Laboratorium 3:
Porównanie implementacji algorytmu Dijkstry

Algorytmy Optymalizacji Dyskretnej
Sara Żyndul 279686

3 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 zdejmowania elementu z wierzchołka kopca sprawdzane jest, czy pobrany dystans jest aktualny (tj. równy `dist[u]`). Jeśli nie, element jest ignorowany.
- **Złożoność czasowa:** $\mathcal{O}(m \log n)$. W najgorszym przypadku do kolejki może trafić m elementów (każda relaksacja krawędzi), a operacje na kopcu zajmują czas logarytmiczny.
- **Złożoność pamięciowa:** $\mathcal{O}(n + m)$ (graf + tablica dystansów + kolejka).

1.2 Algorytm Diala

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

- **Struktura danych:** Zastosowano tablicę kubełków (`buckets`) zaimplementowaną jako `std::vector<std::vector<int»`. Rozmiar tablicy wynosi $C + 1$.
- **Implementacja:** Wykorzystano bufor cykliczny. Wierzchołek o dystansie d trafia do kubełka o indeksie $d \mod (C + 1)$. Algorytm iteruje po kubełkach używając wskaźnika `current_dist`. Podobnie jak w Dijkstrze, zastosowano mechanizm *lazy deletion* dla wierzchołków, które zostały zaktualizowane po wcześniejszym dodaniu do kubełka.
- **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.

1.3 Algorytm Radix Heap

Algorytm ten łączy zalety Dijkstry i Diala, wykorzystując własności reprezentacji binarnej liczb.

- **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.

2 Wyniki eksperymentów

2.1 Poprawność i długości ścieżek

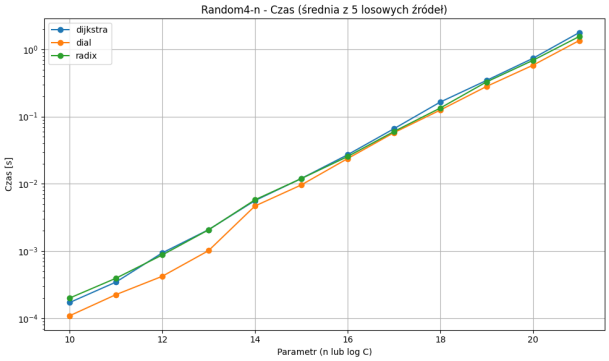
Tabela 1 przedstawia wyniki dla największych instancji testowych z każdej rodziny. Porównano długości znalezionych ścieżek między wierzchołkiem o najmniejszym i największym indeksie (Min-Max) oraz dla 4 losowych par.

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 $1000000000 = 10^9$ co wymagałoby alokacji 24 GB pamięci RAM.

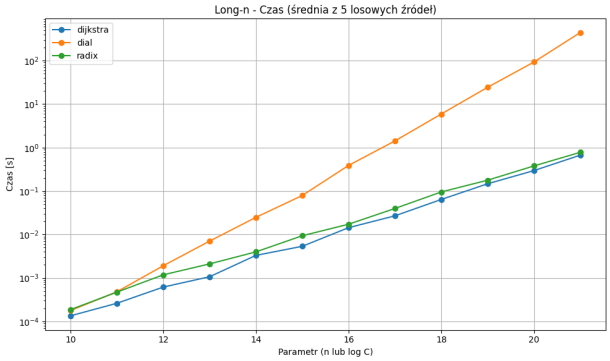
Rodzina	**Instancja**	**Dijkstra / Radix**	**Dial**
Random4-n	Random4-n.21.0	**MM:** 9051281 **Rand:** 8021753, 7962615, 11520120, 7944933	Wyniki zgodne
Random4-C	Random4-C.15.0	**MM:** 3471241820 **Rand:** 4303132782, 3285449479, 3923515585, 4834216768	**MLE** (Brak wyniku)
Long-n	Long-n.21.0	**MM:**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

2.2 Wykresy czasu działania

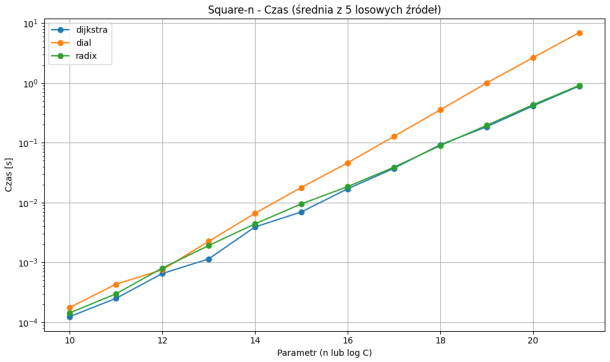
Poniższe wykresy prezentują zależność czasu działania od parametru generatora (liczba wierzchołków n lub logarytm z maksymalnej wagi C). Dla każdego przypadku przedstawiono średni czas dla 5 losowych źródeł.



(a) Random4-n (Zmienne n)

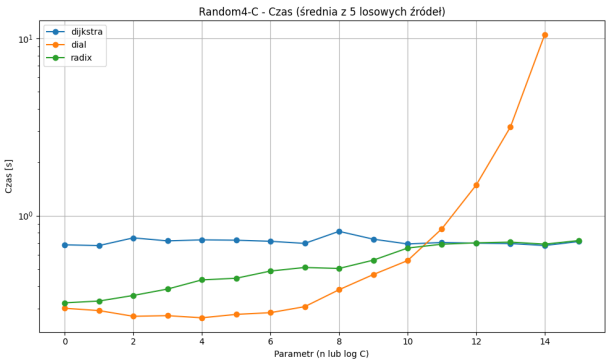


(b) Long-n (Zmienne n)

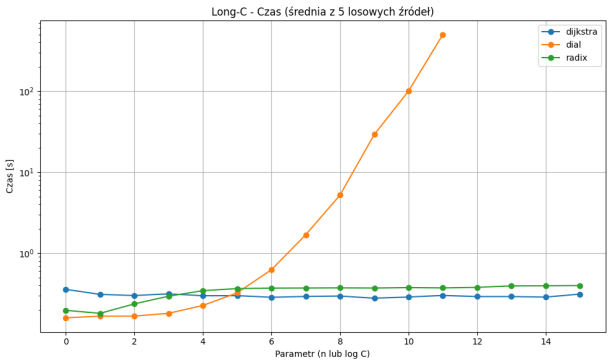


(c) Square-n (Zmienne n)

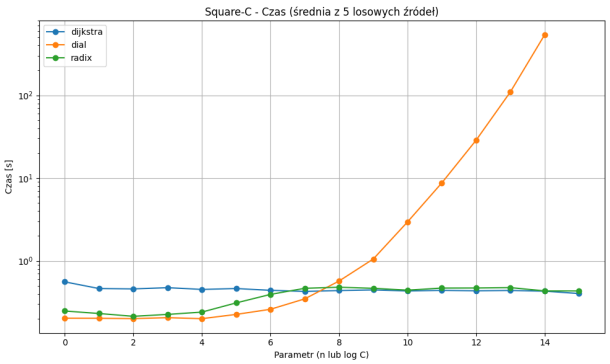
Rysunek 1: Czas działania dla rodzin ze zmienną liczbą wierzchołków.



(a) Random4-C (Zmienne C)

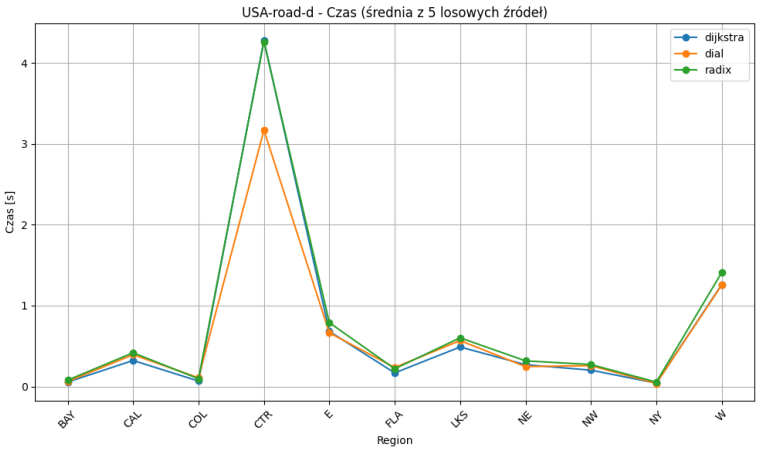


(b) Long-C (Zmienne C)



(c) Square-C (Zmienne C)

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 kubełkach 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.