Zadanie rekrutacyjne – obliczanie cen przejazdów między hangarami

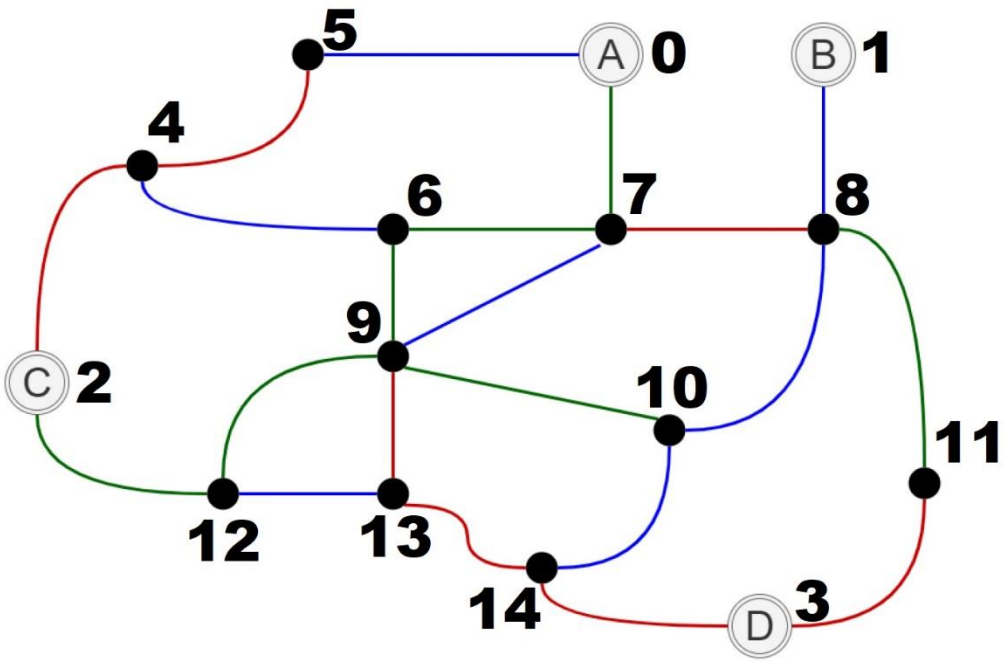
Czytając zadanie zauważyłem że w rozwiązaniu możemy potraktować mapę hangarów jako graf oraz skorzystać z popularnego algorytmu Dijkstry.

Użytkownik, chcąc obliczyć cenę, podaje parametry określone w zadaniu w konsoli. Parametry są od razu sprawdzane – ceny przejazdu w zależności od koloru oraz liczba hangarów muszą być liczbami nieujemnymi. Komunikat (prompt) wyświetla się do skutku – aż użytkownik poda prawidłową wartość. Zastosowałem do przyjmowania danych wejściowych osobną funkcję, jako że dzieje się to kilkukrotnie. Podobnie wygląda przyjmowanie listy odwiedzonych hangarów, użytkownik wpisuje ją w formacie przykładowo:

B A B D C

Komunikat wyświetla się aż użytkownik poda odpowiedniej długości listę, składającą się tylko z liter od A do D.

Dane są przekazywane do funkcji **calculate_cost_of_travel()**. W funkcji **create_graph()** jest tworzona instancja klasy **UndirectedGraph**. Zdecydowałem się na reprezentację grafu poprzez macierz sąsiedztwa (adjacency matrix), każdemu wierzchołkowi grafu przyporządkowałem indeks:



Wierzchołkom A-D nadałem indeksy 0-3 alfabetycznie, natomiast pozostałe wierzchołki mają wyłącznie indeks, jako że nie potrzebują litery. Litery A-D służą jedynie do przyjęcia informacji od użytkownika, a program operuje na indeksach. Za zamianę liter A-D na ich indeksy odpowiada funkcja **translate_letters_to_indexes().**

Graf jest nieskierowany, więc macierz sąsiedztwa jest symetryczna. Przykładowo, dodając połączenie między hangarem B a wierzchołkiem poniżej, podaję indeks 1, indeks 8, oraz kolor krawędzi. Nie muszę dodatkowo dodawać połączenia w drugą stronę, funkcja **add_edge()** robi to symetrycznie. Zarówno na pozycji (1, 8) jak i (8, 1) jest wpisywana waga danej krawędzi (czyli koszt przejazdu).

Mając już obiekt grafu oraz sekwencję hangarów do których podróżujemy zapisaną indeksami wierzchołków, pozostaje przejść iteracyjnie przez wszystkie przejazdy – (przykładowo mając sekwencję B A B D C rozpatrujemy cztery przejazdy: B-A, A-B, B-D, D-C) oraz dla każdego przejazdu obliczyć jego koszt, przy pomocy algorytmu Dijkstry zawartego w metodzie **dijkstra** klasy **UndirectedGraph**. Sumujemy koszty wszystkich przejazdów by otrzymać łączny koszt, który wypisujemy w konsoli.

Nie opisuję tutaj samego algorytmu Dijkstry, wiedzę na jego temat pozyskałem min. z książki o algorytmach autorstwa Aditya Bhargava oraz z kursu Algorytmy i Struktury Danych na drugim semestrze studiów.

Pisząc kod starałem się by był on czytelny ale jednocześnie nie rozwlekły. Każda funkcja posiada **docstring** wyjaśniający pokrótce działanie oraz **type hints** w celu ułatwienia zrozumienia i debugowania kodu. Dodatkowo korzystałem z **list comprehensions** tam gdzie miało to sens, oraz dodawałem dodatkowe komentarze, najczęściej dotyczące kilku następnych linijek kodu. Wszystkie funkcje i zmienne są nazwane po angielsku, jednak komentarze oraz komunikaty do użytkownika są napisane po polsku. Rozbiłem kod na wiele funkcji oraz na dwa pliki, dla większej przejrzystości.