Санкт-Петербургский государственный университет

241 группа

Добряков Дмитрий Алексеевич

Шахматный ИИ

Курсовая работа

Научный руководитель:

доцент Литвинов Ю.В.

Санкт-Петербург

2018

Оглавление

1. Введение и постановка задачи

Шахматы – это очень старинная игра, история которой насчитывает не менее полутора тысяч лет. На протяжении многих веков люди совершенствовали свои навыки игры, с каждым поколением исследуя ее все больше и больше. Шахматы долгое время являлись показателем превосходства человеческого интеллекта над машинами, но все изменилось, когда в 1996 году впервые шахматная программа смогла одолеть действующего чемпиона мира – Гарри Каспарова.

С тех пор технологии развивались, и сейчас не известно ни об одном человеке, который смог бы сравниться с компьютером в этой области.

Целью данной работы является создать подобную шахматную программу, а именно:

- Создать ИИ для компьютерного оппонента, способный показывать достойную игру
- Реализовать удобный графический интерфейс с поддержкой передвижения фигур мышкой
- Провести тестирование результата

2. Обзор существующих решений

К наиболее известным существующим решениям можно отнести такие движки, как *Stockfish* и *Komodo*, которые на протяжении многих лет занимают лидирующие позиции в мировом рейтинге.

Также, совсем недавно компания *Google* выпустила свой продукт под названием *Alpha Zero*, основанный на работе нейросети и, неожиданно для всего шахматного сообщества, с разгромным счетом победивший *Stockfish* в битве из 100 партий (28 побед и 72 ничьи). К сожалению, получить к нему доступ невозможно.

Не бывает идеальных решений: у любого существуют как достоинства, так и недостатки. К примеру, как *Komodo*, так и *Stockfish*, не предлагают никакого пользовательского интерфейса, и чтобы ими воспользоваться – придется скачивать сторонний софт, что может доставить дискомфорт непродвинутому пользователю.

Мы не претендуем на звание лучшей шахматной программы, однако, несмотря на многообразие существующих решений, людьми продолжает вестись постоянная работа над новыми идеями и совершенствованием уже известных алгоритмов в этой области, в следствие чего сила компьютерной игры стабильно растет – и поэтому сама по себе задача написания подобной программы представляет интерес.

Возможно, в нашей реализации не будет каких-то принципиально новых идей, однако мы постарались грамотно воспроизвести уже созданные наработки и объединить их в дружелюбном для пользователя интерфейсе, добавив некоторые дополнительные функции, которые есть не во всех программах: к примеру, возможность поставить компьютер играть с самим собой, а пользователю выступить в роли наблюдателя.

3. Описание решения

Говоря более детально о реализации, написание шахматного движка состоит из нескольких важных этапов. В первую очередь, важно определиться с правильным представлением информации, ведь основная проблема заключается в том, что оптимизация кода здесь встает на первое место. Если не оптимизировать код, то, перебирая миллионы вариантов, программа просто зависнет, и чтобы этого не допустить, иногда приходится жертвовать читаемостью, удобством поддержки и математической общностью кода. Основной механизм работы заключается в том, что существует специальный генератор, который для каждой позиции возвращает список всех возможных ходов и передает их алгоритму поиска, который будет эти ходы умным образом перебирать и выбирать лучший при помощи оценочной функции.

Таким образом, программа состоит из следующих основных модулей, которые необходимо осуществить:

- Представление позиции на доске
- Представление фигур и цветов
- Генератор возможных ходов
- Перебор дерева вариантов
- Оценка позиции
- Графический интерфейс

3.1 Представление доски

Первое, что приходит на ум при попытке описать позицию на доске – это создать массив 8x8, который в своих элементах хранил бы информацию о фигуре, находящейся в соответствующей клетке.

Такой вариант, разумеется, однозначно определяет позицию, но, чтобы, к примеру, найти на доске белого короля, потребуется в худшем случае перебрать все 64 элемента массива, а это непозволительно много, учитывая, что операции, подобные этой, будут производиться миллионы раз подряд.

Другой вариант – это для каждого типа фигуры хранить отдельный массив, который бы содержал все клетки на доске, в которых эти фигуры находятся: все белые пешки, все черные ладьи и так далее. В таком случае найти белого короля проблем не составит, однако, наоборот, проблема возникнет при попытке определить, какая фигура стоит в определенном месте.

Поскольку в ходе работы программы часто возникает как задача найти на доске фигуры данного типа, так и задача определить, что за фигура стоит в определенной клетке, то решением будет объединить оба подхода – несмотря на то, что тогда информация будет храниться в памяти в дублированном виде и это создаст дискомфорт и возможности для ошибок при ее изменении.

Однако, надо сказать, что иметь для каждого типа фигур отдельный массив было бы неразумно, ведь для каждой из 64 клеток доски мы лишь сохраняем информацию о наличии или отсутствии в ней фигуры, которая занимает всего 1 бит. По счастливой случайности, клеток на доске ровно столько, сколько бит в типе long. Тогда, вместо массива мы можем использовать обычное число: если фигура стоит в данной клетке – мы помещаем в

соответствующий бит 1, в противном случае – 0. Такое числовое представление подмножества клеток доски называется **Bitboard**.

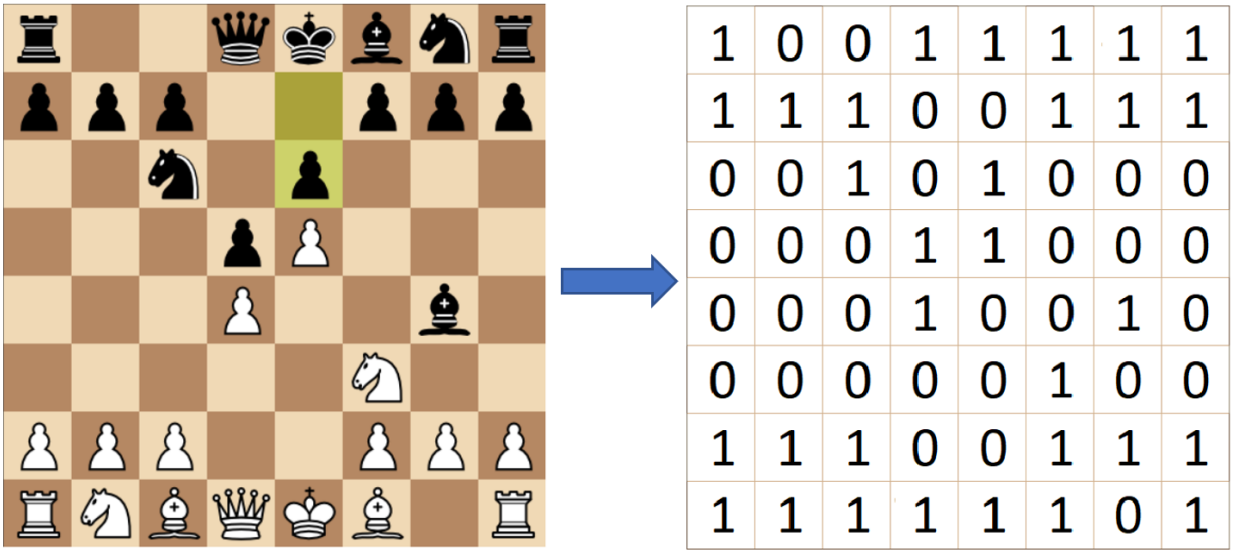

Рисунок 1 Bitboard

Например, положение ладей (черных и белых) в начальной позиции в виде bitboard'а записывается как
1000000100000000000000000000000000000000000000000000000010000001[*]

Помимо Bitboard'ов на каждый тип фигур, для удобства мы можем также поддерживать Bitboard для всех фигур на доске и два Bitboard'а на все фигуры каждого цвета, чтобы не высчитывать их при каждом обращении заново. Таким образом, всего на сохранение информации обо всех фигурах потребуется 15 (6*2 + 1 + 2) long значений и один массив int[64].

Такое избыточное представление обладает массой преимуществ. Выполнение сложных действий сводится к одной-нескольким битовым операциям. Например, как получить все клетки, которые атакуют белые пешки:

```
(WhitePawnBitboard << 7 & ~HFileBitboard) | (WhitePawnBitboard << 9 & ~AFileBitboard)
```

3.2 Представление фигур, цветов

Фигуры нами было также решено представлять при помощи числовых значений, используя на это всего 4 бита.

0000 – ничего, 0010 – пешка, 0100 – конь, 0110 – слон, 1000 – ладья, 1010 – ферзь и 1100 – король. Младший бит везде оставлен нулем для того, чтобы в нем можно было хранить

[*] *Существуют различные соглашения по поводу соответствия номеров битов и клеток доски, мне наиболее удобным видится то, в котором A1 представляет младший бит, далее идут B1 и остальные биты первого ряда, затем A2..H2 и так далее до A8..H8, где H8 – старший бит числа. Я реализовал именно его, хотя особой разницы нет.*

цвет фигуры: белый и черный цвета кодируются нулем и единицей соответственно. Тогда, например, черный конь будет записан как 0101, пустая клетка как 0000 или 0001, а 0110 будет либо белым слоном, либо просто слоном, в случае, когда мы хотим получить только бесцветный тип фигуры.

Используя такое представление, получение типа или цвета занимает всего одну битовую операцию «И» с соответствующей маской.

```
Тип фигуры = Piece & 0b1110;
Цвет фигуры = Piece & 1;
```

3.3 Генератор ходов

В любой позиции для рассмотрения возможных вариантов развития событий необходимо уметь получать список всех возможных ходов. Генерация ходов – это один из наиболее важных этапов и одно из тех мест, в которых процессор будет проводить основную часть времени, поскольку вычисляется это не так очевидно, а требуется очень часто.

Вообще, генераторы ходов делятся на легальные и псевдо-легальные. Ходы, получаемые в псевдо-легальном генераторе, не учитывают, что после них король может оказаться под шахом, а легальные генераторы это исключают и выдают сразу верные ходы. В случае псевдо-легального генератора часть обязательств по определению корректности хода возлагается на переборочный алгоритм. Например, если алгоритм поиска заметил, что короля съели, значит предыдущий ход был нелегальным. Преимущества есть у обоих типов, но мы остановились на подходе с легальной генерацией, поскольку он видится естественнее.

Представление ходов аналогично представлению фигур: в 32 бита помещается информация о клетках, с которой и на которую осуществляется ход, о фигурах на этих клетках, а также дополнительная информация – специальный тип хода (рокировка, взятие на проходе, проведение пешки), и если это проведение пешки, то фигура, в которую она проводится.

Теперь о самом подходе к генерации: для начала забудем про проверку на шах.

Проще всего дела обстоят с конем и королем, поскольку другие фигуры не могут загораживать им путь. Поэтому вместо того, чтобы вычислять их ходы с нуля по мере необходимости, можно еще до начала работы основного алгоритма вычислить все возможные клетки, на которые они нападают и сохранить их в специальные таблицы в виде bitboard’ов. Например, ходы коня с клетки f4:

```
Attack.Knight[f4] =
                    00000000 <- старший бит
                    00000000
                    00001010
                    00010001
                    00000000
                    00010001
                    00001010
     младший бит -> 00000000
```

На каких-то из этих клеток могут стоять фигуры своего цвета, поэтому чтобы теперь получить список всех доступных клеток для псевдо-легальных ходов, нужно их исключить:

```
Ходы коня с f4 = Attack.Knight[f4] & ~PieceBitboard[MovingColor]
              (где PieceBitboard[MovingColor] содержит все фигуры ходящего цвета)
```

Для ладей, слонов и ферзей используется более сложный подход, в котором для каждой клетки доски заранее генерируются bitboard'ы в каждом из 8 направлений. Эти bitboard'ы пересекаются (&) с фигурами на доске и в зависимости от направления вызывается функция lsb (least-significant bit) или msb (most-significant bit) для поиска ближайшей фигуры, часть после которой отрезается, оставляя только нужные клетки. Объединяя (|) эти клетки по разным направлениям получаем список псевдо-легальных ходов.

Для пешек этот список проще всего получить вручную, так как он не симметричен и есть всего 4 возможных клетки, а также нужно учитывать проведение в ферзи.

Если возможны рокировка или взятие на проходе, то они генерируются отдельно.

Чтобы теперь получить из списка псевдо-легальных ходов конечный список легальных, нужно лишь просмотреть каждый ход и оставить те, в результате которых король не находится под шахом. Однако проверять это для каждого хода очень затратно, ведь связанных фигур (прикрывающих короля от нападения) гораздо меньше, чем несвязанных. Поэтому мы сначала вычисляем список всех связанных фигур, и если фигура в него входит и при этом движется не по линии с королем, то делаем проверку на шах, но в противном и гораздо более частом случае этой проверки удается избежать.

3.4 Поиск лучшего хода

Для поиска лучшего хода была использована известная модификация алгоритма «Минимакс» – альфа-бета отсечения.

Минимакс – это алгоритм обхода дерева ходов с целью каждому из них сопоставить некоторую оценку, получаемую при правильной игре обоих игроков и определяемую специальной оценочной функцией. Очевидно, что его сложность возрастает экспоненциально с увеличением глубины и уже на глубине 6 придется просматривать примерно 25^6 = 244 140 625 узлов дерева.

К счастью, существует способ сократить количество узлов, при этом не пропустив варианты. Этот способ называется альфа-бета отсечениями и основывается на том, что ветвь дерева можно не рассматривать, если было найдено, что для нее оценка заведомо хуже, чем вычисленная для предыдущей ветви.

Альфа-бета отсечения оказываются особенно полезны при правильном порядке ходов – действительно, чем раньше встретится хороший ход, тем больше плохих после него рассматриваться не будут. Поэтому для большей эффективности имеет смысл предварительно сортировать ходы из каких-то общих соображений. Одним из таких соображений, например, может быть взятие более значимой фигуры менее значимой фигурой. Чем больше разница, тем скорее всего лучше ход. В своей программе мы также воспользовались идеей, что на взятие с большой вероятностью хорошим ответом будет съедение взявшей фигуры.

Для примера, насколько это действенно, здесь представлена статистика по оценке одних и тех же позиций (на глубине 6) с использованием сортировки ходов (слева), и без использования сортировки (справа):

Time: 00:00:03.8827076	Time: 00:00:24.5993649
Total nodes considered: 868438 (223668 n/s)	Total nodes considered: 8030118 (326436 n/s)
Total positions evaluated: 742379 (191201 n/s)	Total positions evaluated: 7536112 (306354 n/s)
LegalMoves calculated 126060 times	LegalMoves calculated 494007 times
Time: 00:00:07.8448631	Time: 00:00:41.1018542
Total nodes considered: 1702850 (217066 n/s)	Total nodes considered: 12566134 (305732 n/s)
Total positions evaluated: 1418266 (180789 n/s)	Total positions evaluated: 11609348 (282453 n/s)
LegalMoves calculated 284585 times	LegalMoves calculated 956787 times
Time: 00:00:01.8449115	Time: 00:00:19.7134289
Total nodes considered: 342171 (185467 n/s)	Total nodes considered: 6020544 (305403 n/s)
Total positions evaluated: 259584 (140703 n/s)	Total positions evaluated: 5365167 (272158 n/s)
LegalMoves calculated 82588 times	LegalMoves calculated 655378 times

Как видно, производительность возрастает в 10-20 раз при использовании сортировки, несмотря на дополнительные затраты по ее проведению в каждой из многочисленных перебираемых позиций.

3.5 Оценочная функция

Для правильной работы алгоритма поиска нужна хорошая функция оценки. Однако, при этом необходимо поддерживать некоторый баланс, ведь если слишком сильно ее перегрузить, то глубина поиска может заметно сузиться.

Оценка позиции производится на значительной глубине и поэтому может быть довольно примерной: в своей программе мы опирались на материальное положение обеих сторон, а также на позиции фигур на доске. Для каждой фигуры была предварительно создана специальная таблица со значениями, показывающими качество расположения фигуры в определенной клетке. Например, коню не поощряется вставать на край доски, а королю – выходить на центр поля в начале игры, пешки же наоборот получают бóльшую оценку при продвижении к последнему ряду.

3.6 Графический интерфейс

Рисунок 2 Пользовательский интерфейс

В ходе разработки был реализован пользовательский интерфейс, позволяющий перетаскивать фигуры мышкой. Как видно, для удобства, во время перемещения фигуры доступные клетки обозначаются специальным кружком, а последний совершенный на доске ход подсвечивается. Вести игру можно как с компьютером, так и с другим игроком. Можно также поставить двух компьютеров играть друг с другом.

4. Апробация

Приложение было протестировано тремя людьми - любителями. Двое из них сыграли по два раза и еще один – около 10 раз. Несмотря на то, что они утверждают, что были близки к победе, все партии были выиграны движком.

Также в качестве оппонента был поставлен движок Stockfish на различных уровнях. На уровнях 4 и 5 из возможных 8 (1600 и 1700 ELO) наша программа уверенно одержала победу (выиграв 3 партии из 3), но уже на шестом уровне (1900 ELO) ей этого сделать не удалось, и она проиграла обе сыгранные партии. Из этого можно заключить, что уровень игры программы приблизительно равен 1800 ELO, что соответствует границе между вторым и первым разрядами.

5. Заключение

Таким образом, в результате работы:

- Был создан искусственный интеллект для игры в шахматы, показывающий игру на уровне первого-второго разряда
- Был реализован графический интерфейс, позволяющий удобно передвигать фигуры и вести партию
- Результат был протестирован на нескольких людях и компьютере с разными уровнями сложности

Реализация проекта находится по адресу https://github.com/ddobrjakov/Chess-Engine

6. Список источников

[1] URL: https://chessprogramming.wikispaces.com/ (дата обращения 05.06.2018)

[2] URL: https://bitbucket.org/jackalsh/bismark/downloads/book_russian_2014.pdf (дата обращения 05.06.2018)

[3] URL: https://github.com/official-stockfish/Stockfish (дата обращения 05.06.2018)