Analiza Algorytmów – Dokumentacja wstępna

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.

Jeśli posiada Pan jakieś sugestie odnośnie tego, które z przedstawionych wyżej algorytmów mam zaimplementować, to oczywiście zostaną one przeze mnie uwzględnione. W przypadku wystarczającej ilości czasu planuje zaimplementować wszystkie trzy algorytmy (algorytm Dijkstry za pomocą kopca lub zwykłej tablicy).

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

d[poczatek] = 0

for(I od 1 do ilosc pol - 1)
{
    moznaKonczyc = true
    for(X od 0 do ilosc pol -1)
    {
        for(kazdy z sasiadow X w poziomie i w pionie)
        {
            if(d[sasiad] <= d[X] + 1)
                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 docelowym)
        znaleziono najkrotsza sciezke, wyjdz z funkcji
    for(kazdy z sasiadow P w poziomie i w pionie)
    {
        if(sasiad nie jest w Q)
            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. Ponadto planuję zrobić również generator danych pesymistycznych, czyli takich rastrów, w których dojście z punktu startowego A do punktu końcowego B jest zwykle bardziej skomplikowane, niż dla przeciętnego rastra wygenerowanego losowo.
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.