Projektowanie Efektywnych Algorytmów

Kierunek *Informatyka*	**Termin** *Czwartek 19:05*
Temat *Algorytmy dokładne*	**Problem** *(A)TSP*
Skład grupy *241284 Jakub Plona*	**Nr grupy** -
Prowadzący *Mgr inż. Radosław Idzikowski*	**data** *21 listopada 2019*

Spis treści

1 Opis problemu

Jak można przeczytać na Wikipedii,

> problem komiwojażera (ang. travelling salesman problem, TSP) to zagadnienie optymalizacyjne, polegające na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

Problem ten można przedstawić bez wykorzystania terminologii teorii grafów. Problem komiwojażera jest problemem optymalizacyjnym z danymi wejściowymi w postaci zbioru n miast oraz danymi kosztami przejścia między dowolnie wybraną parą miast, za wyjątkiem przejścia z danego miasta do tego samego miasta (przejście to nie istnieje). W problemie tym poszukuje się drogi o najmniejszym koszcie przejścia, przechodzącej przez każde miasto dokładnie raz, wracającej do miasta początkowego. Jakość rozwiązania ocenia się za pomocą tzw. funkcji celu, która jest definiowana jako suma kosztów przejścia między kolejnymi miastami na drodze będącej rozwiązaniem problemu wraz z przejściem łączącym ostatnie odwiedzone (unikalne) miasto z miastem startowym. Rozwiązanie, dla którego funkcja ta przyjmuje wartość minimalną nazywa się rozwiązaniem optymalnym.

W zależności od tego, czy przejście z miasta A do miasta B ma taki sam koszt, co przejście z miasta B do miasta A dla dowolnej pary miast (miasta muszą być różne od siebie) mamy do czynienia z symetrycznym problemem komiwojażera, jeżeli koszty te są równe, zaś w przeciwnym przypadku mówi się o asymetrycznym problemie komiwojażera. Tematem projektu jest asymetryczny problem komiwojażera. Warto dodać, iż każdy symetryczny problem komiwojażera można potraktować tak, jak asymetryczny, uprzednio dublując koszty przejścia pomiędzy każdą parą miast, tak aby przejście było możliwe w obie strony.

2 Metody i algorytmy rozwiązywania problemu

Istnieje kilka znanych metod rozwiązywania problemu komiwojażera; są to

- metoda przeglądu zupełnego (Brute Force),
- programowanie dynamiczne (Dynamic Programming),
- metoda podziału i ograniczeń (Branch and Bound).

W ramach tego projektu zaimplementowano algorytmy rozwiązujące problem komiwojażera zgodnie z założeniami powyższych metod. Algorytmy te zostaną omówione w tej sekcji wraz z odpowiadającymi im metodami.

2.1 Przegląd zupełny

Metoda przeglądu zupełnego polega na wygenerowaniu wszystkich możliwych rozwiązań i wybraniu z nich optymalnego - będzie to rozwiązanie „najlepsze” pod względem zastosowanego kryterium. W przypadku TSP rozwiązanie wybiera się spośród permutacji[1] danych miast, uprzednio wyliczając funkcję celu dla każdej z nich. Rozwiązaniem optymalnym będzie ta permutacja wraz z przejściem powrotnym do miasta początkowego, dla której funkcja celu przyjmie wartość najmniejszą. Oczywiście może wystąpić sytuacja, w której znalezionych zostanie więcej niż jedno rozwiązanie; w takim wypadku algorytm wybiera pierwsze znalezione rozwiązanie optymalne.

Esencją algorytmu opierającego się na tej metodzie jest sposób generowania permutacji. Algorytm ten zaimplementowano w dwóch wersjach:

- polegającej na operacjach typu swap,
- polegającej na przeszukiwaniu struktury drzewiastej.

[1] Przez permutację rozumie się tutaj drogę składającą się z kolejnych elementów permutacji

Do implementacji wersji polegającej na operacjach typu swap wykorzystano algorytm Heap'a w wersji pozbawionej rekurencji. Algorytm ten został dokładnie opisany w [3], a w skrócie polega na takiej zamianie dwóch elementów permutacji miejscami, aby po każdej takiej operacji otrzymać nową, jeszcze do tej pory nie otrzymaną permutację.

Wersję algorytmu bazującą na drzewie przeszukiwań zaimplementowano zgodnie z następującym pomysłem: utworzone zostały dwie tablice (tutaj std::vector) przechowujące przetwarzane numery miast. Jedna z nich zawierała budowane w toku działania algorytmu rozwiązanie, zaś druga zawierała jeszcze niewykorzystane miasta. Za pomocą rekurencji do tablicy przechowującej budowaną drogę dodawano kolejne miasta z zachowaniem własności permutacji, aż do zbudowania rozwiązania. Rozwiązanie to było oceniane za pomocą funkcji celu; spośród wszystkich tak zbudowanych rozwiązań wybierano pierwsze o najmniejszym koszcie przejścia. Główną pętlę iterującą po elementach tablicy zawierającej jeszcze niewykorzystane wierzchołki przedstawiono poniżej.

Listing 1: Kluczowa pętla algorytmu przeglądu zupełnego bazującego na przeszukiwaniu struktury drzewiastej

```
1 for (int i = 0; i != availableElements.size(); ++i) {
2     usedElements.emplace_back(availableElements[i]);
3     availableElements.erase(availableElements.begin() + i);
4     bruteForceTreeRecursiveBuild(availableElements, usedElements,
5         bestSolutionValue, tspInstance, solution);
6     availableElements.insert(availableElements.begin() + i,
7         usedElements.back());
8     usedElements.erase(usedElements.end() - 1);
9 }
```

Funkcja *bruteForceTreeRecursiveBuild()* jest wywoływana rekurencyjnie. Zawiera on w swoim ciele przedstawioną powyżej pętlę oraz warunek obsłużenie przypadku powrotu z wywołania rekurencyjnego. Przypadkiem tym jest zbudowanie poprawnego rozwiązania. Wyżej przedstawiony schemat przeszukiwania jest w rzeczywistości przeszukiwaniem w głąb.

Chociaż, jak się okaże w późniejszych sekcjach, algorytmy te różnią się wydajnością, to nie ulega wątpliwości fakt, iż oba należą do algorytmów o ponad-wielomianowej klasie złożoności obliczeniowej. Jako, iż każdy z nich bada wszystkie $(n-1)!$ permutacji ich złożoność obliczeniowa czasowa wynosi co najmniej $O(n!)$.

2.2 Programowanie dynamiczne

Metoda programowania dynamicznego polega na usunięciu potrzeby wykonywania powtarzających się obliczeń, poprzez zapamiętywanie wyników częściowych potrzebnych wielokrotnie podczas rozwiązywania problemu. Podobnie jak w metodzie „dziel i zwyciężaj" metoda ta polega na dzieleniu problemu na podproblemy. W czasie podziału może się okazać, iż podproblem, który otrzymano został już wcześniej rozwiązany. Pamiętając, że każde rozwiązanie podproblemu zachowuje się, wystarczy w takim wypadku odczytać rozwiązanie podproblemu. Warunkiem koniecznym do jej zastosowania jest posiadanie przez problem tzw. optymalnej podstruktury. Oznacza to, że optymalne rozwiązanie problemu musi być funkcją optymalnych rozwiązań podproblemów. W przypadku (A)TSP własność ta jest spełniona dla wyznaczania optymalnych dróg przejścia - jest więc możliwe zastosowanie metody programowania dynamicznego.

Zaimplementowany algorytm działający zgodnie z tą metodą to algorytm Helda-Karpa. Aby opisać działanie tego algorytmu, należy przedstawić kilka faktów o TSP.

Przed przystąpieniem do analizy równań opisujących działanie algorytmu, należy zdefiniować pojęcia związane z tym algorytmem. Niech x oznacza miasto początkowe drogi, zaś t ostatnie unikalne miasto na drodze. Niech S oznacza zbiór miast na drodze od x do t. Niech $dist(i,j)$ oznacza koszt przejścia z miasta i do miasta j. Dla każdego S takiego, że $x \notin S$ i dla każdego $t \in S$, niech $opt(S,t)$ oznacza minimalny koszt przejścia zaczynając w mieście x, idąc przez miasta w S, kończąc na t. Można zauważyć, że zachodzi równanie 1.

$$opt(S,t) = min(opt(S\backslash\{t\},q) + dist(q,t) : q \in S\backslash\{t\}) \tag{1}$$

Jeżeli natomiast chcielibyśmy wyrazić za pomocą równania 1 optymalne rozwiązanie problemu, to również jest to możliwe, jak pokazuje równanie 2, gdzie N oznacza $S\backslash\{x\}$.

$$v^* = min(opt(N,t) + dist(t,x) : t \in N) \tag{2}$$

Po zauważeniu, iż $opt(\{q\},q) = dist(x,q)$ posiadamy pełny mechanizm do budowy algorytmu rekurencyjnego w wersji zstępującej. Tak więc zaimplementowany algorytm polega na znalezieniu v^*, co wykonuje poprzez rekurencyjne podziały problemu na podproblemu, a następnie rozwiązuje te podproblemy zgodnie z równaniem 1. Im więcej obliczeń wykona algorytm, tym więcej obliczonych wartości *opt* stanie się dostępnych do natychmiastowego odczytu, co będzie skutkowało szybszym otrzymywaniem wyników.

Algorytm ten nie zapisuje znalezionych rozwiązań. Zamiast tego wykonuje drugi przebieg zgodnie z równaniem 1 „w drugą stronę". Mając rozwiązanie v^* „zdejmuje" kolejne miasta końcowe z drogi i sprawdza, które miasto spełnia równanie 1. Procedura odzyskania rozwiązania optymalnego zachodzi, jak następuje. Pierwszym miastem, które będzie zdjęte z drogi będzie miasto x. Następnie algorytm sprawdzi, które minimum spośród tych po prawej stronie równania 2 spełnia warunki optymalnego rozwiązania problemu. Po odpowiedzi na to pytanie otrzyma miasto t, które przetworzy w podobny sposób za pomocą równania 1, jednak ze zaktualizowaną wartością rozwiązania optymalnego. Działanie to będzie powtarzane aż do otrzymania pełnego rozwiązania. Aby operacja ta była możliwa konieczne jest uporządkowanie optymalnych rozwiązań podproblemów. Można to zapewnić wybierając zawsze pierwsze znalezione optymalne rozwiązanie podproblemu.

W algorytmie tym ważne jest odpowiednie dobranie reprezentacji zbioru S. Wybrano reprezentację wektora bitów, gdzie waga bitu oznacza indeks miasta, zaś wartości „1" przynależność miasta do zbioru S („0" oznacza brak przynależności). Rozwiązanie to pozwala na natychmiastowy dostęp do każdego elementu zbioru, za pomocą operacji binarnych.

Złożoność czasowa tego algorytmu wynosi $O(n^2 2^n)$, jednak nie mniej ważna jest tutaj złożoność pamięciowa, która z racji przechowywania rozwiązań dla wszystkich podproblemów w trakcie działania algorytmu wynosi $O(n2^n)$. To właśnie wymagania pamięciowe sprawiają, że algorytm ten bardzo szybko powoduje niedomiar dostępnej pamięci operacyjnej.

2.3 Metoda podziału i ograniczeń

Metoda podziału i ograniczeń polega na ograniczaniu przestrzeni wszystkich możliwych rozwiązań. Ograniczanie to musi zostać przeprowadzone tak, aby rozwiązanie optymalne na pewno znajdowało się w ograniczonej przestrzeni rozwiązań. W metodzie tej wykorzystuje się dwa ograniczenia: górne i dolne. Dla problemu minimalizacji, górnym ograniczeniem jest znalezione w trakcie wykonywania algorytmu rozwiązanie. Jest tak, ponieważ rozwiązania o wyższym koszcie, niż koszt dotychczas znalezionego rozwiązania nie są interesujące. Dolne ograniczenie opisuje minimalny koszt, jaki można uzyskać rozwijając przestrzeń rozwiązań z danego jej punktu. Sposobów wyznaczania dolnego ograniczenia może być wiele, jednak każdy z nich musi zapewniać, iż rozwiązanie optymalne nie zostanie pominięte. Metodę tą można modyfikować ustalając różne strategie rozgałęziania (rozwijania przestrzeni) oraz różne strategie wyboru kolejnego punktu, z którego nastąpi rozwinięcie; w tym przypadku przykładowym kryterium może być najmniejsze dolne ograniczenie.

Zaimplementowany algorytm działający zgodnie z tą metodą to algorytm Little'a. W algorytmie tym ograniczenie dolne oblicza się jako minimalny koszt przejścia, jaki jesteśmy w stanie uzyskać mając na uwadze, iż musimy przejść przez każde miasto dokładnie raz. Wartość tą uzyskuje się sumując najmniejsze koszty przejścia do miast jeszcze nieodwiedzonych. Jeżeli koszty te nie wyznaczają pełnej „ścieżki" dodatkowo do wyliczonego ograniczenia dodaje się najmniejsze koszty dojść do miast, dla których nie ma połączenia na „ścieżce". Dolne ograniczenie posłuży jako kryterium wyboru punktu przestrzeni do dalszego jej rozwijania.

Strategia rozgałęziania algorytmu polega na binarnym wyborze: dołączeniu miasta do ścieżki lub jego pominięciu. Rozgałęzienie w przestrzeni polega na symulacji skutków obu tych decyzji oraz przeliczenia dolnego ograniczenia dla niepełnych ścieżek zaktualizowanych o symulowane decyzje. Do wyboru miasta, które jest rozważane podczas rozgałęziania stosuje się metodę polegająca na wybraniu takiego jeszcze nie odwiedzonego miasta, dla którego kara (wzrost dolnego ograniczenia) byłaby największa.

Główną pętlę algorytmu zamieszczono poniżej.

Listing 2: Kluczowa pętla algorytmu Little'a

```
1  BBNodeData leftNode, rightNode;
2  int calculatedUpperBound;
3  while (bbNodes.top().lowerBound < upperBound && !bbNodes.empty()) {
4      if (!bbNodes.top().isFinal) {
5          leftNode = bbNodes.top();
6          rightNode = bbNodes.top();
7          bbNodes.pop();
8
9          bbUpdateLeftNodeData(leftNode);
10         bbCalculateLowerBoundAndDesignateHighestZeroPenalties(leftNode);
11         if (leftNode.lowerBound < upperBound) {
12             bbNodes.push(leftNode);
13         }
14
15         bbUpdateRightNodeData(rightNode);
16         bbCalculateLowerBoundAndDesignateHighestZeroPenalties(rightNode);
17         if (rightNode.lowerBound < upperBound) {
18             bbNodes.push(rightNode);
19         }
20     } else {
21         calculatedUpperBound =
22         TSPUtils::calculateTargetFunctionValue(tspInstance,
23             bbNodes.top().partialPaths.front());
24         if (calculatedUpperBound < upperBound) {
25             upperBound = calculatedUpperBound;
26             tspSolution = bbNodes.top().partialPaths.front();
27         }
28         bbNodes.pop();
29     }
30 }
```

Algorytm poszerzono o dodatkowe strategie szybszego uzyskiwania górnego ograniczenia. Pierwsza z nich polegała na znalezieniu rozwiązania za pomocą algorytmu najbliższego sąsiada. Druga z nich również była strategią zachłanną (tak jak NN), jednak rozwiązanie wyznaczała za pomocą następującego algorytmu:

1. Utwórz listę przejść między miastami ATSP.
2. Posortuj utworzoną listę niemalejąco wg. kosztów przejść.
3. Iterując po kolejnych elementach listy dodawaj kolejne krawędzie do rozwiązania, o ile powstała ścieżka spełnia wymagania TSP.

Wspomnianymi wymaganiami TSP jest brak cykli w ścieżce (poza powrotem do miasta początkowego) oraz odwiedzenie każdego miasta dokładnie raz.

Na zakończenie należy zaznaczyć, iż w pesymistycznym przypadku złożoność czasowa tego algorytmu może degenerować się do $O(n!)$ (przegląd zupełny).

3 Eksperymenty obliczeniowe

W ramach projektu przeprowadzono eksperymenty obliczeniowe, które polegały na pomiarze czasu wykonania zaimplementowanego algorytmu w zależności od wielkości instancji problemu. Czas ten był mierzony w milisekundach, zaś do samego pomiaru wykorzystano pakiet z biblioteki standardowej C++ - *std::chrono*. Instancje były podawane z pliku - były to wybrane instancje testowe dostarczone razem z zadaniem projektowym. Komputer, na którym przeprowadzono testy posiadał procesor Intel Core i7-6700HQ oraz był wyposażony w 16 GB RAMu. Ustalono limit czasowy na czas trwania jednego wykonania algorytmu dla ustalonej instancji, który wyniósł 3 minuty. Po tym czasie test przerywano, co jest oznaczone w tabeli 1 symbolem *.

Przeprowadzono testy dla obu wersji algorytmu polegającego na przeglądzie zupełnym, algorytmu Helda-Karpa oraz czterech wersji algorytmu Little'a. Wersje te różnią się sposobem obliczania ograniczenia górnego.

Legenda:

- BF - algorytm przeglądu zupełnego
  - (swap) - wersja z iteracyjnym generowaniem permutacji algorytmem Heap'a
  - (DFS) - wersja z przeglądem struktury drzewiastej za pomocą przeszukiwania w głąb (algorytm rekurencyjny) - operacje na dwóch tablicach
- DP (Held-Karp) - algorytm programowania dynamicznego Helda-Karpa
- B&B - algorytm podziału i ograniczeń Little'a
  - (Little) - bez wstępnego obliczania ograniczenia górnego
  - (Little; NN) - wstępne obliczanie ograniczenia górnego za pomocą algorytmu najbliższego sąsiada
  - (Little; G) - wstępne obliczanie ograniczenia górnego za pomocą algorytmu zachłannego opisanego w punkcie 2.3
  - (Little; NN; G) - wstępne obliczanie ograniczenia górnego za pomocą obu algorytmów zachłannych (wybierane jest najlepsze rozwiązanie)

3.1 Wyniki pomiarów czasu

Wszystkie dane pomiarowe uzyskane podczas eksperymentów obliczeniowych zostały umieszczone w tabeli 1. Warto zwrócić uwagę na niewielką liczbę instancji rozwiązanych przez algorytmy polegające na przeglądzie zupełnym; wyraźnie widać, iż algorytmy te są wysoce niepraktyczne. W przypadku algorytmu DP poza granicą 24 wierzchołków narzuconą przez ograniczenie czasowe uzyskano granicę 29 wierzchołków dla posiadanego sprzętu; dla większych instancji nie spełniono wymagań odnośnie zapotrzebowania na pamięć. Nie należy też pominąć ciekawej sytuacji, która występuje w algorytmie podziału i ograniczeń; chociaż nietrudno zauważyć rosnący trend czasowy względem wielkości instancji, co wydaje się być następstwem konieczności przetwarzania coraz większej liczby miast przy tych samych operacjach, to dodatkowo dane pomiarowe uwidaczniają niespodziewane spadki wydajności. Na tej podstawie można wnioskować, iż zachowanie algorytmu B&B zależy również od samej instancji, a nie tylko od jej rozmiaru.

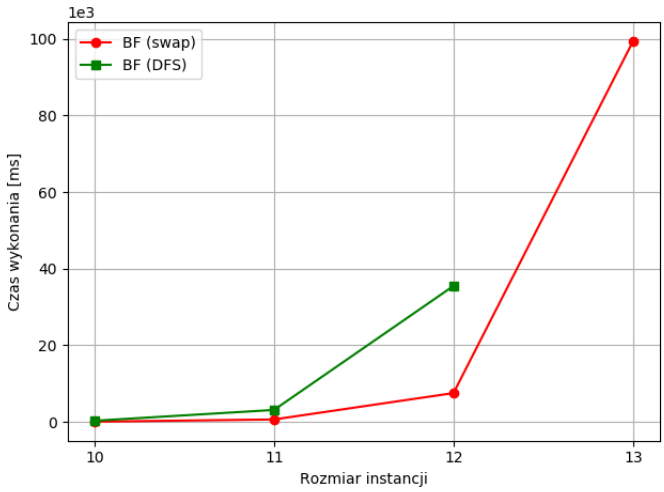
N	BF (swap)	BF (DFS)	DP (Held-Karp)	B&B (Little)	B&B (Little; NN)	B&B (Little; G)	B&B (Little; NN; G)
10	65	309	11	1125	1131	1164	1167
11	651	3133	2	77	77	79	79
12	7535	35493	2	7	6	6	6
13	99340	*	6	2	2	1	1
14	*	*	14	1	1	0	1
15	*	*	31	2	2	2	2
16	*	*	73	8	8	8	8
17	*	*	163	103529	106968	110328	113655
18	*	*	445	6	7	7	7
21	*	*	7703	15	16	16	18
24	*	*	93798	403	402	407	403
26	*	*	*	1038	1040	1033	1048
29	*	*	*	1969	1997	2016	2016
34	*	*	*	8488	8723	8943	9173
36	*	*	*	1686	1700	1720	1752
39	*	*	*	4588	4498	4549	4581
42	*	*	*	*	*	*	*
43	*	*	*	*	*	*	*
45	*	*	*	16868	16951	17451	17484

Tablica 1: Zestawienie czasów wykonania zaimplementowanych algorytmów dla ATSP względem rozmiaru instancji zaokrąglonych matematycznie do części całkowitej

3.2 Analiza zaimplementowanych algorytmów

W sekcji tej dane z tabeli 1 zostaną przedstawione w formie wykresów, a następnie pokrótce omówione.

3.2.1 Przegląd zupełny



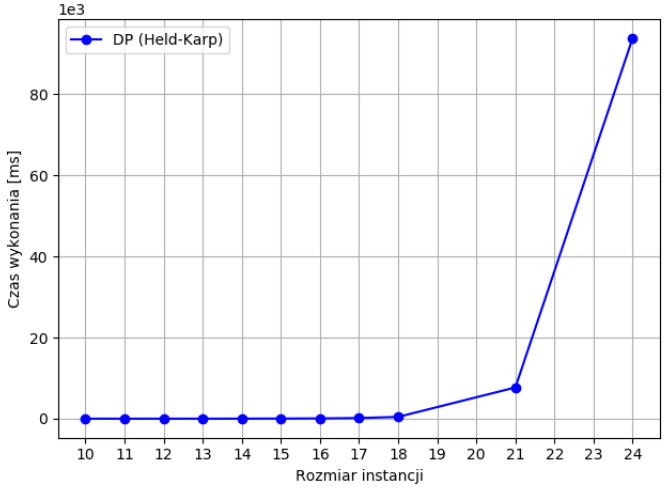
Rysunek 1: Porównanie algorytmów bazujących na metodzie przeglądu zupełnego

Chociaż testy dla algorytmów BF przeprowadzone zostały dla małej liczby instancji, to łatwo jest zauważyć, że algorytm bazujący na operacjach typu swap uzyskuje lepsze rezultaty czasowe, niż algorytm bazujący na przeszukiwaniu struktury drzewiastej. Można wyróżnić dwie przyczyny takiego stanu rzeczy:

1. algorytmy rekurencyjne są zazwyczaj mniej wydajne od swoich odpowiedników iteracyjnych z uwagi na konieczność wykonania dużej liczby operacji na stosie, co jest działaniem wolnym,

2. nową permutację w algorytmie Heap'a uzyskuje się z każdym wykonaniem operacji swap, zaś algorytm DFS wykonuje wiele operacji w sytuacji, gdy z głębokiego poziomu przeszukiwania należy przemieścić się do innej, potencjalnie niepowiązanej wspólnymi elementami, gałęzi drzewa.

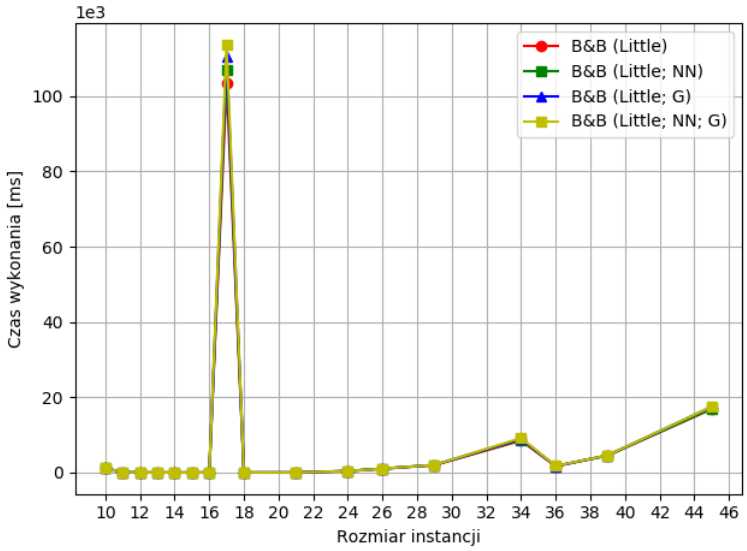
3.2.2 Programowanie dynamiczne



Rysunek 2: Zachowanie algorytmu Helda-Karpa dla rosnącego rozmiaru instancji

Algorytm Helda-Karpa nie zaskakuje ani uzyskanymi czasami, ani wykazanym trendem. Wyniki są zgodne z założeniami teoretycznymi (oszacowanie z góry dla czasu równe $O(n^2 2^n)$). Należy jednak zaznaczyć, iż jest to algorytm o lepszej złożoności czasowej, niż w przypadku przeglądu zupełnego, jednak dla dużych instancji algorytm ten nie ma praktycznego zastosowania, gdyż w dalszym ciąg jest to złożoność czasowa ponad-wielomianowa.

3.2.3 Algorytm podziału i ograniczeń

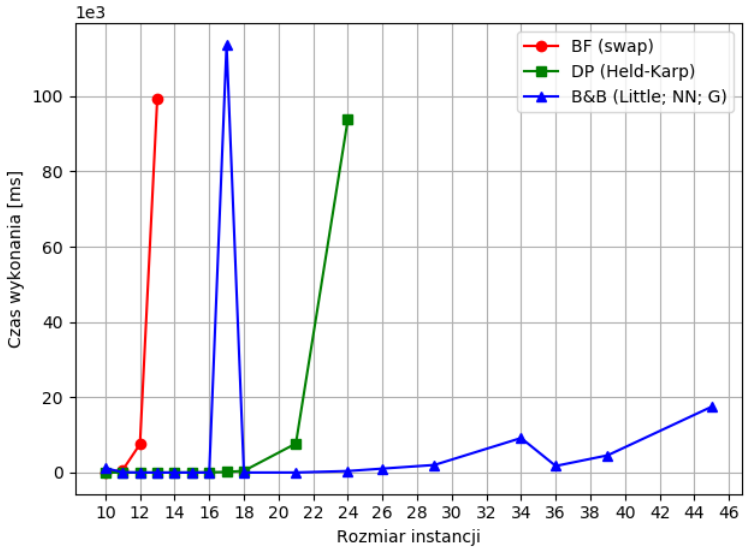


Rysunek 3: Porównanie różnych wersji algorytmu bazującego na metodzie podziału i ograniczeń

Jak się okazuje zastosowanie różnych strategii pozwalających na szybsze uzyskanie oszacowania górnego ma niewielki wpływ na wydajność algorytmu w przypadku przetestowanych instancji. Nie udało się zatem wykazać, iż szybsze wygenerowanie górnego ograniczenia pozwoli na szybsze uzyskanie rozwiązania.

Warto zwrócić uwagę na instancję o rozmiarze 17. Poza omówioną już własnością tego algorytmu do bycia zależnym nie tylko od rozmiaru instancji, ale również od samych danych, widać jeszcze, iż zastosowanie dużej liczby algorytmów wstępnie ograniczających przestrzeń rozwiązań z góry może przynieść efekt odwrotny do zamierzanego; z uwagi na ich wielomianowy czas wykonywania, udział w obliczeniach tych algorytmów uwidacznia się w całkowitym rozrachunku czasowym.

3.2.4 Zestawienie algorytmów



Rysunek 4: Porównanie różnych metod rozwiązywania TSP

Z wykresu wynika, iż zarówno metoda przeglądu zupełnego, jak i metoda programowania dynamicznego w przypadku TSP nie sprawdza się w praktyce. Jest to konsekwencją ich ponad-wielomianowych ograniczeń czasowych z góry. Kuszącym jest, aby wysunąć wniosek o wyższości algorytmu Little'a nad pozostałymi. Chociaż niewątpliwie algorytm ten uzyskuje lepsze rezultaty dla przebadanych instancji, to należy zwrócić uwagę na fakt, iż wykres nie obrazuje przerwania testu dla instancji o rozmiarze 42 oraz 43 ze względu na przekroczony limit czasowy. Biorąc to pod uwagę oraz pamiętając o omówionych wcześniej nieprzewidywalnych spadkach wydajności nawet dla małych rozmiarów instancji można stwierdzić, że również ten algorytm nie będzie miał praktycznego zastosowania. Z racji jego zależności od danych można trafić na instancję, której nie uda się rozwiązać w rozsądnym czasie, gdyż algorytm ten może zdegenerować się do przeglądu zupełnego.

4 Wnioski

W ostatecznym rozrachunku żaden z algorytmów dokładnych nie okazał być się na tyle wydajny, aby mógł mieć praktyczne zastosowanie. Algorytmy przeglądu zupełnego oraz polegające na metodzie programowania dynamicznego w przypadku TSP posiadają złożoność ponad-wielomianową, zaś algorytm bazujący na metodzie podziału i ograniczeń może degenerować się do przeglądu zupełnego. Taki stan rzeczy oznacza osiąganie w praktyce zbyt długich czasów oczekiwania na rozwiązanie dla podanej instancji.

Rozsądnym wydaje się być porzucenie dokładności na rzecz rozwiązań przybliżonych do optymalnego, znajdowanych w osiągalnym czasie, gdyż rzeczywiste instancje problemu komiwojażera mają wielkość rzędu tysięcy miast. Algorytmy te będą tematem kolejnego etapu projektu.

Literatura

[1] S. D. D. S. J. David L. Applegate, William J. Cook. *A Practical Guide to Discrete Optimization*. 2014.

[2] F. N. Gözde Kizilateş. On the nearest neighbor algorithms for the traveling salesman problem. pages 1–4, 2013.

[3] R. Sedgewick. Permutation generation methods. 9(2):1–7, 1977.