МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический
университет Петра Великого»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

КУРСОВАЯ РАБОТА
ПО ДИСЦИПЛИНЕ АЛГОРИТМИЗАЦИЯ И ПРОГРАММИРОВАНИЕ

на тему: "Разработка алгоритма принятия решений для игры в «Крестики-нолики» для пяти в ряд”

Отчет выполнил
студент группы №5130201/40001 _____ Шемякина Е. К.

Отчет принял
преподаватель _____ Сеннов В. Н.

«____» _____2025 г.

Санкт-Петербург, 2025

Содержание

Введение

Крестики-нолики — одна из самых известных и популярных настольных игр, которая сочетает в себе простоту правил и глубину стратегического мышления. Несмотря на кажущуюся элементарность, игра представляет значительный интерес с точки зрения алгоритмизации и искусственного интеллекта, особенно в модификациях с увеличенным полем и количеством символов для победы. Разработка алгоритмов, способных эффективно играть в такие вариации игры, является актуальной задачей, находящей применение в области теории игр, машинного обучения и разработки интеллектуальных систем.

Целью данной курсовой работы является разработка и реализация алгоритма для бота, способного играть в крестики-нолики на поле размером от 15×15 и более с условием победы при составлении линии из пяти или более символов. Алгоритм должен быть не только эффективным с точки зрения времени принятия решений, но и достаточно сильным, чтобы побеждать базового игрока, предоставленного в рамках проекта.

Работа представляет интерес для студентов, изучающих программирование и алгоритмы, так как позволяет применить теоретические знания на практике, развить навыки проектирования и оптимизации программных решений. Кроме того, результаты работы могут быть использованы в образовательных целях, для проведения турниров между алгоритмами, а также как основа для более сложных систем искусственного интеллекта.

1 Постановка задачи

В рамках курсовой работы требуется разработать и реализовать алгоритм для бота, играющего в крестики-нолики на поле размером MxN (M, N $\geq$ 15) с условием победы при построении линии из L = 5 одинаковых символов по горизонтали, вертикали или диагонали.

Исходные данные и условия:

1. Предоставлен каркас проекта на C++17, включающий:
   - Реализацию логики игры (библиотека tttcore), классы State и Game для управления состоянием и процессом игры.
   - Интерфейс IPlayer, который должен быть реализован разрабатываемым алгоритмом.
   - Интерфейс IObserver для получения событий из игры.
   - Исполняемые модули для тестирования (tests) и игры по сети (tttremote).
   - Закрытую библиотеку (libtttcore.a) с реализацией базового игрока ("baseline player") разумного, но не оптимального уровня, который будет использоваться как основной оппонент для тестирования.
2. Алгоритм должен быть реализован в виде класса-наследника IPlayer и интегрирован в проект. Класс должен корректно работать с предоставленным API:
   - Получать свой знак (X или O) через метод set_sign.
   - Принимать текущее состояние игры (const State &) в методе make_move.
   - Возвращать координаты (row, col) следующего хода, которые должны находиться в пределах игрового поля и указывать на свободную клетку.
3. К алгоритму предъявляются следующие требования:
   - Эффективность: Время вычисления одного хода не должно превышать 50 мс на стандартной рабочей станции.
   - Сила игры: Разработанный бот должен стабильно обыгрывать предоставленного базового игрока (baseline player) в большинстве партий (за обе стороны — и за крестики, и за нолики).
   - Универсальность: Алгоритм должен быть применим для полей разного размера (начиная от 15x15) и корректно обрабатывать все правила игры, включая предоставление последнего хода ноликам в случае потенциальной победы крестиков.
4. Критерии успешности выполнения работы:

- Алгоритм реализован, интегрирован в проект и проходит все предусмотренные тесты.
- Реализация побеждает базового бота в более чем 50% случаев в сериях игр (например, 100 партий за крестики и 100 за нолики).
- Время принятия решений укладывается в заданные ограничения.
- Составлен подробный отчет, описывающий алгоритм, его реализацию и результаты тестирования.

2 Математическое описание

2.1 Модель игры

Игровое поле представляет собой матрицу размера MxN (M, N $\geq$ 15). Каждая клетка может находиться в одном из трех состояний: пустая, занятая крестиком (X) или ноликом (O). Выигрышное состояние - непрерывная последовательность из L=5 одинаковых символов по горизонтали, вертикали или диагонали.

2.2 Описание алгоритма

Алгоритм представляет собой систему правил, которые последовательно применяются к текущему состоянию поля для выбора наилучшей клетки для хода. Правила расположены в порядке убывания приоритета: как только находится ход, удовлетворяющий правилу, последующие правила не проверяются.

Правило 1: Немедленная победа.

- Цель: Закончить игру своим выигрышем.
- Действие: Просканировать все пустые клетки на поле. Для каждой пустой клетки мысленно поставить в нее свой символ. Проверить, образует ли эта установка символа где-либо на поле линию длиной 5 или более своих символов. Если такая клетка найдена, выбрать ее для хода.

Правило 2: Немедленная блокировка.

- Цель: Не позволить противнику выиграть на следующем ходу.
- Действие: Просканировать все пустые клетки на поле. Для каждой пустой клетки мысленно поставить в нее символ противника. Проверить, образует ли эта установка символа противника где-либо на поле линию длиной 5 или более его символов. Если такая клетка найдена, выбрать ее для хода (чтобы заблокировать победу противника).

Правило 3: Создание двойной угрозы.

- Цель: Создать ситуацию, при которой противник не сможет предотвратить поражение на следующий ход, так как ему придется блокировать две угрозы одновременно.
- Действие: Просканировать все пустые клетки на поле. Для каждой пустой клетки мысленно поставить в нее свой символ. Далее, необходимо оценить, создает ли этот ход две или более отдельные угрозы.

- Определение угрозы: Угроза — это такая конфигурация символов на поле, при которой на следующем ходу игрок гарантированно может завершить линию из 5 символов. Простейший пример — незаблокированная последовательность из четырех своих символов, где с одной или двух сторон есть свободные клетки для пятого.

- Вывод: Если находится клетка, установка в которую своего символа создает две или более независимые угрозы, выбрать эту клетку для хода. Противник, способный блокировать только одну угрозу за ход, проиграет.

Правило 4: Блокировка сильной атаки противника.

- Цель: Заблокировать потенциально выигрышную последовательность противника до того, как она превратится в немедленную угрозу (как в Правиле 2).

- Действие: Искать на поле конфигурации символов противника, которые являются "опасными". Например, не заблокированная последовательность из трех его символов, с обеих сторон от которой есть свободные клетки. Установка своего символа в одну из ключевых клеток, мешающих развитию этой последовательности, считается сильным блокирующим ходом.

Правило 5: Развитие собственной атаки.

- Цель: Усилить свои позиции, создавая новые или продлевая существующие перспективные последовательности.

- Действие: Найти на поле все свои самые длинные незаблокированные последовательности (например, из 2 или 3 символов). Для каждой такой последовательности определить пустые клетки на ее продолжении. Выбрать ход в одну из этих клеток, отдавая предпочтение тем, которые продлевают самые длинные последовательности или находятся в центре.

Правило 6: Стратегический выбор.

- Цель: Захватить наиболее выгодные позиции в начале игры или при отсутствии явных тактических вариантов.

- Действие: Если предыдущие правила не сработали (например, в самом начале игры), выбирать клетки согласно заранее заданному приоритету. Наивысший приоритет имеют клетки, расположенные ближе к центру поля, так как они предоставляют максимальное количество возможностей для построения линий во всех направлениях. Используется концепция "колец"вокруг центральной точки поля.

Правило 7: Случайный выбор.

- Цель: Сделать корректный ход в ситуации, когда все вышеперечисленные стратегические и тактические соображения неприменимы (крайне редкий случай).
- Действие: Выбрать любую свободную клетку на поле случайным образом. Это гарантирует, что алгоритм всегда сможет сделать ход.

3 Особенности реализации

1. Структуры данных и внутреннее представление состояния

   Для эффективного анализа игровой ситуации реализовано внутреннее представление игрового поля в виде двумерного массива m_board типа Sign**. Размеры поля хранятся в переменных m_rows и m_cols. Данная структура была выбрана для обеспечения быстрого произвольного доступа к клеткам поля в процессе анализа.

   Инициализация и обновление доски происходит в методах:

   - initialize_board(const State& state) - выделяет память и инициализирует массив значениями Sign::NONE.
   - update_board(const State& state) - синхронизирует внутреннее представление с текущим состоянием игры.

2. Реализация алгоритма принятия решений

   Основная логика алгоритма реализована в методе make_move(const State &state), который последовательно применяет правила выбора хода в порядке убывания приоритета. Функция приведена в листинге 1.

```
1         Point MyPlayer::make_move( const State &state )
2         {
3             Point result;
4
5             if (state.get_move_no() == 0)
6             {
7                 result.x = state.get_opts().cols / 2;
8                 result.y = state.get_opts().rows / 2;
9                 return result;
10            }
11
12            update_board(state);
13
14            Sign enemy_sign = (m_sign == Sign::X) ?
                  Sign::O : Sign::X;
15
16            if (find_win_move(result, m_sign)) return
                  result;
17            if (find_block_move(result, enemy_sign, 4))
                  return result;
18            if (find_double_threat(result, m_sign)) return
                  result;
19            if (find_block_move(result, enemy_sign, 3))
                  return result;
20            if (find_strategic_move(result, m_sign))
                  return result;
21
22            for (int attempt = 0; attempt < 50; attempt++)
23            {
```

```
                result.x = std::rand() % m_cols;
                result.y = std::rand() % m_rows;

                if (m_board[result.y][result.x] ==
                   Sign::NONE)
                {
                    for (int dy = -1; dy <= 1; dy++)
                    {
                        for (int dx = -1; dx <= 1; dx++)
                        {
                            if (dx == 0 && dy == 0)
                            continue;
                            int nx = result.x + dx;
                            int ny = result.y + dy;
                            if (nx >= 0 && nx < m_cols &&
                               ny >= 0 && ny < m_rows &&
                               m_board[ny][nx] !=
                               Sign::NONE)
                            return result;
                        }
                    }
                }
            }

            if (find_any_move(result))
            return result;

            result.x = 0;
            result.y = 0;
            return result;
        }
```

Листинг 1: Функция выбора хода

3. Ключевые вспомогательные методы Анализ линий: Метод check_line(int x, int y, int dx, int dy, Sign sign) проверяет потенциальную линию в двух направлениях от заданной точки. Код функции представлен в листинге 2.

```
        int MyPlayer::check_line( int x, int y, int dx,
           int dy, Sign sign ) const
        {
            int count = 0;
            int max_count = 0;

            for (int dir = -1; dir <= 1; dir += 2)
            {
                count = 0;
                for (int i = 1; i <= 4; i++)
                {
                    int nx = x + i * dx * dir;
                    int ny = y + i * dy * dir;

```

```
                    if (nx < 0 || nx >= m_cols || ny < 0
                        || ny >= m_rows) break;
                    if (m_board[ny][nx] == sign) count++;
                    else if (m_board[ny][nx] ==
                        Sign::NONE) break;
                    else { count = 0; break; }
                }
                max_count += count;
            }
            return max_count;
        }
```

Листинг 2: Функция анализа линий

Поиск выигрышного хода: find_win_move ищет ход, завершающий линию из 5 символов. Код функции представлен в листинге 3.

```
        bool MyPlayer::find_win_move( Point &result, Sign
            sign ) const
        {
            for (int y = 0; y < m_rows; y++)
            {
                for (int x = 0; x < m_cols; x++)
                {
                    if (m_board[y][x] != Sign::NONE)
                        continue;

                    int directions[4][2] = {{1, 0}, {0,
                        1}, {1, 1}, {1, -1}};

                    for (int d = 0; d < 4; d++)
                    {
                        if (check_line(x, y,
                            directions[d][0],
                            directions[d][1], sign) >= 4)
                        {
                            result.x = x; result.y = y;
                            return true;
                        }
                    }
                }
            }
            return false;
        }
```

Листинг 3: Функция поиска выигрышного кода

Поиск двойной угрозы: find_double_threat реализует ключевую идею алгоритма. Код функции представлен в листинге 4.

```
        bool MyPlayer::find_double_threat(Point& result,
            Sign sign) const
        {
            for (int y = 0; y < m_rows; y++)
```

```
4               {
5                   for (int x = 0; x < m_cols; x++)
6                   {
7                       if (m_board[y][x] != Sign::NONE)
                            continue;
8
9                       int threat_count = 0;
10                      int directions[4][2] = {{1, 0}, {0,
                            1}, {1, 1}, {1, -1}};
11
12                      for (int d = 0; d < 4; d++)
13                      {
14                          if (check_line(x, y,
                                directions[d][0],
                                directions[d][1], sign) >= 2)
15                          {
16                              threat_count++;
17                              if (threat_count >= 2)
18                              {
19                                  result.x = x; result.y = y;
20                                  return true;
21                              }
22                          }
23                      }
24                  }
25              }
26              return false;
27          }
```

Листинг 4: Функция поиска двойной угрозы

4. Генерация случайных чисел

   Для случайного выбора хода используется стандартная функция std::rand(). В цикле осуществляется до 50 попыток найти подходящую случайную клетку, предпочтительно рядом с уже занятыми клетками.

5. Программный интерфейс

   Класс MyPlayer наследует интерфейс IPlayer и реализует:

   - set_sign(Sign sign) - установка знака игрока.
   - make_move(const State &game) - основной метод выбора хода.
   - get_name() const - возврат имени игрока.

   Приватные методы обеспечивают модульность реализации различных аспектов алгоритма, что упрощает тестирование и модификацию.

6. Оптимизации

   Для обеспечения производительности реализованы:

- локальное кэширование состояния поля,
- поиск только в релевантных направлениях (4 основных направления),
- ранний выход из циклов при нахождении подходящего хода,
- ограничение количества проверяемых случайных клеток (50 попыток).

4 Результаты работы программы

Для всесторонней оценки эффективности разработанного алгоритма были проведены серии тестовых партий, состоящие из 1000 игр, в различных конфигурациях. Тестирование проводилось на одном и том же устройстве с процессором Intel Core i5-11400H.

1. Тестирование против простого игрока Простой игрок выбирает клетку для хода случайным образом, поэтому бот обыгрывает игрока в 100% случаев.

2. Тестирование против базового игрока

   Было проведено две серии тестов по 1000 игр. На рис. 1 и 2 представлены результаты тестов.

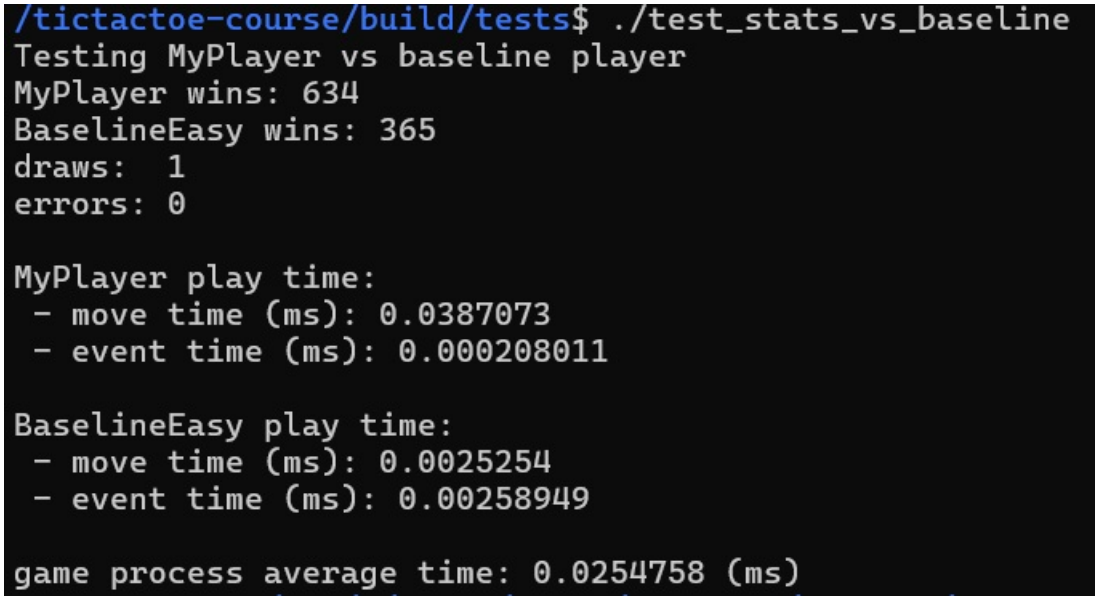

Рис. 1: Результат серии игр MyPlayer/BaselineEasy Player

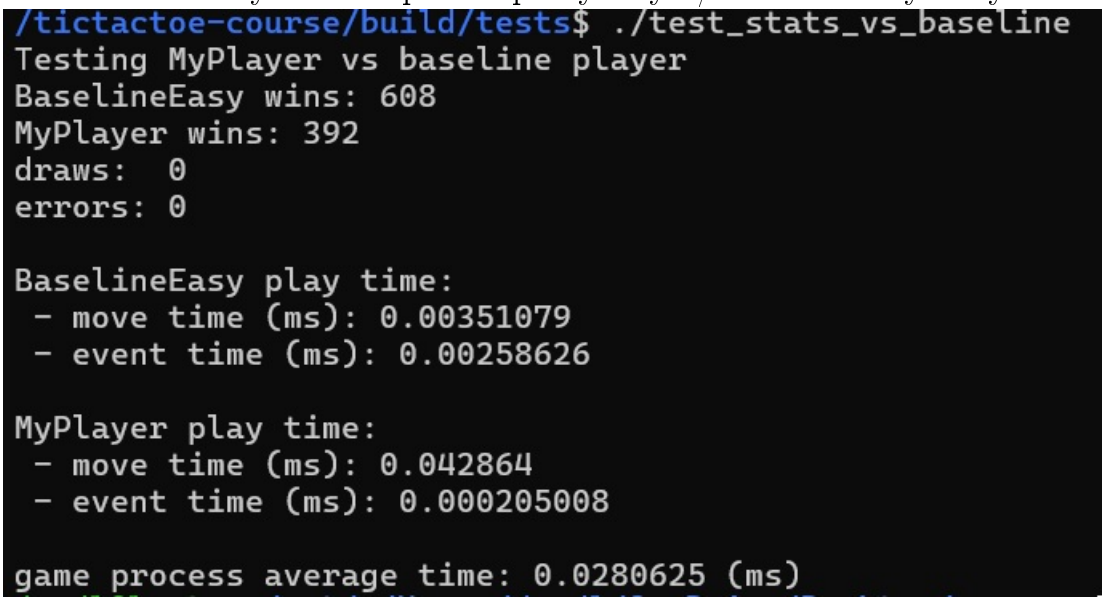

Рис. 2: Результат серии игр BaselineEasy Player/MyPlayer

Время расчета хода:

- Среднее время хода MyPlayer: 0.0258 мс.
- Среднее время хода BaselineEasy: 0.0030 мс.
- Среднее время обработки игры: 0.0268 мс.

В первой серии тестовых игр алгоритм одержал 634 победы против 365 поражений при одной ничьей. Это соответствует 63.4% побед, что превышает целевой показатель в 50%. Отсутствие ошибок (0 errors) подтверждает надежность реализации алгоритма.

Во второй серии алгоритм одержал 392 победы против 608 поражений без ничьих (39.2% побед). Ошибки полностью отсутствуют. Данные результаты неудовлетворительны, т. к. требование в виде 50% побед не пройдено. Для улучшения результатов требуется доработка алгоритма и изменение кода бота.

3. Тестирование против самого себя

   Для проверки сбалансированности алгоритма была проведена серия из 1000 игр. На рис. 3 представлены результаты тестов.

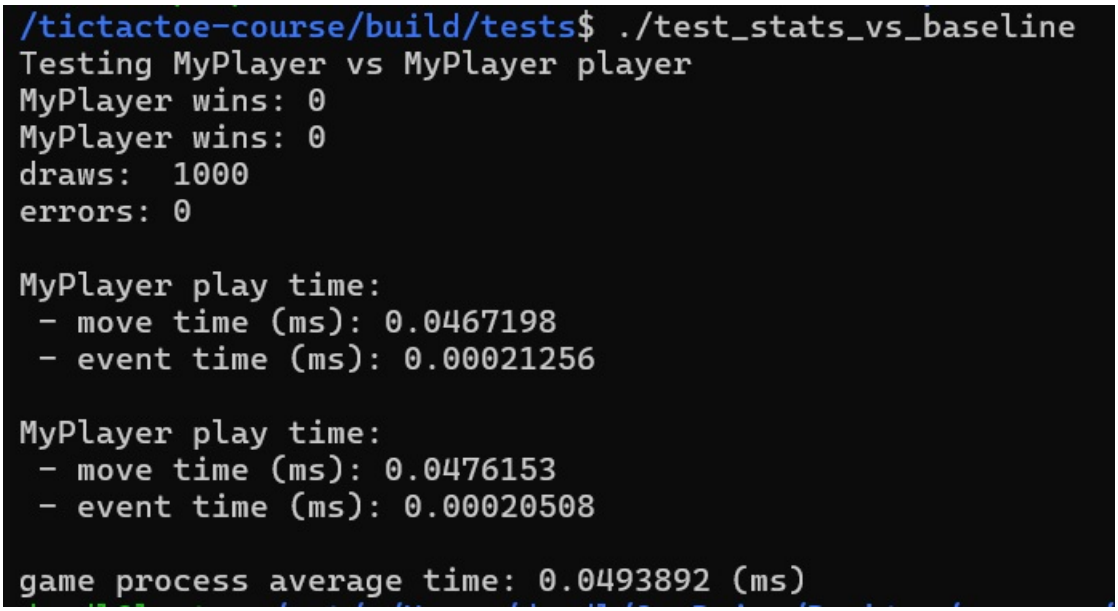

Рис. 3: Результат серии игр MyPlayer/MyPlayer

- Среднее время хода первого MyPlayer: 0.0467 мс
- Среднее время хода второго MyPlayer: 0.0476 мс
- Среднее время обработки игры: 0.0494 мс

При игре против самого себя алгоритм демонстрирует идеальную сбалансированность. Время расчета хода менее 50 мс, что соответствует требованию ко времени хода.

Заключение

В ходе выполнения курсовой работы была успешно решена поставленная задача разработки алгоритма для игры в крестики-нолики на поле увеличенного размера.

Разработан и реализован алгоритм, основанный на системе приоритетных правил, который демонстрирует стабильную работу и конкурентоспособность против базового игрока. Написан код класса ‘MyPlayer‘ объемом 310 строк на C++, который корректно интегрирован в предоставленную кодобазу и реализует интерфейс ‘IPlayer‘.

Общее время разработки алгоритма и кода: 40 часов. Объем кода: 310 строк реализации и 35 строк заголовочного файла. Общее время написания отчета: 5 часов.

Основным слабым местом реализации является нестабильность результатов - разброс между лучшей (63,4%) и худшей (39,2%) сериями составляет 24,2%. Это свидетельствует о недостаточной адаптивности алгоритма к различным игровым ситуациям.

Для улучшения результатов можно реализовать механизм предсказания ходов противника на 2-3 шага вперед. Также для улучшения результатов можно разработать систему весовых коэффициентов для оценки позиций.

Список литературы

1. Полубенцева М. И. C/C++. Процедурное программирование. Санкт-Петербург: БХВ-Петербург, 2008. — 448 с.

2. Т. А. Павловская. C/C++ Программирование на языке высокого уровня. Санкт-Петербург: Питер, 2003. — 461 с.

3. Т. А. Павловская, Ю. А. Щупак. C++. Объектно-ориентированное программирование. Практикум. СПб: Питер, 2005. — 265 с.

Приложение

A my_player.cpp

```
#include "my_player.hpp"
#include <cstdlib>
#include <cstring>

namespace ttt::my_player {

    MyPlayer::~MyPlayer( void )
    {
        if (m_board != nullptr)
        {
            for (int i = 0; i < m_rows; i++)
            delete[] m_board[i];
            delete[] m_board;
        }
    }

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

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

    void MyPlayer::initialize_board( const State &state)
    {
        if (m_board != nullptr)
        return;

        m_cols = state.get_opts().cols;
        m_rows = state.get_opts().rows;

        m_board = new Sign*[m_rows];

        for (int i = 0; i < m_rows; i++)
        {
            m_board[i] = new Sign[m_cols];

            for (int j = 0; j < m_cols; j++)
            m_board[i][j] = Sign::NONE;
        }
    }

    void MyPlayer::update_board( const State &state)
    {
        initialize_board(state);
```

```
49
50              for (int y = 0; y < m_rows; y++)
51              {
52                  for (int x = 0; x < m_cols; x++)
53                  {
54                      Sign current_state = state.get_value(x, y);
55
56                      if (m_board[y][x] != current_state)
57                      m_board[y][x] = current_state;
58                  }
59              }
60          }
61
62          int MyPlayer::check_line( int x, int y, int dx, int
               dy, Sign sign ) const
63          {
64              int count = 0;
65              int max_count = 0;
66
67              for (int dir = -1; dir <= 1; dir += 2)
68              {
69                  count = 0;
70
71                  for (int i = 1; i <= 4; i++)
72                  {
73                      int nx = x + i * dx * dir;
74                      int ny = y + i * dy * dir;
75
76                      if (nx < 0 || nx >= m_cols || ny < 0 || ny
                           >= m_rows)
77                      break;
78                      if (m_board[ny][nx] == sign)
79                      count++;
80                      else if (m_board[ny][nx] == Sign::NONE)
81                      break;
82                      else
83                      {
84                          count = 0;
85                          break;
86                      }
87                  }
88                  max_count += count;
89              }
90
91              return max_count;
92          }
93
94          bool MyPlayer::find_win_move( Point &result, Sign sign
               ) const
95          {
96              for (int y = 0; y < m_rows; y++)
97              {
98                  for (int x = 0; x < m_cols; x++)
```

```
99                  {
100                     if (m_board[y][x] != Sign::NONE)
101                     continue;
102
103                     int directions[4][2] = {{1, 0}, {0, 1},
                           {1, 1}, {1, -1}};
104
105                     for (int d = 0; d < 4; d++)
106                     {
107                         int dx = directions[d][0];
108                         int dy = directions[d][1];
109
110                         if (check_line(x, y, dx, dy, sign) >=
                               4)
111                         {
112                             result.x = x;
113                             result.y = y;
114                             return true;
115                         }
116                     }
117                 }
118             }
119             return false;
120         }
121
122         bool MyPlayer::find_block_move( Point &result, Sign
               enemy_sign, int threat_level ) const
123         {
124             for (int y = 0; y < m_rows; y++)
125             {
126                 for (int x = 0; x < m_cols; x++)
127                 {
128                     if (m_board[y][x] != Sign::NONE)
129                     continue;
130
131                     int directions[4][2] = {{1, 0}, {0, 1},
                           {1, 1}, {1, -1}};
132
133                     for (int d = 0; d < 4; d++)
134                     {
135                         int dx = directions[d][0];
136                         int dy = directions[d][1];
137
138                         int threat = check_line(x, y, dx, dy,
                               enemy_sign);
139
140                         if (threat >= threat_level)
141                         {
142                             result.x = x;
143                             result.y = y;
144                             return true;
145                         }
146                     }
```

```
147                 }
148             }
149             return false;
150         }
151 
152         bool MyPlayer::find_double_threat( Point &result, Sign
               sign ) const
153         {
154             for (int y = 0; y < m_rows; y++)
155             {
156                 for (int x = 0; x < m_cols; x++)
157                 {
158                     if (m_board[y][x] != Sign::NONE)
159                     continue;
160 
161                     int threat_count = 0;
162                     int directions[4][2] = {{1, 0}, {0, 1},
                           {1, 1}, {1, -1}};
163 
164                     for (int d = 0; d < 4; d++)
165                     {
166                         int dx = directions[d][0];
167                         int dy = directions[d][1];
168 
169                         if (check_line(x, y, dx, dy, sign) >=
                               2)
170                         {
171                             threat_count++;
172 
173                             if (threat_count >= 2)
174                             {
175                                 result.x = x;
176                                 result.y = y;
177                                 return true;
178                             }
179                         }
180                     }
181                 }
182             }
183             return false;
184         }
185 
186         bool MyPlayer::find_strategic_move( Point &result,
               Sign sign ) const
187         {
188             int center_x = m_cols / 2;
189             int center_y = m_rows / 2;
190 
191             for (int radius = 0; radius <= center_x; radius++)
192             {
193                 for (int y = center_y - radius; y <= center_y
                       + radius; y++)
194                 {
```

```
                    for (int x = center_x - radius; x <=
                        center_x + radius; x++)
                    {
                        if (x < 0 || x >= m_cols || y < 0 || y
                            >= m_rows)
                        continue;
                        if (m_board[y][x] == Sign::NONE)
                        {
                            for (int dy = -1; dy <= 1; dy++)
                            {
                                for (int dx = -1; dx <= 1;
                                    dx++)
                                {
                                    if (dx == 0 && dy == 0)
                                    continue;

                                    int nx = x + dx;
                                    int ny = y + dy;

                                    if (nx < 0 || nx >= m_cols
                                        || ny < 0 || ny >=
                                        m_rows)
                                    continue;
                                    if (m_board[ny][nx] !=
                                        Sign::NONE)
                                    {
                                        result.x = x;
                                        result.y = y;
                                        return true;
                                    }
                                }
                            }
                        }
                    }
                }
            }
            return false;
        }

        bool MyPlayer::find_any_move( Point &result) const
        {
            for (int y = 0; y < m_rows; y++)
            {
                for (int x = 0; x < m_cols; x++)
                {
                    if (m_board[y][x] == Sign::NONE)
                    {
                        result.x = x;
                        result.y = y;
                        return true;
                    }
                }
            }
```

```
242             return false;
243         }
244
245         Point MyPlayer::make_move( const State &state )
246         {
247             Point result;
248
249             if (state.get_move_no() == 0)
250             {
251                 result.x = state.get_opts().cols / 2;
252                 result.y = state.get_opts().rows / 2;
253                 return result;
254             }
255
256             update_board(state);
257
258             Sign enemy_sign = (m_sign == Sign::X) ? Sign::O :
                    Sign::X;
259
260             if (find_win_move(result, m_sign)) return result;
261             if (find_block_move(result, enemy_sign, 4)) return
                    result;
262             if (find_double_threat(result, m_sign)) return
                    result;
263             if (find_block_move(result, enemy_sign, 3)) return
                    result;
264             if (find_strategic_move(result, m_sign)) return
                    result;
265
266             for (int attempt = 0; attempt < 50; attempt++)
267             {
268                 result.x = std::rand() % m_cols;
269                 result.y = std::rand() % m_rows;
270
271                 if (m_board[result.y][result.x] == Sign::NONE)
272                 {
273                     for (int dy = -1; dy <= 1; dy++)
274                     {
275                         for (int dx = -1; dx <= 1; dx++)
276                         {
277                             if (dx == 0 && dy == 0)
278                             continue;
279                             int nx = result.x + dx;
280                             int ny = result.y + dy;
281                             if (nx >= 0 && nx < m_cols && ny
                                    >= 0 && ny < m_rows &&
                                    m_board[ny][nx] != Sign::NONE)
282                             return result;
283                         }
284                     }
285                 }
286             }
287
```

```
288             if (find_any_move(result))
289             return result;
290 
291             result.x = 0;
292             result.y = 0;
293             return result;
294         }
295 
296     }; // namespace ttt::my_player
```

B my_player.hpp

```
#pragma once

#include "core/game.hpp"

namespace ttt::my_player {

    using game::Event;
    using game::IPlayer;
    using game::Point;
    using game::Sign;
    using game::State;

    class MyPlayer : public IPlayer {
        Sign m_sign = Sign::NONE;
        const char *m_name;

        Sign** m_board = nullptr;
        int m_cols = 0;
        int m_rows = 0;

        public:
        MyPlayer(const char *name) : m_sign(Sign::NONE),
           m_name(name) {}
        ~MyPlayer();

        void set_sign(Sign sign) override;
        Point make_move(const State &game) override;
        const char *get_name() const override;

        private:
        void initialize_board(const State& state);
        void update_board(const State& state);

        int check_line(int x, int y, int dx, int dy, Sign
           sign) const;
        bool find_win_move(Point& result, Sign sign) const;
        bool find_block_move(Point& result, Sign
           enemy_sign, int threat_level) const;
        bool find_double_threat(Point& result, Sign sign)
           const;
        bool find_strategic_move(Point& result, Sign sign)
           const;
        bool find_any_move(Point& result) const;
    };

}; // namespace ttt::my_player
```