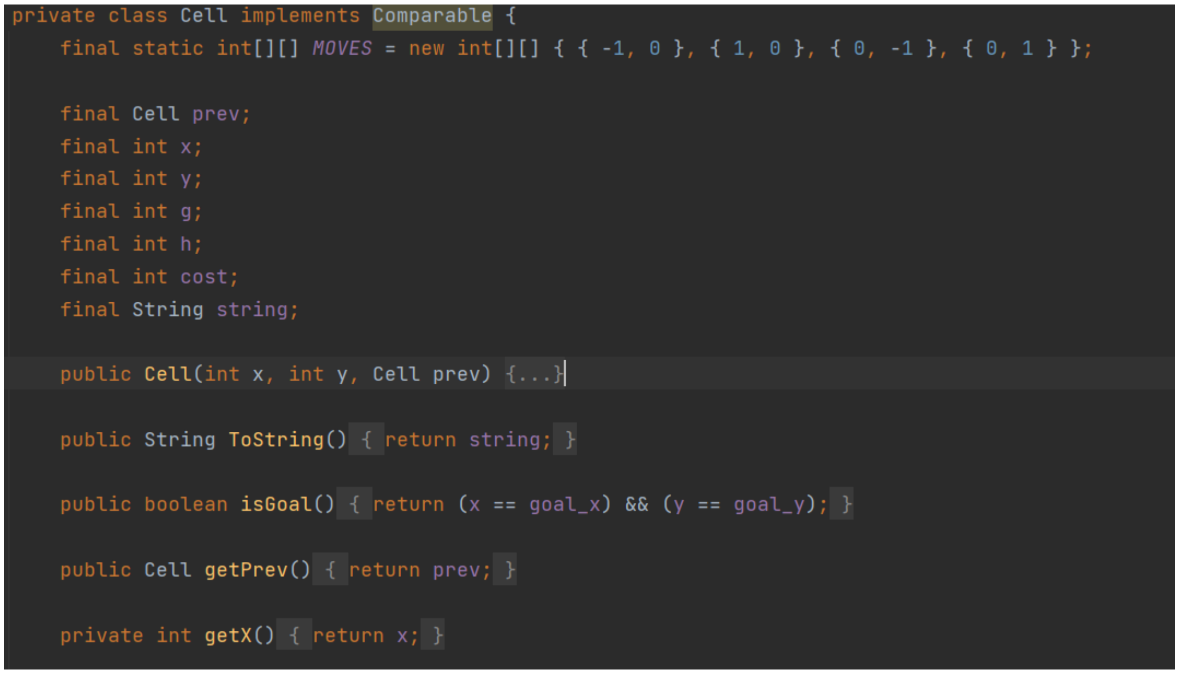
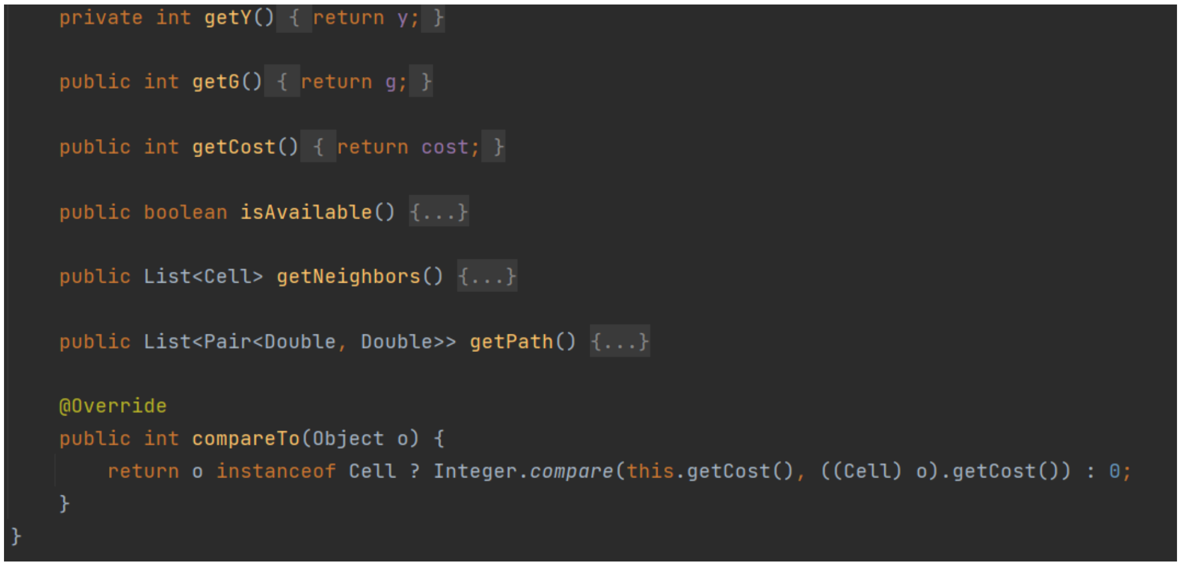
For this project, I split it into 2 parts. Since the A* algorithm needs us to operate “node”, I started with creating a “node” class. After that, I did the A* algorithm, it is pretty straightforward.

This is the “node” class I created for the A* search, called Cell(child class of AStar):



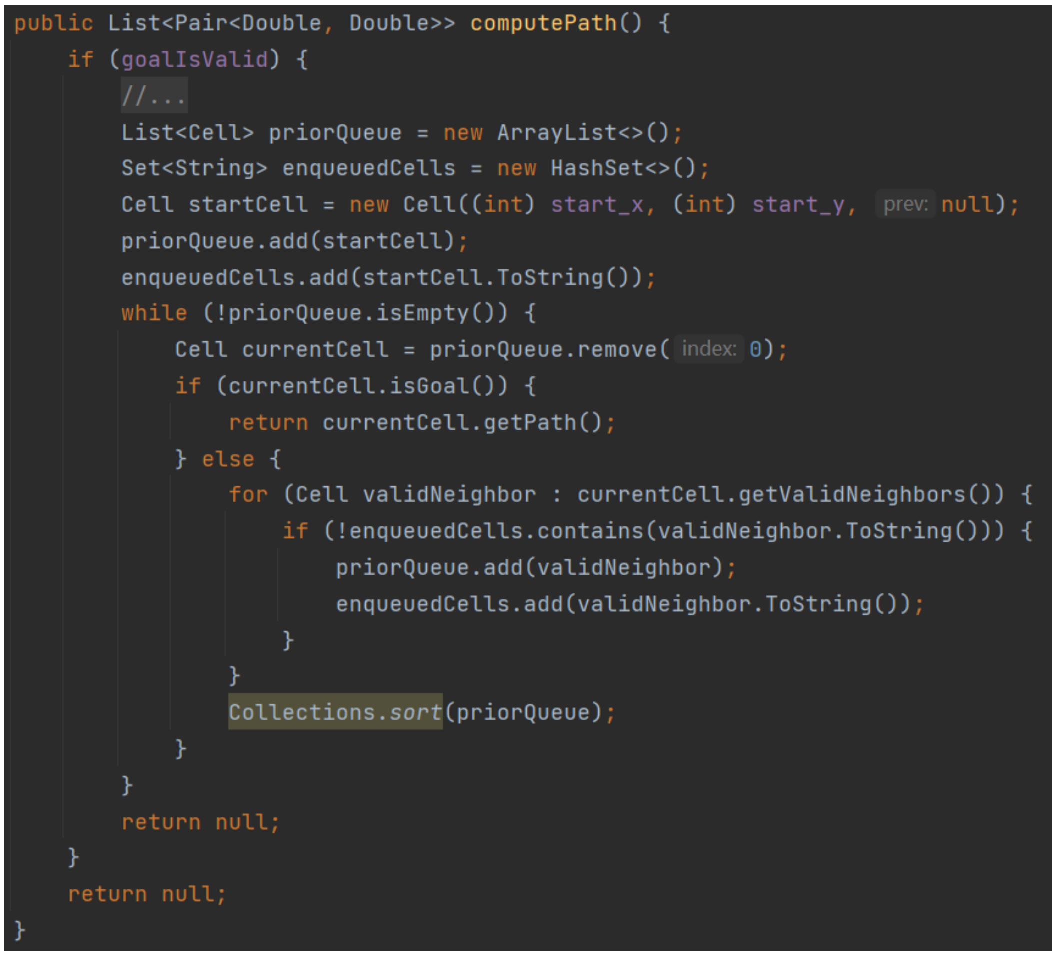
```java
private class Cell implements Comparable {
    final static int[][] MOVES = new int[][] { { -1, 0 }, { 1, 0 }, { 0, -1 }, { 0, 1 } };

    final Cell prev;
    final int x;
    final int y;
    final int g;
    final int h;
    final int cost;
    final String string;

    public Cell(int x, int y, Cell prev) {...}

    public String ToString() { return string; }

    public boolean isGoal() { return (x == goal_x) && (y == goal_y); }

    public Cell getPrev() { return prev; }

    private int getX() { return x; }
```



```java
    private int getY() { return y; }

    public int getG() { return g; }

    public int getCost() { return cost; }

    public boolean isAvailable() {...}

    public List<Cell> getNeighbors() {...}

    public List<Pair<Double, Double>> getPath() {...}

    @Override
    public int compareTo(Object o) {
        return o instanceof Cell ? Integer.compare(this.getCost(), ((Cell) o).getCost()) : 0;
    }
}
```

Basically, each Cell instance should correspond to a cell in the game map. When we want to know whether a cell is reachable in the game, we can call Cell.isAvailable. It will first check whether the x and y coordinates are valid for the game. A cell with a negative coordinate or position out of the map is invalid. Then, it will check whether there is a collision at that position. Also, since we need to put cells into the closed list, I created the ToString method to make the cells hashable. For each x, y coordinate, its corresponding string is unique. I made Cell as a subclass of Comparable and implemented the compareTo method so that we can easily sort the cells in the open list.

For the second part, I added some needed attributes initialization inside the constructor of AStar. Then I start to implement the algorithm body:



```
public List<Pair<Double, Double>> computePath() {
    if (goalIsValid) {
        //...
        List<Cell> priorQueue = new ArrayList<>();
        Set<String> enqueuedCells = new HashSet<>();
        Cell startCell = new Cell((int) start_x, (int) start_y, prev: null);
        priorQueue.add(startCell);
        enqueuedCells.add(startCell.ToString());
        while (!priorQueue.isEmpty()) {
            Cell currentCell = priorQueue.remove( index: 0);
            if (currentCell.isGoal()) {
                return currentCell.getPath();
            } else {
                for (Cell validNeighbor : currentCell.getValidNeighbors()) {
                    if (!enqueuedCells.contains(validNeighbor.ToString())) {
                        priorQueue.add(validNeighbor);
                        enqueuedCells.add(validNeighbor.ToString());
                    }
                }
                Collections.sort(priorQueue);
            }
        }
        return null;
    }
    return null;
}
```

It is pretty straightforward: It first checks whether the goal cell is reachable. Then, it initializes the open list(priorQueue) and the closed list(enqueuedCells). For each cell in the open list, check whether it is the goal. If not, expand it and sort the open list. Repeat the process until it found the goal or the open list became empty.

If the goal is found, Cell.getPath() will work as the backtracking method to return the path we expected.