Analiza Algorytmów – Dokumentacja końcowa

Tomasz Bocheński, 261416

1. Treść zadania:

Najkrótsza droga w mieście.

Dany jest raster MxN o polach białych i czarnych. Opracować algorytm, który znajdzie najkrótszą drogę z białego pola A do białego pola B, pod warunkiem, że można się poruszać jedynie w pionie i w poziomie omijając przy tym pola czarne. Przy generacji danych należy zwrócić uwagę, aby z każdego pola białego można było się potencjalnie przedostać do dowolnego innego pola o tym kolorze. Porównać czas obliczeń i wyniki różnych metod.

2. Proponowane rozwiązania (algorytmy):

W projekcie tym powinienem zaimplementować co najmniej dwa różne sposoby rozwiązywania zadanego problemu.

W każdym z rozwiązań raster będzie traktowany jako graf o wierzchołkach reprezentujących pola rastra oraz krawędziach reprezentujących przejścia między polami. Rozważam następujące sposoby rozwiązanie tego problemu:

- Wykorzystanie algorytmu Bellmana-Forda.
  Złożoność czasowa O(|V| * |E|), gdzie |V| - liczba wierzchołków, |E| - liczba krawędzi.
- Wykorzystanie algorytmu Dijkstry.
  Złożoność tego algorytmu zależy od sposobu implementacji kolejki priorytetowej:
  - za pomocą zwykłej tablicy, złożoność czasowa wynosi $O(|V|^2)$;
  - za pomocą kopca, złożoność czasowa wynosi O(|E| * log|V|);
  - za pomocą kopca Fibonnaciego, złożoność czasowa wynosi O(|E| + |V| * log|V|).
- Wykorzystanie algorytmu A* (algorytm heurystyczny), gdzie kolejka priorytetowa zaimplementowana jest za pomocą kopca.
  Złożoność czasowa wynosi O(|E| * log|V|), zatem tyle samo ile złożoność czasowa algorytmu Dijkstry z wykorzystaniem kopca. Można zauważyć, że algorytm Dijkstry jest najgorszym przypadkiem algorytmu A*.

Zaimplementowane zostaną co najmniej dwa z wyżej wymienionych algorytmów.

3. Opis algorytmu Bellmana-Forda:

Idea tego algorytmu opiera się na metodzie relaksacji. Metoda relaksacji krawędzi polega na sprawdzeniu, czy przy przejściu daną krawędzią grafu (u,v) z 'u' do 'v', nie otrzymamy krótszej niż dotychczasowa ścieżki z 's' do 'v'. Jeśli tak, to odpowiednio zmniejszamy oszacowanie wagi najkrótszej ścieżki.

Pseudokod, schemat działania:

```
inicjalizuj wszystkie elementy tablicy d[ilosc pol] wartoscia nieskonczonosc
// tablica ta przechowuje odleglosci kazdego pola od pola poczatkowego
inicjalizuj wszystkie elementy tablicy poprzednicy[ilosc pol] wartoscia -1 4
d[poczatek] = 0
for(I od 1 do ilosc pol - 1)
{
    moznaKonczyc = true
    for(X od 0 do ilosc pol -1)
    {
        if (X jest polem niedozwolonym)
            continue
        for(kazdy z sasiadow X w poziomie i w pionie)
        {
            if(d[sasiad] <= d[X] + 1 lub sasiad jest polem niedozwolonym)
                nie rob nic
            else
            {
                moznaKonczyc = false
                d[sasiad] = d[X] + 1
                poprzednik[sasiad] = X
            }
        }
        if(moznaKonczyc jest true)
            zakoncz wykonywanie funkcji, wyjdz z funkcji
    }
}
```

Złożoność czasowa algorytmu:

Łatwo zauważyć, że złożoność czasowa tego algorytmu wynosi O(|V| * |E|), gdzie |V| - liczba wierzchołków, |E| - liczba krawędzi.

Pierwsza pętla for powtarzana jest w przybliżeniu tyle razy, ile jest pól w rastrze (czyli wierzchołków w grafie). Druga pętla for, zagnieżdżona w pierwszej pętli for, rozpatrywana jest analogiczną liczbę razy. Trzecia pętla for , zagnieżdżona w drugiej pętli for, powtarzana jest tyle razy, ile jest sąsiadów dla danego pola (w moim przypadku powtarzana będzie od 2 do 4 razy). Można zauważyć, że druga i trzecia pętla for równoważna jest pętli, w której rozpatrywane są wszystkie krawędzie w grafie.

4. Opis algorytmu Dijkstry:

Jest to przykład algorytmu zachłannego. Algorytm ten dokonuje decyzji lokalnie optymalnej, a następnie kontynuuje rozwiązanie podproblemu wynikające z podjętej decyzji. Można powiedzieć, że algorytm Dijkstry jest specjalnym przypadkiem algorytmu A*, gdzie parametr H zawsze wynosi 0 (funkcja heurystyczna każdemu argumentowi przyporządkowuje wartość 0).

Pseudokod, schemat działania:

```
inicjalizuj wszystkie elementy tablicy d[ilosc pol] wartoscia nieskonczonosc
// tablica ta przechowuje odleglosci kazdego pola od pola poczatkowego
inicjalizuj wszystkie elementy tablicy poprzednicy[ilosc pol] wartoscia -1
inicjalizuj Q jako zbior wszystkich wierzcholkow
d[poczatek] = 0
while(Q nie jest puste)
{
    wybierz z Q takie pole P, ze d[P] jest najmniejsze
    wyjmij pole P z Q
    if(P jest polem niedozwolonym)
        continue
    if(P jest polem docelowym)
        znaleziono najkrotsza sciezke, wyjdz z funkcji
    for(kazdy z sasiadow P w poziomie i w pionie)
    {
        if(sasiad nie jest w Q lub sasiad jest polem niedozwolonym)
            nie rob nic
        else if(d[sasiad] > d[P] + 1)
        {
            d[sasiad] = d[P] + 1
            p[sasiad] = P
        }
    }
}
```

Złożoność czasowa algorytmu:

Na początku tworzony jest zbiór Q (kolejka priorytetowa). Znajdują się w nim wszystkie pola rastra (wierzchołki grafu). W każdym przebiegu pętli while rozważane jest jedno z pol w Q, pole P o najmniejszej wartości d[P]. Jednocześnie pole to jest usuwane ze zbioru Q. W każdym przebiegu pętli for, zagnieżdżonej w pętli while, sprawdzani są wszyscy sąsiedzi P (jest ich od 2 do 4).

Można więc zauważyć, że bez względu na sposób implementowania kolejki priorytetowej, złożoność czasowa wynosi O(|V| * koszt_insert + |V| * koszt_delete_min + |E| * koszt_decrease_key).

- W przypadku implementacji kolejki priorytetowej jako zwykłej tablicy:
  koszt_insert = O(1), koszt_delete_min = O(|V|), koszt_decrease_key = O(1).
  Złożoność czasowa wynosi $O(|V|^2)$.
- W przypadku implementacji kolejki priorytetowej jako kopca:
  koszt_insert = O(log|V|), koszt_delete_min = O(log|V|), koszt_decrease_key = O(log|V|).
  Złożoność czasowa wynosi O(|E| * log|V|).
- W przypadku implementacji kolejki priorytetowej jako kopca Fibonacciego:
  koszt_insert = O(1), koszt_delete_min = O(log|V|), koszt_decrease_key = O(1).
  Złożoność czasowa wynosi O(|E| + |V| * log|V|).

5. Opis algorytmu A*:

Jest to algorytm heurystyczny. Szuka on najkrótszej drogi łączącej pole startowe z polem końcowym. W pierwszej kolejności sprawdzane są pola, przez które prowadzą potencjalnie najbardziej obiecujące drogi do celu. Jest to zachowanie charakterystyczne dla algorytmów typu Best First Search, w których w pierwszej kolejności rozpatrywane są potencjalnie najlepsze przypadki. Nie jest to algorytm zachłanny.

Oznaczenia:

- G – długość ścieżki od punktu startowego do aktualnie rozpatrywanego punktu (jest to rzeczywista długość, którą już wyznaczyliśmy);
- H – szacunkowa długość ścieżki prowadząca z aktualnie rozpatrywanego punktu do punktu końcowego. Wartość ta jest wyznaczana metodami heurystycznymi;
- F = G + H – suma długości powyższych ścieżek.

Dobór funkcji heurystycznej:

Aby algorytm działał poprawnie, należy dobrać odpowiednią funkcję heurystyczną, czyli funkcje h, która oblicza wartość parametru H. Gwarancją znalezienia optymalnego rozwiązania jest dobranie takiej funkcji h, która dla każdego pola (punktu) niedoszacowuje faktycznej, najkrótszej odległości pola (punktu) od celu.

Przykładowe funkcje heurystyczne jakie mogę użyć w moim zadaniu to:

- funkcja heurystyczna typu Manhattan (odległość dwóch węzłów to suma ich odległości w pionie i w poziomie);
- funkcja heurystyczna oszacowująca odległość dwóch węzłów przez obliczenie standardowej euklidesowej ich odległości.

Ponieważ w moim zadaniu poruszać się można jedynie w pionie i w poziomie (nie można poruszać się na skos), to lepszym wyborem jest funkcja heurystyczna typu Manhattan. Zapewnia ona niedoszacowanie, jednocześnie wyznacza H szybciej niż druga z przedstawionych funkcji.

Złożoność czasowa algorytmu:

Algorytm ten posiada złożoność analogiczną do algorytmu Dijkstry. Można powiedzieć, że algorytm Dijkstry jest najgorszym przypadkiem algorytmu A*. Chociaż posiadają one taką samą złożoność, to ze względu na użytą heurystykę, algorytm A* powinien działać zdecydowanie szybciej.

Pseudokod, schemat działania:

```
inicjalizuj OL, CL
dodaj punkt startowy do OL
while(OL nie jest pusty)
{
    wybierz z OL pole o najmniejszej wartosci F, nazwij je Q
    umiesc pole Q w CL
    if(Q jest polem docelowym)
        znaleziono najkrotsza sciezke, wyjdz z funkcji
    for(kazdy z sasiadow Q w poziomie lub w pionie)
    {
        if(sasiad jest w CL lub sasiad jest zabronionym polem)
            nie rob nic
        else if(sasiad nie znajduje sie w OL)
        {
            przenies go do OL
            Q staje sie rodzicem sasiada
            oblicz wartosci G, H, F sasiada
        }
        else
        {
            oblicz nowa wartosc G sasiada
            if(nowaG < G)
            {
                G = nowaG
                Q staje sie rodzicem sasiada
                oblicz noweF oraz przypisz F = noweF
            }
        }
    }
}
```

6. Generowanie danych testowych:

Generowanie danych testowych będzie polegało na tworzeniu rastrów o odpowiednich rozmiarach

oraz z odpowiednio pokolorowanymi polami (białymi i czarnymi). Użytkownik będzie mógł wybrać procentowy stosunek pól białych do pól czarnych.

Program będzie miał 3 tryby wykonania:

- według danych dostarczanych ze strumienia wejściowego (standardowego lub pliku)
- według danych generowanych automatycznie z parametryzacją procentowego stosunku pól białych do pól czarnych (również możliwość generowania danych „pesymistycznych”)
- wykonanie z generacją danych, pomiarem czasu oraz prezentacją wyników.

7. Implementacja

Projekt został zaimplementowany w C++ (wykorzystany został C++11). Stworzona przeze mnie aplikacja jest aplikacją konsolową. Działa zarówno na Windowsie jak i Linuxie. W celu zbudowania jej na Windowsie należy użyć VS2015. Budowanie jej na Linuxie przebiega w dwóch etapach. Najpierw należy użyć polecenia cmake, które stworzy plik Makefile, a następnie polecenia make.

8. Zaimplementowane Algorytmy:

Ostatecznie udało mi się zaimplementować większość z przedstawionych powyżej algorytmów, mianowicie:

- Algorytm A*
- Algorytm Bellmana-Ford'a
- Algorytm Dijkstry z tablica jako kolejką priorytetowa
- Algorytm Dijkstry z kopcem jako kolejką priorytetową

9. Generator danych

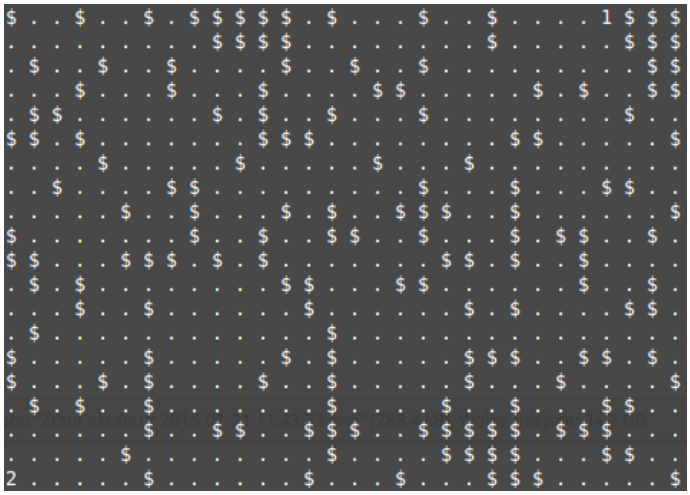
Zadaniem nietrywialnym w moim projekcie była również implementacja generatora danych. Generowane miały być rastry o określonych wymiarach, składające się z białych i czarnych pół. Ponadto musiał być spełniony warunek, aby z każdego pola białego dało się potencjalnie dojść do dowolnego innego pola białego. Algorytm taki został przeze mnie zaimplementowany.

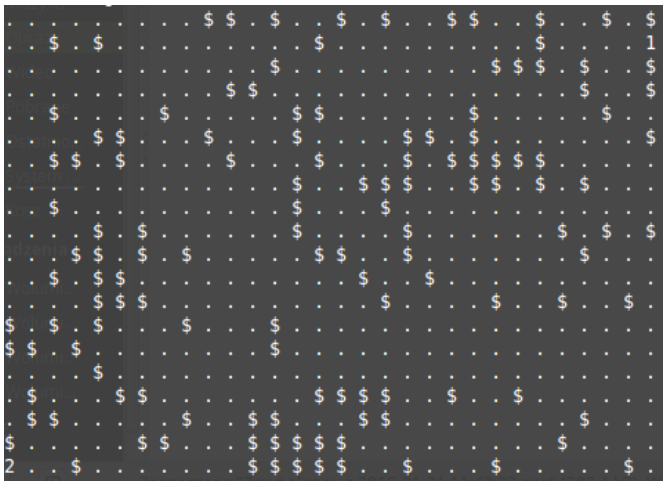
Pseudokod, schemat działania:

```
liczba_bialych_pol = (int)N * M * stosunek_pol_bialych_do_calosci
inicjuj liczba_obecnie_bialych na 0
// pomocnicza macierz w ktorej zaznaczane bedzie czy dane pole zostalo juz
zmienione czy nie
helper[M][N]
// pomocnicza flaga
preparing = true;

wylosuj pierwsze pole białe
zazna
liczba_obecnie_bialych++
zanotuj zmiane na biale w macierzy helper
wloz wylosowane pole do kolejki
wloz wylosowane pole do kolekcjiBialychPol

while(preparing)
{
    wylosuj jedna wartosc z kolejki, przypisz ja do zmiennej Q
    wyjmij Q z kolejki

    for( kazdy sasiad Q)
    {
        if(sasiad jest zaznaczony jako juz zmieniony w helper)
            continue

        // prawdopodobienstwo okreslane jest przez
stosunek_pol_bialych_do_calosci
        kolor = wylosuj z {bialy, czarny} z odpowiednim prawdopodobienstwem

        if(kolor = czarny)
        {
            zmien kolor na czarny
            zanotuj zmiane na kolor czarny w macierzy helper
        }
        else
        {
            zmien kolor na bialy
```

```
                zanotuj zmiane na kolor bialy w macierzy helper
                wloz danego sasiada do kolejki
                wloz danego sasiada do kolekcjiBialychPol
                liczba_obecnie_bialych++

                if (liczba_obecnie_bialych == liczba_bialych_pol)
                {
                    preparing = false
                    break
                }
            }
        }

        if( kolejka jest pusta i liczba_obecnie_bialych < liczba_bialych_pol)
        {
            przejrzyj wszystkie pola macierzy helper, jesli jakies jest czarne to
            ustaw je jako jeszcze nie zmieniane

            wloz do kolejki wszystkie pola z kolekcjiBialychPol
        }
    }

    z kolekcji pol bialych wylosuj pole startowe oraz pole końcowe
    // mozna tu rowniez dodac dodatkowy warunek na minimalna odleglosc pol
    // poczatkowego i koncowego, jednak nie jest to istota tego algorytmu

    na podstawie początku, końca i macierzy pol wygeneruj raster
    zwróć raster
```

Przykłady wygenerowanych rastrów:





Gdzie:

- . oznacza pole białe
- $ oznacza pole czarne
- 1 oznacza początek ścieżki
- 2 oznacza koniec ścieżki

10. Instrukcja obsługi

Program można uruchomić na dwa sposoby. Pierwszy z nich polega na uruchomieniu go z konsoli bez podania żadnych opcji ani parametrów (AAL.exe, ./pathfinding). Wyświetlone wtedy zostaje konsolowe GUI, dzięki któremu użytkownik może wybrać interesująca go działanie.

Drugi sposób aktywacji programu to uruchomienie go z podaniem pewnych opcji i parametrów. Zawsze pierwszym parametrem jest nazwa algorytmu, którym chcemy rozwiązać dany problem. Pierwszy parametr programu jest postaci:

- „-a”: gdy chcemy zastosować algorytm A*;
- „-dh”: gdy chcemy zastosować algorytm Dijkstry z kolejką priorytetową zaimplementowaną jako kopiec;
- „-dt”: gdy chcemy zastosować algorytm Dijkstry z kolejką priorytetową zaimplementowaną jako tablica;
- „-bf”: gdy chcemy zastosować algorytm Bellmana – Forda.

Kolejne opcje i parametry zależą od tego, co chcemy uzyskać. Dozwolone są następujące:

AAL.exe [-a/-dt/-dh/-bf] -f path

./pathfinding [-a/-dt/-dh/-bf] -f path

Program **wyznacza najkrótsza ścieżkę dla rastra wczytanego z pliku**, o ścieżce path.

AAL.exe [-a/-dt/-dh/-bf] -g M N probability

./pathfinding [-a/-dt/-dh/-bf] -g M N probability

Program **wyznacza najkrótsza ścieżkę dla rastra wygenerowanego automatycznie**, przy czym M to liczba wierszy rastra, N to liczba kolumn rastra, natomiast probability to liczba od 0 do 1 określającą prawdopodobieństwo, stosunek, pól białych do wszystkich pól rastra.

AAL.exe [-a/-dt/-dh/-bf] –t initM initN step

./pathfinding [-a/-dt/-dh/-bf] –t initM initN step

Program **generuje tabelkę ze statystykami**. Parametry initM oraz initN określają wymiary rastra początkowego, natomiast step jest to wartość jaka jest dodawana do obu wymiarów rastra przy jego zwiększaniu.

11. Pomiary czasu wykonania:
Ważna uwaga:

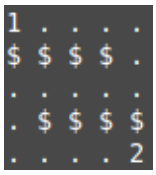
Przy pomiarach czasu wykonania, aby zapewnić wiarygodność pomiarów, w algorytmach wyłączone są wszystkie optymalizacje (są one włączone w każdym innym przypadku).

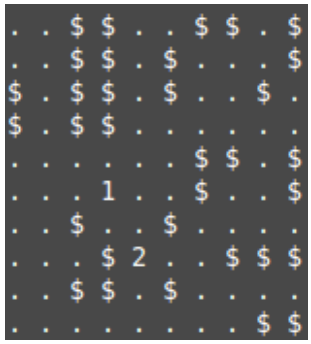
- W algorytmie Bellmana-Forda istnieje warunek, który pozwala przerwać główną pętlę algorytmu, w przypadku gdy w poprzedniej iteracji nie wykryto żadnej nowej krawędzi do relaksacji. Warunek ten zostaje wyłączony.
- W algorytmie Dijkstry istnieje warunek, który pozwala przerwać główną pętlę algorytmu, w przypadku gdy pole dla którego wyznaczono najkrótszą drogę, jest jednocześnie polem końcowym ścieżki. Warunek ten zostaje wyłączony.
- W algorytmie A* istnieje podobny warunek co w algorytmie Dijkstry, ponadto istnieje funkcja heurystyczna. Zostają one wyłączone.

Wytłumaczenie:

Wyłączenie optymalizacji w przypadku pomiarów czasu pozwala zwiększyć wiarygodność wyników. W punkcie tym, dążymy do wyznaczenia czasu działania algorytmu dla danych o różnej wielkości. Po wyłączeniu optymalizacji na czas działania algorytmu ma wpływ tylko wielkość danych. Gdyby optymalizacje nie zostały wyłączone, na czas działania algorytmu miałby również wpływ czynnik losowy, określający położenie początku ścieżki, końca ścieżki oraz rozłożenie pól białych i czarnych. W przypadku znajdowania najkrótszej ścieżki czynnik losowy dominował nad czynnikiem określającym wielkość danych.

Jako przykład porównamy dwa rastry o różnych wielkościach, jeden o wymiarach 5x5 a drugi o wymiarach 10x10.





Jak widać pierwszy raster ma wymiary 5x5, zatem wielkość danych to 25. Długość ścieżki w tym przypadku wynosi 15. Drugi raster ma wymiary 10x10, zatem wielkość danych to 100. Długość ścieżki w tym przypadku wynosi 2. Zatem jak widać, pomimo tego że raster drugi ma 4 razy większy rozmiar danych, ścieżka w nim zostanie znaleziona kilka razy szybciej niż w pierwszym rastrze. Przy generowaniu danych można zawsze dodać warunek na to, żeby długość ścieżki (po linii prostej) była większa niż jakieś minimum (co zrobiłem). Jednak pozwala to wykluczyć tylko najbardziej skrajne, trywialne przypadki. Bardzo dużo zależy również od rozłożenia pól białych i czarnych (łatwo sobie wyobrazić, że nawet przy stosunkowo niedużej odległości między początkiem i końcem w linii prostej, da się tak dobrać kolory pól, aby trasa była bardzo skomplikowana).

Wyłączenie optymalizacji pozwala rozwiązać wszystkie tego typu problemy i w całości uzależnić czas od rozmiaru danych (czyli w całości pozbyć się czynnika losowego).

Przykłady wygenerowanych tabelek:

Bellman-Ford:

```
$ ./pathfinding -bf -t 20 25 2
n =   500    t(n) =            56  T(n) =    250000       q =  0.957365
n =   594    t(n) =            78  T(n) =    352836       q =  0.944825
n =   696    t(n) =           114  T(n) =    484416       q =   1.00581
n =   806    t(n) =           156  T(n) =    649636       q =   1.02632
n =   924    t(n) =           195  T(n) =    853776       q =  0.976158
n =  1050    t(n) =           267  T(n) =   1102500       q =   1.03505
n =  1184    t(n) =           328  T(n) =   1401856       q =          1
n =  1326    t(n) =           411  T(n) =   1758276       q =  0.999043
n =  1476    t(n) =           531  T(n) =   2178576       q =   1.04172
n =  1634    t(n) =           649  T(n) =   2669956       q =   1.03889
n =  1800    t(n) =           814  T(n) =   3240000       q =   1.07376
```

Dijkstra (kolejka priorytetowa jako tablica):

```
$ ./pathfinding -dt -t 90 105 5
n =  9450    t(n) =           262  T(n) =  89302500       q =   1.02798
n =  10450   t(n) =           313  T(n) =  109202500      q =   1.00429
n =  11500   t(n) =           380  T(n) =  132250000      q =   1.00678
n =  12600   t(n) =           454  T(n) =  158760000      q =   1.00199
n =  13750   t(n) =           540  T(n) =  189062500      q =   1.00078
n =  14950   t(n) =           638  T(n) =  223502500      q =    1.0002
n =  16200   t(n) =           749  T(n) =  262440000      q =          1
n =  17500   t(n) =           874  T(n) =  306250000      q =  0.999962
n =  18850   t(n) =          1014  T(n) =  355322500      q =  0.999916
n =  20250   t(n) =          1174  T(n) =  410062500      q =   1.00315
n =  21700   t(n) =          1422  T(n) =  470890000      q =    1.0581
```

Dijkstra (kolejka priorytetowa jako kopiec):

```
$ ./pathfinding -dh -t 450 500 10
n =  225000    t(n) =           206  T(n) =   2772867       q =   1.00963
n =  234600    t(n) =           218  T(n) =   2900978       q =   1.02126
n =  244400    t(n) =           223  T(n) =   3032163       q =  0.999487
n =  254400    t(n) =           227  T(n) =   3166431       q =  0.974273
n =  264600    t(n) =           248  T(n) =   3303788       q =   1.02015
n =  275000    t(n) =           266  T(n) =   3444244       q =   1.04957
n =  285600    t(n) =           264  T(n) =   3587806       q =          1
n =  296400    t(n) =           290  T(n) =   3734481       q =   1.05534
n =  307400    t(n) =           299  T(n) =   3884277       q =   1.04613
n =  318600    t(n) =           294  T(n) =   4037200       q =  0.989674
n =  330000    t(n) =           331  T(n) =   4193259       q =   1.07276
```

A*:

```
$ ./pathfinding -a -t 500 550 10
n =  275000    t(n) =           110  T(n) =   3444244       q =   1.06111
n =  285600    t(n) =           116  T(n) =   3587806       q =   1.07422
n =  296400    t(n) =           113  T(n) =   3734481       q =   1.00534
n =  307400    t(n) =           118  T(n) =   3884277       q =   1.00933
n =  318600    t(n) =           124  T(n) =   4037200       q =   1.02048
n =  330000    t(n) =           133  T(n) =   4193259       q =   1.05381
n =  341600    t(n) =           131  T(n) =   4352460       q =          1
n =  353400    t(n) =           141  T(n) =   4514810       q =   1.03763
n =  365400    t(n) =           142  T(n) =   4680316       q =   1.00804
n =  377600    t(n) =           147  T(n) =   4848984       q =   1.00723
n =  390000    t(n) =           158  T(n) =   5020821       q =   1.04555
```

12. Wnioski:

Analizując tabelki ze współczynnikami zgodności oceny teoretycznej z pomiarem czasu, można zauważyć, że wyniki potwierdzają teoretyczną ocenę. Dla każdego z czterech algorytmów, wszystkie współczynniki q osiągają wartość bliską 1.

Analizując te tabelki można także zauważyć, że chociaż algorytmy Bellmana-Forda oraz Dijkstry z tablicą mają takie same złożoności asymptotyczne O, to nie oznacza to, że mają takie same czasy wykonania. Dużo zależy od współczynnika przy najwyższej potędze. W algorytmie Bellmana-Forda większość operacji ma złożoność n^2. Natomiast w algorytmie Dijkstry złożoność n^2 ma tylko operacja znajdowania minimum w tablicy (kolejce priorytetowej). To właśnie z tego powodu, pomimo tej samej złożoności asymptotycznej O, czasy wykonania tych algorytmów są tak różne (szybciej działa algorytm Dijkstry z tablicą).

Kolejnym wnioskiem z otrzymanych tabelek jest fakt, że dobór odpowiedniej struktury danych do algorytmu pełni bardzo ważną rolę. Najlepiej widać to w przypadku algorytmu Dijkstry. Zarówno algorytm Dijkstry z tablica jak i z algorytm Dijkstry z kopcem działają tak samo. Jedyną różnicą jest implementacja kolejki priorytetowej. Można zauważyć, że zamieniając tablicę na kopiec, jesteśmy w stanie znacznie zmniejszyć czas działania algorytmu.

Ostatnim wnioskiem (otrzymanym z analizy czasów wykonania algorytmów przy włączonej optymalizacji), jest wpływ heurystyki na działanie algorytmów. Okazuje się, że nawet jeśli algorytm heurystyczny ma taką samą złożoność asymptotyczną co algorytm nieheurystyczny, to w praktyce, dla większości przypadków, działa on znacznie szybciej niż algorytm nieheurystyczny.