# Projektowanie efektywnych algorytmów

## Projekt 1

### Asymetryczny problem komiwojażera

Mateusz Tesarewicz 272909

18.11.2024 Politechnika Wrocławska

# Spis treści

# 1 Wstęp

Asymetryczny problem komiwojażera (ATSP, z ang. *Asymmetric Traveling Salesman Problem*) stanowi jedno z kluczowych wyzwań optymalizacyjnych w teorii grafów oraz badań operacyjnych. Zadanie to polega na wyznaczeniu najkrótszej możliwej trasy, która pozwala komiwojażerowi odwiedzić wszystkie zadane miasta dokładnie raz i powrócić do punktu początkowego, przy czym odległości między miastami mogą się różnić w zależności od kierunku przejścia (asymetria). ATSP znajduje szerokie zastosowanie w logistyce, planowaniu tras oraz optymalizacji produkcji, a także w systemach komunikacji miejskiej i transporcie.

Ze względu na swoją złożoność obliczeniową (problem należy do klasy NP-trudnych), rozwiązywanie ATSP wymaga zastosowania specjalistycznych algorytmów, takich jak algorytmy przeglądu zupełnego, podziału i ograniczeń lub programowania dynamicznego. Celem niniejszego badania jest ocena efektywności wybranych algorytmów stosowanych do rozwiązywania ATSP, poprzez analizę ich wyników w kontekście jakości uzyskanych rozwiązań oraz czasu obliczeń. Wyniki badania mają na celu dostarczenie informacji na temat praktycznej przydatności poszczególnych metod oraz wskazanie, które algorytmy mogą być szczególnie skuteczne w różnych przypadkach ATSP.

# 2 Badane algorytmy

## 2.1 Algorytm Brute Force

Algorytm Brute Force dla asymetrycznego problemu komiwojażera (ATSP) polega na wygenerowaniu wszystkich możliwych permutacji tras odwiedzających każde miasto dokładnie raz, a następnie powracających do punktu początkowego. Dla każdej permutacji obliczana jest całkowita długość trasy, a algorytm wybiera tę, która jest najkrótsza. Brute Force gwarantuje

znalezienie rozwiązania optymalnego, jednak jego złożoność obliczeniowa wynosi **O(n!),** co oznacza, że czas działania rośnie wykładniczo wraz z liczbą miast n. Zakładając, że będziemy startować zawsze z pierwszego miasta możemy zmniejszyć złożoność obliczeniową do **O[(n-1)!].** W rezultacie algorytm ten jest praktyczny jedynie dla małych instancji problemu – dla większej liczby miast czas obliczeń staje się nieakceptowalnie długi.

## 2.2 Algorytm Little'a

Algorytm Little'a jest metodą dokładnego rozwiązywania asymetrycznego problemu komiwojażera (ATSP) opartą na technice przeszukiwania z ograniczeniami (*branch and bound*). Algorytm rozpoczyna od skonstruowania macierzy kosztów podróży między miastami, po czym wykonuje redukcję wierszy i kolumn, aby zmniejszyć wartości kosztów i uzyskać oszacowanie dolne dla minimalnego kosztu trasy. Następnie dzieli problem na podproblemy (podział drzewa przeszukiwań) i dla każdego z nich oblicza nowe oszacowania dolne, eliminując te gałęzie, które nie mogą prowadzić do rozwiązania optymalnego (odcięcie).

Algorytm Little'a, jako metoda *branch and bound* dla asymetrycznego problemu komiwojażera, ma w najgorszym przypadku wykładniczą złożoność obliczeniową, zbliżoną do **O(n!).** Wynika to z konieczności przeglądania potencjalnych rozwiązań w przypadku braku skutecznego przycinania gałęzi, czyli w sytuacjach, gdy algorytm musi przeanalizować większość lub wszystkie możliwe permutacje miast.

Jednakże w praktyce, dzięki technice *branch and bound*, algorytm Little'a znacznie ogranicza liczbę badanych rozwiązań, co wpływa na poprawę jego efektywności w wielu przypadkach. Średnia złożoność obliczeniowa jest trudna do jednoznacznego oszacowania, ponieważ zależy od struktury problemu oraz stopnia asymetrii kosztów między miastami. Niemniej, w typowych sytuacjach można przyjąć, że średnia złożoność jest mniejsza niż

wykładnicza, często oscylująca w granicach $\mathbf{O(2^n)}$ lub nawet $\mathbf{O(n^2 \cdot 2^n)}$ dla średnich instancji problemu.

## 2.3 Programowanie dynamiczne

Dla algorytmu rozwiązującego problem komiwojażera metodą programowania dynamicznego, zwykle stosuje się algorytm oparty na podejściu Bellmana-Helda-Karpa.

Algorytm Bellmana-Helda-Karpa oparty na programowaniu dynamicznym rozwiązuje problem komiwojażera, przechowując optymalne wyniki częściowych tras, aby uniknąć wielokrotnego obliczania kosztów dla tych samych podproblemów. W algorytmie tym dla każdego zbioru odwiedzonych miast oraz bieżącego miasta jako końcowego, zapisuje się minimalny koszt dotarcia do tej konfiguracji. Proces ten powtarza się rekurencyjnie dla wszystkich możliwych podzbiorów miast, a na końcu zwraca najkrótszą możliwą trasę.

Algorytm ma złożoność obliczeniową równą $O(2^n \cdot n^2)$, ponieważ musi przetworzyć wszystkie podzbiory miast ( $2^n \cdot n$ kombinacji), a dla każdego z nich przeanalizować trasy do pozostałych miast (n możliwości). Algorytm ten, choć wykładniczy, jest bardziej wydajny niż pełne przeglądanie wszystkich permutacji miast, dzięki przechowywaniu i wykorzystywaniu wyników podproblemów.

# 3 Opisy działania algorytmów

## 3.1 Założenia

Poniżej zostały przedstawione „krok po kroku” opisy działania algorytmów dla przykładowej instancji danego problemu o wartości N = 4 i danej macierzy:

Uwaga! Każdy algorytm zaczyna się od wierzchołka 1 jako startowego.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | -1 | 10 | 15 | 20 |
| 2 | 5 | -1 | 9 | 20 |
| 3 | 6 | 13 | -1 | 12 |
| 4 | 8 | 8 | 9 | -1 |

## 3.2 Algorytm Little’a

### 3.2.1 Przebieg algorytmu „krok po kroku”

1. Pierwsza redukcja macierzy oraz zamiana wszystkich -1 na INF

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | 0 | 4 | 4 |
| 2 | 0 | INF | 3 | 9 |
| 3 | 0 | 7 | INF | 0 |
| 4 | 0 | 0 | 0 | INF |

Koszt redukcji: 36

Kolejka: 1

2. Pobieranie wierzchołka 1 z kolejki. Obecna droga 1. Koszt 36. Poziom 0.

2.1 Tworzenie gałęzi do wierzchołka 2

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | 0 | 6 |
| 3 | 0 | INF | INF | 0 |
| 4 | 0 | INF | 0 | INF |

Koszt redukcji: 3. Całkowity koszt drogi: 36+3 = 39

Kolejka: (1->2)

2.2 Tworzenie gałęzi do wierzchołka 3.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | 0 | INF | INF | 9 |
| 3 | INF | 7 | INF | 0 |
| 4 | 0 | 0 | INF | INF |

Koszt redukcji: 4. Całkowity koszt drogi: 36+4 = 40

Kolejka: (1->2), (1->3)

2.3 Tworzenie gałęzi do wierzchołka 4.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | 0 | INF | 3 | INF |
| 3 | 0 | 7 | INF | INF |
| 4 | INF | 0 | 0 | INF |

Koszt redukcji: 4. Całkowity koszt drogi: 36+4 = 40

Kolejka: Kolejka: (1->2), (1->3), (1->4)

3. Pobieranie wierzchołka 2 z kolejki. Obecna droga 1->2. Koszt: 39.

Poziom 1

3.1 Tworzenie gałęzi do wierzchołka 3.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | INF | INF |
| 3 | INF | INF | INF | 0 |
| 4 | 0 | INF | INF | INF |

Koszt redukcji: 0. Całkowity koszt drogi: 39+0 = 39.

Kolejka: (1->2->3), (1->3), (1->4)

3.2 Tworzenie gałęzi do wierzchołka 4.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | INF | INF |
| 3 | 0 | INF | INF | INF |
| 4 | INF | INF | 0 | INF |

Koszt redukcji: 6. Całkowity koszt drogi: 39+6 = 45.

Kolejka: : (1->2->3), (1->3), (1->4), (1->2->4)

4. Pobieranie wierzchołka 3 z kolejki. Obecna droga 1->2->3. Koszt 39.

Poziom 2.

4.1 Tworzenie gałęzi do wierzchołka 4.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | INF | INF |
| 3 | INF | INF | INF | INF |
| 4 | INF | INF | INF | INF |

Koszt redukcji: 0. Całkowity koszt drogi 39+0=39.

Kolejka: (1->2->3->4), (1->3), (1->4), (1->2->4)

5. Pobieranie wierzchołka 4 z kolejki. Obecna droga. 1->2->3->4. Koszt 39. Poziom 3.

5.1 Znaleziono pełną ścieżkę i ścieżka jest najmniejsza do tej pory. Zapamiętujemy minimalny całkowity koszt = 39.

Kolejka: (1->3), (1->4), (1->2->4)

6. Pobieranie wierzchołka 3 z kolejki. Obecna droga 1->3. Koszt 40.

Poziom 1

6.1 Tworzenie gałęzi do wierzchołka 2.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | INF | 0 |
| 3 | INF | INF | INF | INF |
| 4 | 0 | INF | INF | INF |

Koszt redukcji: 16. Całkowity koszt drogi: 40+16=56.

Kolejka: (1->4), (1->2->4). Nie dodano wierzchołka 2 do kolejki.

6.2 Tworzenie gałęzi do wierzchołka 4.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | 0 | INF | INF | INF |
| 3 | INF | INF | INF | INF |
| 4 | INF | 0 | INF | INF |

Koszt redukcji: 0. Całkowity koszt drogi: 40+0=40.

Kolejka: (1->4), (1->2->4). Nie dodano wierzchołka 4 do kolejki.

7. Pobieranie wierzchołka 4 z kolejki. Obecna droga 1->4. Koszt 40.

Poziom 1

7.1 Tworzenie gałęzi do wierzchołka 2.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | 0 | INF |
| 3 | 0 | INF | INF | INF |
| 4 | INF | INF | INF | INF |

Koszt redukcji: 3. Całkowity koszt drogi: 40+3=43.

Kolejka: Kolejka: (1->2->4). Nie dodano wierzchołka 2 do kolejki.

7.2 Tworzenie gałęzi do wierzchołka 3.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | 0 | INF | INF | INF |
| 3 | INF | 0 | INF | INF |
| 4 | INF | INF | INF | INF |

Koszt redukcji: 7. Całkowity koszt drogi: 40+7 = 47.

Kolejka: Kolejka: (1->2->4). Nie dodano wierzchołka 3 do kolejki.

8. Pobieranie wierzchołka 4 z kolejki. Całkowita droga 1->2->4. Koszt 45. Poziom 2.

8.1 Tworzenie gałęzi do wierzchołka 3.

Redukcja macierzy i usuwanie wiersza oraz kolumny.

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | INF | INF | INF | INF |
| 3 | INF | INF | INF | INF |
| 4 | INF | INF | INF | INF |

Koszt redukcji: 0. Całkowity koszt drogi: 45+0=45.

Kolejka: (pusta). Nie dodano wierzchołka 3 do kolejki.

9. Przekazanie wyniku.

Najkrótszą ścieżką jest ścieżka 1 -> 2 -> 3 -> 4 -> 1 o całkowitym koszcie 39.

### 3.2.2 Przykład redukcji macierzy

Redukowanie macierzy polega na znalezieniu najmniejszej wartości w każdym wierszu i odjęciu jej od wszystkich elementów w tym wierszu. Tak samo robi z kolumnami. Najmniejsze wartości sumujemy i otrzymujemy koszt redukcji. Poniżej przykład:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | 10 | 15 | 20 |
| 2 | 5 | INF | 9 | 20 |
| 3 | 6 | 13 | INF | 12 |
| 4 | 8 | 8 | 9 | INF |

Koszt redukcji wierszy: 10+5+6+8=29. Macierz po redukcji wierszy:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | 0 | 5 | 10 |
| 2 | 0 | INF | 4 | 15 |
| 3 | 0 | 7 | INF | 6 |
| 4 | 0 | 0 | 1 | INF |

Koszt redukcji kolumn: 0+0+1+6 = 7. Macierz po redukcji kolumn:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | 0 | 4 | 4 |
| 2 | 0 | INF | 3 | 9 |
| 3 | 0 | 7 | INF | 0 |
| 4 | 0 | 0 | 0 | INF |

Całkowity koszt redukcji: 29+7 = 36.

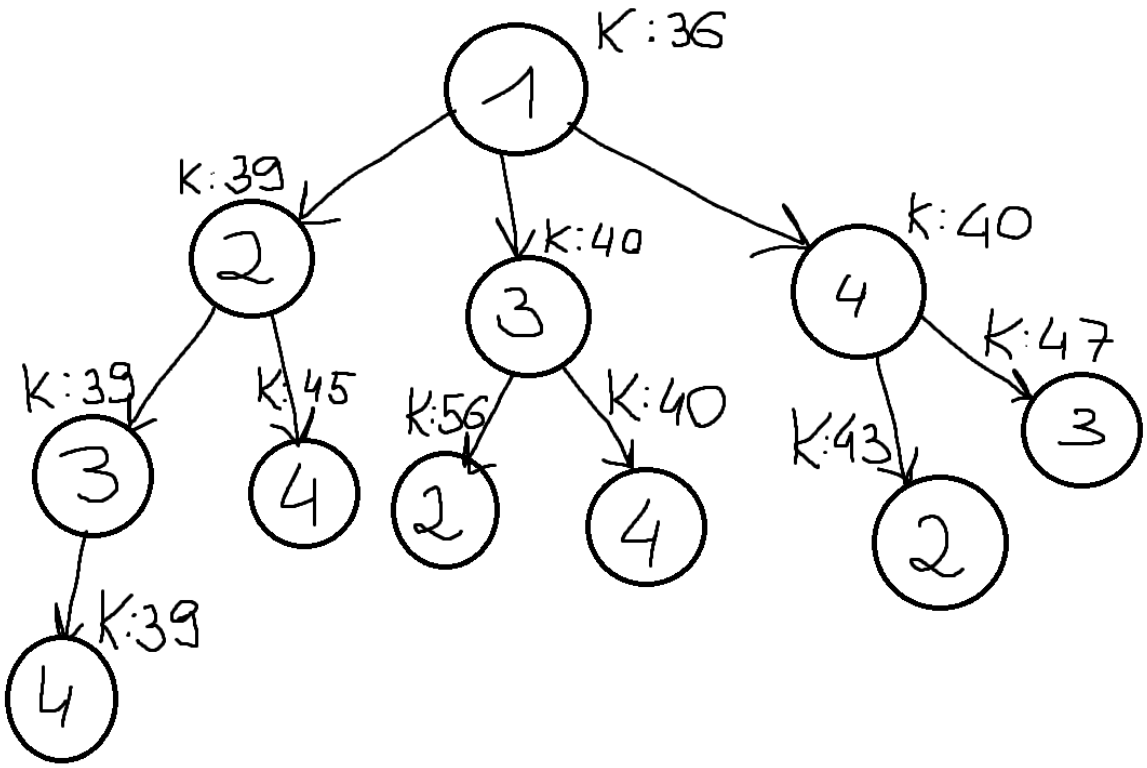
### 3.2.3 Przykład usuwania wiersza i kolumny

Zakładamy, że jesteśmy w wierzchołku 1 i tworzymy gałąź do wierzchołka 2. Musimy wówczas zredukować macierz oraz usunąć wiersz 1 i kolumnę 2 w taki sposób:

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | INF | INF | INF | INF |
| 2 | 0 | INF | 3 | 9 |
| 3 | 0 | INF | INF | 0 |
| 4 | 0 | INF | 0 | INF |

Usuwanie polega tak naprawdę na uzupełnieniu całego wiersza i kolumny wartościami INF.

### 3.2.4 Utworzony graf po przebiegu algorytmu



## 3.3 Programowanie dynamiczne

### 3.3.1 Używane symbole

$$d_{i,j} - odległość\ między\ wierzchołkami\ i\ oraz\ j$$

$D(S,p)$ - optymalna długość ścieżki wychodzącej z punktu 1 i przechodzącej przez wszystkie punkty zbioru *S* tak, aby zakończyć się w punkcie *p* (*p* musi należeć do *S*). Przykład: zapis *D({2, 5, 6}, 5)* to optymalna długość ścieżki wychodzącej z punktu 1, przechodzącej przez punkty 2 i 6, kończącej się w punkcie 5. Liczbę punktów w zbiorze *S* oznaczmy jako *s*.

Wartość D(S,p) wyznaczamy następująco:

Jeśli s=1, to $D(S,p) = d_{1,p}$

Jeśli s>1, to $D(S,p) = min_{x \in (S-\{p\})}(D(S-\{p\},x) + d_{x,p})$

Mówiąc obrazowo, musimy za każdym razem ustalać, który punkt powinien być przedostatni na trasie (który punkt ma poprzedzać punkt *p*).

Na końcu wyznacza się rozwiązanie całego problemu. W tym celu należy znaleźć poprzednika punktu 1.

### 3.3.2 Przebieg algorytmu „krok po kroku”

1. Wyznaczenie wartości D(S,p) dla jednoelementowych zbiorów S (s=1).

$$D(\{2\},2) = d_{1,2} = 10$$
$$D(\{3\},3) = d_{1,3} = 15$$
$$D(\{4\},4) = d_{1,4} = 20$$

2. Wyznaczenie wartości D(S,p) dla dwuelementowych zbiorów S (s=2). W nawiasie kwadratowym znajduje się numer przedostatniego węzła na ścieżce (węzła x dla optymalnego rozwiązania podproblemu).

$$D(\{2,3\},2) = min(\ D(\{3\},3) + d_{3,2}\ ) = min(15 + 13)$$
$$= min(28) = 28\ [3]$$
$$D(\{2,3\},3) = min(\ D(\{2\},2) + d_{2,3}\ ) = min(10 + 9) = min(19)$$
$$= 19\ [2]$$

$$D(\{2,4\},2) = min(\, D(\{4\},4) + d_{4,2}\,) = min(20 + 8) = min(28)$$
$$= 28\ [4]$$
$$D(\{2,4\},4) = min(\, D(\{2\},2) + d_{2,4}\,) = min(10 + 10)$$
$$= min(20) = 20\ [2]$$
$$D(\{3,4\},3) = min(\, D(\{4\},4) + d_{4,3}\,) = min(20 + 9) = min(29)$$
$$= 29\ [4]$$
$$D(\{3,4\},4) = min(\, D(\{3\},3) + d_{3,4}\,) = min(15 + 12)$$
$$= min(27) = 27\ [4]$$

3. Wyznaczamy wartości D(S,p) dla trójelementowych zbiorów S (s=3)

$$D(\{2,3,4\},2) = min\begin{cases} D(\{3,4\},3) + d_{3,2} = 29 + 13 = 42 \\ D(\{3,4\},4) + d_{4,2} = 27 + 8 = 35 \end{cases} = 35\ [4]$$

$$D(\{2,3,4\},3) = min\begin{cases} D(\{2,4\},2) + d_{2,3} = 28 + 9 = 37 \\ D(\{2,4\},4) + d_{4,3} = 27 + 9 = 36 \end{cases} = 36\ [4]$$

$$D(\{2,3,4\},4) = min\begin{cases} D(\{2,3\},2) + d_{2,4} = 28 + 10 = 38 \\ D(\{2,3\},3) + d_{3,4} = 19 + 12 = 31 \end{cases} = 31\ [3]$$

4. Trójelementowy zbiór S jest dla tego zadania największym z możliwych, gdyż nie licząc wierzchołka początkowego mamy trzy wierzchołki. Możemy więc już teraz wyznaczyć rozwiązanie całego zadania:

$$D(\{1,2,3,4\},1) = min\begin{cases} D(\{2,3,4\},2) + d_{2,1} = 35 + 5 = 40 \\ D(\{2,3,4\},3) + d_{3,1} = 36 + 6 = 42 = 39\ [4] \\ D(\{2,3,4\},4) + d_{4,1} = 31 + 8 = 39 \end{cases}$$

5. Przekazanie wyniku

   Możemy odtworzyć trasę wykorzystując indeksy w kwadratowych nawiasach i wychodzi nam najkrótsza ścieżka: 1- > 2 -> 3 -> 4 -> 1 o koszcie 39.

# 4 Plan przeprowadzania badania

Poprawnie zaplanowanie przebiegu badania jest bardzo kluczowe, aby uniknąć ewentualnych komplikacji lub co gorsza, niepoprawnych wyników przeprowadzonego badania. Badaniu poddamy wszystkie algorytmy dla wyznaczonych ilości miast. Odległości między miastami zostaną przedstawione w postaci macierzy kwadratowej. Odległości będą dodatnimi liczbami całkowitymi. Zbadamy również maksymalną ilość miast, którą algorytm jest w stanie obsłużyć.

## 4.1 Środowisko i specyfikacja sprzętu

Do przeprowadzenia badania został napisany obiektowy program w języku C++ kompilowany w środowisku CLion. Badania zostaną przeprowadzone na komputerze o podanej specyfikacji:

Procesor :AMD Ryzen 7 5700X 8-Core Processor 3.40 GHz

Pamięć RAM: 16 GB 3200 MHz

System operacyjny: Windows 10 Home

Typ systemu: 64-bitowy system operacyjny, procesor x64

## 4.2 Przedstawienie odległości między miastami

Odległości między miastami będą zapisane w kwadratowej macierzy, gdzie numer wiersza będzie oznaczał miasto początkowe, a numer kolumny miasto końcowe. Wartością w macierzy będzie odległość między miastem początkowym a końcowym. Drogi między miastami będą zazwyczaj asymetryczne. Po przekątnej macierzy wpisane zostaną wartości -1 lub 0, żeby wykluczyć drogi z miast do siebie nawzajem. W innych komórkach będą mogły się znajdować tylko liczby naturalne dodatnie z przedziału ustalanego w configu. Aby wczytać macierz, należy przygotować ją w pliku tekstowym oddzielając kolejne wartości w wierszach spacją. Każdy wiersz należy pisać w

nowej linii. W pierwszej linii pliku tekstowego powinna się znajdować jedna liczba, która określa rozmiar macierzy kwadratowej.

## 4.3 Rozmiar problemu – reprezentatywne wartości

Podczas przeprowadzania badania ustalone zostały reprezentatywne wartości ilości miast: **4, 6, 8, 10, 12, 13, 14**.

Dla tych wartości zostanie zbadany każdy algorytm pod względem wydajności. Wartości te pozwalają zbadać wydajności wszystkich trzech algorytmów w przyzwoitym czasie. Większe wartości sprawiały problem dla pierwszego algorytmu.

Dodatkowo zostanie przeprowadzony test, który pozwoli wyznaczyć maksymalny rozmiar problemu, który każdy z algorytmów może obsłużyć w przyzwoitym czasie.

## 4.4 Pomiar czasu

Czas zostanie zmierzony za pomocą biblioteki oraz funkcji z std::chrono::high_resolution_clock . Mierzony czas dotyczy jedynie wykonania pojedynczego wybranego algorytmu. Nie uwzględnia się dodatkowych operacji takich jaki generowanie macierzy, wyświetlanie wyników lub ich konwertowanie. Mierzenie czasu polega na odczytaniu czasu bezpośrednio przed wywołaniem funkcji algorytmu i od razu po ukończeniu. Wówczas czas wykonania algorytmu zostanie obliczony z różnicy między czasem końcowym, a czasem początkowym (Dokładna wizualizacja w kodzie). W specjalnym trybie testowania wykonujemy po 100 testów dla każdej reprezentatywnej, ponieważ pojedyncze testy mogą być objęte dużym błędem. Ze 100 testów obliczana jest średnia jako końcowy wynik uśredniony, który jest w wpisany w tabeli.

## 4.5 Przebieg eksperymentu

Zaimplementowany program pozwala badać czas i wydajność wspomnianych wyżej algorytmów dla pojedynczych macierzy, które mogą zostać wygenerowane lub wczytane z pliku tekstowego. Jednak nie nas to interesuje podczas przeprowadzania masowych badań. Do tego został stworzony specjalny podprogram, który jest wygodniejszy i zautomatyzowany. Możemy do niego wejść wybierając go w menu. Jedyne co musimy wybrać to algorytm, który ma zostać poddany testom oraz ilość testów, która zostanie wykonana dla każdej reprezentatywnej wartości (możliwość ustalenia w configu). Wszystkie algorytmy były testowane 100 krotnie, lecz można podać tam inną wartość. W konsoli dostajemy informacje o czasie wykonania dla każdego testu. Na koniec pojawia się podsumowanie, w którym widać uśrednione czasy dla wszystkich badanych wartości.

Ważne! Dla każdego kolejnego testu generowana jest nowa losowa macierz!

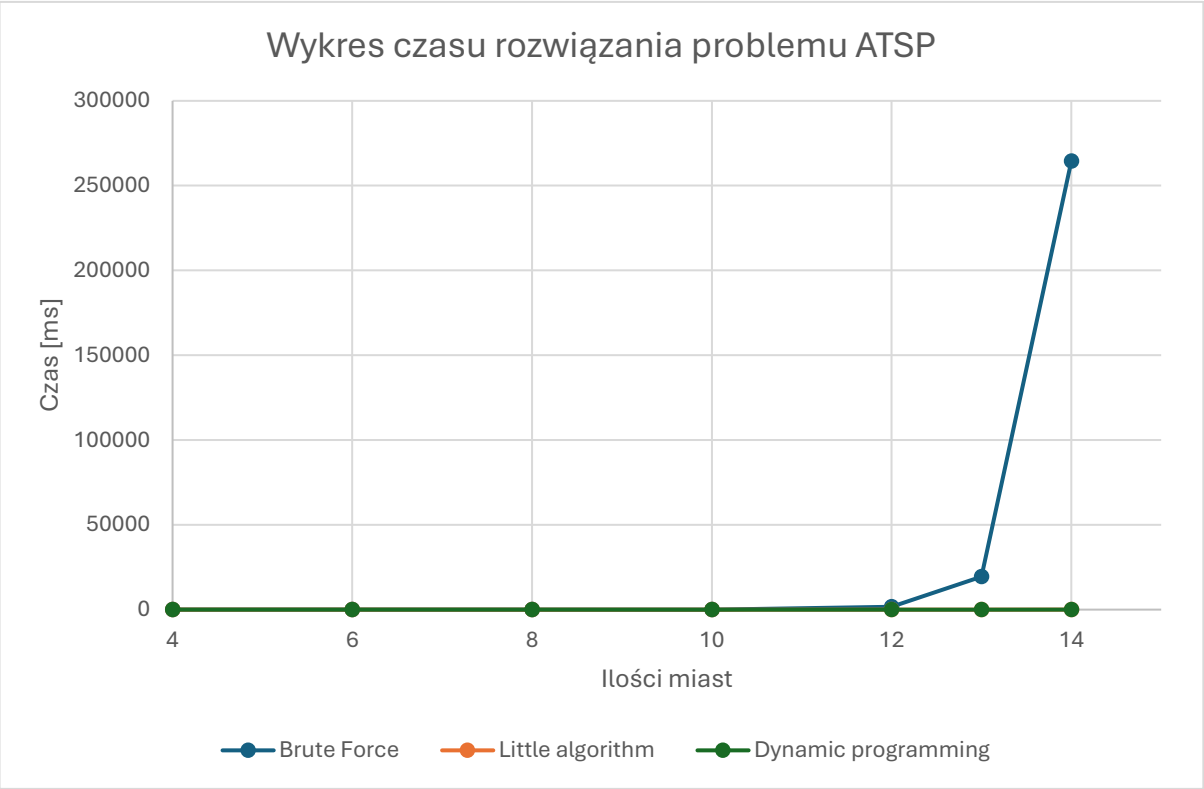
# 5 Wyniki badań

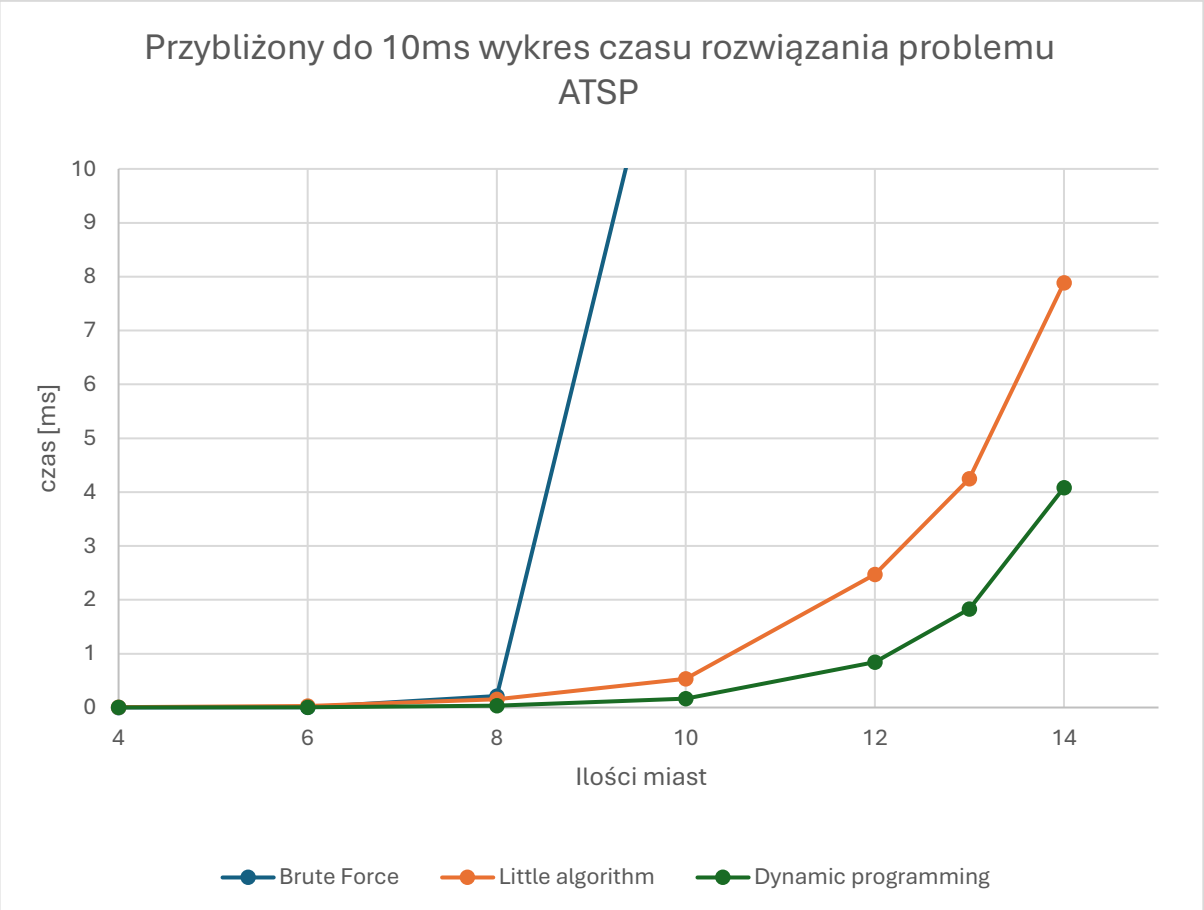
## 5.1 Tabela z głównymi wynikami

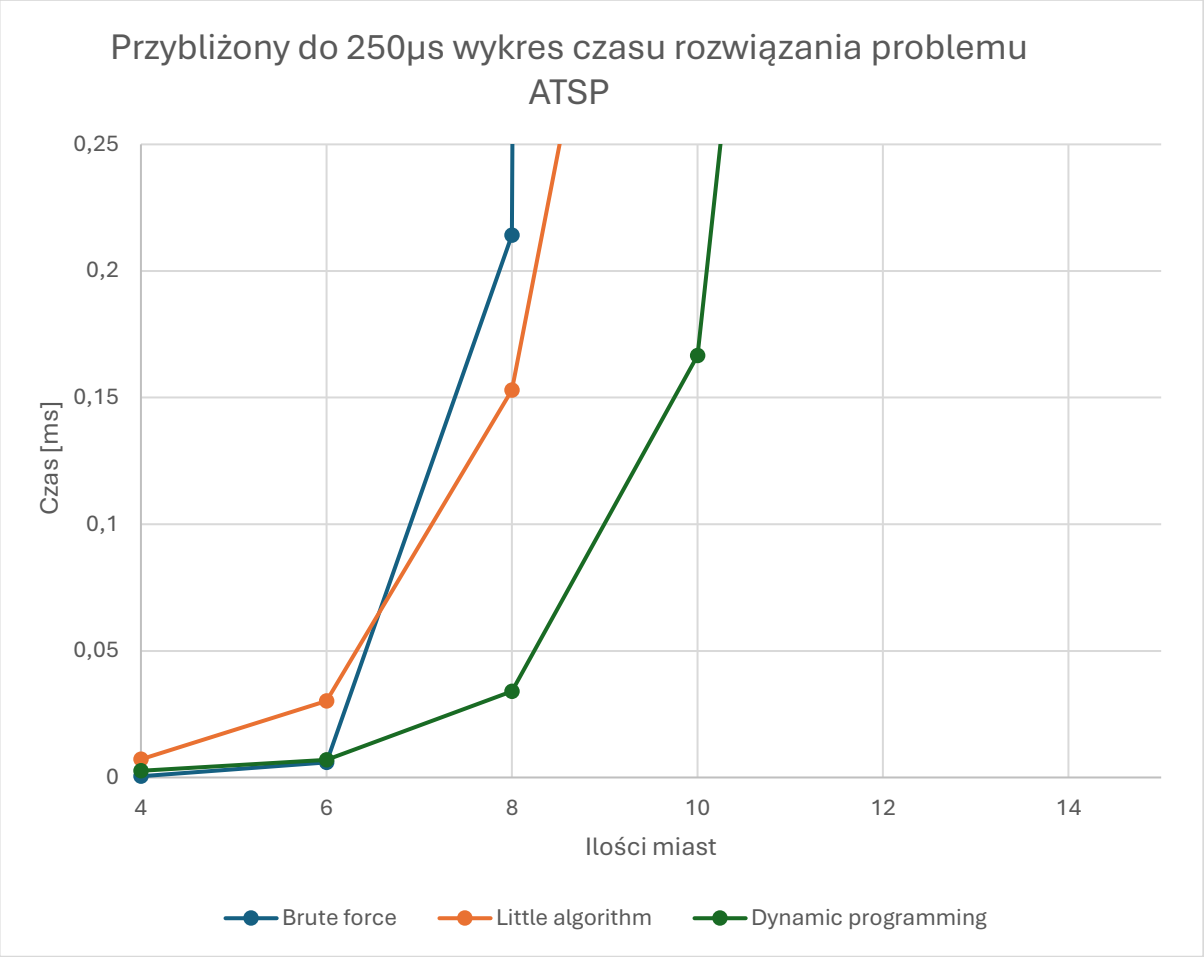
Tabela przedstawia czas wykonania każdego z algorytmów dla każdej z reprezentatywnych wartości.

| Algorytm → / Rozmiar problemu ↓ | Brute Force [ms] | Algorytm Little'a[ms] | Programowanie dynamiczne [ms] |
|---|---|---|---|
| 4 | 0,00048 | 0,00716 | 0,00264 |
| 6 | 0,0059 | 0,0302 | 0,0070 |
| 8 | 0,21 | 0,15 | 0,03 |
| 10 | 14,55 | 0,53 | 0,17 |
| 12 | 1720,22 | 2,47 | 0,84 |
| 13 | 19462,00 | 4,25 | 1,83 |
| 14 | 264348,00 | 7,88 | 4,08 |

## 5.2 Wykres z głównymi wynikami






# 6 Wnioski

Do podsumowania badań przypomnę jeszcze raz ile wynoszą złożoności obliczeniowe algorytmów poddanym badaniom i porównamy stosunek wzorcowy ze stosunkiem czasu, który uzyskałem. Jak dokładnie będzie obliczany stosunek wzorcowy? Złożoność obliczeniowa mówi, że dla N danych potrzebujemy wykonać ilość operacji, która jest uzależniona właśnie od tej złożoności. Najlepiej wytłumaczę to na konkretnym przykładzie : Algorytm o złożoności obliczeniowej O[(n-1)!]. Wynik dla N = 10 to 14,5534 ms, natomiast wynik dla N = 12 to 1720,22 ms. Obliczamy stosunek pary wyników $\frac{1720{,}22\ \text{ms}}{14{,}5534\ \text{ms}} = 118{,}2$. Ten wynik oznacza, że po zwiększeniu N o 2

czas wydłużył się 118,2 krotnie. Teraz obliczymy stosunek wzorcowy, czyli o ile powinien zwiększyć się czas przy takim wzroście N. Nie wiemy ile wynosi czas wykonania pojedynczej operacji, dlatego oznaczmy to jako x. Złożoność czasowa dla N = 10 to $9! * x\ ms$. Złożoność czasowa dla N = 12 to $11! * x\ ms$. Obliczamy stosunek czasów: $\frac{11! * x\ ms}{9! * x\ ms} = \frac{39\,916\,800}{362880} = 110$. Ten wynik oznacza, że według podanej złożoności zwiększając ilość N o 2 czas powinien zwiększyć się 110 razy. Korzystając z tego schematu porównam stosunki faktycznych czasów do stosunków wzorcowych wynikających ze złożoności obliczeniowej algorytmów.

## 6.1 Algorytm Brute Force

Algorytm Brute Force posiada złożoność obliczeniową O[n-1)!], gdy zakłada się startowanie z jednego wierzchołka. Algorytm wykorzystuje permutacje i bada wszystkie możliwe przypadki. Zaletą na pewno jest prostota tego algorytmu i niezawodność. Algorytm świetnie sprawdza się dla małych rozmiarów problemów. Jego największą wadą jest duża złożoność obliczeniowa dla większych problemów. Poniżej znajduje się porównanie stosunków czasów w celu weryfikacji złożoności obliczeniowej:

| Algorytm → / Rozmiar problemu ↓ | Brute Force [ms] | stosunek czasów | stosunek wzorcowy |
|---|---|---|---|
| 4 | 0,00048 | - | - |
| 6 | 0,00597 | 12 | 20 |
| 8 | 0,21 | 36 | 42 |
| 10 | 14,55 | 68 | 72 |
| 12 | 1720,22 | 118 | 110 |
| 13 | 19462 | 11 | 12 |
| 14 | 264348 | 14 | 13 |

Patrząc na stosunki czasów możemy zauważyć, że algorytm radzi sobie nieco lepiej dla rozmiaru 6 niż zakładano. Dla kolejnych rozmiarów stosunki są porównywalne do tych wzorcowych, więc jak najbardziej mogę zaakceptować, że złożoność obliczeniowa wynikająca ze wzoru jest poprawna.

## 6.2 Algorytm Little'a

Algorytm Little'a jest kolejnym algorytmem poddanym badaniom i jednoznaczne można stwierdzić, że radzi sobie o wiele lepiej niż poprzedni algorytm dla większych problemów. Szczególnie możemy zauważyć poprawę dla większych rozmiarów powyżej 8. Dla małych problemów takich jak 4 i 6 algorytm ten wypada gorzej od poprzednika. Algorytm podobno posiada dwie średnie złożoności obliczeniowe: pierwszą $O(2^n)$ i drugą $O(2^n \cdot n^2)$ dla średnich instancji problemu. Sprawdźmy jak wyglądają stosunki czasów i czy wzór na złożoność obliczeniową pokrywa się z jednym z naszych założeń.

| Algorytm → / Rozmiar problemu ↓ | Algorytm Little'a [ms] | stosunek czasów | stosunek wzorcowy $2^n$ | stosunek wzorcowy $2^n \cdot n^2$ |
|---|---|---|---|---|
| 4 | 0,00716 | - | - | - |
| 6 | 0,0302 | 4,22 | 4,00 | 9,00 |
| 8 | 0,15 | 5,06 | 4,00 | 7,11 |
| 10 | 0,53 | 3,47 | 4,00 | 6,25 |
| 12 | 2,47 | 4,65 | 4,00 | 5,76 |
| 13 | 4,25 | 1,72 | 2,00 | 2,35 |
| 14 | 7,88 | 1,86 | 2,00 | 2,32 |

Patrząc na stosunki czasów możemy zauważyć, że nasze czasy pokrywają się z pierwszym stosunkiem wzorcowym. Mimo lekkiego odchylenia wyników również mogę zaakceptować podaną złożoność obliczeniową, która wynosi $O(2^n)$.

## 6.3 Programowanie dynamiczne

Ostatnim badanym algorytmem jest programowanie dynamiczne, które wykorzystuje algorytm Bellmana-Helda-Karpa. Ten algorytm poradził sobie najlepiej z rozwiązaniem problemów dla naszych rozmiarów. Podobnie jak algorytm Little'a charakteryzuje się małym wzrostem czasu wykonania dla zwiększających się problemów. Jedynie dla rozmiarów 4 i 6 algorytm Brute Force okazał się być najlepszy. Programowanie dynamiczne posiada złożoność obliczeniową równą $O(2^n \cdot n^2)$, czyli taką samą jak wariant drugi w

algorytmie Little'a, który się nie sprawdził w naszych badaniach. Sprawdźmy, czy może w tym przypadku ta złożoność będzie poprawna.

| Algorytm → / Rozmiar problemu ↓ | Programowanie dynamiczne [ms] | stosunek czasów | stosunek wzorcowy |
|---|---|---|---|
| 4 | 0,00264 | - | - |
| 6 | 0,0070 | 2,65 | 9 |
| 8 | 0,03 | 4,85 | 7,11 |
| 10 | 0,17 | 4,89 | 6,25 |
| 12 | 0,84 | 5,05 | 5,76 |
| 13 | 1,83 | 2,17 | 2,35 |
| 14 | 4,08 | 2,23 | 2,32 |

Stosunki czasów dla małych problemów są o wiele mniejsze, niż wynika ze wzoru, ale dla większych problemów stosunki stają się coraz bardziej przybliżone do wzorcowych, więc w tym przypadku również stwierdzam, że złożoność obliczeniowa jest poprawna i wynosi $O(2^n \cdot n^2)$.

# 7 Dodatek – szukanie maksymalnego rozmiaru problemu.

## 7.1 Przebieg eksperymentu

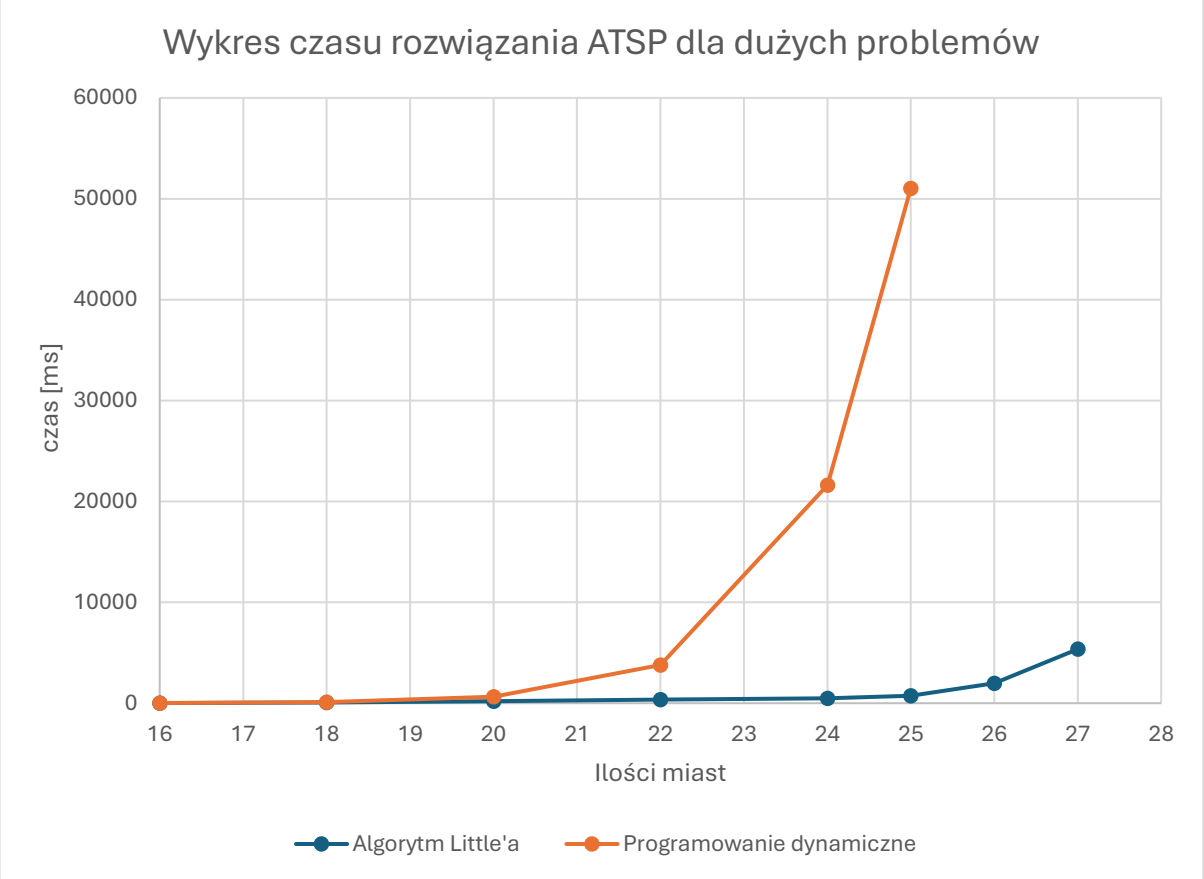
Do znalezienia maksymalnego rozmiaru problemu użyto generowania losowych macierzy o coraz większych rozmiarach i sprawdzono, kiedy program przestaje działać. Następnym krokiem było sprawdzenie, co jest ograniczeniem i w dwóch przypadkach okazało się, że jest to pamięć RAM, dlatego badaniu poddano dodatkową wartość jaką jest zapotrzebowanie na pamięć. Przeprowadziłem serię testów dla podanych rozmiarów problemu: **16, 18, 20, 22, 24, 25, 26, 27, 28, 29**. Aby uśrednić wyniki użyłem podprogramu, którego używałem wcześniej, ale musiałem ograniczyć ilość testów, ponieważ ograniczała mnie pamięć RAM komputera. Poniżej przedstawiam tabelę ze wszystkimi wynikami.

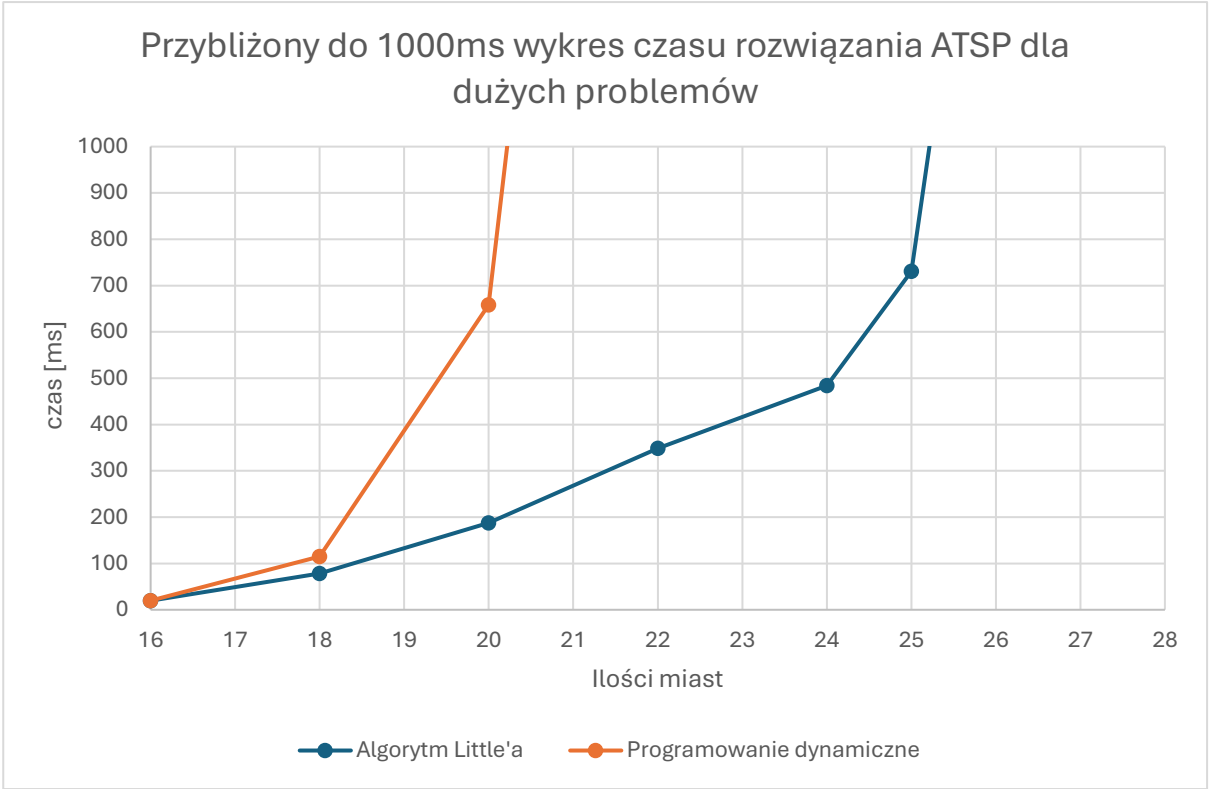
## 7.2 Wyniki

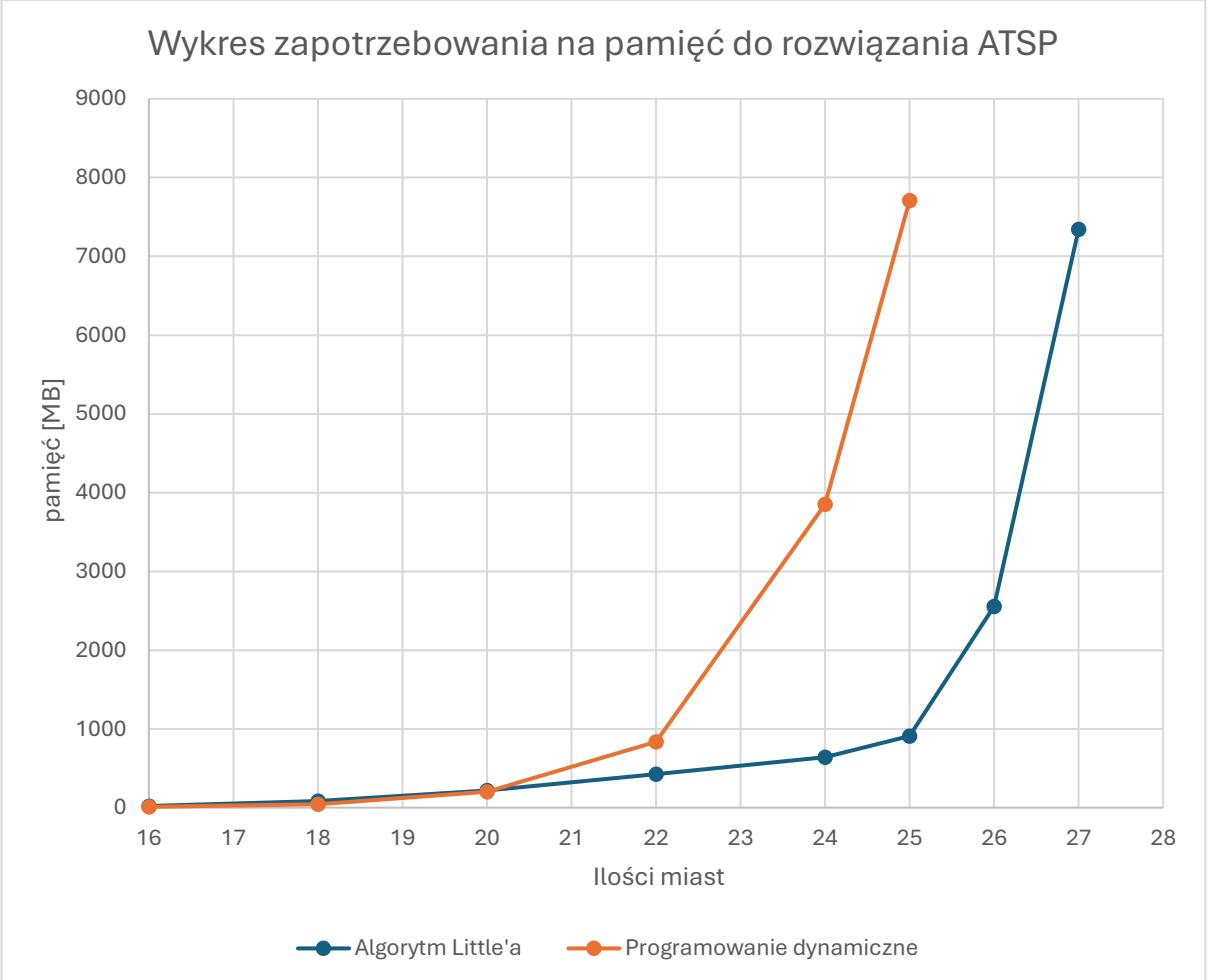
### 7.2.1 Tabela z głównymi wynikami

| | Pomiar czasu | | Zapotrzebowanie pamięci | |
|---|---|---|---|---|
| Algorytm → / Rozmiar problemu ↓ | Algorytm Little'a [ms] | Programowanie dynamiczne [ms] | Algorytm Little'a [MB] | Programowanie dynamiczne [MB] |
| 16 | 19,67 | 19,82 | 21 | 11 |
| 18 | 78,55 | 115,22 | 83 | 44 |
| 20 | 187,45 | 658,31 | 219 | 203 |
| 22 | 348,41 | 3777,00 | 427 | 834 |
| 24 | 484,25 | 21618,70 | 642 | 3850 |
| 25 | 731,07 | 51034,80 | 908 | 7710 |
| 26 | 1994,87 | - | 2555 | Brak pamięci |
| 27 | 5358,70 | - | 7345 | - |
| 28 | * | - | * | - |
| 29 | - | - | Brak pamięci | - |

### 7.2.2 Wykresy z głównymi wynikami




Przybliżony do 1000ms wykres czasu rozwiązania ATSP dla dużych problemów
czas [ms]
1000
900
800
700
600
500
400
300
200
100
0
16
17
18
19
20
21
22
23
24
25
26
27
28
Ilości miast
Algorytm Little'a
Programowanie dynamiczne


Wykres zapotrzebowania na pamięć do rozwiązania ATSP
pamięć [MB]
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
16
17
18
19
20
21
22
23
24
25
26
27
28
Ilości miast
Algorytm Little'a
Programowanie dynamiczne

## 7.3 Wnioski

### 7.3.1 Maksymalne rozmiary problemów

Wnioski zaczniemy od omówienia tabeli. Zaznaczyłem w niej momenty, kiedy zabrakło pamięci i algorytm nie mógł się wykonać do końca. Dla Programowania dynamicznego jest to rozmiar 26. Więc dla mojego sprzętu, w którym jest 16GB pamięci RAM algorytm programowania dynamicznego może rozwiązać problem ATSP maksymalnie dla **25** miast. Teraz przejdźmy do algorytmu Little'a, w którym zaznaczyłem * dla rozmiaru 28. Nie mogłem zrobić uśrednionego wyniku, ponieważ brakowało pamięci RAM dla większej ilości pomiarów, ale algorytm się wykonał przynajmniej 4 razy, lecz wyniki były bardzo różnorodne, dlatego stwierdziłem, że nie będę ich podawał. Zakres czasów jakie uzyskałem to od 904 ms do 5962 ms, natomiast zakres zapotrzebowania pamięci to od 1272 MB do 8218 MB.

Możliwe, że dla złożonego przypadku algorytm dla 28 miast już się nie wykona, ponieważ zabraknie pamięci, dlatego stwierdzam, że algorytm Little'a na pewno może rozwiązać problem ATSP maksymalnie dla **27** miast.

### 7.3.2 Zapotrzebowanie na pamięć

Przyjrzyjmy się teraz zapotrzebowaniu na pamięć obu algorytmów.

Dynamiczne programowanie radzi sobie lepiej dla mniejszych rozmiarów, ale powyżej wartości 20 zaczyna zwiększać swoje zapotrzebowanie o wiele szybciej niż algorytm Little'a, co skutkuje szybszym wyczerpaniem całej pamięci RAM. Algorytm Little'a bardzo dobrze sobie radzi do rozmiaru 25, ale po tym zapotrzebowanie również zaczyna drastycznie rosnąć.

### 7.3.3 Czas wykonywania algorytmów

Na koniec krótko przyjrzymy się czasom algorytmów. Początkowo algorytmy osiągają bardzo podobne oraz niskie czasy do rozmiaru problemu równego 20. Po tej wartości programowanie dynamiczne traci swoją cechę szybkości i

jego czas z każdym kolejnym większym rozmiarem zaczyna drastycznie rosnąc w porównaniu z algorytmem Little'a, dla którego zwiększanie się problemu nie wpływa aż tak bardzo.

### 7.3.4 Napotkane problemy

Podczas badania wydajności algorytmów dla większych problemów spotkałem się z problemem braku pamięci podczas prowadzenia masowych testów na jednym działaniu programu (bez resetu). Mimo przeprowadzenia serii testów dla jednego algorytmu i jej zakończenia, pamięć RAM nie była zwalniania. Gdy przeprowadzałem serię testów dla drugiego algorytmu program przestawał działać, ponieważ brakowało pamięci. Rozwiązaniem problemu było resetowanie programu po każdej serii testów, a dla większych problemów dodatkowo ograniczenie ilości testów.

# 8 Podsumowanie

Podsumowując przeprowadzone badani stwierdzam, że złożoność obliczeniowa jest zgodna ze złożonością wzorcową i algorytmy działają w pełni sprawnie. Kolejną, a zarazem najważniejszą rzeczą jest szybkość algorytmów i który algorytm jest najlepszy. Moim zdaniem nie możemy jednoznacznie stwierdzić, który algorytm jest najlepszy i wszystko zależy od rozmiaru problemu, który chcemy rozwiązać. Dla małych problemów, czyli **od 4 do 8** najlepszym wyborem będzie algorytm **Brute Force**, ponieważ dużą zaletą jest prostota tego algorytmu i jego porównywalnie szybkie działanie z resztą algorytmów, które są bardziej złożone. Dla średnich problemów, czyli **od 8 do 16** proponuję **Programowanie dynamiczne**, ponieważ w tym przedziale radzi sobie najszybciej i zużywa mniej pamięci niż algorytm Little'a. Duże problemy, czyli **od 16 do końca** pamięci RAM zostawiłbym dla algorytmu **Little'a**, ponieważ zwiększające się rozmiary nie wpływają

znacznie na czas wykonywania i wzrost zapotrzebowania na pamięć jest o wiele mniejszy niż w przypadku programowania dynamicznego.

# 9 Źródła

www.eduinf.waw.pl

www.geeksforgeeks.org

www.stackoverflow.com

www.tutorialspoint.com

www.medium.com