CSE 331: Software Design & Engineering Fall 2024

Homework 3

Due: Monday, October 21st, 11pm

As in Problem Sets 1–2, a key part of this assignment is practicing debugging, this time in an application with both client and server components. Task 4 asks you to submit a log describing all the time spent debugging and the nature and causes of the bugs. While you submit your log in Task 4, the debugging itself can take place while working on Tasks 1–3. Do note, however, that it will be challenging to fully debug earlier parts until you are able to test them which will be easier after completing Task 3.

Before you begin Task 1, be sure to read the instructions of Task 4 to learn what information you need to keep track of while debugging. Then, as you work on each coding task, whenever you see a bug, carefully record that information so that you can submit it in Task 4. As in HW2, don't get stuck debugging a task without making sure you at least have an implementation attempt for all parts.

Check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-24au-materials/hw3-campuspaths.git
```

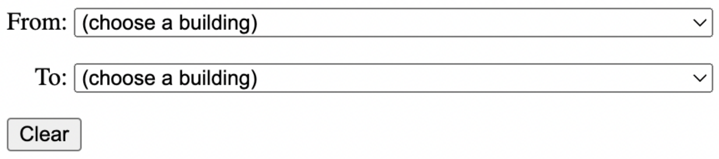
Navigate to the `hw3-campuspaths/server` directory and run `npm install --no-audit`. Then start the server with `npm run start`. In a separate terminal, do the same thing in the `hw3-campuspaths/client` directory. With both the client and server parts running, you can open the application at `http://localhost:8080`.

When you start the application, you will see a map as in HW2, but it will not do anything yet. Eventually, the application will allow the user to select two places on campus and it will display the shortest path between them on the map. Unlike in Problem Set 2, where all calculation is done in the client, here, the shortest path calculation will be performed on the server (in Task 2), so the client will need to send a request to the server to get the answer (which we will do in Task 3).

We encourage you to explore the starter code for this assignment before getting started. You may find useful functions or otherwise benefit from understanding the types we utilize in this assignment. It is a very common mistake for students to do more work than necessary, especially in implementing the algorithm for Task 2.

Task 1 – One Client Leap for Mankind [25 pts]

The provided `App` component of the client displays the campus map and an `Editor` beneath it, but initially, the `Editor` does not do anything. Implement this component to allow the user to choose the two locations between which they want to see a path. You must also include a button to clear the path. Your UI should look something like this[1]:

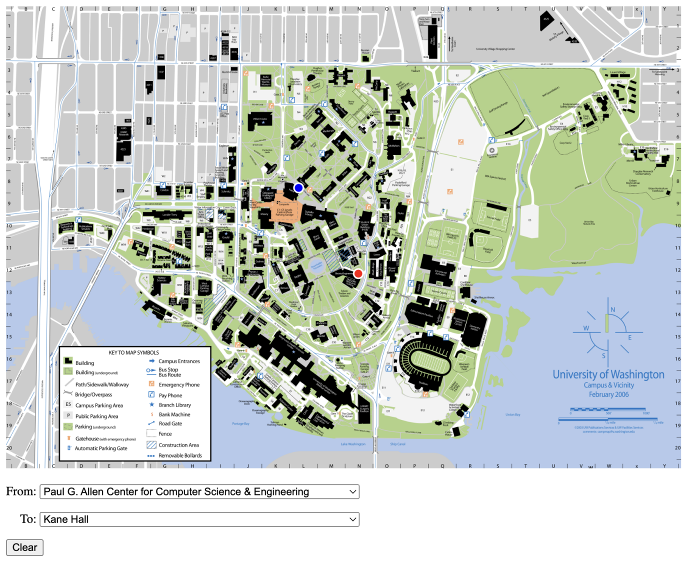


From: (choose a building)

To: (choose a building)

The list of buildings is provided to you in `props`. A callback is also provided to invoke to change the path that is displayed. You should call this when the user has chosen the two endpoints and when they clear the path (in that case, you pass `undefined` to indicate no path). Do not invoke the callback when the user has chosen only one endpoint: the callback wants either two endpoints or no path at all.

At this point, after selecting two buildings, markers should appear for each, but no path will be drawn yet as we need to find that path in the next task.



From: Paul G. Allen Center for Computer Science & Engineering

To: Kane Hall

[1]The precise details of the layout and styling are not important. Once again, this is not a UI design class.

Task 2 – The Full-Short Press [25 pts]

Now, switching to the server, complete the method shortestPath in dijkstra.ts. This method takes a starting (x, y) location and an ending (x, y) location along with a list of all the pairs of points that you can walk between in a straight line. Each of the latter is called an "edge" and also includes the distance of that straight-line walk.

In campus.ts, there is an array called EDGES that is filled in by the function parseEdges, which parses an array of strings (the lines from campus_edges.csv) into the Edge type. We handle calling parseEdges in the starter code for you, but be sure to import and use the EDGES variable when calling shortestPath.

The method should return a Path object describing the shortest path. A path consists of zero or more steps, each of which moves along one edge. The Path object records the starting location, ending location, the sequence of edges to walk along, and total distance covered. With this type we can keep track of intermediate paths between locations and eventually, a shortest path between buildings. For example, the shortest path from CSE2 (found at $(2315.0936, 1780.7913)$) to Moore Hall (found at $(2317.1749, 1859.502)$) in the format $(x, y) \to (x', y')$ is:

$$(2315.0936, 1780.7913) \to (2286.6177, 1825.6619) \to (2322.4782, 1853.4411) \to (2317.1749, 1859.502)$$

You should complete the method by implementing Dijkstra's algorithm. Pseudocode for the algorithm is given on the last page. This describes the basic structure of the code but leaves out many details. In particular, to translate that pseudocode to functional Typescript, you will need to implement the data structures required by the algorithm (descriptions of which are listed in the pseudocode).

For the required map (adjacent) and set (finished), you can use the built-in Map and Set classes provided in Javascript. Note, however, that these classes use "===" to compare keys, which will not do what we want if we try to use Locations as keys. The easiest way to make this work is to convert a Location to a **string** and use that string as the key. You can do the conversion in any way that you like provided that distinct Locations are converted into *different* strings.

Note: The adjacent map (or adjacency list) should be filled in with all the outgoing edges that correspond to each building, prior to Dijkstra's algorithm. This is in contrast to the other data structures you will use for this algorithm that should start empty.

For the priority queue, we have provided a class called Heap in heap.ts that will do the job. It provides all of the required operations: isEmpty, add, and removeMin. This class is generic, so it can be used with any type, but in order to do so, you must provide a "comparator" function to its constructor that allows it to determine which elements are smaller and larger than others.

A comparator function takes two elements, a and b, as arguments and returns a negative value if $a < b$, a positive value if $a > b$, and 0 if $a = b$. For numbers, simply returning $a - b$ would do the trick.

For Dijkstra's algorithm, we need a priority queue of Paths, so you will need to implement a comparator for Paths in order to use Heap.

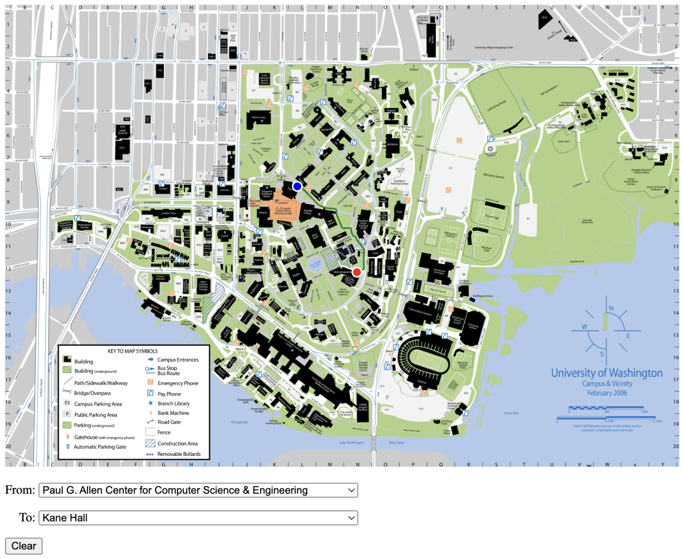
Task 3 – Retrieve You Me [25 pts]

Finally, we will add paths to the application by having the client retrieve a shortest path from the server. We will do so in two steps as follows:

1. On the server, update `index.ts` to have a new route with URL `/api/shortestPath` that calls a `getShortestPath` function you will add in `routes.ts`. The latter should retrieve the starting and ending buildings from the request, invoke `shortestPath` (from `dijkstra.ts`) to calculate the shortest path between them, and then send back the path to the client in the response.

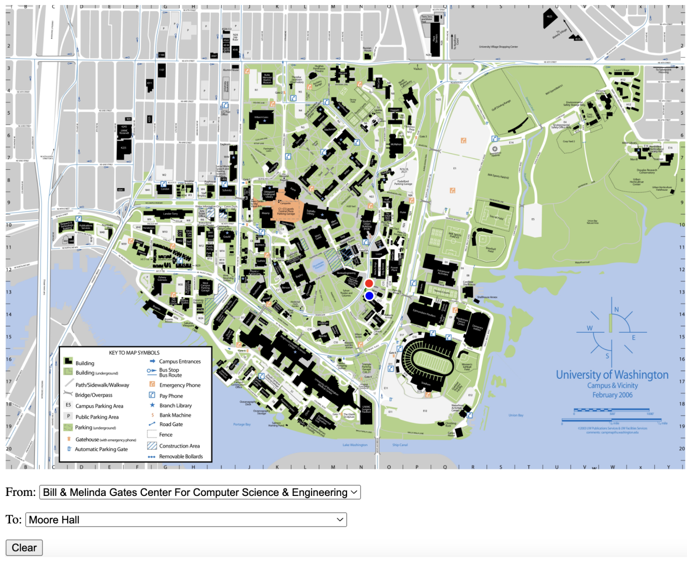
2. On the client, update the `doEndPointChange` method of `App` to initiate a request to the server asking for the path between the two selected buildings, and then, when we get back the path in the response, update the state to store the path in the "`path`" field of `AppState`.

Once you have done these steps, the application should display shortest paths almost immediately after the user selects two buildings in the UI you built in Task 1. (See the examples below in Green)



From: Paul G. Allen Center for Computer Science & Engineering

To: Kane Hall



From: Bill & Melinda Gates Center For Computer Science & Engineering

To: Moore Hall

Task 4 – Go Log Wild! [25 pts]

Submit your log of all time spent debugging, along with an explanation of the cause of the bug. Specifically, for each bug, provide the following information:

- What (incorrect) behavior you saw that told you there was a bug.
- How many **minutes** it took you to find the bug.
- What kind of experiments you performed to try to locate the bug.
- What was the error in the code that led to the incorrect behavior.
- What the defect was that caused the bug (if you ever found it).
- Was the code that produced the failure in a *different function* than the code that contained the defect? What functions (on either the client or server or both) did you need to debug through in order to find this bug?

Again, we have provided a debugging log website for you to record your debugging. Don't forget to save your log!
Since debugging is the most important part of this assignment, you can still get full credit for submitting an incorrect solution provided that you fully document at least **8 hours** (480 minutes) of debugging. You are forbidden from spending significantly more than 8 hours debugging.

Submission

After completing Task 4, download your logging as a PDF following the instructions from HW1. Make sure the downloaded file is called "DebuggingLog.pdf". Submit the following files to the "HW3" assignment on Gradescope:

`DebuggingLog.pdf`	`App.tsx`	`routes.ts`
`Editor.tsx`	`dijkstra.ts`	`index.ts`

After you submit your work, an autograder will run to verify you have submitted the correct files; wait for it to finish and check that the submission looks correct. Don't forget to check that the submitted files are up-to-date with all implementation and debugging you completed!

Dijkstra's Algorithm

The pseudocode below assumes we have the following data structures:

`adjacent` A *map* from an (x, y) location to the list of all edges that start at that location. These give us all the locations you can get to from that location in one step.

`finished` A *set* of (x, y) locations for which we have already found the shortest path. The algorithm will avoid considering new paths to these locations.

`active` A *(priority) queue* containing all paths to locations that are one step from a finished node. The key idea of the algorithm is that the shortest path in the queue to a non-finished node must be the shortest path to that node.[2]

With those data structures in hand, Dijkstra's algorithm proceeds as follows:

```
add a 0-step (empty) path from start to itself to active

while active is not empty:
  minPath = active.removeMin()  // shortest active path

  if minPath.end is end:
    return minPath  // shortest path from start to end!

  if minPath.end is in finished:
    continue  // longer path to minPath.end than the one we found before

  add minPath.end to finished  // just found shortest path to here!

  // add all paths that have one step added to this shortest path
  for each edge e in adjacent.get(minPath.end):
    if e.end is not in finished:
      newPath = minPath + e
      add newPath to active

return undefined  // no path from start to end :(
```

[2]This can be proven formally using tools from CSE 311.