AKADEMIA GÓRNICZO-HUTNICZA im. Stanisława Staszica w Krakowie

**Algorytmy grafowe – algorytmy**

**zachłanne dla zagadnienia komiwojażera**

Stanisław Olech - 412023

Automatyka i Robotyka

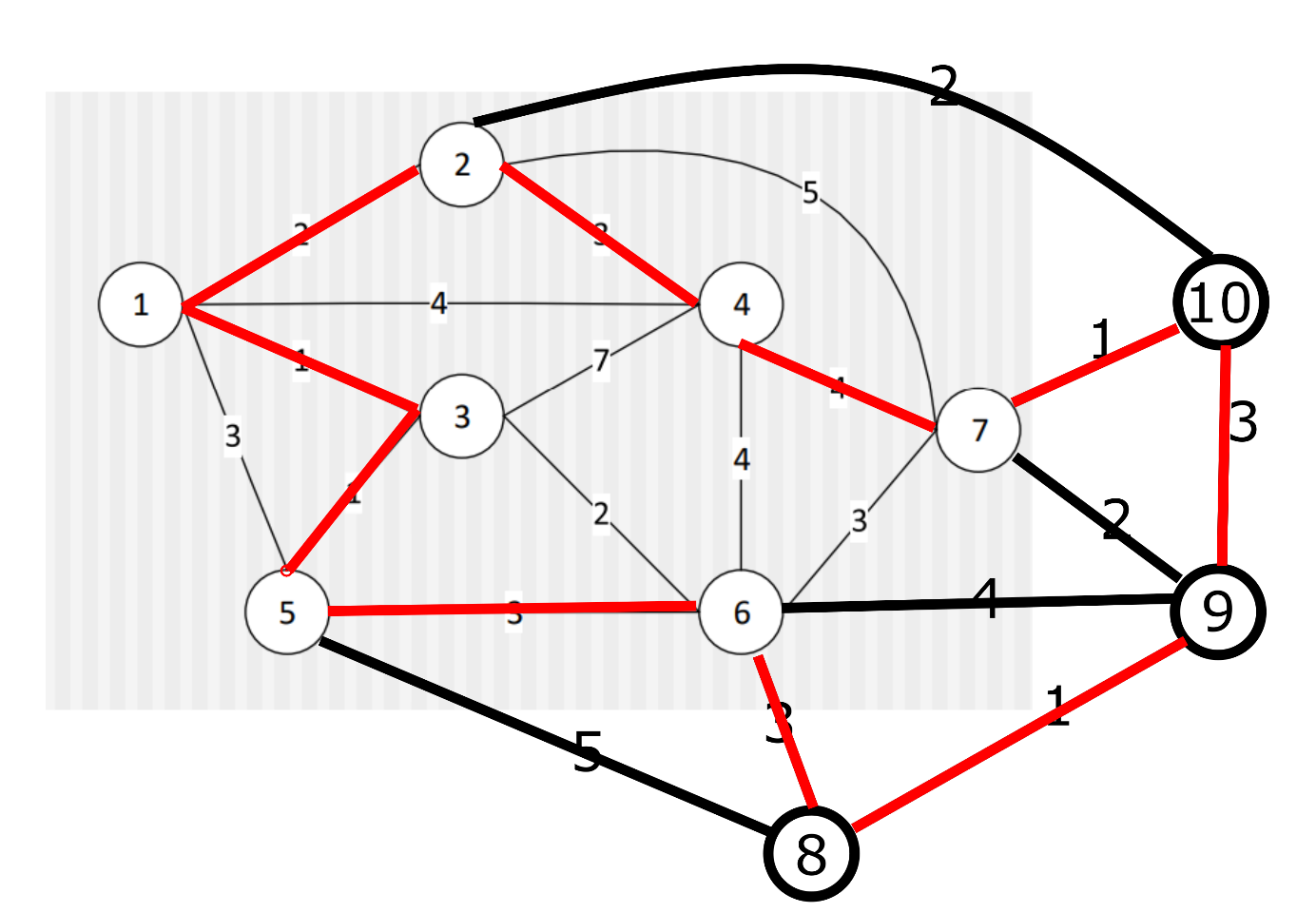
EAIiIB

**Zad. 1**

Kod. 1 Zaimplementowany przez mnie algorytm FARIN.

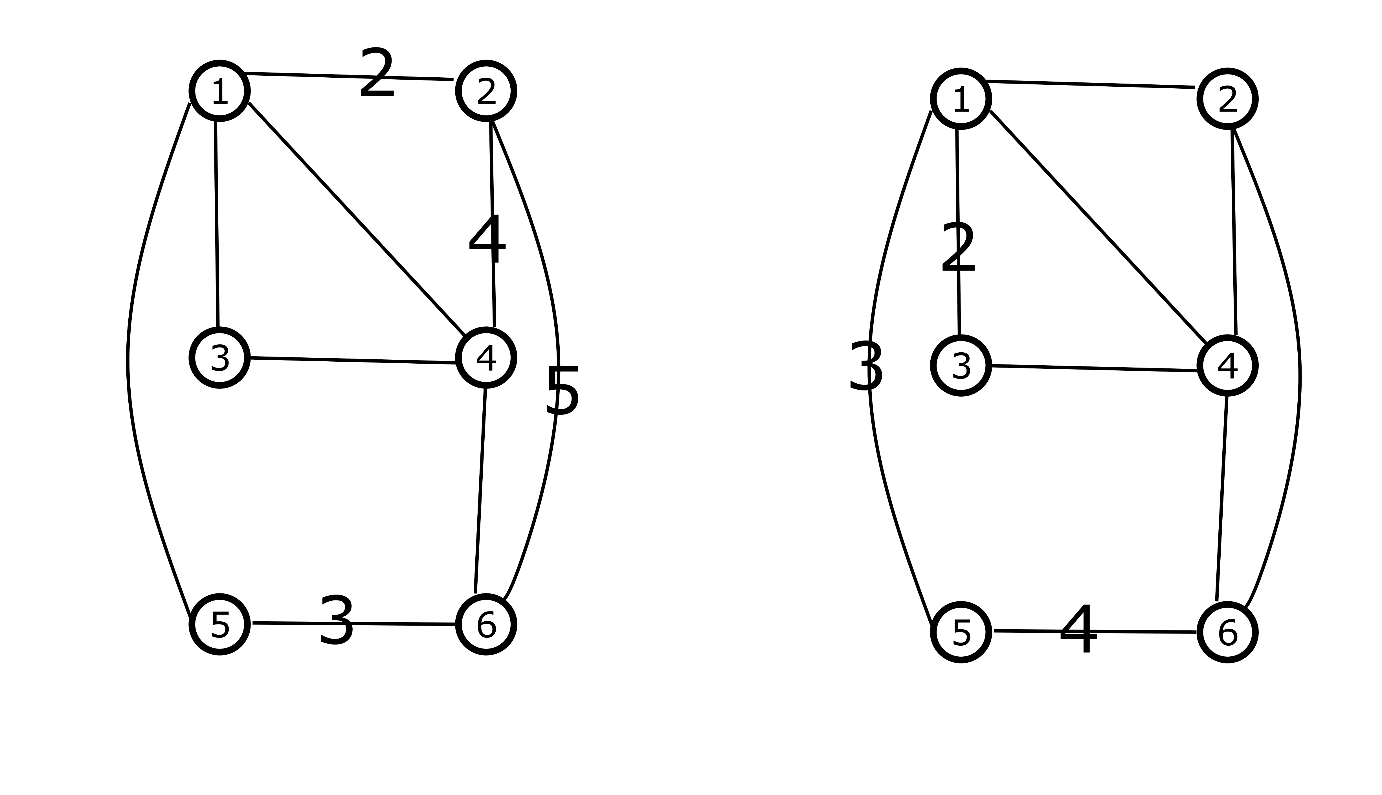
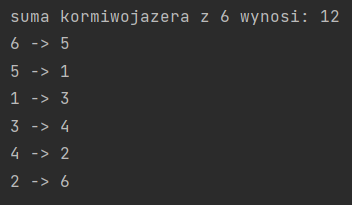
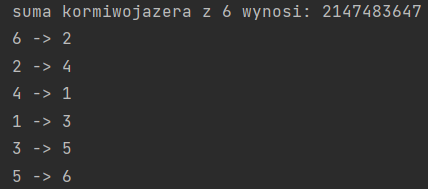
#include <iostream>  
#include <limits>  
#include <map>  
#include <list>  
#include <algorithm>  
  
int inf = std::numeric\_limits<int>::max();  
  
struct path{  
 int origin;  
 int destination;  
};  
  
template <size\_t *size*>  
std::tuple<std::list<path>, int> FARIN(int(&graph)[*size*][*size*], int s){  
  
 // Deklaracja zmiennych  
 std::list<path> A = {};  
 std::list<path> newA = {};  
 int short\_n = -1;  
 int short\_e = -1;  
 std::list<int> visited;  
 int length = 0;  
 int shortest = -1;  
  
 visited.push\_back(s);  
  
 // szukanie następnego wierzchołka  
 for (auto y : visited){  
 for (size\_t x = 0; x < *size*; x++){  
 if (std::find(std::begin(visited), std::end(visited), x) == std::end(visited) and graph[y][x] != inf and graph[y][x] > short\_e){  
 short\_e = graph[y][x];  
 short\_n = static\_cast<int>(x);  
 }  
 }  
 }  
  
 // dodwawanie wierzchołka  
 visited.push\_back(short\_n);  
 A.push\_back({s + 1, short\_n + 1});  
 A.push\_back({ short\_n + 1, s + 1,});  
  
  
  
 while (A.size() != *size* ){  
 short\_e = -1;  
 // szukanie następnego wierzchołka  
 for (auto y : visited){  
 for (size\_t x = 0; x < *size*; x++){  
 if (std::find(std::begin(visited), std::end(visited), x) == std::end(visited) and graph[y][x] != inf and graph[y][x] > short\_e){  
 short\_e = graph[y][x];  
 short\_n = static\_cast<int>(x);  
 }  
 }  
 }  
 visited.push\_back(short\_n);  
  
 // szukanie miejsca gdzie wpisać wierzchołek  
 short\_e = inf;  
 for(auto p = A.begin(); p != A.end(); p++){  
 length = 0;  
 for(auto a = A.begin(); a != p; a++){  
 if (length == inf or graph[a->origin - 1][a->destination - 1] == inf){  
 length = inf;  
 }  
 else{  
 length += graph[a->origin - 1][a->destination - 1];  
 }  
 }  
  
 auto x = p;  
 x ++;  
 for(auto b = x; b != A.end(); b++){  
 if (length == inf or graph[b->origin - 1][b->destination - 1] == inf){  
 length = inf;  
 }  
 else{  
 length += graph[b->origin - 1][b->destination - 1];  
 }  
 }  
  
 int sum = graph[p->origin - 1][short\_n] + graph[short\_n][p->destination - 1] + length;  
  
 if (graph[p->origin - 1][short\_n] == inf){  
 sum = inf;  
 }  
  
 if (graph[short\_n][p->destination - 1] == inf){  
 sum = inf;  
 }  
  
 if (length == inf){  
 sum = inf;  
 }  
  
 // elminacja inf-a  
 if (graph[p->origin - 1][p->destination - 1] == inf and sum != inf){  
 short\_e = sum;  
 shortest = p->origin - 1;  
 continue;  
 }  
  
 // normalny przypadek wybieramy najkrótszy  
 if (sum < short\_e){  
 short\_e = sum;  
 shortest = p->origin - 1;  
 continue;  
 }  
  
 // jak jakikolwiek zadziałał to nie szukamy w infach  
 if (short\_e != inf){  
 continue;  
 }  
  
 // dodawanie wierzchołka mimo nieskończonej ścieżki  
 if (graph[p->origin - 1][short\_n] != inf and graph[short\_n][p->destination - 1] != inf){  
 short\_e = sum;  
 shortest = p->origin - 1;  
 continue;  
 }  
  
 // dodawanie wierzchołka z jednym infem  
 if ((graph[p->origin - 1][short\_n] != inf or graph[short\_n][p->destination - 1] != inf) and sum == inf){  
 shortest = p->origin - 1;  
 }  
  
 }  
 length = 0;  
  
 // dodawanie wierzchołka do ścieżki  
 for(auto p : A){  
 if (p.origin - 1 == shortest){  
 path next = {short\_n + 1, p.destination};  
 path prev = {p.origin, short\_n + 1};  
 //A.erase(i);  
 newA.push\_back(prev);  
 newA.push\_back(next);  
 }  
 else{  
 newA.push\_back(p);  
 }  
 }  
  
 A = newA;  
 newA.clear();  
  
 }  
  
 // liczenie długośći  
 for(auto p : A){  
  
 if (length == inf or graph[p.origin - 1][p.destination - 1] == inf){  
 length = inf;  
 } else{  
 length += graph[p.origin - 1][p.destination - 1];  
 }  
 }  
  
 return {A, length};  
  
}  
  
int main() {  
 int graph[10][10] = {  
 {inf,2, 1, 4, 3, inf, inf, inf, inf, inf},  
 {2, inf,inf,3, inf, inf, 5, inf, inf, 2 },  
 {1, inf,inf,7, 1, 2, inf, inf, inf, inf},  
 {4, 3, 7, inf, inf, 4, 4, inf, inf, inf},  
 {3, inf,1, inf, inf, 3, inf, 5, inf, inf},  
 {inf,inf,2, 4, 3, inf, 3, 3, 4, inf},  
 {inf,5, inf,4, inf, 3, inf, inf, 2, 1 },  
 {inf,inf,inf,inf, 5, 3, inf, inf, 1, inf},  
 {inf,inf,inf,inf, inf, 4, 2, 1 , inf, 3 },  
 {inf,2, inf,inf, inf, inf, 1, inf, 3, inf},  
 };  
  
 int s = 0;  
 auto ans = FARIN(graph, s);  
 std::cout << "suma kormiwojazera z "<< s + 1 << " wynosi: "<< std::get<1>(ans) << std::endl;  
  
 for(auto ele : std::get<0>(ans)){  
 std::cout << ele.origin << " -> " << ele.destination << std::endl;  
 }  
}

kod źródłowy algorytm FARIN poszukiwania minimalnego cyklu Hamiltona.

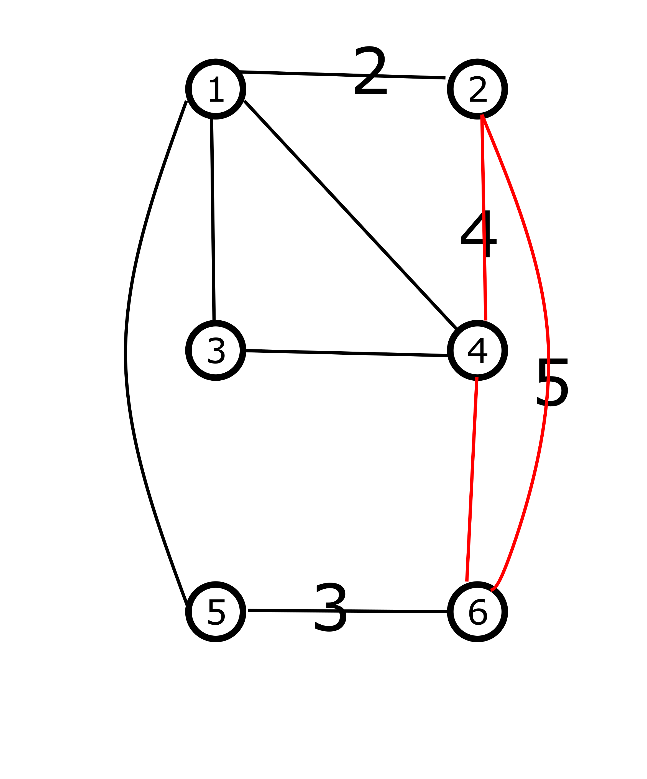


Rys. 1. Wynik algorytmu dla wierzchołków 1, 3, 4.

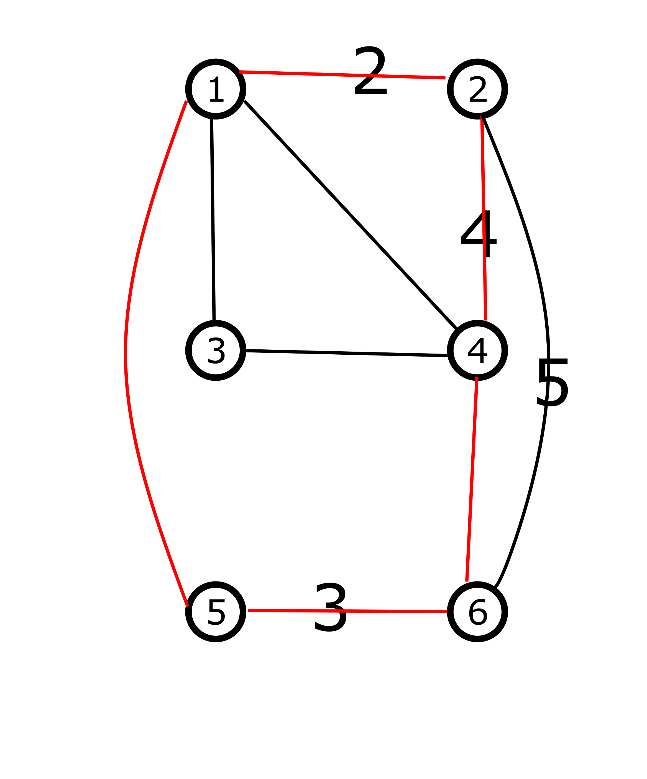
Niestety często wybiera tak wierzchołki że nie może ich połączyć bez nieskończoności.

Rys. 2. Analogiczny przypadek z wagami równymi 1 oraz większością 1 oraz jednym 2, 3,4.

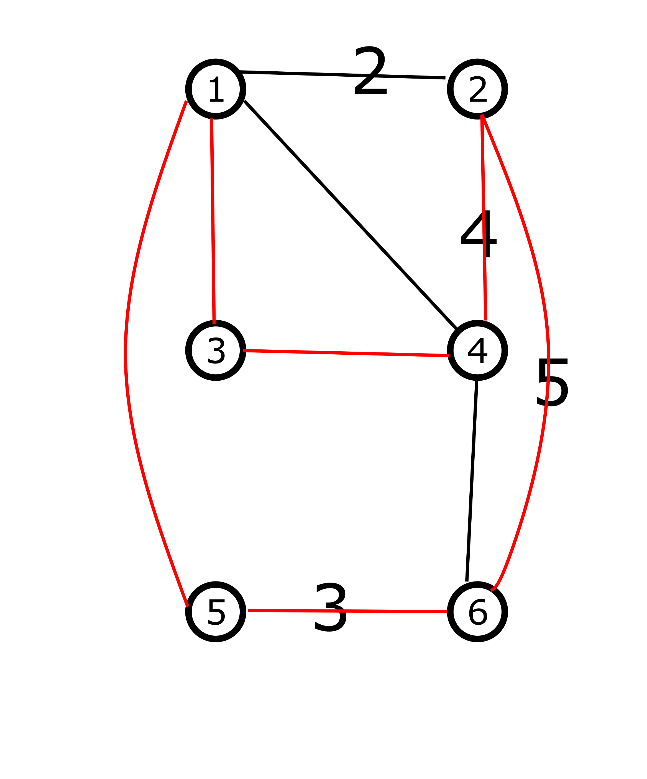


Rys. 3. Kolejny etap rozwiązywania lewego algorytmu. W tej sytuacji algorytm z przyłącza wierzchołek numer 5 i musi wpisać nieskończoność. Na szczęście dzięki temu że algorytm łączy 2 – 5 w następnym kroku może wpisać 1 bez nieskończoności.



Rys. 4. Kolejny etap rozwiązywania lewego algorytmu. W tej sytuacji algorytm musi dorzucić 3 między jakieś wierzchołki i zostaje zmuszony do wpisania nieskończoności.

Graf oczywiście ma cyk Hamiltona np. taki:



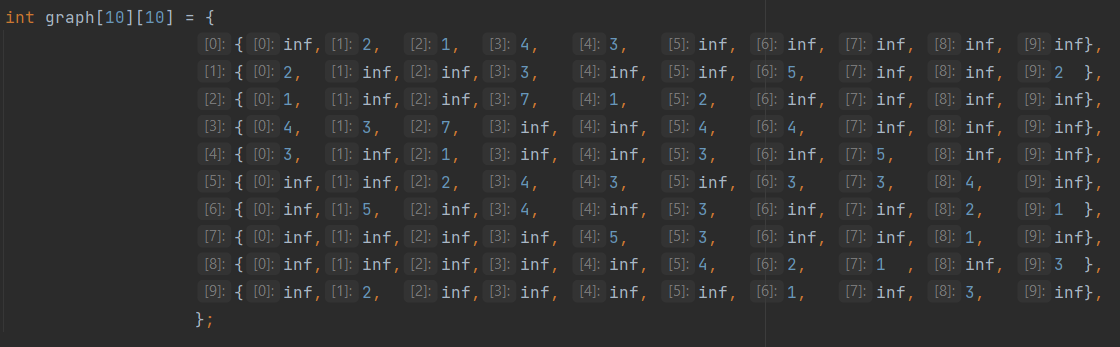
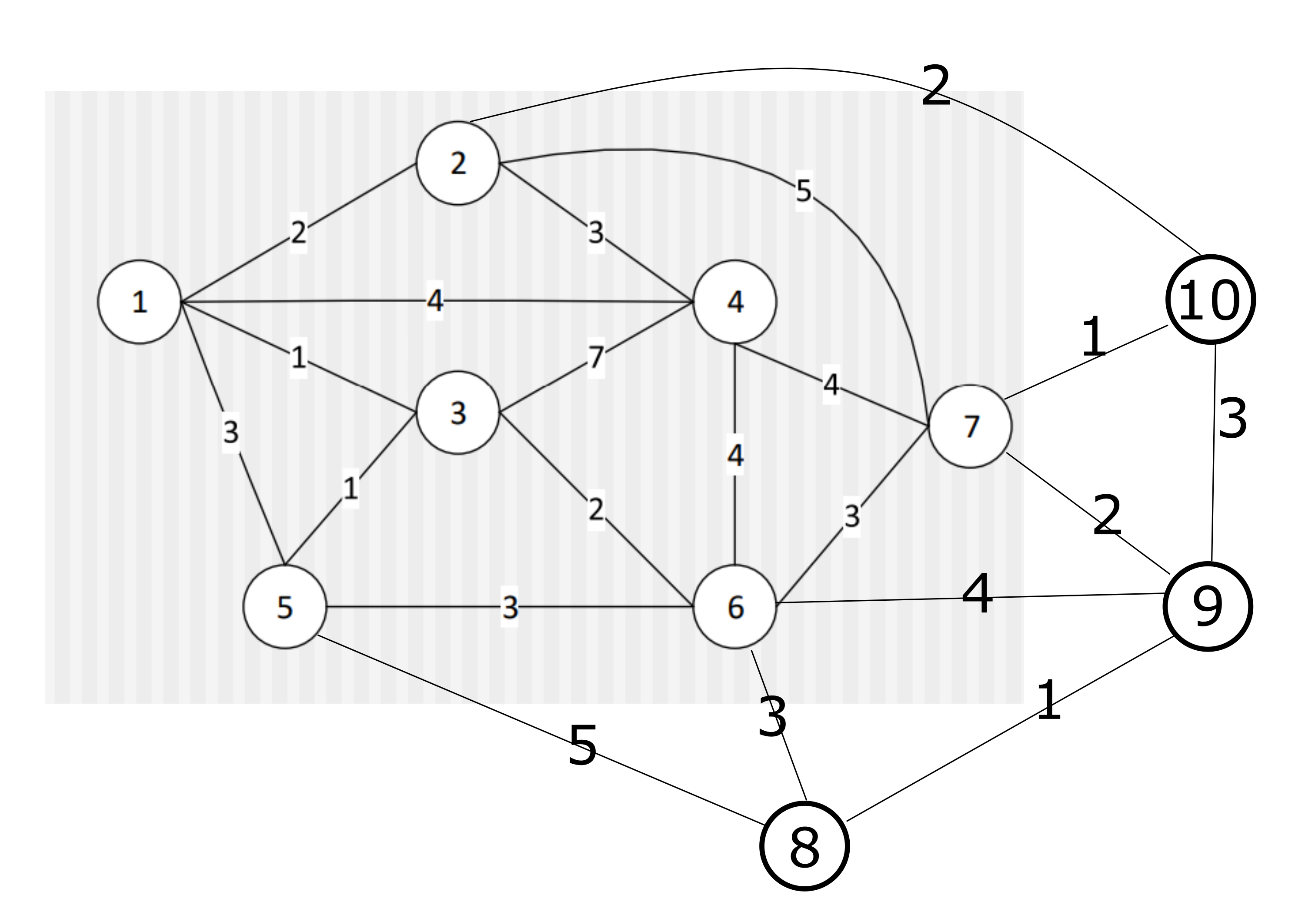
Rys. 5. Cykl Hamiltona w grafie.

Wynika stąd, że algorytm nie jest idealny i w niektórych sytuacjach może nie tylko nie znaleźć nie tylko optymalnej odpowiedzi a odpowiedzi w ogóle.

**Zad. 2**

Z punktu widzenia algorytmu może być istotne:

* czy graf jest spójny
* czy ma cykl Hamiltona
* wagi (jak wybierze zły wierzchołek może nie stworzyć trasy



Rys. 6 Graf oraz jego reprezentacja

odpowiedz na graf z rys. 1 wynosi 22 dla wierzchołków:

1. 22
2. INF
3. 22
4. 22
5. INF
6. INF
7. INF
8. 24
9. 26
10. 30:

**Zad. 3**

Złożoność mojej implementacji to ponieważ wybranie oraz znalezienie najkrótszej z możliwych ścieżek wymaga działań.

Algorytm najbliższego sąsiada oraz G\_TSP wybierają najbliższe wierzchołki.

NEARIN działa praktycznie tak samo jak FARIN ale wybiera wierzchołki w odwrotnej kolejności.

Algorytm 2-opt próbuje tak poukładać wierzchołki by krawędzie się nie przecinały. Robi to przez wybieranie 2 krawędzi i zamieniania ich połączeń. Analogicznie wygląda 3 opt i 4 opt.

Algorytm Christofidesa działa na zasadzie tworzenia minimalnego drzewa rozpinającego ten graf, który przerabiamy w muli graf i wyznaczamy cykl Eulerowski. Pomijamy powtarzające się wierzchołki i mamy cykl Hamiltona.

**Wnioski**

Zadanie sprawiło mi duże trudności. Próbowałem poprawić algorytm, żeby nie zwracał ciągle nieskończoności. Dopiero na mniejszym przykładzie zobaczyłem, że zgodnie z moim rozumieniem algorytmu na podstawie prezentacji z wykładów, w niektórych sytuacjach będzie musiał zwrócić nieskończoność.