Final Project

COMP 250 Fall 2024

posted:	Wednesday, Dec. 4, 2024
due:	Sunday Dec. 15, 2024, at 23:59 for a chance to receive Mastery, OR Friday, Dec. 20, 2024 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 of Dec 20th.** And no submissions received after Dec. 15 will receive Mastery.
  - We encourage you to start early. As always you can submit your code multiple times (**submissions will be capped at 100**) 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).
  - Your task is to complete and submit the following file:
    * `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.
  - **Do not change any of the starter code that is given to you. Add code only where instructed, namely in the "ADD YOUR CODE HERE" block.** You may add private helper methods to the class you have to submit (and in fact you are highly encouraged to do so), but you are not allowed to modify any other class.

- 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. 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 Inconclusive.**
- Whenever you submit your files to Ed, you will see the results of certain exposed tests along with the competency level you have achieved. A small subset of these tests will also be shared with you to help with debugging. We highly encourage you to write your own tests and thoroughly test your code before submitting your final version. Learning to test and debug your code is a fundamental skill to develop.

  You are welcome to share your tester code with other students on Ed and collaborate with others in developing it.
- Your submission will receive an "Inconclusive" if the code does not compile.
- Failure to comply with any of these rules may result in penalties. If something is unclear, it is your responsibility to seek clarification, either by asking during office hours or posting your question on the Ed.
- IMPORTANT: Do NOT wait until you have finished writing the entire project to start testing your code. Debugging will be extremely difficult if you do so If you need help with debugging, feel free to reach out to the teaching staff. When asking for help, be sure to mention the following:
  - The bug you are trying to fix.
  - What steps have already taken to resolve it.
  - where you have isolated the error.

Learning Objectives

This project provides an opportunity to practice working with graphs and tackle practical problems involving graph traversal and pathfinding. Unlike previous assignments, this project offers greater flexibility in your implementation choices, allowing you to exercise creativity and decision making to solve complex tasks.

Through this project, you will:

1. Implement both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms to explore and analyze graphs.
2. Construct two essential data structures, a weighted graph and a priority queue, using object-oriented principles.
3. Develop a functional implementation of Dijkstra's algorithm to find the shortest path on a positively weighted graph.

The skills gained in this project will prepare you for deeper explorations in graph algorithms and optimization in `COMP 251`, while reinforcing core concepts of data structures and algorithms.

Project set up

For this project, you can use a GUI (provided) that is programmed in JavaFX, so you need to set up JavaFX in your IDE properly. Please note, that the use of the GUI is not necessary to successfully complete the project.

- **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, leaving resources scarce. A few years post-apocalypse, mother nature has reclaimed much of the world, covering it in lush greenery. Cities have turned to ruins, serving as hubs for zombies to hide during the day while they roam and hunt for new flesh at night. Resource-gathering time is limited, and over the years, the use of technology has dwindled to a select few who are still capable.

You are among these rare survivors - one of the few still capable of programming. Luckily, one of the elders has entrusted you with a critical mission: to create an app that will help humanity scavenge resources while avoiding the zombie threat. The responsibility now lies on your shoulders, as this app could be a turning point for humanity in its fight for survival.

Thankfully, you do not need to start from scratch. While exploring an old computer, you stumbled upon a map app that provides a simple graphical interface (GUI). However, the core functionalities of the app have been corrupted, and it is now up to you to restore and complete them to ensure its full functionality.

Pathfinding from Your Infected House to a Safe House

Your main task in coding this app is to account for all the different elements of nature, such as deserts, mountains, and more, to devise a plan to safely travel to the safe house.

On your journey, you may need to gather supplies, navigate through metro stations, and even face off against pesky zombies. Successfully computing the best route to the safe house will ensure that the risk of venturing out is calculated and worthwhile.
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 consists of the following sections:

- **Menu:** The menu provides options to navigate through maps and supports functionalities to modify the GUI visual output:
  - **Control:** Basic commands to manipulate the map.
  - **Maps:** Options to initialize maps.
  - **View:** Utility functions related to map display:
    - **Display system log:** A toggle to show or hide the system log.
    - **Display tile text:** A toggle to show or hide text for each tile in the map.
    - **Display grid:** A toggle to show or hide grid borders.
- **Main map display:** Displays different parts of the city in a 2D grid-based format. This section shows the layout of the map, the departure and destination points, and any suggested paths.
- **Commanding panel:** This section allows users to issue commands based on their needs. Your main task is to write the code for each button and ensure their correct functionality.
- **Console panel:** Displays important messages, including system and user-generated messages.

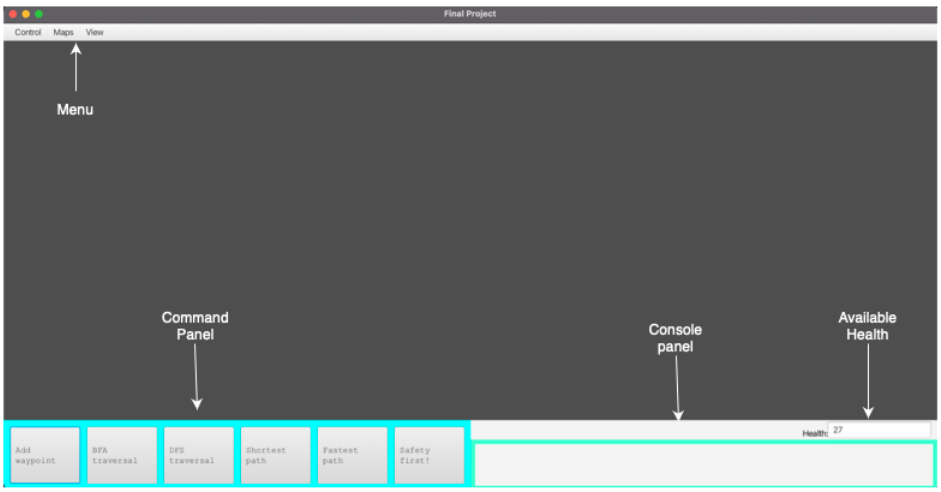


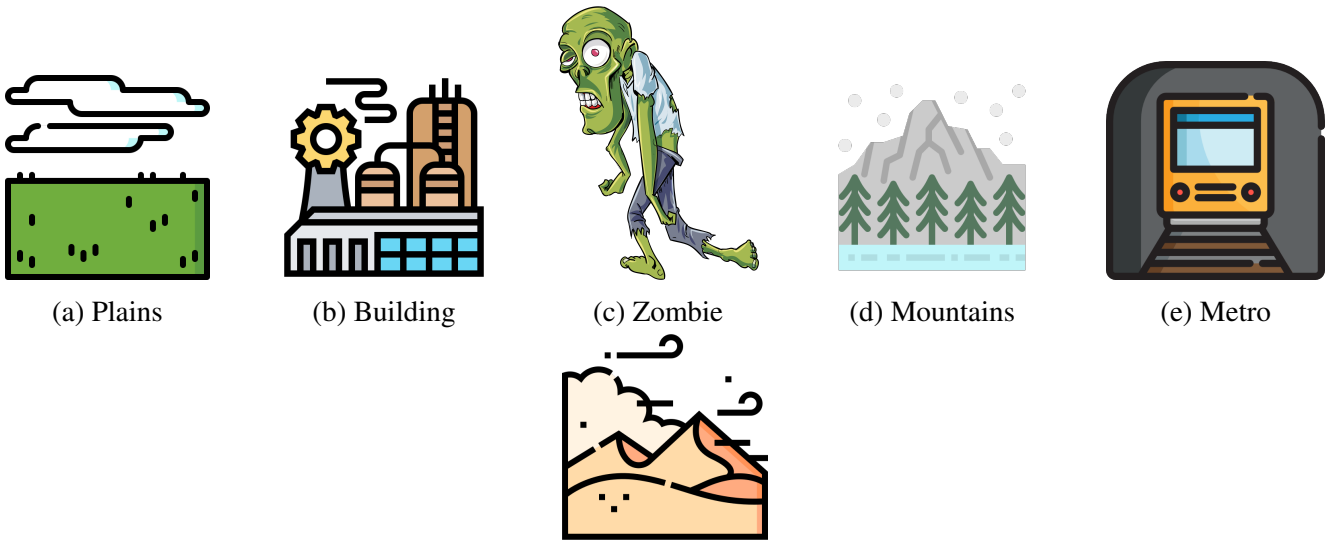
Figure 2: GUI

Map

The map is designed as a 2D grid for easy visualization and demonstration. It consists of **six different base regions**: plains, deserts, mountains, facilities, metro tiles, and zombie-infected ruins. It also identifies the locations of the departure and destination tiles.
Mountains are generally hard to cross and are treated as non-travelable obstacles. The other tiles can be traversed, but each type incurs specific costs in terms of distance, time, and damage. For example:

- The desert region may offer a straightforward path that is short in distance but slow to travel on foot.
- Cutting through an abandoned building may provide a shorter and quicker route but poses a high risk of encountering zombies.

In this project, you will model these regions as data and experiment with how the associated costs influence your choice of pathfinding strategies.



(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 includes a functionality called *simulation*. This feature allows you to simulate your path and visualize your journey.
To start a simulation, *after successfully generating a path*, **press the simulation button from the Control menu**. Don't forget to turn on the volume for some immersive sound effects!

Your Tasks

Level 0: Warming up

As the sun rises on the first day of your mission, you begin setting up the foundation for humanity's survival. The first step is mapping the world around you—a lush yet dangerous landscape with varied terrains. Understanding the lay of the land is essential to plan safe travel routes and avoid perilous obstacles.

The outer world can be modeled using six regions, and your first task is 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.
- `adjacentTiles`: 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: Basic Pathfinding

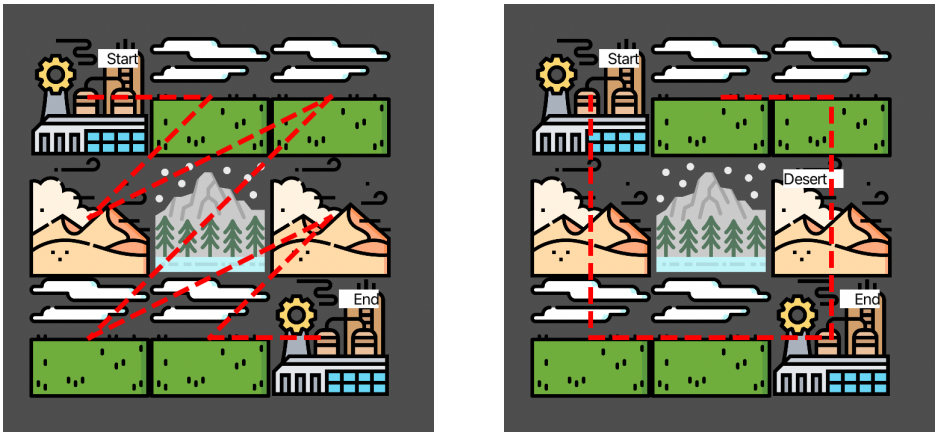
With the map prepared, you set out to explore the area using basic strategies. Guided by your knowledge of Depth-First and Breadth-First Search, you begin scouting paths to the safe house. Though these methods are rudimentary, they lay the groundwork for your ultimate goal: devising an efficient route.
Open the `GraphTraversal` class and implement the following `static` methods:

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

NOTE: Some tiles are not designed to be traversable. Use the method `isWalkable()` from the `Tile` class to filter out these obstacle tiles during 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: Building Weighted Graphs

The world is more complex than it seems. The straightforward traversal methods aren't enough to navigate this dangerous terrain efficiently. You turn to a better representation—a weighted graph—to capture the intricacies of the terrain and prioritize paths with minimal risks.

Hence, you return to your trusted notes, and this time you discover a better algorithm for the task: Dijkstra's algorithm. You recall from class that this algorithm is used to find the shortest path from point A to point B on a positively weighted graph. In a couple of sections, you'll implement this algorithm yourself! To prepare, you first need to consider how to implement the two data structures required by the algorithm. Let's begin by implementing a weighted graph. Open the class `Graph`. This class defines a data type to represent a weighted graph. It is a directed graph where the cost of traveling between two tiles connected by an edge is determined by the destination `Tile`. Specifically:

$$weight(\text{Edge}(t_1, t_2)) = cost(t_2)$$

$$weight(\text{Edge}(t_2, t_1)) = cost(t_1)$$

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 to represent a map of the outer world, on which you will eventually build paths with minimal weight. While the specific implementation details are left up to you, we require you to implement at least the following methods (note that their headers must remain unchanged). You are welcome to add as many fields and methods (`public` or `private`) as you see fit. You may also overload the methods listed below if desired.

- `Graph(ArrayList<Tile> vertices)`: A constructor that builds the graph given a list containing all of its vertices. **This graph should NOT contain any edges. The constructor should be used to initialize the vertices of the graph and any fields you decide to include in this class.**
- `addEdge(Tile origin, Tile destination, double weight)`: A method that adds an `Edge` with the given `weight`, connecting `origin` to `destination`.
- `getAllEdges()`: A method that takes no inputs and returns an `ArrayList` containing all the `Edge`s from this graph.
- `getNeighbors(Tile t)`: A method that takes a `Tile` as input and returns an `ArrayList` containing all the `Tile`s connected to it in this graph.
- `computePathCost(ArrayList<Tile> path)`: This method takes as input a list of `Tile`s 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). You can assume that the input represents a valid path in this graph.

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 `Tile`s 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):

- 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: Priority Queue Construction

To navigate the ever-changing dangers of the world, prioritizing your moves is essential. By leveraging your knowledge of heaps, you will construct a priority queue to dynamically evaluate and select the safest and most efficient paths. This tool will be critical as you progress to more advanced strategies. Since heaps are complete binary trees, you can take advantage of a clever trick you learned to implement the entire data structure efficiently using an array.
Open the `TilePriorityQ` class. This class represents 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. Similar to the `Graph` class, you have some flexibility in deciding how to implement the priority queue.
To earn full points for this task, you must implement the following methods. You are welcome to add any additional fields and methods (`public` or `private`) that you find necessary. Overloading any of the methods listed below is also permitted if it aligns with your design choices.

- `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: Dijkstra's Algorithm

Now equipped with the tools to dynamically assess paths, you are ready to implement Dijkstra's algorithm—a powerful technique for computing the shortest route to safety. Every decision you make brings humanity one step closer to survival.
Open 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 minimum weight path from the source to this node. So, after running the algorithm you will need to use the information 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

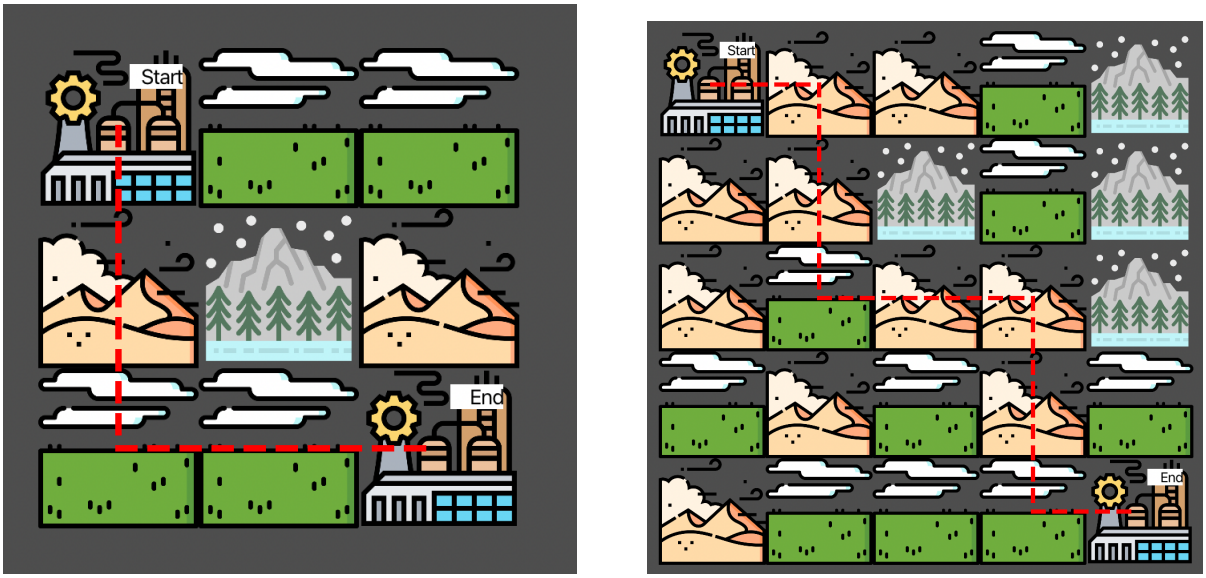
```
RELAX(u,v,w)
    if d[v] > d[u]+w(u,v) then
        d[v] ← d[u]+w(u,v)
        [v] ← u
```

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 8 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: Waypoints

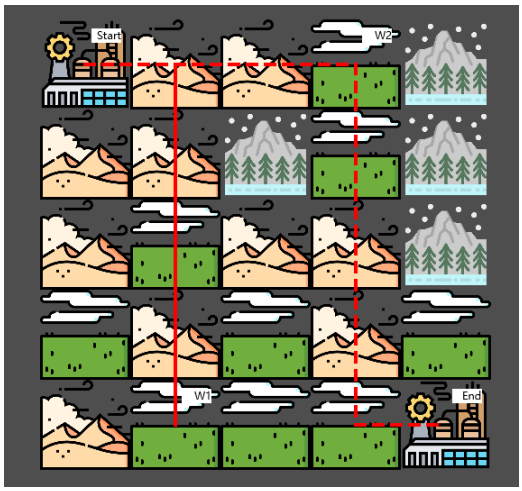
The journey is not just about reaching safety—survival also depends on gathering critical supplies along the way. By integrating waypoints into your algorithm, you can account for these essential stops while still optimizing the overall route.

The app includes a functionality called *"Add Waypoints"*, allowing you to manually place waypoints using the GUI. To place waypoints, click on the *"Add Waypoints"* button and select supply locations on the map. To accommodate these changes, you will need to modify the code by implementing the remaining two `findPath` methods in the `PathFindingService` class. Follow these steps to complete the implementation:

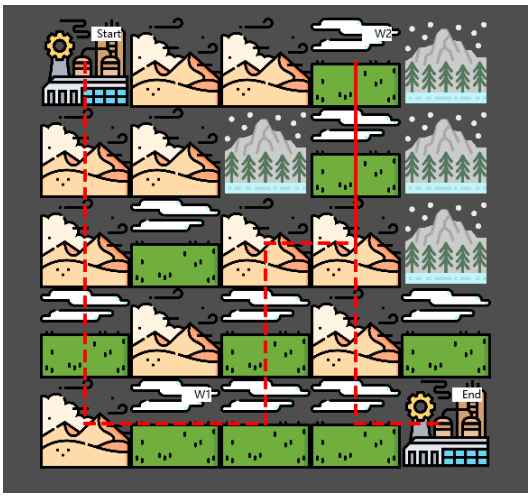
- Step 1: Implement the `findPath` method that takes the start and end Tiles as its input. This method is very similar to the one implemented in Level 4. The only change would be to generate the path to the specific destination tile received as input. For this purpose, notice that Dijkstra's algorithm 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: Fastest Path

Sometimes speed is more important than caution, especially at night when zombies are most active. You adapt your tools to prioritize time efficiency, ensuring a swift escape in the darkest hours.
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.

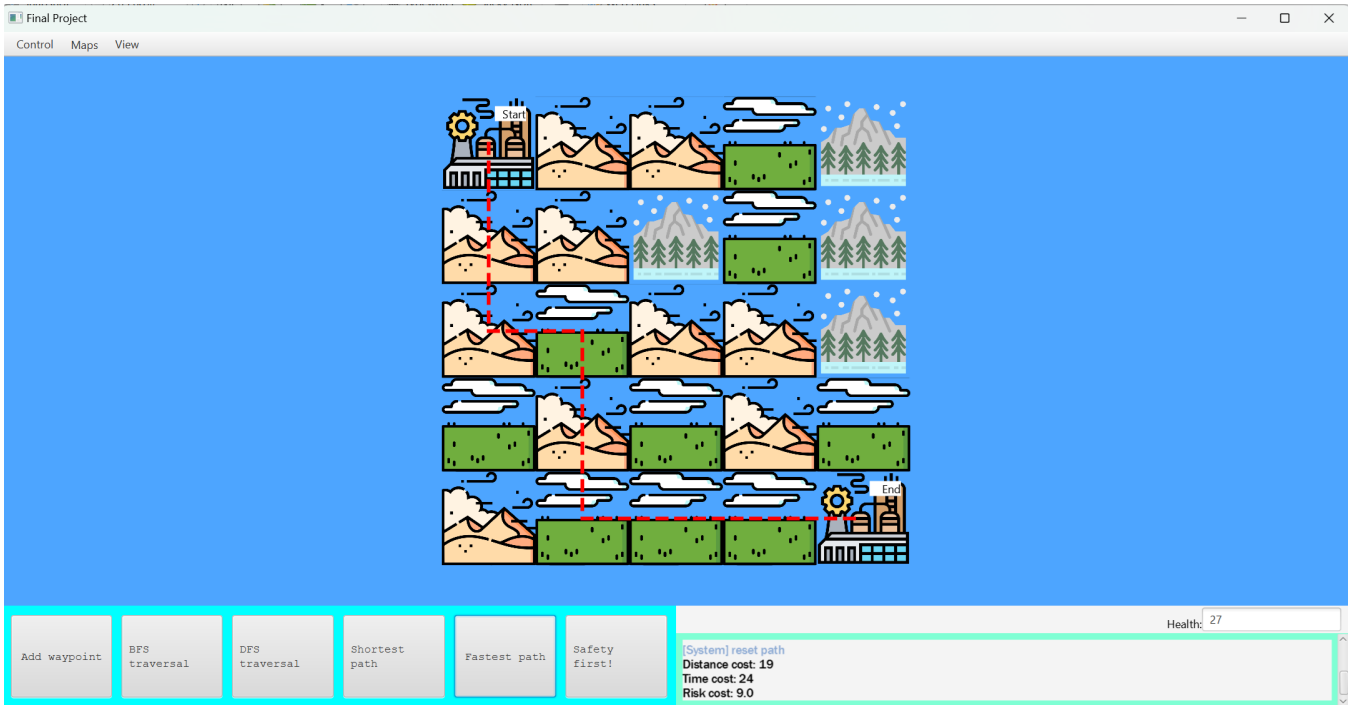


Figure 10: Expected output for the fastest path on Map 2

Level 7: Metro Integration

The ruins of the metro system offer a glimmer of hope for faster travel. By incorporating metro tiles into your pathfinding algorithm, you leverage these remnants of technology to enhance your routes and outpace the undead.

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.

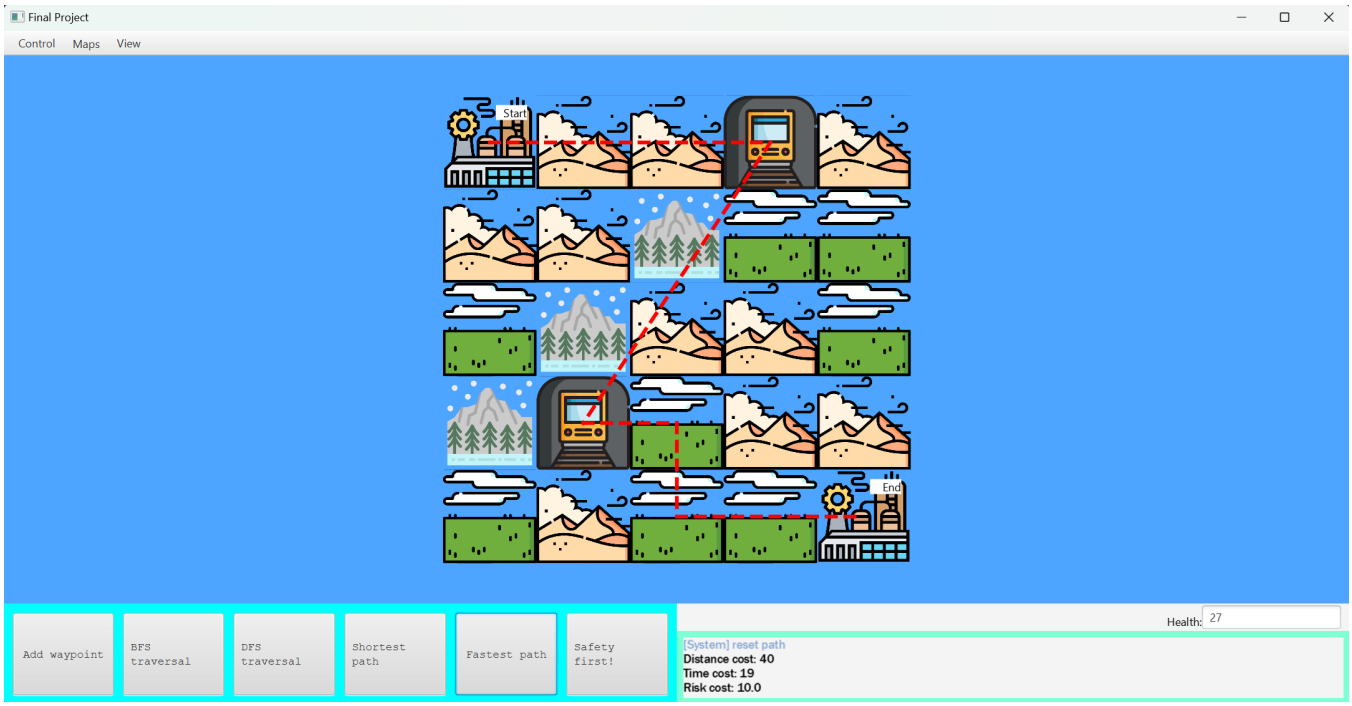


Figure 11: Expected output after integrating metro in Fastest Path

Level 8: Safest Shortest Path

By now, you have implemented Dijkstra's algorithm and developed two strategies: one for the shortest distance and another for the fastest time. Despite these efforts, navigating zombie-infested areas remains a deadly challenge, especially in high-risk districts like those in Map 4. These paths fail to guarantee the safety of our people.
To address this issue, our post-apocalypse pathfinding service must incorporate a new feature that balances safety with efficiency. To simulate and evaluate potential risks, agents now have a fixed health (HP) that decreases when traveling through dangerous areas. If an agent's HP drops below 0, the path must be deemed invalid.
Your task is to develop a solution that finds the shortest path while ensuring the agent's survival.

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.

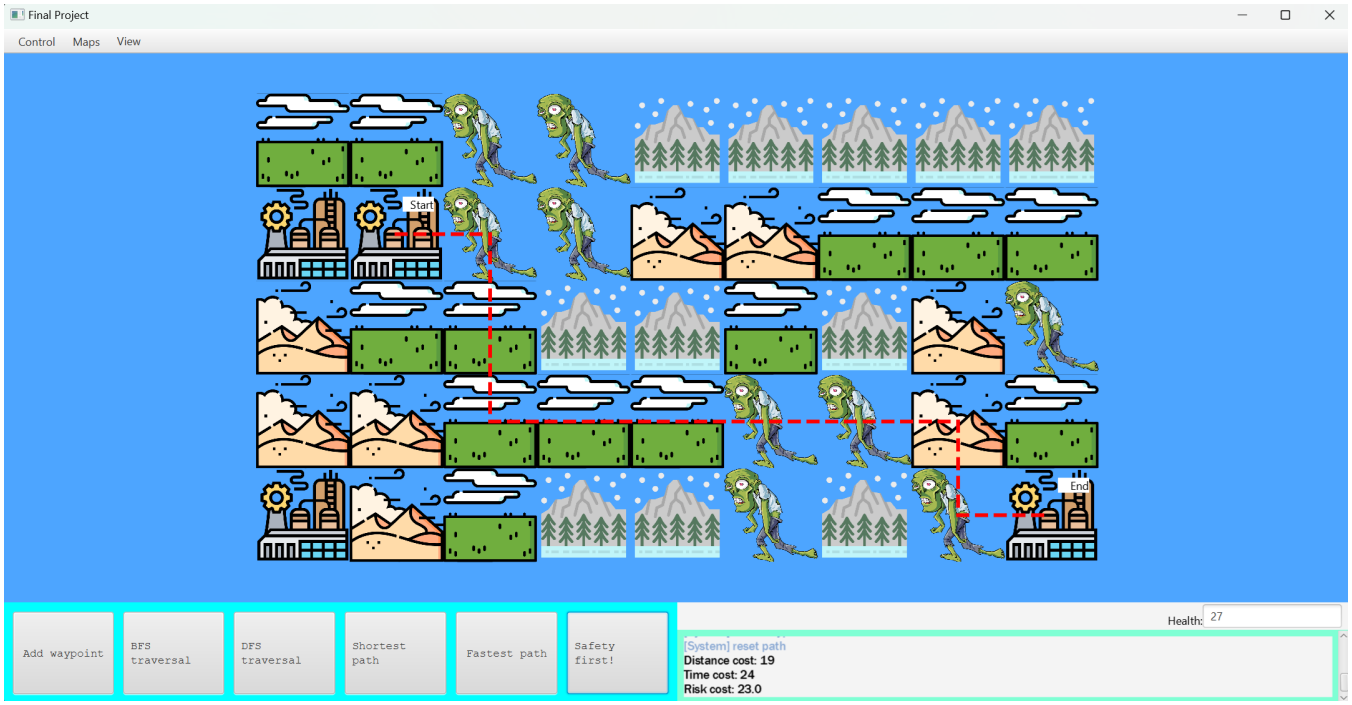


Figure 12: Expected output after implementing LARAC algorithm

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.

Good luck and remember to run the simulation!