HW4

Due: Sun. Jun. 24 by 11:55 pm

This assignment will be marked out of 35 points. You also have an opportunity to earn 5 bonus 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. Hash tables (5 points)

   (a) Consider an open-address hash table with uniform hashing. Give 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.

   (b) Suppose that in the separate chaining scheme, we also keep the lists in sorted order. With this modification, what is the running time for successful searches, unsuccessful searches, insertions, and deletions? State your answers in terms of the load factor α.

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. (7 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 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).

(c) Using the Big-O notation, give an asymptotic expression for the worst-case running time of `KQuickSort`. A proof is not required.

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

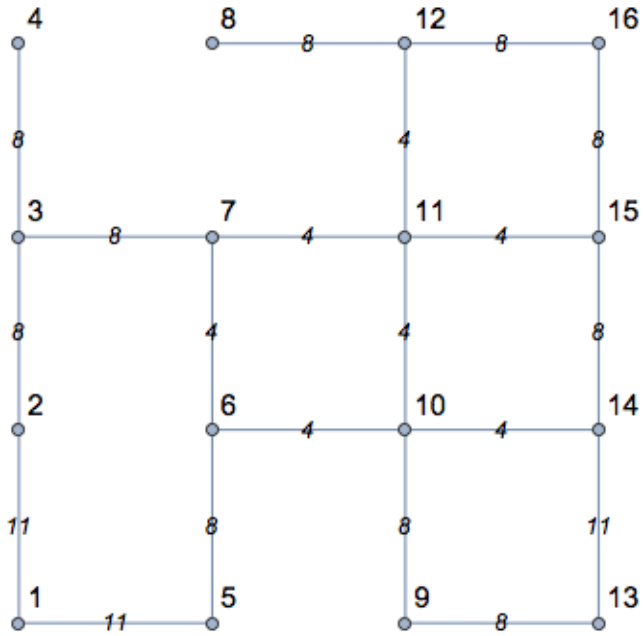
4. (3 points)
Prove that a tree with n vertices has $n - 1$ edges.

Part II

In this coding exercise, you will implement a shortest path algorithm that finds the shortest path between two nodes (if any) in a maze. You can *either* use a breadth first search (BFS) to find the shortest unweighted path, *or* (for an added bonus) use Dijkstra's shortest path algorithm to determine both weighted and unweighted shortest paths.

Introduction

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 the figure below.



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 above, 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 Unweighted Path

For this part, you can ignore the edge weights.

Your task is to determine the shortest path from a source vertex to a target vertex using a breadth first traversal. Source and target vertex pairs will be specified via an input query file that consists of one pair per line. For each pair, please determine the shortest path from the source to the target and use the the supplied `MazeVisualizer` to visualize the result. If there is no path between a given pair, please indicate that by printing out an appropriate message.

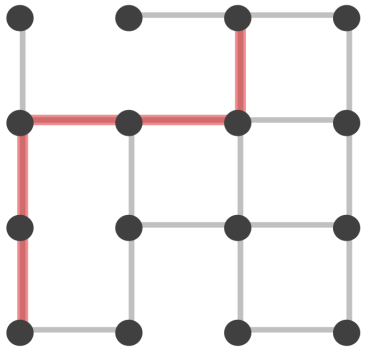
Each line in the input query file has the following format:

```
<source vertex> <target vertex>
```

For example, for the source and target vertices `1` and `12` in the figure above, the input line is:

```
1 12
```

and the result is pictured below.



Please write a Java program that performs the following tasks:

- Reads the maze information from an input file and uses an appropriate data structure to store this information.

- Reads source and target vertex pairs from an input query file. For each pair, perform a BFS to find the shortest path (ignoring edge weights) from the source to the target. Adjacent vertices should be enqueued in ascending order of vertex number. *Alternatively, if you are implementing the bonus portion (see below), your program should implement this using Dijkstra's algorithm with unit edge weights.*

- Visualizes the paths using the supplied `MazeVisualizer`. `MazeVisualizer` is a Java applet that provides methods to add edges and paths to be visualized. Multiple paths can be visualized simultaneously. The main method in `MazeVisualizer` includes relevant calls that demonstrate its use.

Your program will be invoked from the command line as follows:

```
java HW4 maze-file query-file
```

Please use the supplied maze file and the figure above to test your program. Additional test files and associated output 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.

Unlike previous assignments, if you need to make use of auxiliary data structures, please feel free to make use of Java collection classes. The implementation of the shortest path algorithm must be your own. The use of any other code that is not your own is not allowed.

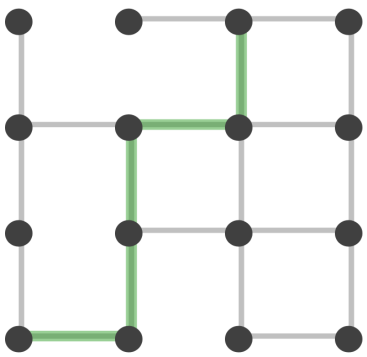
Bonus: Shortest Path using Dijkstra's Algorithm

Optionally, please implement Dijkstra's shortest path algorithm to find the shortest *weighted* path between the source and target vertices supplied in the query file. Visualize these paths using the supplied `MazeVisualizer`. Your program will be invoked from the command line as follows:

```
java HW4 [option] maze-file query-file
```

The command-line argument `[option]` can take one of two values:
`--unweighted`: determine the shortest unweighted path.
`--weighted`: determine the shortest weighted path.

To help you test your program, the shortest path using Dijkstra's algorithm for the pair `1` and `12` (for the example above) is pictured below.



Additional test files and output will be supplied closer to the due date.

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.

Grading

You will be graded based on the functionality of your program as demonstrated by test input files and the corresponding visual output. The following high level grading scheme will be used:

- Program compiles (3 points).
- Functional implementation of unweighted shortest paths as demonstrated by test files (12 points). Partial credit will be awarded based on the level of functionality achieved:
  - File I/O (2 points).
  - Graph representation (3 points).
  - Implementation of shortest path algorithm (5 points for BFS, 10 points for Dijkstra's).
  - Demo (2 points).
    If your program is not working, please let your TA know (during your demo) which parts you have implemented for partial credit.

Submission

Part I

Please use D2L to submit a PDF file called `part1.pdf` (scanned or typeset) that is packaged with the rest of the files for Part II. *If you are scanning your solution, please submit the scanned pages as a single PDF file.*

Part II

Please package *all* Java source files needed to compile your program in a compressed archive called `hw4.zip`. You do not need to include any maze files or query files. Please make sure that your name appears at the beginning of each source file that you authored. In your archive, please also include a `README` file that includes instructions on how to compile and run your program. Please use the `README` file to cite any sources that you used. Please do not place the files in different folders. Please also include the file `part1.pdf` in your archive.