Najkrótsza droga w mieście

Wiktor Franus

22 stycznia 2017

Streszczenie

Aplikacja prezentująca działanie algorytmów znajdujących najkrótszą ścieżkę pomiędzy dwoma białymi punktami na czarno-białym rastrze.

Spis treści

1 Opis problemu

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 mozna się poruszać jedynie w pionie i w poziomie omjając 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.
Raster można traktować jako plan miasta lub labirynt, w którym białe pola reprezentują drogi, a czarne budynki lub ściany. Każde białe pole może być punktem początkowym, z którego chcemy znaleźć najkrótszą drogę do innego wybranego białego pola. Zgodnie z założeniami problemu znalezienie takiej drogi zawsze jest możliwe. Długość drogi z punktu A do punktu B liczona jest jako ilość pól białych, które należy odwiedzić, aby przejść z pola A do pola B. Nie można przy tym odwiedzać 2 razy tego samego pola. Możliwe jest istnienie więcej niż jednej drogi z punktu A do punktu B oraz istnienie kilku dróg o jednakowej długości. Szukana jest długość najktótszej drogi z A do B wraz z ideksami pól tworzących tę drogę. Zadany problem można inaczej przedstawić jako problem szukania najkrótszej ścieżki pomiędzy dwoma wierzchołkami w spójnym, nieskierowanym grafie. W tak sformułowanym zadaniu wszystkie białe pola rastra można interpretować jako wierzchołki grafu. Jeśli białe pola sąsiadują ze sobą na rastrze, tj. są położone obok siebie w jednej kolumnie lub w jednym wierszu, w grafie łączy je krawędź. Długość każdej takiej krawędzi jest równa 1, zatem powstały graf nie jest grafem ważonym. Spośród wszystkich wierzchołków wyróżnione są 2 wierzchołki: startowy i początkowy.

2 Metody rozwiązania

Istnieje wiele algorytmów grafowych, które odnajdują najkrótsze ścieżki pomiędzy wierzchołkami. W niniejszym projekcie wybrane i zaimplementowane zostały trzy algorytmy, wykazujące się zarówno prostotą działania, jak i niską złożonością czasową. Są to: algorytm Breadth-First Search, algorytm Dijkstry oraz heurystyczny algorytm A*.

2.1 Breadth-First Search

Algorytm rozpoczyna pracę w wierzchołku początkowym i odwiedza jego wszystkie sąsiednie wierzchołki, następnie odwiedza sąsiadów każdego sąsiada, itd. W każdym kroku zapamiętuje wskazanie na poprzednika. W ten sposób, po zakończeniu działania algorytmu możliwe jest odtworzenie najkrótszej ścieżki. Algorytm kończy się, gdy analizowany wierzchołek jest wierzchołkiem końcowym.

2.1.1 Pseudokod

G – graf
s – wierzchołek początkowy
e – wierzchołek końcowy
q – kolejka FIFO
p – tablica zawierająca wskazanie na poprzednika i-tego wierzchołka
d – tablica zawierająca odległość i-tego wierzchołka od s

```
function BFS(G, s, e)
    for wierzcholek u w G do
        p[u] ← nieokreslony
    p[s] ← s
    umiesc s w q
    while sa elementy w q do
        zdejmij wierzcholek u z q
        if u = e then return
        for wierzcholek v sasiadujacy z u do
            if p[v] = nieokreslony then
                p[v] ← u
                umiesc v w q

function ODTWORZ_SCIEZKE(p, aktualny)
    sciezka ← aktualny
    while aktualny rozny od s do
        aktualny ← p[aktualny]
        dodaj aktualny do sciezka
    return sciezka
```

Funkcja odtwórz_ścieżkę jest identyczna dla wszystkich algorytmów. Zwraca ona ścieżkę odwrotną, tj. od wierzchołka końcowego do wierzchołka początkowego.

2.1.2 Złożoność czasowa

Pesymistyczna – O(V+E)
gdzie V i E to liczności kolejno wierzchołków (pól białych) i krawędzi w grafie

2.2 Dijkstra

Algorytm wybiera następny wierzchołek do analizy posługując się kolejką priorytetową. Priorytetem kolejki jest aktualnie wyliczona odległość od wierzchołka źródłowego s.

2.2.1 Pseudokod

G – graf
s – wierzchołek początkowy
e – wierzchołek końcowy
q – kolejka priorytetowa
p – tablica zawierająca wskazanie na poprzednika i-tego wierzchołka
d – tablica zawierająca odległość i-tego wierzchołka od s

```
function DIJKSTRA(G, s, e)
    for wierzcholek u w G do
        d[u] ← nieskonczonosc
        p[u] ← nieokreslony
    d[s] ← 0
    p[s] ← s
    umieść s w q z priorytetem 0
    while sa elementy w q do
        zdejmij z q wierzchołek u o najmniejszej wartości d
        if u = e then return
        for wierzcholek v sasiadujacy z u do
            odleglosc ← d[u] + 1
            if odleglosc <d[v] then
                d[u] ← odleglosc
                p[v] ← u
                zmień priorytet v w q na odlegosc
```

2.2.2 Złożoność czasowa

Implementacja opiera się na kolejce priorytetowej opisanej w rozdziale 4.2. Pesymistyczna złożoność czasowa wynosi O(E * logV).

2.3 A*

Algorytm A* jest algorytmem heurystycznym. Działa podobnie jak alg. Dijkstry. Dla każdego analizowanego wierzchołka wyznacza jednak nie tylko odległość od wierzcholka startowego, ale również przewidywaną przez heurystykę odległość od wierzchołka końcowego. W ten sposób większy priorytet mają (są w pierwszej kolejności wybierane) wierzchołki leżące bliżej celu. Heurystyka dla analizowanego problemu liczy odległość pomiędzy zadanym wierzchołkiem, a wierzchołkiem początkowym. Z racji, że rozpatrywane są drogi na rastrze, to odległość ta liczona jest według metryki Manhattan.

2.3.1 Pseudokod

G – graf
s – wierzchołek początkowy
e – wierzchołek końcowy
q – kolejka priorytetowa
p – tablica zawierająca wskazanie na poprzednika i-tego wierzchołka
d – tablica zawierająca odległość i-tego wierzchołka od s
f[i] = d[i] + heurystyka(i,e) – wartość priorytetu dla i-tego wierzchołka

```
function HEURYSTYKA(a, b)
    return |a.x − b.x| − |a.y − b.y|

function A_STAR(G, s, e)
    for wierzcholek u w G do
        d[u] ← nieskonczonosc
        p[u] ← nieokreslony
    d[s] ← 0
    p[s] ← s
    umieść s w q z priorytetem 0
    while sa elementy w q do
        zdejmij z q wierzchołek u o najmniejszej wartości d
        if u = e then return
        for wierzcholek v sasiadujacy z u do
            odleglosc ← d[u] + 1
            if odleglosc <d[v] then
                d[u] ← odleglosc
                p[v] ← u
                prio ← odleglosc + HEURYSTYKA(v, e)
                zmień priorytet v w q na prio
```

2.3.2 Złożoność czasowa

A* posiada najlepszą spośród wymienionych algorytmów pesymistyczną złożoność czasową wynoszącą O(E).

3 Algorytm generujący dane

Zgodnie z założeniami aplikacja umożliwia uruchomienie algorytmów na losowo wygenerowanym rastrze. W tym celu zaimplementowano prosty algorytm opisany w wielu źródłach jako przerzukiwanie DFS (ang. Depth-First Search) z nawrotami (ang. backtracking). Rozszerzono go o możliwość rysowania rastra z zadaną ilością pól białych (wierzchołków grafu) i krawędzi łączących te pola (krawędzi grafu). W ten sposób użytkownik może określić nie tylko wielkość rastra, lecz również podać wymienione wyżej parametry opisujące graf, co umożliwia testowanie złożoności zaimplementowanych algorytmów. Początkowo raster wypełniony jest czarnymi polami. Tworzenie ścieżek rozpoczyna się od punktu startowego, którego współrzędne to 0,0 (lewy górny róg rastra). Punkt ten staje się pierwszym polem białym. Punkt końcowy jest zainicjowany współrzędnymi pola startowego. Następnie algorytm działa według poniższych kroków:

Dopóki raster zawiera pola czarne i nie osiągnięto zadanej liczby wierzchołków i krawędzi:

1. Wylosuj ilość pól k z przedziału 1-3.
2. Jeżeli z aktualnego pola można ruszyć się o k pól w dowolnym kierunku poruszając się wyłącznie po polach czarnych:
   - 2.1. Losowo wybierz kierunek, w którym można poruszyć się o k pól
   - 2.2. Jeśli krok o długości 1 w wylosowanym kierunku nie stworzy zbyt dużo nowych krawędzi:
     - 2.2.1. Zrób 1 krok w wylosowanym kierunku: zmień kolor pola na biały i dodaj pole do stosu
     - 2.2.2. Jeśli obie współrzędne aktualnego pola są większe od współrzędnych punktu końcowego:
       - 2.2.2.1. Ustaw aktualne pole jako pole końcowe
   - 2.3. Zwiększ licznik wierzchołków o 1 i licznik krawędzi o liczbę nowo utworzonych krawędzi
3. Zdejmij pole ze stosu
4. Ustaw je jako aktualne pole

Po zakończeniu działania algorytmu odległość pomiędzy polami startowym i końcowym jest, w zależności od przypadku, zmaksymalizowana lub temu bliska.

4 Opis wykorzystanych struktur danych

4.1 Raster

Raster reprezentowany jest w programie przez dwuwymiarową, dynamicznie alokowaną tablicę wypełnioną liczbami całkowitymi. Ściana (czarne pole) reprezentowana jest przez wartość 0, natomiast droga (białe pole) ma wartość 1.

4.2 Kolejka priorytetowa

Algorytm BFS korzysta z prostej kolejki FIFO. Użyta została w tym celu szybka kolejka deque z biblioteki standardowej. Z kolei algorytmy Dijksta i A* wymagają bardziej złożonego kontenera, w którym kolejność elementów określona jest za pomocą liczby - priorytetu. W projekcie użyta została kolejka priority_queue z biblioteki standardowej. Na potrzeby algorytmu obudowano ją strukturą z pomocnicznymi metodami. Kolejka ta oparta jest na kopcu binarnym przez co wstawianie do niej elementów jest niezwykle szybkie. Jej wadą jest natomiast brak metody zmieniającej priorytet elementu będącego już w kolejce. Brak takiej funkcjonalności jest prawdopodobnie spowodowany niemałym kosztem wspomnianej operacji. Można jednak, co zostało zrealizowane w aplikacji, zrezygnować ze zmiany priorytetu istniejącego elementu, a w zamian dodać do kolejki ten sam element z nową wartością priorytetu. Kolejka może zawierać duplikaty, co z pewnością ją spowalnia, jednak algorytmy na niej oparte działają poprawnie.

4.3 Macierze poprzedników i odległości

Każdy z algorytmów posługuje się macierzą poprzedników - dwuwymiarową, dynamicznie alokowaną tablicą zawierającą pary liczb całkowitych. Liczby te są indeksami w rastrze. Oznacza to, że pod i-tym wierszem i j-tą kolumną w mecierzy poprzedników znajdują się współrzędne poprzednika pola o współrzędnych i,j. Macierz ta jest aktualizowana podczas działania algorytmu szukającego najkrótszej ścieżki. Dzięki temu po zakończeniu działania algorytmu możliwe jest odtworzenie znalezionej ścieżki. Funkcja zwracająca znalezioną ścieżkę, korzystająca przy tym z macierzy poprzedników została przedstawiona w rozdziale 2.1.1.
Algorytmy Dijkstra i A* posługują się dodatkowo macierzą odległości (dwuwymiarową tablicą liczb całkowitych) zawierającą odległości punktu o współrzędnych i,j od punktu startowego. Początkowo wszystkie elementy tablicy zawierają liczbę INT_MAX.

5 Pomiary czasów wykonania

5.1 Breadth-First Search

V	E	t(n)[us]	q(n)
200	308	23.4	1.131
400	663	43.8	1.012
800	1338	90.2	1.036
1600	2757	190.2	1.072
3200	5725	368.2	1.013
6400	11555	735.6	1.006

12800	22744	1487.8	1.028
25600	46417	2933.6	1.000
51200	91238	5809.6	1.001
102400	189028	11626.2	0.979
204800	377151	24930.6	1.052
409600	756946	49056.4	1.032
819200	1520387	96786.8	1.016
1638400	3000025	202555.2	1.072
3276800	5988433	418533.6	1.109

Wartość współczynika q w powyższej tabeli dla różnych wielkości problemu wynosi z dużą dokładością 1, co świadczy o tym, że algorytm posiada złożoność zgodną z teoretyczną.

5.2 Dijkstra

V	E	t(n)[us]	q(n)
200	308	23.2	1.582
400	663	56.4	1.580
800	1338	113.8	1.416
1600	2757	237.8	1.301
3200	5725	508.6	1.225
6400	11555	1011.6	1.112
12800	22744	2046.0	1.059
25600	46417	4233.0	1.000
51200	91238	8856.4	0.996
102400	189028	17460.8	0.891
204800	377151	37164.0	0.897
409600	756946	76418.8	0.870
819200	1520387	158200.0	0.851
1638400	3000025	328326.4	0.851
3276800	5988433	685512.4	0.849

Zaprezentowane w tabeli rezultaty mogę wskazywać, że w ocenie złożoności teoretycznej algorytmu Dijkstry nastąpiło przeszacowanie (q malejące).

5.3 A*

V	E	t(n)[us]	q(n)
200	311	24.6	0.963
400	629	31.8	0.616
800	1173	45.2	0.469
1600	2775	184.6	0.810
3200	5734	455.0	0.966
6400	11439	822.0	0.875
12800	22958	1804.2	0.957
25600	46603	3827.6	1.000
51200	92750	4801.4	0.630
102400	188290	12680.8	0.820
204800	378465	28074.4	0.903
409600	753319	52120.0	0.842
819200	1517679	124889.4	1.002
1638400	3001474	218974.0	0.888
3276800	5989141	601806.8	1.223

Dla niektórych wielkości problemu (np. V=800, V=51200) współczynnik q jest znacznie mniejszy od 1. Zmierzony dla tych przypadków czas jest wyraźnie krótszy od analogicznych czasów dla pozostałych algorytmów. Jest to potwierdzeniem tego, że zastosowanie heurystyki daje w pewnych przypadkach wymierne korzyści.

5.4 Uwagi

Współczynniki q dla mniejszych rozmiarów problemów niż zaprezentowane w tabelach okazywały się kilkukrotnie wyższe od jedności. Przyczyną tego zjawiska może być bardzo szybki czas wykonania funkcji odnajdujących najkrótsze ścieżki i tym samym niemożność dokonania wiarygodnego pomiaru upłyniętego czasu (czasy rzędu kilku us).