Algorithms

Project A

Table of Contents

Answering project main question

Difference between Prim's algorithm for finding minimum spanning tree and Dijkstra's algorithm for finding shortest path is how the distances to be put in an array are calculated.
In Prim's case only distances between two vertices are compared, while in Dijkstra's distances relatively to starting vertex are compared.

Difference implementation in code:

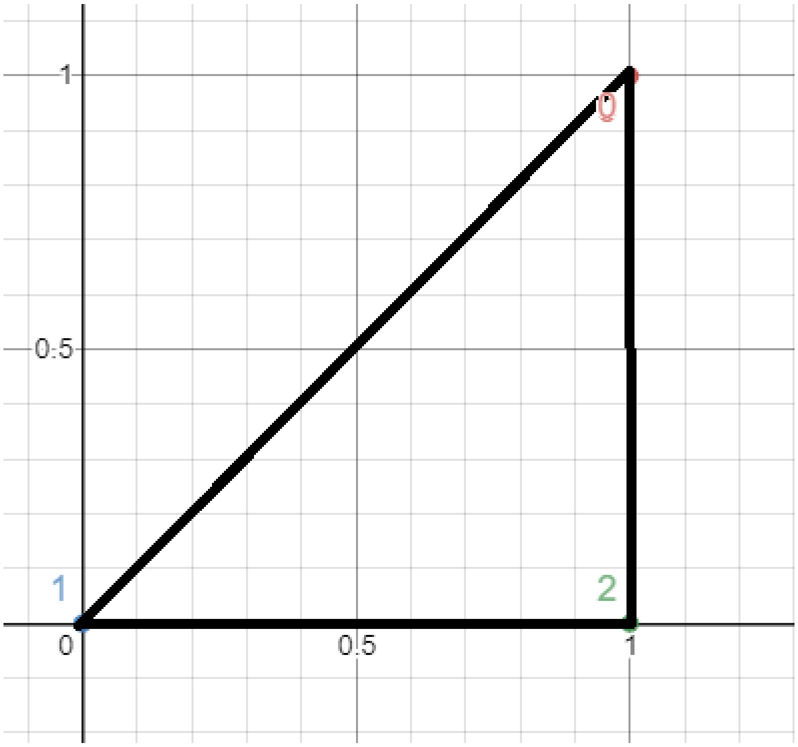
```
float newDistance = int(!prim) * table[processedVertex].distance
+ vertex→distance;,
```

where prim has value of true or false; depending on algorithm used.

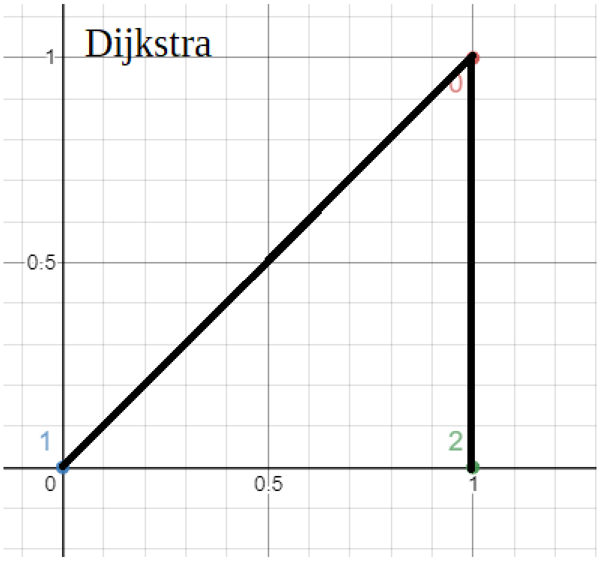
If prim is false then vertex's distance relatively to starting vertex is added to distance between two compared vertices, otherwise only distance between vertices is used.

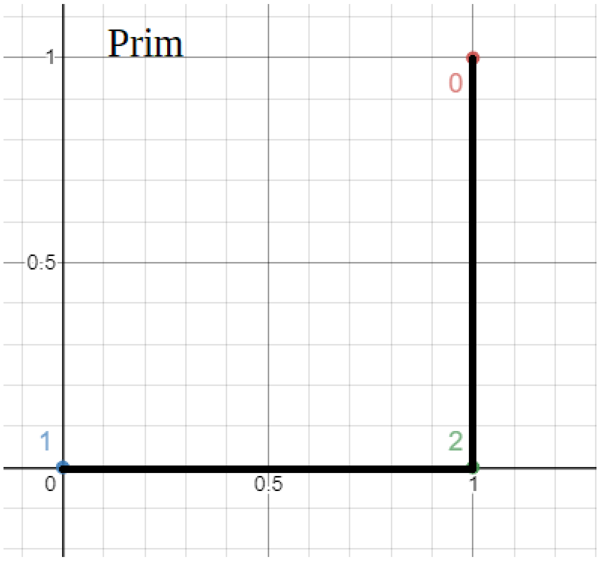
A graph example for which both algorithms give different results starting from the same vertex

Given input: **1,1,0,0,1,0,** which translates to three vertices: 0 – (1,1), 1 – (0,0) and 2 – (1,0), from which a complete graph is created and distances are calculated using the distance formula:



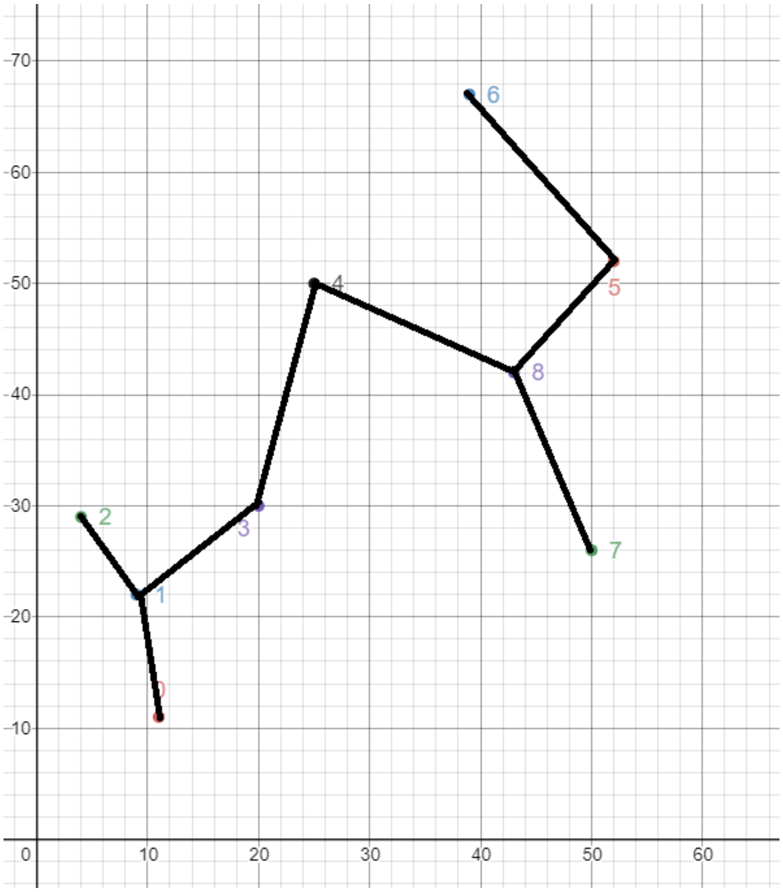
When starting from vertex 0 – (1,1) program outputs **Shortest path tree: 2-0,1-0, MSTree: 2-0,1-2,**





Finding minimum spanning tree

Given points on a plane (11,11),(9, 22),(4,29),(20,30),(25,50),(52,52),(39,67),(50,26),(43, 42). Input: **11,11,9,22,4,29,20,30,25,50,52,52,39,67,50,26,43,42,** and starting from the vertex 4 – (25,50), program outputs **MSTree: 8-4,5-8,7-8,6-5,3-4,1-3,2-1,0-1,**



C++ Implementation

```cpp
#include <queue>
#include <vector>
#include <iostream>
#include <string>
#include <cmath>
#include <limits>

//vertex.h
struct Vertex
{
    int connectedTo;
    float distance;
    Vertex* next = nullptr;
};

//graph.h
struct Graph
{
    Vertex** vertices = nullptr;
    int size = 0;
    std::string tree = "";
};

void connectOneVertex(Vertex*& vertex, int connectedTo, float distance);
void connectVertex(Graph& graph, int from, int to, float distance);
Graph createGraphFromMatrix(float** matrix, int n);
void deleteGraph(Graph& graph);
void deleteVertices(Vertex* vertex);
void printGraph(const Graph& graph);

//point.h
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) );
}

//row.h
struct Row
{
    bool visited = false;
    float distance = std::numeric_limits<float>::infinity(); // infinity
    int previous;
};

//dijkstra.h
Graph dijkstra(const Graph& graph, int startingVertex, bool prim = false);
void updateDistances(Row* table, Vertex** vertices, int processedVertex, bool prim
= false);
```

```
int findLowestDistance(Row* tab, int size);

//prim.h
Graph prim(const Graph& graph, int startingVertex);


//main.cpp
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** createMatrixOfCompleteGraph(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 freeMemoryOfMatrix(float** matrix, int n)
{
    for (int i = 0; i < n; ++i)
        delete[] matrix[i];
    delete[] matrix;
}

int main()
```

```
{
    using namespace std;

    cout << "Input: ";
    vector<Point> points = getInput(cin);

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


    float** matrix = createMatrixOfCompleteGraph(points);
/*
//  PRINT MATRIX
    cout << "graph size: " << points.size() << endl;
    for (int i = 0; i < points.size(); ++i)
    {
        for (int j = 0; j < points.size(); ++j)
            cout << test[i][j] << ' ';
        cout << endl;
    }
*/

    Graph graph = createGraphFromMatrix(matrix, points.size());

    cout << "Graph:\n";
    printGraph(graph);

    cout << "Starting vertex: ";
    int startingVertex = 0;
    cin >> startingVertex;

    cout << endl << endl << "Shortest path:\n";
    Graph shortestPath = dijkstra(graph, startingVertex);
    printGraph(shortestPath);

    cout << endl << "Shortest path tree: " << shortestPath.tree;

    cout << endl << endl << "MST:\n";
    Graph mst = prim(graph, startingVertex);
    printGraph(mst);
    cout << endl << "MSTree: " << mst.tree;

    deleteGraph(graph);
    deleteGraph(shortestPath);
    deleteGraph(mst);
    freeMemoryOfMatrix(matrix, points.size());
    return 0;
}


/*
 * IMPLEMENTATION
 */
//dijkstra.cpp
Graph dijkstra(const Graph& graph, int startingVertex, bool prim)
{
    // initialize shortest path graph (tree)
    Graph shortestPathGraph;
    // graph will have same number of vertices as graph it's created from
    shortestPathGraph.size = graph.size;
    shortestPathGraph.vertices = new Vertex*[graph.size];
    // at first there are no vertices connected to a vertex
    // it is needed because each vertex is points to other,
    // and when there are no more vertices it points to null
    for (int i = 0; i < graph.size; ++i) shortestPathGraph.vertices[i] = nullptr;

    // an array to hold algorithm's table
    Row* table = new Row[graph.size];
    Vertex** vertices = graph.vertices;

    // Distance to starting vertex is 0 and it's visited
    table[startingVertex].distance = 0.0f;
    table[startingVertex].visited = true;
    int visitedSize = 1;

    for (int processedVertex = startingVertex;
            visitedSize < graph.size; ++visitedSize)
    {
        updateDistances(table, vertices, processedVertex, prim);
        processedVertex = findLowestDistance(table, graph.size);
        table[processedVertex].visited = true;

        int previous = table[processedVertex].previous;
        // calculate distance between two vertices from the table
        float distanceBetween = table[processedVertex].distance;
        distanceBetween -= int(!prim) * table[previous].distance;
        // connect previous vertex with vertex with current lowest distance
        connectVertex(shortestPathGraph, processedVertex, previous,
distanceBetween);
    }

    delete[] table;
    return shortestPathGraph;
}

void updateDistances(Row* table, Vertex** vertices, int processedVertex, bool prim)
{
    for (Vertex* vertex = vertices[processedVertex];
         vertex; vertex = vertex→next)
{
        // alias for row of an adjacent vertex
        Row &adjVertexRow = table[vertex->connectedTo];

        if (adjVertexRow.visited) continue;

        // calculate distance accordingly to algorithm used
        float newDistance = int(!prim) * table[processedVertex].distance
                            + vertex->distance;

        if (adjVertexRow.distance > newDistance)
        {
            adjVertexRow.distance = newDistance;
            adjVertexRow.previous = processedVertex;
        }
    }

}

int findLowestDistance(Row* tab, int size)
{
    int lowest = 0;
    // find first not visited vertex's index
    for (int i = 0; i < size; ++i)
        if (!tab[i].visited)
            lowest = i;

    for (int i = 0; i < size; ++i)
        if (!tab[i].visited
            && (tab[i].distance < tab[lowest].distance))
            lowest = i;

    return lowest;
}

//prim.cpp
Graph prim(const Graph& graph, int startingVertex)
{
    return dijkstra(graph, startingVertex, true);
}

//graph.cpp
void connectOneVertex(Vertex* &vertex, int connectedTo, float distance)
{
    Vertex* newVertex = new Vertex;
    newVertex->connectedTo = connectedTo;
    newVertex->distance = distance;
    newVertex->next = vertex;

    vertex = newVertex;
}

void connectVertex(Graph& graph, int from, int to, float distance)
{
    connectOneVertex(graph.vertices[from], to, distance);
    connectOneVertex(graph.vertices[to], from, distance);
    graph.tree+= std::to_string(from) + "-" + std::to_string(to) + ",";
}

Graph createGraphFromMatrix(float** matrix, int n)
{
    Graph graph;
    // table of vertices connected to vertices
    graph.vertices = new Vertex*[n];
    // at first there are no vertices connected to a vertex
    for (int i = 0; i < n; ++i) graph.vertices[i] = nullptr;
    graph.size = n;

    for (int i = 0; i < n; ++i)
        for (int j = i; j < n; ++j)
            if (matrix[i][j])
                connectVertex(graph, i, j, matrix[i][j]);
    return graph;
}

void deleteGraph(Graph& graph)
{
    Vertex** vertices = graph.vertices;
    for (int i = 0; i < graph.size; ++i)
        deleteVertices(vertices[i]);
    delete[] vertices;
}

void deleteVertices(Vertex* vertex)
{
    Vertex* tmp;
    while (vertex)
    {
        tmp = vertex;
        vertex = vertex->next;
        delete tmp;
    }
}

void printGraph(const Graph& graph)
{
    using std::cout;
    using std::endl;

    Vertex* vertex;
    for (int i = 0; i < graph.size; ++i)
    {
        vertex = graph.vertices[i];
        cout << i << "->";
        while (vertex)
        {
            cout << "(" << vertex->connectedTo
                 << "," << vertex->distance << ") ";
            vertex = vertex->next;
        }
        cout << endl;
    }
}
```