CSE 331: Software Design & Implementation Winter 2025

Homework 9

Due: Friday, March 14th, 11pm

Written

Submission

After completing all parts below, submit your solutions as a PDF on Gradescope under "**HW9 Written**".

Don't forget to check that the submitted file is up-to-date with all written work you completed!

Make sure your work is legible and scanned clearly if you handwrite it, or compiled correctly if you choose to use LaTeX. Match each HW problem to the page with your work when you turn in. If your work is not readable or pages are not assigned correctly, you will receive a point deduction.

In this assignment, we will be working with LocTree as we worked with in section. For more information on all of the data structures and definitions used, please refer to the resource found here: https://courses.cs.washington.edu/courses/cse331/25wi/homework/hw9def.pdf.

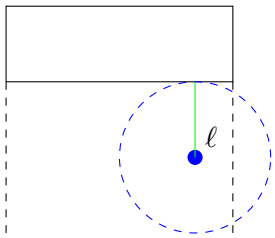
Task 1 – He's Making a Dist, and Checking It Twice [5 pts]

We defined a function "dist" before that calculates the distance between two locations. It will also be useful for us to have a function distTo : (Location, Region) $\to \mathbb{R}$ that calculates the distance from the given location to the *closest* location in the given region. This gives us a *lower bound* on the distance to any location within that region.

To define this function, distTo(ℓ, R), we will break our analysis into a few different cases. The first case is when ℓ is within R. In that case, the distance from ℓ to R is 0.

(a) Write an expression that is true iff location ℓ falls within the region R.

(b) If ℓ is not within R, then our next case is when it is directly below, above, to the left, to the right. For example, if ℓ is directly below, then the picture looks like this:



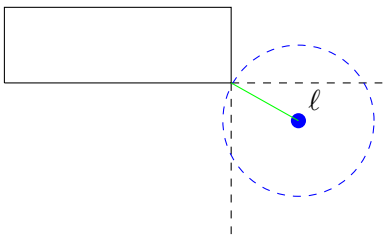
As we can see, the closest location to ℓ in the region R is directly above ℓ.

Write an expression that is true iff location ℓ is directly below R.

(Note that larger y values are farther **down** on the page.)

(c) Write an expression that calculates the distance from ℓ to the closest location in R when ℓ is directly below R.

(d) If ℓ is not within R and not directly below, above, left, or right, then it is in one of the regions diagonally away from R (above and left, above and right, below and left, or below and right). For example, if ℓ is below and right, then the picture looks like this:



In this case, the closest point is the bottom-right corner of R.

Write an expression that is true iff location ℓ is below and right of R.

(e) Write an expression that calculates the distance from ℓ to the closest location in R when ℓ is below and right of R. Feel free to use the function dist defined before.

Finding the Closest Location

The function closest : (LocTree, Location, Location) → Location finds the closest location in the given tree to a given location. More specifically, when invoked as closest(T, ℓ, c), it returns either the closest location in the tree T to ℓ or the location c, whichever is closer to ℓ. It is defined as follows:

$$\begin{aligned}
\text{closest}(\text{empty}, \ell, c) &:= c \\
\text{closest}(\text{single}(\text{s}), \ell, c) &:= c && \text{if } \text{dist}(c, \ell) \leqslant \text{dist}(s, \ell) \\
\text{closest}(\text{single}(\text{s}), \ell, c) &:= s && \text{if } \text{dist}(c, \ell) > \text{dist}(s, \ell) \\
\text{closest}(\text{split}(m, \text{nw}, \text{ne}, \text{sw}, \text{se}), \ell, c) &:= \text{closest}(\text{nw}, \ell, \\
&\qquad \text{closest}(\text{sw}, \ell, \\
&\qquad\quad \text{closest}(\text{se}, \ell, \\
&\qquad\qquad \text{closest}(\text{ne}, \ell, c))))
\end{aligned}$$

The split(. . .) case performs a tree traversal.

It recurses into the **NE** quadrant, and the call results in the closest location to ℓ within the NE quadrant, or c if the NE quadrant has no locations that are closer to ℓ than c is.

The result from traversing the NE quadrant is passed to a recursive call to traverse the **SE** quadrant as the updated closest location found, call it c_2. This SE traversal results in the closest location to ℓ within the SE quadrant, or c_2 if the SE quadrant has no locations that are closer to ℓ than c_2.

This process continues through the **SW** and then the **NW** quadrant.

The location returned from the final recursive call is the closest location of the NW branch to ℓ unless a closer location to ℓ was already found in the SW branch, or the SE branch, or the NE branch, or the originally given 3rd parameter, c. This fact that c is returned from closest if *every* location in the tree is farther away from ℓ will be a useful fact later on! (And can be proven formally by induction.)

The particular order chosen here — NE, SE, SW, NW — traverses the quadrants in *clockwise* order. We could traverse the quadrants counter-clockwise or in any order we want by switching the order of nw, . . . , ne in the recursive calls.

Task 2 – Going Weak in the Trees [18 pts]

Notice that each subtree traversal in closest will traverse all the way down to single location subtrees and directly calculate the distance between that location and ℓ. Distance calculations are expensive, so we want to avoid traversing into subtrees if we know we won't be able to find a closer location than we've already found (c).

We will define a function cl : (LocTree, Location, Region, Location) → Location that looks like closest but takes an extra argument that is a region containing all the points in the tree. With this, it will exclude quadrants of split nodes from the traversal when its region is farther away from ℓ than the known c is (and therefore cannot contain any closer location).

$$
\begin{aligned}
\mathsf{cl}(\mathsf{empty}, \ell, R, c) &:= c \\
\mathsf{cl}(\mathsf{single}(s), \ell, R, c) &:= c && \text{if } \mathsf{dist}(c, \ell) \leqslant \mathsf{dist}(s, \ell) \\
\mathsf{cl}(\mathsf{single}(s), \ell, R, c) &:= s && \text{if } \mathsf{dist}(c, \ell) > \mathsf{dist}(s, \ell) \\
\mathsf{cl}(\mathsf{split}(m, \mathsf{nw}, \mathsf{ne}, \mathsf{sw}, \mathsf{se}), \ell, R, c) &:= c && \text{if } \mathsf{dist}(\ell, c) \leqslant \mathsf{distTo}(\ell, R) \\
\mathsf{cl}(\mathsf{split}(m, \mathsf{nw}, \mathsf{ne}, \mathsf{sw}, \mathsf{se}), \ell, R, c) &:= \mathsf{cl}(\mathsf{nw}, \ell, \mathsf{NW}(m, R), && \text{if } \mathsf{dist}(\ell, c) > \mathsf{distTo}(\ell, R) \\
&\qquad \mathsf{cl}(\mathsf{sw}, \ell, \mathsf{SW}(m, R), \\
&\qquad\quad \mathsf{cl}(\mathsf{se}, \ell, \mathsf{SE}(m, R), \\
&\qquad\qquad \mathsf{cl}(\mathsf{ne}, \ell, \mathsf{NE}(m, R), c))))
\end{aligned}
$$

The split case from closest has become two cases. In the first case, when $\mathsf{dist}(\ell, c) \leqslant \mathsf{distTo}(\ell, R)$, we know that all the locations within this subtree are farther away than c is from ℓ, so we skip the recursive calls and simply return c. It is only in the second case, when $\mathsf{dist}(\ell, c) > \mathsf{distTo}(\ell, R)$, that we perform the recursive calls.

We use the functions NW, NE, SW, SE to calculate a smaller region that only includes the part of R that falls within that quadrant and pass that region in the recursive call. (See the Week 9 Definitions resource for an example of using these functions.)

(a) Prove that, if the region R contains all locations in the tree T, then $\mathsf{cl}(T, \ell, R, c) = \mathsf{closest}(T, \ell, c)$. Your proof should be by structural induction on T.

Feel free to use the fact that, if R contains all the locations in $\mathsf{split}(m, \mathsf{nw}, \mathsf{ne}, \mathsf{sw}, \mathsf{se})$, then $\mathsf{NW}(m, \mathsf{nw})$ contains all the locations in nw and likewise for ne, sw, and se.

Remember that, because $\mathsf{distTo}(\ell, R)$ is the distance from ℓ to the closest location in R, any other location s in R must satisfy $\mathsf{dist}(\ell, s) \geqslant \mathsf{distTo}(\ell, R)$. Hence, if $\mathsf{dist}(\ell, c) \leqslant \mathsf{distTo}(\ell, R)$ ($\leqslant \mathsf{dist}(\ell, s)$), then we know already that s cannot be closer to ℓ than c. Thus, if R contains all the locations in the tree, then none of them can be closer to ℓ than c, and we can simply return c immediately as the closest, which is exactly what cl does in its definition above.

Hints:

- We covered a suspiciously similar problem in section that you should use as a direct reference for how to organize your proof.
- Your base case and inductive step will need to contain a proof by cases.
- Our claim states "if the region R contains all locations in the tree...," but you still need to state the claim holds in the vacuous cases when the region R does not contain all locations.

Coding

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` `index.ts` `MapViewer.tsx`
`App.tsx` `FriendsEditor.tsx`

Set up

Start by checking out the starter code using the command

```
git clone https://gitlab.cs.washington.edu/cse331-25wi/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/`.

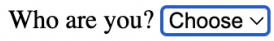
Like HW8, mutation is permitted in the `/server` but not the `/client`.

So your internship project **"CampusMaps"** from week 3 was a hit with the development team of Comfy Inc, so you were offered a return offer with the condition that you could add new features to improve the app. The feature you came up was to be able to find friends that will be nearby when you go to class. Kevin the CEO of Comfy Inc hears this and is over the moon with the idea, so now all that's left for you to do is to implement it.

Before you start Kevin has some wise words he would like to impart onto you. 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.

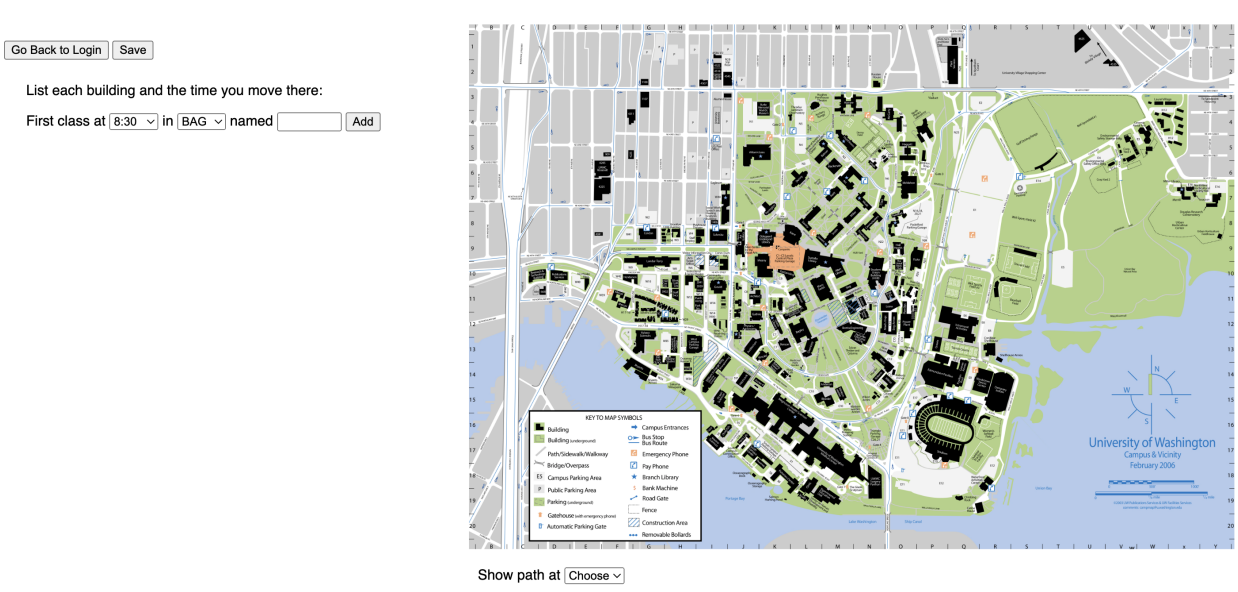
As a note from the development team we've unfortunately been a little lost without our star intern, and this is a work in progress, so if you encounter any bugs please report them to upper management, but don't feel the need to fix them.

Instead of only having the map, the starter code now supports multiple users by beginning with a "login" page[1] using a dropdown to select a user:



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

After logging in, the user will be able to see the map from the original app:



You can specify and add in your classes which will appear in a list. During the 10 minutes before each class, users will be walking from their previous class, and the "Show path at" dropdown allows you to display the path to get to class on the map.

[1] A real version of this application would authenticate with a password or similarly secure mechanism.

Task 3 – Let's Do Dist Thing [10 pts]

Before we can add our super cool feature, we need to implement some helper functions.

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
```

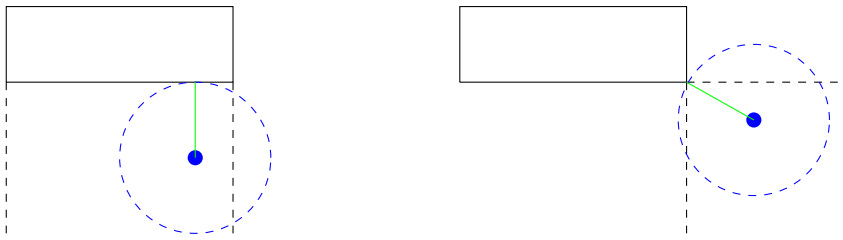
The starter code also includes a function called `squaredDistance` that simply skips the final square root of the distance calculation, making it substantially cheaper, but equally useful when the only requirement is comparing distances (e.g. is point b or c farther away from point a). **You should use `squaredDistance` instead of `distance` whenever possible in the following tasks.**

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 for the user would be to use the distance function to calculate the distance to the region and compare it to `dist` directly, but that would require a square root. By taking `dist` as an argument, we can answer the "more than" question using `squaredDistance` alone.)

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.



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

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!

(b) 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]

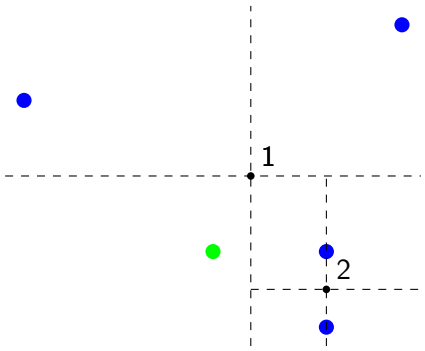
In this part, we will implement a TypeScript function that finds the `Location` in a `LocTree` closest to a given `Location`. This may sound very familiar and that's because you worked with a function that did just this in Task 2 with the function cl!

Our function will have the same results as cl but with a slightly modified recursive search. Instead of searching in counter clock-wise order, we'll search the regions by distance to the given location. This gives us a nice heuristic for finding closest points sooner as they're likely to be within regions that are nearer.

Note that this alternative order does not impact the accuracy of this function! Check out Task 2 where we proved that cl accurately found the closest location, would the proof have still worked if the recursive calls were completed in a different order?[2]

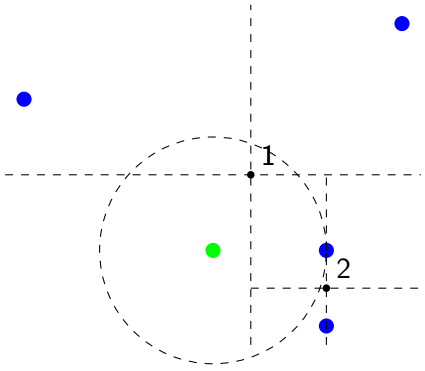
Take the following example. The blue points are part of the location tree, and we want to find the closest of those to the given green point.

(Note, in this case the centroid of the points in the bottom-right corner causes the points to end up on the boundary line. Both points end up in the region to the right of the boundary. See `buildTree` in `location_tree.ts` to see more details!)



Since the green node falls into the SW region of the split at "1", we should start by recursively finding the closest point there. That region is empty, so we move on.

The region next closest to the green point is the SE region, so we recursively search there next. That is itself a split node. 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:



[2] Hint: yes, it would have still worked!

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 "2" split 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, so we need to search it recursively, but when we do so, we find that it has no closer points.

The next closest region in the "2" split is its NE region, but its distance to the green point is larger than the closest point found so far, so we can skip search it all together. The same goes for the SE region of the "2" split. That completes the search of the SE region of the split marked "1". We have a candidate closest point and have performed two distance calculations.

When we return from searching the SE region, we go back to our search of the split marked "1". The next closest region (after SW and SE) is the NE region. We find that the circle does not intersect it, so we can skip it. Likewise for the NW 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 addition to a slightly different recursive search order, we will also change the c parameter and the return value to the following record type:

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

This will allow us to keep track of the closest location found so far as well as its distance to the given location. Since distance calculations are expensive, this will help us avoid recomputing them on every recursive call.

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
```

The `LocTree` type and the other math definitions we defined in the Week 9 definitions sheet are implemented in TypeScript in `locations.ts` and `location_tree.ts`.

(a) 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 point remains the one passed in.)

Otherwise, if the tree is empty, the closest point remains the same. If it is a single node, then we need to perform a distance calculation. If it is closer than the closest so far, we return it; otherwise, we return the existing closest.

Lastly, if we have a split node, then we need to search through the subregions in order by the distance to `loc` (without calling the `distance` function!), make a recursive call for each of them, update the closest each time to the result of the call, and return the closest from the last recursive call as the closest point in the subtree.

There are, only *eight* different orders that you might go through the subregions, so you should implement this with fairly simple `if` / `else` statements.

(b) 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.

Task 5 – I'm All Nears [23 pts]

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

Now we want to add functionality to store a list of friends on the server for each server (check out the next page for a hint on what this will eventually be used for).

We also 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};
```

(a) Add or edit existing route(s) in routes.ts to get and set an updated list of friends per user.

Each "friend" is just represented by their string name.

Feel free to edit any existing data structures, routes, and `index.ts` as you see fit.

(b) 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.

Follow the **TODOS** in getShortestPath to grab friendship information according to how you decided to store it in the last part.

If the two lists contain n and m points respectively, then a naive algorithm would perform $n \times m$ distance calculations. We can use an optimized approach of putting the second list of points into a tree using `buildTree` and then call `findClosestInTree` to find the closest pair of points with only $O(n \log m)$ distance calculations.

(c) 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.

Write/modify tests for setting and getting the list of friends on the server depending on how you implemented this functionality in part a).

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 [24 pts]

Now that the server is working correctly, we will switch over to the **client** and work towards this beautiful final product[3].

We want to change the UI to allow users to choose which users are actually their friends. Then update the map to show all the nearby friends of the user who will also be walking between classes. The bronze dot in the middle of the path represents the point on Kevin's walk to class that is nearest to a point on James' walk to class.

(a) Implement `FriendsEditor` to allow users to Friend and Unfriend the other users. Complete the TODO to render this component in `App` and pass in any needed props.

Add or update fetch request, status code, and response parsing functions in `App` to reflect updated lists of friends on the server and get the list of friends when the main page opens.

(b) 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.[4]

(c) 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] Yours doesn't have to be as beautiful as ours, or even look like this at all, you just need to complete the required functionality. If you want tips on styling things also, feel free to ask!

[4] You're welcome!