Sprawozdanie z listy nr 3
Algorytmy Optymalizacji Dyskretnej

Marek Świergoń (261750)

28 maja 2023, PWr, WIiT INA

Celem tej listy było zaimplementowanie i analiza różnych algorytmów wyznaczania najkrótszych ścieżek w grafie $G = (N, A)$ o $|N| = n$ wierzchołkach (dla uproszczenia $N = \{1, 2, \dots\}$) i $|A| = m$ łukach o nieujemnych kosztach. Dodatkowo, największy koszt łuku będziemy oznaczać jako C. Do każdego z algorytmów zostały napisane programy w języku Julia; kody źródłowe stanowią osobny załącznik.

1 Algorytm Dijkstry

1.1 Opis algorytmu

Jak zaznaczono we wstępie, celem algorytmu Dijkstry jest znalezienie najkrótszych ścieżek z wierzchołka startowego s do wszystkich pozostałych wierzchołków (lub do wybranego wierzchołka t). Algorytm ten korzysta z tablicy etykiet $d(i)$, będących ograniczeniem górnym na długość ścieżki $s \to \dots \to i$. Ponadto, na potrzeby odtworzenia najkrótszej ścieżki w pamięci zapisuje się poprzedników każdego z wierzchołków $pred(i)$. Początkowo, etykiety $d(i) = \infty$ i poprzednicy są nieustaleni (tu np. $pred(i) = 0$).

Poza tablicą poprzedników i etykiet $d(i)$, do działania algorytmu Dijkstry potrzebna jest kolejka priorytetowa; sposób jej implementacji ma kluczowy wpływ na asymptotyczny czas działania algorytmu. Jedną z najpopularniejszych jest kolejka bazująca na kopcu binarnym (w języku Julia struktura PriorityQueue z DataStructures.jl). Pozwala ona na wsadzanie wierzchołków i aktualizację ich priorytetów w czasie logarytmicznym oraz wyciąganie wierzchołka z najniższą wartością (czyli najwyższym dla nas priorytecie) w czasie stałym.

Na początku, do kolejki trafia wierzchołek startowy s, a $d(s) = 0$. Następnie, dopóki kolejka nie będzie pusta, opróżniane są z niej wierzchołki w kolejności malejącej wartości etykiet $d(i)$. Po ściągnięciu wierzchołka, przeglądani są sąsiedzi tego wierzchołka pod kątem ewentualnej aktualizacji ich etykiet $d(j)$. Jeśli ścieżka $s \to \dots \to i \to j$ okaże się krótsza od dotychczasowej najkrótszej ścieżki do j, o wartości zapisanej w $d(j)$, to $d(j) = d(i) + c_{ij}$.

1.2 Teoretyczna złożoność obliczeniowa

Przy analizie złożoności obliczeniowej wyróżnić można dwie podstawowe operacje:

- Wybór analizowanego wierzchołka. Każdy z wierzchołków będzie ściągany z kolejki tylko raz.
- Aktualizacja etykiet sąsiadów (update distance). Każdy wierzchołek jest dodawany do kolejki tylko raz. Pozycja wierzchołków w kolejce może być zmieniana (decrease-key) maksymalnie $|A|$ razy, gdyż każda krawędź jest sprawdzana tylko raz.

Otrzymujemy zatem złożoność $O(|N|*T_{extr.min}+(|A|+|N|)*T_{insert/decreasekey})$. Dla kolejki priorytetowej opartej na kopcu binarnym $T_{extr.min} = O(1)$ i $T_{insert} = T_{decreasekey} = O(\log n)$. Należy zauważyć, że złożoność nie zależy od C, czego nie będzie można powiedzieć o kolejnych algorytmach.

2 Algorytm Diala

2.1 Opis algorytmu

Algorytm ten wzoruje się na algorytmie Dijkstry. Choć jego schemat działania jest podobny, to różni się istotnie strukturą przechowującą wierzchołki z etykietą tymczasową, do których optymalna ścieżka niekoniecznie została jeszcze wyznaczona. W przypadku klasycznego algorytmu Dijkstry wykorzystana została kolejka priorytetowa. W algorytmie Diala wierzchołki o dokładnie tej samej wartości etykiety $d(i)$ są przydzielane do jednego „kubełka". Dzięki temu kubełek nie musi być kolejką priorytetową, gdyż wszystkie wierzchołki w nim mają identyczny priorytet. Operacja aktualizacji wartości etykiety polega tym razem na przeniesieniu sąsiedniego wierzchołka do kubełka z mniejszą wartością etykiety. Informację na temat tego, w którym kubełku obecnie znajduje się wierzchołek z etykietą tymczasową, zawiera sama tablica etykiet $D = \{d(1), d(2), \ldots, d(n)\}$.

Algorytm rozpoczyna od kubełka z wartością etykiety równą 0 (znajduje się w nim na pewno wierzchołek s). Kubełki są opróżniane w kolejności rosnącej wartości etykiety $d(i)$.

W podstawowej, „naiwnej" wersji algorytmu Diala potrzebne jest aż nC kubełków. Okazuje się jednak, że liczbę kubełków można zmniejszyć do $C+1$, gdyż sąsiedzi aktualnie rozważanego wierzchołka są maksymalnie oddaleni od niego o C. Mając $C + 1$ kubełków, tworzymy listę cykliczną, po której algorytm będzie przechodził aż do opróżnienia wszystkich kubełków, co może zostać zauważone po stwierdzeniu $C + 1$ pustych kubełków z rzędu.

2.2 Teoretyczna złożoność obliczeniowa

Dzięki podziałowi na kubełki, jesteśmy w stanie pozbyć się złożoności wynikającej z operacji decreasekey i insert na kolejce priorytetowej. Niestety pojawiają się za to inne koszty, związane z przeglądaniem kubełków, powodujące pseudowielomianowość algorytmu. Jeśli założymy, że operacje usuwania i dodawania konkretnego elementu z kubełka jesteśmy w stanie zrobić w czasie stałym (w Julii można to uzyskać na strukturze Set, która operuje na słownikach), to złożoność obliczeniowa algorytmu diala wynosi $O(m + nC)$. Algorytm zatem ma złą złożoność dla danych o zróżnicowanym koszcie łuków.

3 Algorytm radixheap

3.1 Opis algorytmu

Algorytm radixheap stanowi kompromis pomiędzy wariantem podstawowym algorytmu Dijkstry i algorytmem Diala. Zamiast kubełków KFC zawierających tylko jedną wartość etykiety $d(i)$, w algorytmie radixheap kubełki mają różne zakresy wartości etykiet. Są one dobierane w taki sposób, aby asymptotycznie zmniejszyć liczbę potrzebnych kubełków. Pierwsza dwa kubełki mają szerokość 1, czyli taką jak w algorytmie Diala. Kolejne szerokości kubełków to $2, 4, 8, \ldots, 2^{k-1}$, gdzie $k = \lceil \log_2(nC) \rceil$.

Kubełki są opróżniane w kolejności rosnącej wartości etykiet, czyli tak jak w algorytmie Diala, jednak w przypadku, gdy opróżniany kubełek ma szerokość większą niż 1, to jego zawartość jest rozkładana na wszystkie kubełki na lewo od niego, aktualizując ich zakresy, przy czym szerokości kubełków nie ulegają zmianie przez cały czas trwania algorytmu. Po rozłożeniu zawartości kubełka algorytm ponownie rozpoczyna opróżnianie kubełków od kubełka o najmniejszej wartości lewego krańca zakresu

etykiet. Algorytm kończy pracę po przejrzeniu wszystkich kubełków. Procedura update-distance dla sąsiadów wygląda analogicznie do tej z Diala, czyli sąsiad o aktualizowanej wartości etykiety $d(i)$ jest usuwany z poprzedniego kubełka i dodawany do nowego, którego zakres obejmuje jego nową wartość etykiety $d(i)$.

Ze względu na fakt, iż szerokości kubełków w większości są większe od 1, warto jest zastosować kolejkę priorytetową do przechowywania wierzchołków w ramach kubełka. Przyspieszy to operację usuwania z kubełka wierzchołka o najmniejszej wartości $d(i)$.

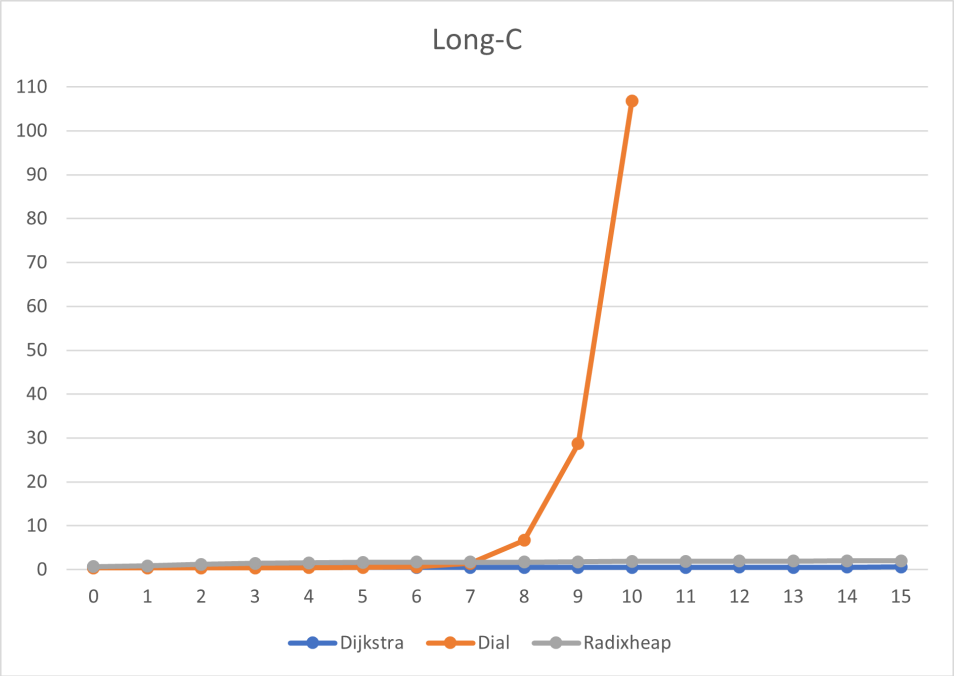
3.2 Teoretyczna złożoność obliczeniowa

Ogólna złożoność obliczeniowa wynosi $O(m+nK)$, gdzie K to liczba kubełków. Wzór ten stosuje się również do algorytmu Diala, gdzie $K = C+1$ dla listy cyklicznej dawało asymptotycznie $O(m+nC)$. W przypadku radixheap, $K = \lceil \log(nC) \rceil$, zatem teoretyczna złożność obliczeniowa tego algorytmu wynosi $O(m + n\log(nC))$. Co więcej, stosując taki sam zabieg jak w algorytmie Diala, możemy zmniejszyć liczbę kubełków do $K = 1+\lceil \log(nC) \rceil$, redukując złożność do $O(m+n\log C)$. Ze względu na fakt, iż zoptymalizowana wersja jest trudna do implementacji, badaniom zostanie poddana wersja o gorszej asymptotycznie złożoności.

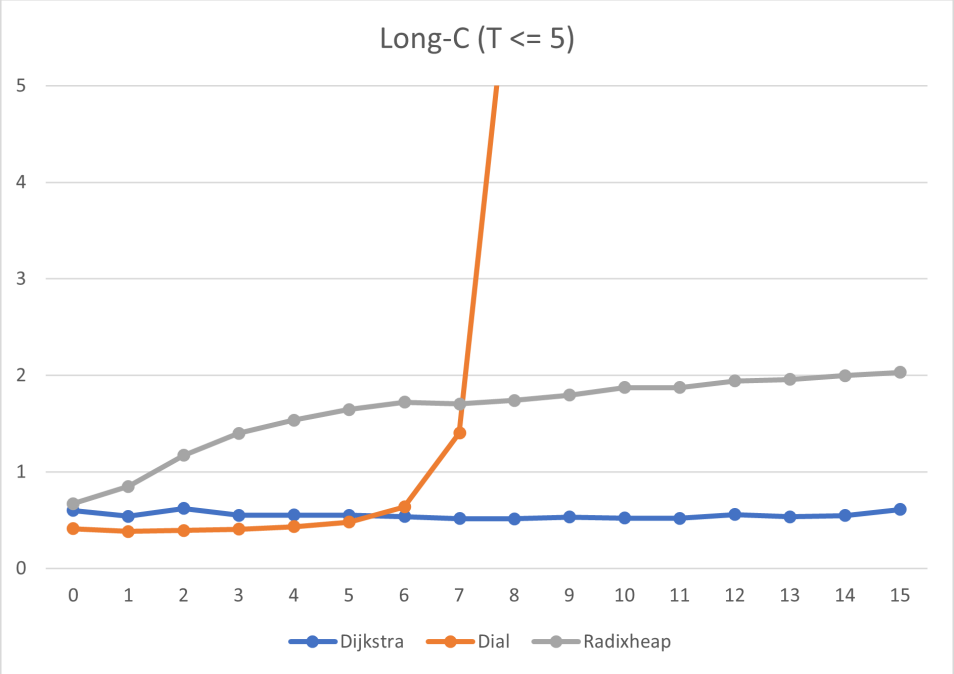
4 Eksperymentalne badanie złożoności obliczeniowej

Badania zostały wykonane na danych testowych pobranych ze strony 9th DIMACS Implementation Challenge – Shortest Paths. Algorytmy były uruchamiane dla sztucznie wygenerowanych instancji rodzin Long-C, Long-n, Random4-n oraz dla pobranej instancji USA-road-t, zawierającej sieci drogowe różnych fragmentów Stanów Zjednoczonych z 2006 roku. Poniższe podrozdziały przedstawiają i omawiają wyniki testów. Wszsystkie testowane grafy są rzadkie, czyli $m = O(n)$. Dzięki temu bardziej widoczne są różnice pomiędzy algorytmami.

4.1 Rodzina Long-C



Rysunek 1: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Long-C (w sekundach).



Rysunek 2: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Long-C (w sekundach) przycięty do wartości czasu równej 5 sekund.

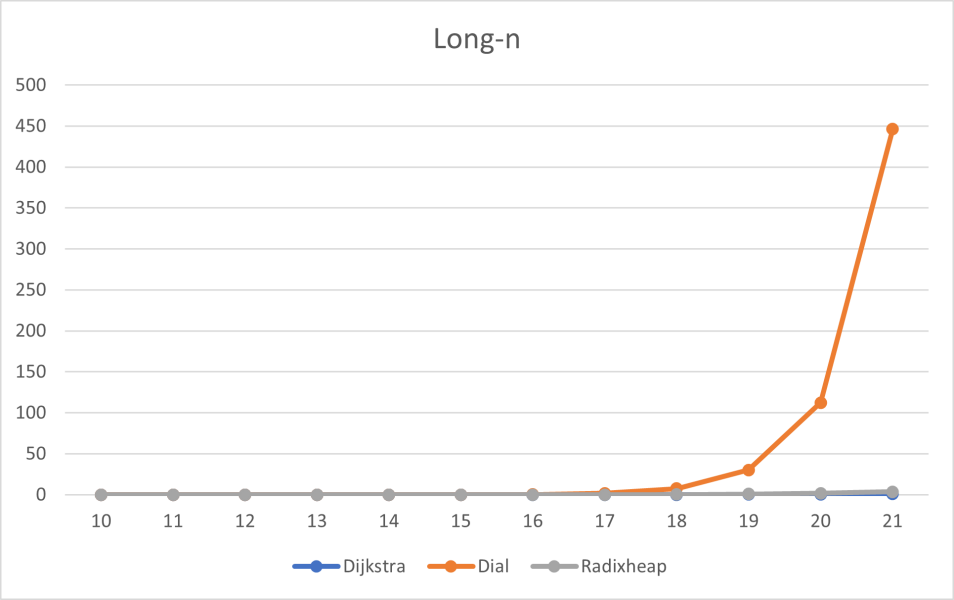
W rodzinie Long-C grafy testowe składają się z podłużnych krat. Liczba wierzchołków jest stała ($n = 2^{20}$), zaś C rośnie, jest równe 4^i dla instancji `Long-C.i.0.gr`, $i \in \{0, 1, \ldots, 15\}$.

Na Rysunku 1 widać, że algorytm Diala bardzo wolno działa dla dużego i. Wynika to z faktu, że jego złożoność istotnie zależy od C; dla $C = 4^i$ otrzymujemy algorytm o złożoności $O(m + n \cdot 4^i)$. Z Rysunku 2 można stwierdzić, że dla małego C algorytm Diala był najszybszy, co również ma podłoże w jego złożoności asymptotycznej.

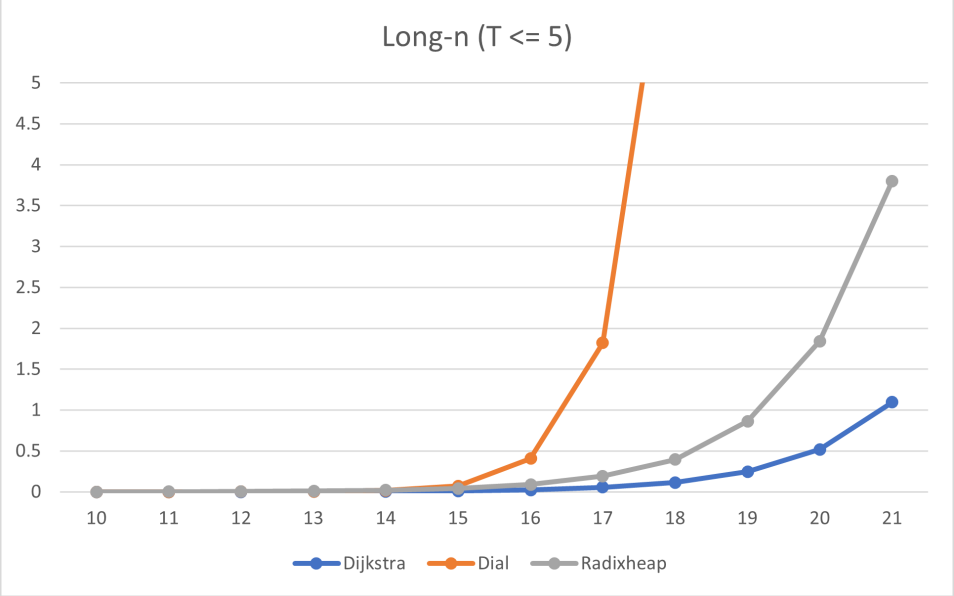
Złożoność obliczeniowa algorytmu Dijkstry nie zależy od wielkości C, co dokładnie można zobaczyć na Rysunku 2, gdzie wartości czasów wykonywania się są w przybliżeniu stałe.

Algorytm radixheap w mniejszym stopniu zależy od C niż algorytm Diala, ponieważ jego złożoność to $O(m+n \log(nC))$. Rysunek 2 potwierdza to stwierdzenie, czasy wykonywania się algorytmu radixheap rosną istotnie wolniej od czasów wykonywania się algorytmu Diala.

4.2 Rodzina Long-n



Rysunek 3: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Long-n (w sekundach).



Rysunek 4: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Long-n (w sekundach) przycięty do wartości czasu równej 5 sekund.

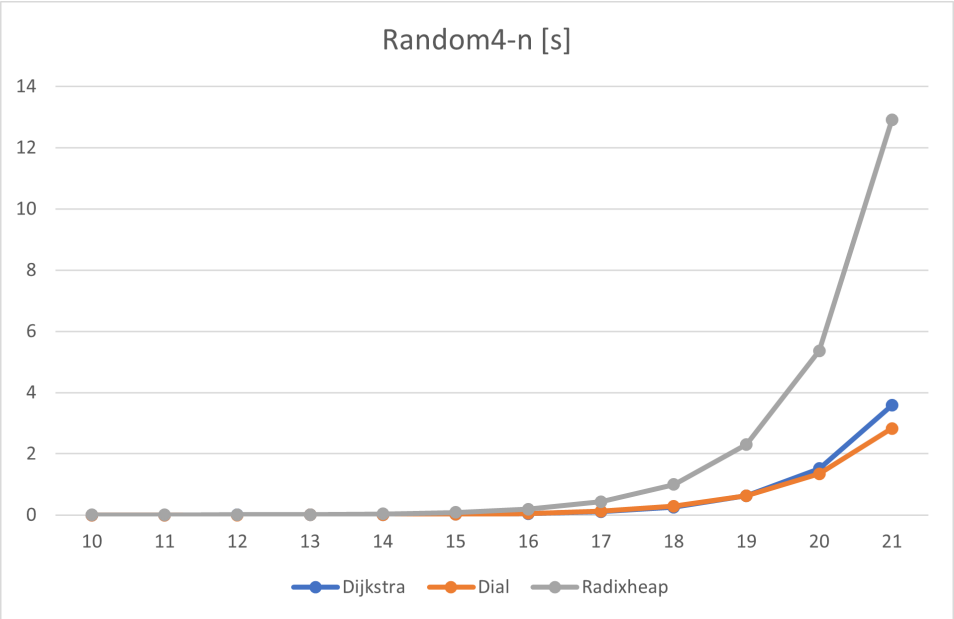
Grafy rodziny Long-n, tak jak rodziny Long-C, również złożone są z podłużnych krat. Wartości C i n są sobie równe i wynoszą $C = n = 2^i$, $i \in \{10, \ldots, 21\}$.

Na czas działania algorytmu Dijkstry wpływ ma tylko rosnące n, więc dla dużych instancji rodziny Long-n algorytm ten był najszybszy.

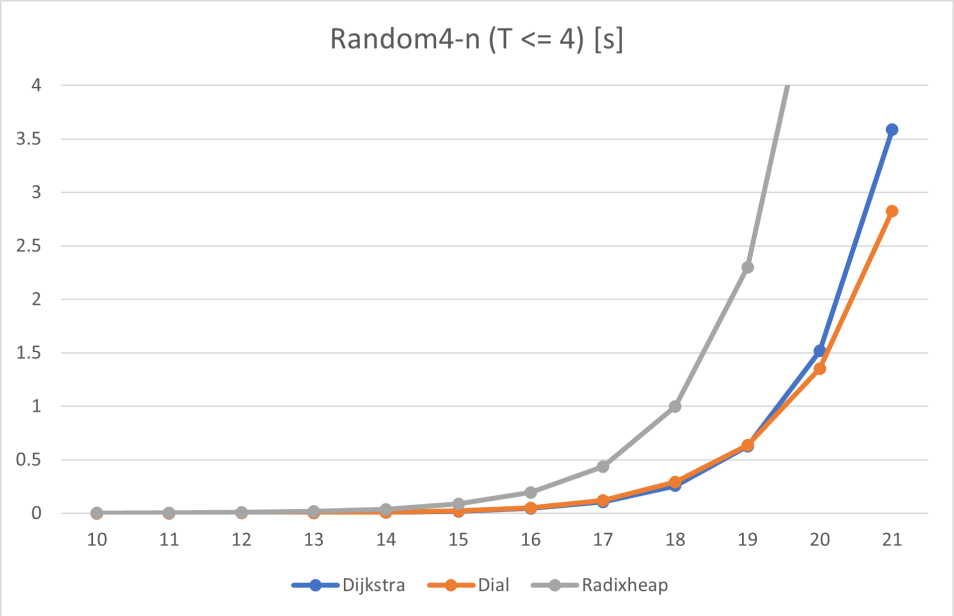
Algorytm Diala wykonywał się najwolniej, gdyż $C = n$ de facto przełożyło się na kwadratową zależność asymptotyczną od rozmiaru grafu.

Analogicznie do testów z rodziny Long-C, algorytm radixheap okazał się szybszy od algorytmu Diala dla dużych instancji, lecz wolniejszy od algorytmu Dijkstry.

4.3 Rodzina Random4-n



Rysunek 5: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Random4-n (w sekundach).



Rysunek 6: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny Random4-n (w sekundach) przycięty do wartości czasu równej 4 sekundy.

W rodzinie Random4-n grafy generowane są w sposób losowy, z zachowaniem $m = 4n$ oraz $C = n = 2^i$, $i \in \{10, \ldots, 21\}$. Wartości kosztów łuków są wybierane losowo z zakresu $[0, \ldots, C]$.

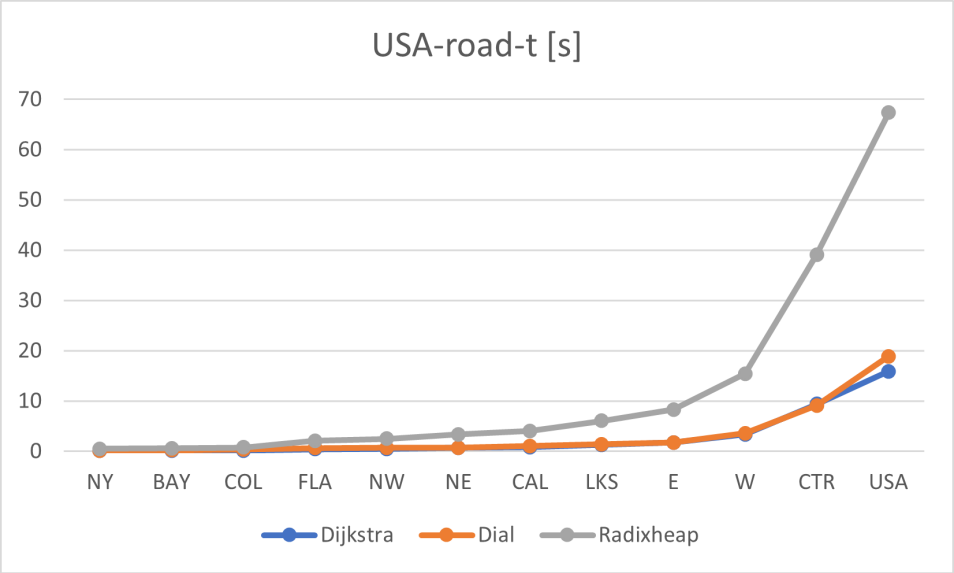
Rysunki 5 i 6 przedstawiają czasy wykonywania się algorytmów dla tejże rodziny.

Najgorszy okazał się algorytm radixheap. Pomimo atrakcyjnej teoretycznej złożności asymptotycznej, koszty stałe związane z nadawaniem nowych zakresów kubełków i rozkładaniem zawartości kubełka na poprzednie kubełki są na tyle duże, że niweczą teoretyczną przewagę algorytmu.

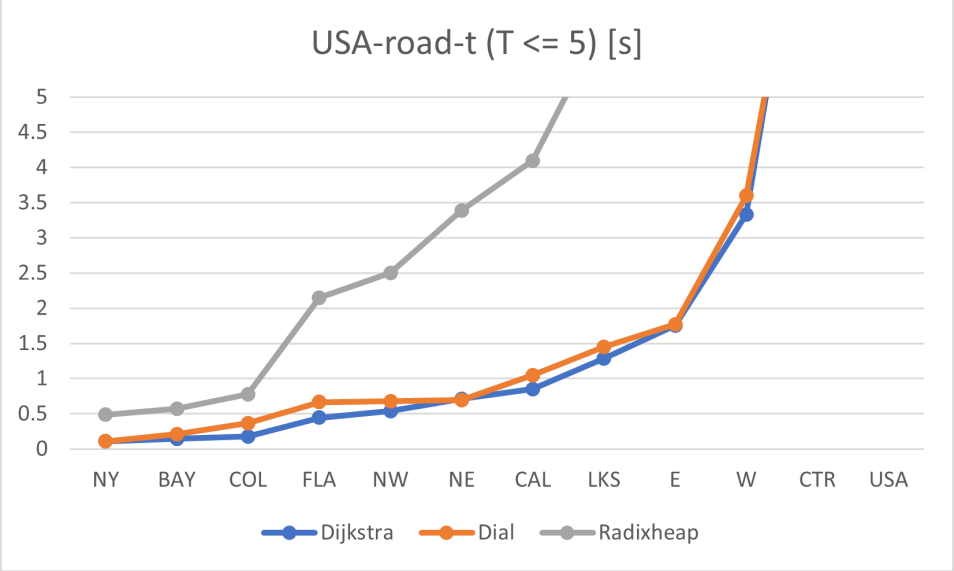
Co ciekawe, algorytmy Dijkstry i Diala uzyskiwały zbliżone czasy, z niewielką przewagą na korzyść algorytmu Diala przy większych instancjach. Jest to pierwsza z testowanych rodzin, dla której algorytm

Diala otrzymuje dobre czasy dla dużych instancji.

4.4 Rodzina USA-road-t



Rysunek 7: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny USA-road-t (w sekundach).



Rysunek 8: Wykres czasu wykonywania się algorytmów dla danych testowych z rodziny USA-road-t (w sekundach) przycięty do wartości czasu równej 5 sekund.

Grafy rodziny USA-road-t stworzone są na podstawie sieci drogowej na terenie Stanów Zjednoczonych z 2006 roku. Jako koszty krawędzi zapisano czasy przejazdu pomiędzy poszczególnymi miejscowościami.

Wyniki dla tej rodziny są podobne do wyników dla rodziny Random4-n. Ponownie radixheap okazał się najgorszy ze względu na kosztowną organizację kubełków. Algorytmy Diala i Dijkstry uzyskiwały podobne czasy: czasami szybszy okazywał się algorytm Diala, a czasem algorytm Dijkstry.

5 Słowo końcowe

Dobór najszybszego algorytmu do wyznaczenia najkrótszych ścieżek nie jest trywialny. Nie istnieje jeden algorytm, który będzie szybszy od pozostałych we wszystkich przypadkach. Przy wyborze algorytmu należy wziąć pod uwagę strukturę grafu, na którym program będzie uruchamiany. Przykładowo, jeżeli koszty łuków są małe i stałe, opłacalny może się okazać algorytm Diala. Z drugiej strony, jeśli C jest duże i związane z rozmiarem grafu, to najpewniejszym wyborem będzie algorytm Dijkstry.