Politechnika Wrocławska

Autor: Filip Strózik 260377 Prowadzący: Mgr Inż. Michał Karol

Sztuczna Inteligencja i Inżynieria Wiedzy

Sprawozdanie z zadania 2

Wstęp teoretyczny i założenia

Celem ćwiczenia jest praktyczne zapoznanie się z algorytmami grania w gry dwuosobowe. Zapoznanie się z algorytmem Minimax oraz jego ulepszeniem z wykorzystaniem alfa-beta cięć.

Na podstawie podanych źródeł w liście laboratoryjnej zostały przeprowadzone zadania na temat gry Reversi, algorytmów oraz szeroko pojętej strategii tej gry. W ramach zadań dodatkowych została zrealizowana możliwość partii pomiędzy 2 programami taki sposób, że użytkownik może grać z innym użytkownikiem, Użytkownik może grać z „ komputerem” oraz „komputer” może grać z drugim „komputerem”. Wprowadzone zostały modyfikacje które pozwalają na jednego ruchu przez po czym następuje oczekiwanie na ruch użytkownika, oraz w przypadku gry „komputer” vs „komputer” można przeprowadzić całą partię do wygranej. Takie funkcjonalności wymagały utworzenie interfejsu użytkownika, który wspierałby powyższe założenia.

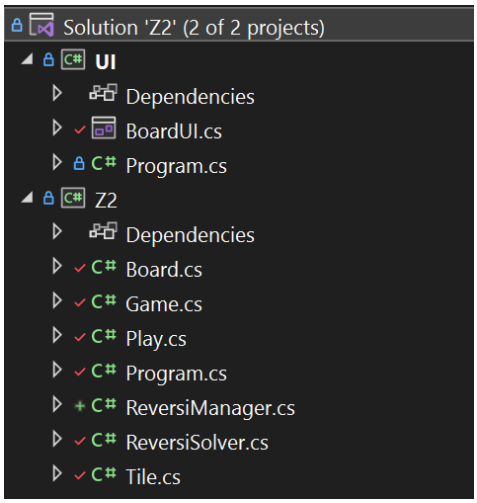
Trzeba podkreślić, że potrzeba dla realizacji interfejsu użytkownika oraz możliwości rozegrania partii na trzy warianty, wymaga bardzo dość przemyślanej struktury programu tak aby można było zrealizować wszystkie założenia nie zmieniając kodu, dlatego struktura programu jest dość kompleksowa.

Technologia

Implementacja rozwiązana została napisana w C# 10.0 uruchamiana w .NET 6. GUI został utworzony za pomocą Windows Forms.

Implementacja

Struktura projektu



Dla możliwości „gry” oraz przeprowadzenia testów wydajnościowych i zgromadzenia wyników w ramach rozwiązania zostały utworzone dwa projekty: UI, Z2.

UI – zawiera interfejs użytkownika oraz jego obsługę. Korzysta z wszystkich klas z projektu Z2 dla realizacji możliwości rozegrania partii między użytkownikiem a czymkolwiek innym.

Z2 – zawiera klasy odpowiedzialne za strukturę reprezentacji planszy, gry, ruchu, manedżera gry, modułu sztucznej inteligencji realizującej algorytmy oraz heurystyki oraz model pionka.

poprawne zdefiniowanie stanu gry i funkcji generującej możliwe ruchy dla danego stanu i gracza

Metoda PossiblePlays zwraca słownik, gdzie kluczem jest krotka reprezentująca koordynaty postawienia pionka na planszy dla ruchu, a wartością jest stan planszy po wykonaniu tego ruchu uwzględniający przemianę pionków odpowiednio z logiką gry.

```
public Dictionary<Tuple<int, int>, Play> PossiblePlays(bool otherPlayer = false)
{

    List<Tuple<int, int>> possiblePositions = Board.OpenAdjacentSpots();
    Dictionary<Tuple<int, int>, Play> results = new();

    TileColor playerColor = IsFirstPlayer ? TileColor.BLACK : TileColor.WHITE;

    if (otherPlayer)
    {
        playerColor = IsFirstPlayer ? TileColor.WHITE : TileColor.BLACK; ;
    }

    foreach (Tuple<int, int> coord in possiblePositions)
    {

        Play possiblePlay = new(Board, playerColor, coord);

        if (possiblePlay.AffectedTiles != null)
        {
            results.Add(coord, possiblePlay);
        }
    }
    return results;
}
```

```
        public List<Tuple<int, int>> OpenAdjacentSpots()
        {
            List<Tuple<int, int>> openAdjacent = new();
            for (int x = 0; x < Size; x++)
            {
                for (int y = 0; y < Size; y++)
                {
                    if (board[x, y] == null && AdjacentToTile(x, y))
                    {
                        openAdjacent.Add(Tuple.Create(x, y));
                    }
                }
            }
            return openAdjacent;
        }

        private bool AdjacentToTile(int x, int y)
        {
            for (int ix = Math.Max(0, x - 1); ix <= x + 1 && ix < Size; ix++)
            {
                for (int iy = Math.Max(0, y - 1); iy <= y + 1 && iy < Size; iy++)
                {
                    if (board[ix, iy] != null) return true;
                }

            }
            return false;
        }
```

zbudowanie zbioru heurystyk oceny stanu gry dla każdego z graczy, każdy z graczy powinien mieć przynajmniej 3 różne strategie

```
        #region heuristics

        public static int RandomHeuristic(Game game, TileColor color)
        {
            Random rand = new();
            int randomValue = rand.Next(100);
            return randomValue;

        }

        // Calculates the heuristic value based on the difference of black tiles and white tiles
        public static int TileCountHeuristic(Game game, TileColor color)
        {
            int black = game.Board.GetNumColor(TileColor.BLACK);
            int white = game.Board.GetNumColor(TileColor.WHITE);

            if (black + white == 0)
            {
                return 0;
            }

            if (color == TileColor.BLACK)
            {
                return 100 * (black - white) / (black + white);
            }
            else if (color == TileColor.WHITE)
            {
                return 100 * (white - black) / (black + white);
            }

            return 0;
        }


        // Calculates a heuristic based on how many moves a player has relative to how many moves the opponent
has
        public static int ActualMobilityHeuristic(Game game, TileColor color)
        {
            TileColor currentPlayer = game.IsFirstPlayer ? TileColor.BLACK : TileColor.WHITE;
            TileColor maxPlayer = color;
            TileColor minPlayer = color == TileColor.BLACK ? TileColor.WHITE : TileColor.BLACK;

            int maxMobility;
            int minMobility;

            if (currentPlayer == maxPlayer)
            {
                maxMobility = game.PossiblePlays().Count;
                minMobility = game.PossiblePlays(true).Count;
            }
            else
            {
```

```
            maxMobility = game.PossiblePlays(true).Count;
            minMobility = game.PossiblePlays().Count;
        }

        if ((maxMobility + minMobility) > 0)
        {
            return 100 * (maxMobility - minMobility) / (maxMobility + minMobility);
        }
        else
        {
            return 0;
        }

    }

    // Calculates a score based on how many more corners one player has than the other

    public static int CornersHeuristic(Game game, TileColor color)
    {
        TileColor currentPlayer = game.IsFirstPlayer ? TileColor.BLACK : TileColor.WHITE;

        List<Tuple<int, int>> corners = new()
        {
            Tuple.Create(0, 0),
            Tuple.Create((int) game.Board.Size - 1, 0),
            Tuple.Create(0, (int) game.Board.Size - 1),
            Tuple.Create((int) game.Board.Size - 1, (int) game.Board.Size - 1)
        };

        int maxScore = 0, minScore = 0;
        foreach (Tuple<int, int> corner in corners)
        {
            if (game.Board[corner.Item1, corner.Item2] != null)
            {
                if (game.Board[corner.Item1, corner.Item2].color == color)
                {
                    maxScore++;
                }
                else if (game.Board[corner.Item1, corner.Item2].color != TileColor.BLANK)
                {
                    minScore++;
                }
            }
        }
        int score = 0;
        if (maxScore + minScore > 0) score = (100 * (maxScore - minScore)) / (minScore + maxScore);
        return score;
    }

    // Calculates the score of a board based on a set of pre-defined weights of each tile
    public static int WeightedHeuristic(Game game, TileColor color)
    {
        if (game.Size() != 8)
        {
            throw new ArgumentException("The game board must be size 8x8 for the weighted heuristic");
        }

        int[,] weights = new int[(int)game.Size(), (int)game.Size()];
        int score = 0;

        #region weights
        weights[0, 0] = 4;
        weights[0, 1] = -3;
        weights[0, 2] = 2;
        weights[0, 3] = 2;

        weights[1, 0] = -3;
        weights[1, 1] = -4;
        weights[1, 2] = -1;
        weights[1, 3] = -1;

        weights[2, 0] = 2;
        weights[2, 1] = -1;
        weights[2, 2] = 1;
        weights[2, 3] = 0;

        weights[3, 0] = 2;
        weights[3, 1] = -1;
        weights[3, 2] = 0;
        weights[3, 3] = 1;

        #endregion


        for (int i = 0; i < game.Size(); i++)
```

```
            {
                for (int j = 0; j < game.Size(); j++)
                {
                    // Mirrors the weights to all 4 corners

                    int im = i < 4 ? i : 3 - i % 4;
                    int jm = j < 4 ? j : 3 - j % 4;

                    if (game.Board[i, j] != null)
                    {
                        if (game.Board[i, j].color == color)
                        {
                            score += weights[im, jm];
                        }
                        else if (game.Board[i, j].color != TileColor.BLANK)
                        {
                            score -= weights[im, jm];
                        }
                    }
                }
            }

            return score;
        }


        #endregion
```

implementacja metody Minimax z punktu widzenia gracza 1 oraz implementacja alfa-beta cięcia z punktu widzenia gracza 1

```
private Tuple<int, Play, int> AlphaBeta(Game game, int currentDepth = 5, bool prune = true, bool max = true,
int alpha = int.MinValue, int beta = int.MaxValue, int nodesVisited = 0)
        {
            //nie oceniaj możliwych zagrań, jeśli jesteś u podstawy drzewa wyszukiwania
            Dictionary<Tuple<int, int>, Play> possiblePlays = currentDepth != 0 ? game.PossiblePlays() : null;

            // W przypadku wyjścia: wynik nie ulega zmianie z powodu bycia zepchniętym na pozycję bez
możliwości ruchu.
            // wyjście jeżeli: doszliśmy do ograniczenia głębokości || gra zakończona || brak możliwych ruchów
            if (currentDepth == 0 || game.GameOver() || possiblePlays.Count == 0)
            {
                return new Tuple<int, Play, int>(heuristic(game), null!, 1);
            }
            // jeśli gracz jest czarny, to maksymalilzuj
            // w przeciwnym razie, to minimalizuj
            Func<int, int, int> Optimizer = max ? Math.Max : Math.Min;

            // jeżeli maksymalizujemy/czarny to zacznij od najmniejszego i szukaj maks
            // jeżeli minimalizujemy/bialy to zacznij od najwiekszego i szukaj min
            int bestScore = max ? int.MinValue : int.MaxValue;

            // Najwyższa wartość znaleziona do tej pory przez funkcję.
            // Ustaw najniższą możliwą wartość, aby każda wartość była wyższa.
            Play bestPlay = null;
            int nodesVisitedAtThisLevel = 0;
            // Dla każdej możliwej gry z zagrań Użyj Game.ForkGame(Play), aby wygenerować różne gałęzie
            foreach (KeyValuePair<Tuple<int, int>, Play> pair in game.PossiblePlays())
            {
                nodesVisitedAtThisLevel++;
                // Przyjmuje maksimum między gałęzią a bieżącą wartością minimalną
                var alphaBeta = AlphaBeta(game.ForkGame(pair.Value), currentDepth - 1, prune, !max, alpha,
beta);

                int childScore = alphaBeta.Item1;
                int nodes = alphaBeta.Item3;
                //nodesVisitedAtThisLevel += nodes;
                nodesVisited += nodes;

                // Jeśli nowa wartość jest lepsza, zapisz ją i ruch, który ją daje
                if (bestScore != Optimizer(bestScore, childScore))
                {
                    bestPlay = pair.Value;
                    bestScore = childScore;
                }
                //Jeśli wartość zwrócona przez algorytm jest większa niż alfa, zaktualizuj
                //wartość alfa na wartość zwróconą.
                if (max)
                    alpha = Optimizer(alpha, bestScore);
                //Jeśli wartość zwrócona przez algorytm jest mniejsza niż beta, zaktualizuj
                //wartość beta na wartość zwróconą.
                else
                    beta = Optimizer(beta, bestScore);
```

```
                // jeśli wartość beta jest mniejsza lub równa alfa, przerwij przeglądanie po_tomków i zwróć
wartość alfa.
                if (prune && (beta <= alpha))
                    break;
            }
            return new Tuple<int, Play, int>(bestScore, bestPlay, nodesVisited);
        }
```

Trzeba zauważyć, że w przypadku strategii Random, to duża część czasu działania alkgorytmu jest poświęcana na pseudolosowe wygenerowanie kolejnej „losowej liczby”, można byłoby zdecydować się na bardziej primitywną metodę, która niezajmowałany tak, dużo czasu.

Strategia Corners ma tylko znaczenie jeżeli jest możliwość wstawienia krążka w jakiś róg planszy i tylko wyłącznie wtedy. Lepszą strategią jest strategia Weighted, która rozszerza strategię Corners.

modyfikacja programu tak, by wykonywał tylko jeden ruch, co umożliwi rozegranie partii pomiędzy dwoma programami.

```
private void ClickCell(object sender, DataGridViewCellEventArgs e)
        {
            Tuple<int, int> destCoords = Tuple.Create(e.ColumnIndex, e.RowIndex);

            // if there exists a valid play at this coordinate, get object
            playable.TryGetValue(destCoords, out Play p);
            Play humanPlay = manager.OutsidePlay(p);
            if (humanPlay == null) return;
            Game next = manager.Next(blackPrune);
            if (next != null)
            {
                game = next;
            }
            else
            {
                throw new ArgumentException("No human player/other game manager error");
            }
            playable = game.PossiblePlays();
            UpdateBoard();
            if (this.automaticPlay)
            {
                this.NextMove(sender, e);
            }
        }



        private void NextMove(object sender, EventArgs e)
        {
            if (game.Winner != null)
            {
                Console.WriteLine(game.Winner);
                Console.WriteLine($"WHITE: {game.Board.GetNumColor(TileColor.WHITE)}\nBLACK:
{game.Board.GetNumColor(TileColor.BLACK)}");
                lblFullBoard.Text = $"WHITE: {game.Board.GetNumColor(TileColor.WHITE)}\nBLACK:
{game.Board.GetNumColor(TileColor.BLACK)}";
                MessageBox.Show(lblFullBoard.Text, game.Winner.ToString());
                return;
            }
            Game next = manager.Next(blackPrune);
            if (next != null) game = next;
            playable = game.PossiblePlays();
            UpdateBoard();
        }
```

Wyniki

Zdecydowano na wykonanie skryptu umożliwiającego pobranie wyników każdej partii porównującej wyniki przy walce strategii ze strategią. Skrypt wprowadza wyniki do pliku .csv by móc lepiej oraz czytelniej porównywać wyniki. Wyniki są w postaci kolejnych skoroszytów w pliku Z2.xlsx. Trzeba zwrócić uwagę, że dobrze, jest odpowiednio skonfigurować ustawienia uruchomienia programu aby działał on w swojej pełnej możliwości, jeżeli chodzi o wydajność. (Release -> Run Without Debugging)

Wyniki wygenerowane przez skrypt znajdują się w załączonym pliku Z2.xlsx

Count vs Weighted, Depth 4, Prune True

Black Heuristic	White Heuristic	Depth	Prune	Winner	Strategy	Black Score	White Score	Moves	Game Time	Black Time	Black Moves	Black Avg Time	White Time	White Moves	White Avg Time	Black Nodes	White Node
Count	Weighted	4	True	WHITE	Weighted	15	49	60	359	128	30	4.27	231	30	7.7	10330	11821
Weighted	Count	4	True	BLACK	Weighted	34	30	60	703	374	30	12.47	329	30	10.97	19746	22351

Jak widać w obu przypadkach strategia Weighted wygrała mimo zamiany kolejności. Rozpoczynając strategią Weighted partia trwała dłużej i był gorszy wynik ze względu na liczbę krążków. Odwiedzono w tym przypadku prawie dwa razy więcej węzłów drzewa decyzyjnego. Ze względu na zamianę kolejności, stan planszy z każdym krokiem był inny niż w pierwszej partii.

Count vs Weighted, Depth 2, Prune True

Black Heuristic	White Heuristic	Depth	Prune	Winner	Strategy	Black Score	White Score	Moves	Game Time	Black Time	Black Moves	Black Avg Time	White Time	White Moves	White Avg Time	Black Nodes	White Nodes
Count	Weighted	2	True	BLACK	Count	13	0	9	15	15	5	3	0	4	0	58	86
Weighted	Count	2	True	WHITE	Count	31	33	60	62	31	30	1.03	31	30	1.03	1337	1126

W tym przypadku, bardziej „Zachłanna” strategia wygrywa ze strategią Weighted, ze względu na głębość poszukiwań równą 2. Jest to zbyt mała wartość dla realizacji strategii, która jest korzystna ze względu na umiejscowienie krążków podczas całej gry. Dopiero od głębokości 3, Weighted zaczyna wygrywać:

Count vs Weighted, Depth 3, Prune True.

Black Heuristic	White Heuristic	Depth	Prune	Winner	Strategy	Black Score	White Score	Moves	Game Time	Black Time	Black Moves	Black Avg Time	White Time	White Moves	White Avg Time	Black Nodes	White Nodes
Count	Weighted	3	True	WHITE	Weighted	21	43	60	141	62	30	2.07	79	30	2.63	2192	7006
Weighted	Count	3	True	BLACK	Weighted	41	23	60	234	95	31	3.06	139	29	4.79	8155	3976

Count Depth 5 vs Weighted Depth 3, Prune True.

```
Testing: Weighted depth: 3 against Count depth: 5 | prune: True
winner: WHITE
strategy: Count
BLACK (Weighted) 27 - 37 WHITE (Count)
moves: 60  / 60
gameTime: 1641ms
blackTime: 143ms with 30 moves. avg: 4.77ms
whiteTime: 1498ms with 30 moves. avg: 49.93ms
blackNodes: 8359
whiteNodes: 70739
```

```
Testing: Count depth: 5 against Weighted depth: 3 | prune: True
winner: BLACK
strategy: Count
BLACK (Count) 34 - 30 WHITE (Weighted)
moves: 60  / 60
gameTime: 2828ms
blackTime: 2578ms with 29 moves. avg: 88.9ms
whiteTime: 250ms with 31 moves. avg: 8.06ms
blackNodes: 42341
whiteNodes: 6936
```

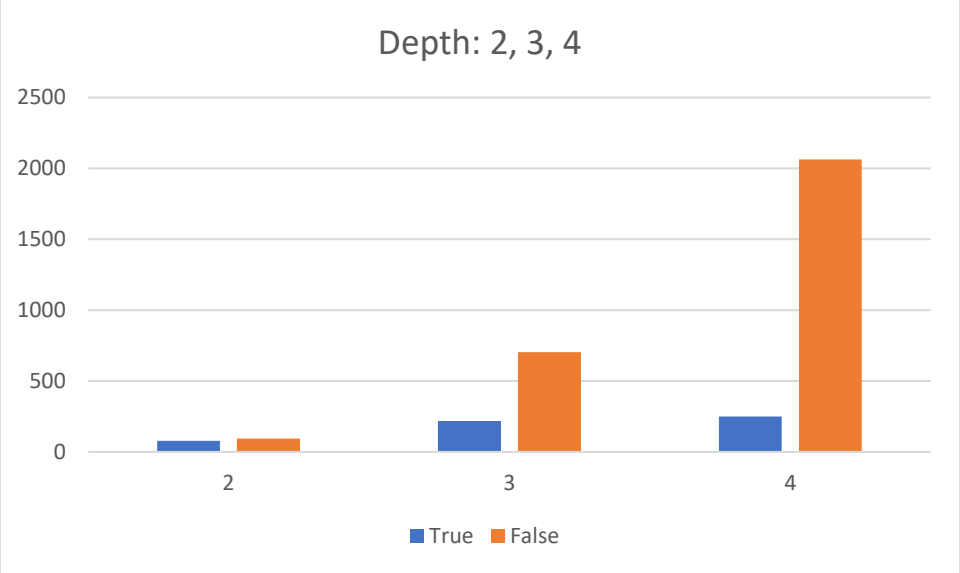
Wyniki przedstawiają, że przy nierównym ustawieniu głębokości przeszukiwania drzewa, dopiero przy Count 5 i Weighted 3, Count będzie w stanie mógł wygrać, z lepszą teoretycznie strategią.

Politechnika Wrocławska

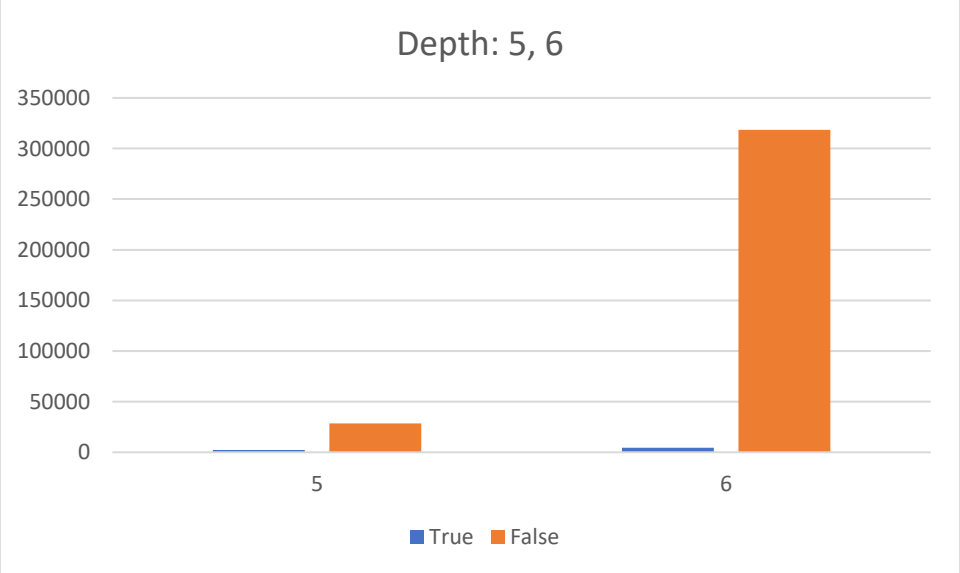
Autor: Filip Strózik 260377 Prowadzący: Mgr Inż. Michał Karol

Porównanie szybkości w zależności od głębokości oraz użycia odcięcia gałęzi (Alpha-Beta pruning). False reprezentuje zwykły Min-Max, a True reprezentuje Alpha-Beta. Wartości w komórkach są czasem całej partii Game Time, wyrażone w ms.

Depth	True [ms]	False [ms]
2	78	93
3	219	703
4	250	2063



5	2125	28500
6	4407	318422



W porównaniu z klasycznym algorytmem Min-Max, który przeszukuje całe drzewo gry, Alpha-Beta Pruning przycina gałęzie drzewa, które już wiemy, że nie prowadzą do najlepszego wyniku.

Dzięki temu Alpha-Beta jest bardziej wydajny niż Min-Max, ponieważ redukuje liczbę węzłów, które muszą być przeszukane. Może to zwiększyć szybkość obliczeń i umożliwić graczom szukanie optymalnych ruchów w bardziej złożonych grach.

Wyniki rozgrywek są takie same, to znaczy, że użycie Alpha-Beta Pruning'u wpływa tylko na przeszukiwanie drzewa rozwiązań.

Wnioski

Chcąc zrealizować grę reversi, w której rzeczywiście będzie można przeprowadzić partię gry zarówno z człowiekiem jak i z komputerem, zdecydowano się na implementację interfejsu użytkownika który by na to pozwalał. Niestety podejście obiektowe, sprawiło że nie osiągnięto najwyższej wydajności ze względu na złożoność powiązań między obiektami które tworzymy podczas gry. Gdybyśmy zdecydowali się na rozwiązanie tylko konsolowe, to stracilibyśmy na prostocie użytkowania tego interfejsu, aczkolwiek struktura programu miałaby niższy poziom abstrakcji oraz program byłby wydajniejszy.

Potwierdzono że istnieją lepsze strategię które wygrywają nad tymi gorszymi. Potwierdzono że gorsze strategie z większą głębokością przeszukiwań drzewa dla algorytmu Min-Max mogą wygrać z lepszym i strategiami które posiadają głębokość drzewa przeszukiwań.

Dla optymalnego przeszukiwania drzewa Alpha-Beta Pruning jest czymś niezbędnym. Pozwala on odciąć gałęzie dla których wiemy że nie prowadzą dla najlepszego wyniku przez co redukuje liczbę węzłów i pozytywnie czas znalezionego rozwiązania.

Podczas realizacji samego zadania odkryto narzędzie deweloperskie wbudowanym w Visual Studio, które skutecznie pozwala zidentyfikować wywołania metod, które zajmują najwięcej czasu, jest to bardzo przydatne w celach optymalizacji własnej implementacji.