минестерство науки и высшего образования
российской федерации

федеральное государственное бюджетное образовательное учереждение
высшего профессионального 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

Основы программирования и алгоритмизации

Отчет по курсовой работе «Игрок для игры в крестики нолики»

Обучающийся: _____ Попов И. А.

Преподаватель: _____ Сеннов В.Н.

«____»_____ 20__г.

Санкт–Петербург, 2024

Содержание

1 Введение

2 Постановка задач

Цель: написать алгоритм для игрока бота для игры в крестики нолики на поле 40 на 40 до 5 в ряд. Для этого нужно было выполнить ряд задач:

- изучить исходный код игры в крестики нолики,
- придумать алгоритм и согласовать его с преподавателем,
- написать реализацию этого алгоритма и протестировать ее,
- написать отчет по выполненой работе.

3 Реализация

3.1 Понятия алгоритма

3.1.1 Атака

Атака - это подряд несколько идущих знаков одного типа (крестики или нолики)

3.1.2 Таблица веса

У атаки есть длина в промежутке от 1 до 5 и количество путей для развития (количество пустых клеток с концов атаки): 0, 1 или 2. На основе этих параметров составляется таблица веса. Это таблица, которая помогает оценивать пустые клетки рядом с уже занятыми.

Она выглядит так:

Таблица 1. Таблица веса

Длина \ Пути	0	1	2
1	0	0.1	0.25
2	0	2	5
3	0	7	50
4	0	100	1000
5	1000	1000	1000

Значения в ней были подобраны эмпирически.

3.2 Принцип работы алгоритм

Программа проходится по всему полю и ищет уже занятые клетки любым игроком. Когда она находит такую, то идет проверка всех ближайших 8 незанятых клеток: 4 по бокам и 4 по диагоналям. После этого подсчитывается вес каждой клетки на основе таблицы веса. Это происходит так: в проверяемую клетку ставиться крестик или нолик и цикл проходит вдоль каждой линии проверяемой клетки (горизонталь, вертикаль, диагонали 45 и 135 градусов) находит длину атаки и количество путей для ее развития. Этот цикл проходит два раза, т.к. подсчитывает вес клетки со стороны игрока и со стороны опонента, то есть, ставиться сначала знак игрока, а потом знак опонента. Веса по 4 линиям суммируются. После этого находиться максимальный вес из этих 8 ближайших клеток. Далее программа проверяет так все оставшиеся занятые клетки и находит клетку с максимальным весом, куда в итоге и будет поставлен знак.

3.3 Методы

Класс бота UserPlayer является наследуемым от класса Player и включает в себя следующие основные методы:

Листинг 1. Основные методы класса UserPlayer

```

Point play(const GameView& game) override;

WeightAndPoint calculate_near_points(const GameView& game, const Point&
    currently_checking_point, const Boundary& field_size);

float calculate_weight(const GameView& game, Point&
    currently_checking_point, const Boundary& field_size);

void calculate_weight_horizontal(const GameView& game, const Boundary&
    field_size, Point& currently_checking_point, float& calculating_weight);

void calculate_weight_vertical(const GameView& game, const Boundary&
    field_size, Point& currently_checking_point, float& calculating_weight);

void calculate_weight_diagonal_45(const GameView& game, const Boundary&
    field_size, Point& currently_checking_point, float& calculating_weight);

void calculate_weight_diagonal_135(const GameView& game, const Boundary&
    field_size, Point& currently_checking_point, float& calculating_weight);

```

3.3.1 Метод play

Первый метод play вызывается, когда бот должен сделать ход, в нем происходит проход по всему полю для поиска занятых клеток, чтобы найти самую эффективную, используя метод calculate_near_points и вернуть ее координаты в виде класса Point. Код релизации этого метода приведен в листинге 2.

Листинг 2. Релизация метода Play

```

Boundary field_size = game.get_settings().field_size;
Point result((field_size.min.x + field_size.max.x) / 2, (field_size.min.y +
     field_size.max.y) / 2);
if (game.get_state().field.get()->get_value(result) != Mark::None)
{
 result.x += 1;
}
WeightAndPoint calculated;
float max_weight = 0;
for (int x = field_size.min.x; x <= field_size.max.x; x++)
{
 for (int y = field_size.min.y; y <= field_size.max.y; y++)
 {
  Point searching_point(x, y);
  if (game.get_state().field.get()->get_value(searching_point) != Mark::
    None)
```

```
  {
   calculated = calculate_near_points(game, searching_point, field_size);
   if (calculated.first > max_weight)
   {
    max_weight = calculated.first;
    result = calculated.second;
   }
   else if (calculated.first >= weight_table[5][2] && game.get_state().
   field.get()->get_value(searching_point) == mark)
   {
    return result;
   }
  }
 }
}
return result;

```

3.3.2 Метод calculate_near_points

У метода calculate_near_points возвращаемым значением является псевдоним типа pair<float, Point> WeightAndPoint. Этот псевдоним на основе объекта стандартной библиотеки шаблонов stl используется, чтобы функция могла сразу возвращать вес клетки и ее координаты. Данный метод реализует подсчет веса ближайших незанятых клеток на основе вызовов метода calculate_weight и возвращает координаты клетки вместе с ее весом. В листинге 3 будет приведена часть реализации этого метода, так как для 8 клеток она совпадает за исключнием некоторых отличий.

Листинг 3. Метод calculate_near_points

```

Point currently_checking_point(searching_point);
float calculated_weight = 0;
WeightAndPoint result(0, currently_checking_point);
if (currently_checking_point.x + 1 <= field_size.max.x)
{
 currently_checking_point.x += 1;
 if (game.get_state().field.get()->get_value(currently_checking_point) ==
   Mark::None)
 {
  if (result.first == 0){result.second = currently_checking_point;};
  calculated_weight = calculate_weight(game, currently_checking_point,
   field_size);
  if (calculated_weight > result.first)
  {
   result.first = calculated_weight;
   result.second = currently_checking_point;
  }
  else if (calculated_weight >= weight_table[5][2] && game.get_state().
   field.get()->get_value(searching_point) == mark)
  {
   result.first = calculated_weight;
   result.second = currently_checking_point;
```

```
    return result;
   }
  }
  currently_checking_point.x -= 1;
}

```

3.3.3 Метод calculate_weight и его подметоды

Метод calculate_weight раздроблен на 4 части и нужен лишь для того, чтобы поочередно вызывать методы calculate_weight_horizontal, calculate_weight_vertical, calculate_weight_diagonal_45 и calculate_weight_diagonal_135, которые уже выполняют схожие действия: проходятся по своей линии и подсчитывают вес клетки со стороны игрока и опонента. После вызова 4 методов метод calculate_weight возвращает итоговый вес клетки на основе проходов по 4 линиям. В листинге 4 будет приведен код для метода calculate_weight, а в листинге 5 будет приведен код метода calculate_weight_horizontal, который схож с кодом оставшихся 3 методов.

Листинг 4. Метод calculate_weight

```
float calculating_weight = 0; //Will contain the sum of weight of all 4 lines
// For horizontal
calculate_weight_horizontal(game, field_size, currently_checking_point, calculating_weight);
// For vertical
calculate_weight_vertical(game, field_size, currently_checking_point, calculating_weight);
// For 45 degrees diagonal
calculate_weight_diagonal_45(game, field_size, currently_checking_point, calculating_weight);
// For 135 degrees diagonal
calculate_weight_diagonal_135(game, field_size, currently_checking_point, calculating_weight);
return calculating_weight;

```

Листинг 5. Метод calculate_weight_horizontal

```
// Calculating weight from the point of our view
Point for_calculating(currently_checking_point);
int amount_of_this_marks = 1; //Always starts with 1, because on this empty point we imaginary put "mark"
int amount_of_free_ways_1 = 0;//It can be 1 or 0, because we going trough one "for" only forward of backward, relatively speaking
bool is_out = false;
for (int x = currently_checking_point.x + 1; x < currently_checking_point.x + 5; x++) //calculating weight to the right of currently_checking_point
{
 if (is_out){break;}
 for_calculating.x += 1;
 if (x <= field_size.max.x)
 {
```

```
  switch (game.get_state().field.get()->get_value(for_calculating))
  {
  case Mark::Cross:
   if (mark == Mark::Cross)
   {
    amount_of_this_marks += 1;
    amount_of_free_ways_1 = x == field_size.max.x ? 0 : 1;
   }
   else
   {
    amount_of_free_ways_1 = 0;
    is_out = true;
   }
   break;
  case Mark::Zero:
   if (mark == Mark::Zero)
   {
    amount_of_this_marks += 1;
    amount_of_free_ways_1 = x == field_size.max.x ? 0 : 1;
   }
   else
   {
    amount_of_free_ways_1 = 0;
    is_out = true;
   }
   break;
  default:
   amount_of_free_ways_1 = 1;
   is_out = true;
  }
 }
 else {
  amount_of_free_ways_1 = 0;
  is_out = true;
 }
}
for_calculating = currently_checking_point;
int amount_of_free_ways_2 = 0;
is_out = false;
for (int x = currently_checking_point.x - 1; x > currently_checking_point.x
    - 5; x--) //calculating weight to the left of currently_checking_point
{
 if (is_out) { break; }
 for_calculating.x -= 1;
 if (x >= field_size.min.x)
 {
  switch (game.get_state().field.get()->get_value(for_calculating))
  {
  case Mark::Cross:
   if (mark == Mark::Cross)
   {
    amount_of_this_marks += 1;
    amount_of_free_ways_2 = x == field_size.min.x ? 0 : 1;
   }
   else
```

```
    {
     amount_of_free_ways_2 = 0;
     is_out = true;
    }
    break;
   case Mark::Zero:
    if (mark == Mark::Zero)
    {
     amount_of_this_marks += 1;
     amount_of_free_ways_2 = x == field_size.min.x ? 0 : 1;
    }
    else
    {
     amount_of_free_ways_2 = 0;
     is_out = true;
    }
    break;
   default:
    amount_of_free_ways_2 = 1;
    is_out = true;
   }
  }
  else {
   amount_of_free_ways_2 = 0;
   is_out = true;
  }
}
amount_of_this_marks = amount_of_this_marks > 5 ? 5 : amount_of_this_marks;
calculating_weight += weight_table[amount_of_this_marks][
    amount_of_free_ways_1 + amount_of_free_ways_2];
// Calculating now weight from the point of view of oponent
for_calculating = currently_checking_point;
int amount_of_other_marks = 1;
int amount_of_other_free_ways_1 = 0;
is_out = false;
for (int x = currently_checking_point.x + 1; x < currently_checking_point.x
    + 5; x++)
{
 if (is_out) { break; }
 for_calculating.x += 1;
 if (x <= field_size.max.x)
 {
  switch (game.get_state().field.get()->get_value(for_calculating))
  {
  case Mark::Cross:
   if (mark != Mark::Cross)
   {
    amount_of_other_marks += 1;
    amount_of_other_free_ways_1 = x == field_size.max.x ? 0 : 1;
   }
   else
   {
    amount_of_other_free_ways_1 = 0;
    is_out = true;
   }
```

```
    break;
   case Mark::Zero:
    if (mark != Mark::Zero)
    {
     amount_of_other_marks += 1;
     amount_of_other_free_ways_1 = x == field_size.max.x ? 0 : 1;
    }
    else
    {
     amount_of_other_free_ways_1 = 0;
     is_out = true;
    }
    break;
   default:
    amount_of_other_free_ways_1 = 1;
    is_out = true;
   }
  }
  else {
   amount_of_other_free_ways_1 = 0;
   is_out = true;
  }
}
for_calculating = currently_checking_point;
int amount_of_other_free_ways_2 = 0;
is_out = false;
for (int x = currently_checking_point.x - 1; x > currently_checking_point.x
    - 5; x--)
{
 if (is_out) { break; }
 for_calculating.x -= 1;
 if (x >= field_size.min.x)
 {
  switch (game.get_state().field.get()->get_value(for_calculating))
  {
  case Mark::Cross:
   if (mark != Mark::Cross)
   {
    amount_of_other_marks += 1;
    amount_of_other_free_ways_2 = x == field_size.min.x ? 0 : 1;
   }
   else
   {
    amount_of_other_free_ways_2 = 0;
    is_out = true;
   }
   break;
  case Mark::Zero:
   if (mark != Mark::Zero)
   {
    amount_of_other_marks += 1;
    amount_of_other_free_ways_2 = x == field_size.min.x ? 0 : 1;
   }
   else
   {
```

```
173     amount_of_other_free_ways_2 = 0;
174     is_out = true;
175    }
176    break;
177   default:
178    amount_of_other_free_ways_2 = 1;
179    is_out = true;
180   }
181  }
182  else {
183   amount_of_other_free_ways_2 = 0;
184   is_out = true;
185  }
186 }
187 amount_of_other_marks = amount_of_other_marks > 5 ? 5 :
      amount_of_other_marks;
188 calculating_weight += weight_table[amount_of_other_marks][
      amount_of_other_free_ways_1 + amount_of_other_free_ways_2];
189
```

4 Тестирование приложения

Тестирование бота проходило с помощью предоставленного вместе с исходным кодом игры класса RandomPlayer, который устанавливает знак в случайную клетку. После исправления ошибок, допущенных при написаний кода, было достигнуто 100% побед в 50 игр, 40 из которых закончились в 5 ходов со стороны игрока UserPlayer, оставшиеся 10 были выиграны за 6 ходом со стороны игрока UserPlayer.

5 Заключение

В процессе работы над ботом были выполнены все поставленные задачи. Приобретенные навыки могут помочь в дальнейшем, например, при работе с чужим кодом или при создании более сложных ботов.

Работа над реализацией игрока и отладкой кода заняла около 8 часов. Написание отчета заняло около 3 часов. Код насчитывает более 1000 строк. Он был написан на языке C++ версии стандарта ISO C++ 98. Оптимизирующий компилятор MIcrosoft (R) C/C++ версии 19.38.33134 для x86

Полный код игрока находиться в репозитории GitHub, ссылку на который указана в приложении А.

Приложение А.

https://github.com/TruePerajush/UserPlayer