Projektowanie efektywnych algorytmów

Projekt

22/10/2024

Ivan Hancharyk 264511

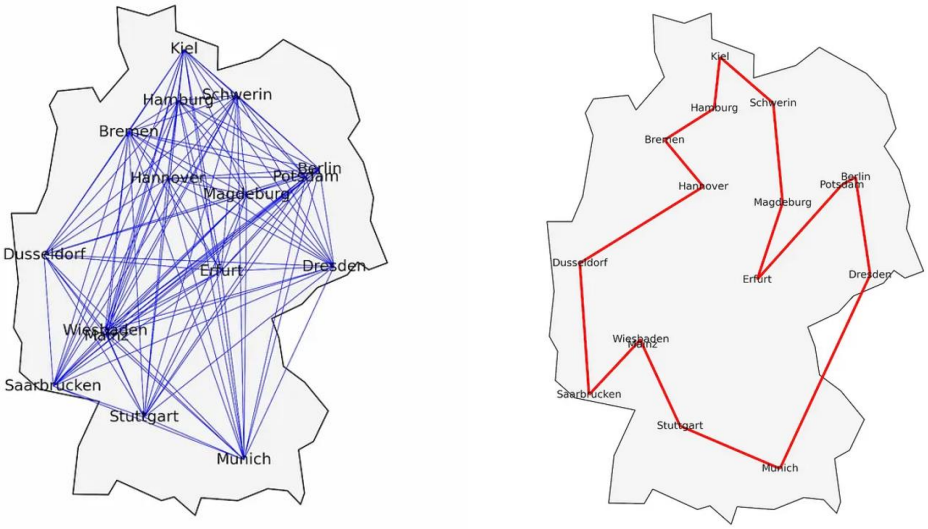
Zadania 1 i 2: brute-force, nearest-neighbor, random, Branch and Bound.

Spis treści

1. Sformułowanie zadania głównego

Problem komiwojażera polega na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Cykl w którym każdy wierzchołek jest odwiedzany dokładnie raz, oprócz początkowego, nazywa się cyklem Hamiltona.

Dobrym przykładem dla przedstawiania problemu w głowie może być potrzeba odwiedzenia każdej stolicy Niemieckich województw.



Rysunki 1 i 2: ścieżki pomiędzy stolicami wszystkich województw w Niemczech oraz rozwiązanie TSP dla podanego przykładu

Problem ten jest klasycznym przykładem problemu NP-trudnego, co oznacza, że dla dużych instancji nie jest znane rozwiązanie w czasie wielomianowym.

1.1. Sformułowanie zadania dla Projektu 1 (Metody BF, NN, Random)

Projekt 1 ma na celu rozwiązanie problemu komiwojażera przy użyciu trzech metod:

- **Metoda brute-force (BF)**, algorytm przeglądający wszystkie możliwe permutacje miast, gwarantujący znalezienie rozwiązania optymalnego. Charakteryzuje się złożonością **O(n!)**, co oznacza, że jego czas działania rośnie wykładniczo wraz ze wzrostem liczby miast.
- **Algorytm nearest-neighbor (NN)**, heurystyczna metoda rozpoczynająca od danego miasta i wybierająca w każdej iteracji najbliższe nieodwiedzone miasto. Jest szybka **O(n^2)**, ale może nie gwarantować optymalności.
- **Algorytm losowy (Random)**, generuje losowe trasy i porównuje ich koszt z najlepszym dotychczasowym rozwiązaniem. Może znaleźć rozwiązanie optymalne, ale brak gwarancji skuteczności przy ograniczonym czasie wykonania.

Celem projektu jest porównanie efektywności czasowej i dokładności uzyskanych wyników dla każdej z metod oraz zbadanie wpływu typu danych na działanie algorytmów. Wyniki pozwolą na ocenę skuteczności metod dla różnych instancji TSP.

1.2. Sformułowanie zadania dla Projektu 2 (Metoda Branch and Bound)

Projekt 2 koncentruje się na zastosowaniu metody podziału i ograniczeń (Branch and Bound) do rozwiązania problemu TSP. Metoda ta, będąca algorytmem optymalizacyjnym, pozwala na zredukowanie liczby analizowanych ścieżek poprzez przycinanie gałęzi drzewa rozwiązań, które nie mogą prowadzić do optymalnego rozwiązania. W ramach projektu zaimplementowane zostaną trzy strategie przeszukiwania w kontekście metody Branch and Bound:

- **Przeszukiwanie po szerokości (BFS)**, w którym analizowane są wszystkie gałęzie drzewa równocześnie na tym samym poziomie, zanim przejdzie się do głębszych poziomów. Dzięki temu BFS gwarantuje, że wszystkie możliwe ścieżki są rozważane równocześnie, co zapewnia znalezienie optymalnego rozwiązania, ale kosztem dużego zużycia pamięci. Złożoność BFS w TSP wynosi **O(n!)**, ale jego wymagania pamięciowe rosną szybko wraz ze wzrostem liczby miast, co czyni go nieefektywnym dla dużych instancji.
- **Przeszukiwanie po głębokości (DFS)**, które wnika w drzewo rozwiązań, eksplorując ścieżki do końca przed analizowaniem sąsiednich gałęzi. DFS może być bardziej efektywny pamięciowo niż BFS, ponieważ śledzi tylko bieżącą ścieżkę, co oznacza, że zużycie pamięci jest znacznie niższe. Złożoność czasowa również wynosi **O(n!)**, ale DFS może czasem analizować nieopłacalne ścieżki przez długi czas, zanim znajdzie rozwiązanie optymalne
- **Przeszukiwanie przy minimalnym koszcie (Lowest Cost / Best First Search)**, które sortuje i eksploruje gałęzie drzewa według minimalnego kosztu bieżącego. Strategia ta pozwala na bardziej efektywne przycinanie gałęzi i zmniejszenie liczby analizowanych ścieżek. Dzięki temu jest bardziej wydajna niż klasyczne DFS lub BFS w kontekście znajdowania optymalnych rozwiązań. Mimo to, złożoność czasowa pozostaje **O(n!)**.

Badanie koncentruje się na analizie efektywności czasowej wszystkich algorytmów oraz pamięciowej. TSP w projekcie 1 analizuje tylko czas wykonania, ponieważ algorytmy heurystyczne i losowe nie wymagają dużych zasobów pamięciowych.

1.3. Dane testowe oraz plik konfiguracyjny

Dane wejściowe pochodzą ze strony Dr. Jarosława Mierzwy, ponieważ są to dane wiarygodne (http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/) oraz wygenerowane losowo (są to pliki zawierające symboli „as", co oznacza asymetryczny oraz plik tsp-101). Dane są zapisane w plikach tekstowych w następującym formacie:

- **Pierwsza linia**: Liczba miast (np. 14), co oznacza, że macierz odległości jest rozmiaru 14x14.
- **Kolejne linie**: Macierz odległości, gdzie każda komórka **A[i][j]A[i][j]A[i][j]** reprezentuje koszt przejścia z miasta **iii** do **jjj**. Wartości -1/0 na przekątnej oznaczają brak możliwości pozostania w tym samym mieście.
- **Optymalny wynik** i **ścieżka**: Są dostępne w niektórych instancjach do oceny błędów. W projekcie 2 te informacje nie są używane.

Dane obejmują instancje dla [4, 6, 8, 10, 12, 13, 14, 15, 17, 101] miast, co umożliwia zbadanie wydajności algorytmów dla małych, średnich, dużych instancji oraz osiąganie limitu czasowego, żeby sprawdzić czy algorytmy potrafią zbliżyć się do rozwiązania optymalnego, nawet gdy pełne przeszukiwanie jest niemożliwe. Dane dotyczą grafów symetrycznych i asymetrycznych, co pozwala ocenić wpływ struktury danych na efektywność algorytmów.

```
12
-1 29 82 46 68 52 72 42 51 55 29 74
29 -1 55 46 42 43 43 23 23 31 41 51
82 55 -1 68 46 55 23 43 41 29 79 21
46 46 68 -1 82 15 72 31 62 42 21 51
68 42 46 82 -1 74 23 52 21 46 82 58
52 43 55 15 74 -1 61 23 55 31 33 37
72 43 23 72 23 61 -1 42 23 31 77 37
42 23 43 31 52 23 42 -1 33 15 37 33
51 23 41 62 21 55 23 33 -1 29 62 46
55 31 29 42 46 31 31 15 29 -1 51 21
29 41 79 21 82 33 77 37 62 51 -1 65
74 51 21 51 58 37 37 33 46 21 65 -1
264
0 -> 1 -> 8 -> 4 -> 6 -> 2 -> 11 -> 9 -> 7 -> 5 -> 3 -> 10 -> 0
```

Rysunek 3: zawartość pliku tsp_12.txt

Plik konfiguracyjny o rozszerzeniu .txt zawiera informacje takie jak:

- **data_file**: ścieżka do pliku z danymi.
- **output_file**: ścieżka do pliku z wynikami.
- **repeats**: liczba powtórzeń testów.
- **show_progress**: opcja wyświetlania postępu.
- **algorithm**: wybrany algorytm (BF, NN, random, BFS, DFS, LowestCost).
- **time_limit**: limit czasu w sekundach.
- **run_all_start_vertices**: czy sprawdzać wszystkie wierzchołki początkowe (true/false).
- **start_vertex**: wierzchołek startowy.

Pozwala to na konfigurację parametrów. Ostatnie dwa parametry są używane tylko dla algorytmu NN, ponieważ wierzchołek startowy ma wpływ na wynik zwrócony przez algorytm. Tych dwóch parametrów nie ma w pliku konfiguracyjnym 2 projektu.

1.4. Hipotezy badawcze

Celem badania jest potwierdzenie lub odrzucenie następujących hipotez dotyczących efektywności algorytmów:

1. **Brute-force (BF)**:

    - Hipoteza: Algorytm BF zawsze znajdzie optymalne rozwiązanie, ale jego czas działania rośnie wykładniczo wraz ze wzrostem liczby miast. Spodziewamy się, że BF będzie skuteczny tylko dla bardzo małych instancji (do 10 miast).

2. **Nearest-Neighbor (NN)**:

    - Hipoteza: NN jest efektywny czasowo dla dużych instancji, ale jego skuteczność w znajdowaniu optymalnych ścieżek maleje wraz z rosnącą liczbą miast. Wynik zależy od wybranego wierzchołka początkowego, co może wpływać na jakość rozwiązania.

3. **Random**:

    - Hipoteza: Algorytm losowy może znaleźć rozwiązanie optymalne, ale z racji losowości nie gwarantuje go przy ograniczonym czasie. Chcemy potwierdzić, że algorytm Random jest w stanie znaleźć rozwiązania z akceptowalnym błędem, jeśli liczba powtórzeń jest odpowiednio duża, mimo że optymalność rozwiązania nie jest gwarantowana.

4. **Branch and Bound (B&B) - Lowest Cost (LC)**:

    - Hipoteza: Strategia LC z dolnym ograniczeniem jest efektywna czasowo w porównaniu do pełnego przeszukiwania, ponieważ odcina nieobiecujące gałęzie. Oczekujemy, że dodatkowe obliczenia związane z wyznaczaniem dolnych ograniczeń zmniejszą liczbę przetworzonych węzłów, co skróci czas całkowity.

5. **Branch and Bound (B&B) - DFS**:

    - Hipoteza: DFS jest bardziej efektywny pamięciowo niż BFS, ale może przeszukiwać mniej optymalne gałęzie przez dłuższy czas. Spodziewamy się, że będzie dobrze działał dla średnich instancji, ale jego efektywność spadnie przy większej liczbie miast.

6. **Branch and Bound (B&B) - BFS**:

    - Hipoteza: BFS znajdzie optymalne rozwiązanie, ale jego wysoka złożoność pamięciowa sprawi, że będzie mniej efektywny dla dużych instancji. Oczekujemy, że BFS sprawdzi się przy małych instancjach, ale z powodu dużego zapotrzebowania na pamięć, jego wydajność spadnie wraz ze wzrostem liczby miast.

Metody uznają się za efektywne, jeśli potrafią znaleźć rozwiązanie optymalne lub zbliżone do optymalnego w akceptowalnym czasie i przy rozsądnym zużyciu pamięci. Efektywność będzie oceniana na podstawie zdolności algorytmu do skalowania się dla większych instancji problemu TSP, a także porównania wyników z innymi metodami pod kątem dokładności oraz oszczędności zasobów. Algorytm uznaje się za efektywny, jeśli jego działanie zapewnia kompromis między czasem obliczeń a jakością wyników.

2. Lista instancji i środowisko badawcze

Lista instancji(nazwa pliku, wartość optymalna oraz ścieżka):

tsp_4_1.txt 66 [0 2 3 1 0]

tsp_6_1.txt 132 [0 1 2 3 4 5 0]

tsp_6_2.txt 80 [0 5 1 2 3 4 0]

tsp_10.txt 212 [0 3 4 2 8 7 6 9 1 5 0]

tsp_12.txt 264 [0 1 8 4 6 2 11 9 7 5 3 10 0]

tsp_13.txt 269 [0 10 3 5 7 9 11 2 6 4 8 1 12 0]

tsp_14.txt 282 [0 10 3 5 7 9 13 11 2 6 4 8 1 12 0]

tsp_101.txt 2000 [-]

tsp6_as.txt 105 [0 4 3 1 5 2 0]

tsp8_as.txt 155 [0 4 7 3 2 6 1 5 0]

tsp10_as.txt 142 [0 7 4 2 9 6 8 3 5 1 0]

tsp12_as.txt 203 [0 2 11 7 10 4 8 1 5 6 3 9 0]

tsp13_as.txt 132 [0 4 7 5 11 8 12 2 10 1 3 6 9 0]

tsp14_as.txt 158 [0 1 9 10 4 2 8 11 12 13 3 5 7 6 0]

Badania zostały przeprowadzone na laptopie HP Pavilion 15:

- Procesor 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz, 2419 Mhz, 4 Core(s), 8 Logical Processor(s)

- Pamięć RAM DDR4 16.0 GB

- System Windows 11 Home 23H2 64-bit

Jest tym samem urządzeniem, na którym były prezentowane programy. Laptop pracował przy podpiętym zasilaniu, a podczas robienia pomiarów wszystkie programy poza środowiskiem pracowniczym były zamknięte.

3. Algorytmy

3.1. Bruteforce

Wszystkie możliwe ścieżki (permutacje miast) są generowane z wyłączeniem wierzchołka początkowego, który jest zawsze pierwszym miastem. Do wygenerowania permutacji używana jest standardowa funkcja **next_permutation()** z **<algorithm>** - biblioteki standardowej C++.

Funkcja **next_permutation()** wykorzystuje algorytm Narayana Pandita, generując kolejne permutacje w kolejności leksykograficznej. Algorytm ten przeszukuje kontener od końca, szukając pierwszej pary sąsiednich elementów, gdzie poprzedni element jest mniejszy od kolejnego. Jeśli taka para nie zostanie odnaleziona, oznacza to, że obecna permutacja jest największa, więc algorytm odwraca kolejność kontenera i zwraca false, kończąc generowanie.

Każda wygenerowana ścieżka jest oceniana za pomocą funkcji, która oblicza koszt trasy (czyli sumę wszystkich krawędzi). Koszt ścieżki jest porównywany z dotychczas najlepszym wynikiem, aby zaktualizować najlepszą znalezioną trasę.

W implementacji algorytm kończy działanie, gdy czas działania osiągnie zadany w pliku konfiguracyjnym limit albo po wykonaniu przeglądu zupełnego.

3.2. Nearest Neighbor

Implementacja umożliwia przeprowadzenie obliczeń dla każdego miasta jako wierzchołka startowego. Może to być jedno wybrane miasto lub wszystkie miasta w instancji, w zależności od konfiguracji.

Algorytm iteracyjnie odwiedza najbliższe nieodwiedzone miasto. Jeśli dla danego miasta istnieje kilka najbliższych sąsiadów, metoda korzysta z rekurencyjnej funkcji eksploracji, aby przeanalizować wszystkie możliwe ścieżki prowadzące przez miasta o takim samym dystansie.

Metoda najbliższego sąsiada kończy działanie po osiągnięciu limitu czasu lub po przeszukaniu wszystkich wierzchołków.

3.3. Random

Algorytm losowo przetasowuje wszystkie miasta, zaczynając od wierzchołka początkowego. W implementacji zastosowano funkcję **shuffle()** z nagłówka **<algorithm>**, która jest implementacją algorytmu Fishera-Yatesa, zapewniającą równomierne (jednakowo prawdopodobne) generowanie permutacji. Algorytm ten iteruje od ostatniego elementu danego kontenera i zamienia i-ty element z innym pseudolosowo wybranym elementem.

Po wylosowaniu ścieżki dodawany jest powrót do wierzchołka początkowego, aby utworzyć pełną trasę cykliczną (cykl Hamiltona). Następnie obliczana jest całkowita długość tej trasy, która jest porównywana z dotychczas najlepszym wynikiem.

Algorytm może działać do momentu znalezienia ścieżki optymalnej lub przekroczenia zadanego limitu czasu.

3.4. DFS

Algorytm DFS eksploruje każdą ścieżkę od miasta początkowego aż do momentu, gdy odwiedzi wszystkie miasta, a następnie wraca do miasta początkowego. Realizowane jest to przy użyciu **stosu (stack)**, co pozwala na pełne rozwijanie jednej ścieżki przed przejściem do następnej. DFS zaczyna od wierzchołka początkowego i stara się dojść jak najdalej, odwiedzając kolejne miasta. Dopiero gdy zakończy się jedna ścieżka, algorytm wraca do poprzedniego wierzchołka, próbując znaleźć inną drogę.

DFS w tej implementacji stosuje przycinanie gałęzi, czyli nie rozwija ścieżek, które już na wczesnym etapie są nieopłacalne. Dla każdej potencjalnej ścieżki przed dodaniem nowego miasta obliczany jest bieżący koszt. Jeżeli ten koszt jest większy lub równy obecnie najlepszej znalezionej trasie (minCost), to dalsze rozwijanie tej gałęzi jest przerywane.

3.5. BFS

Algorytm BFS rozwija wszystkie możliwe ścieżki na danym poziomie drzewa przeszukiwania, zanim przejdzie do kolejnego poziomu. Jest to realizowane za pomocą **kolejki (queue)**, która umożliwia przechodzenie przez wierzchołki poziom po poziomie. BFS zaczyna od miasta początkowego, a następnie rozwija wszystkie możliwe miasta, które można odwiedzić. Każda ścieżka jest rozpatrywana w kontekście pełnego cyklu, co oznacza, że po osiągnięciu ostatniego miasta, wraca do miasta startowego, obliczając koszt całkowity.

3.6. Lowest Cost

Algorytm Lowest Cost rozwija ścieżki w oparciu o priorytet kosztowy. Wykorzystuje do tego **kolejkę priorytetową (priority queue)**, w której zawsze na początku znajduje się ścieżka o najniższym koszcie. Dzięki temu algorytm rozwija ścieżki, które mają największe szanse prowadzić do optymalnego rozwiązania.

4. Procedura badawcza

Procedura miała na celu potwierdzenie lub odrzucenie postawionych hipotez dotyczących efektywności czasowej oraz zużycia pamięci przez algorytmy.

Badania przeprowadzono na instancjach o rozmiarach **4, 6, 8, 10, 12, 13, 14, 101**. Instancje obejmowały **macierze symetryczne** i **asymetryczne**, aby ocenić wpływ rodzaju macierzy na wyniki działania algorytmów.

Algorytm NN był uruchomiany 50 razy dla wszystkich wierzchołków. Pozostałe algorytmy były uruchomione 15 dla małych i 5 razy dla dużych instancji.
Dla niektórych metod wielokrotne powtórzenia mają kluczowe znaczenie, np. dla Random wynik może zależeć od losowości.

Do pliku wyjściowego **,,results.xls”** zapisywany były: Nazwa instancji, Wynik i ścieżka optymalna z pliku z danymi (nie ma w 2 projekcie), znaleziony koszt, błąd względny/bezwzględny (nie ma w 2 projekcie), czas wykonania (dla każdego powtórzenia), czas średni. Plik wyjściowy zapisywany był w formacie xls. Poniżej przedstawiono przykładowy fragment zawartości pliku wyjściowego:

Nazwa instancji	Wynik optymalny	Sciezka optymalna
C:\Users\jaded_tin338\Desktop\studia\PEA\project_1\tsp\tsp_4_1.txt	66	0 -> 2 -> 3 -> 1 -> 0

Minimalny koszt	Blad bezwzgledny	Blad wzgledny (%)	Czasy wykonania (s)	Sredni czas (s)
66	0	0.0000	0.0000	0.0000

Tabela 1: przykładowa zawartość pliku results.xls

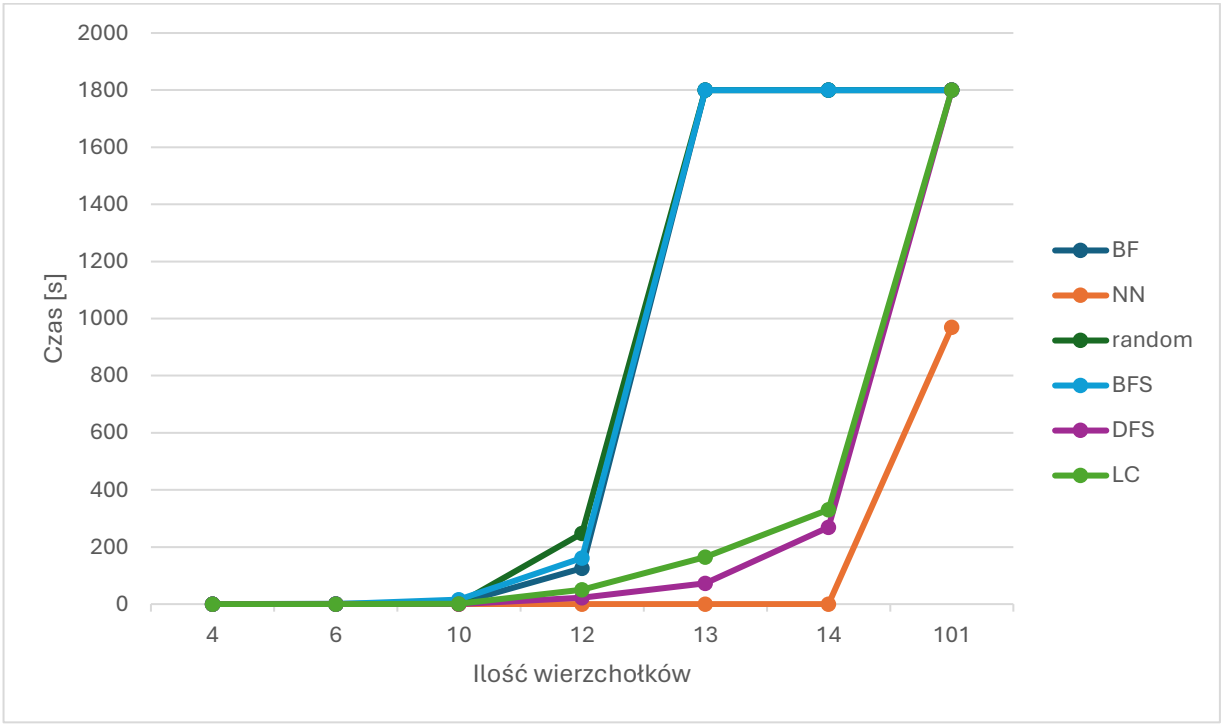
Sposób pomiaru czasu w badaniach opiera się na wykorzystaniu funkcji z biblioteki **<chrono>** w języku C++. Przed wykonaniem algorytmu rozpoczynamy pomiar za pomocą funkcji **high_resolution_clock::now().** Wynik tej różnicy jest następnie przeliczany na sekundy.

Pamięć była monitorowana z wykorzystaniem **Task Managera** systemu Windows.

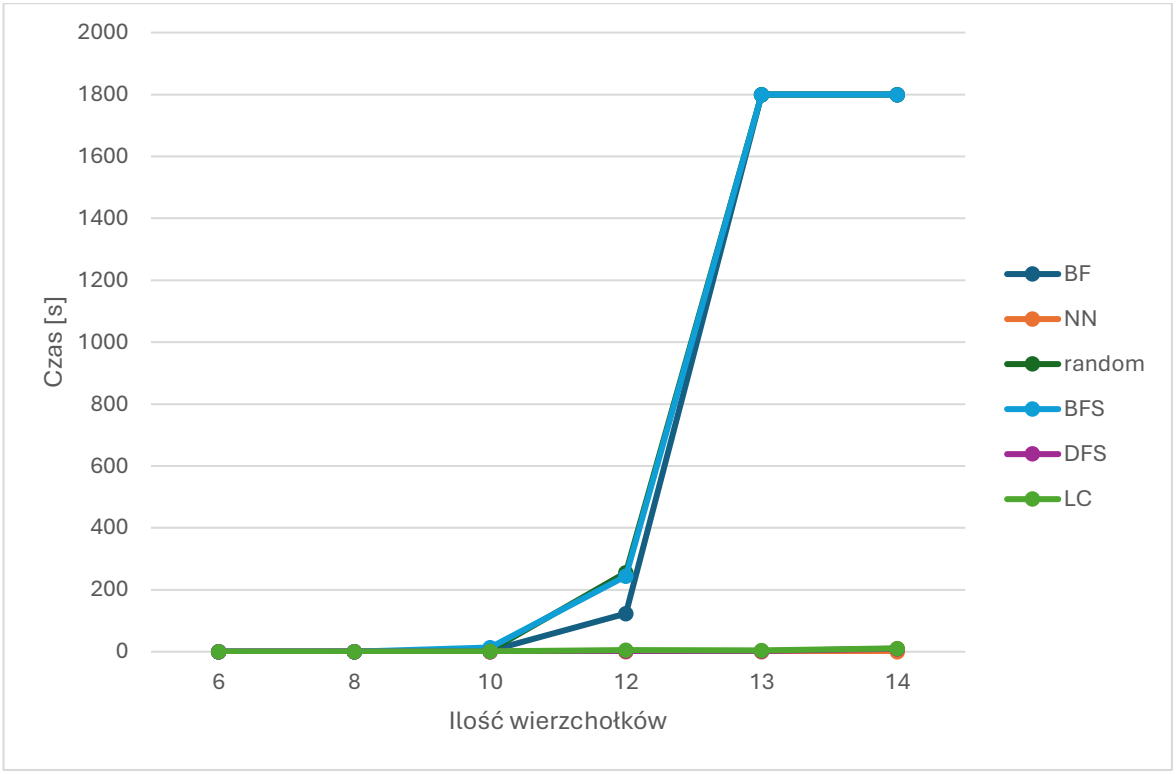
5. Wyniki

Nazwa pliku	BF	NN	Random	BFS	DFS	LowestCost
tsp_4_1	0	0,00035	0	0	0	0
tsp_6_1	1	0,00197	0,002	0,005	0,003	0,008
tsp_10	2,932	0,045	1,761	15,683	0,345	0,6
tsp_12	125,215	0,016	247,457	161,732	22,695	50,48
tsp_13	1800	0,02	1800	1800	73,057	165
tsp_14	1800	0,022	1800	1800	269,23	330,43
tsp_101	1800	969,906	1800	1800	1800	1800
tsp6_as	0,001	0,007	0,004	0,004	0,001	0,009
tsp8_as	0,048	0,005	0,068	0,187	0,019	0,427
tsp10_as	3,105	0,007	3,89	13,837	0,149	0,8
tsp12_as	122,87	0,011	255,63	244,486	1,83	5,2
tsp13_as	1800	0,012	1800	1800	2,142	4,06
tsp14_as	1800	0,012	1800	1800	9,87	10,90

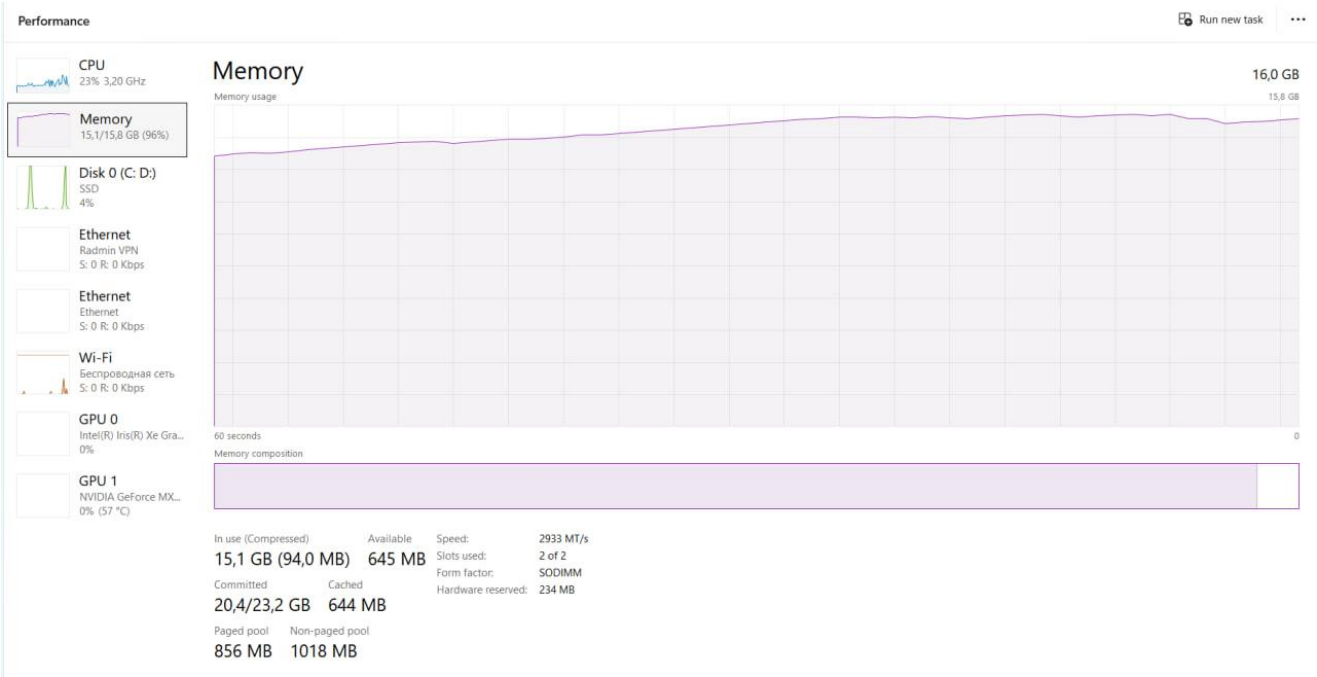
Tabela 2: wyniki pomiarów czasu dla wszystkich algorytmów w sekundach



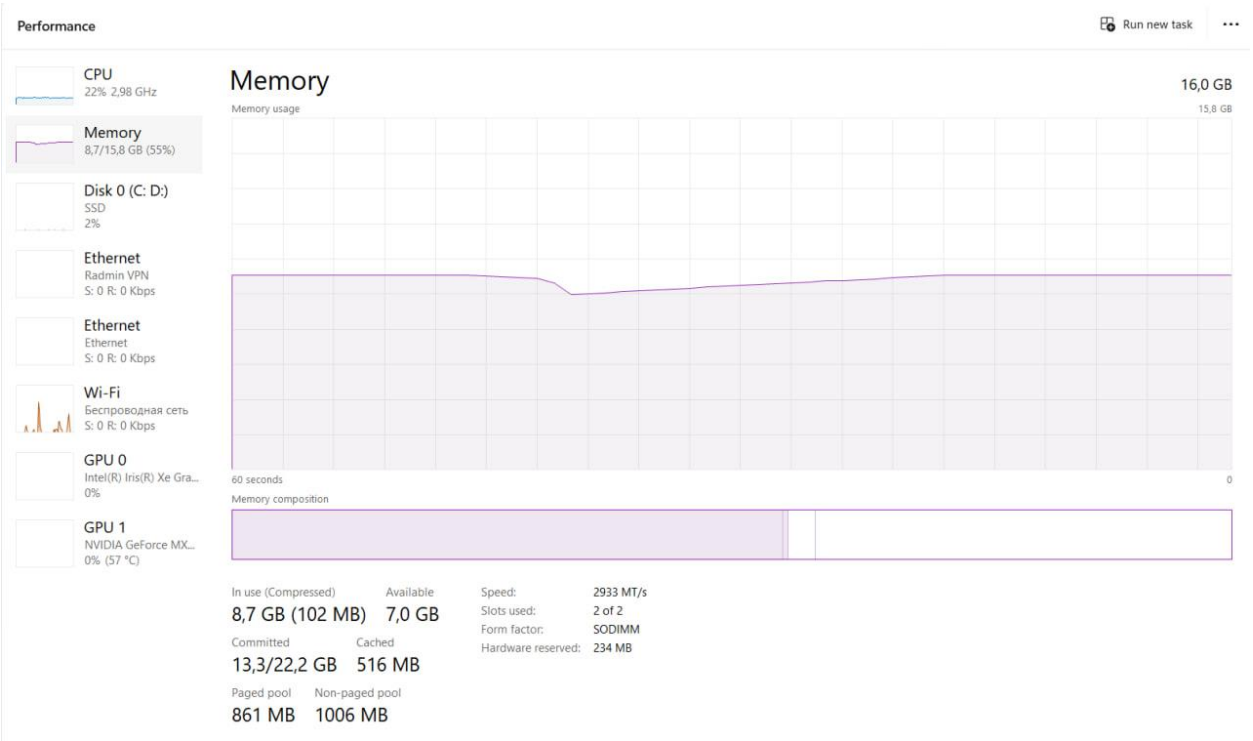
Rysunek 4: wykres zależności średniego czasu wykonania algorytmów od rozmiaru dla instancji symetrycznych



Rysunek 5: wykres zależności średniego czasu wykonania algorytmów od rozmiaru dla instancji asymetrycznych



Rysunek 6: pamięć wykorzystywana przy wykonywaniu algorytmu metodą BFS



Rysunek 7: pamięć wykorzystywana przy wykonywaniu algorytmu metodą DFS

6. Analiza wyników i wnioski

6.1. Analiza wyników

1. **Algorytm Brute-Force (BF)**:

   Czas wykonania BF gwałtownie rośnie wraz ze wzrostem liczby wierzchołków. W przypadku instancji większych niż 10 wierzchołków (np. dla tsp_13 i tsp_14) algorytm osiąga limit czasowy wynoszący 1800 sekund, co wskazuje na jego nieefektywność dla większych problemów. Jest to zgodne z oczekiwaniami, ponieważ algorytm BF ma złożoność wykładniczą.

2. **Algorytm Najbliższego Sąsiada (NN)**:

   NN wykazał dużą efektywność czasową zarówno dla małych, jak i średnich instancji. Jest to jedyny algorytm, który zdołał rozwiązać problem tsp_101 w rozsądnym czasie (około 970 sekund).

3. **Algorytm Losowy (Random)**:

   Wyniki algorytmu Random były bardzo zróżnicowane. W niektórych przypadkach (np. tsp_10_as) algorytm znalazł rozwiązanie stosunkowo szybko (3,89 s), ale dla innych instancji (np. tsp_13_as) osiągnął limit czasowy.

   Losowość tego algorytmu prowadzi do wysokiej zmienności wyników, a brak gwarancji optymalności sprawia, że jest mniej niezawodny w porównaniu do innych metod.

4. **Przeszukiwanie BFS**:

   Algorytm BFS charakteryzował się rosnącym czasem wykonania wraz ze wzrostem liczby wierzchołków. Dla instancji tsp_13 i większych osiągnął limit czasowy.

   BFS zapewnia znalezienie optymalnego rozwiązania, jednak wymaga dużych zasobów pamięci i jest mało efektywny dla większych instancji. Zużycie pamięci dla BFS osiągnęło maksymalną wartość 15,1 GB, co jest bliskie pełnemu zużyciu dostępnej pamięci.

5. **Przeszukiwanie DFS**:

   DFS był bardziej efektywny czasowo niż BFS, szczególnie dla instancji średniej wielkości, np. dla tsp_12 i tsp_13. DFS w przypadku instancji tsp_14 uzyskał wynik w około 269 sekund, co jest lepszym rezultatem niż BFS.

   Algorytm DFS, dzięki swojej naturze eksplorowania ścieżek w głąb, zużywał mniej pamięci – na poziomie 8,7 GB – co jest bardziej optymalne w porównaniu z BFS.

6. **Algorytm Lowest Cost (LC)**:

   Algorytm LC działał efektywniej niż BFS i DFS w wielu przypadkach, osiągając znacznie lepsze wyniki czasowe dla średnich instancji (tsp_12, tsp_13). Na przykład dla tsp_13 czas wykonania wyniósł 165 sekund, podczas gdy BFS osiągnął limit 1800 sekund.

   LC był w stanie lepiej wykorzystać dolne ograniczenia i skutecznie przycinać nieopłacalne gałęzie, co skróciło czas całkowity przeszukiwania.

6.2. Wnioski

1. **Efektywność Czasowa**:

   Algorytm Nearest Neighbor (NN) okazał się najbardziej efektywny czasowo, nawet dla dużych instancji (tsp_101), choć nie zawsze gwarantuje rozwiązanie optymalne.

   Algorytmy BFS i DFS wykazały ograniczoną efektywność przy dużych instancjach z powodu wysokiej złożoności obliczeniowej oraz wymagań pamięciowych.

2. **Skalowalność Algorytmów**:

   Algorytmy bruteforce (BF) oraz BFS nie są skalowalne dla instancji większych niż 12-14 wierzchołków z powodu ich złożoności O(n!). Przeszukiwanie przestrzeni rozwiązań staje się zbyt czasochłonne.

   LC oraz DFS radzą sobie lepiej w porównaniu do BFS i są bardziej efektywne przy średnich instancjach dzięki zastosowaniu przycinania nieopłacalnych gałęzi.

3. **Rodzaj Instancji**:

   Wyniki dla instancji asymetrycznych i symetrycznych są zbliżone pod względem trendów czasowych, ale dla niektórych algorytmów asymetryczność ma znaczenie, co może wpływać na wybór algorytmu w zależności od rodzaju danych.

Ogólnie rzecz biorąc, analiza wykazała, że efektywność algorytmów zależy od rozmiaru instancji oraz struktury danych, a także od zdolności algorytmów do ograniczania przestrzeni rozwiązań. Algorytmy heurystyczne oferują szybkie wyniki, ale bez gwarancji optymalności, podczas gdy algorytmy dokładne są bardziej czasochłonne i wymagające pamięciowo, szczególnie dla większych problemów.