**Оглавление**

[**Оглавление** 2](#_Toc100169202)

[Глава 1 Теоретический обзор по выбранной теме 3](#_Toc100169203)

[1.1 Минимакс 3](#_Toc100169204)

[1.2 Альфа-бета-отсечение 4](#_Toc100169205)

[1.2.1 Ключевые идеи алгоритма 5](#_Toc100169208)

[1.2.2 Как найти лучший ход 6](#_Toc100169209)

[Глава 2 Практическое решение задачи: описание алгоритма работы программы 7](#_Toc100169210)

[2.1 Оценка хода 7](#_Toc100169211)

[2.2 Описание классов и методов 7](#_Toc100169212)

[2.3 Порядок работы приложения 9](#_Toc100169213)

[2.4 Пример работы приложения 10](#_Toc100169214)

[Глава 3 Оптимизация кода 12](#_Toc100169218)

[3.1 Методы оптимизации 12](#_Toc100169219)

[3.1.1 Макросы 12](#_Toc100169220)

[3.1.2 Динамическое выделение памяти 12](#_Toc100169221)

[3.1.3 Тернарный оператор 13](#_Toc100169222)

[3.1.4 Контейнеры класса 13](#_Toc100169223)

[Заключение 15](#_Toc100169224)

[СПИСОК ЛИТЕРАТУРЫ 16](#_Toc100169225)

[**Приложение** 17](#_Toc100169226)

Глава 1 Теоретический обзор по выбранной теме

* 1. Минимакс

*Минимакс* — правило принятия решений, используемое в теории игр, теории принятия решений, исследовании операций, статистике и философии для минимизации возможных потерь из тех, которые лицу, принимающему решение, нельзя предотвратить при развитии событий по наихудшему для него сценарию.

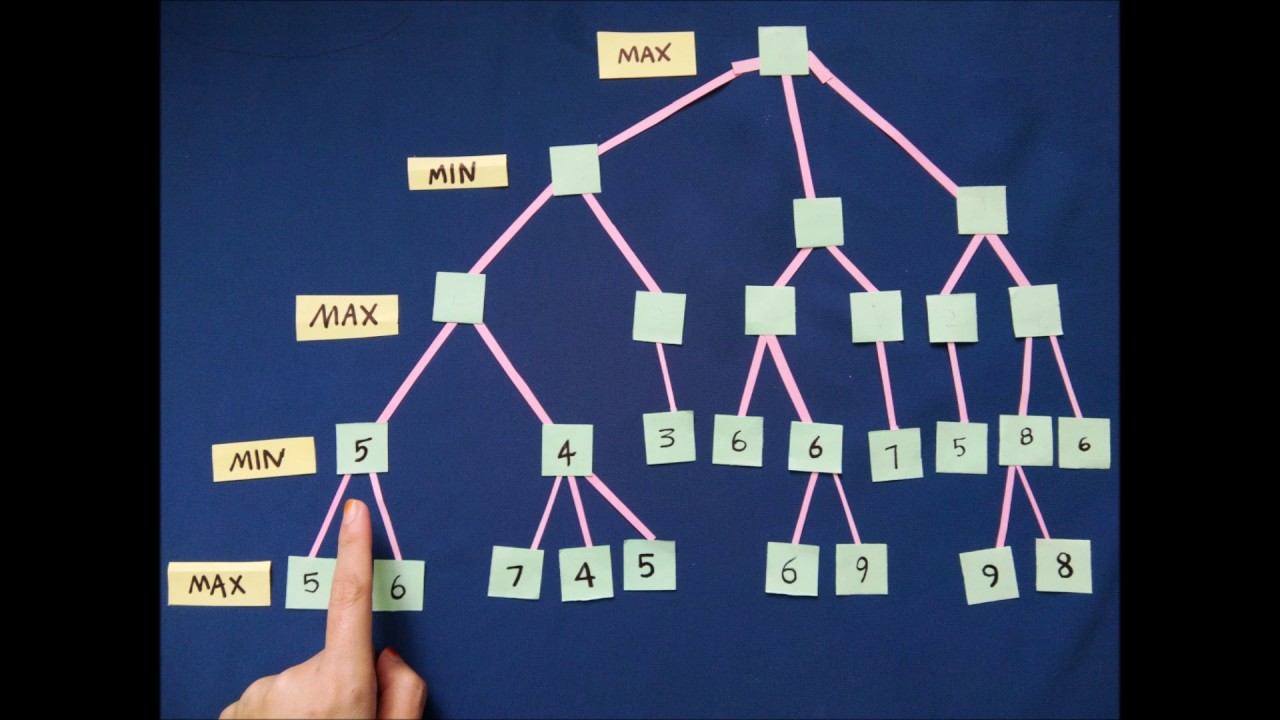


Рисунок 1. Пример минимаксового дерева

Критерий минимакса первоначально был сформулирован в теории игр для игры двух лиц с нулевой суммой Джеймсом Уолдгрейвом в 1713 году, в случаях последовательных и одновременных ходов, впоследствии получил развитие в более сложных играх и при принятии решений в условиях неопределённости

В дальнейшем этот алгоритм был оптимизирован, получив новое название: альфа-бета-отсечение.

* 1. Альфа-бета-отсечение

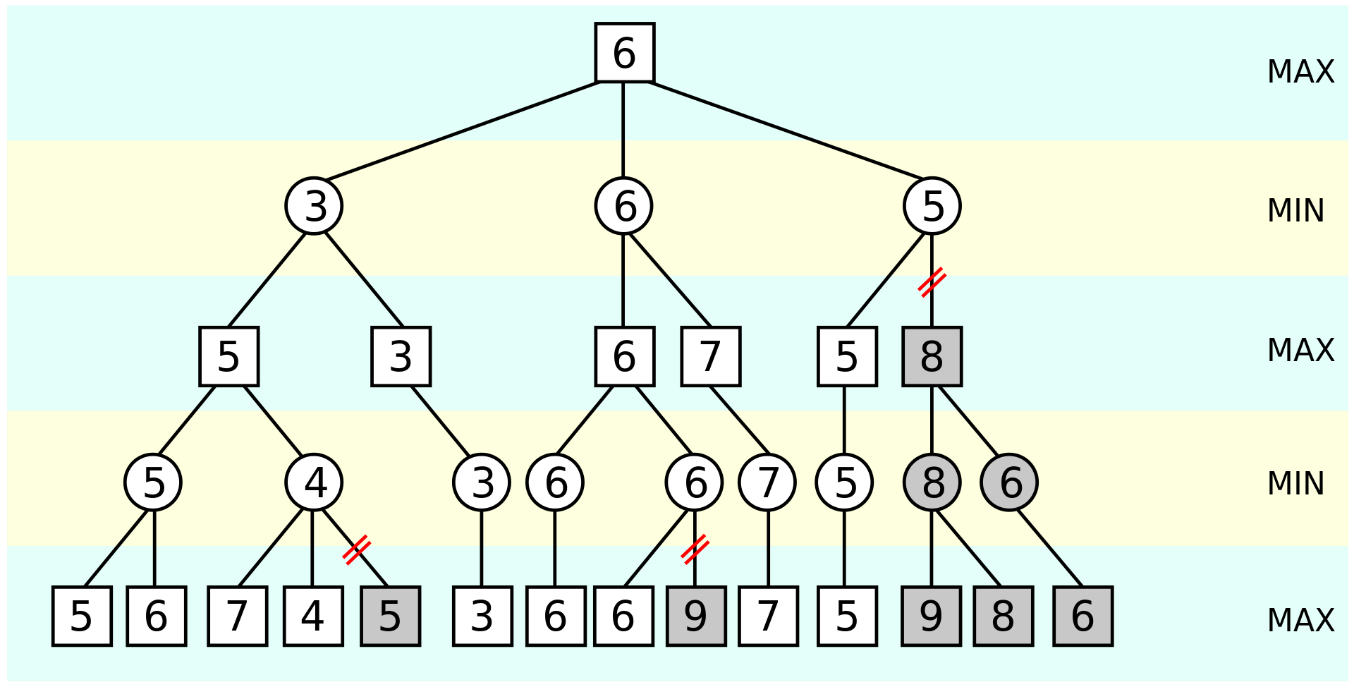


Рисунок 2. Пример работы алгоритма

*Альфа-бета-отсечение* — алгоритм поиска, стремящийся сократить количество узлов, оцениваемых в дереве поиска алгоритмом минимакса.

При минимаксном поиске проблема состоит в том, что количество состояний игры, которые должны быть исследованы в процессе поиска, зависит экспоненциально от количества ходов. К сожалению, такую экспоненциальную зависимость устранить невозможно, но существует возможность сократить ее наполовину. Вычисление правильного минимаксного решения возможно без проверки каждого узла в дереве игры. Это означает, что можно исключить из рассмотрения большие части дерева.

Конкретный метод, рассматриваемый в данной главе, называется альфа-бета-отсечение. Будучи применен к стандартному минимаксному дереву, этот метод возвращает такие же ходы, которые вернул бы минимаксный метод, но отсекает ветви, не способные повлиять на окончательное решение партии.

Напомним, что минимаксный поиск осуществляется в глубину, поэтому в любой момент времени достаточно рассматривать узлы вдоль единственного пути в дереве. Алгоритм альфа-бета-отсечения получил свое название по следующим двум параметрам, которые представляют пределы в зарезервированных значениях, присутствующих во всех узлах вдоль этого пути:

*а = значение наилучшего варианта (т.е. варианта с самым высоким значением), который был до сих пор найден в любой точке выбора вдоль пути для игрока МАХ;*

*Р = значение наилучшего варианта (т.е. варианта с самым низким значением), который был до сих пор найден в любой точке выбора вдоль пути для игрока MIN.*

Алгоритм альфа-бета-поиска в процессе своей работы обновляет значения а и Р, а также отсекает оставшиеся ветви в узле (т.е. прекращает рекурсивные вызовы), как только становится известно, что значение текущего узла хуже по сравнению с текущим значением а или Р для игрока *МАХ* или *MIN* соответственно.

* + 1. Ключевые идеи алгоритма

Ключевая идея алгоритма применительно к играм состоит в поиске наилучшего хода. На рис. 3 изображена ситуация, в которой игра в «крестики-нолики» близится к завершению, а ход предстоит сделать игроку Х. Алгоритм рассматривает три доступных хода, построив тем самым дерево ходов. Ветвь считается законченной, если ее листовой узел выполняет условия окончания игры (один из игроков выиграл или на поле не осталось пустых клеток), данное условие является «крайним случаем» рекурсии. В данном случае вершиной дерева является состояние игровой доски, которую требуется заполнить в контексте алгоритма ходом игрока Х, называемым максимизирующим.

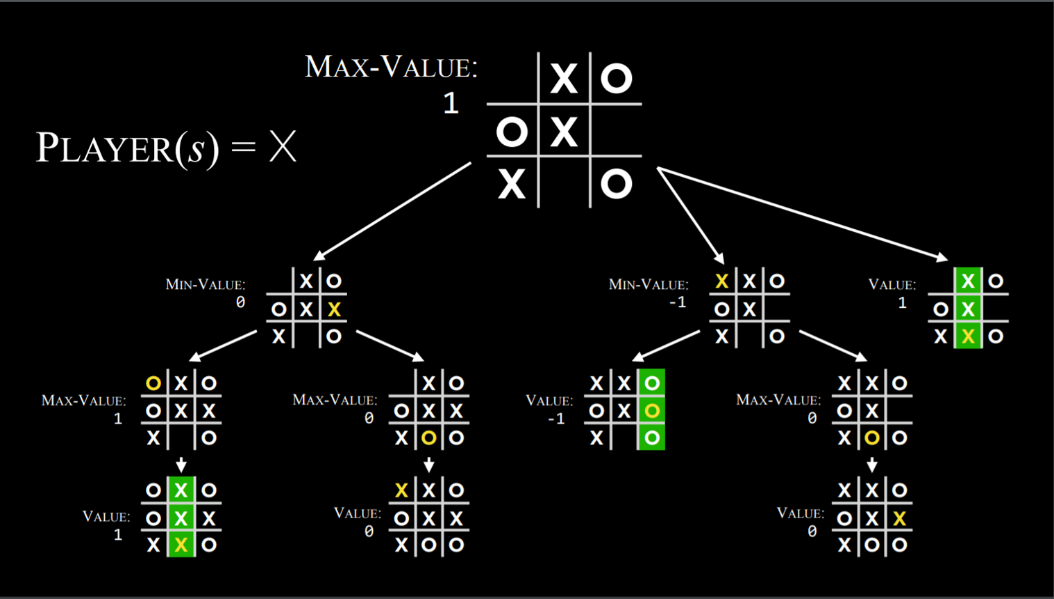


Рисунок 3. Дерево принятия решений алгоритма

Каждый из возможных вариантов порождает новые гипотетические ситуации, результат которых зависит от хода игрока О, называемого минимизирующим. Для принятия оптимального решения алгоритму необходимо оценивать возможные ходы. Описав условия окончания игры и определив ключевые понятия, оптимальной оценкой победы максимизирующего игрока следует задать значение, равное единице, минимизирующего — обратной величиной (-1), а случай ничьей — нулем. Оценивание и возврат результатов происходит снизу вверх, причем максимизирующий игрок ожидает получить максимально возможную оценку, а минимизирующий —наоборот.

Таким образом программа получает наилучшую оценку с учетом того, что ход соперника был оптимальным. Результатом работы алгоритма является выбор хода, обеспечивающего оптимальный результат, в контексте ситуации, изображенной на рис. 3, следует, что наиболее оптимистичным исходом максимизирующего игрока является победа, поскольку оценочная функция вернула значение 1, а значит, следует выбрать ход гарантированно обеспечивающий этот исход (рис. 4).

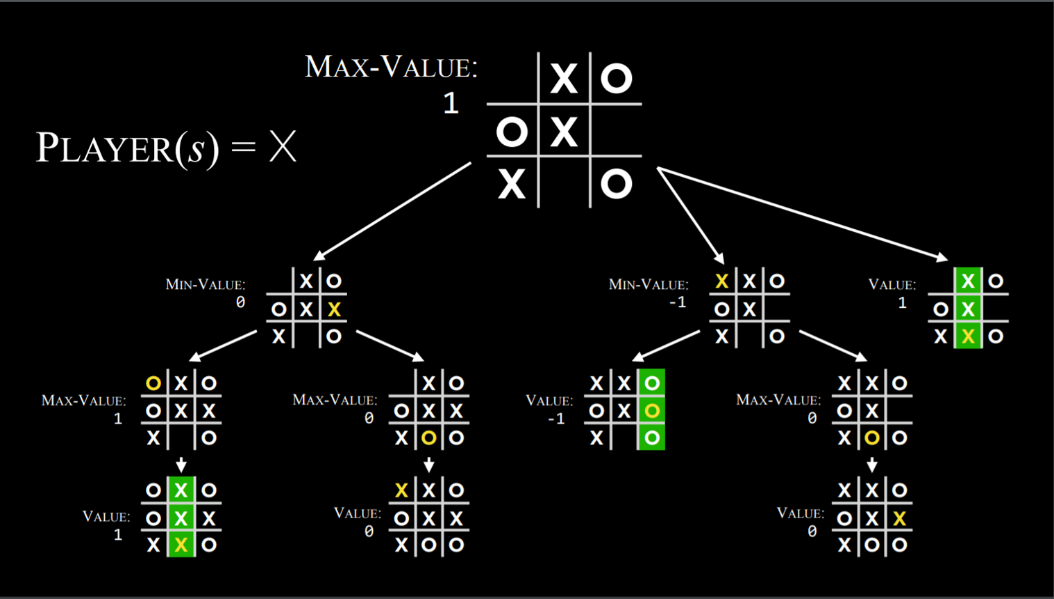


Рисунок 4. Лучший ход

* + 1. Как найти лучший ход

Чтобы найти самый лучший ход, программа перебирает все возможные ходы рекурсивной функцией (подробнее в главе 2), каждому просчитанному исходу партии бот присуждает ценность (переменная типа int). Значение этой переменной зависит от количества ходов, которое нужно совершить и его исхода.

Если можно выиграть партию в один ход, то лучше сделать его, а не рисковать. Также для оптимизации процесса поиска хода программа использует алгоритм Альфа-бета-отсечение. Это даёт возможность отсекать целые поддеревья, что, несомненно, даёт очень хорошую прибавку к производительности. Как я уже описал выше, в данной реализации алгоритма оценки, количество очков, которое функция оценки присудит тому или иному ходу напрямую зависит от глубины (количества ходов для достижения результата). Так что бот будет выбирать ходы, которые максимально быстро приведут его к победе.

Также программе не нужно просчитывать все возможные ходы оппонента так как бот играет от того, что оппонент будет выбирать максимально выгодные для него и максимально невыгодные для программы ходы. Получается, что просчитывая ход максимизирующего игрока и ответ минимизирующего, можно получить взвешенное древо вариантов исходов, а с помощью алгоритма Альфа-бета-отсечение это древо можно получить намного быстрее.

Глава 2 Практическое решение задачи: описание алгоритма работы программы

2.1 Оценка хода

В этой реализации бот не учитывает количество победных исходов в ветке для её оценки. На очки влияет только сам исход партии и глубина, которую пришлось просчитать для достижения результата. Главная задача бота - выиграть за как можно меньшее количество ходов.

В коде программы количество очков, присуждаемых какой-либо ветки просчитывается следующим образом:

|  |
| --- |
| best\_score = score - depth \* 10; |

Листинг 1. Запись нового лучшего счета

2.2 Описание классов и методов

В классе *Game* есть ряд функций, которые выполняют определенные задачи:

Рекурсивная функция *fieldState* определяет, есть ли победитель (или победил ли игрок marker). Атрибуты r и c обозначают на какой ячейке игрового поля сейчас алгоритм, cells - какой длины текущая найденная линия, move - текущее выполняемое действие (-1 если это первый заход в алгоритм).

|  |
| --- |
| char fieldState(const int i, const int j, const int cells, Move move, char marker) const {  if (marker == ‘ ‘)  return prevMove;  else {  if (operator[](i)[j] == marker) return true;  else return false;  }  } |

Листинг 2. Рекурсивная функция fieldState

Эта функция *fieldState* также служит для определения победителя

|  |
| --- |
| char fieldState(short int marker = ‘ ‘) const {  char winner;  for (int i = 0; i < size; ++i)  for (int j = 0; j < size; ++j)  if (operator[](i)[j] != ‘ ‘) {  winner = fieldState(i, j, 1, noMove, marker);  if (winner != -1) return winner;  }  return -1;  } |

Листинг 3. Функция определения победителя

Ниже представлен сам алгоритм минимакс(альфа-бета отсечение)

|  |
| --- |
| pair < int, pair < int, int >> miniMax(char marker, int depth, int alpha, int beta) const {  pair < int, int > best\_move = make\_pair(-1, -1);  int best\_score = (marker == player2) ? INFMINUS : INFPLUS; |

Листинг 4. Часть алгоритма "Альфа-бета отсечение"

Где переменная *make\_pair* хранит наилучший шаг, представляем из себя пару

(r, c), где r - строка игрового поля, а c - столбец)

Здесь мы максимизируем ход игрока

|  |
| --- |
| if (marker == player2){  int score = miniMax(player1, depth + 1, alpha, beta).first;  if (best\_score < score){  best\_score = score - depth \* 10;  best\_move = make\_pair(i, j);  alpha = max(alpha, best\_score);  operator[](i)[j] = ‘ ‘;  if (beta <= alpha) {  break;  }  } |

Листинг 5. Максимизация хода игрока

А тут минимизурем ход оппонента

|  |
| --- |
| if (marker == player2){  int score = miniMax(player1, depth + 1, alpha, beta).first;    if (best\_score < score){  best\_score = score - depth \* 10;  best\_move = make\_pair(i, j);    alpha = max(alpha, best\_score);  operator[](i)[j] = ‘ ‘;  if (beta <= alpha) {  break;  }  } |

Листинг 6. Минимизация хода оппонента

Так же у нас имеются функция служащие для отработки шагов игрока *humanMove,* функция *areFieldsFull* отвечает определения заполнено ли поле.Функция *aiMove* вызывает алгоритм минимакс, записывает шаги в ячейки матрицы(таблицы).

|  |
| --- |
| void clearField() {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  operator[](i)[j] = ‘ ‘;  }  }  } |

Листинг 7. Функция очистки полей

|  |
| --- |
| bool areFieldsFull() const {  for (int i = 0; i < rows; ++i)  for (int j = 0; j < cols; ++j)  if (operator[](i)[j] == ‘ ‘)  return false;  return true;  } |

Листинг 8. Функция определения заполнено поле или нет

2.3 Порядок работы приложения

После того как мы запустили нашу игру нам следует выбрать режим игры: игра 1 на 1 с другим человеком, либо же игра с роботом.

Далее выбираем каким знаком мы будем играть (Х/О).

После нам следует выбрать размер поля, и условие победы (сколько клеток нужно заполнить).

Прошу обратить внимание, что при совершении хода ряды и колонки начинают свой отсчет от 0.

В связи с тем, что робот долго выбирает ход было принято решение ограничить игровые поля: в случае доски 3 на 3 количество уникальных возможных шагов с учетом отсечения неэффективных - ~255000, в случае доски 4 на 4 количество возможных досок в 2187! раз больше.

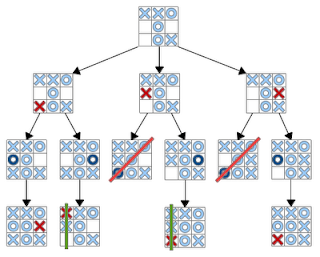


Рисунок 5. Частичное дерево игровых ситуаций для игры в крестики-нолики

2.4 Пример работы приложения



Рисунок 6. Главное меню приложения

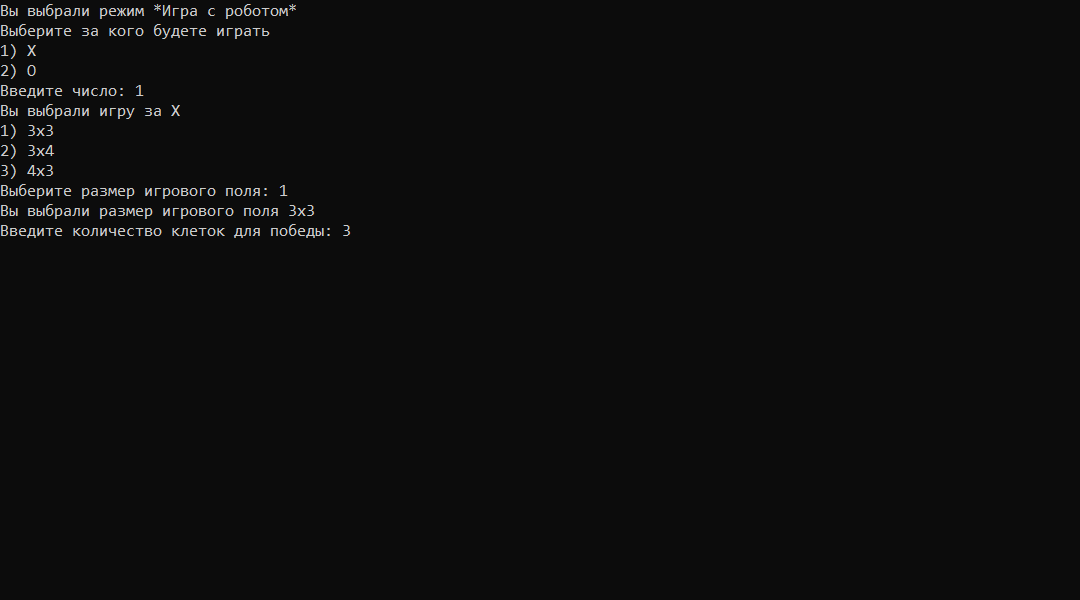


Рисунок 7. Настройка параметров партии

|  |  |
| --- | --- |
| https://i.imgur.com/enUJYVr.png  Рисунок 8. Выбор первого хода | https://i.imgur.com/eydThaB.png  Рисунок 9. Выбор следующего хода |

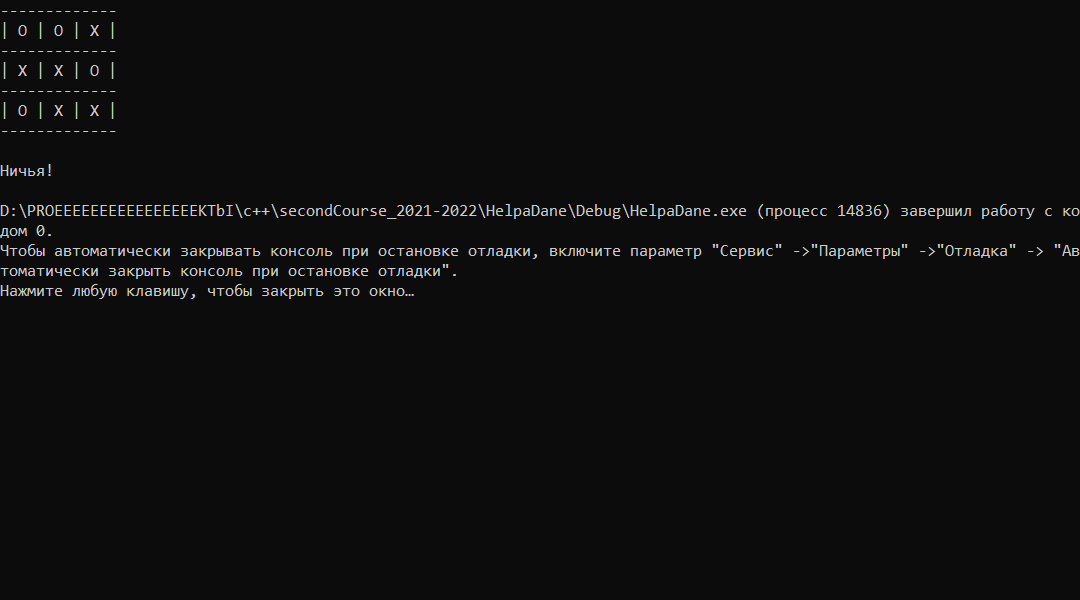


Рисунок 10. Результат партии

Глава 3 Оптимизация кода

3.1 Методы оптимизации

3.1.1 Макросы

В приложении были задействованы макросы для упрощения работы. Если имя макроса распознается в исходном тексте программы или в аргументах некоторых других команд препроцессора, оно рассматривается как вызов этого макроса. Имя макроса заменяется копией тела макроса. Оно помогает оптимизировать код, не использовать несколько раз одни и те же функции и значения.

|  |
| --- |
| #define INFPLUS 1000  #define INFMINUS - 1000  #define char1 'X'  #define char2 'O' |

Листинг 9. Использование макросов в коде

Макросы - это препроцессорные "функции", т.е. лексемы, созданные с помощью директивы *#define*, которые принимают параметры подобно функциям. Они обрабатывают исходный код программы ДО того, как она будет скомпилирована. Существует два типа макросов. Макросы, подобные объектам, не имеют аргументов. Макросы, подобные функциям, можно определить для приема аргументов, чтобы они выглядели и действовали как вызовы функций. Поскольку макросы не создают фактические вызовы функций, вы иногда можете сделать программы быстрее, заменив вызовы функций на макросы. Это помогает сократить код, сделать его более читабельным, так же они влияют на размер исходника и скорость работы.

### 3.1.2 Динамическое выделение памяти

Объекты и массивы можно создавать и удалять динамически с помощью *new* и *delete* либо с помощью *malloc* и *free*. Это может быть полезным, когда требуемый объём памяти неизвестен во время компиляции.

Преимущества динамического выделения памяти следующие.

1) В некоторых случаях структура программы становится яснее.

2) Выделяется ровно столько места, сколько необходимо, что делает кэширование данных более эффективным, нежели для случая, когда массив фиксированного размера сделан очень большим, чтобы охватить наихудший случай максимально возможного требования к памяти.

3) Оно полезно, когда неразумно заранее ограничивать сверху потребный объём памяти.

Таблица 1. Сравнение результатов оптимизации при динамическом выделение памяти

|  |  |  |  |
| --- | --- | --- | --- |
|  | 3x3 | 4x3 | 3x4 |
| До оптимизации | 0,16 с | 12,20 с | 23,20 с |
| После оптимизации | 0,14 с | 11,19 с | 20,44 с |

### 3.1.3 Тернарный оператор

Ради отказа от работы предсказателя ветвления перешел к тернарным операторам.

|  |
| --- |
| int best\_score = (marker == player2) ? INFMINUS : INFPLUS; |

Листинг 9. Пример работы тернарного оператора

Таблица 2. Сравнение результатов оптимизации при тернарных операторах

|  |  |  |  |
| --- | --- | --- | --- |
|  | 3x3 | 4x3 | 3x4 |
| До оптимизации | 0,17 с | 12,70 с | 22,24 с |
| После оптимизации | 0,14 с | 11,33 с | 19,4 с |

### 3.1.4 Контейнеры класса

При каждом использовании динамического выделения памяти рекомендуется обёртывать выделяемую память в контейнерный класс. У контейнерного класса должен иметься деструктор, обеспечивающий освобождение всего, что выделено. Это наилучший способ предотвратить утечки памяти и другие распространённые ошибки при программировании на С++, связанные с динамическим выделением памяти.

Контейнерные классы также могут быть удобны для добавления к массиву проверки границ и для более совершенных структур данных с обращением по принципу ”первым пришёл — первым ушёл“ или по принципу ”первым пришёл — последним ушёл“ (FIFO, LIFO), возможностей сортировки и поиска, двоичных деревьев, хэш–таблиц и т.п.

Общепринято создавать контейнерные классы в виде шаблонов, которым тип элемента контейнера передаётся в качестве параметра шаблона. С точки зрения производительности нет никаких затрат из–за использования шаблонов.

Имеются готовые шаблоны контейнерных классов для многих различных целей. Наиболее широко используемый набор контейнеров — Стандартная библиотека шаблонов (STL), поставляемая совместно с большинством современных компиляторов с Си++. Преимущество использования уже готовых контейнеров состоит в том, что вы не должны заново изобретать велосипед. Контейнеры из STL универсальны, гибки, хорошо проверены и очень полезны для многих различных целей.

Однако STL проектируется для общности и гибкости, тогда как скорость выполнения, экономия памяти, эффективность кэша и размер кода имели низкий приоритет. В особенности в STL излишне расточительно динамическое выделение памяти. Некоторые шаблоны STL, такие, как list, set и map, склонны даже выделять больше блоков памяти, чем имеется объектов в контейнере.

Частое выделение и освобождение памяти с помощью *new* и *delete* (или *malloc* и *free*) вызывает фрагментацию памяти и неэффективное кэширование. Как упомянуто выше, имеются значительные накладные расходы на управление памятью и сборку мусора.

Заключение

Алгоритм *Альфа-бета отсечение* по своей сути является методом «ветвей и границ» и заключается в том, чтобы после «прохода» по одной из ветвей дерева решений «отсекать» ветви, заведомо не имеющие оптимального решения, относительно коэффициентов альфа и бета, полученных на первом проходе.

Альфа — это минимально возможная оценка, которую может получить максимизирующий игрок, инициализируется значением минус бесконечности и наоборот. Если в одном из узлов ветви значения альфа больше либо равно бета, найден ход, который гарантирует победу максимизирующему игроку. Если значения этих параметров в текущем узле не улучшаются, ветвь не следует рассматривать, поскольку все ее «потомки», как и она сама, не содержат оптимального решения. Таким образом, описанный метод позволяет находить оптимальное решение, заметно сократив при этом число рекурсивных вызовов.

К позитивным сторонам данного алгоритма следует отнести адекватную модель принятия решений. Данный эффект достигается благодаря тому, что алгоритму заранее известны все возможные исходы партии уже в самом начале игры. Однако алгоритм ограничен в реализации по ряду причин. Во-первых, очевидным является факт, что рассмотрение каждой гипотезы влечет за собой создание копии игровой доски, а это говорит о высокой ресурсоемкости алгоритма, т. е. эффективность алгоритма по памяти весьма низкая. То же касается и скорости выполнения: при линейном росте размера игрового поля число возможных решений возрастает экспоненциально, а поскольку прием рекурсии использует стек вызовов, имеющий ограниченный ресурс, выполнения программы может не произойти вовсе.

СПИСОК ЛИТЕРАТУРЫ

1. Нистрем, Р. Паттерны программирования / Р. Нистрем. – Москва : Эксмо, 2021. – 432 с.
2. Кормен, Т.Х. Алгоритмы: построение и анализ / Х.Т. Кормен, П . – : Вильямс, 2006. – 1296 с.
3. Братко, И. Алгоритмы искусственного интеллекта на языке PROLOG / И. Братко, Д.Э. – : Вильямс, 2017. – 640 с.
4. Ахо, А.В. Структуры данных и алгоритмы / А.В. Ахо, Д.Э. Хопкрофт, Д.Д. Ульман. – : Вильямс, 2018. – 400 с.
5. Сикорд, С Безопасное программирование на С и С++ / С Сикорд. – : Вильямс, 2015. – 498 с.

**Приложение**

|  |
| --- |
| #include <iostream>  #include <string>  #include <limits>  #include <locale.h>  #include <cstdio>    using namespace std;    #define INFPLUS 1000  #define INFMINUS - 1000  #define char1 'X'  #define char2 'O'    enum Moves {  noMove,  moveUp,  moveRightUp,  moveRight,  moveRightDown,  moveDown,  moveLeftDown,  moveLeft,  moveLeftUp  };    class Game {  char\* field;  int cellsToWin;  int rows;  int collums;  char player1;  char player2;  char prevMove;      // Функция очистки поля  void clearField() {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < collums; j++) {  operator[](i)[j] = ' ';  }  }  }  //фукнция для определения победителя  char fieldState(short int marker = ' ') const {  char winner; //результат вызова функции для каждой ячейки игрового поля  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < collums; ++j) { //для каждой ячейки игрового поля  if (operator[](i)[j] != ' ') { //если в ней есть знак  winner = fieldState(i, j, 1, noMove, marker); //вызываем от этой ячейки рекурсивную функцию определения победителя  if (winner != -1) return winner; //если вызов дал результат - возвращаем  }  }  }  //иначе не один из вызовов не дал результат - победителя нет, возвращаем -1  return -1;  }    //функция определения заполнено ли поля  bool areFieldsFull() const {  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < collums; ++j) {  if (operator[](i)[j] == ' ') {  return false;  }  }  }  return true;  }    // Рекурсивная функция определения, победил ли игрок со знаком marker  char fieldState(const int i, const int j, const int cells, Moves move, char marker) const {    if (cells == cellsToWin) { //если мы нашли линию нужного размера  //если определяется есть ли победитель, возвращаем победителя  if (marker == ' ') return prevMove;  //иначе определяется победил ли игрок со знаком marker  else {  //если знак marker - значит победил, возвращаем true. Иначе проиграл - возвращаем false  if (operator[](i)[j] == marker) return true;  else return false;  }  }    //результат для поочередного рекурсивного вызова функций  char winner;  //если можем пойти вверх и текущее действие либо первое, либо шаг вверх  if ((i != 0) && ((move == noMove) || (move == moveUp))) {  if (operator[](i - 1)[j] == operator[](i)[j]) {  winner = fieldState(i - 1, j, cells + 1, moveUp, marker);  if (winner != -1) return winner;  }  }  if ((j != 0) && ((move == noMove) || (move == moveLeft))) {  if (operator[](i)[j - 1] == operator[](i)[j]) {  winner = fieldState(i, j - 1, cells + 1, moveLeft, marker);  if (winner != -1) return winner;  }  }  if (((i != 0) && (j != 0)) && ((move == noMove) || (move == moveLeftUp))) {  if (operator[](i - 1)[j - 1] == operator[](i)[j]) {  winner = fieldState(i - 1, j - 1, cells + 1, moveLeftUp, marker);  if (winner != -1) return winner;  }  }  if ((i != rows - 1) && ((move == noMove) || (move == moveDown))) {  if (operator[](i + 1)[j] == operator[](i)[j]) {  winner = fieldState(i + 1, j, cells + 1, moveDown, marker);  if (winner != -1) return winner;  }  }  if ((j != collums - 1) && ((move == noMove) || (move == moveRight))) {  if (operator[](i)[j + 1] == operator[](i)[j]) {  winner = fieldState(i, j + 1, cells + 1, moveRight, marker);  if (winner != -1) return winner;  }  }  if (((i != rows - 1) && (j != collums - 1)) && ((move == noMove) || (move == moveRightDown))) {  if (operator[](i + 1)[j + 1] == operator[](i)[j]) {  winner = fieldState(i + 1, j + 1, cells + 1, moveRightDown, marker);  if (winner != -1) return winner;  }  }  if (((i != rows - 1) && (j != 0)) && ((move == noMove) || (move == moveLeftDown))) {  if (operator[](i + 1)[j - 1] == operator[](i)[j]) {  winner = fieldState(i + 1, j - 1, cells + 1, moveLeftDown, marker);  if (winner != -1) return winner;  }  }  if (((i != 0) && (j != collums - 1)) && ((move == noMove) || (move == moveRightUp))) {  if (operator[](i - 1)[j + 1] == operator[](i)[j]) {  winner = fieldState(i - 1, j + 1, cells + 1, moveRightUp, marker);  if (winner != -1) return winner;  }  }  //если не один из вызовов сверху не дал результат - победителя от текущей ячейки нет, возвращаем -1  return -1;    }    // Функция хода пользователя  void humanMove() {  int row, col;  cout << "Ваш ход" << "**\n**";  cout << "Ряд (0-" << to\_string(rows - 1) << "):";    // Проверка ввода  if (!(cin >> row)) {  cout << "**\n**Ошибка ввода**\n**";  cin.clear();  cin.ignore(numeric\_limits < streamsize > ::max(), '**\n**');  return humanMove();  }    cout << "Колонка (0-" << to\_string(collums - 1) << "):";  // Проверка ввода  if (!(cin >> col)) {  cout << "**\n**Ошибка ввода**\n**";  cin.clear();  cin.ignore(numeric\_limits < streamsize > ::max(), '**\n**');  return humanMove();  }    // Проверка ввода  if ((row >= rows) || (col >= collums) || (operator[](row)[col] != ' ')) {  cout << "**\n**Неверные координаты**\n**";  cin.clear();  cin.ignore(numeric\_limits < streamsize > ::max(), '**\n**');  return humanMove();  }    // Записываем шаг в ячейку таблицы (записываемый шаг - противоположный от последнего)  operator[](row)[col] = (prevMove == char2) ? char1 : char2;  // Обновляем переменную, хранящую последний шаг  prevMove = operator[](row)[col];  }    // Алгоритм минимакс (альфа-бета отсечение)  pair < int, pair < int, int >> miniMax(char marker, int depth, int alpha, int beta) const {  // Лучший ход (представляем из себя пару (r, c), где r - строка игрового поля, а c - столбец)  pair < int, int > best\_move = make\_pair(-1, -1);  // Лучший счет  int best\_score = (marker == player2) ? INFMINUS : INFPLUS;    // Условия завершения  if (areFieldsFull() || fieldState(player2) != -1) {  char winner = fieldState(player2);  if (winner != -1) best\_score = winner ? INFPLUS : INFMINUS;  else best\_score = -1;    return make\_pair(best\_score, best\_move);  }    Game current(\*this);    for (int i = 0; i < rows; ++i) {  for (int j = 0; j < collums; ++j) {  if (current[i][j] == ' ') {  operator[](i)[j] = marker; // Делаем предположительный ход    // Максимизируем ход игрока  if (marker == player2) {  int score = miniMax(player1, depth + 1, alpha, beta).first;    // Получаем лучший ход  if (best\_score < score) {  best\_score = score - depth \* 10;  best\_move = make\_pair(i, j);    alpha = max(alpha, best\_score);  // Отменяем ход  operator[](i)[j] = ' ';  if (beta <= alpha) {  break;  }  }    } // Минимизируем ход оппонента  else {  int score = miniMax(player2, depth + 1, alpha, beta).first;    // Получаем лучший ход  if (best\_score > score) {  best\_score = score + depth \* 10;  best\_move = make\_pair(i, j);    beta = min(beta, best\_score);  // Отменяем ход  operator[](i)[j] = ' ';  if (beta <= alpha) {  break;  }  }    }  // Отменяем ход  operator[](i)[j] = ' ';    }  }  }  return make\_pair(best\_score, best\_move);  }    // Функция хода бота  void aiMove() {  // Вызываем алгоритм минимакс (альфа-бета отсечение)  pair < int, pair < int, int > > ai\_move = miniMax(player2, 0, INFMINUS, INFPLUS);  // Записываем шаг в ячейку таблицы (записываемый шаг - противоположный от последнего)  operator[](ai\_move.second.first)[ai\_move.second.second] = (prevMove == char2) ? char1 : char2;;  // Обновляем переменную, хранящую последний шаг  prevMove = operator[](ai\_move.second.first)[ai\_move.second.second];  }    public:  Game(const int cos,  const int ros,  const int cells,  const char first) {  collums = cos;  rows = ros;  cellsToWin = cells;  player1 = first;  player2 = (player1 == char2) ? char1 : char2;  prevMove = player2;  field = new char[rows \* collums];    }    Game(const Game& other) : rows(other.rows), collums(other.collums), cellsToWin(other.cellsToWin), player1(other.player1), player2(other.player2), prevMove(other.prevMove) {  field = new char[rows \* collums];  for (int i = 0; i < rows; ++i) {  for (int j = 0; j < collums; ++j) {  // Инициализируем ячейку матрицы  operator[](i)[j] = other[i][j];  }  }  }    ~Game() {  delete[] field;  }    // Прегрузка оператора [] сделана для быстрой работы массива при вводе очень большой размерности поля  // принцип выделения памяти как при создании обычного двумерного массива только без массы циклов  char\* operator[](const int row) const {  return field + row \* collums;  }    // Функция вывода поля  // Передаем в функцию наше поле обьявленное в main (оно может называться в маине не только field, в данном случае это та переменная которая использыется только в этой функции)  void printField() {  // Очистка экрана  system("cls");  for (int j = 0; j < collums; j++) {  cout << "----";  }  cout << "-**\n**";  for (int i = 0; i < rows; i++) {  // Ряд символов  for (int j = 0; j < collums; j++) {  cout << "| " << operator[](i)[j] << " ";  }  cout << "|**\n**";  for (int j = 0; j < collums; j++) {  cout << "----";  }  cout << "-**\n**";  }  }    // Функция запуска игры  void run(const bool aiMode = true) {  clearField();  char winner = fieldState(); // Инициализируем начальное значение переменной, хранящее победителя  while ((winner == -1) && !areFieldsFull()) { // Пока нет победителя и поле не заполнено  printField(); // Выводим поле  humanMove(); // Делаем ход игрока  if (aiMode) { // Если играем против ИИ  winner = fieldState(); //Смотрим не победил ли игрок  if ((winner != -1) || areFieldsFull()) break; // Если игрок победит или поле заполнено - завершаем игру  aiMove(); // Делаем ход ИИ  }  winner = fieldState(); // Обновляем переменную с победителем  }  printField(); //Выводим таблицу  if (winner != -1) { //Если есть победитель  cout << "**\n**Игрок " << ((winner == player1) ? "1" : "2") << " выиграл**\n**"; //Выводим его  }  else {  cout << "**\n**Ничья!**\n**"; //Иначе результат игры - ничья  }  }  };    int main() {  setlocale(LC\_ALL, "Rus");    int aiMode = true;  int modePlayer = char1;  int choice;  cout << "**\n**";  cout << "**\t\t**\*-------------------Меню-------------------\*" << endl;  cout << "**\t\t**|1) Игра против другого человека |" << endl;  cout << "**\t\t**|2) Игра против робота |" << endl;  cout << "**\t\t**|3) Выйти |" << endl;  cout << "**\t\t**\*------------------------------------------\*" << endl;  TryAgain:  cout << "**\n\n**Введите число: ";  cin >> choice;    switch (choice) {  case 1:  aiMode = false;  choice = 0;  system("cls");  cout << "Вы выбрали режим \*Один на один\*" << endl;  break;  case 2:  aiMode = true;  choice = 0;  system("cls");  cout << "Вы выбрали режим \*Игра с роботом\*" << endl;  break;  case 3:  return 0;  default:  cout << "Некорректное значение" << endl;  goto TryAgain;  break;  }    cout << "Выберите за кого будете играть" << endl;  cout << "1) X" << endl;  cout << "2) O" << endl;  TryAgain2:  cout << "Введите число: ";  cin >> choice;    switch (choice) {  case 1:  cout << "Вы выбрали игру за X" << endl;  modePlayer = char1;  break;  case 2:  cout << "Вы выбрали игру за O" << endl;  modePlayer = char2;  break;  default:  cout << "Некорректное значение" << endl;  goto TryAgain2;  break;  }    int rows, collums, cellsToWin;    cout << "1) 3x3" << endl;  cout << "2) 3x4" << endl;  cout << "3) 4x3" << endl;  TryAgain3:  cout << "Выберите размер игрового поля: ";  cin >> choice;    switch (choice) {  case 1:  cout << "Вы выбрали размер игрового поля 3x3" << endl;  rows = 3;  collums = 3;  break;  case 2:  cout << "Вы выбрали размер игрового поля 3x4" << endl;  rows = 4;  collums = 3;  break;  case 3:  cout << "Вы выбрали размер игрового поля 4x3" << endl;  rows = 4;  collums = 4;  break;    default:  cout << "Некорректное значение" << endl;  choice = 0;  goto TryAgain3;  break;  }    cout << "Введите количество клеток для победы: ";  cin >> cellsToWin;    Game field(rows, collums, cellsToWin, modePlayer);  field.run(aiMode);    return 0;  } |