Projektowanie efektywnych algorytmów

Zadanie 1: Implementacja i analiza efektywności algorytmu podziału i ograniczeń lub programowania dynamicznego dla problemu komiwojażera

Autor: Michał Sikacki (259152)

Grupa projektowa: czwartek 13:15- 15:00

Prowadzący: mgr inż. Antoni Sterna

1. Wstęp teoretyczny

Rozpatrywanym problemem pierwszego zadania projektowego jest problem komiwojażera. Polega on na odnalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym. Cykl Hamiltona polega na przejściu od wierzchołka startowego przez wszystkie wierzchołki dokładnie raz i na powrót do tego samego wierzchołka. Pełny graf ważony to graf, w którym każdy wierzchołek jest połączony ze wszystkimi innymi wierzchołkami grafu. Problem komiwojażera można przedstawić również bardziej intuicyjnie. Załóżmy, że wierzchołkami grafu są miasta. Problem polega na takim odwiedzeniu wszystkich tych miast i na powrót do początkowego aby pokonana droga była jak najkrótsza.
Rozpatrywany przez nas problem komiwojażera jest typu asymetrycznego tzn. koszt przejścia pomiędzy dwoma wybranymi wierzchołkami nie musi być taki sam w jedną i drugą stronę. Istnieją różne algorytmy rozwiązujące ten problem:

a) Przegląd zupełny (ang. brute force)- polega na wyznaczeniu wszystkich możliwych permutacji. Należy przejrzeć wszystkie możliwe drogi, jakie są możliwe i wyznaczeniu spośród wszystkich tych permutacji wartości minimalnej. Algorytm jest bardzo prosty w implementacji. Nie jest jednak efektywny. W asymetrycznym problemie Komiwojażera mamy bowiem do wyznaczenia (N-1)! permutacji. Już dla 19 wierzchołków mamy do wyznaczenia łącznie (19-1)! = 6 402 373 705 728 000 dróg. Złożoność czasowa tego algorytmu wynosi **O(n!)**. Z tego powodu, algorytm ten będzie zwracał wynik w akceptowalnym czasie jedynie dla kilkunastu wierzchołków.
b) Metoda podziału i ograniczeń (ang. branch and bound)- jak sama nazwa wskazuje, algorytm ten stosuje dwie techniki. Można powiedzieć, że jest to w zasadzie usprawniona wersja algorytmu brute force. Algorytm opiera się na analizie drzewa przestrzeni stanów. Metoda ta w każdym węźle oblicza ograniczenie, które pozwala określić na ile obiecujące jest dane rozgałęzienie. W następnych etapach, kiedy algorytm pokonuje kolejne rozgałęzienia, jeśli okaże się, że ograniczenie występujące jest gorsze od wcześniej obliczonego najbardziej optymalnego, to pomijamy rozgałęzienie drzewa wychodzące od danego węzła. Dzięki temu można oszczędzić czas. Złożoność tego algorytmu jest trudna do ustalenia. Najczęściej zależy od implementacji i nie dla każdego wygenerowanego problemu Komiwojażera, algorytm będzie działał tak samo. Jeśli szukany wynik odnajdziemy bardzo szybko, to algorytm szybko pominie inne rozgałęzienia drzewa, dzięki czemu zwróci rezultat bardzo szybko. Wszystko zależy tu jednak od konkretnego przypadku.

2. Przykład praktyczny

a) Branch and Bound

Rozważmy następujący pełny graf ważony:

	1	2	3	4	5
1	x	7	11	13	5
2	6	x	9	2	5
3	14	13	x	9	8
4	6	7	2	x	6
5	4	2	6	9	x

Pierwszym etapem algorytmu jest wyznaczanie dolnego ograniczenia. Będzie ono najmniejszą możliwą drogą, która mogłaby być końcowym rozwiązaniem. Na początku w każdym wierszu należy wyznaczyć minimalną wartość a następnie od każdego elementu wiersza, odejmujemy tę wartość. Macierz po redukcji wierszy będzie wyglądała następująco:

	1	2	3	4	5	min_row
1	x	2	6	8	0	5
2	4	x	7	0	3	2
3	6	5	x	1	0	8
4	4	5	0	x	4	2
5	2	0	4	7	x	2

Postępujemy analogicznie w przypadku kolumn:

	1	2	3	4	5
1	x	2	6	8	0
2	2	x	7	0	3
3	4	5	x	1	0
4	2	5	**0**	x	4
5	0	0	4	7	x
min_col	2	0	0	0	0

Sumując wartości z min_row i min_col otrzymujemy dolne ograniczenie. W tym przypadku wynosi ono 5 + 2 + 8 + 2 + 2 + 2 = 21. Teraz należy przenalizować wszystkie zera znajdujące się w macierzy. Dla każdego z nich bierzemy wiersz oraz kolumnę, w których się znajdują i ponownie szukamy w nich minimum. Sumujemy minimalną wartość z wiersza i kolumny i otrzymujemy wynik. Należy odszukać takie zero w macierzy, dla którego taka obliczona wartość jest największa. Zero to będzie wskazywać, która następna krawędź zostanie dodana (jeśli przejdziemy do lewego poddrzewa) lub usunięta (w przypadku prawego poddrzewa) z listy krawędzi wynikowych. W przypadku powyżej zero dające maksymalną wartość to element

(4,3). Na początku przyjmiemy, że ta krawędź zabierać się będzie w zbiorze wyników. Warto także pamiętać, że w przypadku jej niewybrania dolne ograniczenie w prawym poddrzewie wzrośnie o wcześniej obliczony dla zera wynik (w naszym przypadku jest to 6). Zatem bez krawędzi (4,3) ograniczenie dolne osiąga wartość 21 + 6 = **27**.

	1	2	4	5
1	x	2	8	0
2	2	x	**0**	3
3	4	5	x	0
5	0	0	7	x

W powyższej macierzy analizujemy macierz z wybraną wcześniej krawędzią (4,3). W przypadku jej dołączenia do zbioru wyników musimy usunąć z macierzy wiersz 4 i kolumnę 3, co widać na powyższym rysunku. Symbolem „x" usuwamy element (3,4), ponieważ wybranie tego elementu do zbioru wyników utworzyłoby niespójny graf. Postępujemy analogicznie jak na początku. Obliczamy koszt redukcji macierzy (szukamy minimum w każdym wierszu i kolumnie). W tym przypadku koszt ten wyniesie 0. Zatem dolne ograniczenie nadal będzie wynosić 21. Ponownie szukamy odpowiednie zero i obliczamy dla niego wartość. W tym przypadku będzie to element (2,4). Jeśli krawędź nie zostanie dodana dolne ograniczenie w prawym poddrzewie wzrośnie do **30**.

	1	2	5
1	x	2	0
3	4	x	0
5	**0**	0	x

Dodajemy krawędź (2,4) do zbioru wynikowego. Zatem dotychczasowy zbiór to (4,3), (2,4). Musimy pamiętać aby nie dołączać do rozwiązania krawędzi, które mogłyby utworzyć niespójny graf. Dlatego symbolem „x" usuwamy z macierzy element (3,2). Koszt redukcji na tym etapie nadal pozostaje taki sam- 21. Tym razem niewybranie krawędzi (5,1) spowoduje wzrost dolnego ograniczenia w prawym poddrzewie do **25**.

	2	5
1	2	x
3	x	0

	2	5
1	0	x
3	x	0

Usuwamy wiersz 5 i kolumnę 1. Koszt redukcji na tym etapie wyniesie teraz 2, zatem dolne ograniczenie wynosi już 23. Zostały nam tylko dwie krawędzie do wyboru (1,2) oraz (3,5) (pozostałe zera). Jak widać dolne ograniczenie po wybraniu jednej z tych krawędzi nie wzrośnie, zatem wynik całego tego rozgałęzienia wyniesie 23. Dodajemy ostatecznie krawędzie (1,2) i (3,5) do zbioru wynikowego.

W tym momencie musimy powrócić do analizy reszty rozgałęzień, które nie rozwinęliśmy. Brak krawędzi (2,4) w zbiorze wyników spowodowałby, że w takiej sytuacji minimalny optymalny rezultat wyniósłby jak wcześniej obliczyliśmy- 25, co jest gorszym wynikiem niż 23. Nie ma więc sensu rozważać dalej tej gałęzi- wynik i tak byłby większy. Bez krawędzi (2,4)- najlepszy wynik wyniósłby 30, a 30 > 23. Bez krawędzi (4,3)- 27. Zatem z wykorzystaniem metody branch and bound od razu byliśmy w stanie wyliczyć optymalny rezultat wynoszący 23. Czasami jednak dolne ograniczenie obliczone po przejściu przez wszystkie krawędzie nie będzie optymalne. Wtedy jesteśmy zmuszeni rozwinąć prawe poddrzewo. Dla przykładu jeśli usunęlibyśmy krawędź (4,3) ze zbioru wyników, nasza macierz będzie wyglądać następująco:

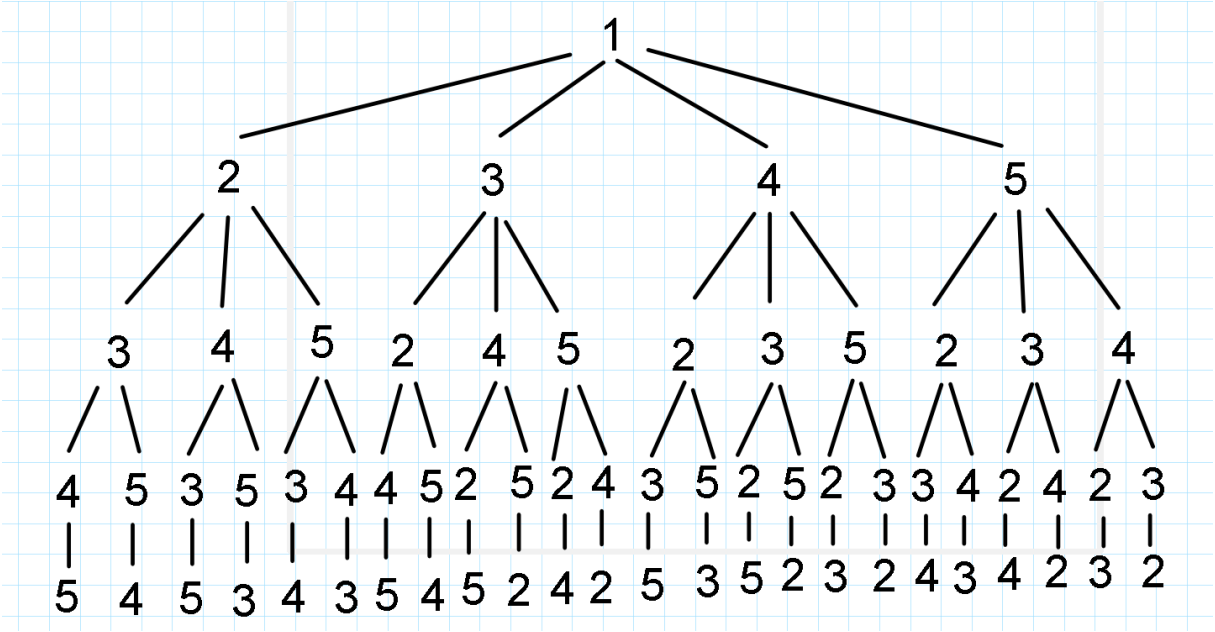
	1	2	3	4	5
1	x	2	2	8	0
2	2	x	3	0	3
3	4	5	x	1	0
4	0	3	**x**	x	2
5	0	0	0	7	x

Krzyżykiem został usunięty element (4,3), a następnie macierz ponownie została zredukowana. W tym miejscu ponownie wykonujemy podobne kroki.

Rozwiązaniem naszego przykładu jest:

23- [4, 3, 5, 1, 2, 4]

b) Brute Force



Na powyższym rysunku zostało przedstawione drzewo wszystkich możliwych permutacji dla problemu TSP.. Aby rozwiązać problem ten metodą należy przejść przez wszystkie możliwe rozgałęzienia drzewa i w każdym z nich obliczyć wynik. Wybieramy najmniejszy ze wszystkich obliczonych i jest to nasze rozwiązanie.

3. Opis implementacji algorytmu

Program rozwiązujący problem komiwojażera został zaprogramowany w C++. Przedstawioną w punkcie 2. macierz jest tzn. macierzą kosztów i każdy z elementów reprezentuje potrzebną do pokonania drogę między dwoma wybranymi wierzchołkami. Macierz ta została w programie zaimplementowana w postaci dwuwymiarowego wektora (klasa <vector> w C++). Pojedyncze elementy są strukturami typu Edge. W strukturze Edge jest przechowywana standardowo koszt przejścia między wierzchołkami oraz indeks kolumny i wiersza elementu. Macierze są przechowywane w specjalnej klasie Matrix, w której znajduje się dodatkowo atrybut „s" oznaczający wielkość macierzy (ile wierszy i kolumn jest w macierzy). W klasie Matrix.cpp znajdują się wszystkie istotne w programie metody- wczytywanie macierzy, losowe jej generowanie, redukcja macierzy, wyświetlanie i zwracanie wielkości.

Algorytm Brute Force:

Wszystkie potrzebne dla algorytmu zmienne, struktury i metody znajdują się w klasie BruteForce. W klasie tej przechowywana jest m. in. lista wynikowa krawędzi (użyto biblioteki <list>). Przechowuje ona struktury typu Edge, która reprezentuje przejście między dwoma wierzchołkami. Oprócz tego istnieje atrybut przechowujący finalny wynik. Po uruchomieniu algorytmu, utworzona zostaje tablica, która będzie przechowywać tymczasowe permutacje (będą w niej gromadzone wierzchołki).

Algorytm działa w bardzo prosty sposób. Iterujemy po utworzonej wcześniej tablicy i pobieramy pomiędzy każdymi sąsiednimi w niej elementami, wartość przejścia pomiędzy nimi. Musimy pamiętać też o tym, żeby pobrać wartość pomiędzy ostatnim elementem tablicy i pierwszym. Wszystkie otrzymane wartości sumujemy i wychodzi nam łączna droga dla danej permutacji. Na końcu porównujemy wynik otrzymany po danej iteracji pętli, z tym który do tej pory był minimalny. Kolejność wierzchołków oraz wynik zostanie zapisany jeśli, otrzymany rezultat będzie mniejszy niż dotychczasowe minimum. Następnie z pomocą funkcji std::next_permutation() z biblioteki <algorithm>, uzupełniamy tablicę o nową permutację i powtarzamy proces.

Algorytm Branch And Bound:

Algorytm ten został zaimplementowany w klasie BranchAndBound i jego działanie jest analogiczne jak dla przykładu w punkcie nr 2. Tutaj po raz kolejny, wykorzystujemy listę (bibilioteka <list>) przechowującą krawędzie wynikowe (struktura Edge). W samym algorytmie, tworzymy dodatkowo listę krawędzi, przechowującą krawędzie dodane przy danym węźle drzewa.

Dwie bazowe funkcje tego algorytmu to *executionLeft()* i *executionRight()*. Służą one do wykonania operacji na danym węźle (lewym lub prawym). Są to funkcje rekurencyjne. Stan macierzy po przejściu przez dany węzeł oraz krawędzie przechowywane na liście tymczasowej są przekazywane kolejnym węzłom.

Funkcja w naszym programie odpowiedzialna za obliczanie ograniczeń dla lewego poddrzewa nosi nazwę *reduceRowColumn()*. Tworzymy listę pomocniczą, która będzie przechowywać wartości z danego wiersza. Następnie używając funkcji std::min() (biblioteka <algorithm>, obliczamy wartość minimalną w tej liście. Lista jest następnie czyszczona i ponownie zostaje zapełniona dla kolejnego wiersza. Wszystkie wartości minimalne zostają zsumowane. Wiersze macierzy są dodatkowo redukowane o minimalne wartości. Analogicznie postępujemy w przypadku kolumn. Koszt redukcji wszystkich wierszy i kolumn jest wzrostem dolnego ograniczenia, który dodajemy do wyniku otrzymanego na wyższym węźle drzewa. Dla prawego poddrzewa wzrost dolnego ograniczenia obliczamy nieco inaczej. Po zredukowaniu macierzy w lewym poddrzewie, iterujemy po wszystkich elementach macierzy, żeby odnajdywać zera. Dla zera znajdującego się w danym wierszu i kolumnie wyznaczamy wartości minimalne podobnie jak to wyżej zostało omówione. Sumujemy je ze sobą. Wartość maksymalna z zer będzie wzrostem dolnego ograniczenia w prawym poddrzewie. Po znalezieniu krawędzi, dodajemy ją do listy tymczasowej.

Wywołania rekurencyjne trwają, dopóki nie dojdzie do sytuacji, w której nie występują żadne zera w macierzy. Jeśli tak jest, to zapisywana jest finalna droga i finalne krawędzie uzyskane z danego pełnego rozgałęzienia. Zostaną one przechowane w pamięci, żeby móc obliczony wynik porównać z innymi węzłami drzewa. Jeśli dojdzie do sytuacji, w której dolne ograniczenie dla prawego poddrzewa nie wzrośnie a jednocześnie będzie więcej niż jedno zero, należy rozpatrzyć wszystkie zera. Drzewo niestety się wtedy rozrasta, ale jest to wymagane dla poprawnego

działania algorytmu. Została zaimplementowana także metoda *checkIfConnected()*, która sprawdza, czy graf pozostaje spójny po dodaniu określonej krawędzi do grafu. Jeśli nie krawędź taka nie będzie brana pod uwagę. Bez tej metody algorytm również nie zawsze zwróci dobry wynik.

Algorytm kończy się, jeśli cały wszystkie wywołane funkcje dla lewego i prawego węzła zwrócą wartość.

Algorytm działa efektywnie. Przed rozpoczęciem pracy każdej z metod reprezentującej rozgałęzienia, i po obliczeniu wzrostu dolnego ograniczenia dla danego węzła sprawdzamy czy otrzymany nie jest już od razu gorszy niż wcześniej otrzymany optymalny. Jeśli tak jest, to funkcja kończy swoją pracę, i nie będzie tworzyła łańcucha wywołań.

4. Plan eksperymentu

Należy przetestować czas działania algorytmu branch and bound oraz brute force w celu zbadania ich efektywności. Dla algorytmu B&B wybieramy dziewięć wartości reprezentujące liczbę wierzchołków {5, 10, 15, 20, 25, 30, 35, 40}. Dla podanych punktów pomiarowych pomiar czasu nie jest bowiem ani za krótki ani też na tyle długi, żeby testy miały trwać bardzo długo. Dla brute force zostały wybrane wartości {4, 5, 6, 7, 8, 9, 10, 11}. Powyżej 11 wierzchołków algorytm brute force działa już za długo. Generujemy 100 losowo wypełnionych macierzy o danym rozmiarze. Zakres wartości elementów to <0, 1000>. Każda wartość to struktura z trzema zmiennymi typu int (12 bajtów). Dla każdej macierzy zapisujemy wynik pomiaru a następnie uśredniamy wynik dla wszystkich 100 pomiarów. Do pomiaru czasu została użyta funkcja QueryPerformanceCounter() a także QueryPerformanceFrequency(). Pomiar czasu rozpoczynamy poprzez zapisanie do zmiennej typu long long int wyniku zwróconej przez specjalnie zaimplementowaną do programu funkcji read_QPC(). Ponowne jej wywołanie zakończy pomiar czasu.

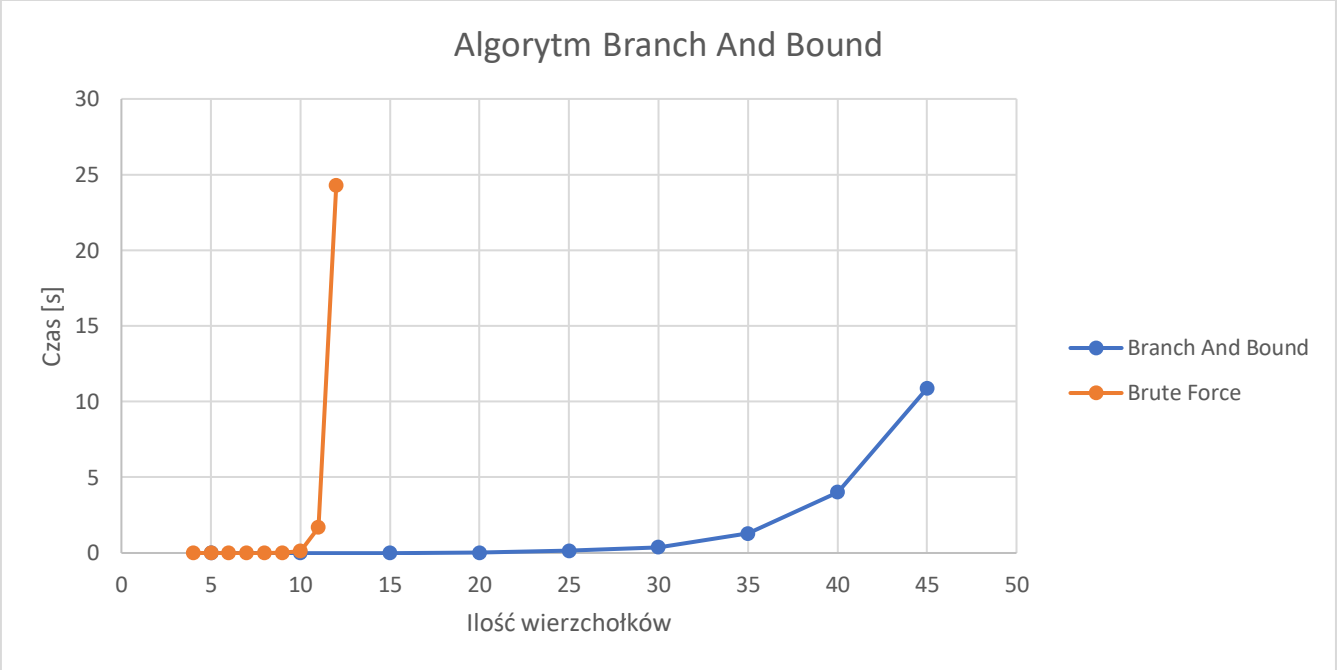
5. Wyniki eksperymentu.

Tabela 1. Wyniki dla algorytmu Branch And Bound

	Liczba wierzchołków								
Czas	5	10	15	20	25	30	35	40	45
[us]	76	1019	7587	26967	154069	375496	1277422	4016748	10868765
[ms]	0.076	1.019	7.587	26.967	154.069	375.496	1277.422	4016.748	10868.77
[s]	0.000076	0.001019	0.007587	0.026967	0.154069	0.375496	1.277422	4.016748	10.86877

Tabela 2. Wyniki dla algorytmu Brute Force

	Liczba wierzchołków								
Czas	4	5	6	7	8	9	10	11	12
[us]	7	11	32	194	1375	12827	131244	1686435	24268806
[ms]	0.007	0.011	0.032	0.194	1.375	12.827	131.244	1686.435	24268.81
[s]	0.000007	0.000011	0.000032	0.000194	0.001375	0.012827	0.131244	1.686435	24.26881



6. Wnioski

Na podstawie wyników eksperymentu, można dojść do wniosku, że zaimplementowany algorytm branch and bound, usprawnia rozwiązanie problemu TSP. Czas rozwiązania problemu zaczyna wyraźnie wzrastać dopiero od 25 wierzchołków. Niemniej jednak trzeba się liczyć, że implementacja takiego algorytmu jest trudniejsza. Jeśli jednak użytkownikowi nie zależy na rozwiązaniu problemu dla bardzo dużej liczby wierzchołków warto rozważyć użycie brute force (dla mniejszych grafów potrafił być nawet minimalnie szybszy niż branch and bound). Warto także wspomnieć, że w trakcie testów branch and bound dało się zauważyć, że dla większych grafów wynik potrafił być różny. Czasami algorytm potrafił się wykonać bardzo szybko a czasami znacznie dłużej niż obliczona średnia. Dzieje się tak, ponieważ można na trafić na różne przypadki- bardziej korzystne i mniej korzystne dla algorytmu.