Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интелекта
Направление: 02.03.01 «Математика и компьютерные науки»

Программирование и алгоритмизация

Отчёт по курсовой работе «Крестики-нолики»

Выполнил студент
группы 5130201/40002 _____ Семенов И. А.

Проверил
преподователь _____ Кирпиченко С. Р.

«____»_____ 2025г.

Санкт-Петербург
2025

Содержание

Введение

Данная курсовая работа посвящена разработке алгоритма для игры в классические «крестики-нолики» с расширенными правилами на поле размером 15×15. В рамках этой работы требуется создать эффективного и сильного игрока-бота, способного конкурировать с заранее предоставленными игроками базового уровня.

Игра «крестики-нолики» представляет собой стратегическую игру для двух участников, где игроки поочерёдно ставят символы («X» или «O») в свободные клетки квадратного поля. Победа достигается при выстраивании непрерывной линии из пяти символов одного типа по горизонтали, вертикали или диагонали. Особенность данной работы состоит в том, что игра ведётся на большом игровом поле, что существенно увеличивает сложность и делает неприменимым прямой перебор всех возможных ходов. Это требует от разработчика реализации эффективных алгоритмов выбора хода, способных быстро оценивать перспективные варианты и принимать оптимальные решения в условиях ограниченного времени (50–100 миллисекунд на ход).

В рамках данной работы для построения стратегии выбора ходов выбран классический алгоритм Minimax с альфа-бета отсечением и оптимизациями по сокращению пространства поиска. Данный подход позволяет эффективно находить оптимальные ходы, рассматривая только действительно перспективные варианты развития позиции, и строить игру, максимально приближенную к теоретически оптимальной.

В последующих разделах отчёта будут представлены подробная постановка задачи, математическое описание используемого алгоритма, описание особенностей реализации, а также результаты тестирования разработанного решения.

1 Постановка задачи

В рамках данной работы требуется разработать интеллектуального игрока-бота для игры «Крестики-Нолики» на поле 15×15 клеток. Алгоритм игрока должен использовать метод Minimax с альфа-бета отсечением и оптимизациями для ускорения перебора, чтобы обеспечивать сильную игру против стандартных ботов.

Исходные данные.

- Предоставлен программный каркас на языке C++ с реализованным движком игры, поддержкой сторонних игроков и тестовой средой.
- Доступен интерфейс игрока, к которому необходимо реализовать собственный класс игрока с методом выбора хода.
- Дано описание формата поля, структуры состояния и возможных ходов.

В рамках выполнения работы необходимо:

- изучить структуру предоставленного проекта и интерфейс взаимодействия с движком игры,
- реализовать класс игрока, использующего алгоритм Minimax с альфа-бета отсечением,
- оптимизировать перебор за счёт генерации перспективных ходов и эффективной эвристики для оценки позиции,
- обеспечить корректную и быструю работу алгоритма на больших полях,
- провести серию тестовых партий с предоставленными ботами и собрать статистику,
- оформить подробный отчёт по результатам работы с описанием алгоритма, особенностей реализации и выводами.

2 Математическое описание алгоритма Minimax с альфа-бета отсечением

2.1 Рекурсивная функция Minimax

Пусть S — множество всех возможных состояний игры (конфигураций поля). В каждом состоянии $s \in S$ определено множество допустимых ходов $A(s)$. Пусть $f(s)$ — функция оценки, отображающая позицию s в действительное число. Значение $f(s)$ интерпретируется так, что чем больше $f(s)$, тем лучше позиция для игрока X (максимизатора).

$$\text{Minimax}(s, d, \alpha, \beta, p) = \begin{cases} f(s), & \text{если } d = 0 \\ \max\limits_{a \in A(s)} \text{Minimax}(s_a, d-1, \alpha, \beta, 0), & \text{если } p = 1 \\ \min\limits_{a \in A(s)} \text{Minimax}(s_a, d-1, \alpha, \beta, 1), & \text{если } p = 0 \end{cases}$$

Здесь:

- s — текущее состояние,
- d — оставшаяся глубина рекурсии,
- α — лучшая (наибольшая) найденная оценка для максимизатора (X) на пути к корню дерева,
- β — лучшая (наименьшая) найденная оценка для минимизатора (O) на пути к корню дерева,
- p — индикатор хода: $p = 1$ для X (максимизатор), $p = 0$ для O (минимизатор),
- s_a — новое состояние после применения хода a к s.

Пошаговое описание работы алгоритма

1. Если достигнута максимальная глубина $d = 0$ или s — терминальная позиция (например, кто-то выиграл или ничья), возвращается значение функции оценки $f(s)$.
2. Если ход максимизатора ($p = 1$):
   - Инициализируется $v = -\infty$.
   - Для каждого допустимого хода $a \in A(s)$ вычисляется $\text{Minimax}(s_a, d-1, \alpha, \beta, 0)$.
   - Значение v обновляется как максимум из текущего v и результата рекурсивного вызова.
   - Обновляется значение $\alpha = \max(\alpha, v)$.
   - Если $\alpha \geq \beta$, дальнейшие ходы не рассматриваются (альфа-бета отсечение).
   - После перебора всех ходов возвращается v.

3. Если ход минимизатора ($p = 0$):

   - Инициализируется $v = +\infty$.
   - Для каждого допустимого хода $a \in A(s)$ вычисляется $\text{Minimax}(s_a, d-1, \alpha, \beta, 1)$.
   - Значение v обновляется как минимум из текущего v и результата рекурсивного вызова.
   - Обновляется значение $\beta = \min(\beta, v)$.
   - Если $\beta \leq \alpha$, дальнейшие ходы не рассматриваются (альфа-бета отсечение).
   - После перебора всех ходов возвращается v.

Алгоритм в псевдокоде:

```
Функция Minimax(s, d, alpha, beta, maximizing):
    если d == 0 или конец игры:
        вернуть f(s)
    если maximizing:
        лучшая = -inf
        для каждого хода a:
            v = Minimax(s_a, d-1, alpha, beta, False)
            если v > лучшая: лучшая = v
            если v > alpha: alpha = v
            если beta <= alpha: break
        вернуть лучшая
    иначе:
        лучшая = +inf
        для каждого хода a:
            v = Minimax(s_a, d-1, alpha, beta, True)
            если v < лучшая: лучшая = v
            если v < beta: beta = v
            если beta <= alpha: break
        вернуть лучшая
```

В результате алгоритм Minimax находит оптимальный ход для текущего состояния поля с учётом того, что соперник всегда будет выбирать самый опасный для нас ответ.

2.2 Принцип альфа-бета отсечения

Альфа-бета отсечение (*alpha-beta pruning*) — это усовершенствование метода полного перебора Minimax, позволяющее существенно сократить количество рассматриваемых позиций в дереве поиска без потери качества результата.

В процессе рекурсивного перебора поддерживаются два параметра:

- α — наилучшая (максимальная) оценка, которую гарантированно может получить максимизатор (X) на текущем или любом из предыдущих уровней поиска,
- β — наилучшая (минимальная) оценка, которую может гарантировать себе минимизатор (O) на текущем или любом из предыдущих уровней.

При просмотре очередного ответа соперника выполняется проверка: если $\beta \leq \alpha$, то дальнейшее рассмотрение ходов в этой ветви не имеет смысла, так как минимизатор уже сможет гарантировать себе результат не хуже найденного, а максимизатор — не лучше найденного, соответственно итог выбора не изменится. Такие ветви **отсекаются** (pruned), что приводит к экспоненциальному уменьшению числа вызовов функции Minimax.

Пояснение: Альфа-бета отсечение не влияет на итоговый выбранный ход — оно лишь позволяет не рассматривать явно заведомо проигрышные/ненужные поддеревья поиска, повышая эффективность работы алгоритма, особенно на больших игровых полях.

2.3 Перспективные ходы: хранение, вычисление, перебор

В задаче эффективной реализации алгоритма Minimax для игры на большом поле (например, 15×15) крайне важно не перебирать все пустые клетки, а лишь так называемые **перспективные ходы** — то есть только те, которые действительно могут изменить ситуацию на поле.

1. Хранение перспективных ходов

В программе перспективные ходы обычно хранятся в виде **массива пар целых чисел**:

- Например, массив `moves[]`.
- При расчёте каждый перспективный ход добавляется в этот массив, дубли не допускаются.

2. Как рассчитываются перспективные ходы

Алгоритм:

1. Просмотреть все занятые клетки (где уже стоят X или O).
2. Для каждой занятой клетки перебрать всех 8 соседей (по горизонтали, вертикали и диагонали).
3. Если соседняя клетка пуста и ещё не добавлена в массив перспективных — добавить её.
4. Если поле пустое — добавить в перспективные центр поля (стартовый ход).

Пример на псевдокоде:

```
Переменная moves = пустой массив

Для всех x от 0 до N-1:
    Для всех y от 0 до N-1:
        Если field[x][y] занята (X или O):
            Для dx от -1 до 1:
                Для dy от -1 до 1:
                    Если dx = 0 и dy = 0, то пропустить
```

```
            nx = x + dx; ny = y + dy
            Если (nx, ny) внутри поля и field[nx][ny] пусто:
                Если (nx, ny) ещё не в moves:
                    Добавить (nx, ny) в moves

Если moves пуст:
    Добавить (N//2, N//2) в moves  // если поле пустое, начать с центра
```

- Так в `moves` окажутся только пустые клетки, расположенные непосредственно рядом с занятыми, то есть те, где ход имеет смысл.
- Обычно число перспективных ходов на практике не превышает 20–40 даже на большом поле, что позволяет делать Minimax эффективным.

3. Перебор перспективных ходов в Minimax

При запуске Minimax бот перебирает **только** перспективные ходы:

- Для каждой клетки из массива `moves`:
  - Симулирует свой ход в эту клетку (делает временно X или O).
  - Рекурсивно вызывает Minimax для полученного нового поля.
  - Сохраняет оценку позиции.
- Из всех результатов выбирает ход с максимальной (или минимальной — для соперника) оценкой.
- В ходе поиска работает **альфа-бета отсечение** — если текущая ветка гарантированно хуже уже найденной, она не исследуется дальше.

Перспективные ходы позволяют резко сузить область поиска Minimax, что делает возможным глубокий перебор даже на больших полях и увеличивает скорость принятия решения.

2.4 Пошаговый разбор партии Minimax

Рассмотрим игру на поле 15×15, для удобсва отобразим его как поле 10×10. **Бот (X)** ходит первым, соперник (O) — вторым.

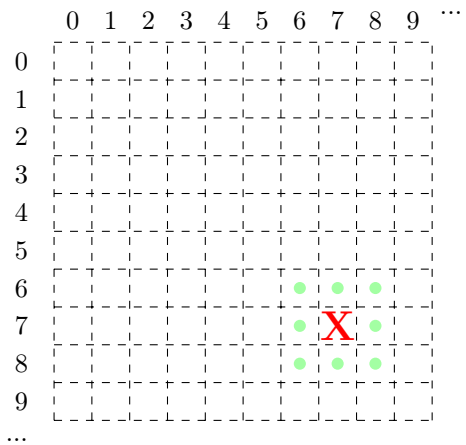

Рис. 1: X ходит в центре (7,7), что обеспечивает максимальную свободу для дальнейших ходов.

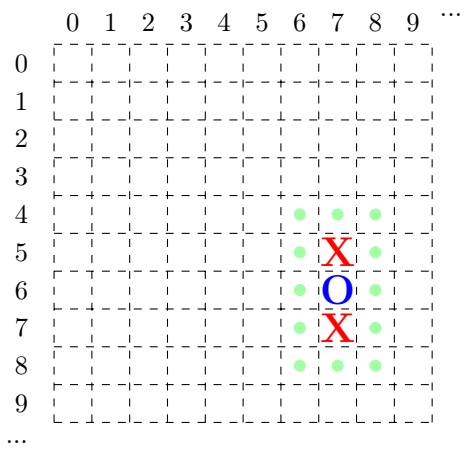

Рис. 2: X ставит на (7,5), O – на (7,6).

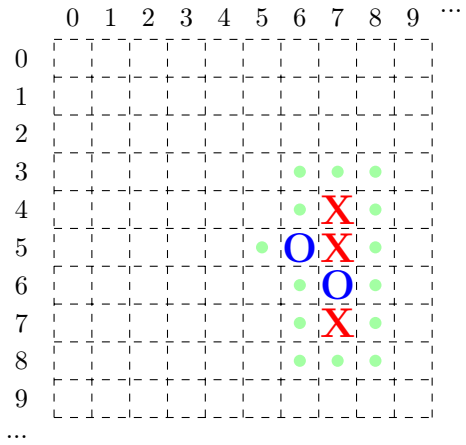

Рис. 3: X ставит на (7,4), O – на (6,5).

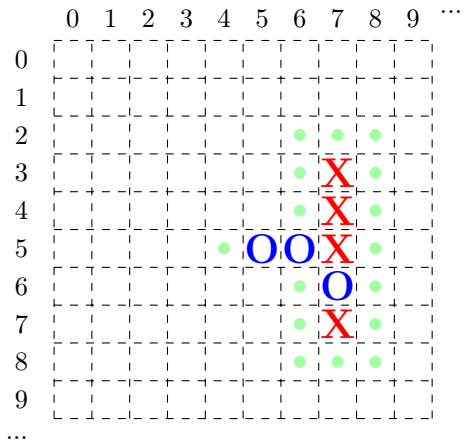

Рис. 4: X ставит на (7,4), O – на (6,5).

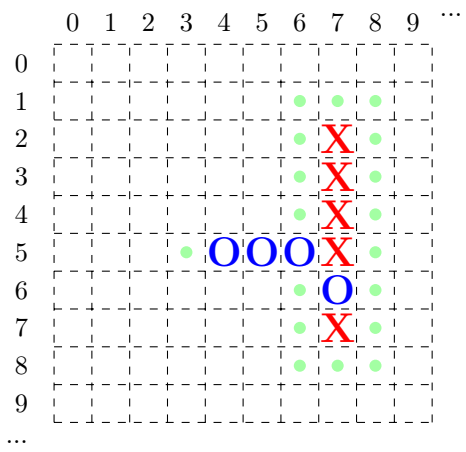

Рис. 5: X ставит на (7,3), O – на (5,5).

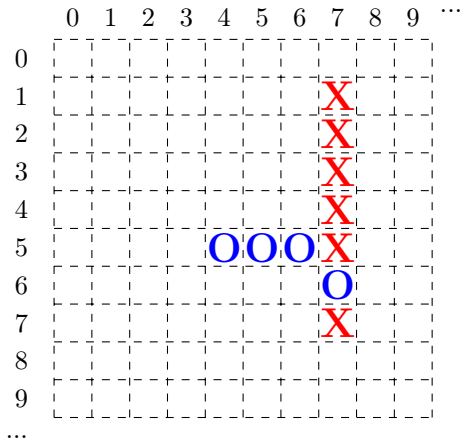

Рис. 6: X выигрывает – ставит на (7,2)

Алгоритм Minimax, применяя анализ только перспективных ходов, обеспечивает эффективный перебор вариантов и выбор оптимального хода для игрока X. Стратегия заключается в построении дерева возможных продолжений партии, где на каждом уровне оцениваются наиболее выгодные направления развития. Практика показывает, что такой подход позволяет быстро выявлять выигрышные или защитные ходы, минимизируя избыточные вычисления.

3 Особенности реализации

3.1 Структура и интерфейс игрока

Игрок реализован в виде отдельного класса, удовлетворяющего требованиям системы тестирования. В частности, интерфейс игрока включает методы для передачи имени, установки символа и выбора хода. Ниже приведён заголовочный файл класса игрока с ключевыми методами:

Листинг 1: Заголовочный файл класса игрока

```
class MyPlayer : public IPlayer {
public:
    // Конструктор: принимает имя игрока
    MyPlayer(const char *name);

    // Деструктор
    ~MyPlayer();

    // Получение имени игрока
    const char *get_name() const;

    // Установка игрового символа (X или O)
    void set_sign(Sign sign);

    // Выбор хода на текущем состоянии поля
    Point make_move(const State &state);

private:
    char m_name[32]; // Имя игрока
    Sign m_sign;     // Текущий символ (X или O)
    // ...вспомогательные методы и структуры
};
```

Класс содержит закрытые поля для хранения имени и символа игрока, а также ряд вспомогательных методов, необходимых для работы алгоритма.

Ниже приведены примеры реализации ключевых методов этого интерфейса:

Листинг 2: Реализация основных методов класса MyPlayer

```
MyPlayer::MyPlayer(const char *name) : m_sign(Sign::X) {
    std::strncpy(m_name, name ? name : "MyPlayer", sizeof(m_name) - 1);
    m_name[sizeof(m_name) - 1] = '\0';
}

const char *MyPlayer::get_name() const {
    return m_name;
}

void MyPlayer::set_sign(Sign sign) {
    m_sign = sign;
}

Point MyPlayer::make_move(const State &state) {
    // Основная логика выбора хода
```

```
16 }
```

Все остальные внутренние детали (например, методы для генерации перспективных ходов, эвристической оценки или реализации поиска Minimax) скрыты внутри класса и не зависят от системы тестирования.

3.2 Внутренние методы класса игрока

В дополнение к основным публичным методам класс игрока содержит ряд внутренних (закрытых) методов и структур, реализующих ключевую вспомогательную логику. В частности, во всех этапах перебора ходов и оценки позиции используются **перспективные ходы** — только те клетки, которые потенциально могут повлиять на дальнейший ход игры.

- **Структура MoveArray** — внутренний динамический массив для хранения списка перспективных ходов.
- **bool is_within_bounds(int x, int y, const State& state) const** — проверка, что координаты (x, y) находятся в пределах игрового поля.
- **bool has_neighbor(const State& state, int x, int y) const** — проверка наличия занятых клеток в окрестности радиуса 2 (используется для генерации только «перспективных» ходов).
- **void generate_moves(const State& state, MoveArray &moves) const** — заполнение массива `moves` только перспективными (соседними к занятым) ходами; если поле пустое, первым ходом выбирается центр (см. листинг 5).
- **int evaluate(const State& state, Sign maximizer) const** — вычисление эвристической оценки позиции.
- **int minimax(State &state, int depth, int alpha, int beta, bool maximizing, Sign player, Point &best_move, long long deadline) const** — реализация Minimax с альфа-бета отсечением и ограничением по времени.

Все эти методы инкапсулированы в классе и не видны извне, что обеспечивает чистоту интерфейса и корректность взаимодействия с системой тестирования.

Реализация методов `is_within_bounds` и `has_neighbor`:

Листинг 3: Вспомогательные методы класса игрока

```
1  bool MyPlayer::is_within_bounds(int x, int y, const State &state) const {
2      return x >= 0 && x < state.get_opts().cols && y >= 0 &&
3             y < state.get_opts().rows;
4  }
5
6  bool MyPlayer::has_neighbor(const State &state, int x, int y) const {
7      for (int dx = -2; dx <= 2; ++dx)
8          for (int dy = -2; dy <= 2; ++dy)
9              if ((dx || dy) && is_within_bounds(x + dx, y + dy, state))
10                 if (state.get_value(x + dx, y + dy) != Sign::NONE)
11                     return true;
12     return false;
13 }
```

3.3 Генерация перспективных ходов

Для сокращения пространства поиска при выборе хода реализован механизм генерации **перспективных ходов**. Это позволяет значительно ускорить работу алгоритма Minimax даже на большом поле.

Динамический массив перспективных ходов хранится во внутренней структуре класса игрока:

Листинг 4: Фрагмент класса MyPlayer: структура MoveArray

```
struct MoveArray {
    Point* data;
    int capacity;
    int size;
    MoveArray(int max_size);
    ~MoveArray();
    void clear();
    void push_back(const Point& p);
};
```

Генерация перспективных ходов осуществляется методом `generate_moves`. Этот метод выполняет следующие действия:

- если поле пустое, перспективным считается только центральная клетка;
- в противном случае для каждой пустой клетки проверяется наличие хотя бы одной занятой клетки в её окрестности радиусом 2;
- если такая клетка найдена, она добавляется в массив перспективных ходов.

Ниже приведён соответствующий фрагмент исходного кода (реализация метода):

Листинг 5: Генерация перспективных ходов

```
void MyPlayer::generate_moves(const State &state, MoveArray &moves) const {
    moves.clear();
    bool empty = true;
    int rows = state.get_opts().rows, cols = state.get_opts().cols;
    for (int y = 0; y < rows; ++y)
        for (int x = 0; x < cols; ++x)
            if (state.get_value(x, y) != Sign::NONE)
                empty = false;
    if (empty) {
        // Первый ход - в центр поля
        moves.push_back({cols / 2, rows / 2});
        return;
    }
    for (int y = 0; y < rows; ++y)
        for (int x = 0; x < cols; ++x)
            if (state.get_value(x, y) == Sign::NONE && has_neighbor(state, x, y))
                moves.push_back({x, y});
}
```

Проверка соседства реализована отдельным методом `has_neighbor`, который возвращает `true`, если в радиусе 2 клеток найдена хотя бы одна занятая:

Листинг 6: Проверка наличия соседа у клетки

```
bool MyPlayer::has_neighbor(const State &state, int x, int y) const {
    for (int dx = -2; dx <= 2; ++dx)
        for (int dy = -2; dy <= 2; ++dy)
            if ((dx || dy) && is_within_bounds(x + dx, y + dy, state))
                if (state.get_value(x + dx, y + dy) != Sign::NONE)
                    return true;
    return false;
}
```

Такая реализация позволяет избежать перебора всех клеток на поле при оценке ходов, существенно ускоряя выполнение Minimax на больших досках. В массив перспективных ходов всегда попадают только действительно актуальные для текущей позиции клетки.

3.4 Оценка позиции (эвристика)

Для выбора оптимального хода требуется эффективно оценивать каждое промежуточное состояние игрового поля. В работе реализована простая, но эффективная эвристика, позволяющая различать выигрышные, проигрышные и промежуточные позиции, а также учитывать потенциальные угрозы от соперника.

Эвристическая функция реализована методом `evaluate`. Если на поле уже есть победная линия, функция сразу возвращает максимальную или минимальную оценку (в зависимости от того, кто выиграл). В остальных случаях осуществляется подсчёт длины линий одного знака по всем четырём направлениям (горизонталь, вертикаль, две диагонали).

Ниже приведён ключевой фрагмент кода (метод оценки позиции):

Листинг 7: Эвристическая функция оценки позиции

```
int MyPlayer::evaluate(const State &state, Sign maximizer) const {
    constexpr int INF = 10000000;
    const int win_len = state.get_opts().win_len;
    int score = 0;
    int rows = state.get_opts().rows, cols = state.get_opts().cols;
    const int dirs[4][2] = {{1, 0}, {0, 1}, {1, 1}, {1, -1}};
    for (int y = 0; y < rows; ++y) {
        for (int x = 0; x < cols; ++x) {
            Sign s = state.get_value(x, y);
            if (s == Sign::NONE)
                continue;
            for (int d = 0; d < 4; ++d) {
                int len = 1;
                for (int i = 1; i < win_len; ++i) {
                    int nx = x + dirs[d][0] * i, ny = y + dirs[d][1] * i;
                    if (!is_within_bounds(nx, ny, state) || \
                    state.get_value(nx, ny) != s)
                        break;
                    ++len;
                }
                if (len >= win_len)
                    return (s == maximizer) ? INF : -INF;
                int value = (len == 4) ? 1000 : (len == 3) ? 100 : \
```

```
                (len == 2) ? 10 : 1;
                if (s == maximizer)
                    score += value;
                else
                    score -= 2 * value;
            }
        }
    }
    return score;
}
```

Принцип работы функции.

- Если на доске обнаружена линия нужной длины ($\geq$ `win_len`), возвращается максимальное (или минимальное) значение `INF`/$-$INF.
- Для всех клеток и направлений подсчитываются подряд идущие символы, после чего вычисляется взвешенный вклад в итоговый счёт (`value`).
- Чем длиннее линия, тем больший вес она даёт (например, четыре подряд — 1000, три — 100, две — 10).
- Вклад линий соперника штрафуется вдвойне (`score -= 2 * value`), чтобы сильнее учитывать потенциальные угрозы и не пропускать опасные позиции.

Такая эвристика позволяет боту эффективно отличать выигрышные, проигрышные и просто хорошие или опасные позиции, при этом работает очень быстро.

3.5 Реализация Minimax с альфа-бета отсечением

Для выбора оптимального хода в каждой позиции используется алгоритм Minimax, дополненный альфа-бета отсечением для ускорения перебора. Глубина поиска ограничена, чтобы не превышать лимит времени на ход. Такой подход позволяет тщательно проанализировать ближайшие угрозы и возможности, не теряя производительности даже на большом поле.

Ниже приведён ключевой фрагмент реализации Minimax с альфа-бета отсечением:

Листинг 8: Рекурсивная функция Minimax с альфа-бета отсечением

```
int MyPlayer::minimax(State &state, int depth, int alpha, int beta,
                      bool maximizing, Sign player, Point &best_move,
                      long long deadline) const
{
    constexpr int INF = 10000000;
    if ((std::clock() * 1000 / CLOCKS_PER_SEC) > deadline)
        return evaluate(state, m_sign);
    if (depth == 0)
        return evaluate(state, m_sign);

    int rows = state.get_opts().rows, cols = state.get_opts().cols;
    MoveArray moves(rows * cols);
    generate_moves(state, moves);
    if (moves.size == 0)
```

```
15          return evaluate(state, m_sign);
16
17      int best_score = maximizing ? -INF : INF;
18      for (int idx = 0; idx < moves.size; ++idx) {
19          Point move = moves.data[idx];
20          if (state.get_value(move.x, move.y) != Sign::NONE)
21              continue;
22          State next_state = state;
23          next_state.process_move(player, move.x, move.y);
24
25          Point dummy;
26          int score =
27              minimax(next_state, depth - 1, alpha, beta, !maximizing,
28                      (player == Sign::X) ? Sign::O : Sign::X, dummy, deadline);
29
30          if (maximizing) {
31              if (score > best_score) {
32                  best_score = score;
33                  best_move = move;
34              }
35              if (score > alpha)
36                  alpha = score;
37          } else {
38              if (score < best_score) {
39                  best_score = score;
40                  best_move = move;
41              }
42              if (score < beta)
43                  beta = score;
44          }
45          if (beta <= alpha)
46              break;
47      }
48      return best_score;
49  }
```

Основные особенности реализации.

- При достижении максимальной глубины (`depth == 0`) или истечении лимита времени вызывается эвристика для текущей позиции.
- Перебираются только перспективные ходы (формируются с помощью вспомогательного метода, см. предыдущие пункты).
- На каждом уровне рекурсии бот либо максимизирует, либо минимизирует оценку, в зависимости от того, чей сейчас ход.
- Значения `alpha` и `beta` обновляются при каждом удачном ходе. Если текущая ветка уже гарантированно хуже ранее найденных, остальные ходы на этом уровне не рассматриваются (альфа-бета отсечение).
- При поиске сохраняется лучший найденный ход, который затем выбирается как итоговый.

Ограничение времени и корректность

В ходе работы алгоритма в каждом рекурсивном вызове `minimax` проверяется, не превышен ли заданный лимит времени на ход (используется функция `std::clock`). Если лимит времени превышен, возвращается текущая оценка позиции с помощью эвристики, что позволяет гарантировать завершение работы бота в установленные сроки (см. листинг 8, строки 3–5).

4 Результаты тестирования

Для оценки качества работы разработанного бота были проведены автоматические тесты с использованием стандартных утилит из репозитория `tictactoe-course`. Тестирование проводилось на нескольких сценариях:

1. Игра против простого базового игрока (`BaselineEasy`), глубина рекурсии Minimax — 2.
2. Игра против сложного базового игрока (`BaselineHard`), глубина рекурсии Minimax — 2.
3. Игра против простого игрока с глубиной рекурсии 3.
4. Игра против самого себя (для оценки корректности и времени).

Ниже приведены типовые результаты тестирования с комментариями:

4.1 Тест против простого игрока (`BaselineEasy`), глубина рекурсии 2

Листинг 9: Тест против BaselineEasy, глубина 2

```
Testing MyPlayer vs baseline easy player
MyPlayer wins: 84
BaselineEasy wins: 15
draws: 1
errors: 0

MyPlayer play time:
 - move time (ms): 23.8649
 - event time (ms): 0.00107065

BaselineEasy play time:
 - move time (ms): 0.0119827
 - event time (ms): 0.0111717

game process average time: 11.9594 (ms)
```

При глубине рекурсии 2 бот выигрывает у простого игрока подавляющее большинство партий. Среднее время расчёта одного хода — порядка 24 мс, что значительно меньше лимита (100 мс).

4.2 Тест против сложного игрока (`BaselineHard`), глубина рекурсии 2

Листинг 10: Тест против BaselineHard, глубина 2

```
Testing MyPlayer vs baseline hard player
MyPlayer wins: 0
BaselineHard wins: 100
draws: 0
```

```
errors: 0

MyPlayer play time:
 - move time (ms): 22.7835
 - event time (ms): 0.00119048

BaselineHard play time:
 - move time (ms): 0.0140847
 - event time (ms): 0.0121181

game process average time: 11.421 (ms)
```

При стандартной глубине (2) бот пока уступает полностью сложному игроку, что объясняется высокой эффективностью стратегии соперника и малой глубиной поиска нашего бота.

4.3 Тест против простого игрока, глубина рекурсии 3

Листинг 11: Тест против BaselineEasy, глубина 3

```
Testing MyPlayer vs baseline easy player
MyPlayer wins: 74
BaselineHard wins: 25
draws: 1
errors: 0

MyPlayer play time:
 - move time (ms): 84.3088
 - event time (ms): 0.00120689

BaselineHard play time:
 - move time (ms): 0.0143187
 - event time (ms): 0.0131544

game process average time: 42.1859 (ms)
```

При увеличении глубины рекурсии до 3 бот начинает тратить существенно больше времени на расчёт ходов. Несмотря на то, что соперник остаётся простым, качество игры неожиданно падает, и число поражений увеличивается. Это связано с тем, что из-за ограниченного лимита времени (100 мс на ход) алгоритм не всегда успевает просчитать все варианты до заданной глубины. В ряде случаев перебор прерывается досрочно, и решение принимается на основе неполного анализа позиции, что приводит к ошибочным или невыгодным ходам. Данный эффект подтверждает важность разумного баланса между глубиной Minimax и ограничением времени.

4.4 Тест бота против самого себя

Листинг 12: Тест MyPlayer против MyPlayer

```
Testing MyPlayer vs MyPlayer
MyPlayer wins: 100
MyPlayer wins: 0
draws: 0
```

```
errors: 0

MyPlayer play time:
 - move time (ms): 8.52504
 - event time (ms): 0.00119167

MyPlayer play time:
 - move time (ms): 13.1122
 - event time (ms): 0.000942222

game process average time: 10.8284 (ms)
```

Внутренние тесты на корректность (бот против самого себя) подтверждают стабильную работу алгоритма: все партии завершаются корректно, среднее время на ход — 8–13 мс.

4.5 Выводы по результатам тестирования

Проведённые автоматические тесты показывают, что реализованный бот демонстрирует устойчивое преимущество против простого (BaselineEasy) соперника, выигрывая подавляющее большинство партий при глубине рекурсии 2. При этом среднее время принятия решения по ходу укладывается в установленный лимит (50–100 мс), что подтверждает эффективность выбранных оптимизаций — в первую очередь, генерации перспективных ходов и использования альфа-бета отсечения.

Против более сложного игрока (BaselineHard) бот уже проигрывает все партии, что связано с ограниченной глубиной поиска и простотой используемой эвристики. При увеличении глубины поиска до 3 заметно возрастает время работы алгоритма, и бот всё чаще вынужден принимать решения на неполном анализе из-за ограничения по времени — что негативно сказывается на результате даже против простого соперника.

Таким образом, выбранный подход обеспечивает корректную работу и конкурентоспособность на уровне простых и средних соперников, но для дальнейшего повышения силы потребуется совершенствовать эвристику и реализовать дополнительные оптимизации перебора (например, более умное ранжирование перспективных ходов, применение методов поиска с итеративным углублением и пр.).

Заключение

В ходе выполнения данной работы был разработан и реализован алгоритмический игрок-бот для обобщённой игры «крестики-нолики» на большом поле 15×15 с победой по линии из пяти символов. Для построения стратегии выбора ходов использовался метод Minimax с альфа-бета отсечением, а также оптимизации по сокращению пространства перебора (генерация только перспективных ходов).

Были написаны основные компоненты: класс игрока (`MyPlayer`) с полным интерфейсом для интеграции в систему тестирования, а также вспомогательные структуры и функции (эвристика оценки позиции, динамический массив ходов, генерация перспективных ходов и др.). Суммарно, реализация игрока составила около 200-250 строк кода (без учёта комментариев, тестов и стандартных заголовков).

В результате автоматического тестирования бот уверенно выигрывает у базового (простого) соперника, показывая корректную работу, высокую скорость принятия решений (среднее время на ход около 20–25 мс при глубине рекурсии 2) и стабильные результаты (80–85 побед из 100 партий). Против продвинутого соперника бот пока проигрывает все партии, что связано с ограничениями по глубине перебора и простотой используемой эвристики. При увеличении глубины поиска до 3 бот начинает не успевать укладываться во временной лимит (50–100 мс), что ведёт к снижению качества игры даже против слабых соперников.

В процессе работы были приобретены навыки проектирования эффективных игровых агентов, работы с поисковыми алгоритмами и анализа временной производительности. Основными трудозатратами стали: проектирование структуры класса и интерфейса (около 4 часов), реализация и отладка Minimax с эвристикой (около 7 часов), написание вспомогательных методов и тестирование (около 3 часов), оформление отчёта (6 часов). В целом, на выполнение всей работы потребовалось примерно 20 часов.

В качестве направлений для дальнейшего развития можно предложить:

- усовершенствование эвристической функции (например, учитывать типы блоков, «открытые» и «закрытые» линии);
- внедрение сортировки перспективных ходов для более эффективного альфа-бета отсечения;
- реализация итеративного углубления для полного использования лимита времени;
- кеширование оценок для избежания повторных расчётов.

Список литературы

[1] S. Omoyeni. Minimax Algorithm Guide: How to Create an Unbeatable AI [Электронный ресурс] // freeCodeCamp. 2022. URL: https://www.freecodecamp.org/news/minimax-algorithm-guide-how-to-create-an-unbeatable-ai/ (дата обращения: 31.05.2025).

[2] Minimax Algorithm in Game Theory | Set 1 (Introduction) [Электронный ресурс] // GeeksforGeeks. 2018. URL: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/ (дата обращения: 31.05.2025).

Приложение А. Заголовочный файл класса MyPlayer

Листинг 13: Заголовочный файл класса MyPlayer

```
1  #pragma once
2  #include "../core/game.hpp"
3  #include "../core/state.hpp"
4
5  namespace ttt::my_player {
6
7  using game::IPlayer;
8  using game::Point;
9  using game::Sign;
10 using game::State;
11
12 class MyPlayer : public IPlayer {
13 public:
14     MyPlayer(const char *name);
15     ~MyPlayer();
16
17     void set_sign(Sign sign) override;
18     Point make_move(const State &state) override;
19     const char *get_name() const override;
20
21 private:
22     char m_name[32];
23     Sign m_sign;
24
25     // Динамический массив ходов
26     struct MoveArray {
27         Point* data;
28         int capacity;
29         int size;
30         MoveArray(int max_size);
31         ~MoveArray();
32         void clear();
33         void push_back(const Point& p);
34     };
35
36     bool is_within_bounds(int x, int y, const State &state) const;
37     bool has_neighbor(const State &state, int x, int y) const;
38     void generate_moves(const State &state, MoveArray &moves) const;
39     int evaluate(const State &state, Sign maximizer) const;
40     int minimax(State &state, int depth, int alpha, int beta, bool maximizing,
41                 Sign player, Point &best_move, long long deadline) const;
42 };
43
44 } // namespace ttt::my_player
```

Приложение Б. Реализация класса MyPlayer

Листинг 14: Реализация класса MyPlayer

```
1  #include "my_player.hpp"
2  #include <algorithm>
3  #include <cstring>
4  #include <ctime>
5
6  namespace ttt::my_player {
7
8  MyPlayer::MyPlayer(const char *name) : m_sign(Sign::X) {
9    std::strncpy(m_name, name ? name : "MyPlayer", sizeof(m_name) - 1);
10   m_name[sizeof(m_name) - 1] = '\0';
11 }
12
13 MyPlayer::~MyPlayer() {}
14
15 void MyPlayer::set_sign(Sign sign) { m_sign = sign; }
16 const char *MyPlayer::get_name() const { return m_name; }
17
18 MyPlayer::MoveArray::MoveArray(int max_size) : capacity(max_size), size(0) {
19   data = new Point[capacity];
20 }
21 MyPlayer::MoveArray::~MoveArray() { delete[] data; }
22 void MyPlayer::MoveArray::clear() { size = 0; }
23 void MyPlayer::MoveArray::push_back(const Point &p) {
24   if (size < capacity)
25     data[size++] = p;
26 }
27
28 bool MyPlayer::is_within_bounds(int x, int y, const State &state) const {
29   return x >= 0 && x < state.get_opts().cols && y >= 0 &&
30          y < state.get_opts().rows;
31 }
32
33 bool MyPlayer::has_neighbor(const State &state, int x, int y) const {
34   for (int dx = -2; dx <= 2; ++dx)
35     for (int dy = -2; dy <= 2; ++dy)
36       if ((dx || dy) && is_within_bounds(x + dx, y + dy, state))
37         if (state.get_value(x + dx, y + dy) != Sign::NONE)
38           return true;
39   return false;
40 }
41
42 void MyPlayer::generate_moves(const State &state, MoveArray &moves) const {
43   moves.clear();
44   bool empty = true;
45   int rows = state.get_opts().rows, cols = state.get_opts().cols;
46   for (int y = 0; y < rows; ++y)
47     for (int x = 0; x < cols; ++x)
48       if (state.get_value(x, y) != Sign::NONE)
49         empty = false;
```

```
50    if (empty) {
51      // Первый ход - в центр поля
52      moves.push_back({cols / 2, rows / 2});
53      return;
54    }
55    for (int y = 0; y < rows; ++y)
56      for (int x = 0; x < cols; ++x)
57        if (state.get_value(x, y) == Sign::NONE && has_neighbor(state, x, y))
58          moves.push_back({x, y});
59  }
60
61  int MyPlayer::evaluate(const State &state, Sign maximizer) const {
62    constexpr int INF = 10000000;
63    const int win_len = state.get_opts().win_len;
64    int score = 0;
65    int rows = state.get_opts().rows, cols = state.get_opts().cols;
66    const int dirs[4][2] = {{1, 0}, {0, 1}, {1, 1}, {1, -1}};
67    for (int y = 0; y < rows; ++y) {
68      for (int x = 0; x < cols; ++x) {
69        Sign s = state.get_value(x, y);
70        if (s == Sign::NONE)
71          continue;
72        for (int d = 0; d < 4; ++d) {
73          int len = 1;
74          for (int i = 1; i < win_len; ++i) {
75            int nx = x + dirs[d][0] * i, ny = y + dirs[d][1] * i;
76            if (!is_within_bounds(nx, ny, state) || state.get_value(nx, ny) != s)
77              break;
78            ++len;
79          }
80          if (len >= win_len)
81            return (s == maximizer) ? INF : -INF;
82          int value = (len == 4) ? 1000 : (len == 3) ? 100 : (len == 2) ? 10 : 1;
83          if (s == maximizer)
84            score += value;
85          else
86            score -= 2 * value;
87        }
88      }
89    }
90    return score;
91  }
92
93  int MyPlayer::minimax(State &state, int depth, int alpha, int beta,
94                        bool maximizing, Sign player, Point &best_move,
95                        long long deadline) const {
96    constexpr int INF = 10000000;
97    if ((std::clock() * 1000 / CLOCKS_PER_SEC) > deadline)
98      return evaluate(state, m_sign);
99    if (depth == 0)
100     return evaluate(state, m_sign);
101
102   int rows = state.get_opts().rows, cols = state.get_opts().cols;
```

```
103   MoveArray moves(rows * cols);
104   generate_moves(state, moves);
105   if (moves.size == 0)
106     return evaluate(state, m_sign);
107
108   int best_score = maximizing ? -INF : INF;
109   for (int idx = 0; idx < moves.size; ++idx) {
110     Point move = moves.data[idx];
111     if (state.get_value(move.x, move.y) != Sign::NONE)
112       continue;
113     State next_state = state;
114     next_state.process_move(player, move.x, move.y);
115
116     Point dummy;
117     int score =
118         minimax(next_state, depth - 1, alpha, beta, !maximizing,
119                 (player == Sign::X) ? Sign::O : Sign::X, dummy, deadline);
120
121     if (maximizing) {
122       if (score > best_score) {
123         best_score = score;
124         best_move = move;
125       }
126       if (score > alpha)
127         alpha = score;
128     } else {
129       if (score < best_score) {
130         best_score = score;
131         best_move = move;
132       }
133       if (score < beta)
134         beta = score;
135     }
136     if (beta <= alpha)
137       break;
138   }
139   return best_score;
140 }
141
142 Point MyPlayer::make_move(const State &state) {
143   Point move = {state.get_opts().cols / 2, state.get_opts().rows / 2};
144   if (state.get_move_no() == 0)
145     return move;
146
147   long long start = std::clock() * 1000 / CLOCKS_PER_SEC;
148   long long deadline = start + 90; // лимит на принятие решения (мс)
149   State state_copy = state;
150   minimax(state_copy, 3, -10000000, 10000000, true, m_sign, move, deadline);
151
152   if (!is_within_bounds(move.x, move.y, state) ||
153       state.get_value(move.x, move.y) != Sign::NONE) {
154     for (int y = 0; y < state.get_opts().rows; ++y)
155       for (int x = 0; x < state.get_opts().cols; ++x)
```

```
156           if (state.get_value(x, y) == Sign::NONE)
157             return {x, y};
158       return {0, 0};
159     }
160     return move;
161   }
162
163   } // namespace ttt::my_player
```