Sprawozdanie

Algorytmy optymalizacji Dyskretnej
Laboratorium - Lista 3

Paweł Stanik

11. December 2024

0.1 Przedmowa

Repozytorium na githubie

Zadanie polega na implementacji 3 wersji algorytmu Dijkstry, algorytmy zostały zaimplementowane w języku Rust, dane wejściowe zostały wygenerowane za pomocą programu z „9th DIMACS Implementation Challenge – Shortest Paths". Wykonałem eksperymenty dla otrzymanych danych, a wykresy wygenerowane za pomocą biblioteki plotters zamieściłem w dalszej części sprawozdania.

1 Algorytm Dijkstry

Algorytm Dijkstry to algorytm wyznaczania najkrótszych ścieżek w grafie, opiera się on na dosyć prostej zasadzie działania.

1. Dzielimy wierzchołki na:
   Odwiedzone: wierzchołek startowy
   Nieodwiedzone: reszta wierzchołków
2. Wybieramy nieodwiedzony wierzchołek i sąsiadujący z już odwiedzonym j, o najmniejszej odległości od wierzchołka startowego.

   Odległość obliczamy za pomocą wzoru, gdzie $w(i, j)$ to waga krawędzi z i do j:

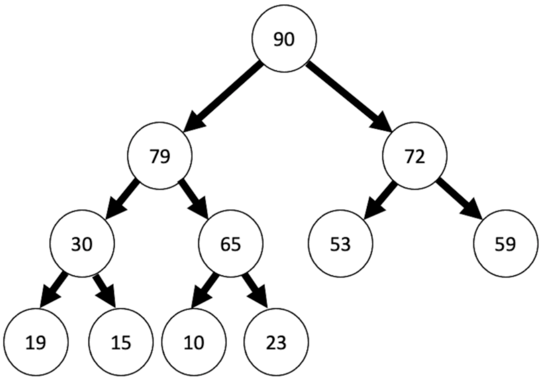
   $$d(i) = d(j) + w(j, i)$$
3. Oznaczamy wierzchołek i jako odwiedzony, i zapisujemy jego odległość $d(i)$
4. Powtarzamy do momentu gdy odwiedziliśmy wszystkie możliwe wierzchołki

Jak widzimy nie jest to zbyt skomplikowany algorytm i wiemy że będzie on poprawny. Żeby jednak działał optymalnie niezbędna jest nam struktura - kolejka priorytetowa, pozwalająca na szybkie wyznaczenie najbliższego wierzchołka

2 Binary Heap

Pierwszą strukturą jaką zaimplementujemy będzie **Kopiec**. Jest to struktura podobna do drzewa o następującej własności:

Każde z dzieci danego elementu jest od niego mniejsze lub równe



Rysunek 1: Wizualizacja Kopca

Moja implementacja kopca to standardowa wersja oparta na tablicy gdzie dla indeksu i

Rodzic: $parent(i) = \lfloor \frac{i-1}{2} \rfloor$

Lewe dziecko: $left(i) = 2 \cdot i + 1$

Prawe dziecko: $right(i) = 2 \cdot i + 2$

2.1 Pola struktury

Heap: tablica elementów na podstawie której działa kopiec

Weight: tablica określająca wagę elementu

Position: tablica wskazująca gdzie w kopcu znajduje się element

ponieważ wiemy że wierzchołki należą do przedziału $[0, n]$ możemy dla wag i pozycji użyć zwykłej tablicy, w innym wypadku można użyć mapy lub drzewa.

2.2 Metody

Bubble up: metoda sprawdza czy element jest mniejszy od rodzica i odpowiednio „podnosi" go w kopcu do osiągnięcia odpowiedniego miejsca

Heapify: metoda sprawdza czy element jest mniejszy od dzieci i odpowiednio „opuszcza" go w kopcu do osiągnięcia odpowiedniego miejsca

Decrease Key: metoda pozwalająca na dodanie elementu lub zmniejszenie jego wagi, dzięki tablicy position możemy go łatwo znaleźć, zmienić i wywołać metodę Bubble Up aby „naprawić" kopiec

Pop: metoda wyciągająca najmniejszy element z kopca, i „naprawiająca" kopiec metodą Heapify

2.3 Złożoność

Bubble Up, Heapify: przejście całej tablicy $\longrightarrow O(\log(V))$

Pop: wyciągnięcie $O(1)$, naprawa **Heapify** $O(\log(V)) \longrightarrow O(\log(V))$

Decrease Key: dodanie/zamiana $O(1)$, naprawa **Bubble Up** $O(\log(V)) \longrightarrow O(\log(V))$

Dijkstra: maksymalnie V wyciągnięć oraz E zmian klucza $\longrightarrow O((E + V)\log(V))$

3 Algorytm Diala

Algorytm diala przypomina w działaniu algorytm Dijkstry, wykorzystuje on jednak specjalną strukturę którą nazwiemy **Dial Bins**. Jest to struktura zawierająca $C + 1$ kubełków gdzie C to maksymalna waga krawędzi. W naszej strukturze kubełek o indeksie i zawiera wierzchołki o dystansie $d(v) \bmod (C + 1) = i$.

Algorytm cyklicznie przechodzi po kubełkach wybierając elementy i wkładając nowe. Wiemy że dla wyciągniętego elementu l nie dodamy nigdy elementu o dystansie większym niż $l + C$ widzimy więc że takie przechodzenie da nam niemalejący ciąg elementów

Tabela 1: Przykładowy stan kubełków

Kubełek	**0**	**1**	**2**	**(3)**	**4**	**5**	**6**	**7**	**8**	**9**	**...**
Zawartość	3 14 12	4		15		9	13 17	10	11 21	8 7	

Więc jeśli aktualnie wybranym kubełkiem jest $i = 3$ to:

$$
\begin{aligned}
d(15) &= 3 \\
d(9) &= 5 \\
d(3) &= C + 1 \\
d(4) &= C + 2
\end{aligned}
$$

Widzimy więc że struktura działa poprawnie i zwraca aktualnie najmniejszy element. Wydajność struktury zależy jednak od C co oznacza że radzi ona sobie bardzo dobrze dla grafów o małych wagach, a dla grafów o wagach dużych jest bardzo wolna. Wymaga ona również aby wagi były liczbami naturalnymi.

3.1 Pola struktury

Cursor: numer aktualnie wybranego kubełka

Bins: (pseudo) cykliczna tablica kubełków na podstawie której działa struktura, w strukturze każdy kubełek to osobna tablica

Distance: tablica określająca dystans od źródła dla danego wierzchołka

Position: tablica wskazująca gdzie w w którym kubełku znajduje się element, oraz na którym miejscu w kubełku

3.2 Metody

Next: iteruje po tablicy i wyciąga wierzchołek z pierwszego niepustego kubełka

Add: metoda pozwalająca na dodanie elementu lub zmniejszenie jego wagi, dzięki tablicy position możemy go łatwo znaleźć, zmienić i dodać ponownie na nowe miejsce

3.3 Złożoność

Next: przechodzimy maksymalnie C kubełków czyli $\longrightarrow O(C)$

Decrease Key: dodanie/zamiana $O(1)$, naprawa $O(1) \longrightarrow O(1)$

Dial: maksymalnie V wyciągnięć oraz E zmian klucza $\longrightarrow O(E + V \cdot C)$

4 Radix Heap

Radix heap to kolejny algorytm wywodzący się od algorytmu Dijkstry, wykorszystuje on kolejkę priorytetową **Radix Heap** od której wzięła się nazwa algorytmu. Również wykorzystuje ona kubełki, jednak przedział dystansów elementów należących do kubełka jest inny.

Tabela 2: Przedziały poszczególnych kubełków

Kubełek	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**...**
Przedział	$\{0\}$	$\{1\}$	$[2,3]$	$[4,7]$	$[8,15]$	$[16,31]$	$[32,63]$	

Jak widzimy przedziały kubełków zwiększają się eksponencjalnie, pozwala nam to na użycie znacznie mniejszej liczby kubełków do przechowywania wierzchołków dla danego C, gdzie liczba kubełków sprowadza się do $\log(V \cdot C)$ co jest znacznie mniejszą liczbą od algorytmu Diala.

Algorytm tak jak Dial przechodzi po kubełkach i szuka pierwszego niepustego, wyciąga najmniejszy element, po tym jednak rozdysponowuje pozostałe elementy z kubełka do wcześniejszych kubełków, zmieniając ich przedziały. Wyjątkiem jest wyciąganie z kubełka o długości przedziału równej 1, wtedy nie wykonujemy redystrybucji.

Tabela 3: Przedziały po wyciągnięciu elementu e gdzie $d(e) = 8$

Kubełek	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**...**
Przedział	$\{8\}$	$\{9\}$	$[10,11]$	$[12,15]$	$\emptyset$	$[16,31]$	$[32,63]$	

Widzimy że struktura działa i zwraca najmniejszy element. Dodatkowo jej wydajność czasowa nie zależy od C jak w algorytmie Diala, wymaga ona jednak, tak samo jak w algortmie Diala, aby wagami były liczby naturalne.

4.1 Pola struktury

Last: dystans ostatnio wybranego elementu

Bins: tablica kubełków na podstawie której działa struktura, w strukturze każdy kubełek to osobna tablica

Distance: tablica określająca dystans od źródła dla danego wierzchołka

Position: tablica wskazująca gdzie w w którym kubełku znajduje się element, oraz na którym miejscu w kubełku

4.2 Metody

Next: iteruje po tablicy i wyciąga najmniejszy wierzchołek z pierwszego niepustego kubełka, oraz redystrybuuje pozostałe elementy.

Add: metoda pozwalająca na dodanie elementu lub zmniejszenie jego wagi, dzięki tablicy position możemy go łatwo znaleźć, zmienić i dodać ponownie na nowe miejsce

4.3 Złożoność

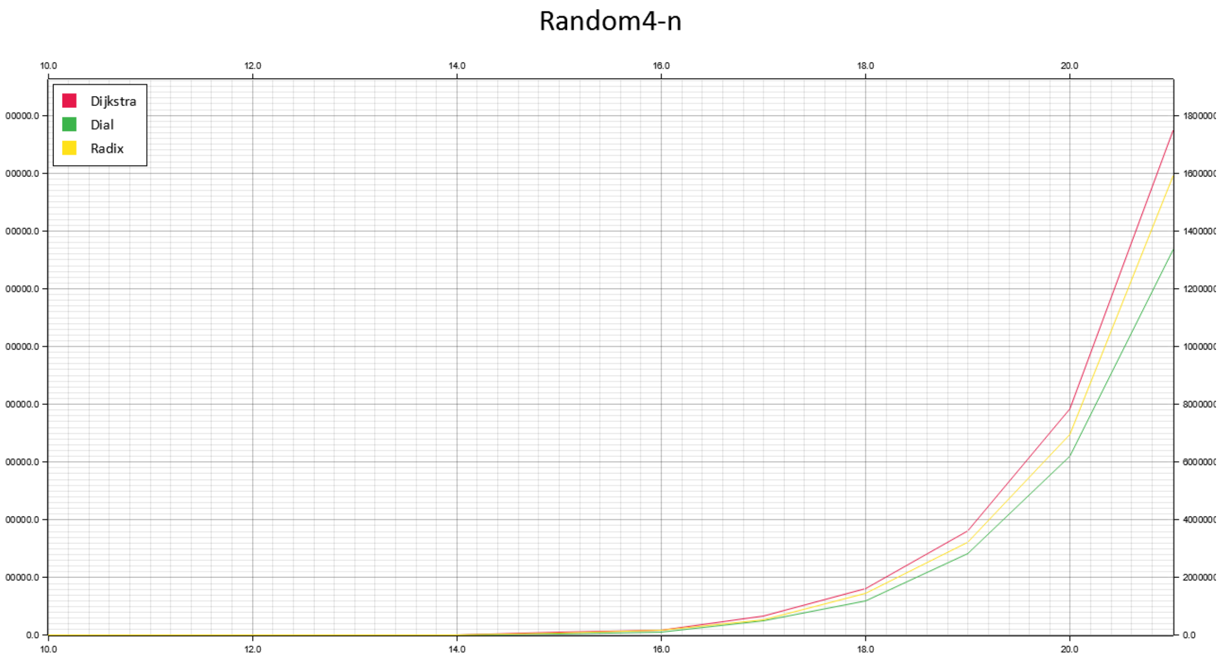
Przeniesienie: każdy węzeł może być przeniesiony conajwyżej $\log(V \cdot C)$ razy

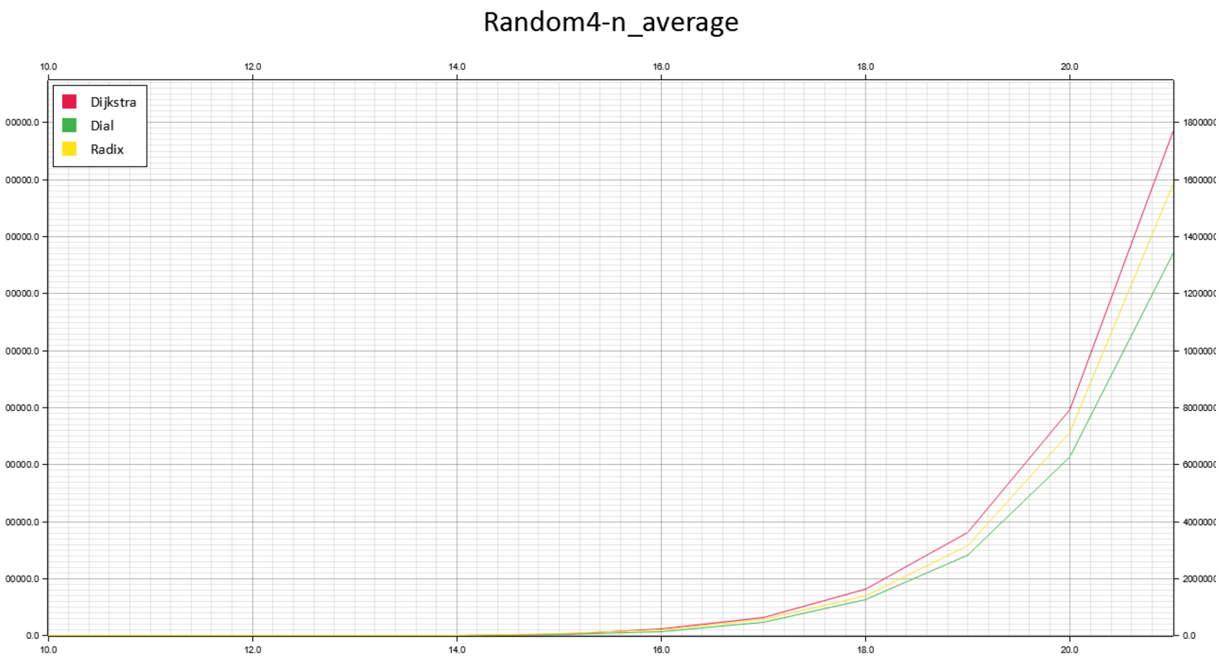
Next: przechodzimy maksymalnie $\log(V \cdot C)$ kubełków + **Przeniesienia** $\longrightarrow \log(V \cdot C)$

Decrease Key: dodanie/zamiana $O(1)$, naprawa $O(1) \longrightarrow O(1)$

Radix Heap: maksymalnie V wyciągnięć oraz E zmian klucza $\longrightarrow O(E + V \cdot \log(V \cdot C))$

5 Random-4-n

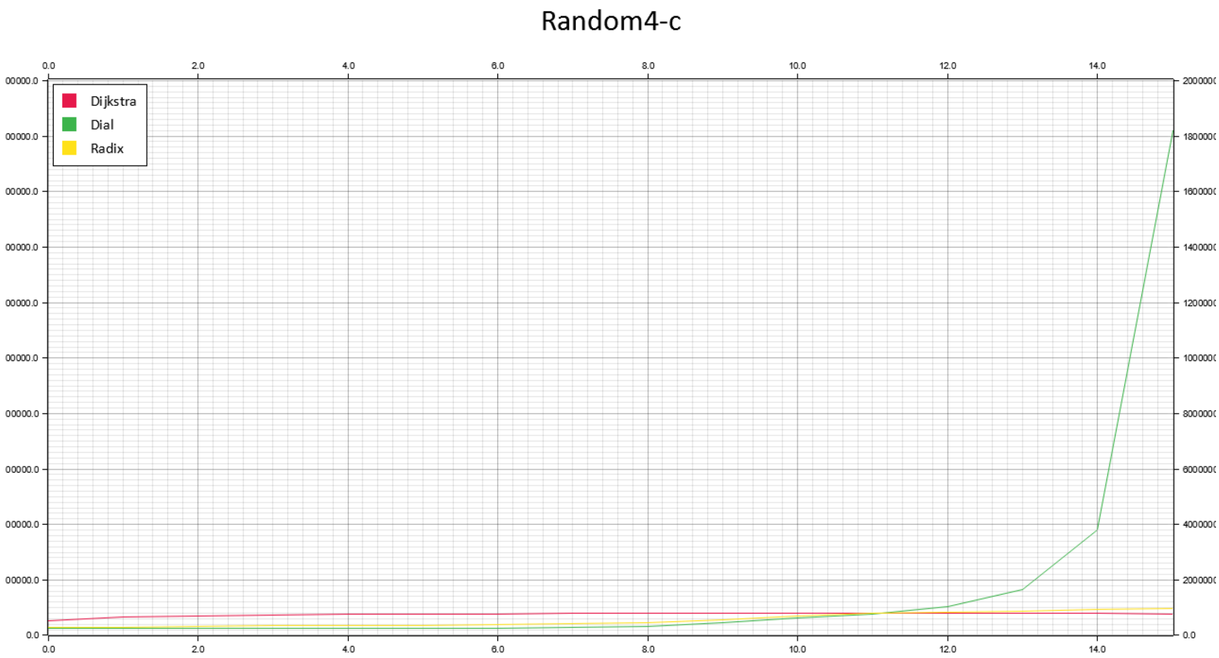


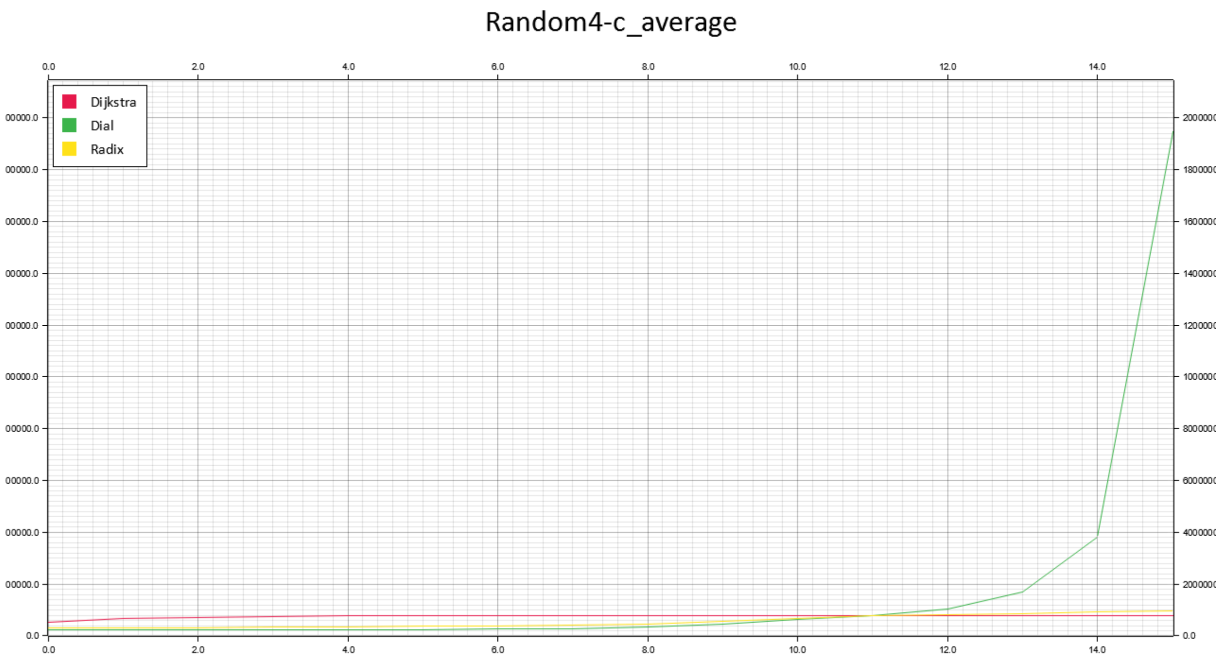


Widzimy że dla grafów losowych o $C = V$ najszybszy jest algorytm Diala, następny jest Radix Heap a na końcu Dijkstra. Pokazuje to szybkość Diala dla stosunkowo niskich C dla których został zaprojektowany.

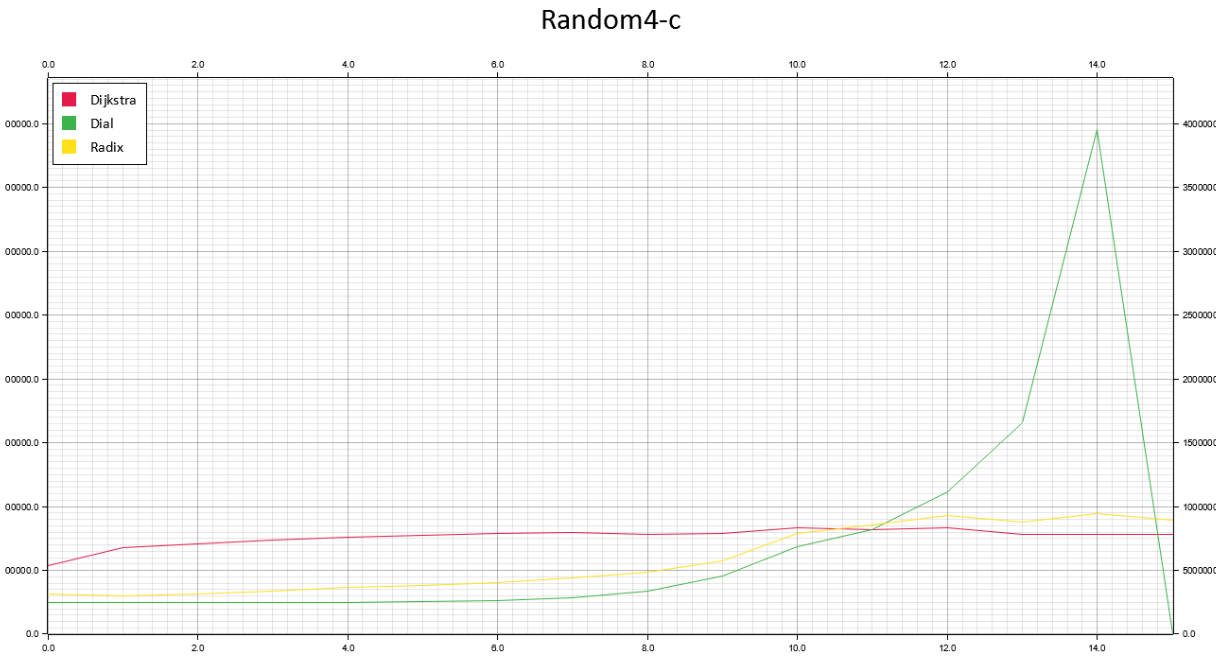
Wszystkie algorytmy zwiększają czas działania eksponencjalnie wraz z eksponencjalnym wzrostem n

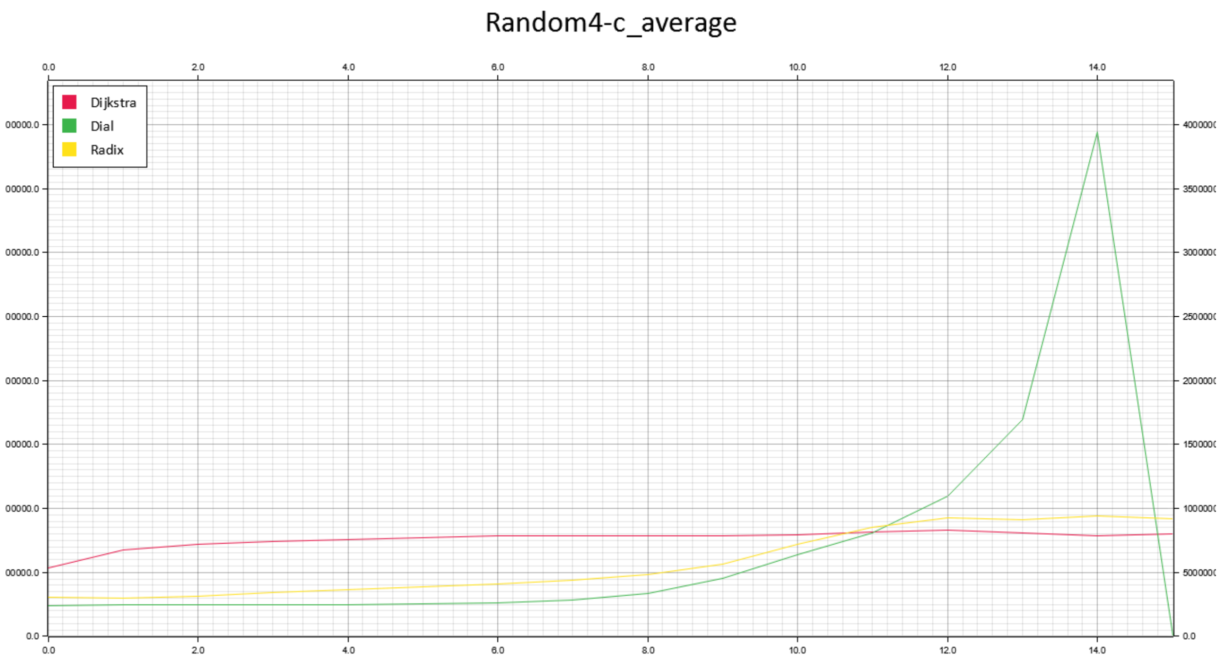
6 Random-4-C


Random4-c
Dijkstra
Dial
Radix


Random4-c_average
Dijkstra
Dial
Radix

6.1 Wykresy z obcięciem Diala



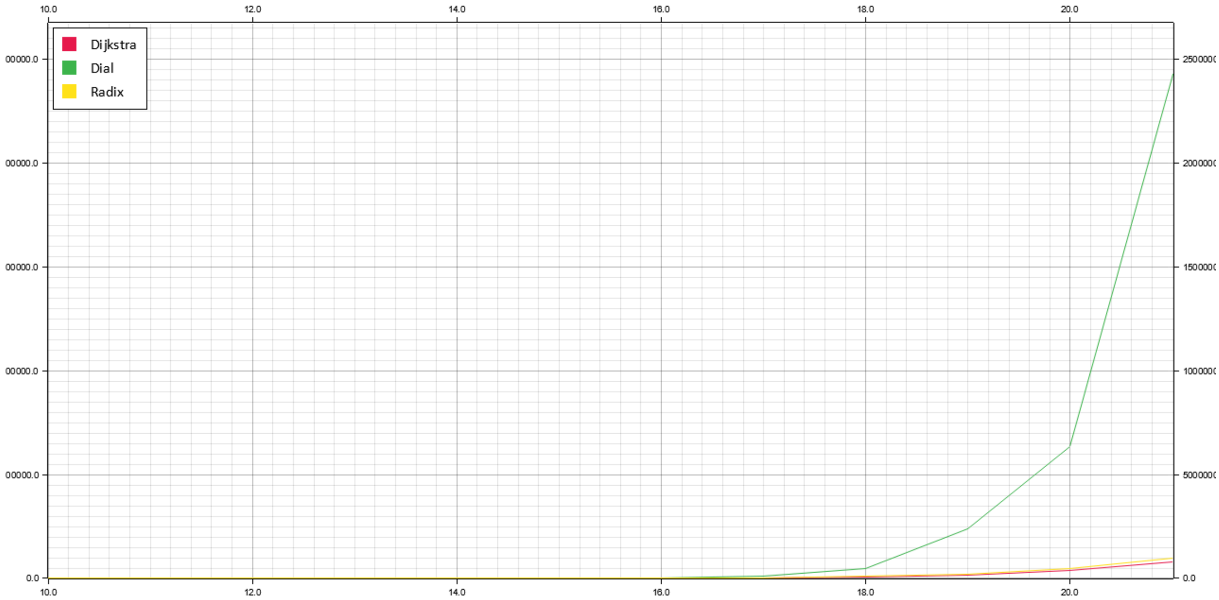


Początkowo widzimy wyniki podobne do poprzednich algorytm Diala jest niemal dwa razy szybszy od Dijkstry, Radix Heap jest niewiele wolniejszy. Jednak dla $C = 4^{11}$ wydajność Diala drastycznie spada a Radix Heap staje się nieco wolniejszy, przez co na prowadzenie wychodzi algorytm Dijkstry który przez cały czas pozostaje niemal stały.

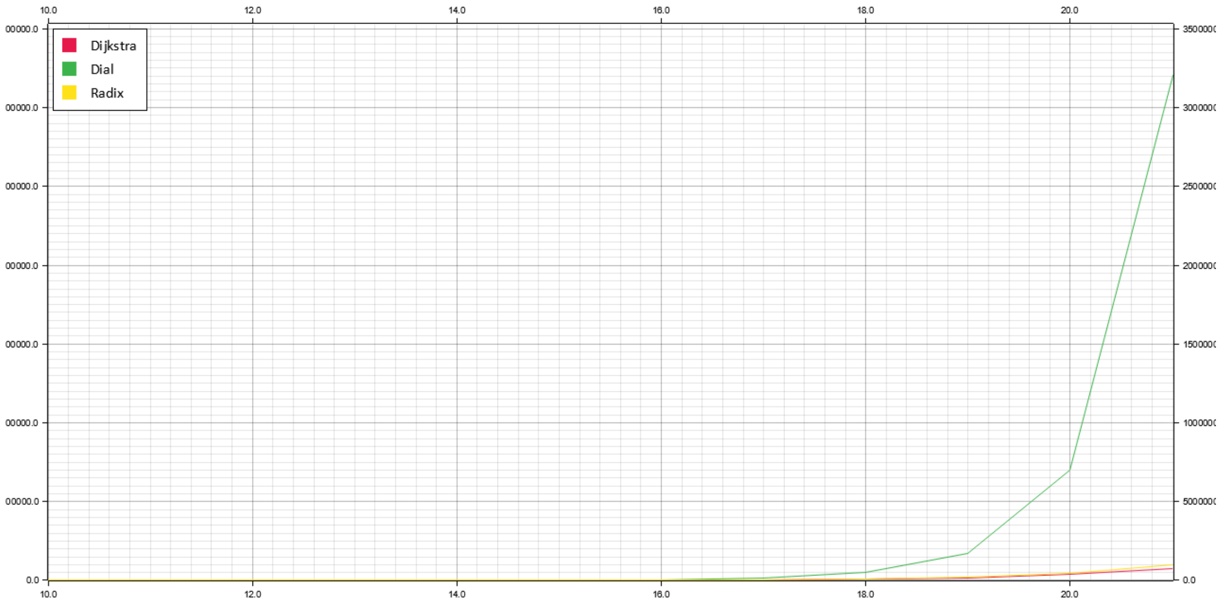
Obserwujemy eksponencjalny wzrost czasu działania Diala dla eksponencjalnego wzrostu C ponieważ jego wydajność czasowa zależy od C. Algorytm Dijkstry nie zależy czasowo od C więc pozostaje stały a Radix Heap zależy od C jedynie logarytmicznie więc nie zmienia się zbytnio.

7 Long-n

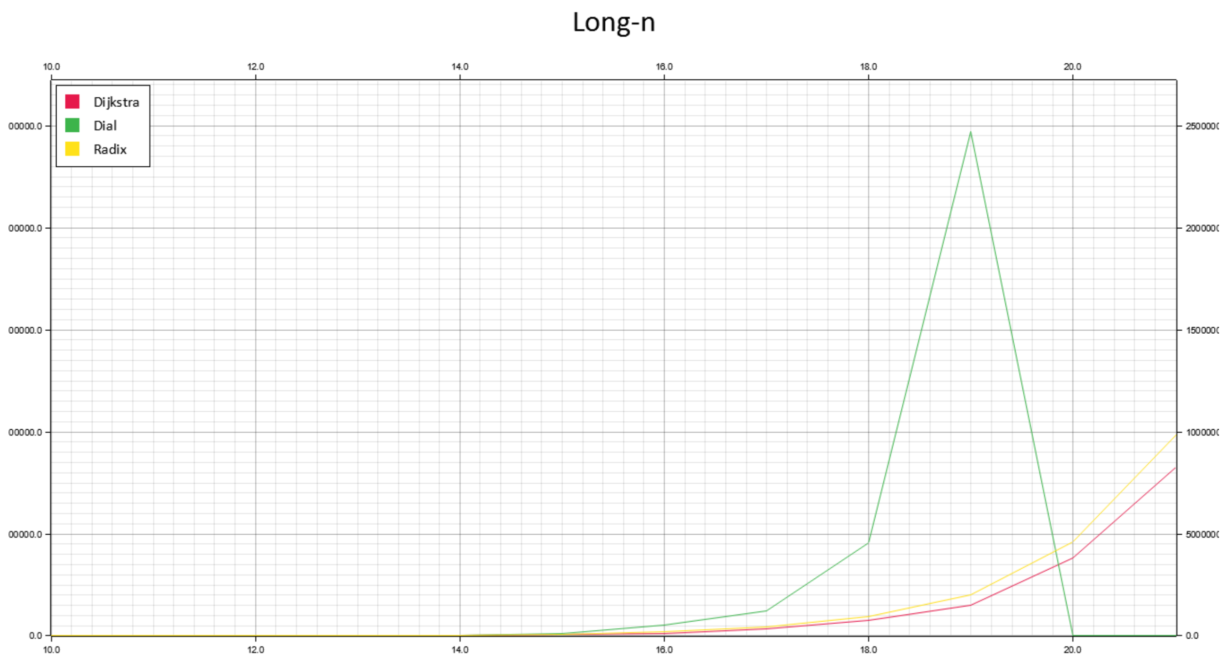
Long-n

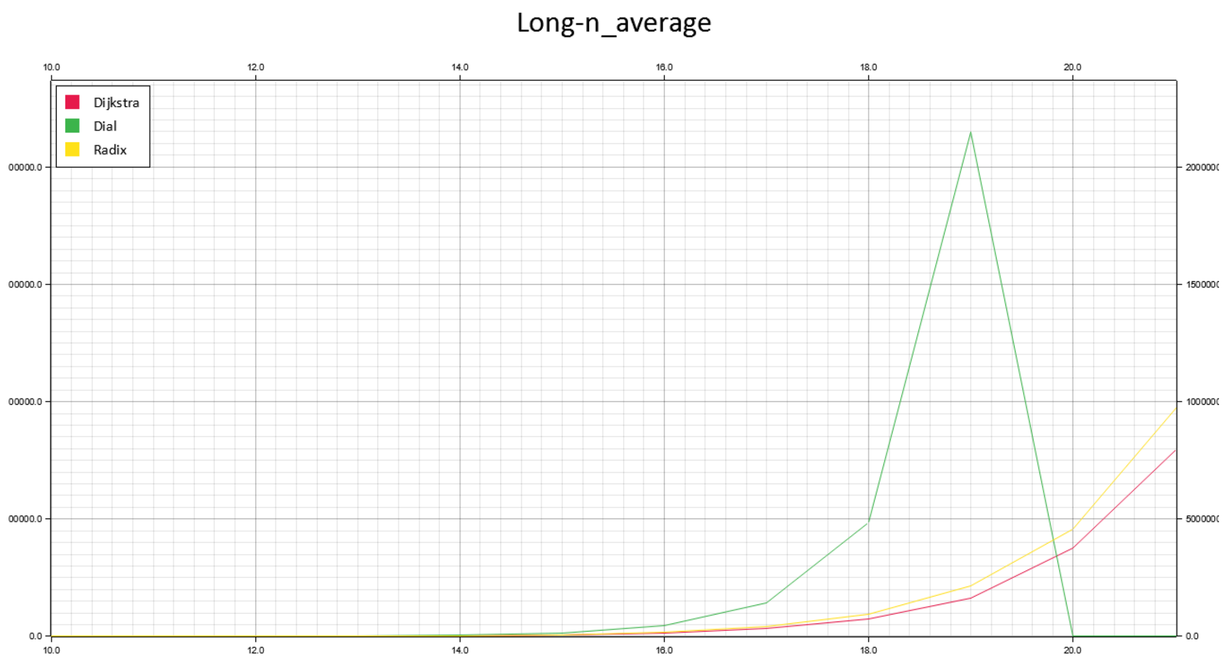


Long-n_average



7.1 Wykresy z obcięciem Diala

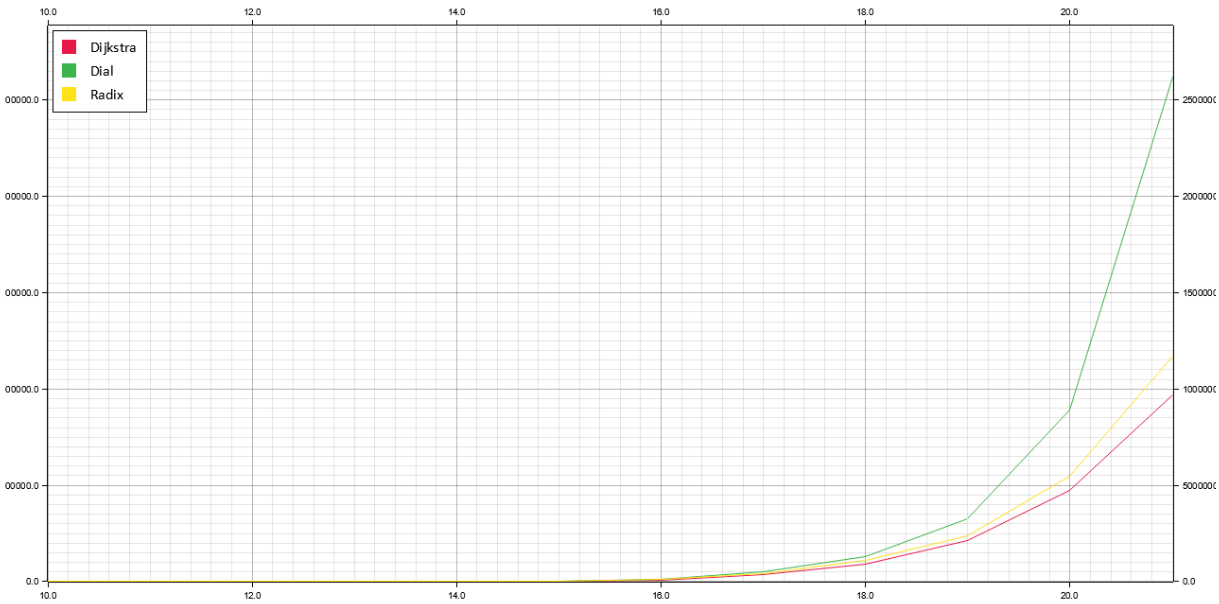




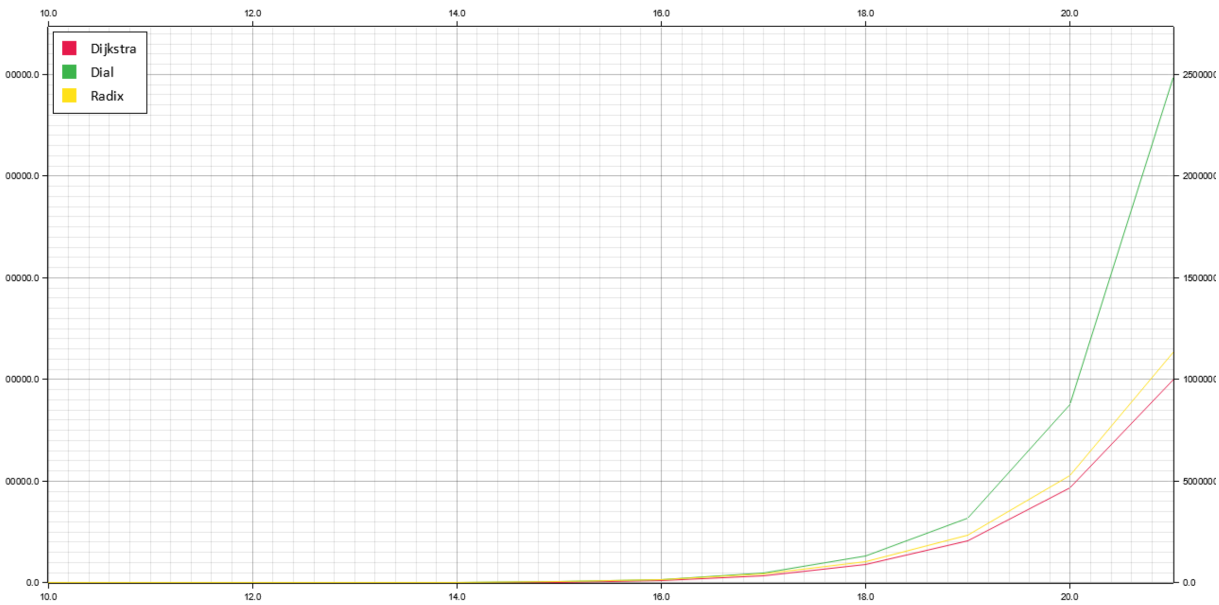
Widzimy tutaj pierwszą osobliwość, mimo tego że dla grafów losowych algorytm Diala był sybszy, to dla długiego grafu kratowego jest on znacznie wolniejszy od pozostałych - zaczyna odbiegać w okolicy $n \approx 2^{15}$. Podobnie jak dla Random-4-c, w późniejszym etapie najszybszy staje się algorytm Dijkstry.

8 Square-n

Square-n

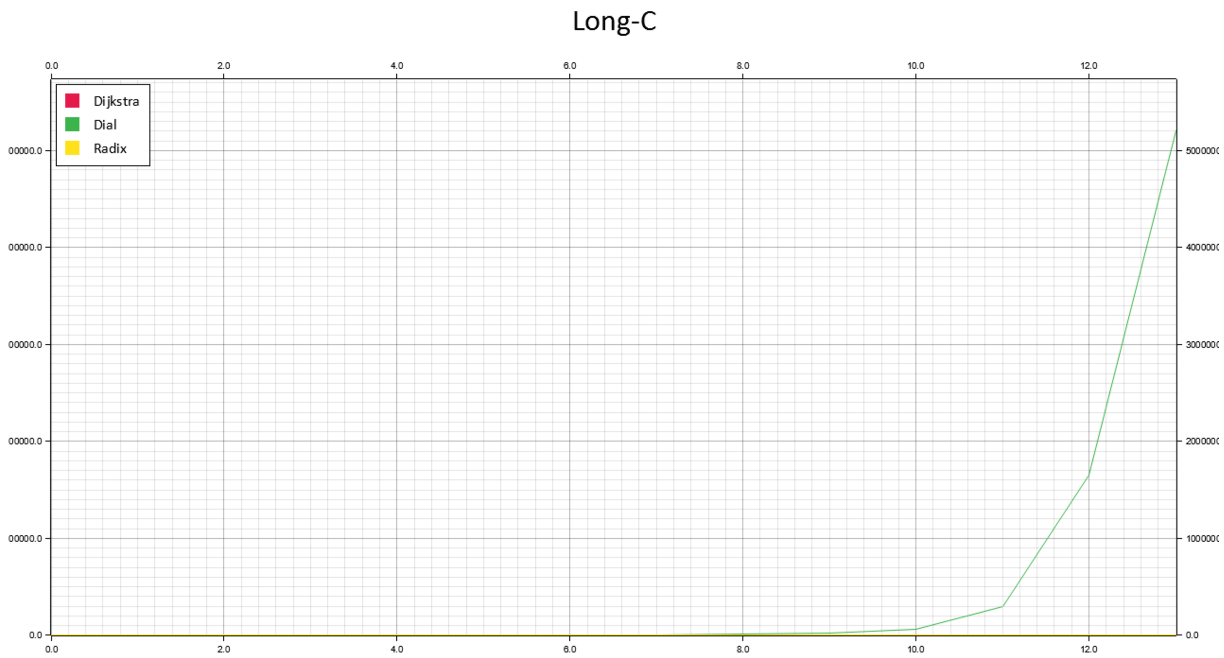


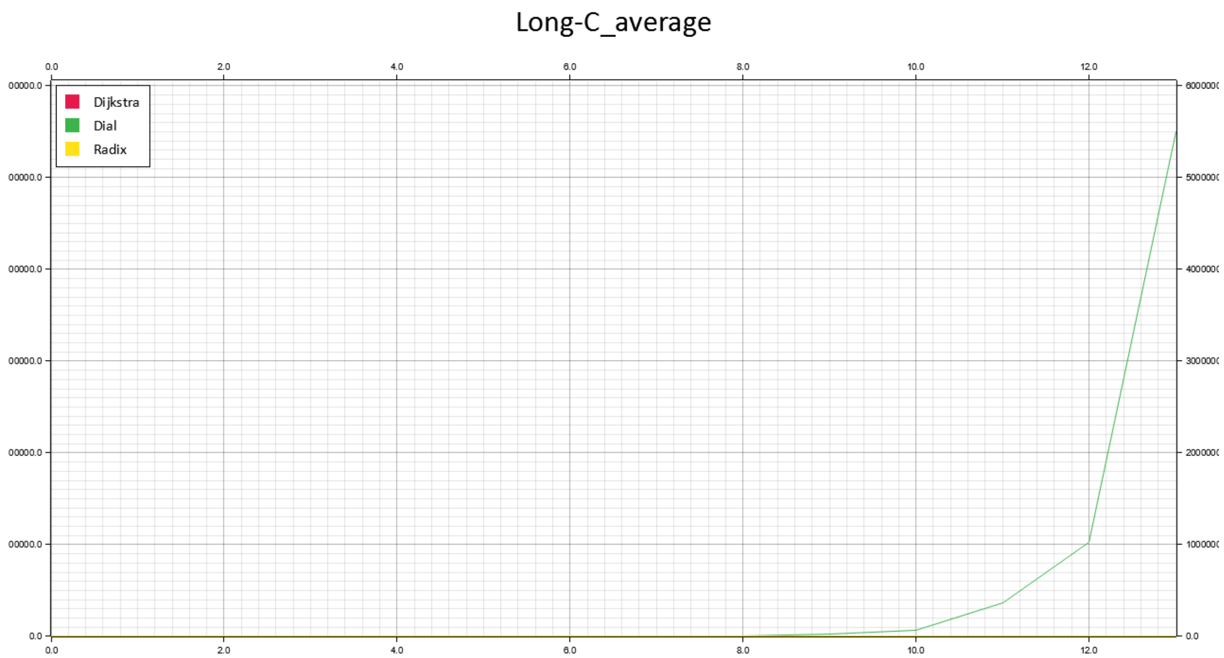
Square-n_average



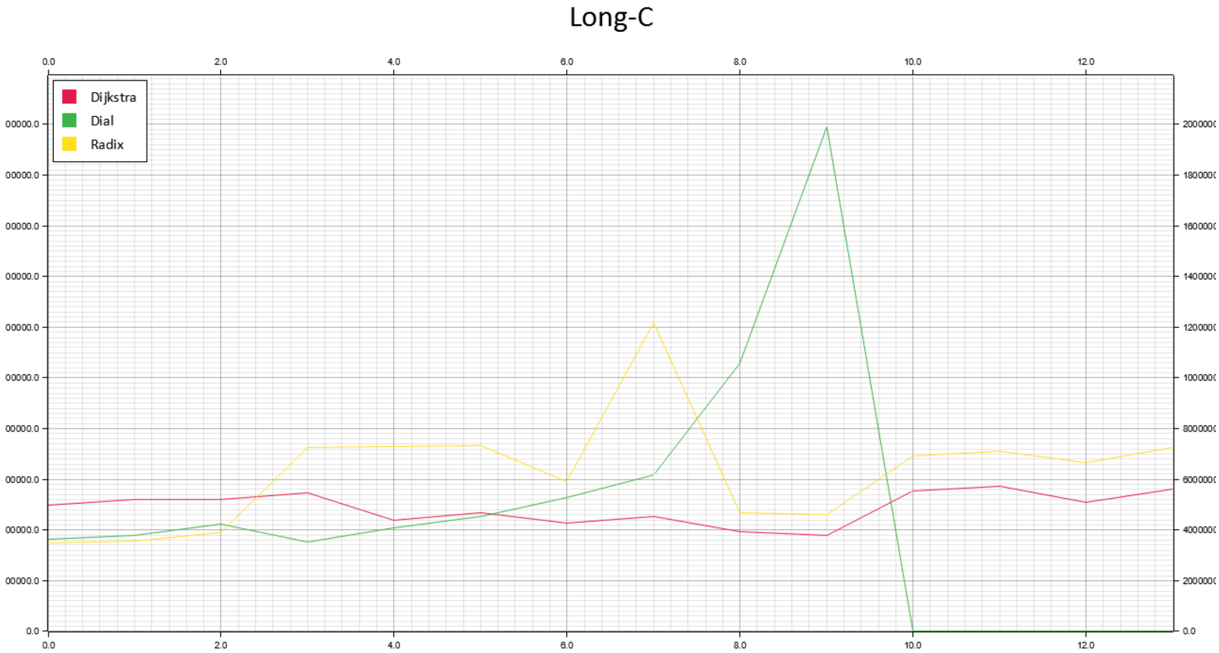
Widzimy tutaj podoby eksponencjalny wzrost tak jak w grafach losowych, jednak dla grafów kratowych algorytm Diala okazuje się być wolniejszy od reszty, chociaż w tym wypadku tylko nieznacznie. Ponownie w dalszej części wykresu najszybszy jest algorytm Dijkstry.

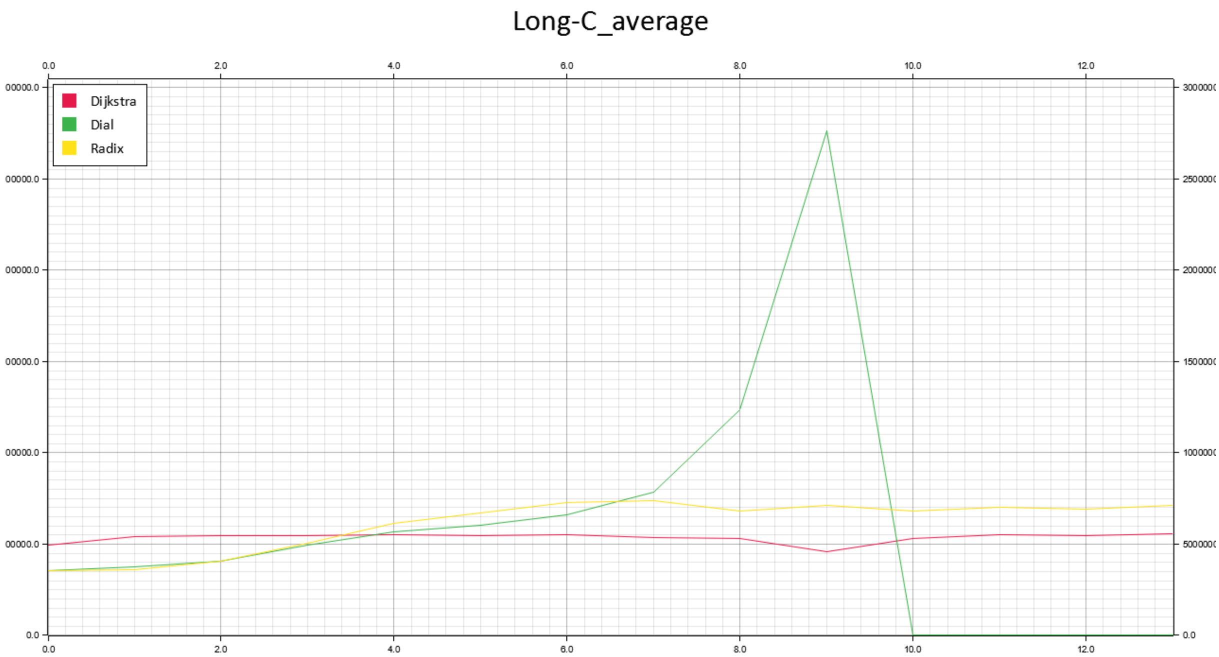
9 Long-C


Long-C
Dijkstra
Dial
Radix


Long-C_average
Dijkstra
Dial
Radix

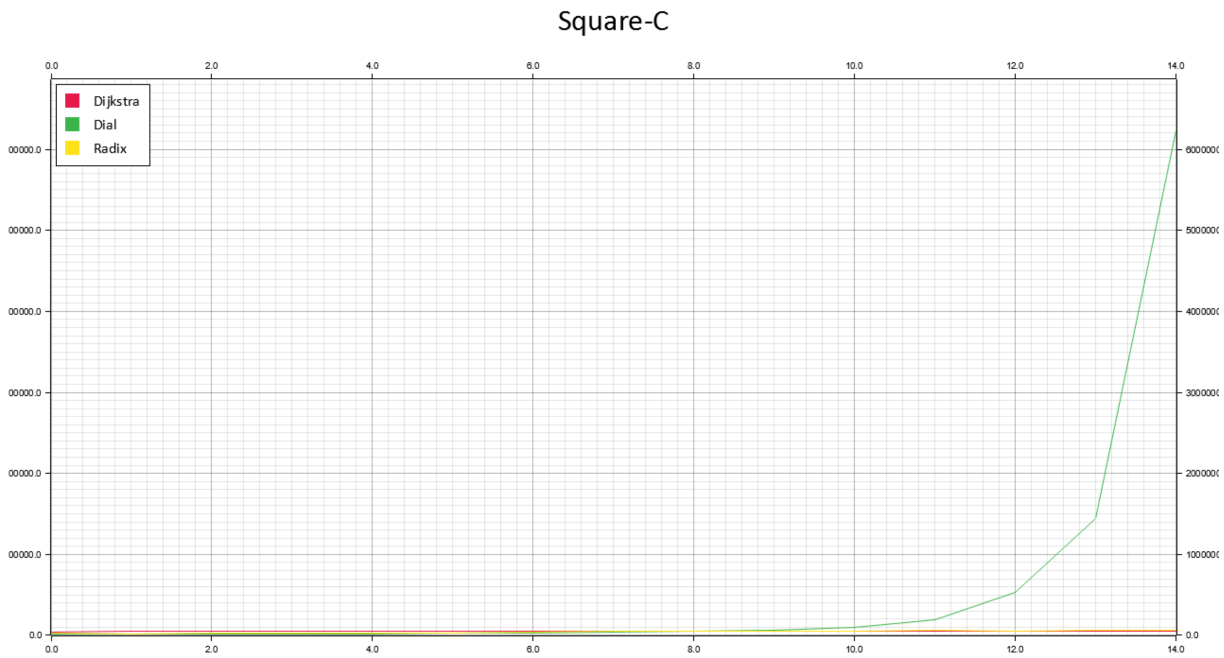
9.1 Wykresy z obcięciem Diala

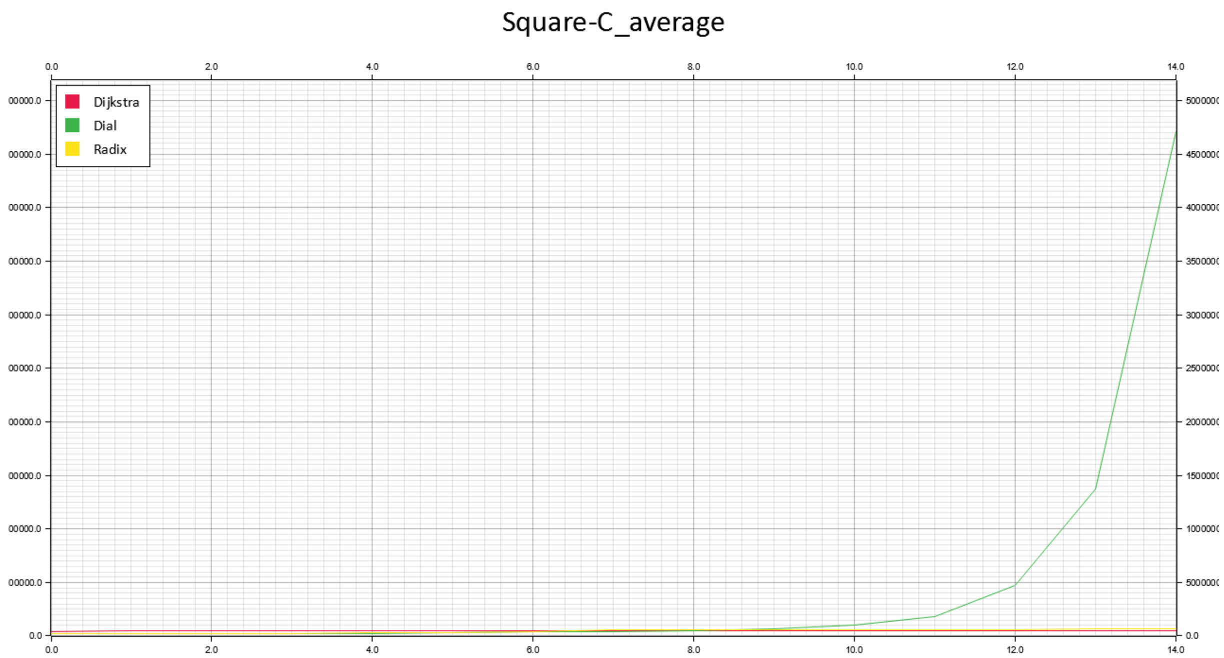




Obserwujemy tutaj podobne zachowanie jak dla Random-4-c, „punkt załamania” jest jednak wcześniej, bo już dla $C \approx 4^4$. Dla małych wartości najszybszy jest Radix Heap, jednak dla dużych najszybszy staje się algorytm Dijkstry.

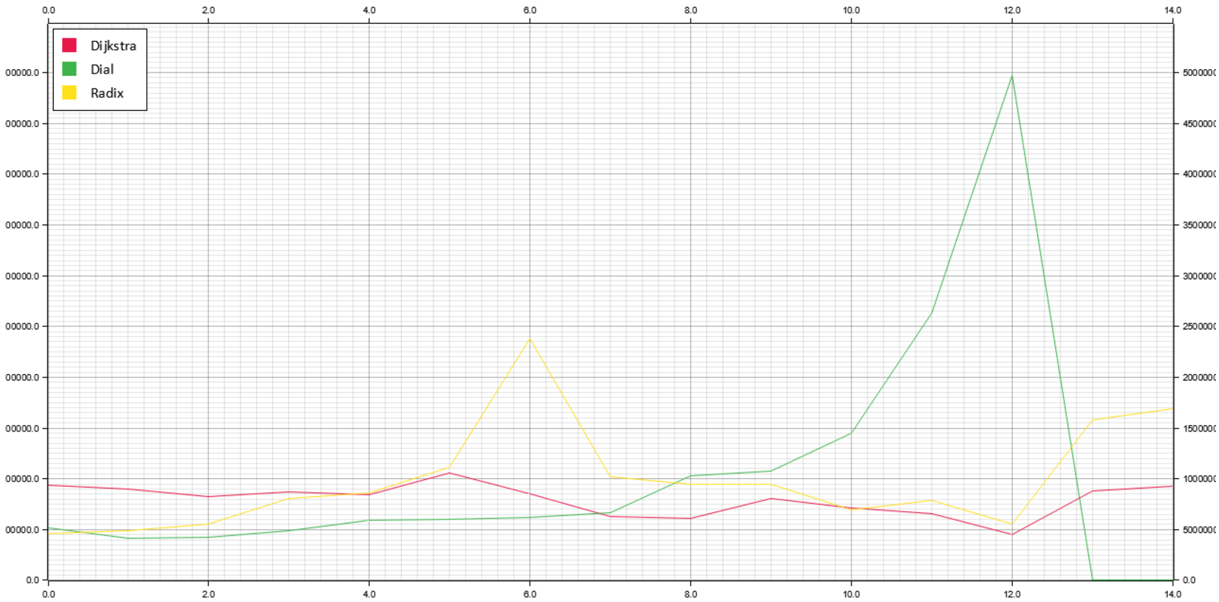
10 Square-C


Square-C
Dijkstra
Dial
Radix

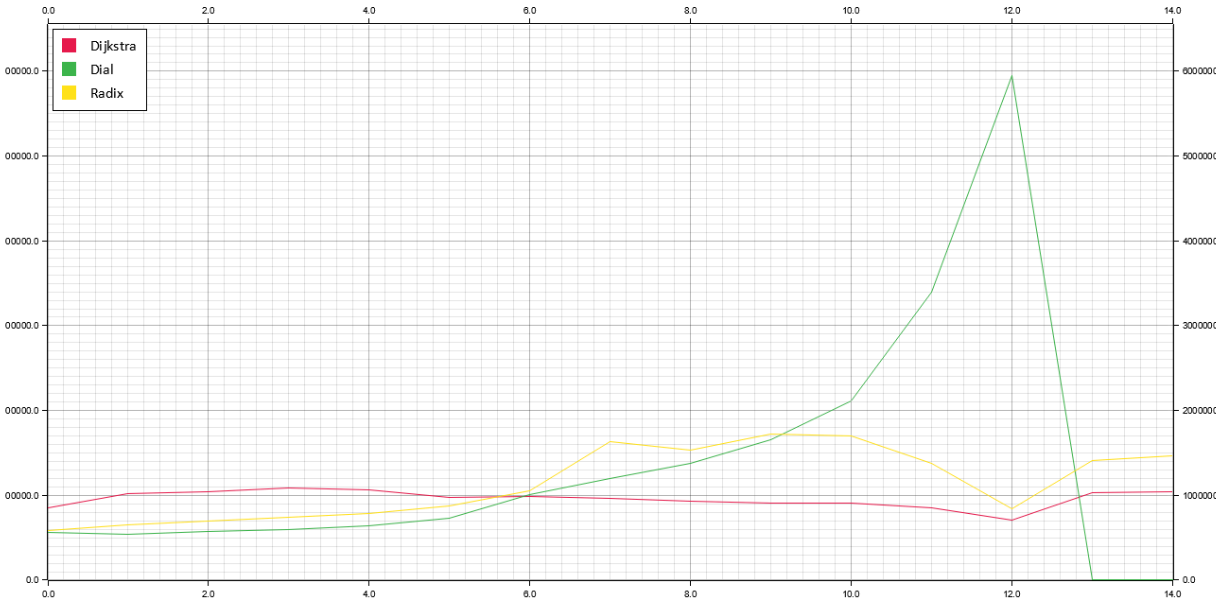

Square-C_average
Dijkstra
Dial
Radix

10.1 Wykresy z obcięciem Diala

Square-C



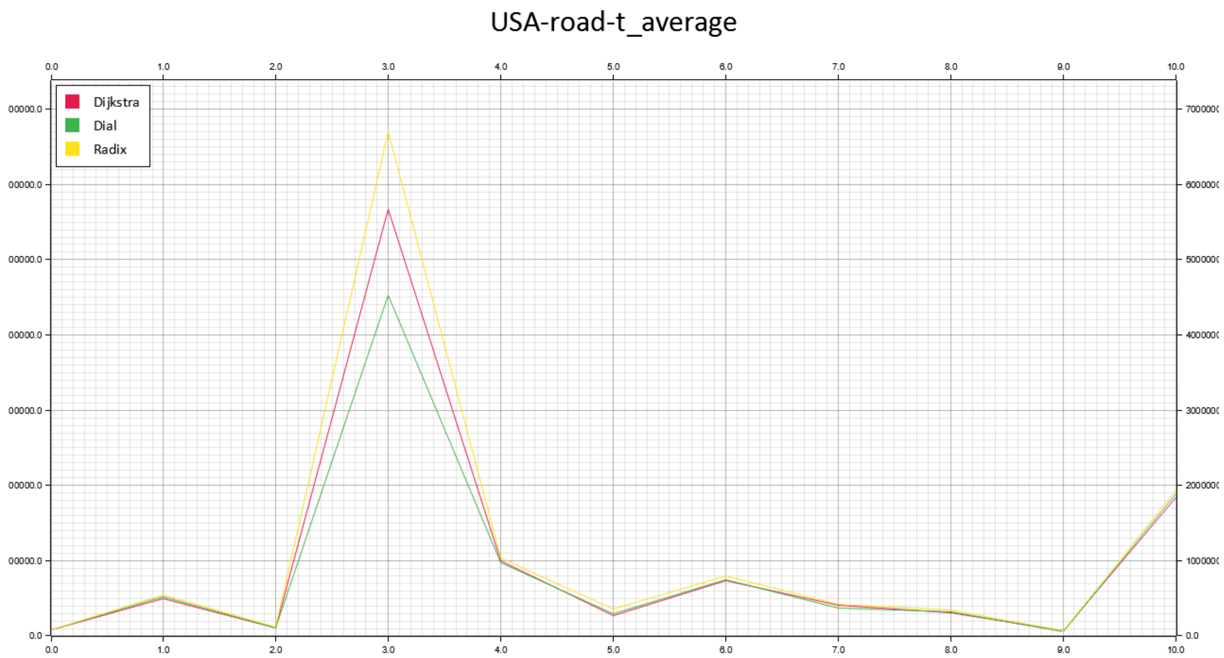
Square-C_average



Ponownie obserwujemy „punkt załamania” taki jak w Random-4-c w tym wypadku występuje on dla $C \approx 4^5$

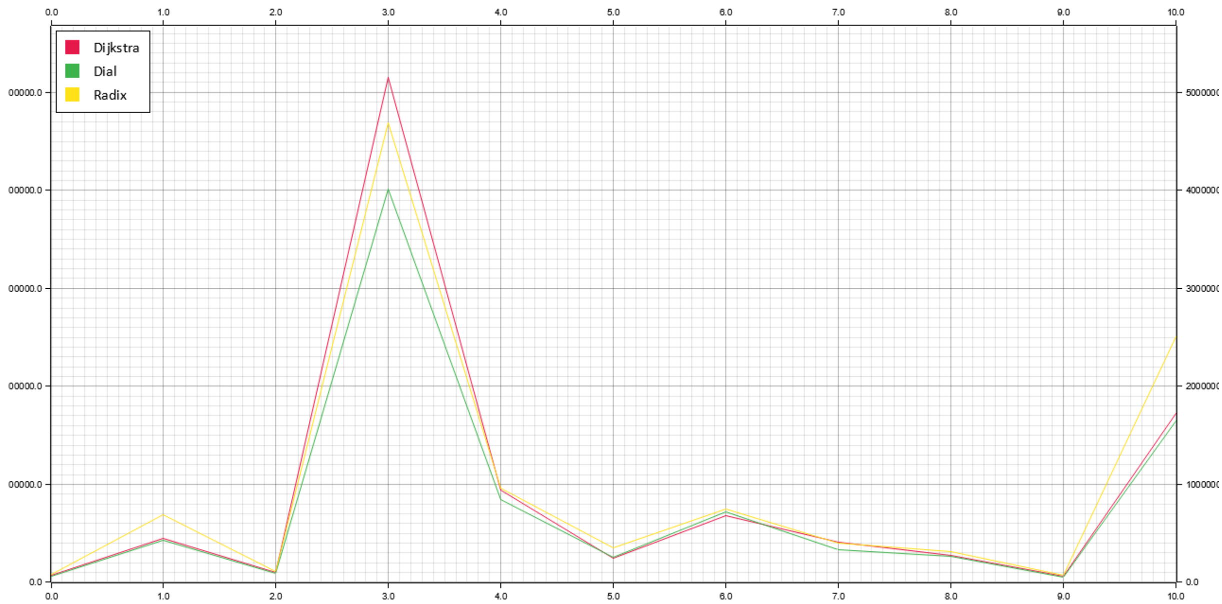
11 USA-road-t


USA-road-t
Dijkstra
Dial
Radix

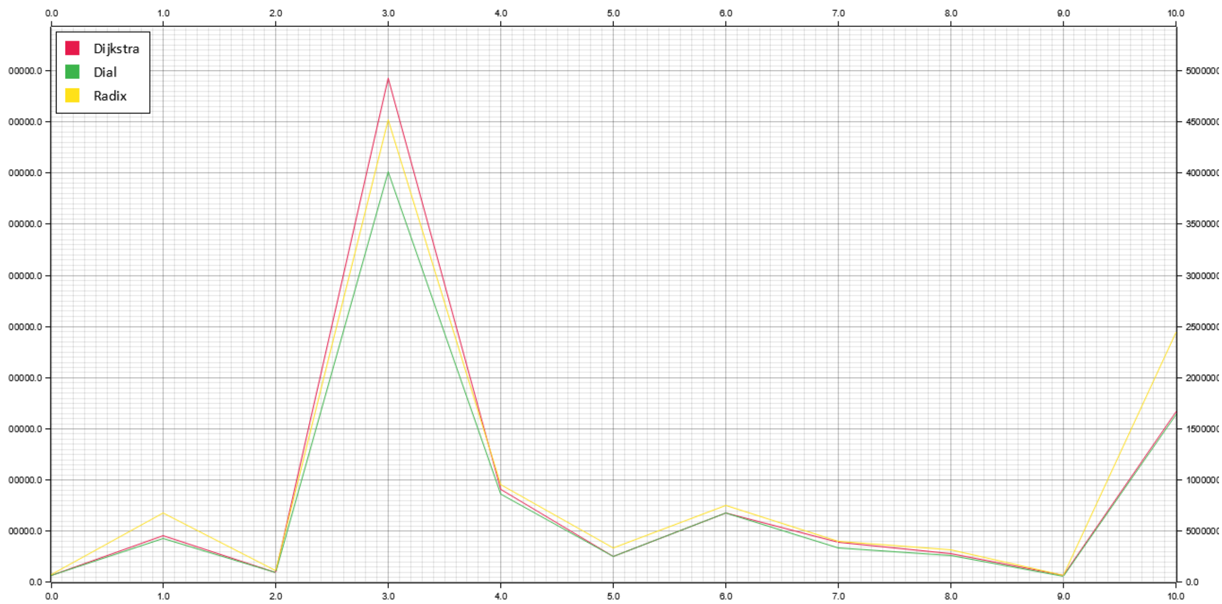

USA-road-t_average
Dijkstra
Dial
Radix

12 USA-road-t

USA_road-d



USA_road-d_average



Widzimy że dla prawdziwych danych, choć nieznacznie, najlepszy jest algorytm Diala.

13 Ścieżki pomiędzy 2 wierzchołkami

Tabela 4: Random-4-n-21

Start	Koniec	Dijkstra	Dial	Radix Heap
164161	1669611	9337637	9337637	9337637
798026	1343221	7517600	7517600	7517600
1408297	1872901	10642970	10642970	10642970
1850062	678699	8847986	8847986	8847986
1664626	1993657	8484659	8484659	8484659
0	2097151	9051281	9051281	9051281

Tabela 5: Random-4-c-15

Start	Koniec	Dijkstra	Dial	Radix Heap
92016	901540	3509491597	3509491597	3509491597
264197	286556	5583518735	5583518735	5583518735
953811	370805	5165186308	5165186308	5165186308
45032	851668	5032881608	5032881608	5032881608
795940	255834	4404331181	4404331181	4404331181
0	1048575	3471241820	3471241820	3471241820

Tabela 6: Long-n-21

Start	Koniec	Dijkstra	Dial	Radix Heap
1450965	1938289	70599140663	70599140663	70599140663
1964550	1755689	51736861927	51736861927	51736861927
545991	2070092	47578185662	47578185662	47578185662
964232	1497990	14716528525	14716528525	14716528525
474314	1031903	9786234471	9786234471	9786234471
0	2097151	31336751771	31336751771	31336751771

Tabela 7: Square-n-21

Start	Koniec	Dijkstra	Dial	Radix Heap
733504	92631	629673961	629673961	629673961
1013081	73307	401159909	401159909	401159909
1610689	1261870	613316462	613316462	613316462
215686	904767	404530081	404530081	404530081
2033025	1778454	509608247	509608247	509608247
0	2096703	714640488	714640488	714640488

Tabela 8: Long-c-13

Start	Koniec	Dijkstra	Dial	Radix Heap
148253	527281	967429490887	967429490887	967429490887
896274	946586	1038614355389	1038614355389	1038614355389
368960	1002167	172933112112	172933112112	172933112112
442052	15922	142662990871	142662990871	142662990871
140013	819944	570603506193	570603506193	570603506193
0	1048575	203423278447	203423278447	203423278447

Tabela 9: Square-C-14

Start	Koniec	Dijkstra	Dial	Radix Heap
871365	406366	53038376661	53038376661	53038376661
841607	765242	40974919449	40974919449	40974919449
721155	882124	68376204755	68376204755	68376204755
487511	614151	49507292282	49507292282	49507292282
100047	7527	18278721752	18278721752	18278721752
0	1048575	89231836050	89231836050	89231836050

Tabela 10: USA-road-t-CTR

Start	Koniec	Dijkstra	Dial	Radix Heap
8367159	5041405	11958012	11958012	11958012
3257417	4414171	1263959	1263959	1263959
7735795	11598095	8371833	8371833	8371833
10669035	13352547	15525453	15525453	15525453
9132943	4725134	18145469	18145469	18145469
0	14081815	9709456	9709456	9709456

Tabela 11: USA-road-t-CTR

Start	Koniec	Dijkstra	Dial	Radix Heap
10459858	3242341	7569652	7569652	7569652
11914261	13728217	4878744	4878744	4878744
5586986	11209586	19081270	19081270	19081270
4501042	976076	19608970	19608970	19608970
12600156	9264085	6521295	6521295	6521295
0	14081815	8352221	8352221	8352221

14 Wnioski

Na postawie eksperymentów widzimy że średnio najszybszy jest algorytm Diala, jednak dla niektórych przypadków, takich jak wysokie C lub niektóre specyficzne rodziny grafów, staje się on ekstremalnie nieefektywny.

Najbardziej regularnie zachowuje się algorytm Dijkstry, zachowując prędkość dla wszystkich zbadanych rodzin grafów.

Algorytm Radix Heap stanowi jakoby środek pomiędzy dwoma powyższymi algorytmami, jest szybki dla większości danych, a patologiczne przypadki spowalniają go jedynie nieznacznie.