Middle East Technical University - Department of Computer Engineering

CENG 213

Data Structures

Fall '2022-2023

Programming Assignment 3

Due Date: 3 January 2023, Tuesday, 23:55
Late Submission Policy will be explained below

Objectives

In this programming assignment, you are expected to implement a 3D object painter. In order to implement such an application, you will implement a graph data structure that will contain the given 3D object. The such graph data structure will be used to color the given vertices. There will be multiple painting methods which will be discussed later. In this graph data structure, you are required to do the shortest path algorithm. To accomplish this task you are required to implement your own binary heap data structure. Details of these data structures will be explained in the following sections.

Keywords: C++, Data Structures, Graphs, Dijkstra Shortest Path Algorithm, Breadth-First Search, Binary Heap, Decrease Priority

1 Binary Heap Implementation (35 pts)

Binary heap implementation consists of classic heap data structure functionality. In addition to that, it will have a change priority functionality which will be explained in the next section. This functionality will be useful when you implement you shortest path algorithm on the MeshGraph data structure.

There are key differences between the classic heap and this data structure. First of all, each element of this data structure must be **unique**. This will be determined by the "uniqueId" variable on the heap elements. This data structure does not accept another element with the same unique id.

It is important to mention that this data structure is implicitly a min heap; meaning that, the root of the data structure will have the smallest item that is determined by the weight variable. The data layout of the Graph class and its helper structures can be seen on Listing 1.

Graph class and its helper structures are declared in *BinaryHeap.h* header file and their implementations (although most of it is empty) are defined in *BinaryHeap.cpp* file. We will explain the details of each function below.

1.1 `BinaryHeap();`

There is not much to say about the constructor. Since heap class uses `std::vector<T>` as its internal data structure. You may do initialization if your implementation requires so or leave it empty.

1.2 `bool Add(int uniqueId, double weight);`

This function returns true if add operation is successful. Add operation can fail; thus returning false, when there is another item with the same id as the `uniqueId`. Otherwise, this function should return true.

```cpp
struct HeapElement
{
    int     uniqueId;
    double  weight;
};

class BinaryHeap
{
    private:
    std::vector<HeapElement> elements;

    // Do not remove this the tester will utilize this
    // to access the private parts.
    friend class HW3Tester;
    ....

    public:
    // Constructors & Destructor
                        BinaryHeap();
    //
    bool                Add(int uniqueId, double weight);
    void                PopHeap(int& outUniqueId, double& outWeight);
    bool                ChangePriority(int uniqueId, double newWeight);
    int                 HeapSize() const;
};
```

Listing 1: Binary Heap Helper Structure data layout

1.3 `void PopHeap(int& outUniqueId, double& outWeight);`

Pops the item with the lowest priority. This is similar to the basic heap function except that it returns the weight and the unique id.

1.4 `bool ChangePriority(int uniqueId, double newWeight);`

This functionality of the data structure is required to efficiently implement shortest path algorithm. This function should search the item with the id `uniqueId` and returns true when this item exists and it should change the item's weight with the given `newWeight`. Do not forget to make the data structure heap again after this operation.

1.5 `int HeapSize() const;`

Returns the element count of the data structure.

2 MeshGraph Implementation (65 pts)

The MeshGraph data structure will contain the main functionality that will be used in this assignment. It follows the adjacency list style of implementation with slight differences. Graph Data Structure has non-negative weights for its edges. However, these weights are not explicit. These weights will be calculated by the parameters of the vertices. Additionally, edges on this data structure is **bi-directional** meaning that the graph is **not** a directional graph. The data layout of the Graph class and its helper structures can be seen on Listing 2.

Almost all the surfaces defined in computer games and computer-generated imagery (CGI) films, can be considered as a graph. In this assignment, all of the data you are processing (in the tests or given with the assignment pack) are 3D objects that can be read with appropriate programs.

We have a simple program that visualizes your output (you need to output the result of your functions this will be announced later.)

Graph holds its data in a dynamic array and an array of linked list. For these, the class utilizes `std::vector<T>` class and `std::list<T>` class from the *Standard Template Library (STL)*. Graph class and its helper structures are declared in *MeshGraph.h* header file and their implementations (although most of it is empty) are defined in *MeshGraph.cpp* file.

```
struct Color
{
    unsigned char r, g, b;
};

struct Vertex
{
    int id;
    Double3 position3D;
};

enum FilterType
{
    FILTER_BOX,
    FILTER_GAUSSIAN,
};

class MeshGraph
{
    private:
    std::vector<Vertex>                 vertices;
    std::vector<std::list<Vertex*>>     adjList;

    // Do not remove this the tester will utilize this
    // to access the private parts.
    friend class HW3Tester;

    public:
    // ...
};
```

Listing 2: MeshGraph and Helper Structure data layout

2.1 Color Struct

Color structure holds the information about a color. For most traditional applications color is held by three 8-bit values red green and blue. Technically, these fundamental colors are "mixed" to get different colors. Please notice that 8-bit unsigned values can hold numbers between 0 and 255; both of which are included, respectively.

For example; the color [r: 255, g: 0, b: 0], will be the brightest red that can your system (and in the end your monitor) produces. [r: 10, g: 0, b: 0] will produce a dark red, [r: 0, g: 0, b: 0] will be black and [r: 255, g: 255, b: 255] will be white.

2.2 Vertex Struct

Vertex structure holds information about the vertices of the graph. In this implementation, each vertex will have a uniqueId which will be held by the variable `id`, and three dimensional position parameter which will be held by `position3D`.

> **Notice:** You must use the index of the vertex on the `vertices` array as a unique id. Testing functions assume this is true. For example; the testing function of `void ImmediateNeighbours(...) const` expects the returned ids are in this manner.

2.3 Filter Enumeration

This enumeration will be used as an input to the painting functions. The specificity of this enumeration will be explained on these functions.

2.4 MeshGraph Class

MeshGraph class implements a non-directional, non-negative weighted graph. It only allows a single edge between two vertices. It utilizes classic adjacency list implementation with linked lists and arrays. One key difference is the weights. Weights are not explicitly defined by each edge, instead, weights of the edges can be found by the vertices of this edge. Given position $p_1 = [x_1, y_1, z_1]$ of the vertex v_1 and the position $p_2 = [x_2, y_2, z_2]$ of the vertex v_2 which are the vertices of the edge e; weight of this edge is d_e and it is defined the euclidean distance between p_1 and p_2. Euclidean distance equation is given in Equation 1.

$$d_e = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \tag{1}$$

Thus when calculating the shortest path, this parameter d_e should be used as the edge weight. Now we will explain each member function and constructor of this class.

2.4.1 `MeshGraph(const std::vector<Double3>& vertexPositions, const std::vector<IdPair>& edges);`

The constructor should create an adjacency list of vertices and an array of vertices using the arguments `vertexPositions` and `edges`.
`vertexPositions` constructor argument is self-explanatory and holds the positions of the vertices of the graph. Each element on this array represents a unique graph vertex. `edges` array holds index pairs of the `vertices` variable and it represents that there is an edge between these two vertices. You can assume there are unique elements on this array. For example there is an edge between vertex v_5 and v_10 then on the `edges` array there will be either $\{5, 10\}$ or $\{10, 5\}$.

> **Hint:** Please do not forget to add edge information to the adjacency list of **both** vertices. Since as discussed before this graph is **not a directed graph**. Informally, edges can be used both ways.

2.4.2 `double AverageDistanceBetweenVertices() const;`

This function should calculate the **all** weights (distances) of the edges on this graph. Then it should calculate the arithmetic average of these weights. If there are a total of n edges, the mean calculation can be seen in Equation 2.

$$E = \frac{1}{n} \sum_{i=1}^{n} d_e \tag{2}$$

2.4.3 `double AverageEdgePerVertex() const;`

Like the function in Section 2.4.2, it calculates an average. Here the average is the number of edges of each vertex. This function should find out the edge count between each vertex and calculate the average like in Equation 2.

2.4.4 `int TotalVertexCount() const;`

This function should return the vertex count of the graph.

2.4.5 `int TotalEdgeCount() const;`

This function should return the total edge count of the graph.

> **Hint:** Do not count edges twice!

2.4.6 `int VertexEdgeCount(int vertexId) const;`

This function should return a specific edge count of a vertex with "`vertexId`". If vertexId does not exist this function should return `-1`.

2.4.7 `void ImmediateNeighbours(std::vector<int>& outVertexIds, int vertexId) const;`

This function should return all of the neighbors of the given vertex with the index `vertexId`. `id` of the vertices should be written to the `outVertexIds` array. Order is not important; while testing output array will be sorted. You can **not** assume vertexId is available on the graph. In such cases `outVertexIds` should be empty.

2.4.8 `void PaintInBetweenVertex(std::vector<Color>& outputColorAllVertex, int vertexIdFrom, int vertexIdTo, const Color& color) const;`

This function will "paint" all of the vertices in between `vertexIdFrom` and `vertexIdTo`. Both of these vertices are included. The function should return an output color for **ALL** vertices on the graph. Only the vertices on the shortest path should have the color `color` and, the rest should be black.

Algorithm Should:

- Resize the output array `outputColorAllVertex` so that it contains value for **all** vertices on the Graph.
- Set all colors in the `outputColorAllVertex` to black (R:0,G:0:B:0).
- Find the shortest path between vertex $v_{vertexIdFrom}$ and $v_{vertexIdTo}$.
- change the color of the vertices of the shortest path to "color" on the `outputColorAllVertex`.

Remarks

- As discussed before, edge weights should be the euclidean distance between two vertices.
- You are **not** allowed to use *`std::priority_queue`* STL library class. You should use `BinaryHeap` class from Section 1. Also, you should utilize `bool ChangePriority(..)` function of that class.
- In order to remain consistent between implementations, do **not** update the path weight if it is **equal** to the previously found path weight. This means that the "first" shortest path will be chosen as the shortest path if any other equally weighted shortest path exists.
- Still some inconsistencies may occur depending on the heap implementation. Because of that, comparisons should be using "less than" operation inside of the heap instead of "less than and equal".

For edge cases, namely, graph not having either vertexIdFrom or vertexIdTo, this function should return empty `outputColorAllVertex`.

2.4.9 `void PaintInRangeGeodesic(std::vector<Color>& outputColorAllVertex, int vertexId, const Color& color, int maxDepth, FilterType type, double alpha) const;`

Given a vertexId $v_{vertexId}$, this function will color the all vertices of the $v_{vertexId}$ up to a certain depth `maxDepth`.
For example; if `maxDepth` is 0, only $v_vertexId$ should be colored. If `maxDepth` is 1 the immediate neighbours and $v_vertexId$ should be colored. For `maxDepth` equals 2, the vertices than can be reached by at most two jumps should be colored. You can implement this functionality using **breadth-first** search. While implementing your breadth-first search, you are **not allowed to use** `std::queue`. You need to use your binary heap implementation (in Section 1) as a queue. How to use it as a queue is up to you.

> **Hint:** Think about strictly-increasing weights for the priority queue.

However there is a catch, color outputs of the vertices will be multiplied by the filter function (either Gaussian or Box). The filter function accepts a distance value that returns a value between [0,1] (both included).
For this function (`PaintInRangeGeodesic`), this distance will be the sum of edge distances to that vertex from $v_{vertexId}$. To clarify; we will use the graph in Figure 1. Lets assume this function is called with $vertexId = 0$ and $maxDepth = 2$. Assuming filter function is $f(x)$, color of the value of "Vertex4" will be multiplied by the $f(2.4 + 9.2)$. (Why it is not multiplied by the $f(2.9 + 1.3)$ will be explained in the remarks section.

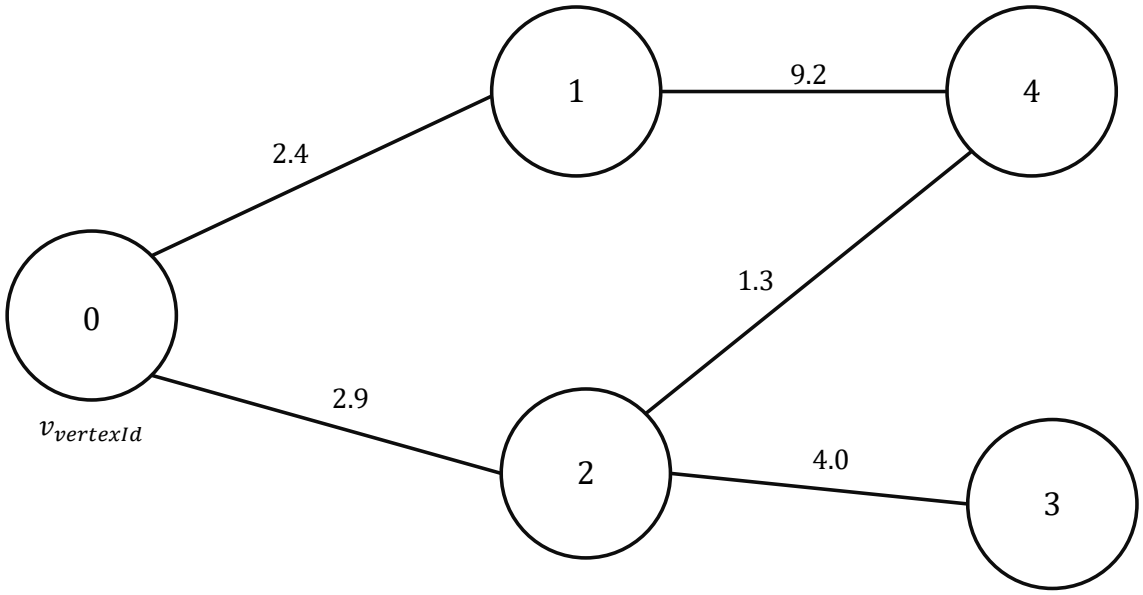

Figure 1: An example graph for illustration purposes. Each circle represents a vertex. Vertex positions are not shown, but calculated distances between vertices are over the appropriate edges. Numbers inside the vertices represent the vertex ids of that vertex.

Filter functions have internal parameter namely α. This function changes the behavior of the filtering function. Filter functions are defined on equations 3 and 4.

$$f(x) = e^{-\frac{x^2}{\alpha^2}} \tag{3}$$

For the Gaussian filter function, you can assume α is never zero. Minimum value of α for this case is 0.01.

$$f(x) = \begin{cases} 1 & -\alpha \leq x \leq \alpha \\ 0 & otherwise \end{cases} \tag{4}$$

We have used a graphing calculator to show how the α parameter changes the shape of the filtering functions so that you can understand it better. Graphs of these functions can be found here.

Remarks

- In order to be consistent between algorithms, your breadth-first algorithm should add items to the queue in vertexId order (minimum first).

  To further clarify, let us use Figure 1. Doing a classic breadth-first search; we add v_0 to the queue. We pop it to do our coloring work and then we add all of the unchecked (nothing is checked at the moment) neighbors of v_0 to the queue (don't forget to maintain distances of the added vertices here). While doing this we should add v_1 first, then v_2 since $1 < 2$. Since v_1 is added first. It will come out of the queue first. Then we add v_4 to the queue (and when we calculate the distance of the v_4 we add d_{0-1} and d_{1-4}).

- For exponentiation, you can use `std::exp(...)` function defined in the `<cmath>` header. For an explanation, check here.

- Again for consistency, you should change the color value to double type (you can use double3 in this case), do your calculations (find the filter value and multiply the r, g, and b with this value); and finally, you need to convert the double values to the unsigned char. While doing this **round-down the value to the nearest integer**.

  To further clarify, the argument "color" is `[255,0,0]`, and a vertex did calculate its filter value `0.25`. Resulting color is `[63.75,0.0,0.0]`. Now we round down the value and the actual color will be (which is unsigned char type) `[63,0,0]`.

- Just like in section 2.4.8, this function should return an array for **all** vertices, not the vertices that had a resulting color. All other colors should be black.

- If `vertexId` does not in the graph. This function should return empty `outputColorAllVertex`.

2.4.10 `void PaintInRangeEuclidian(std::vector<Color>& outputColorAllVertex, int vertexId, const Color& color, int maxDepth, FilterType type, double alpha) const;`

This function is highly similar to the function described in Section 2.4.9 except the distance value. Distance value does not rely on the edges between `vertexId` and the vertex that is being processed. Instead; distance is the direct euclidean distance between $v_{vertexId}$ and v_n.
To further clarify using Figure 1; distance value between v_0 and v_4 (for this function) will **not** be $2.9+1.3 = 4.2$ but it will be the direct calculation of `Distance(v0.position3D, v4.position3D)`. Because of that, you do not need to maintain distance. When you reach a vertex, distance can be calculated using the positions.
The rest of the function is exactly the same as `void PaintInRangeGeodesic(...)`.

2.4.11 `static void WriteColorToFile(const std::vector<Color>& colors, const std::string& fileName);` `static void PrintColorToStdOut(const std::vector<Color>& colors);`

These print functions can be used to print the color values either to a file or to the console. The file print function can be used in conjunction with the visualizer to see the results of your algorithm. **These functions are already implemented for you.**

Regulations

1. **Programming Language:** You will use C++.
2. Standard Template Library (STL) is not allowed except "`std::vector`", "`std::list`", "`std::string`" as data structures. Math functions such as "`std::exp`" and "`std::sqrt`" are allowed.
3. "`std::priority_queue`" and "`std::queue`" should strictly never be used.
4. Using external libraries other than those already included is not allowed.
5. Changing or modifying already implemented functions are not allowed
6. You can add any private member functions unless it is explicitly stated that you should not.
7. **Late Submission Policy:** Each student receives 5 late days for the entire semester. You may use late days on programming assignments, and each allows you to submit up to 24 hours late without penalty. For example, if an assignment is due on Thursday at 11:30pm, you could use 2 late days to submit it on Saturday by 11:30pm with no penalty. Once a student has used up all their late days, each successive day that an assignment is late will result in a loss of 5% on that assignment.

   No assignment may be submitted more than 3 days (72 hours) late without permission from the course instructor. In other words, this means there is a practical upper limit of 3 late days usable per assignment. If unusual circumstances truly beyond your control prevent you from submitting an assignment, you should discuss this with the course staff as soon as possible. If you contact us well in advance of the deadline, we may be able to show more flexibility in some cases.
8. **Cheating:** We have zero tolerance policy for cheating. In case of cheating, all parts involved (source(s) and receiver(s)) get zero. People involved in cheating will be punished according to the University Regulations. Remember that students of this course are bounded to the code of honor and its violation is subject to severe punishment.
9. **Newsgroup:** You must follow the Forum (`odtuclass.metu.edu.tr`) for discussions and possible updates on a daily basis.

Submission

- Submission will be done via CengClass, (`cengclass.ceng.metu.edu.tr`).
- Don't write a "main" function in any of your source files. It will clash with the test case(s) main function and your code will not compile.
- A test environment will be ready in CengClass :
  - You can submit your source files to CengClass and test your work with a subset of evaluation inputs and outputs.
  - Additional test cases will be used for the evaluation of your final grade. So, your actual grades may be different than the ones you get in CengClass.
  - Only the last submission before the deadline will be graded.