Sprawozdanie do Projektu końcowego

Arkadiusz Duliban
Wiktor Gruszczyński

Politechnika Śląska
Wydział Matematyki Stosowanej
Kierunek Informatyka
Inżynierskie, stacjonarne, sem. VII
Rok akademicki 2017/2018
Gliwice

Spis treści

1 Temat projektu **2**

2 Cel projektu **2**

3 Sposób realizacji projektu **2**

4 Komentarz do zadania **2**

5 Algorytmy **4**
5.1 Breadth First Search 4
5.2 Depth First Search 4
5.3 Greedy Best-First Search 5
5.4 A* 5
5.5 Porównanie 6
5.5.1 Przykład 1 6
5.5.2 Przykład 2 6
5.5.3 Przykład 3 7
5.5.4 Przykład 4 7
5.5.5 Przykład 5 7
5.5.6 Podsumowanie 8

6 Implementacja **8**
6.1 Elementy pomocnicze 8
6.1.1 Znajdowanie sąsiadów 8
6.1.2 funkcja isOutOfRange() 8
6.1.3 funkcja includesNode() 8
6.1.4 Klasa "node" 8
6.1.5 funkcja nodeToList() 8
6.1.6 Kolejka priorytetowa 9
6.2 Algorytmy 9
6.3 Heurystyki 10

1 Temat projektu

Tematem projektu jest zaimplementowanie i porównanie kilku przykładowych algorytmów służących do przeszukiwania grafów: przeszukiwanie wszerz (BFS), przeszukiwanie w głąb (DFS), General Best First (GBF) oraz A*.

2 Cel projektu

Naszym celem jest zaimplementowanie algorytmów BFS, DFS, GBF oraz A*. W tym celu stworzone zostanie narzędzie służące do demonstracji działania algorytmów przeszukiwania grafów. Porównywanie będzie się odbywać poprzez znajdowanie drogi między dwoma punktami umieszczonych na planszy składającej się z siatki kwadratów o wymiarze n x n. Dodatkowo niektóre elementy siatki będą mogły zostać wyłączone – tzn. droga nie będzie mogła przeciąć żadnego z nich. Algorytmy zostaną przetestowane dla różnych wariantów danych wejściowych (położenia punktu startowego i końcowego, inna konfiguracja i liczebność przeszkód). Porównane zostaną: liczba iteracji, długość drogi, ilość wyznaczonych sąsiadów.

3 Sposób realizacji projektu

Projekt zostanie przygotowany w formie aplikacji webowej. Użytkownik będzie zadawać parametry wejściowe poprzez modyfikowanie interaktywnego schematu mapy (w postaci siatki). W ten sposób będzie dokonywany wybór punktu startowego, końcowego oraz bloków – przeszkód. Algorytm będzie wywoływany po każdorazowej akcji wykonanej przez użytkownika. Do uruchomienia aplikacji wystarczy przeglądarka internetowa wspierająca obsługę elementu canvas (każda zaktualizowania na przełomie ostatnich 3-4 lat). Uruchomić należy plik **index.html**

4 Komentarz do zadania

1. Stworzono graf – „mapę" do przeszukiwania. Mapa składa się z n wierszy (rowsCount) oraz z m kolumn (boxPerRow). Mapa opisywana jest poprze tablicę net składającą się z nm obiektów, których struktura przedstawia się następująco:

   - x - pozycja poziomo (w pikselach) obiektu na graficznych odwzorowaniu siatki
   - y - pozycja pionowo (w pikselach) obiektu na graficznych odwzorowaniu siatki
   - type - rodzaj bloku (startowy, końcowy, przeszkoda, zwykły)
   - neighId – jeśli blok jest sąsiadem, zmienna przechowuje informacje, który z kolei jest to sąsiad
   - isSearched – czy element był brany pod uwagę przy poszukiwaniach
   - lastSearched – czy element występował jako ostatni przy przeszukiwaniu

   Do prawidłowego działania algorytmu jest wymagana jedynie zmienna type. Pozostałe są użyte jedynie w celu wizualizacji wyniku.

2. Zaimplementowane zostały kolejno wszystkie z algorytmów. Procedury BFS_search, DFS_search jako parametr początkowy przyjmują numer bloku startu oraz numer bloku końca. Procedura GBF_search z kolei wymaga podania dokładnie tych samych parametrów oraz – dodatkowo – referencji do funkcji heurystycznej A lub B. Związane jest to z tym, iż algorytm pozwalamy uruchomić w dwóch konfiguracjach, z różnymi heurystykami.
   Jako wynik otrzymywane są dwie listy:

   - lista zawierająca kolejne numery bloków tworzących wyznaczoną trasę między blokami startu i końca (A i B)
   - lista składająca się z list zawierających identyfikatory sąsiadów punktów wyznaczanych w kolejnych

3. Wyniki algorytmów przedstawiane są na sąsiadujących ze sobą „mapach" – obiektach typu canvas. Każde kliknięcie (akcja) powoduje wywołanie funkcji uruchamiającej algorytm – OnCanvasClick(). Rolę pomocniczą pełni funkcja ChangeSquareType(), która zmienia typ ostatnio kliknięty obiektu: jeśli nie było jeszcze bloku startowego – w start; jeśli nie było końcowego – w końcowy; jeśli jest to blok zwykły – w przeszkodę; jeśli jest to blok inny niż zwykły – zamienia blok w blok zwykły.
   Za rysowanie jest odpowiedzialna funkcja DrawNet, która rysuje siatkę na mapie zadanej jako argument. Wszelkie dane potrzebne do wyrysowywania obiektu pochodzą z danych wyjściowych algorytmu.

4. Dodatkowo skonstruowane narzędzie zostało wyposażone w następujące opcje:

   - Możliwość wyświetlenia na mapie sytuacji aktualnej dla i-tej iteracji. W tym celu wywoływana jest funkcja SetIteration(i) – wyświetlanie wyniku jest ograniczone i nie wyświetlamy wszystkich danych N iteracji, lecz wyświetlanie kończymy na iteracji i.
   - Odtwarzanie animacji przedstawiającej poszukiwanie drogi – realizowane przez funkcję playIterations(), która samoczynnie przestawia numer iteracji od 1 aż do N.
   - Czyszczenie siatki ze zmian wprowadzonych poprzez uruchomienie algorytmu – funkcja EraseCanvas(clearAlsoObjects) odpowiada za zerowanie ustawień do stanu początkowego. Jako parametr przyjmowana jest zmienna logiczna informująca, czy należy również wyczyścić bloki startowe, końcowe i przeszkody.
   - Zmiana rozmiaru siatki net

5. Powyżej ograniczono się jedynie do wymieniania najważniejszych funkcji – dużą część kodu odgrywają funkcje pomocnicze oraz odpowiadające za interakcję z użytkownikiem aplikacji.

5 Algorytmy

5.1 Breadth First Search

Przeszukiwanie wszerz, algorytm który wyszukuje znajdujące się pola jak najbliższej pola startowego. Dzięki temu każde nowe dodane pole wie jaka jest najkrótsza droga by do niego dotrzeć. Dzieje się to jednak kosztem dużej ilości iteracji. Algorytm poszukuje na ślepo w swoim najbliższym sąsiedztwie.

1. Zainicjowanie pamięci: stos, odwiedzone
2. Dodanie punktu początkowego do listy odwiedzonych oraz do stosu
3. Póki stos nie jest pusty:
    - pobieranie punktu ze stosu
    - znajdowanie jego sąsiadów
    - dla każdego z sąsiadów:
        - jeżeli dany punkt nie znajduje się w liście odwiedzonych:
            - jeżeli punkt jest końcem:
                - zwracanie ścieżki
            - w przypadku gdy punkt nie jest na liście odwiedzonych:
                - dodawanie sąsiada na koniec stosu
                - dodawanie sąsiada do listy odwiedzonych

5.2 Depth First Search

Przeszukiwanie wgłąb, algorytm który przeszukuje pola kierujące się w daną stronę, sprawdzając najmłodsze dodawane pola do listy. Skutkiem tego algorytm może łatwo ominąć punkt leżący niedaleko od niego i wykonać sporą ilość iteracji zanim na niego trafi.

1. Zainicjowanie pamięci: stos, odwiedzone
2. Dodanie punktu początkowego do listy odwiedzonych oraz do stosu
3. Póki stos nie jest pusty:
    - pobieranie punktu ze stosu
    - znajdowanie jego sąsiadów
    - dla każdego z sąsiadów:
        - jeżeli dany punkt nie znajduje się w liście odwiedzonych:
            - jeżeli punkt jest końcem:
                - zwracanie ścieżki
            - w przypadku gdy punkt nie jest na liście odwiedzonych:
                - dodawanie sąsiada na początek stosu
                - dodawanie sąsiada do listy odwiedzonych

5.3 Greedy Best-First Search

Przeszukiwanie chciwe, algorytm który wykorzystuje heurystykę do znalezienia najkrótszej drogi w celu przeszukiwania najbardziej obiecujących pól. Pola o największym priorytecie (w przypadku tej implementacji o najmniejszej wartości zmiennej *priority*) są brane wcześniej pod uwagę niż pola o niższym priorytecie. Dzięki temu algorytm kieruje się w stronę celu póki nie napotka przeszkód.

1. Zainicjowanie pamięci: kolejka priorytetowa, odwiedzone
2. Dodanie punktu początkowego do listy odwiedzonych oraz do kolejki
3. Póki kolejka nie jest pusta:
    - zdejmowanie punktu z kolejki z najlepszym priorytetem
    - znajdowanie jego sąsiadów
    - dla każdego z sąsiadów:
        - jeżeli dany punkt nie znajduje się w liście odwiedzonych:
            - jeżeli punkt jest końcem:
                - zwracanie ścieżki
            - w przypadku gdy punkt nie jest na liście odwiedzonych:
                - dodawanie sąsiada do kolejki, z priorytetem równym heurystyce dystansu do punktu docelowego
                - dodawanie sąsiada do listy odwiedzonych

5.4 A*

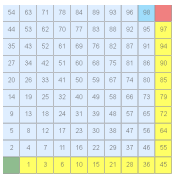
A gwiazdka, algorytm który jest połączeniem algorytmu Dijkstry i chciwego. Wykorzystuje on funkcję kosztu przypisywaną do każdego pola. Algorytm, podobnie jak **GBF**, kieruje się najkrótszą drogą w stronę celu. Jednak jeżeli trafi na przeszkodę i będzie zmuszony się wrócić będzie on dalej w stanie wyznaczyć najkrótszą możliwą trasę. Dzieje się tak dzięki temu, że przy każdym znajdywaniu sąsiadów algorytm jeżeli uzna, że istnieje krótsza droga dojścia do danego pola nadpisuje ją. Sprawia to iż algorytm staje się bardziej czasochłonny lecz zawsze znajduje najkrótszą ścieżkę.

1. Zainicjowanie pamięci: kolejka, odwiedzone
2. Dodawanie punktu początkowego do listy odwiedzonych oraz do stosu
3. Póki kolejka nie jest pusta:
    - pobieranie punkt z kolejki z najlepszym priorytetem
    - nadanie punktowi parametru kosztu - 0
    - znajdowanie jego sąsiadów
    - dla każdego z sąsiadów:
        - jeżeli dany punkt nie znajduje się w liście odwiedzonych:
            - jeżeli punkt jest końcem:
                - zwracanie ścieżki
            - policzenie wartości nowego kosztu = koszt punktu z którego przybył + heurystyka dystansu pomiędzy znalezionym sąsiadem a punktem z którego przyszedł
            - w przypadku gdy punkt nie jest na liście odwiedzonych lub nowy koszt jest mniejszy od parametru kosztu tego punktu:
                - przypisanie sąsiadowi nowego parametru kosztu
                - dodawanie sąsiada do kolejki, z priorytetem równym jego kosztowi + heurystyce od tego punktu do punktu celu
                - dodawanie sąsiada do listy odwiedzonych

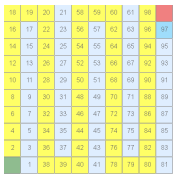
5.5 Porównanie

- **i** - ilość iteracji
- **n** - ilość przeszukanych sąsiadów
- **s** - długość drogi

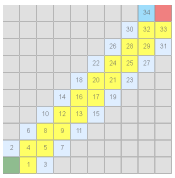
5.5.1 Przykład 1



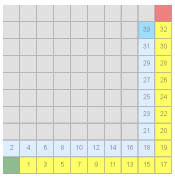
BFS
i=89
n=98
s=19



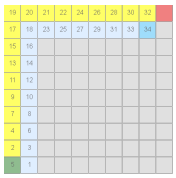
DFS
i=53
n=98
s=55



GBF A
i=17
n=34
s=19



GBF B
i=18
n=33
s=19

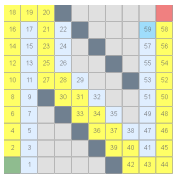


A*
i=18
n=34
s=19

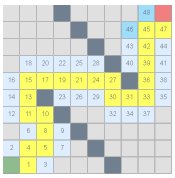
W tym przykładzie pokazana jest droga nie zakłócana żadnymi przeszkodami. Wszystkie algorytmy poza **DFS** były w stanie wyznaczyć najkrótszą ścieżkę. Algorytm **BFS** jednak musiał wykonać dużo iteracji i przeszukać wiele sąsiadów by błądząc trafić do pola końcowego.

5.5.2 Przykład 2

BFS
i=64
n=82
s=21

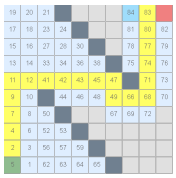


DFS
i=36
n=59
s=37



GBF A
i=21
n=48
s=23

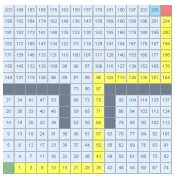
GBF B
i=25
n=49
s=25



A*
i=51
n=84
s=21

Ten przykład pokazuje jak potencjalnie zachowują się algorytmy napotykając przeszkodę w postaci ściany. Żadnemu z algorytmów nie zawsze znajdujących najkrótszą drogę (DFS, GBF) nie był w stanie wyznaczyć najkrótszej ścieżki. Algorytm **DFS** zgodnie ze swoim założeniem - sprawdza najmłodsze pola dodawane do stosu: najpierw pola w dół, następnie w górę, prawo i dopiero w lewo. Dlatego idąc w górę i ścieżka wyznaczona przez algorytm zatrzymuje się i skręca w prawo. Znowu napotykając przeszkodę, tym razem u dołu, postanawia pójść w prawo. Ale jak tylko jest możliwość znowu usiłuje iść w dół. Ten tok rozumowania sprawia, że ten algorytm poza kilkoma przypadkami nigdy nie znajdzie najkrótszej możliwej drogi.
Widoczny jest również problem algorytmów **GBF**, które mają problem gdy należy wrócić się z przeszukiwaniem. Szczególnie dobrze ilustrować będą to kolejne przykłady.

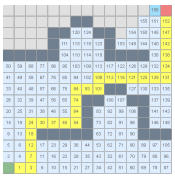
5.5.3 Przykład 3



BFS	DFS	GBF A	GBF B	A*
i=179	i=59	i=32	i=51	i=43
n=205	n=112	n=60	n=90	n=84
s=29	s=59	s=29	s=43	s=29

W tym przypadku jeszcze algorytmowi **GBF A** z heurystyką Euklidesa udaje się otrzymać ścieżkę z najkrótszą drogą przebytą. Dzieję się to dlatego iż przeszkody jedynie przecinają drogę do punktu końcowego wyznaczoną przez heurystykę algorytmu. Gorzej to wygląda w przypadku **GBF B**, gdzie użyty został dystans Manhattana. Ścieżka zwrócona przez algorytm musi się sporo wrócić.
Bardzo dobrze prezentuje się tu algorytm **A***. Widać, że sprawdza on wszystkie pola we wgłębieniu a następnie efektownie wycofuje się z niego dając najkrótszą ścieżkę.

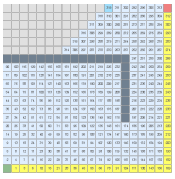
5.5.4 Przykład 4



BFS	DFS	GBF A	GBF B	A*
i=123	i=90	i=60	i=60	i=37
n=156	n=154	n=101	n=104	n=29
s=29	s=49	s=29	s=55	s=29

Implementacje **DFS** oraz **GBF B** znowu błądzą po mapie. Tutaj dobrze widać przewagę **A*** ponad pozostałymi algorytmami - nie robi on aż tyle zbędnych przeszukiwań i zawsze jest skuteczny.

5.5.5 Przykład 5



BFS	DFS	GBF A	GBF B	A*
i=292	i=127	i=74	i=38	i=136
n=319	n=149	n=120	n=73	n=261
s=39	s=121	s=51	s=39	s=39

Znów widoczna jest wada algorytmów typu **GBF** - póki heurystyka nie napotka przeszkody takiej, która nie zmusza algorytmu do wracania z przeszukiwaniem algorytmy będą znajdować najkrótszą ścieżkę.
A* rozwiązuje ten problem robiąc porównania ścieżek prowadzących do danego pola od początku wielokrotnie, zawsze ustalając najkrótszą drogę dojścia do każdego pola.

5.5.6 Podsumowanie

	BFS	DFS	GBF	A*
zawsze najkrótsza możliwa ścieżka	✓	X	X	✓

Moim zdaniem najlepszym algorytmem jest **A***. Znajduje on bowiem zawsze najkrótszą drogę, a to jest z reguły naszym zadaniem. Wykorzystuje on wskazywanie ogólnego kierunku drogi jak w przypadku **GBF** ale potrafi wyjść z "wklęsłości" bez stracenia na długości ścieżki. Algorytmy jak **DFS** oraz **GBF** nie dadzą najkrótszej ścieżki chyba, że będą mieć szczęście. **BFS** oraz A* to gwarantują, tylko **A*** robi to mniejszym kosztem (w większości przypadków).

6 Implementacja

Idea algorytmów jest prosta czego nie można powiedzieć o ich implementacji. W celu ułatwienia pracy stworzony został szereg pomocniczych funkcji i klas.

6.1 Elementy pomocnicze

6.1.1 Znajdowanie sąsiadów

Funkcja odpowiedzialna za znajdowanie sąsiadów *getNeighbours()* sprawdza czy na czterech polach można utworzyć ścieżkę. Robi to poprzez sprawdzenie czy pole: nad, po lewej, po prawej oraz pod zadanym w argumencie polem znajdują się w obrębie planszy oraz czy nie stoi na nich przeszkoda. Jako, że tablica w której przetrzymywane są dane na ten temat jest jednowymiarowa odbywa się to poprzez dzielenie modulo pozycji pola poprzez szerokość planszy.
Funkcja zwraca listę dostępnych pól.

6.1.2 funkcja isOutOfRange()

Funkcja sprawdza czy zadany identyfikator pola znajduje się na mapie.

6.1.3 funkcja includesNode()

Funkcja sprawdza czy w danej liście znajduje się obiekt typu *node*.

6.1.4 Klasa "node"

Korzystając z dobrodziejstw języka *JavaScript* utworzona została pomocnicza klasa *node*, która miała oznaczać przeszukiwane pole. Podstawowymi atrybutami tej klasy są:

- **id** - oznaczające id danego pola w tablicy *net*
- **last** - będące odnośnikiem do *node'a* z którego ten został wywołany

Klasa ta ułatwia odtwarzanie ścieżki utworzonej poprzez algorytmy.

6.1.5 funkcja nodeToList()

Funkcja ta odpowiedzialna jest za stworzenie listy identyfikatorów pól z listy obiektów typu *node*. W argumencie podawany jest *node*, z którego pobierany jest *id* parametru *last* i dodawany do listy zwrotnej. To samo dzieje się dla obiektu last i tak do końca.
W efekcie funkcja zwraca listę identyfikatorów ścieżki utworzonej z poprzedników *node'a* podanego w argumencie.

6.1.6 Kolejka priorytetowa

Nasza implementacja nie jest do końca kolejką priorytetową tylko ją symuluje.

1. Tworzona jest lista **L**
2. Pętla:
   - **L** jest odwracane
   - **L** jest sortowane po priorytecie
   - Operacje:
     - ...
     - dodaj element do **L** z danym priorytetem
     - ...

Lista jest odwracana by obiekty z najniższą liczbą priorytetową, czyli największym priorytecie, zawsze są brane pod uwagę wcześniej.
Początkowy algorytm działał źle gdyż nie brał on pod uwagi wieku dodanych elementów. Dlatego wymyśliłem sztuczne obejście, które dodaje wiek elementom:

1. Tworzona jest lista **L**
2. Inicjuj iterator
3. Pętla:
   - **L** jest odwracane
   - **L** jest sortowane po priorytecie
   - Operacje:
     - iterator++
     - ...
     - dodaj element do **L** z danym priorytetem + iterator*0.001
     - ...

6.2 Algorytmy

Wszystkie algorytmy mają wspólne cechy:

1. każdy algorytm zwraca:
   - ścieżkę od początku do końca (jeżeli istnieje)
   - historię przeszukiwanych pól
2. w argumentach wszystkich funkcji algorytmów podany jest początek i koniec, a w przypadku algorytmu **GBF** jeszcze typ heurystyki.
3. ścieżka jest tworzona poprzez dodawanie do danego node'a parametru last, który ustanawia z którego pola można dojść do niego
4. każde odwiedzone pole jest dodawane do historii przeszukiwanych pól

6.3 Heurystyki

- Heurystyka A - dystans Manhattana

$$|bx - ax| + |by - ay|$$

- Heurystyka B - dystans Euklidesa

$$(bx - ax)^2 + (by - ay)^2$$