Шахматное программирование

2 іюля 2014 г.

Оглавление

Глава 1

Введение

Эта работа преследует несколько целей.

Во первых, служить введением в мир шахматного программирования. Бороздя интернет в поисках полезных материалов как профессионалов, так и любителей, автор с прискорбием отметил, что если первые делают упор на математическое изложение, то вторые, как правило, ограничиваются поверхностным обзором. Эта работа - всего лишь попытка переработать материал, выразив его в максимально простой форме.

Во-вторых, эта работа является своего рода дневником, отчетом о всем пройденном мною пути в шахматном программировании и, в частности, о движке, который я поставил целью себе написать. Как правило, кропотливый процесс написания всегда остается за кадром, являясь всегда побочным эффектом конечного результата — о нем не говорят и его не документируют. Понятный сперва код обрастает, как снежный ком, наслоениями эвристик для увеличения силы игры в ущерб читаемости и вот уже никому, кроме авторов, неизвестны все тонкости алгоритмов и все их нюансы. А вскоре и сами авторы затрудняются сопровождать программу ввиду ее сложности.

Усугубляет задачу тот факт, что для того, что бы побороть экспоненциальную сложность алгоритмов, часто появляется заманчивое желание отвоевывать глубину перебора используя низкий к уровню машины подход к программированию и организации программы. Использование указателей везде, где можно для возвратов сложных структур данных из функций, соблазн не удалять объекты во время перебора, плодя гигабайты ненужного мусора и тому подобные грязные трюки. Паранойя усиливается, когда начинаешь искать подвох в переменных, определяемых внутри функций[1]. А об STL[2] я вообще молчу... мягко говоря, не все алгоритмы и структуры данных оптимизированы с параноидальной агрессивностью.

Опять же, как говорил умный человек, преждевременная оптимизация — корень всех зол, поэтому сосредоточившись собственно на самой задаче, попробуем приоткрыть завесу над миром компьютерных шахмат. Вооружившись здравым смыслом и компилятором, начнем.

Итак, из чего состоит шахматная программа? В общем, это всего несколько модулей, которые интегрируются друг с другом:

1. Представление доски
2. Представление фигур

[1] Думали ли вы как можно поменять значения двух переменных не используя временную третью?
[2] STL — стандартная библиотека шаблонов языка C++

3. Генератор ходов
4. Переборочные алгоритмы + эвристики
5. Функция оценки
6. Взаимодействие с пользователем / GUI

Написать эти модули без надлежащего плана довольно сложно. Нет, вы можете написать вполне удачную реализацию для представления позиции и генератора ходов, но неожиданно обнаружите, что они не вполне эффективно работают вместе. Так, задержка в тысячную долю секунды для генерации всех ходов в позиции не заметна глазом, но когда вы напишите простой переборочный алгоритм, который мы рассмотрим позже, то обнаружите, что эти задержки ощутимо тормозят поиск.

Решение кроется в хорошей интеграции между этими модулями. Порой, лучше сделать избыток представления для доски или чуть упростить генератор ходов, жертвуя общностью, но добиться того, что взаимная работа всех модулей будет работать быстрее. К сожалению, добиться этого, не подсматривая в чужой код можно только методом проб и ошибок. И то нет гарантии, что вы написали качественный код до того, как напишите переборочный алгоритм.

Подобная интеграция, однако, лишает модули независимости и каждый из них становится зависим от конкретной реализации. Так, становится невозможным перенос переборочной функции из программы в программу и каждая шахматная программа становится монолитным произведением искусства, уникальным и неповторимым. Удивительным является тот факт, что хорошие идеи и алгоритмы из других программ могут быть менее эффективны или даже вредны в вашей программе. Пусть это вас не прельщает: скорее всего, если это случилось в вашей программе, то ищите ошибку в коде.

За более, чем 50 лет развития шахматного программирования, множество людей пытались придумать способ увеличить силу программ. Были перепробованы тысячи алгоритмов и их улучшений, придумать что то новое в этой области чрезвычайно сложно. Многие успешные идеи пришли из математики, например техника магичеких битбоардов, упрощающая и ускоряющая генератор была создана именно так. Но и тем, кто далек от математики есть что принести в теорию - метод проб и ошибок очень эффективен и здесь.

К тому же необходимо учитывать и то, что авторы программ не всегда спешили открывать найденные удачные схемы. Многие из этих методов открывались повторно другими людьми и со временем становились достоянием общественности.

Глава 2

Соглашения, относительно стиля кодирования

Глава 3

Наивный способ и его недостатки.

Итак, шахматные правила известны, программирование освоено и есть непреодолимое желание научить компьютер играть в шахматы. С чего же мы начнем ?

Прежде всего, еще до того, как мы приоткроем читателю сердце любой программы - переборочный алгоритм и функцию оценки - необходимо условиться о представлении шахматной позиции в памяти компьютера и кодировании правил игры на машинном языке. Что есть шахматная позиция?

Это доска 8 на 8 клеток в каждой из которых может не быть, а может и быть одна из семи фигур, и мы, воспользовавшись этой аналогией, можем представить доску в виде массива 8 на 8 типа int.

```
int position[8][8]
```

где каждой фигуре сопоставлено определенное число, ее представляющее. Тогда в программе будут логичными макросы вида:

```
#define ROOK 5
```

и проверки вида:

```
if( position[y][x] == ROOK )
{
   ...
}
```

Белые и черные фигуры можно отделять друг от друга, например, заменив определение позиции на

```
#define WHITE true
#define BLACK false

struct piece
{
   bool color;
   int dignity;
}
```

```
piece position[8][8];
```

Наивный метод представления доски, не является, однако, самым лучшим. Когда мы доберемся до программирования генератора ходов, везде и всюду будут появляться конструкции вида:

```
if(x < 0 || x >= 8) continue;
```

являющиеся проверками, не вышли ли мы за границы массива. Кроме того, сам проход по двумерному массиву не является оптимальным, так как обращение к элементу вида

```
position[y][x]
```

неявно делает вычисление $8 \cdot y + x$ для обращения к нужному месту в памяти.

Чтобы решить эту и другие проблемы были изобретены разные ухищрения, как, например, представлять позицию, как одномерный массив из 64 элементов, или использовать двумерный массив[1], где специальные боковые места зарезервированы для служебных целей — именно определять границы доски. Эти методы не будут в полной мере рассмотрены нами ввиду того, что их немало. В моей программе я таки использую двумерный массив с некоторыми добавлениями, но особенности генератора ходов не предполагают проверки выхода за границы массива и использование его приемлемо.

[1] Или вообще не использовать массивы. А представлять доску списком фигур (с координатами) или набором битбоардов

Глава 4

Битбоарды

Несмотря на то, что метод, описанный выше и является наиболее общим способом представления доски в памяти, и несмотря на то, что его успешно используют некоторые из очень сильных программ, подобное представление все реже и реже встречается.

Идея заключается в том, что компьютерные процессоры имеют встроенные сверхбыстрые побитовые операции AND, OR, NOT и XOR. Прибавим к тому тот факт, что 64 битные процессоры и 64 битные операционные системы становятся стандартом в современном мире. Напомним читателю, что каким-то неведомым стечением обстоятельств шахматная позиция также содержит 64 поля, и все преимущества битбоардного преимущества станут ясны.

Например, для того, чтобы узнать, может ли конь по правилам игры ходить с поля c3 на любое другое поле, стоит лишь к числу 345879119952, которое в бинарном виде выглядит как

$$0000000000000000000000000101000010001000000000001000100001010000(bin)$$

применить операцию AND с другим числом, например, для поля $d5$ это будет 68719476736 или

$$0000000000000000000000000010000000000000000000000000000000000000(bin)$$

Все станет еще понятней, если мы эти числа перегруппируем в блоки по 8 так, что каждый бит визуально встанет на свое “шахматное” место.

```
00000000        00000000
00000000        00000000
00000000        00000000
01010000        00010000
10001000        00000000
00000000        00000000
10001000        00000000
01010000        00000000
```

Тогда в результате операции AND получится число, отличное от нуля, только в том случае, если ход возможен. Всего-навсего для каждого поля и для каждой фигуры надо хранить соответствующие битбоарды и правильным образом с ними работать. Это и другие ухищрения могут превратить программу в настоящего монстра, с которым могут справиться лишь единицы из людей. Недостаток битбоардов заключается в том, что программа становится плохо

читаемой, так как часть логики переносится на неудобный для восприятия уровень двоичных чисел.

Можно лишь добавить, что для работы с битбоардами вида 2^n то есть, именно теми, что нужны для представления одной позиции и быстрого перехода между ними, можно использовать процессорные команды *shl* или *shr* (или команды сдвига влево/вправо), эквивалентами которых в языке C++ являются $\ll$ и $\gg$.

Программы, которые придерживаются битбоардной философии до конца, используют множество трюков над битами и не всегда они делают код прозрачным. Стандартной проблемой, плохо решаемой с битбоардами, это проблема определения количества ненулевых бит в 64-битном числе(например, мы храним битбоард для представления белых пешек и хотим узнать их количество). Невозможно сделать это в один такт процессора и в худшем случае необходимо сканировать число побитово[1].

Однако и это еще не все. В какой то момент мы захотим описать шахматные правила в полном объеме. И тут битбоарды оказываются полезны. Существуют методики, позволяющие закодировать весь объем правил для всех фигур на языке битов, что позволит быстро проверять на валидность того или иного хода[2]. Эти методы поистине вершина человеческой мысли, однако их использование порождает трудно читаемый код, поэтому я от них отказался.

Хоть битбоарды используются повсеместно, метод выбранный мной для отображения шахматных правил несколько иной. Я использую списки ходов в представлении массива фигура-доска. Этот метод будет мной рассмотрен отдельно и, на мой взгляд он не медленней, чем общепринятые методы[3].

Всякий раз когда я захочу использовать битбоардные вычисления, я буду использовать тип

```
typedef unsigned long long int U64;
```

А когда переменная всего лишь хранит число в своем обычном смысле, подойдет и простой int. На 32 битной системе преимуществом быстродействия обладает int, который реализуется, как 32 битное число, а unsigned _int64 эмулируется с помощью двух 32 битных чисел, тогда как на 64 битной системе все происходит с точностью наоборот. 64 битный тип является родным, а следовательно наиболее быстрым, а int может быть как 32 битным, так и 64 битным, в зависимости от архитектуры компьютера, настроек компилятора и т.д.

[1] С новой командой popcnt, добавленной расширением SSE4.2, это можно сделать

[2] Точнее, быстро генерировать полный набор всех возможных ходов за несколько быстрых операций

[3] Однако имеются и недостатки: т.к. битбоарды полезны для оценочной функции их в любом случае придётся добавить, но они уже не будут использоваться для генерации ходов

Глава 5

Представление фигур

Итак, для того, чтобы представить фигуру в моей программе, мне нужно 5 битов (подойдет простой int) со следующими условностями : первые 3 бита представят тип фигуры, а 2 последующих - ее цвет

```
#define PAWN   1    //00 001
#define KNIGHT 2    //00 010
#define BISHOP 3    //00 011
#define ROOK   4    //00 100
#define QUEEN  5    //00 101
#define KING   6    //00 110

#define WHITE 8     //01 000
#define BLACK 16    //10 000

#define COLOR_BITS 24 //11 000
#define DIGNI_BITS 7  //00 111
```

Одно число хранит полную информацию о фигуре и она может быть получена используя простые битовые операции.

- Установить цвет фигуры:

  ```
  info = (info & DIGNI_BITS) | color;
  ```

- Установить тип фигуры:

  ```
  info = (info & COLOR_BITS) | digni;
  ```

- Установить цвет и тип фигуры:

  ```
  info = color | digni;
  ```

- Узнать если цвет фигуры color:

  ```
  ((info & COLOR_BITS)==color);
  ```

- Узнать если тип фигуры digni:

```
((info & DIGNI_BITS) == digni);
```

- Если фигура цвета color и типа digni

```
(info == (color | digni));
```

- Узнать какой цвет фигуры:

```
(info & COLOR_BITS);
```

- Узнать какой тип фигуры:

```
(info & DIGNI_BITS);
```

Интуитивная работа достигается работой с предопределенными константами[1], где digni может принимать значения PAWN, KNIGHT, BISHOP, ROOK, QUEEN, KING, а color - WHITE и BLACK.

Напрашивается вопрос: зачем использовать одно число и извлекать из него составляющие, если можно, введя два числа, одно для типа фигуры, а второе для цвета, избежать ненужных битовых операций. Нет, дело не в экономии памяти, а в том, что в программе повсеместно используются таблицы фигура-доска, которые будут нами рассмотрены далее. Очень удобно использовать в подобных таблицах как индекс именно объединенные значения. Благо, их всего 32 комбинации[2]. Если бы мы выбрали хранить цвет и тип отдельно, именно здесь появились бы битовые объединения[3]. Так что вопрос эффективности остается открытым. Я же выбрал наиболее удобный способ.

[1] На самом деле были определены не константы, а макроопределения

[2] Если сжать данные покомпактней, то таких значений будет всего-лишь 12

[3] Или двумерные массивы, что тоже не очень хорошо

Глава 6

Генератор ходов

Ну, вот и подобрались мы к одному из самых критичных, в плане скорости, моменту. Образно говоря, нам необходимо уметь перебирать для заданной позиции все возможные ходы с точки зрения шахматных правил. Легко сказать! Ибо шахматные правила не обладают очевидной математической общностью для компактного описания, несмотря на свою видимую простоту[1].

Так, все фигуры, кроме пешек, имеют полную симметрию для белых и черных, а также могут бить на те же поля, что и ходить, не перепрыгивая при этом другие фигуры. Пешки же имеют явно выраженную черно-белую асимметрию в поведении, да, к тому же, некоторые правила : ход на два поля, взятие на проходе, умение превращаться в фигуры и разные ходы для взятий и простых ходов делают алгоритмизацию сущим адом для программиста, стремящегося экономить каждое сравнение, каждое обращение к переменной.

Наивным способом генерации этих правил являются простые сравнения с помощью циклов. Пробегаются четырьмя вложенными циклами все комбинации для ходов

```
for( unsigned int from_x = 0; from_x<8; ++from_x )
   for( unsigned int from_y = 0; from_y<8; ++from_y )
      for( unsigned int to_x = 0; to_x<8; ++to_x )
         for( unsigned int to_y = 0; to_y<8; ++to_y )
               if ( isValidMove( from_x, from_y, to_x, to_y ) )
               {
                  ...
               }
```

и для каждой такой комбинации при наличии на позиции from фигуры вызывается соответствующая функция *isValidMove*, возвращающая для данного хода true или false, в зависимости от легитимности хода. Это очень медленный способ, от которого я отказался. Программа с таким генератором ходов не покажет сколь-нибудь удовлетворительную игру, и вряд ли будет перебирать более миллиона позиций в секунду при переборе.

Идея, которой я воспользовался, очень проста — будем для каждой позиции и фигуры хранить специальным образом построенный связный список всех легитимных ходов. В программе будет определен массив, который инициализируется при запуске программы

```
#define BOARD_SIZE 8
```

[1] Правила шахмат красивы и понятны только для человека, а их алгоритмизация очень сложна

```
#define PIECE_ID_MAX 32 //11 111

move_list all_comb[PIECE_ID_MAX][BOARD_SIZE][BOARD_SIZE];
```

Тогда для хода белого коня, например, при вызове all_comb[WHITE | KNIGHT][1][0] будет получен связный список всех легитимных ходов с поля $g1$. Для фигур, которые не являются пешками, нет разницы между белыми и черными списками — списки ходов будут одинаковыми. Однако для пешек ситуация другая.

Первый индекс в этом массиве содержит то самое число, которое мы определили при определении фигуры. 3 бита для типа фигуры и 2 для цвета дадут в общем 5 бит. Отсюда и 32 возможные комбинации. Второй и третий индекс представляет позицию на доске, в шахматной нотации

Второй: "a"=7 "b"=6 "c"=5 "d"=4 "e"=3 "f"=2 "g"=1 "h"=0

Третий: "1"=7 "2"=6 "3"=5 "4"=4 "5"=3 "6"=2 "7"=1 "8"=0

Идея использовать таблицы фигура-поле[2] весьма распространена в шахматном программировании. Это быстрый и элегантный способ получить во время исполнения нужную информацию по некоторым критериям, не используя циклы и условия. Мы еще вернемся к этому способу при исследовании оценочной функции и техники магических битбоардов. Сейчас же уясним другое чрезвычайно важное свойство подобных таблиц: они строятся до запуска основного переборочного алгоритма поиска!

Идея заключается в следующем. Во время поиска затратно проверять легитимность того или иного хода для генератора ходов. Нам бы потребовались функции пестрящие условиями и циклами, с проверкой на выход за пределы доски и т.д. И все это может происходить сотни миллионов раз за один поиск. Вместо этого затратного способа во время запуска программы строится информативная структура и далее без изменений используется в программе.

Однако и это еще не все. Связный список, о котором идет речь представляет собой несколько модифицированный вариант классического одностороннего связного списка. Отличие в том, что помимо ссылки на следующий по списку элемент, хранится еще ссылка на следующий “специальный” элемент.

Предположим, мы хотим получить все ходы слона, стоящего на каком-то поле. У слона есть, в общем случае, четыре диагонали, куда он может ходить. Если на позиции, принадлежащей любой из этих диагоналей, есть любая другая фигура, то дальнейшее сканирование этой диагонали бессмысленно — это невозможный ход, и поиск следует продолжать уже со следующей диагонали. Поэтому есть смысл хранить для каждого элемента-хода, помимо указателя на следующий ход, еще один указатель на элемент, на который следует перепрыгнуть в случае, если исследуемая позиция содержит фигуру. Этот элемент и есть специальный.

Для коней, короля и пешечных взятий этот механизм вырождается и специальный элемент всегда равен следующему. Но для всех остальных ходов, это мощнейший и, главное, универсальный механизм.

Допустим, мы сканируем такой список для некой фигуры. Каждый элемент такого списка хранит помимо координат хода, еще другую служебную информацию, как то является ли ход рокировкой, взятием на проходе, превращением или только взятием/тихим ходом. Последнее поле необходимо для обеспечения общего механизма для пешек.

[2] Такая техника (заранее вычислять частоиспользуемые значения и сохранять их) называется кэширование (caching). Не спутайте с похожим термином - хеширование (hashing)

Так, все ходы пешек делятся на две категории: только тихие ходы, которые не могут быть взятием и ходы взятия, которые не могут не брать фигуру. Рокировка является тихим ходом. Взятие на проходе, иначе называемое енпассан, является только взятием. Так вот: мы проверяем является или нет поле, на которое ходит фигура занятым фигурой противоположного цвета и если этот ход удовлетворяет другим условиям, как те, которые мы рассмотрели выше. Если полностью удовлетворяет, добавляем этот ход в список легальных ходов. После этого, если ход не взятие мы прыгаем на следующий элемент в списке. Если взятие, мы прыгаем на следующий специальный элемент. Так, используя на максимум заранее построенную таблицу, мы имеем очень быстрый генератор ходов.

Генераторы ходов бывают двух типов: легальный и псевдо-легальный. Грубо говоря, легальный генератор всегда возвращает полностью легальные шахматные ходы. Это замечательно, но тут есть подводные камни. Есть группа ходов, которые выбиваются из общей картины. Это ходы, приводящие к взятию короля и рокировка через битое поле. Допустим, мы имеем некий ход, который открывает линию и вражеская фигура может побить короля. Если бы король стоял в другом месте, этот ход был бы вполне законен, но теперь фигура связана и не имеет права подставлять короля под бой. Другими словами, этот вид правил не оперирует самой фигурой, ее типом и местоположением, но учитывает всю позицию на доске как целую.

Описать эти правила статически в генераторе ходов можно, но очень затратно. Этот метод настолько замедляет программу, что сведет на нет все наши попытки сделать быстрый генератор ходов. Поэтому очень распространена другая методика, называемая псевдо-легальной генерацией. Попросту, мы не учитываем эти правила при генерации ходов, поскольку их немного, это не критично. Но заботу об обнаружении и устранении этих ходов берет на себя сам переборочный алгоритм. Поскольку правила динамичны, это идеально вписывается в динамичный поиск.

Технически, мы разрешаем и совершаем нелегальные ходы, однако стоит их сделать, в следующем узле это легко обнаруживается и весь ход отменяется, возвращая вместо оценки специальное значение для ошибки. Этот механизм практически не затратен по времени исполнения, хотя и страдает некоторая математическая общность, поскольку теперь генерация легальных ходов не отделена полностью от переборочного алгоритма.

Еще один недостаток псевдо-легального генератора в том, что мы не можем до начала перебора узнать количество полностью легальных ходов. Это не критично, но желательно для использования некоторых эвристик поиска, например эвристики единственного хода, о которых пойдет речь в дальнейшем.

Глава 7

Реализация шахматной доски

Теперь мы почти готовы перейти к алгоритмам. Но пока порассуждаем о представлении шахматной доски.

Наиболее простым и интуитивным способом было бы определить массив 8 на 8 указателей на объект класса piece, инкапсулирующий информацию о фигуре, и таким образом, меняя указатели между собой, можно реализовать ходы и взятия. Однако, подобная реализация нам не подходит, так как для того, что бы пройтись циклом по всем фигурам, стоящим на доске, необходимо обойти все 64 элемента массива, что было бы расточительством, так как большинство из них фигур не содержат. На месте, где фигуры нет, указатель будет приравнен к NULL. К тому же, при такой реализации нам не понадобится информация о координатах фигуры, сохраненная внутри самой фигуры. Вся информация содержится в координатах массива.

С другой стороны, можно определить одномерный массив указателей на piece и таким образом все фигуры компактно расположатся в одном массиве. Пробежаться циклом по ним будет сущим удовольствием и обеспечит минимальные затраты времени. Однако этот вариант тоже плох, так как, выиграв в одном, мы неизбежно проиграли в другом — теперь получить информацию о фигуре, которая стоит на определенном месте доски, невозможно без сканирования в худшем случае всех элемента массива. То, что в случае реализации с двумерным массивом доставалось бесплатно, теперь невозможно.

Итак, что же делать?

В результате компромисса мы реализовали эти два метода одновременно, что непременно ведет к избыточному представлению и дублированию данных. Теперь при ходе фигуры нам необходимо переписать указатель на фигуру в другое место двухмерного массива и изменить координату внутри фигуры, на которую этот указатель указывает. Приняв это неизбежное зло, мы теперь можем эффективно использовать преимущества обоих подходов.

Итак, для представления позиции создаются массивы 8x8 и одномерный массив, как это было определено ранее. С тем небольшим изменением,что массив 8x8 создается из элементов структуры square, которая, в свою очередь, содержит указатель на фигуру и на объект класса coord (он хранит полную информацию о позиции, включая ее битбоард).

Далее следует информация об очередности хода - тут используются уже определенные ранее макросы WHITE и BLACK. И два интересных битбоарда turn_side_bit и not_turn_bit. Они хранят бит-позиции всех ходящих фигур игрока и оппонента соответственно. При смене хода эти битбоарды меняются местами. А при изменении позиции инкрементно обновляются. Интересно, что конструкцию вида (turn_side_bit | not_turn_bit) можно использовать, чтобы получить

бит-позицию всех фигур на доске.

Важность этих бит-позиций трудно переоценить при генерации ходов. Это очень быстрый способ проверить если нужное поле занято противником или своей фигурой.

Отдельно заметим, что мы стараемся создавать код, который является симметричным для белых и черных. Так, проверки вида

```
if(color == WHITE) {}
else {}
```

всячески избегаются и вся асимметричная логика по возможности перенесена в построенные заранее таблицы. Единственное место, где этим условием пришлось пренебречь является оценочная функция, которую мы рассмотрим в дальнейшем.

Помимо генерации ходов, при переборе ходов нам понадобятся две очень важные функции

```
void do_move_low(move_el* move);
void undo_move_low(move_el* move);
```

Первая проводит всю закулисную рутину при передвижении фигуры, а вторая делает обратную операцию. Эти функции не делают четырех вещей, которые должны обеспечиваться функциями, их вызывающими :

1. Они предполагают, что на позиции from есть фигура
2. Эта фигура принадлежит игроку, чья очередь ходить
3. Если на позиции to есть фигура, то она обратного цвета с фигурой на from
4. Если этот ход есть рокировка, то на нужной позиции должна находится ладья

Подобные ограничения являются примером интеграции между разными модулями программы. В данном случае представления доски и генератора ходов. Ход, полученный из генератора уже удовлетворит всем этим условиям. Поэтому нет необходимости проверять их еще раз. Опять мы пожертвовали независимостью модулей в угоду быстродействия. Конечно, при ручном вводе ходов от пользователя, эти проверки должны быть сделаны отдельно, что нисколько не скажется на быстродействии, поскольку находятся вне наиболее критичного к времени исполнения перебора.

Глава 8

Минимаксный поиск

Если вы задумаетесь о том, как играет в шахматы человек, то придете к выводу, что игрок совсем не просчитывает все возможные варианты развития. Он, руководствуясь своей интуицией, выбирает несколько перспективных вариаций и развивает их на определенное число ходов вперед. Более того, некоторые фигуры не участвующие в комбинации, выпадают из поля зрения игрока. В своем анализе он, руководствуясь абстрактными понятиями, находит слабое звено в обороне противника и пытается провести свои планы вскрытия позиции в выгодном для него ключе, попутно нарушая те же самые устремления противника.

Компьютер, конечно, не способен на подобные абстракции, и единственное его преимущество - это способность к скорости вычислений, которая и не снилась человеку. Попытка обучить компьютер мыслить, как человек не удалась, несмотря на всю заманчивость этой идеи. Возможно, некоторые приведут спорный пример, находящийся на уровне теории заговора, разработок шахматного гения М.Ботвинника, чья программа «Пионер» так и не заиграла. В опровержение этих доводов приведу следующее утверждение : при наличии исходных кодов этой программы, или ходя бы воспоминаний работавших с Ботвинником людей, весь научный мир смог бы восстановить работу мастера за короткий срок. Уж слишком падки люди науки на новые гениальные идеи — дай лишь за что зацепиться.

Итак, в нашем распоряжении вычислительные мощности компьютера и уже готовый эскиз генератора ходов. Допустим, мы имеем некую начальную позицию. Генератор вырабатывает все возможные ходы игрока, и, соответственно, получает определенное количество новых позиций, полученных в результате этих ходов.Теперь определенная функция оценки для каждой из этих позиций, учитывая материальные, позиционные и любые другие преимущества, мыслимые и немыслимые, дает каждой позиции объективную оценку согласно какой-то шкале. Причем, ситуации вроде «мат в три хода» или «в результате размена выигрывается ладья» тоже должны содержаться в данной оценке. И игроку-программе не остается ничего лучшего, как выбрать вариант с наилучшей оценкой.

Будет ли такая программа играть в шахматы? Безусловно! Однако, написать такую оценочную функцию не представляется возможным из-за универсальности, заложенной в ее определении. Никто еще четко не сформулировал правила, согласно которым шахматист отделяет хорошие позиции от плохих, а некоторые даже находят опровержения дебютам, которые годами считались правильными.

Однако, вполне можно выработать определенное приближение к такой функции, согласно перевесу одной из сторон в материале, защищенности королей, открытости фигур, занятости

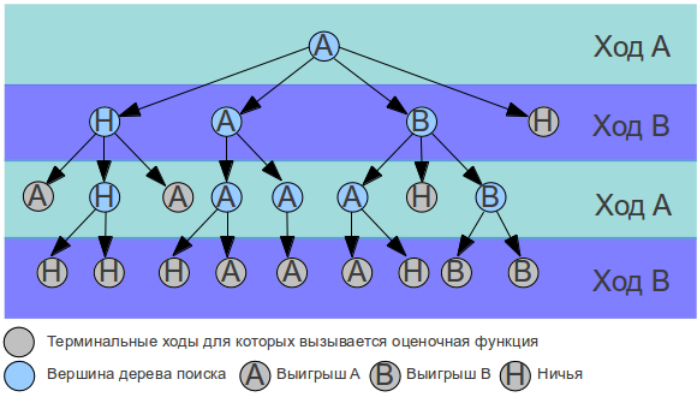

Рис. 8.1: Пример дерева вариантов для минимакса с идеализированной оценочной функцией

центральных позиций доски и так далее. Тогда описанный нами метод будет определенно играть в шахматы и с этой функцией, но не составит определенной конкуренции даже самому слабому игроку. Нужен другой подход.

Можно усовершенствовать данный метод, когда для каждой, полученной в результате перебора ходов, скажем для белых, позиции, мы будем вызывать тот же перебор, который строит позиции, получившиеся в результате хода черных. По аналогии, так же будет продолжен расчет и далее, и в результате будут рассмотрены все ветви шахматной игры, вплоть до однозначно определенного результата — победа одной из сторон или ничья. Хотя и тут существуют позиции, когда ничья не может быть определена без спора. Игрок, на стороне которого закончился последний узел, выберет тот исход из всех возможных ходов в узле, где его выгода максимальна. Он вернет оценку, характеризующею его максимальную выгоду, противнику, который получил такие множественные оценки и с других вариантов ходов. И, соответственно, и на этом уровне противник попытается максимизировать свою выгоду или минимизировать выгоду игрока, и вернуть ее на узел выше по дереву вариантов. Таким образом, по рекурсии оценка поднимется в корень дерева, и опять же максимизацией собственных интересов для каждой из сторон, будет выбран ход, который с наилучшей игрой противника даст наилучший результат.

Препятствием к реализации этого вполне логичного замысла служит величина дерева. Приняв к сведению тот факт, что в среднем на каждой позиции в момент игры существует 20-30 ходов, и взяв число 50 за среднее количество ходов в партии, мы сможем оценить примерное количество всех возможных комбинаций шахматной игры равным 20^{50}. Это астрономическое число и просчет всего дерева займет порядка $3 \cdot 10^{50}$ лет при расчете 10 миллионов позиций в секунду.

Вместо этого просчитывание позиций будем вести не на 50 ходов вперед, а ограничимся некоторой глубиной depth которая будет лежать в диапазоне 1-10.На максимальной глубине будем вызывать оценочную функцию. Однако не ту абстрактную функцию, которую мы идеализированно определили выше, а функцию, которая будет возвращать для каждой позиции числовую оценку ее стоимости.

Псевдокод реализующий алгоритм минимакс будет следующим: Игрок A начинает игру в корне дерева и его стратегия максимизировать оценку, а игрок B, его противник, минимизирует.

```
1  function integer minimax(node,player,depth)
2   if depth <= 0:
3     return the heuristic value of node for player A
4   alpha = -infinity
5   for child in node
6     if(player is A)
7        alpha = max( alpha, minimax(child,not(player),depth-1) )
8     end
9     if(player is B)
10       alpha = min( alpha, minimax(child,not(player),depth-1) )
11    end
12  next for
13  return alpha
14 end function
```

Для того что бы получить лучший ход необходимо запомнить, какая позиция получила максимум в корне дерева и соответственно выбрать ход, с помощью которого она была получена.

Представленный алгоритм описывает ту процедуру, которую мы описали выше словами. Она рекурсивна, асимметрична для обоих игроков и использует неопределенную пока оценочную функцию. И даже некая константа “бесконечность” в ее определении не должна нас смущать. Эта константа по определению больше, чем любое значение, которое может вернуть оценочная функция и может рассматриваться нами как очень большое число.

Глава 9

Оценочная функция

Для того, что бы научно подойти к проблеме оценки шахматной позиции, необходимо четко определить стоящую перед нами задачу и вывести предположения, в рамках которых мы будем выводить теорию. Наша задача сводится к написанию процедуры, которая сможет для любой, или по крайней мере подавляющего количества шахматных позиций, установить их обьективную ценность.

Во первых, мы не зря не избегаем ярлыка “функция” так как функция по определению возвращает только одно число, что несомненно является частным случаем. Ничего не мешает, теоретически, использовать другую модель, где критерий победы определяется вектором, комплексным числом, вероятностью или любым другим математическим обьектом. Практически, я не слышал, что бы кто то эксперементировал в этом направлении. Последуем за общепринятой теорией и постулируем, что модуль оценки является функцией $E_{eval}(x)$ над множеством коректных шахматных позиций, каждой из них противопостовляя одно целое число, которое мы назовем оценкой шахматной позиции. Поскольку множество целых чисел есть множество упорядоченное, позиции можно сравнивать, и если оценки двух любых позиций A и B находятся в соотношении $E_{eval}(A) < E_{eval}(B)$ то из этого однозначно следует, что позиция В лучше, чем позиция А, и наоборот. А так же любые другие закономерности, навязанные природой чисел, как то транзитивность, например.

Поскольку, диапазон возможных значений целых чисел, ограничен архитектурой компьютера, а на деле мы его вскоре искуственно ограничим еще более, при любой оценочной функции, как бы мы не старались, неизбежно будут полностью разные позиции с одинаковой оценкой. Этот факт - недостаток модели, эти позиции действительно одинаково предпочтительны, хотя шахматы достаточно сложная игра, что бы утверждать обратное.

Еще одним таким же свойством является линейность оценочной функции. Это можно обосновать следующим образом. Пусть $\mathscr{F}$ множество независимых признаков позиции. В него входят каждая из стоящих на поле фигур, местоположения этих фигур и некоторые другие. Оценочная функция определена над шахматной позицией, которую полностью описывают множество независимых признаков из $\mathscr{F}$. В первом приближении функция оценки может быть представлена как

$$E_{eval}(\mathscr{F}) = \sum_i F_i \cdot W_i$$

, где i пробегается по всем элементам из $\mathscr{F}$. Веса W_i являются эмпирически подобраными числами. Исходя из этой формы, каждый признак независим от других - фигура, стоящая на

некотором поле не может повлиять на оценку другой фигуры. Грубо говоря, оценку можно высчитывать произвольно деля определяющие факторы на группы. Так, оценка фигуры зависит только от ее местоположения и ее достоинства, а стоящие рядом фигуры ничего не знают о существовании друг друга, поэтому их взаимодействие не может выразится на изменении оценки.

Дополнительное свойство, которому может удовлетворять ОФ, не являющееся, однако, необходимым, является симметричность относительно выбора игрока. Так, если мы вычислим оценку относительно белых и относительно черных, то в сумме эти оценки дадут ноль. Математически, это свойство выражается в том, что мы можем отдельно высчитать оценку для белых и черных фигур, а их разность даст искомую общую оценку. Большинство программ именно так и делают - они хранят отдельно материал белых и черных фигур отдельно, инкрементно изменяя его при необходимости от хода к ходу. Поскольку взятия относительно редки, а расчет материала довольно медленная операция, требующая перебора всех фигур, этот метод довольно выгоден. [1].

Можно построить всю внутреннюю структуру программы, что бы симметрия между белой и черной стороной выражалась более ясно. Вместо того, что бы хранить и работать с белыми и черными фигурами, можно пойти еще дальше и хранить все данные относительно стороны с очередностью хода. Это, конечно потребует при смене очередности производить обмен значений между этими структурами данных. Мы выигрываем в том, что теперь оценочная функция пишется и запускается алгоритмом относительно ходящей стороны. Подобный подход, как мы увидим далее, хорошо вписывается в алгоритм потиска негамакс.

Симметрия не обязательное условие и легко можно придумать ситуацию, когда ей стоит частично пожертвовать. Например, имеет смысл считать защищенность своего короля и короля противника по разному. Это может придать движку более осторожный или авантюрный стиль. Или, например, считать мобильность фигур по разному при движении вперед и назад.

Линейная симметричная оценочная функция является хорошим приближением, зарекомендовавшим себя на практике и удобное в теоретических размышлениях. Поэтому для начала, мы будем рассматривать только факторы, которые не нарушают этих свойств.

9.1 Материальная оценка

Наиболее важным фактором оценки является материальная ценность фигур. Любой шахматист знает, что конь важнее пешки, ладья предпочтительнее легких фигур, а ферзя не стоит за зря менять ни на какую фигуру. Но как можно численно выразить эти соотношения? Примем материальную оценку пешки равную 100 единиц на нашей оценочной шкале. Назовем эту единицу измерения *cp*, сотая доля пешки. Все остальные оценки будем выражать в этой системе единиц. Эмпирически были установлены примерно следующие соотношения:

[1] Подобное хранение материала так же полезно при определении цунгцванга и фазы игры - об этом пойдет речь далее

Фигура	Обозначение	Оценка
Пешка	W_{pawn}	100
Конь	W_{knight}	300
Слон	W_{bishop}	300
Ладья	W_{rook}	500
Ферзь	W_{queen}	1000
Король	W_{king}	?

Конечно, эти цифры должны быть уточнены многократными экспериментами и в разных программах могут отличатся. Уже только одна их настройка представляет собой огромное поле для исследований. Впоследствии мы постараемся обьяснить этот выбор и покажем как можно его улучшить.

Давать оценку королю излишне, поскольку по правилам игры его нельзя потерять. Несмотря на это, на ранних стадиях разработки движка это может быть полезно, когда перебор не может определять шах другими путями, как через потерю короля[2]. В последствии, стоит переложить задачу определения шахов на сам перебор, и формально считать оценку короля равной нулю[3].

Самым простым способом реализовать оценочную функцию будет через подсчет разницы материала игроков. С точки зрения белых материальная оценка будет равна

$$E_{eval}^{white}\left(P_{1\ldots N_w}^{white}, P_{i\ldots N_b}^{black}\right) = \sum_{i}^{N_w} W_{P_i^{white}} - \sum_{i}^{N_b} W_{P_i^{black}}$$

, где P^{white}, P^{black} , белые и черные фигуры соответственно, а W_P обозначают веса этих фигур, взятые из таблицы выше. Заметим, что исходя из этой таблицы, фигуры одного достоинства, будь они белыми или черными, имеют одинаковый вес. Если коэффициенты W_i не зависят от стороны, то очевидно, что функция линейна и симметрична. Она вычисляет сумму материала для игрока белых и вычитает из нее сумму материала для игрока черных. Полученное число и будет искомой оценкой. Пользуясь всеми этими свойствами полученную формулу можно переписать так:

$$\begin{aligned} E_{eval}^{white} &= W_{pawn} \cdot (\text{количество белых пешек} - \text{количество черных пешек}) + \\ &= W_{knight} \cdot (\text{количество белых коней} - \text{количество черных коней}) + \\ &= W_{bishop} \cdot (\text{количество белых слонов} - \text{количество слонов пешек}) + \\ &= W_{rook} \cdot (\text{количество белых ладей} - \text{количество черных ладей}) + \\ &= W_{queen} \cdot (\text{количество белых ферзей} - \text{количество черных ферзей}) \end{aligned}$$

То, что хорошо для белых, плохо для черных. С их точки зрения выполняется соотношение $E_{eval}^{black} = -E_{eval}^{white}$. Можно придать последнему выражению более симметричный вид

$$E_{eval}^{\text{turn side}}\left(P_{1\ldots N_t}^{\text{turn side}}, P_{i\ldots N_n}^{\text{not turn side}}\right) = \sum_{i}^{N_t} W_{P_i^{\text{turn side}}} - \sum_{i}^{N_n} W_{P_i^{\text{not turn side}}}$$

[2] В таком случае, его оценка должна быть очень большим числом, что бы сумма всех остальных фигур даже теоретически не могла его превысить. В определенной нами шкале, любое число порядка 10000 вполне подойдет.

[3] Это так же хорошо сочетается с вычислением фазы игры, где материальная оценка короля будет только мешать.

9.2 Позиционные факторы

Материальные факторы - наиболее значемые при вычислении оценки. Если бы мы имели практическую возможность запускать поиск на неограниченную глубину, их одних было бы достаточно для идеальной игры. Однако, если мы запустим поиск на конечную глубину и этот поиск будет руководствоваться только материальной оценкой фигур, то очень быстро обнаружим, что игра носит довольно странный характер. Компьютер будет играть довольно глупо, бесмысленно передвигать фигуры, пока игрок не допустит ошибку и в пределах перебора будет обнаружен форсированный выигрыш материала. Тогда игроку останется только молится, видя резко усилившуюся игру противника.

Это происходит потому, что материальный перебор слеп к стратегической обстановке на доске. Насколько централизованы и мобильны фигуры. Насколько защищен король и продвинуты пешки. Именно эти позиционные факторы, накопившись и усилившись, перерастают в материальное преимущество. Поэтому стоит предпочитать позицию, где фигуры централизованны и мобильны, позиции с фигурами, забитыми по обочинам доски. Или сильную пешечную структуру предпочитать позиции с пешками, не поддерживающими друг друга. Хотя во всех этих случаях будет одинаковое количество материала.

Хорошо подобранные позиционные бонусы должны статистически предсказывать материальные изменения на меньшей глубине, чем они реально произойдут. Так, допустим, что мы даем бонус за близость наших фигур к королю противника. До объявления ему мата еще далеко - материальный перебор не увидит. Однако, если движок сможет поставить свои фигуры близко к вражескому королю, вероятность опасных комбинационных маневров повышается и в результате может быть форсирован выигрыш фигуры или даже мат.

Но тут нужно остерегаться! Все позиционные бонусы, которые мы начисляем, не должны превышать материальную стоимость пешки. Иначе, слепо гонясь за максимизацией, может быть пожертвована фигура за позиционные факторы. Само по себе это не так уж страшно и может дать движку “человеческий” стиль игры. Но все же, основным правилом для нас должно стать то, что позиционные преимущества ведут к материальным, а не наоборот.

Вторым по значимости, после материальных весов фигур, следуют позиционные бонусы за централизацию и продвижение фигур. Это даст движку, кроме комбинационной игры, еще примитивное позиционное видение. Эти бонусы легко реализуются в виде таблиц PST, которые мы подробно расмотрим позже. Их преимущество в том, что они не нарушают линейности, а множество коеффициентов представляют большую свободу для кодирования в них множества эмпирических правил, которыми шахматист руководствуется в игре. Как мы увидим в последствии, материальная оценка в сочетании с таблицами PST могут дать вполне адекватную игру, внешне неотличимую от игры хорошего мастера.

Позициоые факторы зачастую нелинейны и учитывают интеракцию между независимыми факторами. Примером таких взаимодействий могут быть:

1. Зависимость от фазы игры. Позиционные факторы имеют тенденцию изменятся в процессе игры. Так, например, в эндшпиле короля нужно вывести в центр доски, а в начале игры держать как можно дальше от центра.

2. Закрытости позиции. В закрытой позиции кони строят больше, чем слоны, а в открытой наоборот. В открытой позиции два слона получают дополнительный бонус.

3. Фигуры на пути движения. Слон, который не имеет большой свободы перемещений плохой и его стоит поменять на другую легкую фигуру противника при первой возможности. И наоборот.

4. Форпосты для коней. Когда конь стоит на позиции, где он теоретически не может быть атакован пешками противника.

5. Защищенность своего короля и оголенность короля противника.

Все это и многое другое нужно учитывать для хорошей оценочной функции, а в идеале и найти хороший баланс между всеми позиционными факторами, что бы программа не стала играть странно, гонясь за несуществующими преимуществами. Сталкивая два эквивалентных движка[4], один с хорошо сбалансированными позиционными весами, а другой с произвольно выбраными, вы увидите, что второй часто ошибается - его позиционная оценка может плавно возрастать на протяжении нескольких ходов, а потом резко оборвется в противоположную сторону. Первый движок тоже может ошибаться, но будет делать это гораздо реже. Важно понимать, что данный эффект следует из за эффекта горизонта, но отнюдь не эквивалентен ему. Это понятно, если смотреть на позиционную оценку как на вероятностную.

9.3 Особые случаи оценки

Хорошая оценочная функция должна так же должна определять теоретически ничейные позиции, где даже обладая материалом, невозможно поставить мат. Такие позиции должны возвращать ничейную оценку, равную нулю. Такую же оценку должны возвращать патовые позиции.

Несколько сложнее с оценками матовых позиций. Мат это самое крайнее состояние игры и оценка должна быть соответствующей. Удобно определить константу INFINITY, большую, чем самое хорошее значение, которое способна вернуть оценочная функция, основанная на материале и позиционных факторах. Тогда, вызванная для игрока, INFINITY будет соответствовать мату противника, а -INFINITY мат игроку.

Удобно переложить функцию определения пата и мата на переборочный алгоритм. Поскольку эти состояния проще определить динамичным перебором, а не статичной оценкой. Мы опять вынуждены интегрировать модули, нарушая их общность, однако это окупается выигрышем в скорости.

9.4 Тактика или стратегия?

При проектировании оценочной функции существуют два подхода.

Представим себе функцию, которая подсчитывает только материал. Она очень быстрая для подсчета и, соответственно, за то же самое время минимаксный поиск сможет перебрать гораздо больше ветвей и, соответственно, спустится на большую глубину. Такой движок будет монстром в тактике и способным находить совершенно сумасшедшие комбинации. Его недостаток в том, что в погоне за чистым материалом его позиционная игра будет слаба. Игрок сможет форсировать долгий размен, результатом которого будет не материальное, а позиционное преимущество, к которому алгоритм слеп. Например, проходная пешка.

[4] Что бы перебор не мешал, можно установить глубину перебора минимально возможной. Так поиск будет большей частью руководствоваться оценочной функцией и все ее слабые места будут ярче выражены.

Другой подход сделать оценочную функцию, считающей помимо материального баланса еще и сотни других мелких параметров, примеры некоторых мы уже давали ранее. Минимаксный алгоритм с такой оценочной функцией будет отбирать мельчайшие, незаметные глазу, позиционные преимущества, однако, ввиду своей тяжеловесности, время работы алгоритма возрастет и, соответственно, мы будем вынуждены сократить глубину перебора. Комбинационная игра от этого, соответственно, только пострадает.

Реальные функции оценки, применяемые в современных шахматных движках являются компромиссом между этими двумя крайностями. Подбираются различные техники оценки позиции, которые по мнению авторов программы являются необходимыми и подбираются весы для их эмпирических параметров, что приводит к определенному балансу в игре и формирует уникальный стиль движка. Золотой грааль шахмат, настройка идеальных параметров оценочной функции, находится скорее в области искусства, чем в области математической истинности.

9.5 Общая философия при программировании ОФ

Оценочная функция очень сложна и при ее проектировании и реализации важно придерживаться некоторых правил:

1. Ортогональность. Пока это возможно, стоит избегать написание разных компонентов оценки, которые описывают одно и то же свойство позиции. Если вы добавляете компонент, имеющий “неортогональную проекцию” на уже существующие компоненты, постарайтесь ее минимизировать или переписать код, скомбинировав эти два компонента в один.

2. Гладкость оценочной функции. Между близкими шахматными позициями оценка также должна быть близкой. Так, например, если мы даем бонус за нахождение коня на так называемом форпосте, где он теоретически не может быть атакован пешками противника, то и поля, с которых он может попасть на этот форпост тоже должны получать бонус, правда меньший. Пешка, которая грозит вырваться в ферзи должна начать получать позиционный бонус еще за несколько ходов до превращения. Причем этот бонус должен увеличиваться с продвижением, поощряя пешку двигаться дальше.

3. Понимание прогресса. Гораздо важнее, что бы функция оценки могла способна адекватно оценить, какая из двух очень близких позиций лучше, чем решить тоже самое для абсолютно различных позиций. Функция оценки не должна судить о том, какой из дебютов лучше разыграть. Ее задача сводится к тому, что бы решить, стоит ли менять коня на слона и в какую сторону предпочтительней рокироваться.

4. Лучшее - худшее поведение. Лучше ошибатся в пределах 10 сотых пешки все время, чем быть абсолютно правым в 99.9% случаев и ошибится на 300 сотых пешки один раз.

5. Перенести, насколько это возможно, все шахматные правила и прочую логику в заранее построенные таблицы и коеффициенты, избавляя от необходимости вычислять эквивалентные им данные во время перебора.

9.6 Научный подход к построению ОФ

Попробуем на основе всего сказанного разработать максимально простую и быструю ОФ, но в то же время вместить в нее максимальное количество шахматных эмпирических данных, которыми

шахматисты руководствуются в игре. К тому же она должна быть линейна и антисимметрична. Я настоятельно рекомендую использовать эту главу[5] при разработке своего движка, один в один скопировав коэффициенты. Иначе, будет тяжело отделить баги перебора от плохо подобранных весов.

Итак, в первую очередь мы хотим обеспечить следующее поведение ОФ:

1. Избегать размена одной легкой фигуры на три пешки
2. Поощрять движок к сохранению пары слонов
3. Избегать размена двух легких фигур на ладью и пешку

Подберем материальные веса фигур таким образом, что бы все эти условия выполнялись.

Первый пункт гарантируется простым условием

$$\begin{aligned} W_{bishop} &> 3 \cdot W_{pawn} \\ W_{knight} &> 3 \cdot W_{pawn} \end{aligned}$$

Конечно, существуют позиции, где три(а зачастую и две) пешки более, чем достаточная компенсация за легкую фигуру. Но как правило менять фигуру ради трех пешек не стоит, так как минимум одна из этих пешек может быть легко отыграна впоследствии противником и вся позиция станет критической. Поэтому, в дальней перспективе обладание легкой фигурой предпочтительней.

Второй пункт обеспечивается выполнением простого условия

$$W_{bishop} > W_{knight}$$

Конечно это условие само по себе ничего не гарантирует, поскольку мы можем остатся к концу партии с слоном против коня и двумя голыми королями. Однако, шахматисты часто предпочитают менять своего коня на слона противника и редко наоборот. Например, белый слон на $d3$ редко берет черного коня на $e4$. С другой стороны, белый конь на $e4$ часто бьет черного слона на $f6$. Конечно, можно возразить, что нередко белый слон с $g5$ бьет коня на $f6$. Но, как правило, это происходит после вынуждающего $h6$ или других подобных соображений, которые мы не будем здесь рассматривать. То же самое со слоном на $b4$ в системе Нимцовича, который должен брать слона на $c3$ только после вынуждающего хода $a3$.

Итак, мы препятствуем не необходимому размену слона на коня. Это повышает шанс остатся с парой слонов ближе к концу игры. С другой стороны, существует риск попасть в эндшпиль с “плохим” слоном или со слоном против коня и пешками только с одной стороны. Или в закрытой позиции. Что бы правильно разрешить этот вопрос, можно написать отдельную процедуру, которая учитывает все эти специальные случаи, но на данном этапе мы не можем себе этого позволить, оставаясь в рамках упрощенной модели.

Обьеденяя все вышесказаное, можно постулировать

$$W_{bishop} > W_{knight} > 3 \cdot W_{pawn}$$

Третий пункт достигается соотношением

$$W_{bishop} + W_{knight} > W_{rook} + W_{pawn}$$

[5] Большая часть материала для этого раздела является переводом замечательной статьи, опубликованной на http://chessprogramming.wikispaces.com/Simplified+evaluation+function

Тут есть небольшая проблема, так как согласно определенному мнению две легкие фигуры стоят ладьи плюс две пешки. Однако, таким образом материальная ценность двух легких фигур может быть завышена. Поэтому вместо

$$W_{rook} + 2 \cdot W_{pawn} > W_{bishop} + W_{knight} > W_{rook} + W_{pawn}$$

Мы возьмем компромисный вариант

$$W_{bishop} + W_{knight} = W_{rook} + 1.5 \cdot W_{pawn}$$

Здесь самое время упомянуть термин “симметрия”. В шахматах, чем более симметрична позиция, тем более велика вероятность для ничьи. Последнее уравнение ломает эту симетрию, поскольку исходя из него, движок будет предпочитать несимметричные комбинации фигур, как то две легкие фигуры против ладьи и пешки или ладью и две пешки против двух легких фигур.

Если мы найдем решение, удовлетворяющее всему вышесказаному, то оценка двух коней должна быть немного меньше $W_{rook} + 1.5 \cdot W_{pawn}$, а двух слонов немного больше.

Нам необходимо еще одно уравнение, связывающее оценку ферзя с оценкой остальных фигур. Исходя из эмпирических наблюдений шахматистов, имеет место быть следующее равенство:

$$W_{queen} + W_{pawn} = 2 \cdot W_{rook}$$

Итак, мы имеем систему уравнений

$$\begin{aligned} W_{bishop} &> W_{knight} > 3 \cdot W_{pawn} \\ W_{bishop} + W_{knight} &= W_{rook} + 1.5 \cdot W_{pawn} \\ W_{queen} + W_{pawn} &= 2 \cdot W_{rook} \end{aligned}$$

Конечно, этого мало для решения. Нужно выбрать относительную шкалу, ее обеспечит условие $W_{pawn} = 100$. Одним из легитимных решений этой системы, в правильности которого можно убедится простой проверкой, является

$$\begin{aligned} W_{pawn} &= 100 \\ W_{knight} &= 320 \\ W_{bishop} &= 330 \\ W_{rook} &= 500 \\ W_{queen} &= 900 \end{aligned}$$

Вы уже видите, куда я клоню. Несмотря на то, что эти материальные оценки, статическая разница в оценках коня и слона определяет шкалу влияния позиционных факторов в целом. Она составляет 10 десятых от материальной стоимости пешки.

Можно проверить, что данный выбор весов автоматически удовлетворяет еще одному соотношению.

$$W_{bishop} + 2 \cdot W_{pawn} > W_{knight} + 2 \cdot W_{pawn} > W_{rook}$$

Мы могли бы использовать его, включив в систему уравнений, хотя над его обоснованостью можно поспорить. Для того, что бы произвести размен, согласно неравенству, нужно взять

ладьей легкую фигуру, а затем пешку, которая ее защищает. И после этого понадобится еще два полухода, что бы отыграть дополнительную пешку. В данном случае компенсация не столь очевидна.

Применив последнее соотношение к системе , можно получить следующие уравнения

$$\begin{aligned} 2 \cdot W_{bishop} + W_{knight} &> W_{queen} \\ W_{rook} + W_{bishop} + W_{pawn} &> W_{queen} \\ W_{rook} + W_{knight} + W_{pawn} &> W_{queen} \end{aligned}$$

Мне не нравится опиратся на эти уравнения, в виду их спорности. С другой стороны, я предпочитаю использовать более классическое соотношение $W_{queen} + W_{pawn} = 2 \cdot W_{rook}$.

Мы уже неоднократно упоминали, что, поскольку по правилам игры короля нельзя потерять, нет никакой необходимости определять его материальную оценку. Однако, если ввиду неких соображений это все таки необходимо сделать, можно постановить $W_{king} = 20000$. Это число с одной стороны достаточно большое, что бы быть больше любой другой комбинации материала, а с другой максимально возможный материал в игре для каждой их сторон ограничен сверху числом $20000 + 9 * 900 + 2 * 500 + 4 * 300 = 30300$, что поместится в два бита, учитывая знак. Так, что если есть необходимость хранить две таких оценки вместе, то для этого вполне подойдет стандартный 4 битный тип.

9.7 Таблицы Поле-фигура (PST)

Одним из лучших механизмов вычисления позиционной оценки является использования таблиц поле-фигура, в литературе обозначаемых как PST. Такими таблицами можно закодировать многие эмпирические правила, которыми шахматисты руководствуются при игре. Так, для каждой фигуры создается массив 8 на 8, где каждый его элемент соответствует шахматному полю. При нахождении фигуры, ей добавляется оценка, согласно полю на котором она находится. Отсюда и название - таблицы позволяют связать бонусами или штрафами игровые поля и находящиеся на них фигуры.

Так, при прочих равных условиях, фигура будет стремится перейти на поле с максимальной оценкой и избегать минимальных. В самом простом случае, который мы рассмотрим далее, к материальной стоимости фигуры добавляется позиционный бонус, взятый из таблицы. Это оставляет функцию линейной, то есть разные фигуры по прежнему ничего не знают о существовании друг друга. Но мы все же постараемся подобрать коеффициенты таким образом, что бы с определенной степенью точности эмулировать и взаимодействия.

Все таблицы и примеры даны с точки зрения белой стороны. Для черных необходимо зеркально отобразить ось $y \to 8 - y$.

9.7.1 Пешки

Мы хотим поощрять пешки к движению вперед на позиции неприятеля. Особенно, центральные пешки, которые должны прийти в движение как можно раньше. Это может противоречить правилу не двигать пешки, защищающие позицию короля. Об этом нам тоже придется позаботится. На данном этапе проигнорируем проверку, является ли пешка проходной. Для этого впоследствии могут быть написаны дополнительные процедуры.

```
 0,  0,  0,  0,  0,  0,  0,  0,
50, 50, 50, 50, 50, 50, 50, 50,
10, 10, 20, 30, 30, 20, 10, 10,
 5,  5, 10, 25, 25, 10,  5,  5,
 0,  0,  0, 20, 20,  0,  0,  0,
 5, -5,-10,  0,  0,-10, -5,  5,
 5, 10, 10,-20,-20, 10, 10,  5,
 0,  0,  0,  0,  0,  0,  0,  0
```

Хорошо, давайте прокоментируем таблицу. Прежде всего щит перед крепостью, в которую попадет король в случае короткой рокировки - пешки на $f2$,$g2$ и $h2$ получают небольшие бонусы. Аналогично для длинной рокировки. Так же мы устанавливаем штрафы полям $f3$ и $g3$, пешки на которых создают бреши в защите короля. Пешка h имеет одинаковый вес на полях $h2$ и $h3$, так что здесь движок может оголить короля в случае необходимости.

Поля $f4$, $g4$, $h4$ штрафуют за игру пешками у позиций короля. Продвижение же этих пешек на пятую горизонталь тоже ничего не даст - зачем двигать пешки так далеко, если на второй горизонтали веса не хуже.

Аналогично для длинной рокировки на ферзевом фланге. На данном этапе таблицы не делают различия, в какую сторону действительно рокировался король.

Самые большие штрафы имеют пешки на $d2$ и $e2$. Пешки на них не желательны, как и нет смысла сдвигать их $d3$ и $e3$. Эти пешки должны занять центр поля - четвертую, а лучше пятую горизонталь.

Начиная с шестой горизонтали все пешки начинают получать серьезный бонус, оправдывающий их движение вперед. На седьмой горизонтали этот бонус еще больше, пешка торопится вперед для превращения в фигуру.

9.7.2 Кони

Заметим, что конь относительно медленная фигура. Находится у края доски не желательно. В углу и того хуже. Поэтому, мы просто поощряем их находится как можно ближе к центру.

```
-50,-40,-30,-30,-30,-30,-40,-50,
-40,-20,  0,  0,  0,  0,-20,-40,
-30,  0, 10, 15, 15, 10,  0,-30,
-30,  5, 15, 20, 20, 15,  5,-30,
-30,  0, 15, 20, 20, 15,  0,-30,
-30,  5, 10, 15, 15, 10,  5,-30,
-40,-20,  0,  5,  5,  0,-20,-40,
-50,-40,-30,-30,-30,-30,-40,-50
```

Легко видеть, что движок с радостью разменяет за три пешки любого коня, запертого в углу доски. Ко всему прочему, добавим небольшие бонусы к полям $e2$, $d2$, $b5$, $g5$, $b3$ и $g3$, основным позициям для ввода коней в игру.

9.7.3 Слоны

```
-20,-10,-10,-10,-10,-10,-10,-20,
-10,  0,  0,  0,  0,  0,  0,-10,
```

```
−10,  0,  5, 10, 10,  5,  0,−10,
−10,  5,  5, 10, 10,  5,  5,−10,
−10,  0, 10, 10, 10, 10,  0,−10,
−10, 10, 10, 10, 10, 10, 10,−10,
−10,  5,  0,  0,  0,  0,  5,−10,
−20,−10,−10,−10,−10,−10,−10,−20
```

В общем мы избегаем краев доски и углов в частности. Предпочитаем располагать слонов на полях $b3$, $c4$, $b5$, $d3$, а так же центральных полях. Не желательно так же менять слона на $d3$(или $c3$) на коня $e4$, поэтому поля $c3(f3)$ получают бонус в 10 сотых пешки. В результате слон, стоящий на $d3(c3)$ стоит $330+10$, а конь на $e4$ стоит $320+20$. Поэтому решение о том, стоит ли менять слона на коня зависит от других факторов. С другой стороны, слон на $e4(330+10)$ будет взят конем $f6(320+10)$. А слон на $g5(330+5)$ не побьет черного коня на $f6(320+10)$.

9.7.4 Ладьи

```
 0,  0,  0,  0,  0,  0,  0,  0,
 5, 10, 10, 10, 10, 10, 10,  5,
−5,  0,  0,  0,  0,  0,  0, −5,
−5,  0,  0,  0,  0,  0,  0, −5,
−5,  0,  0,  0,  0,  0,  0, −5,
−5,  0,  0,  0,  0,  0,  0, −5,
−5,  0,  0,  0,  0,  0,  0, −5,
 0,  0,  0,  5,  5,  0,  0,  0
```

Главная идея в централизации и захвате седьмой горизонтали. Мы избегаем выводить ладью через вертикали a и h, что бы не было соблазна защищать пешку $b3$, выводя ладью на $a3$ и тому подобные варианты.

9.7.5 Ферзи

```
−20,−10,−10, −5, −5,−10,−10,−20,
−10,  0,  0,  0,  0,  0,  0,−10,
−10,  0,  5,  5,  5,  5,  0,−10,
 −5,  0,  5,  5,  5,  5,  0, −5,
  0,  0,  5,  5,  5,  5,  0, −5,
−10,  5,  5,  5,  5,  5,  0,−10,
−10,  0,  5,  0,  0,  0,  0,−10,
−20,−10,−10, −5, −5,−10,−10,−20
```

В общем, мы добавили штрафы тем клеткам, где присутствие ферзя не желательно. В добавок, уствновлены небольшие бонусы для ферзя на центральных клетках, а так же клетках $b3$ и $c2$. Все остальное должно быть сделано тактикой.

9.7.6 Короли

Поведение короля, в отличии от других фигур, сильно зависит от фазы игры, что нельзя выразить одной таблицей. В середине игры мы хотим, что бы король держался как можно дальше от центра, за надежным пешечным щитом

```
−30,−40,−40,−50,−50,−40,−40,−30,
```

```
−30,−40,−40,−50,−50,−40,−40,−30,
−30,−40,−40,−50,−50,−40,−40,−30,
−30,−40,−40,−50,−50,−40,−40,−30,
−20,−30,−30,−40,−40,−30,−30,−20,
−10,−20,−20,−20,−20,−20,−20,−10,
 20, 20,  0,  0,  0,  0, 20, 20,
 20, 30, 10,  0,  0, 10, 30, 20
```

Ближе к концу игры приоритеты меняются и король должен ринуться в центр на поддержку проходных пешек.

```
−50,−40,−30,−20,−20,−30,−40,−50,
−30,−20,−10,  0,  0,−10,−20,−30,
−30,−10, 20, 30, 30, 20,−10,−30,
−30,−10, 30, 40, 40, 30,−10,−30,
−30,−10, 30, 40, 40, 30,−10,−30,
−30,−10, 20, 30, 30, 20,−10,−30,
−30,−30,  0,  0,  0,  0,−30,−30,
−50,−30,−30,−30,−30,−30,−30,−50
```

Конечно, необходимо четко определить, когда и как король переходит между этими двумя фазами. Не существует однозначной стратегии, согласно которым можно решать, находимся ли мы в миттельшпиле или игра уже перешла в стадию эндшпиля. Простейшим решением будет соблюдение следующих условий. Эндшпиль, если

- Обе стороны не имеют ферзей или
- Каждая сторона, которая имеет ферзя, не имеет никаких других фигур или одну легкую фигуру максимум.

Главный недостаток этого решения в том, что оценочная функция перестает быть гладкой, что вносит некоторую непредсказуемость в поиск.

9.8 Фаза игры

Другим решением является использование интерполяции между ними, где оценка плавно переходит из первой таблицы во вторую в зависимости от фазы игры. Гладкость, как мы видели, необходима для сглаживания эффекта горизонта. Вопрос о функции, с помощью которой лучше делать интерполяцию остается открытым. Я использую простую линейную функцию.

$$score = phase \cdot PST_{middlegame} + (1 - phase) \cdot PST_{endgame}$$

где фаза игры определяется количеством материала на доске[6]. Единица сответствует началу игры, а ноль эндшпилю с двумя голыми королями.

$$phase = \frac{CurrentMaterial}{BeginMaterial}$$

Теперь понятно, зачем оценка короля должна быть равна нулю - его материал излишен и будет только мешать при вычислении фазы игры.

[6] Необходимо помнить, что при превращении пешки в фигуру может статься $CurrentMaterial > BeginMaterial$, что ведет к нежелательному значению для фазы игры. В таком случае, я считаю фазу игры равной единице.

Глава 10

Негамакс

Алгоритм минимаксного поиска, псевдокод которого мы представили, наиболее легок для понимания, однако не предпочтителен для реализации в шахматной программе из за четкой асимметрии кода для двух сторон. Улучшением его будет служить так называемый алгоритм негамакс, его код для каждой из сторон будет искать только максимум :

```
1 function integer negamax(node, depth)
2  if depth <= 0:
3    return the heuristic value of node for current player
4  alpha = -infinity
5  for child in node:
6    alpha = max(alpha, -negamax(child, depth-1))
7  next for
8  return alpha
9 end function
```

Главное новшество этого алгоритма это полная симметрия кода для каждой из сторон, что позволяет исключить лишние проверки. Оценочная функция вызывается на любой глубине не с точки зрения игрока A, а со стороны игрока,чья очередь ходить на данной глубине. Предположим, функция на определенной глубине вернула оценку для своей стороны. Вернувшись на уровень выше, этой оценке будет добавлен «минус» и будет найден максимум всех этих «минусовых» позиций, что эквивалентно нахождению минимума для оценок без «минуса», таких какие были получены на следующей глубине. Мы все еще неявно проводим чередования поиска максимума с поиском минимума, но делаем это более элегантно.

Таким образом негамакс полностью эквивалентен минимаксу и это ясно из следующего математического тождества :

$$max(a,b) = -min(-a,-b)$$

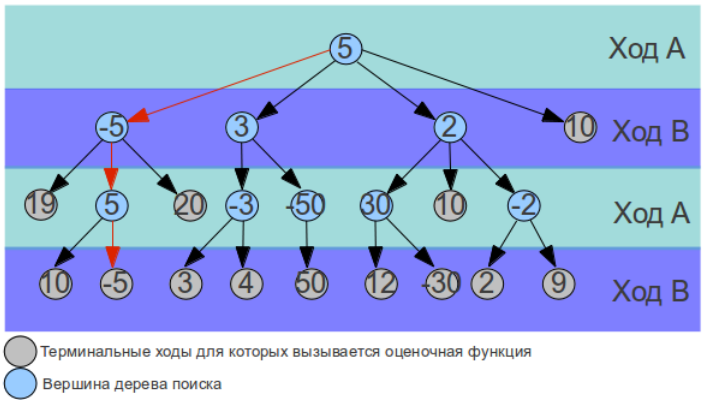

Рис. 10.1: Пример дерева для алгоритма негамакс

Глава 11

Альфа-бета

Существует безопасная оптимизация негамаксного поиска, при которой результат работы алгоритма не изменится, однако некоторые ветви дерева не будут рассмотрены как заведомо неперспективные. Дабы избежать путаницы с минусами рассмотрим минимаксный алгоритм: на каждой глубине мы будем хранить два числа — лучшая оценка, найденная до сих пор для обеих сторон. И если на определенной глубине поиска при переборе позиций найдена лучшая оценка, то при дальнейшем переборе эта оценка будет только увеличиваться. Однако если на глубине выше настоящей противник уже имеет ход с оценкой ниже найденной, то он наверняка не выберет эту ветвь, так как его стратегия минимизировать ход игрока на любой глубине.

Запишем псевдокод этого алгоритма, известного под названием альфа-бета поиск:

В минимаксной нотации :

```
1  function integer alphabeta(node,player,alpha,beta,depth)
2   if depth <= 0:
3     return the heuristic value of node for player A
4   if(player is A)
5     for child in node
6        alpha = max( alpha, alphabeta(child,not(player),alpha,beta,depth-1) )
7        if(beta<=alpha)
8           return beta
9     next for
10    return alpha
11  end
12  if(player is B)
13    for child in node
14       beta = min( beta, alphabeta(child,not(player),alpha,beta,depth-1) )
15       if(beta<=alpha)
16          return alpha
17    next for
18    return beta
19  end
20 end function
```

И в негамаксной нотации:

```
1  function integer alphabeta(node,alpha,beta, depth)
2   if depth <= 0:
3     return the heuristic value of node for current player
4   for child in node:
5     alpha = max(alpha, -alphabeta(child,-beta,-alpha, depth-1))
6     if(alpha>=beta)
7        return beta;
8   next for
9   return alpha
10 end function
```

Что имеет, конечно более компактный вид. Поскольку сложность алгоритма экспоненциальна, альфа-бета поиск как правило позволяет за то же время просмотреть на один-два полухода глубже, что немало.

Так, альфа переменная хранит гарантируемый минимум оценки, найденный до сих пор для ходящего игрока, а бета хранит максимум гарантируемый максимум для противника. Другими словами, при наилучшей игре обеих сторон, игрок не допустит, что его оценка в результате хода будет ниже, чем альфа, а его противник не допустит оценку выше бета значения.

Полный перебор возвращает точную оценку для лучшего хода. Ту же оценку вернет алгоритм альфа-бета. Разница между ними в том, что на каждой глубине (в частности в корне поиска) альфа-бета гарантированно возвращает точную оценку для лучшего из уже найденных ходов. Оценка хода, который заранее хуже,чем уже найденный, вычисляется не точно.

Например, на начальной глубине первый сгенерированный ход на проверку рекурсией оказался нейтральным. Второй сгенерированный ход приводит к форсированному варианту, в середине которого игрок теряет фигуру и в финале получает мат. Альфа-бета алгоритм не увидит мат, так как уже потеряв фигуру, вся ветвь отсечется как неперспективная. Поэтому в корень вернется оценка, соответствующая потере фигуры, а не мата.

Данная эвристика является безопасной, так как математически обоснованно вернет тот же ход, какой вернула бы функция полного перебора с той же оценкой. Фактор экспоненциального роста может уменьшится в разы. Причем, алгоритм тем эффективнее, чем раньше в каждом узле будет сгенерирован лучший ход. Однако, мы не можем знать какой ход лучший, не произведя перебор.

Решением этого парадокса являются всевозможные эвристики. Например, первыми мы можем проверять взятия и только после них тихие ходы. Взятия приводят к более значимому изменению статической оценки и поэтому в среднем они будут чаще являться лучшими ходами. Или можно произвести вначале поиск на меньшую глубину, получив примерные оценки всех возможных ходов и потом, отсортировав ходы по убывающей используя полученные оценки запустить более глубокий альфа-бета поиск.

Перечислим некоторые свойства этого алгоритма.

1. Он никогда не вернет оценку, которая находится вне окна, задаваемого значениями альфы и беты.

2. Если алгоритм вернул оценку равную альфа, это означает, что настоящая оценка меньше или равна альфа.

3. Если алгоритм вернул оценку равную бета, это означает, что настоящая оценка больше или равна бета.

4. Если значения альфы и беты в корне алгоритма таковы, что альфа теоретически меньше любого значения, возвращаемого оценочной функцией, а бета, соответственно больше любого такого значения, то при правильной работе алгоритма будет гарантированно возвращен лучший ход. Или, что эквивалентно, на корневой глубине будет гарантированно поднята оценка альфа как минимум один раз.

5. Или наоборот, если альфа и бета не покрывают весь спектр возвращаемых значений оценочной функции, возможен случай, когда лучший ход не будет найден. Для рядового узла в глубине дерева это не страшно и является частью специфики алгоритма. Тогда будет возращена та же начальная альфа, с которой алгоритм и начал изучать узел. А поскольку альфа это уже гарантированная оценка для игрока в результате хода в другой части дерева, то вернув альфу мы по сути дискредитируем эту ветвь — она не сможет перезаписать тот ход, так как эта перезапись возможна только при увеличении оценки, но не при равенстве. В корне дерева это может вызвать проблему, поскольку предположение, что алгоритм гарантированно находит лучший ход неверно.

6. Чем меньше окно значений между альфой и бетой, тем агрессивнее производятся отсечения и алгоритм более эффективен, чем простой минимакс. Запуская поиск с максимальным окном в корне, альфа и бета изменяются в процессе поиска в сторону сближения. Так, при нахождении хода, лучшего, чем предыдущий, альфа меняется на оценку этого хода и все последующие ветви в узле исследуются с этим новым значением альфы. Как часть этого эффекта, с увеличением глубины, альфа и бета последовательно сближаются и теоретически могут сравнятся, показывая, что эта ветвь не ведет к лучшей игре двух сторон и может быть отброшена.

7. Интересным частным случаем является поиск с нулевым окном (не путать с эвристикой нулевого хода). Когда поиск в узле запускается с $\beta = \alpha + 1$. В результате этого, алгоритм вернет α, если все возможные ходы имеют оценку хуже, чем альфа, а β, если существует ход, оценка которого лучше альфа. При этом, сами оценки не уточняются. Этот метод весьма эффективен и применяется в эвристике нулевого хода и алгоритме PVS, одном из улучшений альфа-беты, которые мы рассмотрим позже.

8. При работе альфа-беты критическую роль играет порядок в котором мы рассматриваем ходы в узле. Это настолько непростая тема, что мы ее рассмотрим отдельно.

Исходя из всего сказаного, при корневом вызове алгоритма необходимо задать максимально возможное окно поиска.

```
returned_value = alphabeta(initial_node,-INFINITY,+INFINITY, search_depth)
```

Хочу обратить ваше внимание на один эффект. В разделе об оценочной функции мы пришли к выводу, что чем проще эта функция, тем больше времени остается на перебор. И тем глубже этот перебор сможет просмотреть за одно и то же время. В общем и целом это так, однако поменяв простой минимакс на альфабету, вы, как и я когда то, можете быть удивлены, что простая материальная функция оценки хуже, чем относительно навороченая, включающая более продвинутые критерии отбора. Изучив ситуацию, вы придете к выводу, что в альфабете с навороченной оценкой было произведено гораздо больше отсечений. И это действительно особенность алгоритма - чем больше критериев, выделяющих одну позицию над другой, тем эффективнее будут отсечения. В разумных пределах, конечно.

Предположим, мы даем бонус за центрированность фигур в центре. Альфабета уже в начале поиска может зацепиться за ход, приводящий к такой позиции и любой другой ход, не имеющий достаточной компенсации, не будет рассмотрен. Это достигается сужением окна, что приводит к большим отсечениям.

Тогда как используя оценку только с материалом, это будет происходить крайне редко. Поскольку, выигрыш фигуры, в общем, довольно редкий случай в шахматах, не иначе, чем при грубой ошибке противника.

Глава 12

Некоторые тонкости

12.1 Определение мата

В принципе тут все ясно, если нет легальных ходов при переборе и королю шах, то это и есть мат. Хотя способов это узнать может быть много, в зависимости от представления доски и алгоритмов. Я бы хотел обратить внимание на один подводный камень, который в свое время съел немало моих нервов. Предположим, ход белых, и на определенной глубине найден мат в 5 ходов с победой белых и была возвращена оценка $-INFINITY$, соответствующая самой плохой оценке черных для своей позиции. При форсированном мате, например, эта оценка поднимется до вершины дерева перебора как $+INFINITY$. Белые, соответственно выберут ход с этой оценкой. Противник сделает свой ход и очередь ходить вернется к белым. Этот же мат будет виден теперь как мат в 4 хода. Однако существует опасность, что поскольку горизонт отодвинулся, раньше, чем будет найден этот форсируемый мат в 4 хода, будет найден другой форсируемый мат в 5 ходов и алгоритм переключится на него.

В результате, может создаться ситуация, когда игра зайдет в бесконечный цикл. Решением этой проблемы может служить следующее усовершенствование. При нахождении мата, будем возвращать $-INFINITY + ply$, где ply увеличивается с глубиной. Так, ply в корне дерева равен нулю, а на терминальных узлах максимален. Таким образом, если будут найдены несколько форсируемых матовых вариантов, где игрок проигрывает, то алгоритм перебора предпочтет тот, который длиннее. И, соответственно, если мат противнику, выберем тот, который достигается меньшим количеством ходов.

Если максимальная глубина на которую может спустится поиск, со всеми мыслимыми ухищрениями, равняется max_ply, то теперь мы имеем конечное множество значений, символизирующие матовую ситуацию на доске. Излишне напоминать, что эти оценки не должны пересекаться с обычными нематовыми значениями. Впрочем, если $INFINITY$ достаточно велик, этого не должно произойти.

12.2 Запоминание лучшего хода

При работе рекурсивного алгоритма поиска, возвращается оценка лучшего хода, но не сам этот ход. Что бы узнать что это был за ход есть несколько способов. Самый простой из них, ввести некую булеву переменную для определения корневой глубины. Если при повышении альфы в узле, этот узел является корневым, тогда стоит сохранить этот ход во внешнюю переменную.

Так же полезно при нахождении лучшего хода вывести его и другую полезную информацию в стандартный вывод программы, что бы пользователь или подключенный к программе GUI смог увидеть настоящее состояние алгоритма и то, что он подает признаки жизни. Вообще, корневой узел настолько особенный, что многие программы дублируют функцию поиска для него отдельно. А уже корневой поиск запускает обычный, облегченный вариант поиска.

Можно так-же модифицировать алгоритм поиска таким образом, что по мере поиска будет запоминаться лучшая линия ходов. Этот метод будет нами рассмотрен подробно впоследствии. А имея эту линию, ничего не стоит узнать ее первый ход, который и является лучшим в начальной позиции.

12.3 Эффект горизонта

Если вы запустите поиск до некоторой терминальной глубины, и сравните эти оценки для разных глубин, то увидите, что эти значения имеют тенденцию "прыгать". К примеру, на глубине N вы получите 100, соответствующий выигрышу пешки, но уже на глубине N+1 оценка станет -100. Но стоит увеличить глубину на еще один полуход и вновь получим 100.

Это происходит из за жестко определенной глубины перебора. Будучи остановлен посредине размена, определяется потеря материала, хотя этот материал, возможно, будет восполнен уже следующим ходом. Этот отрицательный эффект так или иначе будет преследовать любую шахматную программу, однако существуют методы его смягчения. Мы рассмотрим некоторые такие эвристики далее.

Эффект горизонта так же приводит к слепоте программы к некоторым глубоким угрозам. Например, если существует форсированный мат на глубине N, то вплоть до этой глубины программа его не увидит. Или, если поиск остановился в позиции с пешкой на 7 вертикали, то программа не увидит превращение пешки в ферзя. К слову сказать, это не значит, что программа не выберет ход, ведущий к форсированному мату, поскольку большая угроза часто сопровождается несколькими более мелкими. Так например, если для того, что бы защитится от мата в 3 сторона жертвует фигуру, это все еще выигрыш, даже если эта жертва является частью форсированной комбинации, ведущей к мату в 5. Остается только надеяться, что оценочная функция сможет правильно оценить угрозу еще до ее реализации.

Решение тут только одно: увеличить глубину перебора. Это замечательно, но время работы алгоритма экспоненциально возрастет, чего мы не можем себе позволить. Поэтому есть два стандартных метода, практически сводящих на нет всю проблему горизонта.

Первый это продления поиска. Мы можем выборочно увеличить глубину перебора для некоторых ветвей дерева, которые считаем опасными. Так например стандартным решением является продление на один полуход на шахах. Это резко увеличивает тактическую силу движка. Так же, опасным является останавливать поиск если оценка безопасности короля резко изменилась. Все эти методы мы по возможности рассмотрим далее.

Однако, совершенно необходимым для любого современного движка является использование форсированного поиска на терминальных узлах, который мы рассмотрим в последствии.

12.4 Фактор ветвления

Фактор ветвления шахматного дерева (branching factor) это величина, характеризующая среднее количество исследованных ходов в каждой шахматной позиции. Понятно, что это наиважнейшая величина для исследования любого дерева. Поскольку любое дерево разрастается со-

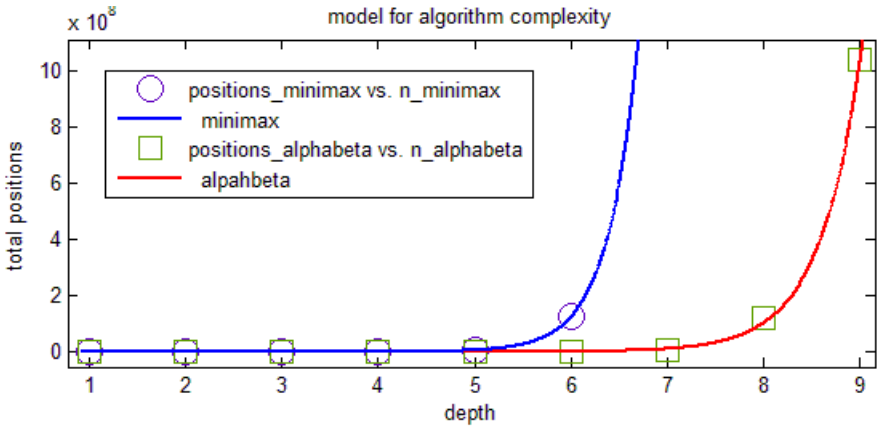

Рис. 12.1: Зависимость просчитанных узлов от глубины поиска

гласно степенному закону, количество просчитанных узлов для глубины n и фактора ветвления b будет равным b^n.

Однако, если в нашей программе есть отсечения, пусть то благодаря альфабете или любой другой природы, этот фактор может уменьшиться. Когда я впервые реализовал алгоритм альфа-бета, с элементарнейшей сортировкой ходов и без форсированного варианта, то провел небольшое исследование, построив график зависимости количества просчитанных узлов от глубины на начальной позиции.

Используя степенную модель, фактор ветвления для минимакса получился равным 22, а тот же фактор для альфа-беты 10. В среднем половина ходов в узле отбрасываются без проверки, что позволяет просчитать на 2 полухода глубже, чем полный поиск. И это не предел! Используя качественную сортировку и всевозможные эвристики можно его уменьшить до 2-3, не теряя качество поиска.

12.5 Алгоритм Зобриста

Допустим, перед нами стоит задача сравнить две позиции на идентичность. Наивным способом будет просканировать все клетки доски на наличие одинаковых фигур, одинаковую очередность ходов, а так же любых других параметров, определяющих позицию. К сожалению, этот метод слишком медленный.

Ранее мы оценивали количество возможных шахматных партий. Попробуем теперь оценить количество возможных позиций. На доске может существовать $4 \cdot 8 = 32$ фигур, которые могут находиться в любой из клеток. Используя уникальность фигур (комбинаторика!) это всего навсего $64 \cdot 63 \cdot 62 \cdot ... \cdot 32 = \frac{64!}{32!}$ позиций. Умные люди возразят, что поскольку не все наши фигуры обладают неразличимым свойством (например две белых пешки) и не все эти позиции легальны, а так-же не учитываются превращения, то эта оценка грубо неверна. Что-же, вы правы, однако эта лишь приближенная оценка. Считаете, что полученное число позиций нужно поделить на 2, 10, 1000, а может умножить на миллион? Уверяю вас, что любое ваше предположение не изменит результат — это количество все еще огромно.

Предположим, мы умеем приводить каждой позиции в соответствие некоторое уникальное число. Тогда сравнивая эти числа можно быстро сказать, являются ли они одинаковыми.

Конечно, на практике это невозможно, поскольку хранить числа такого порядка в памяти компьютера невозможно и не эффективно, а так же непонятен способ их генерации.

Неожиданным может показаться решением использовать хеш функции для идентификации позиций. Такая функция будет приводить каждой позиции в соответствие некоторый ключ - 64-битное число, которое и будет использоваться для сравнения позиций.

Читатель может возразить, что подобное соответствие не будет единственным: теоретически будут существовать классы позиций, имеющих один ключ. Задача хорошей хеш функции обеспечить как можно меньшую корреляцию даже между очень близкими позициями и их хеш ключами. Тогда вероятность коллизии ключа в переборе можно свести к минимуму. И на самом деле, алгоритм, который мы предлагаем полностью удовлетворяет всем условиям:

1. Он позволяет быстро вычислять хеш ключ, обновляя его инкрементно от позиции к позиции.

2. Он практически не создает коллизий, на практике их вероятность равна нулю.

3. Сравнение позиций на идентичность сводится к сравнению двух чисел на равенство.

Алгоритм Зобриста состоит в следующем. При запуске программы инициализируется

```
BIT random_table[PIECE_ID_MAX][BOARD_SIZE][BOARD_SIZE][2];
BIT black_turn_random;
```

и заполняются 64 битными случайными числами. Для вычисления хеш ключа позиции сканируются все фигуры, стоящие на доске, и для каждой вычисляем

```
random_table[тип фигуры][координата y][координата x][цвет фигуры]
```

Теперь все эти числа мы последовательно объединяем операцией XOR. Под конец делаем XOR с числом black_turn_random, если ход черных и в результате имеем хеш ключ позиции. Выбор XOR обеспечивается тем, что эта операция быстро выполняется на уровне команд процессора, она обратима, коммутативна, ассоциативна. Более того, ее обратная функция равна самой себе

$$(A \oplus B) \oplus B = A$$

Отсюда проистекает возможность избирательно добавлять и убирать фигуры на уровне хеш ключа, не пересчитывая его заново. Так, смена очередности хода также требует всего лишь произвести XOR с числом black_turn_random.

Эффективная работа алгоритма требует использование качественного генератора случайных чисел. Иначе возможны частые коллизии. Не рекомендуется использовать генератор, поставляемый со стандартной библиотекой C/C++. В своей программе я использовал 64 битный Mersenne Twister.

Если вы реализовали в своей программе поддержку Зобриста и проверка показывает наличие частых коллизий хеш функции, то скорее всего в ваш код вкралась ошибка. В среднем используя 64 бита для ключа должна быть одна коллизия в сутки непрерывной работы алгоритма или около того. Используя 32 битный ключ, коллизии будут случатся чаще, однако и тут они слабо влияют на лучший ход (смотриRobert Hyatt and Anthony Cozzie (2005). The Effect of Hash Signature Collisions in a Chess Program. ICGA Journal, Vol. 28., No. 3.).

12.6 Определение повторов

Широко известно правило, что если одна и та-же позиция встречается в игре 3 раза, то партия считается ничейной. Если мы запустим наш движок против других движков, очень часто можно наблюдать картину, когда единственный шанс спастись от разгрома это, например, объявить вечный шах королю противника. Что бы научить движок обходить такие вечные шахи и, напротив, если это выгодно, стимулировать их использование, необходимо научить программу находить повторения позиции. Причем, имеются ввиду только позиции в одной цепочке ходов. Этой позиции должна быть присуждена ничейная оценка 0 и это значение должно быть возвращено выше по дереву.

Идея проста: если в определенной вариации, лучшая линия содержит цикличность, приводя через несколько ходов к такой же позиции, что уже встречалась, то даже если мы увеличим поиск на сотню полуходов, эта цикличность будет сохранятся. Так не проще ли обрезать всю эту линию, дав ей ничейную оценку.

Так, помимо основной задачи выявления повторений, мы еще отсекаем немало заранее неперспективных ветвей, что не может не радовать.

Что бы решить эту задачу, необходимо найти способ быстро и компактно сохранять и сравнивать позиции. Проблема в том, что любой прямой метод будет слишком медленным для использования в поиске, решением будет использовать хеширование позиции методом Зобриста. Для каждой позиции вычисляется зобрист-ключ и записывается в специальный массив, где индексом будет служить текущая глубина. Так, что мы будем иметь зобрист ключи для всех предыдущих позиций, приведших к оной вплоть до корня дерева.

Теперь не составит труда циклом проверить не предыдущие позиции, а их хеш ключи на равенство с хеш ключом настоящей позиции и если есть хоть одно равенство, смело даем ей статус повтора.

Что бы повысить эффективность этого метода, можно помимо массива хеш ключей хранить глубину, на которой было произведено последнее в этой цепочке ходов взятие или ход пешки. Любой такой ход делает позицию необратимой: никакие ходы уже не смогут привести к повторению любой позиции до его совершения. Поэтому логичным будет проверять позиции на повторение не с корневой глубины, а именно с глубины, на которой был совершен такой ход.

Эту переменная замечательна еще тем, что с ее помощью можно реализовать правило 50 ходов. Еще один пример хорошей интеграции и экономного использования ресурсов.

Глава 13

Форсированный поиск

Допустим мы написали программу, способную считать за приемлемое время на 8 полуходов. Что бы уменьшить эффект горизонта, нельзя останавливать поиск посреди разменных комбинаций, а так же среди комбинаций, где форсированные взятия теряют или выигрывают материал.

Идея в том, что бы написать еще одну, облегченную функцию поиска, подобно алгоритму Альфа-бета, которая будет рассматривать только взятия. Соответственно, когда взятий не останется, она будет возвращать статичную оценку. Поэтому, полезно иметь в программе отдельный генератор ходов только для взятий.

Взятий гораздо меньше, чем всех ходов в целом, поэтому мы не ожидаем большого разрастания дерева. И тем не менее, при спешной реализации ФВ вы вполне можете получить тормозную функцию, не оправдывающуя своего создания. Правильная реализация должна быть практически незаметна на фоне основного поиска.

Приведем псевдокод такой функции

```
1  function integer forced_alphabeta(node,alpha,beta)
2   val = the heuristic value of node for current player
3   if(val >= beta)
4     return beta;
5   if(val > alpha)
6     alpha = val;
7
8   for child in node: #only captures
9     alpha = max(alpha, -minimax(child,-beta,-alpha, depth-1))
10    if(alpha>=beta)
11       return beta;
12  next for
13  return alpha
14 end function
```

Первое, на что необходимо обратить внимание это то, что поиск не ограничивается глубиной. Взятия необходимо рассматривать все до конца, иначе мы вернемся к тому же с чего начали - проблеме с горизонтом. В общем, это обыкновенный альфа-бета поиск, как мы его определили ранее за исключением двух условий в начале функции.

Насчет первого условия

```
if(val >= beta)
   return beta;
```

то тут все просто. Если статическая оценка позиции больше, чем бета, то вся ветвь отрезается. Если при переборе обнаружится, что есть ход, чья оценка больше, чем бета, то этот ход все равно обрежется. Если же будет найдено взятие, повышающее альфу, то оно все равно будет хуже, чем статичная оценка изначальной позиции, поэтому тоже обрежется.

Второе условие несколько сложнее

```
if(val > alpha)
   alpha = val;
```

Оно уменьшает окно поиска, запрещая рассмотрение неинтересных взятий, которые в итоге ведут к ухудшению статичной оценки.

Без этих условий размены будут рассматриваться до конца. Учитывается тот факт, интуитивно используемый людьми, что размен может быть прекращен в любой момент, если это не выгодно ходящей стороне. Действительно, если на некой терминальной глубине запускается ФВ, то нет смысла рассматривать жертву ферзя, если она, конечно, не ведет к еще лучшей оценке.

Существует нерешенный до конца вопрос о том, стоит ли помимо взятий в ФВ рассматривать так же тихие ходы, приводящие к шахам? А так же стоит ли загружать ФВ другими эвристиками, которыми мы нашпигуем в будущем основной алгоритм поиска?

Что бы попытаться ответить на этот вопрос следует вспомнить зачем мы используем ФВ и в чем его отличие от основного поиска. ФВ это НЕ поиск и его задача не в нахождении тактических маневров, а в том, что бы сгладить эффект горизонта, приведя позицию к состоянию, когда безопасно вызвать оценочную функцию. Поэтому ФВ должна быть максимально легковесной, что бы дать основному поиску больше времени на размышление.

Насчет включения шахов в генератор ходов ответ так же отрицателен. Исключение составляют только программы с полностью легальным генератором ходов, когда шахи могут быть легко определены. Поскольку архитектура моего движка содержит псевдо-легальный генератор, шахи не включаются.

Глава 14

Протокол UCI

Это стандартный протокол связи между движком и графическим интерфейсом. Уже с первых этапов тестирования движков он позволяет не ограничиваться консолью для управления движком и отображения его вывода, а использовать один из уже написанных профессиональных GUI. Так же обычно GUI имеет инструменты для работы с позициями: расстановкой позиций, загрузка и сохранение. Что незаменимо для отловки багов на начальном уровне. Но настоящее преимущество подключение движка к оболочке заключается в игре с другими движками, благо протокол UCI поддерживают движки на любой вкус и цвет. Поэтому обязательно необходимо реализовать этот протокол[1] когда движок научится делать более менее осмысленные ходы.

Дадим пример популярных оболочек

1. Arena. Стандарт де факто, наиболее богатая функционалом. Из минусов: раздражает плохая русификация. Отовсюду вылазят немецкие слова.
2. Chessbase. Целый комплекс проприетарных программ поддерживающих установку UCI движков
3. Jose. Неплохая оболочка на Java. Из минусов: невозможность проводить партии между несколькими движками.

Для связи используется стандартный буфер ввода/вывода. Это означает, что для отправки информации можно использовать printf в C и cout в C++. После каждой отосланной команды нужно отправить символ конца строки. Для приема команд, можно использовать стандартные scanf в C и cin в C++, однако они обладают серьезным недостатком. Их вызов останавливает программу до того, как оболочка не пошлет что-то. Понятно, что так мы не сможем параллельно выполнять вычисления и быть готовыми обработать команду от GUI, если она поступит. Решением будет использовать некоторые нестандартные функции для проверки того, если в буфере потока есть команды, а уже после считать их.

Мы не будем перечислять все возможные команды в протоколе, а ограничимся тем набором, которого, на мой взгляд, достаточно для комфортной работы в оболочке.

При запуске движка, он должен произвести как можно быструю инициализацию внутренних параметров и ждать указаний.

Движок ждет команды

[1] или одну из его альтернатив, например Winboard

```
uci
```

говорящей о попытке оболочки подключится к нему.

Приняв эту команду он должен назвать себя

```
id name put_your_engine_name_in_here
id author my_name
```

И закончив внутреннюю инициализацию, подтвердить, что он действительно UCI движок и готов к работе. Для чего он посылает оболочке команду

```
uciok
```

Теперь движок готов получать команды от оболочки и выполнять их. Необходимо учитывать, что в любой момент оболочка может запросить с ним синхронизацию. Для этого она шлет команду

```
isready
```

движок должен ответить

```
readyok
```

При этом не предполагается, что движок должен прервать или как то изменить свою работу, синхронизация нужна только для проверки того, что связь между ним и оболочкой установлена и любая последующая команда поступит на выполнение незамедлительно[2].

Перед началом поиска оболочка должна послать позицию для анализа. Посылается команда position, за которой следует или предикат startpos, говорящий, что требуется установить начальную шахматную позицию, или предикат fen, с последующей строкой, описывающий позицию в FEN формате. За этим может следовать предикат move с последующей серией ходов, записанных в длинной алгебраической нотации. Движок должен выполнить их относительно позиции, которую пользователь установил стартовой.

- position
  - startpos

    ```
    position startpos
    ```

  - fen

    ```
    position fen ...
    ```

  - moves

    ```
    position startpos moves e2e4 e7e5
    ```

Обычно за этим последует команда go. Она должна запустить поиск, причем параметры поиска должны настраиваться последующими предикатами.

[2]Например, пользователь задал путь к таблицам Налимова. Поскольку скорость работы с диском маленькая, может потребоваться несколько секунд для выполнения подключения. Оболочка не должна посылать в это время другие команды.

- go
  - infinite. Бесконечный поиск, пока его не остановит пользователь.

    ```
    go infinite
    ```

  - depth. Ограничение по глубине.

    ```
    go depth 8
    ```

  - movetime. Время на ход в миллисекундах. По истечению поиск прерывается и движок должен немедленно выдать ход, который он считает лучшим.

    ```
    go movetime 1000
    ```

Принято, что бы поиск время от времени подавал признаки жизни, посылая оболочке данные о своем внутреннем состоянии. Делается это командой info, за которой следуют желаемые предикаты с нужной информацией.

- info
  - depth. Глубина, на которой производится поиск в данный момент.
  - score
    - cp. Оценка в сотых долях пешки
    - mate. Матовая оценка. Количество ходов до мата должно указываться в целых ходах, а не в полуходах.
  - pv. Движок сообщает, что нашел лучшую линию. Команда обычно совмещается с оценкой и глубиной

    ```
    info score 12 depth 3 pv e2e4 e7e5 b1c3
    ```

  - nodes. Общее количество просчитанных узлов в поиске на данный момент
  - nps. Среднее количество перебираемых узлов в секунду
  - hashfull. Заполненность хеш таблицы в единицах промилле[3]. Оболочка обычно переводит и показывает заполненность в процентах.

Когда движок решит, что за ход он хочет сделать в позиции, он должен объявить это оболочке командой bestmove

```
bestmove e2e4
```

Программа должна быть готова к тому, что GUI в любой момент решит прервать поиск. Для этого движку посылается команда stop

```
stop
```

[3]От нуля при пустой таблице, до 1000 при полной

Он должен моментально перехватить эту команду, остановить поиск и сделать ход, который он считает лучшим на этот момент.

Многие движки так-же дают пользователям возможность менять свои настройки прямо из оболочки. Для этого при инициализации, до того, как движок отошлет оболочке uciok, он может послать ей название необходимой опции и ее свойства, например

```
option name Max Depth type spin default 8 min 0 max 30
```

Приведем полный список предикатов, которые могут следовать за командой option

- name . Задает имя свойства.
- type. Тип значений, которые свойство может принимать
  - check. Может принимать значения true и false.

    ```
    option name Nullmove type check default true
    ```

  - spin. Может принимать числа в заданном интервале значений

    ```
    option name Selectivity type spin default 2 min 0 max 4
    ```

  - combo. Выпадающий список с некоторым количеством предопределенных слов.

    ```
    option name Style type combo default Normal var Solid var Normal var Risky
    ```

  - button. Кнопка, указывающая движку сделать некоторое действие.

    ```
    option name Clear Hash type button
    ```

  - string. Устанавливает произвольную строку.

    ```
    option name NalimovPath type string default c:\
    ```

- default. Устанавливает стандартное для движка значение опции.
- min. Минимальное значение, которое может принимать поле.
- max. Максимальное значение, которое может принимать поле.
- var. Устанавливает предопределенные значения для полей.

Если GUI решит изменить одну из опций, он может сделать это командой setoption, за которой следует ее название. Для указания нового значения используется предикат value

```
setoption name Nullmove value true
setoption name Selectivity value 3
setoption name Style value Risky
setoption name Clear Hash
setoption name NalimovPath value c:\chess\tb\4;c:\chess\tb\5
```

Имейте ввиду, что оболочки имеют тенденцию сохранять значения опций с прошлой сессии поиска и навязывать их без предупреждения в новой.

Глава 15

Эвристики

Формально, эвристика это небольшое усовершенствование переборочного алгоритма, порой математически необоснованное, которое тем не менее оказывает положительное влияние на качество игры программы.

До сих пор, мы имели дело только с математически обоснованными усовершенствованиями, к которым относится альфа-бета алгоритм. Он гарантировано дает такой-же лучший ход и его оценку, как и не использующий альфа-бета отсечения негамакс. Однако, если бы все было так просто, программы бы и сегодня не могли бы составить конкуренцию лучшим шахматным игрокам. Задача эвристик, порой в ущерб точности и скорости отдельных алгоритмов, используя интуитивные шахматные рассуждения, ускорить перебор и тем самым успеть просчитать позицию на большую глубину.

Здесь стоит прояснить следующее. Можно проводить сколь угодно изощренные оптимизации кода, шлифовать до безумия генератор ходов и оценочную функцию, но не получить столь желаемого усиления. И напротив, возможно, написать «умный» код, который согласно неким представлениям обрезает целые ветви дерева, считая их с большой вероятностью неперспективными. Поскольку алгоритм перебора экспоненциален в своей сложности, это может сократить намного размер дерева и тем самым позволит увеличить глубину перебора. Отсюда вывод — отсечение ветвей потенциально намного эффективнее в плане усиления силы движка, чем любые возможные оптимизации кода.

Однако, за все надо платить и в результате подобных отсечений можно выкинуть и блестящие шахматные комбинации. Приведем пример. В миттельшпиле, когда все фигуры мобильны и идет острая тактическая игра, как правило на доске еще остаются много пешек, чей ход, мягко сказать, неразумен в данной ситуации. Представьте себе стоящие на обочине доски пешки; они могут ходить, но их время еще не пришло и наступит ближе к финалу. Так не правильней ли на время забыть про них и сосредоточится на мобильных, находящихся в центре событий, фигур. Это сокращение активных фигур резко скажется на величине дерева перебора и, возможно, полученный в результате прирост к глубине 1-2 полуходов даст решающий перевес для победы.

Все это заманчиво, однако попробуйте доказать это французскому гению Филидора, мастеру восемнадцатого века, утверждавшему, что пешки — душа партии, а фигурам доводится вспомогательная роль.

Итак, поняв что такое эвристика, перечислим некоторые из используемых в сильнейших программах.

15.1 Сортировка ходов

Из за своих особенностей, альфа-бета алгоритм тем эффективнее, чем раньше при переборе ходов будет выбран сильнейший. Это происходит из за того, что найденный первым хороший ход увеличит альфу, сужая тем самым окно поиска. И последующие ветви будут исследоваться с гораздо более частыми отсечениями.

Если каким то чудом во всех узлах первым будет исследоваться лучший ход, мы получим теоретически минимальное дерево. Математические исследования алгоритма альфа-бета дают точную формулу для его размера. Если фактор ветвления для дерева постоянен и равен b, то количество просчитанных узлов для глубины n будет равен

$$b^{\lceil \frac{n}{2} \rceil} + b^{\lfloor \frac{n}{2} \rfloor} - 1$$

, где $\lceil x \rceil$ это округление x до ближайшего целого, большего x, а $\lfloor x \rfloor$ соответственно округление до ближайшего целого, меньшего x. Это теоретическая граница, которую может нам дать идеальная сортировка. И это немало. Заметим, что в этой модели фактор ветвления постоянен, изменяется только функция - представленная выше формула дает ее нижнюю границу, тогда как в модели, рассмотренной нами раньше любые сокращения уменьшали фактор ветвления, оставляя формулу неизменной.

Поскольку, без конкретного перебора нельзя узнать какой ход сильнейший, мы можем использовать статистические соображения и другие ухищрения, что бы заранее “угадать” хороший ход. После этой процедуры, когда различным ходам назначаются числовые оценки в зависимости от перспективности, необходимо отсортировать массив ходов, что бы лучшие ходы были рассмотрены первыми. Конечно, сортировка несколько замедлит перебор, но эффективные сокращения сторицей окупят эту потерю.

Все сортировочные эвристики условно можно разделить на две категории:

- Первая использует статичные соображения, порой довольно сложные, которые не используют предыдущий поиск для улучшения сортировки
- Вторая использует сам поиск для накопления информации и ее анализ используется для сортировки последующих узлов.

В первую очередь при написании шахматной программы необходимо реализовать эвристики первой категории и воздержатся от эвристик второй категории, пока вы не уверитесь, что код работает без ошибок и вы интуитивно понимаете что происходит внутри алгоритма поиска. Поверьте, непросто понять, а еще хуже отлаживать рекурсивный альфа-бета алгоритм.

Различные эвристики сформируют категории, где сортировкой внутри этих категорий руководят сами эти эвристики, а между категориями существует связь больше-меньше. Некоторые эвристики выявляют более важные ходы и должны быть рассмотрены первыми, а некоторые менее и следуют после них.

15.2 LVA/MVV

Простейшая эвристика для сортировки взятий, в названии которой и зашифрован ее основной принцип: Less Valuable Atacker / Most Valuable Victim[1]. Смысл этой эвристики в том, что бы

[1] Менее Ценный Атакующий / Наиболее Ценная Жертва

предугадать какое из возможных взятий лучше, без сообственно перебора, а основываясь только на статической информации, как то материальная оценка фигур

Во первых, ходы, отыгрывающие больший материал лучше, чем ходы отыгрывающий меньший материал. Это конечно условно, поскольку можно придумать множество позиций, где это правило нарушается. Однако, при переборе обрабатываются миллионы взятий и в среднем это правило оправдывает себя.

Во вторых, мы делаем попытку учитывать безопасность этого взятия. Так, если мы можем на выбор побить фигуру ферзем или конем, то безопасней это сделать конем, так как при возможном ответном взятии потерять коня предпочтительнее, чем потерять ферзя.

Математически, мы должны сортировать ходы по убыванию оценки взятой фигуры, а в случае, если существуют несколько ходов, отыгрывающих равный материал, то мы их повторно сортируем, уже по возрастанию оценки бьющей фигуры. Таким образом и реализуется принцип, что первыми идут те взятия, которые самой слабой фигурой бьют наиболее сильную. Другими словами, взятия, отыгрывающие максимальный материал.

Програмно, достаточно сортировать по значению $K \cdot S_{from} - S_{to}$, где S_{from} и S_{to} оценки бьющей и битой фигур соответственно, а K - большое число, достаточное, что бы категории не пересекались[2].

Бьющая фигура	Битая фигура	Програмная оценка
Пешка	Ферзь	$2^{10} \cdot 1000 - 100 = 1023900$
Конь или Конь	Ферзь	$2^{10} \cdot 1000 - 300 = 1023700$
Ладья	Ферзь	$2^{10} \cdot 1000 - 500 = 1023500$
Ферзь	Ферзь	$2^{10} \cdot 1000 - 1000 = 1023000$
Пешка	Ладья	$2^{10} \cdot 500 - 100 = 511900$
Конь или Слон	Ладья	$2^{10} \cdot 500 - 300 = 511700$
Ладья	Ладья	$2^{10} \cdot 500 - 500 = 511500$
Ферзь	Ладья	$2^{10} \cdot 500 - 1000 = 511000$
...		
Пешка	Пешка	$2^{10} \cdot 100 - 100 = 102300$
Конь или Слон	Пешка	$2^{10} \cdot 100 - 300 = 102100$
Ладья	Пешка	$2^{10} \cdot 100 - 500 = 101900$
Ферзь	Пешка	$2^{10} \cdot 100 - 300 = 101400$

Таблица 15.1: Как мы видим, LVA/MVV максимизирует выгоду от взятия и минимизирует вероятность потери от ответного взятия

Все замечательно, но как будут рассмотрены взятия королем? Оценка короля чисто формальна, так как короля нельзя потерять. Долгое время в моей программе, как дань традиции, она была равна очень большому числу. Следовательно, согласно вышеозначенному принципу взятия королем имели наименьший приоритет среди равных взятий. После, я обнулил оценку короля, что бы дать им больший приоритет. Идея проста: если уже дошло до того, что король может бить, это должен быть статистически важный ход. Данная модификация в среднем немного увеличила количество отсечений.

[2]Поскольку в моей программе максимальную материальную оценку имеет ферзь и она порядка 1000 сотых пешки, достаточно взять $(S_{from} \ll 10) - S_{to}$, где под $\ll$ подразумевается операция побитового сдвига.

В процессе работы над движком вы можете корректировать оценки фигур, например разумно дать слону оценку чуть больше, чем коню. Таким образом, вы влияете на сортировку, что не обязательно благотворно скажется на отсечениях. Поэтому есть идея использовать не сами оценки фигур, а другие оценки, добавляющие симметрию и не зависящие от материала. Например

Пешка - 1, Конь - 2, Слон - 2, Ладья - 3, Ферзь - 4, Король - 0 или

Пешка - 1, Конь - 3, Слон - 3, Ладья - 5, Ферзь - 9, Король - 0 .

Я опробовал обе эти схемы и остановился на последней.

Сортировка ФВ в погоне за быстродействием использует для сортировки взятий только LVA/MVV и очень хорошо себя зарекомендовала. Однако в основном поиске существуют еще тихие ходы, для которых эта схема не имеет смысла. Поэтому, для не взятий нужно использовать другие эвристики, которых немало. Многие из них мы рассмотрим далее.

Простейшая схема, которую можно использовать на начальных стадиях разработки движка, по аналогии с LVA/MVV можно сортировать тихие ходы по убыванию материала ходящей фигуры. Так, первыми будут рассмотрены ходы ферзя, потом ладьи, слона/коней и пешек в конце.

Как общее правило, необходимо первыми рассматривать взятия, а уже после тихие ходы. Это правило мы еще уточним. Оно работает отлично, но некоторые другие эвристики могут давать приоритет некоторым ходам над взятиями, руководствуясь другими соображениями.

15.3 Ходы-убийцы

Очень эффективное улучшение для сортировки тихих ходов, основанное на свойствах альфабета алгоритма. Поскольку отсечения тем эффективнее, чем раньше будет проверены хорошие ходы, то есть идея первыми проверять те из них, которые являются лучшими на этой же глубине поиска. Обычно, у противника есть один — два хода, которые игрок должен всячески предотвратить и которые являются частью плана противника. Такие ходы, которые является лучшими на одной глубине поиска, с большой вероятностью будет хорошим на этой же глубине в другой части дерева. Поэтому полезно для каждой глубины сохранять два хода которые проверяются первыми и заменять эти ходы на более сильные по мере поиска.

В эту эвристику записываются только тихие ходы и только если этот ход отсекся по бете. Тогда мы переписываем первый киллер в ячейку для второго киллера, а на место первого киллера записываем новоиспеченный ход. Необходимо только убедится, что эти два хода не будут равны, в таком случае мы неэффективно используем два слота, что нежелательно.

При сортировки первый киллер имеет больший приоритет над вторым киллером.

Эта эвристика вполне может уменьшить дерева перебора на 10-20 процентов.

Как идея, можно для каждого из игроков хранить свою таблицу киллеров, поскольку хорошие ходы для одной из сторон как правило бессмысленны для второй. Требует дополнительного исследования.

15.4 Эвристика истории

Эвристика для сортировки тихих ходов.

Идея очень схожа с эвристикой ходов убийц, за тем исключением, что сохраненные ходы не привязаны напрямую к глубине поиска. Идея в том, что бы создать массив 64x64, где первый

индекс показывает на клетку, с которой производится ход, а второй на клетку, куда производится ход. До начала поиска мы обнулим этот массив и будем прибавлять некоторое число для каждого тихого хода, который отсекся по бете в поиске. Хорошие ходы отсекаются по бете в среднем чаще плохих, поэтому таблица будет хорошим индикатором перспективности тихого хода. Все оставшиеся в узле тихие ходы, по какой то причине не выделенные другими эвристиками, лучше сортировать по убыванию этих весов.

Остается открытым вопрос о том, какое число нужно добавлять в таблицу. Заметим, что в отличии от эвристики ходов-убийц, теперь все ходы в таблице равны, однако понятно, что ход, который отсекся ближе к корню дерева более важен, чем аналогичный ход, на большей глубине. Тому есть два соображения:

- С каждой глубиной качество ходов, которые оказываются лучшими в поиске падает по вполне объективным причинам. Понятно, что поиск с глубиной 7 даст лучший ход, чем тот же поиск на глубине 2. То же относится к поиску в одном дереве, где на максимальной глубине поиск руководствуется только статичной оценкой, без учета тактики.

- Дерево разрастается по степенному закону, поэтому высока вероятность, что один и тот же ход на большой глубине множество раз запишется в таблицу, тогда как лучший ход у корня запишется гораздо меньше раз. Поэтому, таблица будет захламляться ходами с большой глубины, которые вырвутся в лидеры только по причине разрастания дерева.

Можно сделать добавляемое значение зависящим от глубины, так что ходы с меньшей глубины получат больший вес. Так, есть несколько стратегий того, какова именно должна быть зависимость.

1. $val = val + depth$
2. $val = val + depth^2$
3. $val = val + b^{depth}$, где b некоторое число, обычно 2

Все эти схемы эмпирические и должны тестироваться методом проб и ошибок. Я же выбрал первую, с небольшим дополнением, что узлы с $depth \leqq 2$ не имеют права записывать в таблицу истории. Хотя, конечно, здесь еще много материала для исследования.

Когда в таблицу записывается множество раз один и тот же ход, его оценка может стать слишком большой и это приведет, что этот ход переползет в другую сортировочную категорию(см. главу 16). Для этого рекомендуется следить за максимальным значением в таблице и если оно превышает некоторое пороговое значение, делить все элементы таблицы на 2 с помощью команды битового сдвига rightshift. Благо, это будет происходить крайне редко.

Есть идея, которую стоит попробовать. Для близких позиций, например тех, что следуют друг за другом в партии, хорошие и плохие ходы в общем не сильно меняются. Поэтому можно не обнулять таблицу истории, а делить ее на некое число, так что бы начинать новый поиск с информацией из предыдущего хода. Число должно быть достаточно большим, что бы информация достаточно быстро перекрывалась новыми данными и не мешала новому анализу. И достаточно маленьким, что бы перекрытие происходило не быстрее, чем эта полезная информация была использована максимально эффективно.

Есть мнение, что поскольку для белых и черных хорошие ходы редко будут одинаковыми, можно завести и использовать две таблицы истории для каждой из сторон соответственно. Идея требует проверки.

15.5 Хеш таблица

Шахматное дерево содержит множество одинаковых позиций, полученных разными путями. И каждый раз, встречая такую позицию, для нее заново запускается поиск. Было бы хорошо сохранять информацию об уже изученных позициях, тогда повторное изучение смогло бы быть более эффективным. Здесь, как и прежде мы стоим перед задачей идентификации позиции по некому ключу и уже описанный алгоритм хеширования Зобриста хорошо подойдет. Прелесть заключается в том, что мы используем тот же механизм, что и при определении повторов в линии, описанному нами в секции 12.6.

Здесь нужно решить несколько проблем: способ, которым мы будем сохранять информацию и какую именно информацию стоит сохранять.

Можно придумать много схем, но, как показывает практика, не все они одинаково эффективны.

Первым делом я реализовал хеш таблицу на основе списка отображений на основе стандартного в C++ шаблона map<BIT,hash_table_el>. Где hash_table_el это элемент сохраненной информации. Его преимуществами являются экономное использование памяти, возможность использовать полный хеш ключ для индексации позиции, а недостатками медленная скорость за счет постоянной аллокации памяти и логарифмического поиска. Этот способ не столь быстр, но заслуживает упоминания.

Второй способ, очень быстрый, но требующий больше памяти заключается в создании массива длинной 2^n элементов типа hash_table_el. Но, если мы захотим использовать весь 64 битный зобрист ключ как индекс массива, полученный массив будет содержать 2^{64}элементов, что не поместится в любую оперативную память.

Однако вместо того, что бы использовать весь ключ, можно использовать только n правых его бит. Определим mask как 64 битное число, у которого только n правых бит в двоичном представлении равны единице, остальные же обнулены. Тогда индекс в хеш таблице будет вычисляться очень просто

```
index = hash_key & mask
```

Можно подобрать n, что бы массив не был слишком большим. Я использую $n = 22$, размер массива примерно 100 MB.

Используя эту методику, мы провоцируем еще большие коллизии, поскольку теперь длина хеш ключа, идентифицирующего ячейку в таблице фактически равна n бит. Способ обойти это, хранить в самой ячейке полный 64 битный зобрист ключ, и при использовании таблицы сравнивать его с полным ключом изучаемой позиции. Будем использовать информацию из таблицы только если полные ключи совпадут, а не только если совпадают их младшие n бит. Так же можно условиться, что если сохраненный внутри ячейки полный хеш ключ равен нулю, то данная хеш ячейка не используется.

Очевидно, что чем больше длина такого хеш массива, тем реже будут появляться эти вторичные коллизии. Так, если n небольшой, то различные позиции чаще будут перезаписывать друг друга в одну и ту же ячейку и информация о поиске может быть потеряна. Для большого n будет использовано больше памяти, но коллизии будут реже. Соответственно, данные в них будут более качественные и надежные. Необходимо найти баланс между этими двумя подходами.

Заметим, что используя этот метод, нельзя в общем случае, задать произвольный обьем используемой программой памяти, подстраивая ее под обьем свободной оперативной памяти

компьютера. Однако, подобные попытки довольно спорный способ увеличить силу игры, за счет уменьшения вторичных коллизий. Поскольку каждое последующее удвоение памяти под хеш тоблицу будет давать все меньший прирост к отсечениям.

Так какую же полезную информацию о поиске мы будем хранить в ячейке?

- Лучший найденный ход в позиции
- Глубина, с которой осуществлялся перебор
- Оценка, полученная в результате перебора
- Режим оценки

Сортировка ходов Во первых, первейшая задача хеш таблицы - способствовать сортировке ходов. Если для некой позиции был получен лучший ход, то встретив эту позицию в другой части дерева, этот ход необходимо рассмотреть первым. После изучения узла, с выявлением в нем лучшего хода или другой полезной информации, мы запишем их в таблицу, если соответствующая ячейка удовлетворяет определенным правилам

Стратегия перезаписи Глубину в хеше мы храним, что бы оценить качество содержащейся в нем информации. Если мы повторно исследовали позицию и обнаружилось, что лучший ход изменился, то мы позволим себе переписать его в таблице только если глубина нового поиска больше или равна глубине, сохраненной в хеш таблице. При записи, мы конечно обновим эту глубину.

Может случится, что из за вторичной коллизии индексов просчитанный ход должен заменить информацию об анализе другой позиции. Тогда при перезаписи ценная информация предыдущего поиска будет потеряна. Вопрос о выявлении условий, когда стоит переписать, а когда сохранить уже существующий ход, требует дополнительных исследований и не до конца закрыт. Мы же и здесь придерживаемся самой простой политики, перезаписывая всегда, если глубина нового поиска больше или равна глубине старого поиска, даже если он не соответствует одной и той же позиции.

При использовании этой политики, со временем, в среднем, ходы в хеш таблице будут заполнятся все более надежными ходами, просчитанными с большей глубиной.

Оценка и ее режим Помимо лучшего хода, мы еще храним оценку предыдущего поиска. А именно переменная mode, сохраненная внутри ячейки содержит указание на природу этой оценки

- mode = INSIDE, score содержит точную оценку, полученную в границах альфа - бета окна
- mode = BIGGER_EQ, score отрезалась по бете. Можно быть уверенным, что настоящая оценка больше или равна score
- mode = LOWER_EQ, score отрезалась по альфе. Можно быть уверенным, что настоящая оценка меньше или равна score

На основе этой информации корректируются границы альфа-бета окна, как в случае с BIGGER_EQ, LOWER_EQ или буквально возвращается оценка, как в случае с INSIDE.

Эта информация релевантна только с учетом глубины поиска. Так, безопасно использовать эту информацию только если глубина в хеш таблице больше или равна настоящей глубине в поиске. И это в отличии от лучшего хода, который используется всегда, вне зависимости от глубины.

Оценка в режиме LOWER_EQ не предполагает наличие лучшего хода. От этого страдает сортировка ходов, особенно если такая ячейка замещает ячейку с лучшим ходом. Поэтому мы запрещаем такую перезапись даже если она удовлетворяют всем другим условиям.

В общем, хеш таблица это важнейшая эвристика, способная намного уменьшить фактор ветвления дерева. Она математически безопасна при правильном использовании, хотя следует понимать, что лучшие ходы и оценки, получаемые с включенной хеш таблицей могут отличаться от ходов и оценок без нее. Дело в том, что повторное использование анализа фактически приводит к тому, что некоторые ветви эффективно рассматриваются с большей глубиной, делая анализ более надежным. При включенной хеш таблице может статься, что поиск на глубине в N полуходов найдет форсированный мат на глубине большей N, куда поиск в принципе не мог добраться.

В любом случае, интегрируя в свою программу хеш таблицу, убедитесь, что вы понимаете природу всех подобных эффектов. И что они не следуют из ошибок и плохого понимания написанного кода.

Некоторые современные программы вольно обращаются с данными из хеш таблицы, не утруждая себя обнулить ее при переходе к новой позиции. Это может быть полезно, если программа играет партию и от хода к ходу позиция меняется плавно. Тогда окажется, что львиная доля позиций была изучена уже на предыдущем ходу. Подобная стратегия может привести к переполнению хеш таблицы и новый поиск будет менее эффективен. Нужно провести эксперименты в этом направлении.

15.6 Нулевой ход

Проведем мысленный эксперимент: Вы играете в шахматы и ваша очередь ходить. Представьте, что против правил, вы отказываетесь от хода и передаете его противнику.

Вопрос: Что выгодно мне как игроку? Пропустить ход или нет?

- Какой глупый вопрос - скажете вы - конечно передавая ход противнику я делаю себе хуже, ведь два подряд хода моего противника не дают мне возможности вести контригру против него или банально защищаться от его возможных ходов. Другими словами, целый ход я абсолютно беспомощен!

Не спешите. Вы правы в подавляющем количестве случаев, но ради математической точности заметим, что существует ряд позиций, когда любой мой возможный ход немедленно приводит к проигрышу. Поэтому в таких случаях отказаться от хода выгоднее, чем сделать его. Такие случаи настолько редки и происходят, как правило, в эндшпиле, что даже получили особое название: цунгцванг. Мы пока что будем рассматривать правило, на время забыв об этих редких исключениях.

Переведем наш диалог на язык программирования. В неком узле есть альфа и бета и мы готовы рекурсивно проверять оценки всех возможных ходов. Начиная с альфы, уже гарантированной оценки, мы будем пытаться ее повысить, пока не закончатся ходы или оценка одного из ходов перевалит через бету.

До начала этого перебора мы делаем нулевой ход, попросту передаем ход противнику. Его альфа теперь это моя минус бета и он запускает поиск в надежде найти ход, который повысит

свою альфу. Допустим, ему удается найти такой ход и он ликует! Тогда это удовлетворяет эмпирическому правилу, которое мы озвучили выше и передать ход действительно не выгодно мне как игроку. Для меня, согласно минимаксному принципу, оценка этого хода обязана быть меньше беты.

Но если противник произвел перебор и не нашел для себя ни одного хода, способного улучшить его оценку, тогда наше эмпирическое правило нарушается. Парадокс? Для игрока это означает, что оценка осталась равна бете, что соответствует отсутствию найденного хода для противника.

Парадокса нет. Просто такой узел слишком хорош, что бы противник позволил его. Поэтому произведем отсечение этого узла, вернув на уровень выше бету. По сути мы предсказали бета отсечение прежде чем оно фактически произошло.

Так, в чем же выигрыш, если мы для того, что бы отрезать узел все равно должны его просчитать? Дело в том, что поиск для противника запускается с нулевым окном: достаточно убедится, что оценка повышается, не имеет значения насколько. Во вторых, этот поиск запускается с несколько меньшей глубиной.

Это сокращение глубины обычно обозначается в литературе как R и обычно равно двум или трем, хотя существуют и гораздо сложные схемы, где R функция от глубины. Если R слишком большое, движок может начать пропускать тактические ходы и все преимущество от использования нулевого хода будет утрачено. В моем движке $R = 3$. Это значение зарекомендовало себя довольно хорошо.

Существуют позиции, где пользоваться этой эвристикой категорично нельзя.

- При шахе нулевой ход приведет к нелегальной позиции
- Поскольку цунгцванг чаще всего происходит в эндшпиле, разрешаем нулевой ход только если у противника есть хотя бы одна легкая фигура и выше. Существуют и другие, более изощренные схемы обнаружения угрозы цугцванга
- Используем нулевой ход только если статическая оценка позиции для игрока больше бета. Точно не знаю зачем, но так вроде правильно.
- Если вы решили сделать рекурсивный нулевой ход, нельзя допустить использовать его 2 раза подряд, так как при смене ходящей стороны 2 раза мы вернемся к той же самой позиции и все преимущество его использования потеряно.

Как вы видите, эта эвристика обладает огромным количеством степеней свободы и подогнать их все под оптимальную работу не так просто.

То, что менее очевидно, это интеграция нулевого хода с другими эвристиками и алгоритмами поиска. Дело в том, что когда я в первый раз реализовал эту эвристику, она оказалась далеко не столь эффективной, как ее описывают в литературе, да и порой давала прямо противоположный результат, увеличивая количество просчитанных узлов.

Однако, методом проб и ошибок удалось заставить его хорошо работать. Для этого пришлось сделать нулевой ход не рекурсивным, запрещая использование нулевого хода внутри нулевого хода.

Каково же было мое удивление, когда выяснилось, что модифицируя алгоритм альфа-бета до его безопасной модификации, называемой PVS, нулевой ход резко снизил свою эффективность. В результате некоторых изменений, о которых пойдет речь далее, удалось добиться хорошей

интеграции между ним и PVS. В результате, нулевой ход вновь стал рекурсивным. Чудеса, да и только.

Рассматривать код нулевого хода отдельно от переборочного алгоритма, на мой взгляд, неправильно, однако мы все же сделаем это, чтобы сложить у читателя основное представление

```
1  function integer search_function(node,alpha,beta,depth)
2   if depth <= 0:
3     return the heuristic value of node for current player
4
5   if( is_null_move_allowed )
6     make_null_move
7     score = - any_search_function(node,-beta,1-beta,depth-R)
8     unmake_null_move
9
10    if(score >= beta)
11       return score
12  end if
13
14  .....
15 end function
```

15.7 Итеративный поиск

Играя в шахматы на время, движок должен быть способен в любой момент остановить поиск и сделать разумный ход. Однако, есть опасность, что запустив поиск на слишком большую глубину, пользователь попросит сделать ход еще до окончания работы алгоритма. А запустив на маленькую глубину, останется напрасно потраченное лишнее время и откровенно слабый ход.

Решением этой проблемы будет производить серию поисков, начиная с маленькой глубины и постепенно ее наращивая, обычно на 1 полуход. В любой момент, когда пользователь остановит поиск, даже если он останется незаконченным, всегда будет уже готовый ход с уже завершенного поиска на меньшую глубину.

Все это замечательно, но таким образом производятся множество повторных вычислений и, если я хочу просто посчитать на глубину 8, мне незачем считать последовательно на глубину от 1 до 7 включительно.

И тут проявляется настоящая сила интеграции между различными эвристиками. Дело в том, что запускаясь последовательно на меньших глубинах, такие эвристики как хеш таблица, история, ходы-убийцы и, как мы увидим далее, эвристика главной линии, начинают собирать информацию о позиции и ее анализе. Закончив поиск на одной глубине и перейдя на большую, начинает использоваться накопленная информация для улучшения сортировки ходов и обрезания ветвей поиска. Так, что в целом, итеративный поиск происходит не медленней, а даже быстрей, чем одиночный поиск сразу на целевую глубину.

15.8 Главная линия

Во время работы алгоритма полезно видеть, помимо лучшего хода, еще и цепочку ходов, которые компьютер считает оптимальной игрой обеих сторон. Так, эта линия предоставляет программисту большее понимание того, что происходит внутри программы, а шахматисту эта информация нужна для анализа позиции.

Можно выводить линию после завершения поиска, но более захватывающе выводить ее во время поднятия альфы на корневой глубине, то-есть нахождения более сильного хода.

Технически, сбор этой линии практически не замедляет перебор, и состоит буквально из нескольких строчек кода. Интересно то, что, применяя в программе последовательный поиск, можно использовать лучшую линию предыдущего поиска на меньшую глубину для сортировки ходов в поиске на большую глубину. Можно ожидать, что лучшая линия не претерпит сильных изменений, в крайнем случае она окажется не лучшей, но довольно сильной, поэтому рассмотрение ее в первую очередь способствует сортировке в альфа-бете и в результате будет просмотрено гораздо меньше узлов.

Производя тесты можно заметить, что в момент, когда главная линия полностью опровергается у корня, поиск замедляется и эта итерация просчитывается больше обычного. Это происходит от того, что резко падает количество отсечений, стимулируемых сортировкой. Ходы, которые считались лучшими, теперь плохи и только мешают. Поэтому очень важно увидеть угрозу как можно раньше, снизив вероятность спонтанного изменения линии. Не последнюю роль в этом выполняют продления и качественная оценочная функция, способная заранее предсказать опасность.

К слову сказать, некоторые программы вообще отказались от использования лучшей линии в сортировке ходов. Поскольку хеш таблица практически делает то-же самое и справляется не хуже. Данный вопрос требует дополнительного исследования.

15.9 Продления / Сокращения

Моя программа на данный момент использует только продления на шахах на один полуход и это является критически важным для тактики.

Я где то читал, что взятия, где материал выигрывается в любом случае, так называемые “хорошие взятия”, можно так же продлевать без особого разрастания дерева, но для их надежного выявления нужно реализовать SEE. Требует дополнительного исследования.

Две эвристики, которые очень заманчивы для исследования:

Первая, это эвристика единственного хода (one reply extension). Идея заключается в том, что если в узле существует только один возможный ход, что случается при шахах и прочих близких к мату позициях, то стоит продлить поиск еще на узел, поскольку это не приведет к сколь либо значимому разрастанию дерева. Проблема с реализацией этой идеи заключается в том, что мы используем псевдо-легальный генератор ходов, и узнать, что в узле есть только один легальный ход, без проверки их всех нельзя.

Вторая эвристика, это эвристика сингулярности (singular extension). Идея заключается в том, что если некий ход в узле намного лучше, чем все его альтернативы, то его нужно рассмотреть с увеличенной глубиной. И опять мы столкнулись с якобы парадоксом. Для увеличения глубины в поиске нужно этот самый поиск провести! Ну не сканировать же одну и ту же ветвь два раза.

Общим решением для этих эвристик может служить их интеграция со следующей эвристикой, которая называется внутренний итеративный поиск (Internal Iterative Deepening). Прежде, чем начать сканировать узел, можно запустить в нем поиск с уменьшенной глубиной, для того, что бы примерно узнать, что нас ждет при его изучении. Информация об этом поиске запишется в хеш таблицу[3], и последующий поиск, выполненный на полную глубину будет оперировать этими данными для улучшения сортировки в нем. Внутренни итеративный поиск обычно используют только в том случае, если хеш таблица для узла не содержит информации об уже найденном лучшем ходе. Можно попробовать использовать поиск на меньшую глубину для эвристики единственного хода и сингулярности. Требует дополнительного исследования.

[3]Конечно, и другие эвристики, как таблица киллеров и истории будут участвовать в обучении, но в данном случае нас наиболее интересует запись лучшего хода в хеш таблице.

Глава 16

Категории в сортировке ходов

Теперь, когда различные эвристики умеют выявлять хорошие ходы и давать им приоритет внутри своей группы, единственным нерешенным вопросом является отношения этих групп относительно друг друга.

В настоящий момент в моей программе используется следующий приоритет

1. Ход, находящийся на лучшей линии поиска на меньшую глубину(если есть)
2. Ход из хеш таблицы(если есть)
3. Взятия сортируются по LVA/MVV
4. Первый киллер
5. Второй киллер
6. Эвристика истории

LVA/MVV замечательная эвристика, но все же некоторые движки от нее отказываются в пользу более медленной эвристики SEE, с которой я еще не экспериментировал. В общем, здесь еще огромное поле для исследований.

Конечно, нельзя допускать случайное переползание ходов между категориями. Например, что бы какой-то ход из истории стал выше, чем второй киллер.

Глава 17

Алгоритм PVS

Алгоритм PVS это безопасная оптимизация альфа-беты. Она очень эффективна в острых тактических позициях, хотя в спокойных позициях ее эффективность не столь очевидна, но точно не хуже, чем старая версия программы только с альфа-бетой[1].

Идея алгоритма, грубо говоря, использовать поиск с нулевым окном везде, где есть шанс, что ход не является лучшим. Огромную роль здесь играет сортировка ходов. Благодаря ей первый ход в узле[2] с большой вероятностью лучший, а поэтому велик шанс, что он поднимет альфу. Поэтому он рассматривается с полным окном. После того, как был рассмотрен первый ход в узле, все остальные рассматриваются с нулевым окном, проверяя если они поднимают альфу. Для этих ходов шанс на это меньше, поэтому оценка этих ходов будет повышать альфу относительно редко. Если мы угадали, то целая ветвь проверена и уточнять оценку не нужно. Если же поиск с нулевым окном показывает, что ход поднимает альфу, то необходимо пересчитать этот ход для уточнения оценки.

Заметим, что в алгоритме PVS поиск с нулевым окном используется 2 раза: первый в эвристике нулевого хода, второй собственно для проверки если ход поднимает альфу. Таким образом все узлы, которые просмотрит алгоритм можно условно разделить на две категории: те, которые рассматриваются с полным окном и те, которые рассматриваются с нулевым окном[3].

Первые назовем PV-node. Это особые узлы: в них не используют эвристику нулевого хода, в них разрешены PVS проверки и они с большей долей вероятности составляют сильнейшие линии в шахматном дереве.

Однако вот алгоритм PVS начинает запускать проверки с нулевым окном и порождается множество не PV узлов. Для таких узлов правила другие: в них разрешен нулевой ход, но PVS проверки запрещены. Интересно, что с таким подходом нулевой ход становится рекурсивным. Необходимо только следить за тем, что он не запускается в двух узлах подряд.

[1] Такое сравнение не совсем корректно. Ведь для этого алгоритма пришлось изменить стратегию работы с эвристикой нулевого хода, прежде чем алгоритм эффективно заработал. Это еще раз показывает очень сложную интеграцию между всеми эвристиками в программе.

[2] Напомним читателю, что под первым ходом я понимаю первый легальный ход, поскольку в программе с псевдо-легальным генератором ходов первый ход не обязательно легален.

[3] Напомним, что алгоритм альфабета просматривает каждый узел с полным окном, поскольку сужение окна поиска не может привести к искажению оценки и математически безопасен. Здесь же мы имеем в виду поиск с нулевым окном, который в общем случае не дает точную оценку благодаря искуственному обнулению размеров окна, а поэтому является поиском с не полным окном.

Для позиций, где преобладает тактика, будет узкое дерево PV узлов, решение будет большей частью форсировано и большая часть дерева насмотрится с нулевым окном. Это будет максимально выгодно в плане отсечений.

Для позиций тихих, с позиционной игрой, дерево из PV узлов широкое и меньшая часть дерева рассматривается с нулевым окном. Выигрыш будет меньше.

Приведем, собственно псевдокод

```
1  function integer PVS(node,alpha,beta, depth,is_PV_node)
2   if depth <= 0:
3     return the heuristic value of node for current player
4
5   bSearchPv = true
6   for child in node:
7       if(bSearchPv || not(is_PV_node))
8         score = PVS(child,-beta,-alpha, depth-1)
9     else
10        score = PVS(child,-alpha-1,-alpha, depth-1)
11        if(score > alpha)
12           score = PVS(child,-beta,-alpha, depth-1)
13
14    if(score>alpha)
15       alpha = score
16    if(alpha>=beta)
17       return beta
18    bSearchPv = false
19  next for
20  return alpha
21 end function
```

При правильной реализации, алгоритм PVS является качественным улучшением алгоритма альфабета. Максимальная отдача достигается в острых, тактических позициях, где выигрыш в большей мере форсирован. И тем не менее, я бы не рекомендовал в новой шахматной программе начинать имено с него, перепрыгнув альфабету.