

# Laboratorium 3: Porównanie implementacji algorytmu Dijkstry

Algorytmy Optymalizacji Dyskretnej  
Sara Źyndul  
279686

9 grudnia 2025

## 1 Opis implementacji i analiza złożoności

W ramach laboratorium zaimplementowano trzy warianty algorytmu wyznaczania najkrótszych ścieżek w grafie skierowanym z nieujemnymi wagami krawędzi. Poniżej przedstawiono szczegóły implementacyjne oraz analizę złożoności, gdzie  $n$  oznacza liczbę wierzchołków,  $m$  liczbę krawędzi, a  $C$  maksymalną wagę krawędzi.

## 1.1 Algorytm Dijkstry (Wariant podstawowy)

Wykorzystano standardową implementację algorytmu Dijkstry z użyciem kolejki priorytetowej typu Min-Heap.

- **Struktura danych:** Użyto kontenera `std::priority_queue` z biblioteki standardowej C++, przechowującego pary (`dystans`, `wierzchołek`).
- **Implementacja:** Zastosowano podejście *lazy deletion*. Zamiast kosztownej operacji `decrease-key`, nowe, lepsze odległości są dodawane do kolejki jako nowe elementy. Podczas zdajmowania elementu z wierzchołka kopca sprawdzane jest, czy pobrany dystans jest aktualny (tj. równy `dist[u]`). Jeśli nie, element jest ignorowany.
- **Złożoność czasowa:** W implementacji została wykorzystana `std::priority_queue`, dla której złożoność `extract_min` jest logarytmiczna, więc złożoność czasowa algorytmu Dijkstry to  $\mathcal{O}((m + n) \log n)$ . W najgorszym przypadku do kolejki może trafić  $m$  elementów (każda relaksa-cja krawędzi).
- **Złożoność pamięciowa:**  $\mathcal{O}(n + m)$  (graf + tablica dystansów + kolejka).

### Pseudokod algorytmu

---

#### Algorithm 1: Algorytm Dijkstry

---

Wejście: Graf  $G(V, E)$ , wierzchołek startowy  $s$ , cel  $target$

Wyjście: Tablica  $dist[]$  z najkrótszymi odległościami od  $s$

```
1 foreach  $v \in V$  do
2    $dist[v] \leftarrow \infty$ ;
3    $dist[s] \leftarrow 0$ ;
4    $PQ \leftarrow$  pusta kolejka priorytetowa (Min-Heap);
5    $PQ.push(\{0, s\})$ ;
6 while  $PQ$  nie jest pusta do
7    $\{d, u\} \leftarrow PQ.top()$ ;
8    $PQ.pop()$ ;
9   if  $u == target$  then
10    return;
11   if  $d > dist[u]$  then
12    continue; // Lazy deletion
13   foreach krawędź  $e \in G.adj[u]$  do
14      $v \leftarrow e.target$ ;
15      $w \leftarrow e.weight$ ;
16     if  $dist[u] + w < dist[v]$  then
17        $dist[v] \leftarrow dist[u] + w$ ;
18        $PQ.push(\{dist[v], v\})$ ;
```

---

## 1.2 Algorytm Diala

Algorytm ten jest optymalizacją Dijkstry dla grafów o całkowitych, niewielkich wagach krawędzi.

- **Struktura danych:** Zastosowano tablicę kubelków (**buckets**) zaimplementowaną jako `std::vector<std::vector<int>>`. Rozmiar tablicy wynosi  $C + 1$ .
- **Implementacja:** Wykorzystano bufor cykliczny. Wierzchołek o dystansie  $d$  trafia do kubelka o indeksie  $d \bmod (C + 1)$ . Algorytm iteruje po kubelkach używając wskaźnika `current_dist`. Podobnie jak w Dijkstrze, zastosowano mechanizm *lazy deletion* dla wierzchołków, które zostały zaktualizowane po wcześniejszym dodaniu do kubelka.
- **Złożoność czasowa:** Teoretycznie  $\mathcal{O}(m + D)$ , gdzie  $D$  to maksymalny dystans. W pesymistycznym wariancie (duże odległości między wierzchołkami) algorytm wykonuje wiele pustych przebiegów pętli, co daje złożoność zależną od sumy wag. Ogólnie  $D < nC$ , więc pesymistyczna złożoność to  $\mathcal{O}(m + nC)$
- **Złożoność pamięciowa:**  $\mathcal{O}(n + C)$ . Wymaga alokacji pamięci zależnej liniowo od maksymalnej wagi krawędzi.

### Pseudokod algorytmu

---

#### Algorithm 2: Algorytm Diala

---

**Wejście:** Graf  $G$ , źródło  $s$ , max waga  $C$   
**Wyjście:** Tablica  $dist[]$

```

1  $B\_size \leftarrow C + 1;$ 
2 Inicjalizacja  $buckets[0 \dots B\_size - 1]$ ;
3 foreach  $v \in V$  do
4    $dist[v] \leftarrow \infty$ 
5  $dist[s] \leftarrow 0;$ 
6  $buckets[0].push\_back(s);$ 
7  $current\_dist \leftarrow 0$ ,  $num\_elements \leftarrow 1$ ;
8 while  $num\_elements > 0$  do
   // Przesunięcie kurSORA do niepustego kubelka
   9   while  $buckets[current\_dist \bmod B\_size]$  jest pusty do
      10     $current\_dist \leftarrow current\_dist + 1$ ;
   11    $idx \leftarrow current\_dist \bmod B\_size$ ;
   12   while  $buckets[idx]$  nie jest pusty do
      13     $u \leftarrow buckets[idx].back();$ 
      14     $buckets[idx].pop\_back();$ 
      15     $num\_elements \leftarrow num\_elements - 1$ ;
      16    if  $u == target$  then
      17       $\text{return}$ 
      18    if  $dist[u] < current\_dist$  then
      19       $\text{continue}$ 
      20    foreach krawędź  $e \in G.adj[u]$  do
      21       $v \leftarrow e.target$ ,  $w \leftarrow e.weight$ ;
      22      if  $dist[u] + w < dist[v]$  then
      23         $dist[v] \leftarrow dist[u] + w$ ;
      24         $buckets[dist[v] \bmod B\_size].push\_back(v);$ 
      25         $num\_elements \leftarrow num\_elements + 1$ ;
   26    $current\_dist \leftarrow current\_dist + 1$ ;

```

---

### 1.3 Algorytm Radix Heap

Algorytm ten łączy zalety Dijkstry i Diala, wykorzystując własności reprezentacji binarnej liczb. Wielkości kubełków są tutaj kolejnymi potęgami dwójką: 1, 1, 2, 4, 8, ...

- **Struktura danych:** Użyto statycznej liczby kubełków (64 dla 64-bitowego typu long long), gdzie zakresy kubełków rosną wykładniczo.
- **Implementacja:** Indeks kubełka wyznaczany jest na podstawie pozycji najbardziej znaczącego różniącego się bitu (MSB) między ostatnio zdjętym dystansem a dystansem kandydata. Do szybkiego obliczania MSB wykorzystano funkcję wbudowaną kompilatora `__builtin_clzll` (Count Leading Zeros). Zastosowano mechanizm redystrybucji elementów z wyższych kubełków do niższych.
- **Złożoność czasowa:**  $\mathcal{O}(m + n \log C)$ . Operacje bitowe są wykonywane w czasie stałym, a każdy element jest przenoszony między kubełkami najwyżej  $\log C$  razy.
- **Złożoność pamięciowa:**  $\mathcal{O}(n + m)$ . Niezależna liniowo od wartości  $C$ .

## Pseudokod algorytmu

**Algorithm 3:** Algorytm Radix Heap

---

```
1 Function get_bucket_index(last, key):
2     diff ← (last ⊕ key);
3     if diff == 0 then return 0;
4     return 64 - clz(diff);                                // most significant bit index
5 Wejście: Graf G, źródło s
6 Inicjalizacja buckets[0 … 64];
7 last_dist ← 0, size ← 0;
8 foreach v ∈ V do
9     dist[v] ← ∞
10    dist[s] ← 0, buckets[0].push({0, s}), size ← 1;
11    while size > 0 do
12        idx ← pierwszy niepusty indeks w buckets;
13        if idx > 0 then
14            min_key ← min(k,u)∈buckets[idx] k;
15            last_dist ← min_key;
16            foreach element {k, u} ∈ buckets[idx] do
17                if k == dist[u] then
18                    new_idx ← get_bucket_index(last_dist, k);
19                    buckets[new_idx].push({k, u});
20                else
21                    size ← size - 1
22            buckets[idx].clear();
23            continue;
24        while buckets[0] nie jest pusty do
25            {k, u} ← buckets[0].back(), buckets[0].pop_back();
26            size ← size - 1;
27            if k ≠ dist[u] then
28                continue
29            if u == target then
30                return
31            foreach krawędź e ∈ G.adj[u] do
32                nd ← dist[u] + e.weight;
33                if nd < dist[v] then
34                    dist[v] ← nd;
35                    b ← get_bucket_index(last_dist, nd);
36                    buckets[b].push({nd, v}), size ← size + 1;
```

---

## 2 Wyniki eksperymentów

### 2.1 Poprawność i długości ścieżek

Poniższa tabela przedstawia wyniki dla największych instancji testowych z każdej rodziny. Porównano długości znalezionych ścieżek między wierzchołkiem o najmniejszym i największym indeksie (Min-Max) oraz dla 4 losowych par.

Zgodność wyników potwierdza poprawność implementacji wszystkich trzech algorytmów.

Tabela 1: Długości najkrótszych ścieżek dla pierwszego i ostatniego wierzchołka. Oznaczenie **MLE** (Memory Limit Exceeded) wskazuje błąd braku pamięci dla algorytmu Diala. W każdym z tych trzech przypadków masymalna waga to około  $10^9$ , co wymagałoby alokacji 24 GB pamięci RAM.

Instancja	Dijkstra / Radix	Dial
Random4-n.21.0	<b>Min-Max:</b> 9051281 <b>Rand:</b> 8021753, 7962615, 11520120, 7944933	Wyniki zgodne
Random4-C.15.0	<b>Min-Max</b> 3471241820 <b>Rand:</b> 4303132782, 3285449479, 3923515585, 4834216768	<b>MLE</b>
Long-n.21.0	<b>Min-Max</b> 31336751771 <b>Rand:</b> 4137488546, 20534900152, 46406387800, 63037605316	Wyniki zgodne
Square-n.21.0	<b>Min-Max</b> 714640488 <b>Rand:</b> 627400780, 773584334, 820931845, 312060678	Wyniki zgodne
Long-C.15.0	<b>Min-Max</b> 1308259008765 <b>Rand:</b> 1122339012956, 6279817984398, 629828042459, 10523381128436	<b>MLE</b>
Square-C.15.0	<b>Min-Max</b> 122219500320 <b>Rand:</b> 109667679059, 239894218968, 183454171539, 261947371811	<b>MLE</b>
USA-road-d.W	<b>Min-Max</b> 13133160 <b>Rand:</b> 8070957, 3015305, 5366628, 12272763	Wyniki zgodne

## 2.2 Czasy znajdywania ścieżek

Tabela 2 przedstawia czas działania algorytmów dla największych instancji testowych z każdej rodziny. Czas I - czasy znalezienia ścieżek między wierzchołkiem o najmniejszym i największym indeksie (Min-Max). Czas II - średnie czasy znalezienia ścieżek dla 4 losowych par.

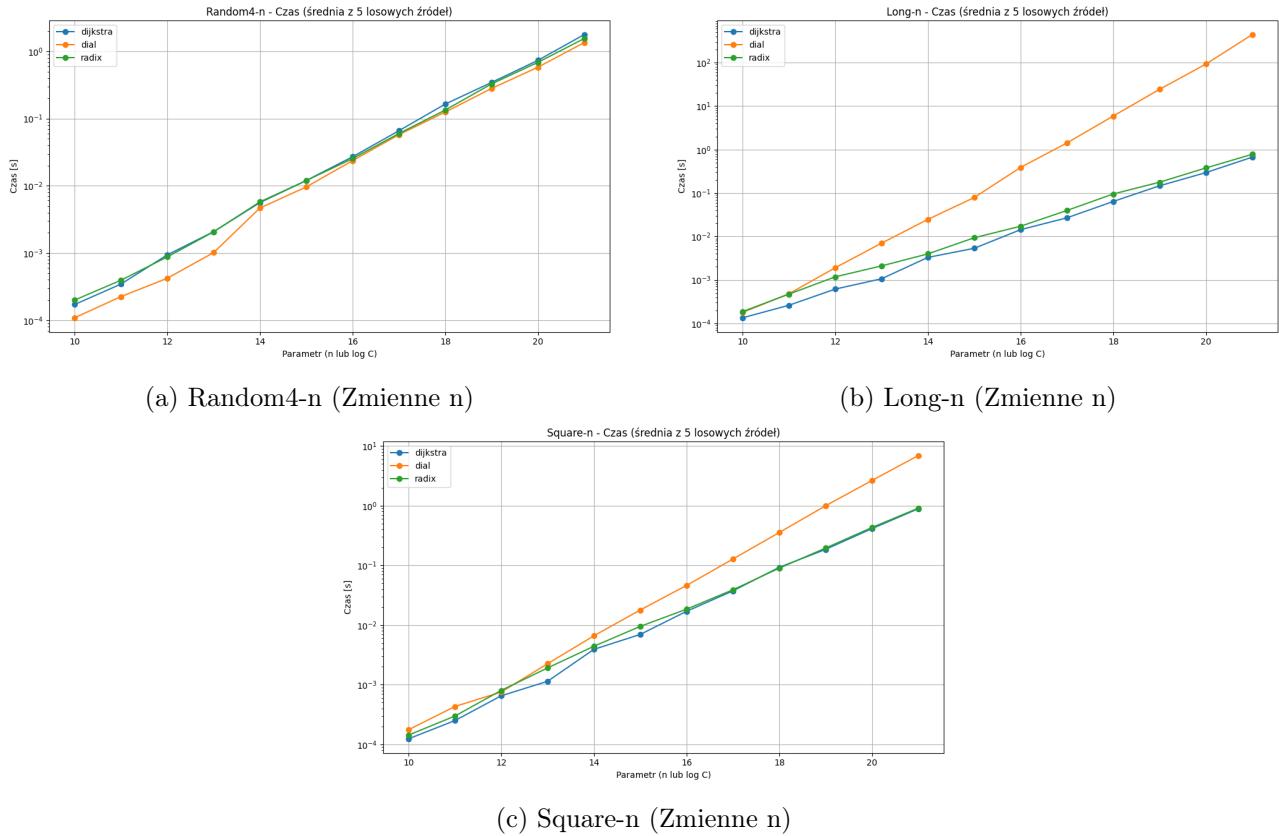
Rodzina - Algorytm	Czas I (ms)	Czas II (ms)
Long-C - Dijkstra	28.04	114.98
Long-C - Dial	<b>MLE</b>	<b>MLE</b>
Long-C - RadixHeap	31.6	160.58
Long-n - Dijkstra	291.2	216.61
Long-n - Dial	95545.73	117392.49
Long-n - RadixHeap	308.11	265.49
Random4-C - Dijkstra	139.37	210.35
Random4-C - Dial	<b>MLE</b>	<b>MLE</b>
Random4-C - RadixHeap	109.35	145.75
Random4-n - Dijkstra	547.91	487.54
Random4-n - Dial	860.5	992.31
Random4-n - RadixHeap	371.79	322.88
Square-C - Dijkstra	105.87	129.67
Square-C - Dial	<b>MLE</b>	<b>MLE</b>
Square-C - RadixHeap	112.87	145.99
Square-n - Dijkstra	257.69	239.23
Square-n - Dial	2750.22	2300.17
Square-n - RadixHeap	255.78	219.63
USA-road-d - Dijkstra	3253.2	2272.93
USA-road-d - Dial	2860.24	2134.75
USA-road-d - RadixHeap	2845.07	2036.93

### Obserwacje:

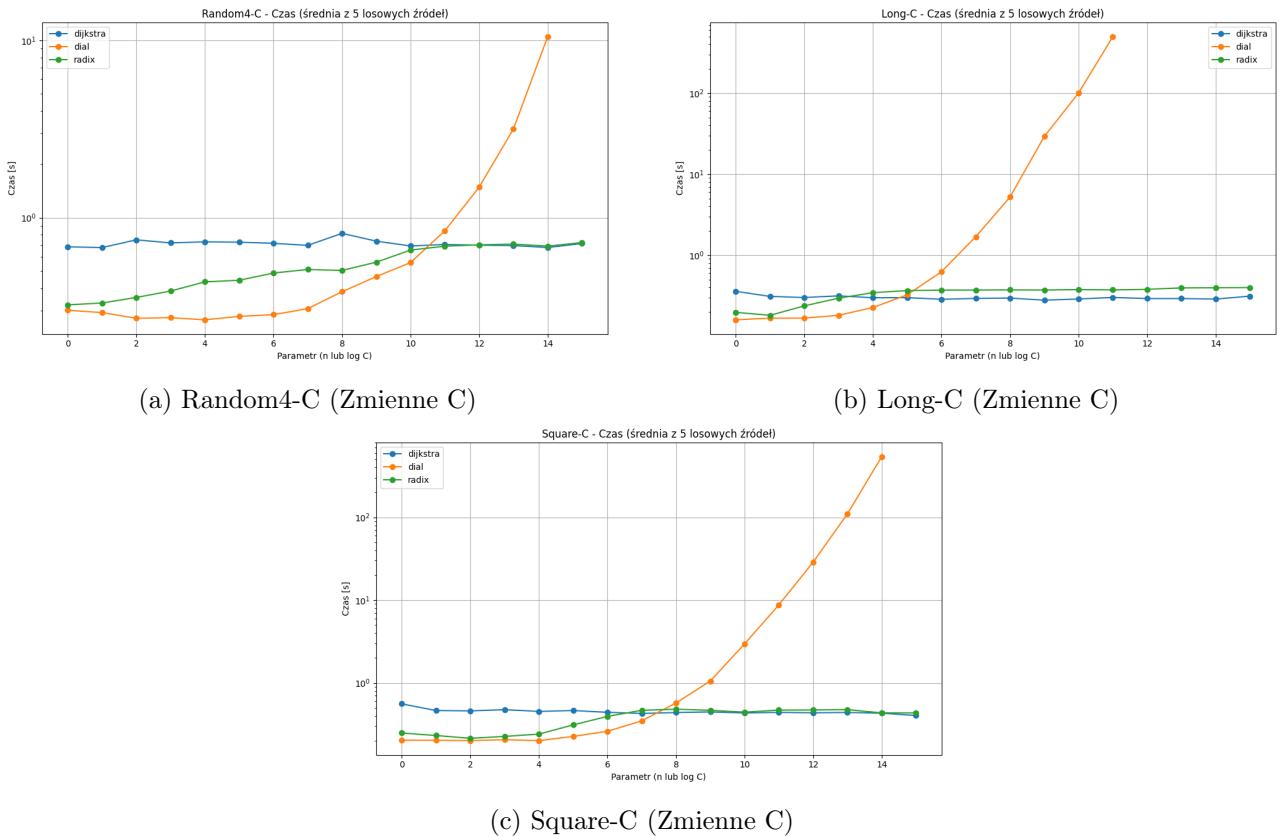
- Widzimy, że algorytm Radix Heap radzi sobie wyraźnie lepiej od Dijkstry przy grafach losowych oraz przy grafach z rodziną USA-road-d.
- Jeśli algorytmowi Diala udało się zaalokować wystarczającą ilość pamięci to za wyjątkiem rodziny USA-road-d zawsze był najwolniejszym algorytmem. Widać to szczególnie na przykładzie rodziny Long-n, dla której wartości C rosną proporcjonalnie do ilości wierzchołków.
- Dla grafów z rodziny USA-road-d algorytm Diala radzi sobie lepiej od algorytmu Dijkstry, jednak wciąż jest wolniejszy od Radix Heap. Zauważmy, że jest to jedyna z testowanych rodzin z relatywnie małymi wartościami C.
- Algorytmy Dijkstry i Radix Heap wykazują się wysoką stabilnością.

## 2.3 Wykresy czasu działania

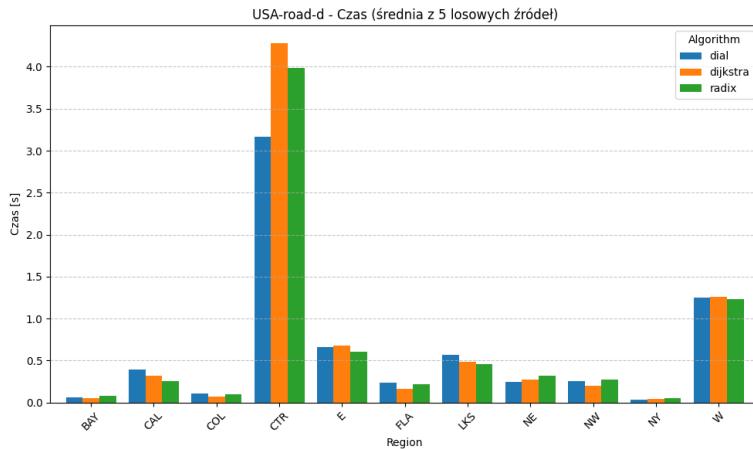
Poniższe wykresy prezentują zależność czasu działania od parametru generatora (liczba wierzchołków  $n$  lub logarytm z maksymalnej wagi  $C$ ). Dla każdego przypadku przedstawiono średni czas dla 5 losowych źródeł.



Rysunek 1: Czas działania dla rodzin ze zmienną liczbą wierzchołków.



Rysunek 2: Czas działania dla rodzin ze zmienną maksymalną wagą krawędzi ( $C$ ). Oś Y logarytmiczna.



Rysunek 3: Czas działania dla rzeczywistych sieci drogowych (USA-road-d).

### 3 Interpretacja wyników

Przeprowadzone eksperymenty potwierdzają teoretyczne różnice w złożoności zaimplementowanych algorytmów.

1. **Wpływ maksymalnej wagi ( $C$ ):** Jest to najbardziej widoczna różnica. Dla rodzin grafów, w których parametr  $C$  rósł wykładniczo (Random4-C, Long-C, Square-C):

- Algorytmy **Dijkstry** i **Radix Heap** wykazały się dużą stabilnością (wykresy płaskie). Czas działania Dijkstry jest niezależny od  $C$ , a Radix Heap zależy logarytmicznie ( $\log C$ ), co przy rozmiarze grafów jest wartością pomijalną.

- Algorytm **Diala** okazał się całkowicie nieefektywny dla dużych wag. Jego złożoność pamięciowa  $\mathcal{O}(n + C)$  spowodowała wyczerpanie dostępnej pamięci RAM (błąd `std::bad_alloc`) dla instancji o  $C > 300 \cdot 10^6$ . Na wykresach widoczny jest wykładniczy wzrost czasu przed wystąpieniem błędu.
2. **Wpływ liczby wierzchołków ( $n$ ):** Dla rodzin ze zmiennym  $n$  i stałym, małym  $C$ , wszystkie algorytmy zachowują się stabilnie. Algorytm Radix Heap często osiągał czasy zbliżone lub lepsze od Dijkstry dzięki uniknięciu kosztownych operacji na kopcu (sortowania), zastępując je szybkimi operacjami bitowymi.
  3. **Struktura grafu:** W przypadku grafów typu *Long-n* (długie łańcuchy) i *Square-n* (siatki), algorytm Diala działał wolniej niż oczekiwano (czasem gorzej od Dijkstry). Wynika to z faktu, że przy rzadkich grafach o dużej średnicy, algorytm traci czas na iterowanie po pustych kubelkach w buforze cyklicznym.
  4. **Podsumowanie:** Algorytm Dijkstry z kolejką priorytetową jest najbardziej uniwersalnym rozwiązaniem, odpornym na charakterystykę wag. Radix Heap stanowi doskonałą alternatywę dla wag całkowitych, oferując wydajność zbliżoną do liniowej. Algorytm Diala ma wąskie zastosowanie – tylko do grafów o małych, całkowitych wagach.

## 4 Wnioski

- **Zgodność teorii z praktyką:** Wyniki eksperymentalne jednoznacznie potwierdziły teoretyczne ograniczenia algorytmu Diala. Testy na danych o dużej rozpiętości wag wykazały jego krytyczną niewydajność, wynikającą z konieczności obsługi olbrzymiej tablicy kubelków, co czyniło go najwolniejszym z badanych rozwiązań.
- **Stabilność algorytmu Dijkstry:** Dla rodzin grafów o nieznanej specyfice lub bardzo dużych wagach, klasyczny algorytm Dijkstry pozostaje najbezpieczniejszym wyborem. Jego czas działania jest przewidywalny i niezależny od rzędu wielkości kosztów krawędzi.
- **Przewaga Radix Heap:** W scenariuszach praktycznych, gdzie wagi są liczbami całkowitymi (nawet 64-bitowymi), algorytm Radix Heap okazuje się rozwiązaniem optymalnym. Łączy on zalety algorytmu Diala (szybkość) z odpornością na duże wagi, charakterystyczną dla Dijkstry.
- **Bilans zysków i strat:** Ewentualny narzut czasowy algorytmów Dijkstry i Radix Heap w przypadku małych wag jest znikomy. W przeciwnieństwie do tego, koszt użycia algorytmu Diala przy dużych wagach jest olbrzymi. Dlatego w ogólnym rozrachunku algorytmy oparte na kopcach (binarnym lub radix) są rozwiązaniami znacznie bardziej wszechstronnymi.