

LAB 17 – Shortest Paths

- (1) The goal of this lab is to make use of a pre-computed visibility graph to determine the shortest path for the robot. Download the **Lab17_ShortestPaths.zip** file and unzip it. We will be using pre-computed vector maps that we used before.

The **MapperApp** contains the same menus as in the previous lab but an additional **ShortestPath** menu has been added. The options allow you to show the Shortest Path Tree, the Shortest Path between the predefined start and end locations, the Shortest Path to Mouse Location from the predefined start location and the Shortest Turn Path to Mouse Location. In order for these to work, you will need to first load an **Obstacle Map** from the **PathPlanner** menu.

ShortestPath

- ☒ Hide Shortest Path
- ☐ Show Shortest Path Tree
- ☐ Show Shortest Path
- ☐ Show Shortest Path To Mouse Location
- ☐ Show Shortest TURN Path To Mouse Location

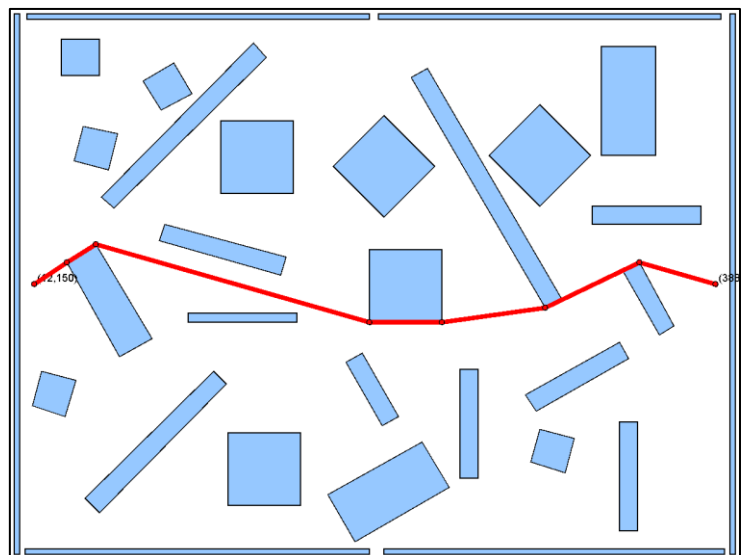
The same java classes (although some have been updated) from the last lab have been provided. You will ONLY be working within the **PathPlanner** class, but you may want to open the **Obstacle**, **Graph**, **Edge** and **Node** classes as well, since you will make use of their various methods.

- (2) We will follow the algorithm in the notes to determine the shortest path. A method called **computeShortestPath()** has been written already which first ensures that the visibility graph has been created and then calls a method called **runDijkstrasAlgorithmFrom(Node s)** ... which is the method you need to write. This method must run Dijkstra's algorithm on the visibility graph (stored as an attribute variable called **visibilityGraph**), where the parameter **s** is the start **Node** of the graph.

There is a **PriorityQueue<Node>** data structure in JAVA that you can use to maintain a list of **Nodes** sorted by their distance from the source. The nodes are sorted by their **distance** attribute as defined in the **compareTo()** method of the **Node** class. The **PriorityQueue** has methods such as **isEmpty()**, **remove()**, **remove(item)** and **add(item)**. The **remove()** method removes and returns the item in the queue with highest priority (i.e., the item at the head of the queue).

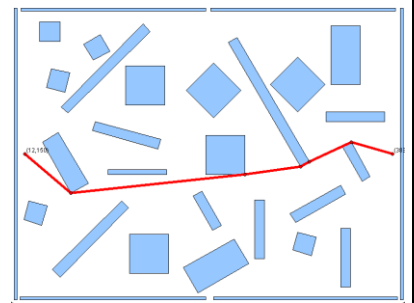
The code for tracing back the path (i.e., slide 22) has already been implemented for you. But for it to work, you will need to properly set the parent notes (see slide 21).

Once you feel that your code is working, run it on the **PathPlannerConvex** environment and select the **Show Shortest Path** radio button. You should see what is shown here ... take a screenshot and save as **Snapshot1.png**:



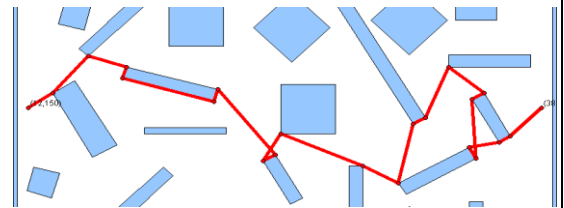
Debugging Tips:

- If you end up with the path shown here, then you are likely making a **WRONG** assumption that **u** is the **endNode** of the incident edge. Each edge in the graph has a **startNode** and an **endNode**. In the algorithm, **u** may be the **startNode** or the **endNode**. So look for a better method to use instead of **getEndNode()**.



- If you do not see any path appear, the problem could be one of a few things:
 - Make sure that you are properly setting the **previous** attribute for each node as you update their costs.
 - Make sure that all your nodes have their distance set to a high number and that they are all added to the PriorityQueue. But make sure that the source node has a distance of **0** to begin.

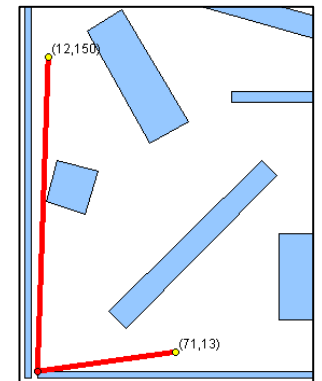
- If you end up with some strange path (such as what is shown here), then likely you are not comparing the distance properly.



- If your code hangs ... then you are not exiting your while loop. This could happen if you never update the costs to the nodes. Double-check your **IF** statement condition. Or perhaps you are not setting the updated distance properly.

Select the **Show Shortest Path to Mouse Location** radio button.

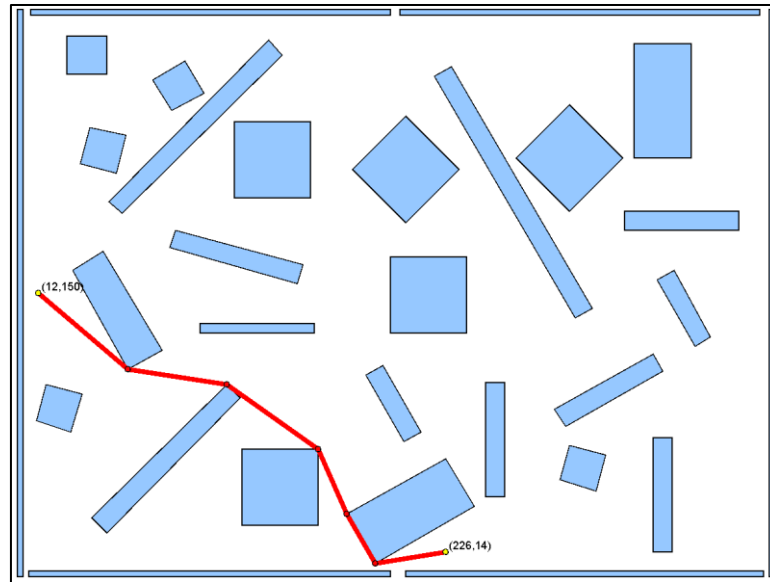
Move the mouse around and you should be seeing an updated path from the start location to the mouse location. Move the mouse location to **(71,13)**. If you see the path shown here on the right, then something is wrong because this is **NOT** the shortest path. In fact, the algorithm is correct, but the problem is in the PriorityQueue data structure. The queue does **NOT** re-sort itself after we update the nodes in the middle of the queue. We need to make sure to sort properly, otherwise the node that we just updated will not be in the proper location in the queue. It will work most of the time, but not all the time.



Unfortunately, there is no method to re-sort/re-locate an updated node in the queue. However, the queue will re-sort itself whenever you add an item to the queue. But we do not want to be adding items. We can remove the updated node from the queue and then add it back to the queue again. That will ensure that it gets to the proper location in the sorted queue ordering.

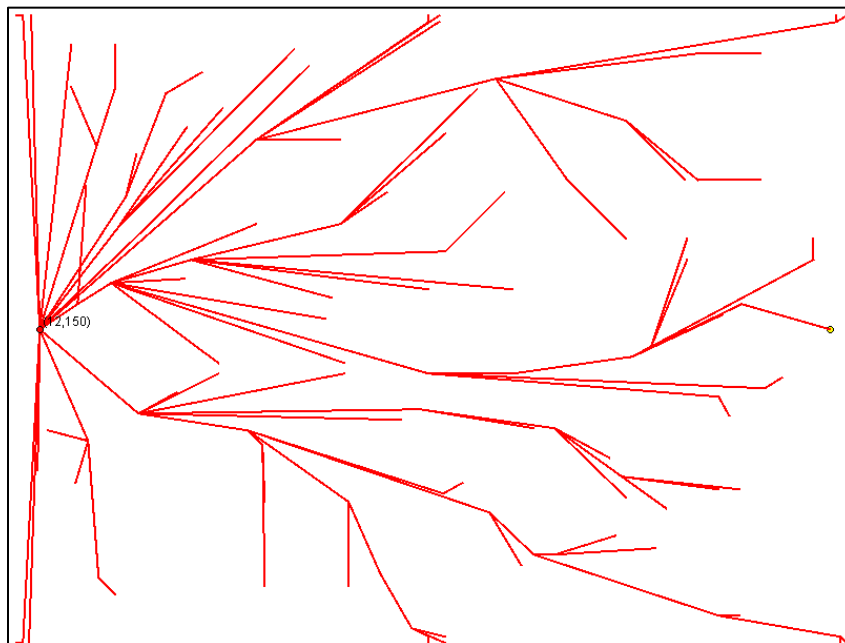
The **remove(item)** method is different from **remove()** in that removes the given **item** from the queue instead of removing the head item. The **add(item)** method adds an item to the queue. See if you can find a way to ensure that the re-sorting takes place when you need to in your code by making use of these functions.

Once you make your changes, ensure that the code works for destination **(71,13)**. Then take a screen snapshot of the path as shown here (on the next page) and save it to a file called **Snapshot2.png**.

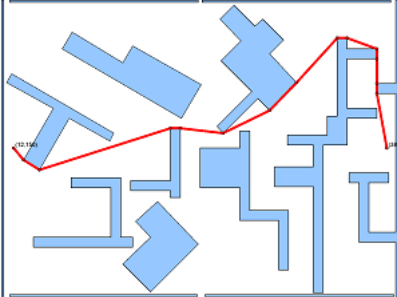
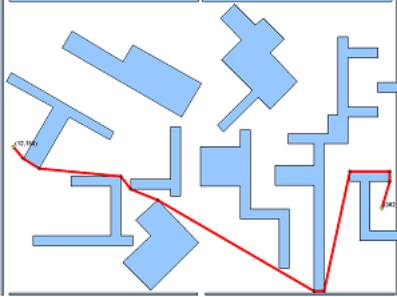
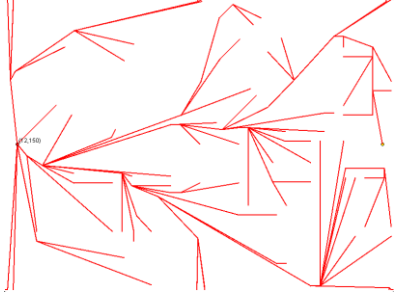


Finally, re-select the **Show Shortest Path** button so that it resets the start and destination to their original locations. Then select the **Show Shortest Path Tree** radio button and unselect **Show Original Obstacles** from the **PathPlanner** menu.

Save the result (as you see below) to a file called **Snapshot3.png**:

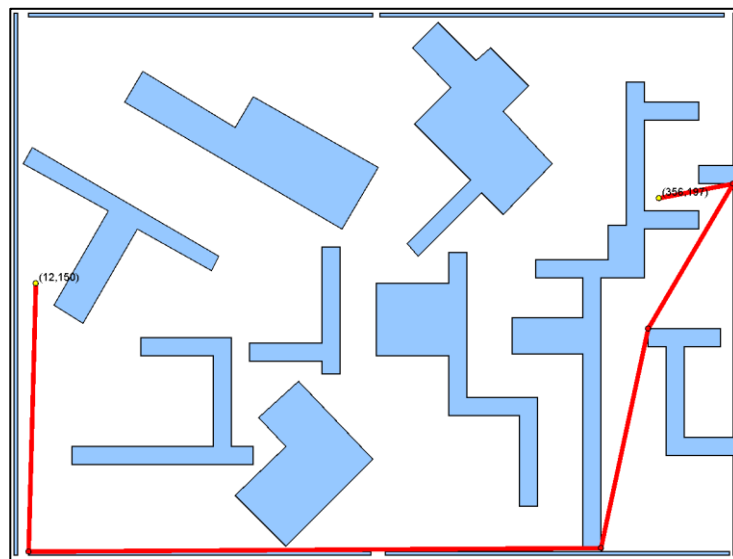


(3) Load the **PathPlannerNonConvex** world and save **3** similar snapshots as follows:

		
Snapshot4.png Show Shortest Path	Snapshot5.png Show Shortest Path to Mouse Location	Snapshot6.png Show Shortest Path then Show Shortest Path Tree

(4) Another blank method has been created called **runShortestTurnPathFrom(Node s)**. Write this code as well. It should determine the best path that minimizes the number of turns along the path. The algorithm is essentially the same as Dijkstra's algorithm ... but with a small change to the cost estimates. See if you can figure out a way to do this by identifying when a turn takes place in the pseudo code of slide **19** and then adjusting it so that making the turn will result in a high cost/weight. The TAs cannot help you much because the solution requires you to write very little code to get the solution. Just share your screen (showing slide **19**) with your partner and put your heads together to think of a way to do this.

When you believe that your code is working, use the **Show Shortest **TURN** Path to Mouse Location** radio button and move the end location to **(356,197)**. You should see something like what is shown below. Take a screen snapshot and save it as **Snapshot7.png**:



Submit **ALL of your java code**, including the files that were given to you. It is important that there are no package statements at the top of your source files (some IDEs require you to put everything into a project or package). Just submit the source files without the package lines at the top (if any from a different IDE). If you are using IntelliJ and are unsure how to do this, please see step **(9)** of the "Using the IntelliJ IDE" file provided.

Submit the **7** snapshot **.png** files.