Middle East Technical University - Department of Computer Engineering

CENG 213

Data Structures

Fall '2023-2024

Programming Assignment 3

Due Date: 1 January 2023, Monday, 23:55
Late Submission Policy will be explained below

Objectives

In this programming assignment, you are expected to implement the functional equivalent of a flight finder application (i.e., flightsfinder.com, skyscanner.net etc.). To implement such an application, you will implement *MultiGraph* and *Hash Table* data structures. Details of these data structures will be explained in the following sections.

Keywords: C++, Data Structures, Graphs, Dijkstra Shortest Path Algorithm, Hash Table, Quadratic Probing

1 MultiGraph Implementation (40 pts)

The MultiGraph data structure used in this assignment follows the adjacency list implementation style with slight differences. Graph Data Structure has two non-negative weights for its edges. Additionally, edges can be *parallel*, meaning that each vertex pair can have multiple edges with different weights. This essential makes the graph data structure a **multigraph**. The multigraph **is** a directional graph. Finally, each edge has a string identifier that will be used to express the airline name. Additional functionality consists of issuing the shortest path algorithm with the specific airline(s), heuristically combining two weights of edge for the shortest path algorithm. The data layout of the MultiGraph class and its helper structures can be seen on Listing 1.

Instead of a linked list, the graph holds its data (in this case, edges and vertices) in dynamic arrays. For dynamic array implementation, the class utilizes `std::vector<T>` class from the *Standard Template Library (STL)*. Instead of pointers, indices of these vectors will be utilized for all of the internal references. This enables the graph to be movable without invalidating all of the intra-references and reduces data duplication. Graph class and its helper structures are declared in *MultiGraph.h* header file, and their implementations (although most of it is empty) are defined in *MultiGraph.cpp* file.

1.1 GraphVertex Struct

GraphVertex structure holds the information about a vertex. A GraphVertex instance holds *edges* for which edges originated *from* this vertex. Finally, *name* variable holds the unique name of that vertex.

```cpp
struct GraphEdge
{
    std::string name;       // Name of the vertex
    float       weight[2];  // Weights of the edge
                            // (used on shortest path)
    int         endVertexIndex;
};

struct GraphVertex
{
    std::vector<GraphEdge> edges; // Adjacency List
    std::string            name;  // Name of the vertex
};

class MultiGraph
{
    private:
    std::vector<GraphVertex>    vertexList;

    ...
};
```

Listing 1: MultiGraph and Helper Structures data layout.

1.2 GraphEdge Struct

GraphEdge structure holds the information about an edge. *weight* static array holds the weights of the edge. Finally, *name* holds the edge's unique name and *endVertexIndex* holds the vertex index that this edge connects to.

The property *endVertexIndex* is the index of the *Graph::vertexList* dynamic array.

1.3 MultiGraph Class

MultiGraph class implements a directional, non-negative weighted multigraph. It allows multiple edges between two vertices. However, it supports *unique-named edges* between two vertices. For example, given vertices v_0 and v_1 and an edge name e_n, there can only be one edge named e_n between $v_0 \rightarrow v_1$. Since the graph is directional, an edge $v_1 \rightarrow v_0$ with a name e_n can also be defined.

The graph utilizes a single dynamic array for its data, capable of some functionalities that will be explained shortly.

1.3.1 `MultiGraph();`

The default constructor constructs an empty graph. It relies on the default constructors of the std::vector.

This function is implemented for you.

1.3.2 `MultiGraph(const std::string& filePath);`

This constructor reads a ".map" file and constructs the Graph data structure using the information in the file.

This function is implemented for you however; it relies on *`InsertVertex(...)`* and *`InsertEdge(...)`* functions.

1.3.3 `void InsertVertex(const std::string& vertexName);`

This function tries to insert a vertex named "vertexName" without any edge. However, it should throw **"DuplicateVertexException"** if the same-named vertex already exists on the graph.

1.3.4 `void RemoveVertex(const std::string& vertexName);`

Unlike the function above, this function removes a vertex "vertexName" from the graphs. **"VertexNotFoundException"** is thrown when the vertex does not exist. It should be noted that the removal operation will invalidate some of the index-based references of edges. These should be resolved before the function is completed.

1.3.5 `void AddEdge(const std::string& edgeName, const std::string& vertexFromName, const std::string& vertexToName, float weight0, float weight1);`

This function tries to set an edge between two vertices, namely "vertexFromName" and "vertexToName". If either of these vertices does not exist, the function should throw **"VertexNotFoundException"**. If an edge named "edgeName" exists between these two vertices the function should throw **SameNamedEdgeException**.

1.3.6 `void RemoveEdge(const std::string& edgeName, const std::string& vertexFromName, const std::string& vertexToName);`

This function tries to remove an edge between two vertices, namely "vertexFromName" and "vertexToName". If either of these vertices does not exist, the function should throw **"VertexNotFoundException"** and **"EdgeNotFoundException"** should be thrown when either vertices or the edges are not available on the graph.

1.3.7 `bool HeuristicShortestPath(std::vector<int>& orderedVertexEdgeIndexList, const std::string& vertexNameFrom, const std::string& vertexNameTo, float heuristicWeight) const;`

Unlike traditional graphs, this graph implementation will have multiple weights for its edges. Users can set the "heuristicWeight" variable (α) to mix the edge weights w_0 and w_1 and find the weight β that will be used for shortest path calculations. Equation 1 shows a combination function.

$$\beta = w_0\alpha + w_1(1 - \alpha) \tag{1}$$

If the shortest path exits, the function returns true, otherwise, it returns false. The actual path should be written on the variable "orderedVertexEdgeIndexList". This variable holds indices of the shortest path starting from the index of "vertexNameFrom" and ends with the index of "vertexNameTo". Since the class is a multigraph, only vertex indices would not suffice for a valid path. To this end, there is an edge index between vertices. In an example, a valid path between v_3 and v_5 can be $v_3 \rightarrow (e_3) \rightarrow v_7 \rightarrow (e_2) \rightarrow v_2 \rightarrow (e_0) \rightarrow v_5$. In this case, edge indices are the indices of the originated vertex's edge vector.

> Please check the implemented `void MultiGraph::PrintPath(...)` function for further explanation. This function reads and prints the path that is represented by the resulting vector to the console.

Finally, this function should throw **VertexNotFoundException** when appropriate. In any case, this function fails, or it returns a false output array that should not be modified.

1.3.8 `bool FilteredShortestPath(std::vector<int>& orderedVertexEdgeIndexList, const std::string& vertexNameFrom, const std::string& vertexNameTo, float heuristicWeight, const std::vector<std::string>& edgeNames) const;`

This function is highly similar to the function above. However, when calculating the shortest path, this function disregards the edges that their name is on the list "edgeNames". Otherwise, the behavior is the same.

Remarks for Shortest Path Algorithms

- You **are** allowed to use *`std::priority_queue`* STL library class. A template instantiation example is given in `IntPair.h` header file. By default, `std::priority_queue` is a *max heap* and you should convert it to a min heap.
- 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. Do iterate arrays in order when required.
- Still some inconsistencies may occur depending on the heap implementation. Because of that, comparisons should be using the "less than" operation inside of the heap instead of "less than and equal".

1.3.9 `int BiDirectionalEdgeCount() const;`

This function returns the bidirectional edge count in the graph. Bidirectional edge is defined as follows: Given vertices v_0 and v_1; if an edge named e_n exists between $v_0 \rightarrow v_1$ and $v_1 \rightarrow v_0$, there is a bidirectional edge. The total count of these pairs should be the output of this function.

1.3.10 `int MaxDepthViaEdgeName(const std::string& vertexName, const std::string& edgeName) const;`

Given the edge name "edgeName" and vertex name "vertexName" return the maximum depth that can be reached from "vertexName" using only "edgeName" named edges. If the vertex is not available on the graph, the function should throw **VertexNotFoundException**. The function should return zero when no other vertex can be reached from the starting vertex.

> You may use either DFS or BFS for this functionality. Iterative implementation will require a queue or stack. You can repurpose the `std::priority_queue` for this purpose.
>
> Think about strictly increasing/decreasing priorities.

1.3.11 `void PrintPath(const std::vector<int>& orderedVertexEdgeIndexList, float heuristicWeight, bool sameLine) const;`

Prints the path between vertices `list.front()` and `list.back()`. The "orderedVertexEdgeIndexList" variable should be the result of a shortest path function. The function will neatly print provided weights using the "heuristicWeight" variable. If *sameLine* is true, this function prints on a single line

This function is implemented for you however; it relies on *`Lerp(...)`* function.

1.3.12 `void PrintEntireGraph() const;`

This function prints the entire graphs. It is provided for you for debugging purposes. **This function is implemented for you.**

1.3.13 `static float Lerp(float w0, float w1, float alpha);`

This function should behave as described in Equation 1. The function *`void PrintPath(...)`* will use this function to combine the weights of the edges.

> You may want to use this function on your shortest-path implementations as well.

1.4 Exceptions

All of the exceptions for the graph are provided for you in the `Exceptions.h` header file. All of these exceptions are implemented for you. You can check the file to how to utilize these exception classes.

2 Hash Table (30 pts)

In addition to the Graph data structure, flight application requires a hash table for efficiency. We assume the shortest path algorithm takes a long time and we store the most-asked flight paths instead of calculating these over and over again. You are going to implement compile-time sized *HashTable* class which stores key and value (data) pairs. The key parameters will be used to determine the location of the data using a hash function. Unlike other hash tables, each entry has a least recently used (LRU) counter which will be used to evict table entries when the table is at full capacity. The data layout of the *HashTable* can be seen on Listing 2. The hash table will resolve its collisions using **quadratic probing**.

KeyedHashTable class and its helper structures are declared in *HashTable.h* header file and their implementations are defined in another header file namely *HashTable.hpp* due to this class being a template class.

2.1 HashTable Class

HashTable class has the main implementation. This class holds its data as a static array. The size of the array is given as a template parameter "MAX_SIZE". The hash table never increases in size, but only half of it can be filled at maximum.

2.1.1 `static int Hash(int startInt, int endInt, bool isCostWeighted);`

This function should be utilized for hashing, in which two integers and a single boolean value are used to calculate the hash. The hash function multiplies the input with primes in the array "PRIMES". Given prime array $P = p_0, p_1, p_2$ and the input set $S = start_i, end_i, cost$; the resulting hash h is given in Equation 2.

$$h = \sum_{i=1}^{3} P(i)S(i) \tag{2}$$

In this calculation "isCostWeighted" is considered 1 if it is true and zero if it is false.

2.1.2 `void PrintLine(int tableIndex) const;`

This function neatly prints a single table entry over the standard output. **This function is implemented for you.**

2.1.3 `HashTable();`

The constructor initializes the hash table. For each entry, it sets the lruCounter to zero and marks every spot on the table as empty.

```cpp
// Sentinel for probing
#define SENTINEL_MARK 0xFFFFFFFF
#define EMPTY_MARK    0xFFFFFFFE
#define OCCUPIED_MARK 0x00000000
// Capacity threshold is the multiplicative inverse of the 1/2 (%50)
#define CAPACITY_THRESHOLD 2

struct HashData
{
    // Data
    std::vector<int> intArray;
    unsigned int     sentinel;
    // Key
    bool             isCostWeighted;
    int              startInt;
    int              endInt;
    // LRU Counter (to determine least recently used entry)
    int              lruCounter;
};

template <int MAX_SIZE>
class HashTable
{
    private:
    static int  PRIMES[3];

    // Properties
    HashData    table[MAX_SIZE];
    int         elementCount;

    ...
};
```

Listing 2: Hash Table Structures

2.1.4 `int Insert(const std::vector<int>& intArray, bool isCostWeighted);`

This function adds an entry to the hash table. First, it tries to find an element that has the same key. Keys are defined by the first and last element of the "intArray" variable and the "isCostWeighted" variable. If there is a same-keyed entry on the table, the function increments its `lruCounter` instead of inserting it. If this insertion defines a unique entry to the table, the function checks the table capacity and throws **TableCapFullException** if the table is full. If the table has space it inserts to the appropriate position.

To generate keys, "intArray" should at least have a single element. If this is not true, the function should throw **InvalidTableArgException**.

As discussed above, collisions are resolved via quadratic probing. The quadratic probing function should be similar to the one discussed in class. However; it is given in Equation 3 due to completeness.

$$q(x) = hash(\cdots) + x^2 \tag{3}$$

The function returns the pre-modified `lruCounter` of the returned item. If the inserted item is new, it should return zero.

2.1.5 `bool Find(std::vector<int>& intArray, int startInt, int endInt, bool isCostWeighted, bool incLRU = false);`

The find function tries to find the entry uniquely defined by the "startInt", "endInt" and "isCostWeighted" variables. It returns true if the entry exists, or returns false when it is unavailable. Function increments the `lruCounter` when the "incLRU" variable is true.

2.1.6 `void Remove(std::vector<int>& intArray, int startInt, int endInt, bool isCostWeighted);`

Similar to the function above, the remove function finds the entry removes it, and returns entries data. If this uniquely-defined entry does not exist it silently returns. In this case, the resulting "intArray" should not be modified.

2.1.7 `void RemoveLRU(int lruElementCount);`

This function removes the top "lruElementCount" items from the table. Lowest `lruCounter` valued items should be removed from the table.

2.1.8 `void InvalidateTable();`

Similar to the constructor, this function removes every entry from the table. Mark the spots as empty and delete the data of the present entries.

2.1.9 `void GetMostInserted(std::vector<int>& intArray) const;`

This function returns the data of the most inserted (or found) entry. In essence, it should return the highest `lruCounter` valued item in the table

2.1.10 `void PrintSortedLRUEntries() const;`

This function prints the entries with non-zero `lruCounter` in a high-to-low fashion.

> Please use the provided `PrintLine` function for printing due to consistency between tests and your code.
>
> To sort the data, you may want to use `std::priority_queue`.

2.1.11 `void PrintTable() const;`

Prints the entire table into the standard output. **This function is implemented for you.**

3 CENG Flight Finder (30 pts)

"CENGFlight" class combines the MultiGraph data structure and HashTable data structure. The data layout of the class can be seen on Listing 3. Continuing with the same pattern; METUMaps class is declared on *CENGFlight.h* header file and its definitions are in *CENGFlight.cpp* file.

3.1 CENGFlight Class

CENGFlight class mostly delegates the functionality to the MultiGraph and/or HashTable. However, some extra functionalities do also exist. CENGClass should not throw any exceptions, exceptions should be handled internally and appropriate print should be displayed on the console. In order to achieve consistency, the print functions (Section 3.1.1) are already provided for you.

```
struct HaltedFlight
{
    std::string airportFrom;
    std::string airportTo;
    std::string airline;
    float w0;
    float w1;
};

class CENGFlight
{
    private:
    HashTable<FLIGHT_TABLE_SIZE> lruTable;
    MultiGraph                   navigationMap;

    ...
};
```

Listing 3: METUMaps Class

Throughout the class, vertex names are given as airport names. Edge names are given as airline names. Edge's first weight is the cost (flight ticket price of that specific flight) and the second weight is the time (although it is not important, it is in minutes) that this specific flight took.

Users can query optimal flights by selecting start/end airports and some weighting factor between cost and price. CENG Flight System will return paths corresponding to the users' needs.

3.1.1 Print Functions

There are many print operations in this class. To provide consistency, all print operations are provided as functions. These functions are not explained here but all of which are self-explanatory. You should use these functions for printing.

These functions are:

- `void PrintCanNotHalt(const std::string& airportFrom, const std::string& airportTo, const std::string& airlineName);`
- `void PrintCanNotResumeFlight(const std::string& airportFrom, const std::string& airportTo, const std::string& airlineName);`
- `void PrintFlightFoundInCache(const std::string& airportFrom, const std::string& airportTo, bool isCostWeighted);`
- `PrintFlightCalculated(const std::string& airportFrom, const std::string& airportTo, bool isCostWeighted);`
- `void PrintPathDontExist(const std::string& airportFrom, const std::string& airportTo);`
- `void PrintSisterAirlinesDontCover(const std::string& airportFrom);`

3.1.2 `CENGFlight(const std::string& flightMapPath);`

This function directly delegates its argument "flightMapPath" to the graph, and initializes the hash table. Initially, the hash table is empty.

3.1.3 `void HaltFlight(const std::string& airportFrom, const std::string& airportTo, const std::string& airlineName);`

Sometimes a flight is not available due to plane maintenance. In this case, the edge with the corresponding should temporarily be removed from the flight map. This function should try to remove the edge defined by the airport names and the airline name. If such an edge exists CENG Flight System should temporarily store the names and weights of this particular edge in a data structure.

> We did not specify the actual temporary data structure, you can use any data structure you want. We do provide `HaltedFlight` structure for you to use, however.

If such an edge or airport does not exist, the function should print `PrintCanNotHalt(...)`.

3.1.4 `void ContinueFlight(const std::string& airportFrom, const std::string& airportTo, const std::string& airlineName);`

Similar to the function above this function re-enables previously halted flight. If this flight either does not exist or it is not halted, it should print `PrintCanNotResumeFlight(...)`

3.1.5 `void FindFlight(const std::string& startAirportName, const std::string& endAirportName, float alpha);`

Prints the shortest flight with the given parameters. First, it checks the hash table if a path has already been calculated before. If so, it prints `PrintFlightFoundInCache(...)`. If the flight does not in the cache function calculates the path and prints `PrintFlightCalculated(...)` and then prints the actual path using the graph function `PrintPath(...)` function. In this case, the path should be printed on multiple lines. If a path does not exist, this function should print `PrintPath DontExist(...)`.

Caching only occurs when the given flight is either fully cost or time-weighted. Please check Section 3.2 for details.

3.1.6 `void FindSpecificFlight(const std::string& startAirportName, const std::string& endAirportName, float alpha, const std::vector<std::string>& unwantedAirlineNames) const;`

Similar to the function above, this function only finds the shortest path with the given parameters. "unwantedAirlineNames" is provided to filter the airlines that user **do not want** to travel with. These paths are **not cached** since they are highly user-specific. This function should print `PrintFlightCalculated(...)` before printing the actual flight path. Flight path printing should use the graph's functionality and nothing else.

3.1.7 `void FindSisterAirlines(std::vector<std::string>& airlineNames, const std::string& startAirlineName, const std::string& airportName) const;`

The sister airlines are defined as follows: A sister airline set $S = a_0, a_1, \cdots a_n$ entire navigation map can be traversed by only using these airlines. There can be many sister airlines for a given navigation map. To achieve consistency between implementations, the algorithm should:

- Start visiting every airport from "airportName" using "startAirlineName"
- If at least one other airport is reached, continue the operation if not print the error.

  Do
  - Select a visited airport. This airport should have the maximum non-visited neighbors. (If there is a tie use the lower indexed one).

- Select the first non-utilized airline on that visited airport (this is the lowest-indexed non-utilized airline). If all airlines from that airport are on the sister airline list, print the error.
- Visit every non-visited airport only using this airline. Add this airline to the sister airline list.

Until all airports are visited or your visited airports did not change from the last iteration.

- If the visited airport list did not change compared to the previous iteration, print the error.
- Finally, copy the found sister airlines to "airlineNames" and return.

The above algorithm indices are the indices of the edgeList or vertexList respectively. Although this function is defined in the CENGFlight class it should delegate the arguments directly to the graph and the graph should handle the operation. "The error" above is `PrintSisterAirlinesDontCover(...)` function. The output "airlineList" should only be modified if there is no error.

If the airport named "airportName" is not on the navigation map, print `PrintSisterAirlinesDontCover(...)` as well.

3.1.8 `int FurthestTransferViaAirline(const std::string& airportName, const std::string& airlineName) const;`

This function should return the maximum transfer amount (depth) that can be done by only using a single airport. If the airport does not exist, this function should return -1;

3.1.9 `void PrintMap();`

Prints the entire graph for debugging purposes. **This function is implemented for you.**

3.1.10 `voidPrintCache();`

Similar to the function above, this function prints the cache. **This function is implemented for you.**

3.2 Caching

When the shortest path is found with an *alpha* either **zero or one**, the CENGFlight class should cache these onto the hash table. LRU entry on the table should be removed if and only if the table is full. Otherwise, the empty space should be utilized. The key value of the hash table is the start and end indices of the path (airport indices) and false if the *alpha* is zero or true when it is one. Blended paths (where $0 < \alpha < 1$) **should not be cached**. **Specific flights should never be cached.**

Regulations

1. **Programming Language:** You will use C++.
2. Standard Template Library is **not** allowed except "`std::priority_queue`" "`std::string`" and "`std::vector`". "`std::priority_queue`" should only be used on the appropriate functions as a temporary data structure.
3. **You are strictly forbidden to use "`std::queue`" and "`std::stack`" classes of the STL**. There is a neat trick to convert a heap to a stack or queue. Please utilize "`std::priority_queue`" for this purpose.
4. Using external libraries other than those already included is not allowed.
5. Changing or modifying already implemented functions is 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 a 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 bound 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 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.