10 Heapsort; StressTests

Portfolio Programs: The Java Collections Framework

Read through the documentation for Java Collections Framework library. Find how you can run the sorting algorithms defined there. Write a simple program that will test these algorithms and measure their timing in a manner similar to the last two labs.

Pair Programming Assignment

10.1 Priority Queue

1. Implement the heap-based priority queue algorithm as described in part 2 of Lab 11.
2. Now implement a simple variant of *heapstort* as follows:
   - In the first step insert the given data into your `PriorityQueue`, one item at a time.
   - In the second step, remove the data from your `PriorityQueue` and insert them into the resulting `ArrayList`, one at a time.
     If you just add each item at the end, you will end up with a list ordered in descending order. If you wish to get the correct ordering, insert each item at the index 0.

10.2 Stress Tests

For this problem finish the work on the first problem in Lab 11, especially focusing on the set of questions and the answers you may find by running the timing tests.

Part 1

Add your implementation of the *heapsort* to the sorting algorithms that will be measured. The skeleton for this is already in place.

Part 2

Run the program a few times with small data sizes, to get familiar with what it can do. Then run experiments and try to answer the following questions:

1. Which algorithms run mostly in quadratic time, i.e. $O(n^2)$?
2. Which algorithms run mostly in $O(n.log_n)$ time?
3. Which algorithms use the functional style, using `Cons` lists?
4. Which algorithm is the *selection sort*?
5. Why is there a difference when the algorithms use a different `Comparator`?
6. Copy the results into a spreadsheet. You may save the result portion in a text editor with a .csv suffix and open it in *Excel* (or some other spreadsheet of your choice). You can now study the data and represent the results as charts. Do so for at least three algorithms, where there is one of each — a quadratic algorithm and a *linear-logarithmic* algorithm.

   Produce a report with a paragraph that explains what you learned, using the *Excel* charts to illustrate this.

   Your report should have a professional look – typed, computer generated charts, reasonably organized. It does not have to be more than 2 pages long, one page is OK.

10.3 Graph Traversals

In the assignment 5 we have designed methods that displayed the map of the USA and showed the user the given route from one city to another. However, we did not know how to find the route.

We now design the method that computes the route from one city to another in one of three different ways. The algorithms that perform the search for the route are called *Breadth-First Search (BFS)*, *Depth-First Search (DFS)*, and *Shortest Path (Dijkstra)*.

The Model

The Graph and User Interactions

Your program needs to represent a graph with places that represent capitals of the 48 US states. Each place has a name — the name of the state. Two states are neighbors if they have a common border. You may consider the *four corner states: Colorado Utah, Arizona and New Mexico* as connected to

each other. The distance between two states is the distance between the capitals of the two states.

The code in the file **GraphAlgoView.java** allows the user to type in the choice for the origin and the destination of the route and choose which algorithm should be used to find the route. *Do not start using it until you are sure most of your program works (through standalone tests).*

Similarly, you can add to the project your old code that shows the route on Canvas and displays the routing instructions, or animates the route — but only once the rest of the program works correctly.

The Data

All three algorithms use the same initial data and methods:

- The state map that is a list of places where each place has a list of its neighbors. (You can reuse the code from Assignment 5, or design a new list using `ArrayList`.

- There is a method that computes the distance between two places. Look up what you have done before and re-use it.

- The desired origin and destination for the route you are trying to find.

- You will need a *To Do* checklist that contains the places you plan to look at next, together with some additional information: for each place, the place you came from and the distance you need to travel to get to this place. You need to define a new class `FromTo` to represent this information. The way this checklist is organized determined which one of the three algorithms will be used.

  We explain later how the distance value should be computed and used.

- You will need a *backtrack list* that also contains data of the type `FromTo`.

- Finally, the route you compute will be either a list of places, or a list of `FromTo` data, as long as you can them translate the information it provides into routing directions.

The Three Algorithms

The only difference between these three algorithms is in the way they keep track of the *ToDo* checklist.

BFS: The Breadth-First Search wants you to look at all nearest neighbors before you look at neighbors two steps away. It uses a *FIFO* (first in - first out) organization for the *ToDO* checklist. Of course, we know that this is just a queue. Implement the queue in any way you wish, including using the Java Collections Framework classes. The distance is ignored here and can be anything.

DFS: The Depth-First Search wants you to explore a path through the first neighbor as far as it goes, before trying out another neighbor. It uses a *LIFO* (last in - first out) organization for the *ToDO* checklist. Of course, we know that this is just a stack. Implement the stack in any way you wish, including using the Java Collections Framework classes. The distance is ignored here and can be anything.

Dijkstra: The Shortest Path Search has been invented by Edgar Dijkstra. Your goal is to keep track of the shortest way known so far to all neighbors that are in the *ToDo* checklist. This checklist is kept as a *priority queue* recording the shortest distance to the *To* places that we have found so far. When you find a new way to a node one of three things may happen:

- this place has not been anyone's neighbor and is not among the nodes we have included in the *ToDo* checklist. In this case, we just add it to the checklist, using the distance to this place to determine the priority - shorter distances have a higher priority. We already know the distance to the place we are coming from, we add to it the distance between that place and its neighbor we are just considering.

- this item has been a neighbor of some other node we have seen already (there are two ways to get to this place). You need to compute the distance to this place using the new route (adding the distance to already computed to the *From* place to the distance between the *From* place and the *To* place.

  If this is a better way (i.e. a shorter one), replace the other way to *To* place with this one.

When asked to remove an item from the *ToDo* checklist in the *Dijkstra algorithm*, we remove the item with the highest priority — which is the **shortest** distance to the origin.

Search Algorithms

Here is the actual description of the three algorithms:

1. Start with an empty *To Do* checklist. Find in the collection of places the origin and add it to the *To Do* checklist, as `new FromTo(empty, origin, 0)` i.e., with an empty place as the place we came from and the distance equal to zero.

   Of course, you replace the `empty` and `origin` with appropriate objects that represent this information.

2. Repeat the steps 3. though 4. until one of the conditions in the next step is satisfied.

3. Remove a `FromTo` item `fromTo` from the the *To Do* checklist. If one of the condition below holds, stop the loop and take the specified action.

   - The *To Do* checklist is empty, in which case no path has been found.
   - The *To* place in the `fromTo` item we remove from the *To Do* checklist is the destination. If this is the case, add the `fromTo` item to the *backtrack list* and finish the work with the *Backtracking algorithm*

   Otherwise, add the `fromTo` item to the *backtrack list* and continue as follows:

4. Add all neighbors *M* of the *To* node in the `fromTo` item to the *To Do* checklist as follows:

   - Do not add *M* to the *To Do* checklist if *M* has been already visited (it appears as *To* place in the *backtrack list*).
   - When adding *M* to the *To Do* checklist do the following:
     - For the DFS and BFS do not add, if the *To Do* checklist if the place *M* already appears as *To* place in the *To Do* checklist.

– For the SP, add if the place *M* does not already appear as *To* place in the *To Do* checklist.
If the place *M* already appears as *To* place in the *To Do* checklist as item `fromToAlt`, you may need to replace the item `fromToAlt` with the item `fromTo`. More on this later.

Backtracking algorithm

Suppose this is our *backtrack list* and we are looking for a route from `A` to `F`.

```
0 -> A, A -> B, A -> D, B -> C, D -> E, C -> F
```

We first remove the item `C -> F` indicating the route to `F` on its last leg went through `C` and add it to our *route*. We then look to find how did we get to `C` and see that the item `B -> C` shows we came through `B`. So, we remove the item `B -> C` from the *backtrack list* and add it to our *route*. Repeating the same reasoning, we find that we got to `B` from `A`, remove the item `A -> B` from the *backtrack list* and add it to our *route*. We repeat one more time, but here we see that there is no place before `A` and so our route is complete. We just read it in the correct order:

```
A -> B -> C - > F
```

1. Remove the item `fromTo` where the *To* place is the destination from the *backtrack list* and add it to the final path.
2. Repeat: For the *From* place *P* for this item find the `fromTo` in the *backtrack list* where the *To* place is *P*.
3. Add it to the final path.
4. If it is the starting node, stop and print the routing, otherwise return to the step 2.