Projekt
Zwiedzanie Muzeów

Ewelina Preś, Franciszek Saliński

Grudzień 2024

Raport z projektu z przedmiotu Algorytmy i struktury danych

Spis treści

1 Treść zadania

Bitowice to miasto o bardzo wielu atrakcjach turystycznych. Jest tam k bardzo ciekawych muzeów. Sieć komunikacyjna w Bitowicach obejmuje n przystanków autobusowych, które łączy m ulic. Przystanek nr 1 to dworzec kolejowy. Parę przystanków p i q może łączyć co najwyżej jedna ulica, a autobusy kursują w obie strony. Znany jest czas przejazdu autobusem pomiędzy przystankami. Przy każdym z k muzeów jest dokładnie jeden przystanek autobusowy. W najbliższy weekend Bajtazar postanowił przyjechać wczesnym rankiem do Bajtowic i zwiedzić wszystkie muzea. Swoją trasę rozpoczyna i kończy na przystanku autobusowym przy dworcu kolejowym. Bajtazar postanowił, że będzie zwiedzał muzea w kolejności rosnącej numerów przystanków, przy których znajdują się te muzea. Niestety, czas ma ograniczony, więc chce tak określić trasę przejazdu autobusami, by łączny czas przejazdu był minimalny.

Wejście:
Należy wylosować liczbę n (liczba przystanków autobusowych), liczbę m (liczba połączeń między przystankami) oraz liczbę k muzeów ($k \leqslant n$). Następnie, dla każdej pary przystanków, które łączy trasa autobusowa, losowo podać czas przejazdu.

Wyjście:
Program podaje minimalny czas przejazdu autobusami na trasie wycieczki Bajtazara.

2 Rozwiązanie

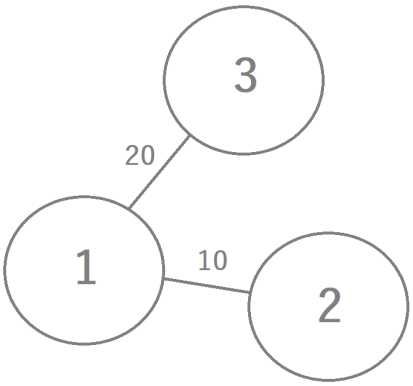
Problem zwiedzania muzeów przez Bajtazara zamodelowaliśmy za pomocą grafu ważonego, gdzie wierzchołki to przystanki, krawędzie to ulice, a muzea znajdują się w pewnych wyróżnionych wierzchołkach. Do rozwiązania problemu wykorzystaliśmy algorytm Dijkstry, który pozwalał nam na znalezienie najkrótszej ścieżki od pewnego wierzchołka, w którym się aktualnie znajdujemy, do wierzchołka, do którego chcemy się dostać. Algorytmu Dijkstry użyliśmy w sumie $k + 1$ razy (gdzie - przypomnijmy - k to liczba muzeów). Na początku za jego pomocą liczyliśmy najkrótszą trasę od dworca do pierwszego muzeum (lub drugiego muzeum, o ile pierwsze muzeum znajduje się przy dworcu). Następnie, używaliśmy go w celu znalezienia najkrótszej ścieżki między pierwszym muzeum a drugim, itd., aż do momentu wyznaczenia najkrótszej ścieżki między ostatnim muzeum a dworcem (przystankiem końcowym). W funkcji route odtwarzamy najkrótszą ścieżkę wyznaczoną przez algorytm Dijkstry wywoałany $k + 1$ razy i obliczamy całkowity czas przejazdu. Zadanie rozwiązywaliśmy przy użyciu języka programowania Python.

2.1 Reprezentacja grafu

W naszym rozwiązaniu reprezentowaliśmy graf ważony jako słownik, w którym kluczami są numery wierzchołków grafu, a wartościami - również słowniki. Dla czytelności będziemy te słowniki nazywać słownikami wag, bo wartościami w tych słownikach są wagi krawędzi. Kluczami w słownikach wag są wierzchołki. W grafie występuje krawędź między wierzchołkami A i B, jeśli wartością dla klucza A jest słownik wag, w którym występuje klucz B (lub odwrotnie: w grafie występuje krawędź między wierzchołkami A i B, jeśli wartością dla klucza B jest słownik wag, w którym występuje klucz A. Jest to równoważne, bo w naszym zadaniu nie rozpatrujemy grafów skierowanych - jeśli da się dojść z A do B, to da się również dojść z B do A).

Przykład: {1: {2: 10, 3: 20}, 2: {1: 10}, 3: {1: 20}}
Taki słownik reprezentuje graf przedstawiony na rysunku poniżej (rysunek 1).

Wartością dla klucza 1 jest słownik wag o kluczach 2 i 3 i wartościach 10 i 20 odpowiednio. Oznacza to, że wierzchołek 1 jest połączony z wierzchołkami 2 i 3, a wagi krawędzi wynoszą 10 i 20. Ponieważ dla kluczy 2 i 3 wartościami są słowniki wag, w których jedynym kluczem jest 1, oznacza to, że te wierzchołki połączone są wyłącznie z wierzchołkiem 1.



Rysunek 1:

Mówiąc jeszcze prościej, nasz słownik koduje następującą symetryczną macierz wag:

$$\begin{bmatrix} 0 & 10 & 20 \\ 10 & 0 & 0 \\ 20 & 0 & 0 \end{bmatrix}$$

gdzie wartość w i-tej kolumnie i j-tym wierszu oznacza wagę krawędzi między i-tym a j-tym wierzchołkiem.

2.2 Algorytm Dijkstry

Algorithm 1 Algorytm Dijkstry

```
1:  procedure DIJKSTRA(V, E, s)
2:      forall v ∈ V do
3:          if (s, v) ∈ E then
4:              D[v].dist ← l(s, v);
5:          else
6:              D[v].dist ← +∞;
7:          fi;
8:          D[v].prev ← null;
9:      od;
10:     D[s].dist ← 0;
11:     S ← {s};
12:     while S ≠ V do
13:         mindist ← +∞;
14:         forall w ∈ V \ S do
15:             if D[w].dist < mindist then
16:                 v ← w;
17:                 mindist ← D[w].dist;
18:             fi;
19:         od;
20:         S ← S ∪ {v};
21:         forall w ∈ V − S do
22:             if D[w].dist > mindist + l(v, w) then
23:                 D[w].dist := mindist + l(v, w);
24:                 D[w].prev := v;
25:             fi;
26:         od;
27:     od;
28: end procedure
```

Algorytm Dijkstry służy do wyznaczania najkrótszych tras z pewnego wierzchołka startowego, do dowolnego innego wierzchołka grafu.

Algorytm Dijkstry przebiega następująco. Na wejściu dostajemy zbiór wierzchołków i wagi krawędzi grafu oraz pewien wierzchołek początkowy s. Na początku inicjalizujemy wektor odległości wierzchołków (mamy tu na myśli odległości od wierzchołka początowego s) oraz wektor „wierzchołków poprzedników". Wektor

odległości jest początkowo wypełniony znakami $+\infty$ tam, gdzie nie ma bezpośredniego połączenia między wierzchołkami, a tam, gdzie występuje połączenie między danym wierzchołkiem a wierzchołkiem początkowym s, wektor przechowuje wagę krawędzi (odległość s od s to 0, więc uzupełniamy odpowiednie miejsce tablicy zerem). Wektor „wierzchołków poprzedników" jest początkowo pusty. Będziemy teraz uzupełniać oba wektory informacjami. Aby to zrobić, musimy jeszcze wprowadzić pewną zmienną - u nas S. S, czyli zbiór wierzchołków odwiedzonych, będziemy stopniowo uzupełnić indeksami wierzchołków, które już odwiedziliśmy. Na początku do zbioru należy tylko wierzchołek startowy. Przejdźmy teraz do uzupełniania naszych wektorów. Wchodzimy w pętlę *while*, która zostanie przerwana dopiero wtedy, gdy odwiedzimy wszystkie wierzchołki. Na początku każdego kroku pętli przypisujemy zmiennej *mindist* wartość $+\infty$. Następnie wchodzimy do pętli for i porównujemy odległości od wierzchołka startowego s wszystkich nieodwiedzonych wierzchołków ze zmienną *mindist*. Jeśli odległość od nieodwiedzonego wierzchołka jest mniejsza niż obecna wartość *mindist*, aktualizujemy wartość *mindist* na tę odległość, a wierzchołek, który rozpatrywaliśmy, przypisujemy do zmiennej v. W ten sposób, w momencie zakończenia pętli for, v będzie oznaczało najbliższy wierzchołkowi s jeszcze nieodwiedzony wierzchołek, a *mindist* będzie odległością między tymi dwoma wierzchołkami. Teraz dodajemy ten najbliższy wierzchołek do zbioru odwiedzonych wierzchołków. Musimy zaktualizować wektor odległości - uzupełnić go o odległości tych wierzchołków, z których da się bezpośrednio dojść do wierzchołka, który właśnie dodaliśmy do zbioru wierzchołków odwiedzonych. Ponieważ teraz może przez niego przebiegać nasza trasa od wierzchołka startowego, możliwym jest, że teraz będą istniały krótsze trasy dojścia do pewnych wierzchołków, a dojście do niektórych innych wierzchołków zacznie być możliwe, mimo że wcześniej nie było. Dla każdego nieodwiedzonego wierzchołka w, jeśli suma najmniejszej odległości od s do v i odległości od v do w, jest mniejsza niż odległość w od s obecnie zapisana w naszym wektorze, to aktualizujemy element wektora odległości oznaczający odległość w od s o nową, krótszą odległość. Wierzchołek v zapisujemy jako poprzednik wierzchołka w. Oznacza to, że na najkrótszej trasie od s do wierzchołka w, przedostatnim wierzchołkiem (przed wierzchołkiem w) jest v. Procedura kończy się, gdy wszystkie wierzchołki zostaną odwiedzone. W ten sposób wyznaczone zostaną najkrótsze odległości między wierzchołkiem s a każdym innym wierzchołkiem. Ponadto, możemy odtworzyć trasę, którą trzeba wybrać, aby iść trasą o najkrótszej drodze. Jeśli chcemy dotrzeć do wierzchołka w, wystraczy spojrzeć, jaki jest jego poprzednik, przypuśmy, że jest to z. Podobnie możemy sprawdzić, jaki wierzchołek jest poprzednikiem wierzchołka z, itd., dochodząc w ten sposób do wierzchołka początkowego i kończąc przy tym odtwarzanie trasy. Algorytm Dijkstry zwraca więc 2 wektory, które zawierają informacje o długości

najkrótszych tras z wierzchołka s do dowolnego wierzchołka oraz o przebiegu tych tras.

Jak wiemy, algorytm **Dijkstry** w implementacji opartej na macierzy sąsiedztwa ma złożoność czasową $\mathcal{O}(n^2)$, gdzie n oznacza liczbę wierzchołków grafu.

2.3 Algorytm rozwiązujący zadanie

Algorithm 2 Funkcja ROUTE - wyznaczanie optymalnej trasy

```
1: procedure ROUTE((M, G))                    ▷ M - lista muzeów, G - graf
2:     sort M;                                ▷ Sortowanie listy muzeów
3:     if M ≠ [1] then
4:         append 1 to M;
5:     fi;
6:     current_stop ← 1;
7:     time ← 0;
8:     route ← [1];
9:     forall i ∈ M do
10:        distances ← DIJKSTRA(current_stop, G);
11:        time ← time + distances[i].dist;
12:        bus_stop ← i;
13:        subroute ← [bus_stop];
14:        while bus_stop ≠ current_stop do
15:            bus_stop ← distances[bus_stop].previous;
16:            append bus_stop to subroute;
17:        od;
18:        route ← route + reverse(subroute)[1 :];
19:        current_stop ← i;
20:    od;
21:    return (time, route);
22: end procedure
```

Jak już wspomnieliśmy, nasze rozwiązanie zadania polega na użyciu algorytmu Dijkstry $k+1$ razy (gdzie k to liczba muzeów) w funkcji, którą nazwaliśmy $ROUTE$. Na wejściu dostajemy graf G (jego wierzchołki i krawędzie wraz z wagami) oraz listę muzeów M (zawierającą numery przystanków, przy których znajdują się muzea). Ponieważ Bajtazar ma odwiedzać muzea w kolejności rosnącej numerów przystan-

ków, sortujemy listę muzeów. Dodajemy też na końcu tej listy przystanek 1, bo po odwiedzeniu wszystkich muzeów musimy wrócić na dworzec, który znajduje się przy przystanku 1 (robimy to zawsze oprócz sytuacji, w której jedyne muzeum znajduje się przy dworcu). Ustawiamy też, zgodnie z poleceniem, zmienną *current_stop* na 1, ponieważ zaczynamy zawsze z przystanku przy dworcu. Wprowadzamy też zmienną czasu przejazdu *time*, której początkowa wartość wynosi 0, i zmienną trasy *route*, która jest listą i będzie zawierać numery przystanków, przez które kolejno przejeżdżamy. Początkowo do listy należy tylko przystanek 1. Następnie wchodzimy do pętli for, w której będziemy w kolejności rosnącej przechodzić przez numery przystanków z listy M, czyli przez wszystkie muzea oraz na końcu przez przystanek 1 (dworzec, w którym musimy zakończyć trasę). W każdym obrocie pętli wyznaczamy za pomocą algorytmu Dijkstry najkrótszą odległość od przystanku, w którym się obecnie znajdujemy, do przystanku, przy którym znajduje się muzeum (lub dworzec), które chcemy teraz odwiedzić i zwiększamy o wartość tej odległości zmienną *time* (ponieważ w naszym zadaniu czas przejazdu możemy interpretować jako odległość wyznaczoną przez algorytm Dijkstry). W ten sposób po wyjściu z pętli będziemy znali najkrótszy czas przejazdu przez wszystkie muzea w zadanej kolejności, zaczynając od dworca i kończąc na dworcu. Chcemy, aby nasz algorytm zwracał również trasę przejazdu, dlatego w for wykonujemy również następujące instrukcje: przypisujemy numer przystanku (tego, przez który przechodzimy w danym obrocie pętli) do zmiennej *bus_stop*, a obecny *bus_stop* dodajemy do listy *subroute* jako jej pierwszy element (*subroute* to zmienna pomocnicza). Następnie wchodzimy do pętli *while* i dodajemy w niej do listy *subroute* od tyłu wszystkie przystanki, przez które musieliśmy przejechać, aby dojechać od poprzedniego muzeum (lub dworca) do miejsca (muzeum lub dworca), do którego chcieliśmy dotrzeć. Po wyjściu z pętli *while* odwracamy kolejność przystanków w zmiennej *subroute* i dodajemy je do naszej trasy (zmiennej *route*). Pod koniec każdego obrotu pętli for, gdy odwiedziliśmy już odpowiedni przystanek, zmieniamy zmienną *current_stop* na jego numer. Po wyjściu z pętli zmienna *time* przechowuje czas najkrótszego przejścia od dworca do dworca, odwiedzając wszystkie muzea w odpowiedniej koljeności, a *route* przechowuje trasę tego przejścia jako listę numerów przystanków. Funkcja $ROUTE$ zwraca parę zmiennych *(time, route)*.

2.4 Złożoność obliczeniowa naszego programu

Analiza złożoności algorytmu **ROUTE**:

- Algorytm najpierw sortuje listę muzeów M o długości k, co ma złożoność $\mathcal{O}(k \log k)$.
- Następnie dla każdego muzeum wykonuje się algorytm Dijkstry, którego złożoność wynosi $\mathcal{O}(n^2)$, gdzie n to liczba wierzchołków grafu.
- Algorytm Dijkstry jest wywoływany $k+1$ razy (dla każdego muzeum i powrotu do punktu początkowego).
- Dla każdej trasy algorytm rekonstruuje ścieżkę o długości co najwyżej n, co ma złożoność liniową $\mathcal{O}(n)$.

Podsumowując, złożoność czasowa algorytmu **ROUTE** wynosi:

$$\mathcal{O}(k \log k) + (k+1) \cdot \mathcal{O}(n^2 + n),$$

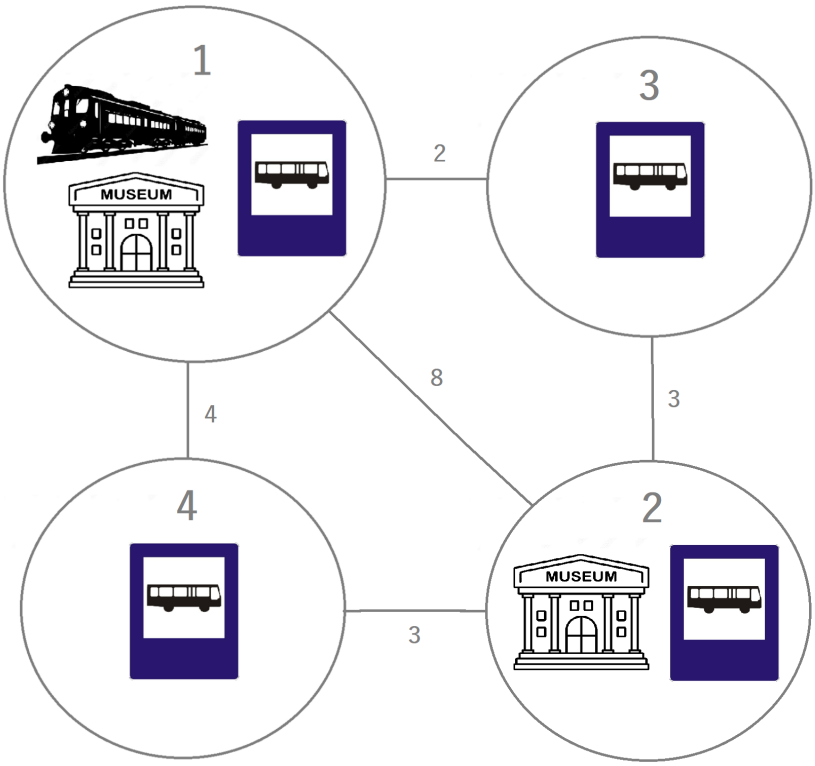
co w przybliżeniu daje $\mathcal{O}(kn^2)$.

3 Przykłady

3.1 Przykład 1

W pierwszym przykładzie mamy w Bitowicach liczbę muzeów równą 2 i liczbę przystanków równą 4. Przystanki połączone są 5 ulicami. Układ dróg i przystanków ukazuje rysunek 2.

Według naszego programu najbardziej optymalną droga zwiedzania muzeów to
$1 \to 3 \to 2 \to 3 \to 1$.
Czas tego przejazdu wynosi 10.



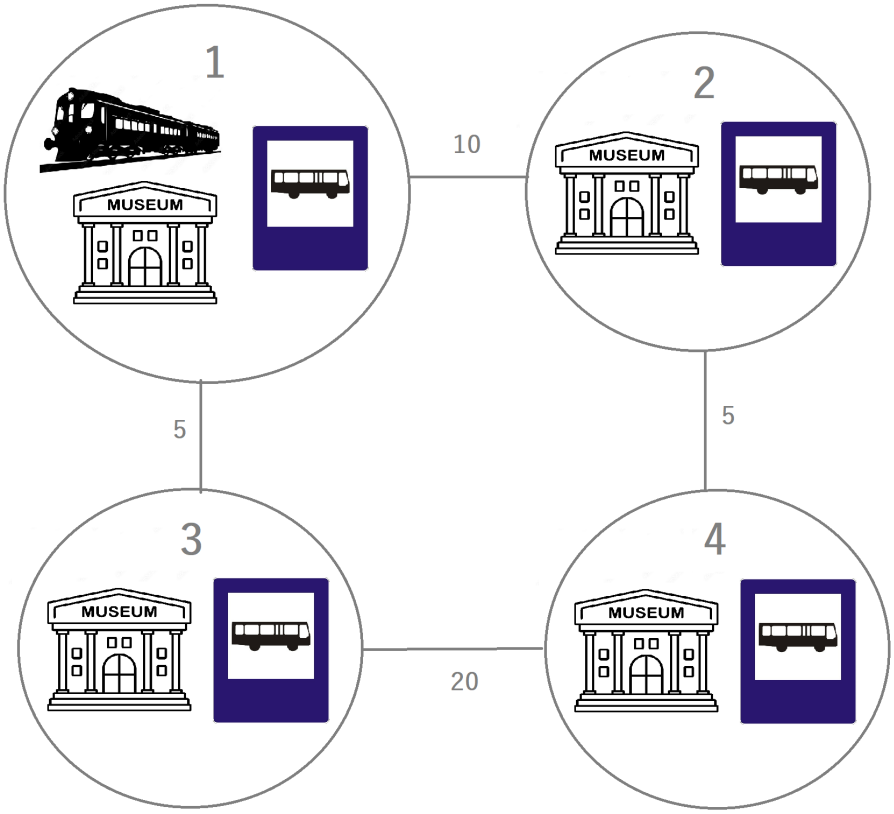
Rysunek 2:

Rzeczywiście, jeśli zwiedzamy muzea w kolejności rosnącej numerów przystanków, to zaczynając od razu w muzeum przy dworcu, najkrótsza droga dotarcia do muzeum przy przystanku 2 to droga przez przystanek 3. Najkrótszą drogą powrotną jest oczywiście ta sama droga, przez przystanek 3. Łatwo policzyć, że jest to faktycznie najkrótsza trasa, a czas przejazdu wynosi 10.

3.2 Przykład 2

W drugim przykładzie mamy 4 przystanki, połączone czterema drogami i przy każdym przystanku jest muzeum (rysunek 3).

Trasa wskazana przez nasz program to $1 \to 2 \to 1 \to 3 \to 4 \to 2 \to 1$.
Czas tego przejazdu wynosi 60.



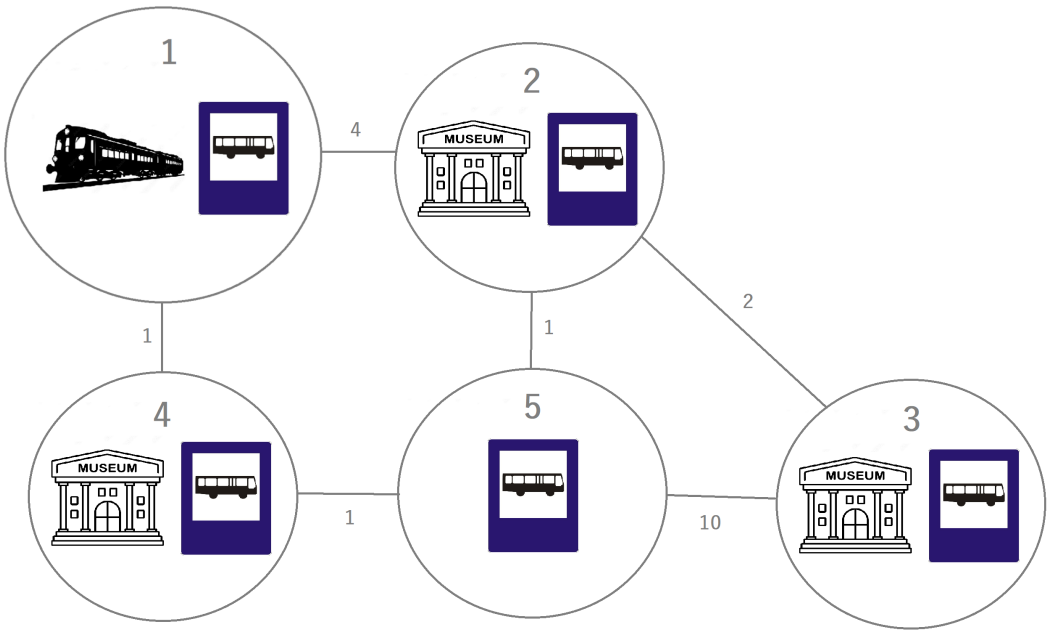
Rysunek 3:

Przeanalizujmy w jaki sposób przebiega trasa. Musimy odwiedzić muzea w kolejności 1, 2, 3, 4 znajdujących się przy nich przystanków. Startujemy z przystanku numer 1, a najkrótsza trasa do przystanku 2 to trasa bezpośrednia. Aby z przystanku 2 dotrzeć do przystanku 3, bardziej opłaca nam się wybrać trasę przez przystanek 1 (z czasem przejazdu 15) niż przez przystanek 4 (z czasem przejazdu 25). Z przystanku 3 do przystanku 4 można przejechać na dwa równie optymalne sposoby, które mają czas przejazdu 20. Nasz program w tym przypadku wybrał trasę bezpośrednią z 3 do 4. Powrót z przystanku 4 do przystanku 1 odbywa się przez przystanek 2, jest to najkrótsza trasa. Podliczając wszystkie wagi krawędzi, którymi przechodziliśmy, widzimy, że faktycznie czas przejazdu wynosi 60.

3.3 Przykład 3

W przykładzie 3 mamy graf składający się z 5 wierzchołków, połączonych 6 krawędziami. Przy trzech z tych przystanków znajdują się muzea (rysunek 4).

Trasa wyznaczona przez nasz program: $1 \to 4 \to 5 \to 2 \to 3 \to 2 \to 5 \to 4 \to 1$
Czas przejazdu: 10



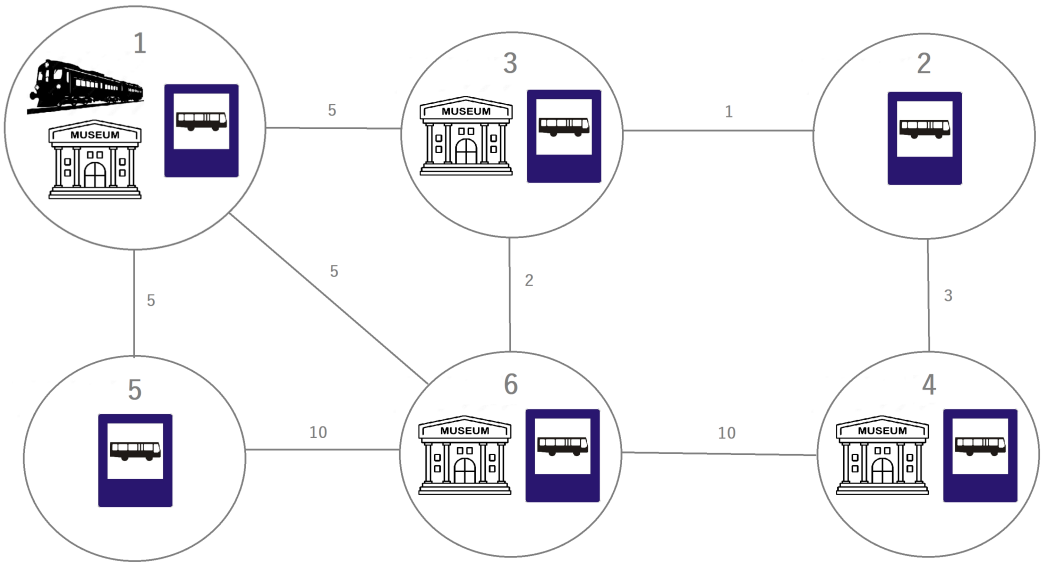
Rysunek 4:

Przeanalizujmy trasę przejazdu. Zaczynamy na przystanku 1 i chcemy udać się na przystanek 2, przy którym jest muzeum. Najkrótszą drogą jest droga przez przystanek 4 i 5 (z czasem przejazdu 3). Przejazd z przystanku 2 na 3 najszybciej odbywa się trasą bezpośrednią (czas przejazdu: 2). Następnie z przystanku 3 na przystanek 4 wracamy taką samą trasą, jaką jechaliśmy na początku: przez przystanek 2 i 5. Z przystanku 4 wracamy bezpośrednim połączeniem na przystanek 1. Czas takiego przejazdu faktycznie wynosi 10.

3.4 Przykład 4

W przykładzie 4 w Bitowicach mamy 6 przystanków, przy czym przy 4 z nich znajduje się muzeum. Przystanki połączone są 8 krawędziami (rysunek 5).

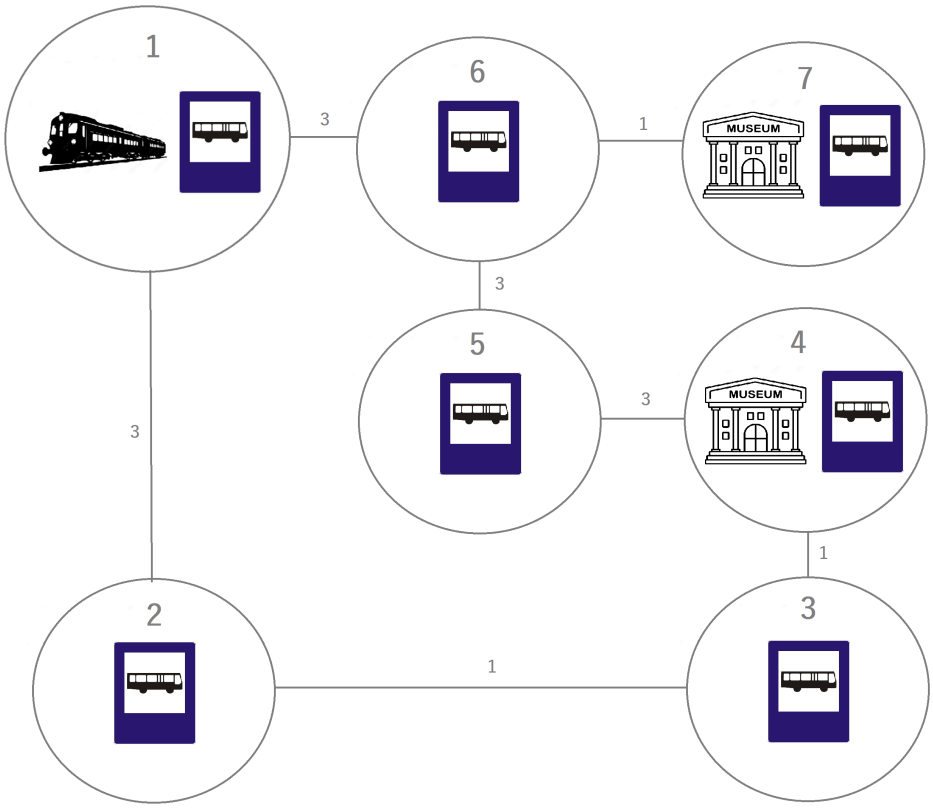
Trasa wyznaczona przez nasz program: $1 \to 3 \to 2 \to 4 \to 2 \to 3 \to 6 \to 1$
Czas przejazdu: 20



Rysunek 5:

Istotnie, najktrótszą drogą z przystanku 1 do przystanku 3 jest trasa bezpośrednia (z czasem przejazdu 5). Następnie chcąc przejechać z przystanku 3 na przystanek 4, najoptymalniej jest wybrać drogę przez przystanek 2 (z łącznym czasem przejazdu 4). Aby z przystanku 4 dostać się na przystanek 6, najlepiej jest wybrać drogę przez przystanek 2 i 3 (z łącznym czasem przejazdu 6). Najkrótszą drogą z przystanku 6 z powrotem na dworzec jest bezpośrednie połączenie z czasem przejazdu 5. Całkowity czas przejazdu faktycznie wynosi 20.

3.5 Przykład 5



Rysunek 6:

W ostatnim przykładzie mamy w Bitowicach 7 przystanków autobusowych, ale tylko 2 muzea. Przystanki połączone są 7 krawędziami (rysunek 6).

Trasa wyznaczona przez nasz program: $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 6 \to 1$
Czas przejazdu: 16

Łatwo zauważyć, że najkrótszą drogą z przystanku 1 do przystanku 4 jest faktycznie droga przez przystanek 2 i 3 (czas przejazdu: 5). Z przystanku 4 na przystanek 7 najszybciej jest dotrzeć przez przystanki 5 i 6 (czas przejazdu: 7). Z przystanku 7 dotrzemy szybko na dworzec kolejowy przejeżdżając przez przystanek 6 (czas przejazdu: 4). A zatem łączny czas przejazdu faktycznie wynosi 16.

4 Literatura

- Materiały z wykładów z przedmiotu *Algorytmy i Struktury Danych*, dr Anna Radzikowska