CSE 331: Software Design & Engineering Winter 2025

Homework 3

Due: Wednesday, January 29th, 11pm

As in Homeworks 1–2, the focus of this assignment is practicing debugging, this time in an application with both client and server components. Task 2 asks you to submit a log describing all the time spent debugging and the nature and causes of the bugs. Like HW2, you can debug after completing each section of the app, but it will be challenging to fully debug earlier parts until the whole app is complete.

Check out the starter code for this assignment:

```
git clone https://gitlab.cs.washington.edu/cse331-25wi/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 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 Homework 2, where all calculation is done in the client, here, the shortest path calculation will be performed on the server, so the client will need to send a request to the server to get the answer.

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 Dijkstra's Algorithm.

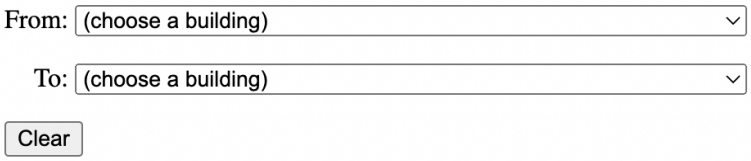
Task 1 – Don't be caught App-ing [25 pts]

Unlike Homework 1 and 2, **you will turn in your code for this assignment**. This will allow us to give you feedback on your code and verify that you are following our class coding conventions and making an attempt at all parts of the app. Our primary focus is still your debugging experience, so the majority of the points for this assignment will still go to your debugging log, and it is okay if you turn in code that has remaining defects, provided you've made an attempt at all parts.

Don't get stuck debugging a section without making sure you at least have an implementation attempt for all parts. In general, it's a good debugging strategy to move on to a problem and look at it later with a fresh set of eyes.

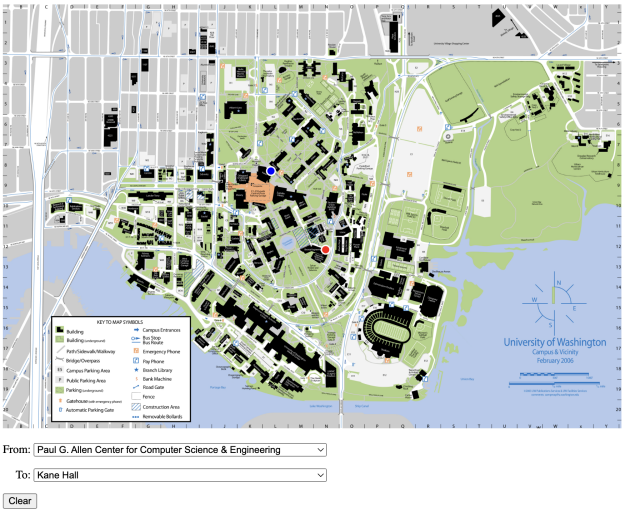
One Client Leap for Mankind

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]:



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 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 section.



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

The Full-Short Press

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 pairs of points representing straight line walking paths (for this app, this is all walking paths on the UW campus). Each path 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 holds 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 $(x, y) \rightarrow (x', y')$ is:

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

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: Prior to Dijkstra's algorithm, the `adjacent` map (or adjacency list) should be filled in with all the outgoing edges that correspond to each starting Location in `edges` (which represents all the edges on the map). Other data structures you will use for this algorithm should start empty.

For the priority queue, we have provided am interface called `Heap` in `heap.ts` that will do the job. It provides all the required operations: `isEmpty`, `add`, and `removeMin`. This interface 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`.

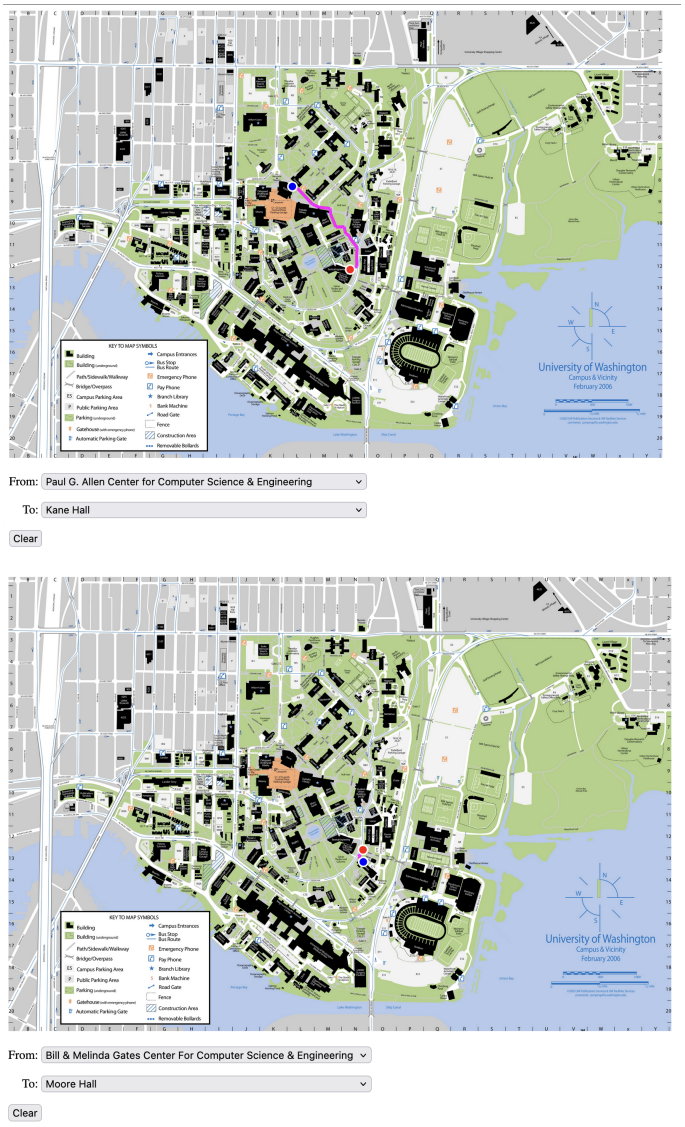
Retrieve You Me

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 the first part. (See the examples below in fuchsia)



Linter

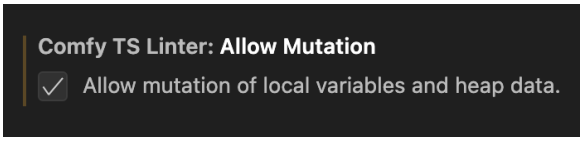
Starting with this assignment, we will also be using a custom linter, comfy-tslint, which checks that your code follows our course-specific coding conventions. The linter is not able to catch every coding convention mistake, so you should write code that looks similar to the examples we go over in class, and refer to the coding conventions document for a written explanation of our expectations.

We will run the linter on your code through the autograder when you submit on Gradescope, so you should make sure it also passes locally. You can always run the linter in the command line with `npm run lint`, or you can download the recommended VS Code extension for the linter. The linter warnings will appear as helpful popups while you're coding (similar to the type checker). If you run the command, the output will always have the first two lines listed in the image below, and if your code has any errors, those will be listed below.

```
> cse331-hw-fib@0.0.1 lint
> comfy-tslint --no-mutation src/*.{ts,tsx}

src/fib.ts:7:32: any type is not allowed
src/index.tsx:6:10: top-level variable declarations must have a type
```

The linter restrict mutations use, but for this assignment mutation is allowed on the server. The `npm run lint` command is already configured to allow this mutation. To allow mutation with the VSCode extension, open the extnsion, select the gear icon, open "Settings", and check the checkbox to "Allow Mutation".



The linter also has some restrictions on loops. It disallows standard `for` loops and requires that all `while` loops have an "invariant". If you want to use a standard `for` loop, use a `for...of` loop instead. We haven't covered invariants yet, so if you want to use a `while` loop, you can stop the warnings by placing a comment above the loop that starts with `// Inv:`, as so:

```
// Inv:
while(true) {
```

If you have any trouble with loop or mutation related linter warnings, definitely reach out on Ed or OH so we can help you resolve it!

Code Submission

After you finish debugging your code (details of which are described in the next task), you should turn it in by submitting the following files to the **"HW3 code"** assignment on Gradescope:

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

After you submit your work, an autograder will run to verify you have submitted the correct files (as well as run the linter). Verify that the submitted files are up-to-date with all implementation and debugging you completed. Then, wait for the autograder to finish, to check that the submission looks correct and fix any errors, if needed.

Task 2 – Go Log Wild! [75 pts]

Submit your log of all time spent debugging, along with an explanation of the cause of the bug.

For each bug, you must also provide the following information:

- What **failure**, (incorrect) app behavior, did you see that told you there was a bug?
- Which **experiments** did you perform to try to locate the defect? (Checking the network tab, scanning for typos, `console.log`s, etc.)
- What the **defect** was that caused that bug (if you ever found it)?
- How many **minutes** did you spend on the bug after noticing the failure?
- 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 the 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!

Once you have finished debugging your app, you will select *only* 3 log entries to turn in. Like Homework 1–2, you should try to select "interesting" entries, though you will not be penalized if some of your bugs were simple. However, **each log entry you select to turn in should have, at minimum, 1 experiment, and at least one log entry should have, at minimum, 2 experiments.** Experiments (your process) are the most important part of this assignment.

Your log should capture all the necessary context around each bug. Your TAs should be able to understand the interactions and inputs that led to the failure, and follow your experience through each debugging step. Experiments should start with some hypotheses with a leading question that you hope to answer with your experiment; then, upon seeing the result, we want to know what you learned which may lead to a next experiment (or to finding the defect).

It's understandable that you may hit some dead-ends and need to try a totally unrelated experiment idea, or that you may not find the defect, so it's okay if these show up in your log entries. Remember, that the goal here is that you improve your debugging skills, so try to make each experiment choice intentionally and avoid just "trying stuff" (but still log it if you do!).

You only need to turn in 3 log entries, but we encourage you to continue debugging your app to try to get all the behavior working because it's fun to have a working app! and it's good practice.

If you have been debugging for more than ~8 hours and have yet to find 3 bugs to turn in, we *highly* encourage you to reach out to the staff for help! Come to office hours or make a private ed post and let us know what's going on, so we can try to give you some extra debugging support. Sometimes bugs takes days or weeks to debug, in the "real world", but *extremely* time-consuming bugs are not the intention for this class, so make sure you're getting our help if you need it!

If you think your implementation is correct, and you have not found 3 bugs to log, or you have not found 3 bugs that required experiments (e.g. immediately obvious typos), you can send us your implementation, and we will return it to you as soon as possible with new bugs for you to debug. To send us your work, you should upload `Editor.tsx` containing your completed component to Google Drive, configure the share settings so we can access it by link, and email that link to cse331-staff@cs.washington.edu. If you need to send us your implementation for bugs, you **MUST** do so by **Monday, Jan 27th at 7pm**.

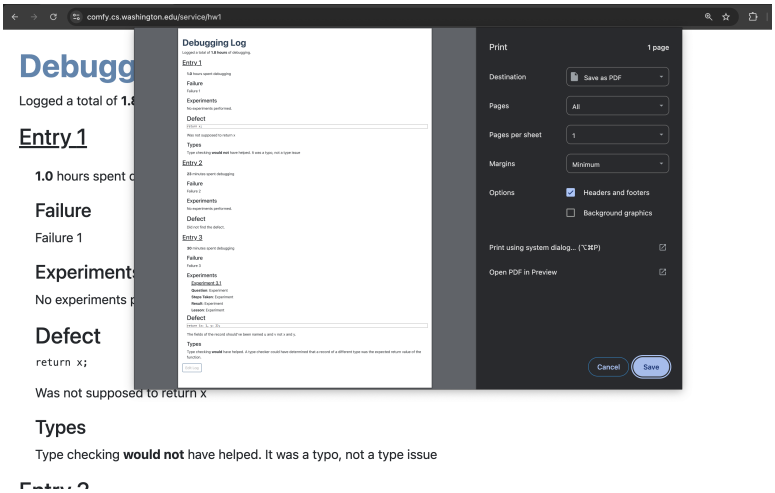
Debugging Log Submission

After you finish debugging:

1. Open each log entry that you want to include in your submission and select Show in "View" box.

   Show: ☑ (in "View")

2. From the main page, select "View Log" to open all of your selected log entries.
3. Select File > Print or ctrl+P/Command+P to open the print dialog.
4. Set the print destination as "Save as PDF" and "Save"



5. Submit your downloaded log to the **"HW3 Log"** assignment on Gradescope.

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.