Algorytmy Optymalizacji Dyskretnej
Eksperymentalna analiza algorytmów SSSP

Arkadiusz Lewandowski

May 2016

1 Wprowadzenie

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.

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

function DIJKSTRA(G,source)
    for all $v \in G$ **do**
        v.distance = ∞ ▷ każdy wierzchołek jest za daleko
    end for
    [$source$].distance = 0 ▷ źródłowy sam do siebie ma odległość 0
    $Q \leftarrow G$ ▷ Skopiuj wierzchołki Grafu do Kolejki
    while Q is not Empty **do** ▷ Sprawdź wszystkie wierzchołki
        $u \leftarrow extractMin(Q)$ ▷ weź najtanszy i usun z Kolejki
        for all v : adjacent(u,v) **do** ▷ Dla nastepnych sąsiadów u
            $Relax(u, v)$ ▷ Wykonaj Update i zaetykietuj na stałe
        end for
    end while
    return $distances$
end function=0

Algorithm 2 Relax

function RELAX(u, v)
    for all $v \in G$ **do**
        if distance(v) > distance(u) + cost(u,v) **then**
            distance(v) = distane(u) + dist(u, v) ▷ update
        end if
    end for
    return $updatedVertices$
end function

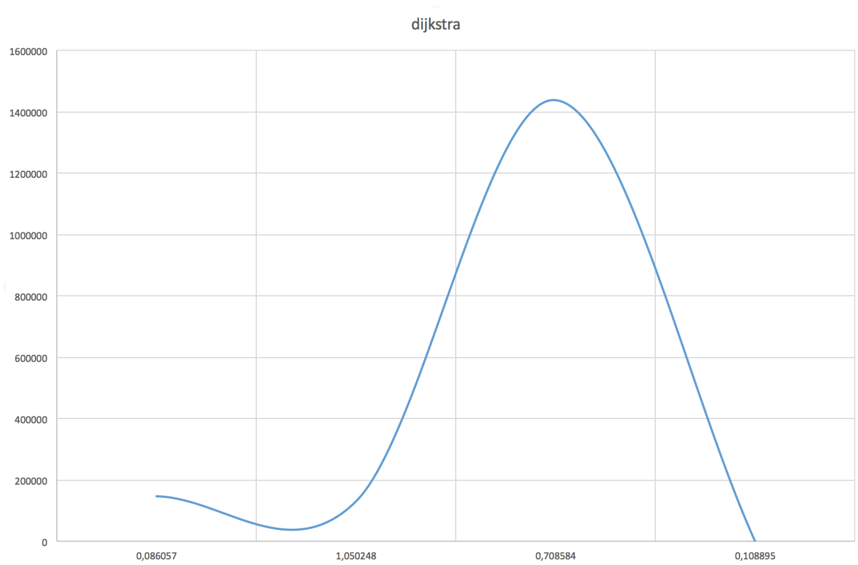
Algorithm 3 extractMin

function EXTRACTMIN(queue) ▷ Koszt wyciągania kosztem wstawiania
    return $queue.min- > vertex$
end function

Algorithm 4 addQElement

```
function addQElement(Queue, element)
    for all iterated-vertex ∈ Queue do
        if element.distance < iterated-vertex.distance then
            insert(iterated-vertex.prev,element,iteratex-vertex)
        end if                  ▷ wstawianie miedzy mniejszy, a wiekszy rowny
    end for
    return Queue
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$(n * C + 1)$

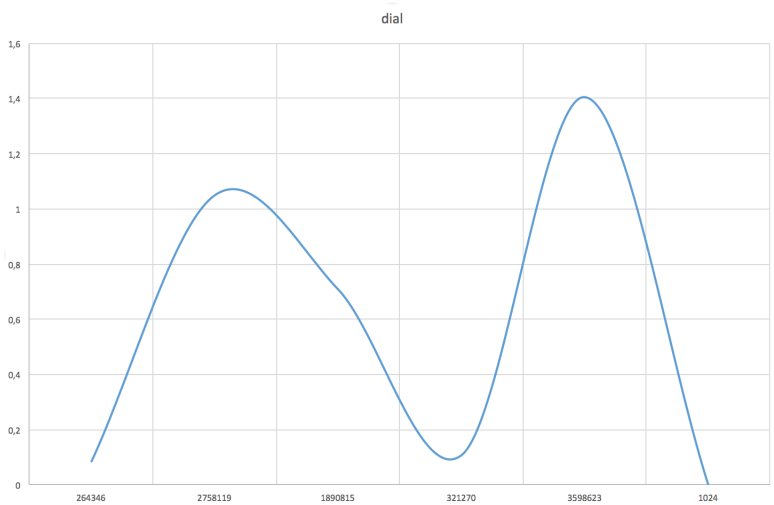
Dodawanie i wyjmowanie z kubełka O(1)

1.2.2 Algorytm

Algorithm 5 Algorytm Dial

function DIAL(G,source)
  Przypisz kazdemu wierzchołkowi INF dystans
  Wierzchołkowi source przypisz 0
  Przygotuj V*C kubełków
  for all $b \in Bucket$ **do**
    for all $i \in Dist$ **do**
      b[dist[i]] ← dist[i] ▷ Kubełek i −ty dostaje v z dist równym i
    end for
  end for ▷ Wierzchołki w każdym kubełku są jako listy
  while getMinElem(Bucket) $\neq NULL$ **do** ▷ Każdy z tych kubełków (niepustych) jest oznaczony minimalnym dystansem
    getMinElem(Bucket) i usuń go z Kubełka
    while (**do**v ∈ sąsiedzi minElem ≠ NULL)
      if v ma mniejszy koszt minElem + jego waga **then**
        Add it to the Bucket with index proper to the distance
        return d

1.2.3 Wydajność



1.2.4 Wnioski do algorytmu Dial

Algorytm szybki dla dużych danych, prównywalny dla małych danych.

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

Kiedy C jest bardzo duże, algorytm nieporównywalnie zwalnia.

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 $O(E+C)$.