Vadym Liss

257264

Zadanie1

Algorytm Dijkstry, opublikowany w 1959 roku i nazwany na cześć jego twórcy, znajduje najkrótsze drzewo ścieżek z pojedynczego węzła źródłowego, budując zestaw węzłów o minimalnej odległości od źródła.

Algorytm Dijkstry jest bardzo podobny do algorytmu Prima dla minimalnego drzewa rozpinającego. Podobnie jak w Prim'ie, generujemy drzewo najkrótszych ścieżek z danym źródłem jako korzeniem. Utrzymujemy dwa zestawy, jeden zestaw zawiera wierzchołki zawarte w drzewie najkrótszej ścieżki, drugi zestaw zawiera wierzchołki, które jeszcze nie zostały uwzględnione w drzewie najkrótszej ścieżki. Na każdym kroku algorytmu znajdujemy wierzchołek, który znajduje się w drugim zbiorze (zestaw jeszcze nieuwzględniony) i ma minimalną odległość od źródła.

Ważne jest, aby zwrócić uwagę na następujące punkty:

- Algorytm Dijkstry działa tylko dla grafów, które nie zawierają żadnych krawędzi o ujemnej wadze.
- Podaje tylko wartość lub koszt najkrótszych ścieżek.
- Algorytm działa dla grafów skierowanych i nieskierowanych.

Algorytm Dijkstry wykorzystuje BFS do rozwiązania opisanego problemu. Jednak w przeciwieństwie do oryginalnego BFS używa kolejki priorytetowej zamiast zwyklej kolejki. Priorytetem każdego wierzchołka jest koszt dotarcia do niego ze źródła.

Aby dowiedzieć się, jak algorytm Dijkstry działa, spójrzmy na poniższe kroki:

1. Przede wszystkim oznaczymy wszystkie wierzchołki jako nieodwiedzone.
2. Następnie oznaczymy wierzchołek źródłowy jako 0, a wszystkie inne wierzchołki jako nieskończoność.
3. Traktujemy źródłowy wierzchołek jako aktualny wierzchołek.
4. Obliczamy długość ścieżki wszystkich sąsiednich wierzchołków z aktualnego wierzchołka, dodając wagę krawędzi w aktualnym wierzchołku.
5. Jeżeli nowa długość ścieżki jest mniejsza niż poprzednia długość ścieżki, zastępujemy ją, w przeciwnym razie zignorujemy ją.
6. Oznaczamy aktualny wierzchołek jako odwiedzony po odwiedzeniu sąsiednich wierzchołków.
7. Wybierzemy kolejny wierzchołek o najmniejszej długości ścieżki jako nowy aktualny wierzchołek i wracamy do kroku 4.
8. Powtarzamy ten proces, aż wszystkie wierzchołki zostaną oznaczone jako odwiedzone.

Złożoność algorytmu Dijkstry wynosi O(V^2), gdzie V jest liczbą wierzchołków grafu. Jednak zauważmy, że transformacja grafu ważonego w nieważony zmniejsza tą złożoność. Jeżeli liczba węzłów i krawędzi transformowanego grafu pozostaje liniowa względem n i m, to możemy uruchomić BFS, aby rozwiązać problem w O(n+m).

Transformację możemy opisać następująco: dla każdej krawędzi (u,v) z wagą w, stwórzmy w−1 węzłów, aby połączyć u,v. Na przykład dla krawędzi (a,b) o długości 3 zajdzie coś takiego jak: $a - d_1 - d_2 - b$. Łatwo sprawdzić, czy ta transformacja zachowuje najkrótszą ścieżkę między dowolnymi parami węzłów. Liczba węzłów wynosi teraz O(n+cm) = O(n+m), a liczba krawędzi to O(m+cm) = O(m). Ponieważ BFS jest liniowy w stosunku do liczby węzłów i krawędzi, mamy czas działania O(n+m+m) = O(n+m).

Pseudokod:

```
Algorithm 1: Dijkstra's algorithm
Input: G = (V, E): the input graph

source: the source node
/* Initialization                                    */
dist[source] ← 0
Initilize dist[v] = 0, for all v ∈ V and v ≠ source
Add v to Q, for all v ∈ V
S ← ∅
/* The main loop for the algorithm                   */
while Q ≠ ∅ do
    v ← vertex in Q with min dist[v]
    remove v from Q
    Add v to S
    foreach neighbour u of v do
        if u ∈ S then
            continue ;       // if u is visited then ignore it
        alt ← dist[v] + weight(v, u)
        if alt < dist[u] then
            dist[u] ← alt ;              // update distance of u
            update Q
    return dist
```

Zadanie2

Algorytm Dijkstry można modyfikować za pomocą innej struktury danych - buckets, co umożliwia lepszą implementacją algorytmu Dijkstry dla przypadków, gdy niejednokrotnie zakres wag na krawędziach jest niewielki. Złożoność czasowa modyfikacji zwanej Dial to O(m + Cn), gdzie C jest maksymalną wagą na dowolnej krawędzi grafu, więc jeżeli C jest małe, to ta implementacja działa znacznie szybciej niż tradycyjny algorytm Dijkstry.

Poniżej kompletny algorytm Dial:

1. Utrzymujemy kilka kubełek ponumerowanych 0, 1, 2,…,Cn.
2. Kubełek k zawiera wszystkie tymczasowo oznaczone węzły o odległości równej k.
3. Węzły w każdym kubełku są reprezentowane przez listę wierzchołków.
4. Kubełki 0, 1, 2,.. Cn są sprawdzane sekwencyjnie, aż do znalezienia pierwszego niepustego kubełka. Każdy węzeł zawarty w pierwszym niepustym kubełku ma z definicji etykietę minimalnej odległości.
5. Jeden po drugim węzły z etykietą minimalnej odległości są trwale oznaczane i usuwane z kubełku podczas procesu przechodzenia.
6. Zatem operacje z udziałem wierzchołków obejmują:
    - Sprawdzanie czy kubełek jest puste.
    - Dodawanie wierzchołka do kubełku .
    - Usuwanie wierzchołka z kubełku.
7. Pozycja tymczasowo oznaczonego wierzchołka w kubełku jest odpowiednio aktualizowana, gdy zmienia się etykieta odległości wierzchołka.
8. Proces powtarzany, aż wszystkie wierzchołki zostaną oznaczone (lub zostaną znalezione odległości wszystkich wierzchołków).

Pseudokod:

```
dial() {
   initDistanceTable(D = [inf,inf,...,inf]);
   D[source] = 0;
   initBucketList(BucketList = [0,1,...,maxCost]);

   while isNotEmpty(BucketList):
      vertex u = vertexWithMinDistance(BucketList);
      delete(List, u);
      for every adjacent vertex v:
         if D[v] > D[u] + edge(u, v) then
            D[v] = D[u] + edge(u, v);
            updateDistance(List, v, D[v]);
         endif
      endfor
   endwhile
}
```

Zadanie3

Algorytm RadixHeap jest taka modyfikacją algorytmu Dijkstry, że ma złożoność O(m + n log C). Co więcej, RadixHeap jest trudniejszy do implementacji niż Dijkstry. Zauważmy, że RadixHeap ma również dwa ograniczenia: (a) wszystkie klucze w kopcu są liczbami całkowitymi oraz (b) nigdy nie można wstawić nowego elementu, który jest mniejszy niż wszystkie inne elementy znajdujące się obecnie na kopcu.
Początkowo złożoność czasowa wyboru minimalnego wierzchołka odległości wynosi O(m), jeśli zastosowana jest naiwna implementacja. Następnie, zastępując kolejką priorytetową, złożoność może się zmniejszyć. Kolejka priorytetowa jest oparta na kopcu, zatem, złożoność czasowa wynosi O(log C) dla każdego wyboru z modyfikacją stosu. Aby jednak wykorzystać kolejkę priorytetową, BucketList nie wystarczy. Krok z ustawieniem nowego wierzchołku oraz updateDistance również musi zostać spełniony. Ta modyfikacja powoduje, że złożoność czasowa tego kroku jest O(n log C) dla jego całego wykonania. Zatem złożoność wynosi O(m + n log C).

Poniżej kompletny algorytm RadixHeap:

1. Zaznacz wszystkie wierzchołki jako nieodwiedzone.
2. Ustaw długość wierzchołka źródłowego na 0 i zaznacz go jako odwiedzony. W przypadku pozostałych wierzchołków odległości próbne są ustawione na ∞.
3. Zaktualizuj odległości próbne nieodwiedzonego wierzchołka przez wierzchołki odwiedzone.
4. Wybierz nieodwiedzony wierzchołek z minimalną odległością i zaznacz go jako odwiedzony.
5. Jeśli wszystkie wierzchołki są oznaczone jako odwiedzone, zakończ ten algorytm. W przeciwnym razie cofnij krok 3.

Pseudokod:

```
radixheap() {
   initDistanceTable(D = [inf,inf,...,inf]);
   D[source] = 0;
   initBucketList(BucketList = [0,1,...,log(maxCost)]);

   while isNotEmpty(BucketList):
      vertex u = vertexWithMinDistance(BucketList);
      updateBucketsLabels(BucketList);
      delete(List, u);
      for every adjacent vertex v:
         if D[v] > D[u] + edge(u, v) then
            D[v] = D[u] + edge(u, v);
            updateDistance(List, v, D[v]);
         endif
      endfor
   endwhile
}
```

Wyniki testów:

Testy przeprowadzone dla **Square-n**.

Tabela reprezentująca wyniki dla flagi SS:

N	Dijkstry	Dial	Radix
14	0.005025	0.004627	0.006262
15	0.016454	0.015452	0.018405
16	0.026605	0.031602	0.028813
17	0.076183	0.108636	0.085488
18	0.127633	0.217350	0.152647
19	0.306006	0.649557	0.317806
20	0.687194	1.874424	0.719624
21	1.715831	6.222203	1.732100

Graficznie:

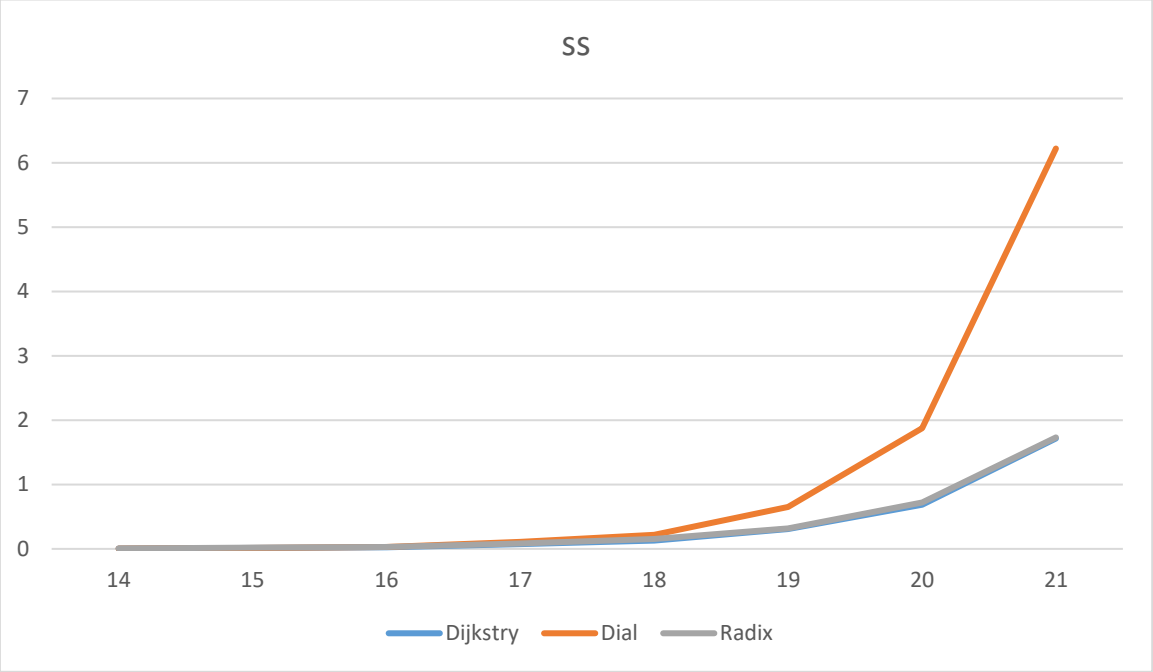
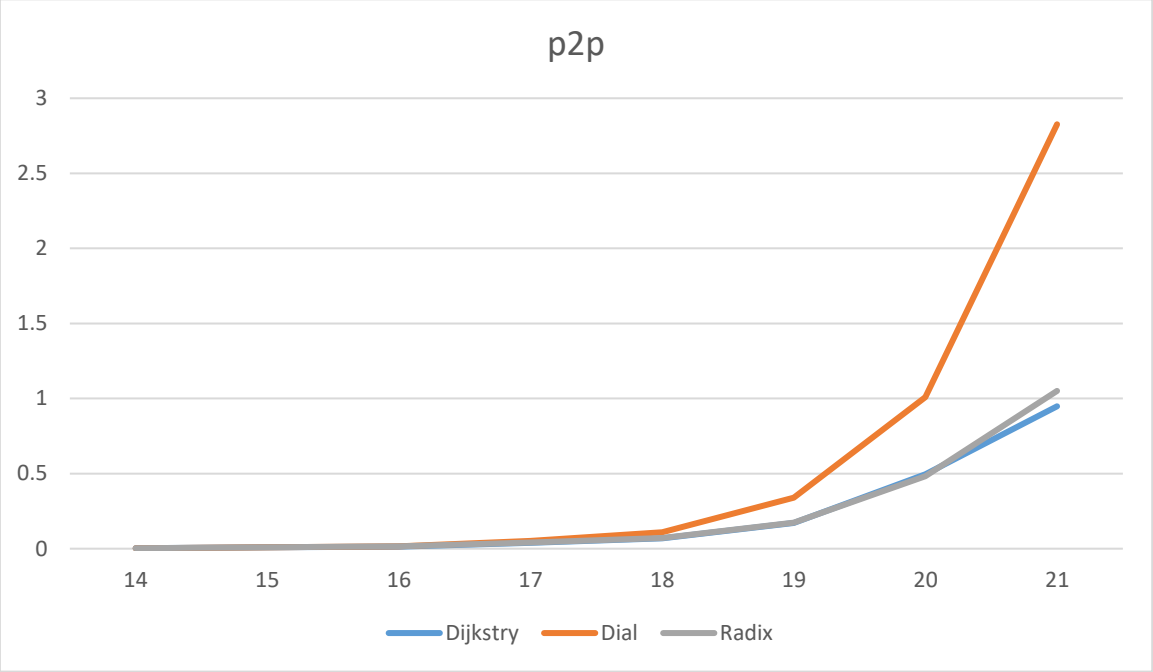


Tabela reprezentująca wyniki dla flagi P2P:

N	Dijkstry	Dial	Radix
14	0.003062	0.002491	0.002946
15	0.008385	0.008276	0.008944
16	0.013795	0.015088	0.014554
17	0.039598	0.051909	0.041801
18	0.069322	0.109870	0.071240
19	0.171402	0.340113	0.173187
20	0.494345	1.008251	0.482685
21	0.948270	2.826552	1.050844

Graficznie:



Testy przeprowadzone dla **Long-n**.
Tabela reprezentująca wyniki dla flagi SS:

N	Dijkstry	Dial	Radix
14	0.003834	0.013863	0.005179
15	0.009042	0.051744	0.010719
16	0.018368	0.198861	0.024026
17	0.041094	0.777681	0.050726
18	0.081544	3.203530	0.107272

Graficznie:

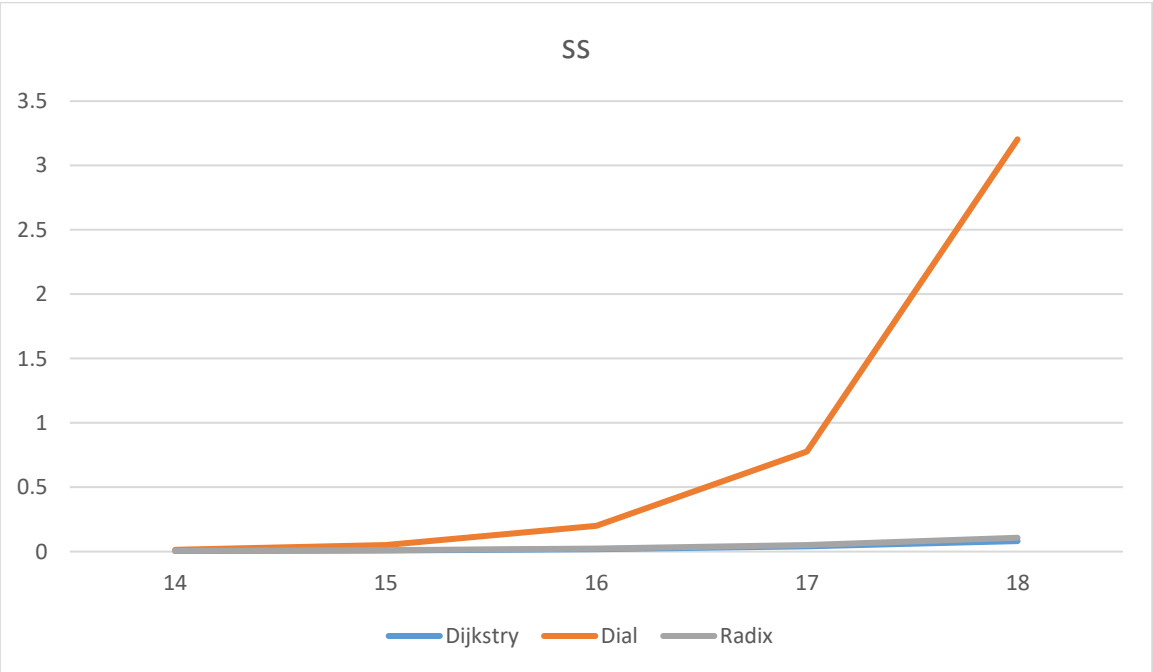
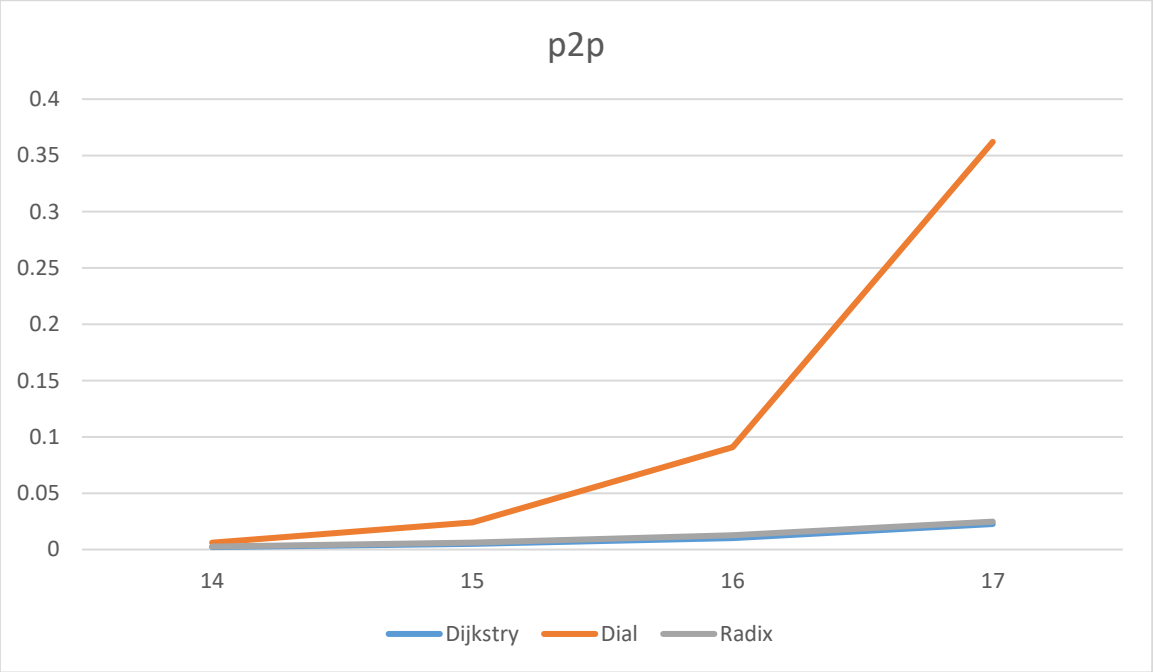


Tabela reprezentująca wyniki dla flagi P2P:

N	Dijkstry	Dial	Radix
14	0.002172	0.006099	0.002625
15	0.005132	0.023987	0.006113
16	0.010273	0.091026	0.012749
17	0.022653	0.362041	0.024790
18	0.046119	1.390598	0.053112

Graficznie:



Testy przeprowadzone dla **Random-n**.
Tabela reprezentująca wyniki dla flagi SS:

N	Dijkstry	Dial	Radix
14	0.014978	0.008233	0.014831
15	0.035226	0.020657	0.036904
16	0.076831	0.054139	0.097795
17	0.232186	0.126653	0.251895
18	0.575801	0.296517	0.609839
19	1.518562	0.654394	1.513205
20	3.355205	1.525965	3.372799

Graficznie:

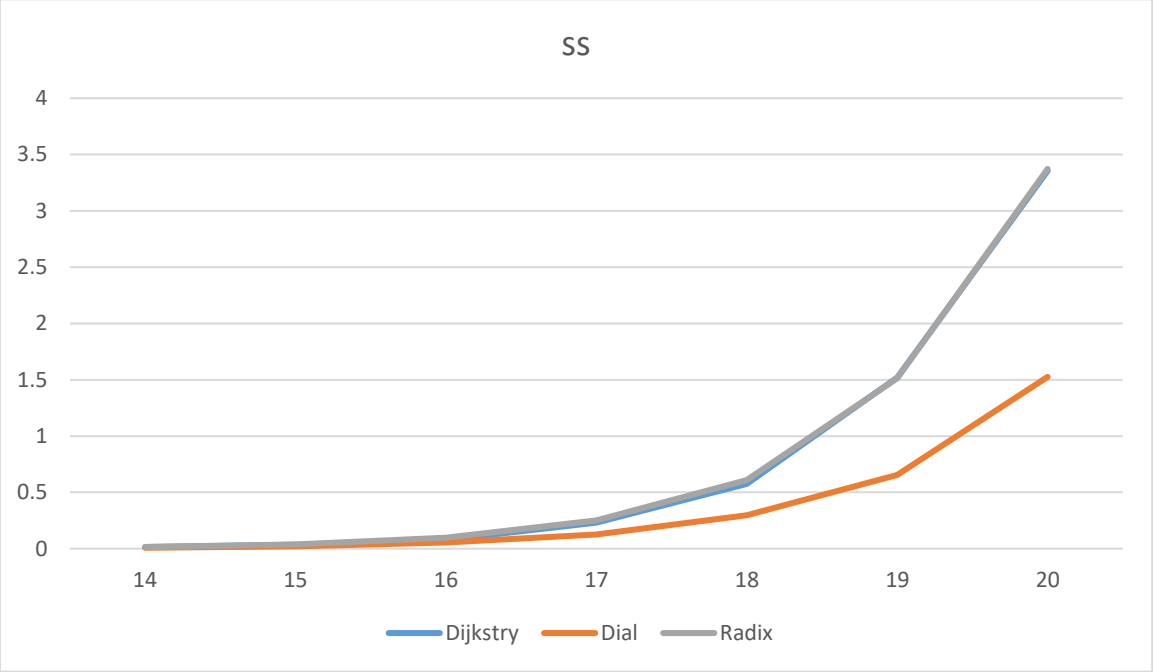
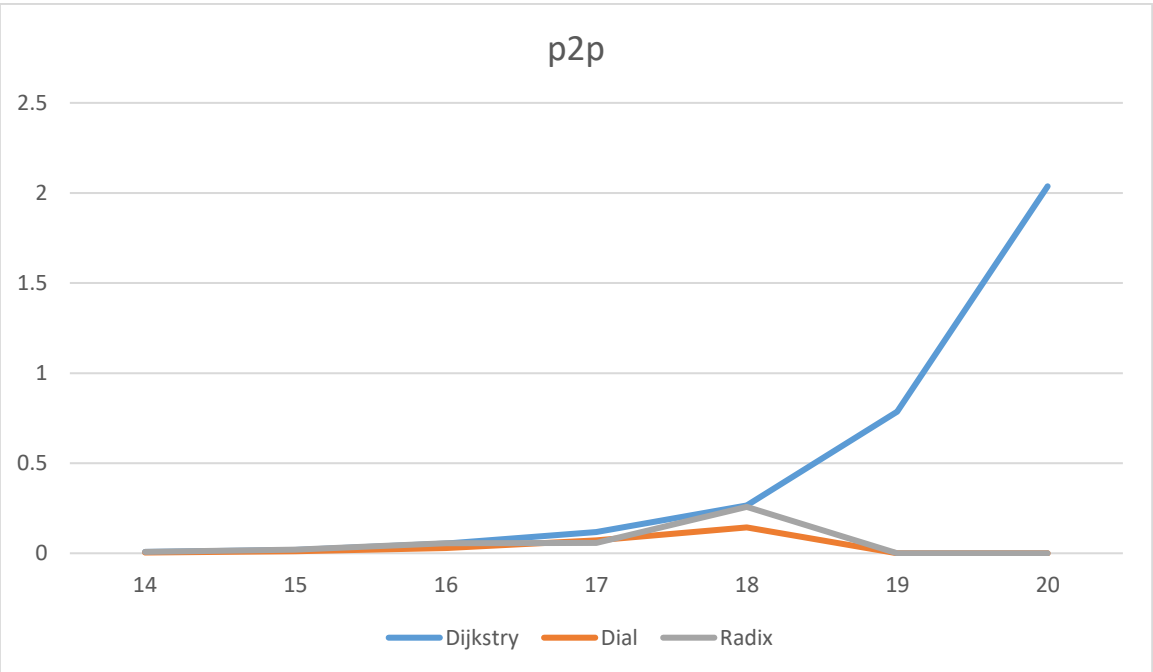


Tabela reprezentująca wyniki dla flagi P2P:

N	Dijkstry	Dial	Radix
14	0.007524	0.003768	0.008395
15	0.018796	0.011241	0.019906
16	0.053457	0.029842	0.055012
17	0.117522	0.071587	0.057522
18	0.264773	0.143617	0.257618
19	0.785172	0	0
20	2.036922	0	0

Graficznie:



Wnioski:

Obserwując wyniki operacje wykonanych przez modyfikacje algorytmu Dijkstry, łatwo wywnioskować to, że sam w sobie algorytm Dijkstry jest efektywny dla pewnych większych n, niż na przykład Dial. Dial działa szybciej dla małych n, ponieważ kolejka z „kubełek" jest efektywna wtedy i tylko wtedy, gdy wagi są tak małe, że C nie będzie nagle wzrastał. Natomiast, dalej chciałbym porównać to jak działa RadixHeap oraz Dijkstry. Na samym początku eksperymentu wydaje się, że RadixHeap nie jest tak wygodny jak Dijkstry, ponieważ złożoność czasowa tej implementacji algorytmu jest naprawdę wielka, natomiast dla większych n RadixHeap zachowuje się tak samo jak i Dijkstry oraz nawet lepiej, ale chciałbym zauważyć, że czas jest prawie taki sami, ponieważ w RadixHeap są złożone operacje na kopcu, co także wymaga dodatkowego czasu. Sam w sobie pomysł na „znalezienie" ścieżek w RadixHeap jest bardziej optymalny niż korzystanie z samej kolejki w Dijkstry. Czasami RadixHeap wyprzedza Dijkstry, czasami czas działania jest taki samy. Moim zdaniem taki wyniki eksperymentu jeszcze wynikają z tego ograniczenia, że nigdy nie można wstawić nowego elementu, który jest mniejszy niż wszystkie inne elementy znajdujące się obecnie na kopcu. Na samy koniec chciałbym zauważyć, że wyniki zależy od postaci wygenerowanych danych. Dla jednego typu danych jest wygodniejszy Dial (np. Radnom), dla innych RadixHeap oraz Dijkstry.