2020

Struktury Danych I
Złożoność obliczeniowa

ALIAKSEI TOKARAU

Zadanie projektowe nr. 2

Data wykonania projektu: 10.05.2020

Data oddania sprawozdania: 20.05.2020

Grupa zajęciowa: cz/TP+1/2 11:15-13:00

Skład grupy: Aliaksei Tokarau 250930

Prowadzący: Dr inż. Jarosław Mierzwa

Cel projektu

Implementacja następujących reprezentacji grafu:

- Macierz sąsiedzictwa
- Lista sąsiadów

Pomiar czasu wykonania następujących algorytmów:

- Algorytm Prima
- Algorytm Kruskala
- Algorytm Dujkstry
- Algorytm Bellmana-Forda

Przyjęte założenia

- Podstawowym elementem struktur danych jest 4 bajtowa liczba całkowita ze znakiem
- Wszystkie struktury danych są alokowane dynamicznie
- Ponieważ pojedynczy pomiar czasu jest obarczony dużym błędem, pomiary wykonane są wielokrotnie (100 razy) i otrzymana wartość jest uśredniona
- Środowiskiem programistycznym jest Clion 2019.3
- Wszyskie algorytmy były stworzone bez użycia STL
- Każdy wykres jest zależnością czasu od ilości wiezchołków

Wstęp teoretyczny

Często zdarza się, że chcemy wykonać schemat relacji pewnych elementów, czy też przedstawić graficznie daną sytuację. Zazwyczaj stosujemy prowizoryczne linie, które łączą punkty przedstawine jako wybrane figury geometryczne. Taki schemat znacznie ułatwia nam pracę i pozwala osiągnąć lepsze rezultaty zamierzonego celu. Poznajmy więc nową konstrukcję matematyczną innego rodzaju, cieszącą się ogromnym zainteresowaniem w wielu dziedzinach nauki.

Cała widoczną strukturę będziemy nazywać **grafem**, wktórym **wierzchołki** są oznaczone literami *z, x, y, w, u, t*. **Krawędzie** natomiast, będą oznaczane symbolem typu *vz*, co rozumiane jest jako krawędź łącząca wierzchołek *v* z wierzchołkiem *z*.

Ogólnie możemy powiedzieć, że graf *G* składa się z nieppustego zbioru wierzchołków, który oznaczamy symbolem *V(G)* (ang. verticies – wierzchołki) oraz zbioru krawędzi, oznaczonego symbolem *E(G)* (ang. edges – krawędzie). Ilość elementów zbioru *V(G)* – liczbę wierzchołków będziemy często ozmaczać symbolem *n*, a ilość elementów zbioru *E(G)* – liczbę krawędzi – symbolem *m.* Zatem możemy naposać:

$$G = (V(G), E(G))$$

Jest to barzo przydatny model matematyczny i pozwala arozwiązanie dużej ilości problemów. Do implementacji tego modelu w programie potrzebukemy jakieś reprezentacji grafu. Innymi słowy: jak możemy przechowywać informacje dotyczące stanu grafu. Jest na to kilka sposobów. Jak już było wspomniane wyżej, w tym projekcie wykorzestujemy macierz sąsiedzictwa oraz listę sąsiadów.

Najprostrą, a zarazem najefektuwniejszą metodą reprezentacji grafów jest **lista sąsiadów**. Polega na wypisaniu wszystkich sąsiadów dla każdego wierzchołka *v* grafu *G* którego chcemy reprezentować. Na przykład jeśli wirzchołek *v* sąsiaduje z wierzchołkami *z,w, x,* i *y* to wtedy zapiszemy:

$$v: z, w, x, y.$$

Powtarzając tę czynność dla pozostałych wierzchołków otrzymay reprezentacje przez listę sąsiadów.

Drugi sposób reprezentacji grafu jest **macierz sąsiedztwa**. Oznaczmy graf *G*, w którym wierzchołki są kolejno przedstawione liczbami ze zbioru *{1,2,…,n}*, gdzie *n* jest ilością wierzchołków grafu *G*. Następnie weźmy pod uwagę macierz wymiaru n×nn×n, w której wyraz aijaij jest równy liczbie krawędzi łączących wierzchołek *i* z wierzchołkiem *j*.

Minimalne drzewo rozpinające

Minimalne drzewo rozpinające (ang. minimum spanning tree, w skrócie MST), inaczej drzewo rozpinające o minimalnej wadze – drzewo łączące wszystkie wierzchołki pewnego grafu spójnego mające najmniejszą możliwą sumę wag krawędzi. Jeśli graf ma *n* wierzchołków, to jego drzewo rozpinające zawsze będzie miało *n – 1* krawędzi. Jeśli ten graf ma *m* krawędzi, aby utworzyć drzewo rozpinające, trzeba usunąć z grafu *m – n + 1* krawędzi. Liczba ta jest określana jako liczba cyklomatryczna.

Istnieje sporo przypadków użycia dla minimalne drzewo rozpinające. Przykładem może być firma telekomunikacyjna próbuje położyć kabel w nowej dzielnicy. Jeżeli struktura jest zakopać przewód tylko wzdłuż określonych ścieżek (np drogi), to nie będzie to wykres zawierający punkty (na przykład domów) połączone tych ścieżkach. Niektóre z tych ścieżek mogą być droższe, ponieważ są one dłuższe, albo wymagają kabla leżeć głębiej; Te ścieżki będzie reprezentowane przez krawędzie z większymi ciężarami. Waluta dopuszczalne jednostką obciążnikiem - nie jest wymagane do długości krawędzi przestrzegać normalnych zasad geometrii, takie jak nierówności trójkąta . Drzewo rozpinające dla tego wykresu będzie podzbiorem tych ścieżkach, które nie ma cykli ale nadal łączy każdy dom; tam może być kilka obejmujące drzewa możliwe. Minimalne drzewo rozpinające będzie jednym z najniższym całkowitym koszcie, co stanowi najtańszą ścieżkę do układania kabla.

W tym projekcie będziemy wykorzystywali dwóch algorytmów na wyznaczenia minimalnego drzewa ropinającego: **algorytm Prima** oraz **algorytm Kruskala** na różnych reprezentacjach. Na koniec porównajmy praktyczną oraz teoretyczną wydzjność tych algorytmów.

Algorytm Prima

Dla każdego wierzchołka określamy, że jego koszt wynosi nieskończoność, a poprzednik jest nieokreślony. Wybranemu wierzchołkowi przypisujemy wartość 0, a następnie tworzymy kolejkę wierzchołków do rozpatrzenia. W kolejce dane są posortowane po koszcie elementu. Następnie, dopóki kolejka nie jest pusta, wybieramy wierzchołek o najniższym koszcie i aktualizujemy jego sąsiadów, którzy występują w kolejce. Jeśli krawędź z wybranego wierzchołka z kolejki do sąsiada ma niższą wartość niż sąsiad ma koszt to aktualizujemy informacje.

Zastosowanie algorytmu:

- Projektowanie sieci komputerowych
- Projektowanie sieci rurociągów wody pitnej lub gazu ziemnego.
- Umieszczenie wież mikrofalowych lub podobnych projektów

Złożoność obliczeniowa: $O(|V|^2)$, paięciowa: $O(|V|)$.

Założenia eksperymentu:

- Testujemy algorytm dla dwóch reprezentacji grafu. Graf zawsze się generuje w postaci macierzowej. W przypadku testowania listy sąsiedzictwa, graf przekształcamy w tą formę po zgenerowaniu.
- Nie liczymy czas generacji oraz przepisywanie grafu z jednej postaci do drugiej. Genracja grafu prepływa s następujący sposób: generujemy drzewo rozpinające, dalej w zależności od gęstości dodajemy krawędzie.
- Testujemy 7 punktów obserwowanych: 50, 60, 70, 80, 90, 100, 110 wierzchołków dla trzech gęstości: 20%, 60%, 99%.
- Mierzymy czas 100 prób, zatem uśredniamy wynik.

Log programy

Macierz sąsiedzictwa:

```
Speed test of Prim's algorithm.
Adjacency matrix:
Density: 20%
Size: 50 result(nanoseconds): 11988
Size: 60 result(nanoseconds): 36966
Size: 70 result(nanoseconds): 67935
Size: 80 result(nanoseconds): 113890
Size: 90 result(nanoseconds): 174836
Size: 100 result(nanoseconds): 263784
Size: 110 result(nanoseconds): 367708
Density: 60%
Size: 50 result(nanoseconds): 377715
Size: 60 result(nanoseconds): 394689
Size: 70 result(nanoseconds): 420662
Size: 80 result(nanoseconds): 495608
Size: 90 result(nanoseconds): 572545
Size: 100 result(nanoseconds): 649485
Size: 110 result(nanoseconds): 789382
Density: 99%
Size: 50 result(nanoseconds): 797373
Size: 60 result(nanoseconds): 813356
Size: 70 result(nanoseconds): 840841
Size: 80 result(nanoseconds): 871809
Size: 90 result(nanoseconds): 915784
Size: 100 result(nanoseconds): 989731
Size: 110 result(nanoseconds): 1082676
```

Lista sąsiadów:

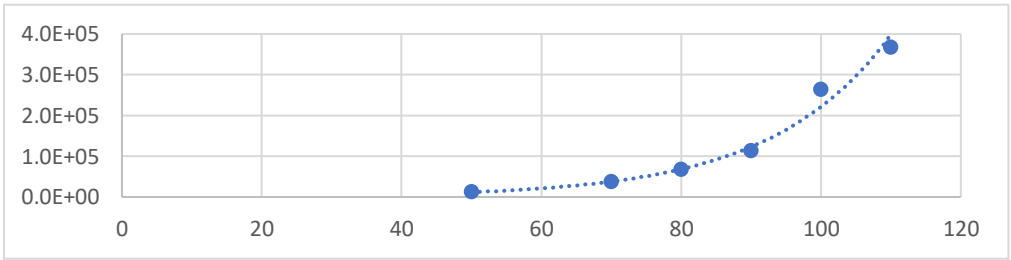
```
Speed test of Prim's algorithm.
Adjacency list:
Density: 20%
Size: 50 result(nanoseconds): 2015
Size: 60 result(nanoseconds): 6024
Size: 70 result(nanoseconds): 14034
Size: 80 result(nanoseconds): 26038
Size: 90 result(nanoseconds): 42027
Size: 100 result(nanoseconds): 69000
Size: 110 result(nanoseconds): 102959
Density: 60%
Size: 50 result(nanoseconds): 111952
Size: 60 result(nanoseconds): 131504
Size: 70 result(nanoseconds): 154480
Size: 80 result(nanoseconds): 191425
Size: 90 result(nanoseconds): 247390
Size: 100 result(nanoseconds): 326323
Size: 110 result(nanoseconds): 451799
Density: 99%
Size: 50 result(nanoseconds): 466783
Size: 60 result(nanoseconds): 492754
Size: 70 result(nanoseconds): 538710
Size: 80 result(nanoseconds): 631636
Size: 90 result(nanoseconds): 751550
Size: 100 result(nanoseconds): 939424
Size: 110 result(nanoseconds): 1202251
```

Wykresy

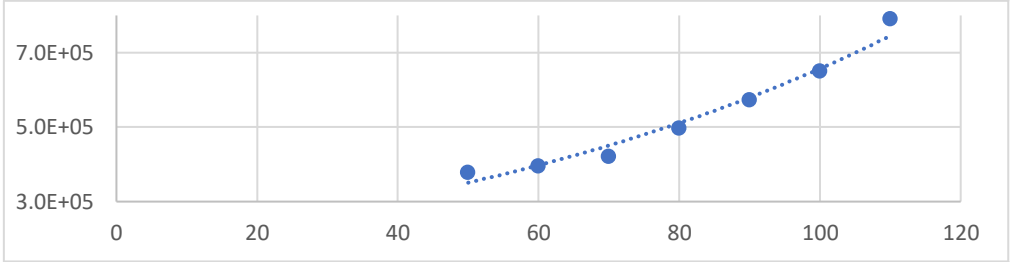
Gęstość

Macierz sąsiedzictwa

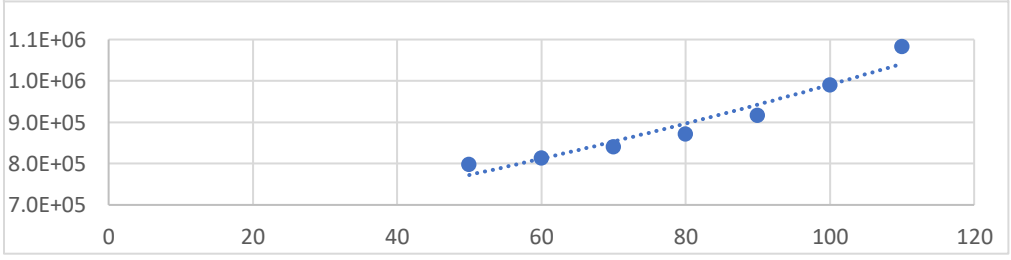
20%



60%



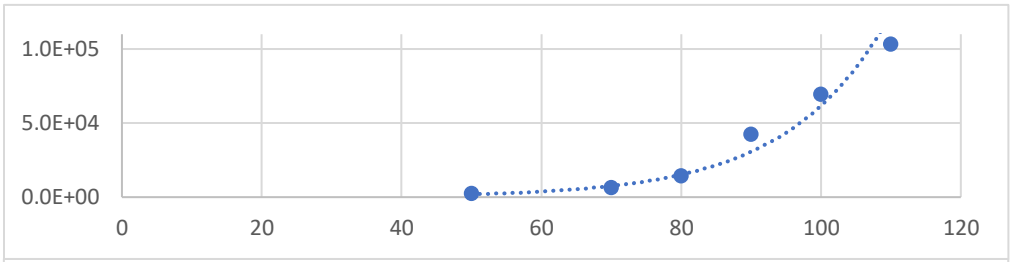
99%



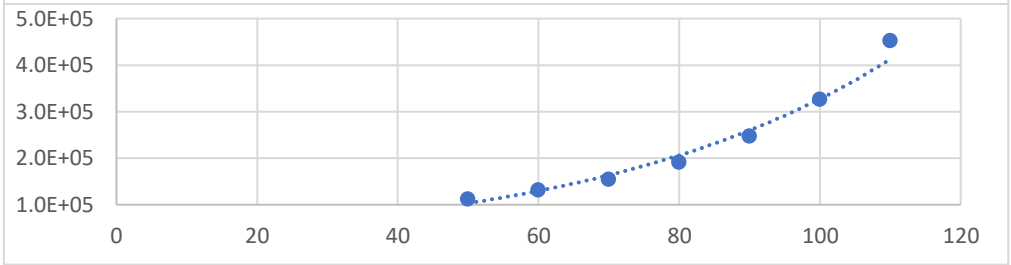
Gęstość

Lista sąsiadów

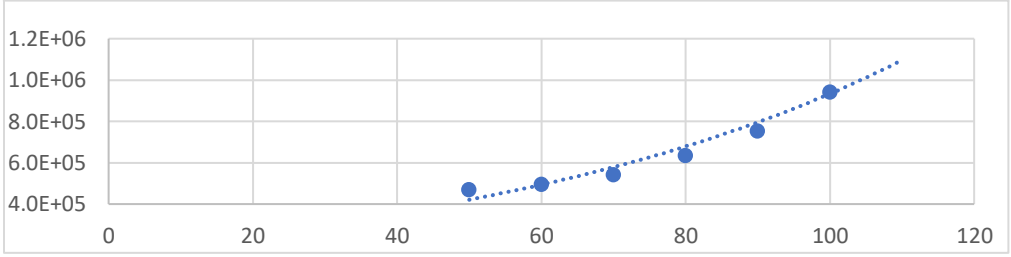
20%



60%



99%



Wnioski

Prosta implementacja algorytmu Prima, używając macierzy sąsidzictwa lub listy sąsiadów i liniowo przeszukiwając tablicę wag w celu znalezienia minimalnej krawędzi wagowej do dodania, wymaga czasu pracy O ($|V|^2$). Jednak ten czas działania może być znacznie poprawiony poprzez użycie kopca do implementacji znajdowania krawędzi minimalnej wagi w pętli wewnętrznej algorytmu.

Praktyczna złożoność obliczeniowa jest zgodna z teoretyczną, co widać na wykresach i logu programy. W obu reprezentscjach czas wykonania algorytmu rośnie eksponencjalnie wraz z gęstością grafu. To jest dla tego, że algorytm polega na sprawdzaniu długości poszczególnych krawędzi. Widać, że generalnie algorytm działa szybciej na liście sąsiadów, to jest z powodu specyfiki implementacji różnych reprezentacji.

Algorytm Kruskala

Algorytm Kruskala do znalezienia minimalnego drzewa rozpinającego wykorzystuje podejście chciwości. Algorytm ten traktuje graf jako las, a każdy jego węzeł jako pojedyncze drzewo. Drzewo łączy się z innym tylko i tylko wtedy, gdy, ma najmniejszy koszt spośród wszystkich dostępnych opcji i nie narusza właściwości MDR.

Zastosowanie algorytmu:

- Tworzenie kabli
- Sieci TV
- Operacje turystyczne

Złożoność obliczeniowa: $O(E \log V)$, pamięciowa: $O(|V|)$.

Założenia eksperymentu:

- Testujemy algorytm dla dwóch reprezentacji grafu. Graf zawsze się generuje w postaci macierzowej. W przypadku testowania listy sąsiedzictwa, graf przekształcamy w tą formę po zgenerowaniu.
- Nie liczymy czas generacji oraz przepisywanie grafu z jednej postaci do drugiej. Genracja grafu prepływa s następujący sposób: generujemy drzewo rozpinające, dalej w zależności od gęstości dodajemy krawędzie.
- Testujemy 7 punktów obserwowanych: 50, 60, 70, 80, 90, 100, 110 wierzchołków dla trzech gęstości: 20%, 60%, 99%.
- Mierzymy czas 100 prób, zatem uśredniamy wynik.

Log programy

Macierz sąsiedzictwa:

```
Adjacency matrix:
Density: 20%
Size: 50 result(nanoseconds): 34961
Size: 60 result(nanoseconds): 101881
Size: 70 result(nanoseconds): 222812
Size: 80 result(nanoseconds): 414682
Size: 90 result(nanoseconds): 708493
Size: 100 result(nanoseconds): 1169212
Size: 110 result(nanoseconds): 1896792
Density: 60%
Size: 50 result(nanoseconds): 2012728
Size: 60 result(nanoseconds): 2294553
Size: 70 result(nanoseconds): 2851216
Size: 80 result(nanoseconds): 3802651
Size: 90 result(nanoseconds): 5108878
Size: 100 result(nanoseconds): 7128690
Size: 110 result(nanoseconds): 10014008
Density: 99%
Size: 50 result(nanoseconds): 10280839
Size: 60 result(nanoseconds): 10802538
Size: 70 result(nanoseconds): 11827930
Size: 80 result(nanoseconds): 13451958
Size: 90 result(nanoseconds): 15989472
Size: 100 result(nanoseconds): 19789261
Size: 110 result(nanoseconds): 25308057
```

Lista sąsiadów:

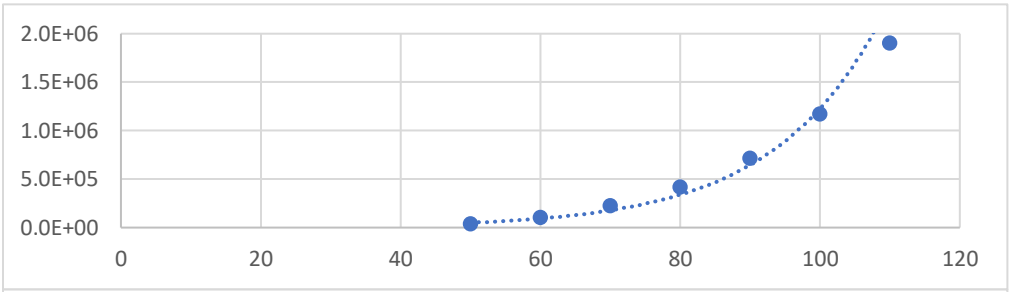
```
Speed test of Kruskal's algorithm.
Adjacency list:
Density: 20%
Size: 50 result(nanoseconds): 17983
Size: 60 result(nanoseconds): 44955
Size: 70 result(nanoseconds): 85899
Size: 80 result(nanoseconds): 142845
Size: 90 result(nanoseconds): 222774
Size: 100 result(nanoseconds): 328691
Size: 110 result(nanoseconds): 478601
Density: 60%
Size: 50 result(nanoseconds): 538546
Size: 60 result(nanoseconds): 648469
Size: 70 result(nanoseconds): 820349
Size: 80 result(nanoseconds): 1117173
Size: 90 result(nanoseconds): 1607872
Size: 100 result(nanoseconds): 2200530
Size: 110 result(nanoseconds): 3034031
Density: 99%
Size: 50 result(nanoseconds): 3167940
Size: 60 result(nanoseconds): 3396793
Size: 70 result(nanoseconds): 3783548
Size: 80 result(nanoseconds): 4415152
Size: 90 result(nanoseconds): 5376597
Size: 100 result(nanoseconds): 6842744
Size: 110 result(nanoseconds): 8878548
```

Wykresy

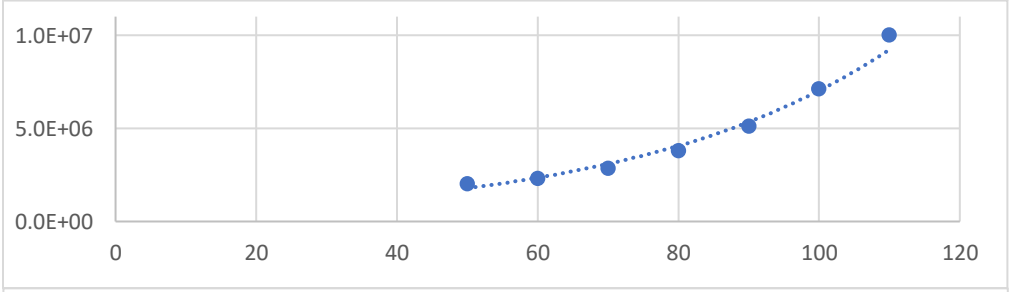
Gęstość

Macierz sąsiedzictwa

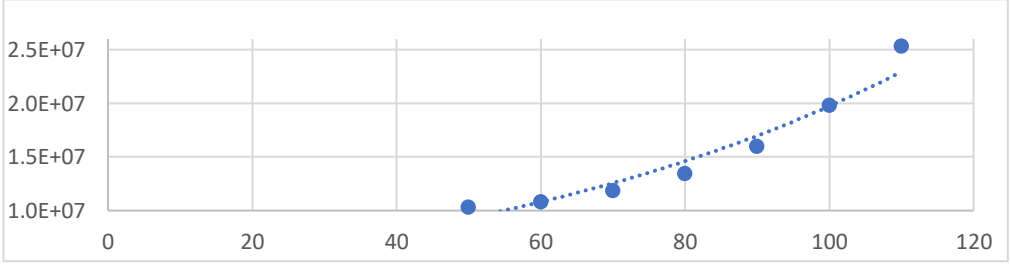
20%



60%



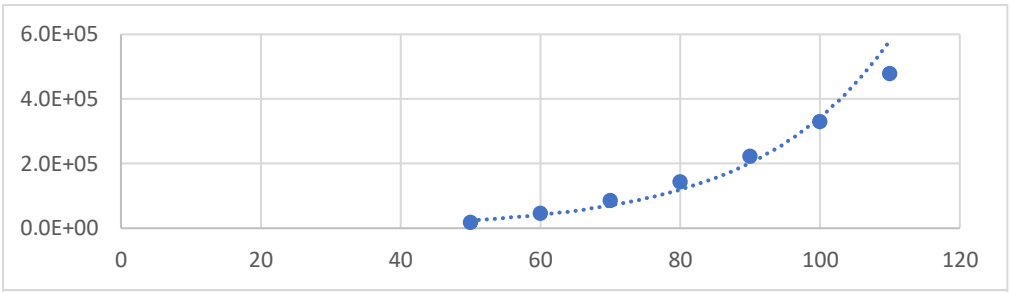
99%



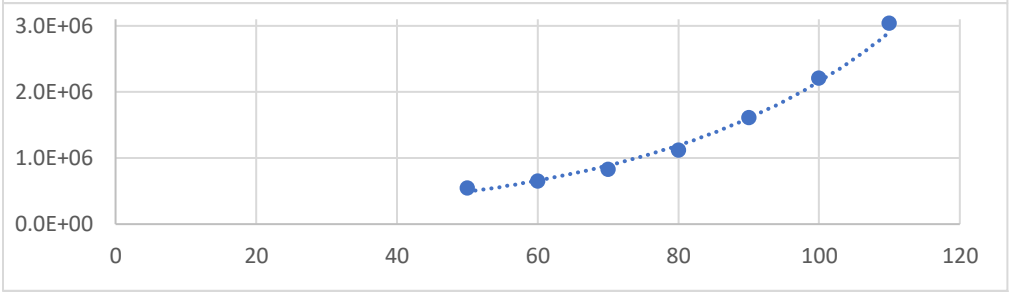
Gęstość

Lista sąsiadów

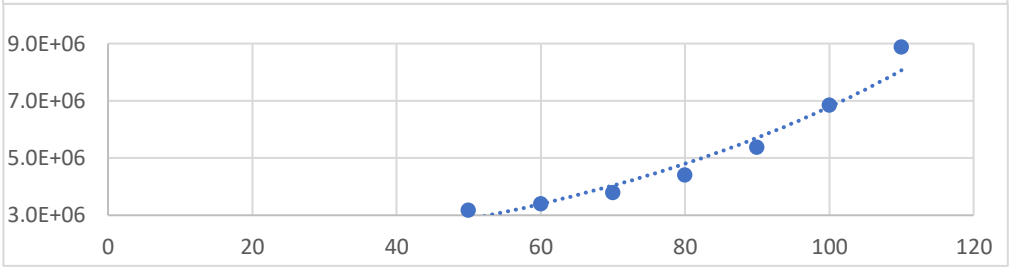
20%



60%



99%



Wnioski

Zarówno algorytm Prima,jak i algorytm Kruskala są algorytmami wyszukiwania MST. W przypadku algorytmu Prima graf musi być połączony, jednak nie jest to prawdą w przypadku algorytmu Kruskala.

Z wzrostem gęstości bardzo wzrasta czas wykonania algorytmu, dlatego że teoretyczna złożoność algorytmu na wprost zależy od ilości krawędzi.

Lista sąsiadów działa szybcij, bo nie mamy robić niepotrzebnych iteracji jak w macierzy sąsiedzictwa.

Problem najkrótrzej ścieżki

Jest to zagadnienie polegające na znalezieniu w grafie ważonym najkrótszego połączenia pomiędzy danymi wierzchołkami. Szczególnymi przypadkami tego problemu to problem najkrótszej ścieżki od jednego wierzchołka do wszystkich innych oraz problem najkrótszej ścieżki pomiędzy wszystkimi parami wierzchołków.

W przypadku pesymistycznym do wyznaczenia optymalnej ścieżki z wierzchołka A do wierzchołka B konieczne jest wyznaczenie najkrótszych ścieżek z wierzchołka A do wszystkich pozostałych wierzchołków w grafie. Zagadnienie takie jest określane jako poszukiwanie najkrótszych ścieżek z jednego źródła. Do rozwiązywania tego zagadnienia będziemy wykorzystywali następujące algorytmy:

- Algorytm Dijkstry
- Algorytm Bellmana-Forda

Aby znalezienie najkrótszej ścieżki było możliwe, graf nie może zawierać ujemnych cykli osiągalnych z wierzchołka źródłowego. Jeśli taki cykl istnieje, to poruszając się nim „w kółko” cały czas zmniejszamy długość ścieżki. Dopuszczalne jest natomiast występowanie krawędzi o ujemnej wadze, choć nie wszystkie algorytmy dopuszczają ten przypadek.

Algorytm Dijkstry

Algorytm Dijkstry służy do wyznaczania najmniejszej odległości od ustalonego wierzchołka *s* do wszystkich pozostałych w grafie skierowanym. W algorytmie tym pamiętany jest zbiór Q wierzchołków, dla których nie obliczono jeszcze najkrótszych ścieżek, oraz wektor D[i] odległości od wierzchołka *s* do *i*. Początkowo zbiór Q zawiera wszystkie wierzchołki a wektor D jest pierwszym wierszem macierzy wag krawędzi A.

Zastosowanie algorytmu:

- Występuje na mapach geograficznych.
- Znalezienie lokalizacji mapy, która odnosi się do wierzchołków grafu.
- Odległość między lokalizacją odnosi się do krawędzi.
- Jest używany w trasowaniu IP, aby najpierw znaleźć otwartą najkrótszą ścieżkę.
- Stosowany jest w sieci telefonicznej.

Złożoność obliczeniowa: $O(|E|+|V|\log|V|)$, pamęciowa: $O(|V|)$.

Założenia eksperymentu:

- Testujemy algorytm dla dwóch reprezentacji grafu. Graf zawsze się generuje w postaci macierzowej. W przypadku testowania listy sąsiedzictwa, graf przekształcamy w tą formę po zgenerowaniu.
- Nie liczymy czas generacji oraz przepisywanie grafu z jednej postaci do drugiej. Genracja grafu prepływa s następujący sposób: generujemy drzewo rozpinające, dalej w zależności od gęstości dodajemy krawędzie.
- Testujemy 7 punktów obserwowanych: 50, 60, 70, 80, 90, 100, 110 wierzchołków dla trzech gęstości: 20%, 60%, 99%.
- Mierzymy czas 100 prób, zatem uśredniamy wynik.
- Graf jest skirowany. Znajdujemy scieżkę do wszystkich punktów od zerowego

Log programy

Macierz sąsiedzictwa:

```
Speed test of Dijkstra's algorithm.
Adjacency matrix:
Density: 20%
Size: 50 result(nanoseconds): 3996
Size: 60 result(nanoseconds): 8991
Size: 70 result(nanoseconds): 15982
Size: 80 result(nanoseconds): 22974
Size: 90 result(nanoseconds): 33968
Size: 100 result(nanoseconds): 48954
Size: 110 result(nanoseconds): 69934
Density: 60%
Size: 50 result(nanoseconds): 73930
Size: 60 result(nanoseconds): 79923
Size: 70 result(nanoseconds): 86935
Size: 80 result(nanoseconds): 94925
Size: 90 result(nanoseconds): 104915
Size: 100 result(nanoseconds): 116903
Size: 110 result(nanoseconds): 132905
Density: 99%
Size: 50 result(nanoseconds): 136900
Size: 60 result(nanoseconds): 145890
Size: 70 result(nanoseconds): 171868
Size: 80 result(nanoseconds): 186850
Size: 90 result(nanoseconds): 197839
Size: 100 result(nanoseconds): 210826
Size: 110 result(nanoseconds): 227811
```

Lista sąsiadów:

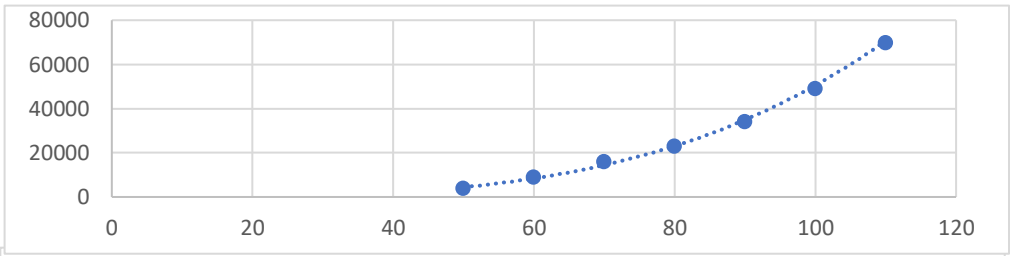
```
Speed test of Dijkstra's algorithm.
Adjacency list:
Density: 20%
Size: 50 result(nanoseconds): 1997
Size: 60 result(nanoseconds): 4995
Size: 70 result(nanoseconds): 8991
Size: 80 result(nanoseconds): 12986
Size: 90 result(nanoseconds): 18980
Size: 100 result(nanoseconds): 27971
Size: 110 result(nanoseconds): 37960
Density: 60%
Size: 50 result(nanoseconds): 40956
Size: 60 result(nanoseconds): 45968
Size: 70 result(nanoseconds): 51978
Size: 80 result(nanoseconds): 59988
Size: 90 result(nanoseconds): 68971
Size: 100 result(nanoseconds): 81966
Size: 110 result(nanoseconds): 96944
Density: 99%
Size: 50 result(nanoseconds): 101939
Size: 60 result(nanoseconds): 107952
Size: 70 result(nanoseconds): 115943
Size: 80 result(nanoseconds): 124932
Size: 90 result(nanoseconds): 135904
Size: 100 result(nanoseconds): 150891
Size: 110 result(nanoseconds): 168891
```

Wykresy

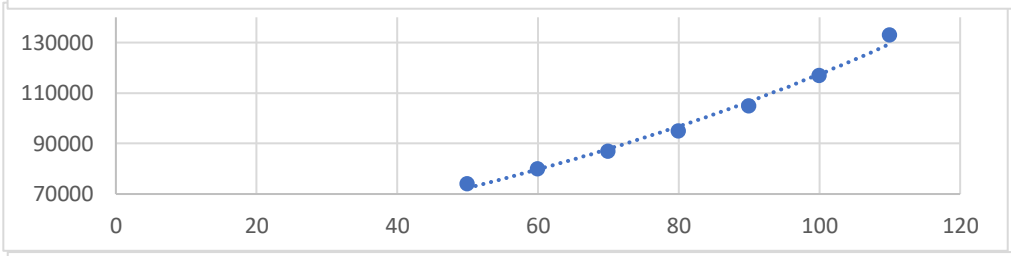
Gęstość

Macierz sąsiedzictwa

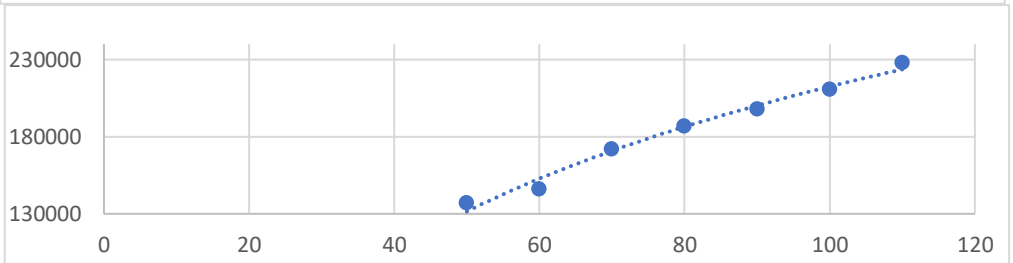
20%



60%



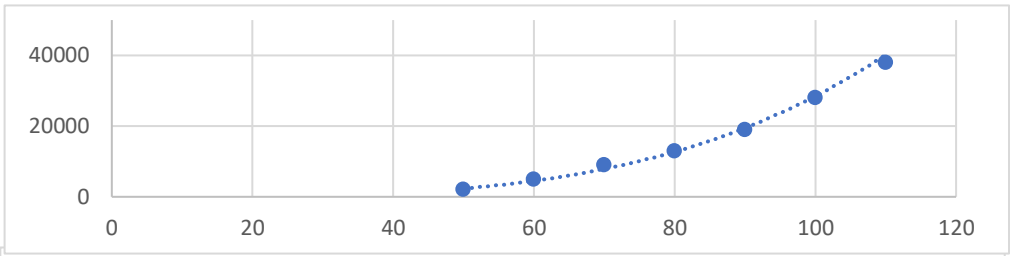
99%



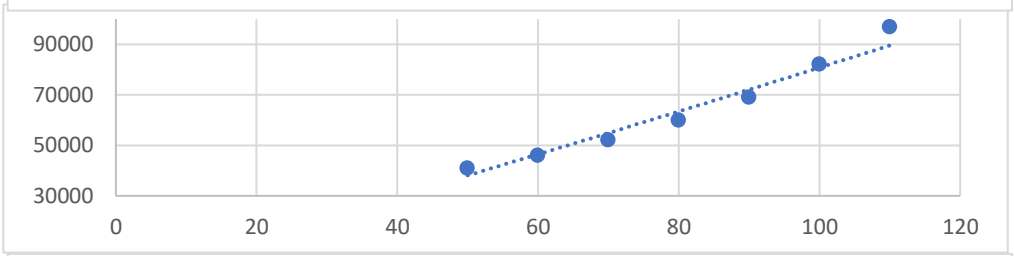
Gęstość

Lista sąsiadów

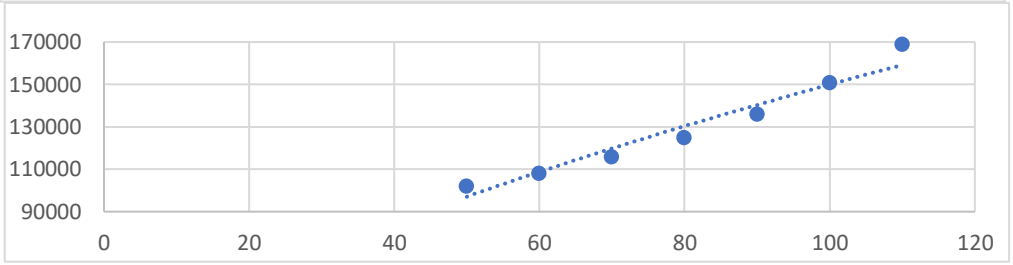
20%



60%



99%



Wnioski

Algorytm Dijkstry nie działa, jeśli w grafie występują krawędzie z ujemnymi wagami – w tym wypadku używa się wolniejszego, lecz bardziej ogólnego algorytmu Bellmana-Forda. Jeśli graf nie jest ważony (wszystkie wagi mają wielkość 1), zamiast algorytmu Dijkstry wystarczy algorytm przeszukiwania grafu wszerz. Algorytm A* jest pewnym uogólnieniem algorytmu Dijkstry, które pozwala przeszukiwać tylko część grafu, jednak wymaga dodatkowej wstępnej informacji (heurystyki) o odległościach wierzchołków. Algorytm Prima znajdowania minimalnego drzewa rozpinającego oparty jest o bardzo podobny pomysł co algorytm Dijkstry.

Zobaczmy co się dzieje z praktyczną złożonością obliczeniową w porównaniu z teoretyczną. Przy nizkiej gęstości graf jest bardzo bodopny na funkcję potęgową, bo na postać wykresu $V*logV$ ma większy wpływ. Podczas 60% gęstości wykres jest zbliżony do funkcji liniowej, w tym momencie obu składniki mają w przybliżeniu taki sam wpływ na wykres. Ale gdy gęstość jest 99%, wykres już jest bardzo podobny do funkcji liniowej. Tutaj decyduje duża ilość krawędzi i E ma większy wpływ.

Jak zawszę algorytm działa szybciej na liście sąsiadów.

Algorytm Bellmana-Forda

Algorytm ten służy do wyznaczania najmniejszej odległości od ustalonego wierzchołka *s* do wszystkich pozostałych w grafie skierowanym bez cykli o ujemnej długości.
Warunek nieujemności cyklu jest spowodowany faktem, że w grafie o ujemnych cyklach najmniejsza odległość między niektórymi wierzchołkami jest nieokreślona, ponieważ zależy od liczby przejść w cyklu.

Zastosowanie algorytmu:

- Występuje na mapach geograficznych.
- Znalezienie lokalizacji mapy, która odnosi się do wierzchołków grafu.
- Odległość między lokalizacją odnosi się do krawędzi.
- Jest używany w trasowaniu IP, aby najpierw znaleźć otwartą najkrótszą ścieżkę.
- Stosowany jest w sieci telefonicznej.

Złożoność obliczeniowa: $O(|E|*|V|)$, pamięciowa: $O(|V|)$.

Założenia eksperymentu:

- Testujemy algorytm dla dwóch reprezentacji grafu. Graf zawsze się generuje w postaci macierzowej. W przypadku testowania listy sąsiedzictwa, graf przekształcamy w tą formę po zgenerowaniu.
- Nie liczymy czas generacji oraz przepisywanie grafu z jednej postaci do drugiej. Genracja grafu prepływa s następujący sposób: generujemy drzewo rozpinające, dalej w zależności od gęstości dodajemy krawędzie.
- Testujemy 7 punktów obserwowanych: 50, 60, 70, 80, 90, 100, 110 wierzchołków dla trzech gęstości: 20%, 60%, 99%.
- Mierzymy czas 100 prób, zatem uśredniamy wynik.
- Graf jest skirowany. Znajdujemy scieżkę do wszystkich punktów od zerowego

Log programy

Macierz sąsiedztwa:

```
Speed test of Bellman-Ford algorithm.
Adjacency matrix:
Density: 20%
Size: 50 result(nanoseconds): 16988
Size: 60 result(nanoseconds): 43957
Size: 70 result(nanoseconds): 83932
Size: 80 result(nanoseconds): 141878
Size: 90 result(nanoseconds): 223812
Size: 100 result(nanoseconds): 341747
Size: 110 result(nanoseconds): 496654
Density: 60%
Size: 50 result(nanoseconds): 518648
Size: 60 result(nanoseconds): 552596
Size: 70 result(nanoseconds): 603542
Size: 80 result(nanoseconds): 678476
Size: 90 result(nanoseconds): 784399
Size: 100 result(nanoseconds): 933297
Size: 110 result(nanoseconds): 1140160
Density: 99%
Size: 50 result(nanoseconds): 1167148
Size: 60 result(nanoseconds): 1211103
Size: 70 result(nanoseconds): 1278061
Size: 80 result(nanoseconds): 1375001
Size: 90 result(nanoseconds): 1512890
Size: 100 result(nanoseconds): 1689771
Size: 110 result(nanoseconds): 1940624
Speed test of Bellman-Ford algorithm.
```

Lista sąsiadów:

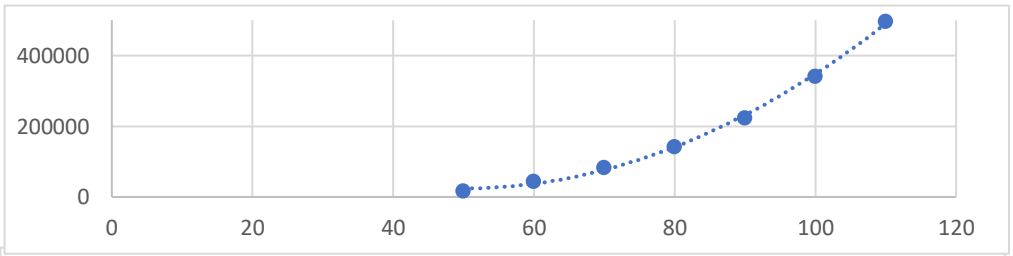
```
Speed test of Bellman-Ford algorithm.
Adjacency list:
Density: 20%
Size: 50 result(nanoseconds): 6995
Size: 60 result(nanoseconds): 15988
Size: 70 result(nanoseconds): 31971
Size: 80 result(nanoseconds): 50936
Size: 90 result(nanoseconds): 77911
Size: 100 result(nanoseconds): 117890
Size: 110 result(nanoseconds): 164840
Density: 60%
Size: 50 result(nanoseconds): 179825
Size: 60 result(nanoseconds): 203818
Size: 70 result(nanoseconds): 235782
Size: 80 result(nanoseconds): 288750
Size: 90 result(nanoseconds): 360684
Size: 100 result(nanoseconds): 463627
Size: 110 result(nanoseconds): 602548
Density: 99%
Size: 50 result(nanoseconds): 624546
Size: 60 result(nanoseconds): 657535
Size: 70 result(nanoseconds): 718483
Size: 80 result(nanoseconds): 793436
Size: 90 result(nanoseconds): 900355
Size: 100 result(nanoseconds): 1067236
Size: 110 result(nanoseconds): 1292090
```

Wykresy

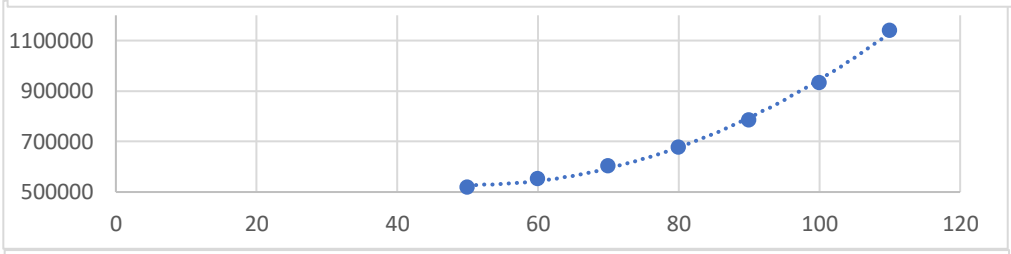
Gęstość

Macierz sąsiedztwa

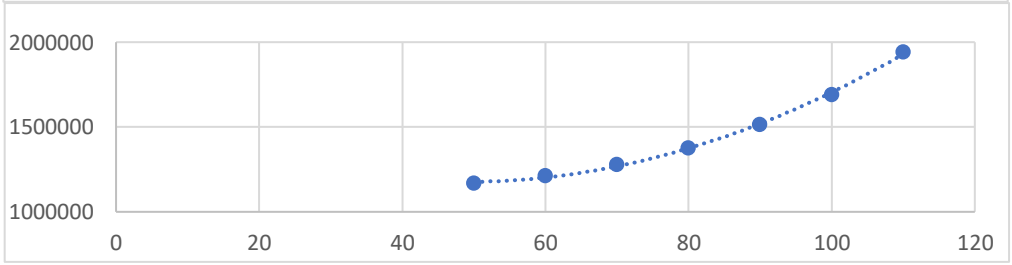
20%



60%



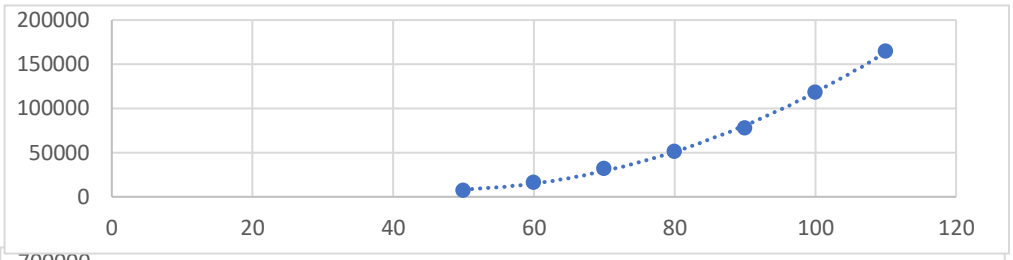
99%



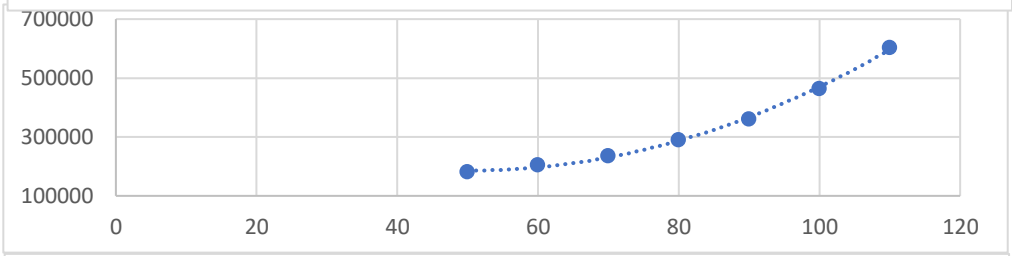
Gęstość

Lista sąsiadów

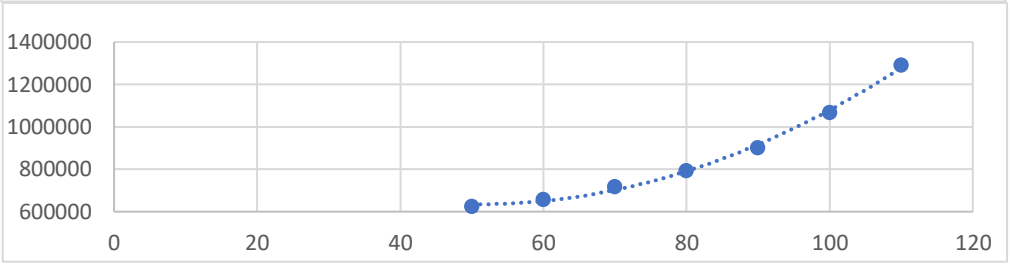
20%



60%



99%



Wnioski

Ujemne wagi krawędzi rzeczywiście nie są zbyt użyteczne w trasowaniu zastosowań algorytmu. Zdolność do obsługi wag ujemnych jest jedną z własności, w której Bellman-Ford różni się od Dijkstry, ale nie jest to powód, dla którego jest preferowana w algorytmach wektorowych odległości. Uruchomienie algorytmu Dijkstry wymaga pełnej znajomości wszystkich krawędzi i wag w sieci. W przeciwieństwie do algorytmów trasowania Stanów łącza, algorytmy wektorowe odległości nie konstruują takiej pełnej mapy sieci. Oznacza to, że algorytm Dijkstry nie może być stosowany w algorytmach wektorowych odległości. (Rozproszony) Bellman-Ford działa bez pełnego widoku sieci i tym samym może być wykorzystywany w algorytmach wektorowych odległości.

Na wykresach widać że ze wzrostem gęstości grafu dramatycznie rośnie czas wykonania algorytmu. Wzrost czasu jest tak duży bo w odróżnieniu od algorytmu Dijkstry ilość krawędzi w algorytmie Bellmana-Forda mnoży się.

Literatura

https://pl.qwe.wiki/wiki/Minimum_spanning_tree

http://home.agh.edu.pl/~zobmat/2017/2_tarkowskijakub/teoria.php

https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

https://www.tutorialspoint.com/data_structures_algorithms/kruskals_spanning_tree_algorithm.htm

https://www.hackerearth.com/blog/developers/kruskals-minimum-spanning-tree-algorithm-example/