Projekt B

Spis treści

Opis Projektu

Początkowa droga (permutacja) wylosowana zostaje algorytmem Fishera–Yatesa. W głównej pętli algorytmu (funkcja annealingFindPath) ustawione są dwa warunki stopu:

- jeśli temperatura jest mniejsza lub równa 0.1f (temperature > 0.1f),
- jeśli przez 10000 obniżeń temperatury nie dokonały się żadne zmiany drogi
  (int threshold = 10000), (withoutChangeCounter < threshold).

Następnie nowa droga może być wylosowana na dwa sposoby, zależnie od temperatury.
W funkcji getNewPath wyliczane jest prawdopodobieństwo użycia metody zamiany dwóch wylosowanych wierzchołków w permutacji. Im większa temperatura tym większa szansa na zamianę (float p = exp(-1/temperature)), w przeciwnym wypadku wybrana zostanie metoda odwrócenia wierzchołków w permutacji pomiędzy wylosowanymi.

Porównanie wydajności metod

Dane wejściowe:

- Wprowadzone współrzędne:

  334,195,404,193,240,235,327,231,91,216,374,177,372,214,275,307,555,218,481,448,201,147,268,392,198,456,170,289,426,378,385,246,325,359,

- Wprowadzona temperatura początkowa:

  5

- Wprowadzony mnożnik q:

  0.99999

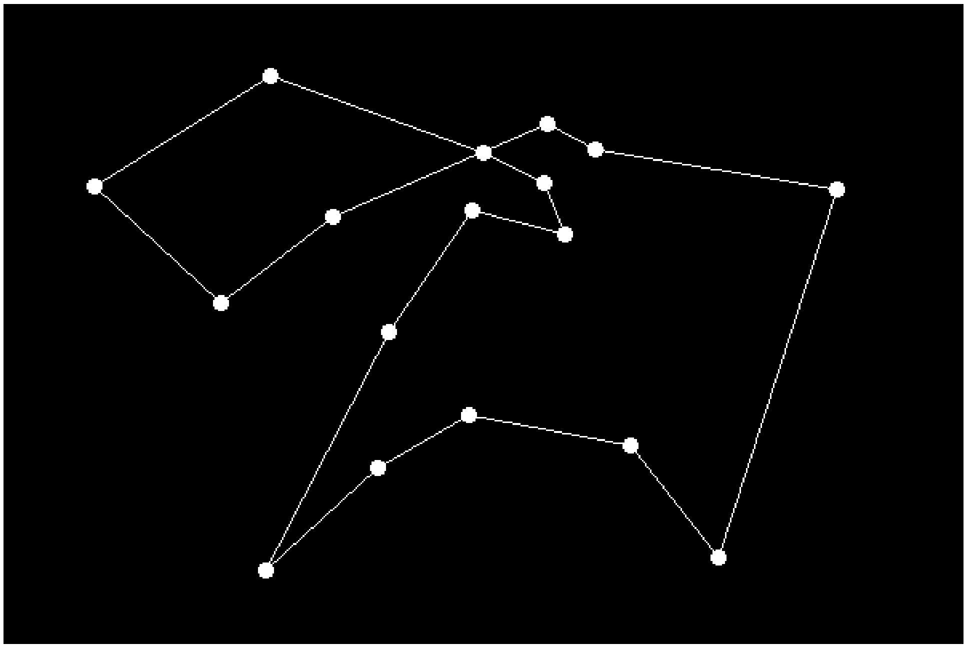
Załączone obrazki zostały wygenerowane przez program załączony w mailu.

Zamiana

Po zmianie w funkcji getNewPath linijki bool swap = (p > r) na bool swap = true, używana będzie tylko metoda zamiany.

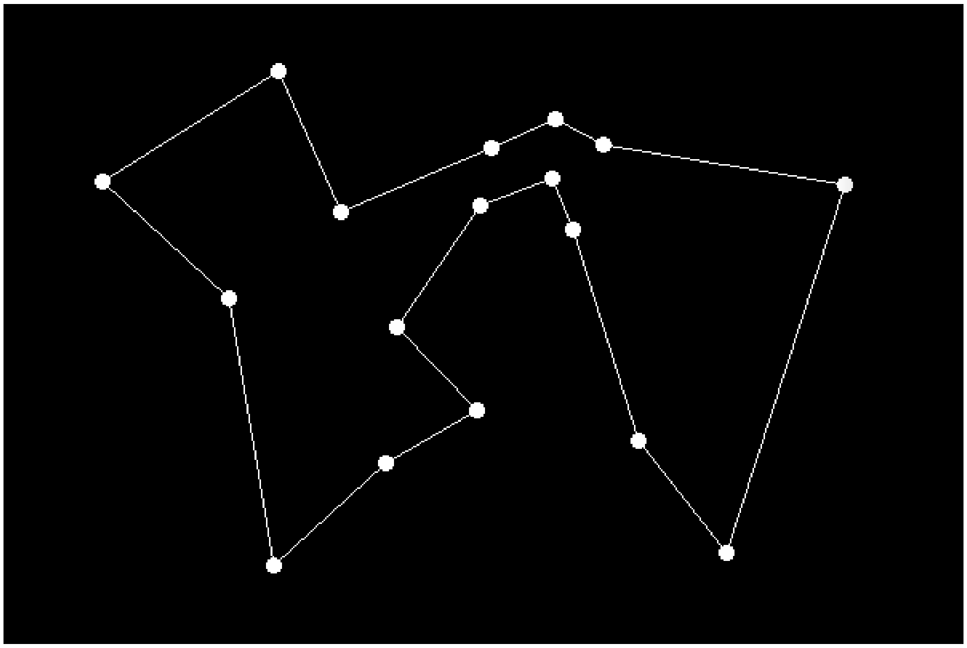
Sciezka: **8, 1, 5, 2, 13, 4, 10, 0, 6, 15, 3, 7, 12, 11, 16, 14, 9,**

Odleglosc: **1791.14**



Sciezka: **13, 12, 11, 16, 7, 3, 6, 15, 14, 9, 8, 1, 5, 0, 2, 10, 4,**

Odleglosc: **1712.51**

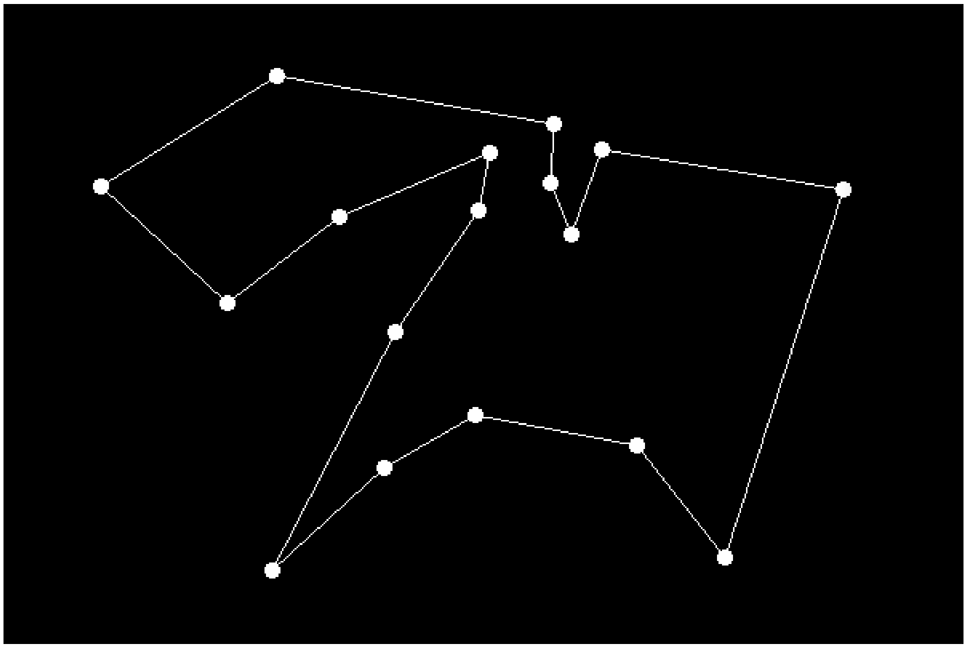


Odwrócenie

Po zmianie w funkcji getNewPath linijki bool swap = (p > r) na bool swap = false, używana będzie tylko metoda odwracająca.

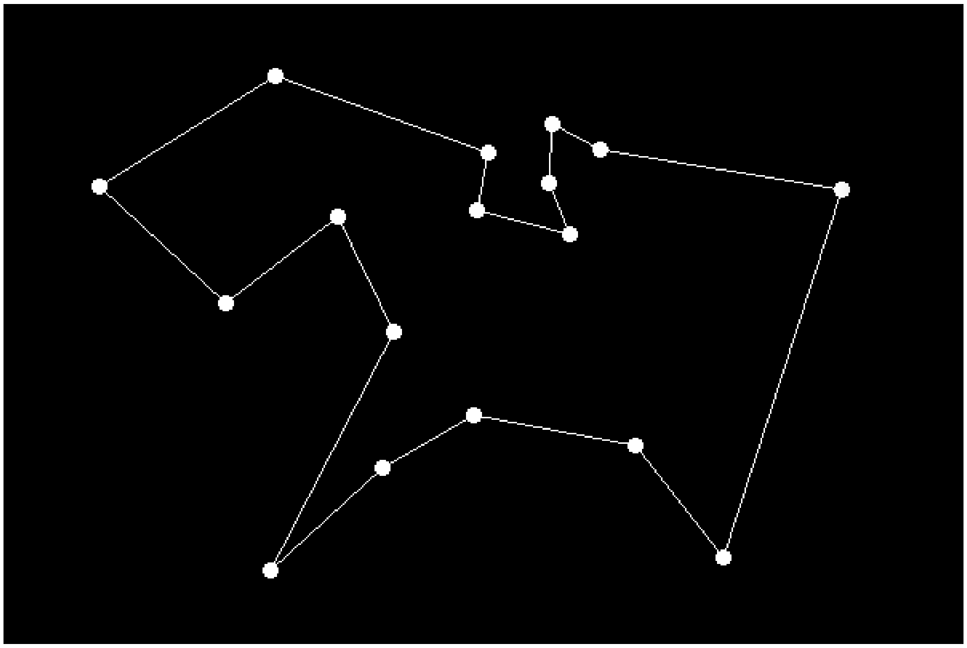
Sciezka: **5, 6, 15, 1, 8, 9, 14, 16, 11, 12, 7, 3, 9, 2, 13, 4, 10,**

Odleglosc: **1775.11**



Sciezka: **2, 7, 12, 11, 16, 14, 9, 8, 1, 5, 6, 15, 3, 0, 10, 4, 13,**

Odleglosc: **1664.34**

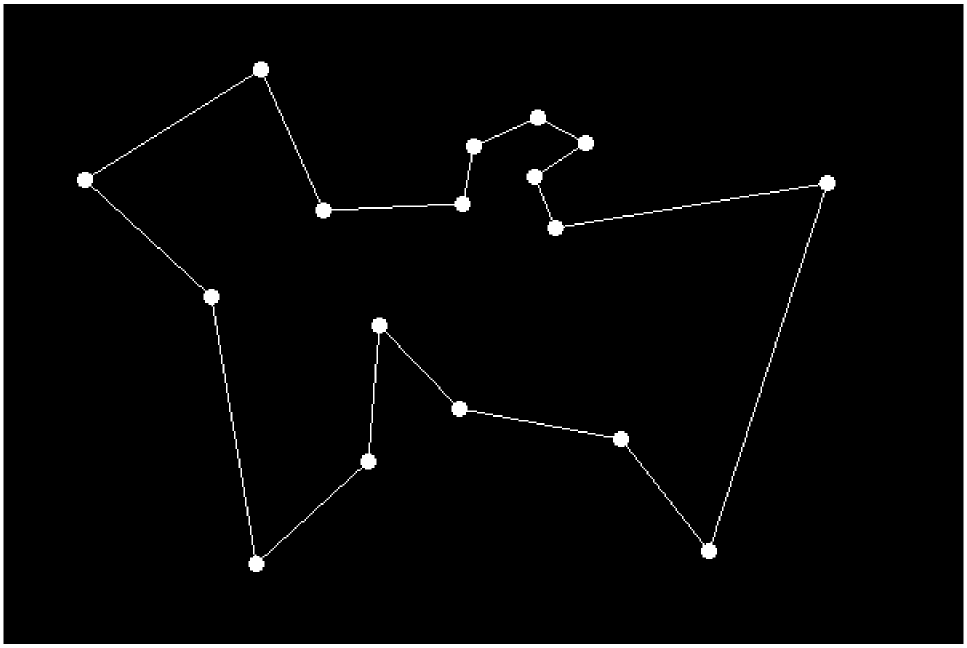


Obie metody

Przy prawdopodobieństwie wyboru metody zależnego od temperatury.

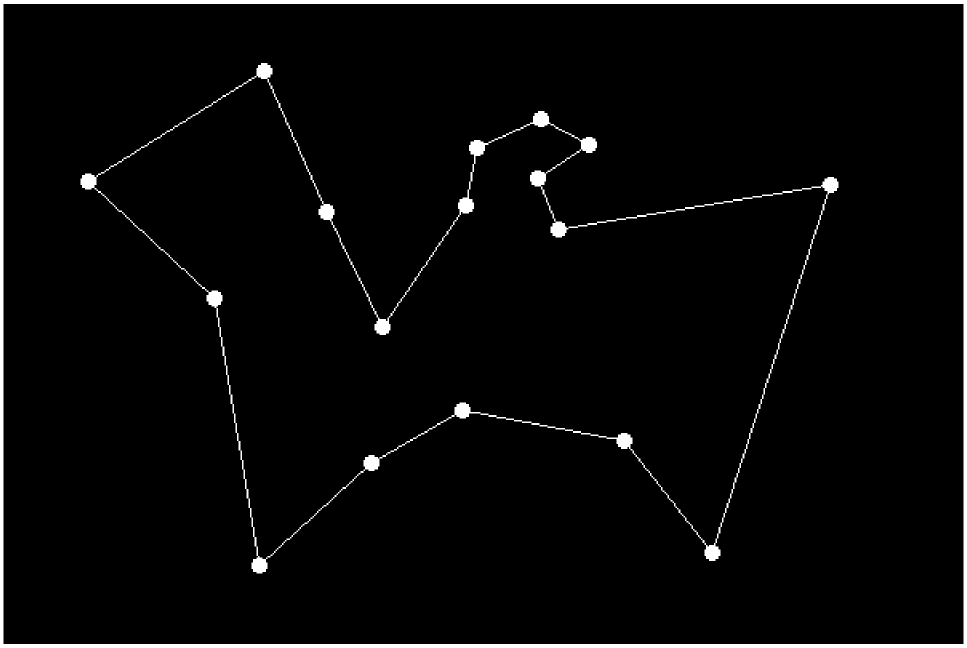
Sciezka: **16, 7, 11, 12, 13, 4, 10, 2, 3, 0, 5, 1, 6, 15, 8, 9, 14,**

Odleglosc: **1635.41**



Sciezka: **9, 8, 15, 6, 1, 5, 0, 3, 7, 2, 10, 4, 13, 12, 11, 16, 14,**

Odleglosc: **1628.9**



Implementacja

```cpp
#include <iostream>
#include <vector>
#include <random>
#include <chrono>
#include <string>
#include <cmath>

// point
struct Point
{
    int x;
    int y;
};

float distanceBetweenPoints(const Point& a, const Point& b)
{
    return sqrt( float(a.x - b.x)*(a.x - b.x) + (a.y - b.y)*(a.y - b.y) );
}

// random and shuffles
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);

float randomProbability()
{
    std::uniform_real_distribution<float> distribution(0.0f, 1.0f);
    return distribution(generator);
}

int randomBetween(int a, int b)
{
    std::uniform_int_distribution<int> distribution(a, b);
    return distribution(generator);
}

void shuffle(int* tab, int n)
{
    for (int i = n-1; i > 0; --i)
    {
        int j = randomBetween(0, i);
        std::swap(tab[i], tab[j]);
    }
}

std::vector<Point> getInput(std::istream& cin)
{
    std::vector<Point> points;
    Point point;
    unsigned int pos = 0;

    std::string s,t;
    getline(cin,s);

    while ((pos = s.find_first_of(','))!=std::string::npos)
    {
        point.x = stoi(s.substr(0,pos));
        s.erase(0,pos+1);
        if ((pos = s.find_first_of(','))==std::string::npos)
            break;
        point.y = stoi(s.substr(0,pos));
        s.erase(0,pos+1);
        points.push_back(point);
    }

    return points;
}

// annealing
enum Direction {LEFT = -1, RIGHT = 1};

struct Path
{
    int* path = nullptr;
    int sizePath = 0;
    float energy = 0.0f;
};

void printPath(Path& path)
{
    using std::cout;
    using std::endl;
    cout << "Sciezka: ";
    for (int i = 0; i < path.sizePath; ++i)
        cout << path.path[i] << ", ";
    cout << endl << "Odleglosc: " << path.energy << endl;
}

void deletePath(Path& path)
{
    delete[] path.path;
}

int* copyTable(int* tab, int n)
{
    int* result = new int[n];
    for (int i = 0; i < n; ++i)
        result[i] = tab[i];
    return result;
}

Path copyPath(const Path& path)
{
    Path newPath = path;
    newPath.path = copyTable(path.path, path.sizePath);
    return newPath;
}

void calculateSwapEnergy(Path& newPath, float** matrix, int i1, int i2)
{
    // get selected vertices to swap
    int v1 = newPath.path[i1];
    int v2 = newPath.path[i2];
    // get their connected vertices
    int v1L, v1R, v2L, v2R;
    if (i1 == 0)
```

```
        v1L = newPath.path[newPath.sizePath-1];
    else
        v1L = newPath.path[i1+LEFT];
    v1R = newPath.path[i1+RIGHT];

    v2L = newPath.path[i2+LEFT];
    if (i2 == newPath.sizePath-1)
        v2R = newPath.path[0];
    else
        v2R = newPath.path[i2+RIGHT];

    // delete distances of nonexisting connections
    newPath.energy-= matrix[v1][v1L] + matrix[v1][v1R] +
                     matrix[v2][v2L] + matrix[v2][v2R];

    // swap two vertices
    std::swap(newPath.path[i1], newPath.path[i2]);
    // now i1->v2 and i2->v1, so i1+RIGHT=v2R
    // get their connected vertices
    if (i1 == 0)
        v2L = newPath.path[newPath.sizePath-1];
    else
        v2L = newPath.path[i1+LEFT];
    v2R = newPath.path[i1+RIGHT];

    v1L = newPath.path[i2+LEFT];
    if (i2 == newPath.sizePath-1)
        v1R = newPath.path[0];
    else
        v1R = newPath.path[i2+RIGHT];

    // add distances of new connections
    newPath.energy+= matrix[v1][v1L] + matrix[v1][v1R] +
                     matrix[v2][v2L] + matrix[v2][v2R];
}

void calculateReverseEnergy(Path& newPath, float** matrix, int i1, int i2)
{
    // get selected vertices to swap
    int v1 = newPath.path[i1];
    int v2 = newPath.path[i2];
    // get their connected vertices
    int v1R = newPath.path[i1+RIGHT];
    int v2L = newPath.path[i2+LEFT];
    // calculate new distance
    newPath.energy+= matrix[v1][v2L] + matrix[v2][v1R] -
                     matrix[v1][v1R] - matrix[v2][v2L];
}

void reverseSetPath(Path& path, int i1, int i2)
{
    while(++i1 < --i2)
        std::swap(path.path[i1], path.path[i2]);
}

Path calculatePath(Path& oldPath, float** matrix, int i1, int i2, bool swap)
{
    Path newPath = copyPath(oldPath);
    if (swap)
        calculateSwapEnergy(newPath, matrix, i1, i2);
    else
        calculateReverseEnergy(newPath, matrix, i1, i2);

    return newPath;
}

Path initializeNewPath(float** matrix, int matrixSize)
{
    Path currentPath;
    currentPath.sizePath = matrixSize;

    currentPath.path = new int[matrixSize];
    // make path equal to [0,1,2,3,4,...]
    for (int i = 0; i < matrixSize; ++i)
        currentPath.path[i] = i;

    // shuffle, take random initial state
    shuffle(currentPath.path, matrixSize);

    // calculate energy of initial state
    for (int i = 0; i < matrixSize-1; ++i)
        currentPath.energy+= matrix[currentPath.path[i]][currentPath.path[i+1]];
    currentPath.energy+= matrix[currentPath.path[0]][currentPath.path[matrixSize-
1]];

    return currentPath;
}

void getTwoRandomIndices(int& i1, int& i2, int size)
{
    i1 = randomBetween(0, size-1);
    i2 = randomBetween(0, size-1);
    while (i2 == i1)
        i2 = randomBetween(0, size-1);
    // s1 is to the left of s2
    // so swapping and reversing takes fewers steps
    if (i2 < i1) std::swap(i2, i1);
}

// returns true if path was changed, false otherwise
bool getNewPath(Path& currentPath, float** matrix, float temperature)
{
    // get two random indices for swapping or reversing
    int i1, i2;
    getTwoRandomIndices(i1, i2, currentPath.sizePath);
    // depending on the temperature choose a method of calculating neighbouring
state
    float r = randomProbability();
    float p = exp(-1/temperature);
    // higher probability for swapping for higher temperature
    bool swap = (p > r);

    // if reverse is chosen, and random indices are i.e. 2 and 4
    // then between them is only one vertice to be reversed, so no reverse happens
    // same for 2 and 3, then there are no vertices between to be reversed
    // so distance between random indices must be 3 or greater.
    while ((i2 - i1 < 3) && !swap)
```

```
        getTwoRandomIndices(i1, i2, currentPath.sizePath);

    Path newPath;
    // get new neighboring state
    newPath = calculatePath(currentPath, matrix, i1, i2, swap);

    if (newPath.energy >= currentPath.energy)
    {
        float dEnergy = currentPath.energy - newPath.energy;
        p = exp(dEnergy/temperature);
        if (p < r)
            return false;
    }

    // since reverse does not need to physically reverse vertices to calculate
distance
    // if the new reversed path is to be used it needs to be physically reversed
    if (!swap)
        reverseSetPath(newPath, i1, i2);

    // delete old path and set it to new path
    deletePath(currentPath);
    currentPath = newPath;
    return true;
}

Path annealingFindPath(float** matrix, int matrixSize, float initialTemperature,
float q)
{
    Path currentPath = initializeNewPath(matrix, matrixSize);

    int withoutChangeCounter = 0;
    // after how many temperature changes without changing path the algorithm will
stop
    int threshold = 10000;

    for (float temperature = initialTemperature; (withoutChangeCounter < threshold)
         && (temperature > 0.1f); temperature*= q)
    {
        if (getNewPath(currentPath, matrix, temperature))
            withoutChangeCounter = 0;
        else
            ++withoutChangeCounter;
    }

    return currentPath;
}

// graph and main
float** createMatrixOfACompleteGraph(const std::vector<Point>& points)
{
    // memory for matrix
    float** matrix = new float*[points.size()];
    for (unsigned int i = 0; i < points.size(); ++i)
    {
        matrix[i] = new float[points.size()];
        // main diagonal is all zeros
        matrix[i][i] = 0;
    }
    for (unsigned int i = 0; i < points.size(); ++i)
        for (unsigned int j = i+1; j < points.size(); ++j)
            matrix[i][j] = matrix[j][i] =
                distanceBetweenPoints(points[i], points[j]);

    return matrix;
}

void deleteMatrix(float** matrix, int n)
{
    for (int i = 0; i < n; ++i)
        delete[] matrix[i];
    delete[] matrix;
}

int main()
{
    using namespace std;

    cout << "Wprowadz wspolrzedne:\n";
    vector<Point> points = getInput(cin);

    cout << "Wprowadz temperature:\n";
    float temperature = 0.0f;
    cin >> temperature;
    cout << "Wprowadz q:\n";
    float q = 0.0f;
    cin >> q;

    cout << "Polozenia wierzcholkow grafu na plaszczyznie:" << endl;
    for (unsigned int i = 0; i < points.size(); ++i)
        cout << i << ": (" << points[i].x << ", " << points[i].y  << ")" << endl;

    float** matrix = createMatrixOfACompleteGraph(points);

    Path path = annealingFindPath(matrix, points.size(), temperature, q);

    printPath(path);

    deleteMatrix(matrix, points.size());
    deletePath(path);
    return 0;
}
```