

# Sprawozdanie – Laboratoria 3.

Jakub Kogut

9 grudnia 2025

## 1 Wprowadzenie

Na liście 3. mamy za zadanie przeprowadzić implementację i analizę algorytmów znajdowania najkrótszych ścieżek w grafach. Za dane mamy użyć grafy wygenerowane przez generator oraz sieć drogową USA napisane przez DIMACS.

## 2 Implementacja

Wszystkie zadania zaimplementowałem w języku C++. Kody algorytmów znajdują się w `algo.cpp`, natomiast instrukcje uruchamiania w `README.md`. Wszystkie programy korzystają wyłącznie z biblioteki standardowej C++, można je kompilować przy użyciu `make`

### 2.1 Algorytm DIJKSTRY

Graf przechowuję w klasycznej postaci list sąsiedztwa: dla każdego wierzchołka  $u$  tablica  $g[u]$  zawiera listę par  $(v, w)$  oznaczających łuk  $u \rightarrow v$  o koszcie  $w$ . Typ `Distance` jest aliasem na typ całkowity (w praktyce `long long`), a wartość `INF` reprezentuje „nieskończoną” odległość.

Implementacja funkcji `dijkstra` odpowiada kopcowej wersji algorytmu Dijkstry, w której operację wyboru wierzchołka o najmniejszej etykiecie realizuje się przez kolejkę priorytetową. `std::priority_queue` przechowuje pary (`dystans, wierzchołek`), a dzięki komparatorowi `std::greater` działa jak kopiec minimalny. Zamiast operacji `decrease-key` stosuję typowy „leniwy” wariant: przy każdej poprawie etykiety wierzchołka dokładam nową parę do kolejki, a przy wyjmowaniu ignoruję wpisy nieaktualne (sprawdzam warunek `if (d > dist[u]) continue;`).

W uproszczonym pseudokodzie:

```
dijkstra(G, s):
    for v in V:
        dist[v] = INF
    dist[s] = 0
    PQ = min-kolejka priorytetowa par (odległość, wierzchołek)
    PQ.push( (0, s) )

    while PQ niepusta:
        (d, u) = PQ.pop_min()
        if d != dist[u]:
            continue      // stary wpis w kolejce

        for (v, w) w G[u]:
            if d + w < dist[v]:
                dist[v] = d + w
                PQ.push( (dist[v], v) )
```

Zgodnie z klasyczną analizą, dla kopca binarnego każda operacja `push/pop` ma koszt  $O(\log n)$ . Wykonujemy  $O(n)$  operacji `pop` oraz  $O(m)$  wstawek, więc łączna złożoność czasowa wynosi

$$T(n, m) = O((m + n) \log n),$$

co w grafie spójnym zwykle zapisuje się jako  $O(m \log n)$ . Złożoność pamięciowa to  $O(n + m)$  na przechowywanie grafu i tablicy odległości.

## 2.2 Algorytm DIALA

Algorytm Diala zakłada, że wszystkie koszty łuków są całkowite i nieujemne oraz że znamy z góry maksymalny koszt pojedynczej krawędzi

$$C = \max\{c_{ij}\}.$$

W implementacji przekazuję  $C$  jako dodatkowy parametr funkcji `dial`. Korzystam z wersji „cyklicznej”: zamiast  $nC + 1$  kubełków używam tylko  $C + 1$  kubełków indeksowanych modulo  $C + 1$ .

Tablica `buckets` ma rozmiar `binCount = C+1`, a kubełek o indeksie  $k$  przechowuje wierzchołki  $v$  z tymczasową etykietą

$$d(v) \equiv k \pmod{C + 1}.$$

Dzięki właściwości, że w każdym momencie wszystkie skończone etykiety leżą w przedziale  $[d_{\min}, d_{\min} + C]$ , wystarczy  $C + 1$  kubełków i indeksowanie modulo.

W dużym uproszczeniu implementacja wygląda tak:

```
dial(G, s, C):
    for v in V:
        dist[v] = INF
    dist[s] = 0

    binCount = C + 1
    buckets[0..binCount-1] = puste listy
    currentBucket = 0
    buckets[0].push_back(s)

    while istnieje niepusty kubełek:
        // szukamy pierwszego niepustego kubełka
        steps = 0
        while buckets[currentBucket] pusty
            if steps < binCount:
                currentBucket = (currentBucket + 1) mod binCount
            steps++

        if steps == binCount i buckets[currentBucket] pusty:
            break // wszystkie kubełki puste

        u = ostatni element z buckets[currentBucket]
        usuń u z tego kubełka
        du = dist[u]

        for (v, w) w G[u]:
            if du + w < dist[v]:
                dist[v] = du + w
```

```

idx = (dist[v]) mod binCount
buckets[idx].push_back(v)

```

Kolejne wierzchołki są wybierane „wiadrami” o rosnących etykietach, co symuluje sortowanie po odległości, ale w czasie stałym na pojedynczą relaksację. Każdą krawędź relaksujemy co najwyżej raz z sukcesem, więc łączny koszt części *distance update* jest  $O(m)$ . Przeszukanie kubełków daje w najgorszym wypadku koszt  $O(nC)$ , co prowadzi do złożoności:

$$T(n, m, C) = O(m + nC), \quad S(n, C) = O(n + C).$$

Jest to więc algorytm pseudowielomianowy – zależny liniowo od maksymalnej wagi  $C$ . Przy małych wartościach  $C$  (np. gdy wagi to małe liczby całkowite) algorytm Diala bywa w praktyce znacznie szybszy niż klasyczny Dijkstra.

### 2.3 Algorytm RADIX HEAP

Trzeci wariant to algorytm Dijkstry, w którym kolejkę priorytetową zaimplementowałem jako specjalizowaną strukturę `RadixHeap<Distance>`. Jest to monotoniczny *radix heap*: zakładamy, że klucze (tu: aktualne odległości  $d(u)$ ) są całkowite, nieujemne oraz że wartości wyjmowane z kopca są niemalejące. Ta własność zachodzi dla Dijkstry z wagami  $c_{ij} \geq 0$ , dlatego w implementacji asercja `assert(x >= last);` sprawdza, czy nowy klucz nie jest mniejszy od ostatnio wyjętego.

Struktura `RadixHeap` składa się z 65 kubełków `buckets[0..64]`, gdzie `U = unsigned long long` jest wewnętrznym typem klucza (wystarcza to dla `Distance = long long`). Kubełki mają zróżnicowane „szerokości”, wyznaczane przez najwyższy bit, na którym klucz różni się od `last`. Funkcja `bucketIndex(x)` oblicza indeks kubełka jako:

- jeśli  $x = \text{last}$ , zwraca 0,
- w przeciwnym razie oblicza  $\text{diff} = x \oplus \text{last}$ , znajduje pozycję najstarszego ustawionego bitu w `diff` i zwraca `msb + 1`, co daje kubełki numerowane od 1 do 64.

Operacja `push` jedynie wrzuca element (`key, v`) do odpowiedniego kubełka. Gdy kubełek 0 (z aktualnym minimum) się opróżni, wywoływana jest funkcja `pull`, która:

1. wyszukuje pierwszy niepusty kubełek  $i > 0$ ,
2. wyznacza nową wartość `last` jako minimalny klucz znajdujący się w tym kubełku,
3. przenosi wszystkie elementy z kubełka  $i$  do kubełków o mniejszych indeksach, zgodnie z nową wartością `last`.

Dzięki temu kubełek 0 zawsze zawiera elementy o najmniejszym aktualnym kluczach.

```

radixheap(G, s):
    for v in V:
        dist[v] = INF
    dist[s] = 0

    RH = pusty RadixHeap
    RH.push(0, s)

    while RH niepusty:
        (d, u) = RH.top()
        RH.pop()
        if d > dist[u]:
            continue      // stary wpis

        for (v, w) w G[u]:
            if dist[u] + w < dist[v]:

```

```

dist[v] = dist[u] + w
RH.push(dist[v], v)

```

W strukturze Radix Heap każdy wierzchołek może zostać „przepakowany” pomiędzy kubelkami co najwyżej  $K = O(\log(nC))$  razy, gdzie  $C$  jest górnym ograniczeniem na długości najkrótszej ścieżki (tzn. na etykiety  $d(\cdot)$ ). Łączny koszt wszystkich przeniesień i operacji wyboru wierzchołka to  $O(nK)$ , natomiast relaksacje krawędzi kosztują  $O(m)$ . Ostatecznie:

$$T(n, m, C) = O(m + n \log(nC)).$$

Przy odpowiednim doborze liczby kubeli można tę złożoność zapisać w postaci  $O(m + n \log C)$ . W praktyce, dla grafów o umiarkowanych wagach całkowitych, wariant z RadixHeap łączy zalety algorytmu Diala (operacje w czasie prawie stałym dla małych wag) z dobrą skalowalnością dla większych wartości  $C$ .

### 3 Analiza rodzin grafów

W części doświadczalnej korzystamy z rodzin grafów zdefiniowanych w ramach *9th DIMACS Implementation Challenge*. Rodziny te zostały zaprojektowane tak, aby w kontrolowany sposób zmieniać:

- strukturę grafu (losowy vs. siatka vs. sieć drogowa),
- głębokość drzewa najkrótszych ścieżek i typowy rozmiar zbioru aktualnie etykietowanych wierzchołków,
- zakres wag krawędzi  $C$  (istotny m.in. dla algorytmu Diala i RadixHeap).

Dzięki temu można obserwować, jak poszczególne struktury danych do Dijkstry zachowują się w różnych warunkach.

#### 3.1 Grafy losowe: rodziny Random4-n i Random4-C (podpunkt 3.1)

##### 3.1.1 Definicja i własności strukturalne

Grafy losowe są skierowane, z liczbą wierzchołków  $n$  i liczbą łuków  $m = 4n$ . Generator najpierw tworzy cykl Hamiltona (aby zapewnić silną spójność), a następnie losuje dodatkowe łuki, wybierając pary  $(v, w)$ ,  $v \neq w$ . Długości łuków są niezależnie i jednostajnie losowane z przedziału całkowitego  $[0, C]$ .

Takie grafy:

- są dobrymi *expanderami* (przy średnim stopniu  $\geq 4$ ),
- mają płytkie drzewa najkrótszych ścieżek (głębokość rzędu  $\Theta(\log n)$ ),
- w trakcie działania algorytmu Dijkstry typowa liczba aktualnie etykietowanych wierzchołków jest duża,
- mają słabą lokalność pamięciową (są „przemieszane”).

**Rodzina Random4-n.** W rodzinie Random4-n rośnie liczba wierzchołków, a zakres wag jest związany z rozmiarem grafu:

$$m = 4n, \quad C = n, \quad n = 2^{10}, 2^{11}, \dots, 2^{21}.$$

Zmiana  $n$  powiększa zarówno liczbę krawędzi, jak i zakres wag.

**Rodzina Random4-C.** W rodzinie Random4-C rozmiar grafu jest stały, a rośnie tylko zakres wag:

$$n = 2^{20}, \quad m = 4n, \quad C = 4^i, \quad i = 0, 1, \dots, 15.$$

### 3.1.2 Wpływ na badane algorytmy

- **Dijkstra (kopiec binarny).** Dla **Random4-n** złożoność teoretyczna to  $O(m \log n)$ , więc oczekujemy prawie liniowego wzrostu czasu z  $n$  pomnozonego przez czynnik  $\log n$ . Dla **Random4-C** struktura grafu się nie zmienia, więc czas powinien być praktycznie niezależny od  $C$  (zmienia się jedynie rozkład etykiet, nie zaś liczba operacji na kopcu).
- **Dial.** Złożoność  $O(m + nC)$  powoduje, że:
  - w rodzinie **Random4-n** czynnik  $nC = n^2$  szybko dominuje, przez co Dial jest konkurencyjny tylko dla małych  $n$ ,
  - w rodzinie **Random4-C** dla stałego  $n$  czas rośnie w przybliżeniu liniowo z  $C$ ; dla większych  $C$  algorytm staje się wyraźnie wolniejszy niż Dijkstra i **RadixHeap**.
- **Dijkstra + RadixHeap.** Teoretyczna złożoność to  $O(m + n \log(nC))$ . W praktyce:
  - w **Random4-n** czas rośnie z  $n$ , ale czynnik  $\log(nC) \approx \log n$  powoduje trochę łagodniejszy wzrost niż w kopcu binarnym,
  - w **Random4-C** zależność od  $C$  jest bardzo słaba (logarytmiczna), więc czas praktycznie nie zmienia się wraz z powiększaniem zakresu wag, w ostrym kontraście do Diala.

## 3.2 Grafy siatkowe: Long-n, Square-n, Long-C, Square-C (podpunkt 3.2)

### 3.2.1 Definicja i własności strukturalne

Grafy siatkowe mają strukturę prostokątnej siatki  $x \times y$ . Wierzchołki są połączone z sąsiadami w góre, w dół, w lewo i w prawo (z wyjątkiem brzegów), a długości łuków są losowane z  $[0, C]$ .

Rozróżniamy:

- **Long grids (Long-\*)** – „długie” siatki o stałej wysokości  $y = 16$  i rosnącej szerokości  $x$ . Drzewa najkrótszych ścieżek są bardzo głębokie  $\Theta(n)$ , a liczba aktualnie etykietowanych wierzchołków jest mała.
- **Square grids (Square-\*)** – siatki „prawie kwadratowe”,  $x \approx y \approx \sqrt{n}$ . Drzewa najkrótszych ścieżek mają umiarkowaną głębokość  $\Theta(\sqrt{n})$ , a liczba etykietowanych wierzchołków jest umiarkowana.

### 3.2.2 Rodziny Long-n i Square-n

W rodzinach „-n” rośnie liczba wierzchołków, a zakres wag jest związany z rozmiarem:

$$C = n, \quad n = 2^{10}, 2^{11}, \dots, 2^{21}.$$

Porównanie:

- **Struktura drzewa najkrótszych ścieżek.**
  - W **Long-n** najkrótsze ścieżki są długie (prawie liniowe), więc Dijkstra „idzie” po grafie dość sekwencyjnie; typowy rozmiar kolejki priorytetowej jest stosunkowo mały.
  - W **Square-n** ścieżki są krótsze, ale front fali BFS/Dijkstry jest szerszy, co prowadzi do większego zbioru etykietowanych wierzchołków jednocześnie.
- **Dijkstra (kopiec binarny).** Dla obu rodzin złożoność asymptotyczna pozostaje  $O(m \log n)$ , ale w praktyce:
  - w **Long-n** kolejka jest mniejsza, więc mimo tego samego rzędu złożoności można oczekiwać nieco krótszych czasów,
  - w **Square-n** więcej operacji na kopcu (większy front), co przekłada się na gorsze czasy niż w **Long-n** przy tym samym  $n$ .
- **Dial i RadixHeap.** Zakres wag  $C = n$  powoduje, że Dial teoretycznie ma koszt  $O(m + n^2)$  i przy większych rozmiarach będzie wyraźnie przegrywał z pozostałymi algorytmami, mimo

korzystnej struktury frontu. RadixHeap pozostaje dużo bardziej odporne na wzrost  $C$ , więc różnice między Long-n i Square-n wynikają głównie z rozmiaru frontu, nie z zakresu wag.

### 3.2.3 Rodziny Long-C i Square-C

W rodzinach „-C” liczba wierzchołków jest stała ( $n = 2^{20}$  w definicji; w przypadku bardzo dużych  $C$  dopuszcza się zmniejszenie np. do  $n = 2^{15}$ ), a rośnie jedynie zakres wag:

$$C = 4^i, \quad i = 0, 1, \dots, 15.$$

- **Dijkstra (kopiec binarny).** Ponieważ struktura grafu się nie zmienia, a liczba operacji na kopcu nie zależy bezpośrednio od wartości wag, czas działania praktycznie nie zależy od  $C$  (dla bardzo dużych  $C$  może pojawić się jedynie subtelny wpływ na kolejność relaksacji).
- **Dial.** Tutaj wpływ  $C$  jest kluczowy:
  - dla małych  $C$  (np.  $C = 1, 4, 16$ ) liczba kubelków jest mała, więc algorytm Diala bywa bardzo szybki,
  - wraz ze wzrostem  $C$  czas rośnie praktycznie liniowo (koszt  $O(nC)$ ), co dla największych wartości  $4^{15}$  może prowadzić do bardzo długich czasów lub wręcz przerwania testów; w takiej sytuacji sensowne jest zmniejszenie  $n$  (np. do  $2^{15}$ ), aby mimo wszystko zaobserwować trend zależności od  $C$ .
- **Dijkstra + RadixHeap.** Teoretyczna złożoność  $O(m + n \log(nC))$  oznacza, że:
  - przy rosnącym  $C$  obserwujemy bardzo łagodny wzrost czasu (logarytmiczny w  $C$ ),
  - RadixHeap jest dzięki temu dużo bardziej skalowalne niż Dial; nawet dla dużych  $C$  pozostaje konkurencyjne wobec kopca binarnego.
  - różnice między Long-C i Square-C wynikają głównie z rozmiaru frontu (jak w rodzinach „-n”), a nie z  $C$ .

## 3.3 Grafy drogowe: rodziny USA-road-d i USA-road-t (podpunkt 3.3)

### 3.3.1 Definicja i struktura sieci drogowych

Rodziny USA-road-d oraz USA-road-t opisują rzeczywiste sieci drogowe USA, zbudowane na podstawie danych TIGER/Line. Poszczególne instancje (np. USA, CTR, CAL, NY) różnią się rozmiarem, ale wszystkie:

- są bardzo rzadkie (średni stopień niewielki, bliski stałemu),
- mają silną strukturę geometryczną (drogi osadzone w 2D),
- charakteryzują się dobrą lokalnością pamięciową (wierzchołki odpowiadające bliskim położeniom geograficznym są często blisko również w numeracji),
- zawierają dwie naturalne funkcje wag:
  - długości odcinków (USA-road-d),
  - czasy przejazdu (USA-road-t).

Typowe drzewa najkrótszych ścieżek mają umiarkowaną głębokość, a liczba aktualnie etykietowanych wierzchołków jest relatywnie mała (ścieżki zwykle podążają wzduż kilku głównych korytarzy komunikacyjnych).

### 3.3.2 Wpływ na badane algorytmy

- **Dijkstra (kopiec binarny).** Dla grafów drogowych jest to klasyczny punkt odniesienia. Duża rzadkość grafu sprawia, że część  $O(m)$  zdominowana jest przez liczbę łuków, a czynnik  $\log n$  jest stosunkowo łagodny. W praktyce obserwuje się dobre skalowanie nawet dla największych instancji.

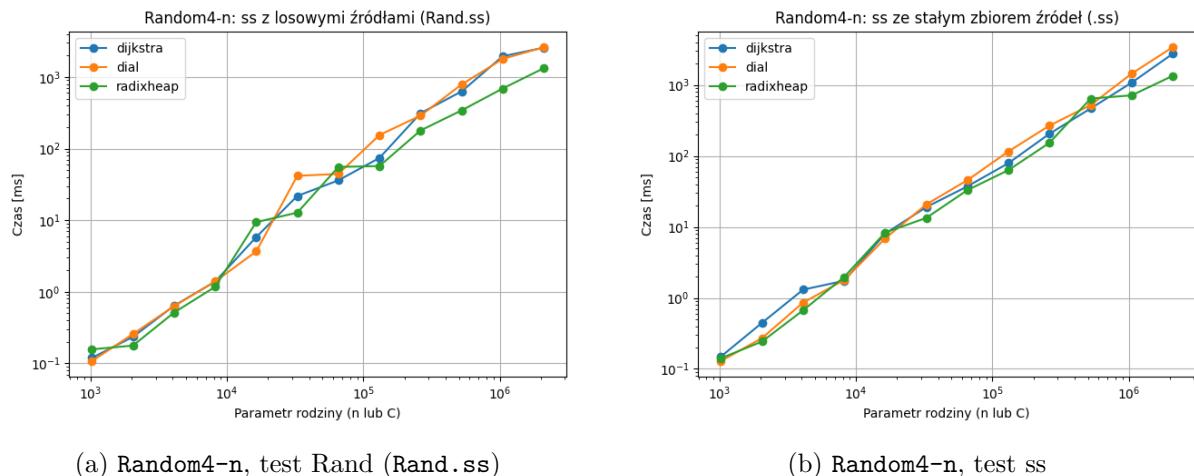
- **Dial.** Wagi na sieciach drogowych są zwykle przechowywane jako liczby całkowite, ale ich zakres może być duży (setki tysięcy lub więcej po odpowiednim przeskalowaniu). Oznacza to:
  - dla lokalnych podgrafów o małych rozpiętościach wag Dial może być konkurencyjny,
  - dla pełnych instancji o dużym  $C$  liczba kubełków rośnie na tyle, że koszt  $O(nC)$  eliminuje przewagę nad Dijkstrą.
- **Dijkstra + RadixHeap.** Dzięki monotonicznemu charakterowi etykiet w Dijkstrze i umiarowanemu zakresowi wag w praktycznych danych, RadixHeap dobrze dopasowuje się do grafów drogowych:
  - złożoność  $O(m + n \log(nC))$  jest bliska liniowej w liczbie krawędzi,
  - silna lokalność i mały stopień wierzchołków przekładają się na dobrą efektywność pamięciową i czasową,
  - w porównaniu do kopca binarnego zwykle obserwuje się podobne lub nieco lepsze czasy, szczególnie na większych instancjach.

Podsumowując, rodziny grafów Random, Grid i USA-road pozwalają osobno zbadać wpływ:

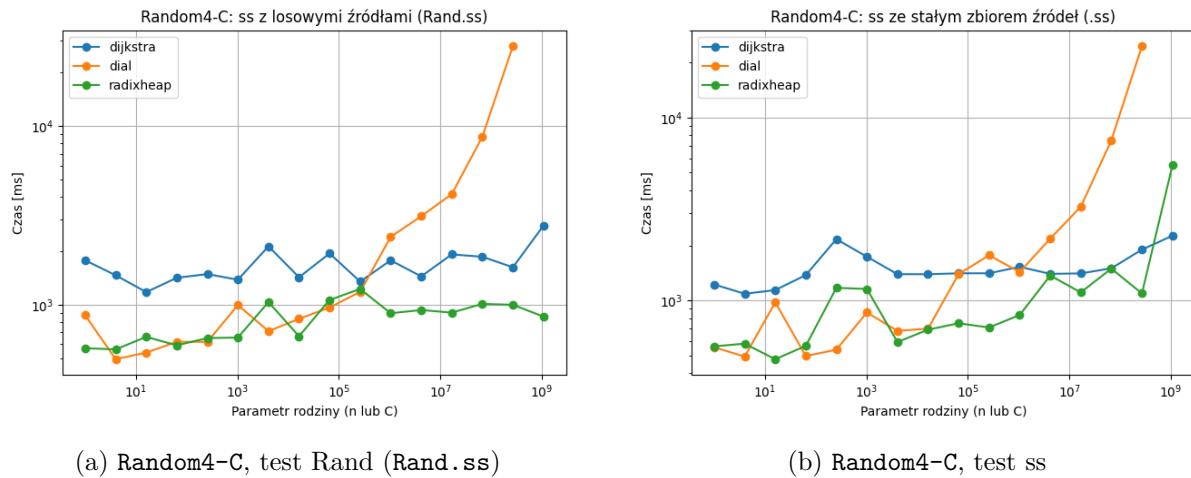
1. rozmiaru grafu i liczby krawędzi (Random4-n, Long-n, Square-n),
2. zakresu wag  $C$  przy stałym grafie (Random4-C, Long-C, Square-C),
3. realistycznej struktury geometrycznej i lokalności pamięciowej (USA-road-d/t),

na względną wydajność trzech wariantów algorytmu Dijkstry: z kopcem binarnym, z algorytmem Diala oraz z kolejką RadixHeap.

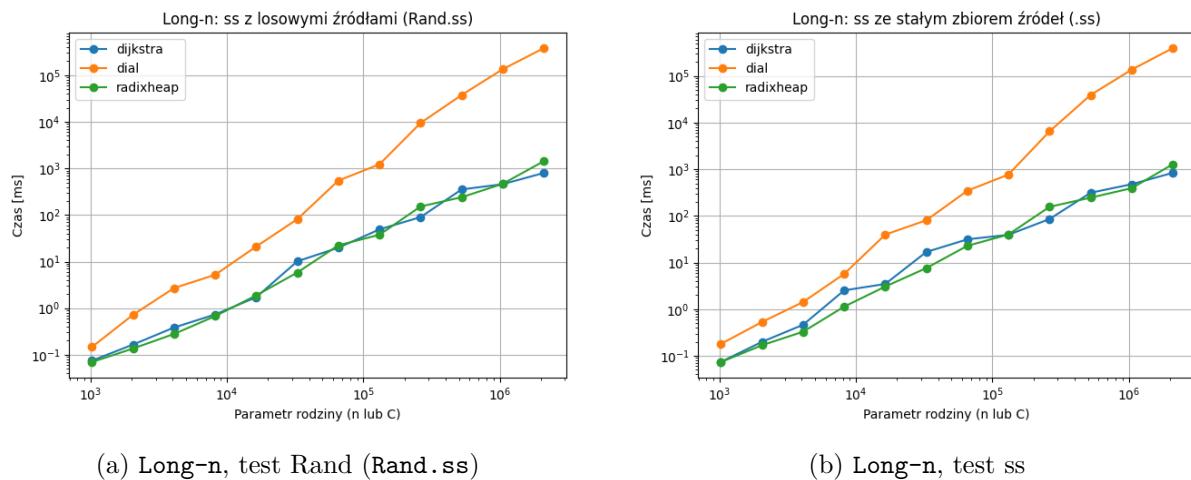
## 4 Wyniki, wnioski



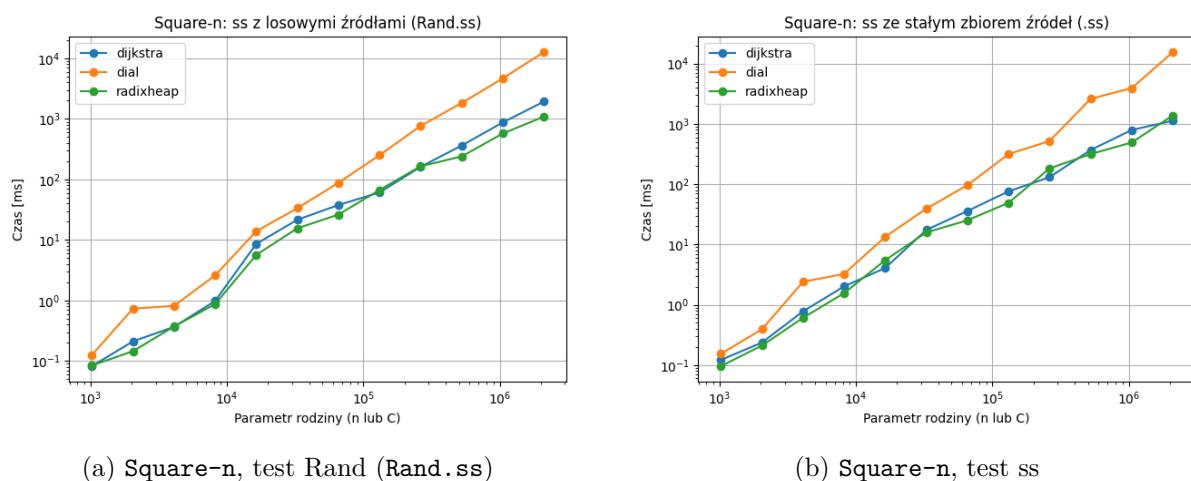
Rysunek 1: Czasy działania algorytmów dla rodziny Random4-n.



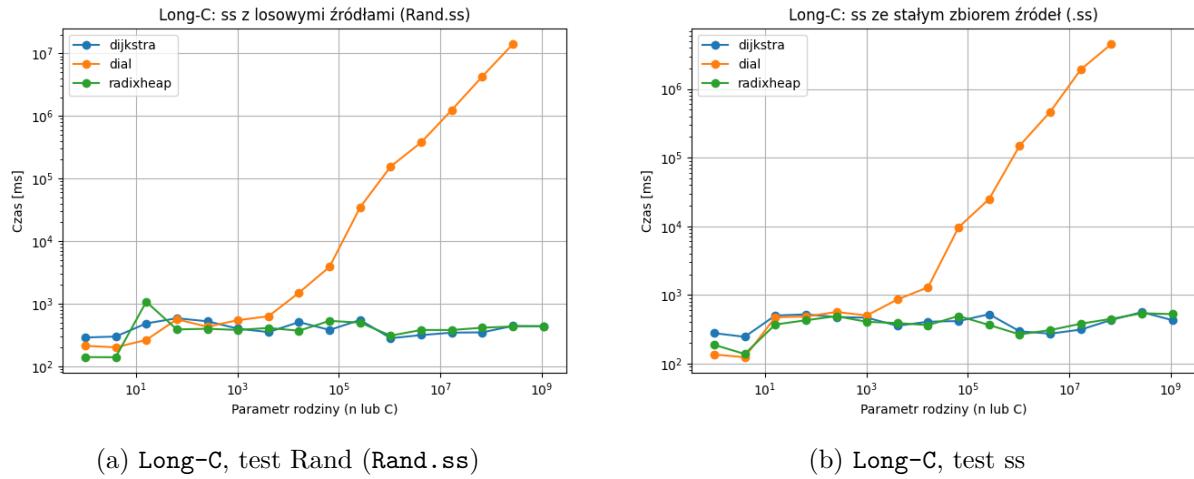
Rysunek 2: Czasy działania algorytmów dla rodziny Random4-C.



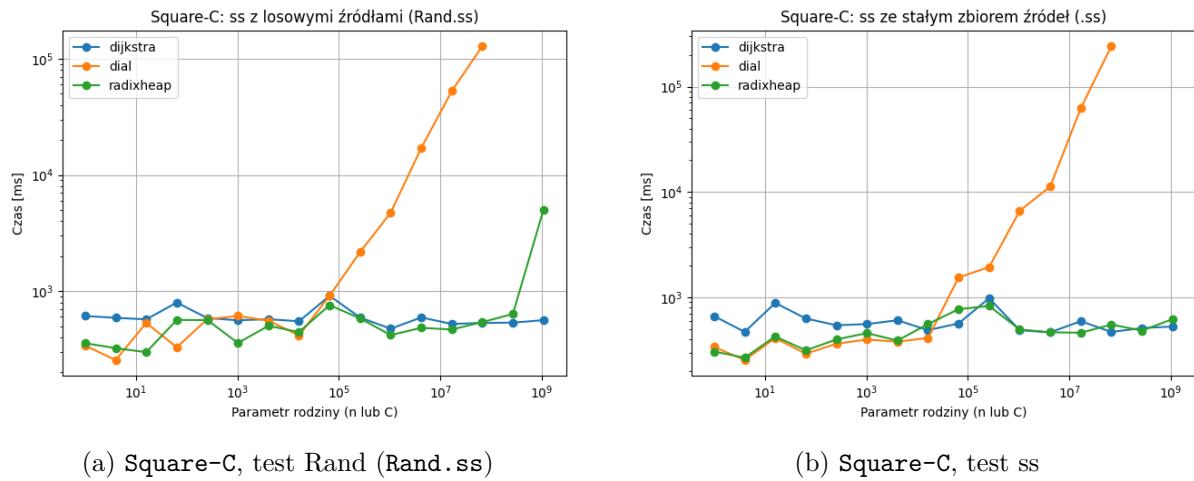
Rysunek 3: Czasy działania algorytmów dla rodziny Long-n.



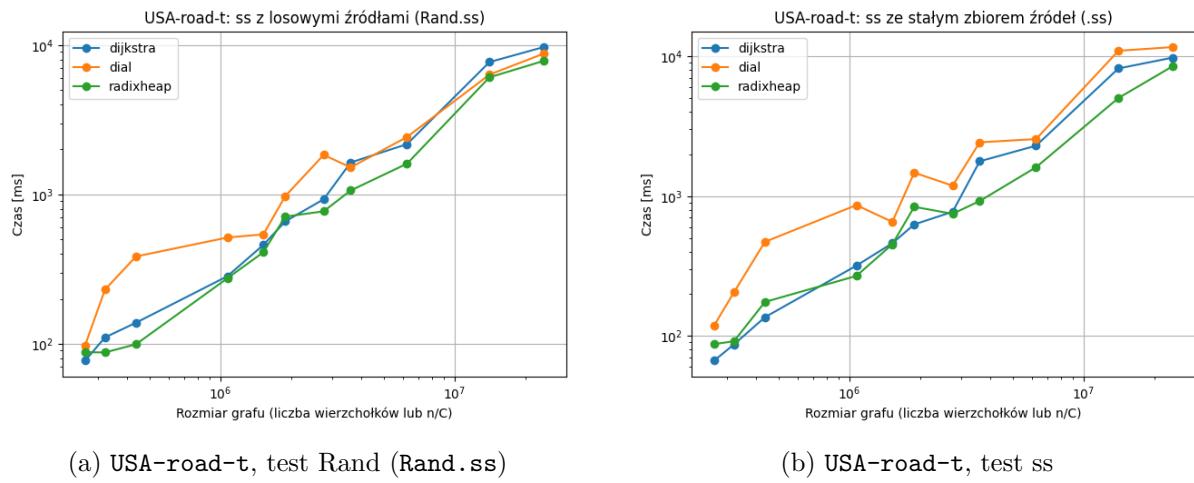
Rysunek 4: Czasy działania algorytmów dla rodziny Square-n.



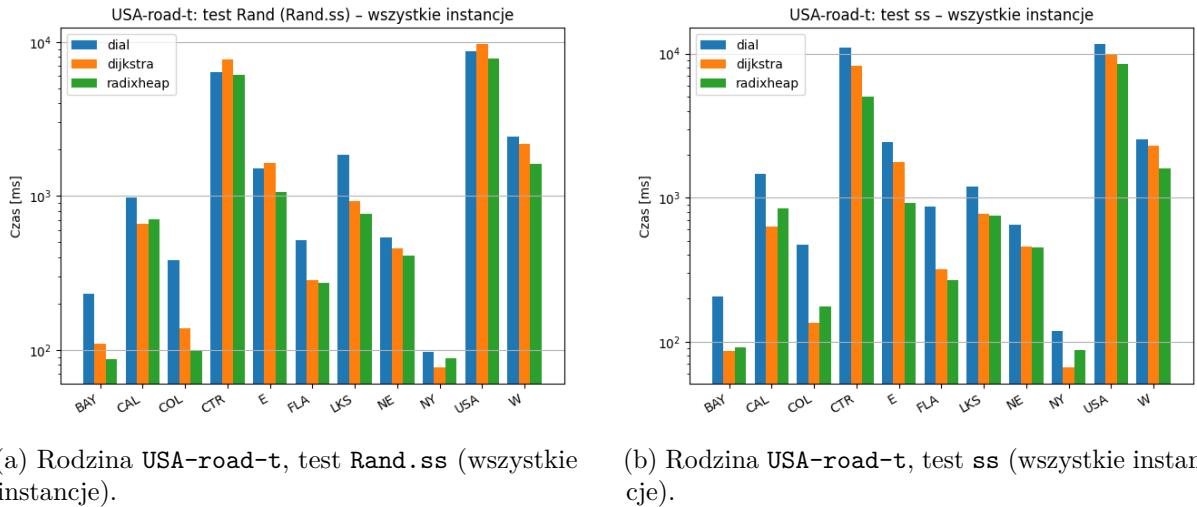
Rysunek 5: Czasy działania algorytmów dla rodziny Long-C.



Rysunek 6: Czasy działania algorytmów dla rodziny Square-C.



Rysunek 7: Czasy działania algorytmów dla rodziny USA-road-t.



Rysunek 8: Wykresy słupkowe porównujące czasy działania algorytmów dla wszystkich instancji rodziny USA-road-t.

## 5 Wnioski

Z wykresu widać zachowanie zgodne z przewidywaniami teoretycznymi:

- W rodzinach Random4-n i Random4-C wszystkie algorytmy zachowują się zgodnie z przewidywaniami. Dla Random4-n Dial jest szybki dla małych  $n$ , ale szybko staje się niepraktyczny ze względu na rosnący koszt  $O(nC)$ . RadixHeap oferuje lepszą skalowalność niż kopiec binarny, szczególnie dla większych  $n$ . W Random4-C Dial wykazuje liniowy wzrost czasu z  $C$ , podczas gdy RadixHeap pozostaje praktycznie niezmieniony.
- W rodzinach Long-n i Square-n różnice w strukturze drzewa najkrótszych ścieżek wpływają na wydajność. W Long-n Dijkstra z kopcem binarnym radzi sobie lepiej ze względu na mniejszy front, podczas gdy w Square-n większy front prowadzi do gorszych czasów. Dial ponownie staje się niepraktyczny dla dużych  $n$ , a RadixHeap oferuje stabilne czasy działania.
- W rodzinach Long-C i Square-C Dial wykazuje wyraźny wzrost czasu z  $C$ , co czyni go niepraktycznym dla dużych wartości. RadixHeap pozostaje stabilny, a Dijkstra z kopcem binarnym nie wykazuje znaczącej zmiany czasu z  $C$ .
- W rodzinie USA-road-t wszystkie algorytmy działają efektywnie ze względu na strukturę grafu drogowego. Dial jest konkurencyjny dla mniejszych instancji, ale dla większych staje się wolniejszy niż pozostałe algorytmy. RadixHeap oferuje najlepszą wydajność na największych instancjach, przewyższając kopiec binarny.

Natomiast tabelki z długościami najkrótszych ścieżek potwierdzają poprawność implementacji wszystkich algorytmów, gdyż wyniki są zgodne dla każdej pary wierzchołków we wszystkich testowanych rodzinach grafów.

Tabela 1: Długości najkrótszych ścieżek dla największych instancji rodzin Long-n, Square-n, Long-C oraz Square-C.

Rodzina	Para $(s, t)$	dijkstra	dial	radixheap
Long-n.21.0	(1, 2097152)	31336751771	31336751771	31336751771
	(1009993, 89642)	12136501400	12136501400	12136501400
	(932923, 1298742)	12611272258	12611272258	12611272258
	(811963, 913618)	45409673212	45409673212	45409673212
	(1003612, 453665)	43487757770	43487757770	43487757770
Square-n.21.0	(1, 2096704)	714640488	714640488	714640488
	(818431, 1576966)	478833912	478833912	478833912
	(1726360, 687497)	302704631	302704631	302704631
	(114272, 1210275)	271654165	271654165	271654165
	(904270, 1818305)	344164231	344164231	344164231
Long-C.9.0	(1, 1048576)	1584591029	1584591029	1584591029
	(326707, 371251)	1534446468	1534446468	1534446468
	(908055, 618433)	2350573131	2350573131	2350573131
	(351719, 291140)	4595631045	4595631045	4595631045
	(786139, 488347)	1876680437	1876680437	1876680437
Square-C.9.0	(1, 1048576)	49020430	49020430	49020430
	(771039, 330684)	32554885	32554885	32554885
	(585891, 589853)	79093637	79093637	79093637
	(979929, 582692)	77716688	77716688	77716688
	(35783, 6350)	61010637	61010637	61010637

Tabela 2: Długości najkrótszych ścieżek dla największych instancji rodzin Random4-n i Random4-C.

Rodzina	Para $(s, t)$	dijkstra	dial	radixheap
Random4-n.21.0	(1, 2097152)	9051281	9051281	9051281
	(1408389, 480125)	8694005	8694005	8694005
	(1025312, 1741784)	7393981	7393981	7393981
	(381995, 158098)	10205500	10205500	10205500
	(226050, 1125125)	6399149	6399149	6399149
Random4-C.9.0	(1, 1048576)	966465	966465	966465
	(378832, 605128)	1079939	1079939	1079939
	(587273, 140016)	1017314	1017314	1017314
	(796379, 514564)	1108414	1108414	1108414
	(99102, 374607)	1032348	1032348	1032348

Tabela 3: Długości najkrótszych ścieżek dla największej instancji rodziny USA-road-t (USA-road-t.USA).

Rodzina	Para $(s, t)$	dijkstra	dial	radixheap
USA-road-t.USA	(1, 23947347)	26227528	26227528	26227528
	(16113059, 19149306)	11528080	11528080	11528080
	(19211217, 11341550)	36852237	36852237	36852237
	(11651134, 6881299)	9791692	9791692	9791692
	(13228455, 14481649)	14620757	14620757	14620757