CSE 331: Software Design & Engineering Fall 2024

Homework 9

Due: Saturday, December 7th 11pm

If you have not completed HW9 written. Close this file and finish the written component before starting the coding parts in this worksheet.

Submission

After completing all tasks to follow, submit your solutions on Gradescope. The following completed files should be submitted to **"HW9 Code"**:

`locations.ts` `locations_test.ts` `location_tree.ts` `location_tree_test.ts`
`routes.ts` `routes_test.ts` `MapViewer.tsx`

Set up

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-24au-materials/hw9-campusfriends.git
```

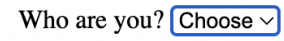
Install the node modules: Create two terminals. Navigate to the `client` directory in one and the `server` directory in the other, and run `npm install --no-audit` in both. To run the app, you'll need to run the command `npm run start` in both the `client` and `server` directories. Then navigate to where the client is running `http://localhost:8080/`.

Like HW8, mutation is permitted in the `/server` but not the `/client`.

Most coding work in industry is making changes to existing code and, most of the time, those changes are being made to add new features or improvements to the software.

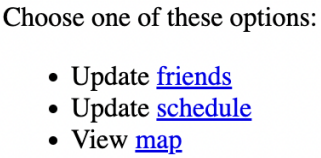
In this assignment, we will add a feature to the campus map application we wrote in HW3. You will implement the client and server portions of an application that allows users to identify friends whose walking paths come close to *some point on* this user's path and then to display those on the map and in the legend. The user can then have an opportunity to see their friends during their walk without having to go too far off their usual path.

Rather than having a single page showing the campus map, the starter code now supports multiple users by beginning with a "login" page[1]:



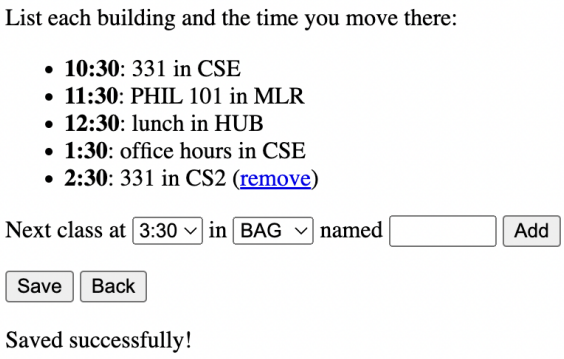
Feel free to update the code in `users.ts` to include yourself and your friends.

After logging in, the user sees a menu with three options:



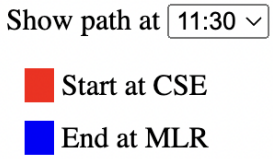
The first option takes them to a page where they can select which of the other users are their friends. As we will see below, our new feature will allow some information about their friends schedules to show up when they view the map.

The second option takes them to a page where they can record their schedule:



The schedule is recorded as a list of locations where they will be throughout the day. We will assume that the user is at each location until approximately 10 minutes before the next item on the list. During those 10 minutes, they will be walking between the two locations.

The map page now lets them choose a time on their schedule when they walk between locations:



Once they do so, the map will show the walk that they will need to make and, below that, a legend showing information about the endpoints of the path.

[1] Obviously, a real version of this application would authenticate with a password or similarly secure mechanism. To make this easier to test, we will skip that.

Building on an existing app requires **first understanding existing code**. Know that it is expected that you may have to spend time reading starter code (and tests), using the app, drawing diagrams and/or anything else that's useful to understanding the context around the tasks you'll need to complete. The spec also requires a lot of reading, but its worth it to take time to understand.

Task 3 – Let's Do Dist Thing [18 pts]

The following types and functions we began working with in HW9 written (some of which we also saw in HW3) have been translated to typescript in `server/src/locations.ts`.

```
type Location = {x: number, y: number};

type Region = {x1: number, x2: number, y1: number, y2: number}

const distance = (loc1: Location, loc2: Location): number
```

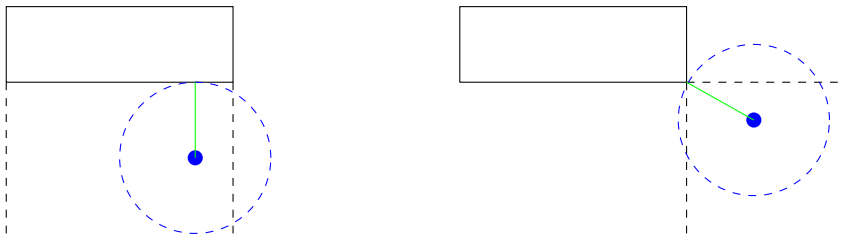
The starter code also includes a function called `squaredDistance` that simply skips the final square root, making it substantially cheaper.

We will start by implementing the following helper function in `locations.ts`:

```
const distanceMoreThan = (loc: Location, region: Region, dist: number): boolean
```

This should check whether the distance from the given location to the *closest* point in the given region is more than `dist`. (An alternative would be to have the function return the distance to the region and let the user compare it to `dist` themselves, but calculating distances often requires a square root. By taking `dist` as an argument, we can answer the "more than" question using `squaredDistance` instead and avoid the need for any square roots.)

It remains to figure out what point in the region is closest to the given point. Since the closest point must be along a boundary, it is either one of the corners or a point along one of the four sides. As the following figures demonstrate, when the location is in line with the region — either vertically or horizontally – then the closest point is the first point along that line, which is on a side. Otherwise, the closest point is the nearest corner.



1. Implement the function `distanceMoreThan` in `locations.ts`. Make sure that your function does not call `distance` or otherwise use any square root calculations.

   Be sure to carefully think through your code to make sure it is right. Finding and fixing a bug in this code will only get harder later on, so do the work now to make sure it is right!

2. Write tests for your function in `locations_test.ts`. Make sure that you cover all required cases.

   The debugging will only get harder if you miss a bug now.

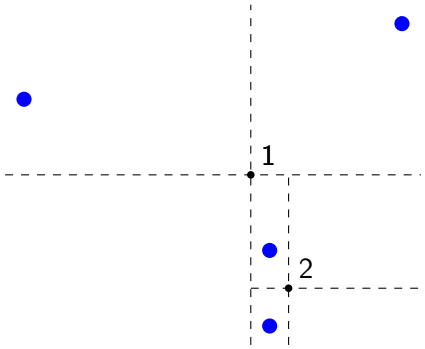
Task 4 – The Best Things in Life are Tree [20 pts]

Recall the definition of `LocTree`, as introduced in the written part, which allows us to efficiently find the closest point to a given location with the recursive structure.

type LocTree := empty
| single(loc : Location)
| split(at : Location, nw : LocTree, ne : LocTree, sw : LocTree, se : LocTree)

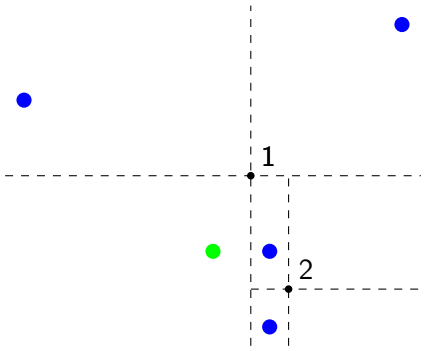
This (and all types and functions we use for this task) is implemented in TypeScript as `LocationTree` in `location_tree.ts`.

As a reminder, to create a tree for the four points below, we would start by splitting them at point "1" (the centroid of those locations). The NW and NE regions then contain only a single point and the SW region is empty. The SE region, however, still contains two points, so we would split them again at point "2" (the centroid of those two), which leaves four regions containing at most one point.



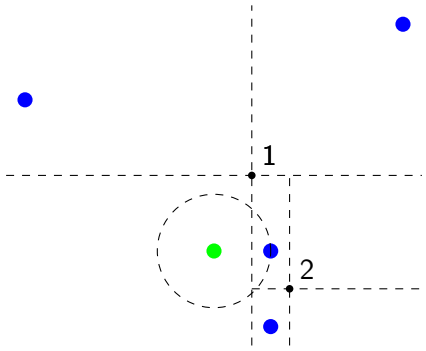
Each node represents some rectangular region in space, but the bounds of that region are not recorded in any field. However, we can easily calculate them bounds as we work down the tree recursively.

We want to be able to find the `Location` in the tree closest to a given `Location`. For example, to find the closest (blue) point of the location tree to the given (green) point.



The root of the tree is the split marked "1". Since the green node falls into the SW region, we should start by recursively finding the closest point there. That region is empty, so we have no bound on how small the closest point might be.

The region of "1" next closest to the green point is the SE region, so we recursively search there next. That is itself a split node, with the split marked "2" in the picture. The green node is closest to the NW region of "2", so we search there. That contains a blue node that is at a distance shown by the radius of the circle in this figure:



This distance becomes an upper bound on the closest distance of any point in the tree. We may later find points that are closer, but if they are, they would need to be closer than this region.

The next closest region in the split "2" is the SW region. We can see that its north-west corner falls just within the circle, so it is possible for it to contain a closer node, which means we need to search it recursively. When we do so, we will calculate the distance to the lowest blue point and find that it is farther away. Thus, the first point we found remains the closest, but we have now performed two distance calculations.

The next closest region in the split marked "2" is its NE region, but its distance to the green point is larger than the closest point found so far (the circle does not intersect that region), so we can skip search it all together. The final region of the split marked "2" to consider is the SE region. It, too, does not intersect the circle, so it can be skipped.

That completes the search of the SE region of the split marked "1". We have a candidate closest node and have performed two distance calculations.

Since we were searching the SE region recursively, when we return, we go back to our search of the split marked "1". The next closest region (after SW and SE) is the NW region. We find that the circle does not intersect it, so we can skip it. Likewise for the NE region. Thus, we now know for sure that the first point we found was the closest, and we did so using only two distance calculations.

In HW9 written, we introduced the function cl which finds the closest Location in a LocTree to a given Location and takes an additional Region argument containing all the Locations in the LocTree which it uses to avoid recursing into quadrants of split nodes that cannot contain a Location closer than the closest location found so far. The recursive search we described above is very similar to this function.

In our implementation here, however, we additionally want to keep track of what steps our code took to find the closest location. Since distance calculations are expensive, rather than returning the closest point, we will return both the point and its distance. (We already calculated it when we determined that it was the closest point, so there is no need to ever calculate it again.) In addition, we will also keep track of the number of distance calculations performed so far in order to make our code more testable. Without that information, we would not be able to tell via tests that our code was actually checking every region in the tree!

To do this, we will change the return value to the following record type that includes all of the information mentioned above:

```
type ClosestInfo = {loc: Location | undefined, dist: number, calcs: number}
```

In addition, rather than keeping track of just the closest `Location` we have found so far, we will instead pass a `ClosestInfo` recording the closest point found so far *and* the number of calculations performed so far between recursive calls.

With that in mind, the actual signature of the function we want to implement is the following:

```
const closestInTree = (tree: LocationTree, loc: Location,
                       bounds: Region, closest: ClosestInfo): ClosestInfo
```

1. Implement the function `closestInTree` in `location_tree.ts`.

   Your function should *begin* by checking whether the distance from `loc` to the `bounds` of that region is more than the distance to the closest point found so far. If it is, then you can skip searching this subtree. (The closest node remains the one passed in.)

   Otherwise, if the tree is empty, then there is nothing to search and the closest node remains the same. If it is a single node, then we will need to perform a distance calculation. If it is closer than the node passed in, then we return it as the closest node; otherwise, we return the node passed in. In either case, we have performed 1 more distance calculation.

   Lastly, if we have a split node, then we need to search through the subregions in order by the distance to `loc`. Make a recursive call for each of them, updating the closest node each time to whatever that call returns, and returning the closest node from the last recursive call as the closest node in the subtree.

   Note that this step differs from the cl definition from HW9 written, which searched the subtrees in counter-clockwise order rather than by distance. The approach we take here will help us find the closest location more efficiently. In HW9 written we completed a proof that cl accurately found the closest location, convince yourself now that this alternate order of recursive calls will also be accurate (your answer from Task 2b of the written part may be helpful).

   Hint: While it is possible to implement this last case with a loop[2], it is worth noting that there are, in fact, only *eight* different orders that you might go through the subregions, so you can also implement this with fairly simple `if` / `else` statements.

2. Write tests for your function in `location_tree_test.ts`. Make sure you cover all required cases.

   Again, the debugging will only get harder if you miss a bug now.

3. Uncomment the tests for `findClosestInTree` in `location_test_test.ts`. If the function you wrote is correct, these should all pass.

[2]This would require extra square root calculations. E.g., if the location is in the NW region, then calculating its distance to the SE region involves a square root. The distances to the NE and SW regions, however, do not require square roots!

Task 5 – I'm All Nears [20 pts]

We now have all the helper functions we need to implement the server part of our new feature.

We want to change the `shortestPath` route so that, in addition to returning the shortest path for the user, it also returns the closest point in the path of any friend that is also walking at that time. We will provide that information in an array of records, each having the following shape:

```
type Nearby = {friend: string, dist: number, loc: Location};
```

1. Modify the function `getShortestPath` in `routes.ts` so that the JSON response sent back includes a list of `Nearby` records for each friend who is walking at the same time.

   The starter code already has a loop that identifies the friends who are walking at the same time. You only need to add the code to find the nearest point on the user's path (`locs`) to any of the points on the friend's path (`fLocs`).

   (Note: If you saw a previous version of the spec, the instructions instead read "You only need to add the code to find the nearest point on their path (`fLocs`) to any of the points on the user's path (`locs`)." If you implemented this alternative version, that is also okay; you will not be penalized. The instructions now are what we intended them to be, matching examples from our solution we have given.)

   If these two lists contain n and m points respectively, then a naive algorithm would perform $n \times m$ distance calculations. Instead, if we put the second list of points into a tree (using `buildTree`) and then call `findClosestInTree` for each point, we can find the closest pair of points with roughly $O(n \log m)$ distance calculations, which is substantially faster for large n and m.

2. Modify the tests this function in `routes_test.ts` so that it also tests the cases where friends are nearby. This will require adding a schedule for another friend and making sure they will be walking between locations at the same hour.

   As annoying as it may seem to write these tests, debugging a problem in the UI that is actually a bug in the server is much more work, so be sure that your code is correct before continuing.

Task 6 – Jolly Green Client [20 pts]

Now that the server is working correctly, we will switch over to the **client**.

We want to change the UI to show all the nearby friends of the user when they are looking at any of the walks they make on campus.

1. Modify `doHourChange` in `MapViewer.tsx` to record the list of nearby points in the state when it is returned by the server.

   We have provided a function called `parseNearbyList` in `nearby.ts` that turns the JSON data back into the TypeScript `Nearby` type.[3]

2. Modify `renderEndPoints` to draw a circle showing the location of each of the nearby friends and `renderLegendItems` to identify that circle on the legend.

   At the bottom of the file, you'll see a commented out list of colors `FRIEND_COLORS` that you are free to use (or you can pick your own colors). If there are more friends than colors, you can either reuse colors or only show that many friends.

Congratulations!! on completing your biggest app of the quarter!

[3] You're welcome!