Laboratorium 1 - Przeszukiwanie grafów (sudoku, puzzle $n^2 - 1$)

Marcin Pietrzykowski

mpietrzykowski@wi.zut.edu.pl

wersja 1.10

1 Algorytmy przeszukiwania grafów

Celem laboratorium jest zapoznanie się za algorytmami Best First Search, A*, językiem programowania C# oraz napisanie w języku C# programów rozwiązujących Sudoku oraz puzzle przesuwne w oparciu o klasy bazowe. Za dwa programy wystawiana jest jedna ocena łączna. Pojedynczy program nie będzie oceniany. W celu wykonania zadania należy pobrać rozwiązanie Visual Studio zwierające klasy bazowe z implementację algorytmów przeszukiwania. Pliki należy pobrać stąd. **Nie należy modyfikować pobranych plików z klasami bazowymi znajdującymi się w rozwiązaniu. Jedynym plikiem, który należy uzupełnić jest plik `Program.cs`.**

W pobranym archiwum znajduje się katalog rozwiązania (*ang. Solution*) Visual Studio. Pojedyncze rozwiązania może zawierać w sobie wiele projektów np.: bibliotekę dll, drugą bibliotekę dll, aplikację konsolową, aplikację okienkową itd. W archiwum znajdują się również przykładowe implementacje programów o rozszerzonej funkcjonalności. **Dodatkowe funkcjonalności w zaprezentowanych programach nie są wymagane od studenta. Podstawą oceny są polecenia zawarte poniżej.** W pobranym rozwiązaniu znajdują się następujące pliki.:

IState.cs Interfejs zawierający w sobie deklaracje metod i właściwości które później będą używane przez klasę `AStarSearch.cs` i `BestFirstSearch.cs`.

State.cs Klasa abstrakcyjna dziedzicząca po interfejsie `IState.cs` zawierający w sobie częściową implementację metod i właściwości używanych przez `AStarSearch.cs` i `BestFirstSearch.cs`.

AStarSearch.cs Klasa abstrakcyjna implementująca algorytm A*. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.

BestFirstSearch.cs Klasa abstrakcyjna implementująca algorytm Best First Search. W trakcie działania operuje na stanach implementujących interfejs `IState.cs`.

PriorityQueue.cs Implementacja kolejki priorytetowej na kopcu binarnym pozwalająca na szybkie sprawdzenie czy dany element istnieje oraz aktualizację elementu. Używana w klasie `AStarSearch.cs`.

SimplePriorityQueue.cs Implementacja kolejki priorytetowej na kopcu binarnym pozwalająca na szybkie sprawdzenie czy dany element istnieje. Używana w klasie `BestFirstSearch.cs`.

Program.cs Plik główny programu zawierający w sobie metodę `Main`.

Uwaga! Korzystanie z nowych wersji plików opisanych powyżej jest obligatoryjne do wykonania zadania. Korzystanie ze starych plików źródłowych będzie traktowane jako plagiat bez możliwości poprawy zadania.

2 Algorytm Best First Search i program rozwiązujący Sudoku

Celem zadania jest zapoznanie się za algorytmem Best First Search, oraz napisanie w języku C# programu rozwiązującego Sudoku w oparciu o klasy bazowe.

2.1 Implementacja

W trakcie implementacji należy utworzyć dwie klasy potomne `SudokuState.cs` i `SudokuSearch.cs` dziedziczące odpowiednio po klasach bazowych `State.cs` i `BestFirstSearch.cs`. Klasa `SudokuState` będzie reprezentować pojedynczy stan planszy sudoku. Zadaniem klasa `SudokuSearch` będzie zastosowanie algorytmu Best First Search do rozwiązania konkretnej planszy Sudoku. Dobra praktyka programowania mówi, że pojedynczy plik powinien zawierać w sobie implementację pojedynczej klasy. Najpierw należy wczytać plik rozwiązania `Laboratory1.sln` w Visual Studio. Aby dodać klasę do projektu należy kliknąć PPM na Projekcie *Laboratory1* w *Solution Explorer* w *Visual Studio*. Projekt będzie pogrubiony i będzie znajdował się poniżej rozwiązania *Solution 'Laboratory1'*. Następnie należy kliknąć Add → New Item. W otworzonym oknie dialogowym zaznaczyć *Class*, oraz wpisać odpowiednią nazwę w polu *Name:* i kliknąć *Add*.

2.1.1 SudokuState.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej.

```
1  public class SudokuState : State
```

Następnie należy kliknąć PPM na nazwie klasy bazowej (`State`) i wybrać opcję *Implement Abstract Class*. W przypadku nowszej wersji Visual Studio należy skorzystać z opcji Refactor, w której znajduje się równoważne polecenie. W przypadku gdy w Visual Studio taka opcja nie działa, metody znajdujące się w ramce poniżej należy dodać ręcznie. W wyniku tego działania zostały utworzone dwa szablony wymagające samodzielnego wypełnienia. Jest to metoda abstrakcyjna i właściwość abstrakcyjna, które muszą zostać zaimplementowana w klasie potomnej. Szablony zawierają automatycznie dodane wyjątki `throw new NotImplementedException()`, należy je usunąć i zastąpić właściwym kodem.

```
1  public override string ID {
2          get { throw new NotImplementedException(); }
3  }
4
5  public override double ComputeHeuristicGrade() {
6          throw new NotImplementedException();
7  }
```

Właściwość ID Na temat czym są właściwości odsyłam do punkt 4.3. Ta konkretna właściwość ma na celu zwrócenie string'a jednoznacznie identyfikującego konkretny stan planszy sudoku. Nie powinno dochodzić do konfliktów czyli dwa odmienne stany powinny posiadać dwie różne wartości **ID**. Natomiast dwa stany reprezentujące ten sam układ na planszy ale będące dwoma różnymi instancjami klasy powinny zwracać tą samą wartość **ID**. Proponuje się zaimplementowanie właściwości w następujący sposób. Uwaga! Do kodu nie należy ponownie dodawać właściwości `public override string ID`, a jedynie usunąć `throw new NotImplementedException()` i zastąpić je poprawnym kodem tj. `return this.id;`

```
1  private string id;
2
3  public override string ID {
4          get { return this.id; }
5  }
```

Gdzie `private string id` jest polem klasy inicjalizowanym w konstruktorze.

ComputeHeuristicGrade Metoda ma na celu zwrócenie wartości heurystyki dla konkretnego stanu sudoku. Heurystyki zostały omówione na wykładzie. **Uwaga!** Dla niektórych heurystyk umieszczających potomków jedynie w polach o najmniejszej liczbie możliwości, część logiki powinna zostać przeniesiona do metody `buildChildren` (Podrozdział 2.1.2).

Print W klasie należy również dodać metodę `Print` wyświetlającą stan sudoku na ekranie. Bezwzględnie wymaga się aby metoda wyświetlała stan w postaci macierzy 9x9 w czytelny sposób tj. wszystkie puste pola (zawierające 0) muszą być zamieniane na spacje, ewentualnie nowo wstawiany stan powinien być wyświetlany innym kolorem. W razie potrzeby należy zwiększyć bufor konsoli aby wszystkie możliwe stany mogły zostać wyświetlone. Można zrobić to ręcznie albo umieszczając w metodzie `Main` polecenie `Console.BufferHeight = 1000;`.

```
public void Print() {
        //...
}
```

Pola klasy i właściwości Proponuje się oprócz właściwości zdefiniowanych powyżej dodać do klasy następujące pola i właściwości:

```
public class SudokuState : State {
        public const int SMALL_GRID_SIZE = 3;

        public const int GRID_SIZE = SMALL_GRID_SIZE * SMALL_GRID_SIZE;

        private int[,] table;

        public int[,] Table {
                get { return this.table; }
                set { this.table = value; }
        }

        // reszta implementacji
}
```

Konstruktory W klasie nie może zabraknąć konstruktorów. Do poprawnej implementacji potrzebne będą dwa konstruktory. Pierwszy przyjmujący stringa np.: postaci "00070080000004003..." reprezentującego początkowy stan sudoku. Drugi konstruktor jest odpowiedzialny za utworzenie potomka stanu podanego w parametrze o wartościach podanych w parametrze. Części kodu `: base()` i `:base(parent)` są odpowiedzialne za wywołanie konstruktora z klasy bazowej.

```
public SudokuState(string sudokuPattern) : base() {
        if (sudokuPattern.Length != GRID_SIZE * GRID_SIZE) {
        throw new ArgumentException("SudokuSring posiada niewlasciwa
           dlugosc.");
        }

        //utworzenie id
        this.id = sudokuPattern;
        //alokacja i wypelnienie tablicy przechowujacej stan sudoku
        this.table = new int[GRID_SIZE, GRID_SIZE];

        for(int i = 0; i < GRID_SIZE; ++i) {
                for(int j = 0; j < GRID_SIZE; ++j) {
                        this.table[i, j] = sudokuPattern[i * GRID_SIZE + j]
                           - 48;
                }
        }

        //obliczenie heurystyki
        this.h = ComputeHeuristicGrade();
```

```
}

public SudokuState(SudokuState parent, int newValue, int x, int y) :
    base(parent) {
        this.table = new int[GRID_SIZE, GRID_SIZE];
        //Skopiowanie stanu sudoku do nowej tabeli
        Array.Copy(parent.table, this.table, this.table.Length);
        //Ustawienie nowej wartosci w wybranym polu sudoku
        this.table[x, y] = newValue;

        //Utworzenie nowego id odpowiadajacemu aktualnemu stanowi planszy
        StringBuilder builder = new StringBuilder(parent.id);
        builder[x*GRID_SIZE + y] = (char)(newValue + 48);
        this.id = builder.ToString();

        this.h = ComputeHeuristicGrade();
}
```

2.1.2 SudokuSearch.cs

Nowo utworzoną pustą klasę należy zdefiniować jako publiczną i dziedziczącą po klasie bazowej. Oraz zaimplementować metody abstrakcyjne.

```
public class SudokuSearch : BestFirstSearch
```

Pola klasy Klasa nie będzie wymagała żadnych dodatkowych pól.

Konstruktor W klasie wystarczy zdefiniować pusty konstruktor

```
public SudokuSearch(SudokuState state) : base(state) { }
```

Metody Wymagane jest zaimplementowanie dwóch metod abstrakcyjnych. Metoda `isSolution` zwraca informację czy dany stan jest stanem końcowym. Metoda `buildChildren` ma za zadanie zbudowanie potomków wybranego stanu. Poniżej przestawiona jest podstawowa wersja metody działająca dla heurystyki naiwnej. Buduje one potomków w oparciu o pierwsze napotkane puste pole. W przypadku zaimplementowania bardziej zaawansowanej heurystyki (np. heurystyka wg minimum pozostałych możliwości) wymagane jest najpierw odnalezienie odpowiedniego pola na planszy sudoku, w którym wartość heurystyki jest najmniejsza, a następnie wstawiać potomków w tym polu.

```
protected override void buildChildren(IState parent) {
    SudokuState state = (SudokuState)parent;

        //poszukiwanie wolnego pola
    for (int i = 0; i < SudokuState.GRID_SIZE; ++i) {
        for (int j = 0; j < SudokuState.GRID_SIZE; ++j) {
            if (state.Table[i, j] == 0) {
                //wstawianie kolejnych potomkow w wolne pole
                for (int k = 1; k < SudokuState.GRID_SIZE + 1; ++k) {
                    SudokuState child = new SudokuState(state, k, i, j);
                    parent.Children.Add(child);
                }
                return;
            }
        }
    }
}

protected override bool isSolution(IState state) {
    return state.H == 0.0;
}
```

Rozważmy sobie stan reprezentujący planszę sudoku, która ma następującą postać:

1 5 7 9 4	...	...
...	...	...
...	...	...

Jako stany potomne ww. planszy rozumiemy następujące plansze sudoku:

1 1 5 7 9 4	...	...
...	...	...
...	...	...

2 1 5 7 9 4	...	...
...	...	...
...	...	...

3 1 5 7 9 4	...	...
...	...	...
...	...	...

...

9 1 5 7 9 4	...	...
...	...	...
...	...	...

Należy mieć na uwadze, że ww. przykładzie nie wszyscy utworzeni potomkowie będą poprawnymi stanami gry sudoku. W niektórych z nich nowo wstawiona cyfra w polu *(i, j)*, będzie już występowała w danym wierszu, kolumnie lub małym kwadracie. Tego typu stanom metoda `ComputeHeuristicGrade` powinna nadać wartość heurystyki równą $+\infty$ (`double.PositiveInfinity`). Potomków możne generować w oparciu o inne pole *(i, j)* a w przypadku bardziej skomplikowanych heurystyk jest to wymagane. **Uwaga!** potomków zawsze podpinamy w jedno puste pole dlatego każdy ze stanów będzie posiadał maksymalnie tylko 9 potomków.

2.1.3 Program.cs

Klasa `Program.cs` zawiera metodę `Main`. W metodzie Main należy wyświetlić kolejne stany Sudoku. prowadzące do rozwiązania. Można zrobić to w następujący sposób:

```
1  static void Main(string[] args) {
2          string sudokuPattern = "010330218..."; \\sudoku w postaci stringa
               powinno zawierac 81 cyfr
3
4          SudokuState startState = new SudokuState(sudokuPattern);
5          SudokuSearch searcher = new SudokuSearch(startState);
6          searcher.DoSearch();
7
8          IState state = searcher.Solutions[0];
9
10         List<SudokuState> solutionPath = new List<SudokuState>();
11
12         while (state != null) {
13                 solutionPath.Add((SudokuState)state);
14                 state = state.Parent;
15         }
16         solutionPath.Reverse();
17
18         foreach(SudokuState s in solutionPath){
19                 s.Print();
20         }
21 }
```

2.2 Praca samodzielna

Aby dokończyć zadanie należy, samodzielnie zaimplementować heurystykę. Heurystyka naiwna będzie w stanie rozwiązać, tylko najprostsze stany sudoku w skończonym czasie, dlatego wymagane jest zaimplementowanie heurystyki bardziej skomplikowanej: heurystyka wg minimum pozostałych możliwości lub heurystyka wg sumy pozostałych możliwości. Heurystyki te zostały podane na wykładzie. Dla heurystyk innych od naiwnej wymagane jest zmodyfikowanie metody `buildChildren`.

Należy przetestować działanie algorytmu Best First Search dla kilku przykładowych sudoku (można je znaleźć w internecie wpisując hasło *sudoku string*). Wyświetlić różne informacje na temat pracy algorytmu w momencie zatrzymania: liczba stanów w zbiorach Open i Closed, czas pracy.

Należy również wyświetlić wszystkie stany prowadzące do rozwiązania sudoku. Bezwzględnie wymaga się aby stany były wyświetlane w postaci macierzy 9x9 w czytelny sposób tj. wszystkie puste pola (zawierające 0) muszą być zamieniane na spacje, ewentualnie nowo wstawiany stan powinien być wyświetlany innym kolorem. W razie potrzeby należy zwiększyć bufor konsoli aby wszystkie możliwe stany mogły zostać wyświetlone.

Program rozwiązujący Sudoku powinien implementować dwie heurystyki: naiwną i wybraną przez studenta. Ocena za program, który będzie implementował tylko heurystykę naiwną będzie obniżona.

2.3 Sudoku stringi z przykładowego programu dołączonego do kodów źródłowych

- 800030000930007000071520900005010620000050000046080300009076850060100032000040006
- 000000600600700001401005700005900000072140000000000803260000100000068420400029 30
- 457682193600000007100000004200000006584716239300000008800000002700000005926835471
- 000012034000056017000000000000000000048000005127000004800000000350061000760035000
- 000700800000040030000009001600500000010030040005001007500200600030080900007000002
- 100040002050000090008000300000509000700080003000706000007000500090000040600020001
- 600040003010000070005000800000502000300090002000103000008000900070000050200030004
- 000000000000003085001020000000507000004000100090000005000000730020100000000040009
- 000040700080000000010000200008000067000005040000020030207000000000000000006018

3 Algorytm A* i puzzle przesuwne

Celem zadania jest zapoznanie się za algorytmem A*, oraz napisanie w języku C# programu rozwiązującego Puzzle w oparciu o klasy bazowe. W zadaniu tym korzysta się z tych samych klas bazowych co w zadaniu poprzednim.

3.1 Implementacja

W trakcie implementacji należy utworzyć dwie klasy potomne `PuzzleState.cs` i `PuzzleSearch.cs` dziedziczące odpowiednio po klasach bazowych `State.cs` i `AStarSearch.cs`. Implementacja jest analogiczna jak w punkcie poprzednim. Zasadnicza różnica, o której należy pamiętać dotyczy konstruktorów klasy `PuzzleState`, w której należy zdefiniować już przebytą drogą do danego stanu. Informacja ta jest wymagana do poprawnego działania algorytmu A*.

```
1 public PuzzelState(PuzzelState parent, ... /*pozostale parametry*/) :
    base(parent) {
2       //cialo konstruktora
3
4       this.h = ComputeHeuristicGrade();
5       //W stanie potomnym droga ktora przebylismy jest o jeden wieksza
           niz w rodzicu
6       this.g = parent.g + 1;
7 }
```

Jako potomków stanu reprezentującego puzzle o układzie:

2	1	6
	5	7
3	8	4

rozumiemy następujące stany:

	1	6
2	5	7
3	8	4

2	1	6
5		7
3	8	4

2	1	6
3	5	7
	8	4

3.2 Praca samodzielna

Zaimplementować dwie funkcje heurystyczne: "misplaced tiles" oraz "Manhattan".

Wykonać eksperyment porównawczy heurystyk dla 100 problemów losowo pomieszanych przy n = 3. "Losowe" układanki generować rozpoczynając od ułożonej planszy i wykonując 1000 ruchów mieszających nie dbając o ewentualne niwelowanie się ruchów przeciwnych, natomiast dbając o nie zliczanie się ruchów pustych przy brzegach planszy. W eksperymentach mierzyć (w momencie stopu): liczbę stanów w zbiorach Open i Closed, czas wykonania, długość ścieżki. Jako końcowe porównanie podać wartości średnie mierzonych wielkości przypadające na każdą z heurystyk.

Na życzenie prowadzącego mieć możliwość szybkiego nastawienia eksperymentu analogicznego do poprzedniego dla: n = 4, 10 losowych układanek powstałych poprzez 30 ruchów mieszających.

Generowanie zupełnie losowego układu planszy może powodować powstanie planszy, której nie da się rozwiązać. Przykładem takiego układu w planszy 2x2 jest:

1	3
2	

Powyższy układ planszy nie da się doprowadzić do postaci:

1	2
3	

Podobnego typu układy występują w większych planszach. Dlatego zaleca się aby konstruktor tworzący puzzle przyjmował jedynie dwa parametry: rozmiar planszy i liczbę mieszań. Wewnątrz konstruktora powinna zostać utworzona tablica z ułożonymi puzzlami, która następnie zostanie pomieszana.

4 Przydane informacje dla początkujących programistów c-sharpa

W Visual Studio istnieje mechanizm podpowiadania składni *IntelliSense*. Aby wymusić jego wywołania należy wcisnąć kombinację klawiszy **Ctrl+Spacja**. Aby automatycznie przeformatować plik zgodnie ze standardami kodowania (wcięcia, czytelne rozłożenie składni) należy usunąć ostatni nawias klamrowy w pliku **{** i napisać go jeszcze raz. Jeżeli w kodzie nie znajdują się poważniejsze błędy to tekst zostanie przeformatowany zgodnie ze standardami kodowania.

4.1 Tablice

W C# wyróżniamy dwa rodzaje tablic o więcej niż jednym wymiarze. Pierwsze to tablice tablic znane z języków takich jak c++ czy java:

```
1 int[][] tab = new int[10][];
2 for (int i = 0; i < tab.Length; ++i){
3         tab[i] = new int[20];
4 }
5
6 //pobieranie odpowiedniego rozmiaru
7 int l1 = tab.Length;
8 int l2 = tab[0].Length;
```

Drugi rodzaj to tablice wielowymiarowe:

```
1 int[,] tab = new int [10, 20];
2 int i1 = tab.GetLength(0);
3 int i2 = tab.GetLength(1);
```

4.2 String

W C# istniej klasa `string` reprezentująca ciągi znakowe. Należy jednak pamiętać, że jest to klasa tylko do odczytu. W przypadku jakichkolwiek operacji na stringach jak np. konkatenacja nowy string jest tworzony w pamięci. Do wielokrotnych manipulacji na tym samym stringu służy klasa `StringBuilder`, która operuje na jednym obszarze pamięci. Możemy użyć jej np. w następujący sposób.

```
1 StringBuilder builder = new StringBuilder();
2 for(int i = 0; i < 100; ++i){
3         builder.Append(i);
4 }
5 string finalString = builder.ToString();
```

Użycie samej klasy `string` spowodowałoby utworzenie 100 bezużytecznych stringów w pamięci.

4.3 Właściwości

Właściwości są odpowiedzialne za enkapsulację - jedno z założeń programowania obiektowego. Poniższy przykład prezentuje podstawową właściwość. Właściwości mogę być typu zapis-odczyt jak ta poniżej tylko do odczytu (posiadające tylko i wyłącznie część `get { ... }`) jak i tylko do zapisu (posiadające tylko część `set { ... }`).

```
1
2 public class A {
3         private int positiveVal;
4
5         public int PositiveVal {
6                 get { return this.positiveVal; }
7                 set {
8                         if (value <= 0) {
9                                 throw new ArgumentOutOfRangeException();
10                        }
11                        this.positiveVal = value;
12                }
13        }
14 }
```

Z właściwości korzysta się jak z pól klasy, a jednocześnie zachowują się one jak metody.

```
1 A myClass = new A();
2 A.PositiveVal = 10;
3 int i = A.PositiveVal;
4 //spowoduje rzucenie wyjatku
5 A.PositiveVal = -2;
```

Dzieje się tak dlatego, ponieważ w procesie kompilacji są one tak naprawdę zamieniane na metody. W powyższym przypadku będą one miały następującą postać:

```
1 public int PositiveVal_get() { ... }
2 public void PositiveVal_set(int value) { ... }
```

4.4 Współpraca z konsolą

Do współpracy z konsolą służy klasa `Console`.

```
1 //nie wstawia znaku nowej lini na koncu
2 Console.Write("moj napis");
3 //wstawia znak nowej lini na koncu
4 Console.WriteLine("moj napis");
5 string s = Console.ReadLine();
6 int i = Console.Read();
7 ConsoleKeyInfo info = Console.ReadKey();
8 //Ustawienie bufforu konsoli
9 Console.BufferHeight = 1000;
```