Wrocław, 29.11.2018r.  
Prowadzący: dr inż. Zbigniew Buchalski

Projektowanie Efektywnych Algorytmów

Projekt 1: Rozwiązanie problemu komiwojażera przy pomocy Przeglądu zupełnego i Branch and Bound

Nikita Stepanenko

245816

# Założenia projektowe

Zadaniem projektowym była implementacja dla problemu komiwojażera algorytmów Brute force i Branch and Bound oraz porównanie efektywności obu algorytmów.

Instancją tego problemu jest graf pełny ważony o n wierzchołkach, gdzie każdy z wierzchołków reprezentuje jedno miasto. Waga krawędzi między wierzchołkami to odległość między miastami, przy czym w opisywanej wersji asymetrycznej, odległość z miasta A do B może być i zazwyczaj jest inna niż odległość z miasta B do A. Należy znaleźć minimalny cykl Hamiltona dla każdego grafu, który to cykl będzie najlepszym, optymalnym rozwiązaniem naszego problemu.

# Wstęp teoretyczny

Problem komiwojażera to zagadnienie optymalizacyjne, polegające na znalezieniu najkrótszej trasy – cyklu Hamiltona, łączącego wszystkie miasta – wierzchołki grafu ważonego, w taki, że droga zaczyna się i kończy w tym samym mieście, a komiwojażer nie odwiedza żadnego miasta – wierzchołka, dwukrotnie. Instancją problemu komiwojażera jest zbiór miast, które ma do odwiedzenia komiwojażer oraz dróg łączących te miasta, mających przypisane wagi/koszty/długości. Długość trasy jest sumą długości wszystkich dróg łączących miasta, znajdujących się na trasie. W wariancie symetrycznym problemu komiwojażera, długość drogi łączącej dwa wybrane miasta jest identyczna, gdy pokonujemy ją w obie strony. W wariancie asymetrycznym, długość drogi prowadzącej z miasta A do B nie musi być równa długość drogi prowadzącej z miasta B do A. W ramach zadania projektowego, należało przygotować implementację algorytmu asymetrucznego.

Algorytm symulowanego wyżarzania to algorytm metaheurystyczny, opierający swoje działanie na losowości. Nie znajduje on zawsze najlepszego rozwiązania problemu, w taki sposób jak przegląd zupełny, jednak jego działanie prowadzi do znalezienia rozwiązania optymalnego – trasy, która będzie krótka, ale nie koniecznie najkrótsza z możliwych. Jego działanie oraz nazwa bierze się z analogii do procesu wyżarzania w metalurgii. W trakcie działania algorytmu, stopniowo, wraz z kolejnymi iteracjami pętli, obniżana jest temperatura. Algorytm zaczyna od pewnego rozwiązania początkowego i w kolejnych iteracjach zamienia miejscami losowe elementy trasy (miasta/wierzchołki). Jeśli po zamianie, trasa jest lepsza – krótsza, zostaje ona zapisana, jako dotychczasowo najlepsza, jeśli nie, odrzucamy ją i zamieniamy inne elementy. Aby podczas tych zamian, algorytm nie utknął w minimum lokalnym, dopuszczalne są także zmiany pogarszające obecnie uzyskany wynik, ale pozwalające dojść do lepszego rozwiązania w kolejnych iteracjach.

# Opisy algorytmów

# Przegląd zupełny (Brute force)

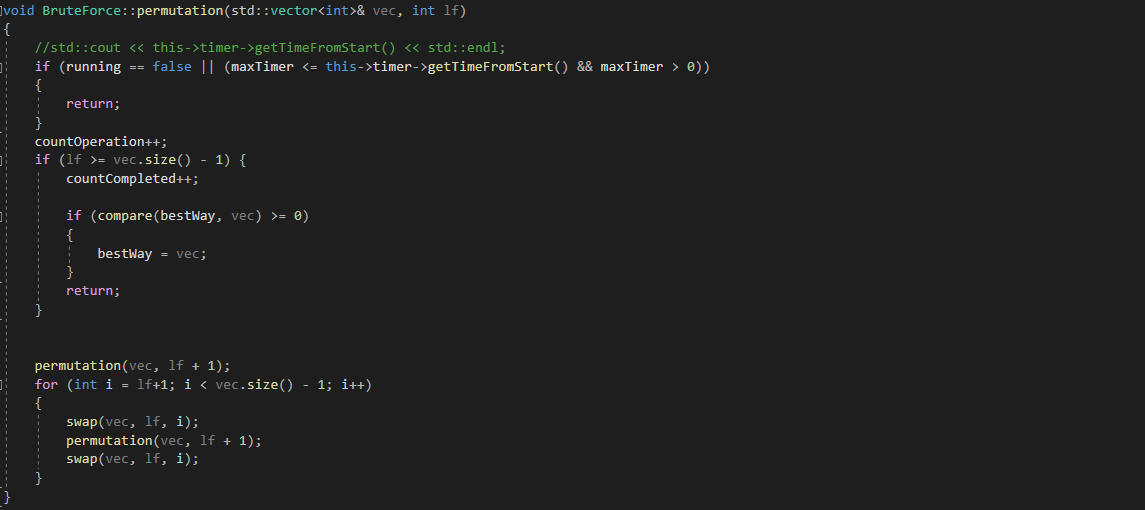
# Idea algorytmu

Przegląd zupełny polega na przeglądaniu wszystkich możliwych drog i wybór najleprzej z nich. Po znalezieniu dowolnej drogi ona jest porównywana z najliepszą drogą która była znaleziona przed tym.

# Realizacja w kodzie programy

Ten algorytm jest zaimpletowany w klasie BruteForce (BruteForce.cpp, BruteForce.h oraz klasie rodziców dla wszystkich zrealizowanych algorytmów dla uproszczenia rozszerzenia programu Algorytm (Algorytm.cpp i Algorytm.h)).

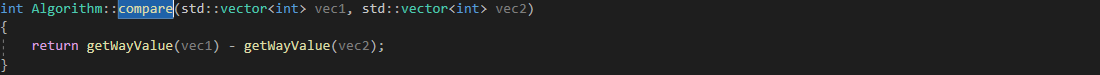
Główną metodą realizująćą algorytm jest rekurencyjna metoda „permutation(std::vector<int>&, int)”



Metoda “swap(std::vector<int>&, int, int)” wymienia miejscami 2 elementy w tablicy.



Metoda “compare(std::vector<int>, std::vector<int>)” porównuje 2 tablicy miedzy sobą.



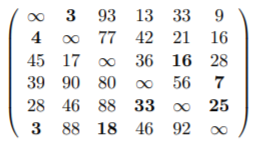
# Podziału i ograniczeń (Branch and Bound)

# Idea algorytmu

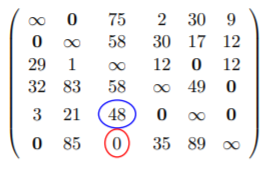
Działanie algorytmu Podziału i ograniczeń opiera się na analizie drzewa przestrzeni stanów. Drzewo to reprezentuje wszystkie możliwe ścieżki jakimi może pójść algorytm rozwiązując dany problem. Algorytm zaczyna w korzeniu drzewa i przechodząc do któregoś liścia konstruuje rozwiązanie. Przeglądanie całego drzewa byłoby bardzo kosztowne ze względu na jego wykładniczy rozmiar, dlatego metoda podziału i ograniczeń w każdym węźle oblicza granicę, która pozwala określić go jako obiecujący bądź nie. W dalszej fazie algorytm przegląda tylko potomków węzłów obiecujących. Pozwala to, razem z dobraniem odpowiedniej strategii odwiedzania wierzchołków oraz liczenia granicy, zmniejszyć ilość odwiedzonych wierzchołków i szybciej znaleźć rozwiązanie problemu.

# Przykład

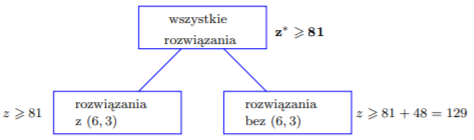
Rozważmy macierz kosztów

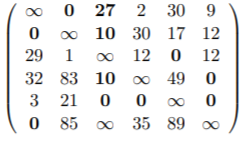


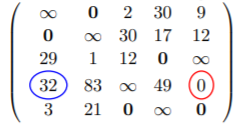
Odejmujemy 3, 4, 16, 7, 25 i 3 od elementów kolejnych wierszy oraz 15 i 8 od elementów kolumny trzeciej i czwartej:



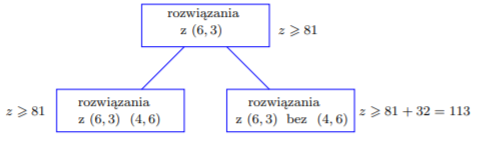
Dolne ograniczenie na długość rozwiązań jest równe 81. Podział zbioru rozwiązań dokonujemy za pomocą łuku (6, 3), ponieważ ten wybór powoduje największy wzrost dolnego ograniczenia w prawym podrzewie, czyli 48.

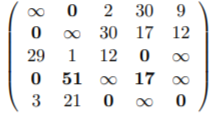


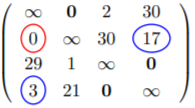
bez wiersza nr 6 i kolumny nr 3 



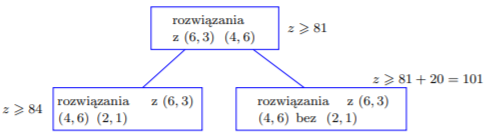
Dzielimy zbiór w lewym podrzewie za pomocą łuku (4, 6). Ten wybór powoduje wzrost dolnego ograniczenia w prawym podrzewie o 32.



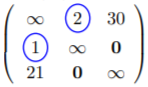
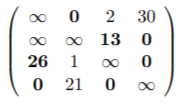
bez wierszy nr 4, 6 i kolumn nr 3, 6



Dzielimy zbiór w lewym podrzewie za pomocą łuku (2, 1). Wybór powoduje wzrost dolnego ograniczenia w prawym podrzewie o 17+3=20. Odejmujemy od wiersza 17, od pierwszej kolumny 3. Zabraniamy przejścia przez łuk (3, 4) (lewe poddrzewo), jeśli w rozwiązaniu są (4, 6) i (6, 3).

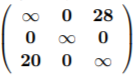


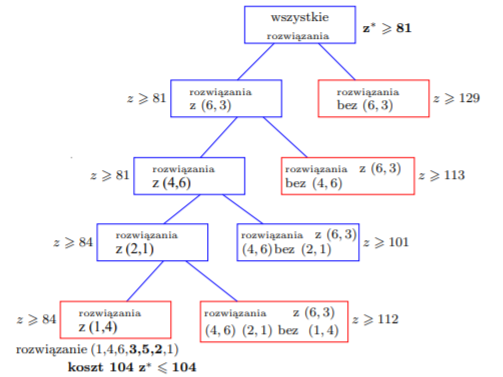
bez wierszy nr 2, 4, 6 i kolumn nr 1, 3, 6



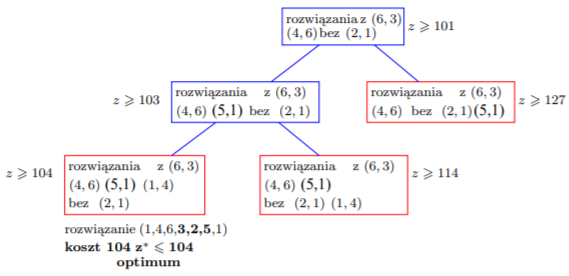
można polepszyć dolne ograniczenie o 3

z ­ 81 + 3 = 84





Należy jeszcze rozpatrzyć rozwiązanie nie zawierające łuku (2, 1).

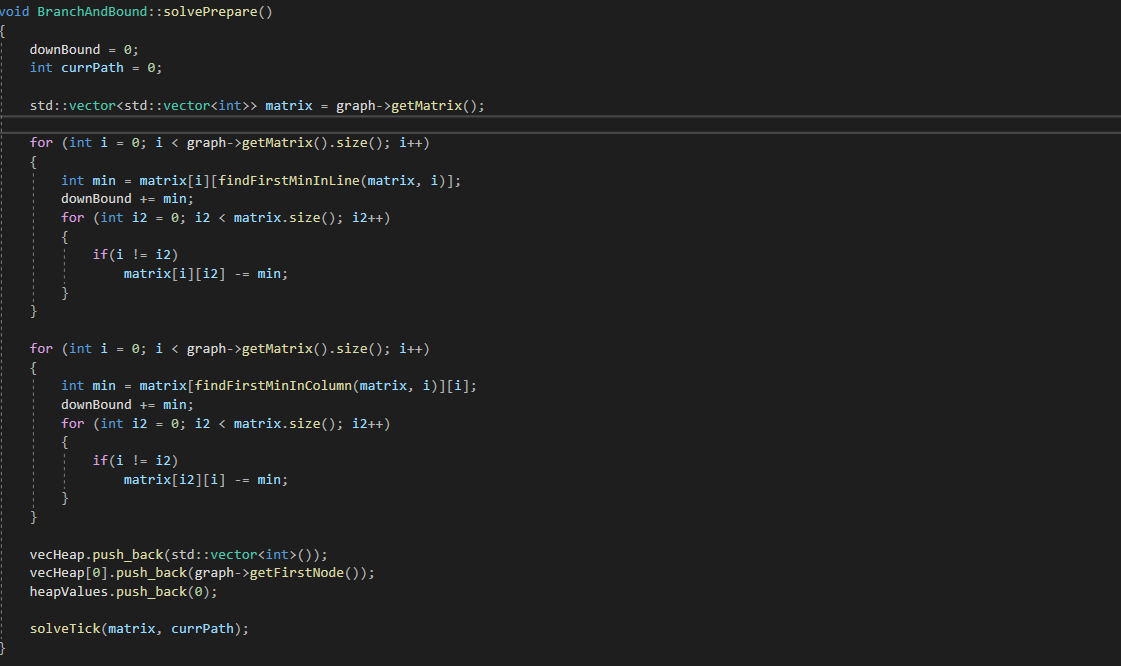


Otrzymaliśmy dwa rozwiązania optymalne o koszcie równym 104.

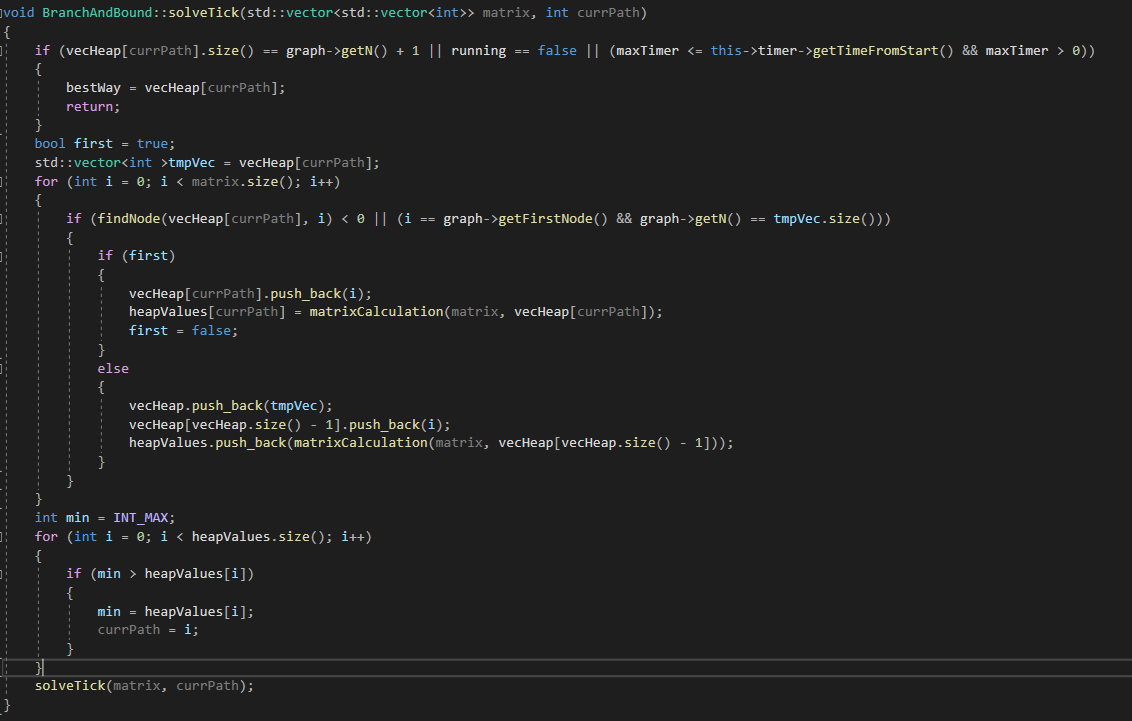
# Realizacja w kodzie programy

Ten algorytm jest zaimpletowany w klasie BruteForce (BranchAndBound.cpp, BranchAndBound.h oraz klasie rodziców dla wszystkich zrealizowanych algorytmów dla uproszczenia rozszerzenia programu Algorytm (Algorytm.cpp i Algorytm.h)).

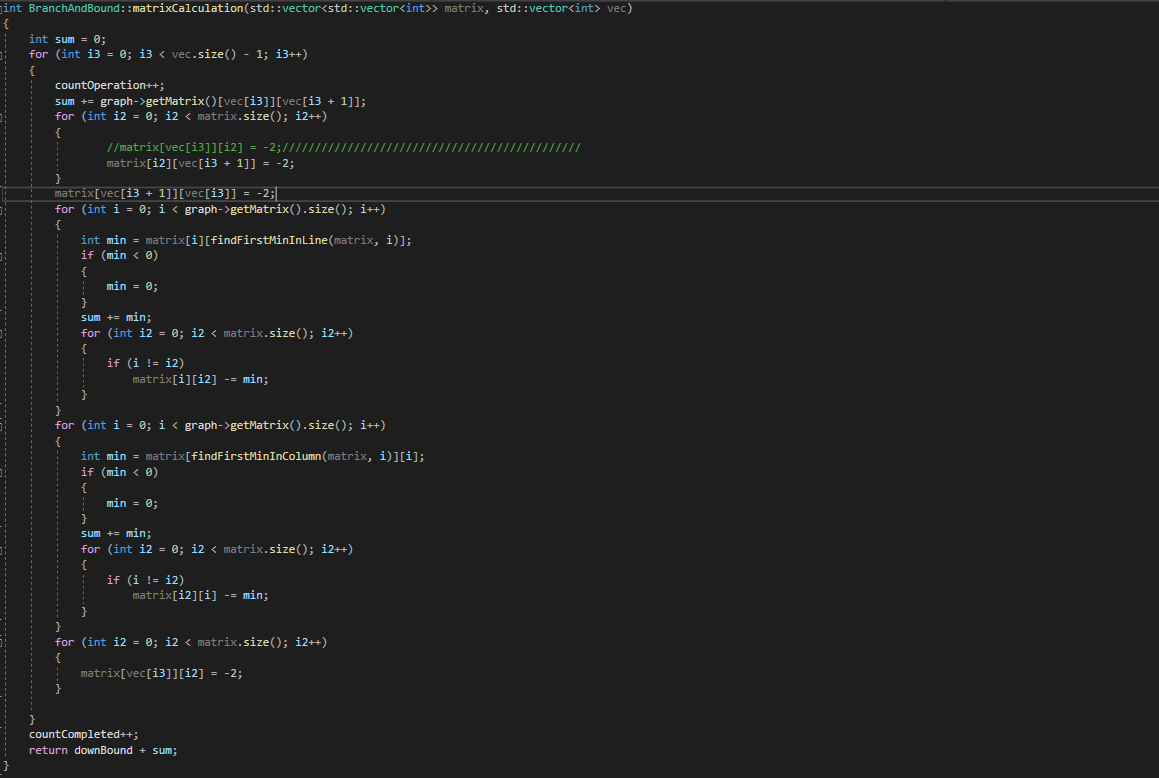
Metoda „solvePrepare()” tworzy czasowe zmienne dla działąnia algorytmu (macierz kosztów, kopiec z drogami i in.) oraz wykonuje odejmowanie kolumn i wierszów oraz określa granicę.



Metoda „solveTick” rozszeża kopiec i wyszukuje najlepszą drogę.



Metoda „:matrixCalculation” usuwa wierzchołki i kolumny(odwiedzane), wykonuje odejmowanie kolumn i wierszów oraz oblicza drogę.

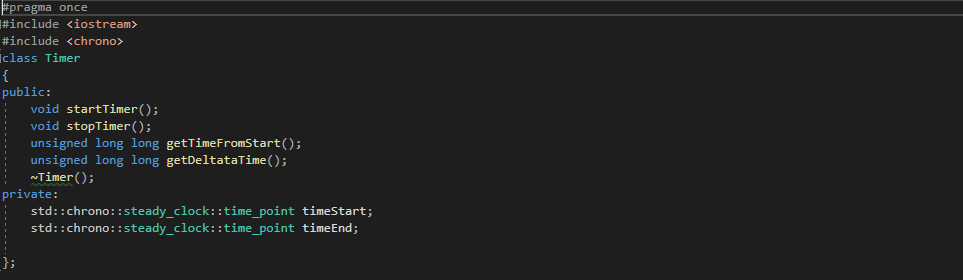


# Obliczenie czasu

Obliczenie czasu jest zrealizowano w klasie Timer (Timer.cpp i Timer.h).

Obliczenie czasu zrealizowano za pamocą biblioteki „chrono”.

Timer.h:



Timer.cpp:



# Wyniki pomiarów

# Przegląd zupełny (Brute force)

# 5.2Podziału i ograniczeń (Branch and Bound)

# 5.3Porównywanie algorytmów

# Wnioski