Algorytmy optymalizacji dyskretnej - Lista 3

Wojciech Sęk

16 grudnia 2021

1 Algorytmy

Wszystkie algorytmy rozważają **silnie spójne** grafy $G = (N, A)$, gdzie $|N| = n, |A| = m$ znajdują długości najkrótszych ścieżek od wierzchołka startowego s do każdego $v \in N \setminus \{s\}$. Niech a_{ij} to krawędź od i do j, a c_{ij} to koszt tej krawędzi. Ponadto $C = \max_{(i,j)\in A} c_{ij}$.

1.1 Algorytm Dijkstry

1.1.1 Idea

Mamy zbiór wierzchołków z ustaloną wartością d i początkowo ustawiamy $d[i] = \infty$ dla $i \neq s$, a $d[s] = 0$. Z kolejki priorytetowej (priorytetem jest najmniejsza wartość d) ściągamy po kolei każdy wierzchołek i rozważamy jego sąsiadów. Jeżeli koszt dojścia do danego wierzchołka i dodany koszt dojścia z niego do jego sąsiada jest mniejszy od dotychczasowego dojścia do jego sąsiada, to aktualizujemy wartość d sąsiada. Robimy to dopóki kolejka jest niepusta.

1.1.2 Implementacja

```
function DIJKSTRA(G, s)
    H ← pusta kolejka priorytetowa
    d ← []
    for i ∈ N \ {s} do
        d[i] ← ∞
    end for
    d[s] ← 0
    H.insert(s)
    while H ≠ ∅ do
        i ← H.extractMin()
        for a_ij ∈ A do
            val ← d[i] + c_ij
            if d[j] > val then
                if d[j] = ∞ then
                    d[j] ← val
                    H.insert(j)
                else
                    d[j] ← val
                    H.decreaseKey(j, val)
                end if
            end if
        end for
    end while
    return d
end function
```

1.1.3 Złożoność

Przyjmijmy, że kolejka priorytetowa jest implementowana przez kopiec binarny. Wtedy:

- złożoność *insert* wynosi $O(\log k)$, gdzie k to aktualny rozmiar kolejki, zatem złożoność wszystkich operacji insert wynosi $O(n \log n)$,
- złożoność *extractMin* wynosi $O(\log k)$, gdzie k to aktualny rozmiar kolejki, bo ściągamy wartość ze kolejki i wykonujemy rekurencyjnie $heapify$ co najwyżej $\log k$ razy. W szczególności jest to operacja $O(\log n)$, bo $k < n$.
- złożoność *decreaseKey* wynosi $O(\log k)$, gdzie k to aktualny rozmiar kolejki, bo zamieniamy wartość i zamieniamy wartość z rodzicem aż będzie zachowana własność kolejki priorytetowej. W szczególności jest to operacja $O(\log n)$, bo $k < n$.

Zauważmy, że operacja $extractMin$ jest wykonywana dokładnie n razy. Operacja $decreaseKey$ jest wykonywana co najwyżej m razy (jest wykonywana lub nie przy każdej krawędzi). Zatem sumaryczna złożoność algorytmu:

$$O(n \log n) + nO(\log n) + mO(\log n) = O((n + m) \log n)$$

1.2 Algorytm Diala

1.2.1 Idea

Mamy dane jak w algorytmie Dijkstry. Wcześniej, wszystkie wierzchołki przechowywaliśmy w jednej kolejce priorytetowej. Możemy natomiast mieć $C+1$ kubełków. Na początku obiegu głównej pętli *curr* (indeksujemy od 0) w kubełkach $content[i], i \geqslant (curr \mod (C+1))$ przechowujemy wierzchołki o $d = \frac{curr}{C+1} \cdot (C+1) + i$, a w mniejszych indeksach o $d = (\frac{curr}{C+1} + 1) \cdot (C+1) + i$. Przetwarzając kubełki po kolei i aktualizując wartości nie doprowadzamy do konfliktów, ponieważ kubełek nie może mieć wpływu na kubełek dalszy niż C kroków. Kubełki nie przechowują wierzchołków o $d = \infty$.

1.3 Implementacja

function DIAL(G, s)
  $content \leftarrow$ tablica list wierzchołków rozmiaru $C + 1$
  $d \leftarrow []$
  for $i \in N \setminus \{s\}$ **do**
    $d[i] \leftarrow \infty$
  end for
  $d[s] \leftarrow 0$
  $content[0].insert(s)$
  $S \leftarrow 1, k \leftarrow 0, curr \leftarrow 0$
  while $S < n$ **do**
    while $conent[k] \neq \emptyset$ **do**
      $i \leftarrow content[k].extract()$
      $S \leftarrow S + 1$
      for $a_{ij} \in A$ **do**
        $old \leftarrow d[j]$, $new \leftarrow curr + c_{ij}$
        if $new < old$ **then**
          if $old < \infty$ **then**
            $content[old \mod (C+1)].remove(j)$
          end if
          $content[new \mod (C+1)].insert(j)$
          $d[b] \leftarrow new$
        end if
      end for
    end while
    $curr \leftarrow curr + 1$
    $k \leftarrow curr \mod (C+1)$
  end while
  return d
end function

1.3.1 Złożoność

Niech listy to listy dwukierunkowe z wartownikiem. Wtedy:

- złożoność *insert* wynosi $O(1)$,
- złożoność *extract* wynosi $O(1)$,
- złożoność *remove* wynosi $O(1)$, jeśli przechowujemy wskaźniki do węzłów w dodatkowej tablicy.

Złożoność inicalizacji tabeli d wynosi $O(n)$. Następnie będziemy wykonywać co najwyżej m razy operację usunięcia z kubełka i wstawienia do nowego, i ponieważ każdy kubełek będziemy przeglądać co najwyżej n razy (więcej nie bo maksymalny koszt jest nC) i mamy $C+1$ kubełków, to przejrzymy $O(nC)$ kubełków. Sumarycznie:

$$O(n) + O(m) + O(nC) = O(m + nC)$$

1.4 Algorytm Radix Heap

1.4.1 Idea

Zmodyfikujmy algorytm Diala. Teraz mamy $K = \lceil \log(nC) \rceil$ kubełków o szerokościach $1, 1, 2, 4, 8 \ldots$, gdzie szerokość to moc zbioru wartości d przyjmowanych przez wierzchołki w danym kubełku. Szukamy pierwszego niepustego kubełka. Jeżeli jego szerokość jest równa 1 to rozważamy wszystkie wierzchołki z niego i updatujemy pozostałe wartości d. Jeśli jest większa od 1, to przerzucamy wierzchołki z tego kubełka do poprzednich kubełków, zachowując kolejność zakresów w kubełkach. Można tak, bo $2^k = 2^{k-1} + 2^{k-2} + \cdots + 1 + 1$.

1.5 Implementacja

function RadixHeap(G, s)
  $K \leftarrow \lceil \log(nC) \rceil$
  $d \leftarrow []$
  $bucket \leftarrow []$
  $content \leftarrow$ tablica list wierzchołków rozmiaru $K + 1$
  for $i \in N \setminus \{s\}$ **do**
    $d[i] \leftarrow \infty$
    $bucket[i] = K$
  end for
  $d[s] \leftarrow 0$
  $bucket[s] = 0$
  $content[0].insert(s)$
  $S \leftarrow 1$
  $k \leftarrow 0$
  $rangeBegin \leftarrow [0, 1, 2, 4, 8, \ldots, 2^{K-1}]$
  $x \leftarrow 1$
  while $S < n$ **do**
    $k \leftarrow 0$
    while $content[k] = \emptyset$ **do**
      $k \leftarrow k + 1$
    end while
    if $k \leqslant 1$ **then**
      $i \leftarrow content[k].extract()$
      $S \leftarrow S + 1$
      for $a_{ij} \in A$ **do**
        $old \leftarrow d[j]$
        $new \leftarrow rangeBegin[k] + c_{ij}$
        if $new < old$ **then**
          $oldBucket \leftarrow bucket[j]$
          if $old < \infty$ **then**

```
                content[oldBucket].remove(j)
            end if
            newBucket ← oldBucket
            while rangeBegin[newBucket] > new do
                newBucket ← newBucket − 1
            end while
            content[newBucket].insert(j)
            d[b] ← new
            bucket[b] ← newBucket
        end if
    end for
else
    minVal ← ∞
    for i ∈ content[k] do
        minVal ← min(minVal, d[i])
    end for
    rangeBegin[0] ← minVal
    x ← 1
    for i ← 1 to k do
        rangeBegin[i] ← minVal + x
        x ← 2 · x
    end for
    while content[k] ≠ ∅ do
        i ← content[k].extract()
        newBucket ← k − 1
        while rangeBegin[newBucket] > d[i] do
            newBucket ← newBucket − 1
        end while
        bucket[i] ← newBucket
        content[newBucket] ← i
    end while
end if
end while
return d
end function
```

1.5.1 Złożoność

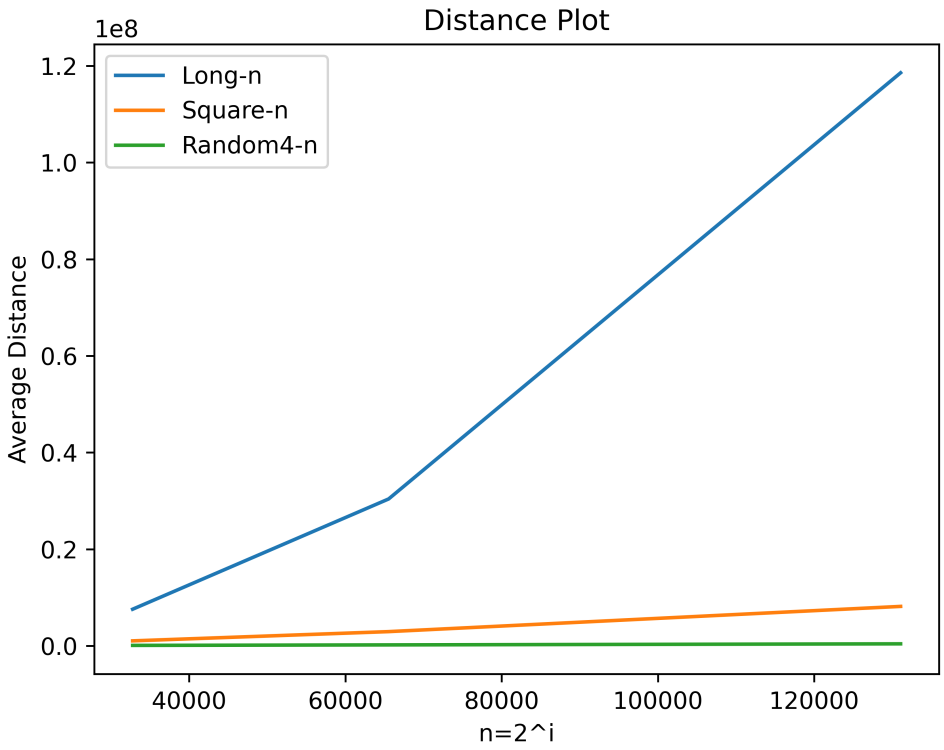
Złożoności operacji na listach dwukierunkowych z wartownikiem tak jak wyżej.
Złożoność inicjalizacji tabeli d wynosi $O(n)$. Każdy węzeł może być przeniesiony co najwyżej K razy i mamy n węzłów. Wybieranie węzła, czyli przeglądanie każdego kubełka aż do znalezienia pierwszego niepustego robimy ze złożonością $O(K)$, robimy tak n razy. Rozważamy każdą krawędź, ze złożonością $O(1)$ przenosimy wierzchołek z kubełka do innego kubełka, jeśli jest taka potrzeba. Całkowita złożoność algorytmu wynosi:

$$nO(K) + nO(K) + O(m) = O(m + nK)$$

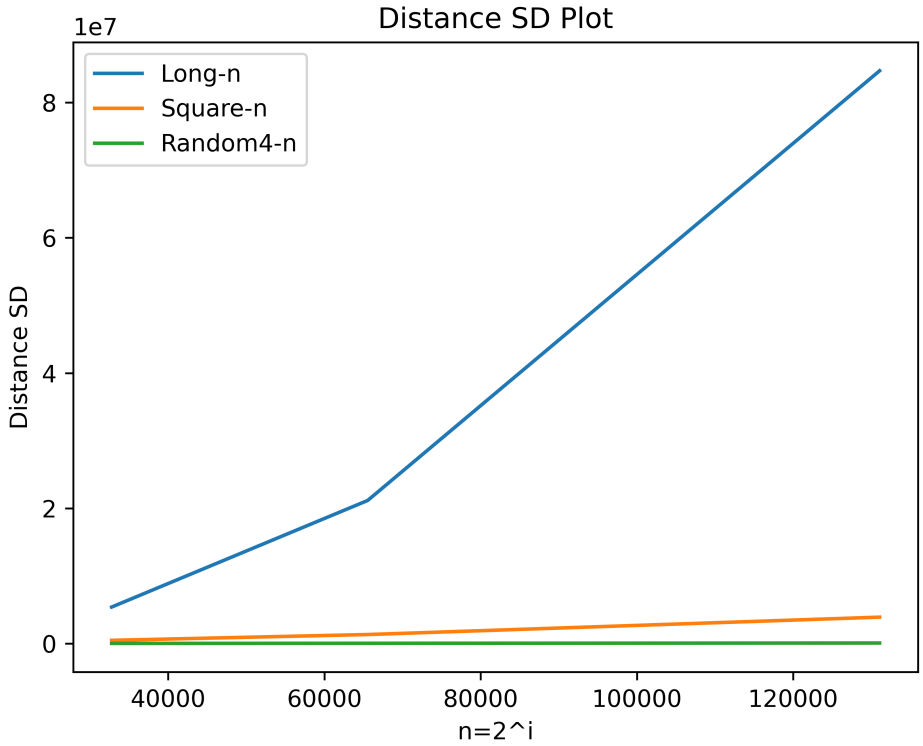
2 Wyniki eksperymentów

Dla testów czasowych rozważamy $n = 2^i$ dla $i \in \{10, 11, \ldots, 17\}$, a dla sprawdzania średnich odległości sprawdzamy rozważamy $n = 2^i$ dla $i \in \{14, 15, 16, 17\}$. Rozważamy 3 rodziny silnie spójnych grafów o $C = n$ (*Square* ma $n' \approx n$ wierzchołków):

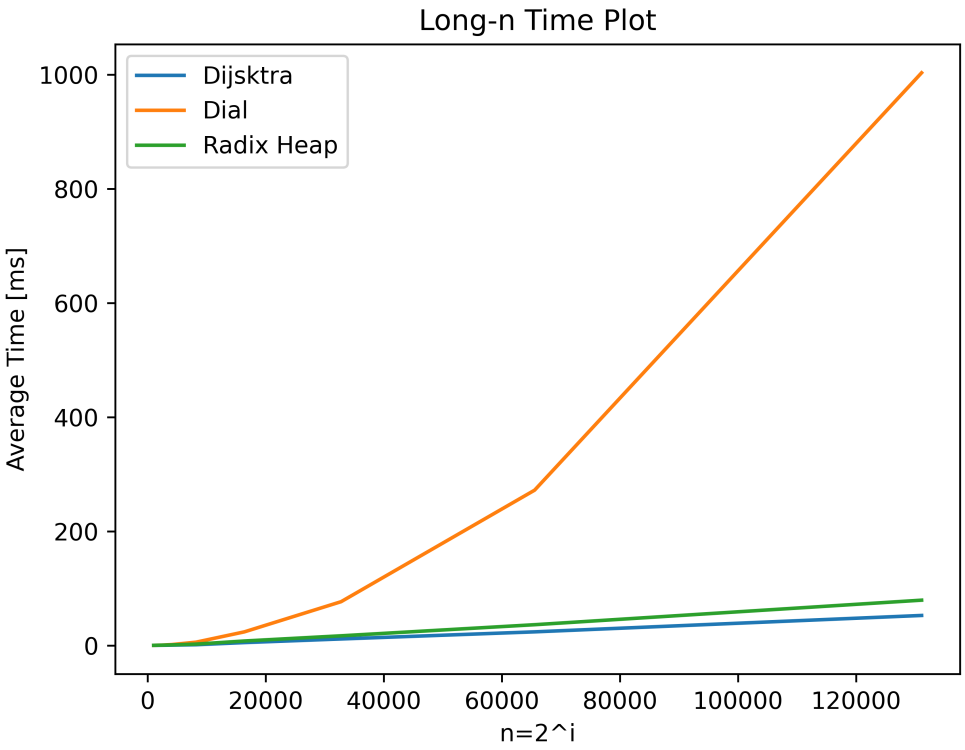
- *Long* - wierzchołki pochodzą z prostokątnej siatki o bokach $x = \frac{2^i}{16} = 2^{i-4}$ i $y = 16$, zatem $m \approx 4n$
- *Square* - wierzchołki pochodzą z prostokątnej siatki o bokach $x = \lfloor\sqrt{n}\rfloor$ i $y = \lfloor\frac{n}{x}\rfloor$, zatem $m \approx 4n$ i $n' = xy \approx n$
- *Random*4 - $m = 4n$, krawędzie są losowe z zachowaniem silnej spójności.



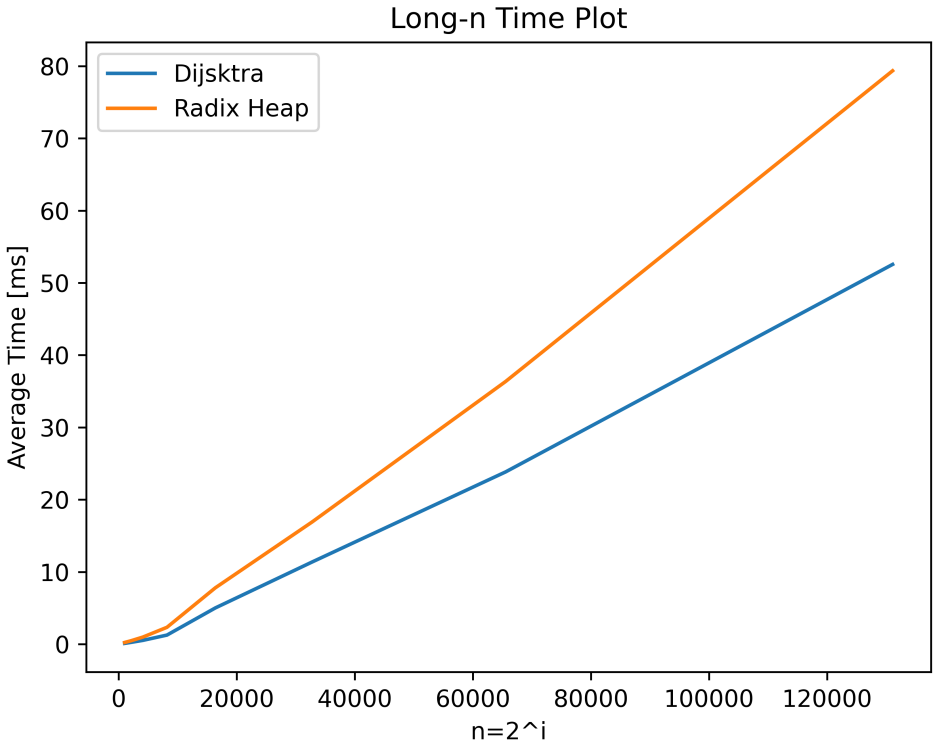
Rysunek 1: Średnie odległości w badanych rodzinach



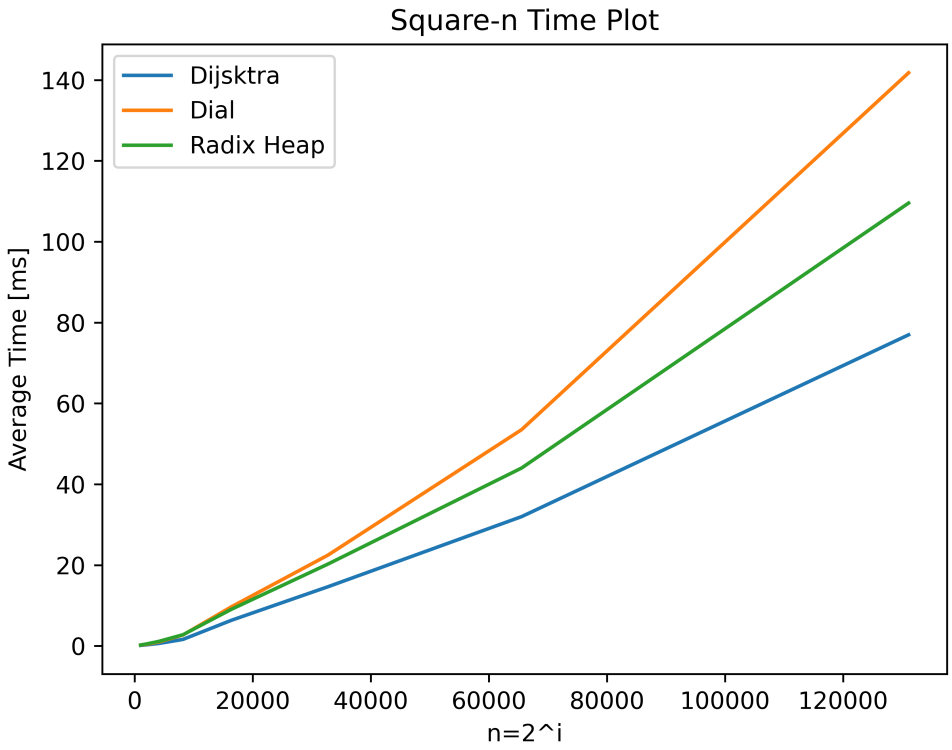
Rysunek 2: Odchylenia standardowe odległości między punktami w badanych rodzinach



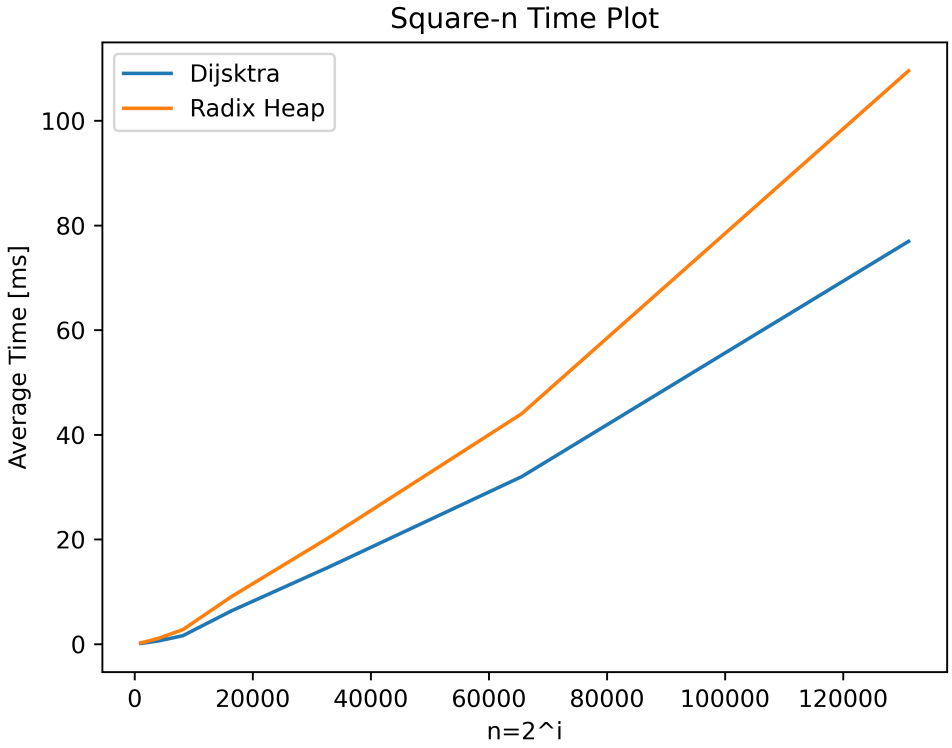
Rysunek 3: Czas działania algorytmów w rodzinie *Long*



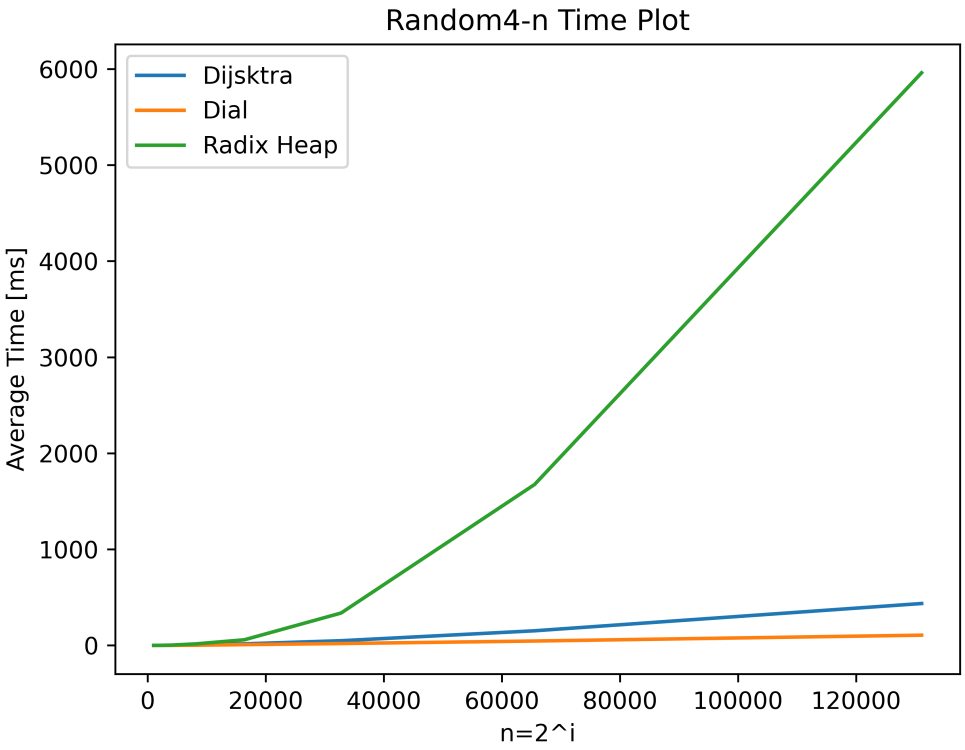
Rysunek 4: Czas działania Dijkstry i Radix Heap w rodzinie *Long*



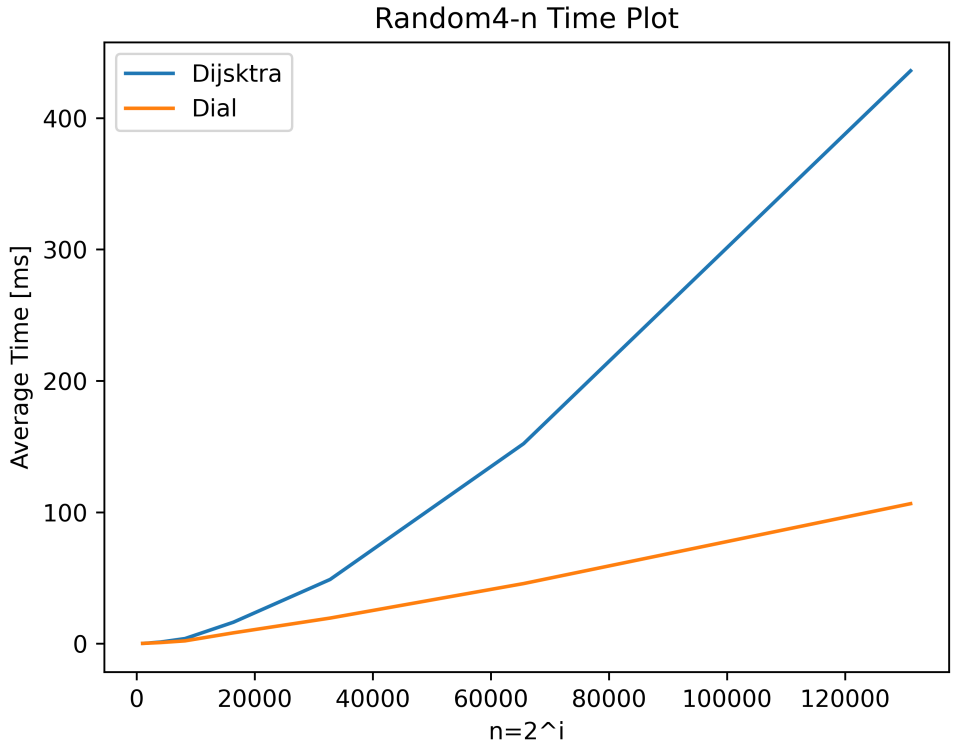
Rysunek 5: Czas działania algorytmów w rodzinie *Square*



Rysunek 6: Czas działania Dijkstry i Radix Heap w rodzinie *Square*



Rysunek 7: Czas działania algorytmów w rodzinie *Random4*



Rysunek 8: Czas działania Dijkstry i Diala w rodzinie *Random4*

3 Intepretacja i wnioski

3.1 Odległości w grafach

Oczywiście, największe średnie odległości między wierzchołkami są w rodzinie *Long*, ponieważ, gdy losujemy dwa punkty z tej rodziny, to mogą one się różnić o 2^{i-4} na współrzędnej x a na y o 16, podczas gdy w kwadracie tylko o $2^{\frac{i}{2}}$ na obu, a najkrótsza ścieżka musi przebiegać przez co najmniej tyle krawędzi ile wynosi odległość w metryce Manhattan między punktami. Co więcej, w rodzinie *Random* odległości są najmniejsze, ponieważ połączenia są losowe i jest małe prawdopodobieństwo, że dwa wierzchołki będą od siebie odległe o wiele krawędzi. Ponadto, odchylenia standardowe też są większe dla rodzin o większych średnich odległościach.

3.2 Czas działa algorytmów w rodzinach

Zauważmy, że w każdej rodzinie $m \approx 4n = O(n)$ i $C = n = O(n)$, więc teoretyczna asymptotyczna złożoność czasowa zadanych algorytmów wynosi:

- dla algorytmu Dijkstry: $O((m+n)\log n) = O(2n \log n) = O(n \log n)$
- dla algorytmu Diala: $O(m + nC) = O(n + n \cdot n) = O(n^2)$
- dla algorytmu Radix Heap: $O(m + nK) = O(n + n \log Cn) = O(n + n \log n^2) = O(n \log n)$

3.2.1 Long-n

Czas działania w tej rodzinie był największy dla algorytmu Diala. Zauważmy, że w tej rodzinie najkrótsza ścieżka jest długości co najmniej odległości Manhattan rozważanych wierzchołków. Odległości Manhattan mogą wynosić tu nawet $16 + \frac{n}{16}$ więc algorytm musi przejść przez C kubełków co najmniej liniową liczbę razy (zakładając, że średni koszt krawędzi to $\frac{C}{2} = \frac{n}{2}$). Natomiast algorytm Radix Heap radzi tu sobie bardzo dobrze, chociaż tworzenie kubełków i przerzucanie wartości i wybory najmniejszej wartości w kubełku generują większe koszta niż klasyczny algorytm Dijkstry z kopcem binarnym, który dla tak dużej liczby rozważanych wierzchołków okazał się być najszybszą strukturą.

3.2.2 Square-n

Średnie czasy są oczywiście mniejsze niż w Long-n, bo średnie odległości są mniejsze i trzeba wykonać mniej kroków, ale sama struktura grafów jest podobna i znowu algorytm Diala wypada najgorzej, następnie na drugim miejscu jest Radix Heap, a najlepiej klasyczny algorytm Dijkstry.

3.2.3 Random4-n

Wynik jest zaskakujący, bo struktura grafu jest zupełnie inna. W grafach losowych możemy z dowolnego wierzchołka średnio w dużo mniejszej liczbie kroków dojść do dowolnego innego wierzchołka. Okazuje się, że algorytm Diala działa najlepiej - tworzenie kubełków jest kosztowne, ale potem szybko dochodzimy do wyniku, a tworzenie kopca jest w tym przypadku bardziej kosztowne i algorytm Dijkstry wypada gorzej. Natomiast najbardziej kosztowne dla niskich odległości okazały się być operacje zmiany zakresów i szukania najmniejszych wartości w kubełkach, dlatego Radix Heap miał najgorszy czas.