Final Project

COMP 250 Fall 2022

posted: Tuesday, Dec. 6, 2022
due: Tuesday, Dec. 20, 2022 at 23:59

General instructions

- Submission instructions
  - Please note that the submission deadline for the final project is very strict. No submissions will be accepted after the deadline.
  - As always you can submit your code multiple times but only the latest submission will be kept. We encourage you to submit a first version a few days before the deadline (computer crashes do happen and code post may be overloaded during rush hours).
- These are the files you should be submitting on Ed:
  - `Tile.java`
  - `DesertTile.java`
  - `FacilityTile.java`
  - `MetroTile.java`
  - `MountainTile.java`
  - `PlainTile.java`
  - `ZombieInfectedRuinTile.java`
  - `Graph.java`
  - `GraphTraversal.java`
  - `TilePriorityQ.java`
  - `PathFindingService.java`
  - `ShortestPath.java`
  - `FastestPath.java`
  - `SafestShortestPath.java`

  Do not submit any other files, especially .class files. Any deviation from these requirements may lead to lost marks
- Starter code is provided for this project. Do not change any of the class names, file names, or method headers. You can add helper methods or fields if you wish. Note also that for this project, you are NOT allowed to import any other class (all import statements other than the one provided in the starter code will be removed). Any failure to comply with these rules will give you an automatic 0.

- The project shall be graded automatically. Requests to evaluate the project manually shall not be entertained, so please make sure that you follow the instruction closely or your code may fail to pass the automatic tests.
- Whenever you submit your files to Ed, you will see the results of some exposed tests. These tests are a mini version of the tests we will be using to grade your work. If your code fails those tests, it means that there is a mistake somewhere. Even if your code passes those tests, it may still contain some errors. We will test your code on a much more challenging set of examples. We highly encourage you to test your code thoroughly before submitting your final version.
- By next week we will share with you a `Minitester` class that you can run to test if your methods are correct. This class is equivalent to the exposed tests on Ed. Please note that these tests are only a subset of what we will be running on your submissions. We encourage you modify and expand this class. You are welcome to share your tester code with other students on Ed. Try to identify tricky cases. Do **not** hand in your tester code.
- You will automatically get 0 if your code does not compile.
- Failure to comply with any of these rules will be penalized. If anything is unclear, it is up to you to clarify it by asking either directly a TA during office hours, or on the discussion board on Ed.

Learning Objectives

This project is meant for you to practice working with graphs and solve some practical problems related to it. You will realize from the starter code and the pdf instuctions, that for this project you have a little more freedom in your implementation choices compared to previous assignments. You will start by implementing both a DFS and a BFS traversa. Then you'll spend sometime implementing two data structures: one representing a weighted graph, and the other a priority queue. Finally, you will dive into the implementation of Dijkstra's algorithm for finding the shortest path on a given graph. You will learn more about why this algorithm finds the optimal solution in COMP 251.

Project set up

For this project, you would be using a GUI that is programmed in JavaFX, so you need to set up JavaFX in your IDE properly.

- **For Intellij user (recommended):**
  - **Windows user:** It should be already included in the SDK if you are using Java 1.8 or higher.
  - **Mac user:** By default you laptop might be using Amazom Correto distribution, you need to change it to Liberica distribution to support media.
    1. open File → Project Structure → SDKs → Add → Download new SDKs → Select Liberica and install it
    2. In your run configuration, select Liberica as your build SDK and build the project
- **For Eclipse user:**
  - **Windows user:** You need to install JavaFX library manually
    1. In Help menu, in Install new software wizzard you should add the new site location to find proper software. Use "Add" button, then in "name" section type "e(fx)clipse (or anything you want, it does not matter). In "location" section type: `https://download.eclipse.org/efxclipse/updates-nightly/site/`
    2. Search downloadable package by applying a filter "e(fx)clipse" you should see a list of options (such as JavaFX SDK)
    3. Install them all, after that Eclipse will restart
    4. In Eclipse select the project, run Project → Preferences → Java Build Path → Add Library → Select JavaFX SDK, then rebuild the project, all errors should go away
  - **Mac user:** switch to Intellij

Introduction

Figure 1: Referred from [1]

In a not-so-distant future, a zombie apocalypse has ravaged the planet and made resources scarce. A few years post-apocalypse, mother nature has taken its course once more and covered the world in lush greenery. Cities turned to ruins and became a hub for zombies to hide during the day while at night they roam and hunt for new flesh. Resource gathering time is limited and over the years, the use of technology has dwindled to a select few who are capable. You are among the rare few who are still capable of programming. Luckily, one of the elders has assigned you the task of creating an app, that will help our people scavenge for resources while also avoiding the zombies. The responsibility now lies on your shoulders as this app could help all of humanity scavenge and fight against the undead.
Luckily, you do not need to start from scratch, because you accidentally found an old map app in the computer that can provide a simple graphic interface (GUI). However, the underlying main functionalities have been corrupted so you need to finish the rest for it to work properly.

Path finding from your infected house to a safe house

Your main task for coding this app is to consider all the different elements of nature like desert, mountains, etc, and to devise a plan to travel to the safe house.
On the way to the safe house, one might have to gather some supplies, travel through the metro and even

kill some of those pesky zombies. Being able to successfully compute the best route to take in order to reach the safe house would make the risk of getting out there calculated and worth taking.
Now hurry up and get to coding before the zombies come knocking on your door!!

GUI

Luckily for you the GUI of the app is still functional and has the following sections:

- **Menu:** The menu provides ways to navigate through maps and also supports functionalities to modify the GUI visual output.
  - Control: basic command to manipulate the map.
  - Maps: where to initialize maps.
  - View: utility functions related to map display:
    - Display system log: a toggle whether or not to show the system log.
    - Display tile text: a toggle on whether or not to show text for each tile in the map.
    - Display grid: a toggle whether or not to show grid border.
- **Main map display:** Different parts of the city are modeled as maps and are displayed here in a 2D fashion. The layout of the map, the departure and destination points, and the suggested paths are displayed here.
- **Commanding panel:** This is where different commands are issued based on users' needs. Your main task would be writing the code for each button and making sure they are executed correctly.
- **Console panel:** This is where important messages are displayed, including system messages and user messages.

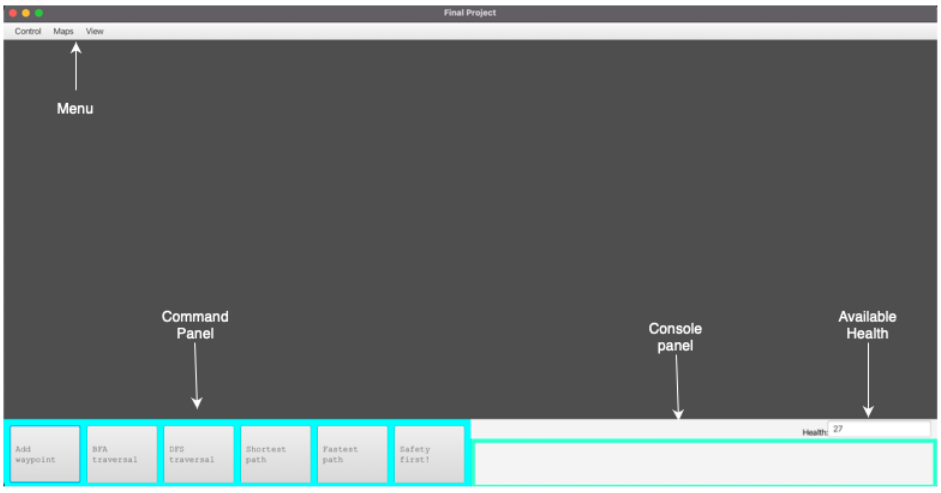
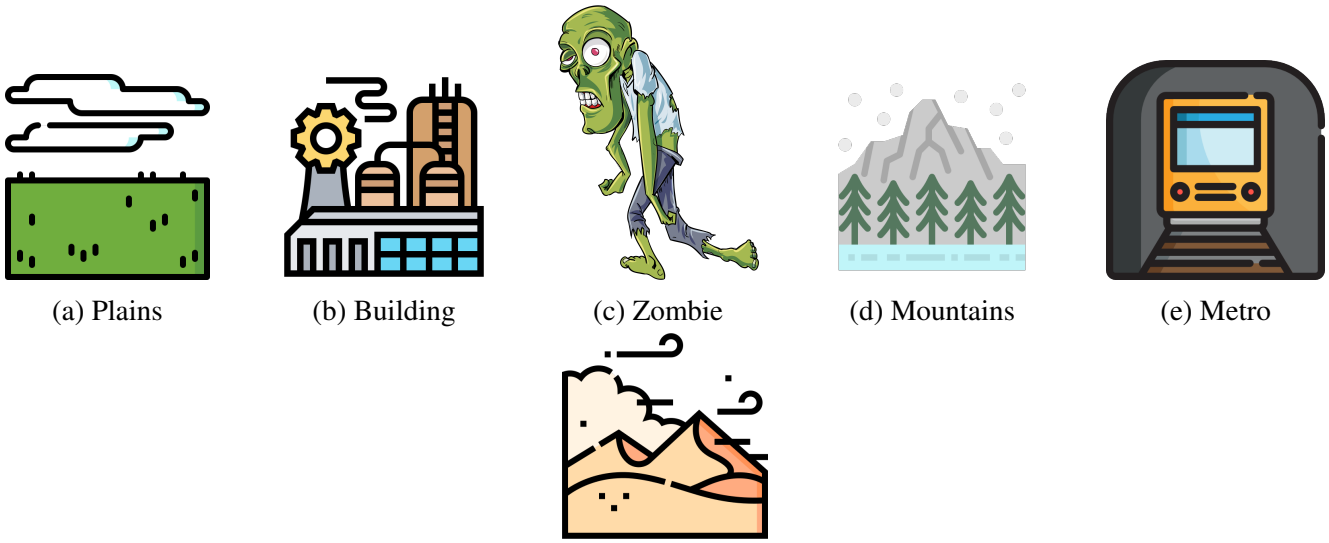


Figure 2: GUI

Map

The map is 2D grid-based for easy demonstration. It consists of **six different base regions**, plains, deserts, mountains, facilities, and zombie-infected ruins. It also labels the location of the departure and the destination. Mountains are generally hard to cross so they can be treated as a non-travelable obstacle. The rest of the tiles can be traveled through with a certain cost in the distance, time, and damage. For example, the desert region may have a straightforward path that is short in distance, but actually traveling through on foot will take a while. Sometimes, you may want to cut through an abandoned building because it is shorter and takes less time, but you might run into zombies and put yourself in danger. In this project, you will try to model these as data and experiment with how this might affect your choice of pathfinding.



(a) Plains (b) Building (c) Zombie (d) Mountains (e) Metro

(f) Desert

Figure 3: Different elements represented in the map[2, 3]

Printing to console

To use the function that shows a message on the GUI, try calling the *logMessage()* function from the `Logger` class. You can use `Logger` to log messages with the following code:

```
Logger.getInstance().logMessage(msg:String)
```

The logger can be accessed anywhere.

Simulating your travel

To ensure that the path devised by your logic is completely accurate, the app has a functionality called "simulation". This would simulate your path and help you visualize your course. To start a simulation, *after generating a path successfully*, **press the simulation button from the Control menu. Turn on the volume for some nice sound effects.**

Your Tasks

Level 0: Warming up [5pts]

In preparation to solve your pathfinding task, you first need to create all the necessary parts for visualizing the outside world on a map. Since the putter world can be modeled using six regions, you first need to make sure that the data related to those regions is correctly initializes. The GUI is provided the template for each region as `Tile` and each specific tile would be a subclass of the `Tile` class.

The `Tile` class has several fields that can be accessed directly from all of the other classes:

- `isDestination`: A boolean variable indicating whether or not this tile is the destination.
- `isStart`: A boolean variable indicating whether or not this is the tile where our path begins.
- `xCoord` and `yCoord`: This tile's x, y coordinates in the map, starting from top left. The row is x and the column is y.
- `nodeID`: A unique index number for each tile object. The only assumption you can make about this number is that it is unique. You can also modify it, if you like, as long as you keep it unique.
- `neighbours`: An array list of all the tiles connected to this tile on the map.
- `distanceCost`, `timeCost`, and `damageCost`: The cost of travelling to this tile in terms of distance, time, and physical damage respectively.
- `predecessor`, and `costEstimate`: two fields which you might find useful when implementing Dijkstra's algorithm.

Find all the subclasses representing each region inside the *tiles* folder, and complete their constructors using the information from the table below:

name/cost	distance	time	damage(risk)
plain	3	1	0
desert	2	6	3
mountain	100	100	100
facility	1	2	0
metro	1	1	2
zombie infected ruins	1	3	5

To test that the costs have been initialized correctly, start GUI and open map 1. Each time you click on an individual tile, the detailed information about that tile should be printed on the console. Fig 3 gives a pictorial representation of different elements of nature which can be found in the GUI.

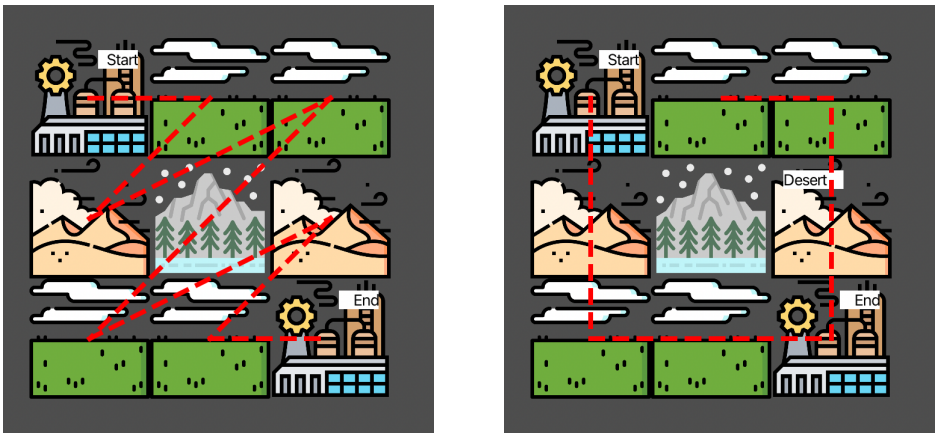
Level 1 [10 pts]

Now that you have made sense of the GUI and you have modeled the world, you are ready to give your first try at implementing an algorithm that would allow you to find a path to the safe house from a given tile. Being a resourceful developer, you quickly open the book "Introduction to Algorithms"[4], a holy grail of Algorithm design, and remember reading something about Breadth First Search(BFS) and Depth First Search(DFS). You decide to implement these two algorithms to see if they do the job.
Open the `GraphTraversal` class and implement the two following `static` methods:

- `BFS(Tile start)` : This method takes a `Tile` as input which represents the starting point of the traversal. It will then traverse the map and find all the reachable tiles from the given input tile using BFS. It returns an `ArrayList` containing the `Tiles` in the same order as they have been visited.
- `DFS(Tile start)` : This method takes a `Tile` as input which represents the starting point of the traversal. It will then traverse the map and find all the reachable tiles from the given input tile using DFS. It return an `ArrayList` containing the `Tiles` in the same order as they have been visited.

NOTE: Some tiles are not designed to be traveled through, thus you can use the method **`isWalkable()`** from the `Tile` class to avoid these obstacle tiles in your traversals.

Testing
To test the correctness of your implementation, open GUI and go to Map 1. Try clicking on **BFS traversal** or **DFS traversal**. You should see a red dotted path that follows the order of visits and visit all reachable tiles on the map. Fig 4 highlights the expected output for Map 1. Please note that depending on your implementation of DFS, you might see a different path and that's ok.



(a) BFS Traversal (b) DFS Traversal

Figure 4: A snapshot of Map 1 for BFS and DFS Traversal

When you work with larger maps, it might be hard to understand the order in which the tiles are reached just by looking at the path drawn. Try opening [Control]→[Start Simulation] after executing the algorithm, it may help you visualize the path better.

Level 2 [15 pts]

Your initial intuition for using BFS and DFS was a good one. However, it has one major flaw: the path obtained traverses through all the points around the world to get to the safe house. You quickly notice that this might not be sustainable, as the complexity(O(n^2)) increases.

Hence, you go back once again to your trusted textbook and this time you find a better algorithm for the task at hand: Dijkstra's algorithm. You remember learning from class, that this algorithm is used to find the shortest path from point A to point B on a positively weighted graph. [1] In a couple of sections you'll finally get to implement it yourself!

To get there though, you first need to think about how to implement the two data structures the algorithm needs. Let's start by implementing a weighted graph. Open the class `Graph`. This class defines a data type to represent a weighted graph. This is also a double directed graph, where the cost of travelling between two tiles connected by an edge is determined by where we are headed. That is, the edges of the graphs are directed, and their weight depends on the destination `Tile`:

$$weight(Edge(t1, t2)) = cost(t2)$$

$$weight(Edge(t2, t1)) = cost(t1)$$

Depending on the graph you need to build, you will refer to the appropriate cost stored in the `Tile` object. Your task is to implement this class so that you can use it to represent a map of the outer world on which you'd like to eventually build paths with minimum weight. The details of the implementation are left up to you. In order for us to assign points for this task, we require you to implement at least the following methods (for which you cannot change the header). You are welcome to add as many fields and methods (`public` or `private`) as you see fit (this includes also overloading any of the methods listed below if this is what you'd like to do).

- `Graph(ArrayList<Tile> vertices)`: a constructor that builds the graph given a list containing all of its vertices.
- `addEdge(Tile origin, Tile destination, double weight)` : a method adds to the graph an `Edge` with the given `weight`, connecting `origin` to `destination`.
- `getAllEdges()` : a method that takes no inputs and returns an `ArrayList` containing all the `Edges` from the graph.
- `getNeighbors(Tile t)`: a method that takes a `Tile` as input and returns an `ArrayList` containing all the `Tiles` adjacent to it.
- `computePathCost(ArrayList<Tile> path)`: this methods takes as input a list of `Tiles` representing a path. It computes and returns a `double` indicating the total weight of the path (i.e. the sum of weights for all edges along the path).

Please note that inside the `Graph` class you can find a `static` nested class called `Edge`. This class is meant to represent a directed edge connecting two `Tiles` in the graph. This class must contain the following methods/fields (as with `Graph` you are welcome to add anything that you might find useful for your own implementation):

[1] For additional details on Dijkstra's algorithm see here.

- Three fields:
  - `origin` : a `Tile` indicating where the edge is originating from.
  - `destination` : a `Tile` indicating where the edge is directed to.
  - `weight`: a `double` indicating the weight associated to this edge.
- `Edge(Tile s, Tile d, int cost)`: A constructor that uses the inputs to initialize an object of type `Edge`.
- `getStart()` and `getEnd()`: two getters used to access the corresponding `Tiles`.

Testing
To be able to test your code for this section using the GUI, you will first need to implement Dijkstra's algorithm. You are encouraged to test your code on your own before moving forward.

Level 3 [15 pts]

The other data structure that you need to implement before tackling Dijkstra's algorithm, is a priority queue. You remember from class that the best way to implement a priority queue is to use a min heap. You also remember that since heaps are complete binary tree, you had learned a wonderful trick that allowed you to implement the entire data structure using an array! Open the `TilePriorityQ` class. This class is meant to represent a priority queue where the elements are `Tiles` and they are compared based on the cost estimated to reach each tile from a source tile. Like for the `Graph`, you will have some freedom in your decisions of how to implement the priority queue. In order to assign points for this task, we will require the following methods to be implemented. You are welcome to add as many fields and methods (`public` or `private`) as you see fit (this includes also overloading any of the methods listed below if this is what you'd like to do).

- `TilePriorityQ (ArrayList<Tile> vertices)`: a constructor that builds a priority queue with the `Tiles` received as input.
- `removeMin()` a method that takes no inputs and removed the `Tile` with highest priority (i.e. minimun estimate cost) from the queue.
- `updateKeys(Tile t, Tile newPred, double newEstimate)`: a method that takes as input a `Tile t`. If such tile belongs to the queue, the method updates which `Tile` is predicted to be the predecessor of `t` in the minimum weight path that leads from a source tile to `t` as well as the estimated cost for this path. Note that this information should be stored in the appropriate fields from the `Tile` class, and after these updates, the queue should remained a valid min heap.

Testing
You are highly encouraged to test that your priority queue works as expected before starting to implement the code from the next section.

Level 4 [25 pts]

Let's now look into the `PathFindingService` class. This class contains the following public methods:

- A constructor that takes a `Tile` as input, representing the starting point of the paths we'd like to compute.
- An `abstract void` method called `generateGraph()`. This method, which you'll need to override in the `PathFindingService`'s subclasses, is supposed to build a graph connecting all reachable tiles (i.e. ignoring the obstacle tiles) from the source tile. It should then use this graph to initialize the corresponding `Graph` field. This will be the graph on which our algorithm will compute the path with a minimum weight.

In addition to the latter, there are the following three methods which will be discussed throughout the next few sections. As with the previous two classes, you are welcome to add any additional method you see fit.

- `findPath(Tile startNode)`
- `findPath(Tile start, Tile end)`
- `findPath(Tile start, LinkedList<Tile> waypoints)`

You are finally ready to implement Dijkstra's algorithm. You have been provided with a class named `ShortestPath` that extends the

Complete the following tasks to get the shortest distance path

- Step 1: In `ShortestPath`, implement `generateGraph()`. The method creates a weighted graph using the **distance cost** as weight. This graph should be then stored in the appropriate field. To make sure that the graph is generated each time a `ShortestPath` object is created, you should add a call to this method inside the constructor.

  Note: You can use BFS or DFS to help you get a list of all reachable tiles. Remember also that the graph you want to build should only connect tiles that are designed to be travelled through. You can use the method `isWalkable()` to help you figure out which tiles are not just obstacles.

- Step 2: Implement Dijkstra's algorithm in `PathFindingService` (Fig 5). Use the algorithm to implement the `findPath(Tile startNode)` method. The method uses Dijkstra's algorithm on the `Graph` stored in the field `g` to find a minimum weight path to the destination from the input `Tile`. Note that the result of running Dijkstra's algorithm is that each node in the graph will contain the information needed to find the minimun weight path from the source to this node. So, after running the algorithm you will need to use the infomation stored in the `predecessor` field to backtrack and find the list of `Tiles` that make up the path to be traversed from the start node to the destination.

```
DIJKSTRA(V, E,w,s):
    INIT-SINGLE-SOURCE(V,s)
    S ←   Q ← V
        while Q   do
        u ← REMOVE-MIN(Q)
        S ← S   {u}
        for each vertex v   Adj[u] do
            RELAX(u,v,w)
```

Figure 5: Pseudo-code for Dijkstra's algorithm

```
INIT-SINGLE-SOURCE(V,s)
    for each v  V do
        d[v]←
        [v]← null
    d[s]← 0
```

Figure 6: Pseudo-code for the initialization

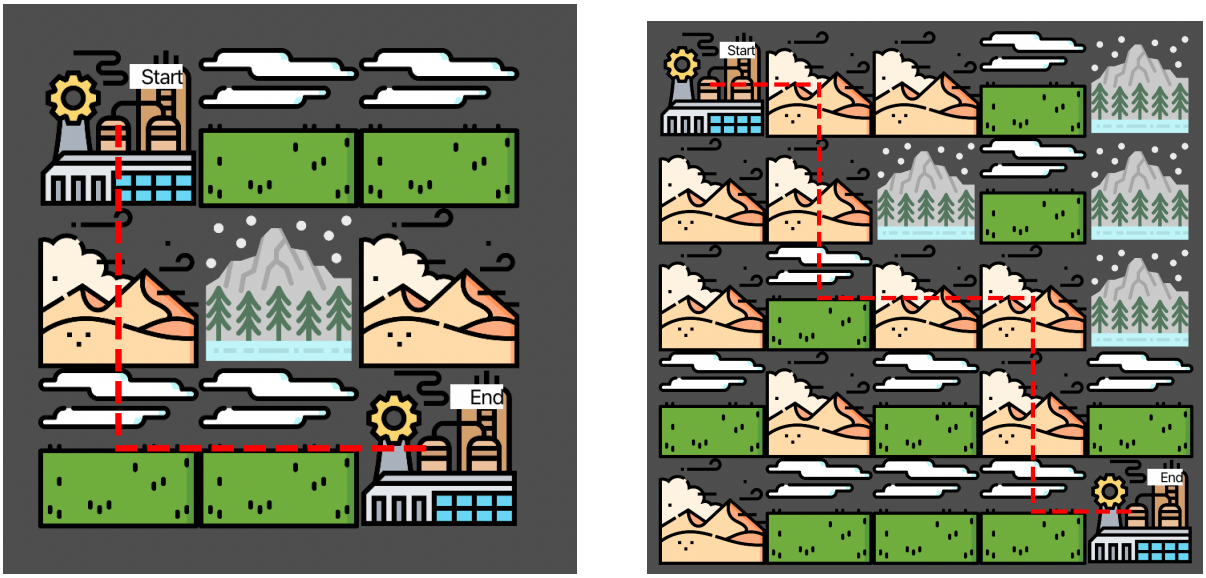
```
RELAX(u,v,w)
    if d[v] > d[u]+w(u,v) then
        d[v] ← d[u]+w(u,v)
        [v] ← u
```

Figure 7: Pseudo-code for the relaxing operation

The reason why we are implementing the path-finding algorithm in the `PathFindingService` class is that when later creating the second strategy for the time cost you would be using the same Dijkstra's algorithm so it is much cleaner to write the code in the parent class. However, you need to write your own graph generation method in the subclass because it is essentially the difference between these path-finding strategies.

Testing

To test this code, you can open either Map 1 or Map 2 and click on the button "Shortest Path". A track will be highlighted in red that will show you the shortest way to reach the safe house. You can click on the *Simulate* button to simulate your path. Fig **??** highlights the expected output for both the maps (please note that the shortest path is not unique. There might be more than one path with the same minimum weight! Your code does not have to generate the same path as in the figure as long as its weight is minimal).



(a) Map 1 (b) Map 2

Figure 8: Shortest path for both maps

Level 5 [5 pts]

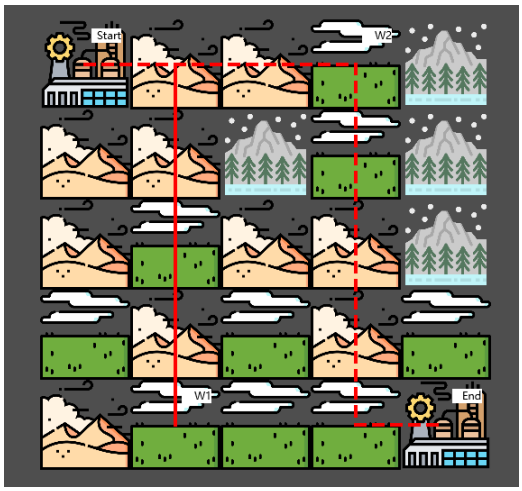
In the previous level, you were able to find the shortest path to the safe house. Unfortunately, that is not enough as you would need to gather supplies to survive in the wild. To make sure that the app is realistic and considers this constraint, you are provided with a functionality called “Add Waypoints”. You can place the waypoints manually using the GUI. For placing waypoints you can click on the button ”Add Waypoints” and then you can add the waypoints to the supply locations.

To accommodate these changes, you would need to make modifications to the code. This will require you to implement the remaining two `findPath` method from `PathFindingService`. These are the steps you should follow:

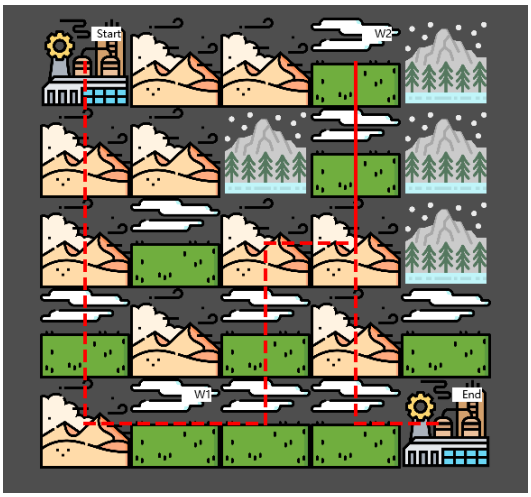
- Step 1: Implement the `findPath` method that takes the start and end Tiles as its input. This method is very similar to the one implemented in Level 4. The only change would be to generate the path to the specific destination tile received as input. For this purpose, notice that Dijkstra’s algorithm will never visit each node in the graph (reachable from the source) exactly once. This means that once a node has been visited by the algorithm, one is already able to figure out what is the shortest path from the source to this node.

- Step 2: Implement the last and final `findPath` method, which takes a starting node and a list of waypoints as input. This method builds the shortest paths from the source to the destination, making sure to visit the each of the waypoints in the order in which they have been provided as input. Use the other methods that you have already implemented to help you find such path. Please note that: the destination tile will not be provided within the list of waypoints. You can figure out which one is the destination tile by accessing the field `isDestination` from the `Tile` class.

Testing
For testing this code, you can open Map 1 or Map 2. You can click on "Add Waypoint" and add waypoints anywhere on the map. Then, you can click on "Shortest Path" to get a path traversing through your supply pointa and going to the final destination. Fig 12 gives a graphical example of sample output. In the figure, we have added two waypoints(W1, W2) and the path traverses through both of them.



(a) Shortest path with waypoints for Map 2



(b) Fastest path with waypoints for Map 2

Level 6 [5 pts]

The elders, who asked you to create this app, seem to be impressed with your ingenuity. However, they feel that this app could use one more feature and that is to find the fastest path. This would be particularly useful for night travel as it makes sense to quickly reach the safe house when zombies are the sharpest.

To find the fastest path, you are provided with a class called `FastestPath` that extends `PathFindingService`. Thankfully, you have already implemented your algorithms to find a minimum weight path inside `PathFindingService`, so all you need to do is generate a graph with the appropriate weights, since in this case you want the algorithm to run on a graph that is weighted by the time cost. Override `generateGraph()` in the class `FastestPath` (similarly to how you did before) to achieve this.

Testing
To test the code, you can open Map 1, and Map 2 and click on the button "Fastest Path". The path should be highlighted using a red line. You can simulate the path by going into [Control] →[Start Simulation], to get a sense of the simulation. You can get a sense of the path by looking at the figure below.

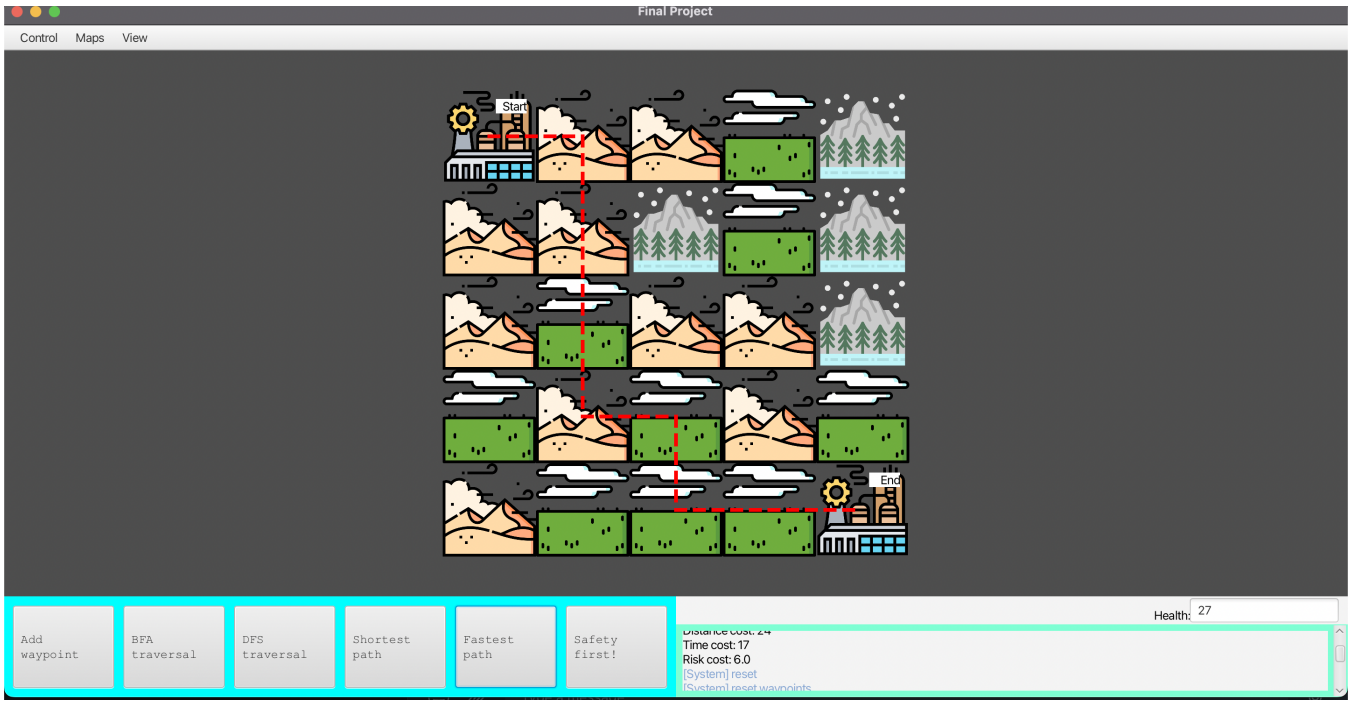


Figure 10: Expected output for the fastest path on Map 2

Level 7 [5 pts]

Building up on your previous level, you thought that it might be better if you integrate the metro stations available all around the city. After mulling over it, you thought that people would reach the safe house at a faster pace using the available metro stations. Hence, you decided to add this feature to the app.
To integrate the subway in your logic, you would need to make a few modifications in your code. The following are the steps to add the subway logic in your code

- You have been supplied with a class named `MetroTile`. You have already initialized a constructor that declares all the variables. Your first task is to implement another method called `fixMetro` that assigns different distance and time costs to metro tiles. This method takes a `Tile` as input. If such tile is another metro tile, then the time and distance costs (i.e. `metroTimeCost` and `metroDistanceCost`) to travel between these two tiles should be computed based on how far the two tiles are. The following are the formulae for calculating the time and distance cost going from one metro station to another:

$$metroTimeCost = M(t1, t2) * metroCommuteFactor$$

$$metroDistanceCost = M(t1, t2)/metroCommuteFactor$$

where the `metroCommuteFactor` variable is set to 0.2 for now. $M(t1, t2)$ is the Manhattan distance between $t1$ and $t2$, use Tile class's *xCoord* and *yCoord* to access their 2-D coordinate and use the formula below:

$$M(t1, t2) = abs(t1.xCoord - t2.xCoord) + abs(t1.yCoord - t2.yCoord)$$

Note that when adding more than 2 metro stations things are getting more complicated because each pair of metro tiles would have their own cost based on the distance, so to simplify this you can assume there are only two metro stations in the district (poor public transportation).

- Modify your code, so that the graph generated by `generateGraph()` in both `FastestPath` and `ShortestPath` class now considers metro weights. That is, whenever you try to add an edge to the graph, if both the start and the end tile for a edge are `MetroTile`, then you need to set the edge's weight/cost using the corresponding value computed in the previous step. Please note that which part of the code you will need to modify really depends on your implementation.

Testing
To test this code, you can go to Map 3 and click on the "Fastest Path" Button. A sample output has been attached below.

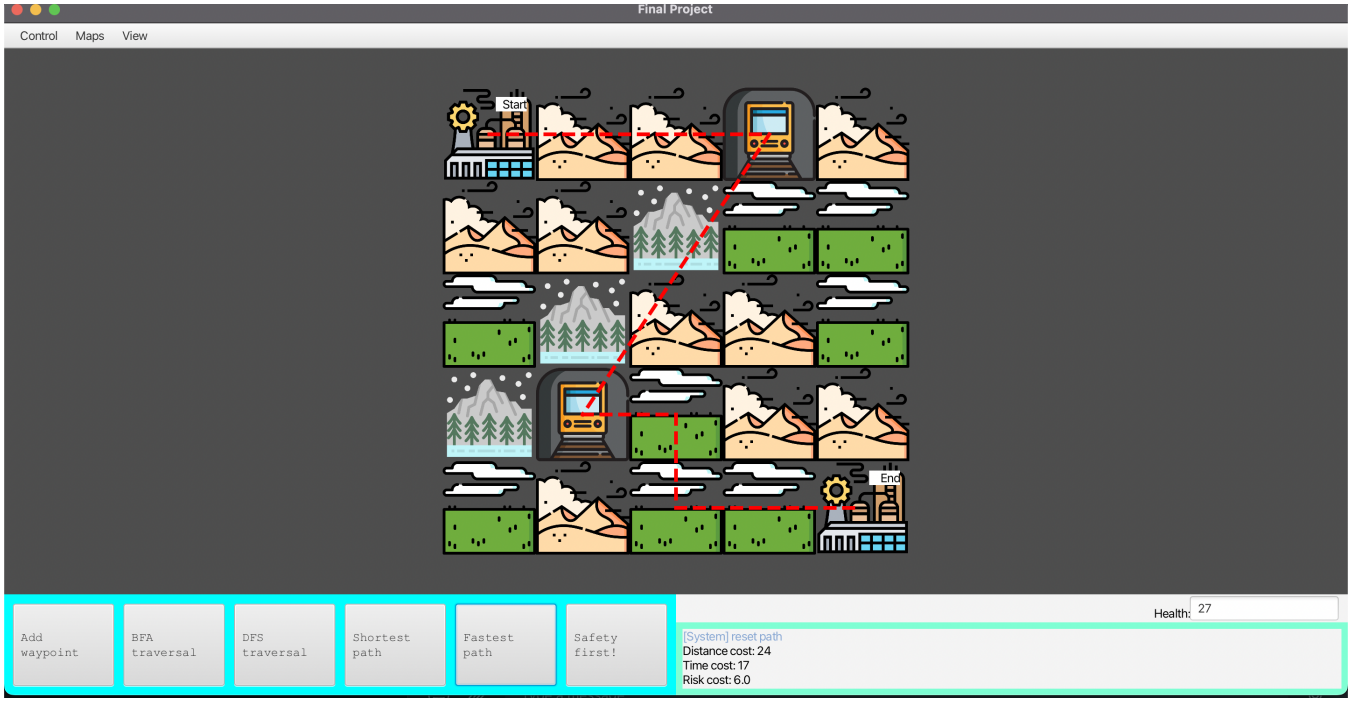


Figure 11: Expected output after integrating metro in Fastest Path

Level 8 [15 pts]

At this stage, you should have implemented Dijkstra's algorithm and created two strategies, one for the shortest distance and the other for the fastest time. However, our agents reported back that the suggested path sometimes would lead our people into the zombie-infected area whenever scavenging more dangerous districts (Map 4). These paths do not seem to guarantee our people's safety. Therefore, our post-apocalypse path-finding service needs a new feature for an optimal path without safety concerns. To simulate and evaluate the possible risk, our agent now has a fixed health (HP) and will take damage when traveling through a dangerous area. Whenever the HP reaches 0 or below, it means our agent's safety is severely affected, thus the path should not be considered. Our goal is to find the shortest path while preventing our agent from dying.

This type of problem is called **constrained shortest path (CSP)** and you will need to implement an algorithm that is known for solving this problem called **LARAC (Lagrangian Relaxation Based Aggregated Cost) algorithm**. In simple words, the algorithm introduces **aggregated cost** to replace graph cost (weight) and optimize it through iterations until it finds the optimal cost (weight) that satisfies the constraint. To know more about the mathematical theory behind it, check out some resources here.

When you first began this project, you have set up various types of costs for each type of tile (region), including `damageCost`, which hasn't been used yet. The field `damageCost` represents how much damage our agent takes when walking on this tile.

For this section, you need to complete the `SafestShortestPath` class which holds the logic for computing the safest shortest path for our agent. This class has the following fields:

- A integer field `health`, model and visualize our agent's life status.
- A Graph called `costGraph` that uses the distance cost as the edges' weights.
- A Graph called `damageGraph` that uses the damage cost as the edges' weights.
- A Graph called `aggregatedGraph` that uses the aggregated cost as the edges' weights.

To complete the class, you need to implement the following methods:

- `generateGraph()` : like for the other two class, you need to override this method so that it initializes the three graphs listed above. For `costGraph` initialize all edges' weight using the distance cost. For the `riskGraph` and `aggreatedGraph`, initialize them with the damage cost. To keep it simple, in this class we will not consider time cost.
- `findPath` : Override the `findPath(startNode, waypoints)` method from the parent class. This method implement the LARAC algorithm that finds the optimal path with our limited HP. Note that the total cost of a path is equal to the sum of the weights of the edges that belong to the path. Now, the algorithm consists of the following steps:

  1. Set the `Graph` field from the superclass to be equal to `costGraph`, and find the optimal path p_c with the least distance cost. If the total damage cost for p_c is less than our health H, return p_c for we have found the optimal path.
  2. Set the `Graph` field from the superclass to be equal to `damageGraph`, find the optimal path p_d with the least damage cost. If the total damage cost for p_d is bigger than our health H, return null for no possible path exists.
  3. Compute the multiplier λ using the equation:

  $$\lambda = \frac{c(p_c) - c(p_d)}{d(p_d) - d(p_c)}$$

  where $c(p)$ is the total distance cost for a path p and $d(p)$ is the total damage cost for a path p. Then update each `aggregatedGraph`'s edge weight to the latest aggregated cost $c_\lambda = c + \lambda * d.$, where c is the distance cost of the edge, and d is the damage cost of the edge.

4. Set the `Graph` field from the superclass to be equal to `aggregatedGraph` and compute the optimal path p_r with the least aggregate cost.
   - If the total aggregated cost for p_r is the same as the total aggregated cost for p_c (our current shortest path without considering any damage factor), then return p_d (our current safest path).
   - else if the total damage cost for p_r is less than or equal to our HP then assign p_r to p_d.
   - else assign p_r to p_c.
5. Repeat Step 3 until a path is returned.

Testing

To test it, start GUI and select Map 4, find a safe path by pressing **Safety first!** button. When you simulate the path, the traveling agent will flash in red if it takes damage and will die if HP is not enough (which is very likely to happen if you naively use the shortest/safest path in Map 4). Feel free to adjust the HP limit using the text field on the screen and see how the resulting path would change based on how much health the agent has.

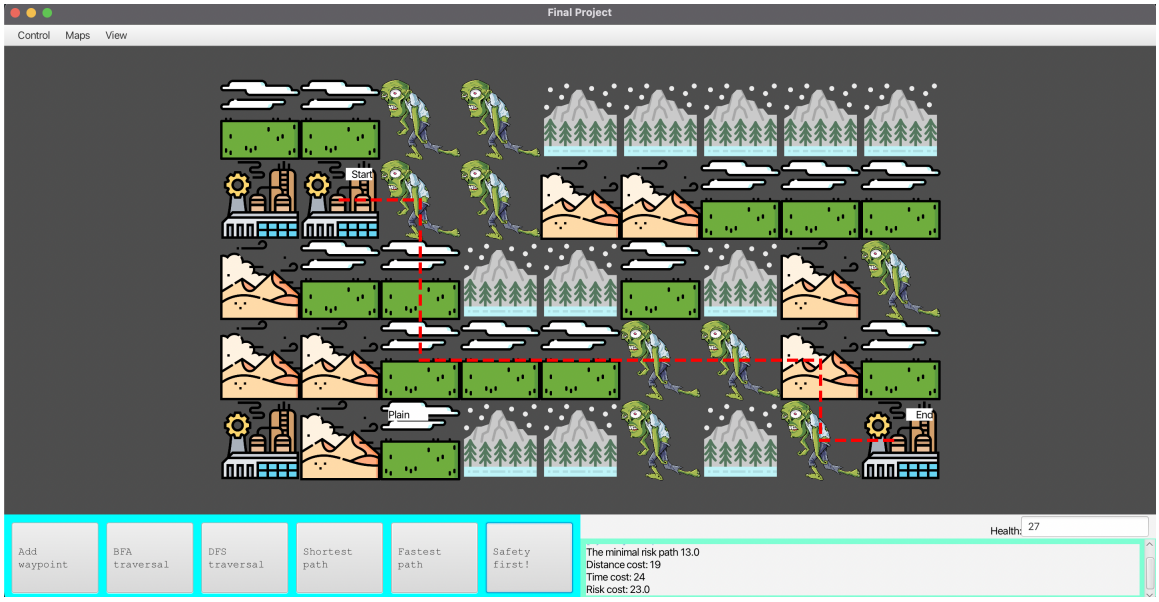


Figure 12: Expected output after implementing LARAC algorithm

References

[1] http://hqdesktop.net/wallpaper.

[2] https://www.pngwing.com/en/free-png-bmvba/download.

[3] https://www.flaticon.com/free-icon/sand-storm_5600772.

[4] Thomas H Cormen et al. *Introduction to algorithms*. MIT press, 2022.

Good luck and remember to run the simulation!