# Algorytmy Optymalizacji Dyskretnej

**Laboratorium**

**Lista 3**

Kinga Majcher
272354

Grudzień 2024

## Wstęp

Celem listy było zaimplementowanie trzech wariantów algorytmu Dijkstry dla problemu najkrótszych ścieżek z jednym źródłem w sieci $G = (N, A)$ o $n$ wierzchołkach i $m$ łukach z nieujemnymi kosztami:

- wariant podstawowy
- algorytm Diala (z $c + 1$ kubełkami)
- implementacja Radix Heap

Celem jest porównanie czasu wykonywania każdej implementacji na pewnych rodzinach grafów oraz implementacja uzyskanych wyników.

Dane testowe zostały pobrane ze strony 9th DIMACS Implementation Challenge: Shortest Paths. Implementacje wariantów algorytmu Dijkstry testowane są na 7 rodzinach grafów.

Implementacje algorytmów napisane są w języku C++.

Używane oznaczenia:

- $n$ - liczba wierzchołków w grafie
- $m$ - liczba krawędzi w grafie
- $C$ - maksymalny koszt krawędzi w grafie

Opis rodzin danych testowych:

- **Long-C** - rodzina grafów składających się z podłużnych kratownic. $n$ jest stałe, $C$ rośnie. $C = 4^i$, gdzie $i \in \{0, \ldots, 15\}$.
- **Long-n** - rodzina grafów składających się z podłużnych kratownic. $n$ rośnie, $C = n$. $n = 2^i$, gdzie $i \in \{10, \ldots, 21\}$.

- **Random4-C** - rodzina losowych grafów, w której $m = 4n$, $n$ jest stałe, a $C$ rośnie. $n = 2^{20}$, $C = 4^i$, gdzie $i \in \{0, \ldots, 15\}$.
- **Random4-n** - rodzina losowych grafów, w której $m = 4n$, $n$ rośnie, a $C = n$. $n = 2^i$, gdzie $i \in \{10, \ldots, 21\}$.
- **Square-C** - rodzina grafów składających się z kwadratowych kratownic. $n$ jest stałe, $C$ rośnie. $C = 4^i$, gdzie $i \in \{0, \ldots, 15\}$.
- **Square-n** - rodzina grafów składających się z kwadratowych kratownic. $n$ rośnie, $C = n$. $n = 2^i$, gdzie $i \in \{10, \ldots, 21\}$.
- **USA-road-t** - rodzina grafów reprezentująca rzeczywiste dane dotyczące dróg w USA, w której koszty krawędzi to czasy podróży pomiędzy wierzchołkami reprezentującymi konkretne miejsca.

# 1 Wariant podstawowy algorytmu Dijkstry

## 1.1 Opis algorytmu

Algorytm Dijkstry jest jedną z typowych metod wyznaczania najkrótszej ścieżki w grafach zarówno skierowanych jak i nieskierowanych z nieujemnymi wagami krawędzi. Algorytm wyznacza, zaczynając od wierzchołka startowego $s$ najkrótsze ścieżki do każdego z pozostałych osiągalnych wierzchołków, badź do jednego wskazanego wierzchołka $t$.

Algorytm działa iteracyjnie, rozszerzając zbiór wierzchołków, dla których odległość od źródła została już określona. W każdej iteracji wybierany jest wierzchołek o najmniejszej odległości od źródła (spośród jeszcze nieprzetworzonych), a następnie jego sąsiedzi są aktualizowani.

## 1.2 Działanie algorytmu

1. Inicjalizacja ($O(n)$):
   - Odległość do wierzchołka startowego $s$ ustawiamy na 0 ($d(s) = 0$).
   - Odległość do pozostałych wierzchołków ustawiamy na nieskończoność ($d(i) = \infty$).
2. Dodajemy źródło do kolejki priorytetowej ($O(1)$).
3. Dopóki kolejka nie jest pusta:
   - Wybieramy wierzchołek z najmniejszą aktualnie odległością od źródła ($O(n\log n)$).
   - Dla każdego sąsiada wybranego wierzchołka $i$ sprawdzamy, czy $d(i) + c_{ij} < d(j)$. Jeśli tak, to aktualizujemy tę wartość i dodajemy wierzchołek do kolejki priorytetowej ($O(m\log n)$)

### 1.3 Złożoność algorytmu

Wydajność algorytmu zależy od struktury danych używanej do implementacji kolejki priorytetowej. W języku C++ do implementacji domyślnej kolejki priorytetowej wykorzystuje się kopiec binarny, co pozwala na efektywne zarządzanie wierzchołkami. Operacje takie jak dodanie elementu czy aktualizacja priorytetu mają złożoność $O(\log n)$, a wyciągnięcie elementu o najmniejszym priorytecie $O(1)$.
Dany wierzchołek jest dodawany i wyjmowany z kolejki jednokrotnie. Odległości mogą być aktualizowane maksymalnie $m$ razy.

Uwzględniając te operacje, całkowita złożoność czasowa algorytmu wynosi:

$$O(n \cdot O(1) + (m+n) \cdot O(\log n)) =$$

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

Jak można zauważyć, złożoność wariantu podstawowego algorytmu Dijkstry zależy tylko od liczby wierzchołków i krawędzi w grafie.

## 2 Algorytm Diala

### 2.1 Opis algorytmu

Algorytm Diala jest usprawnioną wersją algorytmu Dijkstry, która zamiast klasycznej kolejki priorytetowej wykorzystuje strukturę kubełków o rozmiarze $C+1$, gdzie $C$ oznacza maksymalny koszt pojedynczej krawędzi. Wierzchołki o takich samych odległościach $d(i)$ są grupowane w tym samym kubełku. Dzięki temu proces wyboru wierzchołka o najmniejszej wartości $d(i)$ jest uproszczony, gdyż nie musimy już używać kolejki priorytetowej tylko sięgamy do odpowiedniego kubełka.

Odległości $d(i)$ wierzchołków są aktualizowane poprzez przenoszenie ich pomiędzy kubełkami. Informacje o zawartości kubełków przechowywane są w tablicy.

### 2.2 Działanie algorytmu

1. Inicjalizacja ($O(n)$):
   - Tworzymy $C+1$ kubełków.
   - Podobnie jak w algorytmie Dijkstry ustawiamy wartości $d(i)$
2. Dodajemy źródło do kubełka 0 ($O(1)$).
3. Dopóki nie wszystkie kubełki są puste:
   - Znajdujemy pierwszy niepusty kubełek ($O(C)$).
   - Przetwarzamy wierzchołki w tym kubełku, aktualizując odległości ich sąsiadów, podobnie jak w zwykłym algorytmie Dijkstry. Jeśli odległość sąsiada zostanie zaktualizowana, jest on przenoszony do odpowiedniego kubełka.

W podstawowej wersji potrzebne jest aż $nC$ kubełków. Można tę liczbę jednak zmniejszyć, zauważając, że odległość między dwoma wierzchołkami nie może być większa niż $C$ a tablice z kubełkami możemy traktować jako listę cykliczną.

### 2.3 Złożoność algorytmu

Algorytm unika kosztownych operacji na kolejce priorytetowej, jednak wymaga przeglądania kubełków, przez co złożoność jest pseudowielomianowa. Zakładając, że operacje przenoszenia elementów między kubełkami są wykonywane w czasie stałym, całkowita złożoność wynosi:

$$O(m + n \cdot C)$$

Jak można zauważyć, złożoność algorytmu zależy już nie tylko od wielkości grafu ale także od wielkości kosztów krawędzi. Będzie on więc lepiej działał dla grafów o dość małej rozpiętości tych kosztów

## 3 Algorytm Radix Heap

### 3.1 Opis algorytmu

Radix Heap jest kompromisem między algorytmem Diala, a klasycznym algorytmem Dijkstry. Jego kluczowym założeniem jest użycie kubełków o zróżnicowanych zakresach wartości etykiet $d(i)$, co zmniejsza liczbę wymaganych kubełków w porównaniu do algorytmu Diala. W Radix Heap kubełki są dobierane w taki sposób, by zmaksymalizować efektywność poprzez grupowanie elementów na podstawie najbardziej znaczącego bitu ich odległości. Pierwsze dwa kubełki mają szerokość 1. Kolejne kubełki mają szerokości $2, 4, 8, \ldots, 2^{k-1}$, gdzie $k = \lceil \log_2(nC) \rceil$.

### 3.2 Działanie algorytmu

Algorytm działa na zasadzie bardzo podobnej do algorytmu Diala. Kubełki są opróżniane względem rosnącej wartości $d(i)$. Aby ograniczyć konieczność wyszukiwania wartości w kubełkach o większej szerokości to gdy kubełek o szerokości większej niż 1 jest opróżniany, jego zawartość jest rozdzielana na wcześniejsze kubełki o mniejszych wartościach.

Procedura aktualizacji odległości działa tak samo jak w algorytmie Diala.

W celu możliwości wybierania najmniejszej wartości $d(i)$ z kubełka, stosowana jest w nich kolejka priorytetowa.

### 3.3 Złożoność algorytmu

Złożoność algorytmu jest zależna od liczby kubełków podobnie jak w algorytmie Diala. Ogólną złożoność można więc wyrazić w postaci:

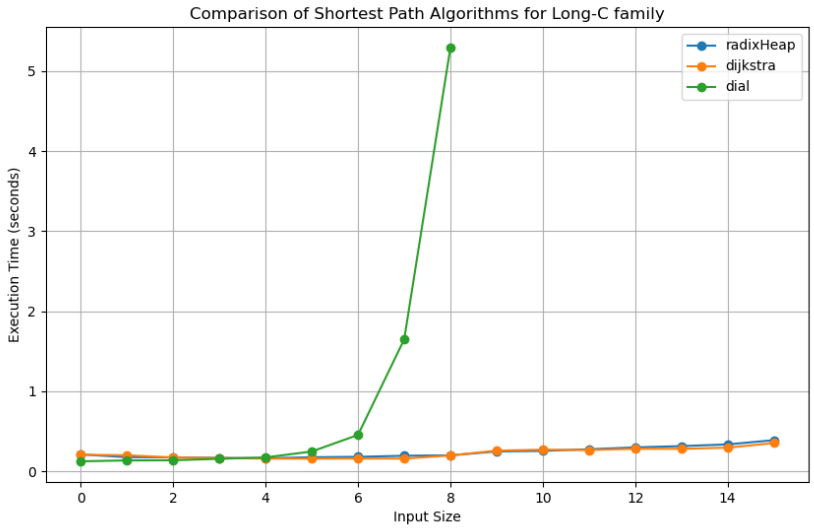
$$O(m + nK), \text{ gdzie } K \text{ to liczba kubełków}$$

Złożoność algorytmu wynosi więc:

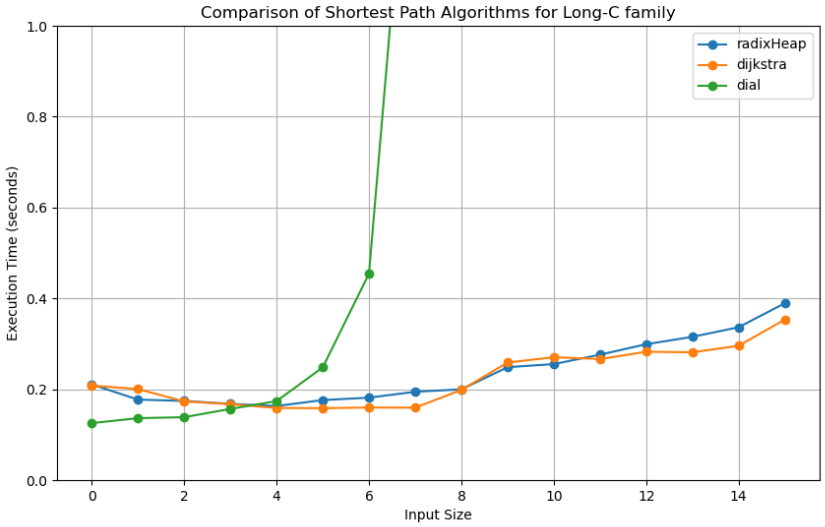
$$O(m + n\log(nC))$$

# 4 Wyniki wyznaczania średniego czasu wyznaczania najkrótszych ścieżek ze źródła do wszystkich wierzchołków

## 4.1 Rodzina Long-C




Rysunek 1: Wyniki dla rodziny Long-C




Rysunek 2: Wyniki dla rodziny Long-C z obcięciem na osi Y

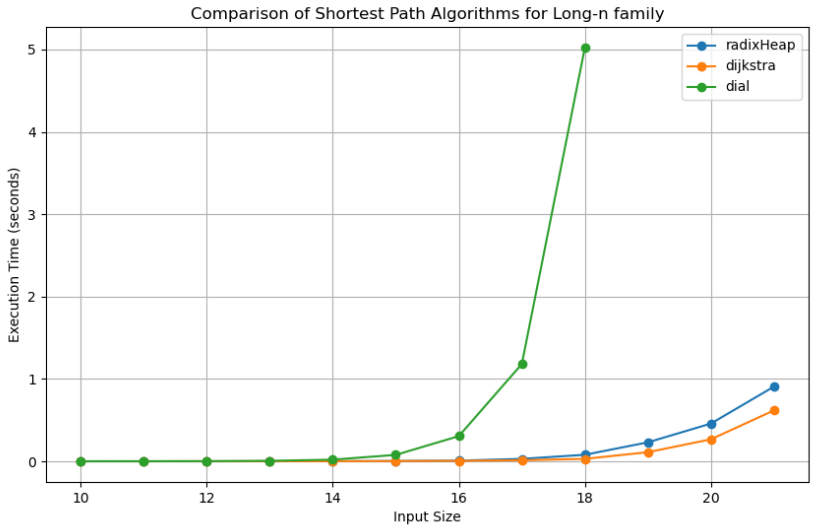
### 4.1.1 Wnioski i obserwacje

- Dla dużych wartości $i$ algorytm Diala działa wyraźnie gorzej niż pozostałe algorytmy.
- Dla małych wartości $i$ algorytmy Diala radzi sobie najlepiej, jednak jest to nieznacznie lepsza złożoność czasowa niż pozostałych algorytmów.
- Algorytmy Dijkstry i Radix Heap dla wszystkich wartości $i$ mają podobną złożoność czasową, aczkolwiek dla Dijkstry jest ona minimalnie lepsza.

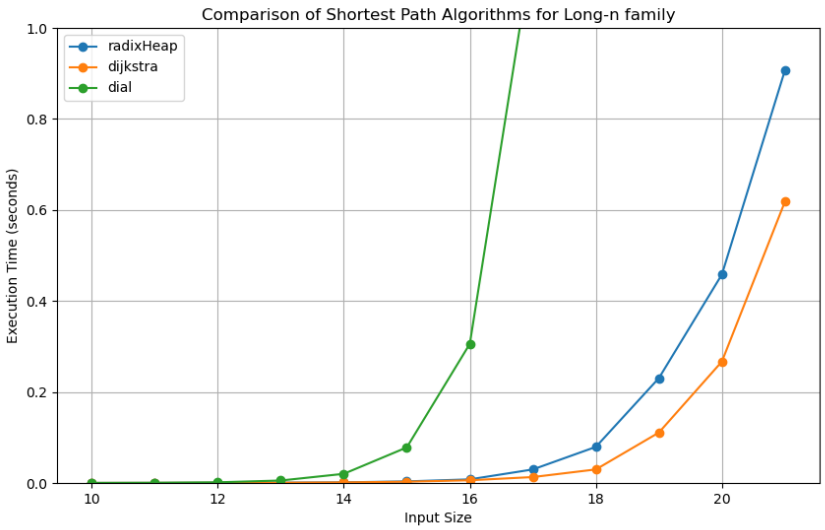
Złożoność algorytmu Dijkstry nie zależy od wartości $C$, a złożoność RadixHeap zależy od $C$ w małym stopniu stąd zachowują się one dla tej rodziny grafów zdecydowanie lepiej niż dla algorytmu Diala, dla którego ta złożoność jest bezpośrednio zależna od $C$.

Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Long-C to najlepiej wybrać algorytm Dijksty lub Radix Heap.

## 4.2 Rodzina Long-n




Rysunek 3: Wyniki dla rodziny Long-n




Rysunek 4: Wyniki dla rodziny Long-n z obcięciem na osi Y

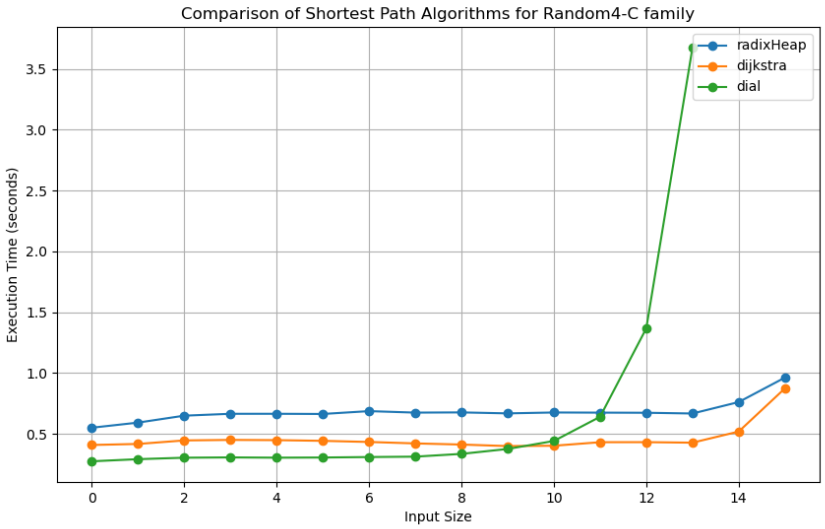
### 4.2.1 Wnioski i obserwacje

- Algorytmy Dijkstry i Radix Heap mają bardzo podobne czasy wykonywania dla wszystkich testowanych danych.
- Algorytm Diala jest gorszy od dwóch pozostałych dla dużych wartości $i$.

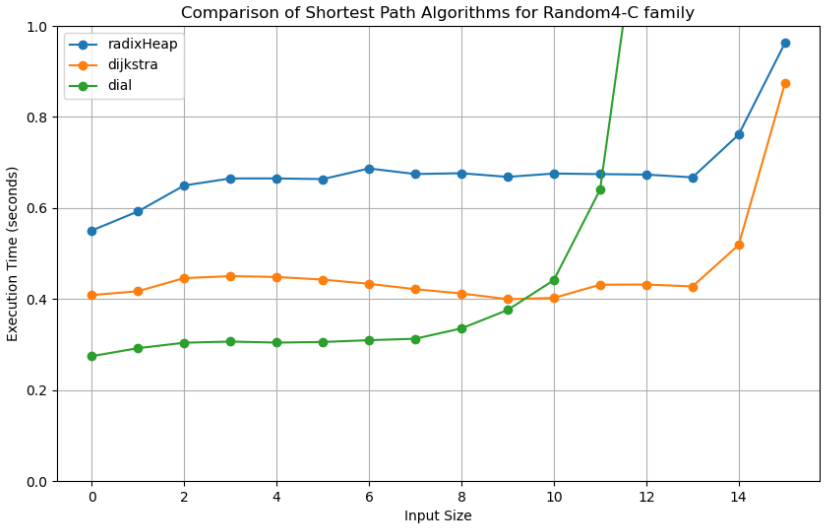
Algorytm Diala jest dla rodziny Long-n najgorszy ponieważ przez jej specyfikę ($C = n$) osiąga bezpośrednio złożoność kwadratową. Na złożoność algorytmu Dijkstry ma wpływ tylko rosnące $n$, dlatego też dla tej rodziny grafów jest on najlepszy.

Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Long-n to najlepiej wybrać algorytm Dijksty lub Radix Heap.

## 4.3 Rodzina Random4-C




Rysunek 5: Wyniki dla rodziny Random4-C




Rysunek 6: Wyniki dla rodziny Random4-C z obcięciem na osi Y

### 4.3.1 Wnioski i obserwacje

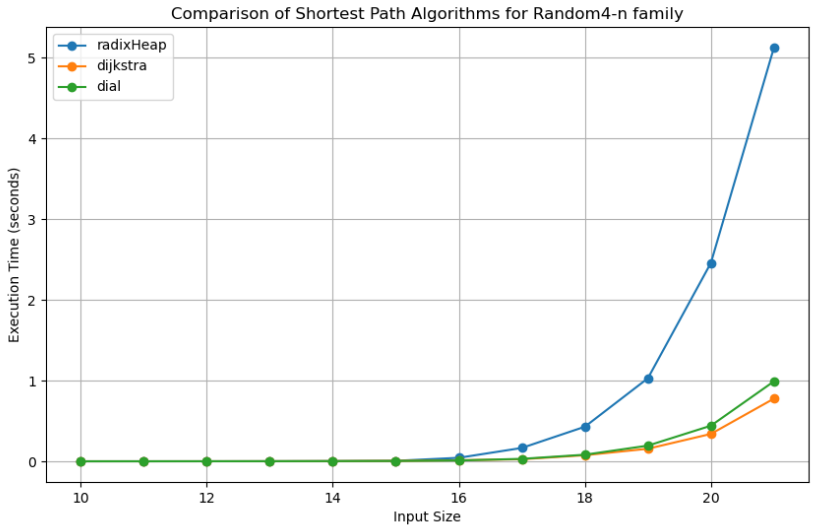
- Dla dużych wartości $i$ algorytm Diala działa wyraźnie gorzej niż pozostałe algorytmy.

- Dla małych wartości $i$ wszystkie algorytmy osiągają bardzo podobną złożoność czasową, jednak najlepszy jest algorytm Diala.
- Algorytmy Dijkstry i Radix Heap dla wszystkich wartości $i$ mają podobną złożoność czasową, aczkolwiek dla Dijkstry jest ona na ogół minimalnie lepsza.

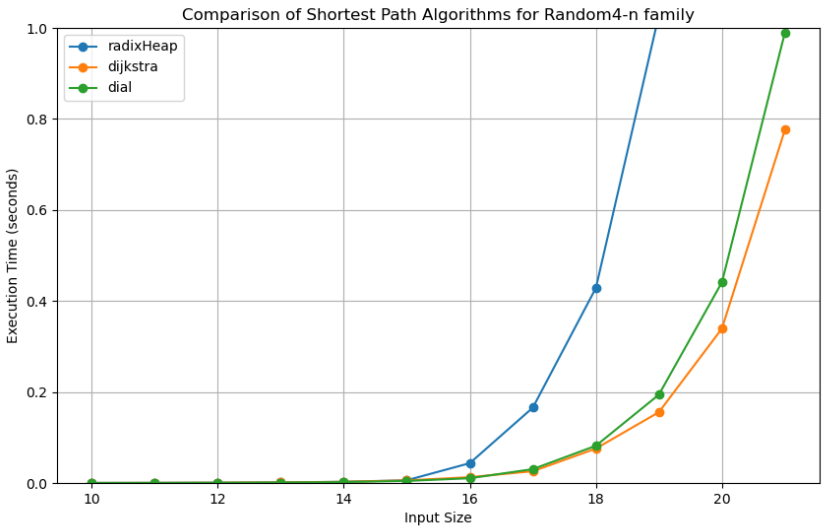
Złożoność algorytmu Diala jest bezpośrednio związana z wartością $C$, a więc logicznym jest duży wzrost złożoności czasowej dla dużych wartości $i$. Złożoność algorytmu Dijkstry nie zależy od wartości $C$, a złożoność RadixHeap zależy od $C$ w małym stopniu stąd zachowują się one dla tej rodziny grafów zdecydowanie lepiej niż dla algorytmu Diala.

Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Random4-C dla dowolnych jego wielkości to najlepiej wybrać algorytm Dijksty lub Radix Heap, gdyż ich złożoność jest względnie stała.

## 4.4 Rodzina Random4-n




Rysunek 7: Wyniki dla rodziny Random4-n




Rysunek 8: Wyniki dla rodziny Random4-n z obcięciem na osi Y

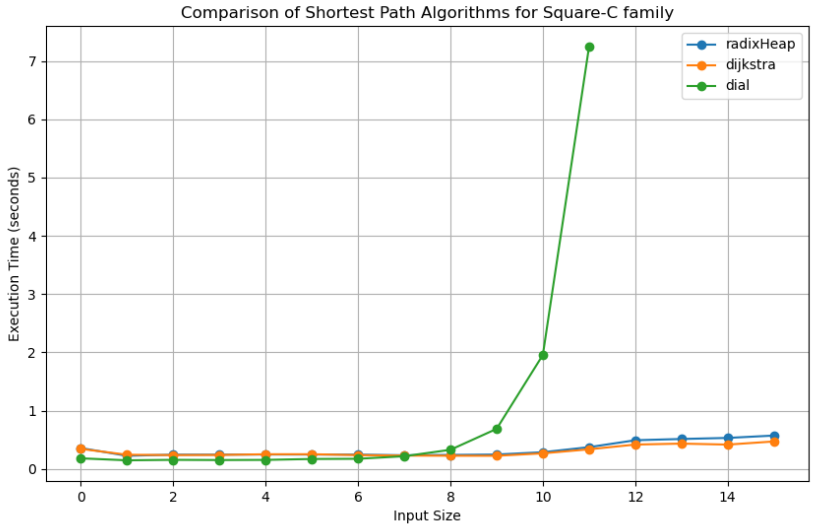
### 4.4.1 Wnioski i obserwacje

- Algorytmy Dijkstry i Diala mają zbliżone czasy wykonywania dla wszystkich testowanych danych.
- Algorytm Radix Heap jest gorszy od dwóch pozostałych.

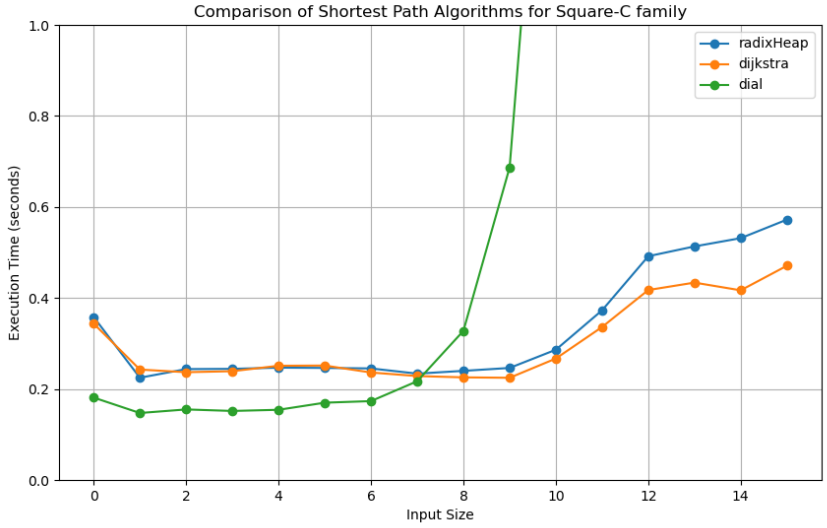
Algorytm Diala jest dla rodziny Random4-n najgorszy ponieważ przez jej specyfikę, przez którą ilość wykonywanych działań na kubełkach przewyższyła teoretycznie dobrą złożoność teoretyczną.

Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Random4-n to najlepiej wybrać algorytm Dijksty lub Diala. .

## 4.5 Rodzina Square-C




Rysunek 9: Wyniki dla rodziny Square-C




Rysunek 10: Wyniki dla rodziny Square-C z obcięciem na osi Y

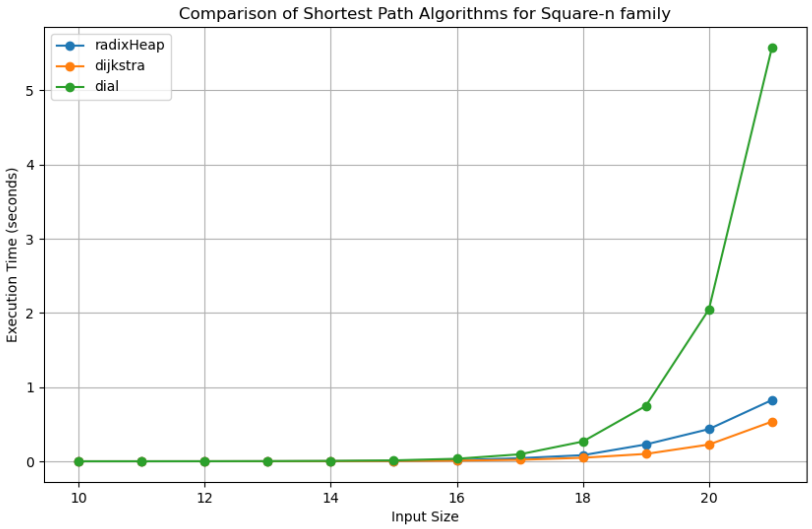
### 4.5.1 Wnioski i obserwacje

- Dla dużych wartości $i$ algorytm Diala działa wyraźnie gorzej niż pozostałe algorytmy.

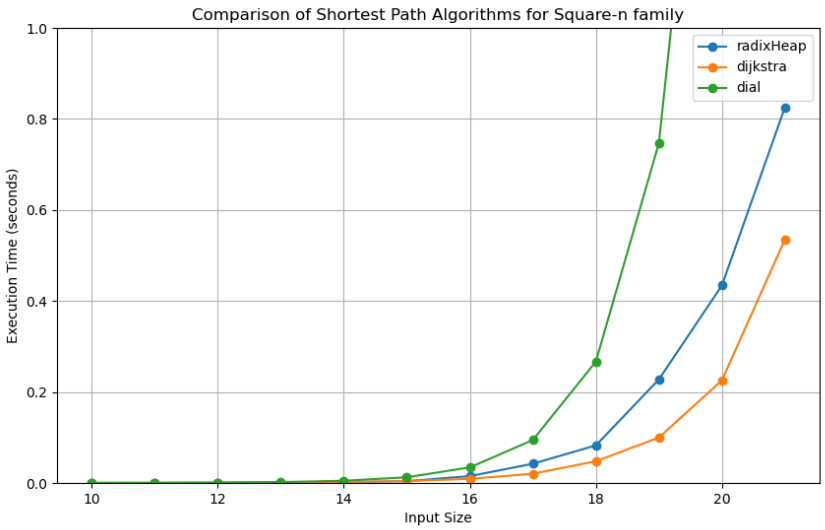
- Dla małych wartości $i$ wszystkie algorytmy osiągają bardzo podobną złożoność czasową, jednakże algortym Diala jest nieznacznie lepszy.
- Algorytmy Dijkstry i Radix Heap dla wszystkich wartości $i$ mają podobną złożoność czasową, aczkolwiek dla Dijkstry jest ona minimalnie lepsza dla dużych wartości $i$, a dla Radix Heap jest lepsza dla małych wartości $i$.

Złożoność algorytmu Diala jest bezpośrednio związana z wartością $C$, a więc logicznym jest duży wzrost złożoności czasowej dla dużych wartości $i$.
Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Square-C to najlepiej wybrać algorytm Dijksty lub Radix Heap.

## 4.6 Rodzina Square-n




Rysunek 11: Wyniki dla rodziny Square-n



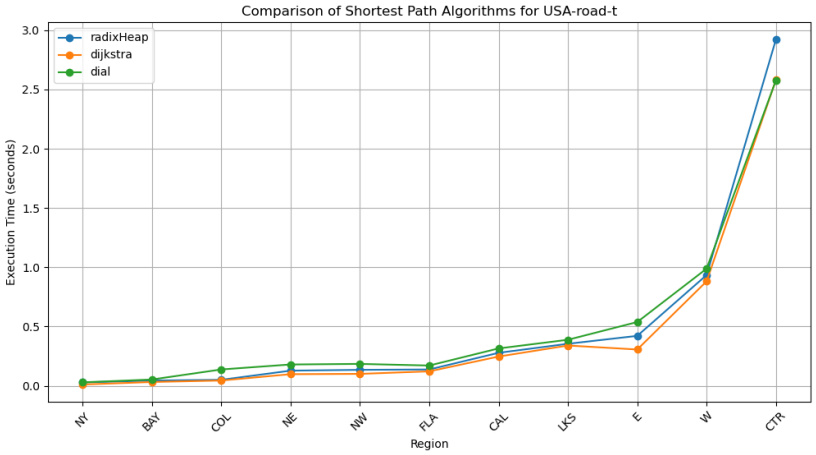

Rysunek 12: Wyniki dla rodziny Square-n z obcięciem na osi Y

### 4.6.1 Wnioski i obserwacje

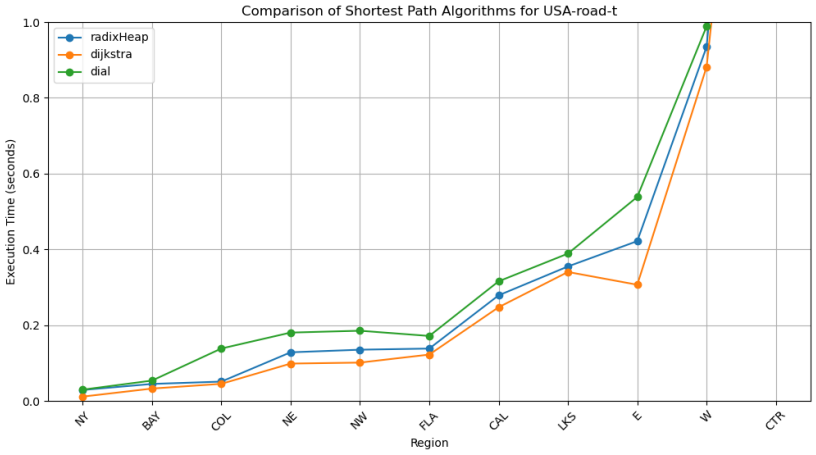
- Algorytmy Dijkstry i Radix Heap mają zbliżone czasy wykonywania dla wszystkich testowanych danych.
- Algorytm Diala jest gorszy od dwóch pozostałych.

Algorytm Diala jest dla rodziny Square-n najgorszy ponieważ przez jej specyfikę ($C = n$) osiąga bezpośrednio złożoność kwadratową.
Jeśli zależy nam na szybkim i poprawnym działaniu algorytmu dla grafu z rodziny Square-n to najlepiej wybrać algorytm Dijksty lub Radix Heap.

## 4.7 Rodzina USA-road-t




Rysunek 13: Wyniki dla rodziny USA-road-t




Rysunek 14: Wyniki dla rodziny USA-road-t z obcięciem na osi Y

### 4.7.1 Wnioski i obserwacje

- Wszystkie testowane algorytmy mają podobne czasy wykonywania dla testowanych danych.

Dla realistycznych danych wszystkie algorytmy osiągnęły stosunkowo podobne złożoności czasowe. Widać, że kształty osiągane na wykresie są zgodne dla wszystkich algorytmów, więc można stwierdzić, że radzą sobie one dla takich danych stosunkowo podobnie.

# 5 Wyniki wyznaczania długości najkrótszych ścieżek między parami wierzchołków

$n$ to liczba wierzchołków, a więc i jednocześnie największy numer indeksu wierzchołka

## 5.1 Rodzina Long-C

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 24155783 | 24155783 | 24155783 |
| $(349, 725)$ | 30799980 | 30799980 | 30799980 |
| $(206, 882)$ | 52041638 | 52041638 | 52041638 |
| $(453, 808)$ | 43761446 | 43761446 | 43761446 |
| $(1, n)$ | 14401679 | 14401679 | 14401679 |

Tabela 1: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Long-C

## 5.2 Rodzina Long-n

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 5382433 | 5382433 | 5382433 |
| $(349, 725)$ | 45840661 | 45840661 | 45840661 |
| $(206, 882)$ | 19147090 | 19147090 | 19147090 |
| $(453, 808)$ | 61693216 | 61693216 | 61693216 |
| $(1, n)$ | 23589324 | 23589324 | 23589324 |

Tabela 2: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Long-n

## 5.3 Rodzina Random4-C

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 19915 | 19915 | 19915 |
| $(349, 725)$ | 17585 | 17585 | 17585 |
| $(206, 882)$ | 16210 | 16210 | 16210 |
| $(453, 808)$ | 15239 | 15239 | 15239 |
| $(1, n)$ | 16356 | 16356 | 16356 |

Tabela 3: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Random4-C

### 5.4 Rodzina Random4-n

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 180094 | 180094 | 180094 |
| $(349, 725)$ | 188251 | 188251 | 188251 |
| $(206, 882)$ | 145430 | 145430 | 145430 |
| $(453, 808)$ | 259045 | 259045 | 259045 |
| $(1, n)$ | 152665 | 152665 | 152665 |

Tabela 4: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Random4-n

### 5.5 Rodzina Square-C

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 479507 | 479507 | 479507 |
| $(349, 725)$ | 562463 | 562463 | 562463 |
| $(206, 882)$ | 1082605 | 1082605 | 1082605 |
| $(453, 808)$ | 655808 | 655808 | 655808 |
| $(1, n)$ | 236461 | 236461 | 236461 |

Tabela 5: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Square-C

### 5.6 Rodzina Square-n

| para | Dijkstra | Dial | RadixHeap |
|---|---|---|---|
| $(16, 512)$ | 1762849 | 1762849 | 1762849 |
| $(349, 725)$ | 5704911 | 5704911 | 5704911 |
| $(206, 882)$ | 1323205 | 1323205 | 1323205 |
| $(453, 808)$ | 3781370 | 3781370 | 3781370 |
| $(1, n)$ | 3596557 | 3596557 | 3596557 |

Tabela 6: Wartości długości najkrótszych ścieżek między parami wierzchołków dla rodziny Square-n

### 5.7 Wnioski

Wszystkie testowane algorytmy dla każdej z rodzin grafów zwróciły takie same wartości. Można więc stwierdzić, że algorytmy działają poprawnie.

## 6 Ogólne wnioski

Nie można jednoznacznie określić, że któryś z algorytmów jest wyraźnie lepszy od pozostałych. To jak algorytmy będą się zachowywać na testowanych danych zależy w ogromnym stopniu od tego, jaka jest struktura grafu. Nawet algorytm Diala, który w zdecydowanej większości testów zachowywał się dość kiepsko, dla grafów o odpowiedniej strukturze jest najszybszy.

Dobór odpowiedniego algorytmu nie jest więc trywialną kwestią i wymaga wiedzy na temat struktury danych. Jeśli jednak nie wiemy o niej zbyt wiele najlepszą opcją będzie celowanie w algorytm Dijkstry. Zachowywał się on bowiem bardzo dobrze dla większości rodzin grafów, a jego złożoność zależy tylko od samych rozmiarów grafu, a nie od dodatkowych czynników jak koszty krawędzi. Można więc powiedzieć, że jest on stosunkowo bezpiecznym wyborem.