Algorytmy optymalizacji dyskretnej - Lista 3

Tomasz Sierzputowski

Grudzień 2024

1 Algorytmy

Każdy algorytm działa na tej samej zasadzie:

1. Stwórz kolejkę priorytetową Q
2. Ustaw odległość od wszystkich wierzchołków oprócz startowego na ∞, a startowemu na 0
3. Dodaj wierzchołek startowy do Q
4. Ściągnij z kolejki wierzchołek v, o najmniejszym kluczu (najniższa odległość od wierzchołka startowego)
5. Dla każdego sąsiada w wierzchołka v
   Jeśli odległość od w jest większa niż odległość od v + koszt krawędzi (v, w), to
   - Ustaw odległość w na odległość od v + koszt krawędzi (v, w)
   - Dodaj w do Q lub (jeśli już znjadował się w Q) przesuń go w Q
6. Jeśli Q nie jest puste, powróć do punktu 4

Główna różnica polega na tym, z jakiej implementacji kolejki priorytetowej korzystamy.

1.1 d-ary heap Dijkstra

Pierwsza implementacja wykorzystuje kopiec d-arny w kolejce priorytetowej. Wartość d jest wyznaczana jako $\min(2, \lceil \frac{m}{n} \rceil)$, gdzie m to liczba krawędzi grafu, a n to liczba wierzchołków. Konstrukcja:

1. Oblicz $d = \min(2, \lceil \frac{m}{n} \rceil)$
2. Alokacja (tablica przechowująca kopiec; tablica przechowująca indeksy, na których znajdują się konkretne wierzchołki)

Operacja insert(v):

1. Dodaj v na koniec kopca
2. Póki klucz rodzica większy niż klucz v, zamień v z rodzicem w kopcu

Operacja extract_min:

1. Zapisz wierzchołek ze szczytu kopca (o indeksie 0)
2. Przenieś ostatni w kopcu wierzchołek v na szczyt
3. Znajdź minimalny klucz spośród v oraz jego wszystkich dzieci
4. Jeśli minimalny klucz należy do v:
   - Przejdź do następnego punktu

W przeciwnym przypadku:

- Zamień v z dzieckiem, do kórego należy minimalny klucz
- Powróć do punktu 3

5. Zwróć zapisany wierzchołek

Operacja decrease_key(v):

1. Póki klucz rodzica większy niż nowy klucz v, zamień v z rodzicem w kopcu

1.2 Dial

Druga implementacja wykorzystuje kubełki, gdzie każdy kubełek przechowuje elementy o takim samym kluczu. Jeśli C to maksymalny koszt krawędzi w grafie, to mamy $C + 1$ kubełków. Kubełki są ustawione cyklicznie (po kubełku $C+1$ jest kubełek 0). Jeśli w pewnym momencie wierzchołek o najmniejszym kluczu znajduje się w kubełku B to każdy wierzchołek o kluczu większym od minimum o n znajduje się w kubełku $B + n$ (gdzie dodawanie rozumiemy jako przesunięcie zachowujące cykliczność). Jest to możliwe, gdyż w jednej iteracji dodajemy sąsiadów konkretnego wierzchołka, których klucz w sposób oczywisty nie może być większy niż klucz aktualnie usuwanego wierzchołka powiększony o C. Tym samym wszystkie dodane w ten sposób wierzchołki między sobą również nie różnią się o więcej niż C. Dzięki temu, że dodajemy sąsiadów wierzchołka minimalnego, to nigdy nie zwiększy nam się możliwa odległość między kluczami dwóch wierzchołków, czyli $C + 1$ kubełków jest wystarczające. Zauważmy również, że wartość klucza, jakie przechowują kubełki, rośnie co 1 między nimi. Oznacza to, że kubełkowi B może zwiększyć się wartość przechowywanego klucza wyłącznie o wielokrotność $C + 1$.
Konstrukcja:

1. Alokacja (kubełki; tablica wskaźników, gdzie znajdują się konkretne wierzchołki w kubełkach)

Operacja insert(v):

1. Dodaj v do kubełka o indeksie równym kluczowi v (zachowując cykliczność)

Operacja extract_min():

1. Szukaj po kolei pierwszego niepustego kubełka
2. Usuń z tego kubełka pierwszy wierzchołek i go zwróć

Operacja decrease_key(v):

1. Przenieś v z aktualnego kubełka do kubełka o indeksie równym nowemu kluczowi v (zachowując cykliczność)

1.3 Radix heap

Trzecia implementacja wykorzystuje kubełki o zmiennej szerokości (liczbie wartości kluczy, które mogą przechowywać). Pierwszy i drugi kubełek mają szerokość 1 każdy następny 2 razy większą niż poprzedni. Dzięki temu możliwe jest rozdzielenie kubłeka pomiędzy mniejsze, gdyż jeśli szerokość kubełka to 2^n to sumaryczna szerokość wszystkich poprzednich wynosi $1 + 1 + 2^1 + 2^2 + ... + 2^{n-1} = 2^n$. W ten sposób szerokość kubełków rośnie wykładniczo, a co za tym idzie, ich ilość wystarczy, że będzie rosła logarytmicznie. Korzystając z tej samej obserwacji jak poprzednio, że klucze dwóch wierzchołków w kolejce mogą różnić się maksymalnie o C, możemy określić, że potrzebne jest $\lceil \log_2(C+1) \rceil + 1$ kubełków na przechowanie takich wartości. Z powodu tego, w jakich tylko momentach oraz którym kubełkom algorytm zmienia zakres kluczy, potrzebny jest jeszcze jeden kubełek. Jego szerokość jest w teorii nieskończona, ale z obserwacji wiemy, że jego faktyczna szerokość nie przekracza $C + 1$, a zatem zawsze będzie możliwe rozdzielenie na wcześniejsze kubełki.

Konstrukcja:

1. Oblicz $K = \lceil \log_2(C+1) \rceil + 2$
2. Alokacja (kubełki; tablica wskaźników, gdzie znajdują się konkretne wierzchołki w kubełkach)
3. Ustaw każdemu i-temu kubełkowi zakres $[2^{i-1}+1, 2^i)$ (w tym przypadku indeksowanie i zaczyna się od 0, faktycznie w algorytmie przechowywany jest jedynie koniec tego zakresu). Nielicząc ostatniego kubełka, któremu koniec zakresu ustaw na ∞.

Operacja insert(v):

1. Znajdź kubełek (szukając od końca), którego zakres odpowiada kluczowi v
2. Dodaj v do tego kubełka

Operacja extract_min():

1. Znajdź pierwszy nie pusty kubełek B
2. Jeśli jest to kubełek pierwszy, drugi lub zawierający tylko jeden element:
   - Usuń pierwszy wierzchołek z tego kubełka i go zwróć

   W przeciwnym przypadku:
   - Znajdź wierzchołek v o minimalnym kluczu w tym kubełku
   - Ustaw pierwszemu kubełkowi zakres równy temu kluczowi
   - Wszystkim kolejnym kubełkom do kubełka $B-1$ zmień zakres tak, by ich szerokość została zachowana oraz przechowywały kolejne wartości kluczy
   - Przenieś wszystkie wierchołki oprócz v z kubełka B do odpowiednich kubełków 1 - $B-1$
   - Usuń wierzchołek v z kubełka B i go zwróć

Operacja decrease_key(v):

1. Znajdź kubełek (szukając od aktualnego kubełka wstecz), którego zakres odpowiada nowemu kluczowi v
2. Przenieś v do tego kubełka

2 Złożoność obliczeniowa algorytmów

Podczas działania algorytmu każdy wierzchołek musi zostać dodany do kolejki oraz z niej zdjęty (wyjątkiem byłby graf niespójny, ale rozpatrujemy maksymalną liczbę operacji, więc zakładamy, że graf jest spójny). Każda krawędź może nic nie zrobić, spowodować dodanie wierzchołka do kolejki lub zmniejszyć klucz wierzchołkowi, który już jest w kolejce. Zatem jeśli n to liczba wierzchołków, a m to liczba krawędzi, to możemy ograniczyć z góry n operacji insert i extract_min oraz $m-n$ operacji decrease_key. Do naszych rozważań przyda się również wartość C oznaczająca maksymalny koszt krawędzi w grafie.

2.1 d-ary heap Dijkstra

Operacje insert oraz decrease_key mogą maksymalnie przesunąć element z końca kopca na szczyt. Oznacza to, że ich złożoność jest równa wysokości kopca. Kopiec ten może maksymalnie zawierać wszystkie wierzchołki, a wtedy jego wysokość wyniesie $O(\log_d(n))$.
Operacja extract_min może maksymalnie przesunąć element ze szczytu na koniec kopca, czyli wykona maksymalnie liczbę iteracji również równą wysokości kopca. Przy każdej iteracji musi jednak również znaleźć minimum spośród wszystkich dzieci, co jest liniowe od liczby dzieci d.

Finalnie otrzymujemy $n + m - n = m$ operacji o złożoności $O(\log_d(n))$ oraz n operacji o złożoności $O(d \log_d(n))$.
Cały algorytm ma złożoność $O(m \log_d(n) + nd \log_d(n))$, ale skoro $d = \min(2, \lceil \frac{m}{n} \rceil)$, to dla $m \sim n$ mamy $O(n \log(n))$, dla $m \sim kn$ - $O(kn \log_k(n))$, a dla $m \sim n^2$ - $O(n^2)$.

2.2 Dial

Operacje insert oraz decrease_key działają w czasie stałym. Mamy bezpośredni dostęp do każdego kubełka na podstawie klucza wierzchołka. Dzięki również przechowywaniu wskaźników dla każdego wierzchołka, gdzie dokładnie się on znajduje, nie musimy nigdy przeszukiwać samych kubełków.
W operacji extract_min usuwanie wierzchołka z kubełka jest w czasie stałym. Problemem jest szukanie pierwszego niepustego kubełka. Za każdym razem może wymagać to przejścia po wszystkich kubełkach, a jest ich $C + 1$.
Na tej podstawie możemy stwierdzić, że dostaniemy m operacji o czasie stałym oraz n operacji o czasie $O(C)$, zatem algorytm ten ma złożoność $O(m + nC)$.

2.3 Radix heap

Do analizy złożoności tego wariantu algorytmu trzeba podejść bardziej całościowo, a nie analizować każdą funkcję osobno. Operacje insert oraz decrease_key wymagają szukania kubełków, jednak dzięki temu, że szukają zawsze wstecz, a decrease_key zawsze zaczyna od aktualnego kubełka, to dla jednego wierzchołka szukanie kubełków może przejść maksymalnie raz po wszystkich kubełkach podczas wszystkich takich operacji. Dodawanie lub przenoszenie jest w czasie stałym (przenoszenie dzięki temu, że przechowujemy informację, w którym kubełku znajduje się wierzchołek o takim indeksie i gdzie dokładnie w nim), zatem te operacje wykonają się maksymalnie m razy, podczas których przeszukiwanie kubełków zajmie $O(n \log(C))$. Czyli łączna złożoność wszystkich operacji insert oraz decrease_key podczas tego algorytmu to $O(m + n \log(C))$.
Operacja extract_min wymaga znalezienia pierwszego niepustego kubełka, co jest liniowo zależne od liczby kubełków. Problematyczne jest również rozdzielanie wierzchołków, zauważmy jednak, że za każdym razem podczas podziału kubełka, każdy wierzchołek przesuwa się o co najmniej jeden kubełek. A zatem podczas wszystkich wykonań funkcji extract_min, przesunięcie wierzchołka może wystąpić maksymalnie tyle razy, ile jest kubełków dla każdego wierzchołka. W teorii funkcja wymaga również przeszukiwania kubełka w celu znalezienia minimum, jednak odbywa się to taką samą pętlą jak rozdzielanie, zatem nie zwiększa to złożoności ponad podział kubełka. Podsumowując, operacja extract_min ma złożoność $O(\log(C))$ dla każdego wykonania oraz również $O(\log(C))$ dla każdego wierzchołka, co daje łącznie $O(n \log(C))$.
Finalnie algorytm Dijkstry oparty na radix heap ma złożoność $O(m + n \log(C))$.

3 Złożoność pamięciowa algorytmów

3.1 d-ary heap Dijkstra

Implementacja kopca d-arnego wykorzystuje tablicę o stałym rozmiarze. W kolejce mogą znajdować się maksymalnie wszystkie wierzchołki, zatem rozmiar tablicy to n, a co za tym idzie złożoność pamięciowa to $O(n)$.

3.2 Dial

Implementacja, jak przeanalizowano wcześniej, wymaga $C+1$ kubełków. Kubełki są listami, w których łącznie mogą znajdować się maksymalnie wszystkie wierzchołki. Finalnie złożoność pamięciowa tego wariantu wynosi $O(n + C)$.

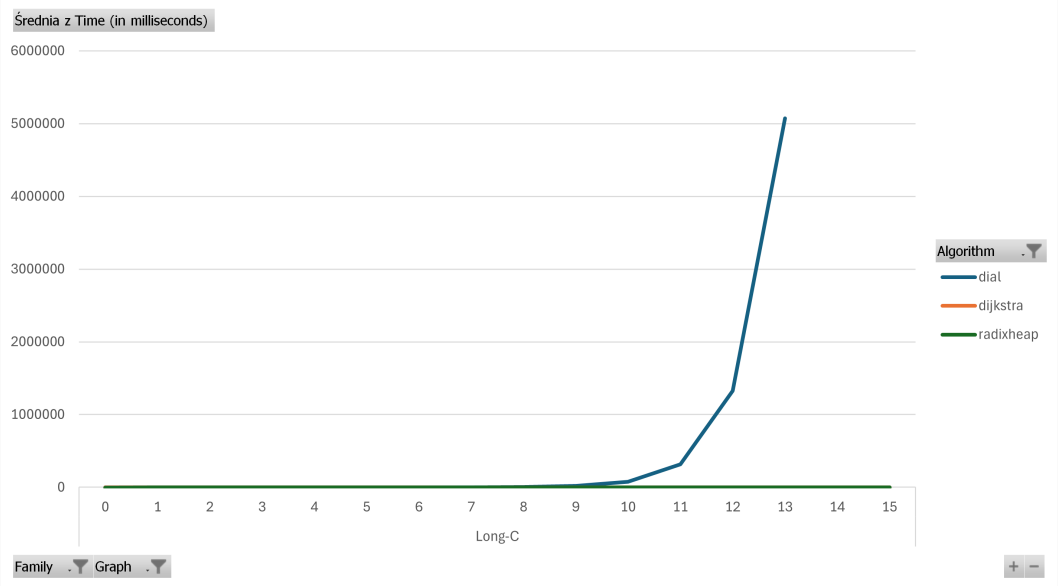
3.3 Radix heap

Implementacja, jak przeanalizowano wcześniej, wymaga $\lceil \log_2(C+1) \rceil + 2$ kubełków. Kubełki są listami, w których łącznie mogą znajdować się maksymalnie wszystkie wierzchołki. Finalnie złożoność pamięciowa tego wariantu wynosi $O(n + \log(C))$.

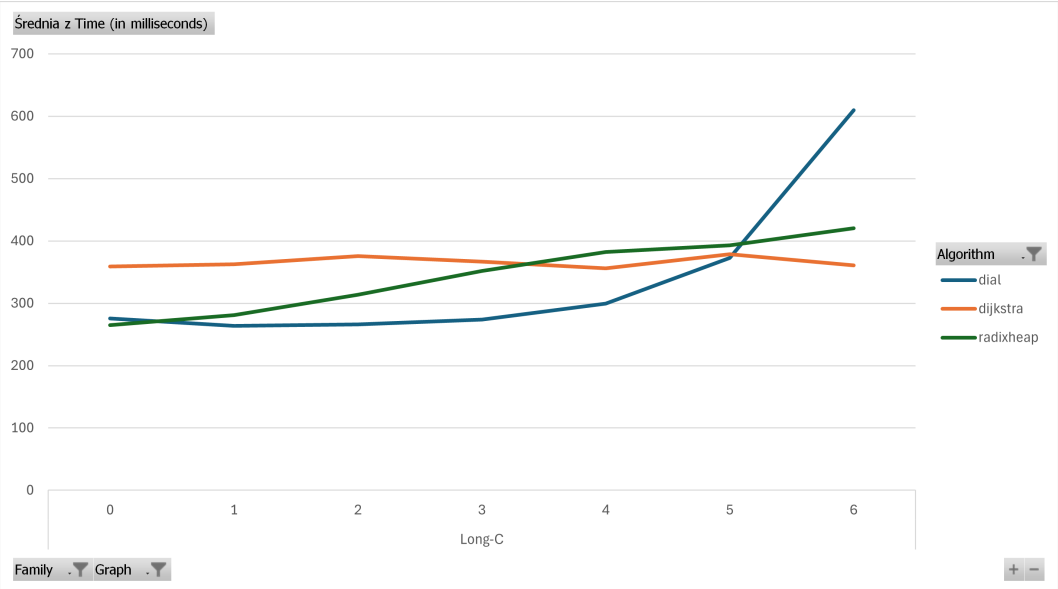
4 Testy

4.1 Wykresu czasu

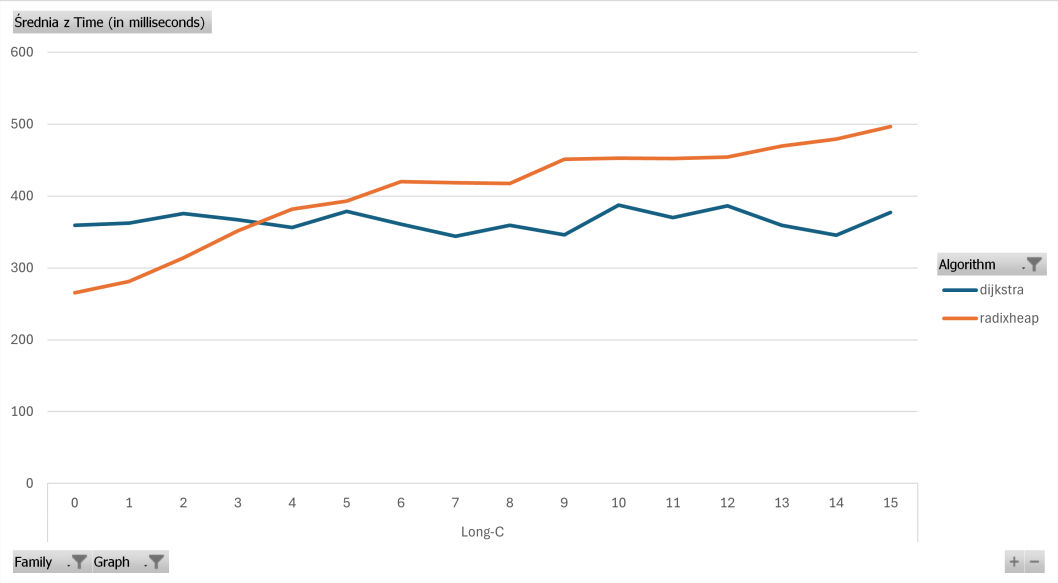
4.1.1 Rodzina grafów Long-C



Algorytm w wersji Diala dla dużych wartości C wymaga BARDZO dużo czasu. Dla $C \sim 4^{13}$ czas przekroczył godzinę. Dla większych wartości testy nie były w stanie odbyć się w sensownym czasie. Przyjżyjmy się bliżej mniejszym wartościom C.

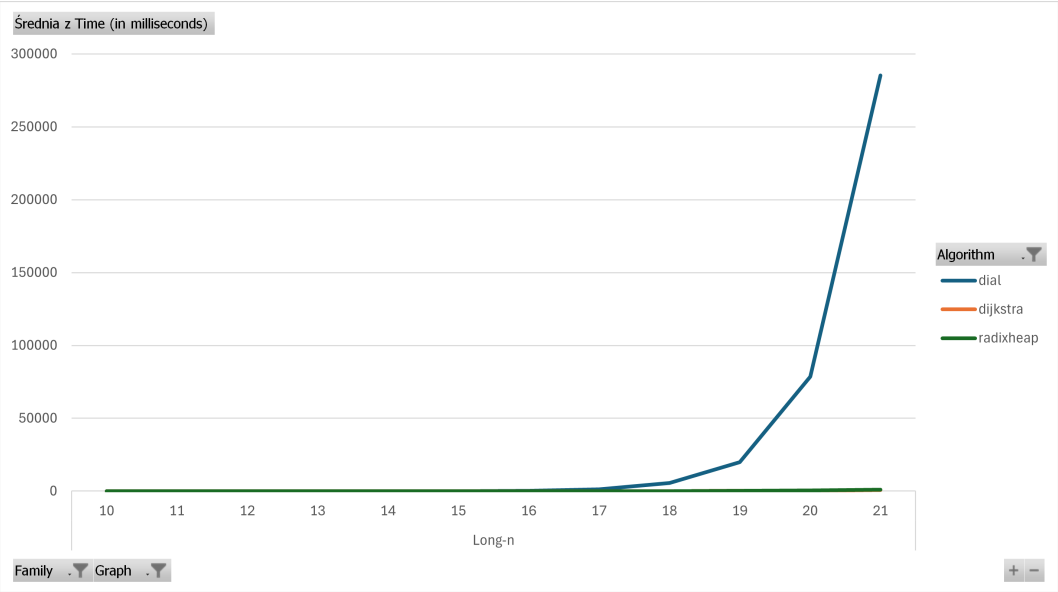


Do pewnego momentu algorytm Diala radzi sobie najlepiej. Wersja standardowa, zgodnie z przypuszczeniami, nie zależy od C, a wersja z radix heap powoli rośnie (C tutaj rośnie wykładniczo, a skoro wersja radix heap w swojej złożoności posiada $\log(C)$, możemy się spodziewać linioiwego wzrostu i taki również widnieje na wykresie).

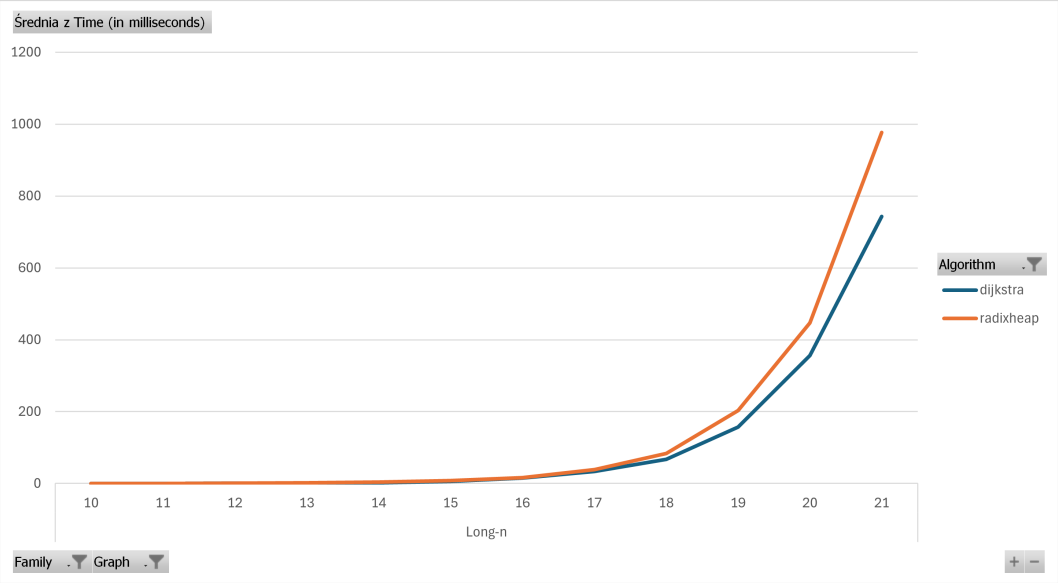


Możemy to również zaobserwować na pełnym wykresie porównującym algorytm z kopcem z algorytmem wykorzystującym radix heap.

4.1.2 Rodzina grafów Long-n

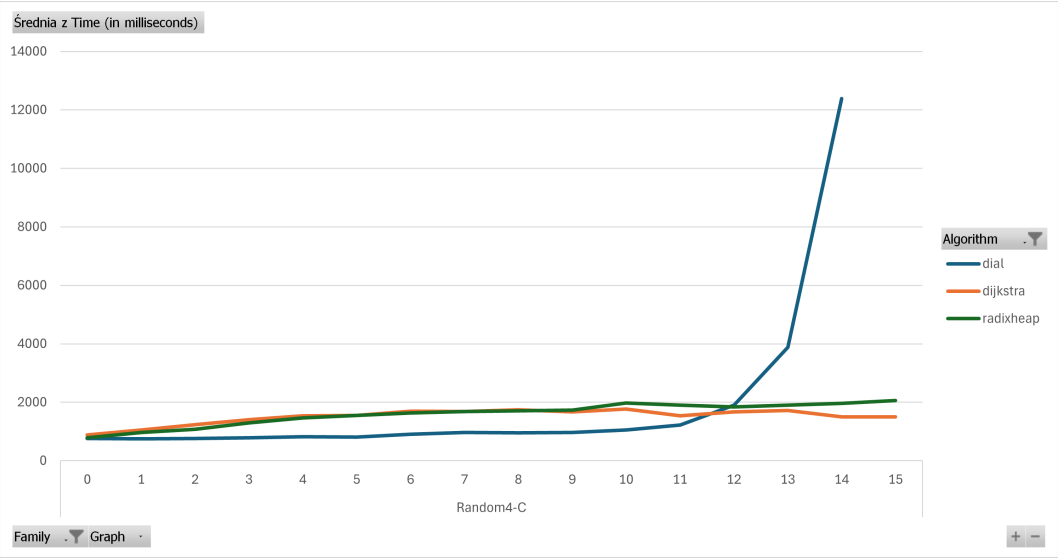


Algorytm Diala ponownie trwa długo dla dużych wartości, tym razem jednak możliwe było przetestowanie go nawet dla największego grafu. Algorytm działa najgorzej od najmniejszych testowanych grafów.



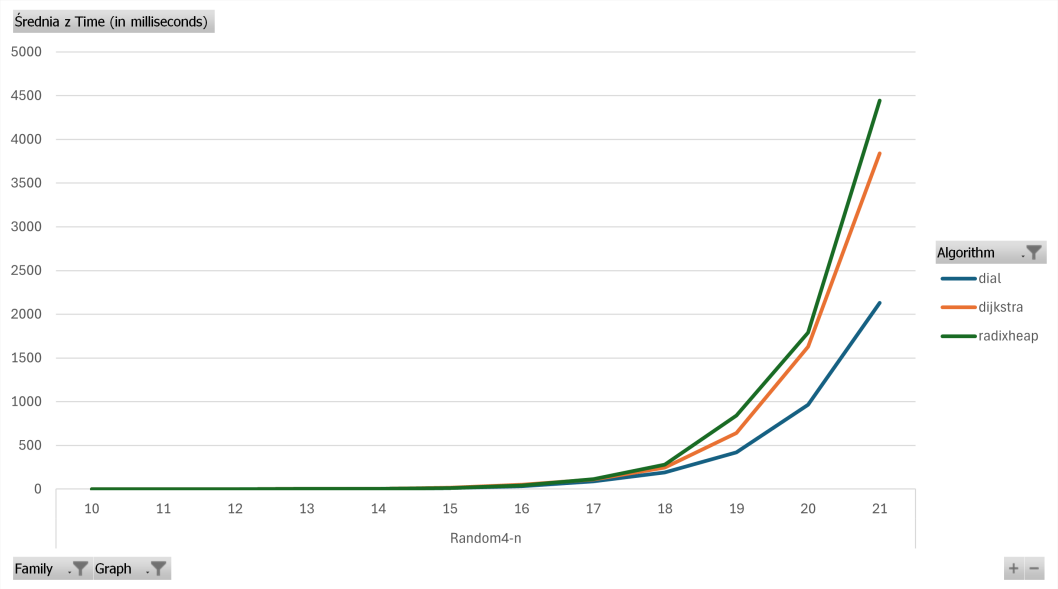
Porównując między sobą wariant standardowy i z radix heap, możemy zauważyć bardzo zbliżone wyniki z lekkim wskazaniem na wersję wykorzystującą zwykły kopiec d-arny.

4.1.3 Rodzina grafów Random4-C



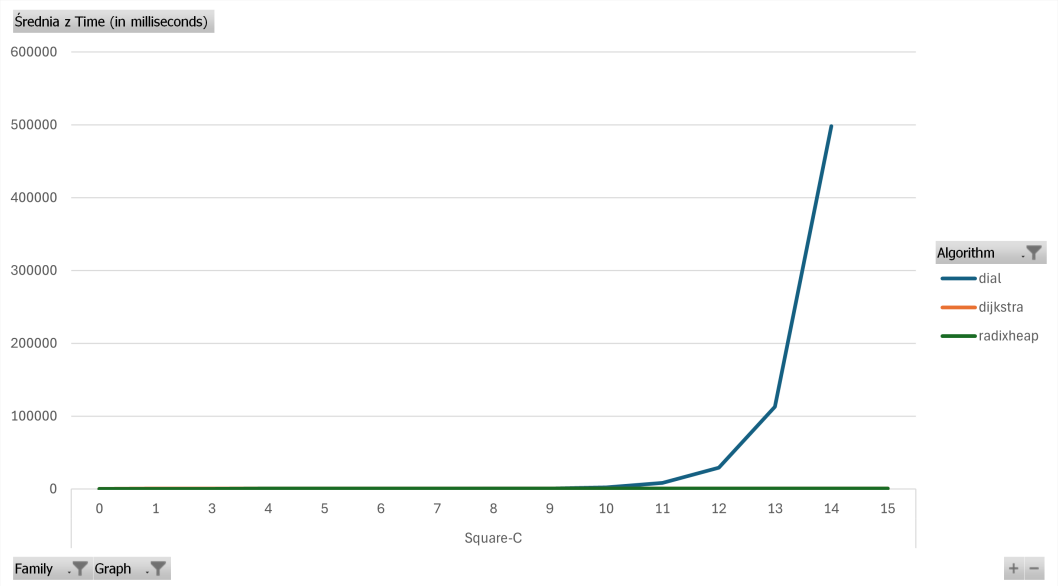
Dla większości grafów algorytm Diala zadziałał najszybciej. Jednak dla opowiednio dużych wartości C czas zaczyna bardzo szybko rosnąć. (Dla grafu 15 nie mam wystarczająco pamięci RAM na komputerze, by zaalokować wszystkie kubełki. Przetestowałem na innym, lepszym komputerze, czas przekroczył pół godziny.)

4.1.4 Rodzina grafów Random4-n

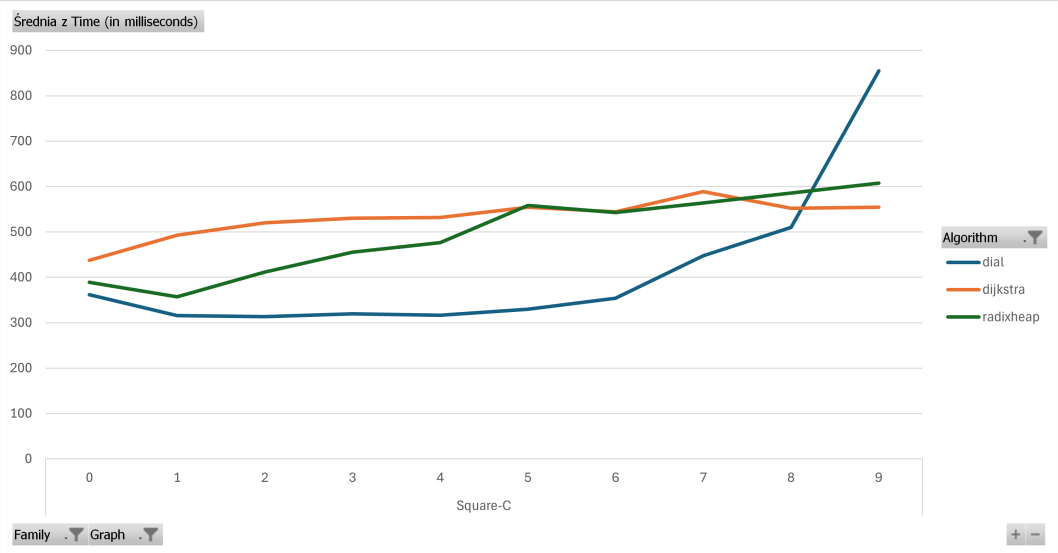


Tym razem algorytm w wersji kubełkowej sprawdził się najlepiej.

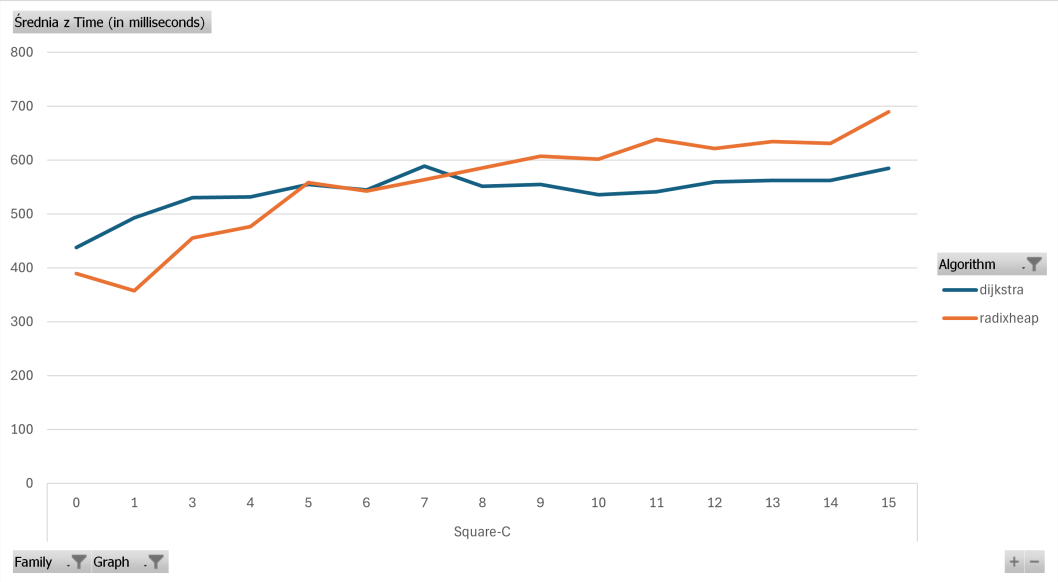
4.1.5 Rodzina grafów Square-C



Algorytm w wersji Diala dla dużych wartości C wymaga dużo czasu. Dla $C \sim 4^{14}$ czas przekroczył godzinę. Dla większych wartości brakowało pamięci RAM i można się spodziewać, że test nie wykonałby się w sensownym czasie. Przyjżyjmy się bliżej mniejszym wartościom C.

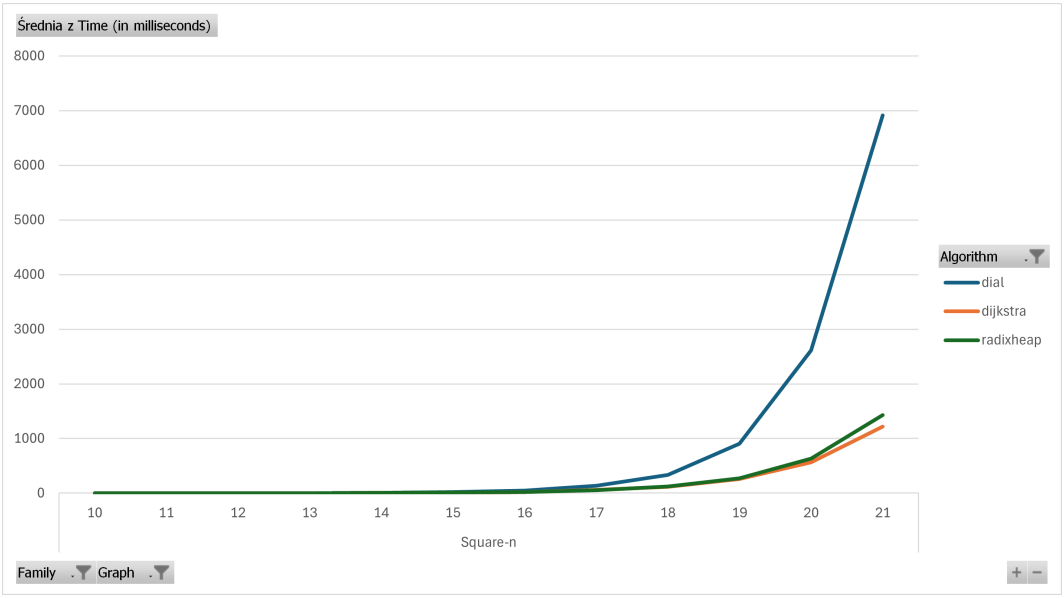


Do pewnego momentu algorytm Diala radzi sobie najlepiej. Wersja standardowa, zgodnie z przypuszczeniami, nie zależy od C, a wersja z radix heap powoli rośnie (C tutaj rośnie wykładniczo, a skoro wersja radix heap w swojej złożoności posiada $\log(C)$, możemy się spodziewać linioiwego wzrostu i taki również widnieje na wykresie).



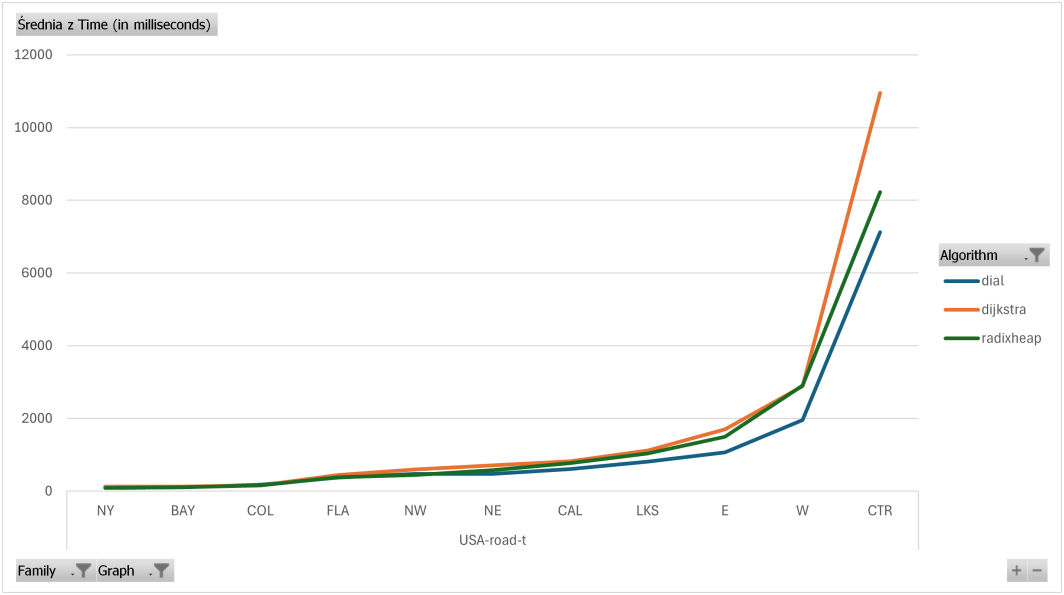
Możemy to również zaobserwować na pełnym wykresie porównującym algorytm z kopcem z algorytmem wykorzystującym radix heap.

4.1.6 Rodzina grafów Square-n



Ponownie algorytm Diala działa najgorzej. Pozostałe dwa podobnie z lekkim wskazaniem na standardowy algorytm z kopcem.

4.1.7 Rodzina grafów USA-road-t



DLa grafu ze świata rzeczywistego wszystkie algorytmy działają w bardzo podobnym czasie. Warto jednak zauważyć, że wyjątkowo niezależnie od wielkości grafu algorytm w wersji Diala działa najszybciej, a standardowy najwolniej.

4.2 Interpretacja wykresów oraz wnioski

Typowy algorytm Dijkstry oraz wersja wykorzystująca radix heap są bardzo wszechstronne. W stosunkowo niedługim czasie były w stanie przerobić wszystkie testy. Standardowy algorytm z kopcem najczęściej radził sobie lepiej niż ten wykorzystujący radix heap. Wyjątkiem są grafy wykorzystujące faktyczne drogi USA oraz duże grafy o stosunkowo małym maksymalnym koszcie krawędzi (małe wartości w rodzinach typu C). Algorytm w wersji kubełkowej (Diala) natomiast, dla grafów o bardzo dużym maksymalnym koszcie krawędzi, nie był w stanie przeprowadzić testów w sensownym czasie. Co jednak ciekawe, w przypadku

małej wartości C często dawał on sobie radę najlepiej. Zwłaszcza, jeśli chodzi o grafy losowe lub na podstawie rzeczywistych dróg w USA.
Dobór algorytmu zatem musi być podejmowany na podstawie rozwiązywanego problemu. Jeśli spodziewamy się grafów o bardzo dużej liczbie losowych krawędzi o niewielkich kosztach, tworzących krótkie ścieżki, powinniśmy wykorzystać wariant Diala. Dla grafów rzadkich, gdzie znajdowane ścieżki są długie, lepiej sprawdzą się wariant z kopcem lub z radix heap. Wybór pomiędzy tymi dwoma powinien zostać podjęty na podstawie szerokości zakresu kosztów krawędzi grafu.
Można również zauważyć, że wykres dla grafów przedstawiających drogi w USA jest bardziej zbliżony do wykresu dla grafów generowanych losowo niż tych generowanych proceduralnie. Można z tego wyciągnąć wniosek, że drogi w USA są ułożone bardziej losowo niż przemyślanie.

4.3 Testy typu point-to-point

Family	Graph	Vertex from	Vertex to	Distance		
				Dial	Dijkstra	Radix heap
Long-C	12	1	1048576	16313024	16313024	16313024
Long-C	12	987702	246356	19790410	19790410	19790410
Long-C	12	984720	80711	16627650	16627650	16627650
Long-C	12	859146	479585	6711620	6711620	6711620
Long-C	12	973482	417264	17323973	17323973	17323973
Long-C	15	1	1048576		10923673	10923673
Long-C	15	614166	436167		6108222	6108222
Long-C	15	910378	861573		21039677	21039677
Long-C	15	137284	477994		13927672	13927672
Long-C	15	15195	964507		10923673	10923673
Long-n	21	1	2097152	5210071	5210071	5210071
Long-n	21	666307	713167	1975525	1975525	1975525
Long-n	21	1509564	184418	13282380	13282380	13282380
Long-n	21	1533931	1592218	5210071	5210071	5210071
Long-n	21	59423	23661	3745092	3745092	3745092
Random4-C	14	1	1048576	16313024	16313024	16313024
Random4-C	14	399648	369209	8918679	8918679	8918679
Random4-C	14	98405	891587	14962594	14962594	14962594
Random4-C	14	568076	411423	8754224	8754224	8754224
Random4-C	14	535293	746889	6857301	6857301	6857301
Random4-C	15	1	1048576		16313024	16313024
Random4-C	15	493463	101208		18594775	18594775
Random4-C	15	734572	309805		9311015	9311015
Random4-C	15	724152	286529		2597729	2597729
Random4-C	15	575329	774802		8496571	8496571
Random4-n	21	1	2097152	7371493	7371493	7371493
Random4-n	21	266381	470618	1008755	1008755	1008755
Random4-n	21	1293134	1660590	1732746	1732746	1732746
Random4-n	21	1093404	1754982	7371493	7371493	7371493
Random4-n	21	294821	1280589	3026645	3026645	3026645

Family	Graph	Vertex from	Vertex to	Distance		
				Dial	Dijkstra	Radix heap
Square-C	14	1	1048576	16313024	16313024	16313024
Square-C	14	441537	965792	15419928	15419928	15419928
Square-C	14	965620	598669	14394526	14394526	14394526
Square-C	14	628279	755528	7188898	7188898	7188898
Square-C	14	57476	18706	4797816	4797816	4797816
Square-C	15	1	1048576		15470117	15470117
Square-C	15	65523	513432		12356141	12356141
Square-C	15	140152	912700		15470117	15470117
Square-C	15	562177	80466		12864949	12864949
Square-C	15	362003	686229		4022293	4022293
Square-n	21	1	2096704	20547320	20547320	20547320
Square-n	21	425801	1051671	10999267	10999267	10999267
Square-n	21	753510	12836	13562916	13562916	13562916
Square-n	21	1182051	497748	8474505	8474505	8474505
Square-n	21	1877933	1938637	20547320	20547320	20547320
USA-road-t	CTR	1	14081816	9709456	9709456	9709456
USA-road-t	CTR	5631866	8913713	14558205	14558205	14558205
USA-road-t	CTR	2011507	7228840	9796128	9796128	9796128
USA-road-t	CTR	257308	1352445	5444610	5444610	5444610
USA-road-t	CTR	4308902	6970609	4616683	4616683	4616683