Algorytmy Optymalizacji Dyskretnej
Eksperymentalna analiza algorytmów SSSP

Arkadiusz Lewandowski

May 2016

1 Wprowadzenie [2]

Niech G będzie grafem skierowanym ważonym, o krawędziach między wierzchołkami s i t oraz wadze v. Problemem tutaj omawianym jest szukanie zbioru najkrótszych ścieżek z danego wierzchołka do wszystkich innych w G lub też szukanie najkrótszej ścieżki z wierzchołka s do wierzchołka t. W przypadku omawianych tu algorytmów brane będą wyłącznie grafy o wagach ze zbioru liczb naturalnych. Single Source Shortest Path problem ma wiele algorytmów szukających tych ścieżek, ale przedstawiono tutaj 2 wraz z implementacjami oraz 1 jako teoretyczną podstawę do implementacji. [3]

1.1 Dijkstra

Algorytm jako należący do klasy SSSP próbuje znaleźć ścieżkę dla podanego wierzchołka *zrodlowego*, której jeden koniec należy do zbioru wierzchołków Q, a drugi do T oraz próbuje zminimalizować sumę dla dystansu ze *zrodla* do v oraz różnicy wag v i w. Przy czym $v \in Q$ i jest oznaczone jako minimalna odległość od źródła, a $w \in T$ i jest oznaczone jako potencjalny wierzchołek zastępujący v, jeśli będzie miał mniejszą wagę. Pierwszym podejściem byłoby porównywanie każdego wierzchołka z każdym, jednak dla rzadkich grafów wymagałoby to nadkładu pamięci i zupełnie sześciennej ilości operacji względem wierzchołków. Stąd Edsgar Dijkstra wywnioskował, że da się ten problem sprowadzić do podproblemu szukania najkrótszych ścieżek na trasie od *zrodla* do *celu*.

1.1.1 Wstęp do złożoności algorytmu

Najważniejsze własności wpływające na złożoność algorytmu:

Wybór wierzchołka - każde przejrzenie wszystkich tymczasowo zaetykietowanych wierzchołków skutkuje oznaczeniem jednego na stałe. Więc przegląda się n, $n-1$, $n-2$,..., 2,1 wierzchołków, stąd czas wyboru $\mathrm{O}(n^2)$.

Update - względem wierzchołka skutkuje w przejrzeniu wszystkich jego sąsiadów, co daje $\mathrm{O}(E)$.

Sumaryczny czas pracy $\mathrm{O}(V * E)$.

1.1.2 Algorytm

Algorithm 1 Algorytm Dijkstra

```
1:  function DIJKSTRA(G,source)
2:      for all v ∈ G do
3:          v.distance = ∞                          ▷ każdy wierzchołek jest za daleko
4:      end for
5:      [source].distance = 0                       ▷ źródłowy sam do siebie ma odległość 0
6:      Q ← G                                       ▷ Skopiuj wierzchołki Grafu do Kolejki
7:      while Q is not Empty do                     ▷ Sprawdź wszystkie wierzchołki
8:          u ← extractMin(Q)                       ▷ weź najtanszy i usun z Kolejki
9:          for all  v : adjacent(u,v)  do          ▷ Dla nastepnych sąsiadów u
10:             Relax(u, v)                         ▷ Wykonaj Update i zaetykietuj na stałe
11:         end for
12:     end while
13:     return distances
14: end function=0
```

Algorithm 2 Relax

```
1: function RELAX(u, v)
2:     for all v ∈ G do
3:         if distance(v) > distance(u) + cost(u,v) then
4:             distance(v) = distane(u) + dist(u, v)                    ▷ update
5:         end if
6:     end for
7:     return updatedVertices
8: end function
```

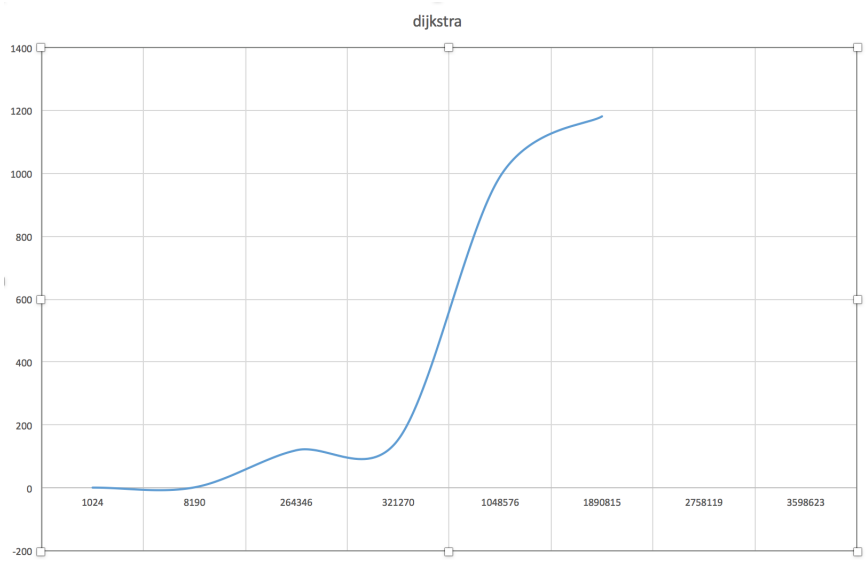
Algorithm 3 extractMin

```
1: function EXTRACTMIN(queue)    ▷ Koszt wyciągania kosztem wstawiania
2:     return queue.min− > vertex
3: end function
```

Algorithm 4 addQElement

```
1: function ADDQELEMENT(Queue, element)
2:     for all iterated-vertex ∈ Queue do
3:         if element.distance < iterated-vertex.distance then
4:             insert(iterated-vertex.prev,element,iteratex-vertex)
5:         end if                    ▷ wstawianie miedzy mniejszy, a wiekszy rowny
6:     end for
7:     return Queue
8: end function
```

1.1.3 Wydajność



1.1.4 Wnioski do algorytmu Dijkstry

Widać, że w miejscach gdzie funkcja opada do zera, algorytm całkowicie zawodził.

Dla liczby krawędzi mniejszej niż liczba wierzchołków można działać na samych krawędziach.

Prosta implementacja Dijkstry daje złożoność czasową $O(V^2)$.

Trzymając oznaczone odległości posortowane, można przyspieszyć algorytm. Wykorzystana idea kolejki priorytetowej.

1.2 Dial

Jest zoptymalizowanym algorytmem Dijkstry, zmienia jedynie podejscie do przechowywania odległych od szukanego wierzchołków. Każda jednostka odgległości daje kolejny kubełek. Każdy kubełek przechowuje wszystkie wierzchołki odległe o jednakową liczbę jednostek względem aktualnie badanego wierzchołka grafu. Podobnie jak w Dijkstrze, jednak struktura danych jest inaczej dobrana. Wynika to z własności, że oznaczone na stałe w algorytmie Dijkstry wierzchołki (dystanse od analizowanego) cechują się tym, że niemaleją. Każdy kubełek k w swoim założeniu ma być stworzony z dwukierunkowych list i posiadać wszystkie tymczasowo zaetykietowane odległości równe k. W pierwszym kubełku domyślnie jest wierzchołek *source*. Każdy następny niepusty kubełek ma wierzchołki z tymczasowymi dystansami. Kiedy wykonuje sie na wierzchołku procedure *update* to trzeba go przenieść do nowego odpowiadającego kubełka.

1.2.1 Wstęp do złożoności

Najważniejsze własności wpływające na złożoność algorytmu Dial:

Niemalejące odległości, z których wynika, że jeśli kubełek k ma wartości k, to następujący po nim niepusty musi być większy.

Całkowity czas przeszukiwania kubełków to $O(n * C + 1)$

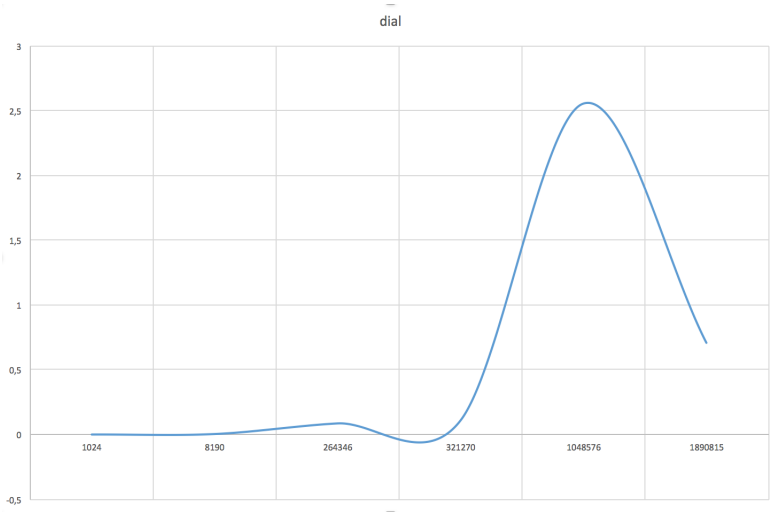
Dodawanie i wyjmowanie z kubełka $O(1)$

1.2.2 Algorytm

Algorithm 5 Algorytm Dial

```
1:  function DIAL(G,source)
2:      Przypisz kazdemu wierzchołkowi INF dystans
3:      Wierzchołkowi source przypisz 0
4:      Przygotuj V*C kubełków
5:      for all b ∈ Bucket do
6:          for all i ∈ Dist do
7:              b[dist[i]] ← dist[i]      ▷ Kubełek i −ty dostaje v z dist równym i
8:          end for
9:      end for                           ▷ Wierzchołki w każdym kubełku są jako listy
10:     while getMinElem(Bucket) ≠ NULL do      ▷ Każdy z tych kubełków
    (niepustych) jest oznaczony minimalnym dystansem
11:         getMinElem(Bucket) i usuń go z Kubełka
12:         while ( dov ∈ sąsiedzi minElem ≠ NULL)
13:             if v ma mniejszy koszt minElem + jego waga then
14:                 Add it to the Bucket with index proper to the distance
15:
16:
17:                 return d
18:
```

1.2.3 Wydajność



1.2.4 Wnioski do algorytmu Dial

Algorytm dla grafów o wielu wierzchołkach jest znacznie szybszy od Dijkstry.

Kiedy C jest małe albo stałe, to algorytm ma złożoność O(V+E).

Kiedy C jest bardzo duże, algorytm nieporównywalnie zwalnia, a jego asymptotyka upodabnia się do algorytmu Dijkstry.

Wymagania pamięciowe są bardzo duże dla grafów o krawędziach z dużymi wagami. $O(E + VC)$ złożoność pamięciowa zarazem.

Możliwe poprawienie algorytmu o zaczepienie maksymalnej różnicy między dwoma tymczasowymi oznaczeniami, które wynosi C. Z czego wynika, że algorytm ten działa na zasadzie koła. Złożoność czasowa pozostaje ta sama, ale pamięciowa maleje do $O(E + C)$.

1.3 Radix-Heap

Jest kolejnym zoptymalizowanym algorytmem Dijkstry, używającym kubełków jako kolejek priorytetowych do szukania minimalnych ścieżek między wierzchołkami. Tak samo jak algorytm Dial, korzysta on z własności monotoniczności niemalejących tymczasowych oznaczeń $d(_i)$. Także podobnie jak w algorytmie Dial, RadixHeap korzysta z kubełków, z tym, że jego sposób wyznaczania indeksu kubełka rośnie exponencjalnie, a zakresy kubełków w czasie działania algorytmu zmieniają się.

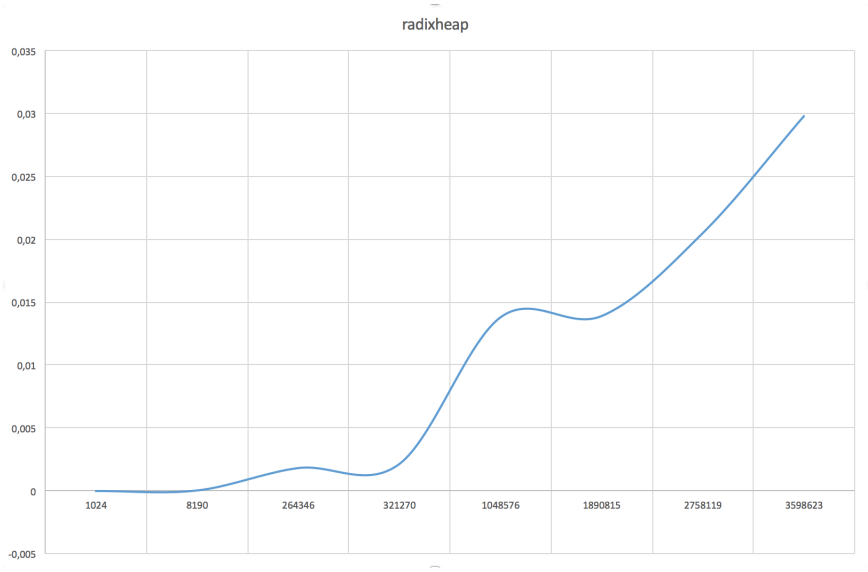
1.3.1 Wstęp do złożoności

Algorytm szukając kolejnych minimalnych krawędzi, przegląda jedynie K kubełków. Każdy kolejny kubełek, który jest pierwszym niepustym zostaje poddany redystrybucji jego elementów. Każdy jego element trafia do odpowiednio mniejszych kubełków, tak żeby tylko jeden był w kubełku z zakresem jednoelementowym. Kolejne kroki są identyczne jak w poprzednim algorytmie, tylko trzymając się zasady kubełków Radix.

1.3.2 Algorytm

Algorithm 6 Algorytm radix heap

```
1:  function RadixHeap(G, source)
2:      for all b ∈ B do
3:          b.size = [2^(i−1),2^i − 1]                    ▷ b - i-ty kubełek
4:      end for
5:      d(source) ← 0
6:      B(0) ← B(0) ∪ source        ▷ Wierzchołek source w pierwszym kubełku
7:      while Istnieje niepusty B do                        ▷ B − kubełek
8:          bucket ← Pierwszy niepusty B
9:          if bucket posiada tylko jeden wierzchołek then
10:             i ← bucket.first                ▷ Pobieramy pierwszy wierzchołek
11:             for all j ∈ G(i).next do
12:                 Relax(i,j,distances) z rozszerzeniem o:
13:                 Przerzucenie wierzchołka do mniejszego kubełka
14:             end for
15:             bucket.first ← bucket.first \ i
16:         else
17:              Modyfikacja kubełków i ich zakresów
18:             - minimalna wagowo krawędź w pierwszym kubełku
19:         end if
20:     end while
21:     return distances
22: end function=0
```



Rysunek 1: Wykres czasu w sekundach jaki potrzeba było na znalezienie wszystkich najkrótszych ścieżek przy podanej ilości wierzchołków.

1.3.3 Teoretyczna wydajność[1]

Algorytm powinien działać lepiej niż algorytm Dial, którego problemem była maksymalna waga krawędzi - determinująca ilość kubełków. Ten algorytm jest odporniejszy na tą zależność, gdyż zamienia ich ilość w logarytmiczną względem maksymalnej wagi krawędzi. Jego złożoność obliczeniowa jest podobna do algorytmu Dial, ale ze względu na łatwiejsze przeglądanie kolejnych tymczasowo oznaczonych krawędzi, jego złożoność zostaje dla tego kroku obniżona i zamiast $\mathrm{O}(E + VC)$ otrzymuje się $\mathrm{O}(E + VlogVC)$ lub nawet $\mathrm{O}(E + VlogC)$.

1.3.4 Dane testowe

Dane dobierane były dla każdego algorytmu, tak by w każdym z nich zademonstrować plusy i minusy jego implementacji. Dijkstra dla coraz większej ilości wierzchołków - niezależnie od kosztów krawędzi między wierzchołkami zwiększał swój czas wykonywania kwadratowo. Kolejny - algorytm Dial, poprawił wyniki Dijkstry jednakże jego implementacja - jest dobra tylko w przypadku, gdy znamy największy koszt krawędzi i jest on mniejszy niż ilość krawędzi w danym grafie. W przeciwnym wypadku otrzymuje się złożoność asymptotyczną taką jak w algorytmie Dijkstry. Trzecim algorytmem - RadixHeap, który jest udoskonaleniem powyższych dwóch, dało się sprowadzić problem ilości kubełków algorytmu Dial, do kubełków z przedziałami, z czego wynikło, że zarówno czas jak i pamięć dla przechowywania i wydobywania z nich wierzchołków drastycznie przyspiesza. Jego minusem jest to, że dla danych z życia wziętych oszczędza się stosunkowo niewiele czasu.

Dane dla powyższych implementacji w każdym wykresie to kolejno:

Nazwa	V	E	C
$Square - n.10$	1024	3968	1023
$Square - n.13$	8190	32398	8190
$USA - road - d.NY$	264346	733846	36946
$USA - road - d.BAY$	321270	800172	94305
$Square - n.20$	1048576	4190208	1048576
$USA - road - d.CAL$	1890815	4657742	215354
$USA - road - d.LKS$	2758119	6885658	138911
$USA - road - d.E$	3598623	8778114	200760

Literatura

[1] MIT. *15.082J / 6.855J / ESD.78J Network Optimization Fall 2010.* MITOpenCourseWare, 2010.

[2] James B. Orlin Ravindra K. Ahuja, Kurt Mehlhorn. *Faster Algorithms for the Shortest Path Problem.* Princeton University, Princeton, New Jersey and A TT Bell Laboratories,MurrayHill, New Jersey, 1990, 2 April.

[3] F. Benjamin Zhan. *Journal of Geographic Information and Decision Analysis, vol.1, no.1, pp. 69-82: Three Fastest Shortest Path Algorithms on Real Road Networks: Data Structures and Procedures.* GIDA.