Projektowanie Efektywnych Algorytmów

Projekt

08.11.2023

263900 Aleksandra Chrustek

(2) Branch and bound

spis treści	*strona*
Sformułowanie zadania	2
Metoda	3
Algorytm	4
Dane testowe	5
Procedura badawcza	6
Wyniki	7

1 Sformułowanie zadania

Zadanie polega na opracowaniu, napisaniu i zbadaniu problemu komiwojażera w wersji optymalizacyjnej algorytmem Branch & Bound. Problem komiwojażera (eng. Travelling salesman problem, TSP) to zagadnienie polegające (w wersji optymalizacyjnej) na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym.

1. Graf pełny to zbiór wierzchołków, przy czym między każdymi dwoma wierchołkami istnieje krawędź je łącząca. [1]
2. Cykl Hamiltona to droga wiodąca przez wszystkie wierzchołki dokładnie raz, z wyjątkiem jednego wybranego, w którym cykl Hamiltona zaczyna się oraz kończy. [2]

Problem komiwojażera rozumiemy jako zadanie polegające na znalezienu najlepszej drogi dla podróżującego chcącego odwiedzić n miast i skończyć podróż w miejscu jej rozpoczęcia. Połączenie między każdym miastem ma swój koszt określający efektywnosć jej przebycia. Najlepsza droga to taka, której całkowity koszt (suma kosztów przebycia wszystkich połączeń między miastami na drodze) jest najmiejszy. Problem dzieli się na symetryczny i asymetryczny. Pierwszy polega na tym, że dla dowolnych miast A i B z danej instancji, koszt połączenia jest taki sam w przypadku przebycia połączenia z A do B, jak z B do A, czyli dane połączenie ma jeden koszt niezależnie od kierunku ruchu. W asymetrycznym problemie komiwojażera koszty te mogą być różne.

2 Metoda

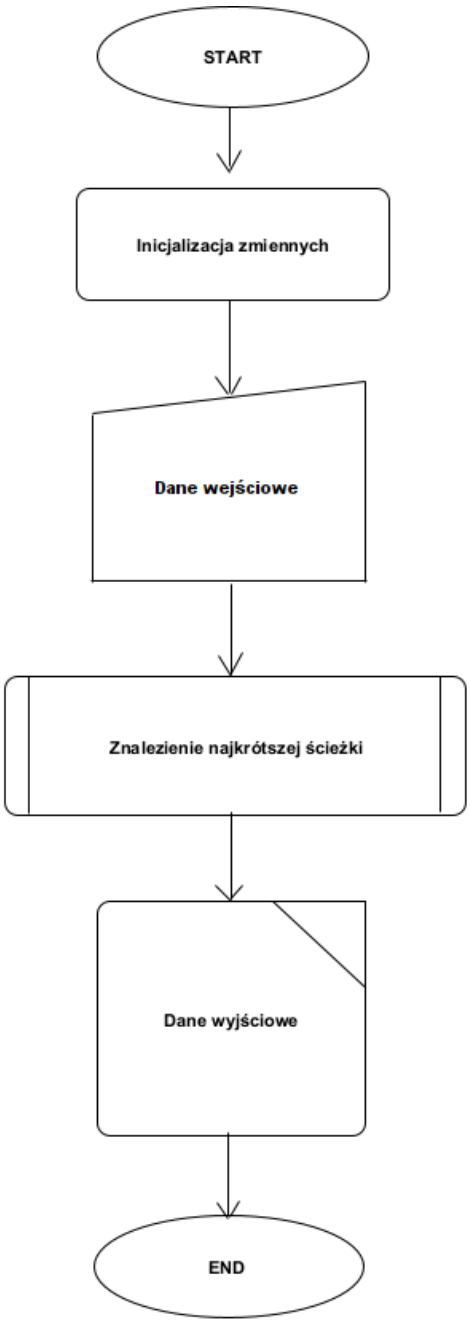
Metoda podziału i ograniczeń (eng. Branch & Bound) polega na analizie drzewa rozwiązań obrazującego każdy możliwy sposób otrzymania rozwiązania końcowego. Przejście i wygenerowanie całego drzewa jest bardzo kosztowne, więc ogranicza się je w wybrany sposób. Istnieją 3 strategie przeszukiwania drzewa:

1. Min/Max Cost – Best Search
2. Depth search – przeszukiwanie drzewa w głąb
3. Breadth search – przeszukiwanie drzewa w szerz

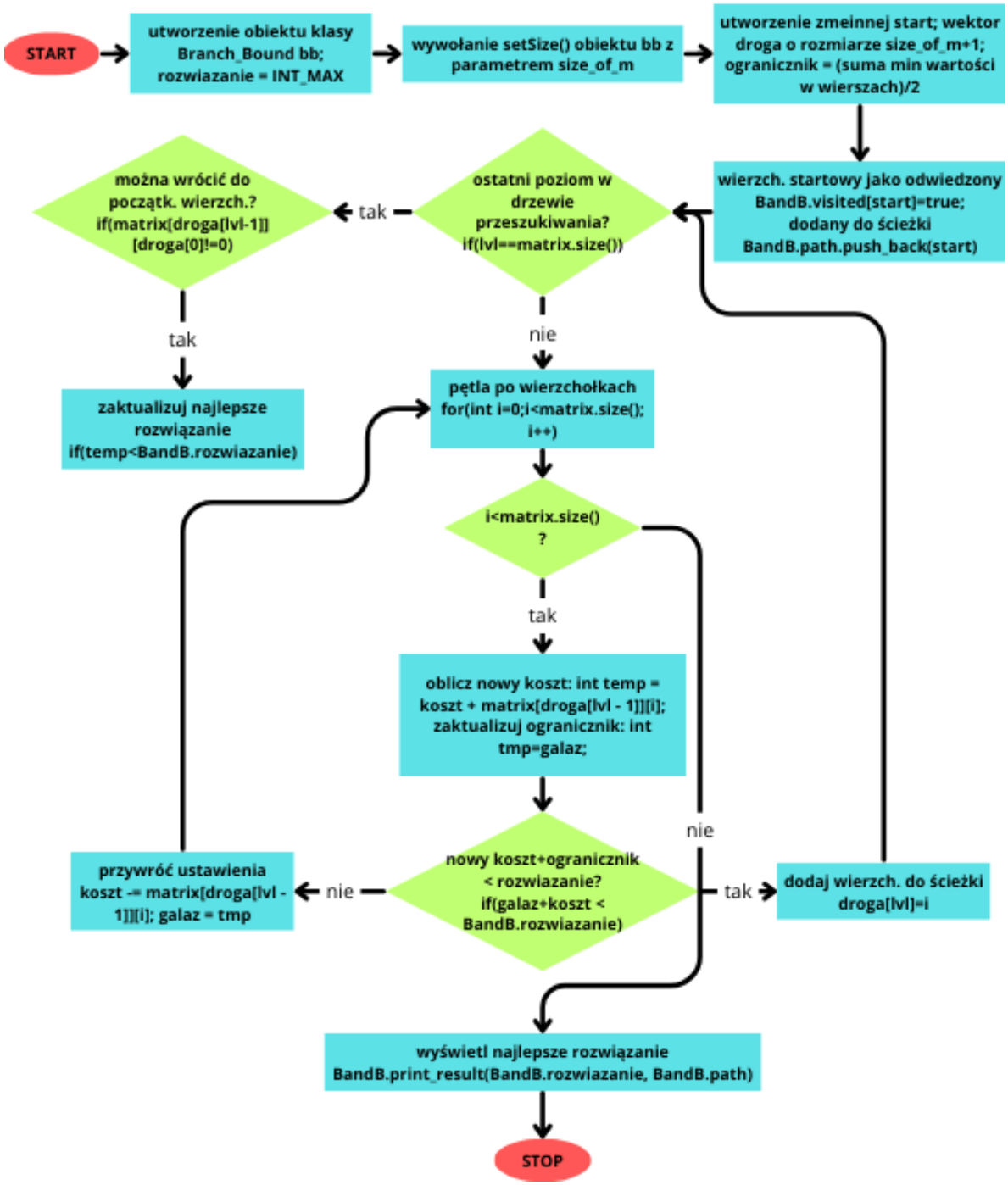
W projekcie zdecydowałam się na wybór wariantu Best Search, gdyż Depth search i Breadth search są nieoptymalne ze względu na bardzo dużą złożoność pamięciową około $O((n-1)! \cdot n^2)$. "Best search"w kontekście algorytmu Branch and Bound polega na wyborze wierzchołka, który ma najmniejszy koszt przejścia do innych wierzchołków. Wybór ten ma na celu ograniczenie przeszukiwania do najbardziej obiecujących gałęzi drzewa rozwiązań. Dzięki "best search", algorytm skupia się na badaniu tych gałęzi, które z największym prawdopodobieństwem zawierają optymalne rozwiązanie, co przyczynia się do efektywności i szybkości działania algorytmu. W mojej implementacji, metoda *find_min* jest wykorzystywana w funkcji *CheckLevel* do identyfikacji najbardziej obiecujących gałęzi do dalszego rozpatrzenia, co pomaga w ograniczaniu przeszukiwania przestrzeni rozwiązań.

3 Algorytm

Rozwiązanie zaimplementowane w programie obrazują następujące schematy.



Rysunek 1: Schemat blokowy programu



Rysunek 2: Schemat blokowy algorytmu opartego na metodzie Branch and Bound

Opis algorytmu:

1. Utworzenie obiektu klasy *Branch_ Bound* o nazwie *bb* - jest to obiekt, który będzie odpowiedzialny za wykonywanie algorytmu Branch and Bound. Ustawienie zmiennej *rozwiazanie* na INT_MAX - jest to zmienna przechowująca najlepsze znalezione rozwiązanie. Skorzystanie z metody *setSize* obiektu *bb*, aby ustawić rozmiar ścieżki i wektora odwiedzonych wierzchołków. Rozmiar ten jest równy *size_ of_ m*. Utworzenie zmiennej *start* - będzie to indeks wierzchołka początkowego, od którego rozpocznie się algorytm. Utworzenie wektora *droga* o rozmiarze *size_ of_ m + 1* - ten wektor będzie przechowywał aktualną ścieżkę w grafie. Utworzenie zmiennej *ogranicznik* i obliczenie jej jako sumę minimalnych wartości w wierszach macierzy kosztów, podzieloną przez 2 (z zaokrągleniem w górę, jeśli wynik jest nieparzysty) - ten parametr ma na celu pomóc w ograniczaniu rozgałęzień w algorytmie.

2. Oznaczenie startowego wierzchołka jako odwiedzony poprzez BandB.visited [start] = true. Oznacza to, że wierzchołek został już odwiedzony na początku algorytmu. Dodanie startowego wierzchołka do ścieżki poprzez BandB.path.push_back(start). Dodajemy go na początek ścieżki.

3. Funkcja *CheckLevel*: W tej funkcji sprawdzamy, czy osiągnęliśmy ostatni poziom w drzewie przeszukiwania. Jeśli tak, oznacza to, że przeszliśmy przez wszystkie wierzchołki w grafie i musimy zakończyć analizę ścieżki. W tym przypadku sprawdzamy, czy można wrócić do początkowego wierzchołka, co jest ważne dla problemu komiwojażera. Jeśli tak, to oznacza, że znaleźliśmy cykl, który zaczyna się i kończy w początkowym wierzchołku. Obliczamy koszt tego cyklu i porównujemy go z najlepszym znanym rozwiązaniem *(BandB.rozwiazanie)*. Jeśli obecne rozwiązanie jest lepsze, następuje zaktualizowanie *BandB.path* i *BandB.rozwiazanie*. Jeśli nie można wrócić do początkowego wierzchołka, następuje powrót, ponieważ nie można zakończyć cyklu.

4. Pętla po wierzchołkach: Przechodzimy pętlą przez wszystkie wierzchołki w grafie, które możemy odwiedzić. Dla każdego wierzchołka wykonujemy następujące kroki:
   -Oblicz nowy koszt, dodając koszt przejścia z ostatniego odwiedzonego wierzchołka do tego wierzchołka *int temp = koszt + matrix[droga[lvl - 1]][i]*.
   -Zaktualizuj ogranicznik *int tmp = galaz*. Jest to ważne, ponieważ ogranicza on dalsze rozgałęzianie.
   -Sprawdź, czy nowy koszt + ogranicznik jest mniejszy niż obecne najlepsze rozwiązanie *if (galaz + koszt < BandB.rozwiazanie)*: Jeśli tak, dodaj wierzchołek do ścieżki *droga[lvl] = i* i rekurencyjnie wywołaj *CheckLevel* dla kolejnego poziomu.
   -Przywróć poprzednie ustawienia, aby móc rozważyć inne możliwe ścieżki *koszt-=matrix[droga[lvl - 1]][i]; galaz = tmp;*.

4 Dane testowe

Do sprawdzenia poprawności działania algorytmu i wykonania badań wybrano następujący zestaw instancji:
tsp_6_1.txt 132
tsp_6_2.txt 80
tsp_10.txt 212
tsp_12.txt 264
tsp_13.txt 269
tsp_14.txt 282
tsp_15.txt 291
tsp_17.txt
tsp_17_1.txt 137
tsp_18_1.txt 149
tsp_19_1.txt 169
tsp_20_1.txt 148
dostępnych na stronie: http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/

5 Procedura badawcza

Należało zbadać zależność czasu rozwiązania problemu od wielkości instancji. W przypadku algorytmu opartego na metodzie Branch and Bound nie występowały parametry programu, które mogły mieć wpływ na czas i jakość uzyskanego wyniku. W związku z tym procedura badawcza polegała na uruchomieniu programu sterowanego plikiem inicjującym *.ini* (format pliku: *nazwa_instancji liczba_wykonań nazwa_pliku_wyjściowego*).
tsp_6_1.txt 10
tsp_6_2.txt 10
tsp_10.txt 10
tsp_12.txt 10
tsp_13.txt 10
tsp_14.txt 10
tsp_15.txt 3
tsp_17.txt 1
tsp_17_1.txt 3
tsp_18_1.txt 3
tsp_19_1.txt 3
tsp_20_1.txt 3
tsp_test.csv

Każda z instancji rozwiązywana była zgodnie z liczbą jej wykonań, np. tsp_12.txt wykonana została 10 razy. Do pliku wyjściowego tsp_test.csv zapisywane były informacje o instancji: jej nazwa, liczba wykonań algorytmu. Nastepnie zapisywane były czasy wykonań algorytmu dla tej instancji. Plik wyjściowy zapisywany był w formacie csv. Poniżej przedstawiono zawartość pliku wyjściowego. Kolorem czerwonym zaznaczone zostały wyniki odbiegające od normy - anomalie - nie były one uwzględniane w obliczaniu średniej. Test dla instancji tsp_17.txt został przerwany ze względu na bardzo długi czas trwania - niemal 4 godziny.

tsp_6_1.txt 10
1658.3
1197.3
715.5
753.1
711
726
733.2
696.1
695
699.7
tsp_6_2.txt 10
1064.4
1032.6

1060.1
1034.7
1034.6
1069.2
1029.6
1028.2
1007.5
1006.8
tsp_10.txt 10
17534.9
17550.1
17269.3
17795
17964.6
17801.4
17814.5
17532.4
18197
17813.8
tsp_12.txt 10
89274.1
88937.7
87330.2
88061.7
88065.2
86993.9
87248
98494.6
87536.4
87746
tsp_13.txt 10
244337
238997
241420
236934
234243
237464
237456
246837
242446
237940
tsp_14.txt 10
481547
462066

459633
460958
461395
461405
466054
462798
462770
468995
tsp_15.txt 3
614974
606168
609036
tsp_17.txt 1
test przerwany ze względu na czas trwania
tsp_17_1.txt 3
5269620
5159610
5200540
tsp_18_1.txt 3
6756580
6665620
6610120
tsp_19_1.txt 3
25223700
24584300
24679900
tsp_20_1.txt 3
6421490
6392490
6387650
tsp_test.csv

Pomiary zostały wykonane na platformie sprzętowej:
procesor: Intel® Core™ i5-01400F CPU @ 2.90GHz × 6
pamięć operacyjna: 16 GiB
system operacyjny: Windows 11 64-bit

W programie czas jest mierzony za pomocą funkcji *read_ QPC()*, która wykorzystuje API systemu Windows do odczytywania licznika wydajności procesora (QPC - QueryPerformanceCounter).

```
long long int read_QPC()
{
    LARGE_INTEGER count;
    QueryPerformanceCounter(&count);
```

```
    return((long long int)count.QuadPart);
}
```

Opis działania:

1. *QueryPerformanceCounter(&count)* pobiera aktualną wartość licznika wydajności procesora i zapisuje ją w strukturze *count*.
2. *count.QuadPart* zawiera odczytaną wartość, która jest zwracana jako long long int.
3. W funkcji *choose_ option()* jest wykorzystywana funkcja *QueryPerformanceFrequency*, która zwraca liczbę tików licznika wydajności na sekundę. Jest to wykorzystywane do przekształcenia różnicy czasu na mikrosekundy:

```
long long int frequency, start, elapsed;
QueryPerformanceFrequency((LARGE_INTEGER *)&frequency);
```

Cały proces mierzenia czasu wygląda więc następująco:

1. Pobierana jest częstotliwość licznika wydajności za pomocą *QueryPerformance-Frequency* i zapisywana w zmiennej *frequency*.
2. Przed wykonaniem operacji, której czas chcemy zbadać, wykonujemy *start = read_ QPC()* aby zapisać aktualną wartość licznika wydajności.
3. Po wykonaniu operacji, ponownie wywołujemy *read_ QPC()* i odejmujemy *start* od odczytanej wartości, aby uzyskać czas, jaki upłynął.
4. Następnie przekształcamy różnicę czasu na mikrosekundy - używając (1000000.0 * elapsed) / frequency).

Taki sposób mierzenia czasu opiera się na tym, że licznik wydajności procesora działa z bardzo wysoką precyzją, co pozwala na dokładne mierzenie czasu wykonania operacji.

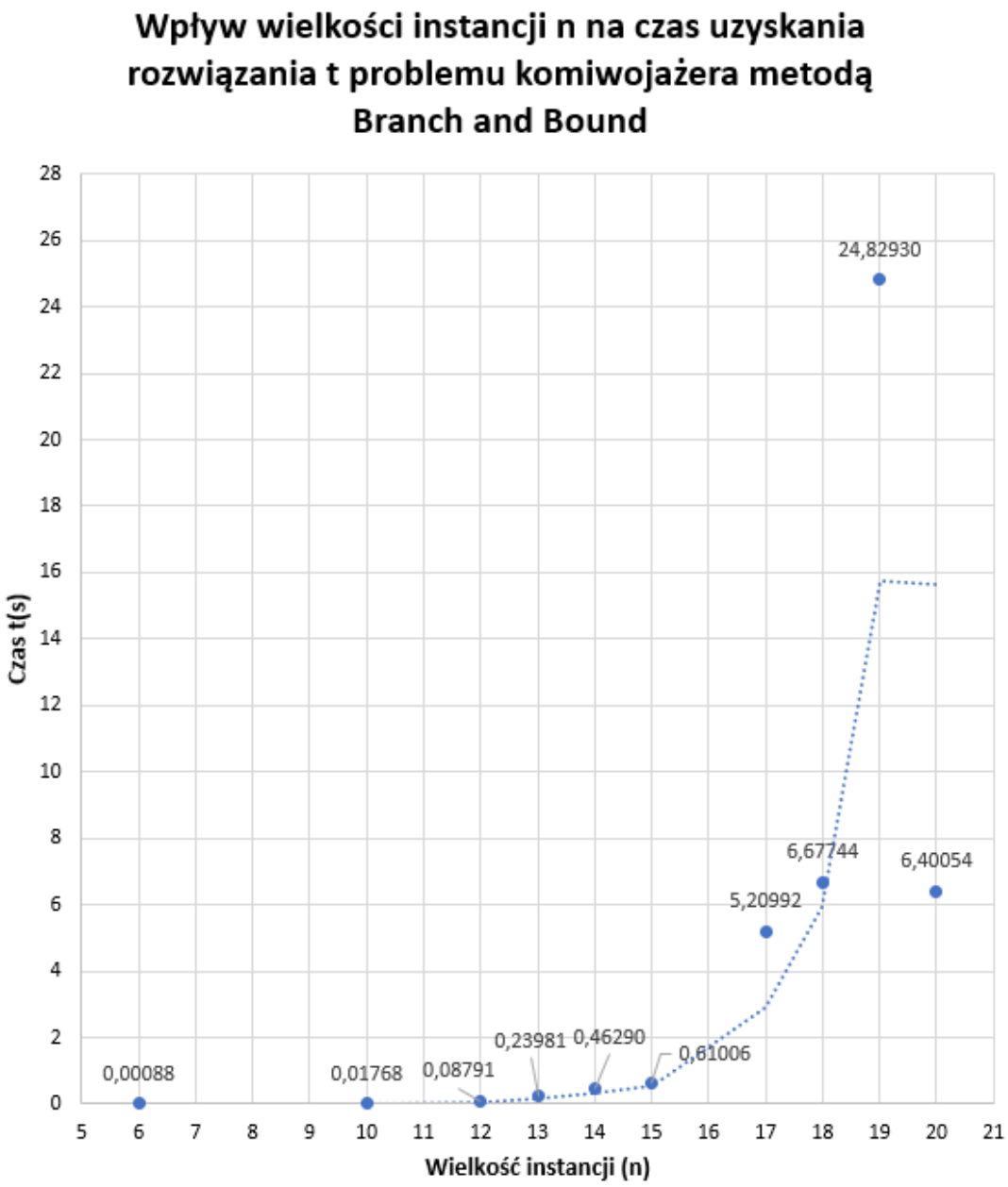
Po każdym powtórzeniu wykonania algorytmu dla danej instancji, program informuje o znalezionej ścieżce oraz jej koszcie. Wyniki zostały opracowane w programie Microsoft Excel.

6 Wyniki

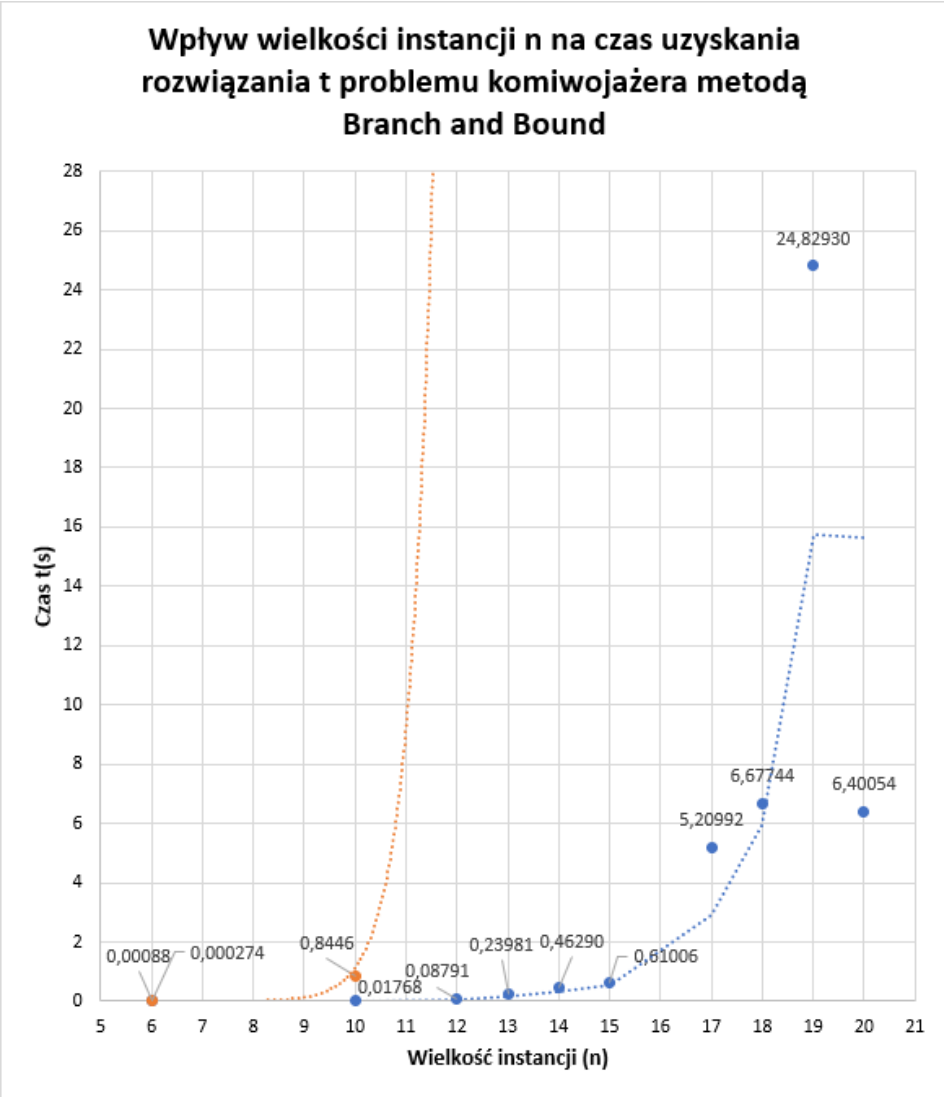
Wyniki zgromadzone zostały w pliku: tsp_test.csv

Tabela 1: Tabela wyników

wielkość instancji (n)	*czas t(s)*
6	0,00088
10	0,01768
12	0,08791
13	0,23981
14	0,46290
15	0,61006
17	5,20992
18	6,67744
19	24,82930
20	6,40054



Rysunek 3: Wpływ wielkości instancji n na czas uzyskania rozwiązania problemu komiwojażera metodą Branch and Bound



Rysunek 4: Wpływ wielkości instancji n na czas uzyskania rozwiązania problemu komiwojażera metodą Brute force oraz Branch and Bound

7 Analiza wyników i wnioski

Badany czas rozwiązania problemu TSP rośnie wraz z wielkościa instancji. Dla instancji do 19 wierzchołków jest to wzrost niewielki, natomiast warto zauważyć, że dla instancji o 19 wierzchołkach, czas rozwiązania problemu TSP rośnie wykładniczo, co sugeruje, że problem staje się znacząco bardziej skomplikowany wraz ze wzrostem liczby wierzchołków. To może wskazywać na potrzebę zastosowania zaawansowanych technik optymalizacyjnych lub algorytmów aproksymacyjnych w przypadku bardzo dużych instancji.
Co więcej, dla instancji o 20 wierzchołkach, obserwujemy interesujący spadek w czasie rozwiązania w porównaniu do instancji z 19 wierzchołkami. Może to sugerować, że istnieje pewna szczególna właściwość dla tych instancji, która sprzyja bardziej efektywnemu rozwiązaniu problemu TSP.
Warto również zaznaczyć, że algorytm Branch and Bound, szczególnie w wersji Best cost, wykazuje znakomitą efektywność dla mniejszych instancji (do 19 wierzchołków) - w przypadku Brute force było to 13 wierzchołków. Jednakże, dla większych instancji, należy wziąć pod uwagę rosnący czas wykonania. W takich przypadkach, możliwe jest, że inne metody lub podejścia mogą być bardziej optymalne.
Podsumowując, analiza porównawcza algorytmów Branch and Bound oraz Brute force pozwala na wyraźne wskazanie zalet pierwszego, zwłaszcza dla instancji o dość dużym rozmiarze (do 19 wierzchołków). Jednakże, wraz z rosnącą liczbą wierzchołków, konieczne może być rozważenie alternatywnych strategii rozwiązania problemu TSP.

Źródła:

[1] https://pl.wikipedia.org/wiki/Graf_pe%C5%82ny
[2] https://pl.wikipedia.org/wiki/Cykl_Hamiltona

Spis rysunków

Spis tabel