Alogrytmy Optymalizacji Dyskretnej

Lista 3 - Najkrótsze Ścieżki w Grafach

Rafał Włodarczyk 279762

2025-11-05

1 Model SSSP

Problem SSSP (Single Source Shortest Path) w grafie skierowanym $G = (V, E)$ polega na znalezieniu najtkrótszych ścieżek ze źródła $s \in S$ do wszystkich innych wierzchołków $V \setminus \{s\}$ z wagami krawędzi $w : E \to \mathbb{R}_{\geq 0}$. Najkrótszą ścieżkę $p = (v_0, v_1, \ldots, v_k)$, z wierzchołka v_0 do v_k definiujemy jako taką ścieżkę, dla której sumaryczna waga:

$$w(p) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

jest najmniejsza. Definiujemy odległość $d(u, v)$ jako wagę najkrótszej ścieżki z u do v:

$$d(u,v) = \begin{cases} \min\{w(p) : u \to v\} & \text{jeśli istnieje ścieżka z } u \text{ do } v \\ \infty & \text{w przeciwnym wypadku} \end{cases}$$

Dla uproszczenia zapisu V w rozważaniach złożności oznacza $|V|$, a E oznacza $|E|$.

2 Algorytm Dijkstry

Chcemy wyznaczyć długości najkrótszych ścieżek z każdego źródła $s \in S$ do wszystkich wierzchołków $V \setminus \{s\}$. Rozpoczynamy od jednego ze źródeł $s \in S$, następnie przechodzimy po wszystkich połączeniach wychodzących z s, aktualizujac odległości do sąsiadów v jako $d(s, v) = w(s, v)$. O kolejności wyboru następnego wierzchołka do odwiedzenia decyduje minimalna odległość od źródła spośród jeszcze nieodwiedzonych wierzchołków - możemy ją efektywnie przechowywać za pomocą struktury implementujacej funkcję `extract-min` - np. kopiec binarny. Po wybraniu wierzchołka u do odwiedzenia, przechodzimy po wszystkich jego sąsiadach v i sprawdzamy, czy odległość do nich może zostać zmniejszona, gdybyśmy przeszli przez u. Proces powtarzamy aż do przejścia wszystkich wierzchołków osiągalnych z s. Dla grafu spójnego będą to wszystkie wierzchołki i wszystkie krawędzie.

2.1 Implementacja Algorytmu Dijkstry

Z dokumentacji `std::priority_queue`[1] wynika, że operacje insert oraz extract-min mają złożność $O(\log n)$, gdzie n to liczba elementów w kolejce, a inicjalizacja kolejki to $O(1)$.

Algorithm 1 Algorytm Dijkstry dla problemu SSSP

for all $v \in V$ **do**
$d[v] \leftarrow \infty$, $\pi[v] \leftarrow$ none ▷ $O(V)$
end for
$d[s] \leftarrow 0$, $Q \leftarrow$ priority queue with $(d[s], s)$
while $Q \neq \emptyset$ **do** ▷ $O(V + E)$
  $(d[u], u) \leftarrow Q$.extract-min() ▷ $O(\log V)$
  for all $(u, v) \in E$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
      $d[v] \leftarrow d[u] + w(u, v)$
      $\pi[v] \leftarrow u$
      Q.insert$(d[v], v)$ ▷ $O(\log V)$
    end if
  end for
end while
return d, π ▷ Lista odległości i poprzedników

Cytując `families.pdf`[2] z załącznika do laboratorium *only sparse graphs are of interest, as for other graphs arc scans dominate the running time*. Oznacza to, że badane grafy będą rzadkie, zatem wybieram listę sąsiedztwa jako wewnętrzną reprezentację grafu.

2.2 Złożność Obliczeniowa Algorytmu Dijkstry

Na podstawie analizy implementacji algorytmu możemy stwierdzić, że jego złożoność wynosi.

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

3 Alogrytm Diala

Algorytm Diala to modyfikacja algorytmu Dijkstry, w której wykorzystujemy fakt, że wagi krawędzi są ograniczone do przedziału $[0, C]$, wobec czego możemy usprawnić wybór następnego wierzchołka z etykietami tymczasowymi. W i-tym kroku algorytmu długości ścieżek znajdą się w przedziale $[0, iC]$. Możemy zatem wykorzystać cykliczną listę kubełków, gdzie j-ty kubełek przechowuje wierzchołki z etykietami tymczasowymi $j, j+(C+1), j+2(C+1), \ldots$. Skoro za każdym razem opróżniamy kubełek, to etykiety nie będą na siebie nachodzić.

[1] `https://en.cppreference.com/w/cpp/container/priority_queue.html`
[2] `http://www.dis.uniroma1.it/challenge9`

3.1 Implementacja Algorytmu Diala

Algorithm 2 Algorytm Diala dla problemu SSSP

```
buckets ← cyclic list(C + 1)
buckets[0] ← {s}
distances[s] ← 0
current ← 0
while true do
    while buckets[current] is empty do
        current ← (current + 1) mod (C + 1)          ▷ Szukamy niepustego kubełka
    end while
    while buckets[current] is not empty do           ▷ Opróżniamy bieżący kubełek
        u ← buckets[current]
        remove u from buckets[current]
        for all (v, w) ∈ graph[u] do                 ▷ Patrzmy na sąsiadów u
            newDist ← distances[u] + w
            if newDist < distances[v] then           ▷ Krótsza ścieżka znaleziona
                remove v from buckets[distances[v] mod C + 1]
                distances[v] ← newDist
                add v to buckets[newDist mod C + 1]
            end if
        end for
    end while
end while
```

3.2 Złożność Obliczeniowa Algorytmu Diala

Zobaczmy, że tak samo jak w algorytmie Dijkstry, rozpatrujemy każdą krawędź dokładnie raz, a każdy wierzchołek zostanie przejrzany maksymalnie VC razy. Zatem złożoność algorytmu Diala wynosi:

$$O(E + VC)$$

Widzimy, że algorytm jest pseudowielomianowy, dla dużych C będzie znacznie mniej wydajny niż algorytm Dijkstry.

4 Algorytm Radix Heap

Algorytm Radix Heap to modyfikacja algorytmu Dijkstry, w której ponownie zakładamy wagi z przedziału $[0, C]$, wykorzystujemy za to inny sposób przetrzymywania etykiet tymczasowych. Delegujemy przechowywanie etykiet do struktury Radix Heap, która pozwala na wykonywanie operacji `insert` oraz `extract-min`. Szerokości kolejnych kubełków są kolejnymi potęgami 2, zatem potrzeba jedynie $O(\log VC)$ kubełków, aby pokryć cały przedział

wag krawędzi. Uproszczenie podobne do algorytmu Diala pozwala zredukować liczbę kubełków do $O(\log C)$. Rozdzielamy zawartość opróżnianego kubełka do wszystkich kubełków mniejszych szerokości. W przypadku Diala robiliśmy tak tylko do poprzedniego kubełka, tutaj musimy rozdzielić do wszystkich mniejszych, ponieważ szerokości kubełków nie są równe.

4.1 Złożność Obliczeniowa Algorytmu Radix Heap

Ponieważ struktura Radix Heap posiada złożoność operacji `insert` w czasie $O(1)$ oraz `extract-min` w czasie $O(\log(VC))$, to złożoność całego algorytmu Radix Heap wynosi:

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

Wykładowe uproszczenie pozwala zredukować złożoność do:

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

Zauważmy natomiast, że dla małych wartości C możemy zbliżać się do złożoności liniowej od wielkości grafu. Jeśli działamy na `64`-bitowych liczbach nieujemnych, to $\log(C) \leq 64$, zatem możemy uprościć złożoność do:

$$O(E + V)$$

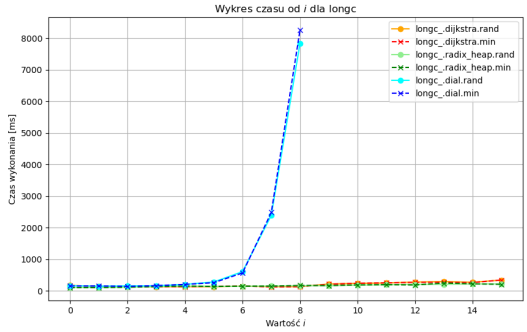
5 Wykresy dla eksperymentów SSSP

Eksperymenty zostaną przeprowadzone dla następujących rodzin grafów:

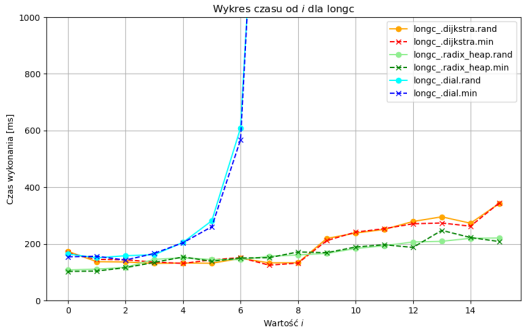
- Long 4C, Long 4N
- Random 4C, Random 4N
- Square C, Square N
- USA-Road-d (distances)

Dla każdej rodziny liczymy SSSP od wierzchołka z najmniejszym indeksem, oraz średnią dla pięciu losowo wybranych wierzchołków startowych. Na wykresach kolorem czerwonym oraz pokrewnymi oznaczam algorytm Dijkstry, niebieskim algorytm Diala, a zielonym algorytm Radix Heap.

5.1 Long C

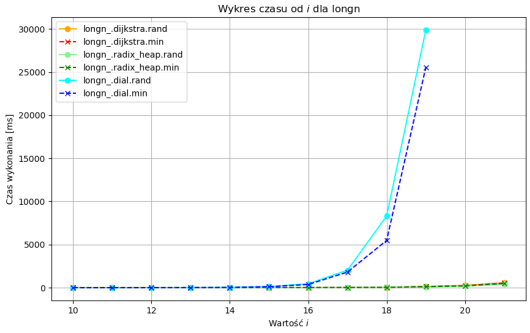


Rysunek 1: longc_plot.png

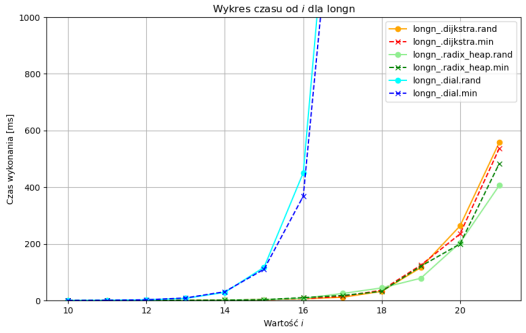


Rysunek 2: longc_plot_zoomed.png

5.2 Long N

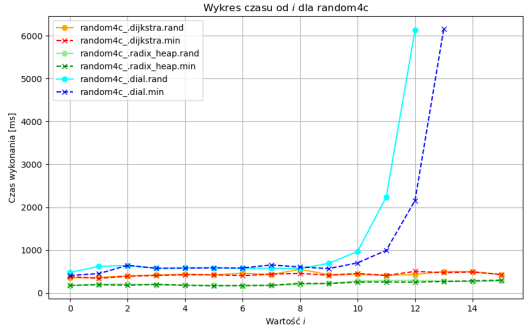


Rysunek 3: longn_plot.png

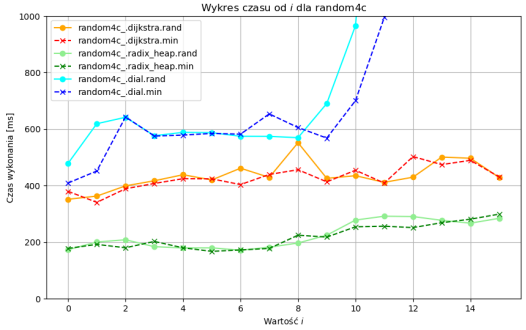


Rysunek 4: longn_plot_zoomed.png

5.3 Random 4C

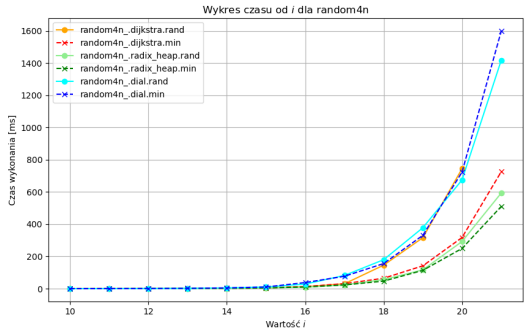


Rysunek 5: random4c_plot.png

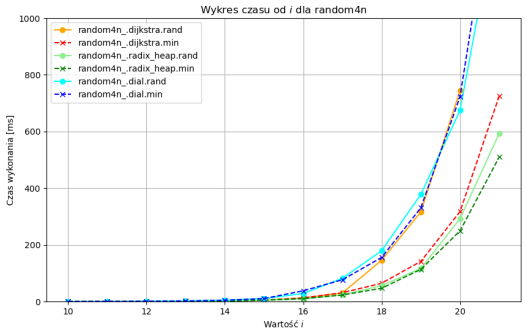


Rysunek 6: random4c_plot_zoomed.png

5.4 Random 4N

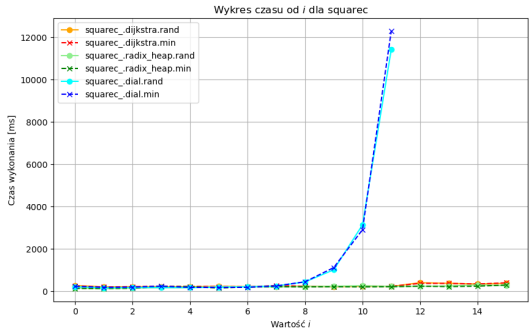


Rysunek 7: random4n_plot.png

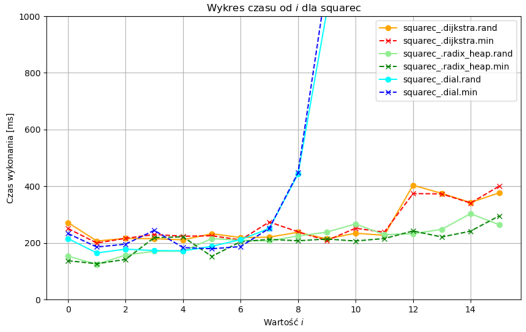


Rysunek 8: random4n_plot_zoomed.png

5.5 Square C

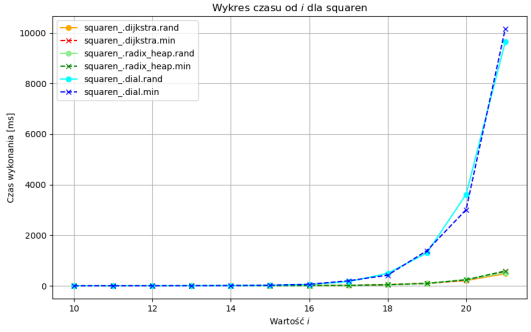


Rysunek 9: squarec_plot.png

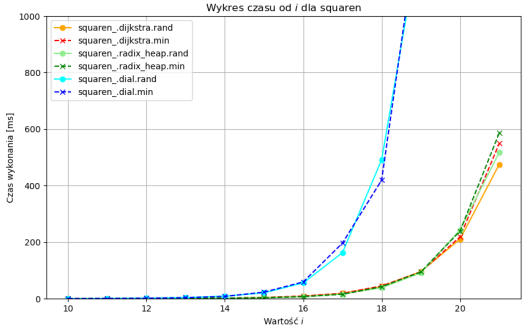


Rysunek 10: squarec_plot_zoomed.png

5.6 Square N

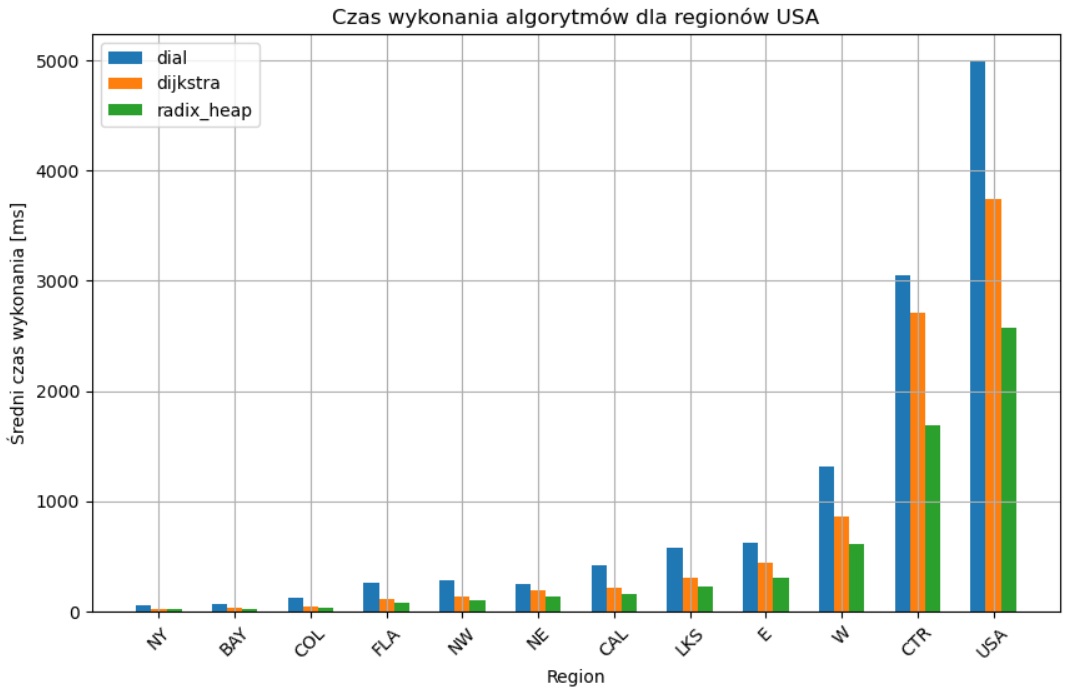


Rysunek 11: squaren_plot.png



Rysunek 12: squaren_plot_zoomed.png

5.7 USA



Rysunek 13: usa_bar_chart.png

5.8 Obserwacje SSSP

Algorytm Dijkstry prezentuje stabilne wyniki dla wszystkich rodzin grafów, zbliżone do teoretycznej złożoności $O((V + E) \log V)$. Widzimy, że dla rodzin grafów LongC, Random4C oraz SquareC zwiększająca się maksymalna waga krawędzi C nie wpływa na czas działania algorytmu Dijkstry, co jest zgodne z teoretyczną analizą złożoności. Widzimy również, jak w rodzinach LongN, Random4N oraz SquareN, czas działania algorytmu Dijkstry rośnie wraz z liczbą wierzchołków N, co również jest zgodne z teoretyczną analizą złożoności.

Zauważmy, że algorytm Diala, ze względu na czynnik C bardzo źle radzi sobie dla rodzin grafów LongC, Random4C oraz SquareC, gdzie maksymalna waga krawędzi jest bardzo duża, z tego również powodu wykonywanie dalszych obliczeń $i > 10$ dla zadanych rodzin grafów jest niepraktycznie i bezcelowe. Osiągnięta asymptotyka wyraźnie pokazuje oczekiwane zjawisko, a teoretyczny model zostaje potwierdzony. Ciekawe jest natomiast to, iż dla małych wielkości C, algorytm Diala radzi sobie porównywalnie, lub nawet nieznacznie lepiej niż pozostałe algorytmy. Najbardziej wymagającą rodziną dla algorytmu Diala okazuje się rodzina LongC, gdzie czas działania staje się ogromny już dla $C = 7$.

Usprawnienie wnoszone przez algorytm Radix Heap jest widoczne w prawie wszystkich rodzinach grafów, gdzie algorytm radzi sobie lepiej niż dwa pozostałe. Dla rodziny Random4N widzimy niewiele gorsze zachowanie niż algorytm Dijkstry.

6 Tabela wyników dla eksperymentów P2P

Eksperyment P2P polega na zmierzeniu odległości i czasu między wierzchołkiem o najmniejszym indeksie, a wierzchołkiem o największym indeksie, ponadto mierzymy średni czas dla czterech innych losowo wybranych par wierzchołków.

Jako przedstawicieli wybieramy największe grafy z każdej rodziny, które udało się przetworzyć w rozsądnym czasie. W przypadku algorytmu Diala pomijam przypadki, gdzie C jest zbyt duże i algorytm nie radzi sobie z przetworzeniem grafu. Usprawnienie $VC + 1$ do $C + 1$ liczby kubełków eliminuje błędy OOM (Out Of Memory) w przypadku dużych i, ale wciąż pozostaje problem absurdalnego czasu działania.

Rodzina - Algorytm	Start	Koniec	Odległość	Czas (ms)
Long-C - dijkstra	1	1048576	1308259008765	21.6291
Long-C - dial	brak (timeout)			
Long-C - radix_heap	1	1048576	1308259008765	21.7272
Long-n - dijkstra	1	2097152	31336751771	277.804
Long-n - dial	1	2097152	31336751771	3456.40
Long-n - radix_heap	1	2097152	31336751771	253.764
Random4-C - dijkstra	1	1048576	3471241820	126.504
Random4-C - dial	1	1048576	3471241820	683.072
Random4-C - radix_heap	1	1048576	3471241820	96.2762
Random4-n - radix_heap	1	2097152	9051281	314.371
Random4-n - dial	1	2097152	9051281	1400.29
Random4-n - dijkstra	1	2097152	9051281	513.474
Square-C - dijkstra	1	1048576	122219500320	117.732
Square-C - dial	brak (timeout)			
Square-C - radix_heap	1	1048576	122219500320	82.4341
Square-n - dijkstra	1	2096704	714640488	258.535
Square-n - dial	1	2096704	714640488	1043.21
Square-n - radix_heap	1	2096704	714640488	192.932
USA-road-d - dijkstra	1	23947347	23228284	3215.28
USA-road-d - dial	1	23947347	23228284	4722.40
USA-road-d - radix_heap	1	23947347	23228284	2351.68

Tabela 1: Wyniki eksperymentów P2P dla różnych rodzin grafów i algorytmów

Zobaczmy następnie średnie czasy przeszukiwania losowych par wierzchołków:

Rodzina - Algorytm	Czas (ms)
Long-C - dijkstra	117.798
Long-C - dial	brak (timeout)
Long-C - radix_heap	84.2331
Long-n - dijkstra	101.535
Long-n - dial	2432.8
Long-n - radix_heap	139.14
Random4-C - dijkstra	248.834
Random4-C - dial	645.152
Random4-C - radix_heap	213.556
Random4-n - dijkstra	300.834
Random4-n - dial	1402.4
Random4-n - radix_heap	355.472
Square-C - dijkstra	192.187
Square-C - dial	brak (timeout)
Square-C - radix_heap	86.0959
Square-n - dijkstra	435.333
Square-n - dial	1134.6
Square-n - radix_heap	330.171
USA-road-d - dijkstra	60.7381
USA-road-d - dial	4427.67
USA-road-d - radix_heap	43.624

Tabela 2: Uśrednione czasy wykonania dla różnych rodzin grafów i algorytmów

6.1 Obserwacje P2P

Zobaczmy, że algorytm Diala ponownie źle radzi sobie z dużymi wartościami C, co jest zgodne z wcześniejszymi obserwacjami. Dla rodziny LongN notujemy absurdalny czas działania. Alogrytm Radix Heap oraz algorytm Dijkstry radzą sobie porównywalnie, ciężko ocenić który z nich jest lepszy, ponieważ w niektórych przypadkach Radix Heap jest szybszy, a w innych Dijkstra. Warto zauważyć, że dla rodziny USA-road-d Radix Heap radzi sobie zauważalnie lepiej, a algorytm Diala nie działa aż tak źle.

7 Wnioski

Wykresy potwierdzają teoretyczne złożoności algorytmów Dijkstry, Diala oraz Radix Heap. Wybór standardowego algorytmu Dijkstry jest wskazany, gdy nie mamy ograniczeń na maksymalną wagę krawędzi C, lub gdy C jest duże. W przypadku małych wartości C, algorytm Diala może być konkurencyjny, jednakże jego złożoność pseudowielomianowa czyni go mniej uniwersalnym. Algorytm Radix Heap wydaje się być najlepszym wyborem w przypadku gdy posiadamy solidną implementację, oraz gdy ograniczamy C do rozmiaru 64-bitowej liczby całkowitej. Wtedy jego złożoność zbliża się do liniowej względem rozmiaru grafu, co jest bardzo korzystne.