Sprawozdanie lista3
AOD

Zofia Tarchalska, indeks: 279699

1 Wstęp

Na laboratorium numer 3 należało zaimplementować 3 algorytmy:

- algorytm (`dijkstry`)
- algorytm (`diala`)
- algorytm (`radix heap`)

Służą one do znajdowania najkrótszych ścieżek w grafie. Problem ten polega na tym, że dla grafu skierowanego $G = (V, E)$, w którym każdy łuk ma przypisaną jakąś wagę, chcemy znaleźć najkrótszą ścieżkę pomiędzy źródłem (ustalonym), a pozostałymi wierzchołkami $V \setminus \{s\}$.
Mając dwa wierzchołki, powiedzmy u i v, ścieżkę

$$p = (v_1, v_2, \ldots, v_n), \quad v_1 = u,\ v_n = v$$

nazywamy najkrótszą, gdy jej sumaryczna waga

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

jest możliwie najmniejsza.

2 Algorytm Dijkstry

Algorytm Dijkstry służy do wyznaczania najkrótszych ścieżek ze źródła s do wszystkich pozostałych wierzchołków w grafie skierowanym o nieujemnych wagach krawędzi. Podstawową ideą jest iteracyjne wybieranie wierzchołka o najmniejszej odległości od źródła (utrzymywanej w strukturze priorytetowej), a następnie relaksacja wszystkich krawędzi wychodzących z tego wierzchołka. Proces powtarza się aż do opróżnienia kolejki priorytetowej.

2.1 Opis działania

1. Inicjalizujemy odległości: dla źródła s ustawiamy $d[s] = 0$, a dla pozostałych $d[v] = \infty$.
2. Umieszczamy wszystkie wierzchołki w kolejce priorytetowej Q, kluczem jest bieżąca wartość $d[v]$.
3. Dopóki Q nie jest pusta:
   (a) Wyjmujemy wierzchołek u o najmniejszej wartości $d[u]$.
   (b) Dla każdej krawędzi (u, v) o wadze w wykonujemy relaksację:

$$\text{jeśli } d[v] > d[u] + w \text{ to ustaw } d[v] \leftarrow d[u] + w.$$

   (c) Jeśli $d[v]$ zostało zmienione, aktualizujemy klucz w kolejce priorytetowej.

2.2 Pseudokod

Algorithm 1 Dijkstra(G, s)

```
1:  for each v ∈ V[G] do
2:      d[v] ← ∞
3:      p[v] ← NIL
4:  d[s] ← 0
5:  Q ← V[G]                          ▷ kolejka priorytetowa według d[v]
6:  while Q ≠ ∅ do
7:      u ← Extract-Min(Q)
8:      for each (u, v) ∈ E[G] do
9:          if d[v] > d[u] + w(u, v) then
10:             d[v] ← d[u] + w(u, v)
11:             p[v] ← u
12:             Decrease-Key(Q, v, d[v])
13:         end if
14:     end for
15: end while
```

2.3 Złożoność czasowa

Złożoność algorytmu zależy od użytej struktury danych:
Z kolejką priorytetową opartą na kopcu binarnym (właśnie tak jest zaimplementowana ta z pakietu `DataStructures` w Julia):

$$O((V + E) \log V),$$

gdzie V to liczba wierzchołków, a E liczba krawędzi. Wyjmowanie najmniejszego elementu ma złożoność $O(1)$.

3 Algorytm Diala

Algorytm Diala jest modyfikacją algorytmu Dijkstry, która wykorzystuje fakt, że wagi krawędzi są ograniczone do przedziału $[0, W]$. Dzięki temu możemy usprawnić wybór następnego wierzchołka poprzez zastosowanie cyklicznej listy kubełków o długości $W + 1$. Każdy kubełek j przechowuje wierzchołki o etykietach tymczasowych równych $j, j + (W + 1), j + 2(W + 1), \ldots$. Ponieważ kubełki są opróżniane w kolejności cyklicznej, etykiety nie nakładają się na siebie.

3.1 Opis działania

1. Inicjalizujemy odległości: $d[s] = 0$, dla pozostałych $d[v] = \infty$.
2. Tworzymy cykliczną listę kubełków o długości $W + 1$.
3. Umieszczamy źródło s w kubełku 0.
4. Utrzymujemy wskaźnik `current`, wskazujący bieżący kubełek.
5. Dopóki istnieją nieprzetworzone wierzchołki:
   (a) Przesuwamy `current` cyklicznie, aż trafimy na niepusty kubełek.
   (b) Opróżniamy bieżący kubełek:
      i. Pobieramy wierzchołek u.
      ii. Dla każdej krawędzi (u, v) o wadze w obliczamy $newDist = d[u] + w$.
      iii. Jeśli $newDist < d[v]$, aktualizujemy $d[v] \leftarrow newDist$ i przenosimy v do kubełka $newDist \bmod (W + 1)$.

3.2 Pseudokod

Algorithm 2 Dial(G, s, W)

```
1:  for each v ∈ V[G] do
2:      d[v] ← ∞
3:      p[v] ← NIL
4:  end for
5:  d[s] ← 0
6:  buckets ← cyclic list of W + 1 empty lists
7:  insert s into buckets[0]
8:  current ← 0
9:  while true do
10:     while buckets[current] is empty do
11:         current ← (current + 1) mod (W + 1)
12:     end while
13:     while buckets[current] not empty do
14:         u ← remove element from buckets[current]
15:         for each (u, v) ∈ E[G] with weight w do
16:             newDist ← d[u] + w
17:             if newDist < d[v] then
18:                 remove v from buckets[d[v] mod (W + 1)]
19:                 d[v] ← newDist
20:                 p[v] ← u
21:                 add v to buckets[newDist mod (W + 1)]
22:             end if
23:         end for
24:     end while
25: end while
```

3.3 Złożoność czasowa

Każda krawędź jest rozpatrywana dokładnie raz, a każdy wierzchołek może być przenoszony maksymalnie W razy. Zatem złożoność czasowa wynosi:

$$O(E + VW).$$

Dla dużych wartości W niestety jego wydajność spada w porównaniu z klasycznym algorytmem Dijkstry.

4 Algorytm Radix Heap

Algorytm Dijkstry można przyspieszyć, gdy wagi krawędzi są nieujemnymi liczbami całkowitymi, poprzez zastosowanie specjalnej struktury danych zwanej Radix Heap.

Radix Heap jest wariantem kolejki priorytetowej, który grupuje elementy w kubełkach w zależności od wartości ich klucza (odległości). Dzięki temu operacje `Extract-Min` oraz `Decrease-Key` mogą być wykonywane szybciej niż w klasycznym kopcu binarnym. Jest to lepsze podejście do sposobu działania Diala.

4.1 Opis działania

1. Inicjalizujemy odległości: $d[s] = 0$, dla pozostałych $d[v] = \infty$.
2. Tworzymy tablicę kubełków (np. 64 kubełki dla liczb całkowitych 64-bitowych).
3. Umieszczamy źródło s w kubełku odpowiadającym wartości 0.
4. Dopóki kolejka nie jest pusta:
   (a) Znajdujemy pierwszy niepusty kubełek.
   (b) Jeśli jest to kubełek 0, wyjmujemy elementy i wykonujemy relaksację ich krawędzi.
   (c) Jeśli jest to kubełek $i > 0$, znajdujemy minimalny klucz w tym kubełku, ustawiamy go jako nową wartość `last_dist`, a następnie przenosimy elementy do odpowiednich kubełków zgodnie z funkcją indeksującą.

4.2 Złożoność czasowa

Dzięki zastosowaniu Radix Heap, złożoność czasowa algorytmu wynosi:

$$O(E + V),$$

W praktyce oznacza to, że algorytm działa w czasie bliskim liniowemu względem wielkości grafu.

4.3 Pseudokod

Algorithm 3 Dijkstra z Radix Heap(G, s)

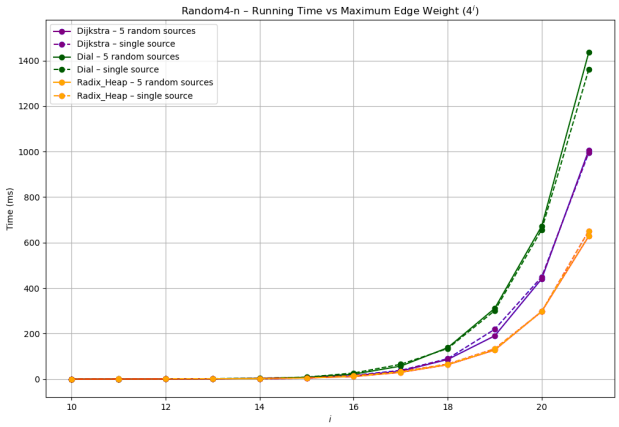
```
1:  for each v ∈ V[G] do
2:      d[v] ← ∞
3:      p[v] ← NIL
4:  end for
5:  d[s] ← 0
6:  utwórz tablicę kubełków B[0..64]
7:  wstaw (0, s) do B[0]
8:  last_dist ← 0
9:  while kolejka niepusta do
10:     znajdź najmniejszy indeks i taki, że B[i] niepusty
11:     if i > 0 then
12:         min_key ← min{ klucze w B[i] }
13:         last_dist ← min_key
14:         przenieś elementy z B[i] do odpowiednich kubełków
15:     else
16:         while B[0] niepusty do
17:             (k, u) ← usuń element z B[0]
18:             if k = d[u] then
19:                 for każde (u, v) ∈ E[G] o wadze w do
20:                     newDist ← d[u] + w
21:                     if newDist < d[v] then
22:                         d[v] ← newDist
23:                         p[v] ← u
24:                         j ← indeks kubełka dla newDist względem last_dist
25:                         wstaw (newDist, v) do B[j]
26:                     end if
27:                 end for
28:             end if
29:         end while
30:     end if
31: end while
```

5 Wyniki doświadczenia dla określonych źródeł

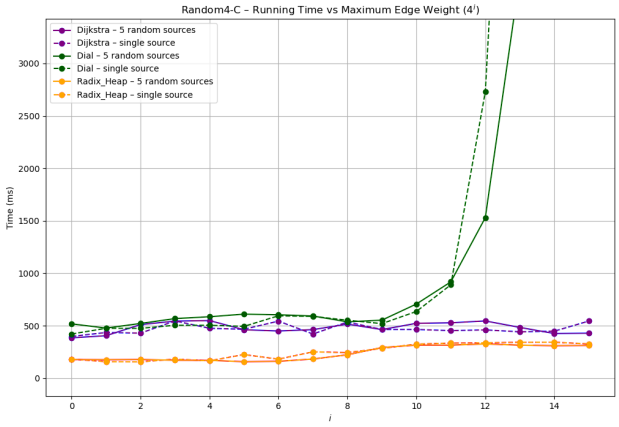
Należało przprowadzić doświadczenia dla określonych rodzin grafów:

- Long-C i Long-n
- Random4-n i Random4-C
- Square-C, Square-n
- USA-Road-d

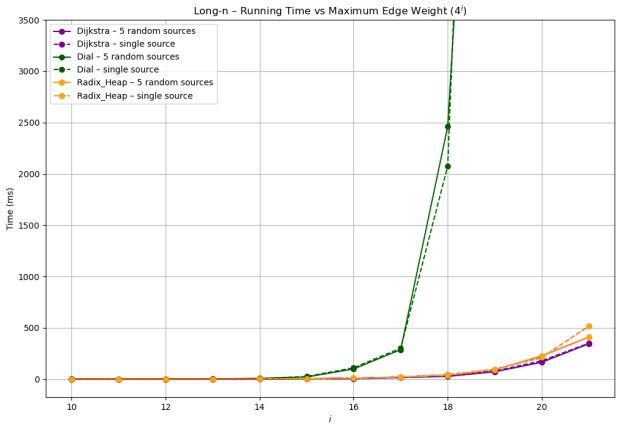
Dla każdej z wymienionych powyżej rodzin obliczamy czas znajdowania najkrótszych ścieżek dla źródła będącego wierzchołkiem o indeksie 1 oraz dla 5 losowo wybranch źródeł (wynik jest średnią czasu).
Ponieważ algorytm Diala jest bardzo wymagający pamięciowo, rozwiązanie powyższych problemów nie było możliwe dla niektórych grafów. Poniżej zamieszczone są wykresy z wynikiami eksperymentów.



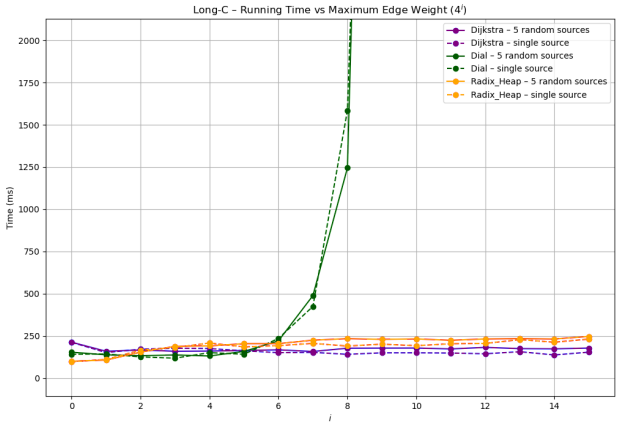
Rysunek 1: Random4-n



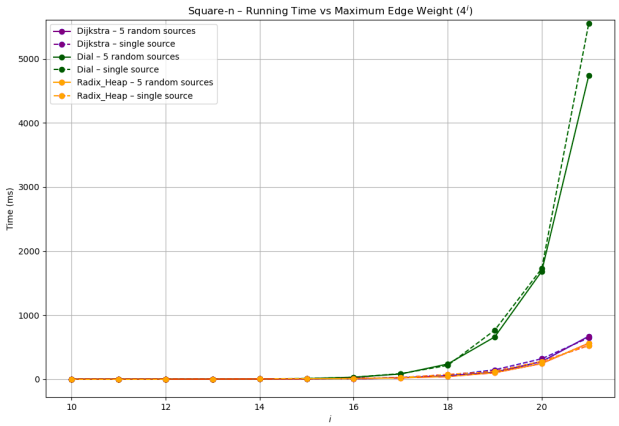
Rysunek 2: Random4-C



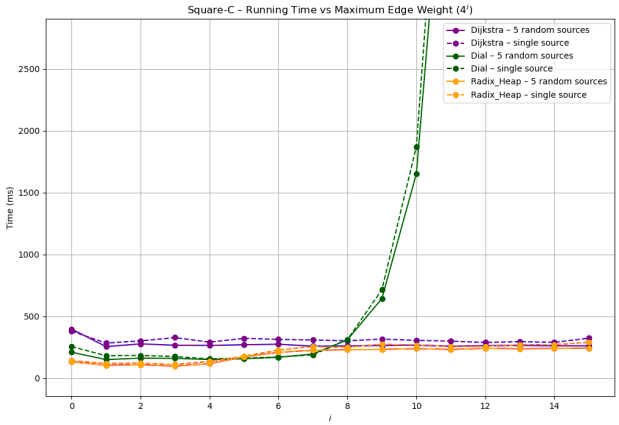
Rysunek 3: Long-n



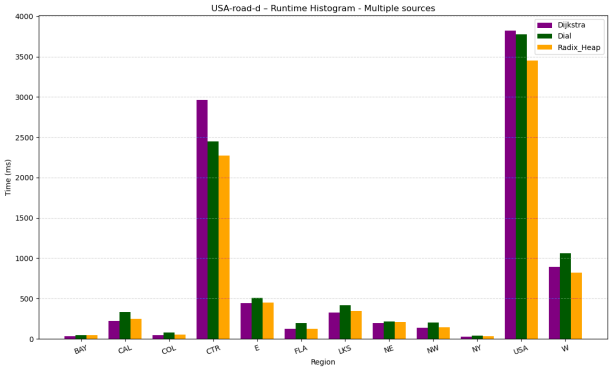
Rysunek 4: Long-C



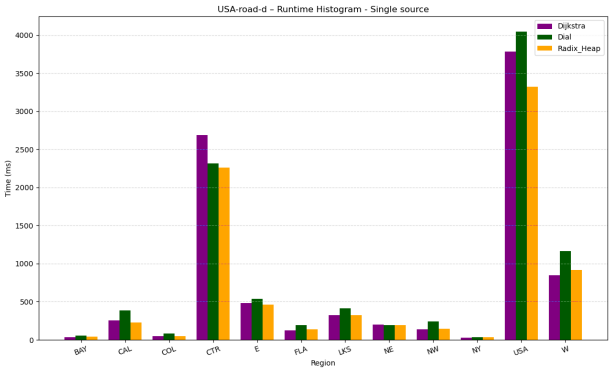
Rysunek 5: Square-n



Rysunek 6: Square-C



Rysunek 7: USA-Road-d (5 sources)



Rysunek 8: USA-Road-d (single source)

6 Wnioski

Algorytmy Dijkstry i RadixHeap dają zbliżone czasy działania, przy czym RadixHeap zwykle jest nieco szybszy, z wyjątkiem rodzin Long-C i Long-n. W przypadku grafów z różnymi wagami krawędzi (warianty „-C"), szczególnie Square-C, algorytm Diala może być bardzo efektywny dla małych wartości wag, czasem przewyższając pozostałe metody. Jednak wraz ze wzrostem wag jego wydajność gwałtownie spada i czasy obliczeń stają się wielokrotnie większe. Tymczasem Dijkstra i RadixHeap utrzymują stabilny czas działania niezależnie od wartości najdroższej krawędzi, co czyni je lepszym wyborem dla grafów o nieznanych lub zróżnicowanych parametrach. Dial sprawdza się tylko przy niewielkich kosztach, natomiast dla rodzin takich jak Long-C przy dużych wagach jego czas wykona-

nia może sięgać minut lub godzin, podczas gdy pozostałe algorytmy kończą obliczenia w ułamku sekundy.

7 Wyniki - najkrótsze ścieżki dla największych grafów

W tej sekcji zajmieny się największymi grafami dla każdej rodziny. Celem zadania było wyznaczenie kosztu najkrótszej ścieżki pomiędzy wierzchołkiem o najmniejszym i największym indeksie oraz średnich czasów dla 4 losowo wybranych par.

Tabela 1: Porównanie czasów działania algorytmów

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	31.60
Long-n	Dijkstra	1	2097152	31336751771	291.20
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	860.50
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	3253.20
USA-road-d	Dial	1	23947347	23228284	2860.24
USA-road-d	RadixHeap	1	23947347	23228284	2845.07

Tabela 2: Wyniki dla losowych ścieżek

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

8 Wnioski

Poza rodziną USA-road-d, wszędzie tam, gdzie algorytm Dial kończy obliczenia w rozsądnym czasie, pozostaje najwolniejszy spośród trzech porównywanych metod. Szczególnie wyraźnie widać to w rodzinie Long-n, gdzie rosnące wartości C są powiązane ze wzrostem liczby wierzchołków. W większości przypadków czasy działania Dijkstry i RadixHeap są zbliżone, z wyjątkiem rodziny Long-C, gdzie dla ścieżki rozpoczynającej się w pierwszym wierzchołku oba algorytmy były kilkukrotnie szybsze. Warto też zauważyć, że w rodzinie USA-road-d, charakteryzującej się stosunkowo niewielkimi wagami krawędzi, Dial wypada lepiej niż Dijkstra, choć nadal ustępuje RadixHeap, który pozostaje najszybszy. Ogromne wagi krawędzi od razu dyskwalifikują algorytm Diala. Wobec tego gdy wagi są nam nieznane lepiej używać np. klasycznej Dijkstry. Gdy znamy ograniczenie na wagi krawędzi możemy pokusić się o RadixHeap, ponieważ potrafi być szybszy od Dijkstry.