Politechnika Wrocławska

Wydział Elektroniki, Fotoniki i Mikrosystemów

PAMSI

Sprawozdanie nr 3

Implementacja algorytmu MiniMax z alfa-beta cięciami w grze kółko i krzyżyk

Prowadzący:
dr hab. inż. Andrzej Rusiecki

Wykonał:
Jakub Kusz

Wrocław, 10 czerwca 2022r.

Spis treści

1. Cel ćwiczenia

Celem ćwiczenia jest zaimplementowanie algorytmu MiniMax z alfa-beta cięciami w grze kółko i krzyżyk. Użytkownikowi dano możliwość wyboru poziomu trudności (głębokość algorytmu) i wielkości planszy.

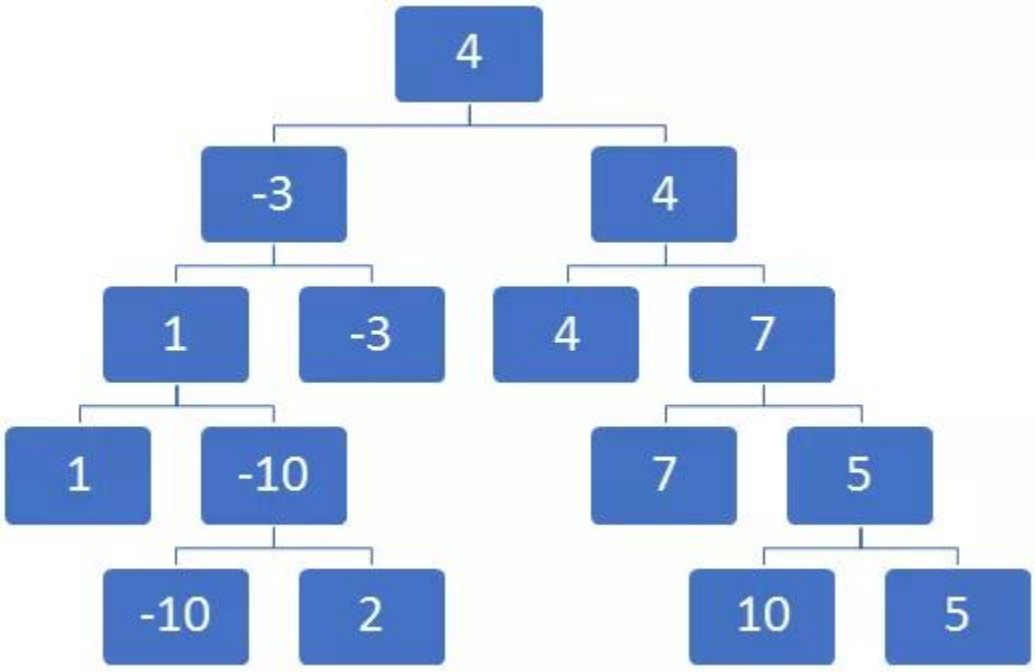
2. Wstęp

2.1. MiniMax

MinimMax jest algorytmem służącym do minimalizowania maksymalnych możliwych strat. Alternatywnie można je traktować jako maksymalizację minimalnego zysku. Wywodzi się to z teorii gry o sumie zerowej, obejmujących oba przypadki, zarówno ten, gdzie gracze wykonują ruchy naprzemiennie, jak i ten, gdzie wykonują ruchy jednocześnie. Zostało to również rozszerzone na bardziej skomplikowane gry i ogólne podejmowanie decyzji w obecności niepewności.

2.1.1. Zasada działania

Najprostszym sposobem przedstawiania zasady działania algorytmu MiniMax jest ukazanie jego pracy na przykładzie. Na rysunku 1, pokazujemy wyniki uzyskane przez gracza x w każdym momencie gry. W bazie, na pierwszym poziomie, decyzję podejmuje przeciwnik. Z tego powodu podane są scenariusze, w których gracz może stracić -10 lub wygrać 5. Na drugim poziomie zależy to od gracza x, więc zmaksymalizuje on swój zysk. Pomiędzy stratą 10 a wygraną 1 wygrasz 1. Podobnie, jeśli wygrasz 5 lub 7, wygrasz 7. Potem znowu kolej na przeciwnika, więc zostaną podane scenariusze, w których gracz x ma najgorszy wynik, -3 i 4, w zależności od przypadku. Wreszcie, pomiędzy przegraną 3 a wygraną 4, gracz x podejmie decyzję, która pozwoli temu drugiemu.



Rysunek 1: Drzewo przeszukiwań

2.2. Alfa-beta cięcia

Korzyść płynąca z algorytmu alfa-beta leży w fakcie, że niektóre gałęzie drzewa przeszukiwania mogą zostać odcięte. Czas przeszukiwania ograniczony zostaje do przeszukania najbardziej obiecujących poddrzew, w związku z czym możemy zejść głębiej w tym samym czasie. Tak samo jak klasyczny min-max, algorytm należy do algorytmów wykorzystujących metody podziału i ograniczeń. Współczynnik rozgałęzienia jest dwukrotnie mniejszy niż w klasycznym MiniMax. Algorytm staje się wydajniejszy, gdy węzły rozwiązywane są układane w porządku optymalnym lub jemu bliskim.

3. Implementacja

Z powodu ograniczeń związanych ze sprzętem nie jesteśmy w stanie rozwinąć całego drzewa od początku do końca, wiec algorytm MiniMax działa do określonej głębokości, na której następuje ocena pozycji za pomocą przeznaczonej do tego funkcji.Implementacja całej gry w języku C++ została umieszczona **tutaj**.

3.1. Ocena pozycji

Do oceny pozycji po osiągnięciu danej głębokości została napisana metoda evaluate_position(), która ocenia pozycje na podstawie ilości kółek/krzyżyków z rzędu.

Listing 1: evaluate_position()

```
1  void t_game :: evaluate_position(){
2      int tmp;
3      for(int i = 0; i < number_of_fields; ++i){
4          tmp = 0;
5          for(int j = 0; j < number_of_fields; ++j){
6              if(gameborad_table[i][j] == 'o'){
7                  tmp++;
8              }else if(gameborad_table[i][j] == 'x'){
9                  tmp = 0;
10                 break;
11             }
12         }
13         if(tmp){
14             position_rating += pow(10,tmp);
15         }
16     }
17
18     for(int i = 0; i < number_of_fields; ++i){
19         tmp = 0;
20         for(int j = 0; j < number_of_fields; ++j){
21             if(gameborad_table[j][i] == 'o'){
22                 tmp++;
23             }else if(gameborad_table[j][i] == 'x'){
24                 tmp = 0;
25                 break;
```

```
26                  }
27              }
28              if(tmp){
29                  position_rating += pow(10,tmp);
30              }
31          }
32          tmp = 0;
33          for(int j = 0; j < number_of_fields; ++j){
34              tmp = 0;
35                  if(gameborad_table[j][j] == 'o'){
36                      tmp++;
37                  }else if(gameborad_table[j][j] == 'x'){
38                      tmp = 0;
39                      break;
40                  }
41                  if(tmp){
42                      position_rating += pow(10,tmp);
43                  }
44              }
45
46          tmp = 0;
47          for(int i = 0, j = number_of_fields - 1; i < number_of_fields; ++i, --j){
48              tmp = 0;
49                  if(gameborad_table[i][j] == 'o'){
50                      tmp++;
51                  }else if(gameborad_table[i][j] == 'x'){
52                      tmp = 0;
53                      break;
54                  }
55                  if(tmp){
56                      position_rating += pow(10,tmp);
57                  }
58              }
59
60
61      }
62
63
```

3.2. MiniMax z alfa-beta cięciami

Metoda znajdująca najlepsze zagranie: Na początku sprawdzane jest czy któryś z zawodników wygrał, jeśli tak to zwracamy maksymalnie/minimalnie duże oceny pozycji i funkcja się kończy. Jeśli nikt nie wygrał, to po osiągnięciu danej głębokości następuje ocena pozycji i zwrócenie jej wartości. Jeśli nie osiągnęliśmy jeszcze zadanej głębokości, algorytm wykonuje naprzemiennie ruchy graczy i wywołuje się rekurencyjnie. Po osiągnięciu ostatniego wywołania funkcji do działania przystępują alfa-beta cięcia.

Listing 2: minimax_alpha_beta()

```
1  int t_game :: minimax_alpha_beta(who_start current_player, int depth, long int a, long int b){
2      this->check_win();
3      if (winner != who_start::draw){
4          if (current_player == who_start :: ai){
5              return INT32_MAX;
6          }else{
7              return INT32_MIN;
8          }
9      }
10     if (this->is_finish() || depth == 0){
11         if (current_player == who_start :: ai){
12             this->evaluate_position();
13             return position_rating;
14         }else{
15             this->evaluate_position();
16             return -position_rating;
17         }
18     }
19     long int best_score;
20     if (current_player == who_start :: human){
21         current_player = who_start :: ai;
22         best_score = INT64_MIN;
23     }else{
24         current_player = who_start :: human;
25         best_score = INT64_MAX;
26     }
27     int tmp;
28     for (int i = 0; i < number_of_fields; ++i){
29         for (int j = 0; j < number_of_fields; ++j){
30             if (gameborad_table[i][j] == '_'){
31                 if (current_player == who_start :: ai){
32                     gameborad_table[i][j] = 'o';
33                     tmp = this->minimax_alpha_beta(current_player, depth-1, a, b);
34                     if (best_score < tmp){
35                         best_score = tmp;
36                     }
37                     if (a < best_score){
38                         a = best_score;
39                     }
40                     gameborad_table[i][j] = '_';
41                     if (a >= b){
42                         return best_score;
43                     }
44                 }else{
45                     gameborad_table[i][j] = 'x';
46                     tmp = this->minimax_alpha_beta(current_player, depth-1, a, b);
```

```
47                      if(best_score > tmp){
48                          best_score = tmp;
49                      }
50                      if(a > best_score){
51                          a = best_score;
52                      }
53                      gameborad_table[i][j] = '_';
54                      if(a >= b){
55                          return best_score;
56                      }
57                  }
58              }
59          }
60      }
61      return best_score;
62  }
63
```

3.3. Ustalenie współrzędnych najlepszego posunięcia

Do wyboru współrzędnych dla najlepszego posunięcia została napisana poniższa metoda:

Listing 3: best_ai_move()

```
1       void t_game :: best_ai_move(int depth){
2
3       long int best_score = INT64_MIN;
4       int tmp;
5       int set_i;
6       int set_j;
7       for(int i = 0; i < number_of_fields; ++i){
8           for(int j = 0; j < number_of_fields; ++j){
9               if(gameborad_table[i][j] == '_'){
10                  gameborad_table[i][j] = 'o';
11                  tmp = minimax_alpha_beta(who_start :: ai, depth, INT64_MIN, INT64_MAX);
12
13                  gameborad_table[i][j] = '_';
14                  if(tmp > best_score){
15                      best_score = tmp;
16                      set_i = i;
17                      set_j = j;
18                  }
19              }
20          }
21      }
22      if(set_i < number_of_fields && set_j < number_of_fields)
23          gameborad_table[set_i][set_j] = 'o';
24  }
```

4. Środowisko graficzne - SFML

W celu przyjemnego przedstawienia rozgrywki została wykorzystana biblioteka graficzna SFML. Cechuje się ona łatwością użycia, nieskomplikowanym procesem konfiguracyjnym i dobrą wydajnością.

4.1. Obliczanie odstępów między liniami stanowiącymi planszę

Aby obliczyć odstępy między liniami stanowiącymi planszę rozmiar okna został podzielony przez ilość pól w kolumnie. Znając rozmiar pola łatwo można narysować linię stanowiące planszę.

Listing 4: set_gameboad_table()

```
1
2       void t_gameboard :: set_gameboad_table( sf :: Vector2i xy){
3           player = who_start :: ai;
4           xy.x = xy.x / filed_size;
5           xy.y = xy.y / filed_size;
6           if ((xy.x < number_of_fields )&& (xy.y < number_of_fields)){
7               if (gameborad_table[xy.x][xy.y] == '_'){
8                   if (player == who_start :: ai){
9                       if (++move_counter % 2 == 0){
10                          gameborad_table[xy.x][xy.y] = 'x';
11                      }else{
12                          gameborad_table[xy.x][xy.y] = 'x';
13                      }
14
15                  }
16                  if (player == who_start :: human){
17                      if (++move_counter % 2 == 1){
18                          gameborad_table[xy.x][xy.y] = 'x';
19                      }else{
20                          gameborad_table[xy.x][xy.y] = 'x';
21                      }
22
23                  }
24              }
25          }
26
27
28  }
29
30
31
```

4.2. Rysowanie planszy, kółek, krzyżyków i zwycięstwa

Poniższa wirtualna z biblioteki SFML została wykorzystana do rysowania całego stanu gry:

Listing 5: draw()

```
1
2  void t_game :: draw(sf::RenderTarget& target, sf::RenderStates states)const{
3
4          t_circe  circle ;
5          t_cross cross;
6          int  circle_win = 0;
7          int  cross_win = 0;
8          circle .setCharacterSize( filed_size );
9          cross.setCharacterSize( filed_size );
10         for(int  i = 0; i < lines.size (); ++i){
11             target.draw(lines[i], states );
12         }
13         for(int  i = 0; i < number_of_fields; ++i){
14             for(int  j = 0; j < number_of_fields; ++j){
15
16                 if(gameborad_table[i][j] == 'x'){
17                     cross.set_position(i * filed_size , j * filed_size );
18                     target.draw(cross,states );
19                 }
20                 if(gameborad_table[i][j] == 'o'){
21                     circle .set_position(i * filed_size  , j * filed_size );
22                     target.draw(circle , states );
23                 }
24             }
25         }
26         int  circles ;
27         int  crosses;
28         circle .setColor(sf  ::  Color  ::  Green);
29         cross.setColor(sf  ::  Color  ::  Green);
30         for(int  i = 0; i < number_of_fields; ++i){
31             for(int  j = 0; j < number_of_fields; ++j){
32                 if(gameborad_table[i][j] == 'o'){
33                     circles ++;
34                 }
35                 if(gameborad_table[i][j] == 'x'){
36                     crosses++;
37                 }
38             }
39             if( circles  == number_of_fields){
40                 for(int  j = 0; j < number_of_fields; ++j){
41                     circle .set_position(i * filed_size  , j * filed_size );
42                     target.draw(circle , states );
43                 }
44                 return;
45             }
46             if(crosses  == number_of_fields){
```

```
47                for(int j = 0; j < number_of_fields; ++j){
48                    cross.set_position(i * filed_size , j * filed_size );
49                    target.draw(cross,states );
50                 }
51                 return;
52
53            }
54             circles = 0;
55            crosses = 0;
56        }
57
58        for(int i = 0; i < number_of_fields; ++i){
59             for(int j = 0; j < number_of_fields; ++j){
60                 if(gameborad_table[j][i] == 'o'){
61                     circles++;
62                }
63                if(gameborad_table[j][i] == 'x'){
64                    crosses++;
65                }
66             }
67             if( circles == number_of_fields){
68                 for(int j = 0; j < number_of_fields; ++j){
69                    circle.set_position(j * filed_size , i * filed_size );
70                    target.draw(circle,states );
71                 }
72                 return;
73
74            }
75            if(crosses == number_of_fields){
76                for(int j = 0; j < number_of_fields; ++j){
77                    cross.set_position(j * filed_size , i * filed_size );
78                    target.draw(cross,states );
79                 }
80                 return;
81
82            }
83             circles = 0;
84            crosses = 0;
85        }
86
87
88        for(int j = 0; j < number_of_fields; ++j){
89                 if(gameborad_table[j][j] == 'o'){
90                     circles++;
91                }
92                if(gameborad_table[j][j] == 'x'){
93                    crosses++;
```

```
94              }
95          }
96          if( circles == number_of_fields){
97              for(int j = 0; j < number_of_fields; ++j){
98                  circle.set_position(j * filed_size , j * filed_size );
99                  target.draw(circle,states);
100             }
101             return;
102
103         }
104         if(crosses == number_of_fields){
105             for(int j = 0; j < number_of_fields; ++j){
106                 cross.set_position(j * filed_size , j * filed_size );
107                 target.draw(cross,states);
108             }
109             return;
110
111         }
112         circles = 0;
113         crosses = 0;
114
115         for(int i = 0, j = number_of_fields - 1; i < number_of_fields; ++i, --j){
116             if(gameborad_table[i][j] == 'o'){
117                 circles++;
118             }
119             if(gameborad_table[i][j] == 'x'){
120                 crosses++;
121             }
122         }
123         if( circles == number_of_fields){
124             for(int i = 0, j = number_of_fields - 1; i < number_of_fields; ++i, --j){
125                 circle.set_position(i * filed_size , j * filed_size );
126                 target.draw(circle,states);
127             }
128             return;
129
130         }
131         if(crosses == number_of_fields){
132             for(int i = 0, j = number_of_fields - 1; i < number_of_fields; ++i, --j){
133                 cross.set_position(i * filed_size , j * filed_size );
134                 target.draw(cross,states);
135             }
136             return;
137
138         }
139         circles = 0;
140         crosses = 0;
```

```
141 }
142
143
```

5. Testy

Gra została przetestowana pod względem poprawności działania i wydajności.

5.1. Poprawność działania

Po rozegraniu wielu gier stwierdzono, że algorytm działa poprawnie, stara się wyszukiwać najlepsze ruchy, blokuje możliwości wygrania przez człowieka, najlepszą możliwością jest remis. Z obserwacji wynika, że już przy głębokości 1 nie człowiek nie jest w stanie wygrać z komputerem.

5.2. Wydajność

Aby gra przebiegała sprawie, dla danych rozmiarów planszy, zostały arbitralnie przydzielone maksymalne głębokości algorytmu:

Tablica 1: Zawierajaca czasy obliczania pierwszego ruchuchu algorytmu MiniMax dla danej wielkości planszy

Wielkość planszy	Głebokość maksymalna	Czas oczekiwania na pierwszy ruch [ms]
3	dowolna, algorytm jest w stanie szybko rozwinąć grę do samego końca	90
4	5	1523
5	4	1816
6	3	480
7	3	1891
8	2	114
9	2	271
10	2	589

Zastosowanie większych głębokości dla danych rozmiarów planszy wiązało się z bardzo długim oczekiwaniem na ruch komputera w początkowej fazie gry. Wraz z każdym ruchem czas oczekiwania na odpowiedź komputera zmniejsza się.

6. Wnioski

- Algorytm MiniMax wraz z alfa-beta cięciami doskonale nadaje się do symulacji gracze w grze o sumie zerowej;
- Dzięki zastosowaniu alfa-beta cięć algorytm znacząco skraca czas działania, dzięki odcinaniu gałęzi niemających znaczenia dla rozwoju gry,
- Algorytm nawet na głębokości 1 jest w stanie skutecznie powstrzymać zwycięstwo człowieka, pozwalając maksymalnie na remis.

- Ważne jest aby dopasować głębokość do wielkości planszy. Zbyt duża głębokość znacząco wydłuża działanie programu.
- SFLM jest potężnym narzędziem do tworzenia GUI, niestety brakuje w tej bibliotece prostej obsługi przycisków i obsługi pól tekstowych przez co poziom trudności i wymiar planszy należy podać w terminalu.