МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ ИМ. ПРОФ. М.А. БОНЧ-БРУЕВИЧА» (СПбГУТ)

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ СЕТЕЙ И СИСТЕМ (ИКСС)
КАФЕДРА ПРОГРАММНОЙ ИНЖЕНЕРИИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ (ПИиВТ)

Научно-исследовательская работа

Дисциплина: “Алгоритмы и структуры данных”

Тема: “Применение минимаксного метода принятия решений в игровых моделях”

Выполнили: Саганенко А. В
Студенты группы ИКПИ-04 Подпись _____
Саганенко Артемий Вадимович Куксин А. А
Куксин Александр Андреевич Подпись _____

Принял: Дагаев А. В
Доцент, кандидат технических наук Подпись _____
Дагаев Александр Владимирович «____»_____ 2023

Содержание

Введение

1. Актуальность

В современном мире очень важно быстро принимать решения, для этого используют самые различные алгоритмы. На принятие решение влияет очень много факторов, так что необходимо найти самый оптимальный алгоритм принятия решения. В данной научной работе будет рассмотрен Минимаксный метод алгоритм принятия решения, чья суть заключается в постоянном “взвешивании” всей ситуации и принятия наиболее выгодного решения из доступных. Данная работа направлена на изучение и моделирование исследуемого метода в практических целях, в частности в игре в “Шахматы”.

Существует множество алгоритмов для использования в игровых моделях. Однако некоторые являются нерелевантными, жадными, неэффективными или имеют слабую производительность. Необходимо найти, разработать алгоритм, который будет иметь свои преимущества в конкретных случаях. Также надо на примере доказать правильность этого алгоритма внутри уже готовой и устоявшейся системы.

2. Цель работы

Практическое и теоретическое доказательство того, что Минимаксный метод принятия решений является полноценным и релевантным для использования в игровых моделях и имеет вполне достойные показатели в реальных прикладных и бытовых задачах. Для практического доказательство нужно разработать программный продукт игру “Шахматы” и алгоритм для игры. В ходе игры против данного алгоритма, и изменения игровых событий внутри системы, нужно определить релевантность Минимаксного метода. Также Минимаксный метод должен быть оптимизирован различными способами, а сам алгоритм улучшен для улучшения совместимости с игрой и для улучшения производительности.

3. Задачи

В ходе работы над данным проектом следует придерживаться поочередного выполнения подзадач для достижения цели работы. Каждый следующий этап должен быть выполнен по факту успешного завершения предыдущего. В таком случае, работа будет целостна, а документация четкой и однозначной. Для анализа и создания алгоритма с внедренным Минимакс

методом необходимо провести теоретическую работу с “Шахматами”, Минимаксным методом, и практическую с программным внедрением алгоритма, использующего Минимаксный метод, в разработанные “Шахматы”. Следует придерживаться данных этапов разработки:

1. Теоретический анализ “Шахмат”
    1.1. Поверхностный анализ игры, в ходе которого будут разобраны общие принципы
    1.2. Более глубокий анализ игры, в ходе которого будут проанализированы тактики игры и изучена актуальность каждой позиции и каждой фигуры на игровой доске
    1.3. Формализация “Шахмат”, перевод всех условий, и возможных ходов из абстрактной модели в математическую. Провести проецирование субъективной модели в исключительно точную и объективную
2. Теоретический анализ Минимаксного метода
    2.1. Необходимо определить преимущества Минимаксного метода и его недостатки
    2.2. На примере понять как данной метод работает при определенных входных условия и узнать какие выходные значение он возвращает
    2.3. Разобрать метод оптимизации Альфа-Бета-отсечение, на примере показать как он улучшает оригинальный Минимакс
3. Математическое внедрение, интеграция метода в алгоритм для игры
    3.1. Рассчитать и представить алгоритм с помощью которого система будет определять цену ходу, передавать ее в Минимаксный метод и получать обратно лучший исход
    3.2. Здесь также необходимо представить способы взвешивание игровой доски и определения общей выгодности хода
4. Практическая реализация
    4.1. Реализовать игру “Шахматы” с визуальным интерфейсом для игры между двумя игроками
    4.2. Реализовать алгоритм для игры в “Шахматы”, теперь должна быть доступна функция игры между игроком и алгоритмом. Алгоритм должен сам производить ходы по решенному Минимаксному древу
5. Тестирование и доказательство гипотезы
    5.1. Протестировать “Шахматы” на наличие семантических, лексических ошибок, и ошибок формата

5.2. Доказать на примере гипотезу: “Минимаксный алгоритм обладает необходимыми и достаточными качествами для использования в игровых моделях”

5.3. Кратко провести сравнение с другой игровой системой. Выдвинуть концепт о принятии решений с помощью Минимаксного метода

5.3.1. Представить гипотезу об эффективности применении Минимаксного метода в прикладных и бытовых задачах

5.4. Проверить игру на намеренные попытки её сломать, на данном этапе найденные ошибки, уже не будут исправлены, но будут помечены в разделе о недостатках и багах

6. Документирование

6.1. Создать документацию проекта

6.1.1. Создать тезис, кратко изложить в нем указанные задачи и также кратко объяснить их решение

6.2. Произвести вывод и закрепить полученную информацию

6.3. Добавить источники и литературу

Глава I Теоретическая часть

1. Теоретический анализ Шахмат

Шахматы, будучи устоявшейся и полноценной игрой, имеет набор четко выверенных правил и методик игры. В данном случае поверхностный анализ провести достаточно просто, так как он не требует никаких специальных знаний и умений. Однако в процессе углубления в теорию, появляется все больше неизвестных и проводить качественных и всеобъемлющий анализ становиться все сложнее.

Полем игры является шахматная доска, которая представлена 8x8 клетками, с поочередным цветом (два легко отличимых цвета, чаще всего белый и черный). За пределы доски заходить нельзя. На каждой клетке может быть только одна фигура. Из 64 клеток в начале игры заполнено 32, где присутствуют 16 белых и 16 черных фигур. Они расположены на противоположных сторонах доски. Среди них: 2 короля, 2 ферзя (королевы), 4 коня (рыцаря), 4 слона, 4 ладьи, 16 пешек.

Всего присутствуют 6 типов фигур:

- Король, может двигаться в любую сторону на одну клетку, если таким образом, он не подставляется под шах, или не переходит ближе чем на две клетки к вражескому королю (рисунок. *Движение короля*). Король является главной целью игры. Его нельзя “съесть”, по факту мата игра завершается в пользу игрока который поставил мат.
- Ферзь, ходит в любую сторону на любое кол-во клеток (рисунок. *Движение ферзя*). Является по некоторым оценкам самой сильной фигурой.
- Ладья, ходит по горизонтали, вертикали на любое количество клеток (рисунок. *Движение ладьи*) С помощью ладьи можно выполнять рокировки.
- Конь, ходит буквой “Г” в любую сторону на один скачок (рисунок. *Движение коня*). Меняет цвет клетки при любом ходе.
- Слон, ходит по диагонали на любое количество клеток (рисунок. *Движение слона*). Один слон игрока всегда находится на клетке одного цвета, другой слон на клетке другого цвета. Не могут поменять цвет клетки.
- Пешка, ходит только вперед на одну клетку (если фигура еще не двигалась, она может передвигаться на две клетки), бьет по диагонали (рисунок. *Движение пешки*) В случае достижения пешкой конца доски,

она может быть превращена в любую другую фигуру. (Чаще всего в ферзя, но бывают случаи, где выгоднее превращать в коня)

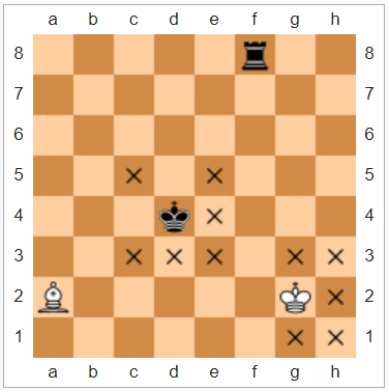

Рис.1 *Движение короля*

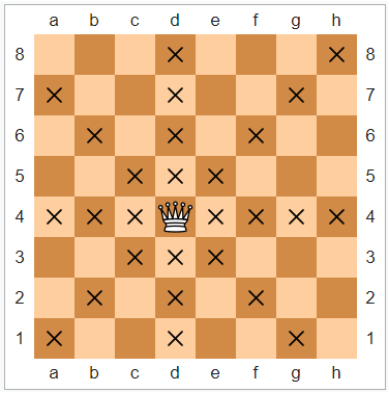

Рис.2 *Движение ферзя*

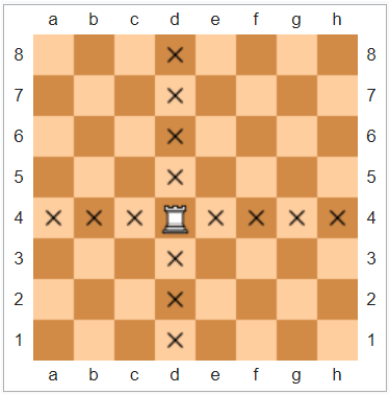

Рис.3 *Движение ладьи*

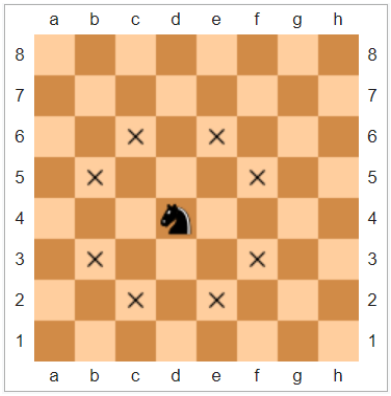

Рис.4 *Движение коня*

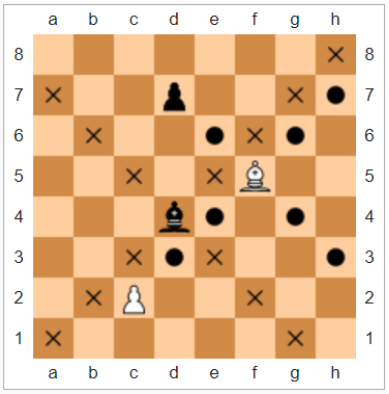

Рис.5 *Движение слона*

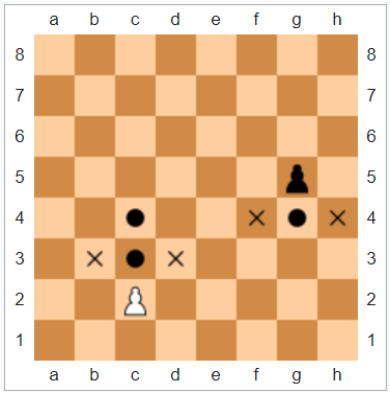

Рис.6 *Движение пешки*

Главной целью игры являться поставить шах и, последующий мат, вражеской фигуре, королю. Игроки имеют в своем распоряжении набор из фигур для достижения поставленной цели. Они поочередно меняют позиции фигур, для улучшения стратегической ситуации на поле и тактического превосходства в данный момент.

Шах — это игровая ситуация в которой король, находится под ударом вражеской фигуры. Теоретически мат происходит в ситуации, когда вражеская фигура имеет возможность “съесть” короля, если бы король мог быть “съеден”. Король под шахом должен быть выведен из-под шаха одним из трех способов. Игрок должен либо переместить короля в клетку, на которую нет шаха, либо прикрыть короля другой фигурой, либо “съесть” фигуру проецирующую шах на клетку. *(*рисунок. *Шах)*

Мат — это игровая ситуация в которой король, не имеет возможности выйти из-под шаха. В таком случае игрок, король которого под матом, проигрывает. *(*рисунок. *Мат)*

Пат — это игровая ситуация в которой ни один из игроков не может завершить партию победой.

Рокировка — это игровая возможность при которой король может поменять место, встав с другой стороны дружеской ладьи. Может быть проделана только один раз за игру, при условии, что ни одна из фигур (ни король, ни ладья) не ходила. (рисунок. *Рокировки*)

Взятие на проходе — это игровая возможность при которой пешка может “съесть” вражескую пешку. При условии, что вражеская совершила первый свой ход, переместившись на две клетки, и что первая пешка находилась справа или слева от клетки на которую переместилась вторая.

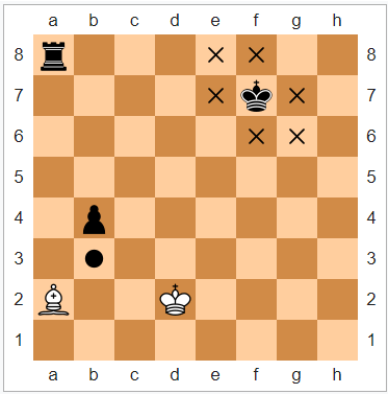

Рис.7 *Шах* (белый слон)

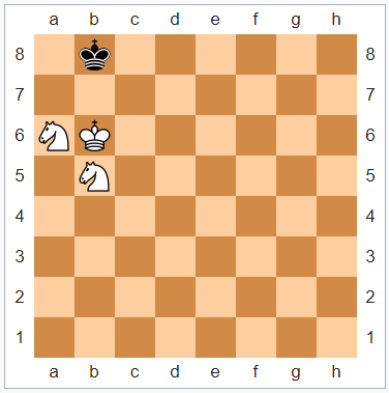

Рис.8 *Мат* (два коня)

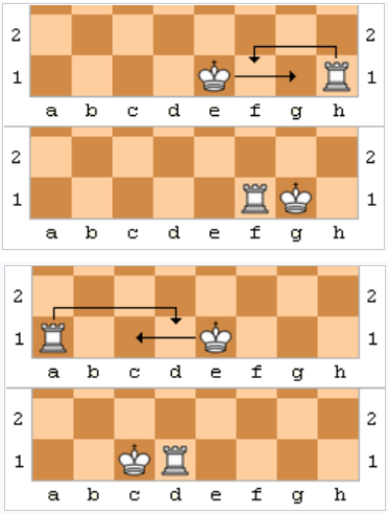

Рис.9 *Рокировки*

В профессиональных, и часто в любительских шахматах, присутствует ограничение по времени, которое в большинстве случаев составляет 5-15 минут на сторону, таким образом ускоряя процесс и заставляя игроков принимать более быстрые, но иногда менее взвешенные решения. В данном случае эта информация является исключительно обще направленной так как в самом алгоритме она не используется.

Шахматы являются одним из лучших примеров игр построенных на математике, каждый ход в Шахматах имеет свой вес, свою стратегическую выгоду. Таким образом задача формализации является очень простой в концептуальном плане, но очень сложной в алгоритмическом. Достаточно

легко, взглянуть на игровую модель с точки зрении математики, представить ее. Добавить веса фигурам, веса каждой клетки доски, в теории просчитать можно всю игру, от первого до последнего хода, однако, когда дело доходит до реализации появляются серьезные проблемы.

1. Современное аппаратное обеспечение не способно удовлетворять требованиям алгоритмов. Если судить по информации представленной на http://tromp.github.io/chess/chess.html существует 10 ^ 45.88 или 772877297796591967716487348768545313732973652 2 возможных позиций в Шахматах, просчитать выгоду каждой и их изменение в ходе игры на данный момент невозможно
2. Несмотря на то, что Шахматы поддаются математическому анализу, формализации и последующая алгоритмизации не могут быть выполнены идеально, так как аналоговая информация, то-есть не математическая (как человек воспринимает имеющуюся информацию), трудно поддается переводу, в данном случае невозможно создать грамотную и надежную систему весов как для фигур так и для доски
3. Человеческая игра имеет фактор, зачастую ломающий логику алгоритмов. Выбор худшего из всех вариантов может привести к победе человека, несмотря на сырую математику, которая говорит алгоритму, что человек выбрал самый невыгодный ход из доступных.

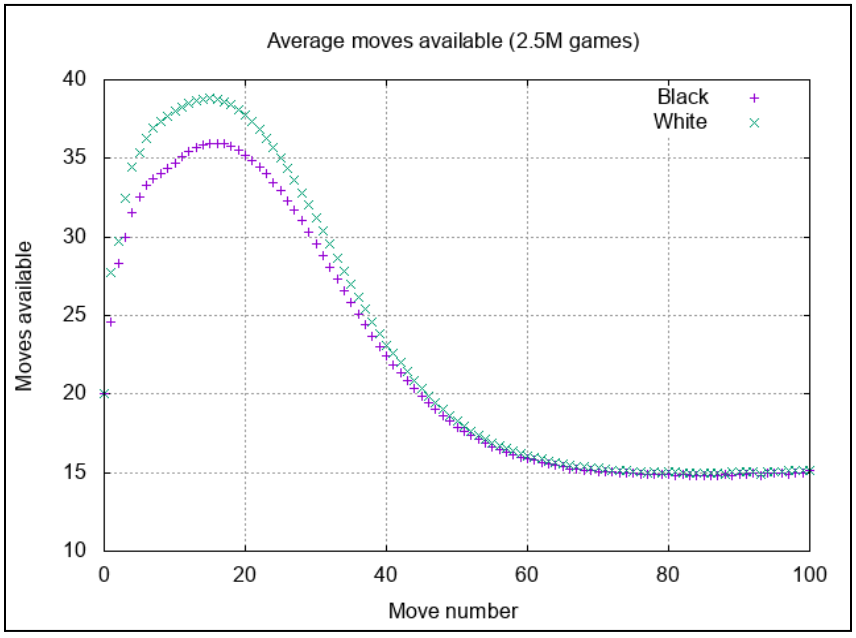

Рис.10 *Среднее количество доступных ходов*. (Источник: https://chess.stackexchange.com)

Перевод: Легенда сверху — Среднее количество доступных ходов (рассчитано на 2,5 миллионах игр), слева — количество доступных ходов,

снизу — номер хода, фиолетовый плюс — черные ходы, зеленый крест — белые.

Несмотря на это было разработано множество разнообразных алгоритмов для игры в шахматы, многие из которых при этом могут похвастаться своей эффективностью. Они используют разные способы формализации Шахмат и оценки ситуаций (позиций). Но в целом следуют одним и тем же тенденциям, что по-своему негативно сказывается на развитии алгоритмов, но положительно на их оптимизацию. В целом судя по статистике представленной на https://aiimpacts.org/historic-trends-in-chess-ai/ и информации взятой из статьи “Computer Science as Empirical Inquiry: Symbols and Search.” (Компьютерная наука как Эмпирический метод исследования: Символы и Поиск), за авторством Аллена Ньювелла и Херберта Симона, шахматный искусственный интеллект на данный момент представляет собой серьезного противника для новичков и любителей. Со средними шансами на победу даже у профессиональных игроков.

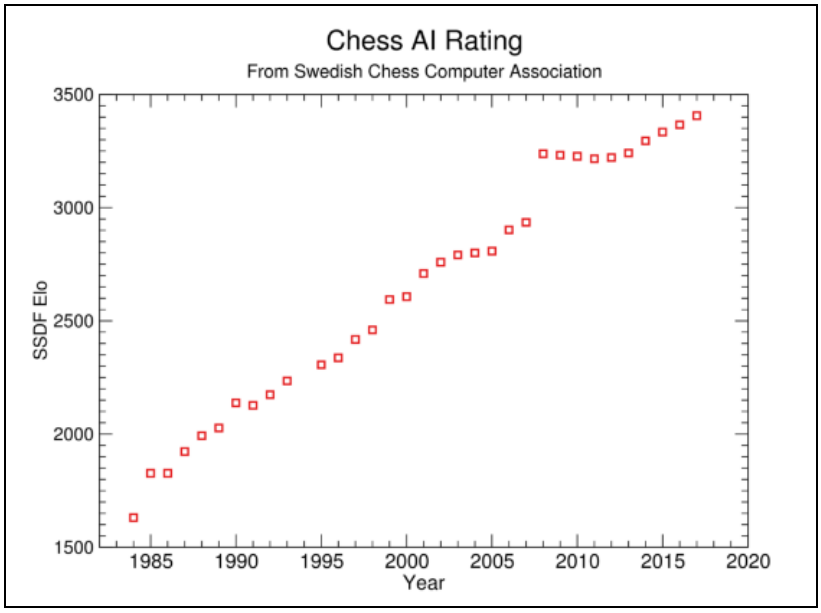

Рис.11 *Рейтинг игры в Шахматы искусственного интеллекта (ИИ).* (Источник: https://aiimpacts.org/historic-trends-in-chess-ai/)

Перевод: Легенда сверху — Рейтинг Шахматных ИИ от Шведского Сообщества Компьютерных Шахмат (ШСКШ), слева - Elo, или же рейтинг от ШСКШ, снизу — год.

Со временем более статичные методы анализа превратились в сложные системы которые динамически (в режиме реального времени), принимают

решения по взвешенным данным. Следуя из этого можно сделать вывод, что в целом математическое сообщество прошло много этапов анализа, формализации, алгоритмизации Шахмат, и на данный момент этого достаточно для победы машины над человеком. При этом, как уже я было упомянуто, почти любой шахматист профессионал, скорее всего все же победит в партии против алгоритма.

В любом случае мы приходим к тому, что есть два решения проблемы создания алгоритма для игры в шахматы. Создания математического алгоритма, который является менее оригинальным и производительным, но при этом будет иметь заведомо неплохие и статичные шансы на победу. Он будет использовать арифметику как способ решения доски, то-есть поиска хода. Или создания нейросети, которая будет использовать уже сыгранные партия для просчета ходов, она будет более производительной и оригинальной, но её шансы на победу будут гораздо более стохастическими.

В первом случае, перед созданием алгоритма необходимо формализовать шахматы. Перевести игровую модель из абстрактных или аналоговых (человеческих) суждений о стратегической и тактической выгоде хода и ситуации на бумагу, иначе говоря в цифры. Для этого нужно создать систему оценки доски, фигур, ходов.

Исходя из грубой математики и эмпирического исчисления весов создается таблица силы каждой из фигур. Существует несколько устоявшихся констант весов для фигур, но одновременно с изменения манеры игры и развитием технологий эти константы изменяются. Чаще всего это происходит опытным путём и закрепляются в ходе математического моделирования. В любом случае решающей формулой является формула материала, которая заключается в вычислении разности между весом всех фигур одной стороны и другой. Саму цену фигуры можно определить статично. В таблице. *Веса фигур*, например указаны некоторые варианты, вычисленные в разные года. Очевидно, что с развитием науки, в частности алгоритмизации Шахмат, эти значение становились лучше, то-есть более оптимизированными, более точно представляющими общую игровую ситуацию.

Source	Year	Pawn	Knight	Bishop	Rook	Queen
H. S. M. Coxeter [7]	**1940**		300	350	550	1000
Max Euwe and Hans Kramer [8]	**1944**	100	350	350	550	1000
Claude Shannon [9]	**1949**	100	300	300	500	900
Alan Turing [10]	**1953**	100	300	350	500	1000
Mac Hack [11]	**1967**	100	325	350	500	975
Chess 4.5 [12]	**1977**	100	325	350	500	900
Tomasz Michniewski [13]	**1995**	100	320	330	500	900
Hans Berliner [14] [15]	**1999**	100	320	333	510	880
Larry Kaufman [16]	**1999**	100	325	325	500	975
Fruit and others [17]	**2005**	100	400	400	600	1200
Larry Kaufman [18]	**2012**	100	350	350	525	1000

Таб.1 *Веса фигур*

Перевод: Первая строка, слева направо по столбцам: источник, год, пешка, конь, слон, ладья, ферзь.

Исходя из главной задачи игры, то-есть шах и мат вражескому королю, королю объявляется самый большой вес на игровой доске, который при любых обстоятельствах должен быть больше самой выгодной ситуации для игрока на поле (все фигуры игрока стоят в лучшей клетке, у врага остался только король в уязвимом месте). Таким образом алгоритм, будет воспринимать шах королю как самый выгодный ход в любой игровой ситуации. Что частично является недостатком, но не настолько критическим.

Для оценки положений фигур на доске была также создана специальная система. Это система в первую очередь предполагает, что алгоритм использует математику для вычисления выгодности хода. Таким образом используя матрицу весов для каждой клетки поля и каждого типа фигуры можно стимулировать алгоритм для перемещения фигур по доске, заставляя его, например выводить коня и пешек как можно раньше.

Матрица подобных коэффициентов зависит и в том числе от количества возможных ходов для фигуры в определенной клетке. На рисунок. *Количество ходов для фигуры*, можно видеть сколько разных ходов может совершить фигура, которая находиться в определенной клетке.

White pawn total

2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3	2	22
3	4	4	4	4	4	4	3	30
13	19	19	19	19	19	19	13	140

white pawn a2

2	3	3	3	3	3			17
2	3	3	3	3				14
2	3	3	3					11
2	3	3						8
2	3							5
3								3
13	15	12	9	6	3			58

white pawn d2

2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3	2	22
2	3	3	3	3	3	3		20
	3	3	3	3	3			15
		3	3	3				9
			4					4
6	12	15	19	15	12	9	4	92

Knight

2	3	4	4	4	4	3	2	26
3	4	6	6	6	6	4	3	38
4	6	8	8	8	8	6	4	52
4	6	8	8	8	8	6	4	52
4	6	8	8	8	8	6	4	52
4	6	8	8	8	8	6	4	52
3	4	6	6	6	6	4	3	38
2	3	4	4	4	4	3	2	26
26	38	52	52	52	52	38	26	336

King

3	5	5	5	5	5	5	3	36
5	8	8	8	8	8	8	5	58
5	8	8	8	8	8	8	5	58
5	8	8	8	8	8	8	5	58
5	8	8	8	8	8	8	5	58
5	8	8	8	8	8	8	5	58
5	8	8	8	8	8	8	5	58
3	5	5	5	5	5	5	3	36
36	58	58	58	58	58	58	36	420

Bishop

7	7	7	7	7	7	7	7	56
7	9	9	9	9	9	9	7	68
7	9	11	11	11	11	9	7	76
7	9	11	13	13	11	9	7	80
7	9	11	13	13	11	9	7	80
7	9	11	11	11	11	9	7	76
7	9	9	9	9	9	9	7	68
7	7	7	7	7	7	7	7	56
56	68	76	80	80	76	68	56	560

Rook

14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
14	14	14	14	14	14	14	14	112
112	112	112	112	112	112	112	112	896

Queen

21	21	21	21	21	21	21	21	168
21	23	23	23	23	23	23	21	180
21	23	25	25	25	25	23	21	188
21	23	25	27	27	25	23	21	192
21	23	25	27	27	25	23	21	192
21	23	25	25	25	25	23	21	188
21	23	23	23	23	23	23	21	180
21	21	21	21	21	21	21	21	168
168	180	188	192	192	188	180	168	1456

Рис.12 *Количество ходов для фигуры*

Перевод: По часовой стрелке с левого верхнего угла: белая пешка общее, белая пешка а2, белая пешка д2, король, ферзь, ладья, слон, конь.

На примере этой таблицы хорошо видна выгодность вывода коня в центр поля, там он достигает максимального количества возможных ходов. Далее в зависимости от позиций и типов фигур игрока вычисляется общий вес (материал) фигур. Происходит сложение весов позиций и фигур. От вычисленного целочисленного значения вычитается также просчитанный вес всех фигур врага. Получается целочисленная величина (положительная либо отрицательная) определяющая общую ситуации на доске, в игровой системе.

Минимаксный алгоритм для игры подразумевает использование именно данной величины для построения древа и впоследствии нахождения самого выгодного хода из заведомо идентичных. Теоретически, и иногда на практике, к этим двум элементам взвешивания, добавляются шаблоны конкретных игровых ситуаций, это может быть набор фигур для успешной игры, например два ферзя могут поставить мат королю без защиты за два хода, либо

расположения фигур на доске которые подразумевают возможность быстрой победы и так далее. На сайте www.chessprogramming.org представлена более подробная информация по данной теме.

Используя данные математические значения и матрицы, мы частично справились с задачей формализации Шахмат. Данные числа можно использовать в алгоритме для оценки доски в уже практической части данной работы.

2. Теоретический анализ Минимаксного метода

Модель метода принятия решений Минимакс

Минимаксный алгоритм — это метод принятия решений который используют в том случае, когда выбор состоит из большого множества ситуаций, условно, такой выбор моделируют в виде древа возможных решений. К примеру, так работает большое количество возможных исходов шахматных партий. Подобные ситуации, на самом то деле, очень часто встречаются почти во всех проявлениях возможности выбора и принятия решений, но более всего распространено в сфере игр. Вспомните любую игру в которую вы играли ранее, и любую ситуацию связанную с этой игрой. Допустим тот факт, что игру можно продолжить огромным количеством возможных исходов. Каждый из исходов будет отличаться друг от друга, и каждый будет иметь свою “цену”, то есть один из возможных исходов будет более выгодным чем другой. Не важен метод получения оценки данной “цены”, для понимания метода нужно понять что “цена” это какое-то определенное число.

Оценивание всех возможных результатов и представление их в виде какого либо числа — это все что нужно для использования в ходе определения наиболее благополучного и выгодного решения. Метод Минимакс описывает использование полученных контрольных сумм оценки исходов для принятия самого лучшего решения для конкретной ситуации.

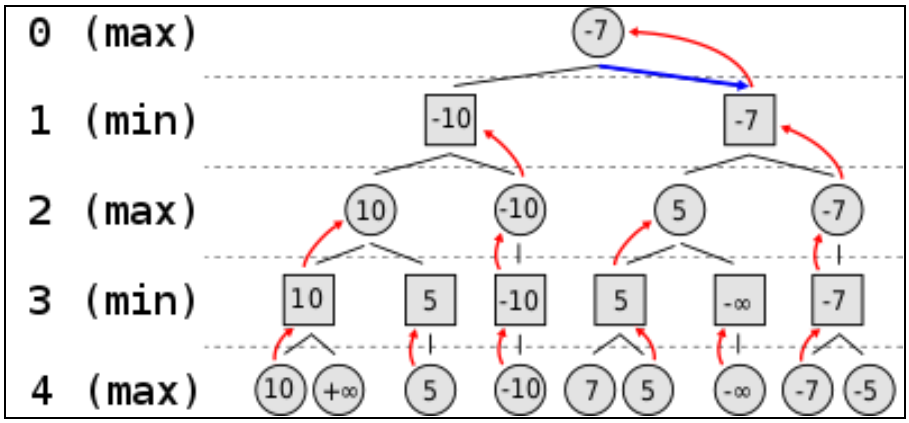

Рис.14 *Минимаксное древо*

На рис. *Минимаксное древо* представлено древо построенное алгоритмом Минимакс, на каждом этапе, алгоритм оценивает и выбирает из доступных решений то, которое более удовлетворяет заданному условию, в частности выбирать для себя максимально выгодную ситуацию, а для противника наиболее проигрышную ситуацию. Как и видно на представленном древе, алгоритм выбирает более выгодные для него стратегии принятия решений.

На сегодня, минимаксный метод принятия решение может применяться в ряде прикладных задач и в множестве имитационных (игровых, абстрактных) моделей. Например: в играх (в основном интеллектуальных), в геодезии, в имитационных моделях транспортных сетей, в радио размещении и так далее.

Главными преимуществами данного метода является простота его внедрения и реализации, средняя эффективность, несмотря на алгоритмическую простоту, а также его очевидность и императивность. Главные недостатки, жадность, что частично исправляется Альфа-Бета-отсечением, слабая производительность на ветвистых (где у главной ячейки очень много наследников), высоких (где много уровней) деревьях. Также следует отметить то, что метод часто применяется в философии и бытовых задачах чтобы эффективно “отрезать” заведомо ненужные (неверные) ветви. При этом необходимо помнить, что в сухом остатке метод работает с сырыми математическими значениями, которые могут быть формализованными суждениями или реальными значениями.

Альфа-бета-отсечение

Альфа-бета-отсечение — алгоритм, который сокращает количество узлов, что существуют в древе построенным алгоритмом принятия решений

Минимакс. В основе идеи алгоритма лежит следующий принцип: при оценивании ветвей древа построенным алгоритмом Минимакс может досрочно прекратиться, если обнаружено, что для этой ветви значение оценивающей функции в любом случае хуже, чем вычисленное для предыдущей ветви. Альфа-бета-отсечение является оптимизацией, так как не влияет на корректность работы алгоритма.

Преимущество альфа-бета-отсечения заключается в следующем, некоторые из ветвей подуровней дерева поиска могут быть исключены после того, как хотя бы одна из ветвей уровня рассмотрена полностью. На эффективность метода существенно влияет предварительная сортировка вариантов — при сортировке чем больше в начале рассмотрено «хороших» вариантов, тем больше «плохих» ветвей может быть отсечено без исчерпывающего анализа.

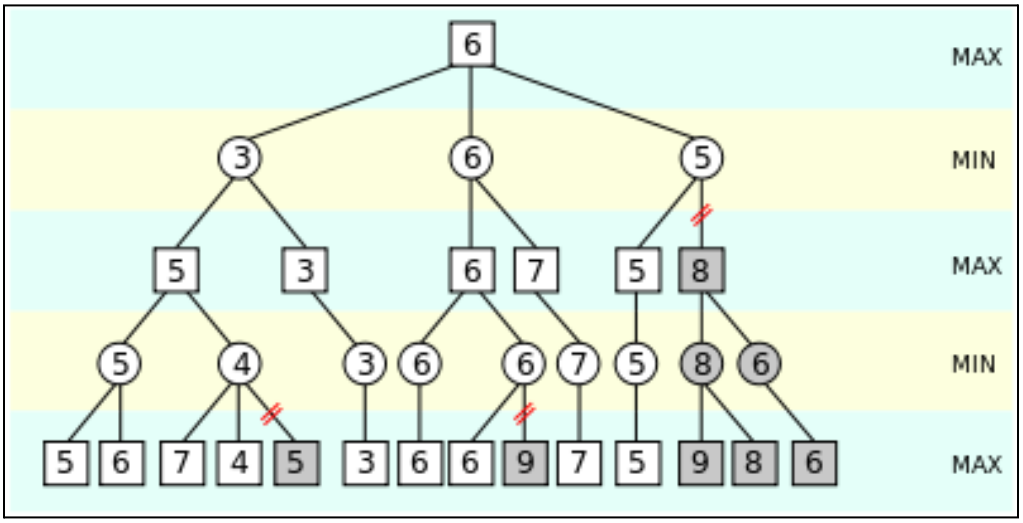

Рис.15 *Альфа-Бета отсечение*

Пусть мы рассмотрели уже один свой первый ход и все возможные ответы соперника и начали рассматривать свой второй ход. Как только среди ответов соперника, появится такой ход, который ставит нас в ситуацию хуже, чем самая плохая ситуация, в которую может нас поставить соперник (если мы сходим первым ходом), в таком случае можно прекращать перебор возможных ответов соперника на второй ход и переходить к рассмотрению следующего нашего третьего хода. Рассуждение выше можно проводить на каждом уровне, рассуждая как за себя, так и за соперника. По сути для реализации идеи мы должны помнить лучший результат, достигнутый на предыдущем уровне. Назовём лучший результат с предыдущего уровня параметром alpha. Можно использовать в своих целях лучший результат с

пред-предыдущего уровня. Это будет параметром beta. Таким образом, если мы угадаем промежуток, в котором лежит реальная оценка ситуации, то получим верную оценку, а иначе мы узнаем лишь с какой стороны от промежутка [alpha, beta] лежит правильный ответ.

Альфа-Бета-отсечение частично исправляет жадность Минимаксного древа, убирая самые невыгодные варианты с ходу тем самым значительно упрощая все древо и улучшая производительность Минимаксного метода прямо пропорционально количеству убранных нодов (nodes, возможных выборов). Альфа-Бета-отсечение являться способом оптимизации, он не может работать в одиночку. Большая часть всех реализованных Минимаксных методов в прикладных и научных задачах используют Альфа-Бета-отсечение.

3. Теоретическое внедрение

Формализованную игровую модель, метод оценки игровой ситуации, метод оценки доски и Минимаксный метод с Альфа-Бета-отсечением можно использовать вместе для создания алгоритма для игры в Шахматы. Мы имеем модель и способ ее реализации. У нас есть все необходимые качества для достижения цели алгоритмизации Минимаксного метода принятия решения вместе с Альфа-Бета-отсечением внутри игровой модели Шахмат.

Распишем теоретически поэтапно алгоритм:

1. Производим оценивание доски в данный момент времени, до совершения хода. Получаем целочисленное значение, которое зависит от фигур и их позиций на поле
2. Получаем все возможные ходы, которые может сделать сторона в игре, получаем список всех возможных ходов
3. Создаем временное виртуальное Шахматное поле
4. Внутри виртуального Шахматного поля совершаем ход
5. Производим оценивание виртуальной доски, получаем новое целочисленное значение, сравниваем со старым значением. Если значение стало лучше для нашей стороны, сохраняем ход и его вес.
6. Проверяем так же каждый возможный ход. Если какой-то ход еще лучше заменяем старый новым.

Если глубина больше 1:

1. На каждый совершенный одной стороной ход, тестируем также все возможные ходы противоположной стороны. В таком случае берем не самое лучшее значение, а самое худшее.

2. Затем берем снова первую сторону и производим вычисления. Производим вычисления до достижения желаемой глубины.
   На примере:

1. Оценка доски в данный момент - 100
2. Кол-во возможных ходов - 23
3. Виртуальная доска с оценкой позиции 100
4. Проверяем каждый из 23 ходов.
    a. Возможный ход 1: совершаем ход, оценка виртуальной доски - 90
    b. Возможный ход 20: совершаем ход, оценка виртуальной доски - 110. Лучше, чем старое значение запоминаем ход 20
    c. Возможный ход 21: совершаем ход, оценка виртуальной доски - 115. Лучше, чем старое значение запоминаем ход 21 вместо 20-ого
    d. Возможный ход 23: совершаем ход, оценка виртуальной доски - 100
5. Ход 21 был лучший. Совершаем его на оригинальной доске.
   На примере с глубиной 2:

1. Оценка доски в данный момент - 250
2. Кол-во возможных ходов - 20
3. Виртуальная доска с оценкой позиции 250
4. Проверяем каждый из 23 ходов.
    a. Ход 20. Оценка - 270 (по пункту v. Превращается в 200)
        i. Оцениваем виртуальную доску - 270
        ii. Кол-во возможных ходов - 25
        iii. Ход 10. Оценка - 240
        iv. Ход 15. Оценка - 290. Лучший исход
        v. **Ход 20. Оценка - 200. Худший исход**
    b. Ход 21. Оценка - 215 (по пункту iii. Превращается в 215)
        i. Оцениваем виртуальную доску - 190
        ii. Кол-во возможных ходов - 10
        iii. Ход 5. Оценка 215. Худший исход
        iv. Ход 9. Оценка 410. Лучший исход
    c. Ход 21 (вес 215) будет выбран между 20 и 21 (между вес 200 и вес 215), так как противник в случае с ходом 20 (вес 210) может создать гораздо более плохую ситуацию нежели, чем при ходе 21 (вес 215).

Даже при условии, что ход первой стороны был лучше.

5. Ход 21 был лучший. Совершаем его на оригинальной доске.

Стоит отметить опять пункт проблем разработки алгоритмов для игры в Шахматы под номером 3 - “Человеческая игра имеет фактор, зачастую ломающий логику алгоритмов. Выбор худшего из всех вариантов может привести к победе человека, несмотря на сырую математику, которая говорит алгоритму, что человек выбрал самый невыгодный ход из доступных.” В пункте из примера с глубиной два под символами iii и v алгоритм рассчитывает, что человек, выберет лучший из его доступных ходов, но это может быть далеко не так. И в таком случае алгоритм может принять неверное решения, исходя из неверного предположения об ответном ходе, при этом при повышении глубины, такая ошибка будет иметь все более серьезные последствия. Даже на представленном примере видно, что если человек выбрал ход 15 (вес 410) после алгоритма ход 20 (вес 270), алгоритм не предполагал бы такого и заведомо пропустил возможность улучшить игровую ситуацию.

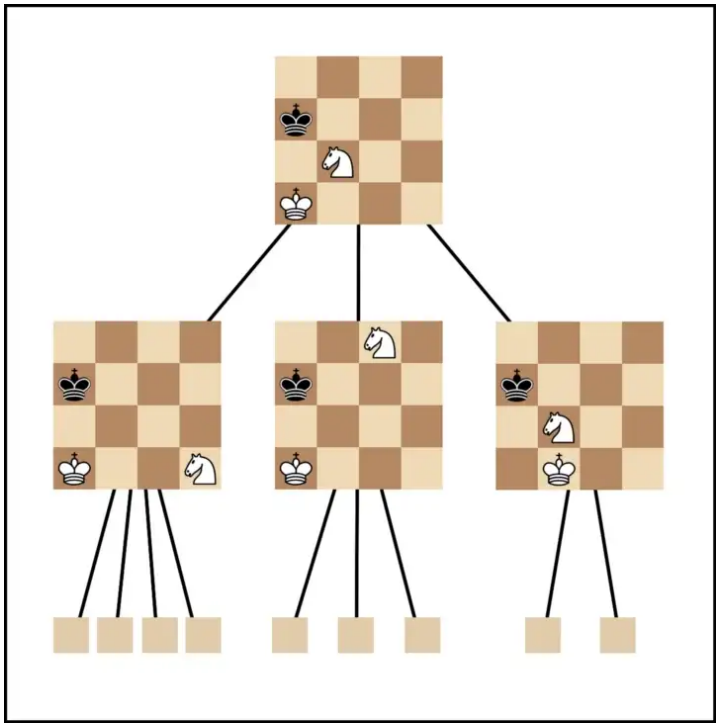

Рис.16 *Минимакс на примере.* (Вариант посередине будет выбран так как при нем ставиться шах)

В то же время с повышением глубины алгоритм создает более цельную картину игры, и в отличие от человека с каждым ходом становиться все

сильнее, имея меньшую выборку ходов, и растущую игровую усталость человека-противника.

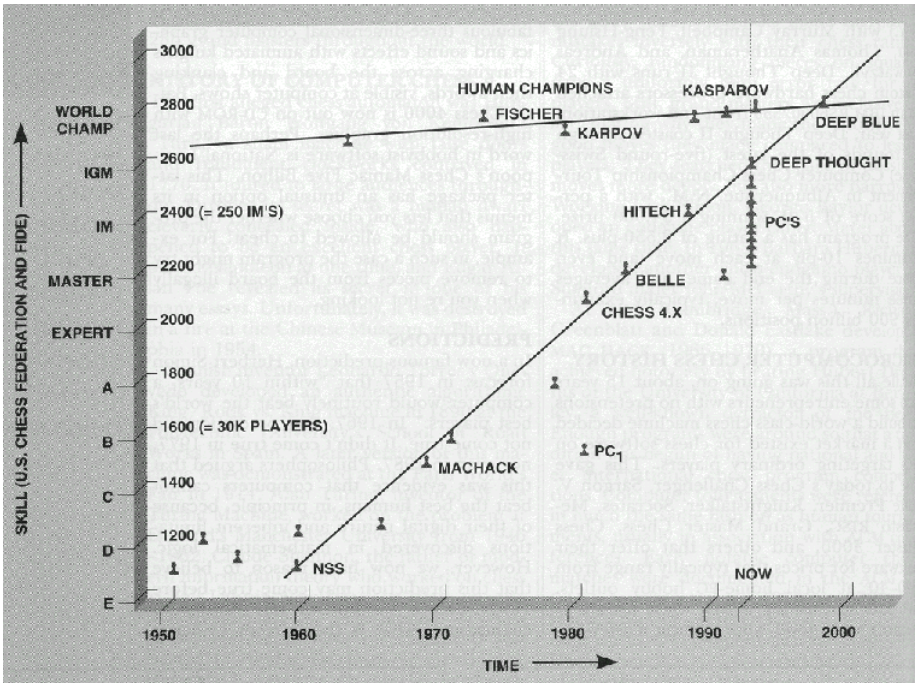

Рис. 17 *Сравнение рейтинга между ИИ и игроками*.
Перевод: Легенда: слева: навык по рангу Федерации Шахмат США, снизу: время.

Рассмотрим также данный метод на примере игры “Крестики-нолики”, это в то же время будет служить примером работы Минимаксного метода как части алгоритма для другой игровой системы. Данная игровая модель является более простой, но при этом все же математической и, более того, полностью формализованной. Внутри “Крестиков-ноликов” Минимаксный метод являться эффективнейшим методом для решения (победы), который выводит игру только в ситуации “победа” или “ничья”, тем самым обеспечивая максимально возможную частоту побед. Рассмотрим алгоритмическую реализацию Минимаксного метода в данной модели на рисунок. *Минимакс в Крестиках-ноликах.*

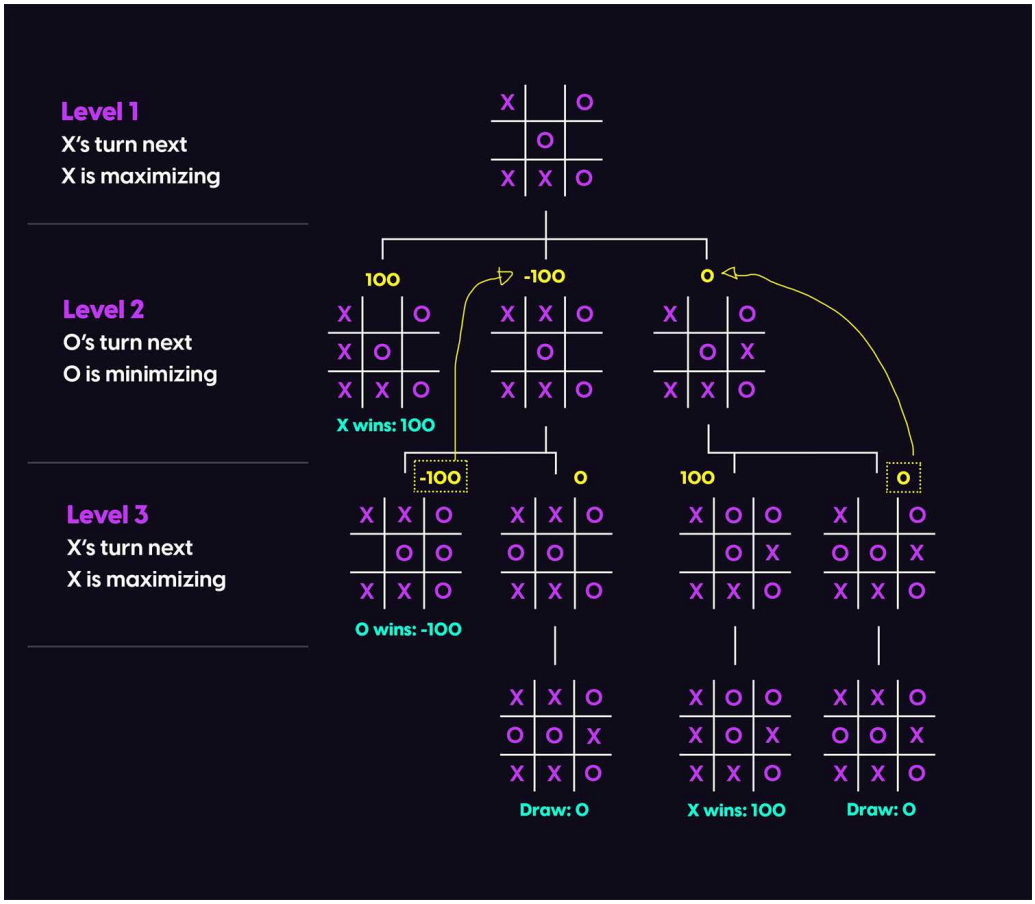

Рис.18 *Минимакс в Крестиках-ноликах.*

Перевод: По строкам разделены точкой с запятой: Уровень 1. Крестик следующий, он максимизируется (алгоритм играет за крестик); Уровень 2. Нолик следующий, он минимизируется, левый случай Крестик побеждает; Уровень 3. Крестик следующий, он максимизируется, левый случай Нолик побеждает; Левый случай ничья, середина Крестик побеждает, правый ничья

По этому примеру видно, что данный алгоритм не проигрывает, так как эта модель намного проще чем шахматная и ее можно просчитать от начала и до конца. Отличный пример того, чтобы было если бы аппаратное обеспечение позволяло просчитывать Шахматы от первого хода до последнего. У человека не было бы шансов на победу.

Глава II Практическая часть

1. Разработка программы

Стэк разработки

Стэк разработки ЯП Python3. В том числе по уровням реализации:

PyGame - визуальное представление, рендеринг содержания окна

Python + Sys + time - для работы с ходами управление временем и окнами

Python + math + numpy + random + copy - для алгоритма для игры в Шахматы

Python + logging + screeninfo - для конфигурации и настройки игры, в том числе определение рабочего экрана и его разрешение

Для написания документации использовались:

Google Docs (https://docs.google.com/)
Draw.io (https://app.diagrams.net/)

Для поддержки, развертки, интерпретации и разработки:

VS code (https://code.visualstudio.com/)
GitHub (https://github.com/)
Replit (https://replit.com/~)

Версии пакетов необходимых для интерпретации и запуска указаны в файле зависимостей requirements.txt

numpy==1.24.1
pygame==2.1.2
screeninfo==0.8.1

Версии Python3, на которых на разных этапах разработки тестировался и реализовывался код:

Первая реализация:
Python 3.9
Полная совместимость:
Python 3.6.9
Python 3.11.0
Каждая версия собиралась на Ubuntu 18.04 и Windows 10 версия 10.0.19044 Сборка 19044 на одинаковых условиях и независимо

Графическое представление структуры программы

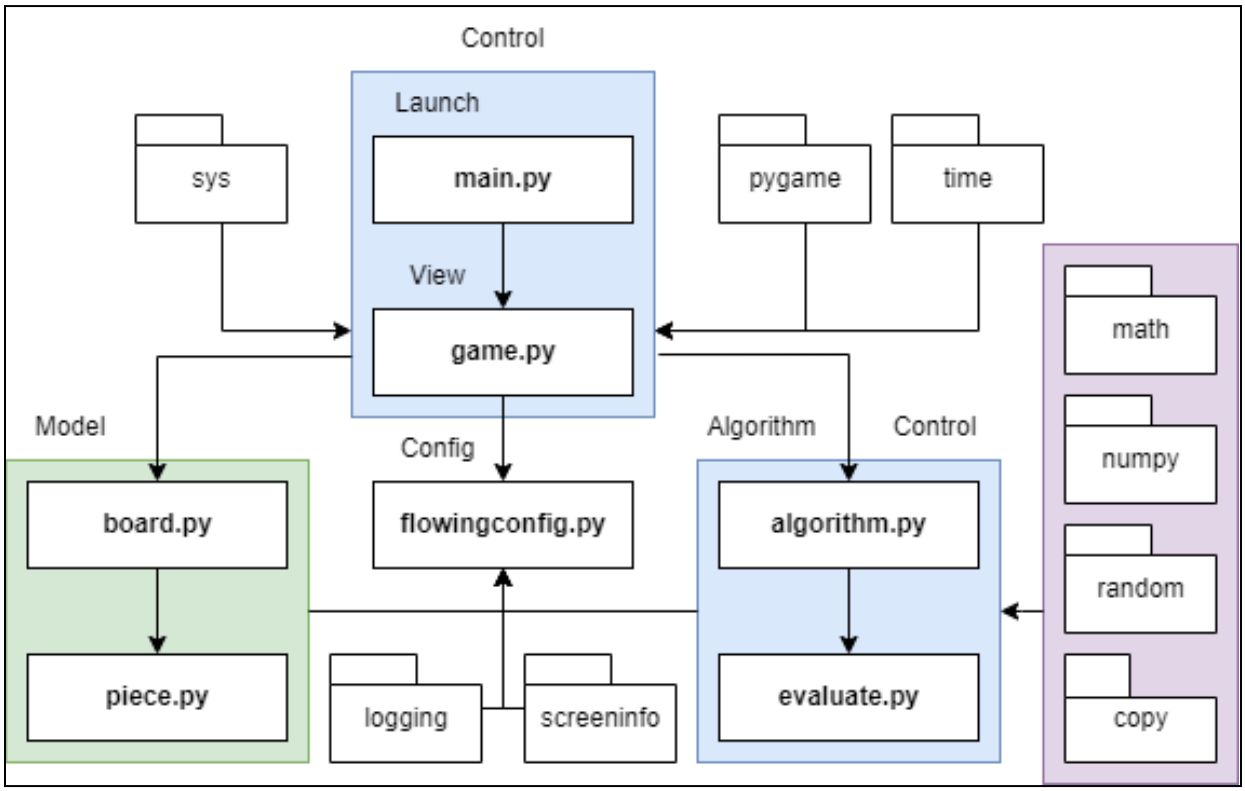

Рис.19 *Диаграмма модулей*

На рисунке 19 изображена схема модулей и зависимостей. На диаграмме отсутствуют номера сборок/версий. Можно обратить внимание на то, что в разделе модели (model) присутствуют два модуля-класса board.py, piece.py, которые соответственно хранят информацию о шахматной доске и шахматных фигурах. Также есть раздел с алгоритмами для игры (algorithm control), где расположены сами алгоритмы в модуле algorithm.py, которые зависят от функций взвешивания из модуля evaluate.py, и раздел контроля за процессом игры, модули game.py, main.py (control). Там же располагается раздел визуального представления (view). Последний раздел представляет конфигурацию проекта (config) вся конфигурация обрабатывается через модуль flowingconfig.py, статичные значения хранятся в txt файле, config.txt.

Модули:

- main.py - управляющий метод. По сути ничего не делает, запускает метод game из game.py
- game.py - главный модуль игры. Хранит в себе саму игру, рендеринг окна, выполнение хода, реализацию победы, поражения, ничьи и так далее. Вызывает методы из board.py, algorithm.py.

- board.py - класс Board для доски. Имеет массив элементов из Piece или int. Содержит ключевые функции контроля select и move, выбрать ячейку и выполнить ход
- piece.py - классы для фигур. Главный абстрактный класс Piece и наследники для каждого типа фигуры
- algorithm.py - алгоритмы для игры. Все 4 уровня описаны здесь, использует две функции из evaluate.py
- evaluate.py - оценка доски. Простая (просто материал) и продвинутая (фигуры в определенных клетках доски).
- flowingconfig.py - “плавающий конфиг”. Динамически просчитанная конфигурация игры, зависит от статичного текстового файла config.txt.

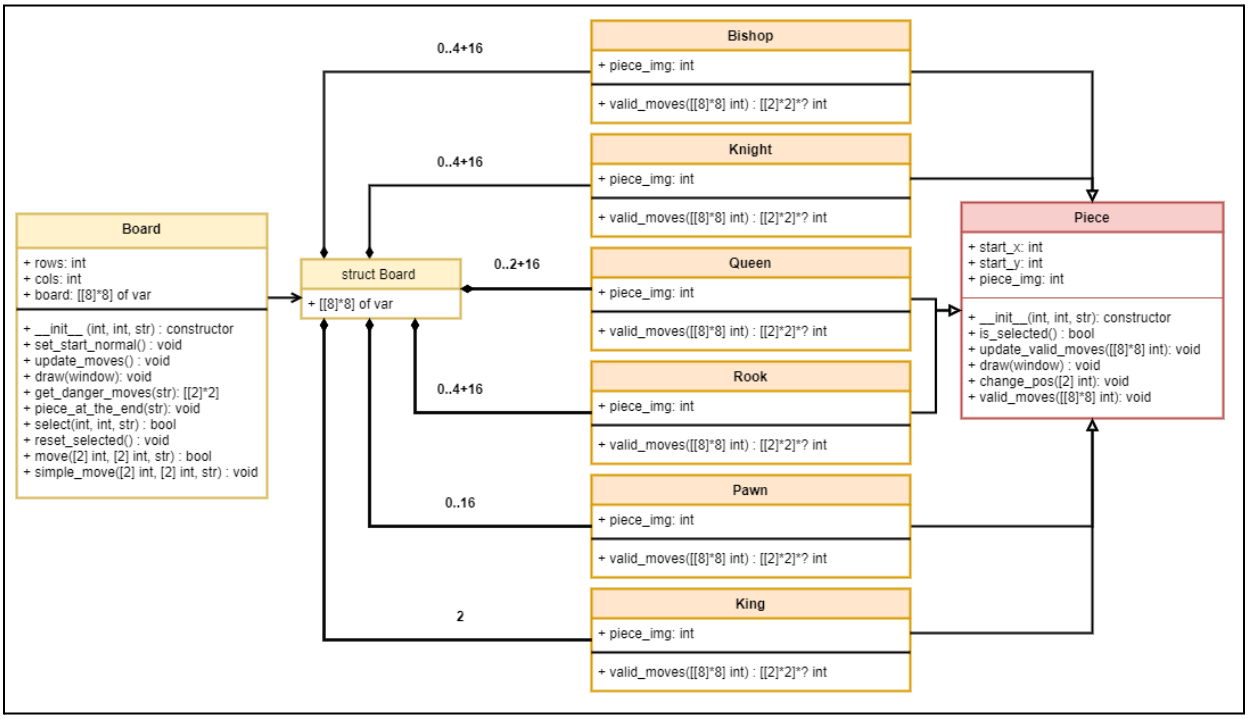

Рис.20 *Диаграмма классов модели*

На рисунке 20 изображена UML-диаграмма классов. Присутствуют не все классы и модули, а только классы модели (model). В первую очередь изображен класс фигуры Piece, и унаследованные конкретные типы фигур Bishop, Knight, Queen, Rook, Pawn (на схеме не указан параметр pawn и first, указывающие на то, что фигура является пешкой и она еще не ходила), King.py (на схеме не указан параметр king - показатель того, что фигура король). Также показан класс шахматной доски Board вместе с матрицей board, которая содержит либо int’ы, либо Piece’ы. Эти два элемента являются

ключевыми в данном работе, так как именно они имитируют саму шахматную доску.

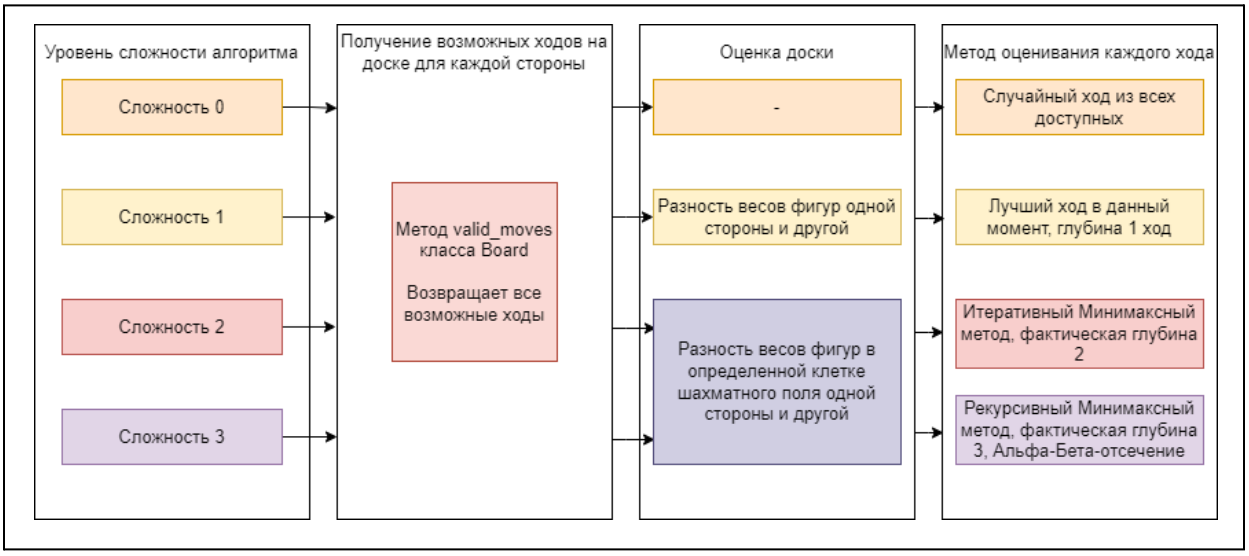

Рисунок.21 *Пользовательская диаграмма уровней сложности*

На рисунке 21 схематично изображены уровни сложности. Можно обратить внимание на разные этапы работы алгоритмов и осуществления их методик. Начиная с метода valid_moves, который возвращает все доступные ходы, заканчивая методом оценивания доски, и (не указанным) методом совершения хода, в данном случае simple_move.

Существует два метода для совершения хода. Разница между ними не велика, по сути алгоритмы идентичны, однако метод move (созданный для человека) имеет большее количество проверок, в первую очередь на шах и мат. Функция simple_move (для алгоритма) такими проверками не обладает, так как если алгоритм не находит никакого хода, который является более эффективным чем мат самому себе, значит других доступных ходов и нет, в таком случае он проигрывает.

Выдержки из кода

```
def tier1_choice(self, color):
    def root_minimax(board, depth, maximizing):
        best_value_root = -inf
        maxed_move_root = self.random_choice(color)
        all_pieces_1 = get_all_pieces(board, color)
        for piece_1 in all_pieces_1:
            for move_1 in piece_1.move_list:
                next_board = copy.deepcopy(board)
                next_board.simple_move((piece_1.row, piece_1.col), (move_1[1], move_1[0]), color)
```

```
                if not next_board.piece_is_checked("b"):
                    value = minimax(next_board, depth - 1, -inf, inf, not maximizing)
                    if value >= best_value_root:
                        best_value_root = value
                        maxed_move_root = (piece_1.row, piece_1.col), (move_1[0], move_1[1])
        return maxed_move_root

    def minimax(board, depth, alpha, beta, maximizing):
        if depth == 0:
            return evaluate_board_advanced(board, "b")
        new_board_1 = copy.deepcopy(board)
        if maximizing:
            all_pieces_1 = get_all_pieces(new_board_1, "b")
            best_value_1 = -inf
            for piece_1 in all_pieces_1:
                for move_1 in piece_1.move_list:
                    new_board_1 = copy.deepcopy(board)
                    new_board_1.simple_move((piece_1.row, piece_1.col), (move_1[1], move_1[0]), color)
                    best_value_1 = max(minimax(new_board_1, depth - 1, alpha, beta, not maximizing),
best_value_1)
                    alpha = max(alpha, best_value_1)
                    if beta <= alpha:
                        return best_value_1
            return best_value_1
        else:
            all_pieces_1 = get_all_pieces(new_board_1, "w")
            best_value_1 = inf
            for piece_1 in all_pieces_1:
                for move_1 in piece_1.move_list:
                    new_board_1 = copy.deepcopy(board)
                    new_board_1.simple_move((piece_1.row, piece_1.col), (move_1[1], move_1[0]), color)
                    best_value_1 = min(minimax(new_board_1, depth - 1, alpha, beta, not maximizing), best_value_1)
                    beta = min(beta, best_value_1)
                    if beta <= alpha:
                        return best_value_1
            return best_value_1

    was_checked = self.board.piece_is_checked(color)
    if was_checked:
        all_pieces = get_all_pieces(self.board, color)
        best_move = self.random_choice(color)
        best_value = -inf
        for piece in all_pieces:
            for move in piece.move_list:
                new_board = copy.deepcopy(self.board)
                new_board.simple_move((piece.row, piece.col), (move[1], move[0]), color)
                if not new_board.piece_is_checked(color):
                    if best_value == -inf:
                        best_move = (piece.row, piece.col), (move[0], move[1])
                        best_value = evaluate_board_advanced(new_board, color)
                    elif evaluate_board_advanced(new_board, color) > best_value:
                        best_move = (piece.row, piece.col), (move[0], move[1])
                        best_value = evaluate_board_advanced(new_board, color)
        return best_move
    else:
```

```
        best_move = root_minimax(self.board, 3, True)
        return best_move
```

Алгоритм для принятия решения 3го уровня сложности. Рекурсивный Минимакс с фактической глубиной 3 и Альфа-Бета-отсечением

```
def evaluate_board_advanced(board, color):
    pawn_list = [[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]]
    knight_list = [[-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]]
    bishop_list = [[-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]]
    rook_list = [[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]]
    queen_list = [[-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]]
    king_list = [[-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]]
    white_score = 0
    black_score = 0
    for row in range(0, 8):
        for col in range(0, 8):
            if board.board[row][col] != 0:
                if board.board[row][col].color == "b":
                    if board.board[row][col].__class__.__name__ == "Rook":
                        black_score += 500 + flipud(rook_list)[row][col]
                    elif board.board[row][col].__class__.__name__ == "Pawn":
                        black_score += 100 + flipud(pawn_list)[row][col]
                    elif board.board[row][col].__class__.__name__ == "Bishop":
                        black_score += 330 + flipud(bishop_list)[row][col]
                    elif board.board[row][col].__class__.__name__ == "Knight":
                        black_score += 320 + flipud(knight_list)[row][col]
                    elif board.board[row][col].__class__.__name__ == "Queen":
                        black_score += 900 + flipud(queen_list)[row][col]
                    elif board.board[row][col].__class__.__name__ == "King":
                        black_score += 20000 + flipud(king_list)[row][col]
                else:
                    if board.board[row][col].__class__.__name__ == "Rook":
                        white_score += 500 + rook_list[row][col]
                    elif board.board[row][col].__class__.__name__ == "Pawn":
                        white_score += 100 + pawn_list[row][col]
                    elif board.board[row][col].__class__.__name__ == "Bishop":
                        white_score += 330 + bishop_list[row][col]
                    elif board.board[row][col].__class__.__name__ == "Knight":
                        white_score += 320 + knight_list[row][col]
                    elif board.board[row][col].__class__.__name__ == "Queen":
                        white_score += 900 + queen_list[row][col]
                    elif board.board[row][col].__class__.__name__ == "King":
                        white_score += 20000 + king_list[row][col]
    if color == "w":
        return white_score - black_score
    else:
        return black_score - white_score
```

Функция для продвинутого “взвешивания” игровой доски

```
game_mode=0
difficulty=2
visual_set=0
time_restriction=15
```

Пример config.txt

Комментарии к коду

В flowingconfig.py происходит генерация конфигурации для каждого конкретного случая. Производиться попытка считать конфигурацию из файла config.txt. Тут же включается режим разработчика, с помощью

которого можно получать логи на все важные события в программе. Реализуется автоматическое масштабирование главного окна с помощью пакета screeninfo. Высота и ширина окна будет равна высоте экрана (дисплея) минус 90 пикселей, эти значения могут быть перегружены и установлены через config.txt. Здесь происходит проверка параметров на семантику, чтобы пользовательская конфигурация не сломала программу. Вычисляются константные значения.

В классе Board, содержаться главные методы манипуляции игровой ситуации. Главная матрица board класса Board имеет общее разрешение 8x8, представлены все 64 клетки шахматного поля. Элементы (клетки) этой матрицы равны либо, 0 - то-есть пустая клетка, либо там находится объект класса фигуры определенного типа. Метод set_start_normal - располагает фигуры для нормальной шахматной партии, метод update_moves - обновляет возможные ходы для каждой фигуры на доске, draw - рендерит каждую клетку доски, точнее её содержание, get_danger_moves - просчитывает клетки на которые проецируется вражеский шах, piece_at_the_end - если пешка достигла конца доски она меняется на ферзя, piece_is_checked - если король под шахом, select - выбрать клетку поля, сначала выбирается фигура, затем клетка для перемещения, reset_selected - сбросить выбранную клетку. Вариации move уже были расписаны.

В начале модуля Piece, происходит загрузка изображений для каждого из типов фигур, они масштабируются пропорционально размеру игрового поля. Абстрактный отцовский класс Piece содержит поля row, col - для отображения положения на доске, color - для цвета фигуры, selected - является ли данная фигура выбранной, move_list - список всех возможных ходов данной фигуры, king, pawn, queen - булевые “подписи” наследников. Присутствуют методы update_valid_moves - ключевой метод, который вычисляет все возможные ходы для фигуры, draw - для отрисовки фигуры. Каждый перегружающий класс наследник, использует изображения своей фигуры и переписывает update_valid_moves - относительно расположения фигуры и её типа.

Класс решения Solution из algorithm.py имеет 4 метода для вычисления хода. В основном алгоритм работает в данном порядке: получает все возможные ходы, оценивает каждый ход отдельно, возвращает свой выбранный ход обратно в метод game из game.py. Везде используются виртуальные board из Board.py. В любом случае в первую очередь проверяется шах, если он есть, тогда вся остальная часть алгоритма

игнорируется и ищется лучший ход с глубиной один для самого эффективного вывода короля из-под шаха. В остальных случаях алгоритм проходит по всему древу в поисках лучшего из всех ходов. Метод get_all_pieces - возвращает все возможные ходы всех доступных фигур.

Методы из evaluate.py используют константную матрицу и константные параметры. Они считывают матрицу доски и считают материал по заданным формулам. Для продвинутой оценки вражеской доски используется метод flipud из numpy, для переворачивания матрицы позиционных констант.

Конфигурация программы

Для конфигурации используйте config.txt

- game_mode=X (0 - PvP, 1 - PvE),
- difficulty=X (0, 1, 2, 3 - сложность),
- visual_set=X (имя подпапки — альтернативный визуальный набор),
- freeze_time=X (0..10 - задержка перед началом игры),
- time_restriction=X (0.5f..60000 - лимит времени)

2. Визуальный вид программы

Области окна программы

Разработанная программа делится на два условных окна, для распределения информационной нагрузки.

Первое окно представляет собой информационное сообщение, которое ознакомляет пользователя с управлением разработанным продуктом, в частности, игра в “Шахматы”. Само поле называется “Hotkeys”, то есть “Горячие клавиши”. Пользователю предоставляется возможность увидеть соответствие клавиш с и их функциональностью. В дальнейшем будем более подробно рассмотрен функционал этих клавиш.

Второе окно является основным окном в данном продукте. Это игровое поле для игры в “Шахматы”, где пользователю предоставлена возможность принять участие в партии с установленных ранее настройками. В дальнейшем рассмотрим данное поле более подробно.

Поле “Hotkeys”

Данное поле выполнено в следующем стиле, для более удобного восприятия информации. Слово “Hotkeys” выполнено в красном цвете, для подчеркивания краткой информации об окне. Сами же горячие клавиши сделаны в стиле “белый по черному”, что обеспечивает ясность и четкость текста. О функционале клавиш будет рассмотрено позднее. Для перехода к следующему полю требуется подождать некоторое время. Это время конфигурируется с помощью config.txt, параметр freeze_time, однако не рекомендовано это значение изменять. Три секунды не такое большое время, а вот посмотреть горячие клавиши во время игры уже нельзя. Размер окна автоматически распознается, но может быть перегружено через конфиг файл.

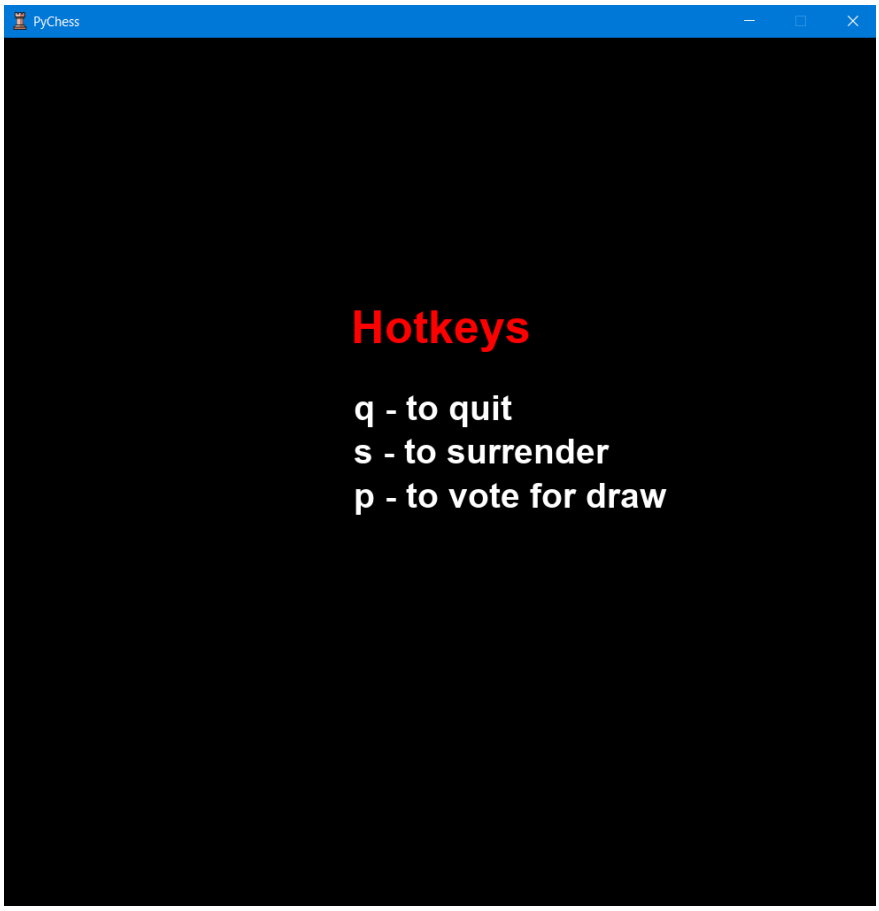

Рисунок.22 *Горячие клавиши* (в режиме против человека), против алгоритма возможность голосовать за ничью отсутствует.

Основное игровое поле “Шахматная доска”

Данное поле является основным для игры в “Шахматы”.

В левом верхнем углу и правом нижнем углу указано время, которое есть у игроков до поражения. Временные рамки соответствуют указанным значениям в config.txt, параметр time_restriction в минутах. По истечению времени объявляется победитель, в частности оппонент игрока, у которого истекло время.

Все игровое поле разделено на 64 квадратных полей, что являются местом для возможного размещения игровых фигур на поле игры. Поля по левому краю игрового поля подписано цифрами, а по верхнему краю подписано английскими буквами, все подписи выполнены в соответствии с общепринятыми правилами в игру “Шахматы”.

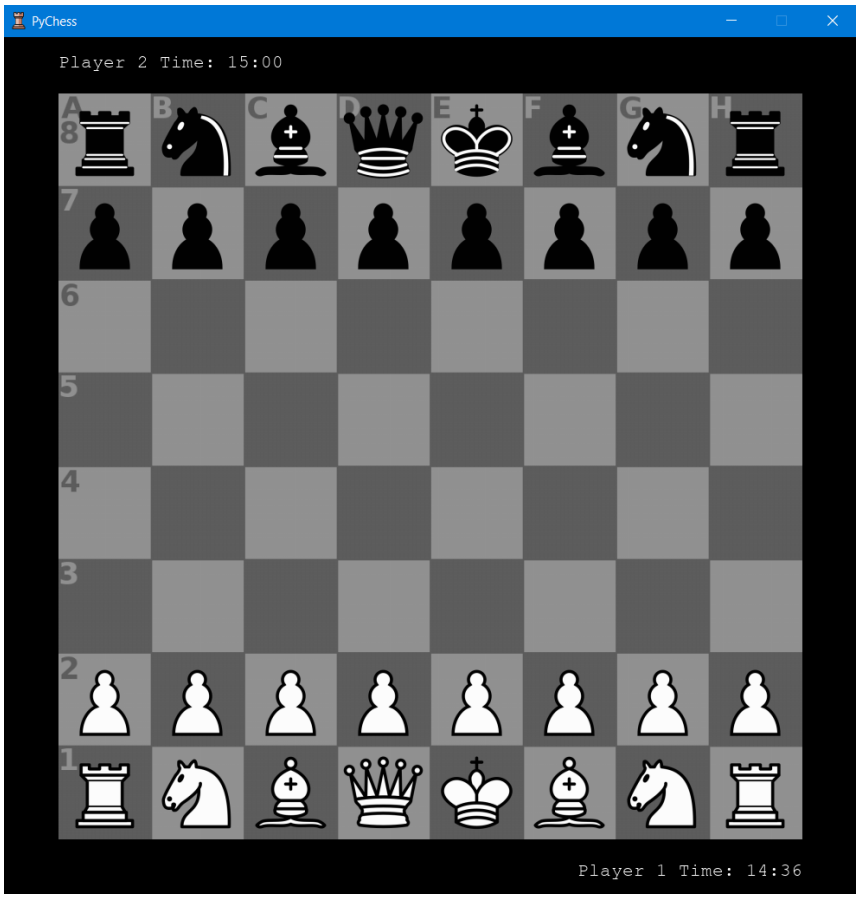

Рисунок.23 *Основное игровое поле* Основное игровое поле “Шахматная доска”.

Ход фигуры

В игре в “Шахматы”, когда пользователю требуется переместить фигуру на другую позицию, разработанный продукт дает возможность увидеть куда именно он может переместить фигуру. Также увеличивается размер фигуры, которой желает походить пользователь. Визуально возможные позиции, на которые может походить фигура, представлено в виде кругов на этих самых позициях.

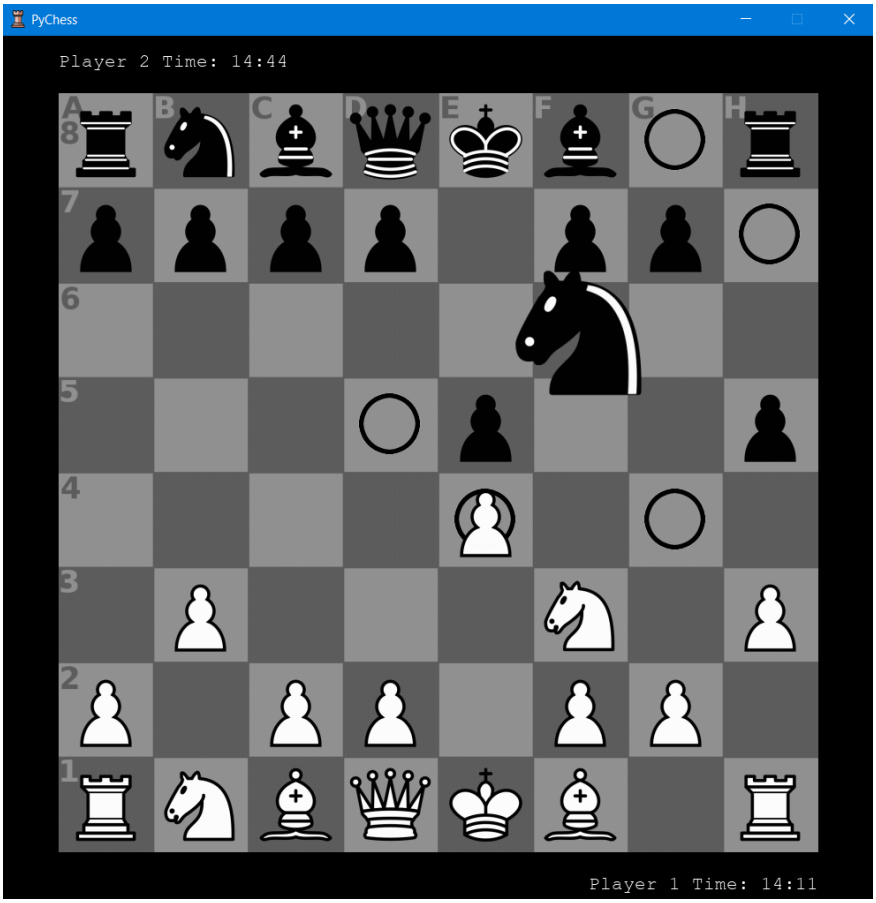

Рисунок.24 *Ход фигуры*

Игровая ситуация. Шах

Во время развития игровой партии, в шахматах может возникнуть ситуация именуемая “Шах”. На поле партии, данная ситуация отмечается в нижней части окна сообщением “White King is under check” либо “Black King is under check” в зависимости от того чей именно король попал в подобную ситуацию.

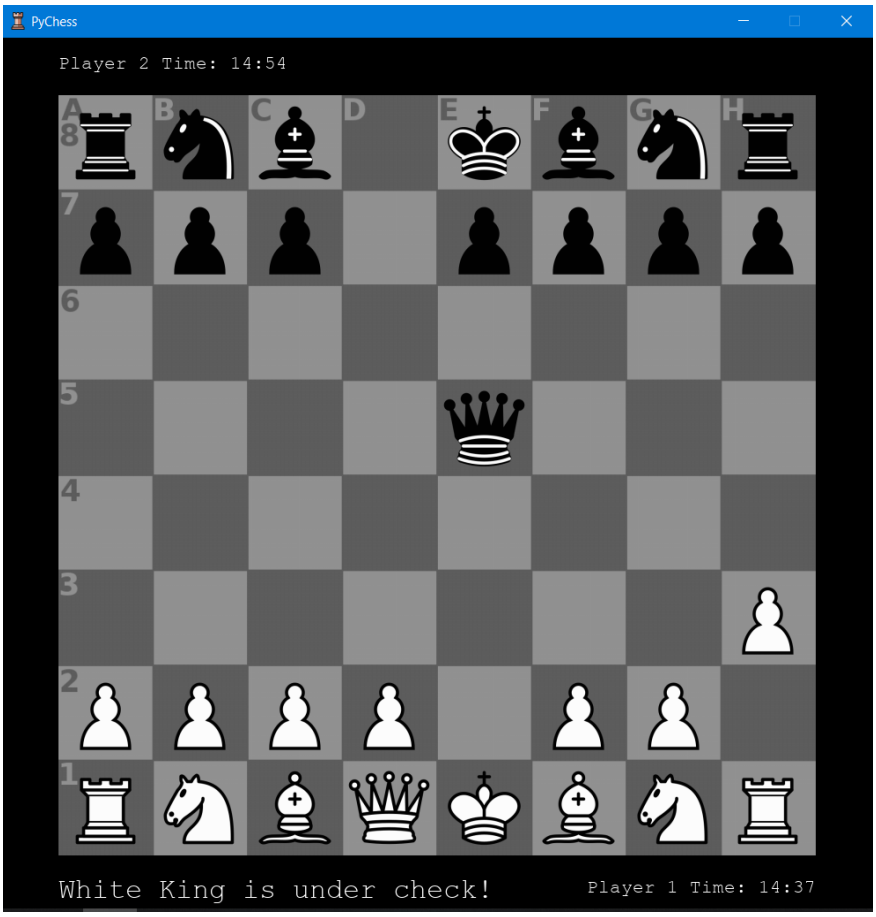

Рисунок.25 *Шах*

Игровая ситуация. Шах и Мат

Во время развития игровой партии, в шахматах может возникнуть ситуация именуемая “Шах и Мат”. На поле партии, данная ситуация отмечается по центру окна с оповещением о том какая именно сторона одержала победу.

Чуть ниже оглашенного победителя показывается время партии, затраченное на проведение игры. Также пользователю предлагается на выбор либо выйти из программы “q - to quit” и перезапустить игру “r - restart”.

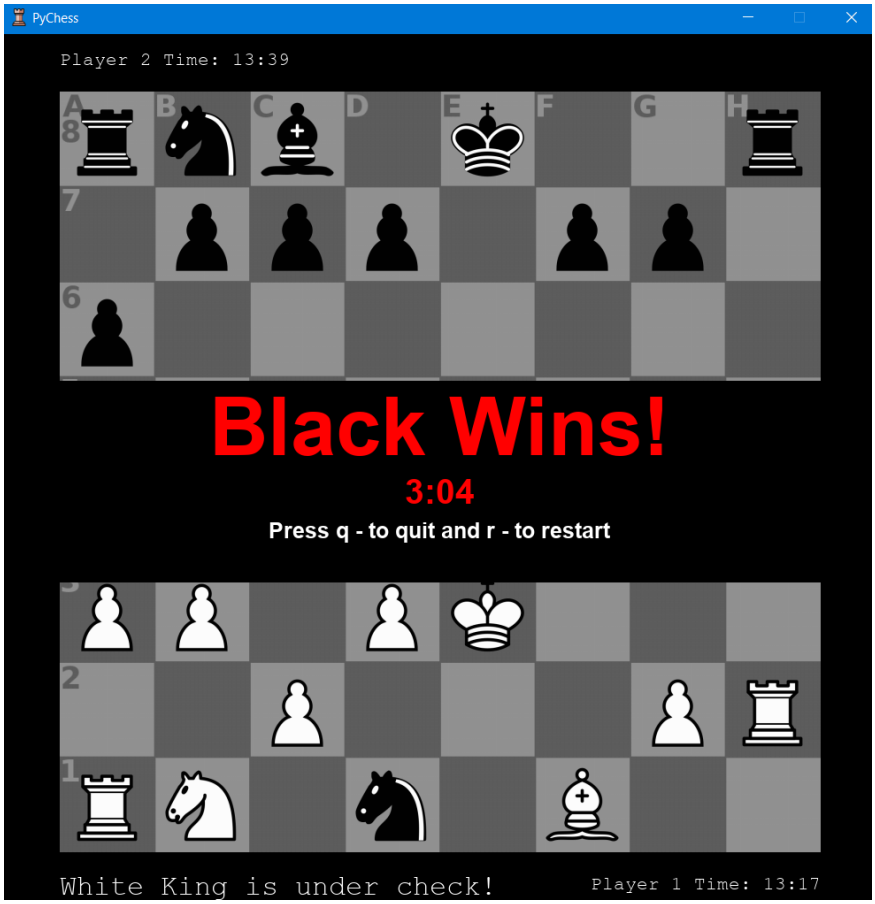

Рисунок.26 *Шах и Мат*

3. Функциональные особенности программы

Горячие клавиши

При пользовании разработанным продуктом, игрок может использовать упрощающие функции. Во время игры в “Шахматы” пользователь имеет доступ к следующим горячим клавишам:

- Клавиша “q” - выход из программы, пользователь может совершить досрочный выход из программы, тем самым завершая ее работу.
- Клавиша “s” - сдаться, пользователь может до конца времени сдаться. Закончить игру признав поражение. В таком случае выводится итоговый экран, будет предложено перезапустить партию, либо завершить работу программы.
- Клавиша “r” - перезапуск игровой партии, пользователь может перезапустить партию без перезапуска всей программы, все

шахматные фигуры возвращаются в исходную позицию, и игра начинается как новая.

- Клавиша “p” - голосование за ничью, пользователь может предложить оппоненту ничью, тем самым заканчивая партию ничьей.

Перемещение фигур

Во время игры в “Шахматы” соперникам каждый ход требуется перемещать фигуры, разработанный продукт дает пользователям такую возможность, игроку требуется нажимать, т. е. выбирать интересующую его фигуру и переместить на другую возможную позицию. Примечательно, что для данного взаимодействия с игрой было разработана вспомогательная для пользователя визуальное выделение позиций на которые может переместиться фигура. Фигура перемещается только по заданным для нее правилам и не может перемещаться на поля занимаемые союзными фигурами.

Также реализована функция “взятие” вражеских фигур, то-есть если фигура может “срубить” вражескую фигуру, она перемещается на её позицию, в то время как фигура, которую “срубили” исчезает.

4. Инструкция пользователю

Порядок взаимодействия с программой

Пользователю предлагается следующий алгоритм действий:

- Выставить все интересующие настройки в файле config.txt
- Запустить программу с помощью любой IDE, которая способна интерпретировать и запускать код на Python3
- С помощью мыши перемещать фигуры и сыграть партию в “Шахматы”, по заранее заданным настройкам
- После любого окончания партии выбрать, перезапуск программы для проведения еще одной партии, либо выйти из программы и завершить ее работу.

Глава III Заключение

Тестирование программы

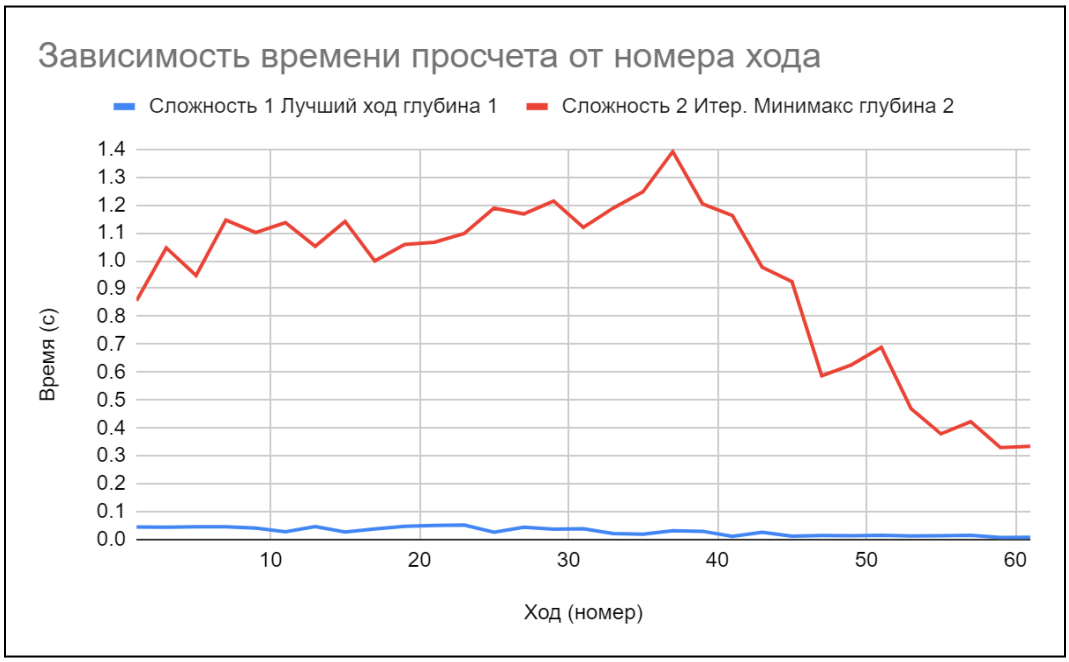

Рисунок.27 (сверху), 28 (снизу) *Зависимость времени от глубины хода*

На рисунках 27, 28 изображена зависимость времени от глубины хода. Время затраченное на расчет наилучшего хода напрямую зависит от количества возможных ходов. На рисунке 10 *Среднее количество доступных*

ходов можно заметить количество возможных ходов. При наложении можно заметить относительно прямую зависимость. Статистические, технические и стохастические погрешности вызваны локальной реализацией программы, особенностями игровой ситуации, уникальностью манеры игры каждого игрока при тестировании, техническими аппаратными обстоятельствами и неточностью мер измерения. Так же стоит учитывать, что ключевой параметр для уменьшения времени решения является уменьшения количества возможных ходов в игре. На примере, можно заметить что последние ходы не требуют времени, так как есть только один выгодный ход, которой ставит шах/мат королю.

Эффективность развертывания при этом составила более 80%. Что обозначает — большинство людей справилось с запуском программы на устройствах, следуя при этом инструкциям, указанным в теле отчета.

Были исправлены критические баги, найденные в ходе тестирования. Некритические ошибки найдены не были. Визуальные ошибки так же не были найдены. Вырезан функционал с программной настройкой конфигурации. При этом содержание игры было нарушено отсутствием рокировок, взятий в проходе и изменения пешки в конце доски на любое другую фигуру (кроме ферзя).

Шахматы имеют оптимизационные проблемы (проблемы производительности), что отчетливо видно при выборе 3 уровня сложности алгоритма, расчет хода может длиться до 80 секунд. Это последствие неправильной структуризации модели (model), и взаимодействия с ней. Построение программы и алгоритма вокруг другой модельной структуры потребовало много времени и полного изменения кода программы. Цель заключалась не в создании самого эффективного или производительного алгоритма для игры в шахматы.

Вылетов или неожиданных выходов из программы обнаружено не было, а логирование при этом помогает исправлять большинство ошибок.

При сравнении разработанного алгоритма с уже существующими, заметна его недостаточная оптимизация, простота и не очень высокая сложность.

Претензий к визуальной, навигационной части нет; к производительности, содержанию, сложности алгоритма претензии присутствуют.

Доказательство теории

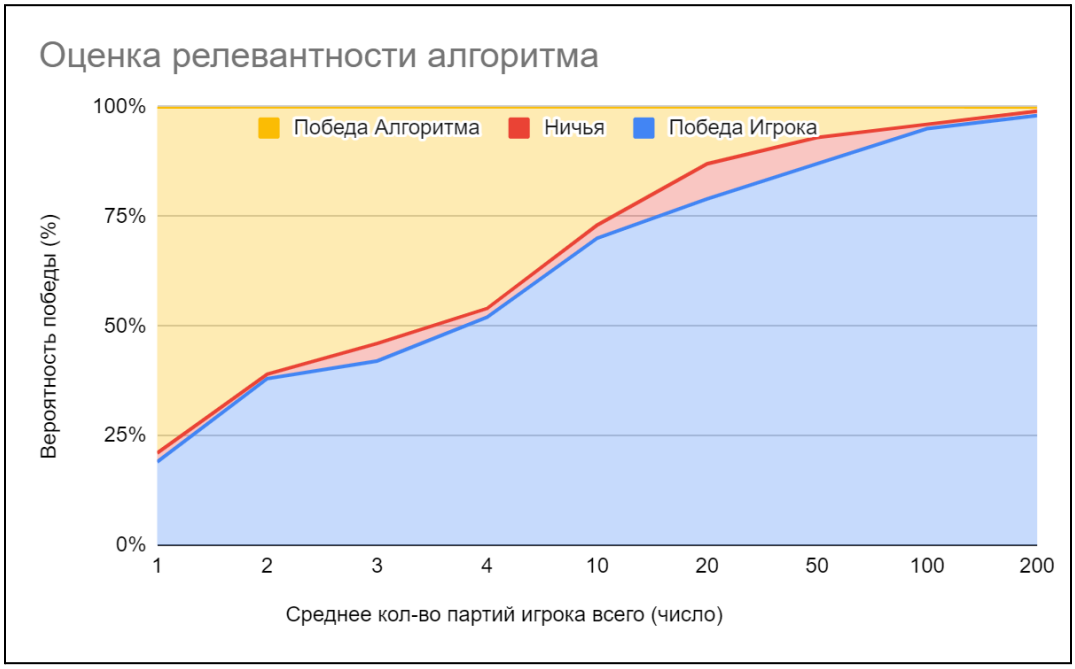

Рисунок. 29 *Зависимость уровня игрока от вероятности победы*

На рисунке изображена вероятность победы алгоритма против игроков разного уровня. Можно заметить, что игроки, у которых эта партия первая (судя по их заявлениям), имеют очень маленькие шансы для победы в шахматы против написанного алгоритма. В то же время, игроки, имеющие 10 и более шахматных партий за спиной имеют очень хорошие шансы на победу. Это связано в первую очередь, очевидно, с опытом игры именно в шахматы, и во вторую с пониманием принципа работы алгоритма. Всего опрошено 41 человек (студенты разных направлений 19-23 лет, взрослые люди разных профессий 30-50 лет) с разными уровнями и манерой игры.

В целом данные показатели на базовом уровне доказывают релевантность алгоритма и эффективность/выгодность найденных им решений. Целью проекта являлось доказательство применимости Минимаксного метода с Альфа-Бета-отсчением в игровых моделях. И в частности эта цель была достигнута, путем эксперимента и анализа.

Достаточные условия выполнены: поиск самого выгодного решения из всех данных, необходимые условия выполнены: поиск решения по формализованной доске. Это доказывает применимость, так как играть против такого алгоритма возможно, и в редких случаях трудно. На примере “крестиков-ноликов” объяснено почему шахматы — не единственная игровая система в которой можно реализовывать Минимаксный метод. Это доказывает применимость вне шахматной модели. Что отдельно доказывает,

что применимость в шахматной модели не вызвана неправильной формализацией и алгоритмизацией самой модели.

Исходя из представленной информации и указанных доказательств можно заявить, что для реализации Минимаксного метода в какой-либо системе нужен способ оценки ситуаций в данной системе. Формализация модели, скорее всего, необходима; обязательна в случае взаимодействия с аналоговыми данными. После создания математической модели, превращения модели в систему, и внедрения в нее алгоритма, Минимаксный метод может быть реализован и может предоставлять решения для системы, удовлетворяющее математическим требованиям, заданным этой самой системе.

В таком случае можно заявить, что любая модель, которая может быть формализована, или просто переведена в систему (если модель уже математическая), такая модель и впоследствии система может быть решена Минимаксным методом. Что в сухом остатке, означает, что гипотеза о применимости в игровых формализованных моделях доказана. А гипотеза о применимости в любых формализованных моделях может быть выдвинута.

Облачное хранение

https://github.com/zeightOFFICIAL/python-chess-minimax

Выводы

В результате работы был разработан программный продукт, в частности игра в “Шахматы”. Для изучения исследуемой темы в разработанной игровой среде был реализован алгоритм Минимакс в виде бота который может принять участие против игрока. В ходе исследуемой работы была выявлено, что метод принятия решений является полноценным и релевантным для использования в игровых моделях и имеет вполне достойные показатели в реальных прикладных и бытовых задачах. В практическом применении было выявлена релевантность данного метода. Следовательно, считаем, что поставленная цель достигнута успешно.

Поэтапно выполнена каждая задача данной научно-исследовательской работы. Приведены аргументы, или как минимум объяснена позиция относительно каждой задачи. Приведены визуальные доказательства и объяснения. Гипотезы о применимости Минимаксного метода в шахматах, игровых системах доказаны. Гипотеза о применимости Минимаксного метода в любых формализованных моделях выдвинута.

Источники и литература

- The Chess Programming Wiki is a repository of information about programming computers to play chess. - Репозиторий технической и математической информации для создания программируемых систем для шахмат [Электронный ресурс]. URL: www.chessprogramming.org (дата обращения: 13.07.2022)
- fokus-lop Создаем несложный шахматный ИИ: 5 простых этапов [Электронный ресурс]. URL: https://habr.com/ru/company/skillbox/blog/437524/ (дата обращения: 20.07.2022)
- На пути к Deep Blue: пошаговое руководство по созданию простого ИИ для игры в шахматы [Электронный ресурс]. URL: https://tproger.ru/translations/simple-chess-ai-step-by-step/ (дата обращения: 25.06.2022)
- Encyclopedia of Mathematics Minimax principle - Принцип Минимакс [Электронный ресурс]. URL:https://encyclopediaofmath.org/index.php?title=Minimax_principle (дата обращения: 29.06.2022)
- Wayback Machine Game Visualization – Визуализация Минимакс алгоритма [Электронный ресурс]. URL:https://web.archive.org/web/20150324045417/http://ksquared.de/gamevisual/launch.php (дата обращения: 30.06.2022)
- AI Impacts Historic trends in chess AI [Электронный ресурс]. URL:https://aiimpacts.org/historic-trends-in-chess-ai/ (дата обращения: 22.01.2023)
- Allen Newel, Herbert A. Simon Completer Science as Empirical Inquiry: Symbols and Search // ACM Turing Award lecture. - 1975 - 19 - С. 3.
- April Walker The Anatomy of a Chess AI [Электронный ресурс]. URL:https://medium.com/@SereneBiologist/the-anatomy-of-a-chess-ai-2087d0d565 (дата обращения: 19.01.2023)
- JavaTPoint: Mini-Max Algorithm in Artificial Intelligence [Электронный ресурс]. URL:https://www.javatpoint.com/mini-max-algorithm-in-ai (дата обращения: 28.01.2023)
- Geeks for Geeks: Minimax Algorithm in Game Theory [Электронный ресурс]. URL:https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ (дата обращения: 13.12.2022)