**PEA**

Marta Głowacka 234999

**Branch and Bound – WERSJA 2**

Wykonałam drugą wersję algorytmu Branch and Bound, jako że pierwszy okazał się nieefektywny ze względu na zbyt wiele pamięci wymaganej przy obliczaniu rozwiązania dla macierzy o rozmiarze 17 (w kolejce było za dużo węzłów, a każdy z nich trzymał w sobie dodatkowo macierz n x n).

Druga wersja algorytmu B&B polega na użyciu funkcji rekurencyjnej i uzyskiwaniu permutacji. Funkcja ograniczająca liczy ograniczenie dolne dla danej permutacji i odcina gałęzie nieobiecujące. Dzięki temu nie wykonujemy części kodu dla wszystkich permutacji, ale tylko tych obiecujących, których ograniczenie jest mniejsze niż aktualne ograniczenie globalne.

Strukturą służącą do mieszczenia permutacji jest vector rozmiaru N (liczba miast) + 1 (na końcu ponownie wpisujemy wierzchołek zerowy), który na samym początku mieści rosnący ciąg: 0,1,2,3,4…0. Podczas algorytmu permutacja jest zmieniana tak, aby żadna konfiguracja się nie powtórzyła (swapowanie wartości na miejscu *step* z każdą i-tą wartością ciągu, poczynając od wartości na miejscu *step*; po tej operacji zwiększenie step i odwrócenie zmian, aby kolejna pętla mogła się wykonać na niezmienionym przez poprzednią pętlę porządku).

Gdy zmienna iterująca kroki – step będzie równa liczbie miast, to znaczy, że zrobiliśmy wszystkie kroki i możemy zaktualizować koszt drogi i podać jako rozwiązanie

**Funkcja ograniczająca**

Jako argument przyjmuje permutację, w skład której wchodzą wszystkie wierzchołki oraz step - piwot, ostatni wierzchołek trasy, miejsce, od którego kolejność wierzchołków nie ma znaczenia, bo nie należą do aktualnie wyliczonej trasy.

Dolne ograniczenie to suma kosztów dla znanej części trasy (do step) i minimalnych kosztów odległości dla jeszcze nie ustalonej trasy.

Zobrazujmy działanie kolejnych trzech pętli przykładowym ciągiem podanym jako argument funkcji calculate\_bound: (pogrubiona 1 to piwot, czyli step). Znana część trasy to 021.

* Najpierw obliczamy drogę dla znanej części trasy, czyli 0-2, 2-1
* Później obliczamy minimalny koszt od jedynki do wszystkich pozostałych wierzchołków, które nie były jeszcze odwiedzone (nie bierzemy pod uwagę zera, bo nie chcemy, aby pojawiło się za szybko i przedwcześnie zakończyło trasę, gdy nie przeszliśmy jeszcze przez wszystkie wierzchołki). 1-3,1-4,1-8,1-7, 1-6, 1-5.

Klamry nie oznaczają zbioru, ale łączą jednie dwa wierzchołki.

* Ostatnia pętla realizuje liczenie minimalnego kosztu od każdego wierzchołka jeszcze nieodwiedzanego. Poniżej przedstawiono tylko pierwszą iterację, dla 3, ponieważ kolejne wyglądają analogicznie:

**Wykresy**

Wykres 1: Wykres czasowy dla B&B i DP

Wykres 2: Wykres logarytmiczny czasowy dla B&B i DP

Wykres 3: Wykres logarytmiczny czasowy dla BF, B&B i DP

**Wnioski**

Teraz można zauważyć, że funkcja B&B oraz DP wyglądają bardzo podobnie. We wcześniejszych eksperymentach algorytm B&B był zaprojektowany nieefektywnie. Jeśli chodzi o tę implementację Branch and Bound (klasa TSP\_Branch), w zależności od optymalizacji funkcji, można było uzyskać nieco inne wyniki – różniące się od minuty po kilkanaście. Najmniej optymalną opcją było zrobienie tablicy visited, która z założenia miała mieścić w sobie indeksy odwiedzonych wierzchołków – wystarczyło jednak zauważyć, że nieodwiedzone wierzchołki są na prawo od step i przeszukiwać nie całą, lecz część macierzy, poczynając od step.

Na początku napisałam funkcję ograniczającą korzystającą z dwóch pętli for – czas przeszukania dla macierzy 17 wierzchołków wynosił ok. 2 minut (zakomentowany kod). Później po próbach optymalizacji i napisaniu 3 pętli for, udało mi się uzyskać wartość ok. 1 minuty dla tej samej macierzy.

Macierz dla 17 wierzchołków:

<http://jaroslaw.mierzwa.staff.iiar.pwr.wroc.pl/pea-stud/tsp/tsp_17.txt>

[dostęp dnia 19.11]