Marcin Całkowski
nr 218191
Śr. 15.15

Data oddania: 8.11.2017r.
Data poprawy: 14.11.2017r.
Prowadzący: Dr inż. Tomasz Kapłon

Projektowanie efektywnych algorytmów

Sprawozdanie

Zadanie projektowe nr 1

Metoda podziału i ograniczeń dla problemu komiwojażera

1. Wstęp

Celem zadania projektowego jest zastosowanie 3 algorytmów dla wcześniej wybranego zagadnienia, w moim przypadku problemu komiwojażera a następnie porównanie wyników otrzymanych przy użyciu każdego z nich. W kolejnych etapach projektowych zostaną wykonane wybrane metody, w tym przypadku są to branch and bound, tabu search oraz algorytm genetyczny. Algorytmy te należy przetestować dla wybranych instancji naszego problemu oraz porównać wyniki poszczególnych algorytmów. Otrzymane w ten sposób wyniki będzie należało porównać ze sobą, dzięki czemu dowiemy się, który algorytm jest lepszym rozwiązaniem dla konkretnych instancji problemu, jakie ma zalety oraz wady oraz czy warto go stosować dla dowolnych instancji problemu. W pierwszej części projektu wykonany został algorytm branch and bound.

2. Opis problemu komiwojażera

Problem komiwojażera (ang. traveling salesman problem) jest zagadnieniem optymalizacyjnym polegającego na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Najczęstszym przykładem tego problemu jest komiwojażer, który musi odwiedzić wszystkie miasta (wierzchołki grafu) i wrócić do miasta początkowego w taki sposób, aby najkrótszą drogą (lub z jak najmniejszym kosztem) przejść przez każde z miast tylko jeden raz. Główną trudnością problemu jest ilość danych, która wraz ze wzrostem liczby wierzchołków zwiększa się wykładniczo.

Rozróżniamy dwie odmiany problemu komiwojażera: **symetryczną (STSP)** - gdzie odległość między dwoma miastami jest zawsze taka sama oraz **asymetryczną (ATSP)** - w której droga od wierzchołka pierwszego do drugiego może mieć inną długość niż droga pomiędzy tymi samymi wierzchołkami w drugą stronę.

Problem komiwojażera jest **NP-trudny**, co oznacza że nie znaleziono jeszcze żadnego szybkiego rozwiązania problemu i bardzo prawdopodobne że takie w ogóle nie istnieje.

3. Metoda branch and bound

Metoda podziału i ograniczeń (ang. branch and bound) używana jest do wyznaczenia prawidłowego rozwiązania dla problemów optymalizacyjnych poprzez analizę **drzewa stanów** problemu. Drzewo to reprezentuje wszystkie możliwe ścieżki jakimi może pójść algorytm rozwiązując dany problem. Jako że przejrzenie całego drzewa stanów wymagało by bardzo dużej ilości obliczeń ze względu na jego wykładniczy rozmiar, algorytm ten wyznacza granicę dzięki której można sprawdzić, czy potomkowie danego węzła są obiecujący oferując możliwe lepsze rozwiązanie od aktualnego (obliczanie **dolnej granicy**). Dzięki tej wiedzy możemy określić który potomek na pewno nie będzie lepszy od aktualnego rozwiązania (**górnej granicy**) porzucając jego dalsze przeszukiwanie, powodując tym samym szybsze przeglądanie całego drzewa. Pozwala nam to na zmniejszenie liczby obliczeń i w konsekwencji szybszego rozwiązaniu problemu.

Efektywność algorytmu zależy od przede wszystkim od **funkcji wyznaczającej granicę** dla danej części rozwiązania. Jako że będzie ona obliczana dla każdego węzła ważne jest by obliczała ona dobrą granicę, ale jednocześnie była ona szybka w realizacji. Odpowiednie zrównoważenie tych dwóch warunków pozwoli nam uzyskać funkcję dającą prawidłowy wynik w czasie lepszym niż np. przegląd zupełny. Ważna również jest wybrana **strategia przeszukiwań** rozwiązań, której odpowiedni wybór może znacząco wpłynąć na czas działania algorytmu (np. przeszukując w pierwszej kolejności najbardziej obiecujące węzły). [1]

4. Implementacja algorytmu

Pseudokod zaimplementowanego algorytmu prezentuje się następująco:

```
initialize(Q)
enqueue(Q, v)
best ← value(v)
best road ← road(v)
while !empty(Q) do
    v ← dequeue(Q)
    for all każde dziecko u węzła v do
        if value(u) lepsza od best then
            best ← value(u)
            best road ← road(u)
        end if
        if bound(u) jest lepsza od best then
            enqueue(Q, u)
        end if
    end for
end while
```

[1] Informacje na temat algorytmu oraz pseudokod zostały napisane na podstawie opracowania Mateusza Łyczka *Metoda podziału i ograniczeń.* Źródło: https://www.ii.uni.wroc.pl/~prz/2011lato/ah/opracowania/met_podz_ogr.opr.pdf

Jako że powyższy pseudokod przygotowany był z myślą o problemie plecakowym, musiał zostać on przystosowany do problemu komiwojażera. Algorytm po przygotowaniu danych rozpoczyna pracę od obliczania **górnej granicy** przy pomocy algorytmu zachłannego. Dzięki szybkiemu ustaleniu jej jesteśmy w stanie szybciej odrzucić rozwiązania nieoptymalne bez konieczności wyszukania jednego pełnego rozwiązania zadania.

Właściwa część algorytmu opiera się na **stosie** przeszukując drzewo stanów problemu w głąb. Dla ostatnio dodanego wektora na szczycie stosu w pętli sprawdzane są kolejno wszystkie potomki danego węzła. Dla każdego potomka pod warunkiem że cykl jest pełny sprawdzamy, czy jest lepszy od aktualnie najlepszego rozwiązania. Jeśli jest lepszy, to jest on zapisywany jako nowa górna granica. Jeśli cykl nie jest pełny, sprawdzamy jego **dolne ograniczenie** przy pomocy funkcji *bound()* czy jest on obiecujący wrzucając go na wcześniej wspomniany stos. Rozwiązania granicy będące gorsze od aktualnie najlepszego rozwiązania nie są dalej sprawdzane.

Funkcja ***bound()*** użyta w programie działa na zasadzie obliczania aktualnej części rozwiązania, dopełniając go najmniejszymi wartościami dla wierzchołków jeszcze nie obliczonych. Powstałe dwa łańcuchy są ze sobą łączone poprzez znalezienie najmniejszego łącznika dla ostatniego wierzchołka łańcucha rozwiązania częściowego, oraz dla wierzchołka zerowego. W ten sposób otrzymujemy dolną granicę algorytmu.

Złożoność obliczeniowa algorytmu w dużej mierze zależy od danych, gdyż w pesymistycznym przypadku zostanie wykonany przegląd zupełny co daje nam złożoność czasową ***O(n!)***. Jednak dla większości problemów dzięki odpowiedniemu odrzucaniu rozwiązań czas ten będzie krótszy.[2] Zastosowany algorytm, pomijając podaną wcześniej złożoność czasową stosu zawierającego kolejne etapy drzewa stanów część obliczająca granicę (dolną i górną) posiada złożoność $O(n^4)$. Ta część algorytmu jest wykonywana dla każdego rozwiązania znajdującego się na stosie, dzięki czemu widać że wybranie dobrej początkowej granicy górnej, oraz odpowiednie obliczanie granic dolnych są kluczem do szybkiego rozwiązywania problemu.

5. Procedura testowania

Pomiary zostały przeprowadzenie poprzez wykonanie 50 prób dla każdej testowanej instancji. Z tych prób będzie brany pod uwagę średni czas działania próby algorytmu, jako że metoda podziału i ograniczeń powinna obliczać najlepsze rozwiązanie. Do pomiaru czasu użyta została funkcja z biblioteki *chrono* dostępnej od standardu c++11, która pozwala na wystarczająco dokładny pomiar czasu.

[2] Źródło: http://www.geeksforgeeks.org/branch-bound-set-5-traveling-salesman-problem/

Testy przygotowanych zestawów zostały wykonane na prywatnym komputerze. Podczas przeprowadzania testów nie były uruchomione żadne dodatkowe aplikacje, a wszelkie procesy w tle zostały ograniczone do minimum poprzez wyłączenie aplikacji w tle oraz odłączeniu komputera od Internetu (zachowanie jedynie procesów wymaganych przez system operacyjny) w celu uzyskania jak najlepszych wyników czasowych. Konfiguracja sprzętowa maszyny testującej wyglądała następująco:

Procesor: Intel Core i7-4700MQ 2.4 GHz
Pamięć RAM: 16 GB
System operacyjny: Windows 10 Pro N

Jako podstawy testowej użyłem zgodnie z poleceniem plików z podanego źródła[3]. Do testowania użyłem 6 różnych instancji problemu, 3 symetryczne oraz 3 asymetryczne:

gr17.tsp, pa561.tsp, si1032.tsp, br17.atsp, ftv170.atsp i rbg443.atsp

Każdy z plików posiada inny zestaw wierzchołków, w różnej liczbie oraz o różnej odległości względem siebie. Również do każdego z nich dołączona jest znaleziona optymalna ścieżka, co pozwoli nam porównać otrzymane wyniki podczas testów z najlepszymi znalezionymi do tej pory, dzięki czemu będziemy mogli określić czy dany algorytm działa prawidłowo. Wczytywanie plików .tsp i .atsp ograniczyłem jedynie do obsługi wybranych przez siebie plików, jednak program powinien wczytać dowolne dane z wcześniej podanego źródła zapisane jako macierz zwykła bądź diagonalna.

Ze względu na charakterystykę metody podziału i ograniczeń w testach użyte zostały wyłącznie najmniejsze pliki: **gr17.tsp** oraz **br17.atsp**. Mimo wszystko wybrana została cała grupa plików testowych o różnej wielkości instancji, ze względu na porównanie wyników między algorytmami w następnych etapach projektu. Pomoże nam to później porównać zalety i wady danego algorytmu względem pozostałych zaimplementowanych.

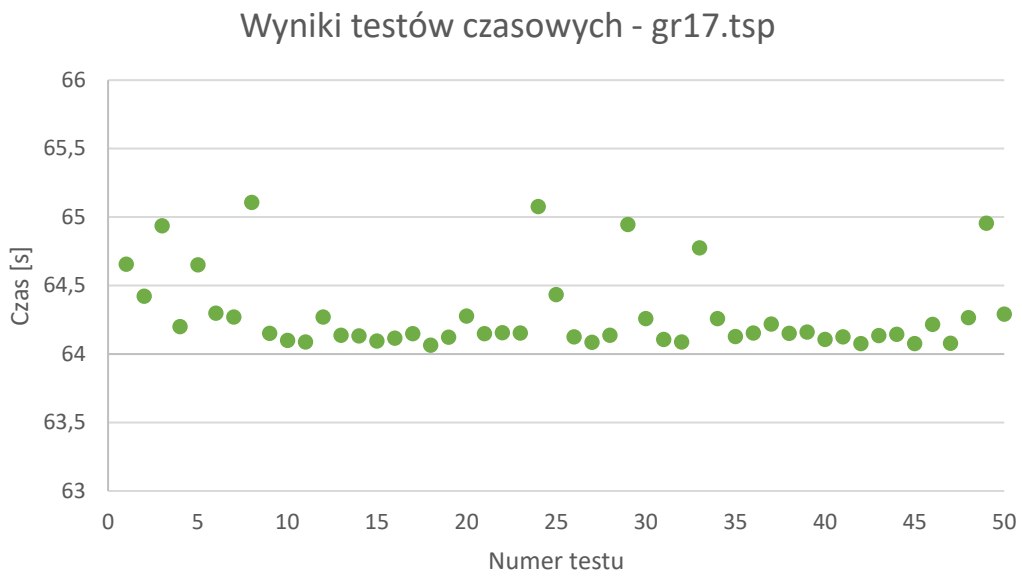
Dodatkowo, poza sprawdzeniem wybranych plików z instancjami zostały przeprowadzone testy na losowych danych o 6 różnych wielkościach instancji: **13, 14, 15, 16, 17 i** 18 Testy danych losowych korzystały z funkcji generującej losową instancję symetryczną dla każdej pojedynczej próby testu. Dzięki temu uzyskany wynik uwzględnia różnicę czasową dal różnego rozkładu danych w testowanej instancji.

Wyniki przeprowadzonych testów znajdują się na następnych stronach.

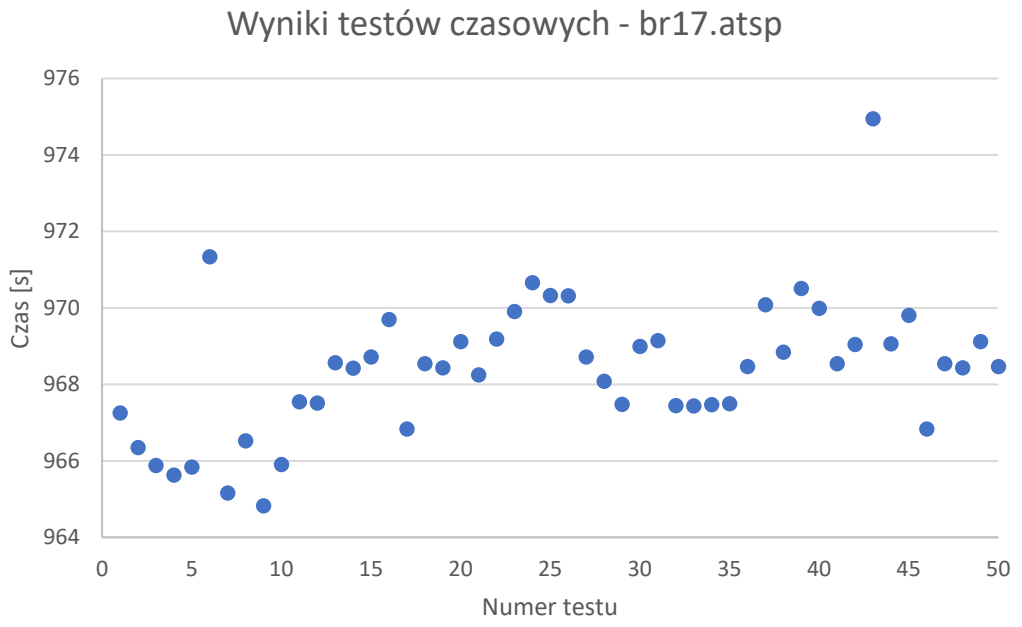
[3] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

6. Wyniki

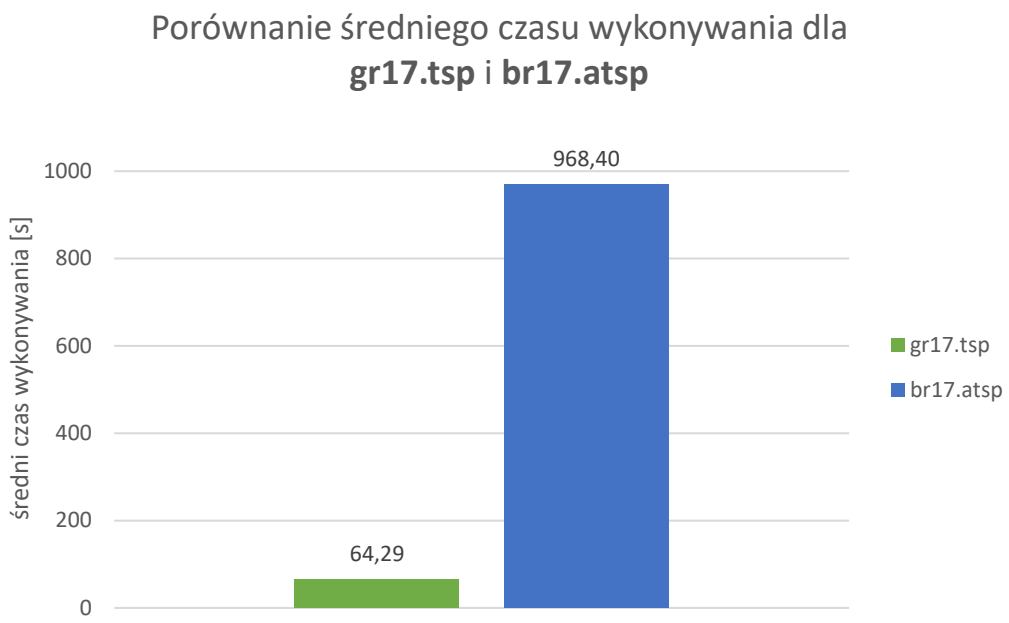
Droga uzyskana dla **gr17.tsp**: 2085 (optymalna)
Droga uzyskana dla **br17.atsp**: 39 (optymalna)



*Wykres 1: Porównanie wyników przeprowadzonych testów dla instancji **gr17.tsp***



*Wykres 2: Porównanie wyników przeprowadzonych testów dla instancji **br17.atsp***



*Wykres 3: Porównanie średnich wyników pomiędzy **gr17.tsp** i **br17.atsp***

Wyniki czasowe dla testów przeprowadzonych dla losowych danych:

Wielkość instancji	13	14	15	16	17	18
Czas [s]	0,055049	0,177178	0,376941	1,303854	3,113497	10,912741

Tabela 1: Średnie wyniki czasowe dla testów przeprowadzonych na losowych instancjach.

Wykres 4: Porównanie średnich wyników czasowych testów na losowych instancjach.

7. Wnioski

Na podstawie testów widać, że algorytm spełnia swoje zadanie jednak nie jest on w pełni zadowalający. Dla wszystkich przeprowadzonych testów generował on optymalną drogę, jednak z długim czasem działania już dla instancji większych niż 16 wierzchołków. Najbardziej rzucająca się w oczy jest różnica czasowa pomiędzy testowanymi plikami, które mimo takiej samej wielkości mają zupełnie różne czasy wykonywania algorytmu (wykres 3). Różnica średnich czasów pomiędzy **gr17** a **br17** wynosi ponad 15 minut. Na różnicę na pewno wpływa niesymetryczność drugiego pliku oraz różne wielkości danych instancji, co widać chociażby po różnorodności optymalnego wyniku, jednak i tak uważam że jest ona bardzo duża.

Przedstawione wyniki dla losowych instancji danych (tabela 1 i wykres 4) pokazują znaczy wzrost średniego czasu wykonywania, co widać szczególnie na instancjach o największej liczbie wierzchołków. Średnie czasy tych testów są zgodne z teorią, jednak porównując losową instancję o wielkości 17 z testowanymi plikami o tej samej wielkości już dla **gr17.tsp** otrzymujemy dużą różnicę czasów wynoszącą ponad minutę, natomiast dla **br17.atsp** ta różnica jest ogromna wynosząc ponad 16 minut. Wyniki te po raz kolejny pokazują jak ważne dla działania algorytmu poza samą wielkością instancji jest to, jakie dane ona posiada.

Sam algorytm zgodnie z teorią zawsze daje nam optymalny wynik dla testowanej instancji, jednak z dużą złożonością obliczeniową już dla instancji wielkości 17. Przeglądanie nawet ograniczonego drzewa stanów i tak stanowi wyzwanie dla szybkości obliczania algorytmu co widać w przedstawionych wynikach. Samo rozłożenie danych w instancji również ma duży wpływ na szybkość jego działania, ponieważ w najgorszym przypadku algorytm będzie musiał przejrzeć całe drzewo stanów wykonując praktycznie przegląd zupełny. Widać więc, że algorytm ten ma sens wyłącznie dla niewielkich problemów absolutnie nie nadając się dla bardziej złożonych instancji.