Муниципальное Автономное Общеобразовательное Учреждение

«Гимназия №14» г. Улан-Удэ

ИНДИВИДУАЛЬНЫЙ ПРОЕКТ

Разработка движка монгольских шахмат «Шатар»

Автор-разработчик: обучающийся 11 класса «М»

Лубсанов Д. А.

Руководитель: учитель информатики

Кононова О. В.

Работа допущена к защите «____» _____ 2023 г.

Подпись руководителя проекта _____ (Кононова О. В.)

г. Улан-Удэ

2023 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 3
ГЛАВА 1. ОБЗОР И ТЕОРИЯ 5
1.1. Используемые технологии 5
1.2. Обзор игры 5
1.3. Шахматный движок 6
1.3.1. Шахматная доска 6
1.3.2. Оценка 8
1.3.3. Алгоритм 9
ГЛАВА 2. РЕАЛИЗАЦИЯ 13
2.1. Классы 13
ЗАКЛЮЧЕНИЕ 17
СПИСОК ИСТОЧНИКОВ 18
ПРИЛОЖЕНИЯ 19

ВВЕДЕНИЕ

Шахматы издавна были популярны во всём мире. Очень часто разные народы вносили свои собственные изменения в правила (собственно, современные шахматы — сильно изменившаяся древнеиндийская игра чатуранга).

Одним из таких вариантов является **«Шатар»** — монгольские шахматы. Само название — видоизменённое персидское «шатранг», а то, в свою очередь, происходит из вышеупомянутого «чатуранга».

Актуальность обусловлена вновь возросшим интересом к этому национальному виду шахмат. Открываются кружки, проводятся соревнования, турниры — например, в рамках фестивалей «Алтаргана» и «Сурхарбан». Они были впервые включены в их программу относительно недавно — в 2008 году.

Мною, к сожалению, не были найдены другие программные реализации этой игры.

2022 год объявлен Годом культурного наследия народов России; также многие направления определены в «Концепции сохранения и развития нематериального культурного наследия народов Российской Федерации...». Проводятся мероприятия по сохранению культур народов. Развивается и культура Бурятии; потому также важны усилия по популяризации народных игр, в том числе и «шатар».

Объект: монгольский вариант шахмат шатар.

Предмет: игровой движок шахмат шатар.

Цель проекта: написать на языке программирования C++ программу шахмат шатар.

Задачи:

1. Изучить информацию о шатар.

2. Изучить теорию шахматных движков

3. Написать шахматный движок на языке C++

ГЛАВА 1. ОБЗОР И ТЕОРИЯ

1.1. Используемые технологии

Создание движка будет вестись на языке C++. Это мощный и популярный язык для разработки настольных приложений.

Разработка будет вестись в среде CLion от компании JetBrains. Она также является популярной, имеет широкий набор возможностей. Кроме того, производитель предоставляет бесплатные лицензии учащимся.

Компилятором является GNU G++ 12.2, а системой сборки — CMake 3.26. Они оба имеют открытый исходный код и бесплатны. Проект создан при помощи дополнения cmake-init.

Для управления терминальным выводом используется библиотека rang[6].

1.2. Обзор игры

Как уже было сказано, шатар — это разновидность шахмат, и большинство правил схожи с европейскими шахматами. Так что имеет смысл описать различия. На самом деле, степень отличия может варьироваться, ведь среди шатар тоже встречаются разные варианты.

Возьмём такую версию[2][8]:

- Отсутствует рокировка;
- Пешка не может совершать двойное перемещение, кроме случаев начала игры. Таким образом, отсутствует взятие на проходе;
- В начале игры обязательно совершаются ходы d2d4 и d7d5 (можно считать это начальным состоянием игры);
- Ферзь может ходить только на одно поле по диагонали.

1.3. Шахматный движок

Шахматный движок — программа, переносящая игру в шахматы в форму компьютерной программы. Движки также обычно содержат в себе некий алгоритм, способный играть в шахматы.

Тогда шахматный движок как бы состоит из двух частей: шахматной доски и алгоритма игры (условно — искусственного интеллекта, ИИ), который, в свою очередь, состоит из механизмов *оценки* и *поиска*.

1.3.1. Шахматная доска

Существует множество способов представления шахматной доски[6] — например, можно было бы использовать «наивный»: создадим массив размером 8x8, в каждой ячейке которого находится некая фигура. Ещё лучше будет создать доску размером 10x12, тогда удобнее будет обработать фигуры, которые могут выйти за пределы поля.

Однако, уже давно существует более удобный (в первую очередь для компьютера) способ, использующий т.н. «битовые доски» (англ. *bitboards*)[5]. Его достоинства в том, что требуется меньше памяти, а также увеличивается скорость (за счёт использования встроенных операций). Так, можно сгенерировать ходы для пешек просто произведя побитовый сдвиг. Многие операции можно очень быстро выполнять — за $\mathcal{O}(1)$.

Назовём *битовой доской* двоичное число длины 64, i-й бит которого обозначает, находится ли в позиции i какая-либо фигура.

Для удобства отметим, что нумерация битов начинается со старшего (most significant) бита. Тогда получится такое соответствие квадратам доски:

Теперь шахматную доску можно представить как набор таких битовых досок — по одной на каждый тип фигуры, ещё две, показывающие, какой сто-

Таблица 1.1: Соответствие

	A	B	C	D	E	F	G	H
8	0	8	16	24	32	40	48	56
7	1	9	17	25	33	41	49	57
6	2	10	18	26	34	42	50	58
5	3	11	19	27	35	43	51	59
4	4	12	20	28	36	44	52	60
3	5	13	21	29	37	45	53	61
2	6	14	22	30	38	46	54	62
1	7	15	23	31	39	47	55	63

роне принадлежит фигура, а также — общая, т.е. показывающая наличие или отсутствие фигуры на доске в принципе. К ним понадобятся ещё несколько вспомогательных (инвертированных).

Реализовать битовые доски достаточно просто на основе `std::bitset`.

Доска и счётчик ходов составляют *позицию*.

Нам потребуется следить за повторяющимися позициями. Можно, конечно, хранить каждую позицию по отдельности, но это достаточно затратно. Поэтому можно каждую позицию *хешировать*, т.е. присвоить *хеш*. Такой хеш называют Zobrist-хешем[6].

Для того, чтобы совершать ходы, нужна некая структура, описывающая ход. Она содержит в себе собственно ход, то, какой фигурой сходили, съеденную фигуру (если таковая была), продвижение (если оно было).

Требуется реализовать правила. Сделаем так: для каждой позиции сгенерируем все возможные легальные ходы (т.е. те, которые не нарушают правил игры). Теперь можно легко проверить ход на правильность, найдя его в списке возможных ходов.

Позиция может быть инициализирована при помощи FEN-строки (строки в формате нотации Форсайта-Эдвардса — нотации, описывающей шахматную позицию).

1.3.2. Оценка

Оценка — это некоторое число, описывающее текущее состояние игры — насколько близко тот или иной игрок к победе (или проигрышу). Очевидно, это требуется для того, чтобы алгоритм поиска понимал, какие ходы выгоднее, а какие — нет.

Зачастую в данный момент времени невозможно сказать, кто выиграет. Но мы можем приблизительно понять, кто сейчас «ведёт». Скажем, что если число *меньше*, то игра идёт в пользу белых, а если оно *больше* — чёрных.

Из чего складывается эта оценка? Тут есть множество факторов, основные из них — это

- Материал — какие фигуры сейчас на доске,
- Мобильность — сколько они бьют полей,
- Расположение фигур.

Материал — это сумма цен всех фигур на доске. Он представлен в таблице 1.2 [6].

Таблица 1.2: Цена фигуры

Фигура	Цена
Пешка	100
Конь	300
Слон	350
Ладья	500
Ферзь	900

Считается, что Король имеет очень большое значение; здесь же это просто не учитывается.

Мобильность рассчитывается из количества полей, которые бьют фигуры. За каждое поле даётся соответствующее кол-во очков (табл. 1.3).

В положениях фигур учитываются только сдвоенные пешки (отнимает

Таблица 1.3: Цена фигуры

Фигура	Цена
Конь	9
Слон	4
Ладья	3
Ферзь	3

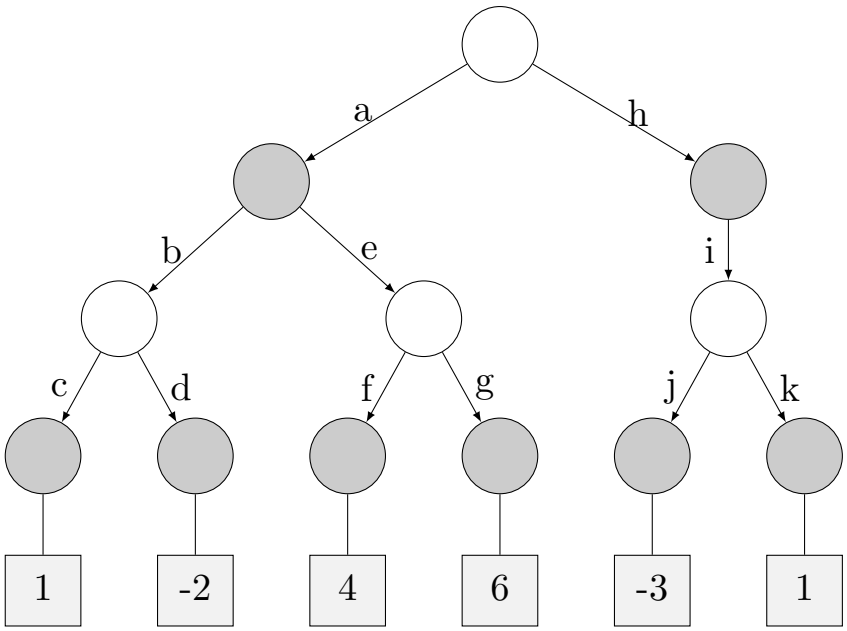
25) и два слона (даёт 50).

Имеется особая оценка для эндшпиля: единица дистанции между королями и королём и центром даёт 10 очков.

1.3.3. Алгоритм

Шахматы — классическая игра для двух игроков. Мы можем оценить каждое состояние игры; оно меняется с каждым ходом. Теперь можно сказать, что для того, чтобы победить, белые хотят максимизировать оценку (а черные — минимизировать).

Рис. 1.1: Игровое дерево



Представим игру в виде дерева, где вершины — это позиции игры, а ребра — ходы; в «листах» производится оценка (там идёт дальнейший поиск

в нашем случае). Это есть *игровое дерево* (пример на рис. 1.1) — графическое представление процесса принятия решений в последовательности игры двух соперников. На графе можно воспользоваться *поиском в глубину*: из каждой вершины-состояния мы проверим влияние на оценку всех ходов, пытаясь максимизировать или минимизировать оценку. Такой алгоритм называется *максиминным*.

Однако следует учесть, что коэффициент ветвления шахмат составляет около 35[6]. Таким образом, на каждой позиции мы имеем около 35 возможных ходов (хотя в начале и конце игры этот показатель может быть меньше, а середине — больше). Для поиска на глубину d ходов необходимо рассмотреть примерно 35^d позиций (т.е., например, на 8 ходов — $35^8 \approx 2,25 \cdot 10^{12}$).

Известно, что временная сложность алгоритма поиска в глубину составляет $\mathcal{O}(n+m)$, в нашем случае — $\mathcal{O}(b^d)$ (b — текущий коэффициент ветвления). Требуется оптимизация, иначе поиск займет слишком много времени.

Можно совершить очень большое число ходов до того, как игра завершится. Так что ограничим глубину поиска.

Другой важной оптимизацией будет сохранение значений позиций (чтобы не считать их каждый раз заново). Для этого можно воспользоваться хеш-таблицей (например, `std::unordered_map`). Будем сохранять туда хеши и значения всех рассмотренных позиций, и при переходе проверять, не находится ли она в хеш-таблице.

Но этого всё ещё недостаточно. Самой важной оптимизацией является применение *альфа-бета отсечения*[9]. Оно просто в реализации и использует следующую идею: не имеет смысла рассматривать поддерево дальше, если мы не сможем добиться лучшей для нас оценки. Мы можем ограничить поиск определённым диапазоном $[\alpha;\beta]$, определяемым динамически. Это позволяет

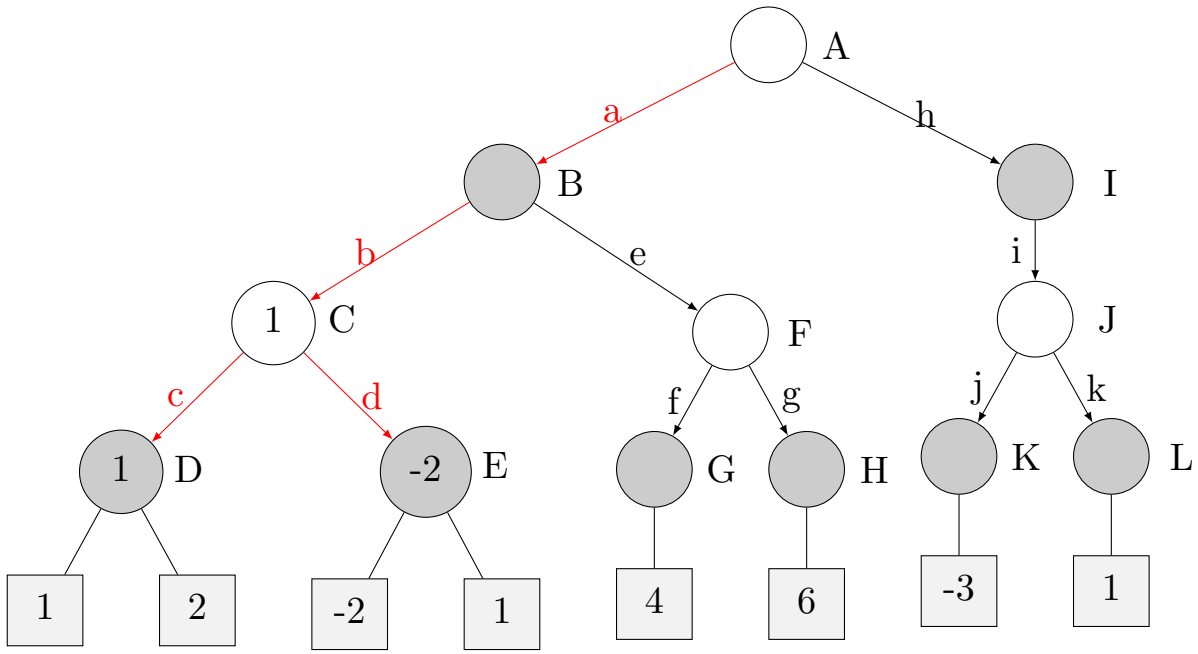
ускорить алгоритм до (в лучшем случае) $\mathcal{O}(\sqrt{b^d})$.

Рассмотрим работу алгоритма:

Изначально, $\alpha = -\infty$, $\beta = +\infty$. α рассчитывается из максимума, β — из минимума.

На рис. 1.2 алгоритм прошел пути abc и abd и оценил значения в них. Это ход черных, выбирается минимум. Теперь родительская вершина C может выбрать значение из этих двух вершин. Так как это ход белых, то выбирается 1. Теперь в C $\alpha = 1$, а в B — $\beta = 1$.

Рис. 1.2



На рис. 1.3 далее обработка следующего хода черных в B. Пройдем по aef. В G значение теперь 4, вернёмся в F. Там мы имеем $\beta = 1$, и теперь $\alpha = 4$. Так как $\alpha > \beta$, то далее смысла идти нет: у черных B уже есть ход лучше. В B теперь 1.

На рис. 1.4 возвращаемся в A. В данный момент тут $\alpha = 1$, и теперь остаётся обработать hij и hik. Тут никаких отсечений не будет: не будет ситуаций, когда для белых $\alpha > \beta$; у черных вариантов вообще нет. В A в итоге останется 1.

Рис. 1.3

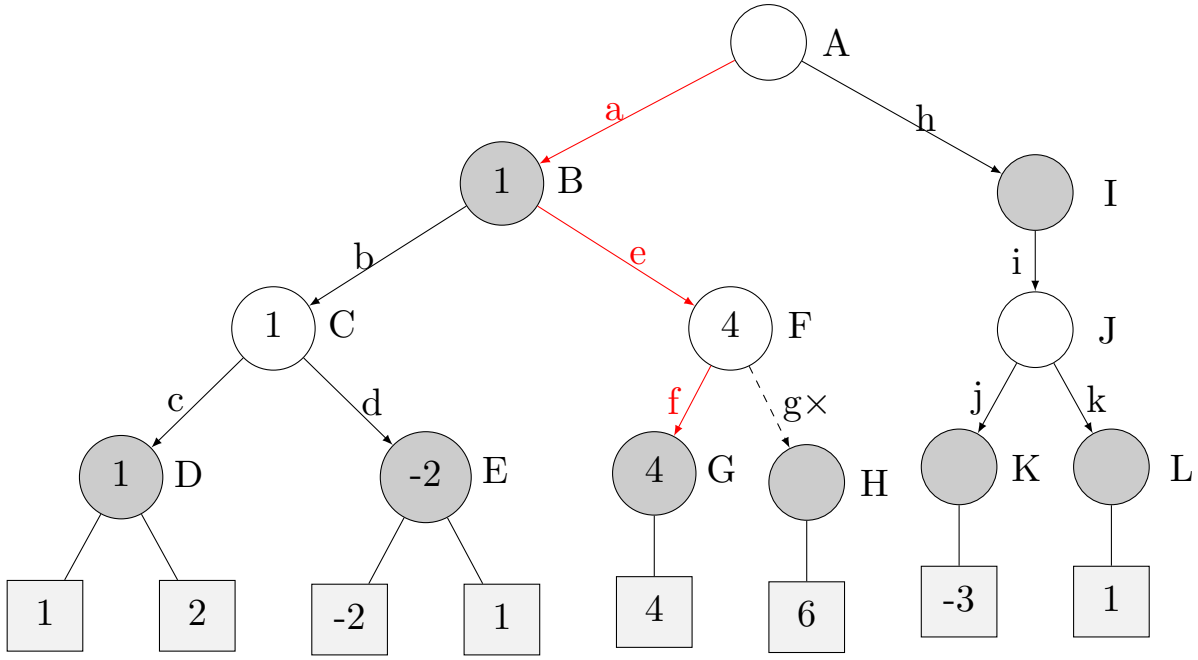
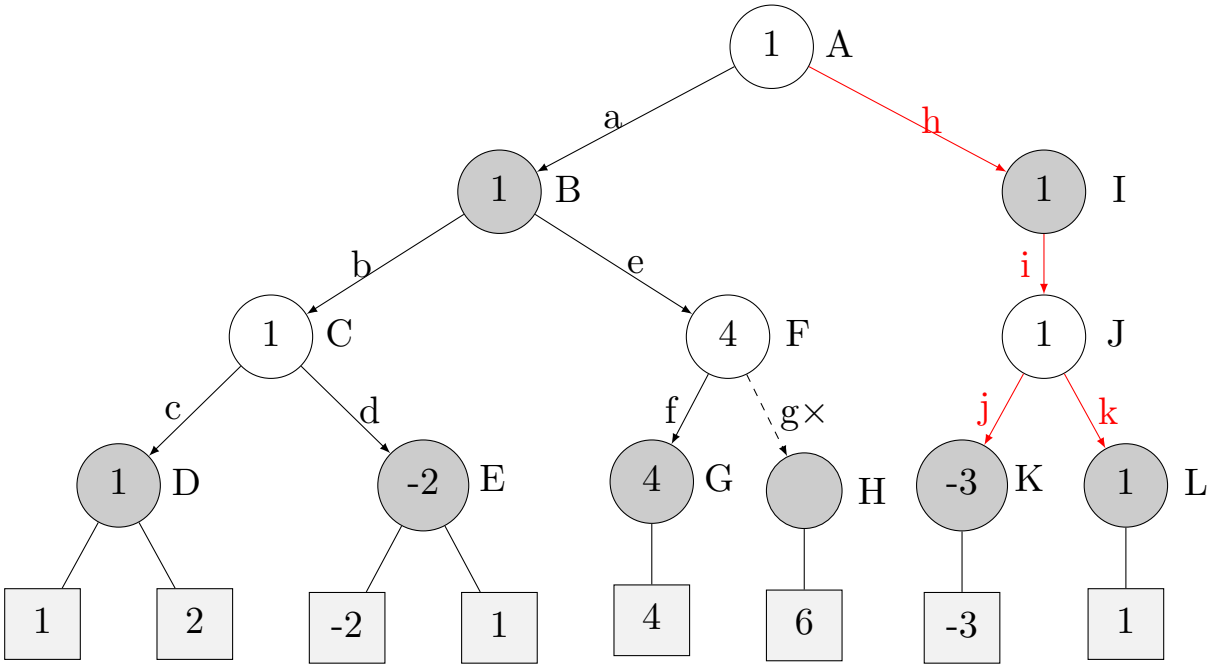

A
a
h
1
B
I
b
e
i
1
C
4
F
J
c
d
f
g×
j
k
1
D
-2
E
4
G
H
K
L
1
2
-2
1
4
6
-3
1

Рис. 1.4


1
A
a
h
1
B
1
I
b
e
i
1
C
4
F
1
J
c
d
f
g×
j
k
1
D
-2
E
4
G
H
-3
K
1
L
1
2
-2
1
4
6
-3
1

ГЛАВА 2. РЕАЛИЗАЦИЯ

Реализация игры разделена на несколько модулей, как и описано выше. Ссылка на листинги см. в приложениях.

2.1. Классы

Класс Bitboard

Как уже было сказано, этот класс построен на основе `std::bitset`. Он имеет следующие основные методы:

`test` проверяет, является ли бит единицей

`set` делает бит единицей

`reset` делает бит нулём

`count` считает количество единиц

`find_first` ищет первую единицу

`find_last` ищет последнюю единицу

Они в основном полагаются на уже встроенные методы `std::bitset`, кроме `find_first` и `find_last`: они используют скан-таблицу[5].

Имеются также некоторые вспомогательные методы (битовых операций и ввода-вывода).

Класс Board

Также реализует сказанное выше: 12 битовых досок на каждый тип фигур каждого цвета, 2 на сторону, 2 на пустые клетки сторон, доски для занятых и свободных клеток.

Метод `update` обновляет вспомогательные доски и вызывается после каждой модификации.

Класс Hash

Реализация хеширования (содержит поле с хешем доски). Использует заранее сгенерированные генератором константы для обозначения фигур на позициях, добавляя или удаляя при вызове функции `put_piece`. При этом используется тождество $a \oplus a = 0$.

Класс Move

Экземпляры этого класса хранят в себе ход. Он содержит несколько полей (откуда, куда, тип и цвет фигуры, съеденную фигуру, флаги продвижения).

Класс RecentHistory

Экземпляры этого класса содержит в себе историю последних ходов. Это нужно для реализации правила 3-х ходов. Строится на основе `std::unordered_map` Позволяет узнавать количество повторов по хешу позиции. Очищается при взятии или продвижении.

Класс Position

Экземпляры класса хранят текущую позицию: доску, счетчики ходов, хеш, историю ходов.

Основной метод — `apply_move`: применяет ход, описываемый в классе Move.

Класс PseudoLegalMoveMaskGen

Этот класс генерирует битовые маски движения для всех фигур, с учетом правил ходов (кроме шахов и других подобных). Отдельно генерируются взятия пешками. К сожалению, в этом классе есть несколько не очень эффективных методов, однако, это не сильно влияет на производительность.

Здесь же находится функция `in_danger`, проверяющая, не бьётся ли какая-либо фигура другими. Для этого генерируются все маски из данной позиции и проверяется, не находятся ли там фигуры, которые могут съесть данную.

Класс MoveList

Экземпляры этого класса хранят в себе список возможных ходов от генератора. В остальном схож с массивом (имеет методы `[]`, `push_back`, `size`).

Класс LegalMoveGen

Этот класс генерирует ходы на основе ранее созданных масок. Делается это так: из маски при помощи `find_last` достается следующий ход. Он проверяется на правильность (король не окажется под ударом) и записывается в список ходов `MoveList`. Имеется возможность генерировать только взятия.

Класс Static

Этот класс реализует оценку позиции как было сказано выше. Для получения оценки вызывается функция `evaluate`.

Класс Human

Это интерфейс взаимодействия игрока и игры (т.е. получения от него хода). Считывается ход игрока (в формате **e2e3**, откуда (**e2**) — куда(**e3**)). Если ход содержит продвижение, то спрашивается, какое именно.

Класс AI

В этом классе содержится вышеупомянутый алгоритм поиска. Однако здесь имеются некоторые интересные особенности.

Алгоритм альфа-бета запускается в отдельном потоке — так можно настроить время (а значит и глубину) поиска; также пользователю не будет казаться, что программа «зависла». По истечении времени взводится флаг остановки, и поиск прекращается. Возвращаемое значение — оценка и ход.

Циклом жестко задаётся глубина основного поиска (она постепенно наращивается). Когда алгоритм достигает этой глубины, запускается алгоритм оценки.

Оба предыдущих класса содержат метод `getMove`, при помощи которого программа получает от них ход.

ЗАКЛЮЧЕНИЕ

В рамках работы над проектом были выполнены следующие задачи:

- Изучена игра шатар
- Изучена теорию шахматных движков
- Разработан шахматный движок

Следующими направлениями работы могут быть:

- Дополнительная оптимизация алгоритма
- Разработка графического интерфейса

СПИСОК ИСТОЧНИКОВ

1. Адельсон-Вельский, Г. М. Машина Играет в шахматы / Г. М. Адельсон-Вельский, В. Л. Арлазаров, А. Р. Битман, М. В. Донской; отв. ред. А. Ф. Волков. — М.:Наука, 1983. — 208 с.
2. Байнов, Е. В. Шатар — бурятские шахматы. / Е. В. Байнов, М. Е. Байнова. — Улан-Удэ: Детско-юношеская спортивная школа №8, 2015. — 26 с.
3. Корнилов, Е. Н. Программирование шахмат и других логических игр. — СПб.:БХВ-Петербург, 2005. — 272 с.
4. Шахматы на C++ // Хабр. — Режим доступа: https://habr.com/ru/post/682122/
5. Browne, Cameron. Bitboard Methods for Games // ICGA Journal. — 2014. — Vol. 37. — No. 2. — pp. 65-84.
6. Chess Programming Wiki : [Electronic Resource]. — URL: https://chessprogrammingwiki.org.
7. rang: A Minimal, Header only Modern c++ library for terminal goodies // GitHub — The world's leading software development platform. — Access mode: https://github.com/agauniyal/rang (access date: 17.03.2023).
8. Shatar // Wikipedia. The Free Encyclopedia. — Access mode: https://en.wikipedia.org/wiki/Shatar (access date: 17.03.2023).
9. Ward D. Maurer. Alpha-Beta Pruning // BYTE Magazine. — 1979. — Vol. 4. — No. 11. — pp. 82-96.

ПРИЛОЖЕНИЯ

Приложение 1

Ссылка на листинги программы: https://github.com/lubsanovdmitry/shatar

Ссылка на исходные файлы этого документа:

https://github.com/lubsanovdmitry/iip-docs