AKADEMIA GÓRNICZO-HUTNICZA im. Stanisława Staszica w Krakowie

**Programowanie sieciowe – algorytmy CPM, PERT**

Stanisław Olech - 412023

Automatyka i Robotyka

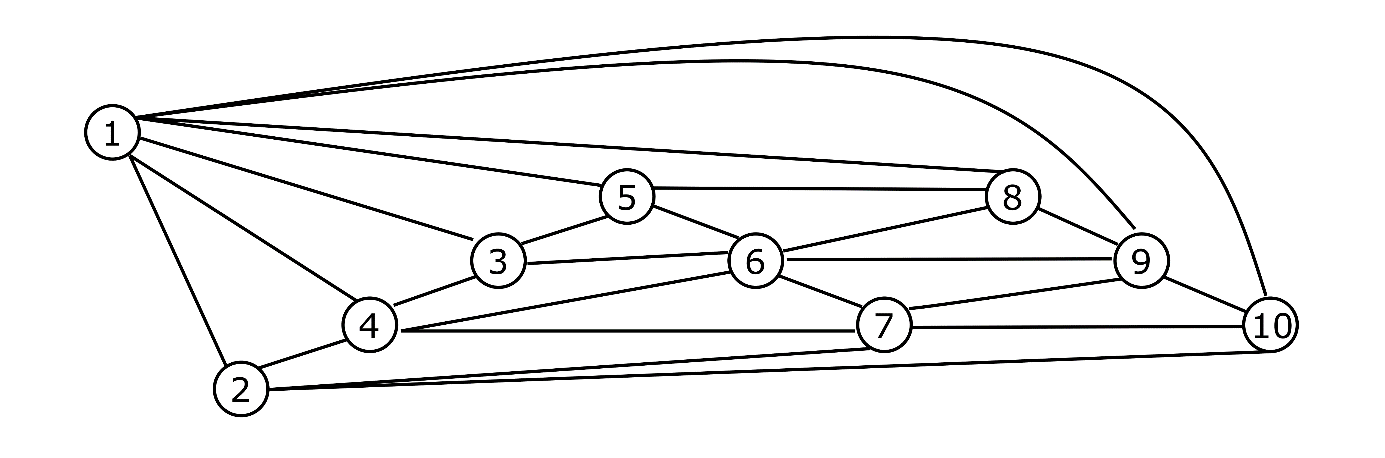
EAIiIB

**Zad. 1**

Kod. 1 Zaimplementowany przez mnie algorytmu

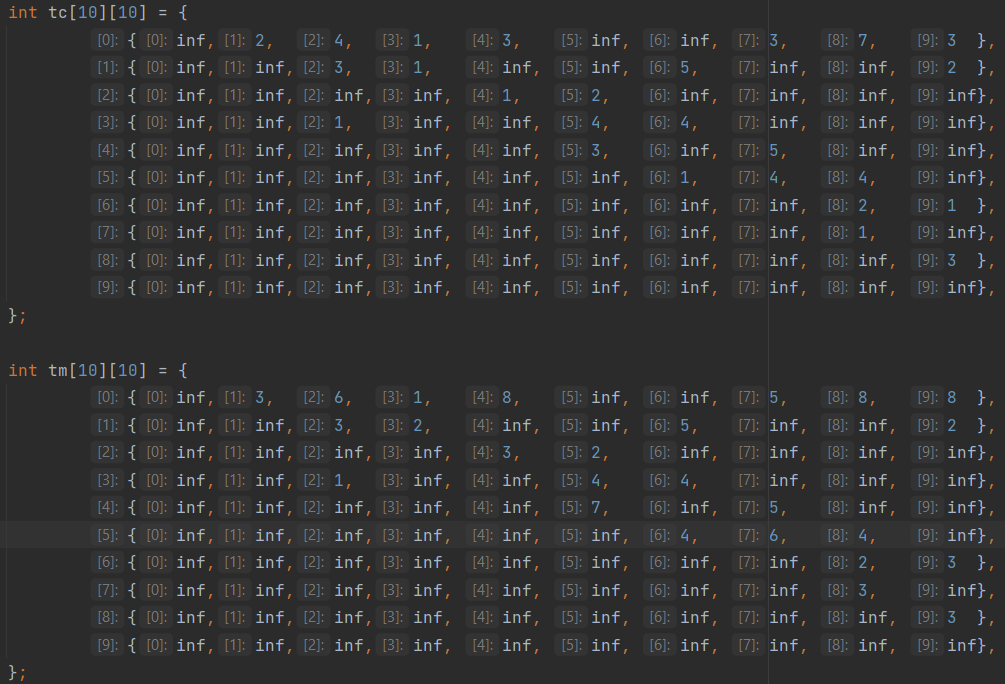
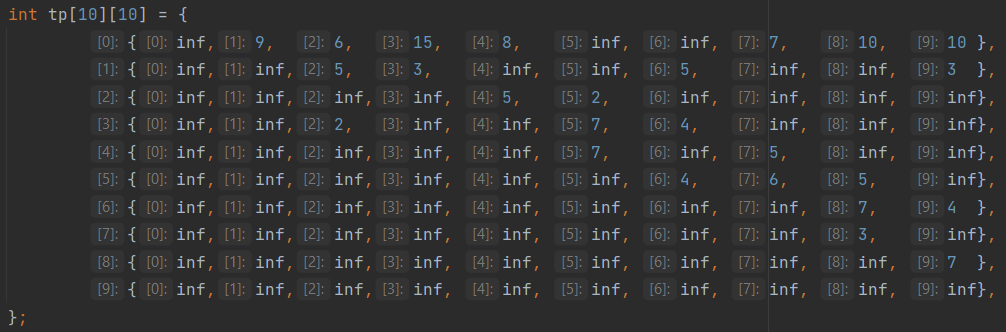
#include <iostream>  
#include <limits>  
#include <map>  
#include <list>  
#include <algorithm>  
#include <cmath>  
  
int inf = std::numeric\_limits<int>::max();  
float float\_inf = std::numeric\_limits<float>::max();  
  
struct path{  
 int origin;  
 int destination;  
};  
  
struct re\_num{  
 int org\_num;  
 int new\_num;  
};  
  
template <size\_t *size*>  
void array(int(&tc)[*size*][*size*], int(&tm)[*size*][*size*], int(&tp)[*size*][*size*], float(&tx)[*size*][*size*]){  
 for (size\_t x = 0; x < *size*; x++) {  
 for (size\_t y = 0; y < *size*; y++) {  
 if (tc[x][y] != inf){  
 tx[x][y] = 1.29 \* pow(pow(tp[x][y] - tc[x][y], 2) / 36, 0.5) + (float(tc[x][y]) / 6 + 4.0 / 6 \* float(tm[x][y]) + float(tp[x][y]) / 6);  
 } else{  
 tx[x][y] = float\_inf;  
 }  
 }  
 }  
}  
  
template <size\_t *size*>  
std::tuple<std::list<path>, float> PERT(float(&tc)[*size*][*size*]){  
  
 // Deklaracja zmiennych  
 std::list<path> path = {};  
 std::list<re\_num> new\_name = {};  
 std::list<size\_t> nodes = {};  
 float tw\_x[*size*];  
 float tp\_x[*size*];  
 float ans;  
  
  
  
 // Przenumerowanie  
 int i = 0;  
 while (nodes.size() != *size*) {  
 for (size\_t x = 0; x < *size*; x++) {  
 if(std::find(nodes.begin(), nodes.end(), x) != nodes.end()){  
 continue;  
 }  
  
 bool flag = true;  
  
  
 for (size\_t y = 0; y < *size*; y++) {  
 if(std::find(nodes.begin(), nodes.end(), y) != nodes.end()){  
 continue;  
 }  
 if (tc[y][x] != float\_inf and y != x) {  
 flag = false;  
 }  
 }  
  
 if (flag) {  
 new\_name.push\_back({static\_cast<int>(x), i});  
 nodes.push\_back(x);  
 i++;  
 }  
 }  
 }  
  
 for(auto& x :new\_name){  
 float sum = 0;  
 for (size\_t y = 0; y < *size*; y++) {  
 if (tc[y][x.org\_num] != float\_inf and tc[y][x.org\_num] + tw\_x[y] > sum){  
 sum = tc[y][x.org\_num] + tw\_x[y];  
 }  
 }  
 tw\_x[x.org\_num] = sum;  
 tp\_x[x.org\_num] = sum;  
 }  
  
 for(auto x = new\_name.rbegin(); x != new\_name.rend(); x++){  
 re\_num ele = x.operator\*();  
 if (ele.new\_num == *size* -1){continue;}  
  
 float sum = float\_inf;  
 for (size\_t y = 0; y < *size*; y++) {  
 if (tc[ele.org\_num][y] != float\_inf and tp\_x[y] - tc[ele.org\_num][y] < sum){  
 sum = tp\_x[y] - tc[ele.org\_num][y];  
 }  
 }  
 tp\_x[ele.org\_num] = sum;  
 }  
  
 int last = 0;  
 bool flag = false;  
 for(auto& x :new\_name){  
 if (!flag){  
 last = x.org\_num;  
 flag = true;  
 continue;  
 }  
 if(std::abs(tw\_x[x.org\_num] - tp\_x[x.org\_num]) < 0.0001){  
 path.push\_back({last, x.org\_num});  
 last = x.org\_num;  
 }  
 ans = tw\_x[x.org\_num];  
 }  
  
 return {path, ans};  
  
}  
  
template <size\_t *size*>  
std::tuple<std::list<path>, float> PERT(int(&tc)[*size*][*size*]){  
 float tx[*size*][*size*];  
  
 for (size\_t x = 0; x < *size*; x++) {  
 for (size\_t y = 0; y < *size*; y++) {  
 if (tc[x][y] != inf){  
 tx[x][y] = float(tc[x][y]);  
 } else{  
 tx[x][y] = float\_inf;  
 }  
 }  
 }  
 return (PERT(tx));  
}  
  
template <size\_t *size*, typename type>  
std::list<size\_t> order(type(&tx)[*size*][*size*], const std::list<path>& cri\_path){  
  
 std::list<size\_t> nodes = {};  
  
 for(auto& ele: cri\_path){  
 nodes.push\_back(ele.origin);  
 nodes.push\_back(std::numeric\_limits<size\_t>::max());  
 for (size\_t x = 0; x < *size*; x++) {  
 if (std::find(nodes.begin(), nodes.end(), x) != nodes.end() or  
 x == ele.destination) {  
 continue;  
 }  
  
 bool flag = true;  
  
  
 for (size\_t y = 0; y < *size*; y++) {  
 if (std::find(nodes.begin(), nodes.end(), y) != nodes.end()) {  
 continue;  
 }  
 if (tx[y][x] != float\_inf and y != x) {  
 flag = false;  
 }  
 }  
 if (flag) {  
 nodes.push\_back(x);  
 }  
 }  
 }  
 for (int x = 0; x != *size*; x++){  
 if (std::find(nodes.begin(), nodes.end(), x) == nodes.end()) {  
 nodes.push\_back(x);  
 }  
 }  
  
 return nodes;  
}  
  
int main() {  
 int tc[10][10] = {  
 {inf,2, 4, 1, 3, inf, inf, 3, 7, 3 },  
 {inf,inf,3, 1, inf, inf, 5, inf, inf, 2 },  
 {inf,inf,inf,inf, 1, 2, inf, inf, inf, inf},  
 {inf,inf,1, inf, inf, 4, 4, inf, inf, inf},  
 {inf,inf,inf,inf, inf, 3, inf, 5, inf, inf},  
 {inf,inf,inf,inf, inf, inf, 1, 4, 4, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, 2, 1 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, 1, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, 3 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, inf},  
 };  
  
 int tm[10][10] = {  
 {inf,3, 6, 1, 8, inf, inf, 5, 8, 8 },  
 {inf,inf,3, 2, inf, inf, 5, inf, inf, 2 },  
 {inf,inf,inf,inf, 3, 2, inf, inf, inf, inf},  
 {inf,inf,1, inf, inf, 4, 4, inf, inf, inf},  
 {inf,inf,inf,inf, inf, 7, inf, 5, inf, inf},  
 {inf,inf,inf,inf, inf, inf, 4, 6, 4, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, 2, 3 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, 3, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, 3 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, inf},  
 };  
  
 int tp[10][10] = {  
 {inf,9, 6, 15, 8, inf, inf, 7, 10, 10 },  
 {inf,inf,5, 3, inf, inf, 5, inf, inf, 3 },  
 {inf,inf,inf,inf, 5, 2, inf, inf, inf, inf},  
 {inf,inf,2, inf, inf, 7, 4, inf, inf, inf},  
 {inf,inf,inf,inf, inf, 7, inf, 5, inf, inf},  
 {inf,inf,inf,inf, inf, inf, 4, 6, 5, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, 7, 4 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, 3, inf},  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, 7 },  
 {inf,inf,inf,inf, inf, inf, inf, inf, inf, inf},  
 };  
  
 float tx[10][10];  
 array(tc, tm, tp, tx);  
  
 auto ans1 = PERT(tc);  
 std::cout << "minimalna dlugosc: "<< std::get<1>(ans1) << std::endl;  
  
 for(auto ele : std::get<0>(ans1)){  
 std::cout << ele.origin + 1 << " -> " << ele.destination + 1 << std::endl;  
 }  
 std::cout << std::endl;  
  
  
 auto ans2 = PERT(tm);  
 std::cout << "przewidywana dlugosc: "<< std::get<1>(ans2) << std::endl;  
  
 for(auto ele : std::get<0>(ans2)){  
 std::cout << ele.origin + 1 << " -> " << ele.destination + 1 << std::endl;  
 }  
 std::cout << std::endl;  
  
  
 auto ans3 = PERT(tx);  
 std::cout << "z prawdopodobienstwem 90% nie przekroczy: "<< std::get<1>(ans3) << std::endl;  
  
 for(auto ele : std::get<0>(ans3)){  
 std::cout << ele.origin + 1 << " -> " << ele.destination + 1 << std::endl;  
 }  
 std::cout << std::endl;  
  
 auto ansprim = order(tx, std::get<0>(ans3));  
 for(auto ele : ansprim){  
 if (ele + 1 == 0){  
 std::cout << "---------" << std::endl;  
 continue;  
 }  
 std::cout << ele + 1 << std::endl;  
 }  
 std::cout << std::endl;  
  
  
  
 auto ans4 = PERT(tp);  
 std::cout << "maksymalna dlugosc: "<< std::get<1>(ans4) << std::endl;  
  
 for(auto ele : std::get<0>(ans4)){  
 std::cout << ele.origin + 1 << " -> " << ele.destination + 1 << std::endl;  
 }  
}

kod źródłowy metody PERT.

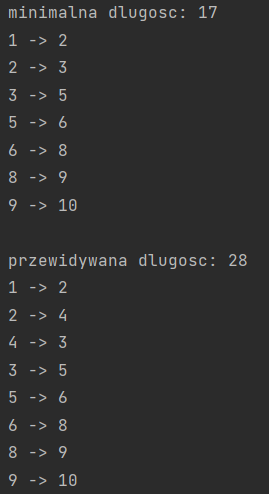
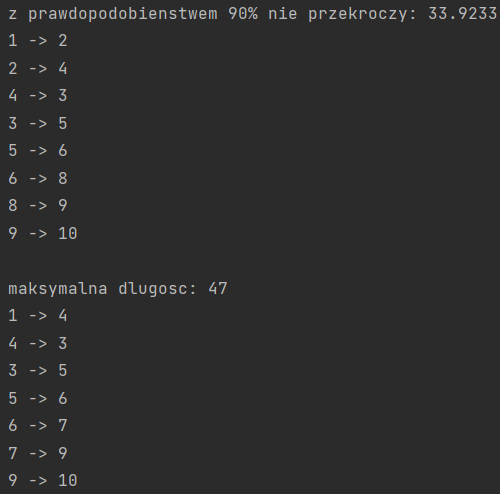


Rys. 1 Graf z 10 wierzchołkami i 20 krawędziami. Oczywiście wbrew temu co rysunek pokazuje jest to graf skierowany i możemy się poruszać jedynie w prawo.

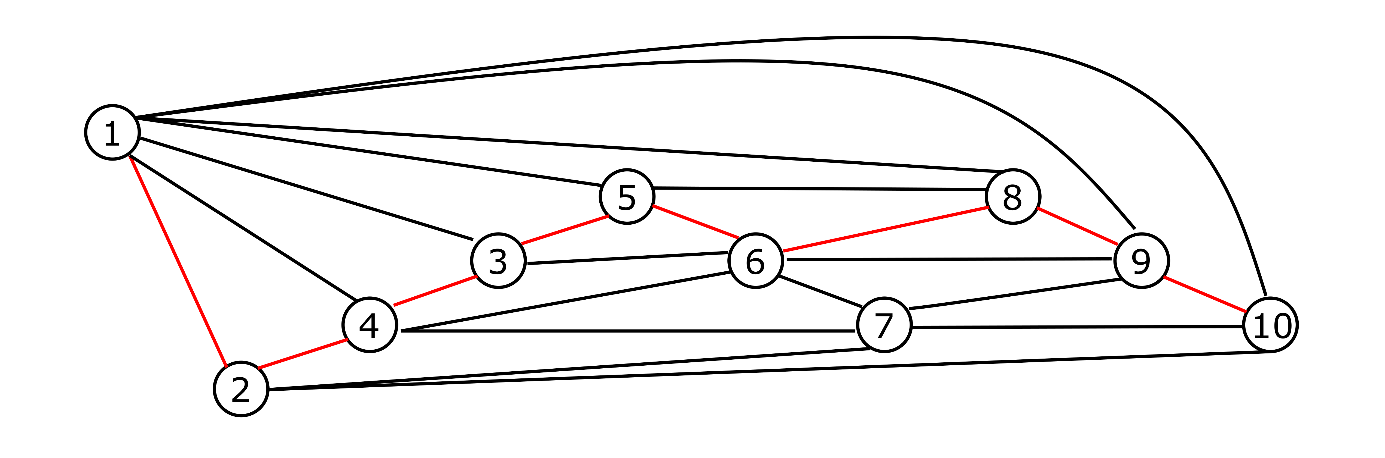
Jako reprezentację przyjąłem tabelę z wartościami optymalnymi, przewidywanymi i pesymistycznymi.

Rys. 2 Reprezentacja.

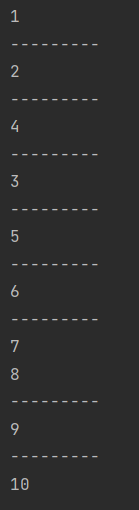
**** ****

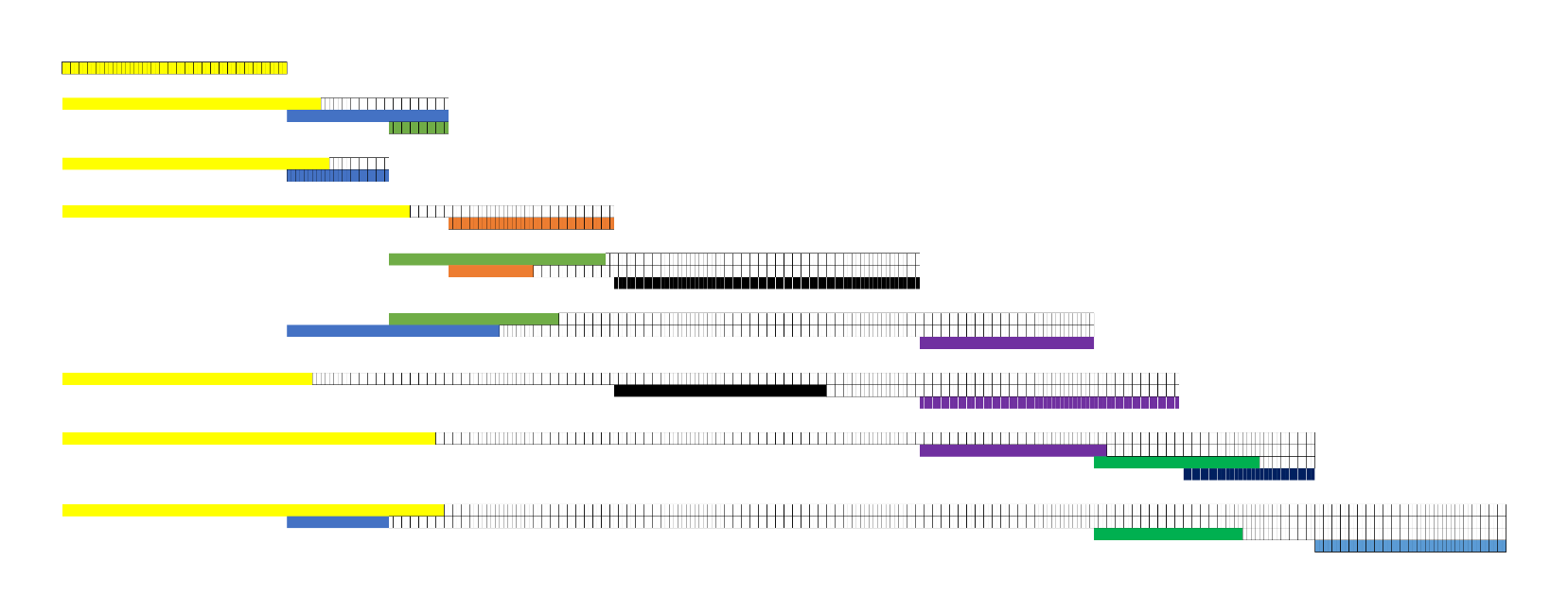
Rys. 3 Działanie algorytmu.

****

Rys. 4 Ścieżka krytyczna dla czasu liczonego z 90% pewnością

**Zad. 2**



Rys. 5 Kolejność wykonywania działań

Rys. 5 wykres Gantt’a. Kolorowe zakreskowane to ścieżki krytyczne a zakreskowane białe to zapas

**Wnioski**

Zadanie sprawiło mi trudnośći, w reprezentacji wykresu Gantt’a. oraz w samej implementacji .Jednak nauczyłem się podczas niego dużo o szablonach klas w c++ oraz różnych metodach zwracania tablicę przez funkcję.