Politechnika
Wrocławska

Algorytmy i złożoność obliczeniowa: projekt nr.2

Analiza wydajności wybranych algorytmów grafowych rozwiązujących problemy minimalnego drzewa rozpinającego oraz najkrótszej ścieżki w zależności od reprezentacji grafu i jego gęstości

Kytrysh Andrii, 275770

Wrocław 2025

Spis treści

1 Cel projektu

Celem danego projektu jest zapoznanie się z algorytmami, które rozwiązują problematy wyszukiwania MST oraz najkrótszej ścieżki, wykorzystując różne implementacje grafów takie jak lista sąsiedztwa i macierz incydencji , i zbadanie działania tych algorytmów w przybliżonych do rzeczywistości warunkach. W tym zbadać i przeanalizować wpływ wielkości grafów, ich gęstości oraz implementacji, w której zostały zapisane dla przyszłego wykorzystania podczas pracy algorytmu. Celem projektu zarówno jest napisanie implementacji algorytmów oraz dodatkowego oprogramowania do obsługi badań czasu wykonania algorytmów, zapisywania wyników sortowania, odczytu danych wejściowych z pliku, generowania danych wejściowych o różnym rozmiarze grafa i jego gęstości, zapisu wyniku działań algorytmów do pliku wraz z demonstracją wyniku w konsoli oraz podania najkrótszej ścieżki z jej wagą albo MST z jego wagą.

2 Wstęp

W tym projekcie będziemy badali różne algorytmy grafowe na rosnącym ciągu rozmiaru danych wejściowych oraz gęstości grafów wejściowych. Zbadane zostaną grafy w następnym zakresie wielkości wierzchołków: 25; 50; 75; 100; 125; 150; 175; 250; 500; 750; 1000. Taki wybór jest przybliżeniem do rzeczywistości. Na przykład, kiedy algorytm musi wyszukać najkrótszą ścieżkę z punktu A do B, na przykład w dużym mieście jak Nowy Jork, gdzie jest 472 stacji metro i jeszcze ponad 300 przystanków autobusowych, co już stanowi 772 możliwych przystanków do odwiedzenia. Najczęściej podróżujemy na krótkich odcinkach, przez co badania dla 500, 750 oraz 1000 wierzchołków stanowili interes techniczny a nie praktyczny. Jak się okaże podczas badań, nie uda się o silić wszystkich pomiarów na rozmiarze grafów 500+ wierzchołków oraz gęstości 50%+ przez ograniczenia wydajnościowe komputera oraz mało zoptymalizowaną implementację niektórych struktur danych oraz algorytmów.

3 Problem wyszukiwania MST

Minimalne drzewo rozpinające (MST) to podzbiór krawędzi grafu skierowanego lub nieskierowanego łączący wszystkie wierzchołki tak, żeby suma wag krawędzi była minimalna. MST znajduje zastosowanie m.in. w projektowaniu sieci telekomunikacyjnych, dróg czy systemów elektroenergetycznych, gdzie chcemy połączyć wszystkie punkty przy jak najniższym koszcie. W projekcie będziemy traktowali graf jako nieskierowany do rozwiązania probleatu wyszukiwania MST.

3.1 Algorytm Prima

Algorytm Prima zaczyna od dowolnego wierzchołka i przyłącza stopniowo do rosnącego drzewa najmniej kosztowną krawędź wychodzącą z już zbudowanego fragmentu. W wersji na liście sąsiedztwa (PrimList) w każdej iteracji przeglądamy tylko krawędzie wychodzące z wierzchołków, które już są w drzewie, co w najgorszym przypadku (bez użycia kolejki priorytetowej) daje złożoność

$$O(V^2).$$

Gdy jednak reprezentujemy graf macierzą incydencji (PrimMatrix), każdorazowo skanujemy wszystkie E krawędzi, przez co czas działania rośnie do

$$O(V \cdot E).$$

Praktycznie PrimList jest zazwyczaj szybszy na grafach rzadkich, natomiast PrimMatrix traci na wydajności właśnie ze względu na „globalne” skanowanie wszystkich krawędzi.

3.2 Algorytm Kruskala

Kruskal operuje na płaskiej liście wszystkich krawędzi: najpierw sortuje je rosnąco według wag, a następnie dodaje krawędzie kolejno do budowanego lasu przy pomocy struktury Union–Find, unikając w ten sposób tworzenia cykli. Etap sortowania zajmuje

$$O(E \log E),$$

a dzięki efektywnej strukturze zbiorów rozłącznych (Union–Find) kolejne operacje łączenia i znajdowania reprezentantów wykonują się w $\alpha(n)$–amortyzowanym czasie, co w sumie daje złożoność pamięciową

$$O(V + E)$$

oraz czasową praktycznie zdominowaną przez sortowanie krawędzi. KruskalList i KruskalMatrix różnią się jedynie sposobem wczytywania oryginalnych krawędzi, ale sam mechanizm sortowania i łączenia zbiorów jest identyczny.

4 Problem najkrótszej ścieżki

W problemie najkrótszej ścieżki chcemy dla zadanego wierzchołka początkowego znaleźć najtańszą (najkrótszą) ścieżkę do każdego innego wierzchołka grafu z uwzględnieniem wag krawędzi.

4.1 Algorytm Dijkstry

Algorytm Dijkstry korzysta z kolejki priorytetowej (min-heap), aby w każdej iteracji wyciągnąć wierzchołek o minimalnej odległości od źródła, a następnie zrelaksować wszystkie krawędzie wychodzące z tego wierzchołka. Nie pracuje z grafami które mają cykle ujemne. W optymalnej implementacji na liście sąsiedztwa (z kolejką priorytetową) osiąga złożoność

$$O\big((V + E) \log V\big).$$

Jeżeli natomiast nie używamy kolejki i „ręcznie" skanujemy wszystkie wierzchołki w poszukiwaniu najmniejszej odległości, dostajemy w wersji listowej:

$$O(V^2),$$

a w wersji macierzowej (skanowanie wszystkich E krawędzi w każdej z V iteracji):

$$O(V \cdot E).$$

4.2 Algorytm Bellmana–Forda

Algorytm Bellmana–Forda nie wymaga braku ujemnych cykli w grafie i działa poprzez relaksację wszystkich krawędzi w kolejnych iteracjach. W każdej z $V - 1$ iteracji skanuje całą listę E krawędzi, dzięki czemu deterministycznie osiąga złożoność

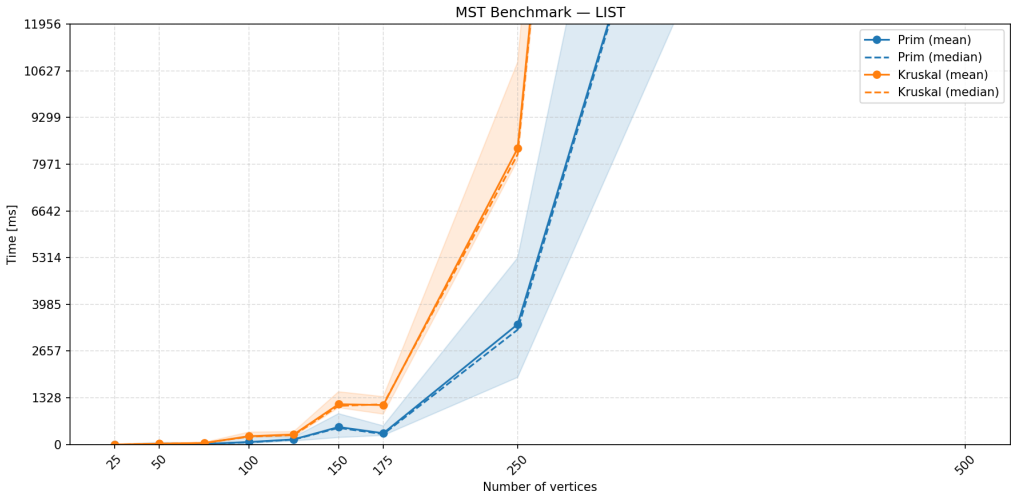
$$O(V \cdot E).$$

Dodatkowo po tych $V - 1$ krokach można wykonać jeszcze jedną rundę relaksacji, aby wykryć obecność ujemnych cykli w grafie.

5 Wpływ rozmiaru grafu na wydajność algorytmów

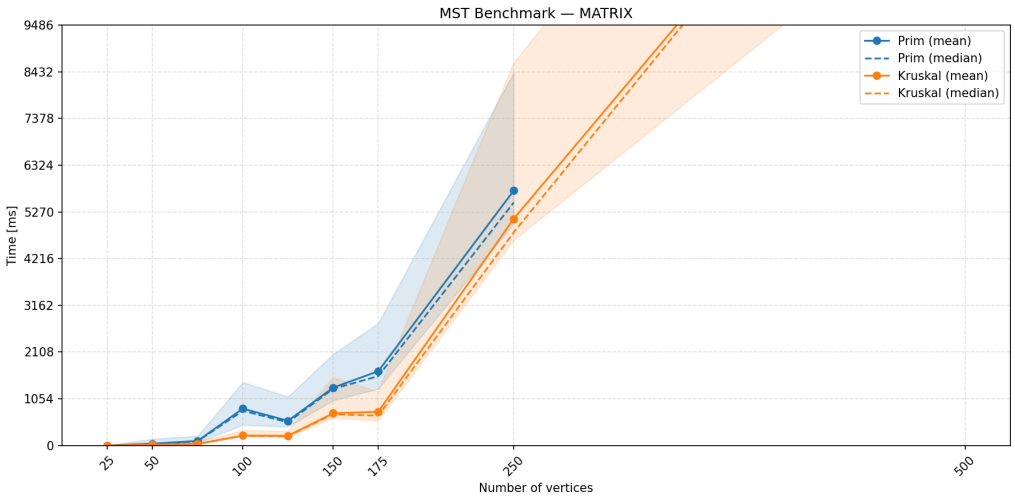
W tym rozdziale przedstawiono wyniki pomiarów czasu działania algorytmów rozwiązujących problemy wyszukiwania MST oraz najkrótszej ścieżki na wykresach zależności średniego czasu od liczby wierzchołków. Każdy punkt na wykresach to uśredniona wartość z serii testów na grafach o danej liczbie wierzchołków i różnych gęstościach krawędzi $25\%, 50\% oraz 99\%$.

Na rysunku [1] widać, że dla małych grafów (n 100) zarówno algorytm Prima, jak i Kruskala wykonują się w czasie rzędu kilkunastu-kilkuset milisekund. Dopiero przy przejściu do około 150 wierzchołków obserwujemy zauważalny wzrost koszt. Oba algorytmy zaczynają rosnąć niemal wykładniczo. Dla n=250 różnica staje się dramatyczna: KruskalList osiąga kilku tysięcy milisekund, a PrimList przekracza 2–3 sekundy, co odzwierciedla wyższą złożoność $O(V^2)$ w przypadku Prima i reprezentacji grafu w postaci listy sąsiedztwa.



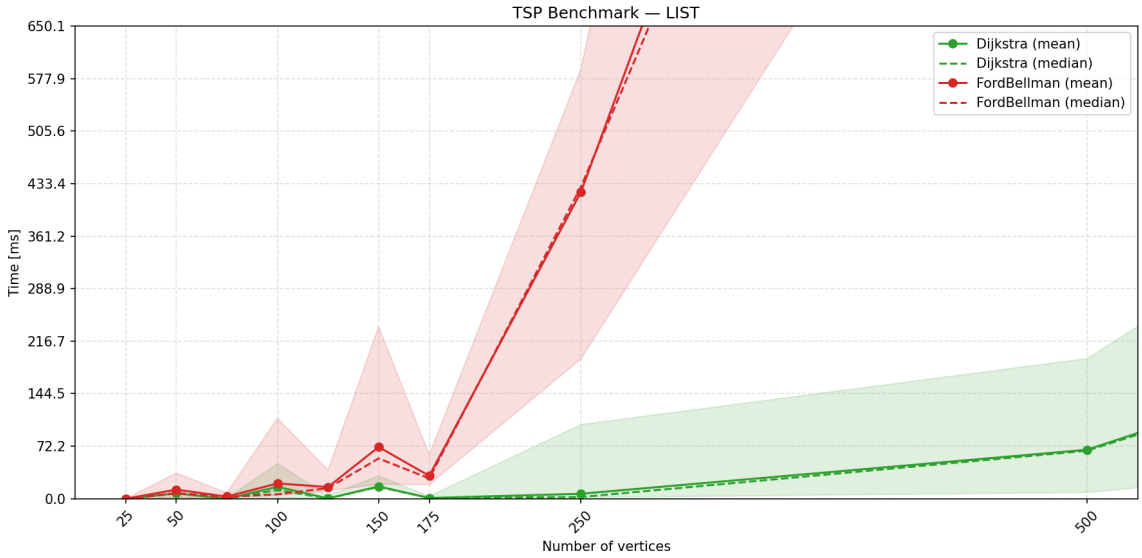
Rysunek 1: Wpływ ilości wierzchołków na wydajność algorytmów dla reprezentacji grafa w postaci listy sąsiedztwa

W reprezentacji macierzowej [2] wzorzec jest podobny, jednak czasy są generalnie wyższe od około 150 wierzchołków. PrimMatrix oraz KruskalMatrix rosną tu podobnie, ale macierzowa wersja Prima $O(V \cdot E)$ zaczyna być wyraźnie wolniejsza niż KruskalMatrix $O(V + E)$ w każdym punkcie.



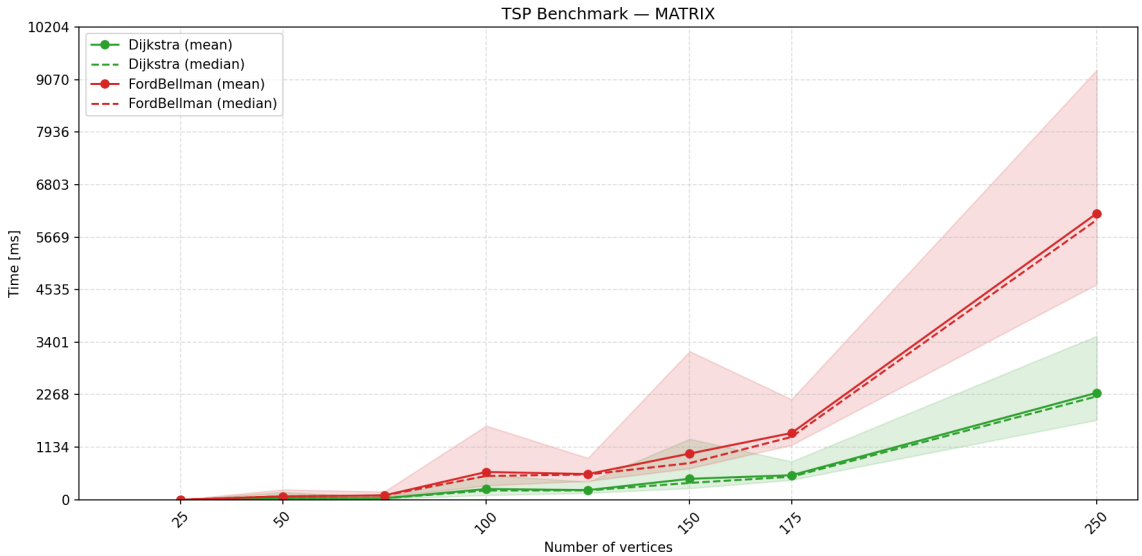
Rysunek 2: Wpływ ilości wierzchołków na wydajność algorytmów dla reprezentacji grafa w postaci macierzy incydencji

Na wykresie dla problemu najkrótszej ścieżki [3] widać, że algorytm Dijkstry (lista + kolejka priorytetowa) zachowuje się bardzo efektywnie i dla większości rozmiarów grafu utrzymuje czasy poniżej kilkudziesięciu milisekund. Z kolei Bellman–Ford, złożony $O(V \cdot E)$, zaczyna wyraźnie odstawać już od n=100, a przy n=250 osiąga kilkaset milisekund do kilku sekund, co idealnie odzwierciedla jego liniowo–iloczynową złożoność. Kolejka priorytetowa w algorytmie Dijkstry została zaimplementowana własnoręcznie, korzystając z własnej implementacji kopca.



Rysunek 3: Wpływ ilości wierzchołków na wydajność algorytmów dla reprezentacji grafa w postaci listy sąsiedztwa

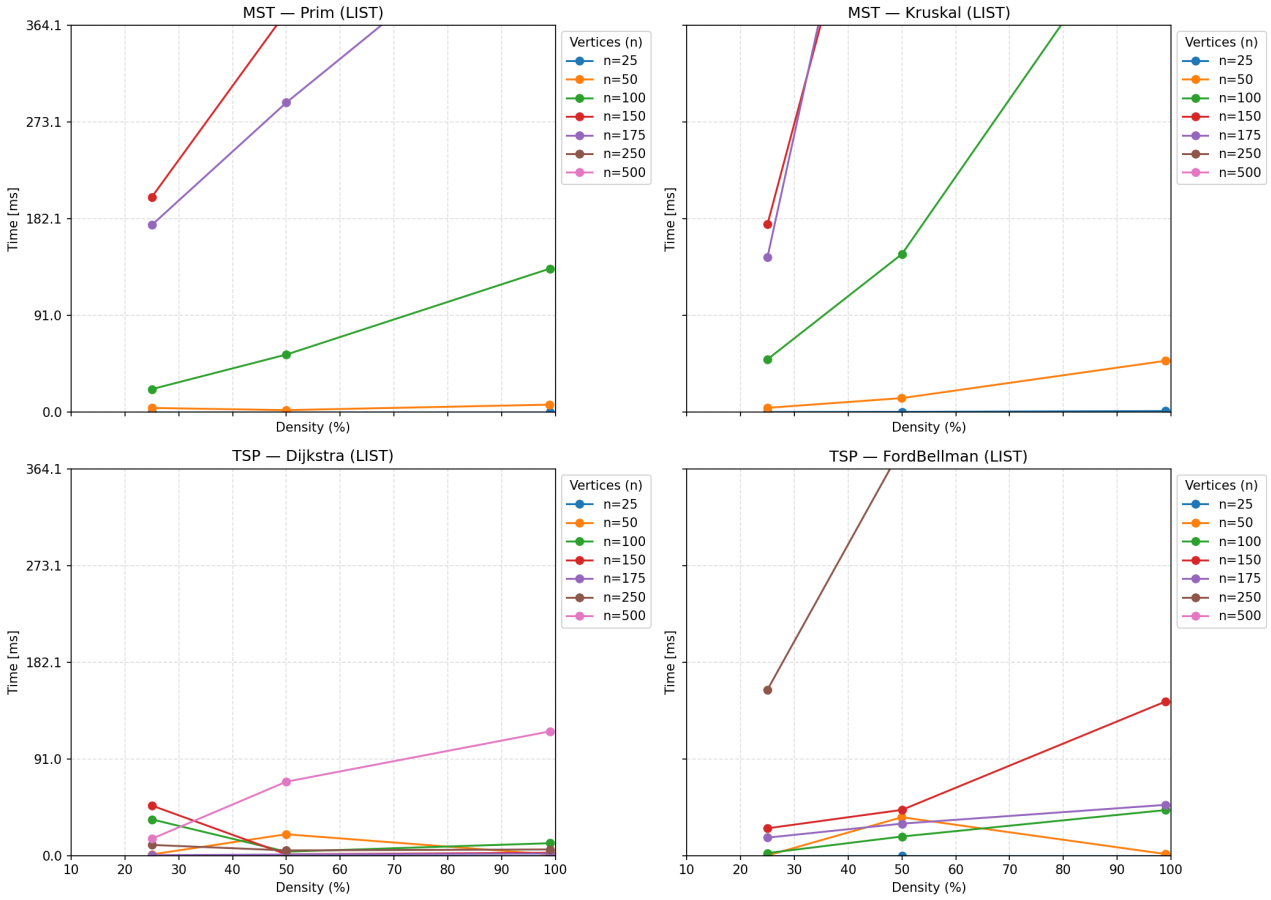
Reprezentacja macierzowa [4] ponownie uwydatnia spadek wydajności Bellman–Forda – jego wykresy są najwyższe ze wszystkich, szczególnie na n=250. Algorytm Dijkstry w wersji macierzowej pozostaje na stosunkowo niskim poziomie czasu (do kilkuset milisekund), ale widać, że także on traci na wydajności wobec rosnącej liczby krawędzi skanowanych w każdej iteracji.



Rysunek 4: Wpływ ilości wierzchołków na wydajność algorytmów dla reprezentacji grafa w postaci macierzy incydencji

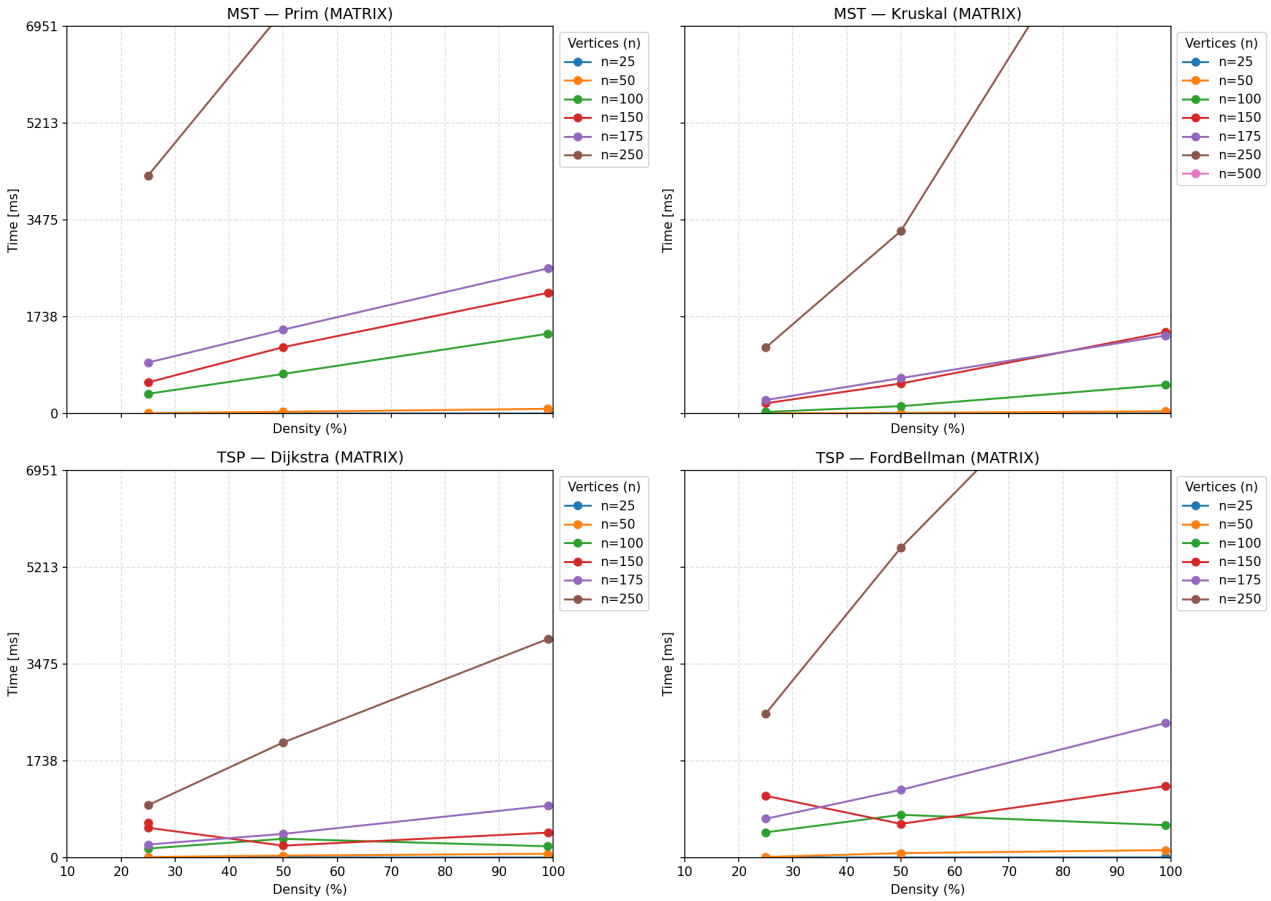
6 Wpływ gęstości grafu na wydajność algorytmów

W tej sekcji analizujemy, jak zmienia się średni czas działania algorytmów wyszukiwania MST i najkrótszej ścieżki (TSP) wraz ze wzrostem gęstości krawędzi, przy stałej liczbie wierzchołków. Wyniki przedstawiono osobno dla reprezentacji listowej [5] oraz macierzowej [6]. Każda krzywa odpowiada innemu rozmiarowi grafu n (ilość wierzchołków) i pokazuje, jak rozkład czasu zależy od procentowego udziału krawędzi w grafie.



Rysunek 5: Wpływ gęstości na czas działania algorytmów na liście sąsiedztwa

Na wykresach dla listy sąsiedztwa (rys. 5) jest wyraźnie widać, że algorytm PrimList i KruskalList mają zupełnie różne wrażliwości na gęstość. Dla niewielkich grafów (np. $n = 25, 50$) czasy pozostają niemal stałe niezależnie od udziału krawędzi — jest to efekt małego E, które nawet przy gęstym grafie nie obciąża znacznie pętli głównej. W miarę wzrostu n (100–175) obserwujemy niemal liniowy wzrost czasu Kruskala w funkcji gęstości, podczas gdy Prim przy tych samych wielkościach grafów rośnie znacznie łagodniej. Dla $n = 250$ (brązowa krzywa) oba algorytmy rosną szybko, lecz PrimList powinien rosnąć szybciej, co wynika z potrzeby wyszukiwania minimalnej krawędzi z każdego wierzchołka w drzewie (złożoność $O(V^2)$), podczas gdy KruskalList jednorazowo sortuje listę krawędzi ($O(E \log E)$), a sama gęstość jedynie zwiększa E, nie wpływając bezpośrednio na pętlę główną algorytmu po sortowaniu. Niestety nie jest ten efekt, wyprzedzającego wzrostu, widoczny na wykresie. Może to być spowodowane złą implementacją algorytmu lub, co jest bardziej wiarygodne, badanie były odpalone na 4 komputerach jednocześnie dla badania kilku algorytmów na raz. Każdy komputer miał częstotliwość procesora w zakresie 3-4.8 GHz. Taki rozrzut mógł spowodować pomiar, który byłby słabo porównany z innymi przez to, że został wykonany na fizycznie wolniejszym komputerze.



Rysunek 6: Wpływ gęstości na czas działania algorytmów na macierzy incydencji

W reprezentacji macierzowej [6] czasy działania rosną jeszcze bardziej stromo. PrimMatrix przegląda wszystkie krawędzie grafu w każdej iteracji, więc jego krzywe rosną z przyspieszeniem wraz z gęstością — szczególnie widoczne dla $n \geqslant 175$ oraz $n \geqslant 250$. KruskalMatrix z kolei, mimo że także sortuje całą listę krawędzi, działa nieco wolniej od wersji listowej sąsiadów, gdyż dostęp do macierzy jest mniej lokalny i koszty pamięci podręcznej są wyższe. Dodatkowo algorytm Bellman–Ford w wersji macierzowej odnotowuje gwałtowny wzrost dla większych n i większej gęstości — każda dodatkowa krawędź mnoży się przez liczbę wierzchołków w czasie działania $O(V \cdot E)$. Algorytm Dijkstry na macierzy również rośnie liniowo z gęstością, lecz jego nachylenie jest znacznie mniejsze niż Bellman–Ford, co dobrze wpisuje się w jego złożoność $O(V^2)$ przy wsparciu listy sąsiedztwa, a w wersji macierzowej – $O(V \cdot E)$. Podsumowując, wychodząc z teorii musimy zobaczyć prawie identyczne wykresy Dijktri Matrix co i jest obserwowane. Jak widać na wykresach szybkość wzrostu czasu jest podobna a wykresy prawie się nakładają na siebie.

7 Wnioski

W ramach tego projektu zaimplementowano i przetestowano cztery warianty algorytmów wyznaczania minimalnego drzewa rozpinającego (PrimList, PrimMatrix, KruskalList, KruskalMatrix) oraz dwa algorytmy wyszukiwania najkrótszej ścieżki (DijkstraList, Bellman–FordList i ich odpowiedniki macierzowe). Każda implementacja została zweryfikowana pod kątem poprawności działania oraz pozbawiona wycieków pamięci, co potwierdziły testy debuggera. Przygotowano również skrypt automatyzujący generowanie losowych grafów o zadanych rozmiarach i gęstości, pomiary czasu wykonania oraz wizualizację wyników.

Analiza wykresów „Time vs Number of vertices" pokazała, że czasy działania algorytmów rosną zgodnie z teoretycznymi złożonościami. PrimList i PrimMatrix wykazują wzrost kwadratowy lub quasi-kwadratowy (kolejno $O(V^2)$ i $O(V \cdot E)$), co uwidacznia się szczególnie dla dużych n. KruskalList i KruskalMatrix natomiast skalują się znacznie łagodniej dzięki jednorazowemu sortowaniu krawędzi ($O(E \log E)$), a następnie łączeniu zbiorów.

W kontekście najkrótszej ścieżki algorytm DijkstraList (implementowany z kolejką priorytetową) okazał się wydajniejszy od macierzowej wersji ze względu na ograniczenie przeszukiwania do faktycznych sąsiadów wierzchołków. Bellman–Ford, mimo że niezawodny nawet dla ujemnych wag, rośnie w czasie liniowo względem iloczynu $V \cdot E$, co w gęstych lub dużych grafach czyni go znacznie wolniejszym od Dijkstry. Na wykresach dla problemu wyszukiwania najkrótszej szczeki widać, że różnica w nachyleniu krzywych obu algorytmów dobrze koreluje ze złożonościami $O((V + E)logV)$ i $O(V \cdot E)$.

Badania wpływu gęstości krawędzi utwierdziły w przekonaniu, że algorytmy jednorazowo sortujące krawędzie (Kruskal) są najmniej wrażliwe na zwiększenie liczby krawędzi — ich czasy rosną proporcjonalnie do dodatkowego kosztu sortowania, ale nie mnożą się przez kolejne iteracje. PrimMatrix i Bellman–FordMatrix natomiast w każdym kroku „przeczesują" całą macierz incydencji, dlatego dla gęstych grafów stają się praktycznie nieużyteczne. Wersje listowe tych samych algorytmów zachowują się znacznie lepiej, bo operują wyłącznie na istniejących krawędziach.

Podsumowując, otrzymane wyniki są spójne z teorią: algorytmy o złożoności $O(E \log E)$ (Kruskal) lub $O((V + E) \log V)$ (Dijkstra z PQ) powinny być preferowane w praktyce nad prostymi kwadratowymi wariantami Prima czy Bellmana–Forda. Implementacje bez zaawansowanych kolejek priorytetowych i z typowymi strukturami danych pozwoliły jednak zweryfikować ich zachowanie w warunkach przybliżonych do rzeczywistych zastosowań oraz zaplanować dalsze optymalizacje, np. wprowadzenie Fibonacci dla Dijkstry. Opisywane w dokumentacji czasy stanowią uśrednione wartości z 50 niezależnych pomiarów dla każdej konfiguracji, co zapewnia 1350 unikalnych grafów w sumie i wiarygodność uzyskanych obserwacji.

Trudności z którymi się spotkało. Czas. Generowanie takiej dużej puli grafów zajęło sumarycznie 40 godzin. I chociaż chciałem wygenerować grafy o mniejszych rozmiarach, pytanie: 'A co jeśli' wygrało. Badania zajęły kolejne 24-25 godzin. Przy tym nie zostały do końca zbadane grafy na 500, 750, 1000 wierzchołków dla tego zostały częściowo wyłączeni z analizy w sprawozdaniu ale są w rekordach benchmarku. Wynika takie 'niedopracowanie' z złożoności algorytmów na reprezentacji macierzowej grafu. Mianowicie te grafy w reprezentacji macierzowej nie zostały do końca zbadane dla powyższych wielkości grafów a jedno przejście algorytmem na macierzowej reprezentacji dla grafu 750+ wierzchołków dla gęstowiu 99% zajmowało niecałe 3 minuty. Kolejnym problemem który został stworzony to jest zaburzenie pomiarów pojedynczymi pomiarami nie specjalnie wolniejszymi. Tak się stało przez już wspomniany sposób prowadzenia badań na 4 różnych komputerach 3 z których mieli taką samą częstotliwość procesora 4.3 GHz ale tylko jeden miał 3.2 GHz i obliczał on małe grafy. To spowodowało zaburzenie wykresów na ich początku dla grafów o większości do 150 wierzchołków włącznie. Wszystkie sortowania które były zapotrzebowane w implementacjach algorytmów były zrobione za pomocą wcześniej zaimplementowanego quick Sort -a z projektu nr.1.. Dodatkowo kolejka priorytetowa została zaimplementowana samodzielnie na bazie też już istniejącego kopca MAX z projektu nr.1. Projekt nr.2 stał kontynuacją projektu nr.1.