Sprawozdanie Lista 3

Paweł Krzyszczak (nr 272379)

Grudzień 2024

1 Wstęp

W ramach zadania zaimplementowano trzy warianty algorytmu Dijkstry dla problemu najkrótszych ścieżek z jednym źródłem w sieci $G = (N, A)$ o n wierzchołkach ($|N| = n$) i m łukach ($|A| = m$) z nieujemnymi kosztami (algorytmy wyznaczają najkrótsze ścieżki między zadanym źródłem $s \in N$ i wszystkimi wierzchołkami $i \in (N/\ s)$:

- wariant podstawowy (wykorzystujący np. kopiec binarny lub inną wydajną implementację kolejki priorytetowej),
- algorytm Diala (z $C + 1$ kubełkami),
- implementację Radix Heap.

Celem było porównanie ich efektywności obliczeniowej i czasowej na wybranych rodzinach grafów oraz interpretacja uzyskanych wyników.

Wszystkie implementacje zostały napisane w języku C++.

2 Wariant podstawowy (z kolejką priorytetową)

2.1 Cel i zastosowanie

Algorytm Dijkstry został zaimplementowany w celu wyznaczenia najkrótszych ścieżek w grafach skierowanych lub nieskierowanych o nieujemnych wagach krawędzi. Jego zastosowania obejmują m.in. znajdowanie tras w sieciach komunikacyjnych, analizę topologii grafów oraz rozwiązywanie problemów logistycznych.

2.2 Opis implementacji

Zaimplementowana wersja algorytmu wykorzystuje kolejkę priorytetową w postaci kopca minimalnego (`std::priority_queue` z odwróconym porządkiem), co zapewnia wydajność $O((V + E) \cdot \log V)$, gdzie V to liczba wierzchołków, a E liczba krawędzi. Kolejka priorytetowa umożliwia szybkie wybieranie wierzchołka o najmniejszym dotychczasowym dystansie.

Algorytm inicjalizuje:

- **Tablicę odległości** (`distances`), w której każda wartość jest początkowo ustawiona na $+\infty$, z wyjątkiem wierzchołka źródłowego, którego odległość wynosi 0.
- **Tablicę poprzedników** (`predecessors`), która pozwala odtworzyć najkrótszą ścieżkę do danego wierzchołka.

2.3 Szczegóły działania

1. Wierzchołek źródłowy jest dodawany do kolejki z odległością równą 0.
2. W każdej iteracji pętli algorytm pobiera wierzchołek o najmniejszym dystansie z kolejki.

3. Jeśli aktualna odległość jest większa niż zapisana w tablicy, wierzchołek jest pomijany (optymalizacja w celu uniknięcia przetwarzania przestarzałych stanów).

4. Dla każdego sąsiada wierzchołka wykonywana jest operacja relaksacji:

   - Obliczana jest nowa odległość jako suma odległości do bieżącego wierzchołka i wagi krawędzi.
   - Jeśli nowa odległość jest mniejsza niż dotychczasowa, odległość i poprzednik są aktualizowane, a sąsiad jest dodawany do kolejki priorytetowej.

2.4 Wnioski

Implementacja algorytmu Dijkstry z użyciem kolejki priorytetowej zapewnia wydajność oraz elastyczność w analizie grafów o różnych rozmiarach. Optymalizacja za pomocą kolejki priorytetowej skutecznie minimalizuje czas przetwarzania, co zostało potwierdzone podczas testów na dużych grafach.

3 Algorytm Diala

3.1 Cel i zastosowanie

Algorytm Diala służy do wyznaczania najkrótszych ścieżek w grafach skierowanych lub nieskierowanych, w których wagi krawędzi są liczbami całkowitymi dodatnimi i mają ograniczoną wartość maksymalną. Jest on szczególnie efektywny w przypadku grafów o małych i ograniczonych wagach krawędzi, ponieważ jego złożoność czasowa wynosi $O(V + E + C)$, gdzie V to liczba wierzchołków, E liczba krawędzi, a C to największa waga krawędzi.

3.2 Opis implementacji

Implementacja algorytmu Diala opiera się na wykorzystaniu struktury zwanej **bucket list** (listą kubełkową), która jest tablicą list. Każdy kubełek odpowiada odległości modulo $C + 1$, gdzie C jest maksymalną wagą krawędzi. Kubełki te pozwalają efektywnie grupować wierzchołki według ich obecnych dystansów, co eliminuje potrzebę stosowania kolejki priorytetowej.

Algorytm inicjalizuje:

- **Tablicę odległości** (`distances`), w której każda wartość jest początkowo ustawiona na $+\infty$, z wyjątkiem wierzchołka źródłowego, którego odległość wynosi 0.
- **Tablicę poprzedników** (`predecessors`), która umożliwia odtworzenie najkrótszej ścieżki.
- **Listę kubełkową** o rozmiarze $C + 1$, gdzie każdy kubełek przechowuje listę wierzchołków o danym dystansie modulo $C + 1$.

3.3 Szczegóły działania

1. Wierzchołek źródłowy jest dodawany do kubełka odpowiadającego odległości 0.

2. Algorytm iteracyjnie przetwarza kubełki, zaczynając od najmniejszego indeksu:

   (a) Jeśli bieżący kubełek jest pusty, indeks jest zwiększany o 1 (modulo $C + 1$).

   (b) Wierzchołki z bieżącego kubełka są przetwarzane kolejno, a ich sąsiedzi poddawani operacji relaksacji:

      - Obliczana jest nowa odległość jako suma bieżącej odległości i wagi krawędzi.
      - Jeśli nowa odległość jest mniejsza niż dotychczasowa, wierzchołek jest przenoszony do kubełka odpowiadającego nowej odległości modulo $C + 1$.

3. Algorytm kończy działanie, gdy wszystkie kubełki zostaną przetworzone i pozostaną puste przez $C + 1$ iteracji.

3.4 Wnioski

Algorytm Diala, dzięki wykorzystaniu list kubełkowych, zapewnia dużą efektywność w przypadku grafów z ograniczonymi wagami krawędzi. Jest szczególnie przydatny w sytuacjach, gdy maksymalna waga krawędzi jest znacząco mniejsza od liczby wierzchołków w grafie.

4 Algorytm Radix Heap

4.1 Cel i zastosowanie

Algorytm Radix Heap jest zaawansowaną wersją algorytmu Dijkstry, który używa specjalnej struktury danych zwanej *radix heap* (stosem radixowym). Jest to struktura oparta na zestawie kubełków, gdzie każdy kubełek zawiera elementy o danej wartości. Celem algorytmu jest efektywne obliczenie najkrótszych ścieżek w grafach, w których wagi krawędzi są liczbami całkowitymi. Radix Heap jest szczególnie efektywny w przypadku grafów, w których wagi krawędzi są stosunkowo małe i w których użycie tradycyjnej kolejki priorytetowej byłoby kosztowne pod względem czasowym.

4.2 Opis implementacji

Algorytm Radix Heap operuje na specjalnych kubełkach, które przechowują pary (odległość, wierzchołek). Kubełki są przypisane do zakresów wartości, a algorytm przetwarza je w sposób podobny do metody *bucket sort*, wykorzystując odpowiednią szerokość kubełków. Działanie algorytmu jest oparte na następujących zasadach:

- **Inicjalizacja kubełków**: Każdy kubełek odpowiada zakresowi wartości, a jego szerokość jest dostosowana do rozkładu wag krawędzi w grafie.
- **Relaksacja krawędzi**: W przypadku poprawy odległości wierzchołka, nowa odległość jest przypisana do odpowiedniego kubełka, a wierzchołek dodawany do kolejki w tym kubełku.
- **Przetwarzanie kubełków**: Algorytm iteracyjnie przetwarza kolejne kubełki, usuwając z nich wierzchołki i rozważając ich sąsiadów. Jeśli sąsiedni wierzchołek może zostać poprawiony, jego nowa odległość jest zapisywana, a wierzchołek jest dodawany do odpowiedniego kubełka.

4.3 Szczegóły działania

1. Inicjalizacja: Na początku algorytm tworzy kubełki odpowiadające różnym zakresom odległości. Każdy kubełek jest przypisany do określonego przedziału wartości, który jest obliczany w sposób dynamiczny w trakcie działania algorytmu.
2. Relaksacja: Dla każdego wierzchołka, który może zostać poprawiony, jego sąsiedzi są badani, a nowe odległości są obliczane. Jeśli nowa odległość jest mniejsza niż dotychczasowa, wierzchołek jest dodawany do odpowiedniego kubełka.
3. Przetwarzanie kubełków: Po dodaniu wierzchołka do kubełka, algorytm kontynuuje przetwarzanie wierzchołków w kolejnych kubełkach, dopóki wszystkie kubełki nie zostaną opróżnione.

4.4 Wnioski

Algorytm Radix Heap znacząco poprawia czas obliczeń w porównaniu do klasycznego algorytmu Dijkstry, szczególnie w przypadku grafów o ograniczonych wagach krawędzi. Dzięki zastosowaniu kubełków i relaksacji tylko tych wierzchołków, które mogą zostać poprawione, algorytm jest bardziej efektywny w porównaniu do tradycyjnych metod opartych na kolejce priorytetowej.

5 Wyniki eksperymentów

5.1 Opis danych testowych

W przeprowadzonych eksperymentach, na danych testowych z 9. DIMACS Implementation Challenge - Shortest Path, analizowane były różne rodziny grafów. Poniżej przedstawiono ich krótkie opisy:

- **Random4-n:** Losowe grafy o n wierzchołkach, w których każdy wierzchołek ma cztery krawędzie. Charakteryzują się niskim stopniem połączenia i służą do testowania algorytmów najkrótszej ścieżki w losowych warunkach.

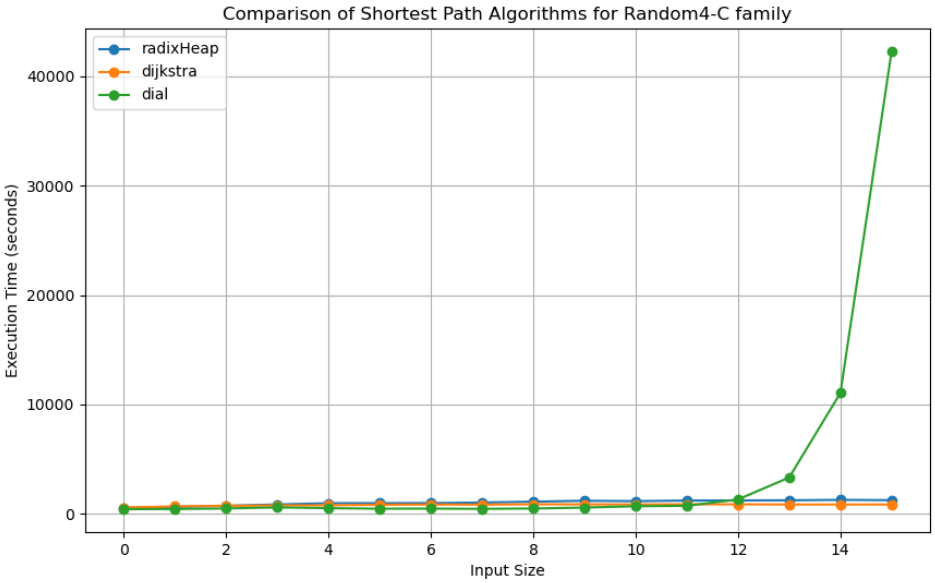
  css Copy code

- **Random4-C:** Grafy o czterech krawędziach na wierzchołek, generowane z dodatkowymi ograniczeniami, co sprawia, że ich struktura jest bardziej złożona. Celem tej rodziny jest analiza wydajności algorytmów w bardziej skomplikowanych scenariuszach.
- **Long-n:** Grafy składające się z długich ścieżek o n wierzchołkach. Mają prostą strukturę liniową, co ułatwia analizę czasów obliczeń i długości najkrótszych ścieżek, będąc idealnym przypadkiem do testowania algorytmów na grafach o niskiej gęstości.
- **Square-n:** Kwadratowe siatki o n wierzchołkach, w których każdy wierzchołek połączony jest z sąsiadującymi. Używane do badania algorytmów w kontekście problemów związanych z sieciami transportowymi i przestrzennymi.
- **Long-C:** Długie grafy o bardziej złożonej strukturze, mogące zawierać cykle. Przeznaczone do testowania algorytmów w sytuacjach, w których występują powtarzające się ścieżki lub cykle.
- **Square-C:** Grafy o strukturze kwadratowej, ale zawierające cykle. Użyteczne do badania efektywności algorytmów w kontekście bardziej skomplikowanych sieci.
- **USA-road-t:** Grafy reprezentujące rzeczywiste sieci drogowe w Stanach Zjednoczonych, zawierające dane dotyczące odległości fizycznych oraz czasu przejazdu. Idealne do analizy algorytmów najkrótszej ścieżki w kontekście rzeczywistych aplikacji transportowych.

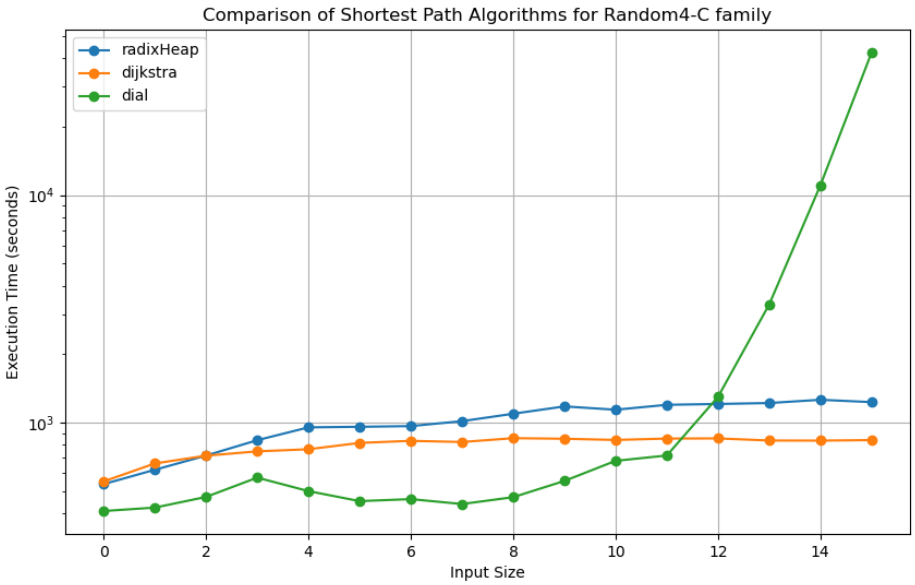
Wszystkie grafy w analizowanych rodzinach są rzadkie, co oznacza, że liczba krawędzi m jest proporcjonalna do liczby wierzchołków n, tzn. $m = O(n)$. Dzięki temu wyniki eksperymentów są łatwiejsze do analizy pod kątem czasu obliczeń oraz długości najkrótszych ścieżek. Wyniki przedstawiono w formie wykresów pełnych oraz obciętych do limitu, aby lepiej zobrazować wyniki eksperymentów.

5.2 Czas wyznaczania najkrótszych ścieżek

- **Random4-C**



Rysunek 1: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 2: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

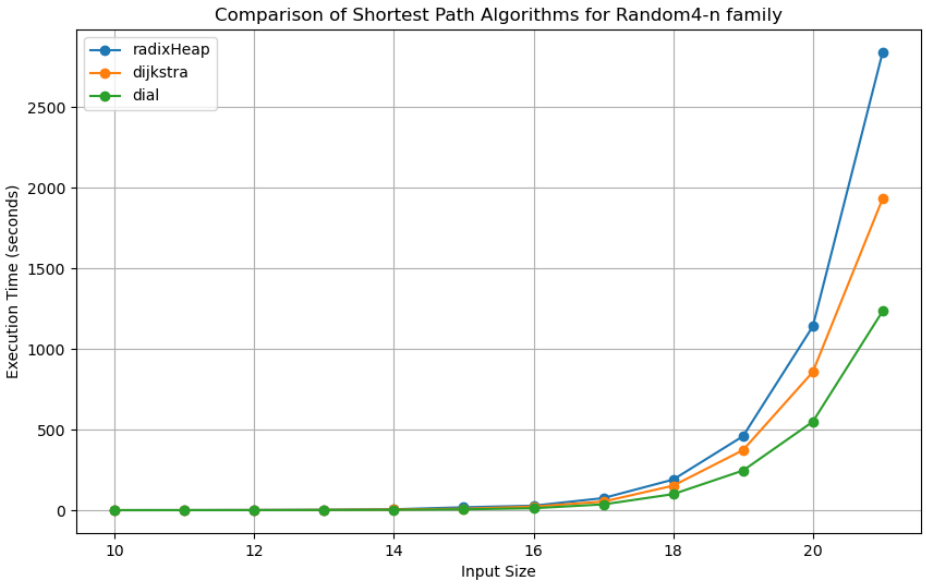
Tabela 1: Caption

Wykresy przedstawiają czas wykonania algorytmów dla grafów typu Random4-C, które charakteryzują się złożoną strukturą, posiadając cztery krawędzie na każdy wierzchołek.

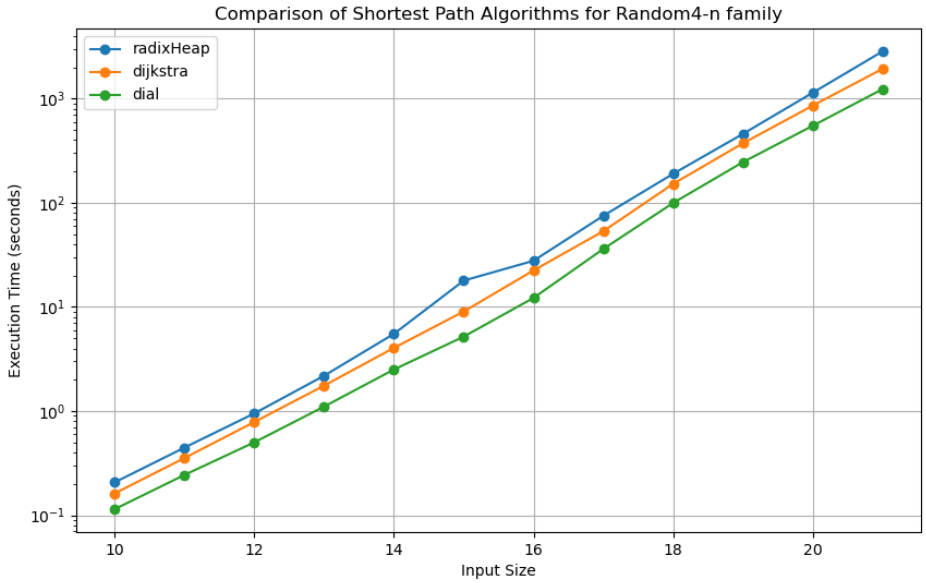
Obserwacje:

- Algorytm podstawowy z użyciem kolejki priorytetowej i RadixHeap zachowuje stabilną wydajność niezależnie od rozmiaru grafu.
- Algorytm Diala wykazuje spadek wydajności przy większych wartościach C, co jest zgodne z jego pseudowielomianową złożonością.

- **Random4-n**



Rysunek 3: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 4: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

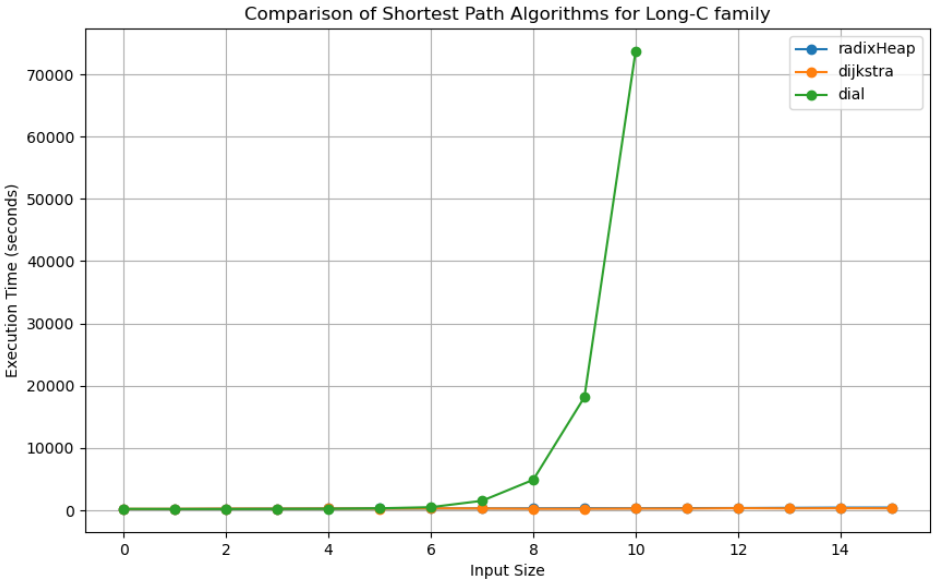
Tabela 2: Caption

Wykresy dla grafów typu Random4-n, które charakteryzują się losową strukturą z czterema krawędziami na wierzchołek.

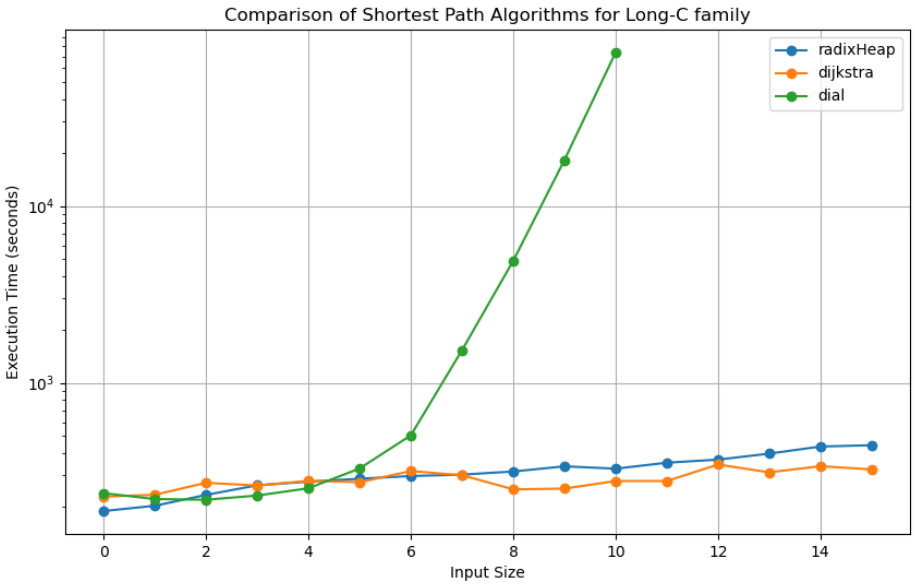
Obserwacje:

- W tym przypadkiu najefektywniejszy wydaje sie algorytm dial'a.
- Algorytm utrzymują stabilną wydajność dla różnych rozmiarów grafu.

- **Long-C**



Rysunek 5: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 6: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

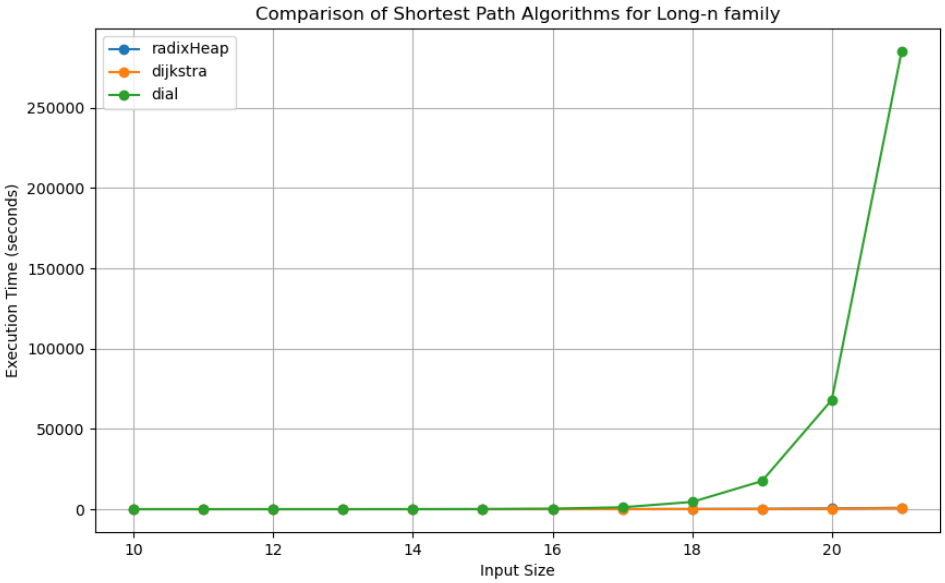
Tabela 3: Caption

Grafy o strukturze długich ścieżek, charakteryzujące się bardziej złożoną strukturą i cyklami.

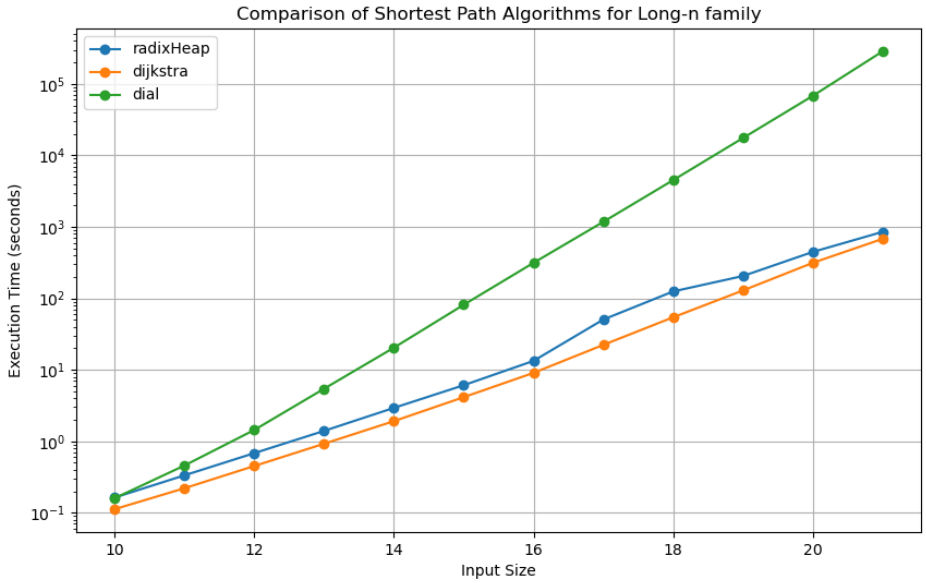
Obserwacje:

- Algorytm Diala jest wyraźnie mniej wydajny w przypadku dużych wartości C.
- Algorytm podstawowy i Radix Heap utrzymują stabilną wydajność dla różnych rozmiarów grafu.

- **Long-n**



Rysunek 7: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 8: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

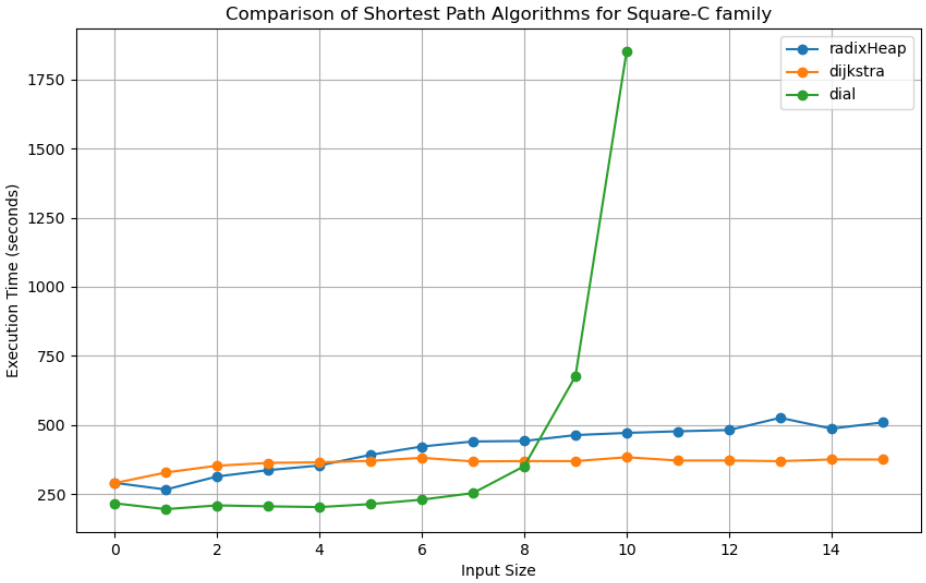
Tabela 4: Caption

Długie ścieżki o prostej strukturze.

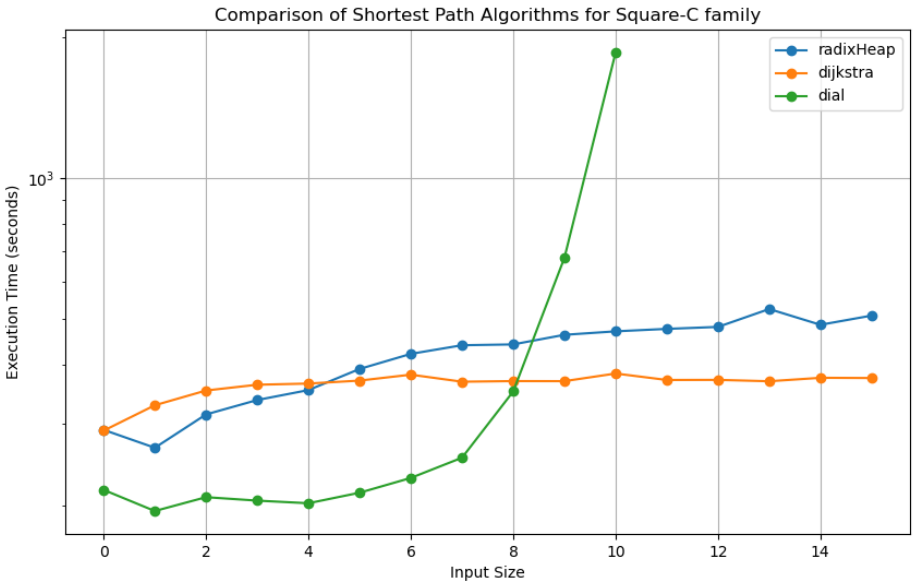
Obserwacje:

- Algorytm Diala jest wyraźnie mniej wydajny w przypadku wiekszych danych.
- Algorytm podstawowy i Radix Heap utrzymują stabilną wydajność dla różnych rozmiarów grafu.

- **Square-C**



Rysunek 9: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 10: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

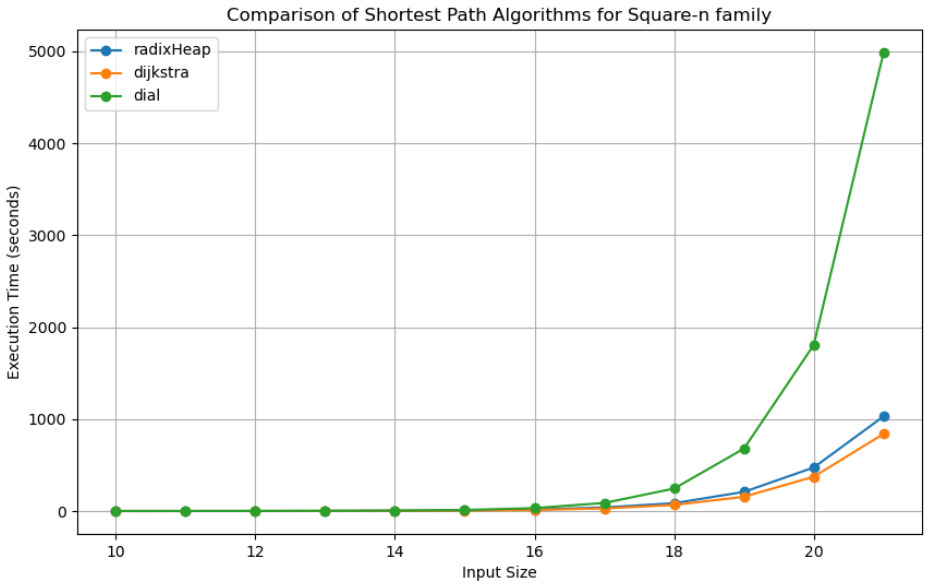
Tabela 5: Caption

Grafy kwadratowe z cyklami.

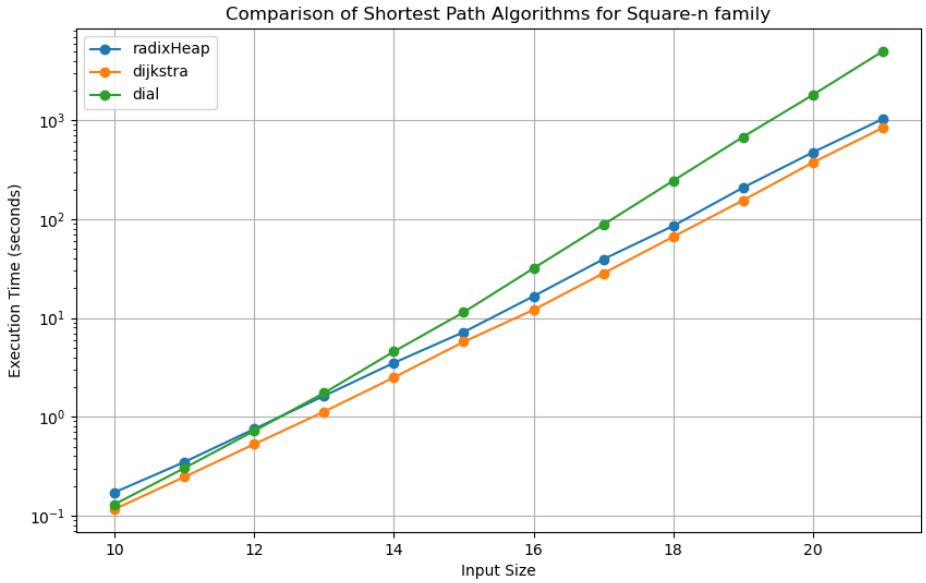
Obserwacje:

- Algorytm Diala jest wyraźnie mniej wydajny w przypadku dużych wartości C.
- Algorytm podstawowy i Radix Heap utrzymują stabilną wydajność dla różnych rozmiarów grafu.

- **Square-n**



Rysunek 11: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 12: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

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

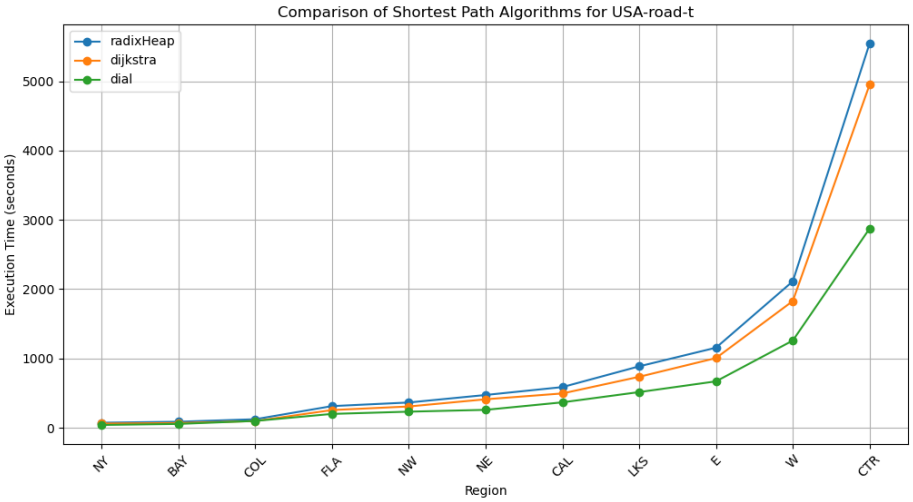
Tabela 6: Caption

Kwadratowe siatki bez cykli.

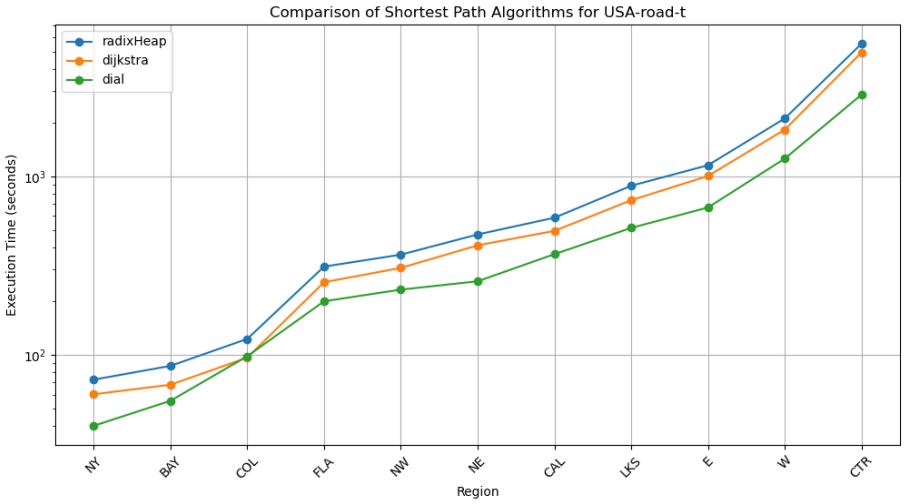
Obserwacje:

- Algorytm Diala jest delikatnie mniej efektywny
- Algorytm podstawowy i Radix Heap utrzymują stabilną wydajność dla różnych rozmiarów grafu.

- **USA-road-t**



Rysunek 13: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.



Rysunek 14: Porównanie czasu wykonania algorytmów dla różnych rodzin grafów.

Rzeczywiste sieci drogowe w Stanach Zjednoczonych

Obserwacje:

– Algorytm Diala jest minimalnie wydajniejszy jednak wszystkie algorytmy wydaja się stabilne.

6 Interpretacja wyników

Analiza uzyskanych danych ujawnia znaczące różnice w efektywności algorytmów w zależności od charakterystyki grafu. W szczególności:

- **Algorytm podstawowy** cechuje się najwyższą wydajnością w przypadku grafów o niewielkiej liczbie wierzchołków.
- **Diala** osiąga lepsze wyniki dla grafów z ograniczonym zakresem wag krawędzi.
- **Radix Heap** okazuje się najbardziej efektywny w przypadku dużych grafów o złożonej strukturze krawędzi.

7 Wnioski

Przeprowadzone eksperymenty umożliwiły ocenę efektywności zaimplementowanych wariantów algorytmu Dijkstry. Nie można wskazać jednego najszybszego algorytmu dla wszystkich przypadków, dlatego wybór odpowiedniego rozwiązania powinien uwzględniać charakterystykę grafu oraz wymagania czasowe.