HW4

Due: Wed. Jun. 17 by 11:55 pm

This assignment will be marked out of 30 points. *This is an individual assignment. Please review the rules on academic misconduct on the course D2L page.*

This assignment consists of two parts. The first part consists of problems that will give you more practice working with hash tables and sorting algorithms. The second part consists of a coding exercise that asks you to implement a shortest path algorithm for a graph. You will also get some practice working with multiple source files and Java applets.

Part I

1. (2 points)
   Consider an open-address hash table with simple uniform hashing. Use Knuth's formulas to determine upper bounds on the expected number of probes in an unsuccessful search and on the expected number of probes in a successful search when the load factor is 3/4 and when it is 7/8. Consider all three schemes discussed in class namely: linear probing, quadratic probing, and double hashing. Tabulate your results for comparison.

2. (5 points)
   This question deals with the heapsort algorithm shown below.

```
function Heapsort(arr)
    Heapify array arr;
    for i = n − 1 down to 1 do
        swap arr[i] with arr[0];
        restore heap property for the tree arr[0], . . . , arr[i − 1] by percolating down the root;
    end
end
```

   - Identify a suitable loop invariant that will help you show the correctness of heapsort.
   - Show that your loop invariant holds by showing the initialization and maintenance steps. You can assume that the *percolate down* operation is correct.
   - Use your loop invariant to show the partial correctness of heapsort.

3. (5 points)
   This question deals with the quick sort algorithm.

   (a) Suppose that we modify the partitioning algorithm so that it always partitions an input array of length n into two partitions in such a way that the length of the left partition is $n - K$ and the length of the right partition is $K - 1$ (for some **constant** K, where $K > 0$). Let us refer to this partitioning algorithm as `KPartition`.

   Now consider the following variation of quick sort called **KQuickSort**:

```
function KQuickSort(int[] A, int low, int high)
    n = high - low + 1;
    if n < K then
        insertionSort(A, low, high) ;
    else
        q = KPartition(A, low, high) ;
        KQuickSort (A, low, q-1) ;
        KQuickSort (A, q+1, high) ;
    end
end
```

Let $T(n)$ denote the worst-case number of steps to run `KQuickSort` on input arrays of size n. Complete the following recurrence relation for $T(n)$. Assume that `KPartition` takes $\Theta(n)$ steps to partition an array of size n.

$$T(n) \leq \begin{cases} & n < K, \\ & n \geq K. \end{cases}$$

(b) Use the iterative substitution method to find an upper bound on $T(n)$. For simplicity, assume that n is a multiple of K, i.e. $n = m \cdot K$ (where m is a non-negative integer). State your answer using an appropriate asymptotic notation.

(c) How does the worst-case asymptotic running time of `KQuickSort` compare with that of quick sort?

4. (3 points)
Prove that a tree with n vertices has $n - 1$ edges.

Submission

Please handwrite or typeset your solutions to these problems. If hand-writing, please write neatly and make sure that your responses are clearly visible in your scanned/photographed images.

Please submit via gradescope to the assignment entitled 'HW4_Written'. Gradescope will accept both PDF and image file formats. Allow yourself plenty of time before the deadline to ensure a smooth submission process.

Part II

For this part, you will implement Dijkstra's shortest path algorithm to determine both weighted and unweighted shortest paths (if any) between two nodes in a maze.

Introduction

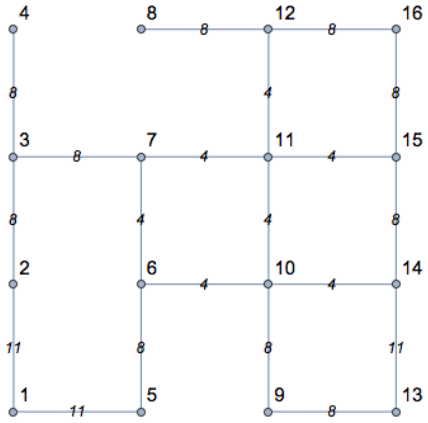

Figure 1: A 4×4 grid graph.

You are given a maze that is represented as a graph consisting of an $n \times n$ grid of vertices. Vertices in the maze are connected via *weighted* undirected edges to form paths as shown in Fig. 1.

The vertices are numbered from 1 to n^2 in a column-major fashion as shown above. The connectivity in the graph is represented in a text file. The first line of this text file consists of the number n and the rest of the lines consist of three integer entries per line in the format:

```
<fromVertex> <toVertex> <weight>
```

In other words, each line after the first corresponds to an edge in the graph and there are as many entries as the number of edges. The edges are bidirectional, i.e. each edge appears twice in the file.

For the example graph pictured in Fig. 1, please see the file `maze.txt`. The first few lines from this file are:

```
4
1 2 11
1 5 11
2 1 11
2 3 8
3 2 8
3 4 8
3 7 8
4 3 8
5 1 11
...
```

Shortest Path using Dijkstra's Algorithm

Your task is to write a class called `GridGraph` that implements the following interface:

```
public interface GridGraphInterface {

  /**
   * Builds a grid graph from a specified file. It is assumed
   * that the input file is formatted correctly.
   *
   * @param filename
   */
  public void buildGraph( String filename ) throws FileNotFoundException;

  /**
   * Finds the shortest path between a source vertex and a target vertex
   * using Dijkstra's algorithm.
   * @param s Source vertex (one based index)
   * @param t Target vertex (one based index)
   * @param weighted Whether edge weights should be used or not.
   * @return A String encoding the shortest path. Vertices are
   *         separated by whitespace.
   */
  public String findShortestPath( int s, int t, boolean weighted );
}
```

1. The method `buildGraph()` should read connectivity information from a file and build a graph. The choice of which representation you use — adjacency list or adjacency matrix — is up to you. You may assume that the input file conforms to the format above.

2. The method `findShortestPath()` should determine the shortest path (either weighted or unweighted) from a source vertex to a target vertex using Dijkstra's algorithm.

   - Source and target vertices are specified as integers using a one-based indexing (see Fig. 1).
   - Whether edge weights are used or not is specified as a boolean flag. If this flag is true, a weighted shortest path should be computed, otherwise, an unweighted shortest path should be computed.
   - For unweighted paths, your implementation should use Dijkstra's algorithm with the same edge weights for all the edges.
   - *Adjacent vertices should be enqueued in ascending order of vertex number.*
   - The determined shortest path should be returned as a String which consists of vertices along the path starting from the source vertex and ending at the target vertex. The vertices should be separated by whitespace.
     If there is no shortest path from the supplied source vertex to the supplied target vertex, an empty string should be returned.

As an example, consider the source and target vertices `1` and `12` in Fig. 1.
For this pair, the shortest *unweighted* path is: `"1 2 3 7 11 12"` (see Fig. 2 (left)).
For this pair, the shortest *weighted* path is: `"1 5 6 7 11 12"` (see Fig. 2 (right)).

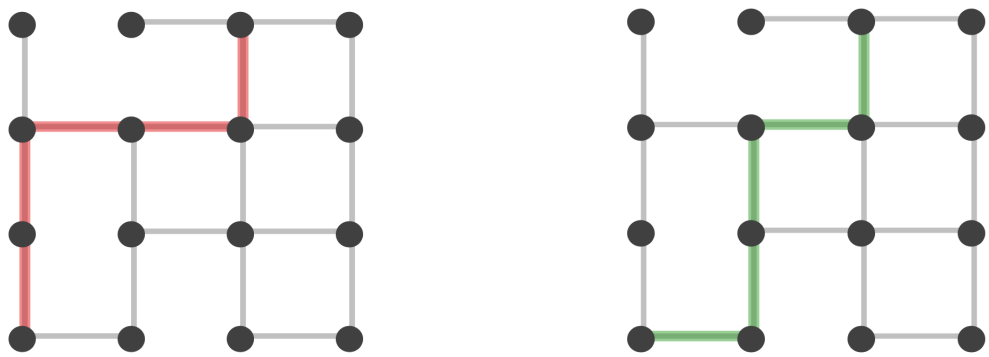

Figure 2: Unweighted (left) and weighted (right) shortest path between vertices `1` and `12`.

Files Provided

The following files are provided to you on D2L:

1. `GridGraphInterface.java` - The interface shown above.
2. `GridGraph.java` - A class skeleton which consists of method stubs that you are required to implement.
3. `GridGraphVisualizer.java` - A Java applet that provides methods to add edges and paths to be visualized. Multiple paths can be visualized simultaneously. You can use this class to help debug and test your implementation. The main method in `GridGraphVisualizer` includes relevant calls that demonstrate its use. You can use `GridGraphVisualizer` to visualize an input maze and overlay paths on it. The visualizations shown in Fig. 2 were generated using `GridGraphVisualizer`.
4. `maze.txt` - The input file corresponding to the maze shown in Fig. 1.

Additional Requirements

- If you need to make use of auxiliary data structures, please feel free to make use of Java collection classes. The implementation of Dijkstra's shortest path algorithm must be your own. The use of any other code that is not your own is not allowed.
- Please pay attention of the packaging structure of the supplied files and don't alter it or your code will not compile and run on gradescope.
- You may use as many helper methods and classes as you need. However, please try to encapsulate all your code within the file `GridGraph.java`. If for any reason, you need to split your implementation into multiple files, please ensure that all required files reside in the package `hw4.student`.

Documentation and Testing

- For all classes and methods that you add, please provide Javadoc comments to explain what the classes and methods are doing.
- Please use the supplied maze file and the figure above to test your program. Additional test files will be made available closer to the due date. Your program should be able to infer the number of vertices in the maze from the maze file. Please do not hard code this information as you will be required to test with larger mazes.
- You may find `GridGraphVisualizer` helpful when testing and debugging your implementation.

Submission and Grading

Your implementation will be tested by an autograder on Gradescope. When you submit, the autograder will run some unit tests and grade your submission based on functionality.

- Please submit via Gradescope to the assignment entitled 'HW4_Coding'.
- Please submit the file `GridGraph.java` and any other files it depends on. Please ensure that the package declaration `hw4.student` is intact. Submit the file(s) directly, do not compress into an archive.
- Please include any citations as comments at the the top of the submitted file(s).
- Once submitted, the autograder will compile and test your implementation providing you with output that shows the test results.
- You may submit multiple times. Only the most recent submission will be kept.
- You will be graded primarily based on functionality as tested by the autograder. The autograded score is subject to adjustments at the discretion of the marker.