TSP - implementacja problemu komiwojażera

Zaawansowane Programowanie Obiektowe – AiR

AGH — 2 listopada 2020

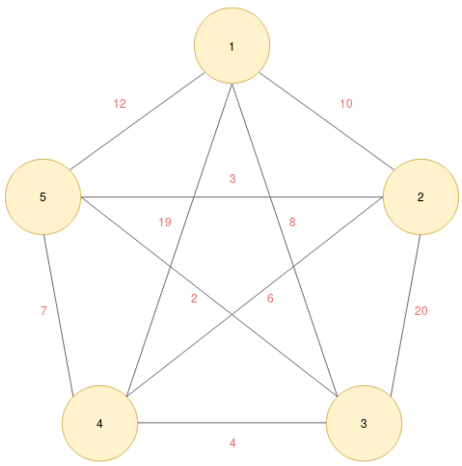
Wstęp

Problem komiwojażera (ang. *Travelling Salesman Problem, TSP*) – czyli problem wędrownego sprzedawcy – określa następujące zadanie: mając listę miast oraz odległości między ich każdą parą, jaka jest najkrótsza ścieżka pozwalająca odwiedzić wszystkie miasta i wrócić do miasta początkowego?

> **Info:** Ćwiczenie polega na uzupełnieniu szkieletu programu rozwiązującego problem komiwojażera. Twoim zadaniem będzie implementacja funkcji realizujących kolejne kroki algorytmu. Laboratorium nie ma na celu szczegółowego opisu zagadnienia komiwojażera od strony teoretycznej – swoją wiedzę uzupełnisz na innych przedmiotach, dedykowanych badaniom operacyjnym.

1 Przybliżenie problemu

Zadanie TSP jest zagadnieniem NP-trudnym, co oznacza, że nie są znane algorytmy o wielomianowej złożoności obliczeniowej pozwalające na rozwiązanie tego problemu (mówiąc wprost: większa liczba miast będzie wpływać na czas obliczeń nie wielomianowo, a na przykład wykładniczo – czyli w znacznie większym stopniu). Dlatego zwykle używa się algorytmów jedynie *aproksymujących* rozwiązanie optymalne. Niemniej jednak, dla małej liczby miast, możliwe jest wykorzystanie metod zdolnych do wyznaczenia wprost tras optymalnych. Takim algorytmem jest na przykład algorytm Little'a.



Rysunek 1: Graf ilustrujący przykładowy problem komiwojażera.

Na Rys. 1 przedstawiono przykładowy problem: komiwojażer ma odwiedzić 5 miast (wierzchołków grafu) odległych między sobą o wartości wpisane w krawędzie grafu. Jak widać jest to problem symetryczny (STSP), gdyż dla dowolnych dwóch miast A i B odległość z A do B jest taka sama, jak z B do A. Można również analizować asymetryczny problem komiwojażera (ATSP), przy czym opisywana metoda Little'a będzie dla obu wariantów taka sama.

1.1 Teoretycznie...

Z teoretycznego punktu, zadanie polega na znalezieniu minimalnego cyklu Hamiltona w pełnym grafie ważonym krawędziowo. Co to oznacza?

Po pierwsze, **graf pełny** jest grafem, w którym każdy wierzchołek jest połączony ze wszystkimi innymi wierzchołkami. **Graf ważony krawędziowo** wprowadza liczbowe etykiety przyporządkowane krawędziom łączącym wierzchołki. **Cykl** jest to ścieżka zamknięta z takim samym pierwszym i ostatnim wierzchołkiem. **Cykl Hamiltona** wprowadza dodatkowe ograniczenia – szuka się **cyklu prostego** (czyli takiej ścieżki, w której każdy – oprócz pierwszego – wierzchołek odwiedzany jest tylko raz) zawierającego wszystkie wierzchołki grafu. Minimalny cykl Hamiltona będzie więc oznaczał taką drogę, dla której suma kolejnych odcinków (krawędzi) będzie najmniejsza.

1.2 Uproszczony przykład

W tym miejscu szczegółowo opisano fragment algorytmu, który jednocześnie stanowi zadanie programistyczne w kolejnych laboratoriach. W opisie zaznaczono kroki, które są realizowane w pełnym algorytmie, i które są już przygotowane w szkielecie.

Rozważmy przykład pokazany na Rys. 1, przepisany do tabeli poniżej (wartości w komórkach oznaczają odległości między miastami 1–5):

	1	**2**	**3**	**4**	**5**
1	–	10	8	19	12
2	10	–	20	6	3
3	8	20	–	4	2
4	19	6	4	–	7
5	12	3	2	7	–

KROK 1 – redukcja. W pierwszej kolejności redukujemy wartości – dla każdego rzędu znajdujemy wartość minimalną i odejmujemy ją od wszystkich elementów w tym rzędzie:

(E1)

	1	**2**	**3**	**4**	**5**
1	–	10	8	19	12
2	10	–	20	6	3
3	8	20	–	4	2
4	19	6	4	–	7
5	12	3	2	7	–

min
8
3
2
4
2

→

	1	**2**	**3**	**4**	**5**
1	–	2	0	11	4
2	7	–	17	3	0
3	6	18	–	2	0
4	15	2	0	–	3
5	10	1	0	5	–

Sprawdzamy, czy w każdym rzędzie i każdej kolumnie występuje co najmniej jedno zero – jeżeli nie, to analogicznie redukujemy wartości w kolumnach:

	1	**2**	**3**	**4**	**5**
1	–	2	0	11	4
2	7	–	17	3	0
3	6	18	–	2	0
4	15	2	0	–	3
5	10	1	0	5	–

min	6	1	0	2	0

→

	1	**2**	**3**	**4**	**5**
1	–	1	0	9	4
2	1	–	17	1	0
3	0	17	–	0	0
4	9	1	0	–	3
5	4	0	0	3	–

Suma wszystkich odjętych wartości wyznacza tzw. **dolne ograniczenie** (ang. *lower bound*, LB) – na danym etapie nie da się znaleźć drogi, której wartość będzie niższa.
W rozważanym przykładzie: $LB = 8 + 3 + 2 + 4 + 2 + 6 + 1 + 2 = 28$.

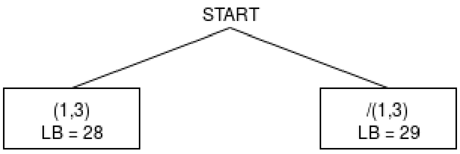
KROK 2 – wybór przejścia. Następnie sprawdzamy wszystkie możliwe przejścia: wybieramy te pary wierzchołków, na których przecięciu w tabeli występuje wartość 0. Następnie obliczamy sumę minimalnych wartości w wyznaczonym wierszu i kolumnie:

$$\begin{array}{ll} (1,3) \rightarrow 1+0=1 & (2,5) \rightarrow 1+0=1 \\ (3,1) \rightarrow 0+1=1 & (3,4) \rightarrow 0+1=1 \\ (3,5) \rightarrow 0+0=0 & (4,3) \rightarrow 1+0=1 \\ (5,2) \rightarrow 0+1=1 & (5,3) \rightarrow 0+0=0 \end{array}$$

Wybieramy krawędź o maksymalnej z wyznaczonych wartości i dodajemy do szukanego rozwiązania (w przypadku gdy więcej niż jedna krawędź ma maksymalną wartość, można wybrać dowolną z tych krawędzi). Wybierzmy krawędź $(1, 3)$.

KROK 3 – aktualizacja macierzy kosztów. Należy teraz usunąć z macierzy kosztów rząd 1 i kolumnę 3 oraz zabronić przejścia powrotnego $(3, 1)$. Należy również sprawdzić, czy istnieje możliwość utworzenia innego cyklu w bieżącym rozwiązaniu – niemniej jednak w pierwszej iteracji jedynym cyklem byłoby przejście powrotne.

Uwaga! W pełnym rozwiązaniu w tym miejscu następuje podział na dwie gałęzie możliwych rozwiązań (zob. Rys. 2): z przejściem $(1, 3)$, czyli bieżące rozwiązanie, oraz bez tego przejścia. Wyznaczamy też LB dla prawego rozwiązania poprzez dodanie do bieżącego LB sumy minimalnych wartości w wybranym wierszu i kolumnie: $(1, 3)$, czyli $LB = 28 + 1$ (w ten sposób określamy koszt nie wybrania wierzchołka).



Rysunek 2: Podział rozwiązania na podproblemy.

Kontynuujemy rozwiązywanie lewej gałęzi. Sprawdzamy, czy w każdym rzędzie i każdej kolumnie występuje wartość 0 – jeżeli nie, dokonujemy odpowiedniej redukcji.

(E2)

	1	**2**	**4**	**5**
2	1	–	1	0
3	–	17	0	0
4	9	1	–	3
5	4	0	3	–

min
0
0
1
0

$\rightarrow$

	1	**2**	**4**	**5**
2	1	–	1	0
3	–	17	0	0
4	8	0	–	2
5	4	0	3	–

min	1	0	0	0

$\rightarrow$

	1	**2**	**4**	**5**
2	0	–	1	0
3	–	17	0	0
4	7	0	–	2
5	3	0	3	–

$LB = 28 + 1 + 1 = 30$

Sprawdzamy możliwe krawędzie:

$$
\begin{array}{ll}
(2,1) \rightarrow 0 + 3 = 3 & (2,5) \rightarrow 0 + 0 = 0 \\
(3,4) \rightarrow 0 + 1 = 1 & (3,5) \rightarrow 0 + 0 = 0 \\
(4,2) \rightarrow 2 + 0 = 2 & (5,2) \rightarrow 3 + 0 = 3
\end{array}
$$

Wybieramy krawędź $(2, 1)$ i usuwamy wybraną drogę z macierzy. W pełnym rozwiązaniu znowu następuje podział na dwie gałęzie (patrz Rys. 3). Dodatkowo należy sprawdzić możliwość innego cyklu. Popatrzmy na bieżące rozwiązanie, złożone z dwóch krawędzi:

$$(1,3),\ (2,1)\ \Rightarrow\ 2 \rightarrow 1 \rightarrow 3 \tag{1}$$

Powrót z wierzchołka 3 do 2 spowodowałby uzyskanie nieprawidłowego rozwiązania – cyklu nie będącego cyklem Hamiltona – dlatego należy takiego przejścia zabronić (patrz macierz E3).

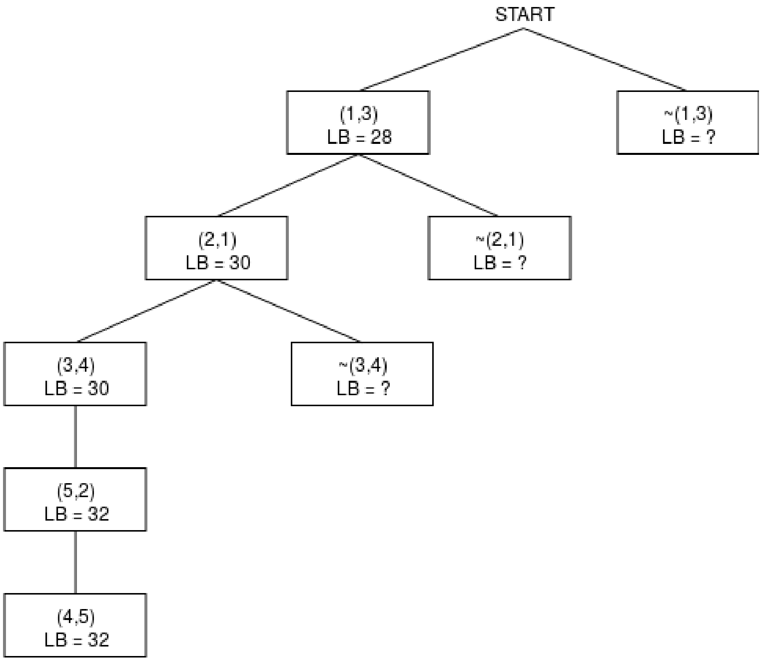
Redukcja nie jest potrzebna, więc od razu przechodzi się do wyznaczenia kolejnych krawędzi:

(E3)

	2	**4**	**5**
3	–	0	0
4	0	–	2
5	0	3	–

$$
\begin{array}{ll}
(3,4) \rightarrow 0 + 3 = 3 & (3,5) \rightarrow 0 + 2 = 2 \\
(4,2) \rightarrow 2 + 0 = 2 & (5,2) \rightarrow 3 + 0 = 3
\end{array}
$$

Wybieramy krawędź $(3, 4)$. Zabraniamy powrotu, jeżeli to konieczne, oraz sprawdzamy rozwiązanie pod kątem pojawienia się cyklu. Rozwiązanie ma postać $2 \rightarrow 1 \rightarrow 3 \rightarrow 4$, więc musimy dodatkowo zabronić przejścia $(4, 2)$



Rysunek 3: Podział rozwiązania na podproblemy – rozwiązana lewa gałąź.

(E4)

	2	**5**
4	–	2
5	0	–

min	0	2

→

	2	**5**
4	0	0
5	0	–

Obecnie $LB = 30 + 2 = 32$. Kolejne krawędzie to: $(5,2) = 0$ oraz $(4,5) = 0$. Wybieramy np. $(5,2)$ i dodajemy do rozwiązania. Sprawdzamy ewentualne cykle. Usuwamy przejście z macierzy.

(E5)

	5
4	0

Odczytujemy ostatnią krawędź: $(4,5) = 0$ Ostatnim krokiem jest posortowanie otrzymanych krawędzi, aby utworzyły drogę:

$$(1,3) \rightarrow (3,4) \rightarrow (4,5) \rightarrow (5,2) \rightarrow (2,1)$$

Wartość dolnego ograniczenia stanowi koszt drogi: $LB = 32$.

W tym momencie otrzymaliśmy jedno z możliwych rozwiązań. Szczęśliwym trafem jest to rozwiązanie optymalne, natomiast nie jest to koniec rozwiązania pełnego algorytmu. W tym momencie musimy się wrócić do prawych gałęzi naszego drzewa. Dla każdej takiej gałęzi najpierw sprawdzamy LB:

- Jeżeli jest wyższe od końcowego LB nie musimy jej w ogóle analizować.
- Jeżeli jest niższe, należy uruchomić rozwiązanie od początku, dla macierzy początkowej z zabronionym danym (dla gałęzi) przejściem. Oczywiście ta gałąź będzie się rozchodzić na kolejne gałęzie.

W ten sposób wyznaczymy inne rozwiązania optymalne (lub w przypadku gdy mieliśmy mniej szczęścia, dopiero je wyznaczymy).

Szczęśliwie obsługę całego drzewa rozwiązań zaimplementowaliśmy za Ciebie. Twoim zadaniem będzie uzupełnienie kilku funkcji.

2 Implementacja

Info: Ponieważ uzupełniasz szkielet implementacji musisz dość szczegółowo trzymać się wytycznych.

Twoim zadaniem będzie uzupełnienie funkcji realizujących jedną iterację rozwiązania:

1. Redukcja w wierszach i kolumnach
2. Wybranie kolejnego przejścia
3. Zaktualizowanie macierzy kosztów

Zapoznaj się z zawartością plików *tsp_setup.hpp* oraz *tsp_setup.cpp*, w których zaproponowano aliasy oraz sposób reprezentacji zabronionych przejść.

W pliku *TSP.hpp* zdefiniowano klasy oraz funkcje potrzebne do pełnego rozwiązania – Twoim zadaniem będzie uzupełnienie ich implementacji w pliku *TSP.cpp*.

2.1 Część 1 – macierz kosztów

W pierwszej części uzupełnij klasę reprezentującą macierz kosztów `CostMatrix`. Klasa przechowuje macierz oraz implementuje następujące operacje na macierzy:

1. `get_min_values_in_rows()` – znajdź minimalne wartości w wierszach
2. `get_min_values_in_cols()` – znajdź minimalne wartości w kolumnach
3. `reduce_rows()` – redukcja minimalnych wartości w wierszach
4. `reduce_cols()` – redukcja minimalnych wartości w kolumnach
5. `get_vertex_cost()` – znajdź koszt nieodwiedzenia **danego** wierzchołka, tj.: znajdź sumę minimalnych wartości dla danej pary wierzchołków, patrz KROK 2 – wybór przejścia.

2.2 Część 2 – etap rozwiązania

W drugiej części uzupełnij klasę reprezentującą jedną gałąź rozwiązania. Klasa przechowuje macierz kosztów i implementuje metody potrzebne do znalezienia rozwiązania w każdej iteracji:

1. `reduce_cost_matrix()` – redukcja macierzy (KROK 1): funkcja zwraca sumę zredukowanych wartości
2. `choose_new_vertex()` – wybór nowego przejścia (KROK 2)
3. `update_cost_matrix()` – aktualizacja macierzy kosztów (KROK 3): zabronienie powrotnego przejścia oraz sprawdzenie ewentualnych cykli
4. `get_path()` – dla zredukowanej macierzy kosztów 2×2 oraz dotychczasowego rozwiązania, zwróć ścieżkę końcową (rozwiązanie) w postaci ciągu kolejnych etykiet wierzchołków

2.3 Część 3 – pełne rozwiązania

Korzystając z Twoich funkcji oraz naszego szkieletu, uzupełnij główną pętle rozwiązania w funkcji `solve_tsp()`. Twoim zadaniem jest uzupełnienie lub poprawienie instrukcji oznaczonych `@TODO` zgodnie z komentarzami w kodzie, dla lewej gałęzi rozwiązania `left_branch`. Każde `@TODO` wiążę się z jedną instrukcją (wywołanie jednej metody na obiekcie `left_branch`).

2.4 Wskazówki

1. W pliku *main.cpp* zawarto kilka przykładowych macierzy wraz z rozwiązaniami.
2. W pierwszej kolejności zamiast uzupełniania funkcji `solve_tsp()` wypróbuj swoje rozwiązanie dla jednej gałęzi (np. opisanej w niniejszym konspekcie). Prościej będzie znaleźć ewentualny błąd.
3. Dobrą praktyką będzie też sprawdzanie kolejnych implementowawnych funkcji – na zasadzie testów jednostkowych.
4. Do wypisywania macierzy w trakcie debugowania wykorzystaj przeciążoną funkcję.
5. Jeżeli coś jest niejasne – pytaj!