|  |  |
| --- | --- |
| Gerb-BMSTU_01 | Министерство науки и высшего образования Российской Федерации  Федеральное государственное бюджетное образовательное учреждение  высшего образования  «Московский государственный технический университет  имени Н.Э. Баумана  (национальный исследовательский университет)»  (МГТУ им. Н.Э. Баумана) |
| ФАКУЛЬТЕТ «Робототехники и комплексной автоматизации»  КАФЕДРА «Компьютерные системы автоматизации производства»  РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА  К КУРСОВОМУ ПРОЕКТУ  НА ТЕМУ:  «Исследование алгоритма минимакс с альфа-бета отсечением на примере игры крестики-нолики»  Студент РК9-12М Р.Р. Хикматуллин  (Подпись, дата)  Руководитель курсового проекта М. А. Блохин  (Подпись, дата) | |
| 2023 г. | |

**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное бюджетное образовательное учреждение**

**высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет)»**

**(МГТУ им. Н.Э. Баумана)**

УТВЕРЖДАЮ

Заведующий кафедрой РК9

С.С. Гаврюшин

« \_\_\_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 20 \_\_\_\_ г.

**ЗАДАНИЕ**

**на выполнение курсового проекта**

по дисциплине «Интегрированные САПР (CAD/CAM/CAE)»

Студент группы РК9-12М

Хикматуллин Рамиль Рустамович**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

(Фамилия, имя, отчество)

Тема курсовой работы: «Исследование алгоритма минимакс с альфа-бета отсечением на примере игры крестики-нолики»

Направленность КП (учебная, исследовательская, практическая, производственная, др.):

исследовательская**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

Источник тематики (кафедра, предприятие, НИР): \_\_\_\_\_\_\_\_\_\_\_НИР\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

График выполнения работы: 25% к \_\_\_ нед., 50% к \_\_\_ нед., 75% к \_\_ нед., 100% к \_\_\_ нед.

***Задание*** провести предпроектное исследование, исследовать и реализовать алгоритм поиска оптимального хода при помощи метода минимакс с различного рода оптимизациями на примере игры крестики-нолики в различных модификациях. Проанализировать эффективность модификации и выбрать наиболее оптимальный.

***Оформление курсового проекта:***

Расчетно-пояснительная записка на 53 листах формата А4. Перечень графического материала TODO слайдов PowerPoint.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата выдачи задания « \_\_\_ » \_\_\_\_\_\_\_\_\_\_\_\_ 2022 г.

**Руководитель курсового проекта**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ М.А. Блохин

(Подпись, дата) (И.О.Фамилия)

**Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** Р.Р. Хикматуллин

(Подпись, дата) (И.О.Фамилия)

# РЕФЕРАТ

Расчетно-пояснительная записка 52 с., 6 рис., 10 источников, 1 приложение.

АЛГОРИТМ МИНИМАКС, АЛЬФА-БЕТА ОТСЧЕЧЕНИЕ, ТЕОРИЯ ИГР.

Объектом исследования является алгоритм минимакс, его оптимизация с помощью альфа-бета отсечения, динамического программирования, и его применения в различных видах игры крестики-нолики.

Цель работы – исследовать и реализовать алгоритм поиска наилучшего хода при помощи метода минимакс с различными видами его оптимизации на примере разного вида игры крестики-нолики.

**СОДЕРЖАНИЕ**

[РЕФЕРАТ 2](#_Toc125099139)

[1 Предпроектное исследование 6](#_Toc125099140)

[1.1 Объект исследования 6](#_Toc125099141)

[1.2 Постановка и актуальность задачи 7](#_Toc125099142)

[1.3 Теория игр 7](#_Toc125099143)

[1.3.1 Краткая история развития теории игр 8](#_Toc125099144)

[1.3.2 Наглядный пример применения теории игр 9](#_Toc125099145)

[1.3.3 Типы игр 10](#_Toc125099146)

[1.3.4 Проблемы практического применения 11](#_Toc125099147)

[1.5 Алгоритм минимакс 12](#_Toc125099148)

[1.5.1 Альфа-бета отсечение 14](#_Toc125099149)

[2 Программная реализация метода минимакс для игры с нулевой суммой 19](#_Toc125099150)

[2.1 Разработка логической архитектуры программы 19](#_Toc125099151)

[2.2 Реализация программного кода 22](#_Toc125099152)

[2.2.1 Язык программирования 22](#_Toc125099153)

[2.2.2 Реализация вспомогательных функции 22](#_Toc125099154)

[2.2.3 Реализация основных функции 24](#_Toc125099155)

[3 Оптимизация алгоритма минимакс 30](#_Toc125099156)

[3.1 Игра 3х3 30](#_Toc125099157)

[3.2 Игра 3х4 32](#_Toc125099158)

[3.3 Игра 4х4 ( 4 в ряд) 33](#_Toc125099159)

[3.4 Игра 4х4 (3 в ряд) 33](#_Toc125099160)

[ЗАКЛЮЧЕНИЕ 34](#_Toc125099161)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 35](#_Toc125099162)

[Приложение А 37](#_Toc125099163)

ВВЕДЕНИЕ

С детства мне интересна тема настольных игр и правил в них. Ведь каждая настольная игра, пережившая десятилетия, является своего рода камнем, который стал настолько гладким под струёй воды, что не имеет изъянов. Говоря иначе, игра менялась, менялись правила со временем, и в конце концов она стала равновесной. Это значит, что в этих играх нет универсальной стратегии, которая всегда выигрывает.

Крестики-нолики – не самый лучший пример развития игры со временем, здесь больше подойдут шахматы. Но крестики-нолики наиболее простой пример игры для понимания алгоритма минимакс в деле.

Крестики-нолики— логическая игра между двумя противниками на квадратном поле 3 на 3 клетки или большего размера (вплоть до «бесконечного поля»). Один из игроков играет «крестиками», второй — «ноликами». В традиционной китайской игре Гомоку[1] используются чёрные и белые камни.

Объектом исследования является алгоритм минимакс, его оптимизация с помощью альфа-бета отсечения, динамического программирования, и его применения в различных видах игры крестики-нолики.

Цель работы – исследовать и реализовать алгоритм поиска наилучшего хода при помощи метода минимакс с различными видами его оптимизации на примере разного вида игры крестики-нолики.

Для достижения цели необходимо реализовать модель игры крестики-нолики в различных модификациях, также создать искусственного оппонента на основе алгоритма поиска наилучшего решения – минимакс. И в результате сравнить эффективность оптимизационных методов для алгоритма.

# 1 Предпроектное исследование

## Объект исследования

Крестики-нолики— логическая игра между двумя противниками на квадратном поле 3 на 3 клетки или большего размера (вплоть до «бесконечного поля»). Один из игроков играет «крестиками», второй — «ноликами». В традиционной китайской игре Гомоку используются чёрные и белые камни.

Правила игры знакомы любому:

Игроки по очереди ставят на свободные клетки поля 3×3 знаки (один всегда крестики, другой всегда нолики). Первый, выстроивший в ряд 3 своих фигуры по вертикали, горизонтали или диагонали, выигрывает. Первый ход делает игрок, ставящий крестики.

Обычно по завершении партии выигравшая сторона зачёркивает чертой свои три знака (нолика или крестика), составляющих сплошной ряд.

Минимакс - это своего рода алгоритм обратного[2] отслеживания, который используется в процессе принятия решений и теории игр, чтобы найти оптимальный ход для игрока, предполагая, что ваш оппонент также играет оптимально. Он широко используется в пошаговых играх для двух игроков, таких как крестики-нолики, нарды, шахматы и т. Д.  
В минимаксе два игрока называются максимизатором и минимизатор. Максимизатором пытается получить максимально возможный балл, в то время как минимизатор пытается сделать наоборот и получить минимально возможный балл.  
С каждым состоянием доски связано значение. В данном состоянии, если максимизатор имеет преимущество, оценка доски будет иметь тенденцию к некоторому положительному значению. Если минимайзер имеет преимущество в этом состоянии платы, то он будет иметь тенденцию к некоторому отрицательному значению. Значения состоянии вычисляются с помощью некоторых эвристик, которые уникальны для каждого типа игры.

## Постановка и актуальность задачи

На первый взгляд кажется, что крестики-нолики совсем уж игра простая и зачем её исследовать в современном мире, где есть множество суперкомпьютеров и даже десяток квантовых. Мой же подход именно в прикладном понимании алгоритма минимакс с его различными оптимизациями, так как основа этого алгоритма окружает нас со всех сторон. От навигатора в машине до робот-пылесоса дома. Поиск оптимального пути. Да, минимакс- это не совсем то, что находит нам оптимальный путь в навигаторе, там скорее всего используется более сложный алгоритм, но минимакс один из базовых видов поиска решении на графе. Также алгоритм минимакс и его разновидности дают основу многим моделям искусственного интеллекта, который используется в практически в каждой единице современной техники.

## Теория игр

**Теория игр** — это раздел математической экономики, изучающий решение конфликтов между игроками и оптимальность их стратегий. Конфликт может относиться к разным областям человеческого интереса: чаще всего это экономика, социология, политология, реже биология, кибернетика и даже военное дело. Конфликтом является любая ситуация, в которой затронуты интересу двух и более участников, традиционно называемых игроками. Для каждого игрока существует определенный набор стратегий, которые он может применить. Пересекаясь, стратегии нескольких игроков создают определенную ситуацию, в которой каждый игрок получает определенный результат, называемый выигрышем, положительным или отрицательным. При выборе стратегии важно учитывать не только получение максимального профита для себя, но также возможные шаги противника, и их влияние на ситуацию в целом.

### 1.3.1 Краткая история развития теории игр

Основы теории игр зародились еще в 18 веке, с началом эпохи просвещения и развитием экономической теории. Впервые математические аспекты и приложения теории были изложены в классической книге 1944 года Джона фон Неймана и Оскара Моргенштерна «Теория игр и экономическое поведение». Первые концепции теории игр анализировали антагонистические игры, когда есть проигравшие и выигравшие за их счет игроки. Несмотря на то, что теория игр рассматривала экономические модели, вплоть до 50-х годов 20 века она была всего лишь математической теорией. После, в результате резкого скачка экономики США после второй мировой войны, и, как следствие, большего финансирования науки, начинаются попытки практического применения теории игр в экономике, биологии, кибернетике, технике, антропологии. Во время Второй мировой войны и сразу после нее теорией игр серьезно заинтересовались военные, которые увидели в ней мощный аппарат для исследования стратегических решений. В начале 50-х Джон Нэш[3] разрабатывает методы анализа, в которых все участники или выигрывают, или терпят поражение. Эти ситуации получили названия «равновесие по Нэшу»[4]. По его теории, стороны должны использовать оптимальную стратегию, что приводит к созданию устойчивого равновесия. Игрокам выгодно сохранять это равновесие, так как любое изменение ухудшит их положение. Эти работы Нэша сделали серьезный вклад в развитие теории игр, были пересмотрены математические инструменты экономического моделирования. Джон Нэш показывает, что классический подход к конкуренции А.Смита, когда каждый сам за себя, неоптимален. Более оптимальны стратегии, когда каждый старается сделать лучше для себя, делая лучше для других. За последние 20 — 30 лет значение теории игр и интерес значительно растет, некоторые направления современной экономической теории невозможно изложить без применения теории игр.Большим вкладом в применение теории игр стала работа Томаса Шеллинга[5], нобелевского лауреата по экономике 2005 г. «Стратегия конфликта».

### 1.3.2 Наглядный пример применения теории игр

Табл.1 Дилемма заключённого[6]

Смысл теории игр проще всего пояснить на «Дилемме заключенного», классическая формулировка которой звучит так:

«Двое преступников, А и Б, попались примерно в одно и то же время на сходных преступлениях. Есть основания полагать, что они действовали по сговору, и полиция, изолировав их друг от друга, предлагает им одну и ту же сделку: если один свидетельствует против другого, а тот хранит молчание, то первый освобождается за помощь следствию, а второй получает максимальный срок лишения свободы (10 лет). Если оба молчат, их деяние проходит по более лёгкой статье, и они приговариваются к 6 месяцам. Если оба свидетельствуют против друг друга, они получают минимальный срок (по 2 года). Каждый заключённый выбирает, молчать или свидетельствовать против другого. Однако ни один из них не знает точно, что сделает другой. Что произойдёт?»

А теперь представим развитие ситуации, поставив себя на место заключенного А. Если мой подельник молчит, лучше его сдать и выйти на свободу. Если он говорит, то так же лучше все рассказать, и получить всего два года, вместо десяти. Таким образом, если каждый игрок выбирает, что лучше для него, оба сдадут друг друга, и получат два года, что не является идеальной ситуацией для обоих. Если бы каждый думал об общем благе, они бы получили всего по полгода.

### 1.3.3 Типы игр

Кооперативная\некооперативная игра

Кооперативной игрой является конфликт, в котором игроки могут общаться между собой и объединяться в группы для достижения наилучшего результата. Примером кооперативной игры можно считать карточную игру Бридж, где очки каждого игрока считаются индивидуально, но выигрывает пара, набравшая наибольшую сумму. Из двух типов игр, некооперативные описывают ситуации в мельчайших деталях и выдают более точные результаты. Кооперативные рассматривают процесс игры в целом. Несмотря на то, что эти два вида противоположны друг другу, вполне возможно объединение стратегий, которое может принести больше пользы, чем следование какой-либо одной.

Более кратко получается так:

Кооперативные или коалиционные игры в теории игр — игры, в которых игроки могут объединяться в группы, взяв на себя некоторые обязательства перед другими игроками и координируя свои действия.  
Некооперативные игры — игры, в которых каждый игрок должен играть за себя.

С нулевой суммой и с ненулевой суммой

Игрой с нулевой суммой называют игру, в которой выигрыш одного игрока равняется проигрышу другого. Например, банальный спор: если вы выиграли сумму N, то кто-то эту же сумму N проиграл. В игре же с ненулевой суммой может изменяться общая цена игры, таким образом принося выгоду одному игроку, не отнимаю ее цену у другого. В качестве примера здесь отлично подойдут шахматы: превращая пешку в ферзя игрок А увеличивает общую сумму своих фигур, при этом не отнимая ничего у игрока Б. В играх с ненулевой суммой проигрыш одного из игроков не является обязательным условием, хотя такой исход и не исключается.

Параллельные и последовательные

Параллельной является игра, в которой игроки делают ходы одновременно, либо ход одного игрока неизвестен другому, пока не завершится общий цикл. В последовательной игре каждый игрок владеет информацией о предыдущем ходе своего оппонента до того, как сделать свой выбор. И совсем не обязательно информации быть полной, что подводит нас к следующему типу.

### 1.3.4 Проблемы практического применения

Безусловно, следует указать и на наличие определенных границ применения аналитического инструментария теории игр. В следующих случаях он может быть использован лишь при условии получения дополнительной информации.  
  
Во-первых, это тот случай, когда у игроков сложились разные представления об игре, в которой они участвуют, или когда они недостаточно информированы о возможностях друг друга. Например, может иметь место неясная информация о платежах конкурента (структуре издержек). Если неполнотой характеризуется не слишком сложная информация, то можно применять опыт подобных случаев с учетом определенных различий.  
  
Во-вторых, теорию игр трудно применять при множестве ситуаций равновесия. Эта проблема может возникнуть даже в ходе простых игр с одновременным выбором стратегических решений.  
  
В-третьих, если ситуация принятия стратегических решений очень сложна, то игроки часто не могут выбрать лучшие для себя варианты. Например, на рынок в разные сроки могут вступить несколько предприятий или реакция уже действующих там предприятий может оказаться более сложной, нежели быть агрессивной или дружественной.  
  
Экспериментально доказано, что при расширении игры до десяти и более этапов игроки уже не в состоянии пользоваться соответствующими алгоритмами и продолжать игру с равновесными стратегиями.  
  
К сожалению, ситуации реального мира зачастую очень сложны и настолько быстро изменяются, что невозможно точно спрогнозировать, как отреагируют конкуренты на изменение тактики. Тем не менее, теория игр полезна, когда требуется определить наиболее важные и требующие учета факторы в ситуации принятия решений в условиях конкурентной борьбы. Эта информация важна, поскольку позволяет учесть дополнительные переменные или факторы, имеющие возможность повлиять на ситуацию, и тем самым повысить эффективность решения.

## 1.5 Алгоритм минимакс

Один из алгоритмов поиска решения на игровых деревьях известен как метод Минимакс[7] (MiniMax), предложенный еще в 1945 г. О. Моргенштер- ном (О. Morgenstern) и Дж. фон Нейманом (J. von Neumann).

Наиболее известен как праотец современной архитектуры компьютеров (архитектура фон Неймана), а также как создатель теории игр и концепции клеточных автоматов.

«Когда математическая дисциплина отходит достаточно далеко от своего эмпирического источника, она все более и более превращается в бесцельное упражнение по эстетике, в искусство ради искусства. При наступлении этого этапа единственный способ исцеления, на мой взгляд, состоит в том, чтобы возвратиться к источнику и впрыснуть более или менее прямо эмпирические идеи. Я убежден, что это всегда было необходимо для того, чтобы сохранить свежесть и жизненность математической теории, и что это положение останется в силе и в будущем».

Если бы у нас было полное дерево игры, мы бы начали снизу, с терминальных вершин, оценка которых задана правилами игры (функция F для терминальных позиций), и все было бы достаточно просто — для хода игрока Мах мы выбираем дочернюю вершину с максимальным значением оценочной функции и переносим эту оценку вверх, а для хода Min — с минимальным. Если в результате работы алгоритма начальная вершина имела бы оценку +ос, то победил Мах, если -ос, то Min. Таким образом, игра была бы решена.

Однако полное дерево игры обычно слишком велико. Например, для шашек это 1040 позиций, для шахмат — 10120 (см. п. 1.1.5). Для практически интересных игр полное дерево игры построить физически невозможно. Но можно эту идею (начать снизу) применить и к частичному дереву. Для этого нужно построить частичное дерево, оценить узлы на нижнем ярусе с помощью оценочной функции и вернуться по дереву назад, т.е. вверх.

Метод Минимакс заключается в следующем. Пусть есть некоторая статическая оценочная функция позиции F, причем если позиция считается выгодной для Мах, то F > 0, если позиция считается выгодной для Min, то F < 0. Если бы Мах выбирал среди концевых вершин, то он выбирал бы позицию с наибольшей оценкой, а если выбирал бы Min, то — с наименьшей оценкой. Также и в методе Минимакс. В построенном частичном дереве игры (если полное дерево построить невозможно) листовые позиции оцениваются придуманной оценочной функцией (поскольку правила игры нe предусматривают оценку промежуточных позиций), а далее вверх по дереву распространяются оценки в точности так, как это делается при статической оценке полного дерева. Если оценочная функция придумана удачно, а дерево построено достаточного глубоко, то метод Минимакса часто дает действительно оптимальное решение, или решение, близкое к оптимальному.

### 1.5.1 Альфа-бета отсечение

Альфа-бета отсечение[8] на самом деле не является новым алгоритмом, а скорее методом оптимизации минимаксного алгоритма. Это значительно сокращает время вычислений. Это позволяет нам выполнять поиск намного быстрее и даже углубляться в уровни игрового дерева. Он отсекает ветви в игровом дереве, которые не нужно искать, потому что уже существует лучший доступный ход. Он называется альфа-бета-обрезкой, потому что он передает 2 дополнительных параметра в минимаксной функции, а именно альфа и бета.

Определим параметры альфа и бета.

**Альфа**- это наилучшее значение, которое максимизатор в настоящее время может гарантировать на этом уровне или выше.  
**Бета** - это наилучшее значение, которое минимайзер в настоящее время может гарантировать на этом уровне или ниже.

В листинге 1 ниже приведён псевдокод для реализации альфа-бета отсечения для алгоритма минимакс

function minimax(node, depth, isMaximizingPlayer, alpha, beta):

**if** node **is** a leaf node :

**return** value of the node

**if** isMaximizingPlayer :

bestVal = -INFINITY

**for** each child node :

value = minimax(node, depth+**1**, false, alpha, beta)

bestVal = max( bestVal, value)

alpha = max( alpha, bestVal)

**if** beta <= alpha:

**break**

**return** bestVal

**else** :

bestVal = +INFINITY

**for** each child node :

value = minimax(node, depth+**1**, true, alpha, beta)

bestVal = min( bestVal, value)

beta = min( beta, bestVal)

**if** beta <= alpha:

**break**

**return** bestVal

// Calling the function **for** the first time.

minimax(**0**, **0**, true, -INFINITY, +INFINITY)

Листинг 1– Псевдокод альфа-бета отсечения

Пройдёмся итеративно на примере с альфа-бета отсечением для понимания в соответствие с рисунком 1.

* Начальный вызов начинается с **A.** Значение альфа здесь равно **-БЕСКОНЕЧНОСТИ**, а значение бета равно **+ БЕСКОНЕЧНОСТИ**. Эти значения передаются последующим узлам дерева. В **A** максимизатор должен выбрать максимальное из **B** и **C**, поэтому **A** сначала вызывает **B**

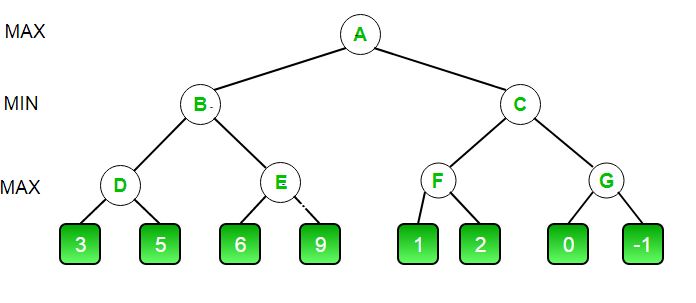


Рисунок 1– Дерево решении для антагонистической игры (начало)

* В **B** он минимизатор должен выбрать min из **D** и **E** и, следовательно, сначала вызывает **D.**
* В **D** он смотрит на своего левого дочернего элемента, который является конечным узлом. Этот узел возвращает значение 3. Теперь значение alpha в **D** равно max( -INF, 3), что равно 3.
* Чтобы решить, стоит ли смотреть на его правильный узел или нет, он проверяет условие beta<=alpha . Это неверно, поскольку бета = + INF и альфа = 3. Таким образом, поиск продолжается.
* **Теперь D** смотрит на своего правого дочернего элемента, который возвращает значение 5.At **D**, alpha = max(3, 5), что равно 5. Теперь значение узла **D** равно 5
* **D** возвращает значение от 5 до **B.** В точке **B** бета = min(+ INF, 5), что равно 5. Теперь минимизатору гарантировано значение 5 или меньше. Теперь **B** вызывает **E**, чтобы узнать, может ли он получить меньшее значение, чем 5.
* В **E** значения alpha и beta не равны -INF и +INF, а вместо этого -INF и 5 соответственно, потому что значение beta было изменено в **B**, и это то, что **B** перешло в **E**
* Теперь **E** смотрит на своего левого дочернего элемента, который равен 6. При **E** alpha = max(-INF, 6), что равно 6. Здесь условие становится истинным. бета равна 5, а альфа равна 6. Таким образом, бета <= альфа истинно. Следовательно, он прерывается, и **E** возвращает 6 в **B**
* Обратите внимание, что не имело значения, каково значение правильного дочернего элемента **E.** Это могло быть +INF или -INF, это все равно не имело бы значения, нам даже не приходилось смотреть на это, потому что минимизатору было гарантировано значение 5 или меньше. Итак, как только максимизатор увидел 6, он понял, что минимизатор никогда не пойдет этим путем, потому что он может получить 5 с левой стороны **B.** Таким образом, нам не нужно было смотреть на это 9 и, следовательно, экономить время вычислений.
* **E** возвращает значение от 6 до **B.** В точке **B** бета = min(5, 6), что равно 5. Значение узла **B** также равно 5

Пока так выглядит наше игровое дерево (см. Рис.2). 9 вычеркнуто, потому что оно никогда не вычислялось.

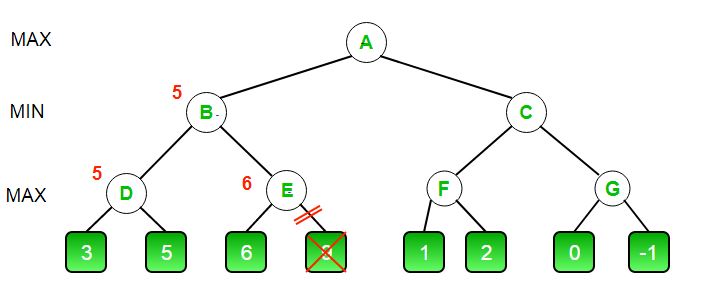


Рисунок 2 – Дерево решении (продолжение)

* **B** возвращает 5 в **A.** При **A** alpha = max( -INF, 5), что равно 5. Теперь максимизатору гарантировано значение 5 или больше. Теперь **A** вызывает **C**, чтобы узнать, может ли он получить более высокое значение, чем 5.
* В точке **C** альфа = 5 и бета = +INF. **C** вызывает **F**
* При **F** альфа = 5 и бета = +INF. **F** смотрит на своего левого дочернего элемента, который равен 1. alpha = max(5, 1), что по-прежнему равно 5.
* **F** смотрит на своего правого дочернего элемента, который равен 2. Следовательно, наилучшее значение этого узла равно 2. Альфа по-прежнему остается 5
* **F** возвращает значение от 2 до **C.** При **C** бета = min(+ INF, 2). Условие beta <= alpha становится истинным как beta = 2 и alpha = 5. Таким образом, он ломается, и ему даже не нужно вычислять все поддерево **G.**
* Интуиция, стоящая за этим разрывом, заключается в том, что при **C** минимизатору было гарантировано значение 2 или меньше. Но максимизатору уже было гарантировано значение 5, если он выберет **B.** Так почему максимизатор когда-либо выбирал **C** и получал значение меньше 2? Опять же, вы можете видеть, что не имело значения, какими были эти последние 2 значения. Мы также сэкономили много вычислений, пропустив целое поддерево.
* **Теперь C** возвращает значение от 2 до **A.** Следовательно, наилучшее значение при **A** равно max(5, 2), что равно 5.
* Следовательно, оптимальное значение, которое может получить максимизатор, равно 5

Вот так выглядит наше финальное игровое дерево. Как вы можете видеть, **G** было вычеркнуто, поскольку оно никогда не вычислялось.

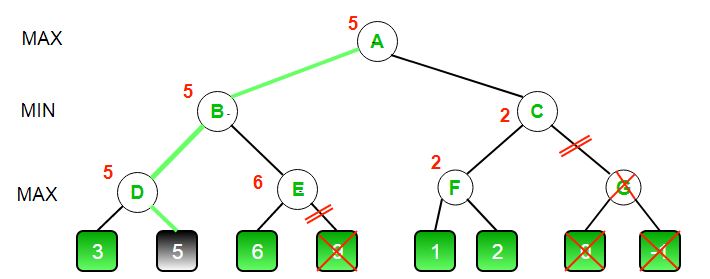


Рисунок 3 – Дерево решении (конец)

## Программная реализация метода минимакс для игры с нулевой суммой

### Разработка логической архитектуры программы

Логическая архитектура описывает программу с точки зрения ее концептуальной организации в слоях, пакетах, основных платформах, классах объектов, интерфейсах и подсистемах и их взаимодействия.

Для нашей программы диаграмма будет небольшой, но важно определить необходимые вспомогательные функции, основные функции и их взаимодействия.

Изначально понадобятся следующие вспомогательные функции:

* Проверка терминального состояния
  + Проверка на выигрыш одно из двух сторон
  + Либо ничейной позиции
* Оценочная функция по текущему состоянию игры
* Поиск пустых клеток для возможных ходов
* Проверка на адекватность выбранного хода
* Функция изменения текущего состояния на следующее состояние
* Визуализация
  + Отрисовка текущего состояния игры в консоли
  + Очищение поля консоли

Используя вспомогательные функции, уже строятся функции более высокого порядка, определяющие логику ходов игроков:

* Функция Минимакс (см. Рис. 4)
* Функция, определяющая логику хода компьютера в игре с использованием Минимакса
* Функция, определяющая логику хода человека в игре.

Для хранения текущего состояния «board» был выбран обычный двумерный массив, где пустые клетки обозначались «0», а символ человека и компьютера соответственно «-1» и «+1» (см. Листинг 2).

HUMAN = -1

COMP = +1

board = [

[0, 0, 0],

[0, 0, 0],

[0, 0, 0],

]

Листинг 2 – Константы для работы программы

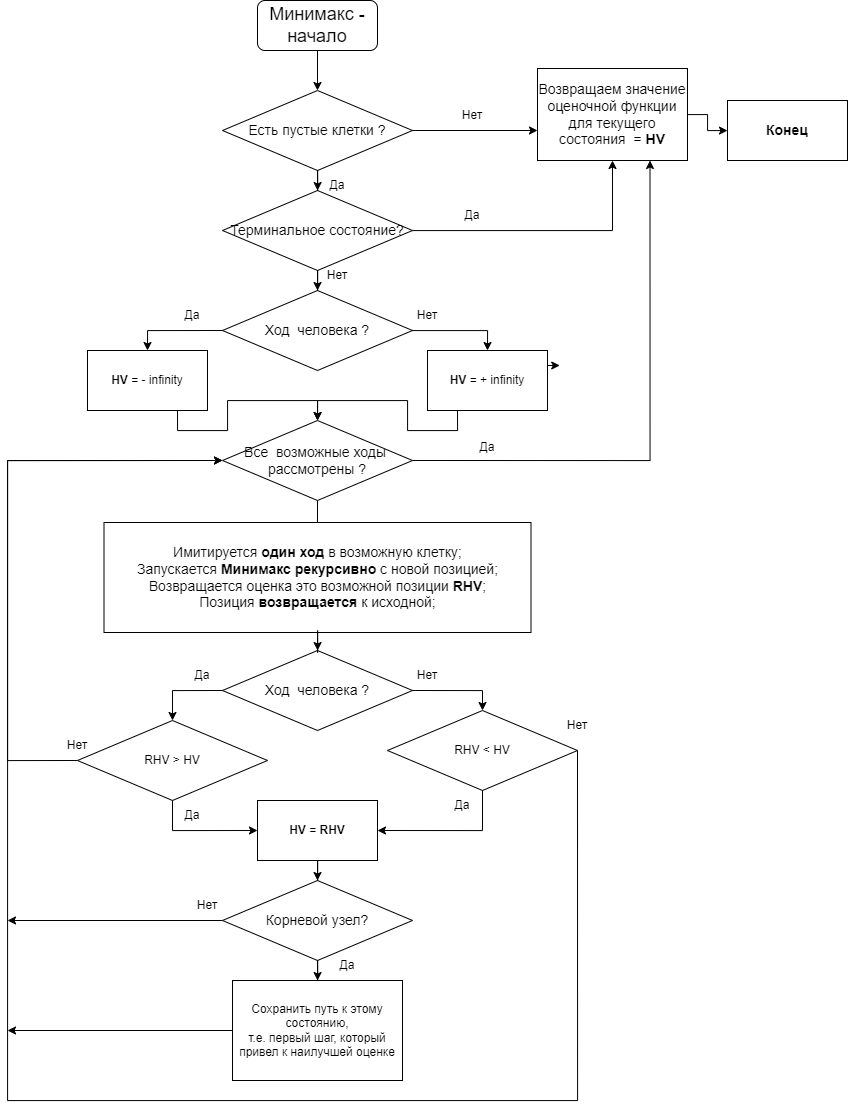


Рисунок 4 – Функция Минимакс

### 2.2 Реализация программного кода

### 2.2.1 Язык программирования

Язык программирования для реализации был выбран Python версии 3.11.

Это высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ.  
  
В настоящее время является самым популярным языком программирования и используется как для автоматизации серверной части веб-приложении и до Data Science ( анализ данных).

### 2.2.2 Реализация вспомогательных функции

1. Проверка выигрышной ситуации:

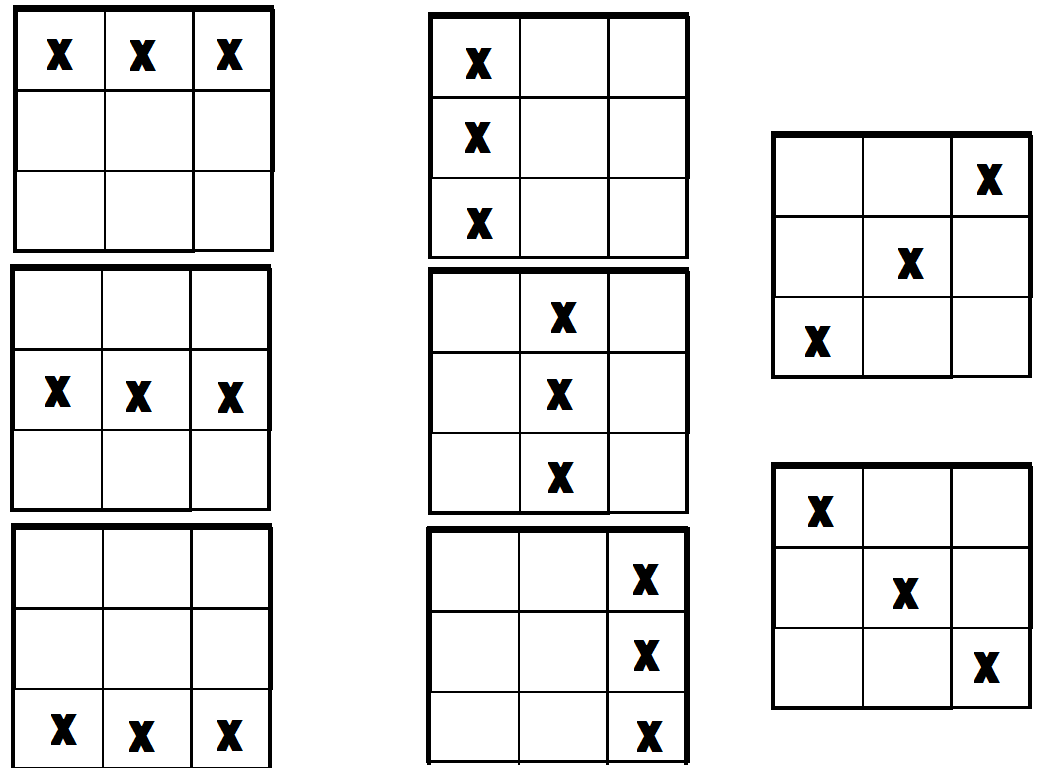


Рисунок 5 – Выигрышные положения в игре

Согласно рис.5 в игре «Крестики – нолики 3х3» 8 возможных выигрышных положении, то есть три в ряд одинаковых символов.

**def** wins(state, player):

win\_state = [

[state[0][0], state[0][1], state[0][2]],

[state[1][0], state[1][1], state[1][2]],

[state[2][0], state[2][1], state[2][2]],

[state[0][0], state[1][0], state[2][0]],

[state[0][1], state[1][1], state[2][1]],

[state[0][2], state[1][2], state[2][2]],

[state[0][0], state[1][1], state[2][2]],

[state[2][0], state[1][1], state[0][2]],

]

**if** [player, player, player] **in** win\_state:

**return** True

**else**:

**return** False

Листинг 3 – Функция проверки текущего состояния на выигрыш игрока

2. Поиск пустых ячеек на доске:

Тут все просто, мы итеративно проходим по двумерному массиву и сравниваем значении. Если значение равно «0» , то ячейка пуста. Возвращает двумерный массив из координат пустых клеток на доске.

3. Проверка на возможность хода:

Также несложная функция, проверяющая пустая ли клетка, куда хочет сходить игрок. Если да возвращает True, иначе False.

4. Оценочная функция:

Является одно из самых важных вспомогательных функции в программе. В моей реализации используется не только, константа в виде +10 баллов либо -10, но также учитывается глубина данного состояния на доске. То есть данная оценочная функция будет знать о ближайшем выигрыше для игрока. Также это глубина поможет более разумно использовать альфа-бета отсечение для вершин графов, которые находятся дальше от текущей позиции.   
Если бы не учитывалась глубина, альфа-бета отсечение отсекало бы все вне зависимости от глубины решения и выбирало путь через наименьший индекс ячейки, которая дает вероятность выиграть для игрока.

**def** evaluate(state, depth):

**if** wins(state, COMP):

score = +10 - depth \* 1

**elif** wins(state, HUMAN):

score = -10 + depth \* 1

**else**:

score = 0

**return** score

Листинг 4 – Оценочная функция

Остальные вспомогательные функции довольно просты в реализации и не требуют подробного описания.

### 2.2.3 Реализация основных функции

1. Функция Минимакс

Самая важная реализация всей программы – сам алгоритм минимакс. Был реализован на основе блок-схемы на рис.4. Сама реализация на базе рекурсии выглядит довольно просто (см. Листинг 5).

**def** minimax(state, potential\_moves, player, depth):

**global** MINIMAX\_LAUNCHES

MINIMAX\_LAUNCHES += 1

**if** player == COMP:

best = [-1, -1, -infinity]

**else**:

best = [-1, -1, +infinity]

**if** potential\_moves == 0 **or** game\_over(state):

score = evaluate(state, depth)

**return** [-1, -1, score]

**for** cell **in** empty\_cells(state):

x, y = cell[0], cell[1]

state[x][y] = player

score = minimax(state, potential\_moves - 1, -player, depth + 1)

**print**(score[2])

state[x][y] = 0

score[0], score[1] = x, y

**if** player == COMP:

**if** score[2] > best[2]:

best = score *# max value*

**else**:

**if** score[2] < best[2]:

best = score *# min value*

**return** best

Листинг 5 – Функция Минимакс

Человеку не близкому к рекурсии будет не понятно, как работает данная функция в деле.

В начале функции написана глобальная переменная для отслеживания сколько раз данная функция была запущена. Это понадобится нам в дальнейшем для сравнения разных модификации игры и алгоритма.

После статистической переменной заданы условия для определения игрока, то есть если ход алгоритма, то он выставляет для себя максимальное отрицательное значение – «-» бесконечность, так как он алгоритм не знает, чего может достичь.  
Если же ход человека, то для него важна максимизация данного значения для выигрыша.

После условии определения игрока задаются терминальные условия для остановки рекурсивного вызова функции вглубь. Условия эти таковы, что если нет пустых клеток для хода либо срабатывает функция проверки выигрыша, то текущее состояние оценивается с помощью оценочной функции возвращает в переменную *score.*

И получается с помощью рекурсии вызовов функции минимакса, реализуется поиск в глубину, так как рекурсивный вызов функции использует стек, как структуру данных для хранения всех вызовов. На этом принципе и работает алгоритм минимакс. Получается. Что данная функция проходится из текущего состояния до всех возможных с помощью поиска в глубину и выбирает тот путь, который в конце даст максимальную/минимальную оценку в зависимости от того, чей был ход при изначальном запуске.

Также считаю, нужным подметить особенность реализации. В использовании под аргументом player число 1 и -1, за счет чего достигается лёгкая смена игрока при рекурсивных вызовах. То есть каждом шаге вглубь функция инвертирует данный аргумент, и следит за тем, чей сейчас ход в игре. Эта особенность нетривиальна и очень полезна для простой реализации.

2. Ход алгоритма

**def** ai\_turn(c\_choice, h\_choice):

potential\_moves = len(empty\_cells(board))

**if** potential\_moves == 0 **or** game\_over(board):

**return**

clean()

**print**(f'Ход алгоритма: [{c\_choice}]')

render(board, c\_choice, h\_choice)

**if** potential\_moves == 9:

x = choice([0, 1, 2])

y = choice([0, 1, 2])

**else**:

move = minimax(board, potential\_moves, COMP, 0)

x, y = move[0], move[1]

set\_move(x, y, COMP)

time.sleep(1)

Листинг 6 – Функция хода алгоритма

Особенность данной функции в одном условном блоке, где проверяется количество пустых клеток на доске и если доска полностью пуста, то компьютер сходит в случайное место. Это условие возможно если только компьютер будет ходить первым. Это условия сильно сокращает количество запусков функции минимакс соответственно и времени расчета на первый шаг.

С другой стороны, мы не сильно теряем в эффективности алгоритма.

Для наглядности сравним две аналогичные игровые ситуации, когда компьютер ходит первым, но в одном случае с этим условием, а в другом случае без него. Также в каждой игре будем придерживаться тактики, чтобы играла закончилась ничьё для адекватности сравнения.

В первом случае с условием на случайный первый ход, функция минимакса была запущена 8318 раз.

Во втором случае с запуском алгоритма минимакс на первом ходе, алгоритм был запущен 557492 раз, что в 67 раз больше.

Этот эффект легко объясняется. При случайном первом ходе, алгоритм просчитывает примерно 7! случаев, но при просчёте первого хода ему приходится обрабатывать примерно 9! вариантов. Точно именно сколько раз запуститься алгоритм теоретически посчитать сложно, так как если перебрать все возможные ходы — это будет завышенная оценка, так как некоторые из рассмотренных позиции являются выигрышными.

3. Ход человека

**def** human\_turn(c\_choice, h\_choice):

potential\_moves = len(empty\_cells(board))

**if** potential\_moves == 0 **or** game\_over(board):

**return**

*# Dictionary of valid moves*

move = -1

moves = {

1: [0, 0], 2: [0, 1], 3: [0, 2],

4: [1, 0], 5: [1, 1], 6: [1, 2],

7: [2, 0], 8: [2, 1], 9: [2, 2],

}

clean()

**print**(f'Ваш ход :[{h\_choice}]')

render(board, c\_choice, h\_choice)

**while** move < 1 **or** move > 9:

**try**:

move = int(input('Число из (1..9): '))

coord = moves[move]

can\_move = set\_move(coord[0], coord[1], HUMAN)

**if** **not** can\_move:

**print**('Плохой ход')

move = -1

**except** (EOFError, KeyboardInterrupt):

**print**('Конец игры ')

exit()

**except** (KeyError, ValueError):

**print**('Введите число !')

Листинг 7 – Функция хода человека

Функция не представляет из себя ничего сложного, обработка исключении при вводе желаемой ячейки для хода, и словарь соответствии индексов одномерного массива с двумерным массивом для более интуитивного выбора ячейки.

Программа для игры в «крестики-нолики 3х3» была реализована. В дальнейшем данная реализация будет использоваться для оптимизации с помощью альфа-бета отсечения и других методов оптимизации алгоритма минимакс.

## Оптимизация алгоритма минимакс

Для поля 3х3 алгоритм работает приемлемо. Первый шаг алгоритма является самым объемным для просчета всех вариантов., поэтому максимальное количество рассматриваемых вершин в графе равно 9! = 362880 при условии, что первый ход совершает алгоритм. Но все же и этот результат можно оптимизировать.

Также интересно поведение алгоритма на других размерах доски:

* 3х4 с игрой до 3 в ряд символов
* 4х4 с игрой до 4 в ряд символов

Начнем с реализации альфа-бета оптимизации на примере обычной игры 3х3.

### Игра 3х3

Для внедрения оптимизации нужно добавить логику отсечения в функцию минимакса. Вот так будет выглядеть функция с реализованным альфа-бета отсечением.

**def** minimax(state, potential\_moves, player, depth, alpha, beta):

**global** MINIMAX\_LAUNCHES

MINIMAX\_LAUNCHES += 1

**if** player == COMP:

best = [-1, -1, -infinity]

**else**:

best = [-1, -1, +infinity]

**if** potential\_moves == 0 **or** game\_over(state):

score = evaluate(state, depth)

**return** [-1, -1, score]

**for** cell **in** empty\_cells(state):

x, y = cell[0], cell[1]

state[x][y] = player

**score = minimax(state, potential\_moves - 1, -player, depth + 1, alpha, beta)**

**print**(score[2])

state[x][y] = 0

score[0], score[1] = x, y

**if** player == COMP:

**if** score[2] > best[2]:

best = score *# max value*

**if score[2] >= beta:**

**return best**

**if score[2] > alpha:**

**alpha = score[2]**

**else**:

**if** score[2] < best[2]:

best = score *# min value*

**if score[2] <= alpha:**

**return best**

**if score[2] > beta:**

**beta = score[2]**

**return** best

Листинг 8 – Функция минимакс с альфа-бета отсечением

Различия подчеркнуты и выделены курсивом.

Теперь сравним эффективность данной оптимизации при одинаковых позициях в игре в таблице 1.

Таблица 1 – Сравнение количества запусков функции минимакс

|  |  |  |  |
| --- | --- | --- | --- |
| 3х3 | Без отсечения | С отсечением | Отношение |
| Кол-во запусков | 557492 | 37350 | 15 |

В результате анализа оптимизации было получено, что альфа-бета отсечение уменьшило количество запусков функции минимакса в 15 раз. Что является очень эффективным алгоритмом оптимизации.

### Игра 3х4

Игра 3х4 в крестики-нолики считается более интересной, так как здесь увеличивает количество выигрышных позиции с 8 до 13. Так как добавляются три выигрышных позиции по вертикали, если мы рассматриваем поле именно 3 в ширину и 4 в высоту, и два диагональные позиции выигрыша (см. Рис.6).

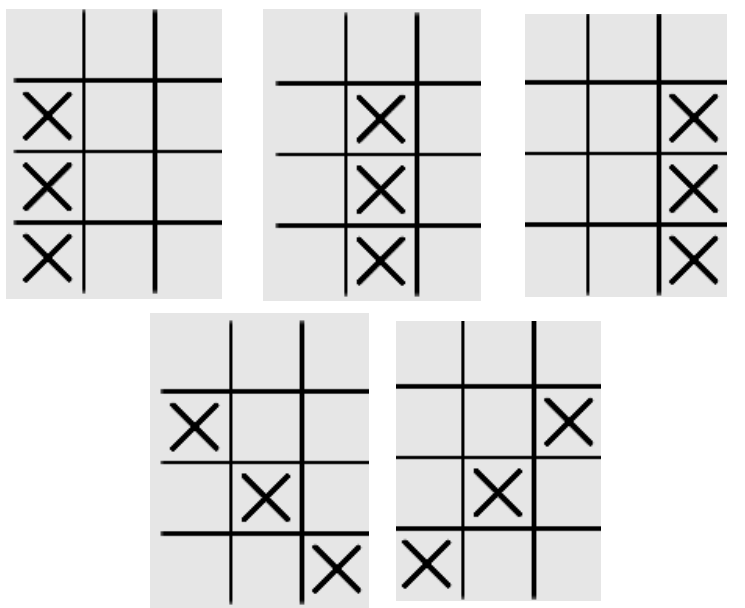


Рисунок 6 – Выигрышные позиции в игре 3х4

Реализация функции минимакса не нуждается в изменениях, поэтому перейдем к сравнению. Было сыграно абсолютно одинаковые игры с оптимизацией и без. Получились результаты, приведенные в таблице 2.

Таблица 2 – Сравнение количества запусков функции минимакс 3х4

|  |  |  |  |
| --- | --- | --- | --- |
| 3х4 | Без отсечения | С отсечением | Отношение |
| Кол-во запусков | 279256667 | 526658 | 53 |

### Игра 4х4 ( 4 в ряд)

Такого размера поля уже требует заметно больше времени для расчёта первых шагов для шагов алгоритма. В этой модификации игра уже невозможна без оптимизации алгоритма. Так как 14! = 87.178.291.200 вариантов для расчета, и для моего железа это занимает больше нескольких часов и с такой задержка игра невообразима. 14, так как 15 и 16 возможных ходов уже нецелесообразно огромны для сравнения, разве только если проверить теоретическое значение факториала и сравнить его с результатом с альфа-бета оптимизацией.

Таблица 3 – Сравнение количества запусков функции минимакс 4х4

|  |  |  |  |
| --- | --- | --- | --- |
| 4х4 | Без отсечения | С отсечением | Отношение |
| Кол-во запусков | 87.178.291.200 | 74.509.004 | 1170 |

В этом эксперименте оптимизация альфа-бета отсечением дало в 1000 большую эффективность работы алгоритма!

### Игра 4х4 (3 в ряд)

Была рассмотрена также игра 3 в ряд на поле 4 на 4. Добавлены все возможные выигрышные ситуации, таких оказалось относительно других модификации игры намного больше: 24 позиции. Но процессе тестирования правил игры и алгоритма, оказалось, что в этой модификации игры побеждает тот, кто ходит первым, но при условии, что он ходит оптимально.

# ЗАКЛЮЧЕНИЕ

В данной работе было проведено предпроектное исследование, включающее в себя глубокое изучение алгоритма поиска решения на графе Минимакс с оптимизацией в виде метода альфа-бета отсечения.

На этапе концептуальном проектировании была разработана логика программного кода: вспомогательных функции, основных функции и их взаимодействия.

Разработан программный код для имитации игры в «крестики-нолики» в разных его модификациях против алгоритма, который не проигрывает. Также был произведен анализ эффективности оптимизации с помощью альфа-бета отсечения.

В дальнейшей работе планируется реализации рассмотрение более объёмной задачи в виде игры Гомоку. И оптимизации алгоритма минимакс с помощью хеширования по Зобристу[10].

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гомоку /[Электронный ресурс] // URL: https://ru.wikipedia.org/wiki/%D0%93%D0%BE%D0%BC%D0%BE%D0%BA%D1%83 (дата обращения: 19.12.2022).
2. What is the difference between Backtracking and Recursion? / [Электронный ресурс] // — URL: https://www.geeksforgeeks.org/category/backtracking/ (дата обращения: 19.12.2022).
3. Нэш, Джон Форбс / [Электронный ресурс] // — URL: https://ru.wikipedia.org/wiki/%D0%9D%D1%8D%D1%88,\_%D0%94%D0%B6%D0%BE%D0%BD\_%D0%A4%D0%BE%D1%80%D0%B1%D1%81 (дата обращения: 19.12.2022)
4. Равновесие Нэша / [Электронный ресурс] // — URL: https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B2%D0%BD%D0%BE%D0%B2%D0%B5%D1%81%D0%B8%D0%B5\_%D0%9D%D1%8D%D1%88%D0%B0 (дата обращения: 20.12.2022).
5. Шеллинг, Томас / [Электронный ресурс] // — URL: https://ru.wikipedia.org/wiki/%D0%A8%D0%B5%D0%BB%D0%BB%D0%B8%D0%BD%D0%B3,\_%D0%A2%D0%BE%D0%BC%D0%B0%D1%81 (дата обращения: 20.12.2022).
6. Теория игр: Введение / [Электронный ресурс] //— URL: https://habr.com/ru/post/163681/ (дата обращения: 20.12.2022).
7. Minimax Algorithm in Game Theory | Set 3 (Tic-Tac-Toe AI – Finding optimal move) / [Электронный ресурс] // — URL: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=rp (дата обращения: : 22.12.2022).
8. Minimax Algorithm in Game Theory | Set 4 (Alpha-Beta Pruning) / [Электронный ресурс] // — URL: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/?ref=rp (дата обращения: 22.12.2022).
9. Чистый код: причины и следствия [Электронный ресурс] // Habr : [сайт]. — URL: https://habr.com/ru/company/dataart/blog/499348/ (дата обращения: 24.12.2022).
10. Minimax Algorithm in Game Theory | Set 5 (Zobrist Hashing) / [Электронный ресурс] // — URL: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/?ref=rp (дата обращения: 19.01.2023).

## Приложение А

