2021

Projektowanie Efektywnych Algorytmów

PROJEKT 1

KAROL WAŻNY (252716)

1. Polecenie

W ramach projektu należy zaimplementować oraz dokonać analizy efektywności algorytmu podziału i ograniczeń (B&B), programowania dynamicznego (DP) oraz przeglądu zupełnego dla asymetrycznego problemu komiwojażera (ATSP).

1.1. Dodatkowe wymagania

- używane struktury danych powinny być alokowane dynamicznie (w zależności od aktualnego rozmiaru problemu),
- program powinien umożliwić weryfikację poprawności działania algorytmu. W tym celu powinna istnieć możliwość wczytania danych wejściowych z pliku tekstowego,
- po zaimplementowaniu i sprawdzeniu poprawności działania algorytmu należy dokonać pomiaru czasu jego działania w zależności od rozmiaru problemu N (badania należy wykonać dla minimum 7 różnych reprezentatywnych wartości N),
- dla każdej wartości N należy wygenerować po 100 losowych instancji problemu (w sprawozdaniu należy umieścić tylko wyniki uśrednione),
- implementacje algorytmów należy dokonać zgodnie z obiektowym paradygmatem programowania.

2. Wstęp teoretyczny

Asymetryczny problem komiwojażera (ATSP – Asymmetric Travelling Salesman Problem) zdefiniowany jest jako zadanie znalezienia dla danego skierowanego grafu ważonego minimalnego cyklu przechodzącego przez wszystkie wierzchołki (minimalnego cyklu Hamiltona).

ATSP należy do klasy problemów NP.-trudnych, co oznacza, że nie ma algorytmu, który daje gwarancję obliczenia wyniku optymalnego w czasie wielomianowym.

Prezentowany program zawiera implementacje trzech algorytmów rozwiązujących problem komiwojażera: algorytmu Brute-Force, algorytmu Helda-Karpa oraz algorytmu podziału i ograniczeń (Branch and Bound). W prezentowanych przykładach praktycznych wykorzystany jest graf reprezentowany przez poniższą macierz incydencji (indeksowanie od 0):

—	81	50	18	75
81	—	76	21	37
50	76	—	24	14
18	21	24	—	19
75	37	14	19	—

2.1. Brute Force

2.1.1. Opis

Czyli rozwiązanie problemu przy użyciu przeglądu zupełnego. Złożoność czasowa zależy proporcjonalnie od przeszukiwanej przestrzeni rozwiązań, ta zaś zależy od liczby wierzchołków w grafie jak n!. Na wykorzystanym sprzęcie największa instancja, dla jakiej udało się otrzymać wynik drogą przeglądu zupełnego miała 12 wierzchołków.

Do zalet tego podejścia należy prostota implementacji oraz liniowa złożoność pamięciowa.

2.1.2. Złożoność

Czasowa	$O(n!)$
Pamięciowa	$O(n)$

2.1.3. Implementacja

Algorytm zaimplementowano w wersji iteracyjnej. Aktualnie rozważany cykl jest przechowywany jako tablica z indeksami wierzchołków. W ramach jednej iteracji obliczana jest wartość aktualnie rozważanego cyklu, ewentualnie jest aktualizowana wartość najlepsza do tej pory (jeżeli aktualna wartość jest mniejsza niż najlepsza dotąd) i wyznaczana jest kolejna permutacja wierzchołków. Do znajdowania kolejnych permutacji wierzchołków użyta jest metoda `nextPermutation()`.

2.2. Branch and Bound

2.2.1. Opis

B&B czyli metoda podziału i ograniczeń jest modyfikacją metody przeglądu zupełnego.

B&B opiera się na założeniu, że niejednokrotnie da się stwierdzić, że dana grupa rozwiązań nie zawiera rozwiązania optymalnego bez rozpatrywania każdego z tych rozwiązań z osobna.

Sercem tej metody jest heurystyka obliczana dla każdego węzła przy przechodzeniu drzewa rozwiązań i pozwalająca stwierdzić, czy dany węzeł jest „obiecujący”, czy też nie. Jeżeli dany węzeł nie jest obiecujący, nie ma potrzeby odwiedzania wszystkich węzłów potomnych. Im bliżej korzenia w drzewie

rozwiązań stwierdzimy, że dany węzeł jest nieobiecujący, tym więcej czasu zaoszczędzimy na odwiedzaniu jego węzłów potomnych.

Zastosowana heurystyka jest obliczana jako suma wag krawędzi dotychczas dodanych do rozwiązania zsumowana z wagami najlżejszych krawędzi wychodzących z węzłów niezawartych dotychczas w rozwiązaniu. Oczywiście istnieją inne, potencjalnie bardziej efektywne heurystyki, na tę zdecydowano się ze względu na jej prostotę.

W przypadku tego algorytmu złożoność czasowa zależy od konkretnej instancji i tego, jak wybrana heurystyka radzi sobie z daną instancją, jednak w praktyce przeważnie algorytm ten jest porównywalnie szybki lub szybszy niż pozostałe prezentowane.

2.2.2. Złożoność obliczeniowa

Czasowa	$O(n)$ do $O(n!)$
Pamięciowa	$O(n)$

2.2.3. Przykład praktyczny

Na początku tworzona jest tablica wartości najlżejszych krawędzi wychodzących z każdego wierzchołka (indeks w tablicy odpowiada wierzchołkowi):

[18, 21, 14, 18, 14]

Następnie wartości w tablicy są sumowane aby otrzymać minimalną możliwą wartość heurystyki:

$$18 + 21 + 14 + 18 + 14 = 85$$

Wartość najlepszej znalezionej dotąd ścieżki jest ustawiana na nieskończoność (reprezentowaną przez stałą INT32_MAX).

Następnie po kolei są odwiedzane kolejne węzły, a dla każdego jest liczona wartość heurystyki (dokładny sposób obliczania heurystyki znajduje się poniżej, w sekcji poświęconej implementacji):

- 0 -> 1 (Heurystyka: 85 – 18 + 81 = 148; mniejsza od najlepszej znalezionej dotąd, więc węzeł jest obiecujący)
- 0 -> 1 -> 2 (Heurystyka: 148 – 21 + 76 = 203; mniejsza od najlepszej znalezionej dotąd, więc węzeł jest obiecujący)
- 0 -> 1 -> 2 -> 3 (Heurystyka: 203 – 14 + 24 = 213; mniejsza od najlepszej znalezionej dotąd, więc węzeł jest obiecujący)
- 0 -> 1-> 2 -> 3 -> 4 (Nie ma więcej węzłów do wyboru, więc kończymy cykl(0-1-2-3-4-0), liczymy jego koszt (213 – 18 + 19 – 14 + 75 = 275) i aktualizujemy najlepszą znalezioną ścieżkę i jej koszt.)

Cofamy się do ostatniego obiecującego węzła, w którym mieliśmy wybór (0-1-2):

- 0 -> 1 -> 2 -> 4 (H: 203 – 14 + 14 = 203 < 275, węzeł obiecujący)
- 0 -> 1 -> 2 -> 4 -> 3 (Kończymy cykl; koszt: 203 – 14 + 19 – 18 + 18 = 208 < 275, więc aktualizujemy najlepszą znalezioną ścieżkę i jej koszt, cofamy się do ostatniego węzła, gdzie mamy niesprawdzony wybór):
- 0 -> 1 -> 3 (H: 148 – 21 + 21 = 148 < 208, węzeł obiecujący)
- 0 -> 1 -> 3 -> 2 (H: 148 – 18 + 24 = 154 < 208, węzeł obiecujący)
- 0 -> 1 -> 3 -> 2 -> 4 (Kończymy cykl, koszt: 154 – 14 + 14 – 14 + 75 = 215 > 208, nie aktualizujemy najlepszej ścieżki, cofamy się do ostatniego węzła, gdzie został niesprawdzony wybór)
- 0 -> 1 -> 3 -> 4 (H: 148 – 18 + 19 = 149 < 208, węzeł obiecujący)

- 0 -> 1 -> 3 -> 4 -> 2 (Cykl, koszt: 149 – 14 + 14 – 14 + 50 = 185 < 208, aktualizujemy najlepszą ścieżkę i cofamy się do ostatniego węzła z niesprawdzonym wyborem)
- 0 -> 1 -> 4 (H: 148 – 21 + 37 = 164 < 185, węzeł obiecujący)
- 0 -> 1 -> 4 -> 2 (H: 164 – 14 + 14 = 164 < 185, węzeł obiecujący)
- 0 -> 1 -> 4 -> 2 -> 3 (Cykl, koszt: 164 – 14 + 24 – 18 + 18 = 174 < 185, aktualizujemy najlepszą ścieżkę i wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 1 -> 4 -> 3 (H: 164 – 14 + 19 = 169 < 174, węzeł obiecujący)
- 0 -> 1 -> 4 -> 3 -> 2 (Cykl, koszt: 169 – 18 + 24 – 14 + 50 = 211 > 174, nie aktualizujemy, wracamy w górę drzewa do węzłów z niesprawdzonymi wyborami)
- 0 -> 2 (H: 85 – 18 + 50 = 117 < 174, węzeł obiecujący)
- 0 -> 2 -> 1 (H: 117 – 14 + 50 = 153 < 174, węzeł obiecujący)
- 0 -> 2 -> 1 -> 3 (H: 153 – 21 + 21 = 153 < 174, węzeł obiecujący)
- 0 -> 2 -> 1 -> 3 -> 4 (Cykl, koszt: 153 – 18 + 19 – 14 + 75 = 215 > 174, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 2 -> 1 -> 4 (H: 153 – 21 + 37 = 169 < 174, węzeł obiecujący)
- 0 -> 2 -> 1 -> 4 ->3 (Cykl, koszt: 169 – 14 + 19 – 18 + 18 = 174 = 174, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami).
- 0 -> 2 -> 3 (H: 117 – 14 + 24 = 127 < 174, węzeł obiecujący)
- 0 -> 2 -> 3 -> 1 (H: 127 – 18 + 21 = 130 < 174, węzeł obiecujący)
- 0 -> 2 -> 3 -> 1 -> 4 (Cykl, koszt: 130 – 21 + 37 – 14 + 75 = 207 > 174, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 2 -> 3 -> 4 (H: 127 – 18 + 19 = 128 < 174, węzeł obiecujący)
- 0 -> 2 -> 3 -> 4 -> 1 (Cykl, koszt: 128 – 14 + 37 – 21 + 81 = 211 > 174, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami).
- 0 -> 2 -> 4 (H: 117 – 14 + 14 = 117 < 174, węzeł obiecujący)
- 0 -> 2 -> 4 -> 1 (H: 117 – 14 + 37 = 140 < 174, węzeł obiecujący)
- 0 -> 2 -> 4 -> 1 -> 3 (Cykl, koszt: 140 – 21 + 21 – 18 + 18 = 140 < 174, aktualizujemy najlepszą ścieżkę i wracamy do węzłów z niesprawdzonymi wyborami)
- 0 -> 2 -> 4 -> 3 (H: 117 – 14 + 19 = 122 < 140, węzeł obiecujący)
- 0 -> 2 -> 4 -> 3 -> 1 (Cykl, koszt: 122 – 18 + 21 – 21 + 81 = 185 > 140, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 (H: 85 – 18 + 18 = 85 < 140, węzeł obiecujący)
- 0 ->3 -> 1 (H: 85 – 18 + 21 = 88 < 140, węzeł obiecujący)
- 0 -> 3 -> 1 -> 2 (H: 88 – 21 + 76 = 143 > 140, węzeł nieobiecujący, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 1 -> 4 (H: 88 – 21 + 37 = 104 < 140, węzeł obiecujący)
- 0 -> 3 -> 1 -> 4 -> 2 (Cykl, koszt: 104 – 14 + 14 – 14 + 50 = 140 = 140, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 2 (H: 85 – 21 + 24 = 88 < 140, węzeł obiecujący)
- 0 -> 3 -> 2 -> 1 (H: 88 – 14 + 76 = 150 > 140, węzeł nieobiecujący, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 2 -> 4 (H: 88 – 14 + 14 = 88 < 140, węzeł obiecujący)
- 0 -> 3 -> 2 -> 4 -> 1 (Cykl, koszt: 88 – 14 + 75 – 21 + 81 = 209 > 140, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 4 (H: 85 – 18 + 19 = 86 < 140, węzeł obiecujący)
- 0 -> 3 -> 4 -> 2 (H: 86 – 14 + 37 = 109 < 140, węzeł obiecujący)

- 0 -> 3 -> 4 -> 2 -> 1 (Cykl, koszt: 109 – 14 + 76 – 21 + 81 = 231 > 140, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 4 -> 1 (H: 86 – 14 + 75 = 147 > 140, węzeł nieobiecujący, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 3 -> 4 -> 2 (H: 86 – 14 + 14 = 86 < 140, węzeł obiecujący)
- 0 -> 3 -> 4 -> 2 -> 1 (Cykl, koszt: 86 – 14 + 76 – 21 + 86 = 213 > 140, nie aktualizujemy, wracamy do ostatniego węzła z niesprawdzonymi wyborami)
- 0 -> 4 (H: 85 – 18 + 75 = 142 > 140, węzeł nieobiecujący, wracamy do ostatniego węzła z niesprawdzonymi wyborami).

Nie ma więcej węzłów do sprawdzenia, zatem najlepszy cykl to [0, 2, 4, 1, 3, 0] z kosztem 140.

2.2.4. Szczegóły implementacji

Algorytm zaimplementowano w wersji rekurencyjnej. Przechowywana jest kolejka (implementowana jako tablica) z niewykorzystanymi węzłami, stos (implementowany w tablicy) zawierający dotychczas zbudowaną ścieżkę oraz tablica zawierający koszty minimalnych krawędzi wychodzących z każdego węzła.

Każde wywołanie funkcji rekurencyjnej składa się z iteracji przez wszystkie niewykorzystane węzły. Pojedyncza iteracja składa się z:

1. Pobrania węzła z kolejki.
2. Obliczenia nowej wartości heurystyki i sprawdzenia, czy ten węzeł jest obiecujący.
3. Jeżeli węzeł jest obiecujący, jest dodawany do dotychczas zbudowanej ścieżki, funkcja jest wywoływana rekurencyjnie, po czym węzeł jest „odczepiany” od ścieżki.
4. Węzeł jest dodawany z powrotem na końcu kolejki.

W przypadku, gdy po wywołaniu funkcji okazuje się, że nie ma więcej niewykorzystanych węzłów, obliczana jest koszt cyklu. Jeżeli koszt jest mniejszy niż koszt najlepszego dotychczas znalezionego cyklu, koszt i najlepszy dotychczas cykl są aktualizowane.

2.2.4.1. Obliczanie heurystyki

Zastosowana funkcja obliczająca heurystykę po dodaniu węzła do ścieżki działa następująco:

1. Odczytuje wartość heurystyki przed dodaniem węzła do ścieżki.
2. Odejmuje wartość minimalnej krawędzi wychodzącej z przedostatniego węzła.
3. Dodaje wartość krawędzi z przedostatniego do ostatniego (świeżo dodanego) węzła w ścieżce.

Wartość heurystyki dla pustej ścieżki to suma wartości minimalnych krawędzi wychodzących z każdego węzła.

Jeżeli obliczona w ten sposób wartość jest większa niż koszt najlepszej znalezionej do tej pory ścieżki, węzeł jest nieobiecujący i należy wrócić do ostatniego węzła z niesprawdzonymi wyborami.

2.3. Programowanie dynamiczne

2.3.1. Opis

Strategia programowania dynamicznego jest reprezentowana przez algorytm Helda-Karpa.

Aby przedstawić zasadę działania tego algorytmu zdefiniujmy sobie najpierw funkcję $g(S, v)$ przyjmującą dwa argumenty: zbiór wierzchołków S, taki że nie należy do niego wierzchołek startowy x_1, ani wierzchołek v, oraz wierzchołek v różny od wierzchołka startowego. Wówczas wartość $g(S, v)$ jest zdefiniowana jako koszt najkrótszej drogi przechodzącej z wierzchołka początkowego x_1 do wierzchołka v i przechodzącej przez każdy wierzchołek z S dokładnie raz.

Działanie algorytmu opiera się na spostrzeżeniu, że:

$$g(S, v) = \min(g(S \backslash x_i) \ + \ d(x_i, v))$$

gdzie x_i to wierzchołki należące do S.

Spostrzeżenie to pozwala rozwiązać problem wyznaczenia $g(S, v)$ przez podział i rozwiązanie tego problemu dla mniejszych instancji.

W przedstawionej implementacji wartości $g(S, v)$ są kolejno obliczane dla coraz większych S, aż do momentu, gdy dochodzimy do zbiorów składających się ze wszystkich wierzchołków oprócz x_1 i v. Wówczas pozostaje znaleźć wartość:

$$\min(g(V \backslash \{x_1, x_i\}) + d(x_i, x_1))$$

Znaleziona wartość jest kosztem minimalnego cyklu odwiedzającego wszystkie wierzchołki. Jeżeli oprócz wartości $g(S, v)$ będziemy przechowywać też kolejność wierzchołków w S, która odpowiada znalezionemu najmniejszemu kosztowi, można na tym etapie łatwo odtworzyć szukany cykl.

Przedstawiona implementacja algorytmu znajduje wartości funkcji g po kolei dla coraz większych S. Dla każdej pary S i v przechowywane jest rozwiązanie częściowe zawierające zbiór S, wierzchołek v, kolejność wierzchołków w dotychczasowej ścieżce i koszt rozwiązania częściowego.

Aby znaleźć rozwiązania częściowe dla następnego rozmiaru S (następnego poziomu rozwiązań częściowych), dla każdego rozwiązania częściowego obliczane są wszystkie możliwe rozwiązania pochodne od tego rozwiązania częściowego w następujący sposób:

1. Tworzony jest nowy zbiór S' przez dodanie v do zbioru S.
2. Dotychczasowa ścieżka jest wydłużana przez dołożenie na końcu wierzchołka v.
3. Dla każdego wierzchołka x_i niebędącego wierzchołkiem startowym i nienależącego do S' tworzone jest rozwiązanie częściowe składające się ze zbioru S', ścieżki utworzonej w punkcie 2., wierzchołka x_i i kosztu obliczonego jako $g(S, v) + d(v, x_i)$.
4. Każde z utworzonych w ten sposób rozwiązań jest porównywane z najlepszym dotychczas znalezionym rozwiązaniem częściowym dla pary S', x_i. Jeżeli nowe rozwiązanie częściowe okaże się mieć lepszy koszt niż najlepsze dotychczas znalezione, aktualizowany jest koszt i kolejność najlepszego rozwiązania i znalezione rozwiązanie częściowe staje się najlepszym dotychczas.
5. Procedura jest powtarzana aż znalezione będą wartości $g(S, v)$ dla wszystkich zbiorów składających się ze wszystkich węzłów oprócz x_1 oraz v.

Za każdym razem przechowywane są tylko dwa „poziomy” rozwiązań częściowych, co pozwala ograniczyć potrzebną pamięć.

2.3.2. Złożoność obliczeniowa

Czasowa	$O(n^2 2^n)$
Pamięciowa	$O(n2^n)$

2.3.3. Implementacja

Na potrzeby przechowywania stanu wykonywania algorytmu utworzono klasę `PartialSolution`. Klasa ta zawiera listę wiązaną z dotychczasową ścieżką (oprócz węzła ostatniego i startowego), drzewo czerwono-czarne zawierające wszystkie węzły na dotychczas skonstruowanej ścieżce (oprócz ostatniego i startowego), ostatnio dodany węzeł i koszt dotychczasowej ścieżki.

Rozwiązania częściowe są przechowywane w tablicy tablic. Pozycja w zewnętrznej tablicy (pierwszy indeks) jest określona jednoznacznie przez zbiór węzłów w drzewie czerwono-czarnym. Pozycja w tablicy wewnętrznej (drugi indeks) jest określana przez węzeł końcowy. Takie podejście pozwala uniknąć wyszukiwania w tablicy i porównywania zbiorów – są to operacje kosztowne. Zamiast tego w oparciu o zawartość drzewa (węzły w zbiorze S rozwiązania częściowego) obliczany jest indeks w tablicy zewnętrznej. Jeżeli takie same zbiory zawsze znajdą się w tym samym miejscu, nie trzeba się zastanawiać, czy są równe – wystarczy, że są pod tym samym indeksem w tablicy.

Wzór poniżej to wzór służący do obliczania indeksu w tablicy zewnętrznej. Zakładamy, że:

- n to ilość węzłów w instancji problemu minus 1
- k to liczba węzłów w zbiorze S
- X to tablica z indeksami wierzchołków grafu wejściowego (wierzchołki numerowane od 0, ale wierzchołek 0 jest wierzchołkiem startowym, także najmniejsza wartość w tablicy X nie będzie mniejsza niż 1) znajdującymi się w zbiorze S i uporządkowanymi rosnąco.

$$Indeks = \sum_{i=0}^{k-1} \binom{n - X[i]}{k - i}$$

Algorytm zaczyna się od przygotowania tablicy dla zerowego poziomu rozwiązań częściowych – zewnętrzna tablica jest jednoelementowa, tablica wewnętrzna zawiera n – 1 rozwiązań częściowych; każde z rozwiązań częściowych składa się z pustego zbioru S, pustej ścieżki, węzła końcowego x_i oraz kosztu rozwiązania równego $d(x_0, x_i)$.

Następnie algorytm przeprowadza n – 1 iteracji przez tablicę zewnętrzną rozwiązań częściowych poprzedniego poziomu, tworząc tablicę tablic rozwiązań częściowych kolejnego poziomu.

Przebieg jednej iteracji:

1. Przypisanie tablicy tablic rozwiązań opracowanej w poprzednim poziomie jako tablicy poprzedniego poziomu.
2. Utworzenie tablicy tablic rozwiązań o odpowiednich wymiarach jako tablicy aktualnego poziomu.
3. Przeiterowanie przez rozwiązania częściowe poprzedniego poziomu i obliczenie rozwiązań częściowych aktualnego poziomu.

Przebieg utworzenia nowych rozwiązań częściowych z dotychczasowego rozwiązania częściowego:

1. Dodanie węzła końcowego do ścieżki dotychczasowej i zbioru S (reprezentowanego jako drzewo).
2. W oparciu o zawartość zbioru S obliczenie indeksu w tablicy zewnętrznej rozwiązań częściowych kolejnego poziomu, pod którym będziemy przechowywać rozwiązania pochodne

od aktualnie przetwarzanego i utworzenie uporządkowanej rosnąco listy węzłów nieznajdujących się w S.

3. Dla każdego z węzłów nieznajdujących się w S obliczany jest koszt rozwiązania częściowego powstałego przez dodanie go jako ostatniego węzła do rozważanego rozwiązania częściowego.
4. Jeżeli koszt rozwiązania częściowego znajdującego się w tworzonej tablicy pod indeksem `[i][j]`, gdzie i to identyfikator obliczony dla zbioru S, a j to indeks węzła końcowego na liście węzłów nienależących do S, jest większy niż koszt obliczony dla właśnie utworzonego rozwiązania, koszt i kolejność węzłów w ścieżce dotychczasowego rozwiązania częściowego są zastępowane kosztem rozwiązania właśnie utworzonego (zawartość drzewa jest kopiowana tylko, jeżeli właśnie obliczone rozwiązanie częściowe jest pierwszym obliczonym dla tej pary S, v).

Po wykonaniu tych wszystkich iteracji dla każdego z uzyskanych rozwiązań częściowych obliczany jest koszt zamknięcia cyklu powstałego z danego rozwiązania częściowego jako suma dotychczasowego kosztu i kosztu krawędzi od ostatniego węzła do węzła startowego; rozwiązanie o najmniejszym koszcie jest zwracane jako rozwiązanie problemu.

3. Pomiary czasu

3.1. Plan

Eksperyment polega na pomiarze czasu dla każdego z algorytmów dla siedmiu różnych wielkości instancji: 4, 6, 8, 10, 12, 15 i 20 węzłów (dla przeglądu zupełnego nie wykonano pomiarów dla 15 i 20 węzłów, ponieważ okazało się to niemożliwe –trwało to zbyt długo). Z większymi niż 20 węzłów instancjami nie radził sobie algorytm Helda-Karpa.

Dla każdego rozmiaru wygenerowano 100 instancji problemu, po czym zmierzono czas obliczenia rozwiązania przez każdy z algorytmów. Czasy zsumowano, aby uzyskać wartość średnią.

3.2. Generowanie instancji problemu

Macierze incydencji dla generowanych instancji problemu generowano jako macierze N x N wartości losowych o rozkładzie równomiernym z zakresu od 0 do 300, zatem nie były to instancje problemu, które mogłyby reprezentować realną sytuację z miastami i drogami.

3.3. Metoda pomiaru czasu

Do pomiaru czasu użyto klasy `StopWatch`. Klasa ta ma cztery metody: jedna do rozpoczęcia pomiaru, jedna do zakończenia pomiaru i dwie do odczytania czasu od rozpoczęcia do zakończenia pomiaru (jedna zwraca wynik jako liczbę całkowitą, druga jako liczbę zmiennoprzecinkową). W tym przypadku użyto metody zwracającej liczbę zmiennoprzecinkową, ponieważ liczba całkowita (zależnie od zastosowanego mnożnika) albo nie dawała zadowalającej precyzji dla małych instancji, albo dochodziło do przekroczenia zakresu – liczba całkowita się „przekręcała” i wynik wychodził nieprawidłowy.

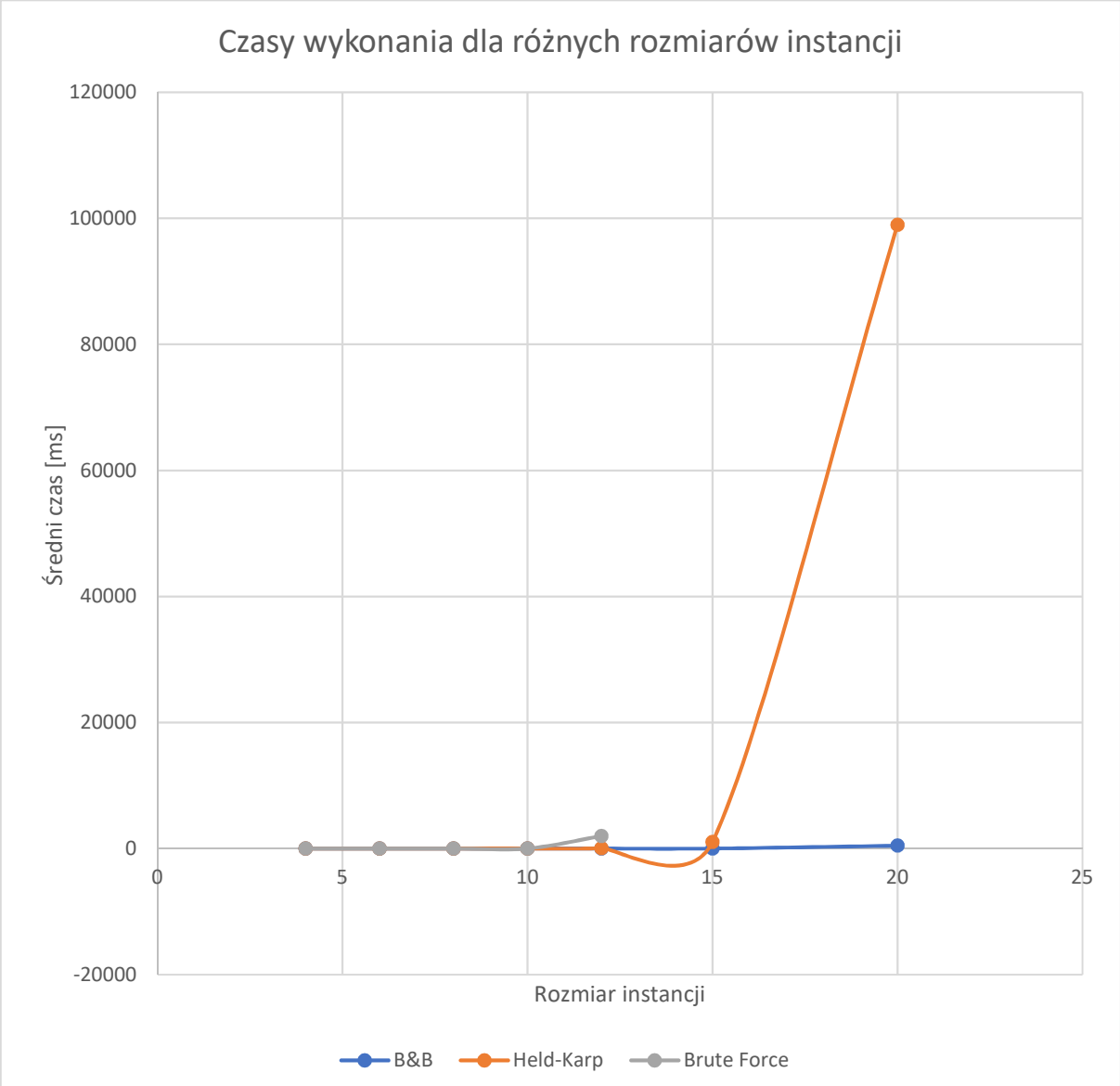
Aby zmierzyć czas, tworzono instancję problemu, instancję klasy reprezentującej algorytm oraz obiekt klasy `StopWatch`, po czym wywoływano metodę `start()` stopera, przekazywano instancję problemu do metody `solveFor()` obiektu algorytmu (co bezpośrednio prowadziło do rozwiązania problemu) i zatrzymywano stoper metodą `stop()`. Następnie odczytywano zmierzoną wartość czasu.

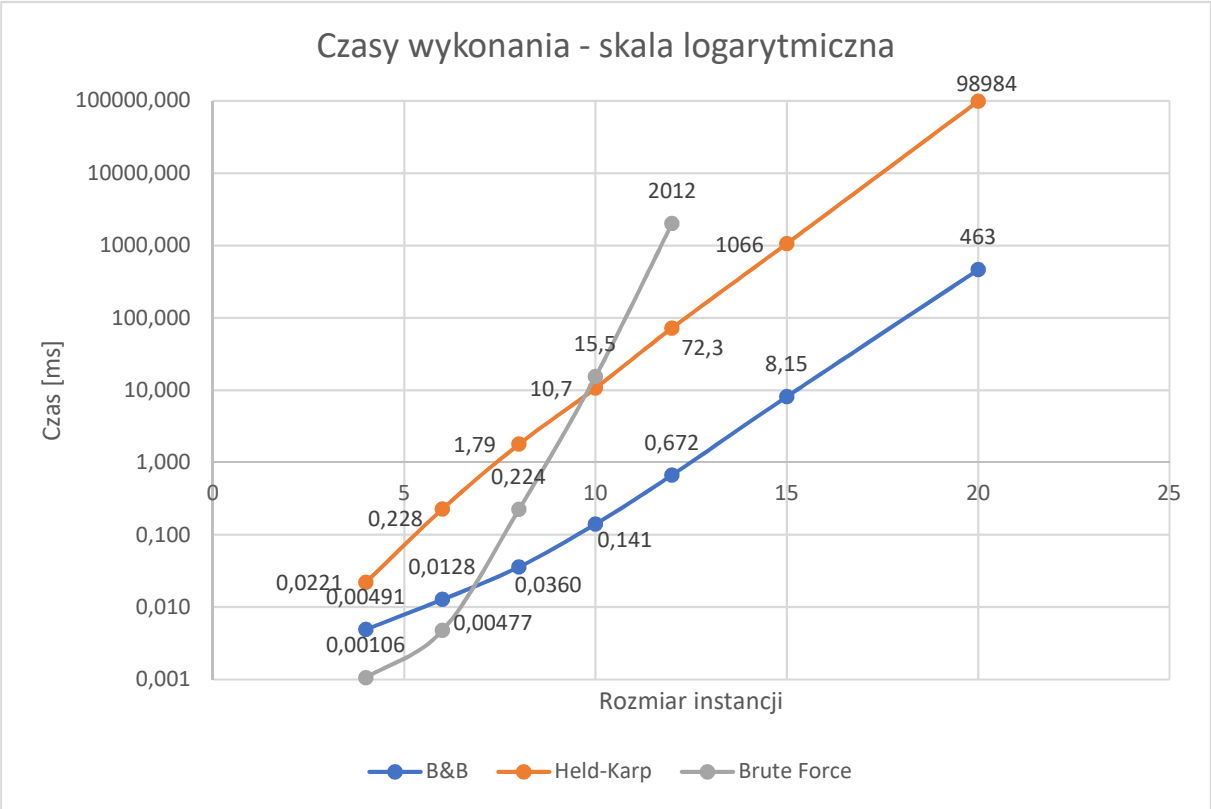
4. Uzyskane wyniki

Uzyskane wyniki przedstawia tabela i wykresy.

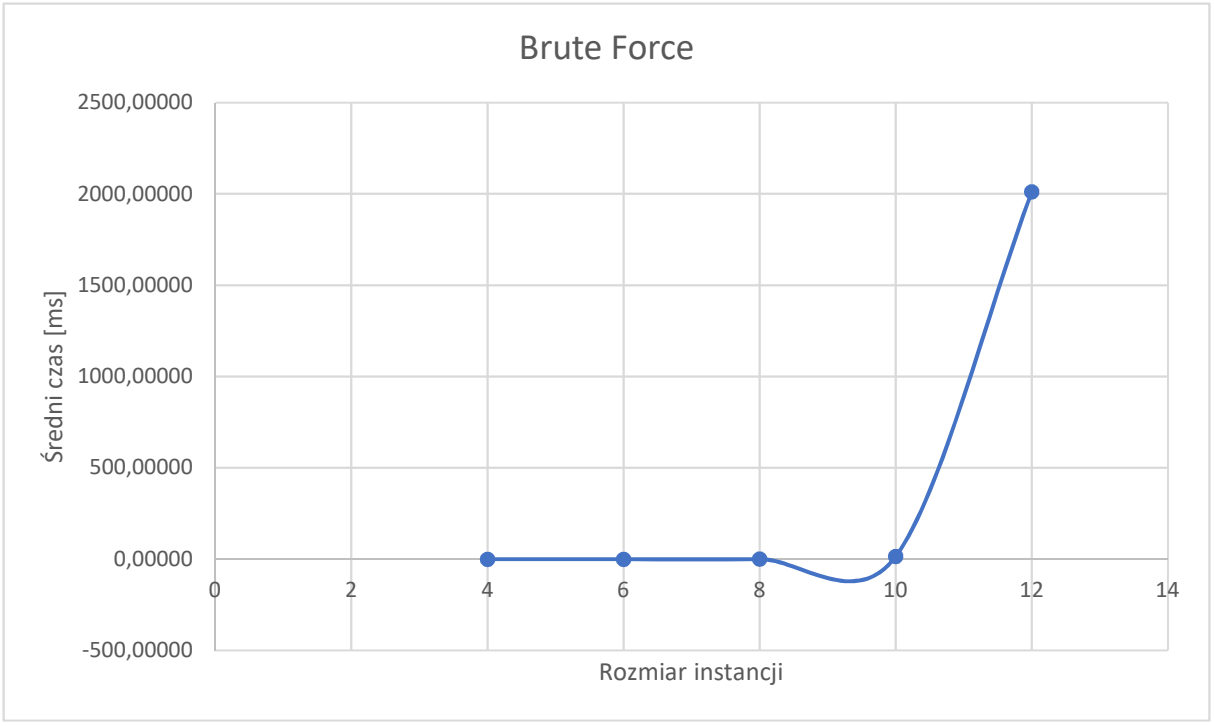
N	B&B	Held-Karp	Brute Force
[1]	[ms]	[ms]	[ms]
4	0,00491	0,0221	0,00106
6	0,0128	0,228	0,00477
8	0,0360	1,79	0,224
10	0,141	10,7	15,5
12	0,672	72,3	2012
15	8,15	1066	
20	463	98984	

4.1. Wszystkie metody

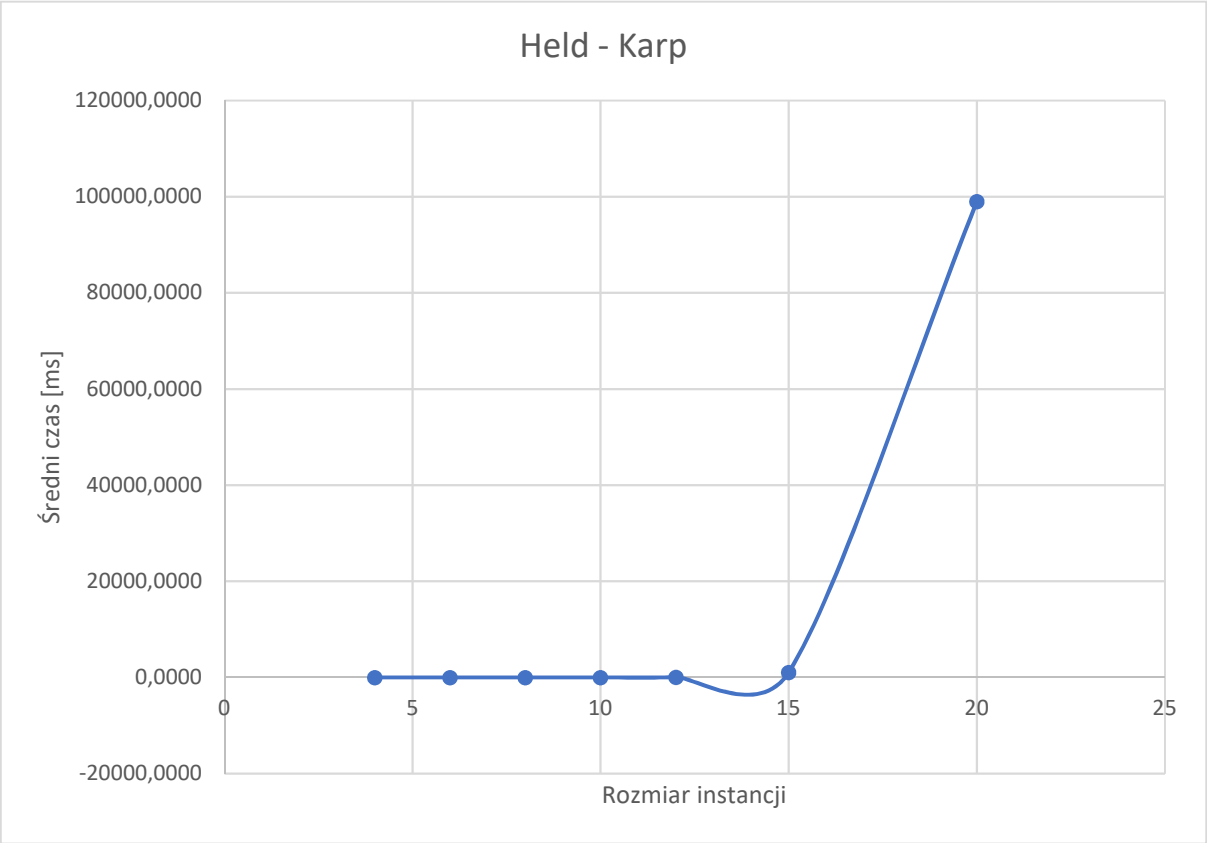




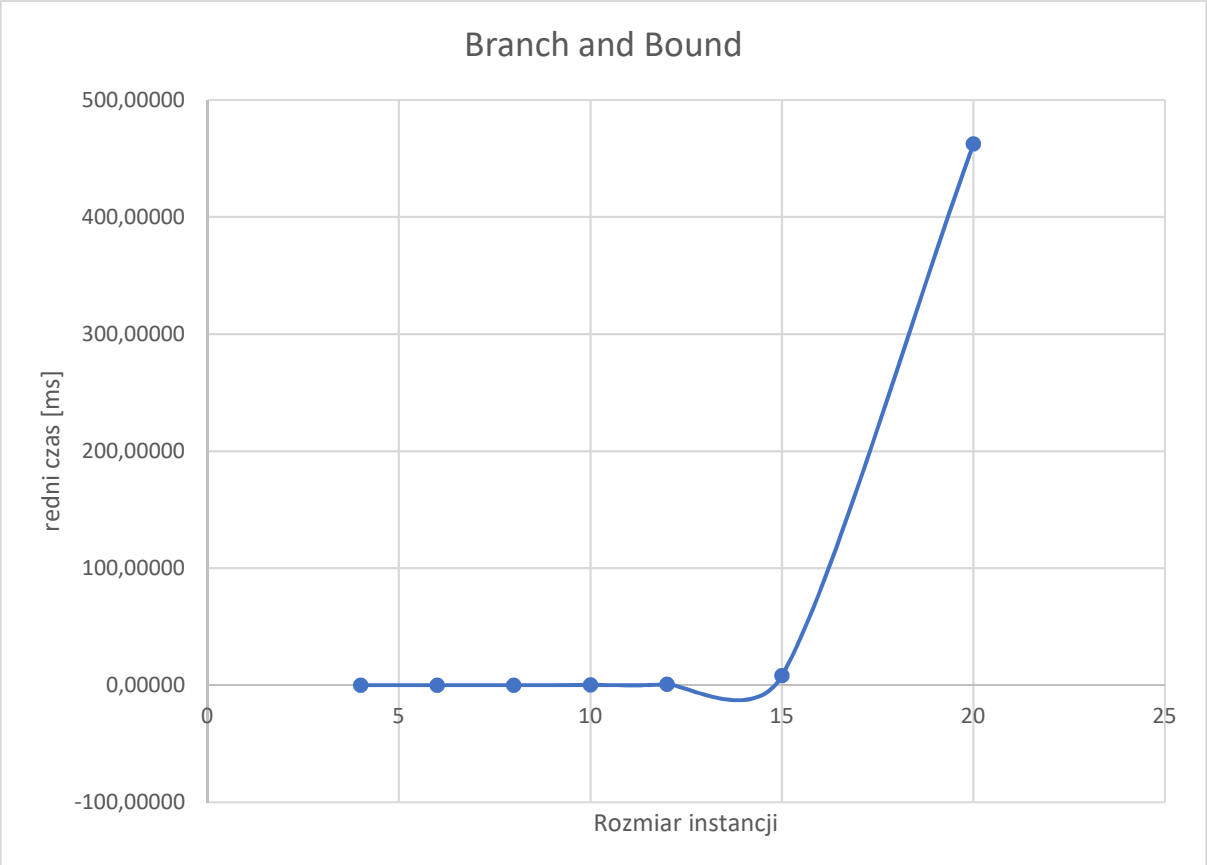
4.2. Brute Force



4.3. Held-Karp



4.4. Branch and Bound



5. Wnioski i uwagi końcowe

Wykresy czasu wykonania od wielkości instancji dla wszystkich algorytmów wyglądają bardzo podobnie. Dopiero narysowanie zależności dla wszystkich metod na jednym wykresie pokazuje relacje pomiędzy nimi. Ze względu na charakter zmian (najpierw bardzo małe, potem – stosunkowo – bardzo duże) lepiej porównuje się wyniki naniesione na skalę logarytmiczną niż liniową.

Zwłaszcza na skali logarytmicznej widać, że dla skrajnie małych instancji metoda przeglądu zupełnego działa najszybciej – narzut związany z większym stopniem skomplikowania (dodatkowe obliczenia, alokacje dodatkowych struktur) metod programowania dynamicznego i podziału i ograniczeń jest większy niż zysk na złożoności obliczeniowej.

Sytuacja zmienia się dla instancji o N=8 – dla tego rozmiaru narzut związany z dodatkowymi obliczeniami w metodzie Branch and Bound zaczyna się opłacać i czas wykonania jest mniejszy niż dla przeglądu zupełnego.

Metoda programowania dynamicznego „wyprzedza" przegląd zupełny dopiero dla 10 miast; jednak dla większych instancji przewaga programowania dynamicznego rośnie – dla 12 wierzchołków czas wykonania przeglądu zupełnego jest już o dwa rzędy wielkości większy niż czas wykonania algorytmu Helda-Karpa; dla większych instancji nie udało się uzyskać wyniku metodą przeglądu zupełnego (stąd brak danych).

Warto zwrócić uwagę na bardzo małe czasy wykonania algorytmu Branch and Bound – najprawdopodobniej sposób generowania instancji faworyzował ten algorytm i wykorzystaną heurystykę. Na potrzeby kolejnych etapów projektu zostanie opracowany nowy algorytm generowania instancji – wagi krawędzi będą odpowiadały odległościom pomiędzy miastom (przypadki bardziej „realne") – to pozwoli zweryfikować użyteczność zastosowanej heurystyki w realnych zastosowaniach.

Wszystkie zgromadzone dane pokazują, że algorytmy dające gwarancję rozwiązania optymalnego dla problemów NP-trudnych są mało efektywne i nie dają się zastosować dla dużych instancji problemu.

Pokazują też, że niekiedy algorytm o gorszej złożoności obliczeniowej (teoretycznej) dla małej instancji problemu wykona się szybciej niż algorytm o złożoności lepszej.

6. Spis treści