AOD - Laboratorium 3

Michał Waluś 279695

Grudzień 2025

1 Problem

Problem SSSP (*Single Source Shortest Path*), czyli problem najkrótszej ścieżki z jednego źródła w grafie skierowanym $G = (V, E)$, z funkcją wag krawędzi $w : E \to R_{\geqslant 0}$, polega jest znalezienie długości najkrótszej ścieżki $d(s, v)$ pomiędzy jednym ustalonym źródłem s oraz każdym pozostałym wierzchołkiem $v \in V \backslash \{s\}$.
Mając dane wierzchołki u i v, ścieżkę $p = (v_1, v_1, v_2, \ldots, v_n)$, $v_1 = u$, $v_n = v$ nazywamy najkrótszą, gdy jej sumaryczna waga

$$W(p) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}) \tag{1}$$

jest najmniejsza.
Odległość między wierzchołkami u i v definiujemy następująco:

$$d(u, v) = \begin{cases} \min\{W(p) : p = (u = v_1, v_2, \ldots, v_n = v)\}, & \text{jeżeli istnieje droga z } u \text{ do } v \\ \infty, & \text{w przeciwnym przypadku} \end{cases} \tag{2}$$

2 Algorytm Dijkstry

Algorytm Dijkstry znajduje najkrótszą ścieżkę poprzez wykorzystanie kolejki priorytetowej do której dodajemy wierzchołki wraz z odległością. Na początku wewnątrz kolejki znajduje się tylko źródło s oraz odległości wszystkich wierzchołków są ustawione na ∞. Algorytm wykonuje w pętli następujące operacje:

1. Wyciągnięcie z kolejki wierzchołka u o najmniejszej odległości (priorytecie).
2. Dla każdego sąsiada v wierzchołka u:
   - Jeśli nowa odległość do v poprzez u jest krótsza niż aktualna minimalna odległość do v, to zastępujemy odległość nową odległością i dodajemy v do kolejki (jeśli go w niej nie ma, jeśli jest to tylko zmieniamy priorytet).
   - Jeśli nowa odległość jest większa to idziemy dalej.

Algorytm kończy działanie kiedy kolejka priorytetowa się opróżni.

Pseudo-kod algorytmu został przedstawiony jako Algorithm 1.

```
Function Dijkstra (G, s):
    Define n ← G.n ;                                   // liczba wierzchołków
    Define PQ ← priority queue;
    Define d[n] ;                                      // odległość od źródła
    for v ← 1 to n do
        d[v] ← ∞;
    end
    d[s] ← 0;
    PQ.insert((d[s], s));
    while PQ is not empty do
        (du, u) ← PQ.extract_min();
        for (v, w) ∈ G.neighbors(u) do
            // w to waga/koszt krawędzi
            new_d ← du + w;
            if new_d < d[v] then
                d[v] ← new_d;
                PQ.emplace((d[v], v)) ;                // Dodajemy lub aktualizujemy
            end
        end
    end
    Return d;
end
```

Algorithm 1: Algorytm Dijkstry

W implementacji użyta została kolejka priorytetowa `std::priority_queue`[1] ze biblioteki standardowej w C++, dla której operacje insert(), emplace() (zastąpienie elementu) oraz extract_min() mają złożoność czasową $O(\log n)$, a inicjacja kolejki $O(1)$.

Inicjacja algorytmu (przed pętlą while) ma złożoność czasową $O(|V|)$. Złożność obliczeniowa kodu procedur *extract_min*() oraz *emplace*() jest logarytmiczna i zauważmy, że *extract_min*() może się wykonać maksymalnie tyle razy ile jest wierzchołków, a także nie rozważamy żadnej krawędzi dwukrotnie, czyli kod w wewnętrznej pętli wykonamy maksymalnie $O(|E|)$ razy. Daje nam to złożoność obliczeniową algorytmu:

$$O\left((|V| + |E|) \log |V|\right)$$

3 Algorytm Diala

Algorytm Diala to modyfikacja algorytmu Dijkstry, która korzysta z faktu, że wagi krawędzi należą do przedziału $[0, C]$. Polega on na zastąpieniu kolejki priorytetowej, listą kubełków, odpowiadającym odległościo. Ponieważ zawsze rozpatrujemy wierzchołek o najmniejszej odległości znajdujący się na liście, oznaczmy tą odległość przez d^*, a wagi krawędzi są nieujemne, to na naszej liście znajdują się tylko wierzchołki w odległościach należących do przedziału $[d^*, d^* + C]$, zatem możemy wykorzystać tylko $C + 1$ kubełków, z których korzystamy cyklicznie.

Przy implementowaniu wybieramy taką implementacje zbioru (tutaj użyty jako kubełek), żeby możliwe było wybranie i usunięcie jakiegoś elementu w stałym czasie (ponieważ potrzebujemy dowolny element z kubełka, to wystarczy nam by stały czas był tylko dla pierwszego elementu).
W algorytmie Diala, podobnie jak w algorytmie Dijkstry każdą krawędź rozpatrzamy tylko raz,

[1] `https://en.cppreference.com/w/cpp/container/priority_queue.html`

```
Function Dial (G, s, C):
    Define n ← G.n ;                                  // liczba wierzchołków
    Define B[C + 1] ;                                          // kubełki
    Define d[n] ;                                     // odległość od źródła
    Define i ← 0 ;                          // indeks rozpatrywanego kubełka
    for v ← 1 to n do
        d[v] ← ∞;
    end
    d[s] ← 0;
    B[0] ← {s};
    while B is not empty do
        if |B[i]| = 0 then
            i ← (i + 1) mod (C + 1);
            continue;
        end
        u ← element z B[i];
        usuń u z B[i];
        for (v, w) ∈ G.neighbors(u) do
            // w to waga/koszt krawędzi
            new_d ← du + w;
            if new_d < d[v] then
                if d[v] ≠ ∞ then
                    usuń v z B;
                end
                d[v] ← new_d;
                dodaj v do B[(i + w) mod (C + 1)];
            end
        end
    end
    Return d;
end
```

Algorithm 2: Algorytm Diala

a dla każdego wierzchołka musimy przeszukać maksymalnie $C + 1$ kubełków by go znaleźć, zatem złożność czasowa algorytmu Diala to:

$$O(|E| + |V| \cdot C)$$

Widzimy, że dla dużych wartości parametru C, jest on znacznie wolniejszy od algorytmu Dijkstry.

4 Algorytm Radix Heap

Algorytm Radix Heap to usprawniona wersja algorytmu Diala, która zamiast wykorzystywać kubełki wielkości 1, wykorzystuje strukturę Rafix Heap i używa kubełków o wielkościach będących potęgami liczby 2: $\{1, 1, 2, 4, 8, 16, \dots\}$. Pozwala to zmniejszyć liczbę kubełków do $O(\log(|V|C)$, a stosując podobne uproszczenie jak w algorytmie Diala, do $O(\log C)$. Gdy podczas działania algorytmu pierwszy kubełek nie jest pusty, to opróżniamy pierwszy niepusty i rodzielamy jego zawartość między mniejsze.
Procedura `extract_min()` ma złożoność czasową $O(\log(|V|C)$, a `insert()` $O(1)$, zatem złoż-

ność całego algorytmu to:

$$O(|E| + |V| \log(|V|C)$$

co stosując usprawnienie możemy redukować do:

$$O(|E| + |V| \log C$$

5 Wyniki - SSSP

Wykresy przedstawiają otrzymane wyniki dla rodzin grafów:

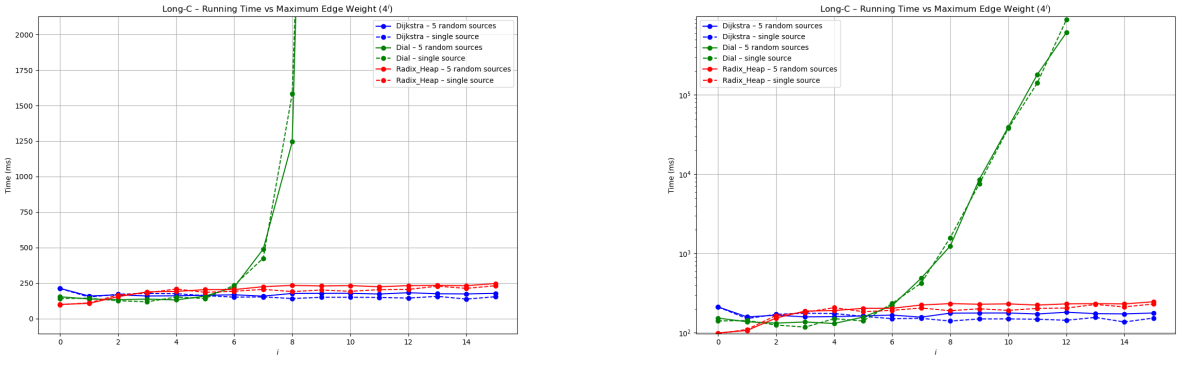
- Long-C, Long-N
- Random4-C, Random4-N
- Square-C, Square-N
- USA-Road-d

Dla każdej rodziny liczymy czasy wykoniania dla:

- SSSP od wierzchołka z indeksem 1
- SSSP od 5 losowo wybranych wierzchołków (te same dla wszystkich 3 algorytmów)

Ponieważ w rodzinach typu -C, wagi krawędzi potrafią osiągać nawet $4^{15} \approx 10^9$, skutkujące bardzo dużym czasem wykonania algorytmu Diala oraz zaporzebowaniem na pamięć operacyjną, rozwiązanie problemu z użyciem tego algorytmu nie było możliwe dla niektórych grafów. Na wykresach ciągłą linią przedstawione są średnie czasy, a przerywaną te wyznaczone od pierwszego wierzchołka. Kolorem niebieskim przedstawione są czasy algorytmu Dijkstry, zielonym Diala, a czerwonym RadixHeap.

5.1 Long-C

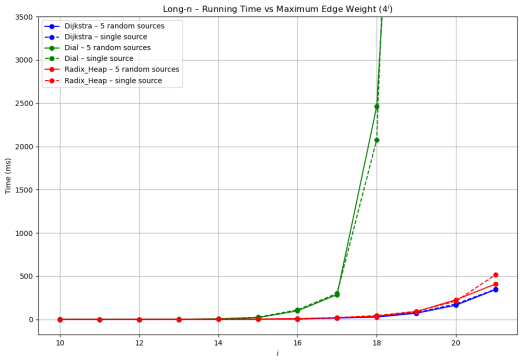


(a) skala liniowa

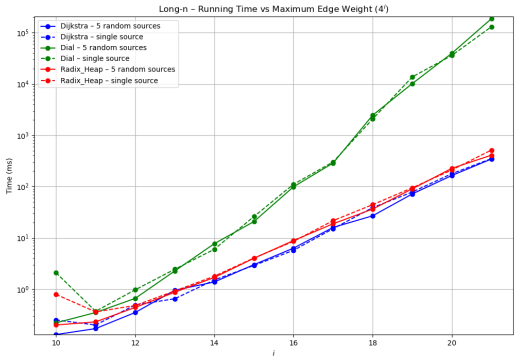
(b) skala logarytmiczna

Rysunek 1: Wykresy dla rodziny Long-C

5.2 Long-n



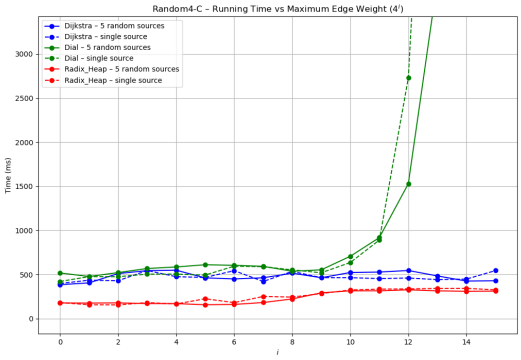
(a) skala liniowa



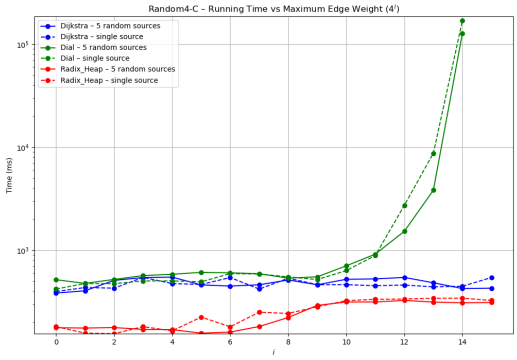
(b) skala logarytmiczna

Rysunek 2: Wykresy dla rodziny Long-n

5.3 Random4-C



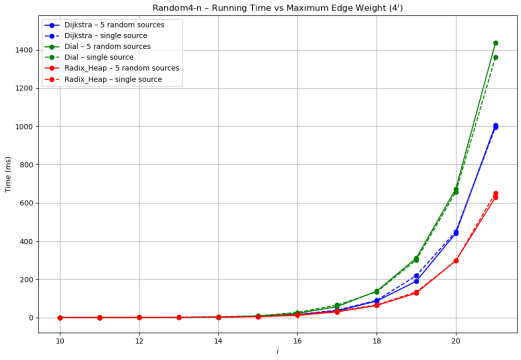
(a) skala liniowa



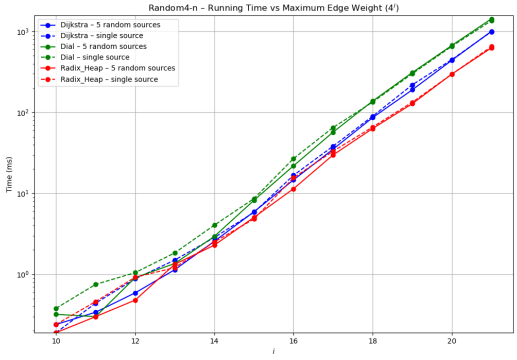
(b) skala logarytmiczna

Rysunek 3: Wykresy dla rodziny Random4-C

5.4 Random4-n



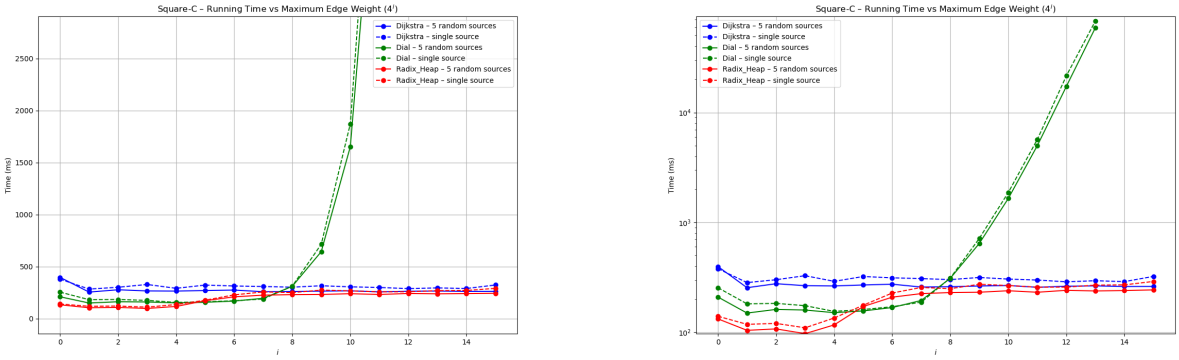
(a) skala liniowa



(b) skala logarytmiczna

Rysunek 4: Wykresy dla rodziny Random4-n

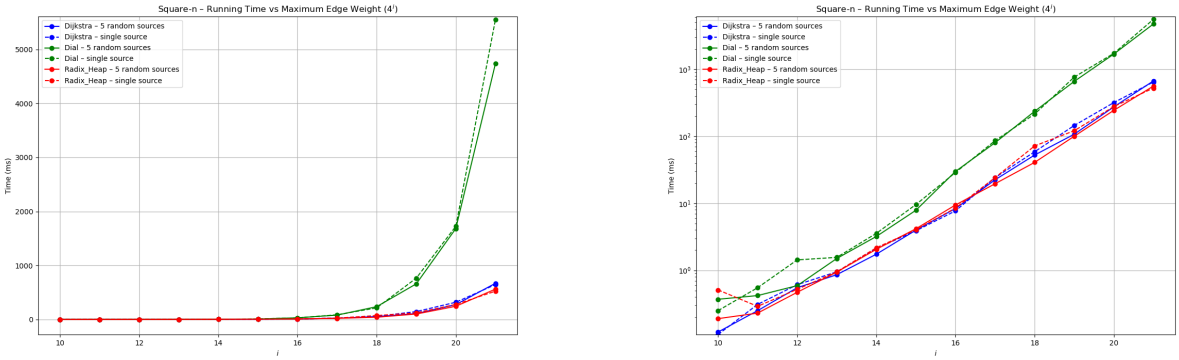
5.5 Square-C



(a) skala liniowa (b) skala logarytmiczna

Rysunek 5: Wykresy dla rodziny Square-C

5.6 Square-n



(a) skala liniowa (b) skala logarytmiczna

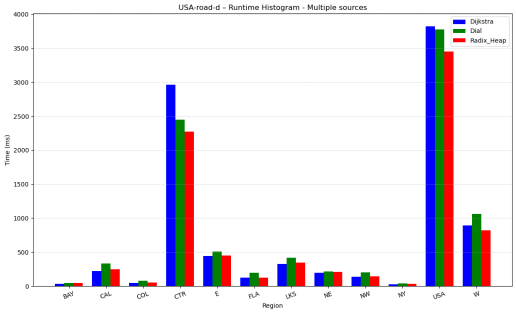
Rysunek 6: Wykresy dla rodziny Square-n

5.7 USA-road-d

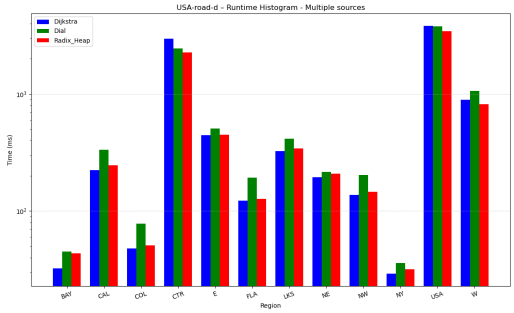
Ponieważ różnice w porównaniu z wykresami dla jednego źródła są niezauważalne, przedstawione zostały tylko wykresy dla uśrednionego czasu. (na Rysunku 7 na kolejnej stronie)

5.8 Obserwacje

Algorytmy Dijkstry oraz RadixHeap we wszystkich przypadkach dają podobne rezultaty, chociaż dla wszystkich rodzin poza Long-C i Long-n, RadixHeap był szybszy. Dla rodzin, które różnią się wagami krawędzi (-C), a w szczególności Square-C, widzimy, że dla małych wartości C algorytm Diala działa szybko, czasem nawet szybciej niż dwa pozostałe, jednak gdy wagi krawędzi stają się duże, jego efektywność znacznie spada, a czasy wykonania są o kilka rzędów wielkości większe niż dla pozostałych algorytmów.
Można zaobserować, że czas działania algorytmów Dijkstry i RadixHeap nie wykazuje znaczących zmian wraz ze wzrostem C, co sprawia że te algorytmy są zdecydowanie lepsze dla grafów,



(a) skala liniowa



(b) skala logarytmiczna

Rysunek 7: Wykresy dla rodziny USA-road-d

których parametrów nie znamy albo do należących do bardzo zróżnicowanych rodzin. Algorytm Diala powinien być stosowany tylko i wyłącznie dla grafów o małych kosztach, bo do tego był stworzony, a dla grafów takich jak te z rodziny Long-C (dla $i \geqslant 10$), otrzymujemy czasy wykonania mierzone w minutach lub nawet godzinach, podczas gdy alternatywny obliczają to poniżej sekundy.

6 Wyniki - Najkrótsza ścieżka

Celem było znalezienie najkrótszych ścieżek dla największego grafu z każdej rodziny wynionionej w części SSSP. Dla każdego grafu szukamy ścieżki od wierzchołka z najmniejszym indeksem do wierzchołka z największym indeksem oraz średniego czasu znalezienia 4 ścieżek pomiędzy wierzchołkami z losowo wybranymi indeksami.

Rodzina - Algorytm	Start	Koniec	Odległość	Czas (ms)
Long-C - Dijkstra	1	1048576	1308259008765	28.04
Long-C - Dial				Timeout
Long-C - RadixHeap	1	1048576	1308259008765	31.6
Long-n - Dijkstra	1	2097152	31336751771	291.2
Long-n - Dial	1	2097152	31336751771	95545.73
Long-n - RadixHeap	1	2097152	31336751771	308.11
Random4-C - Dijkstra	1	1048576	3471241820	139.37
Random4-C - Dial				Timeout
Random4-C - RadixHeap	1	1048576	3471241820	109.35
Random4-n - Dijkstra	1	2097152	9051281	547.91
Random4-n - Dial	1	2097152	9051281	860.5
Random4-n - RadixHeap	1	2097152	9051281	371.79
Square-C - Dijkstra	1	1048576	122219500320	105.87
Square-C - Dial				Timeout
Square-C - RadixHeap	1	1048576	122219500320	112.87
Square-n - Dijkstra	1	2096704	714640488	257.69
Square-n - Dial	1	2096704	714640488	2750.22
Square-n - RadixHeap	1	2096704	714640488	255.78
USA-road-d - Dijkstra	1	23947347	23228284	3253.2
USA-road-d - Dial	1	23947347	23228284	2860.24
USA-road-d - RadixHeap	1	23947347	23228284	2845.07

Tabela 1: Wyniki dla ścieżek od pierwszego do ostatniego wierzchołka

Wykonywanie algorytmu Diala dla tak dużych kosztów jak 4^{15} jest niepraktyczne i na zwykłym komputerze często brakuje na to zasobów.

Rodzina - Algorytm	Czas (ms)
Long-C - Dijkstra	114.98
Long-C - Dial	Timeout
Long-C - RadixHeap	160.58
Long-n - Dijkstra	216.61
Long-n - Dial	117392.49
Long-n - RadixHeap	265.49
Random4-C - Dijkstra	210.35
Random4-C - Dial	Timeout
Random4-C - RadixHeap	145.75
Random4-n - Dijkstra	487.54
Random4-n - Dial	992.31
Random4-n - RadixHeap	322.88
Square-C - Dijkstra	129.67
Square-C - Dial	Timeout
Square-C - RadixHeap	145.99
Square-n - Dijkstra	239.23
Square-n - Dial	2300.17
Square-n - RadixHeap	219.63
USA-road-d - Dijkstra	2272.93
USA-road-d - Dial	2134.75
USA-road-d - RadixHeap	2036.93

Tabela 2: Wyniki dla losowych ścieżek

6.1 Obserwacje

Za wyjątkiem rodziny USA-road-d, to we wszystkich przypadkach, gdzie algorytm Diala był w stanie zakończyć wykonywanie w sensownym czasie, to i tak jest najwolniejszym algorytmem z tych trzech. Szczególnie jest to widoczne dla rodziny Long-n, gdzie wartości C rosną proporcjonalnie do ilości wierzchołków.
Po wynikach nie widać dużych różnic w czasie wykonania pomiędzy dwoma przypadkami, poza rodziną Long-C, gdzie dla ścieżki z pierwszego wierzchołka, algorytmy Dijkstry i RadixHeap zakończyły działanie czterokrotnie szybciej.
Warto również zaobserwować, że dla grafu z rodziny USA-road-d, która jest jedyną tutaj obecną rodziną z relatywnie małymi współczynnikami C, algorytm Diala radzi sobie lepiej od algorytmu Dijkstry, jednak algorytm RadixHeap dalej jest szybszy.

7 Wnioski

Wyniki uzyskane eksperymentalnie potwierdzają wnioski uzyskane z teoretycznej analizy. Ponieważ dane testowe miały zazwyczaj ogromne wagi krawędzi algorytm Diala potrzebował bardzo dużo czasu na wykonanie i był zdecydowanie najwolniejszym.
Możemy z tego wnioskować, że dla rodzin o nieznanych lub zróżnicowanych własnościach algorytm Dijkstry jest znacznie lepszym wyborem, a jeżeli mamy nawet słabe ograniczenie na koszty krawędzi, takie jak możliwość zapisania ich w 64 bitach, to algorytm RadixHeap będzie często jeszcze lepszym wyborem i nawet jeśli dla małych wag krawędzi, algorytmy te czasami są wolniejsze od algorytmu Diala, jest to niewielka różnica.