Algorithms

Course project:

Simulated annealing applied to the traveling salesman problem

Table of Contents

Algorithm summary

The initial path (permutation) is obtained using a Fisher-Yates algorithm. In the main loop there are two stopping conditions:

- the temperature is equal or goes below a number 0.1,
- the path wasn't modified for 1000 consecutive temperature changes.

New path can be obtained in a two ways, depending on the temperature, the function getNewPath calculates a probability for a method to be used on a current permutation:

- swapping two random vertices *(higher temperature = higher chance)*

  1, ***2***, 3, 4, 5, ***6*** → 1, ***6***, 3, 4, 5, ***2***

- reversing the order of vertices between two random vertices *(lower temperature = higher chance)*

  1, ***2***, **3**, **4**, **5**, ***6*** → 1, ***2***, **5**, **4**, **3**, ***6***

Comparison of the methods

Input data:

- (x,y) points coordinates:

  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,

- Initial temperature:

  5

- Multiplier q:

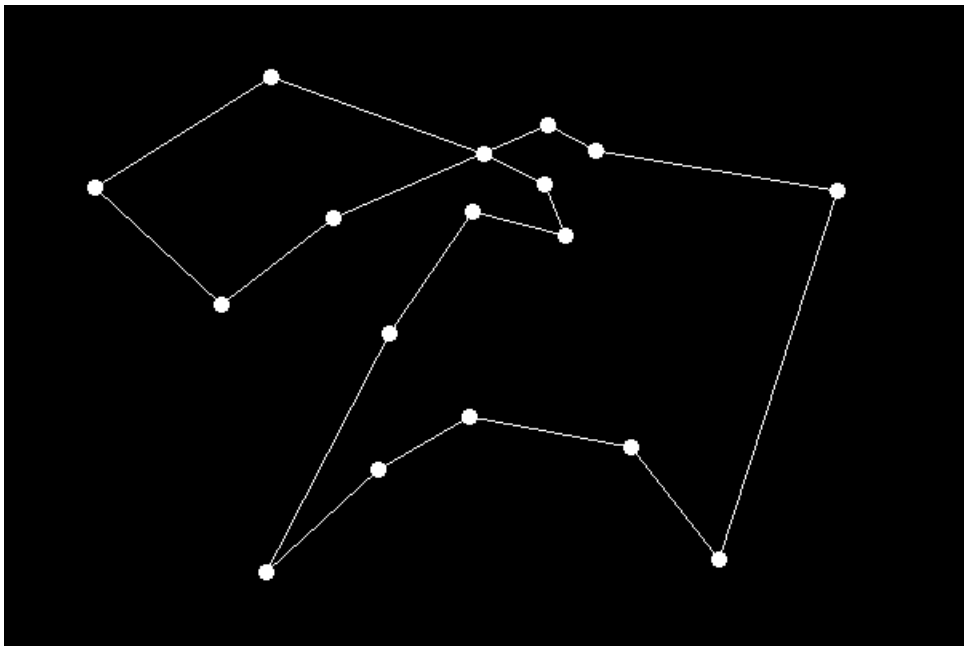
  0.99999

Swapping

Making the algorithm use swapping only is achieved by setting the variable **bool swap = true**.

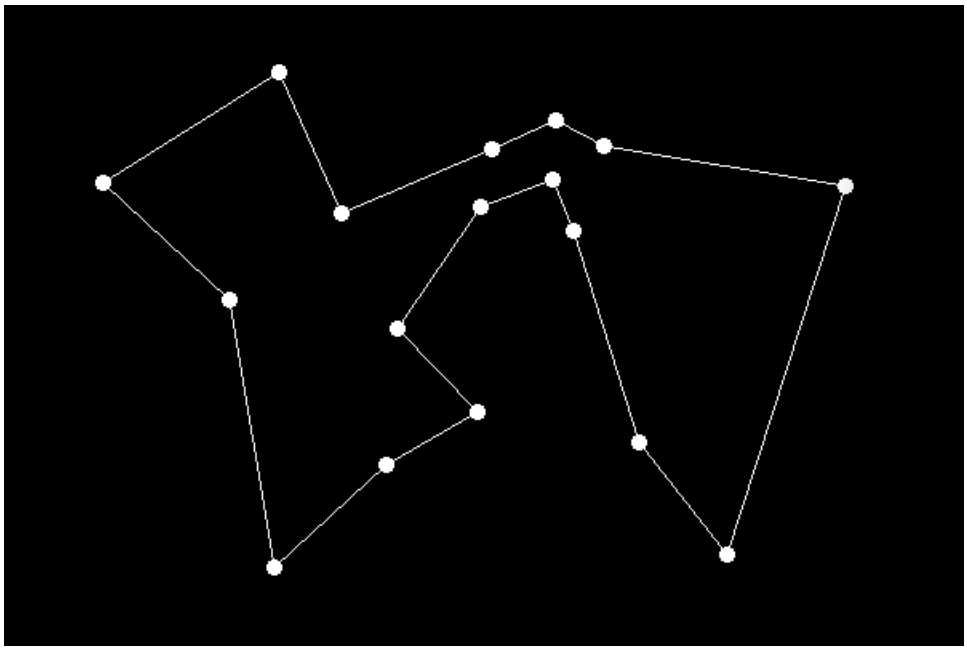
Output path: **8, 1, 5, 2, 13, 4, 10, 0, 6, 15, 3, 7, 12, 11, 16, 14, 9,**

Distance: **1791.14**



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

Distance: **1712.51**

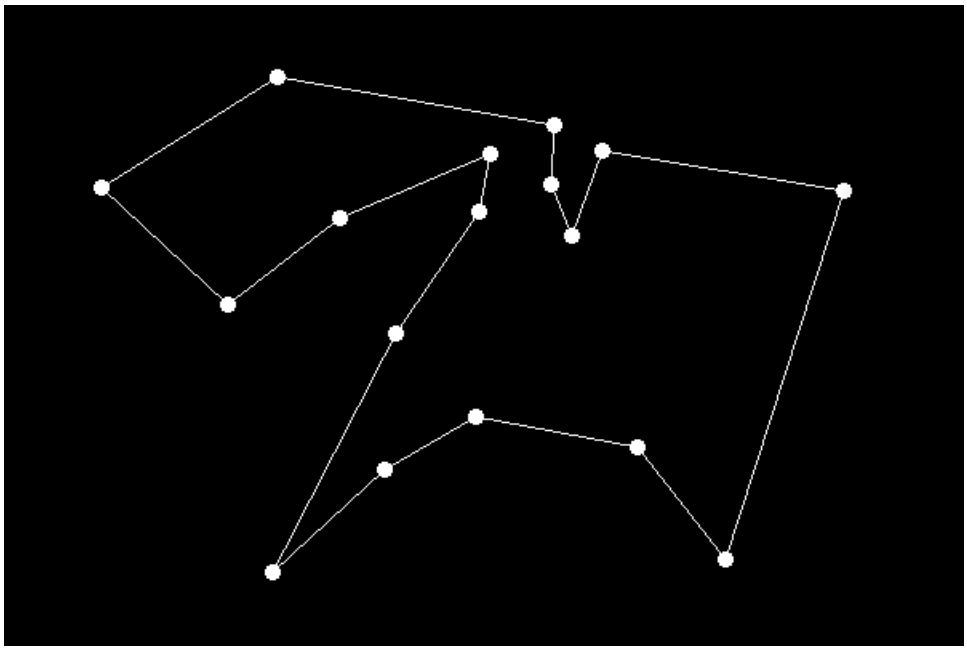


Reversing the order

Making the algorithm use reversing only is achieved by setting the variable **bool swap = false**.

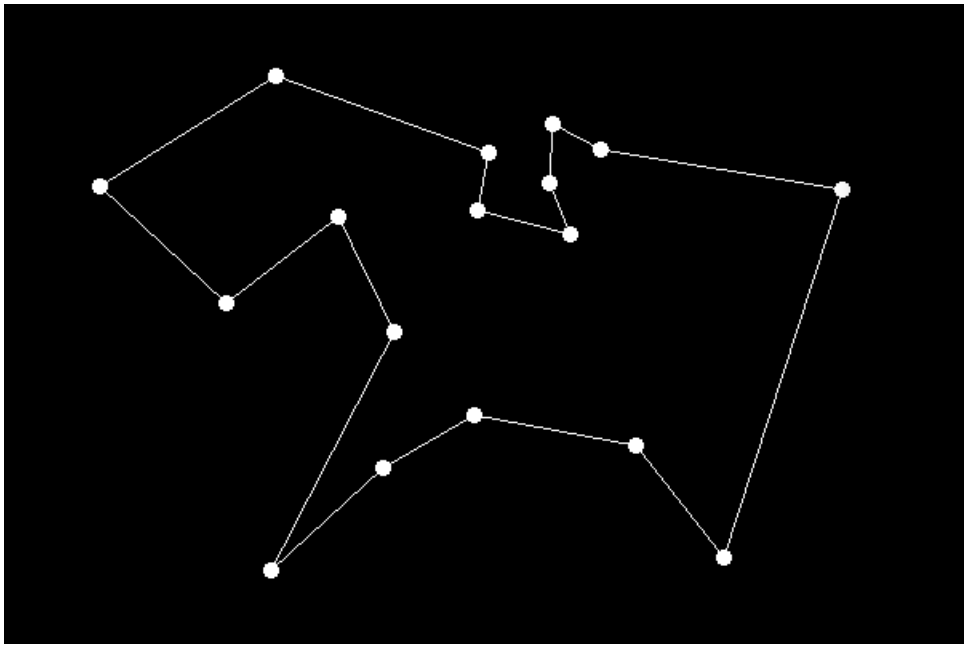
Output path: **5, 6, 15, 1, 8, 9, 14, 16, 11, 12, 7, 3, 9, 2, 13, 4, 10,**

Distance: **1775.11**



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

Distance: **1664.34**

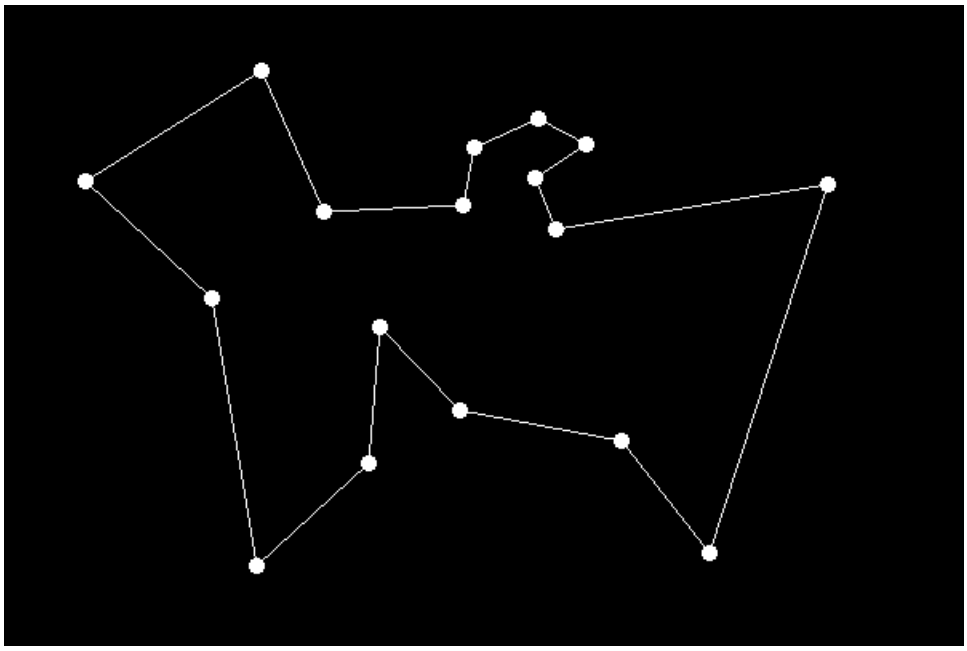


Both methods

With the probability of choosing either method depending on the temperature.

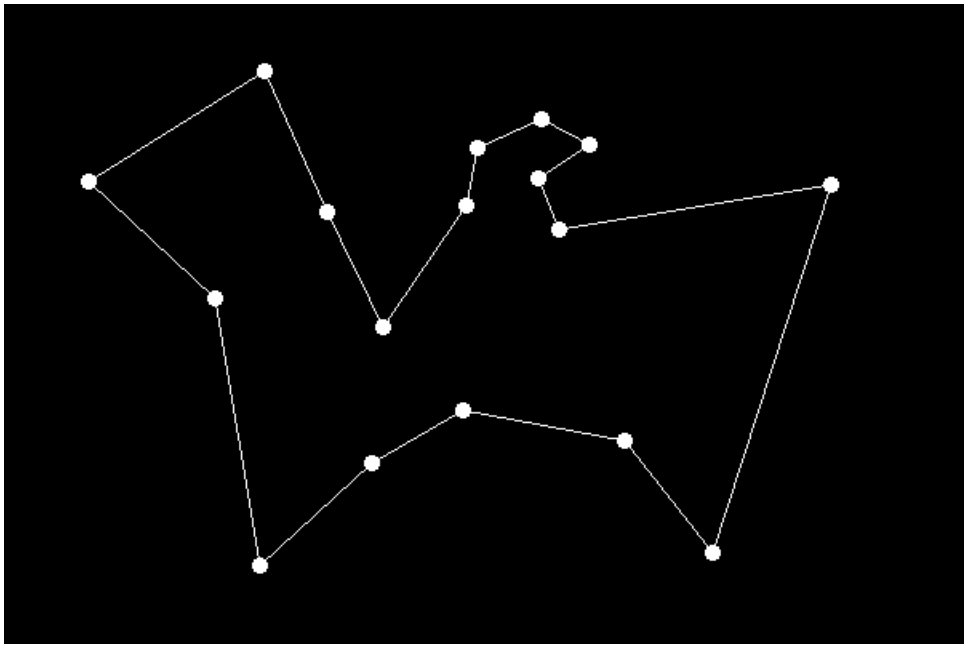
Output path: **16, 7, 11, 12, 13, 4, 10, 2, 3, 0, 5, 1, 6, 15, 8, 9, 14,**

Distance: **1635.41**



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

Distance: **1628.9**



C++ Implementation

```cpp
#include <SFML/Graphics.hpp>
#include <cstdlib>
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <random>
#include <chrono>

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

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) );
}

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]);
    }
}

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

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

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;
}

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()
{
    enum {WIDTH = 800, HEIGHT = 600};

    std::cout << "The visualization can be stopped at any time
    by closing the Finding path window,\n";
    std::cout << "in such case the program will display
    a currently found permutation.\n\n";
    std::cout << "Enter points x,y, coordinates:\n";
    std::vector<Point> points = getInput(std::cin);

    std::cout << "\nEnter temperature:\n";
    float initialTemperature = 0.0f;
    std::cin >> initialTemperature;
    std::cout << "\nEnter q:\n";
    float q = 0.0f;
    std::cin >> q;

    float** matrix = createMatrixOfACompleteGraph(points);

    int sleepRestriction = 0;
    std::cout << "\nSlow down the visualization on initial steps?\n
    0 - No\nOther - Yes\n";
    std::cin >> sleepRestriction;
    std::cin.ignore();
    if (sleepRestriction != 0)
        sleepRestriction = 1;

    // annealing initial state
    Path currentPath = initializeNewPath(matrix, points.size());

    int withoutChangeCounter = 0;
    // after how many temperature changes without changing path
    the algorithm stops
    int changeThreshold = 1000;
    // after reaching what temperature the algorithm stops
    float temperatureThreshold = 0.1f;
    float temperature = initialTemperature;
    bool annealing = true;

    // start visualization
    sf::RenderWindow window(sf::VideoMode(WIDTH, HEIGHT), "Finding path");

    // Create background.
    sf::RenderTexture background;
    if (!background.create(WIDTH, HEIGHT))
        return -1;
    background.clear(sf::Color::Black);

    // connection between vertices
    sf::Vertex line[2];

    // radius of vertices
    const float radius = 5.0f;
```

```
// draw each vertex on background
for (unsigned int i = 0; i < points.size(); ++i)
{
    sf::CircleShape vertex(radius);
    sf::Vector2f middle(float(points[i].x), float(points[i].y));

    vertex.setPosition(middle-sf::Vector2f(radius, radius));
    vertex.setFillColor(sf::Color::White);
    // Draw vertex on the background.
    background.draw(vertex);
}
sf::Sprite backgroundVertices(background.getTexture());
backgroundVertices.setTextureRect(sf::IntRect(0, HEIGHT, WIDTH, -HEIGHT));

// The main loop.
while (window.isOpen())
{
    window.clear(sf::Color::Black);
    sf::Event event;
    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            window.close();
    }

    // annealing
    annealing = (withoutChangeCounter < changeThreshold) &&
                (temperature > temperatureThreshold);

    if (annealing)
    {
        if (getNewPath(currentPath, matrix, temperature))
        {
            withoutChangeCounter = 0;
            sf::sleep(sf::milliseconds(sleepRestriction*temperature*40));
        }
        else
            ++withoutChangeCounter;

        // lower the temperature
        temperature*= q;
    }
    else
    {
        std::cout << "######################" << std::endl;
        std::cout << "PERMUTATION FOUND:" << std::endl;
        printPath(currentPath);
        std::cout << "Press enter to exit." << std::endl;
        std::cin.get();
        window.close();
    }

    // Actual drawing.

    // draw vertices
    window.draw(backgroundVertices);
```

```
            // draw connections between vertices
            for (int i = 0; i < currentPath.sizePath-1; ++i)
            {
                int v1 = currentPath.path[i];
                int v2 = currentPath.path[i+1];
                sf::Vector2f start(float(points[v1].x), float(points[v1].y));
                sf::Vector2f end(float(points[v2].x), float(points[v2].y));

                line[0].position = start;
                line[1].position = end;
                //draw the line
                window.draw(line, 2, sf::Lines);
            }
            // and between first and last
            int v1 = currentPath.path[0];
            int v2 = currentPath.path[currentPath.sizePath-1];
            sf::Vector2f start(float(points[v1].x), float(points[v1].y));
            sf::Vector2f end(float(points[v2].x), float(points[v2].y));

            line[0].position = start;
            line[1].position = end;
            // draw the line
            window.draw(line, 2, sf::Lines);

            window.display();
        }
        // end of visualization

        if (annealing)
        {
            std::cout << "################################" << std::endl;
            std::cout << "VISUALIZATION STOPPED MANUALLY" << std::endl;
            std::cout << "PERMUTATION FOUND:" << std::endl;
            printPath(currentPath);
            std::cout << "Press enter to exit." << std::endl;
            std::cin.get();
        }

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