
                             -alpha);
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        move = move->next;

     } //while
     return alpha;

} //function
```

Функция `InCheck` играет роль детектора шахов. В данном примере введена новая переменная — `ply`. Это абсолютная глубина счета. При первом вызове она равна 0 и увеличивается на 1 в каждом узле. `Depth` уже только косвенно отражает глубину счета. Если `ply` превысит некоторое предельное значение (в нашем примере оно ограничено константой `MAX_PLY`), то поиск завершится и продолжится уже в форсированных вариантах. Продление шахов делает программу намного сильнее. Нужно отметить, что шах — это абсолютно безопасное продление. Сами правила игры подразумевают, что на шах должен быть дан немедленный ответ, иначе конец игре. Общее положение теории игр гласит, что ходы, дающие приращение в узле намного больше, чем другие в этом узле, нужно просчитывать глубже. Это аксиома. Вопрос в том, какие ходы считать лучшими. Для шахмат — это, бесспорно, взятия. Взятия дают приращение оценки намного больше, чем другие ходы, приращение которых ограничено стратегическим приростом. Насчет продления взятий существует множество мнений. Продление всех взятий, кроме того, приведет к взрыву дерева перебора и программной аварии. Взятие не всегда требует немедленного ответа, правдоподобность ответа на взятие значительно ниже, чем на шах. К решению этой проблемы можно подойти различными путями. Например, взятия продлеваются до некоторой глубины, а дальше — только шахи. Мы должны просчитывать дальше лучшие ходы, но ход, лучший для глубины 2, не всегда является лучшим для глубины 10. На глубине 10 можно уже не уточнять вопрос, будет ли точно выиграна пешка, или программе это только кажется. Некоторые программы расширяют взятия в зависимости от глубины и захваченной области. Например, можно воспользоваться массивом, где индекс соответствует `ply` данной глубины. Если ход в данную позицию больше или равен значению элемента таблицы с данным индексом, мы расширяем узел.

```
static int ext[] =
{pawn,pawn,pawn,pawn,pawn,pawn,bishop,rook,queen,INFINITY,....};
```

Еще раз хочется подчеркнуть, что расширять взятия уместно в пределах полного перебора. Приведем стандартную схему расширений:

- в случае шаха;
- если произошел размен фигуры;
- если пешка пошла на предпоследнюю горизонталь.

Эта схема является стандартной и приводится почти во всех поверхностных описаниях расширений поиска. Обратим внимание на расширение при размена. При небольших накладных расходах это расширение увеличивает тактическую прочность программы. Несмотря на это, вокруг него идут непрекращающиеся споры. В чем его слабость? Во-первых, размены, хотя и сглаживают эффект горизонта, довольно условно отражают основные тактические строки. Во-вторых, просчитывая дальше из взятий только размены,

мы тем самым побуждаем программу делать их. Это довольно неприятно, когда программа без видимых причин начинает разменивать фигуры. Дело в том, что мы размен рассмотрели глубже, и программа могла увидеть некоторый тактический прирост от размена. Применять ли это расширение или нет — дело вкуса. Большинство программ его использует, но с некоторыми оговорками.

Дробный Depth

Нужно просчитывать глубже многие ходы, но это может привести к взрыву дерева поиска, и только шахи являются стопроцентно корректными расширениями на любой глубине. Возникла следующая идея: делать `Depth` дробным. В каждом узле `Depth` уменьшается не на 1, а на 4. Начальное значение `Depth` тоже равно глубине просчета, умноженной на 4. Таким образом, в зависимости от нашего представления о правдоподобности данного хода, мы можем при одних ходах увеличивать `Depth` на 4 (при шахах), а при других — на меньшую величину. Приведем примерный список расширений:

- шах — 4;
- размен — 3;
- пешка на предпоследней горизонтали — 3.

Дробный показатель глубины является, в сущности, вероятностным методом, и я отношусь к нему довольно скептически, но очень многие программы используют его, и он является самым простым способом осуществить выборочные продления многих вариантов, не приводя к взрыву дерева поиска. Мое личное мнение, что программа должна считать точно, а не накапливать вероятность правдоподобия в каждой строке, но есть другие взгляды на этот счет. Альтернативным является метод увеличения `Depth` всегда на 1, но, в зависимости от глубины просчета, дальше продлеваются все лучшие и лучшие ходы, а некоторые постепенно отсеиваются.

Искусственный интеллект пока не изобретен, структура и способ организации выборочных продлений, являющихся интеллектуальной составляющей программы, подлежит обсуждению.

Кроме перечисленных простейших случаев, существует еще целый ряд возможных выборочных продлений. Постараюсь привести, по возможности, полный список:

- Расширение, если шах. Обратите внимание на вскрытые шахи, когда фигура при ходе освобождает линию атаки на короля дальнобойной фигуре.
- Расширение при размене. Можно до некоторой степени расширять взятия, можно проводить расширения в зависимости от захваченной части.
- Если в позиции есть только одно фактическое перемещение. При шахе мы не сокращаем глубину. Если есть всего один ход, при котором короля

не бьют (обычно при шахе), то глубина увеличивается на 1. Получается, что шахи рассматриваются на глубину +1, а единственный ответ +2. Конкретная реализация может быть различной.

- Расширение по угрозе. Для обнаружения угрозы может использоваться нулевой ход или детектор атаки.
- Если пешка пошла на предпоследнюю горизонталь.
- Если резко падает безопасность короля. В самом простом случае, когда расстояние между ферзем противника и нашим королем опасно сокращается.
- Если оценка изменяется резко без взятия.
- Если результат некоторого хода существенно выделяется среди остальных.

Какие расширения использовать в конкретной программе — дело вкуса. Нужно лишь помнить, что наиболее ценными являются расширения шахов, взятий и авансов пешек.

Безопасность короля

В шахматах безопасность короля занимает особое место. Цель игры — поставить мат королю. Чем больше программа обращает внимание на безопасность короля, тем лучше она играет. Нужно учесть, что оценка позиции инвертированная, и если мы разрабатываем безопасность короля, то программа лучше чувствует потенциальные угрозы королю и успешнее строит атаки на короля противника. Безопасность короля — это ключевая тема шахматной программы. Шахи, безусловно, отражают безопасность короля, но не в полной мере. Очень глубокого полного перебора и продления шахов может оказаться достаточно для приемлемой игры, но варианты, потенциально опасные для короля, лучше просчитывать глубже. Эти ходы сложно как-то характеризовать. Они увеличивают давление на короля противника, и все. Представим себе на поле игры некоторые воображаемые квадраты: квадрат, равный полю игры, отображает все поле, некоторый квадрат вокруг короля — поле короля. Любые передвижения в квадрате короля могут иметь повышенную опасность для него. Можно еще представить себе квадрат центра поля, отражающий борьбу за центр в начале игры, некоторый квадрат для проходных пешек на половине противника и т. д. Главная идея использования таких квадратов, или мини-полей, заключается в том, что на всем поле перебрать глубже ходы мы не в состоянии, но можно продлевать ходы в некоторых выбранных квадратах. Деление на квадраты довольно условно, но нужно не забывать, что мы не вносим дополнительной погрешности, а только ищем критерии для более углубленного просчета некоторых ходов. Ситуация в форсированных вариантах все равно будет просчитана до конца, мы только можем расширить полный перебор для некоторых строк игры.

Если сравнить с мышлением шахматиста, то строки с взятиями он просматривает чуть глубже (например, на 2), но значительно дальше анализирует ситуацию вокруг короля. На атаке короля строится большинство комбинаций, и самые красивые решения построены именно на этом. С точки зрения формальной теории игр, король — самая ценная фигура, и атаку на него нужно просчитывать глубже, т. к. это, безусловно, самые сильные ходы.

Выборочные продления — хорошая техника, но их не должно быть много. Каким же образом можно ограничить неуправляемый рост дерева перебора при выборочных продлениях? Можно задать дробное значение `Depth`, можно расширять потенциально опасные для короля ходы. И потом, этих расширений все равно не должно быть много.

Есть простой способ соблюсти безопасность короля и ограничить до приемлемого количества расширения в строке: продлевать далее ходы, когда расстояние между ферзем противника и нашим королем опасно сокращается. Таких ходов значительно меньше, и время счета не должно сильно увеличиться. Кроме того, существуют даже методики, позволяющие отсекать ветви в глубоких узлах дерева перебора, если данный узел не расширяется и текущая статическая оценка в узле больше beta на некоторую величину. Величина королевы является, как правило, безопасной для подобного отсечения, т. к. это максимальное приращение материальной оценки, и жертва королевы вообще нонсенс в шахматах, а на большой глубине не стоит тратить время на подобные ветви игры. Если у нас глубина поиска будет больше в самых напряженных строках игры, такая оптимизация не должна на практике привести к ухудшению тактической прочности программы. Но это пока в сторону. Вернемся к ходам королевы, потенциально опасным для нашего короля. Королева — это особая фигура в шахматах. Она в начальной позиции стоит рядом со своим королем и как бы показывает степень его безопасности. В некоторых случаях позиционная оценка королевы даже равна расстоянию до короля противника. Это, конечно, слишком приближенно, и такие варианты должны просчитываться глубже. Во всяком случае, ключевые ходы королевы должны просчитываться глубже (меньше чем шахи, но больше, чем взятия). В шахматах бывают комбинации с жертвой фигур. В большинстве случаев жертва фигур оправдывается далекой форсированной угрозой ферзя королю противника, и только шахов здесь недостаточно. Программа должна иметь либо очень глубокий полный перебор, либо развивать атаку на короля дальше. Еще лучше и то, и другое. У меня был вариант программы, реализующей подобные продления для королевы. Она постоянно выигрывала у очень сильных программ именно на угрозе королю, т. к. такие варианты она просчитывала очень глубоко. Также программа качественно лучше обеспечивала безопасность своего короля, особенно если форсированные варианты были с шахами. В целом, хотя лучше полного перебора и нет ничего, можно построить сильную программу и со сравнительно неглубоким полным широтным поиском (6 полуходов, как правило).

Рассмотрим, как на практике можно осуществить продление опасных для короля выпадов ферзя. Допустим, нам известны координаты последней ходившей фигуры противника. Координаты фигуры противника можно передавать в стеке при рекурсивном вызове, координаты нашего короля нам всегда известны, если мы используем списки фигур, а король стоит в них первым. Допустим, для генерации перемещений мы используем массив 8 × 8, представляющий позицию. Нужно измерить расстояние от последней ходившей фигуры противника до нашего короля, если эта фигура ферзь, в старой позиции и в новой. Если новая позиция входит в воображаемый квадрат короля, и расстояние меньше, то такой узел — кандидат для расширения. Как измерить это расстояние? Мы не можем воспользоваться теоремой Пифагора и вычислить корень квадратный. Это долго, а скорость для нас принципиальна. Можно строить линию от фигуры до короля и подсчитать количество клеток этой линии, но это тоже долго. Можно использовать вспомогательный массив, значительно больший по размеру, чем 8 × 8. Воображаемая позиция короля — в центре этого массива, а воображаемый ферзь на таком же расстоянии от него, что и в массиве игры. В каждой клетке массива расстояние до короля. Нам нужно лишь перевести координаты нашего ферзя в координаты нового массива и посмотреть, что в данной клетке. Это и будет расстояние до нашего короля. Вот пример программного кода:

```
/////////////// расстояние до фигуры ////////
static unsigned char _taxi[20*32]=
{
9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,6,6,6,6,6,6,6,6,6,6,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,5,5,5,5,5,5,5,5,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,4,4,4,4,4,4,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,3,3,3,3,3,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,2,2,2,2,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,2,1,1,1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,2,1,0,1,2,3,4,5,6,7,8,9,3,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,2,1,1,1,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,2,2,2,2,2,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,3,3,3,3,3,3,3,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,4,4,4,4,4,4,4,4,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,5,5,5,5,5,5,5,5,5,5,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,6,6,6,6,6,6,6,6,6,6,6,6,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,7,7,7,7,7,7,7,7,7,7,7,7,7,7,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
9,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,9,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

```
9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,9,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,3,0,0,0,0,0,0,0,0,0,5,0,0,0,0,0,0,0,0,0,0,0
};
////////////// возвращает расстояние от фигуры до фигуры //////////////
#define TAXI(x,y,kingX,kingY)\
( _taxi[ ((9 + ((int)y - (int)kingY)) << 5)\
+ 9 + ((int)x - (int)kingX) ])
```

Воображаемый король стоит в ячейке [9,9]. Ширина массива больше его высоты. Это сделано для того, чтобы можно было вычислить индекс ячейки без умножения простым битовым сдвигом. Умножение на 32 эквивалентно сдвигу влево на 5. Современные процессоры очень быстро перемножают небольшие числа, но злоупотреблять этим не следует. Функция `TAXI` переводит координаты ферзя (`X`, `Y`) и короля (`kingX`, `kingY`) в координаты массива `TAXI` и возвращает расстояние от ферзя до короля. Этот пример только демонстрирует один из способов решения данной задачи. Далее приводится схематичный пример расширения узла при потенциальной угрозе ферзя королю:

```
if(ply <= _maxPly+2 && PIECE(_pos[fy][fx]) == QUEEN)
{
     int oldTaxi = TAXI(oldX,oldY,kingX,kingY);
     int newTaxi = TAXI(fx,fy,kingX,kingY);
     bool isTaxi = false;
     if(newTaxi < 4)
     {
       if(newTaxi < oldTaxi) isTaxi = true;
       if(newTaxi == oldTaxi && fx+fy < oldX+oldY) isTaxi = true;
     }
     if(isTaxi) isAddDepth = true;
}

if(isAddDepth) Depth++;
Depth--;
```

Обозначения в программе:

- `oldTaxi` — расстояние от ферзя до короля в старой позиции ферзя;
- `newTaxi` — расстояние от ферзя до короля в новой позиции ферзя;
- `fy`, `fx` — новые координаты ферзя;
- `oldY`, `oldX` — старые координаты ферзя;
- `kingY`, `kingX` — координаты нашего короля;
- `isAddDepth` — флаг расширения узла;

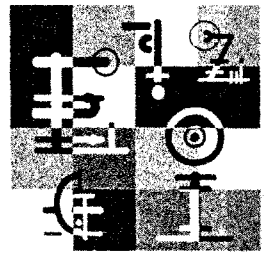
- `ply` — абсолютная глубина узла в строке от 0;
- `depth` — остаток глубины просчета (регулируется);
- `_maxPly` — гарантированная глубина просчета при данной итерации без выборочных продлений.

Если ферзь в новой позиции попадает в квадрат короля, и расстояние до короля меньше старого, то узел расширен. Нужно еще ограничить максимальное количество расширений разумным числом (например, 4). Данное расширение весьма условно, но значительно повышает тактический уровень программы.

Безопасность короля не ограничивается угрозами ферзя. Она может отражаться и в оценочной функции, и в выборочных продлениях. Возможен и силовой вариант: перебор на 8 (лучше — 10) полуходов с продлением шахов. Стандартный подход учитывает безопасность короля только в позиционной оценке и продлевает только шахи. Но с таким подходом не написать программу, играющую выше среднего уровня (если у вас, конечно, нет знакомого гроссмейстера, который потратит на написание и отладку оценочной функции лет тридцать).

Стандартными, с точки зрения теории игр, являются расширения, значительно увеличивающие прирост в узле после хода. Для шахмат — еще шахи и авансы пешек. Перебор для сильной игры при силовом решении должен быть не менее 8 полуходов, кроме того, форсированные варианты с взятиями. Все остальное находится в области экспериментирования. Расширяя узлы, критичные для безопасности короля, но не имеющие четкой формальной характеристики, мы рискуем, и подобный подход требует тонкой настройки. Тем не менее на практике это может работать очень хорошо. Желательно добиться, чтобы программа в дереве перебора всегда дотягивалась до короля, каким бы пешечным и другим экраном он ни был закрыт. Для этого можно расширить другие ходы в квадрате короля (простые перемещения — совсем немного, взятия — значительно больше). В самом начале игры такое расширение использовать не целесообразно. В качестве примера могу сказать, что расширение только угроз ферзя придавало программе, считающей на 6 полуходов, способность видеть острые ситуации, которые могла развить только программа, считающая на 8 или даже 10 полуходов. Просчитывать глубоко самые интересные ситуации — это признак уровня программы и совершенно преобразует ее игру, даже когда подобные ситуации не подпадают под формальные признаки.

ГЛАВА 3



Простая шахматная программа

Мы уже рассмотрели многие вещи, касающиеся построения шахматной программы. Об этом можно много рассуждать, но лучше попробовать построить первую программу своими силами. Тогда многое станет понятнее.

Силовое решение

Здесь мы рассмотрим пример простой программы, осуществляющей так называемое силовое решение: полный перебор на 6 полуходов, выборочные продления и форсированные варианты с взятиями до конца. Такая программа, конечно, не будет играть очень сильно, но уровень первого разряда ей обеспечен. При хорошей оценочной функции и дебютных справочниках, она может показать и лучший результат. Но это в сторону. Наша задача сейчас состоит в построении простейшей программы, а не в погоне за результатами. Данная программа использует alpha-beta алгоритм перебора, NegaScout и сортировку перемещений по значению. Взятия сортируются по принципу MVV/LVA (наиболее ценная жертва — наименее ценный нападающий). Остальные перемещения сортируются в три списка по значению. Хеш-таблица не будет задействована. Наша программа и так должна сосчитать на 6 полуходов, а отвлекаться на второстепенные вещи мы пока не будем.

Сердцевиной любой шахматной программы является alpha-beta перебор:

```
int AlphaBeta(int alpha, int beta, int Depth)
{
   if(Depth == 0) return Evaluate();
   Generate();
   while(moveList && alpha < beta)
   {
      MakeMove();
```

```
        int score = -AlphaBeta(-beta,-alpha,Depth-1);
        UnMakeMove();
        if(score > alpha) alpha = score;
    }
    return alpha;
}
```

Данный алгоритм является основным улучшением полного перебора (NegaMax). Он очень чувствителен к порядку ходов. При наилучшем порядке ходов он рассмотрит $\sqrt{N}$ позиций, где N — число позиций при полном переборе. Среднее количество перемещений в шахматах равно 40. Отсюда мы можем заключить, т. к. нам надо перебрать на 6 полуходов, то при наилучшем порядке ходов нам нужно будет рассмотреть 64 тыс. позиций.

Это совсем немного. Наилучшего порядка перемещений мы не достигнем, но надо учесть небольшие выборочные продления и взятия в форсированных вариантах до конца. Учитывая это, предположим, что число рассматриваемых позиций возрастет. Для упорядочивания мы применим, для простоты, только сортировку. Нам потребуются списки фигур, инициализированные перед началом просчета. В этих списках *(см. главу 2)* король идет первым, затем — ферзь, и далее по убыванию веса фигуры. Рассмотрим типичный элемент списка фигур:

```
typedef struct _TItem {
    int f;    //фигура
    int x,y; //ее координаты
    struct _TItem *next; //следующий элемент списка
}TItem, *PItem;
```

Два списка, для верхних фигур и для нижних, задаются при инициализации:

```
PItem _upFList, _botFList;
```

Рассмотрим некоторые необходимые константы. Массив игры мы определим как символьный массив 8 × 8. Фигуры представим числами по порядку: король — 1, ферзь — 2 и т. д. Каждая клетка в массиве игры должна хранить также информацию о цвете фигуры: белые — 128, черные — 64. Обратите внимание на некоторую битовую хитрость. Максимальное число клетки массива (символьное без знака) составляет 255. Вот десятичные значения, когда соответствующий бит установлен в 1:

номер бита	7	6	5	4	3	2	1	0
десятичное число	128	64	32	16	8	4	2	1

Таким образом, коды фигур уместятся у нас в первых трех битах. Цвет мы обозначим битами 7 и 6. Дополнительно нам потребуется информация о том, перемещалась ли фигура. Под этот флаг выделим третий бит. Это тре-

буется для правильной генерации рокировок. Как вы помните, рокировку делать нельзя, если король или ладья хоть раз ходили. А они вполне могут походить и вернуться. Вот список констант, на основании сказанного:

```
#define KING  1
#define QUEEN 2
#define ROOK  3
#define BISHOP 4
#define KNIGHT 5
#define PAWN   6
#define MOVE  8
#define BLACK 64
#define WHITE 128
#define COLOR(f)  ((f) & (BLACK | WHITE)) //цвет фигур
#define FIGURE(f) ((f) & 7)               //код фигуры
#define IS_MOVE(f) ((f) & MOVE)           //перемещалась ли фигура?
//выход за пределы доски
#define OutBoard(x,y) ((unsigned)(x) >= 8 || (unsigned)(y) >= 8)
```

Определим также вес фигур:

```
#define W_PAWN 1000
#define W_BISHOP (3*W_PAWN)
#define W_KNIGHT W_BISHOP
#define W_ROOK  (5*W_PAWN)
#define W_QUEEN (9*W_PAWN)
#define INFINITY (10*W_QUEEN)
#define W_KING INFINITY
```

Для вычисления веса фигуры по ее коду объявим массив перекодировки:

```
static int _wf[7] = {0,W_KING,W_QUEEN,W_ROOK,W_BISHOP,W_KNIGHT,W_PAWN};
```

Инициализируем списки фигур. Используем для этого простейший способ. Этот блок программного кода выполняется однократно. Для хранения собственно элементов списков используем буфер памяти — простой статический массив:

```
unsigned char _pos[8][8]; //позиция
static int _upFColor; //цвет верхних фигур
static TItem data[20];  //буфер памяти
PItem p = &data[0]; //указатель на первый элемент
//все клетки доски
_upFList = _botFList = NULL; //обнулим списки фигур
for(int n = PAWN; n >= KING; n--)
for(int y = 0; y < 8; y++) for(int x = 0; x < 8; x++)
```

```
{
  int f = _pos[y][x];
  if(!f || FIGURE(f) != n) continue;
  p->x = x;
  p->y = y;
  p->f = f;
  if(COLOR(f) == _upFColor)
  {
    //вставка в начало списка верхних фигур
     p->next = _upFList;
     _upFList = p;
  }else{
     //вставка в начало списка нижних фигур
     p->next = _botFList;
     _botFList = p;
  }
  p++; //указатель на следующий элемент буфера
}
```

Приведенный код достаточно прост и инициализирует списки фигур по убыванию веса. Первым идет король, последними — пешки. Это нужно, чтобы перемещения фигур брались в определенном порядке. Нам будет легче сортировать взятия по принципу "наиболее ценная жертва — наименее ценный нападающий", и ходы наиболее сильных фигур желательно рассматривать первыми, т. к. это потенциально более сильные перемещения. В ситуациях, когда король под шахом, счет ускоряется в несколько раз только из-за того, что перемещения короля идут первыми. Шахи — это основные продления, и их нужно обрабатывать максимально быстро.

В предыдущей главе книги взятая фигура удалялась из списка фигур противника. Для этого использовалась ссылка на указатель и дополнительный массив, где каждой фигуре на доске соответствовала ссылка на указатель для удаления из соответствующего списка. Это требует изощренного программирования и отвлекает внимание на второстепенные вещи. Реальная же прибавка в скорости незначительна. В данном примере мы поступим проще. Взятая фигура просто удаляется с доски, соответствующий элемент списка пустой, мы его пропустим при генерации перемещений и перейдем к следующему. Единственная сложность возникает для парных фигур. Если взяли ладью, то хотя элемент списка и пустой, в данной клетке доски может оказаться вторая ладья, и для нее перемещения будут сгенерированы два раза, т. к. на данную клетку доски указывают два элемента списка фигур. Проблему легко обойти, если пометить парные фигуры некоторым дополнительным флагом. У нас в описателе фигуры свободен четвертый бит. Его мы и используем для метки. Лучше, конечно, удалять элемент списка фи-

гур, но сейчас главная задача не усложнять детали, а показать принцип. С учетом сказанного, позиция в начале игры будет следующая:

```
static unsigned char _pos[8][8] =
{
{ROOK|16|BLAK, KNIGHT|16|BLACK, BISHOP|16|BLAK, QUEEN|16|BLACK,
KING|BLACK,
BISHOP|BLACK, KNIGHT|BLACK,ROOK|BLACK};
{...} //и так все фигуры
};
```

Фигуры в левой половине доски помечаются дополнительным флагом 16. Это их отличает от фигур того же цвета в правой половине доски. Черные вверху, белые — внизу. Перед тем как написать функцию счета, представим основную схему нашей программы. Осуществляется полный перебор на глубину 6 с выборочным продлением шахов. Когда глубина становится равной нулю или превышает максимально допустимую, вызывается функция форсированного поиска, рассматривающая только взятия до конца. Мы используем алгоритм NegaScout и амортизацию отказов. Схема программы примерно следующая:

```
//оценка позиции
int Evaluate()
{
  {...} //подсчитаем оценку
        //лучше это делать пошагово
}
```

Функция оценки должна подсчитать материал для белых, прибавить к нему позиционную оценку белых и вычесть материал черных и позиционную оценку черных. В результате мы получим позиционную оценку белых. Если функция должна вернуть оценку черных, значение снабжается знаком минус. Позиционную оценку можно вычислять разными способами. В главе, посвященной оценке позиции *(см. главу 2)*, были приведены таблицы средней оценки для каждой фигуры. Для простейшей оценочной функции можно воспользоваться ими:

- для короля в начале игры таблица задавала приоритет его горизонтали и углов, в конце игры — приоритет центра;
- для королевы позиционная оценка заключалась в штрафе от расстояния до короля противника;
- для коней — приоритет центра;
- для слонов — приоритет центра и ключевых диагоналей;
- для пешек — стимул двигаться вперед, ладейные пешки получают меньший стимул. Для проходных пешек оценка значительно больше. Можно

отслеживать структуру пешек и давать некоторые штрафы сдвоенным пешкам, изолированным и отсталым (не защищенным пешкой).

Инициализируем таблицы с дополнительными данными о положении фигур, руководствуясь следующими соображениями:

- пока ферзь противника на доске, можно попытаться сохранить пешечный экран короля;
- можно давать премии для X-гау атаки (одна дальнобойная фигура стоит позади другой);
- можно давать штрафы зависающим фигурам;
- можно оценивать потенциальное давление на короля;
- можно содействовать ладьям после дебюта занять открытые вертикали и давать небольшой штраф в зависимости от расстояния до враждебного короля.

Для первой программы не нужно гнаться за сложной оценочной функцией, можно ограничиться таблицами средних значений для каждой фигуры *(см. главу 2)*. Написание хорошей оценочной функции — сложная задача, требующая шахматных знаний и долгого времени. Готовых рецептов здесь нет. Можно воспользоваться и простейшей оценкой, одной для всех фигур, но следует учесть, что для хорошей игры с такой оценкой нужен более глубокий просчет и развитая (я бы даже сказал, интеллектуальная) система выборочных продлений. В данной программе мы продлеваем только шахи. Код приведен в следующем листинге:

```
//форсированный вариант
int Quies(int alpha, int beta, int ply)
{
   int score = Evaluate();   //оценка текущего узла
   int res = -INFINITY,      //результат просчета
       tmp;
   //повышаем alpha статической оценкой
   if(score > res) res = score;
   if(res > alpha) alpha = res;
   if(alpha >= beta) return alpha;
   //все взятия в отсортированном порядке
   PMove move = GenerateAndSortCaptures();
   while(moveList)
   {
     //если съели короля, дальше не считаем
     //возвращаемая оценка с поправкой на глубину,
     //чтобы не оттягивать мат
     if(eatKing) return INFINITY-ply;
     MakeMove();
```

```
      tmp = -Quies(-beta,-alpha,ply+1);
      UnMakeMove();
      if(tmp > res) res = tmp;
      if(res > alpha) alpha = res;
      if(alpha >= beta) return alpha;
    }
    return res;
}

    //основной поиск
int Search(int alpha, int beta, int Depth, PMove bestMove, int ply)
{
    //выборочные продления при шахе
    if(InCheck()) Depth++;
    Depth--;
    //если исчерпана глубина счета или превысили
    //максимальную глубину, вернем
    //результат форсированного поиска
    if(Depth <= 0  ||  ply > MAX_PLY)
      return Quies(alpha,beta);
   int res = -INFINITY,  //результат счета
       tmp;
    //все перемещения в отсортированном порядке
   PMove move = GenerateAndSortAllMoves();
   while(moveList)
   {
        if(eatKing) tmp = INFINITY-ply;
 else{ // NegaScout
        MakeMove(); //сделали ход
        tmp = -Search(-(alpha+1),-alpha,Depth,NULL,ply+1);
        if(tmp > alpha && tmp < beta)
            tmp = -Search(-beta,-tmp,Depth,NULL,ply+1);
        UnMakeMove(); //восстановили позицию
      }
      if(tmp > res) res = tmp;
      if(res > alpha)
      {
         alpha = res;
       //вернули лучший ход при первом вызове
         if(bestMove) memcpy(bestMove,move,sizeof(TMove));
      }
      if(alpha >= beta) return alpha;
    }
```

```
    //проверка на пат
    //если перемещался только король,
    //и на следующем ходу его съели,
    //и он не под шахом в данной позиции,
    //результат равен 0: ничья
   if(moveOnlyKing && !inCheck() &&
      res == -(INFINITY-(ply+1))
    )
     res = 0;
   return res; //вернем всегда результат отсечки
}

///////////////// первый вызов ///////////////
TMove bestMove; //сюда запишется найденный ход
//инициализируем списки фигур и глобальные переменные
{ ... }
Search(-INFINITY,INFINITY,6,&bestMove,0);
///////////////  end ////////////////////////
```

Как видите, все достаточно просто. Для написания программы потребуется создать два генератора перемещений. Один выдает только взятия в отсортированном порядке, второй — все остальные ходы. Описатель перемещения может выглядеть следующим образом:

```
//характеристика хода
typedef enum{ S_MOVE. //простое перемещение
              S_1,    //рокировка
              S_2     //взятие пешкой через битое поле
             } TMoveDesc;
typedef struct _TMove{
   int f; //фигура
   int x1,y1; //откуда пошла
   int x2,y2; //куда пошла
   TMoveDesc moveDesc; //характеристика хода
   struct _TMove *next; //следующий элемент списка
}
```

Функция, делающая ход на доске, может выглядеть как:

```
void MakeMove(PItem item, //элемент списка фигур
              PMove move //описатель хода
             )
{
   switch(move->moveDesc)
   {
      case S_MOVE: //простое перемещение
```

```
      {
        int newF = move->f | MOVE;//фигура на новом месте
                                  //с флагом перемещения
        //если пешка дошла до последней горизонтали,
        //превратится в ферзя
        if(FIGURE(move->f) == PAWN  &&  (move->v2==7 || move->y2==0))
           newF =  (move->f & (~7))  |  QUEEN  |  MOVE;
        //изменения в позиции
        _pos[move->y1][move->x1] = 0;        //убрали фигуру
        _pos[move->y2][move->x2] = newF;//поставили фигуру
        //изменения в списке фигур
        item->x = move->x2;
        item->y = move->y2;
        item->f = newF;
      }break;
      case S_1:{
       //рокировка
       {...}
      }break;
      case S_2:{
       //взятие пешкой через битое поле
       {...}
      }
   }//switch
}
```

Все это несколько избыточно по коду и непрофессионально. Служит только для демонстрации принципа. При ходе сама фигура у нас запомнится в описателе перемещения, и когда мы будем восстанавливать позицию, ее нужно оттуда переписать в старую позицию на доске, а также восстановить начальное состояние списка фигур. В реальной программе я генератор перемещений объединял с функцией счета. Так намного проще, и нет никаких промежуточных вызовов функций. Сначала находятся и сортируются все взятия. Потом — остальные ходы. Если для простых перемещений найдено потенциально лучшее перемещение, можно сразу выполнять счет, а не ждать, пока все перемещения будут сгенерированы. Возможны и многие другие хитрости. Если вы пишете первую программу, то заострять внимание на этом не нужно. Можно обойтись даже без списка фигур, правда, эффективность такого кода будет чрезвычайно низка. Главное — это порядок ходов. Существуют методики, позволяющие уменьшить дерево перебора (нулевой ход и пр.). Это дает возможность даже при неэффективном генераторе ходов создать довольно приличную программу.

Пример функции, восстанавливающей позицию, я не привожу. Она тривиальна. Просто все возвращает на свои места. Если генератор ходов совме-

щен с функцией счета, то не нужно многих проверочных условий. Например, превращение пешки возможно только при ходе пешки, рокировка — только при ходе короля и т. д. При написании генератора ходов обратите внимание на то, что при сортировке взятий первыми лучше рассматривать ходы, бьющие последнюю ходившую фигуру противника (по возрастанию веса нападающей фигуры), а потом — все остальные взятия в отсортированном порядке. Существует множество других методик, позволяющих лучше упорядочить ходы: хеш-таблицы в сочетании с итеративными углублениями (таблицы перестановок, отслеживающие повторы позиций), внутренние итеративные углубления и пр. Пока мы этого не касались. Нужно сказать, что приведенная программа вполне способна просчитать на 6 полуходов и обыграть своего создателя, при условии хорошо настроенной оценочной функции.

Проще некуда

Если вы не можете настроить оценочную функцию, или не хватает опыта программирования, или просто нет времени для написания и отладки большой программы, а написать все-таки хочется, могу предложить простейшую схему. Уровень игры ее я затрудняюсь определить, но любителю, не перехаживая, выиграть будет затруднительно. В этой программе нет списка фигур, генератор перемещений совмещен с функцией счета, а оценочная функция отсутствует как таковая — вся оценка вычисляется динамически. Программа считает на пять полуходов с полным перебором, а дальше — упрощенный выборочный поиск, по существу, играющий роль оценочной функции. Данная программа опровергает все классические каноны шахматного программирования, она не считает точно, а некоторым образом прогнозирует ситуацию.

Рассмотрим основные идеи, используемые в этой программе. Первый вызов функции счета предполагает сразу большую глубину, например двадцать, но счет идет, пока не исчерпан лимит времени и функция `TimeOver()` не возвратит `true`. В самом начале, прежде чем нашли хоть одно перемещение осуществляется поиск лучшего хода на глубину N−2. Если такой ход найден, его пробуем первым, в надежде получить отсечку сразу. Вот пример кода:

```
static TMove glMove; //сюда вернется лучший ход
int Search(int alpha,int beta,int Depth,PMove retMove,int ply)
{
   //пока не нашли лучшего хода
   NoFindMove(retMove);
   if(Depth <= 0 || TimeOver()) return Evaluate();
   TMove tmpMove; //стековая промежуточная переменная
```

```
   //найдем лучший ход на меньшую глубину
   Search(alpha,beta,Depth-2,retMove,ply);
 //если лучший ход найден, попробуем его первым
   if(findMove)
   {
      MakeMove(retMove);
      int tmp = -Search(-beta,-alpha,Depth-1,
                        &tmpMove, //нам не потребуется
                        ply+1);
      UnMakeMove(retMove);
      if(tmp > alpha) alpha = tmp;
      if(alpha >= beta) return alpha;
   }
   //основной счет
   {...}
   ///допустим, нашли ход
   {...}
       MakeMove(move);
       tmp = -Search(-beta,-alpha,Depth-1,&tmpMove,ply+1);
       UnMakeMove(move);
       if(tmp > alpha)
       {
          alpha = tmp;
          if(!TimeOver())
           memcpy(retMove,move,sizeof(TMove));
       }
       if(alpha >= beta) return alpha;
   {...}
   return alpha;
}
```

Первый вызов данной функции будет выглядеть примерно так:

```
Search(-INFINITY,INFINITY,20,&glMove,0);
```

Мы вызываем функцию на большую глубину. Она сразу "ныряет" на глубину N−2 для получения лучшего хода, и первый ход при `ply=0` будет найден при `Depth=2`. Этот лучший ход будет использован при `Depth=4` и т. д. Функция не будет исследовать узел, не попытавшись найти лучший ход для данного узла на меньшей глубине. Сколько полуходов переберет данная функция? Это неизвестно. Пока не исчерпан лимит времени. Лучший ход записывается всегда, когда результат счета больше текущего значения alpha. Если исчерпан лимит — лучший ход не записывается. Таким образом, функция, помимо упорядочивания перемещений, осуществляет постепенное

углубление перебора, пока не исчерпан лимит времени. Сортировка перемещений в узлах все равно нужна. Взятия можно рассматривать первыми и без сортировки. Остальные перемещения берутся позже. Взятия в большинстве случаев дают отсечку. Остальные перемещения нужно сортировать. Лучшие можно обсчитывать сразу, средние и худшие (прирост отрицательный) — оставлять на потом.

Если у нас есть результат просчета на глубину N−2, то на глубине N его можно использовать для отсечения узла без предварительного исследования. Отсекаемый узел должен удовлетворять некоторым требованиям. Как минимум, это не должно быть взятие и шах. Текущая оценка должна превышать значение beta на некоторую величину, называемую полем роста. Поле роста следует делать равным максимальному стратегическому приросту. Обычно, это величина в полпешки. Если узел удовлетворяет этим требованиям, и результат счета на глубине N−2 больше beta, его можно отсечь. Отсечение узла без предварительного исследования является довольно рискованным шагом. В худшем раскладе перебор на N выльется в перебор на N−2, но это очень маловероятно. Все размены и шахи мы не отсекаем, а тихие перемещения на глубине не дают значительного прироста оценки. Данная программа считает не до конца. Ее просчет на глубине является, в сущности, видом оценочной функции, учитывающей некоторые динамические возможности. В реальной программе на определенном этапе мы будем отсекать наиболее слабые перемещения статической оценкой. Этот пример является учебным, и выборочный поиск сильно упрощен.

Итак, определим основные типы данных. Фигуры у нас остаются прежними. Тип, описывающий перемещение:

```
typedef struct _TMove{
 int x1,y1,x2,y2; //откуда и куда пошла фигура
 int f;           //сама фигура
 int newF;        //значение фигуры на новом месте
                  //пешка после превращения меняет значение
 int eatFX, eatFY;//координаты съеденной фигуры
                  //при взятии пешкой через битое поле
                  //эти координаты не совпадают с новым
                  //местом фигуры
 int eatF;        //сама съеденная фигура
 int score;       //приращение оценки при ходе
 bool isExtMoveKing; //рокировка?
 int Depth;       //глубина счета
}TMove,*PMove;
#define B BLACK
#define W WHITE
static unsigned char _pos[8][8] =
```

```
{
 {ROOK|B,KNIGHT|B,BISHOP|B   ....},
 {...}
}
//делает перемещение
void MakeMove(PMove move)
{
   if(!move->isExtMoveKing)
   {
     _pos[move->y1][move->x1] = 0;
     _pos[move->eatFY][move->eatFX] = 0;
     _pos[move->y2][move->x2] = move->newF;
   }else{
    //рокировка
    {...}
   }
}
//восстанавливает позицию
void UnMakeMove(PMove move)
{
  //все в обратном порядке
  {...}
}
//массивы со стратегической оценкой
typedef int TStBoard[8][8];
//стратегическая оценка
        //для верхних
        static TStBoard _upSt =
        {
        { 1, 2, 3, 4, 5, 6, 7, 8},
        { 9,10,11,12,13,14,15,16},
        {17,18,19,20,21,22,23,24},
        {25,26,27,28,29,30,31,32},
        {33,34,35,36,37,38,39,40},
        {41,42,43,44,45,46,47,48},
        {49,50,51,52,53,54,55,56},
        {57,58,59,60,61,62,63,64}
        };
        //для нижних
        static TStBoard _botSt =
        {
        {64,63,62,61,60,59,58,57},
        {56,55,54,53,52,51,50,49},
        {48,47,46,45,44,43,42,41},
```

```
        {40,39,38,37,36,35,34,33},
        {32,31,30,29,28,27,26,25},
        {24,23,22,21,20,19,18,17},
        {16,15,14,13,12,11,10, 9},
        { 8, 7, 6, 5, 4, 3, 2, 1}
        };
//массив быстрой перекодировки кода фигуры в ее вес
static int _wf[7] = {0,W_KING,W_QUEEN,W_ROOK,W_BISHOP,W_KNIGHT,W_PAWN};
//возвращает приращение оценки хода
int delRes( TStBoard *myB, //оценка наших фигур
            TStBoard *opB, //противника
            PMove move     //описатель перемещения
          )
{
   int score;
   if(move->isExtMoveKing)
   {
     //рокировка
     {...}
   }else{
      score = _wf[FIGURE(move->newF) - _wf[FIGURE(move->f)] +
              (*myB)[move->y2][move->x2] -
              (*myB)[move->y1][move->x1];
      if(eatF)
      {
        score +=  _wf[FIGURE(move->eatF);
        score +=  (*opB)[move->eatFY][move->eatFX];
      }
   }
   return score;
}
//цвет верхних фигур
static int _upFColor;
//сама функция счета (схематично)
int Search(PMove node, //описатель хода в эту позицию
           int Depth,  //сколько полуходов осталось
           int ply,    //абсолютная глубина от 0
           int alpha,
           int beta,
           PMove retMove,   //найденный ход
           int score,       //текущая оценка узла
           bool isLazyEval, //идет сокращенный просчет
           int color        //цвет наших фигур
          )
```

```
{
     TStBoard *myB,*opB;
     int nextColor = color==BLACK?WHITE:BLACK;
     //инициализируем переменные
     if(color != _upFColor)
     {
        //если наши фигуры нижние
        myB = &_botSt; opB = &_upSt;
     }else{
        myB = &_upSt; opB = &_botSt;
     }
     retMove->f = 0;          //нет пока лучшего хода
     if(Depth <= 0 || TimeOver()) return score;
     int res = -INFINITY; //возвращаемая оценка
     //в эту стековую переменную будет возвращен лучший ход
     TMove tmpMove;
     //структура выборочного поиска
static int ext[100] =
{-64,-64,-64,-64,
-64,W_PAWN,W_BISHOP,W_ROOK,W_QUEEN,INFINITY,INFINITY,...};
    //повышаем alpha текущей оценкой
    if(ply > 0  &&                  //если не первый полуход
       node->score < ext[ply] &&    //ход не продлевается
      !InCheck(node)                //не шах
     )
    {
        res = score;
        if(res > alpha) alpha = res;
        if(alpha >= beta) return alpha;
    }

     //переберем на глубину Depth-2 для получения
     //лучшего хода
     int tmp = Search(node,Depth-2,ply,alpha,beta,
                      retMove,
                      score,
                      ply==0?false:true,
                      color
                     );
    /// lazy eval /////
    //если данный узел удовлетворяет некоторым требованиям,
    //мы можем отсечь его, основываясь на результате сокращенного счета
    if(retMove->f &&              //если нашли лучший ход
       retMove->Depth >= Depth-2 //достаточная глубина
```

```
    ply != 0 &&                 //не первый вызов
    !isLazyEval &&              //не делаем сокращенного поиска
    score - 64 >= beta    &&    //оценка и поле роста >= beta
    tmp-64 >= beta        &&    //проверка подтверждает
    node->score < W_PAWN &&     //не взятие
    !InCheck(node)              //не шах
  )
  {
    return beta; //целая ветвь подрезана!!!
  }
  //прежде, чем нашли перемещения,
  //попробуем найденный лучший ход
  if(retMove->f) //если такой ход найден
  {
    MakeMove(retMove);
    tmp = -Search( retMove, //описатель хода в эту позицию
                   Depth-1,  //сколько полуходов осталось
                   ply+1,    //абсолютная глубина от 0
                   -beta,
                   -alpha,
                   &tmpMove,   //найденный ход(нам не нужен)
                   -(score+retMove->score), //текущая оценка узла
                   isLazyEval,
                   nextColor
                 );
     UnMakeMove(retMove);
     if(tmp > res) res = tmp;
     if(res > alpha)
     {
       //если не превышен лимит времени
       if(!TimeOver())
       {
         //лучший ход тот же,
         //только корректируем глубину
         retMove->Depth = Depth;
       }
       alpha = res;
     }
     if(alpha >= beta) return alpha;
  }//endif

 ////////  взятия ////////////////////////////
 //нашли взятие - сразу в следующую позицию ////
 {...}
```

```
   ////// допустим, нашли ход ////////////////////
   {...}
      //приращение при ходе
      move->score = delRes(myB,opB,move);
      MakeMove(move);
      tmp = -Search( move, //описатель хода в эту позицию
                     Depth-1,  //сколько полуходов осталось
                     ply+1,    //абсолютная глубина от 0
                     -beta,
                     -alpha,
                     &tmpMove,   //найденный ход(нам не нужен)
                     -(score+move->score), //текущая оценка узла
                     isLazyEval,
                     nextColor
                   );
      if(tmp > res) res = tmp;
      if(res > alpha)
      {
         alpha = res;
         if(!TimeOver())
         {
            memcpy(retMove,move,sizeof(TMove));
            retMove->Depth = Depth;
         }
      }
      if(alpha >= beta) return alpha;
      UnMakeMove(move);
   {...}
   {...}
   return res;
}//end function
```

Пример первого вызова данной функции:

```
static TMove glMove; //сюда вернется лучший ход
memset(&glMove,0,sizeof(TMove));
static TMove tmpMove; //описатель первого перемещения
memset(&tmpMove,0,sizeof(TMove)); //нет этого перемещения
    Search(&tmpMove,   //описатель хода в эту позицию
           20,         //сколько полуходов осталось
           0,          //абсолютная глубина от 0
           -INFINITY,  //alpha
           INFINITY,   //beta
           &glMove,    //найденный ход
```

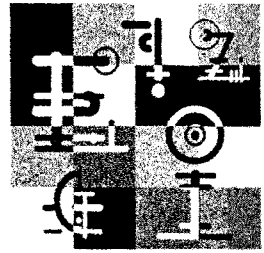
```
    Evaluate(), //текущая оценка узла
    false,      //идет сокращенный просчет
    WHITE       //цвет фигур получаемого хода
  );
```

Данный пример является схематичным. При создании реальной программы нужно быть очень внимательным. Особенно следует обратить внимание на то, что если мы считаем на N−2, то не должны уже в этом сокращенном поиске подрезать ветви поиска без предварительного исследования. Это возможно только в основном поиске, с учетом характеристики узла и результата сокращенного просмотра.

Вместо внутренних итеративных углублений можно использовать большую хеш-таблицу. Это гарантированно ускорит поиск. Конкретная реализация может быть различной. Слишком сильно данная программа упрощенного выборочного поиска играть не будет, но и обижать себя не даст. Как правило, у среднего любителя она будет выигрывать.

В конце игры, когда на доске нет пешек, нужно использовать одну стратегическую оценку с приоритетом центра.

ГЛАВА 4



Более сложные приемы

ZObrist-ключи

Для написания шахматной программы нужно уметь сравнить две позиции, одинаковы ли они? Такая задача возникает довольно часто, и ее выполнять нужно с максимальной скоростью. Например, при построении хеш-таблицы нужно иметь ключ, однозначно идентифицирующий позицию. При поиске конкретной позиции сначала находится начало подсписка, соответствующего данному ключу. Хотя ключ и не совсем точен, поиск значительно сокращается.

В каких случаях возникает необходимость поиска данной позиции? Их очень много. Это поиск в дебютных справочниках, в справочниках по игровым ситуациям на доске, в базах данных эндшпилей, в таблицах самообучения, в таблице перестановок, при определении повторов позиции в строке игры. Программа может при счете даже дотянуться до базы данных эндшпилей и иметь точную оценку некоторой позиции. Если у нас есть хеш-ключ позиции, то число уточняющих сравнений минимально (в десятки и сотни тысяч раз меньше).

Таким образом, определение ключа позиции является принципиальным моментом при построении хорошей шахматной программы. Какие требования предъявляются к хеш-ключу? Он должен вычисляться максимально быстро и как можно точнее характеризовать данную позицию, т. е. должно быть как можно меньше коллизий ключа (хеш-индекса). Хеш-ключ может быть "связан" с позицией, и тогда будет много повторов. Например, если при определении хеш-ключа используется номер клетки доски и величина фигуры в этой клетке, то коллизии неизбежны, и их будет очень много. Например, `key = NSquare × ValuePiece`, где `NSquare` — номер клетки, `ValuePiece` — фигура в клетке.

Для определения не связанного с позицией ключа используется техника ZObrist-ключей. Суть ее в том, что имеется массив вычисленных заранее

чисел, не связанных с данной позицией. Массив может быть трехмерным: `zobrist[pieceType][colType][square]`, где `pieceType` — код фигуры в данной клетке, `colType` — цвет фигуры, `square` — номер клетки. Размеры массива будут, очевидно, 6 × 2 × 64.

Для хранения значений ключей используется тип данных `unsigned_int64`. Это 64-битовое без знака. Раньше использовался `unsigned long` (32-битовое без знака), но сейчас в этом нет необходимости. Данное число не отражает позицию на доске однозначно, т. к. для точного отражения нужно 4 × 64 = 256-битовое число. Но, тем не менее, коллизии достаточно редки. Таблица заполняется случайными числами. Для инициализации можно использовать стандартный генератор случайных чисел:

```
typedef unsigned __int64 U64
U64 rand64(void)
{
  return rand() ^ ((U64)rand() << 15)  ^ ((U64)rand() << 31) ^
                  ((U64)rand() << 47)  ^ ((U64)rand() << 59);
}
```

Используется операция XOR (исключающее ИЛИ), которая в каждом бите возвращает 1, только если биты различны. XOR двух одинаковых чисел вернет нули во всех разрядах. Вместо XOR можно использовать обычное сложение, но считается, что XOR дает лучшее распределение хеш-индекса. Так ли это? Я использовал XOR.

Теперь, когда у нас есть таблица с вычисленными значениями для всех возможных случаев на доске, вычислить хеш-индекс совсем просто. Например, для белого коня на e5

```
key = zobrist[KNIGHT][WHITE][E5]
```

Если нужно иметь полный ключ, нужно сканировать всю позицию, и каждый раз выполнять XOR с новым значением. Особенностью операции XOR является то, что когда мы делаем два раза XOR с каким-то числом, то получаем исходное число. Это дает возможность нам убрать фигуру из ключа, просто применив к ключу операцию XOR с ней. Если эта фигура была в ключе, значит, мы ее вычли, если не было, то добавили. Это дает нам возможность просто по шагам вычислять хеш-ключ.

Хеш-таблицы

Хеширование является одним из самых мощных способов повышения производительности компьютера. Не знаю, кому как, а мне хеширование всегда не очень нравилось из-за своей половинчатости, приблизительности, если можно так сказать. Хеширование не является точным методом. Тем не ме-

нее альтернативы ему нет, и если мы хотим получить высокие скоростные показатели, хеширование нам потребуется.

Для тех, кто не знаком с предметом, несколько слов о хешировании вообще. Хеширование является методом ускорения поиска. Компилятор встречает некоторую лексему и пытается найти ее в своей базе данных или таблице символических имен. Таблица символических имен современного компилятора в MS Windows может содержать несколько тысяч или десятков тысяч лексем. Для ускорения поиска был придуман следующий прием. Компилятор, найдя лексему в тексте программы, определяет ее хеш-ключ. Наша лексема — это просто слово, состоящее из последовательности кодов символов. Для определения хеш-кода следует сложить все коды символов. Лексем с таким хеш-кодом в базе данных уже значительно меньше. Компилятор определяет номер списка с заданным кодом и перебирает этот список, а не всю базу данных. Все это в программе может выглядеть примерно так:

```
//элемент связного списка
typedef struct _tagHIten{
   char lexem[40];           //лексема
   struct _tagHItem* next;   //следующий элемент списка
}THItem,*PHItem;

//количество связных списков
#define MAX_H_INDEX 1000
//массив связных списков
PHItem hashTable[MAX_H_ITEM];
//функция возвращает хеш-код слова
int GetHKey(char *lexem)
{
   unsigned int code = 0;
   int n = 0;
   while(lexem[n])
   {
      code = code ^ (code << n);
      n++;
   }
   return code % MAX_H_ITEM;
}
//поиск лексемы в таблице
int search(char *lexem)
{
  PHItem p;
  // начало списка с данным хеш-кодом
  p = hashTable[GetHKey(lexem)];
```

```
  while(p)
  {
     if(!strcmp(p->lexem,lexem))
        return 1; //точное совпадение
     p = p->next; //следующий элемент списка
  }
  return 0;
}
```

Хеш-код определяется не сложением кодов символов, а применением операции исключающего ИЛИ. Это дает более равномерное распределение. Хеш-индекс не может быть больше размера таблицы, для этого возвращается остаток от деления на размер. Могут быть коллизии: находится несколько лексем с одним хеш-индексом. Для проверки точного совпадения нужно проверить весь список. Основная проблема состоит в поиске хеш-индекса, максимально точно идентифицирующего данную лексему. Идеальный хеш-индекс однозначно должен определять лексему.

При программировании шахмат тоже используется хеширование. Основная особенность состоит в том, что хеш-индекс должен очень точно отражать позицию, и поиск должен производиться максимально быстро. У нас даже нет времени на перебор списков с одним индексом. Списков, как правило, нет вообще. Для каждого значения хеш-ключа существует только одна позиция. Если возникла коллизия и встретилась позиция с таким же ключом, то она записывается поверх прежней. Зачем при программировании шахмат и других подобных игр хеширование и запоминание позиций? Дело в том, что при переборе мы имеем не дерево игры в прямом смысле, а граф. Позиции повторяются через некоторое количество полуходов, и даже в одну и ту же позицию можно прийти различными путями. Можно воспользоваться некоторой информацией о позиции, которая уже встречалась. Самое простое и эффективное — это использовать позицию, чтобы улучшить порядок ходов. Это особенно выразительно при итеративных углублениях. Улучшение упорядочивания ходов — основное назначение хеш-таблицы. Перестановки или повторы позиций тоже играют свою роль. Особенно их много в конце игры.

Если хеш-ключ достаточно точно отражает шахматную позицию, можно даже воспользоваться результатом счета и не перебирать вовсе. Если данная позиция была уже просчитана на такую же или большую глубину, можно просто вернуть результат и не считать вовсе.

Хочется отметить, что хеш-таблицы являются практически неисчерпаемым источником ошибок в шахматных программах. Ошибки эти, как правило, трудно обнаруживаемые и очень досадные. Даже если бы мы имели идеальный ключ, и не было бы коллизий, и наша программная реализация была бы стопроцентно верной, мы все равно получили бы одну досадную про-

блему при использовании результата из хеш-таблицы. Проблема эта называется нестабильностью поиска. Это когда расчет одного хода на одну глубину одной и той же функцией возвращает разный результат. Нестабильность поиска может быть вызвана многими причинами. Это — введение искусственных границ alpha-beta, различные схемы форвардного отсечения (отсечение ветвей без предварительного счета). Основной причиной является использование результата из хеш-таблицы. Даже обычный алгоритм NegaScout может из-за повторных переборов усиливать нестабильность поиска. Типичный пример:

```
tmp = -NegaScout(-(alpha+1),-alpha);
if(tmp > alpha  &&  tmp < beta)
 tmp = -NegaScout(-beta,-tmp);
```

Приведенный пример корректно работает при стабильности поиска. Если в шахматном двигателе есть нестабильность, лучше использовать следующую схему:

```
tmp = -NegaScout(-(alpha+1),-alpha);
if(tmp > alpha  &&  tmp < beta)
  tmp = -NegaScout(-beta,-alpha);
```

При повторном переборе с полным окном мы не повышаем alpha полученным ранее результатом, а используем текущие пределы alpha и beta. Нестабильность поиска является незначительной помехой при грубом усилии, но в выборочном поиске или извлечении главной строки изменения может разрушить всю организацию программы. Но перейдем к практической реализации хеш-таблицы для шахмат.

Основной проблемой является получение ключа, не связанного с позицией. Это достигается ставшей уже стандартной методикой ZObrist. Если нам, например, нужно получить ключ для белого слона, стоящего в поле a4, мы поступаем следующим образом:

```
key = randTable [BISHOP] [WHITE] [A4];
```

Полный ключ для всей позиции вычисляется только в самом начале при помощи операции XOR. Дальше идет его пошаговая корректировка. Например, белая ладья переместилась из a1 в c1. Чтобы внести соответствующие изменения в хеш-ключ, мы должны сделать следующее:

```
nextKey = key ^ randTable [ROOK] [WHITE] [A1] ^
                randTable [ROOK] [WHITE] [C1];
```

Это позволяет быстро по шагам корректировать хеш-ключ. Он автоматически отслеживает все возможные повторы позиций. Системы программирования, как правило, не позволяют получить случайное 64-битовое без знака. Можно легко написать свою функцию `rand64`, дающую такое число:

```
typedef unsigned __int64  U64
   (U64)rand() ^ ( (U64)rand() << 15) ^ ( (U64)rand() << 30) ^ ...
```

При счете, т. к. мы используем alpha-beta пределы, оценка узла при записи в хеш-таблицу может быть трех видов.

- Меньше или равно alpha: мы не имеем лучшего хода, а только результат счета, который представляет собой максимум, достигнутый для этого узла. Реальной оценки мы не знаем, т. к. в процессе перебора сработали отсечения за alpha-beta пределы.
- Больше alpha, но меньше beta: мы имеем абсолютно точную оценку, как если бы мы перебирали алгоритмом NegaMax без всяких alpha-beta пределов. Имеется также лучший ход, посредством которого был получен данный результат.
- Больше или равно beta: оценка не точная, но реальная. При полном переборе для данного узла это минимум. Имеется также лучший ход.

Если мы в некотором узле нашли результат из хеша и хотим его использовать, то нужно проверить, какого типа этот результат. Если это оценка первого вида, то использовать ее можно, только если она меньше или равна текущему значению alpha. Само собой, глубина узла из хеша должна быть, как минимум, равна глубине нашего текущего узла. Если оценка из хеша второго вида, т. е. точная, то использовать ее можно в любом случае. Если оценка узла из хеша третьего вида, то это минимум для данного узла, использовать его можно только в том случае, если он больше или равен текущему значению beta. Что касается этого третьего вида оценки, мы можем повышать alpha, т. к. alpha не может быть меньше данного минимума. При использовании результата из хеша хочется отметить один тонкий момент. В случае взятия короля оценка узла равна разности недосягаемого максимума `INFINITY` и глубины узла от 0, чтобы не оттягивать мат. При манипуляциях с оценкой из хеша нужно производить корректировку `ply`, чтобы соблюсти корректность глубины, на которой был поставлен мат:

```
if(score > INFINITY - MAX_PLY) score += ply;
else if(score < -INFINITY + MAX_PLY) score -= ply;
```

При извлечении из хеша:

```
if(score > INFINITY-MAX_PLY) score -= ply;
else if(score < -INFINITY+MAX_PLY) score += ply;
```

Оценка будет точно отражать глубину, на которой взяли короля. Пример, отражающий использование хеш-таблицы:

```
//виды элементов хеша
#define EMPTY 0
#define EXACT 1
```

```
#define ALPHA 2
#define BETA  3
typedef struct tagHash{
  U64 key;     //хеш-ключ
  int depth;   //глубина узла
  int flags;   //тип узла
  int score;   //оценка узла
  MOVE move;   //лучший ход (если есть)
}HItem;
#define MAX_H_ITEM (1 << 18)
//сама таблица
HItem hash[MAX_H_ITEM];
#define MAX_PLY 100 //максимальная глубина
//запись в таблицу
void RecordHash(int depth, int score, int flag, U64 key, MOVE* move,
                int  ply)
{
   HItem *p = &hash [ key & (MAX_H_ITEM-1) ];  //остаток от деления
   //не перезаписываем узлы с лучшим ходом менее
   //ценными узлами с флагом ALPHA
   if(flag == ALPHA  &&
      (p->flags == EXACT || p->flags == BETA)
      return;
    //схема замены
    //более глубокие узлы не затираем
   if(p->depth <= depth)
   {
     if(score > INFINITY - MAX_PLY) score += ply;
     else if(score < -INFINITY + MAX_PLY) score -= ply;
     p->depth = depth;
     p->score = score;
     p->flags = flag;
     p->key   = key;
     if(flag != ALPHA)
     {
        //скопируем лучший ход
        {...}
     }
   }
}

//использование хеша в функции поиска
int ALphaBeta(int alpha,int beta,int depth,int ply,U64 key)
{
   int oldAlpha = alpha; //запомним alpha
```

```
//получим ход из хеша
HItem *p = &hash [ key & (MAX_H_ITEM-1) ];
if(p->flags != EMPTY  && p->key == key)
{
   //попробуем использовать оценку из хеша
   if(ply > 0  &&  p->depth >= depth)
   {
      int score;
      //корректировка оценки
      score = p->score;
     if(score > INFINITY-MAX_PLY) score -= ply;
     else if(score < -INFINITY+MAX_PLY) score += ply;
      switch(p->flags)
      {
        EXACT:BETA: {
           if(score > alpha) alpha = score;
           if(alpha >= beta) return beta;
        }break;
        ALPHA:{
           if(score <= alpha) return alpha;
        }break;
      }//switch
   }//endif
   if(p->flag != ALPHA)
   {
           //попробуем лучший ход из хеша
           MakeMove(&p->move);
           tmp = -ALphaBeta(-beta,-alpha,depth-1,ply+1,NextKey(key,
                              &p->move);
           UnMakeMove(&p->move);
           if(tmp > alpha)
           {
              RecordHash(depth, tmp, tmp<beta?EXACT:BETA, key,
                         &p->move, ply);
              alpha = tmp;
              if(alpha >= beta) return beta;
           }
   }
}//endif

GenerateAllMoves();
//перебираем все перемещения
//и записываем в хеш с соответствующим флагом
while(move)
```

```
    {
                MakeMove(move);
                tmp = -ALphaBeta(-beta,-alpha,depth-1,ply+1,
                                 NextKey(key,move);
                UnMakeMove(move);
                if(tmp > alpha)
                {
                   RecordHash(depth, tmp, tmp<beta?EXACT:BETA,
                              key,move, ply);
                   alpha = tmp;
                   if(alpha >= beta) return beta;
                }
      move = move->next;
    }//while

    //если программа дошла до этой точки и
    //значение alpha не повышалось, то запишем
    //данную позицию с флагом ALPHA
    //постараемся не перезаписать в хеше более
    //ценный узел с лучшим ходом
    if(alpha == oldALpha)
       RecordHash(depth, alpha, ALPHA,key,NULL,ply);
    return alpha;
}
```

В данном примере значение alpha повышалось всегда, если флаг хода из хеша был EXACT или BETA. Это увеличивает нестабильность поиска, но ошибки здесь нет. Обратите внимание на схему замены. Размер хеш-таблицы, как правило, всегда ограничен, и для одного и того же хеш-индекса могут быть узлы с разной глубиной и одним адресом в хеше. Здесь использовалась схема замены таким же узлом или более глубоким. Поверхностные узлы не затирают более глубокие и ценные. Узлы без лучшего хода (с флагом ALPHA) не затирают более ценные узлы с лучшими ходами. Напомним, что основной эффект хеш-таблицы заключается в использовании итеративных углублений, и узлы с ходами предыдущей итерации затирать нельзя. Так как размер хеша ограничен, то узлы с флагом ALPHA можно не использовать вообще, следовательно, флаг для узла из хеша не обязателен. Его оценкой всегда повышается текущее значение alpha. Узлы с флагом ALPHA имеют более низкий приоритет, по сравнению с другими узлами.

Помимо приведенной схемы замены существуют и другие. Например — "заменять всегда". В случае единственной хеш-таблицы предпочтительнее схема "такой же или более глубокий". Хороший эффект дают две хеш-таблицы: основная с такой схемой замены и вспомогательная (меньшего размера) со схемой замены всегда. Если вы хотите выжать из программы

все, можно ввести вспомогательную хеш-таблицу с незначительным количеством элементов (например, 0 × FFFF). Особенно неплохо использование двух хеш-таблиц сочетается с внутренними итеративными углублениями. Если нет хода в хеше, мы перед началом перебора пытаемся получить лучший ход перебором на глубину N−2. Это, кроме всего прочего, упорядочивает вспомогательную хеш-таблицу и перебор (особенно на большую глубину). В реальных программах размер хеша всегда ограничен реальной величиной. Я обнаружил, что увеличение моей таблицы в два раза дало такой же эффект, что и использование маленькой вспомогательной таблицы со схемой "заменять всегда". Все это очень зависит от конкретной реализации.

В приведенном примере, помимо коллизий хеш-индекса (которыми можно пренебречь, т. к. иного выхода нет), есть опасность при использовании лучшего хода из хеша. Представьте себе ситуацию, когда лучший ход для узла был получен на глубине 10, а потом мы этот же узел считаем на глубине 2 и пытаемся использовать лучший ход из хеша. Из-за повторов позиций такая ситуация вполне может быть. Во избежание этого нужно при использовании лучшего хода из хеша проверять его легальность. Если в исходной клетке стоит та же фигура, то она вполне может пойти в целевую клетку. В целевой клетке не должно быть ничего или фигура противника. Для дальнобойных фигур нужно строить линию, чтобы убедиться, что между исходной и конечной клеткой нет ничего. В целом, схема вполне работоспособная, и даже использование результата из хеша не ведет к развалу просчета. Результат, как правило, используется на большой глубине. Чем больше глубина, тем меньше значение погрешности при вычислении.

При создании хеш-таблицы размер памяти, как правило, ограничен, а хочется поместить как можно больше элементов. Отсюда желание сделать размер одного описателя хеш-таблицы минимальным. Некоторые поля можно представить более компактно, в виде байта. Основной объем занимает хеш-ключ (8 байт) и описатель перемещения. В шахматах основную сложность представляют нестандартные ходы: рокировки и взятия пешки через битое поле. Двумерный массив игры [8][8] может быть заменен одномерным [64]. Таким образом, вместо двух индексов используется один.

Можно воспользоваться следующей схемой. Функция, генерирующая перемещения, возвращает, как правило, указатель на первый элемент списка перемещений. Для описателей перемещения она использует буфер, откуда берутся свободные описатели и соединяются в список. Буфер может располагаться в стеке функции `ALphaBeta`. Важно то, что к любому описателю перемещения можно обратиться по его адресу и номеру в буфере. Хеш-таблица может хранить только номер описателя перемещения в буфере. Таким образом, при использовании хода из хеша не нужно проверять его легальность в данной позиции, и не страшны коллизии хеш-индекса. Единственный недостаток данной схемы — функция `Generate`, генерирующая перемещения, должна быть вызвана перед использованием хода из хеша.

В хеш-таблице поместится больше элементов, что гораздо важнее. Кроме того, упрощается вся схема, и исключаются многочисленные ошибки, связанные с использованием хеш-таблиц. Для простой программы такая схема имеет несомненные преимущества. Описатель элемента хеш-таблицы:

```
typedef struct{
   U64 key;                        //хеш-ключ
   unsigned char moveNumber;       //номер лучшего хода
   signed   char depth;            //глубина
}THItem;
```

Он имеет размер всего 10 байт. Если программа не собирается использовать хеш на большой глубине, или размер хеша ограничен, можно вместо типа U64 (64-битовое без знака) использовать 32-битовое без знака. Размер описателя хеш-таблицы будет всего 6 байт! Какую схему использовать — дело вкуса и опыта в программировании.

Повторы позиций

Определение повторов позиций — не такая тривиальная тема. Она затронута многими корифеями шахматного программирования. Я не буду пытаться объять необъятное, а покажу, как простейшим способом научить программу распознавать повторы позиций.

Допустим, наше перемещение представлено в виде

```
a1b1 a8b8
```

В этой строке игры показано, как белая ладья пошла с a1 на b1, а черная — с a8 на b8. Как может исходная позиция повториться в одной строке игры? Очевидно, ладьи должны встать на исходные позиции:

```
b1a1 b8a8
```

Полная строка, приводящая к повтору, выглядит следующим образом:

```
a1b1 a8b8 b1a1 b8a8
```

Повтор получился через 4 полухода. Мы и будем рассматривать только повторы через 4 полухода в одной строке. Если позиция повторится 3 раза, то строка выглядит

```
a1b1 a8b8 b1a1 b8a8    a1b1 a8b8 b1a1 b8a8    a1b1 a8b8 b1a1 b8a8
```

Как программа в процессе счета может понять, что есть повторение три раза через 4 полухода? Самый простой способ заключается в ZObrist-ключах. Это 8-байтовые хеш-ключи позиции. Вместо строки игры мы работаем со строкой ключей. Определение повтора не составляет труда. Перед началом счета мы инициализируем массив с ZObrist-ключами последних

8—9 позиций. Во время счета переменная ply отслеживает глубину узла в строке от 0. Таким образом, при равенстве текущего ключа и ключа в строке на 4 и на 8 позиций ранее возвращаем 0 — оценку draw. Выглядит этот программный код примерно так:

```
//позиция с индексом 8 соответствует корню дерева перебора при ply = 0
ZObrist  line[100];
//где-то в функции AlphaBeta
int AlphaBeta(int alpha,int beta, int ply, U64 key)
{
   line[ply+8] = key;
   if( ply > 0                  &&
       line[ply+8-4] == key  &&
       line[ply+8-8] == key)
         return 0; //draw

  {....}
}
```

Здесь есть одна небольшая тонкость. Проверять повторы позиций можно только в пределах гарантированного полного перебора. Если мы используем нулевой ход или другой метод подрезки дерева, нужно оставлять несколько полуходов в начале перебора, где эти методы не используются. Если программа пытается раскрыть главную строку изменения, возможно, что она сможет отслеживать повторы позиций и глубже. Если у нас есть гарантированные 4—5 полуходов полного перебора, то мы сможем научить программу распознавать элементарные зацикливания в игре, чего для практических целей вполне достаточно.

Простейшие повторы через 4 полухода можно распознавать и без ZObrist-ключей, написав следующий блок программного кода:

```
TMove line[20];
//ход 12 - корень дерева перебора
int AlphaBeta(PMove node,  int ply,  ...)
{
    line[ply+12] = node;
    if(ply > 0  &&  ply < 5)
    {
      int n;
       for(n = 0; n < 4; n++)
       {
         if(
            !CompareMove(line[ply+12-n], line[ply+12-4-n) ||
            !CompareMove(line[ply+12-n], line[ply+12-8-n)
          )goto next;
```

```
        }//for
        return DRAW;
    }
next:
  {...}
}
```

С ZObrist-ключами получается намного компактнее.

Оценка `draw` в большинстве случаев равна 0. Чтобы заставить программу играть дальше, если у нее небольшой минус в оценке, пытаются возвращать отрицательную величину. Если кто-то желает углубиться в эту тему, ему можно посоветовать почитать корифеев вроде Кена Томпсона.

Детекторы шахов

Шах является исключительной ситуацией и требует незамедлительной реакции. При шахе игра как бы замирает, и счет продолжается на один полуход. Это логически вытекает из того, что король — главная фигура.

Без детектора шахов не обходится ни одна шахматная программа. Причина в том, что программа должна находить наиболее сильные перемещения, стремиться поставить мат противнику и усилить давление на его короля, а шах — это потенциально наиболее сильное перемещение. Шахи очень сказываются на эффекте горизонта. Если программа считает, например, на 6 полуходов, то из-за шаха в конце строки она может выдать неверный результат: ей может показаться, что она теряет ферзя, а нужно только уклониться от шаха. Если мы используем выборочные продления и не сокращаем глубину перебора при шахе, то программа может находить очень глубокие форсированные маты, и в этих ситуациях сказывается все преимущество машины перед человеком. Продление статической оценкой — это примерно то же самое, оно используется для сглаживания эффекта горизонта или нахождения в глубине очень длинных тактических выпадов, построенных на шахах.

Так или иначе, детектор шахов — это одна из основных частей шахматной программы, и имеет смысл рассмотреть его подробнее.

Какие основные требования предъявляются к детектору шахов?

- Он должен работать максимально быстро. Отсутствие шаха должно определяться практически сразу, чтобы не иметь непроизводительных расходов.
- Он должен определять шах, объявленный последней ходившей фигурой противника. Если в строке игры шах уже был объявлен другой фигурой, то короля уже съели в предыдущем узле.

Примечание

Имеются еще так называемые вскрытые шахи, когда последняя ходившая фигура противника освободила линию атаки для его дальнобойной фигуры. Это довольно редкая ситуация, она может не рассматриваться в детекторе шахов или рассматриваться до определенной глубины.

Как видим, детектор шахов — довольно условная вещь и даже не определяет всех шахов. Тем не менее на практике он необходим.

Для определения шахов можно использовать маленькую хитрость, отбрасывающую большинство перемещений, которые не могут объявить шаха. Хитрость заключается в следующем. Нам известны координаты короля и координаты фигуры (последней ходившей), которая может объявить шах. Построим некоторый массив для короля, в каждой ячейке которого задано, может ли фигура объявить отсюда шах или нет. Для коней это истинно всегда (если клетки для коней совпадают), для дальнобойных фигур — это только потенциальная возможность. Для ее уточнения нужно построить линию от фигуры до короля и убедиться, что на ней ничего нет.

Для того чтобы воспользоваться этим массивом, нужно преобразовать координаты нашего короля и фигуры в координаты вспомогательного массива так, чтобы король оказался на предназначенном ему месте. Сделать это несложно.

Как в каждой ячейке массива обозначить, может ли фигура объявить отсюда шах? Для этого можно воспользоваться двоичной арифметикой. Все биты ячеек массива изначально нулевые:

```
0 0 0 0 0 0 0 0
```

Мы можем определить ненулевой бит для ферзя:

```
0 0 0 0 0 0 0 1
```

Сложнее, если две фигуры могут объявить шах из одного места, например слон и ферзь. Для слона мы используем следующий бит:

```
0 0 0 0 0 0 1 0
```

Тогда в ячейке массива, откуда и слон, и ферзь могут объявить шах, будет изображено:

```
0 0 0 0 0 0 1 1
```

Если представить это на языке программирования, то это будет подобно следующему:

```
CONST
  CHECK_QUEEN  = 1;
  CHECK_BISHOP = 2;
```

Чтобы определить возможность шаха только для ферзя, мы должны применить к ячейке массива операцию AND с кодом ферзя:

```
(value AND CHECK_BISHOP)
if (value AND CHECK_BISHOP) <> 0 then ...
```

Поразрядная логическая операция AND оставит только те биты, которые установлены в 1 у обоих операндов, остальные сбросит в 0.

Для ладьи:

```
0 0 0 0 0 1 0 0
```

После того как мы определили возможность вероятного шаха, нужно построить линию для дальнобойных фигур, чтобы убедиться, что на ней ничего нет. Для коня линию строить не нужно. Для него вероятный шах совпадает с действительным. Для пешки необходимо просто проверить, по ходу ли пешки клетка короля. Черные пешки двигаются вниз, а белые вверх. Для черных нужно проверить, что координата Y пешки меньше координаты короля, для белых — наоборот.

Несколько слов о построении линии. Для двумерного массива, представленного как одномерный, нужно знать лишь одно приращение координаты. Для того чтобы быстро находить это приращение, можно ввести еще один вспомогательный массив.

Игровое поле у нас представлено массивом 16 × 16 с массивом игры 8 × 8 в центре и флагом выхода за пределы по краям. Вот пример детектора шахов:

```
{////////////// некоторые предшествующие определения ///////}
TYPE
 TPos16 = array[0..16*16-1] of integer;  {позиция}
CONST
 {фигуры}
 KING   = 1;
 QUEEN  = 2;
 ROOK   = 3;
 BISHOP = 4;
 KNIGHT = 5;
 PAWN   = 6;  {итого первые 3 бита}
 {цыета}
 CBLACK = 64;
 CWHITE = 128;
 CCOLOR  = CBLACK + CWHITE;  {маска для выделения цвета}
 CFIGURE = 7;                {маска для выделения фигуры}
 COUT    = (1 shl 9);        {выход за  пределы доски}
var glUpFColor; {цвет верхних фигур}
```

```
{////////////////////////   детектор шахов //////////}
{вероятный шах}
CONST
  {коды фигур для вероятного шаха}
  N_QUEEN = 1;
  N_ROOK  = 2;
  N_BISHOP= 4;
  N_KNIGHT= 8;
  N_PAWN  = 16;
  N_NO    = 32;
  {массив, где по основному коду фигуры (1..6) записан код
  фигуры для детектора шахов
  }
  _codeF:array[0..6] of integer =
(N_NO,N_NO,N_QUEEN,N_ROOK,N_BISHOP,N_KNIGHT,N_PAWN);

  {массив вероятного шаха, в каждой ячейке
  сумма кодов фигур, которые могут объявить шах
  из этой ячейки}
  BASE = (16*7)+8-1; {координата виртуального короля}
  X = 0;
  _check:TPos16 =
  (
 5, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5, 0,
 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 5, 0, 0,
 0, 0, 5, 0, 0, 0, 0, 3, 0, 0, 0, 0, 5, 0, 0, 0,
 0, 0, 0, 5, 0, 0, 0, 3, 0, 0, 0, 5, 0, 0, 0, 0,
 0, 0, 0, 0, 5, 0, 0, 3, 0, 0, 5, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 5, 8, 3, 8, 5, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 8,21, 3,21, 8, 0, 0, 0, 0, 0, 0,
 3, 3, 3, 3, 3, 3, 3, X, 3, 3, 3, 3, 3, 3, 3, 3,
 0, 0, 0, 0, 0, 8,21, 3,21, 8, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 5, 8, 3, 8, 5, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 5, 0, 0, 3, 0, 0, 5, 0, 0, 0, 0, 0,
 0, 0, 0, 5, 0, 0, 0, 3, 0, 0, 0, 5, 0, 0, 0, 0,
 0, 0, 5, 0, 0, 0, 0, 3, 0, 0, 0, 0, 5, 0, 0, 0,
 0, 5, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 5, 0, 0,
 5, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 5, 0,
 0, 0, 0, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 5
 );

  {приращения при построении линии}
  _del:TPos16 =
```

```
  (
-17,  0,  0,  0,  0,  0,  0,-16,  0,  0,  0,  0,  0,  0,-15,  0,
  0,-17,  0,  0,  0,  0,  0,-16,  0,  0,  0,  0,  0,-15,  0,  0,
  0,  0,-17,  0,  0,  0,  0,-16,  0,  0,  0,  0,-15,  0,  0,  0,
  0,  0,  0,-17,  0,  0,  0,-16,  0,  0,  0,-15,  0,  0,  0,  0,
  0,  0,  0,  0,-17,  0,  0,-16,  0,  0,-15,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,-17,  0,-16,  0,-15,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0,  8,-17,-16,-15,  8,  0,  0,  0,  0,  0,  0,
 -1, -1, -1, -1, -1, -1, -1,  X,  1,  1,  1,  1,  1,  1,  1,  1,
  0,  0,  0,  0,  0,  8, 15, 16, 17,  8,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0,  0, 15,  0, 16,  0, 17,  0,  0,  0,  0,  0,  0,
  0,  0,  0,  0, 15,  0,  0, 16,  0,  0, 17,  0,  0,  0,  0,  0,
  0,  0,  0, 15,  0,  0,  0, 16,  0,  0,  0, 17,  0,  0,  0,  0,
  0,  0, 15,  0,  0,  0,  0, 16,  0,  0,  0,  0, 17,  0,  0,  0,
  0, 15,  0,  0,  0,  0,  0, 16,  0,  0,  0,  0,  0, 17,  0,  0,
 15,  0,  0,  0,  0,  0,  0, 16,  0,  0,  0,  0,  0,  0, 17,  0,
  0,  0,  0,  0,  0,  0,  0, 16,  0,  0,  0,  0,  0,  0,  0, 17
 );

{возвращает true, если фигура f с индексом NFigure объявляет
шах королю с индексом NKing
счет ведется для позиции 16×16 glPos
}
FUNCTION InCheck(F,NFigure,NKing:integer):boolean;
VAR
 del,N:integer;
BEGIN
    InCheck := false;
    {вероятный шах}
    if (_check[ NKing-NFigure+BASE ] and _codeF[F and CFIGURE]) <>
       (_codeF[F and CFIGURE]) then
    exit;
    {для пешки и коня - своя проверка}
    case (f and CFIGURE) of
     KNIGHT: begin
               InCheck := true;
               exit;
             end;
     PAWN:begin
            if (F and CCOLOR) = glUpFColor then
            begin
              if NFigure < NKing then
                InCheck := true;
```

```
            end else begin
               if NFigure > NKing then
                 InCheck := true;
            end;
            exit;
          end;
    end; {case}
    {строим линию между двумя точками}
    del := _del[NFigure - NKing + BASE];
    if del = 0 then exit;
    N := NKing + del;
    {если вышли за пределы
     поля игры,  флаг выхода}
    while (N <> NFigure) and (glPos[N] = 0) do
       inc(N,del);

    InCheck := (N = NFigure);
END;
```

Данный пример взят из учебной программы, и для пояснения можно посмотреть пример генерации перемещений из той же программы. (Если возникло несоответствие типов данных или других особенностей.)

Как видите, кода довольно много для скоростной функции, но нужно учесть, что большая часть функции будет пропущена. На языке C, чтобы избежать накладных расходов, можно вероятное определение шаха оформить в виде макроса с параметрами и основную функцию не вызывать, если макрос вернет 0.

Недействительное перемещение

Нулевой ход, или недействительное перемещение, используется в большинстве современных программ. История его открытия довольно туманна. Такое впечатление, что он известен был всегда, но об этом открыто не говорили. Существовала тенденция сокрытия информации о шахматных программах. Каждому казалось, что он открыл нечто уникальное, до чего никто не додумался, но потом постепенно становилось известно, что все используют, в принципе, одни и те же методы. Так было и с нулевым ходом. Официально его еще не существовало. Франц Морш услышал о нем в 1986 году в Германии на мировом первенстве среди шахматных программ, где Дон Бил участвовал с шахматной программой, использующей технику NullMove (Нулевой ход). Франц Морш использовал его в своей программе Fritz, которая и по сей день входит в тройку сильнейших программ. Позже, когда о нем написали Donninger, Beal, Goetsch & Campbel, нулевой ход стал всеобщим достоянием.

Нулевой ход позволяет очень просто и на сравнительно неплохом уровне организовать выборочный поиск в шахматной программе. Особенно хорошо он подходит для схем, где осуществляется предварительный осторожный стратегический расчет (чтобы программа хоть как-то шевелила фигурами), а затем включается на полную мощность счет одного материала с обрезкой неправдоподобных ветвей нулевым ходом. Таким образом, при некоторой технике программирования осуществляется расчет на 10 полуходов и более, включая взятия до конца. Усиливаются обычно строки с взятиями и шахами. Такая программа просчитывает огромное количество позиций в секунду (оценочная функция у нее, собственно говоря, отсутствует) и производит очень хорошее впечатление в острых тактических ситуациях. Если человек применил беспорядочную тактику, то он пропал, будь он хоть сам Бобби Фишер. В соревнованиях с другими программами такая программа ведет себя превосходно, т. к. стратегический расчет у нее предварительный и очень осторожный, а дальнобойной проверке материала не нужно захватывать пространство доски. Основная слабость других шахматных программ в недостаточной позиционной оценке. Программа с подобной схемой избегает этих ошибок. Если у нее сорвался предварительный стратегический просчет, и проверка материала его не одобрила, программа может просто начать двигать ладьей туда и обратно. Собственно говоря, она за пределами дебютных справочников не играет вовсе, пока противник не подставится. Несмотря на перечисленные недостатки, написать функцию, планирующую осторожную стратегическую игру для такой программы, очень непросто. Люди, как правило, восхищаются такими программами, но сами с ними не играют, а предпочитают что-нибудь поживее. Никому не интересно играть с вычислительным комбайном без малейшего признака фантазии.

Впрочем, я отвлекся от нулевого хода. Он используется в большинстве современных программ. Не все программы считают только материал. Многие считают и позиционную, и материальную оценку сразу. Некоторые получают список легальных ходов, считая только материал, и подают этот список на вход функции, планирующей стратегию.

Нулевой ход позволяет сравнительно просто настроить выборочный поиск и достигать больших глубин счета, немыслимых при полном переборе. Нулевой ход работает сравнительно точно и позволяет получить качество оценки, в большинстве случаев не уступающее полному перебору. Он идеально подходит для усиленной дальнобойной проверки материала, т. к. его огрехи при этом практически не видны.

Нулевой ход используется, как правило, для отсечения ветвей дерева без полного счета. Есть и другие использования нулевого хода (определение угрозы). Вместе с тем важно понимать, что нулевой ход является эвристикой, и использование его для отсечения ветвей дерева перебора возможно не во всех играх и требует особой осторожности. В шашках, например, его использование совершенно невозможно.

В чем состоит суть эвристики нулевого хода? Замечено, что в практически любой позиции в шахматах пропуск хода хуже, чем возможность сделать ход. Если у игрока есть ход, он может получить результат не хуже, чем если бы его у него не было. Исключения редки, и класс позиций, где право хода мешает, получил даже название — цуцванг. Например, в ситуации, где ферзь с королем ставят мат одинокому королю, цуцванг встречается чуть ли не на каждом ходу. Ферзь и король не нападают прямо на одинокого короля, а вынуждают его двигаться навстречу гибели. Нулевой ход здесь не работает.

Как используется нулевой ход? У нас в каждом узле есть две величины — alpha и beta. Alpha — это наш максимум, достигнутый в результате счета, beta — максимум противника. Если мы в результате счета получили оценку больше или равную beta, то счет можно прекратить, т. к. он дальше не имеет смысла. Когда перебор вернется в точку, где было достигнуто значение beta противника, результат все равно отвергается, т. к. он не больше максимума противника в этой точке. Данный алгоритм позволяет отсечь без какой-либо погрешности ненужные ветви дерева перебора и получить результат такой же точности, как и полный перебор на данную глубину:

```
int AlphaBeta(int alpha,int beta, int depth)
{
  //закончилась глубина?
  //вернем оценку узла
  if( depth <= 0) return Evaluate();

  //список ходов
  PMove move = Generate();
  while(move)
  {
    MakeMove(move);  //сделали ход
    int tmp = -ALphaBeta(-beta,-alpha,depth-1);
    UnMakeMove(move); //восстановили позицию
    //если результат улучшает alpha,
    //увеличим ее
    if(tmp > alpha) alpha = tmp;
    //если максимум противника,
    //дальше продолжать нет смысла
    if(alpha >= beta) return alpha; //или beta
    //следующий ход из списка
    move = move->next;
  }
  //вернем наш максимум
  return alpha;
}
```

Данный алгоритм прекращает перебор, если мы получили оценку больше или равную beta. Если в узле, прежде чем сделали хоть один ход, и даже прежде чем получили перемещения, предоставить право хода противнику (он сделает два хода подряд), а полученная оценка все равно больше или равна beta, то предполагается, что результат счета тем более будет больше beta. Выглядит это примерно так:

```
int AlphaBeta(int alpha,int beta, int depth)
{
  //закончилась глубина?
  //вернем оценку узла
  if( depth <= 0) return Evaluate();
  //null move
  if(-ALphaBeta(-beta,-alpha,depth-1) >= beta)
     return beta;
  //список ходов
  PMove move = Generate();
  while(move)
  {
    MakeMove(move);  //сделали ход
    int tmp = -ALphaBeta(-beta,-alpha,depth-1);
    UnMakeMove(move); //восстановили позицию
    //если результат улучшает alpha -
    //увелим ее
    if(tmp > alpha) alpha = tmp;
    //если сравнялись с максимумом противника
    //дальше продолжать нет смысла
    if(alpha >= beta) return alpha; //или beta
    //следующий ход из списка
    move = move->next;
  }
  //вернем наш максимум
  return alpha;
}
```

Выигрыш по скорости мы здесь вряд ли получим, т. к. используем нулевой ход на глубину `depth-1`. Вводят некоторую величину, именуемую коэффициентом затухания R, и выполняют нулевой ход на сокращенную глубину. Так как пропуск хода в шахматах является очень серьезным преимуществом для противника, то эвристика все равно работает. Ускорение получается очень существенное. Если нулевой ход не сработал, то мы все равно перебирали на меньшую глубину, и задержка незначительная. Если сработал — целая ветвь подрезана! Вот пример кода:

```
#define R 2
int AlphaBeta(int alpha,int beta, int depth)
{
  //закончилась глубина?
  //вернем оценку узла
  if( depth <= 0) return Evaluate();
  //null move
  if(-ALphaBeta(-beta,-alpha,depth-1-R) >= beta)
     return beta;
  //список ходов
  PMove move = Generate();
  while(move)
  {
    MakeMove(move);  //сделали ход
    int tmp = -ALphaBeta(-beta,-alpha,depth-1);
    UnMakeMove(move); //восстановили позицию
    //если результат улучшает alpha,
    //увеличим
    if(tmp > alpha) alpha = tmp;
    //если сравнялись с максимумом противника
    //дальше продолжать нет смысла
    if(alpha >= beta) return alpha; //или beta
    //следующий ход из списка
    move = move->next;
  }
  //вернем наш максимум
  return alpha;
}
```

Нулевой ход можно еще усовершенствовать. Совершенно не обязательно перебирать с полным окном, чтобы увидеть, больше результат, чем beta, или нет. Можно использовать нулевое окно:

```
//null move
if(-ALphaBeta(-beta,-(beta-1),depth-1-R) >= beta)
   return beta;
```

Коэффициент затухания R, обычно равный двум, позволяет получить приемлемое качество счета. При счете на большую глубину (больше 6) можно использовать R, равный 3 или 4.

Нулевой ход может вносить погрешность и в строках, не связанных с цуцванг-позициями. Если обвалить напряженную строку нулевым ходом с уменьшенной глубиной просчета, это может вызвать ошибку.

Нулевой ход является методом выборочного поиска, и поэтому отсекаемый узел должен удовлетворять некоторым требованиям. В простейшем случае

нулевой ход можно использовать в любом узле, но для придания программе дополнительной тактической прочности необходимо соблюдать следующие условия:

- Ход в данную позицию не должен быть шахом. Он и так вернет отрицательный ответ, нет необходимости его вызывать.
- Этот ход не должен быть взятием. Это верно, по крайней мере, до некоторой глубины. Взятия можно и не учитывать. Это момент тонкой индивидуальной настройки.
- Данный узел не должен расширяться. В некоторых узлах можно не сокращать глубину поиска. Типичные выборочные продления: при шахе, при размене, при ходе пешки на предпоследнюю горизонталь. Система выборочных продлений может быть индивидуальна для каждой программы. Есть и другие критерии. Это целый принцип поиска. Если узел расширяется, то не имеет смысла обваливать его нулевым ходом.

Текущая статическая оценка узла должна быть, как минимум, равна значению beta. Это условие статического поиска. Узел, удовлетворяющий вышеперечисленным условиям, можно на большой глубине отсечь и без нулевого хода. Получится простейший вариант выборочного поиска. Нулевой ход выступает теперь в качестве дополнительного проверочного условия. Насколько текущая оценка должна быть больше beta? Существует понятие поля роста (`margin`). Его величина либо равна нулю, либо максимальному стратегическому приросту. Для нулевого хода это, как правило, 0. Величина, равная максимальному стратегическому приросту, повысит точность вычисления, но существенно замедлит скорость.

Нулевой ход можно использовать, лишь начиная с некоторой глубины, два или четыре полухода. Если программа расширяет строки с взятиями, то использование нулевого хода на глубине четырех дает очень хорошую тактическую прочность. Важно помнить, что нулевой ход — это не полный перебор, а эвристика, дающая сбои. Вот пример использования нулевого хода с условиями:

```
if( !InCheck                  &&
    moveNotCapture            &&
    !isExtensiens             &&
    score - margin >= beta    &&
    ply >= 4                  &&
    -ALphaBeta(-beta,-alpha,depth-1-R) >= beta)
    return beta;
```

Здесь соблюдается следующий порядок условий:

1) если не шах;

2) если не взятие;

3) если узел не расширяется;

4) если текущая оценка узла и поле роста больше beta;

5) если не первые 4 узла в строке;

6) если результат нулевого хода больше или равен beta.

Нужно отметить, что `margin = 0` незначительно увеличивает время счета, а тактическая прочность повышается существенно. Проверка обычно осуществляется из преузла. Код, реализующий эти действия, такой:

```
{...}
MakeMove(move);
score = MakeScore(move); //оценка после хода
check = InCheck(move);   //объявляет ли ход шах
if( !check  &&         //ход не объявляет шах
    moveNotCapture && //не взятие
    score <= alpha && //не улучшает alpha
    ...
 )
    {
      //do null move
    }
```

В преузле дополнительно может использоваться следующее условие:

4) если наш король не под шахом.

Для нулевого хода это условие не является обязательным, т. к. он и сам точен, и достаточно условий простого статического поиска. Если смотреть из преузла, то

```
score + margin <= alpha
```

означает то же самое, что

```
score - margin >= beta
```

из следующего узла. Почему предпочтительнее выполнять отсечку в преузле? Так эффективнее (не нужно напрасно вызывать функцию `AlphaBeta`), кроме того, есть возможность проверять состояние преузла. Если некоторые ходы были продлены, то желательно иметь точный ответ на них. Это уже больше относится к выборочному статическому поиску. Для продления только шахов статической оценкой это выглядит примерно так:

```
int AlphaBeta(...)
{
    {...}
    PMove move = GenerateAllMoves();
    while(move)
```

```
    {
       score = MakeScore;
       if(king_is_not_in_check  &&       //если наш король не под шахом
          move_does_not_opponent_king && //если ход не объявляет шах
          score <= alpha)                //не улучшаем alpha
          return alpha;

       MakeMove();
       //реальное вычисление
       {...}
       UnMakeMove();
    }
    return alpha;
 }
```

Для нулевого хода такое двойное продление не обязательно. Достаточно перечисленных ранее условий. Хочется отметить еще раз, что нулевой ход работает и без соответствия отсекаемого узла каким-либо условиям, но в напряженных позициях он постоянно будет ошибаться.

Нулевой ход можно использовать для определения угрозы. Это делается довольно редко. Если мы после хода попробовали ход на уменьшенную глубину (R очень большой), и результат говорит о том, что при пропуске хода мы выигрываем материал, то данный узел можно расширить. Чтобы таких расширений было немного, в качестве функции нулевого хода можно использовать только форсированный поиск материала с нулевым окном. Он выполнится довольно быстро. Если этот поиск показал выигрыш материала (только ходившей фигурой), глубину счета данного узла можно не уменьшать. В этом случае нулевой ход (форсированный вариант) работает как детектор атаки. Атака определяется только для ходившей фигуры, может ли эта фигура немедленно что-то выиграть в результате размена. Нулевой ход, таким образом, не отсекает ветви дерева, а наоборот, служит для расширения. Если `depth` не сокращается при атаке, это существенно увеличивает силу игры программы. Такие узлы обрабатываются довольно быстро, т. к. легальных ходов немного. Нужно понимать, что все это довольно условно.

С помощью нулевого хода и хеш-таблицы можно (при наличии оценочной функции) написать быстро считающую программу, даже при неэффективном генераторе перемещений. Таких программ в последнее время появилось великое множество, и они похожи друг на друга, как две капли воды. Приблизительно одна и та же оценка (таблицы средних значений для каждой фигуры), нулевой ход, выборочные расширения шахов, разменов и авансов пешек, форсированные окончания, рассматривающие только взятия до конца. Играют они, в среднем, довольно неплохо, но шахматы сложная игра. Возникнет на доске простейшая тактическая последовательность, не укладывающаяся в формальные критерии выборочных продлений, и такой

переборщик с полной глубиной в шесть полуходов не увидит ничего. Он сразу вылетит в форсированный вариант, где он рассматривает только взятия. Такую последовательность, правда, еще нужно создать или найти. Тем не менее, ставшее очень распространенным стандартное решение не учитывает все нюансы шахматной игры, и лучшие программы пытаются нащупать какие-то свои изюминки в расчете. Я знаю, что многие отказываются от нулевого хода для обвала дерева перебора или используют его очень выборочно. В целом, это очень хорошая эвристика для шахмат, но требующая вдумчивого отношения.

При использовании нулевого хода нужно учесть, что его можно применять один раз в строке. Иначе поиск выродится в процесс с максимально возможным коэффициентом затухания R. Некоторые программы применяют нулевой ход дважды. Я не нашел, что это эффективнее. Все это — тонкая индивидуальная настройка программы. Вот пример, где использование нулевого хода ограничено одним разом в строке:

```
#define R 2
int AlphaBeta(int alpha,int beta, int depth, bool doNullMove)
{
  //закончилась глубина?
  //вернем оценку узла
  if( depth <= 0) return Evaluate();
  //null move
  //если не делали нулевой ход раньше
  if( !doNullMove      &&
      -ALphaBeta(-beta,-alpha,depth-1-R,true) >= beta
   )
       return beta;
  //список ходов
  PMove move = Generate();
  while(move)
  {
    MakeMove(move);  //сделали ход
    int tmp = -ALphaBeta(-beta,-alpha,depth-1,doNullMove);
    UnMakeMove(move); //восстановили позицию
    //если результат улучшает alpha -
    //увелим ее
    if(tmp > alpha) alpha = tmp;
    //если сравнялись с максимумом противника
    //дальше продолжать нет смысла
    if(alpha >= beta) return alpha; //или beta
    //следующий ход из списка
    move = move->next;
  }
```

```
  //вернем наш максимум
  return alpha;
}
```

Эвристика убийцы

Порядок ходов при alpha-beta переборе имеет принципиальное значение. Хеш-таблица позволяет запоминать проделанную работу, и если позиция встретится вновь, мы имеем, как минимум, лучший ход для этой позиции, который можем попробовать сначала. Это все справедливо для очень большой хеш-таблицы и одинаковых позиций. Но часто ситуация бывает следующая: позиции отличаются в несущественных моментах, а лучший ход, вызывающий отсечку, совпадает. Возникла идея: попробовать сначала лучший ход для этого уровня, в надежде, что он будет лучшим и в данной позиции. Это часто срабатывает. Если у двух позиций один общий предок, то часто лучший ход в них один и тот же. Данная эвристика получила название эвристики убийцы (Killer heuristic). Она работает неточно, но может давать хорошую прибавку в скорости. Все дело в приоритете конкретной эвристики для оптимизации перемещений. Я, например, опытным путем пришел к следующему порядку:

1. Пробуем результат из хеша в надежде получить отсечку.
2. Пробуем нулевой ход с уменьшенной глубиной.
3. Пробуем лучший ход из хеша.
4. Взятия в отсортированном порядке. Сначала идут взятия, бьющие только ходившую фигуру противника, имеющие наилучшее соотношение цены атакованной фигуры и нападающего.
5. Прежде чем пробовать основную массу "тихих" перемещений, попробуем ход Killer heuristic.

Из приведенного списка видно, что Killer move не может быть взятием. В качестве описателя перемещения для Killer heuristic можно использовать обычный описатель хода, но тогда придется проверять допустимость данного хода в этой позиции. Можно хранить просто номер хода, что значительно упрощает программу, но возможно не во всех реализациях.

Иногда вместо одного хода хранят два или три. Это дает на практике очень мало, а программная реализация усложняется. В целом, хочется отметить, что при хорошей хеш-таблице ускорение от применения Killer heuristic не является принципиальным. Некоторые программисты даже отказываются от этой идеи. Дальнейшее развитие эта идея получила в эвристике истории (History heuristic).

Я всегда использовал Killer heuristic, т. к. она предельно проста и улучшает порядок ходов. Если программа не использует хеш, то ускорение от Killer

heuristic очень существенно — 50% и более, но это не актуально в наши дни, т. к. практически все программы используют таблицы перестановок.

Killer move можно настроить и более тонким образом. Тогда ускорение от него сравнимо с небольшой хеш-таблицей. Генератор взятий должен первым выдавать взятия последней ходившей фигуры противника. Тонкость состоит в том, что в Killer нельзя записывать взятия, бьющие последнюю ходившую фигуру противника, и в преузле Killer должен сбрасываться, т. к. имеет более низкий приоритет, чем взятия последней ходившей фигуры противника. Работает это очень хорошо. Два узла в дереве, имеющие общего предка, очень часто имеют один и тот же значимый ход. Единственное отличие — нужно попытаться сначала взять последнюю ходившую фигуру противника. Для организации Killer требуется всего 100 байт, если используются номера ходов в буфере, или 100 описателей перемещений. 100 — это просто разумное число использования глубины Killer. А эффективность очень неплохая. Настройка Killer увеличит и без того значительное ускорение от нулевого хода (если он применяется). Для Killer нет никакой разницы, в основном он поиске работает или в нулевом ходе. При поиске на большую глубину задержка от нулевого хода очень значительна (при хорошем порядке ходов и небольшом коэффициенте затухания). Это связано с тем, что перемещения в поиске нулевого хода плохо упорядочены, а программа пытается считать глубоко. Хочется отметить, что эта пара: взятие последней ходившей фигуры противника и Killer move — работает очень хорошо на любой глубине без хеша и нулевого хода. Можно не записывать в Killer взятия вообще и использовать его после взятий.

Эвристика истории

Эвристика истории (History heuristic) является статистическим методом упорядочивания перемещений. Позиция имеет размер 8 × 8 или 0..63. Можно сделать некоторую таблицу `[64][64]`, обнуленную в начале вычисления. Если данное перемещение вызывает улучшение alpha (или отсечку за beta), то в клетке `[N1][N2]` можно увеличить значение на 1, где `N1` — позиция, откуда пошла фигура, а `N2` — куда пошла.

Данную информацию можно использовать для упорядочивания перемещений. Если хеш-таблица дает гарантированное улучшение порядка ходов, то с этим не поспоришь, но статистика? Вот примерный список очередности при генерации перемещений:

- взятие короля (если был шах в преузле);
- взятие последней ходившей фигуры противника;
- определение атаки (последняя ходившая фигура противника напала на большую или незащищенную фигуру);
- если атака, берем перемещения атакованной фигуры;

- если не было взятия последней ходившей фигуры противника или атаки, пробуем Killer — лучший ход соседа на этой глубине (он в preузле сбрасывается);
- остальные взятия берутся по принципу "наиболее ценная жертва — наименее ценный нападающий";
- остальные перемещения добавляются в список и будут рассмотрены в самом конце, если имеет место потеря при взятии;
- Killer не записывается, если было взятие короля, последней ходившей фигуры или атака;
- если не было атаки, "тихие" перемещения просчитываются от лучших (сразу) до худших;
- потери материала в любом случае рассматриваются в конце;
- "тихие" перемещения можно выбирать по приращению позиционной оценки, но если расстояние между ферзем и королем оппонента опасно сокращается, такие перемещения рассматриваются в первую очередь.

Данный порядок ходов не является единственно правильным, но отражает реальное положение вещей. Killer — очень мощная эвристика, если не было взятия последней сходившей фигуры или тяжелой атаки — тогда лучший ход, безусловно, другой. Хочется спросить: куда здесь можно вставить эвристику истории с ее статистикой? Только для тихих перемещений. Но если тихие перемещения брать просто по приращению позиционной оценки, мы гарантированно ускорим процесс в несколько раз. Если атака оппонента, перемещения сортировать вообще не нужно. Не обязательно генерировать все перемещения сразу.

У нас есть списки фигур, и мы можем быстро сгенерировать нужное перемещение и сразу просчитать его. Эвристику убийцы и ход из хеша вообще не нужно получать, их только надо проверить на легальность. Ход из хеша идет самым первым. Тихие перемещения можно упорядочить по статистике, но упорядочивание по приросту оценки дает гарантированный результат.

Если у нас генератор перемещений написан как отдельная функция, и программа ожидает, пока он сгенерирует и отсортирует все перемещения (включая статистический признак), то можно и эвристику истории использовать. Если бы у нас не было взятий, то эвристика истории была бы значительно эффективнее. Возможны хитроумные реализации, но статистическая суть все равно остается.

Поиск стремления

Поиск стремления (Aspiration search) — еще одна идея ускорения alpha-beta перебора. Из корня дерева перебора функция `AlphaBeta` вызывается с пределами alpha = `-INFINITY`, beta = `INFINITY`, где `INFINITY` — максимально возможный результат.

Функция AlphaBeta возвращает оценку лучшего (по ее мнению) хода. Скорость счета сильно зависит от alpha-beta пределов. Возникла простая идея: зачем уточнять alpha-beta пределы, когда результат счета в большинстве случаев примерно известен. В качестве параметров функции ALphaBeta рассматривается ожидаемый результат и окно стремления. Выглядит это примерно так:

```
score = Evaluate();
window = ...;
alpha = score - window;
beta  = score + window;
score = AlphaBeta(alpha,beta,depth);
if(score <= alpha){
  alpha = -INFINITY;
  score = AlphaBeta(alpha,beta,depth);
}else if(score >= beta){
  beta  = INFINITY;
  score = AlphaBeta(alpha,beta,depth);
}
```

Если вышли за пределы окна стремления, пересчитаем с новыми пределами. Начальная ожидаемая оценка равна статической оценке корня дерева. Данный алгоритм можно совместить с итеративными углублениями.

```
score = Evaluate();
window = ...; //полпешки
depth = 1;
while(depth < MAX_DEPTH  &&  !TimeUp())
{
   alpha = score - window;
   beta  = score + window;
   score = AlphaBeta(alpha,beta,depth);
  if(score <= alpha){
    alpha = -INFINITY;
    score = AlphaBeta(alpha,beta,depth);
  }else if(score >= beta){
    beta  = INFINITY;
    score = AlphaBeta(alpha,beta,depth);
  }
  depth++;
}
```

Ожидается, что повторных пересчетов будет немного. Реальное ускорение от данного алгоритма достигается использованием стандартной процедуры alpha-beta. Если же у нас NegaScout и хеш-таблица, то практическое ускорение отсутствует. Хеш при окне стремления будет не таким полным, а

NegaScout выжимает из alpha-beta практически все. При NegaScout или Principal Variation нам нужно быстрее сосчитать первый ход (или главную строку изменения, оставшуюся от предыдущей итерации). Остальные перемещения будут просматриваться с нулевым окном. Дальнейший поиск будет идти значительно быстрее.

Некоторые люди используют Aspiration search в корне дерева дополнительно к NegaScout. Это дело вкуса. Хочется только отметить, что экспоненциальный рост дерева перебора с увеличением глубины поиска сохраняется в Aspiration search при любом окне стремления, и даже при нулевом окне.

MTD(f)

Много было попыток сделать невозможное — решить проблему перебора. Одна из таких попыток — MTD(f). Замечено, что перебор с нулевым окном выполняется быстрее:

```
int alpha = ...
int beta  = alpha+1;
score = AlphaBeta(alpha,beta);
if (score >= beta)  alpha = score;
else if(score <= alpha) beta = score;
```

Так как реальная оценка неизвестна, но лежит в интервале от -INFINITY до +INFINITY, то можно производить двоичный поиск с нулевым окном, пока alpha и beta не сойдутся. Это и есть основная идея MTD(f):

```
int MTDF(int depth)
{
  int min = -INFINITY;
  int max = INFINITY;
  int test = Evaluate();//статическая оценка позиции
  while (min < max)
  {
     int tmp = AlphaBeta(test,test+1,depth);
     if(tmp > test)
     {
       min  = tmp;
       test = min;
     } else {
       max  = tmp;
       test = max-1;
     }
  }
  return min;
}
```

Функция AlphaBeta должна вернуть, кроме оценки, и лучший ход. Он записывается, только если возвращаемый результат больше alpha. Можно предусмотреть проверку, которая выполняет повторный перебор, если результат последней итерации был меньше или равен test. Для MTD(f) используется alpha-beta с амортизацией отказов. Метод возвращает результат не всегда из alpha-beta диапазона, может вернуть отсечку:

```
int AlphaBeta(int alpha, int beta, int depth)
{
   int score = -INFINITY; //возвращаемое значение
   if(depth <= 0) return Evaluate();
   Generate();
   while(MoveList)
   {
      MakeMove();
      int tmp = -ALphaBeta(-beta,-alpha,depth-1);
      UnMakeMove();
      if( tmp > score)     score = tmp;
      if( score > alpha)   alpha = score;
      if( alpha >= beta)   break;
   }
   return score;
}
```

Данный алгоритм может совершить множество проходов, если есть большое разнообразие оценок. Например, каждый проход может увеличивать нижний предел только на 1. Чтобы избежать зависания функции, можно разбить промежуток между min и max пополам:

```
int SearchMT(int depth)
{
int min = -INFINITY;
int max = INFINITY;
test = (max+min)/2;
do{
int tmp = AlphaBeta(test,test+1,depth);
if(tmp > test)
{
min = tmp;
test = min;
} else {
max = tmp;
test = max-1;
}
```

```
if(max-min >= 2) test = (max+min)/2;
}while(min < max);

return min;
}
```

Для определения начального значения и сокращения числа проходов можно совместить функцию счета с итеративными углублениями:

```
int MTDF(int depth, int test)
{
int min = -INFINITY;
int max = INFINITY;
while (min < max)
{
int tmp = AlphaBeta(test,test+1,depth);
if(tmp > test)
{
min = tmp;
test = min;
} else {
max = tmp;
test = max-1;
}
}
return min;
}
void IterativeDeepening(void)
{
int test = Evaluate();
int depth = 1;
while( depth <= MAX_DEPTH && !TimeUp())
{
test = MTDF(depth,test);
depth++;
}
}
```

В MTD(f) можно задать точность, с какой мы извлекаем оценку. Чем ближе нулевое окно к действительному результату, тем дольше считает функция `AlphaBeta`. Если нам не существенна разница ± 1, то можно сужать промежуток поиска до этой величины. Если промежуток поиска сделать равным максимальной стратегической оценке, то алгоритм сосчитает только материал. На практике желательно считать точно, иначе теряется всякий смысл.

Алгоритм использует множество повторных переборов и нуждается в хорошей хеш-таблице для ускорения поиска. Здесь кроется еще одна проблема MTD(f). Она связана с нестабильностью поиска. Если поиск с нулевым окном вернул оценку больше или равную beta, то это не всегда наш минимум. Следующий поиск может вернуть оценку меньше минимума. Теоретически этого быть не должно, но из-за использования результатов из хеша и искусственных отсечений (Forward pruning) это вполне возможно.

Если диапазон оценок мал, и промежуток от `INFINITY` до `INFINITY` незначителен, то MTD(f) может показать хорошую скорость. Если же мы считаем в шахматах материал и стратегическую оценку, то эффективность MTD(f) хуже, чем NegaScout. Из-за нестабильности поиска и сложности взаимодействия с хешем практических реализаций MTD(f) очень мало. На практике порядок ходов значительно важнее и дает принципиальное ускорение. NegaScout выжимает из нулевого окна практически все возможное, и дальнейшее увеличение глубины перебора при хорошем порядке перемещений возможно лишь за счет увеличения мощности процессоров или отсечения некоторых ветвей дерева перебора.

Futility pruning

Автор идеи — Эрнст Хайнц, применивший ее в своей программе DarkThough. Сама идея очень проста. На последних двух полуходах перебора оценка неточна, и можно, основываясь на текущей статической оценке позиции и характеристике узла, подрезать целую ветвь поиска. Нельзя с определенностью сказать, что Futility pruning вносит погрешность, т. к. в двух последних узлах программа и так постоянно ошибается. Можно сказать, что ошибки будут, но несколько другого характера.

Если мы не делаем нулевого хода (или другого поиска сокращенной глубины), ход в текущую позицию не является шахом, взятием, данный узел не расширяется, и текущая оценка позиции больше beta на некоторую величину, называемую полем роста (`margin`), то мы можем просто вернуть beta и подрезать ветвь поиска. Все это относится к двум последним полуходам.

Как видно, всю это очень напоминает нулевой ход. Только на последних двух полуходах поиск недействительного перемещения не вызывается, если поле роста достаточно велико. Для нормальной работы хватит величины максимального стратегического приращения (полпешки).

Почему Futility pruning работает? Дело в том, что в позиции ожидается наш ход. Можно рассуждать так: если мы не сделали хода и имеем преимущество в материале (текущая оценка больше beta на величину материального приращения), то неужели мы не найдем такой ход, который, если и не сохранит нашего материального преимущества, то хотя бы приведет к его минимальной потере? Это маловероятно. Даже если бы мы выполнили полный

поиск и не подрезались, то, т. к. поиск идет в двух граничных узлах, нельзя сказать с уверенностью, что его результат будет точнее. Мы имеем материальное преимущество и рассчитываем, что дополнительный ход поможет нам сохранить оценку. Дополнительной проверкой служит ход в эту позицию: он не шах, не взятие и т. д. Текущая строка является потенциально слабой строкой. Выглядит алгоритм примерно так:

```
margin = wPawn/2; //поле роста
int AlphaBeta(PMove node, int depth, int alpha, int beta, ...)
{
     extend = extensiens();   //выборочные продления
     depth += extend;
     if( !nullMove           && //не нулевой ход
         depth <= 2          && //если 2 последних полухода
         !extend             && //узел не расширяется
         notCapture(node)    && //не взятие
         !InCheck(node)      && //не шах
         Evaluate() - margin >= beta //текущая оценка
       )
         return beta; //возвратим beta

       {...}
}
```

Если смотреть из преузла, то дополнительным условием является шах нашему королю:

```
int AlphaBeta(int depth, int alpha, int beta, ...)
{
    PMove move = Generate();
    while(moveList)
    {
       MakeMove(move);
       if(!nullMove              && //не нулевой ход
          depth-1 <= 2           && //2 последних узла
          notCapture(move)       && //ход не взятие
          notCheck(move)         && //ход не объявляет шах
          notExtensiens(move)    && //узел после хода не расширяется
          notCheckOwnKing        && //наш король не под шахом
          Evaluate() + margin <= alpha //оценка после хода и
                                       // поле роста не
                                       // улучшает alpha
        )
         {
          //пропустим вычисление
            tmp = alpha;
```

```
        }else{
            tmp = -AlphaBeta(...);
            {...}
        }
        UnMakeMove(move);
      move = move->next;
    }//while
    return alpha;
}
```

Если мы отсекаем из преузла, то дополнительным условием служит отсутствие шаха нашему королю. Это подразумевает, что мы хотим получить точный ответ на шах противника и не подрезаемся. Текущая оценка после хода и поле роста не должны улучшать alpha. Эффективность Futility pruning очень неплохая. Мы отсекаем множество статических поисков в форсированных вариантах без практического ухудшения качества счета. Если в конце перебора вызываются сложные (или плохо написанные) форсированные поиски, то ускорение может быть и большим. Можно не подрезаться, пока не выяснили все насчет мата для обеих сторон, но это дело вкуса.

Razoring

Идея та же, что и Futility pruning. Только там подрезались два граничных узла, а теперь мы возьмемся за два следующих. Поле роста (margin) теперь больше и равно максимальной величине материального приращения. Обычно вполне достаточно величины одной королевы. Условия подрезки дерева:

- мы в четырех конечных узлах поиска;
- ненулевой ход и другой поиск сокращенной глубины;
- текущая оценка больше beta на величину margin;
- ход в данную позицию не взятие;
- нет шаха нашему королю;
- данный узел не расширяется (речь идет о выборочных продлениях).

Если эти условия соблюдены, мы можем подрезать целую ветвь и возвратить beta. Есть еще один прием, называемый сокращением глубины поиска. Он работает обратно выборочным продлениям. Если при выборочных продлениях мы не сокращаем глубину поиска в определенных узлах, то теперь, наоборот, вместо того, чтобы подрезать целую ветвь статической оценкой, мы просто сокращаем глубину поиска в данном узле. Делается это в надежде на то, что если строка поиска станет вдруг интересной, она опять расширится, и оценка будет более точной. Вряд ли имеет смысл рассматривать жертву ферзя или аналогичных по ценности фигур на такой глубине. В по-

давляющем большинстве случаев это деградирующие строки поиска, и их можно отсечь. Риска не больше, чем при нулевом ходе. Эффективность Razoring несколько меньше, чем Futility pruning. Вот примерная схема алгоритма:

```
margin = wQueen; //поле роста
int AlphaBeta(PMove node, int depth, int alpha, int beta, ...)
{
     extend = extensiens();   //выборочные продления
     depth += extend;
     if( !nullMove           && //не нулевой ход
         depth <= 4          && //если 4 последних полухода
         !extend             && //узел не расширяется
         notCapture(node)    && //не взятие
         !InCheck(node)      && //не шах
         Evaluate() - margin >= beta //текущая оценка - поле роста
       )
         depth--; //сокращаем глубину поиска
         //  return beta; //возвратим beta

       {...}
         Generate();
       while(moveList)
       {
        {...}
       }
      return alpha;
}
```

Если смотреть из преузла, то к условиям проверки можно добавить шах нашему королю. Итак, если ход не объявляет шах, и нет шаха нашему королю:

```
int AlphaBeta(int depth, int alpha, int beta, ...)
{
    PMove move = Generate();
    while(moveList)
    {
       MakeMove(move);
       if(!nullMove              && //не нулевой ход
          depth-1 <= 4           && //4 последних узла
          notCapture(move)       && //ход не взятие
          notCheck(move)         && //ход не объявляет шах
          notExtensiens(move)    && //узел после хода не расширяется
          notCheckOwnKing        && //наш король не под шахом
```

```
          Evaluate() + margin <= alpha //оценка после хода +
                                       // поле роста не улучшает alpha
        )
         {
             //сократим глубину поиска на 1
             //на 1 она сокращается в любом случае

             nextDepth = depth-2;

         }else nextDepth = depth-1+extensiens(move);

         tmp = -ALphaBeta(nextDepth,-beta,-alpha,...);

         UnMakeMove(move);

         {...}
       move = move->next;
    }//while
    return alpha;
}
```

Приемы вроде Razoring требуют некоторой настройки в конечной программе, и результат сильно зависит от конкретной реализации. Если программа расширяет строки с взятиями, то она строит огромные деревья поиска, и их нужно подрезать такими или аналогичными приемами. В результате мы получаем результат, по качеству сопоставимый с полным перебором, хотя и не просматривали все дерево поиска.

Статический поиск

Статический поиск в том или ином виде используется во всех программах. Обычно он называется форсированным вариантом, хотя статический поиск может рассматривать не только форсированные ходы. В чем основная идея статического поиска? В процедуре поиска можно для каждого узла получить немедленную статическую оценку. Обычно функцией, вычисляющей эту оценку, является `Evaluate`. Что такое статическая оценка? Представьте, что фигуры замерли, и ходов больше нет, просто подсчитывается текущее состояние. Неважно, что белому королю на следующем ходу поставят мат. Движения нет, и этого функция `Evaluate` не увидит. Оценка для белых равна сумме всех весов и позиционных факторов белых фигур за вычетом всех весов и позиционных факторов черных. Позиционные факторы не определены однозначно. Есть средние значения для каждой фигуры, могут учитываться структуры пешек, безопасность короля, проходные пешки, близость легких фигур к центру, близость королевы к королю противника и т. д. Нам в данном случае не важен характер позиционной оценки. Важно, что она

берется для неподвижных фигур. Можно, для простоты, ориентироваться на следующие таблицы, действительные для всех фигур:

```
int black[64] =
      {
       1, 2, 3, 4, 5, 6, 7, 8,
       9,10,11,12,13,14,15,16,
      17,18,19,20,21,22,23,24,
      25,26,27,28,29,30,31,32,
      33,34,35,36,37,38,39,40,
      41,42,43,44,45,46,47,48,
      49,50,51,52,53,54,55,56,
      57,58,59,60,61,62,63,64
      };

 int white[64] =
      {
      64,63,62,61,60,59,58,57,
      56,55,54,53,52,51,50,49,
      48,47,46,45,44,43,42,41,
      40,39,38,37,36,35,34,33,
      32,31,30,29,28,27,26,25,
      24,23,22,21,20,19,18,17,
      16,15,14,13,12,11,10, 9,
       8, 7, 6, 5, 4, 3, 2, 1
      };
```

В нашем простейшем случае для всех белых фигур одна таблица, и позиционный фактор в ячейке таблицы, где стоит фигура. Функция выглядит следующим образом:

```
int ALphaBeta(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    if(depth <= 0) return Evaluate(player);
    PMove move = GenerateAllMoves(player);
    while(move)
    {
  MakeMove(move);
```

```
        int tmp = -AlphaBeta(-beta,-alpha,depth-1,opponent,player);
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
    }
    return alpha;
}
```

Первый вызов может быть таким:

```
#define INFINITY  (10*wQueen)  //десять ферзей
    ALphaBeta(-INFINITY,INFINITY,6,WHITE,BLACK);
```

Мы вызвали функцию `AlphaBeta` на глубину 6 с бесконечными пределами alpha и beta. Как видите, все достаточно просто. Как далеко сумеет просчитать такая функция? Число рассматриваемых позиций при полном переборе равно S в степени N, где S — среднее количество ходов в позиции, N — просчитываемая глубина. При alpha-beta отсечениях минимальное количество рассматриваемых позиций равно корню квадратному из этого числа.

Когда `depth = 0`, мы вызываем статическую оценочную функцию. Слишком глубоко полным перебором мы просчитать не сможем, и поэтому в конце вместо статической оценочной функции `Evaluate()` можно вызвать функцию статического поиска `Quies()`. Она устроена примерно так же, как и основная процедура поиска, только alpha в ней повышается статической оценкой:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    //статическая оценка узла
    int score = Evaluate(player);
    if(depth <= 0) return score;
    //повышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
        MakeMove(move);
        int tmp = -Quies(-beta,-alpha,depth-1,opponent,player);
```

```
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
    }
    return alpha;
}
```

В каждом узле сначала мы получаем статическую оценку данного узла. Ожидается, что, т. к. ход наш, мы сможем сохранить текущую оценку или улучшить ее. Это на практике не всегда так, но наш поиск упрощенный и должен отталкиваться от каких-то статических величин. Если мы повышаем alpha, то поиск идет значительно быстрее. Качественно быстрее. Мы даже можем выбросить некоторые ходы, дающие маленький прирост, и оставить, например, одни взятия:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    //статическая оценка узла
    int score = Evaluate(player);

    if(depth <= 0) return score;
    //повышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllCaptures(player);
    while(move)
    {
        MakeMove(move);
        int tmp = -Quies(-beta,-alpha,depth-1,opponent,player);
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
    }
    return alpha;
}
```

Можно получить выигрыш от взятия или просто вернуть текущую статическую оценку, или alpha. Статический поиск рассматривает только взятия, параметра глубины, как правило, нет, и взятия рассматриваются до конца.

Если смотреть из преузла, то функция не повышает alpha, а понижает beta, или свой потолок роста. Выглядит это так:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    if(depth <= 0) return Evaluate(player);
    //повышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  // beta
       int nextScore; //оценка узла после хода

       MakeMove(move);
       nextScore = Evaluate(player);//оценка после хода
       nextBeta  = beta;
       if(nextScore < beta) nextBeta = nextScore;
       if(nextBeta <= alpha)
       {
          tmp = alpha; //отсечка
       }else{
          int tmp = -Quies(-nextBeta,-alpha,depth-1,opponent,player);
       }

       UnMakeMove(move);
       if(tmp > alpha) alpha = tmp;
       if(alpha >= beta) break;
    }
    return alpha;
}
```

В этой функции мы делаем то же самое, что и раньше, только в преузле. Если оценка после хода меньше beta, мы понижаем beta. Как видите, пока мы не ввели ничего нового, только делаем статическую корректировку в предыдущем узле. На этом примере видно, что при статическом поиске мы для каждого хода ограничиваем прирост его захватываемой частью. Например, взяли пешку, максимум ограничен, и если дальше в строке будет выиг-

рыш больше пешки, программа этого не увидит. Максимум прироста ограничен приростом хода в узле. То же самое относится и к позиционной оценке. Если новый максимум (nextBeta) меньше или равен alpha, то нет смысла считать далее, можно сразу принять оценку хода как alpha. Заметьте, что отрицательный ход автоматически вычеркнется, если значение alpha до перемещения уже повышено текущей оценкой.

Мы ограничиваем максимум всех ходов. Как продлить шахи статической оценкой? Очень просто. Если шах, максимум не ограничиваем:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    if(depth <= 0) return Evaluate(player);
    //повышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  //новое значение beta
       int nextScore; //оценка узла после хода
       MakeMove(move);
       nextScore = Evaluate(player);//оценка после хода
       nextBeta  = beta;
       if( !inCheck(move)  &&
           nextScore < beta
        )
          nextBeta = nextScore;
       if(nextBeta <= alpha)
       {
          tmp = alpha; //отсечка
       }else{
          int tmp = -Quies(-nextBeta,-alpha,depth-1,opponent,player);
       }
       UnMakeMove(move);
       if(tmp > alpha) alpha = tmp;
       if(alpha >= beta) break;
    }
    return alpha;
}
```

Функция inCheck — детектор шахов. Если ход объявляет шах королю противника, мы не ограничиваем его максимум. Такой статический поиск, хотя и считает приближенно, но увидит уже гораздо больше. Заменим функцию inCheck функцией Extensiens, которая возвращает 1, или true, когда ход продлевается. Мы можем просмотреть глубже не только шахи, а взятия или атаки. Чем больше ходов мы продлеваем, тем точнее результат, но тем медленнее идет счет. Введем еще переменную, аргумент функции Quies, которая показывает, продлевался ли ход в преузле. Назовем ее ext.

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent,//цвет фигур противника
              bool ext      //продлевался ли ход в преузле
             )
{
    if(depth <= 0) return Evaluate(player);
    //повышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  //новая beta
       int nextScore; //оценка узла после хода
       int nextExt;   //продлевается ли ход

       MakeMove(move);
       nextScore = Evaluate(player);//оценка после хода
       nextBeta  = beta;
       nextExt   = Extensiens(move);
       if( !nextExt     &&
           nextScore < beta
        )
          nextBeta = nextScore;

       if(nextBeta <= alpha)
       {
          tmp = alpha; //отсечка
       }else{
          int tmp = -Quies(-nextBeta,-alpha,depth-1,
                           opponent,player,nextExt);
       }
```

```
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
    }
    return alpha;
}
```

Теперь мы в каждом узле знаем, продлевался ли ход в предыдущем. Что это нам дает? Если ход продлевался в предыдущем узле, и максимум этого хода не был ограничен, мы можем рассмотреть точный ответ на него. Например, если противник нам объявил шах, мы можем иметь более точный ответ на шах, а не ограниченный максимумом. Выглядит это примерно так:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent,//цвет фигур противника
              bool ext      //продлевался ли ход в преузле
             )
{
    if(depth <= 0) return Evaluate(player);
    //завышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  //новая beta
       int nextScore; //оценка узла после хода
       int nextExt;   //продлевается ли ход

       MakeMove(move);
       nextScore = Evaluate(player);//оценка после хода
       nextBeta  = beta;
       nextExt   = Extensiens(move);
       if( !Ext          &&
           !nextExt      &&
           nextScore < beta
         )
          nextBeta = nextScore;

       if(nextBeta <= alpha)
       {
          tmp = alpha; //отсечка
```

```
      }else{
         int tmp = -Quies(-nextBeta,-alpha,depth-1,
                          opponent,player,nextExt);
      }

      UnMakeMove(move);
      if(tmp > alpha) alpha = tmp;
      if(alpha >= beta) break;
   }
   return alpha;
}
```

Если нам не было шаха, и мы не объявляем шах, то ограничиваем максимум для хода. Таким образом, строки с шахами рассматриваются дальше полностью.

Такой поиск идет все-таки довольно медленно, т. к. мы рассматриваем все перемещения. Взятия мы исключить не можем, они дают значительный прирост оценки и должны рассматриваться далее, хоть и с ограниченным максимумом. Для строк с шахами можно написать такую функцию:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent,//цвет фигур противника
              bool ext,     //продлевался ли ход в преузле
              int evalu     //текущая оценка узла
             )
{
    if(depth <= 0) return evalu;
    //завышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  //новая beta
       int nextScore; //оценка узла после хода
       int nextExt;   //продлевается ли ход

       MakeMove(move);
       nextScore = evalue +  //текущая оценка
                   delEvaluate(player);//приращение при ходе
```

```
            nextBeta  = beta;
            nextExt   = Extensiens(move);

if( !Ext      &&     //нет шаха нашему королю
    !nextExt  &&     //ход не объявляет шах противнику
    !moveNotCapture(move)  //ход не взятие
 )
{
   //нет хода
   tmp := evalue; //оценка узла до хода
}else{

            if(
                !nextExt     &&
                nextScore < beta
             )
               nextBeta = nextScore;

            if(nextBeta <= alpha)
            {
               tmp = alpha; //отсечка
            }else{
               int tmp = -Quies(-nextBeta,-alpha,depth-1,
                            opponent,player,nextExt.-nextScore);
            }

            UnMakeMove(move);
}
            if(tmp > alpha) alpha = tmp;
            if(alpha >= beta) break;
         }
         return alpha;
      }
```

Получился форсированный вариант, который просматривает глубже шахи. Это довольно стандартное решение. Нужно только ограничить глубину просмотра шахов некоторой реальной величиной, а дальше рассматривать только взятия для совсем приблизительной оценки позиции горизонта. Введена новая переменная, содержащая текущую оценку узла. Не нужно вызывать функцию `Evaluate` каждый раз, если приращение оценки можно вычислять пошагово. У ответов на шахи можно тоже ограничить максимум, нельзя их вычеркивать. Есть еще один вариант статического поиска. В нашем случае мы ограничивали beta в преузле или повышали alpha в узле. Можно не делать этого, а подрезаться, только если текущая оценка больше beta.

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    //статическая оценка узла
    int score = Evaluate(player);

    if(depth <= 0) return score;
    //если вышли за beta
     if(score >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       MakeMove(move);
       int tmp = -Quies(-beta,-alpha,depth-1,opponent,player);
       UnMakeMove(move);
       if(tmp > alpha) alpha = tmp;
       if(alpha >= beta) break;
    }
    return alpha;
}
```

Если смотреть из преузла, то это выглядит так:

```
int Quies(
              int alpha,    //наш максимум
              int beta,     //противника
              int depth,    //глубина
              int player,   //цвет наших фигур
              int opponent //цвет фигур противника
             )
{
    if(depth <= 0) return Evaluate(player);
    //завышаем alpha статической оценкой
    if(score > alpha) alpha = score;
    if(alpha >= beta) return beta; //или alpha
    PMove move = GenerateAllMoves(player);
    while(move)
    {
       int nextBeta;  //новая beta
       int nextScore; //оценка узла после хода
```

```
        MakeMove(move);
        nextScore = Evaluate(player);//оценка после хода
        if(nextScore <= alpha)
        {
           tmp = alpha; //отсечка
        }else{
           int tmp = -Quies(-beta,-alpha,depth-1,opponent,player);
        }

        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
    }
    return alpha;
}
```

Это уже имитация полного перебора. Пока текущая оценка в alpha-beta коридоре, отсечки нет. Можно оставлять некоторое поле роста (margin). Можно отсечку оговорить дополнительными условиями.

Устаревший метод выборочного поиска

В процессе счета возникает множество вариантов, бесполезность которых видна человеку с первого взгляда. Как научить машину отсекать бесполезные варианты и оставлять нужные? Это не так просто. Всегда найдется исключение из правил. Тем не менее на большой глубине поиска мы сталкиваемся с дилеммой: либо не считаем ничего, либо считаем выборочно. Применительно к шахматам можно сказать, что существует ряд ходов, когда можно определенно предположить, что поиск нужно продолжить. Это, первым делом, шах. Программа будет терять в качестве, а эффект горизонта оказывать сильное влияние. Во-вторых, взятия. Размены нужно рассматривать до появления спокойной позиции (это в форсированных вариантах делается повышением alpha текущей оценкой). Взятие является ходом, за которым может последовать качественное изменение позиции. Теория игр говорит (это ясно, в принципе, и без теории), что ходы, дающие наибольшее приращение оценки в узле, нужно рассматривать глубже. Кроме шахов и взятий, существует целый ряд так называемых угроз. Простейшая угроза — это атака. Чтобы взять фигуру, ее нужно атаковать. Угрозой может быть и ход пешки на предпоследнюю горизонталь, т. к. следующим ходом пешка может превратиться в ферзя, и счет нельзя прекращать в этой точке. Но чаще всего угроза — это опасность выигрыша фигурой материала. Вес атакующей фигуры должен быть меньше атакуемой. Как правило, это атака на зависающую фигуру. Все это приблизительно и формально, потому что ходившая фигура может не атаковать сама, а освободить линию атаки для

дальнобойной фигуры, возможна и другая комбинация. Но на практике тактические последовательности, продолжающиеся на много (иногда более десяти) полуходов, состоят именно из простейших угроз немедленного выигрыша материала и взятий (шахи относятся к угрозам).

С появлением нулевого хода выборочный поиск стал легче. Нулевой ход автоматически распознает все угрозы немедленного выигрыша материала и потенциальные угрозы. У него есть только один недостаток — дерево перебора все равно очень большое, а самые напряженные строки желательно рассматривать далее. Нулевой ход, кроме всего прочего, несет опасность нераспознанного цуцванга, но для шахмат этим в начале игры можно пренебречь. Итак, что мы имеем? Есть целый ряд ходов, которые можно выделить по формальным признакам и просчет которых желательно продолжить для получения более реалистичной оценки. Делается это примерно так:

```
int ALphaBeta(int alpha, int beta, int ply ...)
{
     mainScore = Evaluate(); //оценка узла (статическая)
     isCheck   = inCheck(); //под шахом ли наш король
     Generate();
     while(moveList)
     {
        MakeMove();
        nextScore = Evaluate();//оценка после хода
        nextCheck = inCheck(); //ход объявляет шах?
        //взятие?
        moveIsCapture = nextScore - mainScore >= wPawn;
        if( !isCheck   &&  //наш король не под шахом
            !nextCheck &&  //ход не объявляет шах
            !moveIsCapture && //ход не взятие
            !pawnMovesRank_7_or_2 &&  //не аванс пешки
            nextScore + margin + threat <= alpha
         )
          {
             tmp = alpha; //подрезали
          }else{
           //вычисление
           tmp = -ALphaBeta(-beta,-alpha,ply+1,...);
          }
        UnMakeMove();
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
     }
     return alpha;
}
```

Если не было шаха нашему королю, и ход не шах, не взятие, не ход пешки на предпоследнюю горизонталь, и

```
nextScore + margin + threat <= alpha,
```

где `margin` — поле роста оценки, `threat` — поле угрозы, то подрезаем целую ветвь.

Поле роста обычно изменяется от 0 до максимального стратегического прироста. Можно поле роста делать различным, в зависимости от глубины. Если мы в двух последних узлах, и поле роста равно полпешки, то можно подрезать без поля угрозы (Futility pruning). Если мы в четырех последних узлах, и поле роста — величина королевы, то тоже можно подрезать без поля угрозы или сократить глубину поиска (Razoring). Как определить поле угрозы?

Нулевой ход определит поле угрозы автоматически. Плохо то, что нулевой ход оставит все равно большое дерево перебора. Можно определять поле угрозы и по-другому. До появления нулевого хода так и делали. Можно для определения поля угрозы использовать некоторый аналог нулевого хода, который ищет только материал, и выигрыш материала должна принести последняя ходившая фигура. Следует уточнять главную строку изменения при таком выборочном поиске. Поиск угрозы будет подобен простому форсированному варианту, который рассматривает только взятия:

```
int ThreatDetect(int alpha,int beta, int square,
                 int player,int opponent)
{
   score = Evaluate(player);
   if(score > alpha) alpha = score;
   if(alpha >= beta) return alpha;
   GenerateAllCapture(player);//список взятий
   while(moveList)
   {
       if( square != 0  &&
           move->square <> square) continue;
       MakeMove(move);
       score = -ThreatDetect(-beta,-alpha,0,
                  opponent,player);
       UnMakeMove(move);
       if (score > alpha) alpha = score;
       if (alpha >= beta) break;
   }
   return alpha;
}
```

Данный вариант работает как нулевой ход, но при первом вызове рассматривает ходы только одной фигуры. Вариант функции с таким определением угрозы будет выглядеть подобно следующему:

```
  int ALphaBeta(int alpha, int beta, int ply,
                int player,    //цвет наших фигур
                int opponent)  //противника
{
     mainScore = Evaluate(player); //оценка узла (статическая)
     isCheck   = inCheck(player); //под шахом ли наш король
     Generate(player); //список перемещений
     while(moveList)
     {
        MakeMove(move);
        nextScore = Evaluate(player);//оценка после хода
        nextCheck = inCheck(opponent); //ход объявляет шах?
        //взятие?
        moveIsCapture = nextScore - mainScore >= wPawn;
        if( !isCheck   &&  //наш король не под шахом
            !nextCheck &&  //ход не объявляет шах
            !moveIsCapture && //ход не взятие
            !pawnMovesRank_7_or_2 &&  //не аванс пешки
            nextScore + margin <= alpha &&
            //последняя ходившая фигура не может
            //немедленно выиграть материал
            ThreatDetect(alpha,alpha+1, move->square,
                 player,opponent) <= alpha
         )
          {
             tmp = alpha; //подрезали
          }else{
           //вычисление
           tmp = -ALphaBeta(-beta,-alpha,ply+1,
                            opponent,player);
          }
        UnMakeMove(move);
        if(tmp > alpha) alpha = tmp;
        if(alpha >= beta) break;
     }
     return alpha;
}
```

С возрастанием глубины поле угрозы может увеличиваться, и под конец остаются только шахи. Можно передавать текущую оценку. Таких продлений немного, и все они представляют несомненный интерес.

Сокращенное вычисление

Я использовал этот прием сам, но не относился к нему серьезно. Каково же было мое удивление, когда я увидел, что он используется в программах, занимающих первые строчки в рейтинге. Существует эвристика нулевого хода, позволяющая, основываясь на сокращенном поиске, отсечь неперспективные ветви дерева. Мы пропускаем ход противника после нашего хода, делаем еще один сокращенный поиск, и если оценка все равно не улучшает alpha, то узел можно подрезать. Ожидается, раз мы сделали два хода подряд и не улучшили alpha, то если противник сделает ход, мы его подавно не улучшим. LazyEval работает по-другому. Мы ищем не на полную глубину, а на сокращенную.

- Текущая оценка после хода должна быть не больше alpha, мы не выполняем сокращенный поиск для всех узлов. Это зависит от alpha-предела, который постоянно уточняется.
- Используют поле роста оценки — `margin`.

```
{...}
MakeMove();
if(score + margin <= alpha)
   tryLazyEval;
```

- Данный ход не должен быть взятием, шахом, и вообще, не должен расширяться.

```
{...}
MakeMove();
score = Evaluate();
if(score+margin <= alpha  &&
   moveNotCapture  &&
   notCheck  &&
   ...
 )
  tryLazyEval;
```

Метод очень хорошо взаимодействует с хеш-таблицей. Мы постоянно перебираем на глубину `depth-2` и упорядочиваем хеш. Данная методика называется Internal Iterative Deepening (Внутренние итеративные углубления). Только мы пользуемся результатом сокращенного счета и характеристикой узла, чтобы подрезать дерево.

Если наша программа уточняет главную строку изменения, то неточность LazyEval вообще сведена к минимуму.

Когда LazyEval работает особенно хорошо? Когда у нас за полным перебором идет выборочный статический поиск. Тогда сокращение глубины на два

и характеристика узла дают почти полную гарантию верной отсечки. Ведь мы даже не сокращаем глубину полного перебора. В LazyEval, как и в NullMove, нельзя подрезать. Если мы уже выполняем поиск сокращенной глубины, то нельзя вызывать еще один сокращенный поиск, чтобы подрезать, иначе поиск деградирует.

```
#define L 2
int ALphaBeta(int alpha,int beta, int depth, bool lazyEval)
{
       if(depth <= 0) return Quies(alpha,beta);
       Generate();
       while(moveList)
       {
           MakeMove();
           extend = extansiens(); //приращение глубины
           nextDepth = depth-1;
           margin = wPawn/2; //полпешки

           if(!lazyEval       && //не поиск сокращенной глубины
              Evaluate() + margin <= alpha   &&
              notCheck()      && //не шах
              moveNotCapture  && //не взятие
              extend == 0     && //узел не расширяется
              -ALphaBeta(-beta,-alpha,depth-1-L,true) <= alpha
            )
             {
                tmp = alpha;
             }else{
                tmp = -ALphaBeta(-beta,-alpha,nextDepth+extend,LazyEval);
             }
             UnMakeMove();
             if(tmp > alpha) alpha = tmp;
             if(alpha >= beta) break;

       }//while

  return alpha;
}
```

Все это требует вдумчивого отношения. На последних полуходах полный перебор превращается в некий статический поиск. Если коэффициент затухания LazyEval равен 1, то на последних полуходах будет очень похоже на Futility pruning. Это значит, что если оценка после хода с учетом поля роста не улучшает alpha, подрезаемся без проверок.

Погрешность программы, в которой сокращается глубина не полного широтного поиска, а выборочного, сведена к минимуму. При некоторой настройке, чтобы подрезать узел, достаточно взять из хеша оценку старой итерации. Даже если `margin = 0`, поиск на глубину N не всегда вырождается в поиск на глубину N−2 или N−1.

Извлечение строки главного изменения

Главная строка изменения — это строка перемещений, которую программа посчитала лучшей. Оценка, возвращаемая функцией `AlphaBeta`, является результатом этой строки. Данная строка как бы является продолжением хода, который программа делает. Насколько информативна эта строка? Это трудно сказать однозначно. Для напряженных строк игры бывает, что вся строка предсказана правильно. Для спокойных позиций бывает, что правильно предсказан только первый ход, т. к. программа его непосредственно сделала. Извлечение главной строки представляет интерес еще и с точки зрения скорости выполнения программы. Если мы после каждой итерации имеем главную строку изменения, нам легче просчитать следующую. До появления хеш-таблиц это был основной (вместе с эвристикой убийцы) способ перебрать более глубоко.

Как же извлекать главную строку изменения? Если в узле была отсечка за beta, то это уже узел не главной строки. Главная строка — это строка, которая начинается в корне дерева перебора, и все узлы имеют точную оценку больше alpha, но меньше бета. Если бы мы перебирали алгоритмом alpha-beta без искусственных отсечений, то получили бы ту же главную строку, что и при полном переборе.

Главная строка называется Principal Variation, или PV. Извлекается PV следующим программным кодом:

```
CONST
  MAX_DEPTH = 20;

TYPE
  TMove = record
     N1,N2:byte; {откуда куда}
  end;
  TLine = record
     line:array[0..MAX_DEPTH] of TMove; {сама строка}
     count:integer; {количество элементов}
  end;
  procedure SaveLine(var sourceLine,destLine:TLine; var bestMove:TMove);
  var
   n:integer;
```

```
begin
  for n := 0 to sourceLine.count-1 do
    destLine.line[n+1] := sourceLine[n];
  destLine.line[0] := bestMove;
  destLine.count := sourceLine.count+1;
end;
procedure ResetLine(var line:TLine);
begin
    line.count := 0;
end;

function ALphaBeta(alpha,beta,depth:integer;
                   var bestLine:TLine; {возвращаемая строка}
                   player,             {цвет наших фигур}
                   opponent:integer;   {противника}
                  ):integer;
var
  move:TMove;
  bufMoves:array[0..MAX_MOVES] of TMove;
  tmp:integer;
  tmpLine:TLine; {временная строка}
begin
     if depth <= 0 then begin
         {приблизительная оценка горизонта}
         ALphaBeta := Quies(...);
         exit;
     end;
     {получим все перемещения}
     Generate(player,bufMoves);
     {покажите весь список
     while NextMove(move) do
     begin
         ResetLine(tmpLine);
         MakeMove(move);
         tmp := -ALphaBeta(-beta,-alpha,depth-1,
                           tmpLine,opponent,player);
         UnMakeMove(move);
         if tmp > alpha then begin
            alpha := tmp;
            if alpha < beta then
              SaveLine(tmpLine,bestLine,move);
            else
              break;
         end;
```

```
      end;
      ALphaBeta := alpha;
  end;
```

Мы записываем лучший ход и всю строку за ним только в случае, если результат больше alpha, но меньше beta. Записываемый ход ставится на первое место, его строка далее. Функция, записывающая PV, выполняется сравнительно редко, она не принципиальна по времени исполнения, так что цикл ничего не значит.

Если у нас есть хорошая хеш-таблица, то не обязательно принудительно извлекать PV. Узлы PV запишутся в таблицу с флагом HASH_EXACT. Можно в стеке функции счета передавать некоторую логическую переменную PV. При первом вызове она всегда true. Далее, если в этом узле PV == true, и данный узел есть в хеше с флагом HASH_EXACT, то PV для следующего вызова true. Во всех остальных случаях — false. Таким образом, мы отслеживаем все строки PV (предположительно), начинающиеся в корне дерева перебора. Это своего рода дерево PV. Там есть и главная строка изменения от предыдущей итерации. Такое дерево не будет отслеживаться абсолютно точно, т. к. узлы в хеше могут перезаписываться другими узлами, но абсолютная точность здесь и не требуется. Можно предпринять некоторые меры для сохранения узлов в хеше или завести для них отдельную таблицу. Таких узлов будет очень немного.

Использование строки главного изменения

В этом разделе мы попытаемся пощупать то, что еще не изобретено, а именно искусственный интеллект. Тем не менее им можно пользоваться и извлекать немалую выгоду. Если в шахматах в качестве оценочной функции пытались использовать нейронные сети, генетические алгоритмы и просто вероятностные методы, то чем это хуже? Некоторые из этих приемов использовались, чтобы построить шахматные программы на совсем уж хилых процессорах. Непонятно, как они играли, но играли же. Но перейдем от вступления к делу.

Мы в alpha-beta поиске извлекли строку главного изменения. Как можно ею воспользоваться? Можно использовать ее для следующей итерации в упорядочивании перемещений. Тут следует вспомнить алгоритм Principal Variation. Он ход из строки главного изменения просматривает с полным alpha-beta окном, а остальные (они, вероятно, хуже) пытается отсечь сначала нулевым окном. Если оценка больше alpha, то ход пересчитывается с полным окном. Таких пересчетов немного, и ускорение достигает двух и более раз (зависит от конкретной реализации).

```
CONST
  MAX_PLY = 20;
```

```
TYPE
  TMove = record
     N1,N2:byte; {откуда куда}
  end;
  TLine = record
     line:array[0..MAX_PLY] of TMove; {сама строка}
     count:integer; {количество элементов}
  end;
  procedure SaveLine(var sourceLine,destLine:TLine; var bestMove:TMove);
  var
   n:integer;
  begin
    for n := 0 to sourceLine.count-1 do
      destLine.line[n+1] := sourceLine[n];
    destLine.line[0] := bestMove;
    destLine.count := sourceLine.count+1;
  end;
  procedure ResetLine(var line:TLine);
  begin
      line.count := 0;
  end;

  {строка главного изменения}
  var mainLine:TLine;


  function PrincipalVariation(alpha,beta,depth:integer;
                      var bestLine:TLine; {возвращаемая строка}
                      player,             {цвет наших фигур}
                      opponent:integer;   {противника}
                      ply:integer         {глубина от 0}
                     ):integer;
  label
   done;
  var
    move:TMove;
    bufMoves:array[0..MAX_MOVES] of TMove;
    tmp:integer;
    tmpLine:TLine; {временная строка}
  begin
       if (depth <= 0) or (ply >= MAX_PLY) then begin
           {приблизительная оценка горизонта}
           PrincipalVariation := Quies(...);
           exit;
       end;
```

```
    {попытаемся получить ход PV из
    глобальной строки}
    if FindPV(move) then begin
         MakeMove(move);
         tmp := -PrincipalVariation(-beta,-alpha,depth-1,
                          tmpLine,opponent,player,ply+1);
         UnMakeMove(move);
        if tmp > alpha then begin
           alpha := tmp;
           if alpha < beta then
             SaveLine(tmpLine,bestLine,move);
           else
             goto done;
        end;
    end;

    {получим все перемещения}
    Generate(player,bufMoves);
    while NextMove(move) do
    begin
        ResetLine(tmpLine);
        MakeMove(move);
        tmp := -PrincipalVariation(-(alpha+1),-alpha,depth-1,
                          tmpLine,opponent,player,ply+1);
        if (tmp > alpha) and (tmp < beta) then
          tmp := -PrincipalVariation(-beta,-alpha,depth-1,
                          tmpLine,opponent,player,ply+1);

        UnMakeMove(move);
        if tmp > alpha then begin
           alpha := tmp;
           if alpha < beta then
             SaveLine(tmpLine,bestLine,move);
           else
             break;
        end;
    end;
done:
    PrincipalVariation := alpha;
end;
{старт }
function IterativeDeepening(var bestMove:TMove):integer;
const
  MAX_DEPTH = 6;
```

```
var
  tmpLine:TLine;
  depth:integer;
  score:integer;
begin
    ResetLine(tmpLine);
    ResetLine(mainLine);
    depth := 1;
    while (depth <= MAX_DEPTH) and not TimeUp do
    begin
      IterativeDeepening := score;
      mainLine := tmpLine;
      bestMove := mainLine.line[0]; {лучший на это время ход}
      score := PrincipalVariation(-INFINITY,INFINITY,
                                  depth,
                                  tmpLine,
                                  player,
                                  opponent,
                                  0
                                 );
      depth := depth+1;
    end;
end;
```

Переменная `ply` показывает абсолютную глубину от 0. `Depth` может не сокращаться во время просчета, например при шахе, а `ply` всегда строго увеличивается на 1. Нашли ход из главной строки изменения — переберем его сразу с полным окном, остальные попытаемся отсечь, т. к. они, вероятно, не улучшают alpha. Для полной корректности нужно не перебирать во второй раз ход PV, если он встретится в основной массе перемещений, но в данном случае это не важно. При каждой итерации извлекается главная строка изменения, а при последующей она просматривается первой. Так как у нас полный перебор, то это просто улучшает порядок ходов и увеличивает скорость вычисления. Не очень много для начала, тем более что хорошая хеш-таблица разгонит программу значительно лучше. Но пойдем дальше. У нас функция была стандартная `AlphaBeta`, не отсекающая ничего и просматривающая все ветви. Давайте перепишем эту функцию как форсированный вариант, или точнее, некий вариант статического поиска. Полного перебора пока вообще нет. Если оценка позиции после хода не улучшает alpha, и ход не шах, не взятие и не ход пешки на предпоследнюю горизонталь, то функция просто подрезает дерево, и оценка хода принимается равной alpha.

```
function Search(alpha,beta,depth:integer;
                var bestLine:TLine; {возвращаемая строка}
```

```
                    player,              {цвет наших фигур}
                    opponent:integer;    {противника}
                    ply:integer          {глубина от 0}
                   ):integer;
label
 done;
var
  move:TMove;
  bufMoves:array[0..MAX_MOVES] of TMove;
  tmp:integer;
  tmpLine:TLine; {временная строка}
begin
     if (depth <= 0) or (ply >= MAX_PLY) then begin
         {приблизительная оценка горизонта}
         Search := Quies(...);
         exit;
     end;
     {попытаемся получить ход PV из
     глобальной строки}
     if FindPV(move) then begin
         MakeMove(move);
         tmp := -Search(-beta,-alpha,depth-1,
                        tmpLine,opponent,player,ply+1);
         UnMakeMove(move);
         if tmp > alpha then begin
            alpha := tmp;
            if alpha < beta then
              SaveLine(tmpLine,bestLine,move);
            else
              goto done;
         end;
     end;

     {получим все перемещения}
     Generate(player,bufMoves);
     while NextMove(move) do
     begin
         ResetLine(tmpLine);
         MakeMove(move);

if (Evaluate(player) <= alpha)  and {оценка после хода не лучше alpha}
    moveNotCapture(move)  and       {ход не взятие}
    moveNotCheck(move)    and       {ход не шах}
    moveNotPawnRank_6_or_7(move) then   {не аванс пешки}
```

```
  begin
    {подрезаем дерево}
    tmp := alpha;
  else begin
         tmp := -Search(-(alpha+1),-alpha,depth-1,
                        tmpLine,opponent,player,ply+1);
         if (tmp > alpha) and (tmp < beta) then
           tmp := -Search(-beta,-alpha,depth-1,
                        tmpLine,opponent,player,ply+1);
 end;
         UnMakeMove(move);
         if tmp > alpha then begin
            alpha := tmp;
            if alpha < beta then
              SaveLine(tmpLine,bestLine,move);
            else
              break;
         end;
     end;
done:
     Search := alpha;
end;
```

Данный вариант не является образцом статического поиска. Обратите внимание, что мы грубо подрезаем дерево, основываясь на характеристике хода и пределе alpha. Если бы мы не извлекали главную строку, то такой вариант не видел бы практически ничего. С главной строкой он несколько умнее. Почему? Это трудно объяснить. Видимо, дело в том, что мы уточняем alpha-beta пределы, и отсечка становится выборочной. Данный статический поиск обладает удивительной особенностью. Если он первыми просчитывает лучшие ходы, он играет сильнее. Намного сильнее. Он может не увидеть простейшей комбинации, но иногда поражает своей игрой. Для упорядочивания ходов мы просто извлекали главную строку изменения. Этого, конечно, не достаточно. В идеале все дерево перебора должно храниться в памяти, и лучшие ходы при каждой итерации передвигаются вверх. В нашем случае в конце перебора вызывается функция `Quies`, которая возвращает приблизительную оценку позиции горизонта. Она должна, как минимум, рассматривать взятия до конца, а желательно, и сглаживать эффект горизонта от шахов.

Теперь внесем небольшое изменение. Так как все дерево игры мы в памяти хранить не можем, то будем хранить все главные строки, которая программа собиралась использовать в игре. Так мы приходим к мысли о необходимости создать таблицу PV или дерево. Это ограниченное дерево игры, которое программа проанализировала. Оно уже не такое большое, несколько десят-

ков элементов. В нашем случае мы будем просто после каждой итерации извлеченную главную строку изменения вставлять в это дерево PV:

```
type
  PPVNode = ^PVNode;
  {узел дерева PV}
  PVNode = record
     move:TMove;     {ход}
     next:PPVNode;   {следующий ход в этой позиции}
     list:PPVNode;   {начало списка ходов оппонента}
  end;
  PVTree = .... {конкретную реализацию опускаем}
  var PVRoot:PPVNode; {корень дерева PV}
{вставляет строку в дерево}
procedure InsertLineInPVTree(var line:TLine);
{возвращает true если ход в  дереве PV}
function MoveInPV(var move:TMove;  node:PPVNode):boolean;
```

Наша функция будет содержать указатель на элемент ограниченного дерева игры.

```
   function Search(   node:PPVNode;       {узел PV-дерева}
                      alpha,beta,depth:integer;
                      var bestLine:TLine; {возвращаемая строка}
                      player,             {цвет наших фигур}
                      opponent:integer;   {противника}
                      ply:integer         {глубина от 0}
                     ):integer;
 label
  done;
 var
   move:TMove;
   bufMoves:array[0..MAX_MOVES] of TMove;
   tmp:integer;
   tmpLine:TLine; {временная строка}
 begin
      if (depth <= 0) or (ply >= MAX_PLY) then begin
          {приблизительная оценка горизонта}
          Search := Quies(...);
          exit;
      end;
      {
       перебираем ходы дерева PV
       и просматриваем их с полным окном
      }
```

```
      while node <> NIL do
      begin
          move := node.move;
           MakeMove(move);
           tmp := -Search(-beta,-alpha,depth-1,
                          tmpLine,opponent,player,ply+1,
                          node^.list);
           UnMakeMove(move);
          if tmp > alpha then begin
             alpha := tmp;
             if alpha < beta then
               SaveLine(tmpLine,bestLine,move);
             else
               goto done;
          end;
          node := node^.next;
      end;

        Generate(player,bufMoves);
             while NextMove(move) do
      begin
         {если ход уже был рассмотрен как часть
          дерева PV, пропустим}
          if moveInPV(move,node) then continue;
          ResetLine(tmpLine);
          MakeMove(move);

if (Evaluate(player) <= alpha)  and {оценка после хода не лучше alpha}
    moveNotCapture(move)  and       {ход не взятие}
    moveNotCheck(move)    and       {ход не шах}
    moveNotPawnRank_6_or_7(move) then   {не аванс пешки}
   begin
     {подрезаем дерево}
     tmp := alpha;
   else begin
          tmp := -Search(-(alpha+1),-alpha,depth-1,
                         tmpLine,opponent,player,ply+1,NIL);
          if (tmp > alpha) and (tmp < beta) then
            tmp := -Search(-beta,-alpha,depth-1,
                         tmpLine,opponent,player,ply+1,NIL);
  end;
          UnMakeMove(move);
          if tmp > alpha then begin
             alpha := tmp;
```

```
            if alpha < beta then
              SaveLine(tmpLine,bestLine,move);
            else
              break;
          end;
      end;
done:
      Search := alpha;
end;


{старт }
function IterativeDeepening(var bestMove:TMove):integer;
const
  MAX_DEPTH = 6;
var
  depth:integer;
  score:integer;
begin
      ResetLine(mainLine);
      PVRoot := NIL;
      depth := 1;
      while (depth <= MAX_DEPTH) and not TimeUp do
      begin
        IterativeDeepening := score;
        bestMove := mainLine.line[0]; {лучший на это время ход}
        InsertLineInPVTree(mainLine);
        score :=  Search(PVRoot,
                         -INFINITY,INFINITY,
                         depth,
                         mainLine,
                         player,
                         opponent,
                         0
                        );
        depth := depth+1;
      end;
end;
```

После каждой итерации мы главное изменение заносим в дерево PV. Это делается, чтобы программа помнила продуманные варианты и избегала старых ошибок. Если этого не делать, то программа может постоянно возвращаться к ошибочным вариантам. В функции счета мы сначала перебираем все ходы PV в узле, если они есть. Обратите внимание, что ходы из PV-

дерева просматриваются с полным окном. Затем смотрим основную массу перемещений. Они подрезаются статической оценкой. Так как alpha-beta пределы постоянно корректируются лучшими ходами, то этот выборочный поиск не так глуп, как кажется. Просто мы создаем дерево обдуманных ходов с полным (!) окном и постоянно его уточняем.

Мы ходы из PV считаем лучшими и рассматриваем их с полным окном, не подрезая статической оценкой. Основной принцип статического поиска гласит, что если мы ход считали хорошим и не подрезали статической оценкой, то и ответ на него нужно рассматривать, не подрезая его, т. е. иметь точный ответ на некоторую угрозу. В нашем случае, можно для простоты ввести некоторую переменную, которая устанавливается в `false`, если ответ в следующем узле нужно рассмотреть с полным окном. Выглядит это примерно так:

```
  function Search(   node:PPVNode;      {узел PV-дерева}
                     alpha,beta,depth:integer;
                     var bestLine:TLine; {возвращаемая строка}
                     player,             {цвет наших фигур}
                     opponent:integer;   {противника}
                     ply:integer;        {глубина от 0}
                     isCut:boolean       {подрезать ли ход}
                    ):integer;
label
 done;
var
  move:TMove;
  bufMoves:array[0..MAX_MOVES] of TMove;
  tmp:integer;
  tmpLine:TLine; {временная строка}
begin
     if (depth <= 0) or (ply >= MAX_PLY) then begin
         {приблизительная оценка горизонта}
         Search := Quies(...);
         exit;
     end;
     {
      перебираем ходы дерева PV
      и просматриваем их с полным окном
     }
     while node <> NIL do
     begin
         move := node.move;
          MakeMove(move);
```

```
          tmp := -Search(-beta,-alpha,depth-1,
                         tmpLine,opponent,player,ply+1,
                         node^.list,  false);
          UnMakeMove(move);
          if tmp > alpha then begin
             alpha := tmp;
             if alpha < beta then
               SaveLine(tmpLine,bestLine,move);
             else
               goto done;
          end;
          node := node^.next;
      end;
      {получим все перемещения}
      Generate(player,bufMoves);
      while NextMove(move) do
      begin
         {если ход уже был рассмотрен как часть
          дерева PV - пропустим}
          if moveInPV(move,node) then continue;

          ResetLine(tmpLine);
          MakeMove(move);
if  IsCut  and                          {если не ответ на PV}
    (Evaluate(player) <= alpha)  and {оценка после хода не лучше alpha}
    moveNotCapture(move)   and          {ход не взятие}
    moveNotCheck(move)     and          {ход не шах}
    moveNotPawnRank_6_or_7(move) then    {не аванс пешки}
   begin
     {подрезаем дерево}
     tmp := alpha;
   else begin
          tmp := -Search(-(alpha+1),-alpha,depth-1,
                      tmpLine,opponent,player,ply+1,NIL,true);
          if (tmp > alpha) and (tmp < beta) then
            tmp := -Search(-beta,-alpha,depth-1,
                           tmpLine,opponent,player,ply+1,NIL,true);
  end;
          UnMakeMove(move);
          if tmp > alpha then begin
             alpha := tmp;
             if alpha < beta then
               SaveLine(tmpLine,bestLine,move);
```

```
            else
              break;
          end;
      end;
done:
      Search := alpha;
end;
```

При первом вызове переменная `isCut` равна `false`, т. к. первый вызов всегда от корня дерева PV. Данная программа будет уже довольно неплохо играть даже без полного перебора. Ходы из таблицы PV можно использовать для гибкой системы выборочных продлений, не основанной на формальных признаках. Здесь надо учесть один фактор. Когда позиция определилась, существует не так много строк игры, которые нужно уточнять (две или три).

Извлеченную строку PV можно добавлять в дерево не обязательно в корне, а и в каждом узле, если он доступен (это практически не улучшает игру). Элементов для построения дерева тогда понадобится не несколько десятков, а несколько сотен. После выдачи хода можно сбрасывать дерево и думать над ответом. Если противник выбрал другой ход, ну и шут с ним, все равно не наше время было. Если угадали — возможен практически мгновенный ответ. Это производит впечатление, тогда как у полного переборщика никогда не будет физической возможности считать с полным окном на такую громадную глубину.

Относительно генерации перемещений. Совершенно не обязательно сканировать все перемещения, а потом листать их. Если мы используем списки фигур, можно очень просто получить взятия последней ходившей фигуры противника, причем в отсортированном порядке. Для этого список наших фигур перебираем, начиная с пешек и заканчивая королем. Для каждой фигуры выясняется принципиальная возможность взять последнюю ходившую фигуру противника. Эту функцию можно назвать `Attack`. Существует массив, где в каждой ячейке записано, может ли фигура отсюда бить конкретную клетку. Если может, надо построить линию. Для пешек, королей и коней линия не нужна. После взятия последней ходившей фигуры можно попробовать эвристику убийцы, чтобы отсечь узел, даже не генерируя перемещений для него. Тихие ходы можно делать по принципу: нашел хорошее перемещение — сразу в следующую позицию. В технике BitBoard можно иметь маски перемещений для всех фигур. Тогда выяснить потенциальную возможность взять конкретную фигуру проще простого.

Теория выборочного поиска

Данная тема является довольно скользкой, и я ни в коем случае не претендую на роль классика в этом жанре. Понятия нечетки, и трудно провести грань между белым и черным. Это напоминает интегральное исчисление:

сумма бесконечно малых величин дает что-то осязаемое. Так и с выборочным поиском. Рассуждая логически, нельзя резать ветви дерева перебора без их предварительного исследования. Данная задача интригует программистов и математиков уже давно, и многие утверждают, что только полный перебор дает правильный результат. Применительно к шахматам, наилучший порядок — вещь труднодостижимая. Но это в сторону.

Эффективность метода alpha-beta научно доказана, когда программа перебирает до самого конца. Мы должны создавать иллюзию счета до конца. Цель игры — поставить мат королю, а прежде, чем объявляется мат, нужно сделать шах. Шахи должны просматриваться максимально глубоко. Для достижения этой цели после основного поиска вызывают упрощенный, который рассматривает только взятия, а в хорошей программе должен рассматривать и шахи, и ответы на них. Это целое искусство — просмотреть шахи достаточно глубоко с минимальными накладными расходами, чтобы не замедлять основной перебор. Например, программа Genius Ричарда Лэнга, выигравшая у Каспарова блиц из двух партий, просматривала тактические выстрелы из шахов на десятки полуходов. Существует подход, при котором, если в строке игры встретился шах, глубина перебора не сокращается. Но если тактическая строка начинается на большой глубине, то программа ничего не увидит. Дело в данном случае не в конкретной реализации. Никакими позиционными коэффициентами не объяснишь программе, что такое безопасность короля, если она дальше носа ничего не видит. Шахи и взятия являются самыми сильными перемещениями. Можно привести еще ряд перемещений, которые следует просматривать более глубоко. Это, в первую очередь, атаки. Подмечено, что в некоторых позициях количество легальных перемещений минимально. Такие позиции нужно просматривать глубже, т. к. это прямая тактика.

Статические понятия

В каждой позиции существует две статические оценки: до хода (`L_SCORE`) и после хода (`R_SCORE`). Оценка до хода является общей для всей позиции, оценка после хода изменяется после каждого перемещения. Рассмотрим эти величины подробнее. `R_SCORE`, как правило, больше `L_SCORE`. Это относится к взятиям, как к перемещениям, дающим наибольший прирост оценки, и к другим ходам, оценка которых больше 0. Существуют еще отрицательные ходы.

Реальная оценка после счета должна быть, как минимум, `L_SCORE`. Это — основное статическое понятие. Игрок после хода противника, как правило, может найти ход, не ухудшающий текущую позицию. Прирост оценки ограничен диапазоном от `L_SCORE` до `R_SCORE`. Если мы взяли пешку, то, согласно статическим критериям, это наш максимум. Дальнейший поиск только выясняет: действительно ли мы можем ее взять, или противник ее отыграет.

Итак, мы имеем:

- L_SCORE — статический минимум
- R_SCORE — статический максимум
- alpha — реальный минимум
- beta — реальный максимум

```
if (L_SCORE >= beta) return beta;
```

Если в узле alpha меньше L_SCORE, то можно сразу повысить alpha до величины L_SCORE.

```
procedure StaticALphaBeta(alpha,beta:integer):integer;
var
  L_SCORE,R_SCORE:integer;
begin
   L_SCORE := Evaluate; {статическая оценка позиции}
   if L_SCORE > alpha then alpha := L_SCORE;
   Generate; {список перемещений}
   while moveList and
         (alpha < beta) do
   begin
       MakeMove;
       tmp := -StaticALphaBeta(-beta,-alpha);
       UnMakeMove;
       if tmp > alpha then alpha := tmp;
   end;
   StaticAlphaBeta := alpha;
end;
```

У нас получился простейший статический поиск. Переменную R_SCORE мы пока не используем. Если мы будем брать не все перемещения, а только взятия, то получим простейший форсированный вариант.

В приведенном примере мы всегда повышали alpha. Можно делать иначе. Поиск прекращается, когда L_SCORE больше beta. Это уже имитация полного перебора. Мы просматриваем все, пока наш статический минимум меньше нашего реального максимума:

```
procedure StaticALphaBeta(alpha,beta:integer):integer;
var
  L_SCORE,R_SCORE:integer;
begin
   L_SCORE := Evaluate; {статическая оценка позиции}
   if L_SCORE >= beta then begin
       StaticAlphaBeta := beta;
```

```
        exit;
    end;
    Generate; {список перемещений}
    while moveList and
          (alpha < beta) do
    begin
        MakeMove;
        tmp := -StaticALphaBeta(-beta,-alpha);
        UnMakeMove;
        if tmp > alpha then alpha := tmp;
    end;
    StaticAlphaBeta := alpha;
end;
```

Если смотреть из преузла, то условие `L_SCORE >= beta` означает, что `R_SCORE <= alpha`. Если статический максимум нашего перемещения не повышает alpha, то оценка хода равна alpha:

```
procedure StaticALphaBeta(alpha,beta:integer):integer;
var
  L_SCORE,R_SCORE:integer;
begin
    Generate; {список перемещений}
    while moveList and
          (alpha < beta) do
    begin
        MakeMove;
        R_SCORE = Evaluate;
        if R_SCORE <= alpha then
          tmp := alpha
        else
          tmp := -StaticALphaBeta(-beta,-alpha);
        UnMakeMove;
        if tmp > alpha then alpha := tmp;
    end;
    StaticAlphaBeta := alpha;
end;
```

Этот пример идентичен предыдущему, только отсечение выполняется в преузле.

Варианты подрезания:

- `L_SCORE >= beta`. Мы режем целый узел и не получаем перемещения.
- `R_SCORE <= alpha` после хода, подрезаем только один ход.
- `alpha >= beta`, натуральная отсечка по результату счета.

Рассмотрим ситуацию:

```
(L_SCORE <= alpha) and (beta <= R_SCORE)
```

Реальные минимум и максимум находятся внутри статического промежутка. Это говорит о том, что мы знаем реальный диапазон статического изменения оценки и только уточняем его. Если промежуток от L_SCORE до R_SCORE устраивает нас как погрешность вычисления, то можно прекратить счет. Варианты возвращаемых значений:

- alpha;
- beta;
- среднее арифметическое, т. к. alpha и beta стремятся друг к другу;
- среднее значение статического приращения;
- случайное число или вероятностную оценку.

Может быть некоторая аномальная ситуация, когда:

```
(L_SCORE >= beta) and (R_SCORE <= alpha)
```

Мы должны бы отсечь данный узел за beta, но данное перемещение не улучшает alpha. Это возможно только для отрицательных ходов вблизи alpha-beta диапазона, когда этот диапазон сократился до выяснения стратегической составляющей. Что возвращать в данном случае? Отсекать ход, когда оценка R_SCORE не улучшает alpha, несколько примитивно. Мы можем отсечь интересные перемещения.

Вместо фиксированного поля роста можно ввести динамическое. Принимаем alpha как оценку хода. В предыдущем узле для противника были свои L_SCORE и R_SCORE. Назовем их PRE_L_SCORE и PRE_R_SCORE. Для нашего узла это инвертированные величины. Мы должны повышать alpha, по меньшей мере, до PRE_L_SCORE. Это значит, если противник взял фигуру, мы не должны подрезать, пока не компенсируем материал. Введем константу:

```
MAX_ST_VALUE = wPawn div 2; {максимальное стратегическое приращение}
```

Поскольку мы не хотим нарушать стратегию, условие подрезания будет следующее:

```
if (PRE_L_SCORE-MAX_ST_VALUE <= alpha) and
   (R_SCORE <= alpha) then ...
```

Итак, мы компенсировали потерю материала, и оценка после хода не улучшает alpha, тогда подрезаем. Это работает довольно прилично, только нужно уточнять главное изменение.

Вообще мы должны дотягивать alpha и до L_SCORE. По статическим понятиям, оценка не должна быть хуже статической оценки узла, и мы должны компенсировать потерю:

```
if(alpha > PRE_L_SCORE) and
  (alpha > L_SCORE)     and
  (alpha > R_SCORE) then ...
```

Данное условие представляет собой гибкое поле роста от 0 до максимального приращения (ферзя). На практике работает очень неплохо. Величины L_SCORE и другие не нужно вычислять всякий раз, а можно добавлять изменение оценки при ходе:

```
function StaticSearch(alpha,beta,L_SCORE,PRE_L_SCORE:integer):integer;
var
 R_SCORE:integer;
begin
    Generate;
    while moveList and
          (alpha < beta) do
    begin
       R_SCORE := DelScore(L_SCORE,move);{пошаговое приращение}
       if (alpha > PRE_L_SCORE) and
          (alpha > L_SCORE) and
          (alpha > R_SCORE) and
          not Check  {не шах}
          then
          tmp := alpha
       else begin
          MakeMove;
          tmp := -StaticSearch(-beta,-alpha,-R_SCORE,-L_SCORE);
          UnMakeMove;
       end;
       if tmp > alpha then alpha := tmp;
    end;
   StaticSearch := alpha;
end;
```

При итеративных углублениях строка главного изменения помещается в таблицу PV. Первым делом мы рассматриваем ходы из PV-таблицы. Для хода из PV или ответа на него узел не подрезается. Таким образом, мы уточняем главное изменение. Поле роста можно ограничить материальной составляющей и не дотягивать alpha до L_SCORE. Зачем нужно уточнять PV? В реальной игре не так много жестких строк с ограниченным количеством легальных перемещений. Они автоматически попадают в PV. Их нужно просто уточнять. Таким образом, в жестких позициях программа может считать невероятно глубоко. В отличие от нулевого хода, это просчет самых жестких строк игры с полным окном, а не подрезка, основанная на догадке (плюс-минус кирпич).

Нужно отметить, что данное поле роста позволяет считать очень точно. Я тестировал программу без полного перебора и уточнения PV (только строка главного изменения просматривалась первой) с полным переборщиком, считающим очень глубоко и имеющим очень хороший тактический уровень. Этот чистый форсированный вариант выдержал все атаки противника и даже под конец получил стратегическое преимущество. В реальной программе выборочный поиск должен вставляться после сокращенного поиска полной ширины (4—5 полуходов), и строка главного изменения должна уточняться.

Если выполнено условие, мы сразу принимаем оценку хода как alpha. Можно не обрезать дерево, а сокращать глубину перебора. Это намного точнее, но медленнее. Выглядит это примерно так:

```
function Search(alpha,beta,L_SCORE,PRE_L_SCORE,
                depth,ply:integer):integer;
var
 R_SCORE,nextDepth:integer;
begin
    if (Depth <= 0) or (ply >= MAX_PLY) then
    begin
        Search := Quies(alpha,beta); {форсированный поиск}
        exit;
    end;
    Generate;
    while moveList and
          (alpha < beta) do
    begin
       R_SCORE := DelScore(L_SCORE,move);{пошаговое приращение}

       if Check then        {если ход объявляет шах}
              nextDepth := Depth;
       else  if notPV      and     {ход не PV}
                (alpha >= PRE_L_SCORE-MAX_ST_VALUE) and
                (alpha >= R_SCORE)
             then
               nextDepth := Depth-2
       else   nextDepth := Depth-1;

       MakeMove;
       tmp := -Search(-beta,-alpha,-R_SCORE,-L_SCORE,
                    nextDepth,ply+1);
       UnMakeMove;
```

```
      if tmp > alpha then alpha := tmp;
    end;
   Search := alpha;
end;
```

Строки PV должны уточняться, и сокращение глубины в них недопустимо. Так как мы первой просматриваем только одну строку PV, и alpha-beta пределы максимально отдалены от корня дерева, она автоматически будет рассматриваться с полной глубиной. Чтобы данный алгоритм считал точно, искусственные alpha-beta пределы не годятся.

Что лучше: подрезка статической оценкой или снижение глубины просмотра? Первое быстрее при приемлемой точности, погрешность сведена к минимуму. Если мы подрезаем статической оценкой, то структура выборочных продлений может основываться на реальных строках, которые программа собирается играть, а не на формальных критериях. Даже шах не является обязательным ходом, и совершенно нет необходимости плодить огромные деревья из шахов, если программа не собирается играть эти перемещения. В форсированном поиске шахи должны очень глубоко просматриваться статической оценкой, чтобы программа могла выбрать в PV разумные перемещения. Программа просматривает наиболее напряженные строки на 12—14 полуходов с полным окном без всяких форвардных отсечений. Причем, строки могут состоять из простых перемещений. Полный переборщик расширяет строки, исходя из формальных критериев: шах, взятие, ход пешки на предпоследнюю горизонталь и т. д. Он не может физически сосчитать простые перемещения на такую глубину без подрезки и оказывается совершенно беспомощным при игре с такой программой. Человеку с ней играть, наоборот, легче. Она не застрахована от сильных позиционных ходов противника, т. к. в перспективе видит угрозу.

Надо отдать дань справедливости, расширение главной строки изменения является тоже довольно условным. Насколько осмысленна эта строка? Она тем ближе к истине, чем меньше легальных ходов в данной позиции. Это напоминает мышление человека. Выбираем наиболее перспективный вариант и анализируем его на большую глубину.

Просмотр шахов в форсированном поиске

Тема эта чрезвычайно важна. Для того чтобы alpha-beta процедура находила лучший ход, мы должны считать до конца или создавать некоторую иллюзию с помощью форсированных вариантов. Относительно того, какие ходы нужно рассматривать в форсированном поиске, не существует единого мнения. Чем больше ходов мы просматриваем, тем точнее результат, но тем медленнее поиск, в ущерб полному перебору.

Почему так важно просматривать шахи в статическом (форсированном) поиске? Дело в том, что цель игры — поставить мат, а прежде, чем поставить мат, нужно объявить шах. Кроме того, при шахе игра как бы продолжается на один полуход, и от противника требуется немедленная реакция. По моему мнению, без просмотра шахов в статическом поиске невозможно сделать хорошую программу. Выборочных продлений явно недостаточно. Результат статического поиска приближенный, но для большой глубины и комбинаций, основанных на шахах, он очень точен. Для примера можно привести программу Genius. Она просматривала тактические выпады, основанные на шахах, на десятки полуходов с минимальным количеством исследованных позиций. Совместить выполнение требований к просмотру шахов в форсированных вариантах позволяет старый алгоритм, который использовался еще Ричардом Лэнгом:

1. Начальная глубина просмотра шахов равна двум.
2. Если нам объявлен шах, и существует только одно легальное перемещение, то глубина просмотра увеличивается на два.
3. Если два легальных перемещения, то глубина просмотра увеличивается на единицу.
4. Если больше двух легальных перемещений, то глубина просмотра остается неизменной, в результате смягчается эффект горизонта от шахов.

Алгоритм прост, но как его реализовать на практике? При некотором мастерстве можно оптимизировать просмотр шахов. Можно иметь некоторый код, который считает количество легальных перемещений под шахом. Первым делом мы рассматриваем взятия последней ходившей фигуры противника. Остальные взятия (кроме взятия короля противника) пропущены, т. к. наш король под шахом. В каждом узле есть описатель хода противника в эту позицию.

Можно использовать счетчик легальных ходов. Вначале считаем, что у нас всего один легальный ход, глубина просмотра шахов при первом ходе всегда увеличивается на два. Если мы в результате счета получили ход, при котором короля не съели на следующем, счетчик увеличивается, и глубина просмотра шахов будет увеличиваться не на два, а на единицу для всех последующих ходов. При величине счетчика два (и далее) глубина просмотра не увеличивается, т. к. легальных ходов много. В пределах глубины просмотра мы обрабатываем шахи, ответы на них и взятия. Что делать с остальными перемещениями? Их можно вычеркнуть. Оценка хода принимается равной alpha. Если вычеркнуть простые перемещения, то скорость почти не будет отличаться от простого форсированного варианта (кроме позиций с множеством шахов).

Данная схема не является догмой, а всего лишь одним из возможных решений:

```
Const
 MAX_CHECK_PLY = 30;
function CheckInQS(alpha,beta,MaxCheckPly,ply,score:integer;
                   check:boolean;
                   player,opponent:integer;
                  ):integer;
var
  legalMoves,NextCheckPly:integer;
begin
    if (ply >= MaxCheckPly) or
       (ply >= MAX_CHECK_PLY) then
    begin
        CheckInQS := Quies(alpha,beta);
        exit;
    end;

    if not check then begin
       if score > alpha then alpha := score;
       if alpha >= beta then begin
           CheckInQS := beta;
           exit;
       end;
    end;

    Generate(player);
    legalMoves := 0;
    while moveList and (alpha < beta) do
    begin
        if moveCaptureKing then
        begin
           CheckInQS := INFINITY-ply;
           exit;
        end;
        nextCheck := CheckDetect(move,opponent);
        if not check  and
           not nextCheck and
           moveNotCapture then
        begin
           tmp := alpha;
        end else begin
           MakeMove(move);
           NextCheckPly := MaxCheckPly;
           if check then begin
```

```
            if legalMoves = 0 then
               NextCheckPly := MaxCheckPly+2
            else if legalMoves = 1 then
               NextCheckPly := MaxCheckPly+1;
         end;

         tmp := - CheckInQS(-beta,-alpha,NextCheckPly,
                  ply+1,-MakeNextScore(score,move),
                  nextCheck,
                  opponent,player
                 );
         UnMakeMove(move);
      end;
      if tmp <> -(INFINITY-(ply+1)) then
        legalMoves :- legalMoves + 1;
      if tmp > alpha then alpha := tmp;
   end;
  CheckInQS := alpha;
end;
```

Генератор перемещений должен первым возвращать взятия короля. Глубину просмотра шахов в реальной программе нужно все равно ограничивать. Вот примерный вызов функции просмотра шахов из основной функции поиска:

```
function AlphaBeta(alpha,beta,ply,depth .....):integer;
begin
     if depth <= 0 then begin
 AlphaBeta :=  CheckInQS(alpha,beta,
                 ply + 2, {!!!!}
                 ply,score,
                 check,
                 player,opponent,
               );
       exit;
     end;

    {...}
end;
```

Приведенный пример только демонстрирует принцип. При вызове функции `CheckInQS` из основного поиска `MaxCheckPly = ply+2`, где `ply` — абсолютная глубина поиска от 0, `Depth` — изменяемая глубина поиска. При первом вызове функции `AlphaBeta` она равна стартовой глубине и уменьшается на единицу в каждом узле (если нет выборочных продлений). Таким образом, мы динамически увеличиваем глубину только в позициях с малым количе-

ством легальных ходов. Если шахов мало, то основной поиск почти не тормозит. Максимальный прирост простого перемещения ограничен приращением при ходе.

Принцип отбора легальных ходов несколько примитивен, но позволяет уточнять комбинации, построенные на шахах.

Сокращения глубины поиска

Зачем еще и сокращения, когда есть выборочные продления? Не одно ли это и то же? Выборочные сокращения позволяют довольно гармонично настроить поиск, а продления все равно остаются, и все мирно уживаются друг с другом.

Основная идея сокращения глубины аналогична подрезанию статической оценкой, только вместо немедленного прекращения перебора мы сокращаем глубину поиска. Сокращения глубины происходят точно в тех же узлах, что и подрезка при статическом поиске.

Например, самая типичная схема статического поиска:

```
int Search(int alpha, int beta, int depth, ...)
{
   //если превышена максимальная глубина, вернем
   //приблизительную оценку позиции горизонта
      if (depth <= 0) return Quies(...);
   //получим список перемещений
   Generate();
   while(moveList   &&   alpha < beta)
   {
       r_score = ... //оценка после хода
       if (move_does_check  ||      //если шах
           move_is_capture  ||      //если взятие
           r_score > alpha)
        {
           MakeMove();
           tmp = -Search(-beta,-alpha,depth-1,...);
           UnMakeMove();
           if(tmp > alpha) alpha = tmp;

        }//if
        nextMove();
   }//while

   return alpha;
}//end function
```

Если статическая оценка позиции после хода (R_SCORE) не улучшает alpha, то ход подрезан. R_SCORE — это наш статический максимум. Ожидается, что противник сумеет, как минимум, сохранить текущую оценку (или улучшить). В случае выборочного сокращения глубины мы делаем то же самое, только не подрезаем, а сокращаем глубину, скажем, на два:

```
int Search(int alpha, int beta, int depth, ...)
{
   if(depth <= 0) return Quies();
   Generate();
   while(moveList && alpha < beta)
   {
      //сокращение глубины счета
      extend = -1;
      if(move_does_check)  extend = 0; //если шах
        else if( !move_is_capture   &&  //не взятие
                 alpha >= r_score) extend = -2;
      MakeMove();
      tmp = -Search(-beta,-alpha,depth+extend,...);
      UnMakeMove();
      if(tmp > alpha) alpha = tmp;

   }//while

   return alpha;
}//function
```

Стандартное сокращение глубины при взятии или другом сильном перемещении равно единице, при шахе глубину не сокращаем. Во всех остальных случаях сбрасываем глубину на два, ожидая, что ветвь подрежется и не улучшит alpha. Тут работает еще один тонкий момент. Сокращая глубину, мы уменьшаем оценку. Для граничных узлов это эквивалентно статическому подрезанию. Для остальных узлов используем алгоритм, устанавливающий искусственные alpha-beta пределы. Следует продумать, как он будет сочетаться с конкретной схемой выборочных уменьшений глубины. В частности, добавить условие баланса мата: пока не выяснили все насчет мата, не подрезаем:

```
if( alpha > -INFINITY+100    &&    beta  < INFINITY-100)
{
 ...
}
```

Можно ввести поле роста (margin) и эмулировать полный перебор:

```
if(r_score - margin  >=  alpha) { ... }
```

Значение `margin` в последних двух полуходах можно взять равным половине пешки. На близких к корню дерева полуходах значение поля возрастает. Это мы получим Futility pruning и Razoring, вместе взятые.

Возвращаясь к первой схеме, обратите внимание на один тонкий момент. Если в преузле была атака, мы не сбросим глубину, пока не сравняем материальную оценку со статической оценкой до хода. Это дает надежду, что программа увидит тактические выпады, где есть единственный ответ, чтобы сохранить материал. Условий сокращения глубины поиска можно придумать множество. Например, если не окончание игры, то ход назад является потенциально слабым перемещением. И если не теряем материал в узле, не шах и не взятие, то можно дополнительно сбросить глубину.

Правдоподобная генерация перемещений

В статическом поиске результат (и лучший ход) зависит от порядка перемещений. Как это происходит? Допустим, у нас есть два тихих перемещения. Оба дают приращение в узле +8. Оба увеличивают alpha до значения больше статической оценки после хода. Если мы просчитаем первым одно перемещение, второе подрежется статической оценкой. Оба хода ведут к подрезанию. Это явление получило название нестабильности поиска. Обычно нестабильность поиска считается досадной помехой, мешающей осуществить грубое усилие более глубоко. Нестабильность поиска возникает в тех случаях, когда отсечка или сокращение глубины основаны на пределах alpha-beta. В качестве примера посмотрим на использование нулевого хода. Мы можем применять нулевой ход без всяких условий:

```
int Search(int alpha, int beta, int player, int opponent, int depth)
{
   if(-Search(-beta,-(beta-1),opponent,player,depth-3) >= beta)
     return beta;
     {...} //сам поиск

}//function
```

Предлагается оговорить использование нулевого хода условием: текущая статическая оценка больше или равна beta. Дело в том, что если статическая оценка меньше beta, и мы пропустим ход, то в подавляющем большинстве случаев противник улучшит позицию, и дорогостоящий вызов нулевого хода пропадет напрасно.

Условие применения нулевого хода следующее:

```
if(L_SCORE >= beta)
{
  //do null-move
  {...}
}
```

Очевидно, что даже применение нулевого хода не спасает от нестабильности поиска. Тем более, если мы подрезаем сразу, без проверочных условий. Отсечение статической оценкой может быть двух видов:

```
int Search(int alpha,int beta,....)
{
   if(L_SCORE > alpha) alpha = L_SCORE;
   if(alpha >= beta) return beta; //можно и alpha
   {...} //сам поиск
}
```

или

```
int Search(int alpha,int beta,...)
{
   if(L_SCORE >= beta) return beta; //или L_SCORE
   {...} //сам поиск
}
```

Это достаточно грубо, но смысл есть. То, что является приблизительным на глубине 2, может быть принято за истину на глубине 15. Особенно это относится к первому варианту. Он используется для поиска взятий до конца (дающих наибольший прирост оценки). Если мы будем повышать alpha текущей статической оценкой и выбросим все перемещения, кроме взятий, то функция сосчитает результат размена фигур.

В таком форсированном варианте можно оставить все перемещения на первых двух полуходах, а потом считать только взятия. Тогда ходы, дающие прирост меньше 0 в узле, автоматически подрежутся статической оценкой, а у остальных максимум будет ограничен приростом оценки при ходе (R_SCORE).

Такой придаток из двух полуходов не сильно задержит поиск, но улучшит понимание программой позиционной оценки. Мне не известен другой способ заставить программу понимать позиционную игру, кроме как повышать alpha текущей оценкой. Это относится к любому виду позиционной оценки. Рассмотренные два полухода часто используются для более глубокого просмотра шахов, но речь сейчас не об этом, а о чувствительности к порядку ходов, хотим мы этого или нет. Приведу простой пример. Допустим, у нас есть списки фигур, инициализированные перед началом счета. Берем перемещения в следующей последовательности: Queen, Rook, Bishop, Knight, King, Pawn. Мы получим один стиль игры. Если мы возьмем перемещения в обратной последовательности: Pawn, King, Knight, Bishop, Rook, Queen, мы получим совершенно другой стиль. В шахматах ход назад считается потенциально слабым (по крайней мере, в начале партии). Если упорядочить ходы так, что ходы назад будут рассмотрены в последнюю очередь, программа

будет пытаться решать возникшие задачи, не отступая. Можно менять приоритет в ходе игры. Мы можем не полагаться на случай, а упорядочить счет:

1. Борьба за центр (начало партии).
2. Охота на короля (если ферзей не разменяли).
3. Проходные пешки (если нет ферзей).

Так как мы не вводим четких позиционных оценок, то стиль игры программы изменяется, но не жестко. Эффект может быть особенно сильным, если у нас более двух полуходов статического отсечения, и мы уточняем главное изменение, оставшееся от предыдущей итерации. Можно попытаться сохранить пешечный экран короля (тоже мягко) и т. д. Не прибегая к утомительной настройке коэффициентов вручную, можно научить программу пониманию некоторых позиционных факторов. Например, ладейные пешки вперед двигать плохо. Как плохо, насколько? Иногда это выигрывает всю партию, хотя двигать ладейные пешки в начале игры — признак дурного стиля. Но как задать эти признаки конкретными значениями? Порядок ходов частично решает эту задачу.

Есть еще один тонкий момент. Если бы мы хранили все дерево игры в памяти, у нас был бы задан порядок ходов и структура выборочных продлений. При каждой итерации лучшие ходы передвигаем вверх (в подсписках для каждого узла), это дерево гораздо лучше отражает многообразие вариантов, в соответствии с заданным критерием приоритетов. На практике, однако, нужен точный результат.

Единственный ответ

Я ни в коем случае не могу претендовать на полное изложение предмета, хотя сам и использую данный прием в своих программах. Если в позиции есть только одно легальное перемещение, то глубину просчета нужно увеличить. Все вроде бы просто. Только что считать легальным перемещением? Если существует некоторая позиция, где все ходы не несут материального выигрыша, а один выигрывает пешку, то можно сказать, что ход, выигрывающий пешку, является легальным, или правильным. Самое простое толкование легальности перемещения — это когда короля не бьют следующим ходом. Если есть только один ход, где король не под боем, то глубину нужно увеличить. В стандартных выборочных продлениях при шахе глубина остается неизменной, а если легальное перемещение единственное, то ее можно даже увеличить.

Существует и другое понятие легальности перемещений. Легальными называются те перемещения, где мы не теряем материал. Например, существует такой способ организации программы: сначала считается только материал (что намного быстрее) и формируется список легальных ходов с максималь-

ной оценкой. Потом этот список подается на вход функции, просчитывающей стратегию. Она может использовать какие угодно критерии классификации ходов по стратегическому принципу. Такой подход встречается во многих средних программах и даже некоторых коммерческих, чтобы доставить пользователю максимальное удовольствие от игры. Слишком сильную программу так не сделать. Дело в том, что все усилия брошены на то, чтобы сосчитать максимально глубоко, с трудом извлекается один лучший ход или главная строка изменения, а на то, чтобы извлечь целый список, просто не хватает времени. Это намного медленнее, чем получить один лучший ход. Тем не менее данная схема вполне работоспособна и годится для простой программы. Получаем список ходов, оценка которых (только материальная) равна максимальной. Если есть один ход, выигрывающий материал, значит, есть только один легальный ход, и думать дальше вообще не о чем. Если существует три хода, при которых мы не теряем материал, то стратегическая функция исследует только эти три хода.

Если мы хотим считать сразу и материальную, и позиционную оценку, то все намного сложнее. Нас интересует количество легальных ходов не только в корне дерева перебора, а в каждом узле. Причем, данный узел еще не просчитан. Как определить количество легальных ходов в нем? Мы не можем для каждого узла считать материал и количество легальных перемещений.

Рассмотрим счетчик легальных ходов, равный нулю перед просчетом данного узла. Как только мы нашли легальный ход, мы увеличиваем счетчик. Пока счетчик равен нулю, у нас одно легальное перемещение (предположительно). Если счетчик равен единице, у нас два легальных перемещения, и т. д. Результат счета не может быть меньше alpha для нашего узла. Как же мы определим легальные ходы, если не можем получить точный результат счета? Если значение alpha уже достигло порога легальности для данного узла, можно предположить, что у нас есть, как минимум, одно легальное перемещение в этом узле, и глубину дальше увеличивать нет смысла. Мы еще не просчитали, а уже делаем вывод о количестве легальных ходов в узле. Оценка данного узла не может быть менее alpha, и совершенно нет никакой разницы, получено ли значение alpha в этом узле или досталась от предыдущих узлов в строке. Мы все равно уменьшаем глубину счета. Можно даже подрезать статической оценкой и принять оценку хода как alpha. Статическая оценка после хода должна быть меньше или равна alpha. Если мы подрезаем статической оценкой, то текущая строка никак не попадет в главное изменение. Можно просто сократить глубину счета, когда alpha достигнет некоторого уровня, а оценка после хода не улучшает alpha. И то и другое не соответствует полному перебору. Мы должны понимать, что мы только регулируем глубину счета, пока оценка узла не достигла некоторого уровня. Ошибка может получиться и при подрезании статической оценкой, т. к. данная строка не попадет в главное изменение. До какой величины

нужно дотягивать alpha прежде, чем подрезать? Однозначного критерия нет. Обязательным может быть только статический критерий — оценка после хода не должна улучшать alpha. Можно дотягивать до корневой оценки дерева перебора. При этом не надо забывать, что корневая оценка может быть не стабильной, а промежуточной в результате размена или другой комбинации. Я лично использовал следующий критерий. Если противник взял фигуру предыдущим ходом, нужно попытаться компенсировать это и не подрезать, пока alpha не компенсирует потерю. Это хорошо работает на практике и в случае подрезания статической оценкой, и при сокращении глубины, если мы уточняем главное изменение. Программа комбинации просматривает намного глубже. Только результат счета нужно уточнять.

Есть и другие критерии для оценки. Например, ждут второй отсечки за beta. Если ее нет, то узел пересчитывается заново на большую глубину. Это очень медленно, и сразу возникает множество вопросов. Может, alpha-beta интервал мал, и мы портим стратегию? Нужно ли в таком случае из-за небольшого стратегического преимущества перебирать заново? Если мы используем NegaScout или другой алгоритм с нулевым окном, как это сочетается с легальностью ходов? Самый простой способ заключается в следующем. Если результат оценочной функции для некоторого хода существенно больше, чем для других, то данный ход нужно рассмотреть на большую глубину. Иными словами, взятия нужно рассматривать глубже. Это очень хорошо работает, только выборочных продлений становится много, и дерево нужно подрезать чем-то вроде Futility pruning. Взятия пешек можно, например, продлевать первые 6 полуходов, потом — по возрастанию веса взятой фигуры. Данный алгоритм хорошо известен из теории игр и качественно улучшает тактический уровень, но разорителен по времени выполнения. Хотя работающую схему получить вполне возможно.

ПРИЛОЖЕНИЯ

ПРИЛОЖЕНИЕ 1

Пример игры "Крестики-нолики"

Напомним правила: игра идет на квадратном поле 3 × 3 клетки. Игроки по очереди ставят в клетках крестики (один) и нолики (другой). Выигрывает тот, кто первым замкнул линию (вертикальную, горизонтальную, диагональную — неважно). Программа написана на языке Turbo Pascal 7 (см. листинг П1.1). Вывод осуществляется в текстовом режиме.

Листинг П1.1. Крестики-нолики

```
uses crt;

TYPE
 SquareType = (
               EMPTY , {пусто}
               CROSS , {крестик}
               NULL    {нолик}
             );
 {массив игры}
 PosType = array[0..8] of SquareType;

 {стартовая позиция}
const pos:PosType =
(
  EMPTY,EMPTY,EMPTY,
  EMPTY,EMPTY,EMPTY,
  EMPTY,EMPTY,EMPTY
);

{данная функция возвращает true, если в массиве
 игры есть замкнутая линия крестиков или ноликов}
function EndGame( z:SquareType):boolean;
```

```
begin
{       номера клеток
  0,1,2,
  3,4,5,
  6,7,8
}
  EndGame := true;
  if (pos[0] = z) and (pos[1] = z) and (pos[2] = z) then exit;
  if (pos[3] = z) and (pos[4] = z) and (pos[5] = z) then exit;
  if (pos[6] = z) and (pos[7] = z) and (pos[8] = z) then exit;
  if (pos[0] = z) and (pos[3] = z) and (pos[6] = z) then exit;
  if (pos[1] = z) and (pos[4] = z) and (pos[7] = z) then exit;
  if (pos[2] = z) and (pos[5] = z) and (pos[8] = z) then exit;
  if (pos[0] = z) and (pos[4] = z) and (pos[8] = z) then exit;
  if (pos[6] = z) and (pos[4] = z) and (pos[2] = z) then exit;
  EndGame := false;

end;


{сюда вернем лучший ход}
var retPos:PosType;

{функция поиска}
function Search(s:SquareType;          {наши фигуры}
                alpha,beta:integer;    {минимум и максимум}
                ply:integer            {глубина}
               ):integer;
var
 n,tmp:integer;
 f,opS:SquareType;
 findMove:boolean;
begin


    {определим цвет фигур противника}
    if s = NULL then opS := CROSS
      else opS := NULL;

    {если есть замкнутая линия, вернем оценку}
    if EndGame(opS) then begin
        {противник в преузле замкнул линию;
         это наш проигрыш}
```

```
        Search := -1;
        exit;
    end;

    {пока не нашли ни одной свободной клетки}
    findMove := false;

    {переберем все клетки доски}
    for n := 0 to 8 do
    begin
       f := pos[n];
       if f = EMPTY then begin
         findMove := true;
         {сделали ход}
         pos[n] := s;
         {просчет}
         tmp := -Search(opS,-beta,-alpha,ply+1);
         if tmp > alpha then
         begin
            alpha := tmp;
            if ply = 0 then
              Move(pos,retPos,sizeof(PosType));
         end;
         {восстановим позицию}
         pos[n] := EMPTY;
         if alpha >= beta then break;

       end;
    end;

    if not findMove then Search := 0 {ничья}
      else  Search := alpha;
end;



{выводит позицию на экран}
procedure ShowPos;
  function Ch(n:integer):char;
  begin
   if pos[n] = CROSS then  Ch :='*'
      else if pos[n] = NULL then Ch := '0'
      else Ch := ' ';
  end;
```

```
begin
   Writeln('   -------');
   Write('   |', Ch(0), '|', Ch(1), '|', Ch(2), '|'); Writeln;
   Writeln('   -------');
   Write('   |', Ch(3), '|', Ch(4), '|', Ch(5), '|'); Writeln;
   Writeln('   -------');
   Write('   |', Ch(6), '|', Ch(7), '|', Ch(8), '|'); Writeln;
   Writeln('   -------');


end;

{возвращает true, если конец игре}
function GameOver:boolean;
var
 n:integer;
begin
    GameOver := false;
    if not (EndGame(CROSS) or EndGame(NULL)) then
    begin
        for n := 0 to 8 do
          if pos[n] = EMPTY then exit;
    end;
    GameOver := true;
end;


VAR
  ch:integer;
  tmp:integer;
  n:integer;
BEGIN
HighVideo;

 repeat

    Writeln;
    Writeln('////////////////////////////////////////');
    ShowPos;
    Writeln;

    if GameOver then begin
        for n := 0 to 8 do pos[n] := EMPTY;
```

```
        Writeln('Game Over!!!');
        Writeln;
        ShowPos;
        Writeln;
    end;

    repeat
      Write('Enter move number [0..8] or -1 (exit):'); Read(ch);
    until  (ch = -1) or
           ((ch >=0) and (ch <= 8) and (pos[ch] = EMPTY)) ;
    if ch = -1 then break;
    pos[ch] := CROSS;

    if not GameOver then begin
       tmp := Search(NULL,-2,2,0);
       pos := retPos;
    end;

  until ch = -1;


LowVideo;
Writeln('done');
END.
```

ПРИЛОЖЕНИЕ 2

Пример генерации перемещений

Можно рассуждать сколько угодно о различных программных решениях и алгоритмах, но лучше попытаться сделать хоть что-то своими силами. Давайте рассмотрим генерацию перемещений. Это в шахматной программе наиболее трудоемкий участок кода. Раньше подобные вещи писали исключительно на ассемблере, чтобы получить требуемую эффективность. Сейчас применение ассемблера совершенно не обязательно. Современные процессоры при хорошей компиляции и грамотном алгоритме обрабатывают код языка высокого уровня ничуть не хуже, чем ассемблер. Можно сказать даже более. Очень непросто написать ассемблерный код, который будет выполняться быстрее. Но это в сторону. Наша задача — рассмотреть алгоритмы, а не их оптимизацию на уровне машинного кода.

Генератор перемещений является очень ответственным фрагментом программы. Он должен, как минимум, безошибочно генерировать перемещения. В шахматах существуют некоторые ходы, существенно усложняющие дело. Рассмотрим их подробнее.

1) Простой ход фигуры из одной позиции в другую. Фигура при этом стирается в старой клетке доски и записывается в новой. Вся необходимая информация об этом перемещении содержится в индексе стартовой клетки и индексе конечной клетки. Можно назвать их N1 и N2. Это самый простой ход, и с ним не будет путаницы.

2) Простой ход с взятием. Фигура при этом покидает стартовую клетку и становится в другую. При этом она съедает фигуру противника в конечной клетке. Для данного перемещения нужно тоже две переменных N1 и N2.

3) Превращение пешки. Пешка дошла до последней горизонтали и превращается в ферзя. Пешка может ступить на последнюю горизонталь двумя способами. Первый — это сделать ход вперед на одну клетку, при этом в целевой клетке не должно быть ничего. Второй — пешка взяла по диагонали фигуру противника и при этом превратилась в ферзя, так как вступила

на последнюю горизонталь. Для описания данного хода тоже достаточно стартовой и конечной координат. Мы просто должны проверить, не вступила ли пешка на последнюю горизонталь при ходе.

4) Взятие пешкой через битое поле. Данный ход является самым сложным при программировании. Для его генерации нужно знать, откуда и куда ходила последняя фигура противника. Если это был первый ход пешки через клетку, и она пересекла поле, битое пешкой противника, она может быть взята. При этом она становится в поле взятия (в котором пусто, так как взятая пешка перепрыгнула через него). С доски снимается съеденная пешка, которая стоит в другом поле. Для описания данного перемещения нужно уже три переменных: стартовая координата фигуры, конечная координата фигуры и `eatFN` — координата съеденной фигуры.

Пешка бьет фигуру не в той клетке, куда становится. Данный ход довольно редок в шахматах, но его все равно нужно учитывать, и лучше об этом подумать заранее, чем потом исправлять готовую программу.

5) Рокировки. Существуют правая и левая рокировки. Король и ладья делают нестандартное перемещение. Рокировку можно делать, только если король и ладья до этого ни разу не ходили, и король не под шахом. Король, так как он перепрыгивает через клетку, не должен пересекать битое поле. Эту особенность можно и не учитывать для простой программы, но что король не делает рокировки под шахом, надо учитывать обязательно. Для генерации перемещения нужно много информации: перемещалась ли фигура до этого, детектор шахов, стартовая и конечная координаты короля, стартовая и конечная координаты ладьи, кроме того, флаг перемещения, характеризующий вид данного хода. Попробуем сформировать тип описателя перемещения. Он нужен для того, чтобы функция `MakeMove`, которая выполнит данное перемещение, могла его сделать, а функция `UnMakeMove` — восстановить позицию. Шахматную позицию, хотя она и имеет два измерения, мы представим в виде одномерного массива:

```
  const pos:array[0..7,0..7] of byte =
(
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0),
  (0,0,0,0,0,0,0,0)
);
```

В шахматах доска имеет размер 8 × 8 клеток. В нашем случае массив проиндексирован 0..7, что одно и то же. Данный двумерный массив можно представить одномерным:

```
const pos:array[0..63] of byte =
(
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0
);
```

Что мы выигрываем, представляя двумерный массив одномерным? Раньше профессионалы бы заметили, что двумерный массив при компиляции подразумевает дополнительную операцию умножения (или битового сдвига для хорошего компилятора). Компилятор, чтобы определить конкретный адрес в массиве, делает следующее:

```
N := y*8 + x;
```

Хороший компилятор сделал бы следующее:

```
  N := (y shl 3) + x;
```

Битовый сдвиг влево на 3 — это то же, что умножить число на 8.

Сейчас это не актуально, и выбор размерности массива диктуется лишь соображениями удобства программирования. Современные компиляторы уже не заменяют маленькие операции умножения битовым сдвигом для скорости. В нашем случае, если мы будем использовать двумерный массив, у нас резко увеличится количество переменных, необходимых для описателя перемещения. Нам будут нужны для каждой клетки две координаты: X и Y. Если мы используем одномерный массив, то вместо двух координат достаточно одной — N. Одномерная индексация имеет еще массу преимуществ. При генерации перемещений потребуется постоянно проверять выход за пределы массива. Например, перемещения коня. Мы 8 раз проверяем, не вышли ли за пределы доски. Это приведет к избыточному коду. Генератор ходов и так довольно непростая часть программы, и лишнего кода следует избегать. Можно прибегнуть к небольшой хитрости. Расширим границы массива, оставив игровое поле в центре массива, а ненужные поля пометим особым флагом, указывающим, что мы вышли из игрового поля. При такой организации количество избыточного кода существенно сократится. Это мы

увидим впоследствии, а пока представим наш целевой массив как следующий:

```
const pos:array[0..16*16-1] of integer =
(
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,

 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,
 F,F,F,F,  0,0,0,0,0,0,0,0,  F,F,F,F,

 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F,
 F,F,F,F,  F,F,F,F,F,F,F,F,  F,F,F,F
);
```

Само игровое поле находится в центре и помечено 0. Клетки за пределами поля помечены константой F. Чтобы преобразовать координату N данного массива в координаты массива 8 × 8, нужно сделать следующее:

```
X := N mod 16  -  4;
Y := N div 16  -  4;
```

Данную тонкость можно представить в более профессиональном виде:

```
X := (N and 15) - 4;
Y := (N shr 4)  - 4;
```

Или на языке C:

```
x = (n & 15) - 4;
y = (n >> 4) - 4;
```

Обратное преобразование несколько избыточно по коду, можно просто воспользоваться массивом перекодировки, где для каждой клетки с координатой N записана координата этой клетки в массиве 16 × 16. Это просто и быстро выполняется.

```
const hlat16:array[0..63] of integer =
(
   68,69,70,71,72,73,74,75 ,
   84,85,86,87,88,89,90,91,
   100,101,102,103,104,105,106,107,
   116,117,118,119,120,121,122,123,
   132,133,134,135,136,137,138,139,
   148,149,150,151,152,153,154,155,
   164,165,166,167,168,169,170,171,
   180,181,182,183,184,185,186,187,
);
```

Нам пока для генерации перемещений нужно лишь знать, что наш массив индексирован как одномерный, и достаточно одного индекса для определения положения фигуры. Схематично представим описатель перемещения. Он должен хранить все поля, необходимые для полного описания любого хода. С учетом нестандартных ходов, рокировки и взятия пешкой через битое поле, этот описатель получится громоздким.

```
CONST
{типы перемещений}
  kNormal      = 1;  {простой ход}
  kCapture     = 2;  {простое взятие}
  kFIrstPawn   = 4;  {первый ход пешкой через клетку}
  kEnPassant   = 8;  {взятие пешкой через битое поле}
  kLeftCastling   = 16; {левая рокировка}
  kRightCastling  = 32; {правая рокировка}
  kPromote  = 64; {превращение пешки}
  kNotMove  = 0;  {нет перемещений}

 {примерный описатель хода}
 Type TMove = record
     N1,N2:integer;  {   откуда и куда пошла фигура  }
     F:integer;      {сама фигура}
     eatFN:integer;  {координата взятой фигуры}
     eatF:integer;   {взятая фигура}
     desc:integer;   {тип хода}
     rN1,rN2:integer; {откуда и куда пошла ладья при рокировке}
 end;
```

Потребуются еще значения фигур в новой позиции. Например, ходил ферзь с d1 на d2. Ячейка в массиве позиции должна отражать, что ферзь перемещался. При сканировании массива позиции мы должны четко знать, какая фигура перемещалась хоть раз, а какая — нет. Это нужно для генерации первых ходов пешки и рокировок. Позиция у нас представлена одномерным

массивом чисел. Давайте разберемся с этим. Если в клетке доски 0, фигуры нет. Введем коды фигур:

```
   {фигуры}
CONST
 KING   = 1;
 QUEEN  = 2;
 ROOK   = 3;
 BISHOP = 4;
 KNIGHT = 5;
 PAWN   = 6;  {итого первые 3 бита}
```

Наши коды фигур представлены числами от 1 до 6 включительно. Это значит, они поместятся в первых 3 битах числа. Остальные биты числа мы можем использовать по своему усмотрению. Определим необходимые константы:

```
{цвет}
CBLACK = 64;
CWHITE = 128;
```

Эти десятичные числа займут биты с номерами 6 и 7. У нас еще остаются свободными биты с номерами 5, 4, 3. Один из них используем под флаг перемещения. Если он установлен, фигура перемещалась до этого хоть один раз.

```
CMOVE   = 16;  {флаг, что фигура перемещалась}
```

Позиция находится в центре массива 16 × 16, а по краям находятся поля, которые нужно заполнить флагом переполнения, чтобы было видно, когда мы вышли за пределы доски. Для этого можно использовать оставшийся свободным бит с номером 3.

```
COUT   = 8;
```

У нас еще остался свободным бит с номером 5. Его мы используем позже. Проверить, какая фигура стоит в клетке доски, легко можно с помощью поразрядной логической операции AND. Она в результате оставит только те биты, которые установлены в 1 у обоих операндов. Чтобы из клетки доски выделить код фигуры (без цвета), мы должны сделать следующее:

```
F := S  and  7;
```

где S — клетка доски.

Чтобы выделить цвет фигуры:

```
COLOR := S and (CBLACK + CWHITE);
```

Введем некоторые дополнительные константы для удобства:

```
CCOLOR  = CBLACK + CWHITE;  {маска для выделения цвета}
CFIGURE = 7;                {маска для выделения фигуры}
```

Для того чтобы выделить фигуру, мы можем написать:

```
F := S and CFIGURE;
```

Цвет:

```
COLOR :+ S and CCOLOR;
```

Намного понятнее в реальной программе. В языке С подобные операции удобно делать с помощью макросов с параметрами.

Чтобы проиллюстрировать все изложенное, возьмем все перемещения ферзя. Наша позиция, как вы помните, представлена массивом 16 × 16. Игровое поле 8 × 8 находится в центре, а лишние поля заполнены флагом переполнения.

```
procedure GetMovesQueen( Nf:integer      {координаты фигуры};
                         opColor:integer {цвет фигур противника}
                       );
const
 addNQueen:array[0..8] of integer =
 (16,-16,1,-1,17,-17,15,-15,0);
var
 N,j:integer;
begin
    j := 0;
    while addNQueen[j] <> 0 do
    begin
        N := Nf + addNQueen[j];
        while pos[N] = 0 do
        begin
          {здесь простое перемещение ферзя в
           пустую клетку (не взятие)
          }
          N := N + addNQueen[j]);
        end;
        { не пустая клетка доски,
          если на ней стоит фигура противника,
           это взятие
        }
        if (pos[N] and CCOLOR) = opColor then
```

```
        begin
           {нашли взятие}
        end;
        j := j + 1;
    end;
end;
```

Как видите, все предельно просто, и объем кода минимален. Если бы мы использовали двумерный массив или не имели по краям полей с флагом переполнения, объем кода значительно возрос бы.

Для хранения приращений перемещения ферзя мы используем массив addNQueen. Мы сканируем его, пока не встретим 0. Это значит, что взяты перемещения ферзя во всех направлениях. В одномерном массиве приращение при движении фигуры не может быть 0. Мы строили линию, пока в клетке доски пусто. Для того чтобы проверить, что в непустой клетке стоит фигура противника, потребовалась всего одна операция сравнения. Эта клетка могла быть за пределами доски (тогда в ней был бы флаг COUT), она могла быть пустой (тогда ее значение было бы 0), в ней могла быть наша фигура. Вместо всех этих многочисленных условий мы просто проверили: цвет фигур в данной клетке совпадает с цветом фигур противника? И все.

```
if (pos[N] and CCOLOR) = opColor then  ...
```

Так как цвета фигур имеют значения 64 и 128, а флаг переполнения 8, то при операции AND они никогда не пересекутся, так как это различные биты.

Данная функция только находила перемещения, но ничего с ними не делала. Мы еще не определились с описателем перемещений. Это довольно непростая задача. Если в средней шахматной позиции примерно 40 ходов, и наш генератор перемещений при каждом найденном ходе будет заполнять огромные структуры данных, то эффективность данного кода будет очень низка. Есть и другие причины, заставляющие стремиться к минимальному размеру описателя перемещения. В самом простейшем случае описатель перемещения следующий:

```
TMove = record
   N1,N2:byte;
end;
```

Или может выглядеть так:

```
TMove = record
   N1,N2:byte;          {откуда и куда}
   moveType:byte;       {тип перемещения}
   next:byte;  {номер следующего описателя в буфере}
end;
```

Это несколько отвлеченный пример, но он демонстрирует основные принципы. Когда мы находим очередное перемещение, мы добавляем его в свой список. Списки потом соединяются, и функция `Generate` (генератор перемещений) возвращает указатель на первый элемент общего списка перемещений. Типичный описатель списка:

```
PList = ^TList;
TList = record
   {...}
   PList next;
end;
```

Поле `next` указывает на следующий элемент списка. Это обычный указатель, и его размер, как правило, составляет 4 байта. В нашем случае функция `Generate` для описателей перемещения использует буфер, откуда берутся свободные описатели. Буфер может располагаться где угодно, обычно в стеке вызывающей функции. Размер буфера ограничен некоторым реальным числом, представляющим максимально возможное количество ходов в данной позиции. Для шахмат это может быть 200. Таким образом, буфер состоит из 200 элементов, и для адресации следующего элемента можно использовать не указатель, а просто номер элемента в буфере. Например:

```
const MAX_MOVES = 200;
      EMPTY = 255; {такого номера не может быть}
buf:array[0..MAX_MOVES-1] of TMove;
list:byte;   {индекс первого элемента списка}
count:byte;  {номер первого свободного описателя в буфере}
list := EMPTY;
count := 0;
{...}
{нашли перемещение}
with buf[count] do begin
   {заполнили все необходимые поля}
   {...}
   {вставим в список}
   next := list;
end;
   list  := count;
   count := count + 1;
```

Здесь не показано, каким образом мы нашли перемещение. Это в данном случае не важно. Чтобы просмотреть весь список, нужно сделать примерно следующее:

```
while list <> EMPTY do begin
    with buf[list] do begin
```

```
      {необходимые действия с данным ходом}
      {...}
      {номер следующего хода}
      list := next;
    end;
end;
```

Я думаю, эти операции не сложнее, чем с указателями. Можно использовать и указатели, это совершенно не принципиально. Просто повысится размер описателя перемещения, а у нас задача ограничить его 4 байтами.

В приведенном ранее примере мы осуществляли вставку в начало списка. Если есть необходимость вставлять в конец списка, можно сделать следующее:

```
type PByte = ^byte;
const MAX_MOVES = 200;
      EMPTY = 255; {такого номера не может быть}
buf:array[0..MAX_MOVES-1] of TMove;
list:byte;   {индекс первого элемента списка}
lastItem:PByte;{указатель на адрес конца списка}
count:byte;  {номер первого свободного описателя в буфере}
list := EMPTY;
lastItem := @list;
count := 0;
{...}
{нашли перемещение}
with buf[count] do begin
   {заполнили все необходимые поля}
   {...}
   {вставим в список}
   next := EMPTY;
   lastItem^  := count;
   lastItem   := @next;
end;
   count := count + 1;
```

В конец списка удобнее, потому что мы при генерации перемещений будем отслеживать несколько подсписков, которые нужно потом соединить в один.

Рассмотрим еще один важный момент. При генерации перемещений нам нужно найти наши фигуры. Сканировать массив доски в поисках своих фигур неэффективно. Генератор перемещений должен работать максимально быстро. Для определения места расположения фигур можно ввести списки фигур. Они инициализируются один раз перед началом просчета и потом

только корректируются. Каждый элемент списка фигур должен содержать координату фигуры и собственно фигуру. Описатель элемента списка фигур может выглядеть как

```
PPiece = ^TPiece;
TPiece = record
  __N:integer;      {номер клетки фигуры}
  __F:integer;      {фигура}
  __index:integer; {номер описателя в буфере}
  __next:PPiece; {следующий элемент списка}
end;
```

К каждому элементу списка фигур можно обратиться по указателю и по номеру данного элемента в буфере, откуда берутся описатели. Списки фигур инициализируются следующим образом. Сначала идет король, потом ферзь и так все фигуры по убыванию. Выглядит это примерно так: King, Queen, Rook, Rook, Bishop, Bishop, Knight, Knight, Pawn. Вот пример функции, инициализующей списки фигур перед просчетом:

```
{глобальные переменные}
var
glFigBuf:array[0..32] of TPiece; {буфер описателей фигур}
glUpFColor:integer;  {цвет верхних фигур}
glUpFList:PPiece;{список фигур верхних}
glBotFList:PPiece;{список фигур нижних}
{инициализирует списки фигур}
PROCEDURE InitFList;
VAR
  p:PPiece;
  count,K,N,f:integer;
BEGIN
     count := 0;
     glUpFList := NIL;
     glBotFList := NIL;
     for K := PAWN downto KING do
     for N := 0 to 16*16-1 do
     begin
         f := glPos[N];
         if (f = 0) or (f = COUT) or ((f and CFIGURE) <> K)
           then continue;
         p := @glFigBuf[count];
         p^.__F := f;
         p^.__N := N;
         p^.__index := count;
```

```
        inc(count);
        if (f and CCOLOR) = glUpFColor then
        begin
           p^.__next := glUpFList;
           glUpFList := p;
        end else begin
           p^.__next := glBotFList;
           glBotFList := p;
        end;
     end;
     glFigBufCount := count; {количество фигур}
END;
```

К любому элементу списка фигур можно обратиться по его указателю и номеру в буфере glFigBuf. Окончательный вариант описателя перемещения, который будет однозначно определять ход:

```
PMove = ^TMove;
TMove = record
  case byte of
  1:(
    __FigIndex:byte;  {индекс описателя перемещения в glFigBuf}
    __N2:byte;        {куда пошла}
    __desc:byte;      {тип перемещения}
    __next:byte;      {индекс следующего перемещения}
    );
  2:(
      __data:LongInt;
    );
end;
```

Стартовая координата и сама фигура отсутствуют, так как у нас в описателе перемещения есть индекс элемента списка фигур для данной фигуры, а он содержит все необходимые данные. Скажем, чтобы получить стартовую координату фигуры и другую информацию, мы должны

```
with glFigBuf[ __FigIndex] do
 begin
     {все поля}
 end;
```

Почему такая сложность и запутанность? Дело в том, что перемещения должны браться максимально быстро. В нашем случае нам нужно всего 4 байта. Это совсем немного и по размеру, и по объему заполнения. Большинство ходов не потребуется, и нет смысла создавать для каждого хода

развернутый описатель перемещения. Функция `Generate` — самая критичная по времени функция выполнения программы. Были многочисленные попытки реализовать ее аппаратно, чтобы повысить производительность при вычислении. Если у нас будет генератор перемещений и функции `MakeMove` и `UnMakeMove`, делающие перемещение и восстанавливающие позицию, на это можно будет не обращать внимания, а заниматься более интересными вещами.

В поле описателя перемещения есть еще элемент `__data`. Он занимает то же пространство в памяти, что и 4 байта основного описания. Это так называемое объединение (`Union`). Если нужно скопировать один описатель перемещения в другой, совершенно не обязательно копировать все поля по очереди. Можно написать

```
move1.__data := move2.__data;
```

Все поля при этом будут корректно перенесены, так как поле `__data` занимает в памяти машины то же место, что и 4 байта основного описания.

Рассмотрим генерацию взятий. Независимо от структуры программы, для генерации взятий потребуется отдельная функция. Это связано с тем, что взятия просматриваются на большую глубину, чем другие перемещения. В форсированных вариантах желательно рассматривать взятия до конца. Некоторые позиции можно быстро отсечь только взятием, и совершенно не потребуются другие ходы. Так или иначе, а генерация взятий — это самое критичное место в программе по скорости. Более того, взятия должны браться в определенном порядке, так как метод alpha-beta очень чувствителен к порядку ходов. Рекомендуют первым попробовать взятие последней ходившей фигуры противника. Это действительно в большинстве случаев отсекает всю ветвь поиска. Первыми идут наиболее вероятные (лучшие) взятия, последними наименее. Ход, когда пешка бьет ферзя, имеет гораздо больше шансов быть разумным, чем когда ферзь бьет пешку.

Чтобы alpha-beta процедура нормально считала, ходы, дающие наибольшие приращения в узле, должны быть рассмотрены первыми. Это верно и для материальной, и для стратегической оценки. Мы должны наиболее ценные взятия рассматривать первыми, а наименее ценные последними, по принципу MVV/LVA (наиболее ценная жертва — наименее ценный нападающий). Итак, определим порядок сортировки взятий:

1. Взяли короля.

2. Взятия, бьющие последнюю ходившую фигуру противника.

3. Взятия по убыванию веса захваченной фигуры.

Как видите, все достаточно непросто. Взятия можно и не сортировать, но скорость просчета при этом резко снизится, и рассматривать форсированные варианты до конца мы просто не сможем.

Какие же требования предъявляются к функции взятия? Она должна максимально быстро сканировать все перемещения фигур и выдавать взятия в отсортированном порядке. Самым простым решением является введение в описатель перемещения дополнительного поля цены хода. Потом каждый проход пузырьковой сортировки подкачивает лучшее перемещение. Так можно поступать, когда много сортируемых значений, и нужно быстро получить лучший ход для немедленного перехода в следующую позицию. К счастью, сортируемых значений у нас не так много (всего 6), и их можно сразу вставлять в свой список, а потом списки соединять в один. Кроме того, обратите внимание, что списки фигур идут по убыванию веса фигуры. Взятия ферзя будем вставлять в начало подсписка ферзя, они расположатся как раз по возрастанию веса атакующей фигуры. Вот пример функции взятия в отсортированном порядке:

```
TYPE
  TMoveArray = array[0..1] of TMove;
  PMoveArray = ^TMoveArray;
  {типы перемещений}
 CONST
{типы перемещений}
  kNormal          = 1;  {простой ход}
  kCapture         = 2;  {простое взятие}
  kFIrstPawn       = 4;  {первый ход пешкой через клетку}
  kEnPassant    = 8;  {взятие пешкой через битое поле}
  kLeftCastling  = 16; {левая рокировка}
  kRightCastling = 32; {правая рокировка}
  kPromote  = 64; {превращение пешки}
  kNotMove  = 0;  {нет перемещений}
 CONST
  M_EMPTY = 255; { указатель NIL}
{приращения при движении фигур
 по коду фигуры
 если приращение 0, конец строки приращений
 }
type tagArray = array[0..8] of integer;
const
_addEmpty:tagArray = (0,0,0,0,0,0,0,0,0);   {пустая строка для кода
0}
_addKing:tagArray  = (16,-16,1,-1,17,-17,15,-15,0); {для короля}
_addQueen:tagArray = (16,-16,1,-1,17,-17,15,-15,0); {для ферзя}
_addRook:tagArray  = (16,-16,1,-1,0,0,0,0,0);       {для ладьи}
_addBishop:tagArray = (17,-17,15,-15,0,0,0,0,0);     {для слона}
_addKnight:tagArray =
(32+1,32-1,16+2,16-2,-(32+1),-(32-1),-(16+2),-(16-2),0);{для коня}
```

```
{адреса строк приращений по коду фигуры}
const glAdd:array[0..6] of PIntArray =
(
@_addEmpty,@_addKing,@_addQueen,@_addRook,@_addBishop,@_addKnight,
@_addEmpty
);
{получает взятия в отсортированном порядке}
PROCEDURE GenerateAndSortAllCaptures( color:integer; {цвет фигур}
                                      node:PMove; {ход в эту позицию}
                                      buf:PMoveArray; {стековый буфер }
                                      size:integer; {сколько TMove в буфере}
                                      {номер первого элемента
                                      списка в буфере }
                                      var firstMove:integer;
                                      var countMoves:integer {сколько
                                      перемещений}
                                     );

  {единица соответствует коду фигуры, чье перемещение не
  требует линии: король и конь}
const  simpleF:array[0..6] of byte =
  (0,1,0,0,0,1,0);
 const pawnInQueen:array[0..15] of byte =
(0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0);
  VAR
   list:PPiece;     {список наших фигур}
   count:integer; {сколько описателей перемещений использовано}
   {список взятий, бьющих последнюю ходившую фигуру}
   lastEatList:byte;    {начало списка}
   pLastEatList:PByte;  {всегда указывает на конец списка}
   {списки остальных взятий}
   eatList:array[0..6] of byte;   {по коду взятой фигуры}
   pEatList:array[0..6] of PByte; {на концы списков}
   n:integer;
   f:integer; {фигура, чьи перемещения берем}
   pAdd:PIntArray; {строка перемещений}
   index:integer; {куда фигура переместилась}
   delPawn:integer; {приращение при движении пешки}
   tmp:integer;
   eatF:integer;
  procedure AddEatMove( codeEatF:integer; {код взятой фигуры}
                        flag:integer      {тип хода}
                      );
```

```
var
   listNumber:integer;
begin
    if count < size then
    begin
       with  buf^[count] do
       begin
          __FigIndex := list^.__index;
          __N2 := index;
          __desc := flag;
          __next := M_EMPTY;
          if (node^.__desc <> kNotMove) and
             (node^.__N2 = index) then
          begin
            { взяли только ходившую фигуру}
            __next  := lastEatList;
            lastEatList := count;
            if __next = M_EMPTY then
               pLastEatList := @__next;
          end else begin
              {остальные взятия}
               listNumber := codeEatF;
               __next := eatList[listNumber];
               eatList[listNumber] := count;
               if __next = M_EMPTY then
                  pEatList[listNumber] := @__next;
         end;
        end;{with}
        inc(count);
      end;
end;
{//////////////////////////////////////}
VAR
 opColor,del:integer;
BEGIN
   {инициализируем списки}
   lastEatList  := M_EMPTY;
   pLastEatList := @lastEatList;
   for n := 0 to 6 do
   begin
      eatList[n] := M_EMPTY;
      pEatList[n] := @eatList[n];
   end;
```

```
firstMove := M_EMPTY; {нет пока перемещений}
countMoves := 0;
count := 0;
{инициализируем переменные в зависимости от того,
верхние ли наши фигуры или нижние}
if(color = glUpFColor) then
begin
   list := glUpFList;
   delPawn := 16;
end else begin
   list := glBotFList;
   delPawn := -16;
end;
if color = CBLACK then opColor := CWHITE
  else opColor := CBLACK;
  while (list <> NIL) do
begin
  f := glPos[list^.__N];
  if f <> list^.__F then
  begin
      list := list^.__next;
      continue;
  end;
 {  /////case//// }
 case (f and CFIGURE)  of
       PAWN: begin
          {/////////PAWN////////////////////}
          {на клетку вперед}
          index := list^.__N + delPawn;
          if(glPos[index] = 0) then
          begin
              {добавим в список}
              if pawnInQueen[index shr 4]  <> 0 then
              {пешка в ферзи}
                 AddEatMove(QUEEN,kPromote);
          end;
          {взятия}
          index := list^.__N + delPawn + 1;
          tmp := glPos[index];
          if (tmp and CCOLOR) = opColor  then
          begin
             if pawnInQueen[ index shr 4 ]  <> 0 then
             {пешка в ферзи}
                AddEatMove(QUEEN,kPromote+kCapture)
```

```
            else
               AddEatMove(tmp and CFIGURE, kCapture);
         end;
         index := list^.__N + delPawn - 1;
         tmp := glPos[index];
         if (tmp and CCOLOR) = opColor then
         begin
            if pawnInQueen[ index shr 4 ]  <> 0 then
            {пешка в ферзи}
               AddEatMove(QUEEN,kPromote+kCapture)
            else
               AddEatMove(tmp and CFIGURE, kCapture);
         end;
        {взятие через битое поле}
         if(node^.__desc and kFIrstPawn) <> 0 then
         begin
           if (node^.__N2+1) = list^.__N then begin
               index := list^.__N + delPawn - 1;
               AddEatMove(PAWN,kEnPassant);
           end;
          if (node^.__N2-1) = list^.__N then begin
               index := list^.__N + delPawn + 1;
               AddEatMove(PAWN,kEnPassant);
           end;
         end;
         {/////////PAWN////////////////////}
   end; {PAWN}
     else begin {все фигуры, кроме пешек}
      {получим строку приращений при движении}
      n := 0; {индекс приращения}
      pAdd := glAdd [ f and CFIGURE ];
      del := pAdd^[n];
      while del <> 0 do
      begin
         index := list^.__N + del;
         {линия}
         if simpleF[ f and CFIGURE ] =  0 then
             while glPos[index] = 0 do inc(index,del);
           {здесь, возможно, не взятие
            в любом случае, ненулевая клетка
           }
         eatF := glPos[index];
         if (eatF and CCOLOR) = opColor then
```

```
            begin
                {добавим взятие}
                AddEatMove(eatF and CFIGURE, kCapture);
            end;
            inc(n);
            del := pAdd^[n];
          end; {while}
        end;{все фигуры}
    end;  {case}
    {  /////case//// }
      list := list^.__next;
  end;{while}
    {собираем подсписки}
  for n := PAWN-1 downto KING do
     pEatList[n]^ := eatList[n+1];
  pLastEatList^ := eatList[KING];
  firstMove := lastEatList;
  countMoves := count;
END;{function}
```

Данный код требует некоторого пояснения. Перемещения пешек берутся отдельно, так как это нестандартные ходы. Перемещения всех других фигур берутся сразу для всех фигур. По коду фигуры вычисляется адрес начала строки приращений. Массив `simpleF` содержит 1 в ячейке, соответствующей коду фигуры, для которой не нужно строить линию: конь или король. Обратите внимание, что объем кода для такой нетривиальной задачи, как взятие и сортировка перемещений, совсем небольшой. Найденные взятия вносятся в свой подсписок по коду съеденной фигуры. Вносятся в начало списка. Так как списки фигур идут по убыванию веса фигуры, а внесение в начало списка инвертирует порядок, происходит правильная сортировка. Фигуры в списке фигур расположены: King, Queen, Rook. Взятия в подсписке ферзя: Pawn, Bishop, Rook. Взятия, бьющие последнюю сходившую фигуру противника, имеют высший приоритет. Они вносятся в особый список, списки можно потом соединить. Сортировка взятий выполняется практически без накладных расходов.

Рассмотрим в качестве примера еще функцию, генерирующую простые перемещения. Она еще более компактна. В ней не выполняются никакие сортировки, так как это учебный пример. В реальной программе можно вести три подсписка по приращению стратегической оценки:

1. Приращение меньше 0.
2. Небольшое приращение (разумное значение).
3. Лучшие ходы.

Несмотря на кажущуюся абсурдность такой сортировки (ведь мы не знаем, какой ход в действительности лучший), alpha-beta считает качественно быстрее. Вместо упомянутой сортировки по значению можно применить нечто более изысканное, например, по статистике полей (эвристика истории). Эвристику убийцы можно рассматривать в основной программе, а не загружать им генератор перемещений. Готовых рецептов здесь нет. Всякий делает по своему вкусу и возможностям. Вот пример функции:

```
{только ходы без приращения материальной оценки}
PROCEDURE GenerteNotCaptures( color:integer; {цвет фигур}
                              node:PMove; {ход в эту позицию}
                              buf:PMoveArray; {стековый буфер }
                              size:integer; {сколько TMove в буфере}
                              {номер первого элемента
                              списка в буфере}
                              var firstMove:integer;
                              var countMoves:integer {сколько
                              перемещений}
                             );
 const  simpleF:array[0..6] of byte =
  (0,1,0,0,0,1,0);
 const pawnInQueen:array[0..15] of byte =
(0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0);
  {стартовые горизонтали пешек}
const pawnStartY:array[0..15] of byte =
(0,0,0,0, 0,1,0,0,0,0,1,0, 0,0,0,0);
LABEL
 done;
VAR
   list:PPiece;     {список наших фигур}
   count:integer; {сколько описателей перемещений использовано}
   {список взятий, бьющих последнюю ходившую фигуру}
   {списки остальных ходов}
   noEat:byte;
   pNoEat:PByte;
   n:integer;
   f:integer; {фигура, чьи перемещения берем}
   pAdd:PIntArray; {строка перемещений}
   index:integer; {куда фигура переместилась}
   delPawn:integer; {приращение при движении пешки}
   tmp:integer;
   canBeLeftCastl,canBeRightCastl:boolean;
   fN:integer;
```

```
VAR
 noEatMove:TMove; {шаблон описателя перемещения}
 del:integer;
 isSimpleFig:boolean;
BEGIN
   {указатель на начало списка - вставка в конец}
   noEat  := M_EMPTY;
   pNoEat := @noEat;
   {шаблон наиболее распространенного хода}
   noEatMove.__desc := kNormal;
   noEatMove.__next := M_EMPTY;
   firstMove := M_EMPTY; {нет пока перемещений}
   countMoves := 0;
   count := 0;
   {инициализируем переменные в зависимости от того,
   верхние ли наши фигуры или нижние}
   if(color = glUpFColor) then
   begin
      list := glUpFList;
      delPawn := 16;
   end else begin
      list := glBotFList;
      delPawn := -16;
   end;


   while (list <> NIL) do
   begin
     fN := list^.__N;
     f := glPos[fN];
     if f <> list^.__F then
     begin
         list := list^.__next;
         continue;
     end;
    noEatMove.__FigIndex := list^.__index;
    {  /////case//// }
    case (f and CFIGURE)  of
          PAWN: begin
             {/////////PAWN/////////////////////}
             {на клетку вперед}
             index := fN + delPawn;
             if(glPos[index] = 0) then
```

```
        begin
            {добавим в список}
            if pawnInQueen[index shr 4]  = 0 then begin
            {пешка не в ферзи}
            {////// добавим в конец списка ////}
               with buf^[count] do
               begin
                 __data := noEatMove.__data;
                 __N2   := index;
                 pNoEat^ := count;
                 pNoEat  := @__next;
               end;
               inc(count);
             {///////  end ////////////////////}
            end;
              if pawnStartY[fN shr 4] <> 0 then
            begin
              {если первый ход пешки}
              inc(index,delPawn);
              if(glPos[ index ] = 0) then begin
                {ход пешки через клетку}
            {////// добавим в конец списка ////}
               with buf^[count] do
               begin
                 __data := noEatMove.__data;
                 __N2   := index;
                 __desc := kFIrstPawn;
                 pNoEat^ := count;
                 pNoEat  := @__next;
               end;
               inc(count);
             {///////  end ////////////////////}
                end;
            end;
          end;
          {/////////PAWN//////////////////////}
           end; {PAWN}
      else begin {все фигуры, кроме пешек}
        isSimpleFig := simpleF[ f and CFIGURE ] <>  0;
        {получим строку приращений при движении}
        n := 0; {индекс приращения}
        pAdd := glAdd [ f and CFIGURE ];
        del  := pAdd^[n];
```

```
        while del <> 0 do
        begin
           index := fN + del;
           {линия}
           while glPos[index] = 0 do
           begin
              {свободная клетка - добавляем не взятие}
              {...}
              {////// добавим в список ////}
                 with buf^[count] do
                 begin
                   __data := noEatMove.__data;
                   __N2   := index;
                   pNoEat^ := count;
                   pNoEat  := @__next;
                 end;
                 inc(count);
               {///////  end ////////////////////}
              {///// end add no eat ///////}
              {если король или конь - линию не строим}
              if(isSimpleFig) then break;
              inc(index,del); {следующая клетка доски}
           end;
           inc(n);
           del := pAdd^[n];
           if count > size-10 then goto done;
        end; {while}
     end;{все фигуры}
end;  {case}
{  /////case//// }
  {если перемещения короля, и он не
перемещался ранее}
if ( (f and CFIGURE) = KING ) and
   ( (f and CMOVE)  = 0 ) then
begin
     {рокировки}
     {левая рокировка}
     canBeLeftCastl := false;
     canBeRightCastl := false;
     n := fN;
     if (glPos[n-1] = 0) and
        (glPos[n-2] = 0) and
        (glPos[n-3] = 0) and
```

```
    ((glPos[n-4] and CFIGURE) = ROOK) and
    ((glPos[n-4] and CMOVE) = 0) then
    canBeLeftCastl := true;
{правая рокировка}
if (glPos[n+1] = 0) and
    (glPos[n+2] = 0) and
    ((glPos[n+3] and CFIGURE) = ROOK) and
    ((glPos[n+3] and CMOVE) = 0) then
    canBeRightCastl := true;
if canBeLeftCastl or canBeRightCastl then
begin
  if(node^.__desc = kNotMove) or
    not InCheck(glPos[node^.__N2],
                node^.__N2,
                fN) then
   begin
     if canBeLeftCastl then begin
       index := fN - 2;
         {////// добавим в конец списка ////}
            with buf^[count] do
            begin
              __data := noEatMove.__data;
              __N2   := index;
              __desc := kLeftCastling;
              pNoEat^ := count;
              pNoEat  := @__next;
            end;
            inc(count);
          {//////  end ////////////////////}
     end;
     if canBeRightCastl then begin
       index := fN + 2;
         {////// добавим в конец списка ////}
            with buf^[count] do
            begin
              __data := noEatMove.__data;
              __N2   := index;
              __desc := kRightCastling;
              pNoEat^ := count;
              pNoEat  := @__next;
            end;
            inc(count);
```

```
                {/////// end ////////////////////}
            end;
          end;
        end;
    end; {рокировки}
      if count > size-10 then break;
    list := list^.__next;
  end;{while}
done:
  firstMove := noEat;
  countMoves := count;
END;{function}
```

В данной функции, так как все перемещения (кроме хода пешки через клетку) с одним флагом, создается шаблон. Например, берем мы перемещения ферзя или слона. Их может быть очень много, а они отличаются только полем N2, куда пошла фигура. Используем общий шаблон.

Обратите внимание на генерацию рокировок. В них выполняется проверка только на шах. Это несколько упрощенно. Используется функция `InCHeck`, аргументами которой являются последняя ходившая фигура противника, куда она пошла, и координаты нашего короля.

Рассмотрим для данной схемы функции, делающие ход и восстанавливающие позицию. Нужно учесть, что они должны выполняться только перед переходом в следующую позицию (`MakeMove`), поэтому страшного ничего нет. Если мы будем создавать развернутый описатель перемещения в генераторе ходов, будет значительно хуже, так как большая часть перемещений, как правило, не используется. Функция `MakeMove` при выполнении заполняет поля структуры `tagMove`. Они содержат всю информацию, необходимую для восстановления позиции функцией `UnMakeMove`. Вот эти функции и сама структура `tagMove`:

```
PTagMove = ^TTagMove;
TTagMove = record
    N1,N2,        {откуда и куда пошла}
    F,            {фигура}
    newF,         {значение на новом месте}
    eatF,         {взятая фигура}
    NEatF,        {координата взятой фигуры}
    desc:         {описатель перемещения}
          integer;
    Fig1,Fig2:PPiece; {элементы списка фигур: первый основной,
                       второй вспомогательный для рокировки}
    N11,N12,F11,newF11:integer; {откуда и куда пошла ладья при рокировке}

end;
```

```
PROCEDURE MakeMove(move: PMove; {описатель перемещения}
                   tag: PTagMove{вспомогательная структура}
                  );
VAR
 p:PPiece;
BEGIN
  with tag^ do begin
     Fig1  := @glFigBuf[ move^.__FigIndex ]; {элемент списка фигур}
     N1    := Fig1^.__N;           {стартовая координата}
     N2    := move^.__N2;          {куда пошла}
     F     := glPos[N1];           {фигура на старом месте}
     newF  := glPos[N1] or CMOVE; {фигура на новом месте}
     NEatF := N2;                  {координаты фигуры}
     eatF  := glPos[NEatF];        {взятая фигура}
     desc  := move^.__desc;        {тип перемещения}
     {если превращение пешки}
     if (desc and kPromote) <> 0 then
         newF := ( glNewF or (newF and (not CFIGURE)) ) or C_NEW_F;
     {если пешка взяла через битое поле}
     if (desc and kEnPassant) <> 0 then
     begin
       {координата фигуры не совпадает}
       if (F and CCOLOR) = glUpFColor then
          dec(NEatF, 16)
       else inc(NEatF, 16);
       eatF := glPos[NEatF];
     end;
     {выполним перемещение}
     glPos[N1] := 0;
     glPos[NEatF] := 0;
     glPos[N2] := newF;
     {изменение в списке фигур}
     Fig1^.__N := N2;
     FIg1^.__F := newF;
     {если рокировка - перемещаем ладью}
     if (desc and (kLeftCastling+kRightCastling)) <> 0 then
     begin
         {находим координату ладьи}
         if (desc and kLeftCastling) <> 0 then
         begin
            N11 := N1-4; {коор. ладьи}
            N12 := N11+3;
         end else begin
            N11 := N1+3; {коор. ладьи}
```

```
          N12 := N11-2;
        end;
        {находим элемент списка фигур ладьи}
        if (F and CCOLOR) = glUpFColor then
          p := glUpFList
        else
          p := glBotFList;
        while (p <> NIL)        and
              (p^.__N <> N11)   do
        begin
          p := p^.__next;
        end;
        Fig2 := p;{если p - 0 то !!!}
        F11 := glPos[N11];
        newF11 := F11 or CMOVE;
        glPos[N11] := 0;
        glPos[N12] := newF11;
        Fig2^.__N := N12;
        Fig2^.__F := newF11;
    end;
  end;
END;
{восстанавливает перемещение}
PROCEDURE UnMakeMove(tag:PTagMove);
BEGIN
    with tag^ do begin
      glPos[N2]     := 0;
      glPos[NEatF]  := eatF;
      glPos[N1]     := F;
      Fig1^.__N     := N1;
      Fig1^.__F     := F;
      if (desc and (kLeftCastling+kRightCastling)) <> 0 then
      begin
        glPos[N12]  := 0;
        glPos[N11]  := F11;
        Fig2^.__N   := N11;
        Fig2^.__F   := F11;
      end;
    end;
END;
```

Позиция 16 × 16 во всех функциях содержится в `glPos`. Данный пример является учебным, но код вполне работоспособен. Основная польза его мне видится в том, что при написании своей программы желательно отталки-

ваться от каких-то идей, а не начинать с нуля. Эффективная генерация перемещений в шахматах — сложная задача и не имеет одного решения.

В реальной программе потребуется еще функция, генерирующая сразу все перемещения: и взятия, и не взятия. Ее можно написать, просто объединив эти две функции. Все зависит от конкретной реализации. Хочется обратить внимание на один тонкий момент, связанный с использованием списков фигур. Если фигуры нет на доске (например, коня), то пустой элемент списка фигур может случайно указать на другого коня, и перемещения для оставшегося коня будут сгенерированы два раза. Чтобы это избежать, можно при инициализации позиции для парных фигур использовать дополнительный бит флага. У нас остался свободным бит 5, который и можно использовать для этой цели. Вот пример стартовой позиции в игре с учетом бита флага парности фигур:

```
 const glStartPos:array[0..7][0..7] of byte = (
(ROOK+CBLACK+32,KNIGHT+CBLACK+32,BISHOP+CBLACK+32,QUEEN+CBLACK+32,KING+CB
LACK,BISHOP+CBLACK,KNIGHT+CBLACK,ROOK+CBLACK),

(PAWN+CBLACK+32,PAWN+CBLACK+32,PAWN+CBLACK+32,PAWN+CBLACK+32,PAWN+CBLACK,
PAWN+CBLACK,PAWN+CBLACK,PAWN+CBLACK),

(0,0,0,0,0,0,0,0),
(0,0,0,0,0,0,0,0),
(0,0,0,0,0,0,0,0),
(0,0,0,0,0,0,0,0),
(PAWN+CWHITE+32,PAWN+CWHITE+32,PAWN+CWHITE+32,PAWN+CWHITE+32,PAWN+CWHITE,
PAWN+CWHITE,PAWN+CWHITE,PAWN+CWHITE),

(ROOK+CWHITE+32,KNIGHT+CWHITE+32,BISHOP+CWHITE+32,QUEEN+CWHITE+32,KING+CW
HITE,BISHOP+CWHITE,KNIGHT+CWHITE,ROOK+CWHITE)

);
```

Если на доске будет второй ферзь (например, у белых), то он уже появится без флага, и коллизий не будет. Если третий... Но это маловероятно, можно и на этот счет что-нибудь придумать.

ПРИЛОЖЕНИЕ 3

Замечания по технике программирования

Профессионал может пропустить эту главу. В ней содержатся некоторые тривиальные вещи, которые, тем не менее, полезно знать.

Стековые переменные

Переменные могут быть стековыми или глобальными. Когда начинается выполнение функции, в стек программы заносится адрес возврата, происходит переход на начало машинных инструкций данной функции. Вот типичная сигнатура функции:

```
function Foo(arg1,arg2:integer):integer;
var
 tmp1,tmp2:integer;
begin
    {...}
end;
```

Стек программы начинается в верхних адресах, и при каждом занесении туда нового значения (равного размерности регистра, аккумулятора AX или EAX) указатель вершины стека (регистр SP или ESP) уменьшается, и стек смещается вниз. Когда значение извлекается из стека, указатель стека, наоборот, увеличивается. Занесение в стек — ассемблерная инструкция `push`, извлечение — `pop`. Переход на начало машинных инструкций функции выполняется командой `call` (имя метки).

Команда `call` автоматически заносит в стек адрес возврата из функции, и когда тело функции завершилось, выполняется инструкция `ret`, которая извлекает из стека адрес возврата.

В теле функции могут быть объявлены переменные и могут быть аргументы функции. В приведенном примере `arg1`, `arg2` — аргументы функции, `tmp1`,

tmp2 — переменные внутри функции. Мы должны понимать, что и аргументы, и переменные располагаются в стеке. Это — стековые переменные. Аргументы функции являются такими же стековыми переменными, что и переменные, объявленные внутри функции. Аргументы функции — такие же переменные, просто инициализированные при вызове функции. В приведенном примере можно также присвоить значение аргументам arg1, arg2, как и переменным tmp1, tmp2.

```
function Foo(arg1,arg2:integer):integer;
var
 tmp1,tmp2:integer;
begin
    arg1 := 0;
    tmp1 := 0;
    {...}
end;
```

Когда тело функции завершилось, ее стековые переменные перестают существовать. Типичной ошибкой является возвращение из функции адреса ее стековой переменной. Этой переменной уже нет, и пространство стека используется для других целей.

```
type PInt = ^integer;
function Foo:PInt;
var
 tmp:integer;
begin
   tmp := 0;
   Foo := @tmp;
end;
```

Это ошибка. Если мы хотим вернуть из функции адрес переменной, такая переменная должна быть статической и существовать вне стека. На языке C это спецификатор static, на Паскале — **const**.

```
function Foo:PInt;
const tmp:integer = 0;
begin
    Foo := @tmp;
end;
```

```
int *Foo(void)
{
  static int tmp = 0;
  return &tmp;
}
```

Инициализация статических переменных выполняется один раз при запуске программы. Когда функция `Foo` начинает выполняться, значение `tmp` уже 0. Если мы присвоим `tmp` при выполнении функции некоторое значение, то при следующем вызове функции `tmp` будет уже с новым значением. Это эквивалентно объявлению `tmp` вне функции обычной глобальной переменной, только ее область видимости ограничена данной функцией.

```
static int tmp = 0;
int *Foo(void)
{
  return &tmp;
}
```

Если мы в функции присвоили `tmp` некоторое значение, оно сохранится при завершении функции.

```
int *Foo(void)
{
  static int tmp = 0;
  tmp = tmp + 1;
  return &tmp;
}
```

Память под стековые переменные выделяется мгновенно. Не так обстоит с их инициализацией. При каждом вызове функции инициализация стековых переменных происходит заново.

```
void Foo(void)
{
   int array[] = {0,1,2,3,4,5,6};
   //...
}
```

При каждом вызове функции массив `array` будет инициализироваться заново. Если он не изменяется, лучше написать:

```
void Foo(void)
{
   static int array[] = {0,1,2,3,4,5,6};
   //...
}
```

Тогда массив будет инициализирован компилятором, и при входе в тело функции уже будет содержать значения, так как это не стековая переменная.

Размер стека программы должен быть достаточным для размещения стековых переменных. Если мы используем рекурсию, то должны четко пред-

ставлять, сколько у нас рекурсивных вызовов функций, и хватит ли стека для этого.

Операции умножения

Это старая заезженная тема. Существуют по сей день люди, не видящие в программе никаких достоинств, кроме отсутствия операций умножения (деления). Чтобы написать ассемблерный код, который бы выполнялся быстрее, чем созданный хорошим компилятором, нужно быть профессионалом высочайшего уровня. Сейчас не нужно хранить переменные в регистрах и заниматься многими другими нудными вещами. Но некоторые особенности полезно учитывать.

Небольшие операции умножения только разгоняют процессор. Количество тепла, им выделяемого при этом, правда, возрастает.

Вот типичный пример. Можно объявить массив как одномерный, а можно как двумерный:

```
data:array[0..7][0..7] of byte;
data:array[0..63] of byte;
```

Когда компилятор обрабатывает двумерный массив, то реальный адрес вычисляется по формуле:

```
N = Y*8 + X;
```

Старый компилятор бы изобразил это по-другому:

```
N = (N << 3) + X;
```

Небольшие операции умножения только разгоняют процессор, и если мы не пишем системного кода, можно руководствоваться только соображениями удобства.

Если у нас не байтовый массив, а некоторый массив с большим размером элементов, то можно призадуматься, стоит ли перегружать процессор.

```
type TRec = record
    {...}
end;

var data:array[0..2000] of TRec;
```

При доступе к каждому элементу такого массива в критичной по скорости программе лучше придумать что-нибудь другое, чем

```
data[N]
```

Лучше всего, чтобы размер структуры был кратен степени двух, и адрес элемента можно было получить битовым сдвигом.

Есть и анекдотичные примеры. Например, у нас в программе используется 64-битовый хеш-индекс. Чтобы получить номер хеш-ячейки, нужно иметь остаток от деления этого неправдоподобно огромного числа на конечный размер хеш-таблицы:

```
type
THItem = record
    {...}
end;
PHItem = ^THItem;
const MAX_ITEM = 100000;
var hashTable:array[0..MAX_ITEM-1] of THItem;

function Look(index:U64):PHItem;
begin
   Look := @hashTable[ index mod MAX_ITEM ];
end;
```

Гораздо эффективнее делать размер хеш-таблицы таким, чтобы в двоичном представлении это было что-то вроде

```
10000000
```

Тогда, чтобы получить остаток от деления любого сколь угодно огромного числа на размер хеш-таблицы, мы должны просто провести поразрядную операцию AND с размером таблицы на единицу меньше:

```
    10000000
-       1
   =01111111
```

Приведенный пример в следующем листинге:

```
type
THItem = record
    {...}
end;
PHItem = ^THItem;
const MAX_ITEM = (1 shl 18);
var hashTable:array[0..MAX_ITEM-1] of THItem;

function Look(index:U64):PHItem;
begin
   Look := @hashTable[ index and (MAX_ITEM-1) ];
end;
```

ПРИЛОЖЕНИЕ 4

Описание прилагаемого компакт-диска

Каталог PROGRAMS

- Игра Mirage — непобедимый чемпион России. Чемпионаты, правда, уже не проводятся. Считать может до 5 минут.

Каталог SOURCE (наиболее известные открытые исходники шахматных программ)

- crafty — Классика. С опубликации этих исходников и наступил конец эре сокрытия информации в шахматном программировании.
- gnu000 — Самый простой и доступный известный мне код шахматной программы. Все последующие версии шахмат GNU используют его в качестве основы.
- gnu3.01 — Подробно задокументированный код.
- gnu5.00 — Bitboard вариант Gnuchess.

Каталог Winboard

Вариант Xboard для Windows. Сам шахматный движок представляет собой консольное приложение, которое через стандартный ввод/вывод связывается с Winboard. Протокол связи хорошо задокументирован.

По умолчанию в Winboard стоит Gnuchess4. В диалоговом окне можно задать другой движок. Для этого его можно скопировать в каталог Winboard и набрать в стартовом окне имя файла.

В каталоге уже имеется несколько движков, и для каждого создан командный файл с расширением bat.

Вы можете для простоты запустить соответствующий командный файл и поиграть в шахматы, скажем, с программой Smarthink, которая сейчас позиционируется как самая сильная российская программа.

Программу автора тоже можно запустить под Winboard, несмотря на то что она имеет свою графическую оболочку.

Winboard удобно использовать для матча между 2 программами.

Для более подробной информации по Winboard смотрите соответствующий файл справки. Интерфейс программирования описан в faq.html.

Для быстрого начала работы запустите файл winboard.exe.

Каталог AUTHOR (программы, написанные автором книги)

- Qchess — Демо-версия шахматной программы. Использует развернутый bitboard и выборочный поиск. Формирует малое дерево игры и уточняет его. Имеет сложную оценочную функцию. Учитываются: близость фигур к центру, безопасность короля, активность в атаке, давление по фронту и пр. Некоторые окончания игры представлены отдельными оценочными функциями. Основная глубина перебора небольшая (6), но некоторые варианты просматриваются значительно глубже.

  Используемые эвристики: хеш-таблица, таблица PV, killer, эвристика истории. Программа использует таблицу самообучения для запоминания проигранных позиций и их оценок.

- Checkers — Демо-версия русских шашек. Bitboard-программа. Для оптимизации перебора использует основные шахматные эвристики. К недостаткам следует отнести отсутствие дебютного справочника. Умеет играть окончания с дамками.

- Krnil — Демо-версия игры "Крестики-нолики".

- Tchess — Демо-версия шахматной программы. Подробное описание приводится далее.

Описание модуля Tchess

Программа написана на языке Turbo Pascal 7. Системные требования: MS DOS 6.22, адаптер VGA.

Это простая, но функциональная программа. При ее написании главной целью было продемонстрировать основные идеи. Позиция представлена байтовым одномерным массивом [0..255]. Это эквивалентно двумерному массиву 16 × 16. Игровое поле 8 × 8 находится в центре. Края заполнены флагом выхода за пределы доски. Размер 16 выбран потому, что это мини-

мальное число больше 8 и кратное степени 2. Кроме того, для детектора шахов тоже используется массив 16 × 16, что упрощает перевод координаты короля и атакующей фигуры из координат одного массива в другой. В игровом поле 8 × 8 каждый байт может быть нулевым, если нет фигуры, или ненулевым, если он занят фигурой или флагом выхода за пределы.

Байт фигуры содержит упакованный код цвета фигуры и номер (индекс) описателя фигуры в массиве списка фигур. Чтобы получить более подробную информацию о фигуре, нужно просмотреть соответствующий описатель в массиве списка фигур. В процессе расчета мы работаем, в основном, со списками фигур. Они инициализируются перед началом вычисления. Сам массив игры используется для определения атаки дальнобойной фигуры. Знания, что в клетке наша фигура или фигура противника, в большинстве случаев оказывается достаточно.

Описатель перемещения содержит всего два поля: откуда и куда пошла фигура. Больше ничего. Функция `MakeMove` на основании описателя перемещения заполняет служебную структуру `tagMove`, которая содержит подробную информацию о перемещении. Тип хода определяется по ходу дела в `MakeMove`. Большинство ходов являются простыми перемещениями, и это практически не отнимает времени.

Основную сложность представляют функции `MakeMove`, `UnMakeMove` и `gGetCurrPos`. Можно и не вникать в устройство этих функций, а только представлять себе, что они делают.

Функция `MakeMove` изменяет клетки поля игры и списки фигур, а также заполняет структуру `tagMove`. Она определяет тип хода, вносит информацию, необходимую для восстановления позиции, определяет приращения оценки при ходе, хеш-ключ хода, применив к которому операцию XOR с ключом позиции можно полvчить хеш-ключ после перемещения.

Функция `UnMakeMove`, на основании заполненного функцией `MakeMove` развернутого описателя перемещения, восстанавливает позицию. Для простых перемещений это несложно. Для взятий пешки на проходе координаты взятой пешки не совпадают с целевыми координатами взявшей пешки. Для рокировок нужно поставить ладью и короля на свои места.

Функция `gGetCurrPos` возвращает текущую позицию в игре. Сама игра представлена строкой перемещений, ограниченной реальным числом (например, 400). Для того чтобы получить текущую позицию в игре, надо инициализировать стартовую позицию в игре и списки фигур. Потом нужно проиграть всю строку, делая перемещения в массиве игры и корректируя соответствующим образом списки фигур. Если фигура изменила положение, то в старую ячейку массива игры запишется 0, а значение из нее будет перенесено в новую (для простых перемещений этого достаточно). Также в соответствующем элементе списка фигур изменятся координаты ходившей фигуры.

В результате мы получим текущую позицию и списки фигур. В каждом элементе списка фигур есть поле, отражающее, перемещалась ли фигура хоть один раз. Это нужно для корректного отслеживания возможности рокировки (если король или ладья ходили, то, хоть они и на старых местах, рокироваться уже нельзя). Если при получении текущей позиции фигуру взяли, списки нужно уплотнить, чтобы не было лишних расходов при просчете. Списки фигур содержатся в двумерном массиве:

```
PieceTab:array [cWhite..cBlack,0..15] of TPiece;
```

Размер `TPiece` четыре байта, индексация и доступ к элементу быстрый. Фигуры расположены по убыванию веса.

Переменная `PawnNo` указывает на индекс последней фигуры. Переменная `KnightNo` указывает на индекс последней фигуры старше пешки. Функция `gGetCurrPos` уплотняет списки фигур (если были взятия в строке игры). Если было превращение пешки, то `gGetCurrPos` сортирует список по убыванию и корректирует переменные `PawnNo` и `KnightNo`. Это довольно нудно и неинтересно. Нужно лишь знать, что `gGetCurrPos` возвращает текущую позицию в игре и соответствующие ей списки фигур.

Функция просчета всего одна. Тем не менее имеет 4 этапа. Начнем с конца.

4) По истечении глубины устанавливается переменная `CaptureSearch`. Дальше будут рассмотрены только взятия, и alpha перед просчетом в узле повышается текущей оценкой.

3) Поиск шахов: на последних двух полуходах устанавливается переменная `CheckSearch` и максимум не ограничивается только у шахов. Все остальные перемещения подрезаются статической оценкой. Когда `CheckSearch` перейдет в `CaptureSearch`, поиск шахов прекращается.

2) Статический поиск: функция `Cut` занижает максимум (beta) статической оценкой после хода у всех ходов, кроме тех, которые продлеваются. Если максимум будет меньше или равен alpha, ход подрезан, его оценка принимается равной alpha.

1) Поиск полной ширины никаких отсечений. Глубину поиска полной ширины можно выставить в функции `Cut`. В настоящем коде его нет, т. е. полного перебора нет вообще.

В корне дерева перебора используются итеративные углубления. Глубина поиска сначала 3, потом 4, 5, 6, 7, 8, ..., пока не исчерпан лимит времени. Минимальная глубина 10. После каждой итерации главная строка изменения извлекается и помещается в таблицу PV. Эти строки при следующей итерации будут рассмотрены первыми и не будут подрезаны статической оценкой.

Хеш-таблица содержит только лучшие ходы и имеет вспомогательное значение. Используется алгоритм Principal Variation.

Так как программа написана на Turbo Pascal 7, и компилятор является 16-разрядным, при перенесении на 32-разрядный компилятор все структуры должны быть упакованными (размер структуры соответствует реальному суммарному количеству байтов во всех переменных).

В программе использованы некоторые вспомогательные файлы. Знать их структуру для понимания работы программы совершенно необязательно.

Файл масок содержит байтовые маски шахматных фигур. Каждая фигура представлена массивом 50 × 50. Единица соответствует черному цвету, ноль белому. Всего масок 24: 6 черных фигур и 6 белых в черной и белой клетке.

Файл дебютной библиотеки — простой массив строк перемещений. Его структура описана в файле lib.pas. Каждый дебют представлен строкой движения фигуры. Для последующего объединения файлов и графического драйвера с исполняемой программой дано представление с расширением obj и компоновка с программой как внешней процедуры.

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Х

Ц

Э

ПРОГРАММИРОВАНИЕ ШАХМАТ И ДРУГИХ ЛОГИЧЕСКИХ ИГР

МАШИНА ИГРАЕТ В ШАХМАТЫ – РАЗВЕ ЭТО НЕ УДИВИТЕЛЬНО?!

КОРНИЛОВ ЕВГЕНИЙ НИКОЛАЕВИЧ, программист, выпускник Ленинградского института авиационного приборостроения (ЛИАП). Автор многочисленных шахматных и других игровых программ, данной темой занимается более 10 лет.

Человек, владеющий некоторыми навыками программирования, но не знакомый с созданием компьютерных логических игр, найдет в этой книге все необходимое для создания программ для игры с компьютером (от простейших "крестиков-ноликов" до шахмат). В книге обобщен опыт, накопленный в данной области, и приводятся примеры практической реализации известных и испытанных приемов программирования (генерация и сортировка перемещений, эвристика недействительного перемещения, схема форвардного отсечения и др.).

Методики и алгоритмы, применяемые при программировании шахмат, могут с успехом использоваться при разработке других компьютерных игр: от шашек до преферанса. Книга поможет начинающим написать свою первую программу логической игры, а профессионалы в этой области найдут здесь справочную информацию, новые идеи и альтернативный взгляд на некоторые вопросы. Примеры программ приведены на языках C++ и Pascal.

КАТЕГОРИЯ:
ПРОГРАММИРОВАНИЕ ИГР

+CD Компакт-диск содержит наиболее известные открытые коды шахматных программ, а также исходные тексты программ, написанных автором.

ISBN 5-94157-497-5

БХВ-ПЕТЕРБУРГ 190005, Измайловский пр., 29
E-mail: mail@bhv.ru Internet: www.bhv.ru
Тел.:(812) 251-4244 Факс:(812) 251-1295