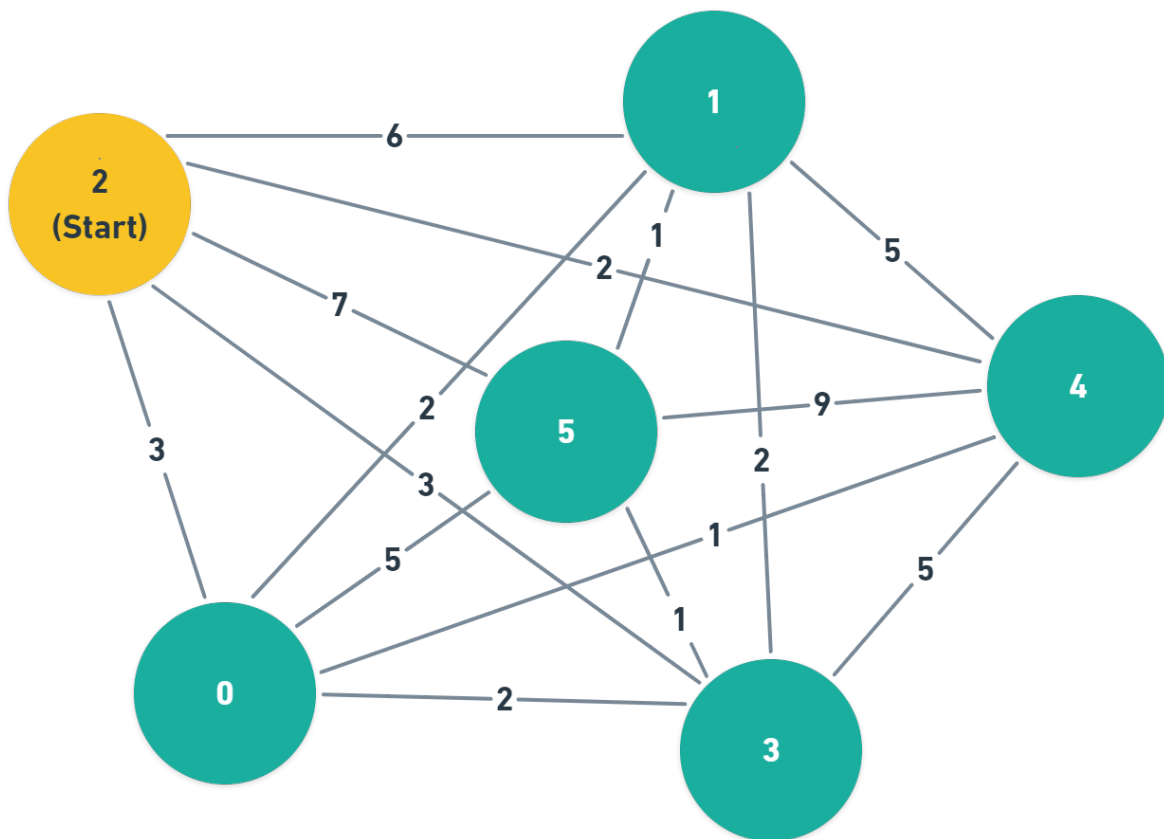


## Problem Komiwojażera

Problem polega na tym że, dane jest **N** miast, które komiwojażer ma odwiedzić, oraz odległość / cena podróży / czas podróży pomiędzy każdą parą miast. Celem jest znalezienie najkrótszej / najtańszej / najszybszej drogi łączącej wszystkie miasta, zaczynającej się i kończącej się w określonym punkcie.

Przykładowy graf, na którym pracowałem, jest on spójny, każdy węzeł ma ścieżkę do każdego węzła:



Na krawędziach podane są koszty przejścia z danego węzła do kolejnego. Przyjąłem za punkt startowy węzeł nr. 2.

Rozwiązanie problemu Komiwojażera przedstawiam w języku C#.

Wykorzystane algorytmy:

- Algorytm najbliższego sąsiada,
- Algorytm symulowanego wyżarzania,

Aby rozwiązać problem Komiwojażera stworzyłem na podstawie grafu odpowiednią macierz sąsiedztwa:

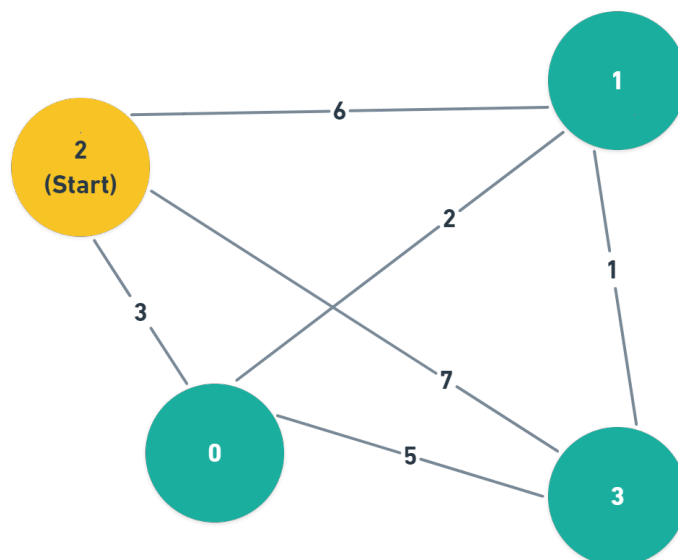
## MACIERZ SĄSIEDZTWA

	0	1	2	3	4	5
0	0	2	3	2	1	5
1	2	0	6	2	5	1
2	3	6	0	3	2	7
3	2	2	3	0	5	1
4	1	5	2	5	0	9
5	5	1	7	1	9	0

W Macierzy sąsiedztwa znajduje się koszt.

## Algorytm najbliższego sąsiada

W ramach przedstawienia działania Algorytmu wykorzystałem graf 4 węzłowy.



oraz jego macierz sąsiedztwa:

	0	1	2	3
0	0	2	3	5
1	2	0	6	1
2	3	6	0	7
3	5	1	7	0

Rozpatrujemy działanie algorytmu dla powyższego 4 węzłowego grafu spójnego:

```
startNode = 2
currentNode = startNode
minCost = inf
visited[currentNode] = true
Oznaczamy węzeł startowy jako Odwiedzony

Pętla do zmiany węzła
działa dopóki i < liczba węzłów - 1      i = 0

    Pętla do rozpatrywania sąsiadów
    działa dopóki i < liczba węzłów - 1

        dla i = 0 i currentNode = 2    graph[2,0] = 3 <-- na podstawie Macierzy Sąsiedztwa
        if(visited[i] == false && graph[currentNode,i] < minCost
        && graph[currentNode,i] != 0)
            minCost = graph[currentNode,i]    minCost = 3;
            tmp = i    tmp = 0
            dla i = 1 i i = 3 i currentNode = 2 koszt jest większy niż dla i = 0, dla i = 2, warunek visited[2] jest false,
            currentCost += graph[currentNode,tmp];    currentCost += 3
            currentNode = tmp    przesuwamy się do węzła 0
            visited[currentNode] = true;    oznaczamy węzeł 0 jako odwiedzony
            minCost = int.MaxValue;    zmieniamy minCost na inf
```

### Pętla do zmiany węzła

działa dopóki  $i < \text{liczba w\u0119z\u0142\u00f3w} - 1$

**i = 1**

#### Pętla do rozpatrywania sąsiad\u00f3w

działa dopóki  $i < \text{liczba w\u0119z\u0142\u00f3w} - 1$

dla  $i = 1$  i  $\text{currentNode} = 0$   $\text{graph}[0,1] = 2$  <-- na podstawie Macierzy S\u0105siedztwa

if( $\text{visited}[i] == \text{false}$  &&  $\text{graph}[\text{currentNode},i] < \text{minCost}$  <sup>inf</sup>  
&&  $\text{graph}[\text{currentNode},i] \neq 0$ )

$\text{minCost} = \text{graph}[\text{currentNode},i]$   $\text{minCost} = 2$ ;

$\text{tmp} = i$   $\text{tmp} = 1$

dla  $i = 2$  i  $i = 3$  i  $\text{currentNode} = 0$  koszt jest wi\u0119kszy ni\u017c dla  $i = 1$ , dla  $i = 0$ ,  $i = 2$  warunek  $\text{visited}[i]$  jest false,

$\text{currentCost} += \text{graph}[\text{currentNode},\text{tmp}]$ ;  <sup>$\text{currentCost} = 3 + 2 = 5$</sup>   
dodajemy koszt przej\u015bcia do nowego w\u0119z\u0142a

$\text{currentNode} = \text{tmp}$  <sup>przesuwamy si\u0119 do w\u0119z\u0142a 1</sup>

$\text{visited}[\text{currentNode}] = \text{true}$ ; <sup>oznaczamy w\u0119ze\u0142 1 jako odwiedzone</sup>

$\text{minCost} = \text{int.MaxValue}$ ; <sup>zmieniamy minCost na inf</sup>

### Pętla do zmiany węzła

działa dopóki  $i < \text{liczba w\u0119z\u0142\u00f3w} - 1$

**i = 2**

#### Pętla do rozpatrywania sąsiad\u00f3w

działa dopóki  $i < \text{liczba w\u0119z\u0142\u00f3w} - 1$

dla  $i = 3$  i  $\text{currentNode} = 1$   $\text{graph}[1,3] = 1$  <-- na podstawie Macierzy S\u0105siedztwa

if( $\text{visited}[i] == \text{false}$  &&  $\text{graph}[\text{currentNode},i] < \text{minCost}$  <sup>inf</sup>  
&&  $\text{graph}[\text{currentNode},i] \neq 0$ )

$\text{minCost} = \text{graph}[\text{currentNode},i]$   $\text{minCost} = 1$ ;

$\text{tmp} = i$   $\text{tmp} = 3$

dla  $i = 0$ ,  $i = 1$ ,  $i = 2$  warunek  $\text{visited}[i]$  jest false,

$\text{currentCost} += \text{graph}[\text{currentNode},\text{tmp}]$ ;  <sup>$\text{currentCost} = 3 + 2 + 1 = 6$</sup>   
dodajemy koszt przej\u015bcia do nowego w\u0119z\u0142a

$\text{currentNode} = \text{tmp}$  <sup>przesuwamy si\u0119 do w\u0119z\u0142a 3</sup>

$\text{visited}[\text{currentNode}] = \text{true}$ ; <sup>oznaczamy w\u0119ze\u0142 3 jako odwiedzone</sup>

$\text{minCost} = \text{int.MaxValue}$ ; <sup>zmieniamy minCost na inf</sup>

### Pętla do zmiany węzła

działa dopóki  $i < \text{liczba w\u0119z\u0142\u00f3w} - 1$

**i = 3**

wszystkie w\u0119z\u0142y s\u0105 odwiedzone

$\text{currentCost} = 3+2+1+0 = 6$

$\text{endCost} = \text{currentCost} +$   
 $\text{graph}[\text{currentNode},$   
 $\text{startNode}]$ ;

wychodzimy z p\u0119tli dodajemy koszt przej\u015bcia z w\u0119z\u0142a w którym  
jestesmy do w\u0119z\u0142a startowego.

$\text{endCost} = 6 + 7 = 13$

sciezka to  $2 \rightarrow 0 \rightarrow 1 \rightarrow 3 \rightarrow 2$

# Algorytm symulowanego wyżarzania

Rozpatrujemy działanie algorytmu dla 4 węzłowego grafu spójnego, przedstawionego na rysunku w zagadnieniu Algorytm Najbliższego Sąsiada.

## Annealing

```
List<int> CurrentNodeList = new List<int>();  
List<int> newNodeList = new List<int>();
```

listy w których przechowujemy aktualna i nowa drogę pomiędzy wierzchołkami

```
alpha = 0.999
```

przyjmuje alphe = 0.999

```
temperature = 400.0
```

temperatura początkowa

```
epsilon = 0.01
```

przyjmujemy epsilon = 0.01

```
distance = computeDistance(graph, newNodeList)
```

**metoda sumDistance** w tej metodzie sumujemy dystans do poszczególnych węzłów na podstawie listy węzłów

### Pętla do zmiany węzła

działa dopóki  $i < \text{liczba węzłów} - 1$

$i = 0$

```
distance += graph[NodeList[i], NodeList[i + 1]]
```

distance = 2 graph[0,1] = 2 (na podstawie macierzy sąsiedztwa)

$i = 1, 2, 3$

```
distance += graph[i, i + 1];
```

graph[i,i+1] (na podstawie macierzy sąsiedztwa)  
distance = 2 + 6 + 7 = 15

wychodzimy z pętli i dodajemy odległość między ostatnim węzłem a startowym.

```
distance += graph[NodeList[0], NodeList[NodeList.Count - 1]];
```

distance = 20 ponieważ  $\text{graph}[0,3] = 5$  (na podstawie macierzy sąsiedztwa)

```
zwracamy obliczony dystans
```

distance = 20

```
distance = 20;
```

### Pętla sprawdzająca temperature

działa dopóki  $\text{temperature} > \text{epsilon}$

```
permutationFunc(CurrentNodeList, out newNodeList);
```

wykonujemy permutacje za pomocą funkcji permutationFunc, przekazujemy aktualna listę wierzchołków oraz listę nowych które ulegną zmianie  
CurrentNodeList aktualnie wygląda tak: 0,1,2,3

### metoda permutationFunc

```
int i1 = (int)(rnd.Next(CurrentNodeList.Count));  
int i2 = (int)(rnd.Next(CurrentNodeList.Count));
```

losujemy dwa indeksy, które posłużą nam do permutacji  
np. :  $i1 = 0$ ,  $i2 = 1$

```
int aux = newNodeList[i1];  
newNodeList[i1] = newNodeList[i2];  
newNodeList[i2] = aux;
```

$i1 = 0$   $i2 = 1$   
 $\text{newNodeList}[0] = 0$ ,  $\text{newNodeList}[1] = 1$

poprzez wylosowane indeksy, zamieniamy węzły -> budujemy nową ścieżkę i zapisujemy ją w newNodeList

newNodeList wygląda tak: 1,0,2,3

```
delta = sumDistance(graph, newNodeList) - distance;
```

delta = -7

### jeśli delta < 0

podmieniamy aktualna listę na nowo utworzoną listę węzłów i obliczamy dystans ->  $\text{distance} = -7 + 20 = 13$

```
CurrentNodeList = newNodeList; distance = delta + distance;
```

### w innym przypadku

```
proba = rnd.NextDouble();
```

losujemy proba

```
if (proba < Math.Exp(-delta / temperature))
```

```
{  
    podmieniamy aktualna listę na nowo utworzoną listę węzłów
```

```
    CurrentNodeList = newNodeList;
```

```
    distance = delta + distance;
```

```
}
```

```
temperature *= alpha;
```

zmniejszamy temperature mnożąc przez alphe