2022 年 秋 季学期研究生课程考核
实验报告（2）

考　核　科　目: 算 法 设 计 与 分 析

学 生 所 在 院: 计 算 机 科 学 与 技 术 学 院

学 生 所 在 学 科: 电 子 信 息

学　生　姓　名: 兰 江 浩

学　　号: 2 2 S 1 3 0 4 8 7

学　生　类　别: 应 用 型

考　核　结　果:　　阅 卷 人

实验二 搜索算法

1. 实验目的

（1）掌握搜索算法的基本设计思想与方法。

（2）掌握 A*算法的设计思想与方法。

（3）熟练使用高级编程语言实现搜索算法。

（4）利用实验测试给出的搜索算法的正确性。

2. 实验学时

4 学时。

3. 实验问题

寻路问题。以图 2-1 为例，输入一个方格表示的地图，要求用 A*算法找到并输出从起点（在方格中标示字母 S）到终点（在方格中标示字母 T）的代价最小的路径。有如下条件及要求：

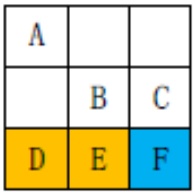
（1）每一步都落在方格中，而不是横竖线的交叉点。

（2）灰色格子表示障碍，无法通行。

（3）在每个格子处，若无障碍，下一步可以达到八个相邻的格子，并且只可以到达无障碍的相邻格子。其中，向上、下、左、右四个方向移动的代价为 1，向四个斜角方向移动的代价为$\sqrt{2}$ 。

（4）在一些特殊格子上行走要花费额外的地形代价。比如，黄色格子代表沙漠，经过它的代价为 4；蓝色格子代表溪流，经过它的代价为 2；白色格子为普通地形，经过它的代价为 0。

（5）经过一条路径总的代价为移动代价+地形代价。其中移动代价是路径上所做的所有移动代价的总和；地形代价为路径上除起点外所有格子的地形代价的总和。比如，在下图的示例中，路径 A→B→C 的代价为$\sqrt{2}+1$(移动)+0(地形)，而路径 D→E→F 的代价为 的代价为 2(移动) +6(地形)。



4. 实验步骤

4.1. 实现单向与双向的 A*搜索算法

以图 1 为样例，实现单向和双向 A*算法并测试输出。输出的路径应该与下图中以圆点标示的路径一致，或者略有不同但是有一样的代价。规则和条件如第 3 节所述。

双向 A*算法应当从 S 点和 T 点同时行 A*搜索。

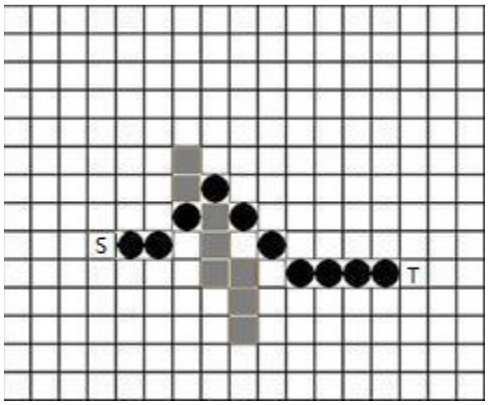


图 1

4. 1. 1. 单向 A*算法介绍

A*算法使用启发式的代价函数进行代价估计。每一次扩展，A*会优先从待扩展结点中选择预估其所在路径从起点到终点预估的总代价最小的结点前进。预估代价的计算公式为：

$$f(n) = g(n) + h(n)$$

其中，n为待扩展的结点，$g(n)$是从起点到n的实际代价，，$h(n)$是一个启发函数，用来预估从结点n到目标结点的代价。根据问题的不同，$g(n)$有不同的计算方式，例如，定义横向移动的代价为 1，斜向移动的代价为$\sqrt{2}$。另外，还可以规定地形代价，比如为平地移动和水面移动定义不同的代价。$h(n)$也可以有多种选择，比如使用欧氏距离。当成功到达终点或没有可扩展结点时，A*搜索终止。

要实现 A*搜索，需要两个辅助表：OPEN 表和 CLOSE 表。OPEN 表的作用是保存待扩展的结点，算法每次从 OPEN 表里选择$f(n)$最小的结点进行扩展，因此在具体实现中，应该使用优先队列来实现 OPEN 表。CLOSE 表的作用是记录已经扩展的结点，被放到 CLOSE 表中的结点不会被再次扩展，故所有结点只能被扩展一次。

A*算法描述如下：搜索开始前，CLOSE 表为空，OPEN 表只有起始结点，然后开始迭代。每次迭代，算法从 OPEN 中取出$f(n)$最小的结点作为当前结点，然后将当前结点的邻接结点按照一定规则放到 OPEN 表中。最后，将当前结点放到 CLOSE 表中，表示该结点已经扩展。如此往复。

将邻接节点放到 OPEN 表中的具体规则如下：（1）当邻接结点超出地图边界，不放入 OPEN 表；（2）当邻接结点是障碍物，不放入 OPEN 表；（3）当邻接结点在 CLOSE 表中，不放入 OPEN 表；（4）当邻接结点不在 OPEN 表中，加入 OPEN 表；（5）当邻接结点在 OPEN 表中，且 OPEN 表中该结点的g值>当前结点的g值+当前结点到该结点的代价，则更新 OPEN 表中的该结点，将其父节点修改为当前结点，修改其g值为新的当前代价。

在结点的具体的实现中，除了其坐标外，还需要保存其g值和h值，因为在迭代时需要和其它结点进行比较；另外，由于最后需要回溯路径，每个结点还需要保存其父节点的指针。起始结点无父结点。

算法伪代码如下：

输入：地图$M_{m\times n}$, 起始位置p_s, 目标位置p_t
输出：从p_s到p_t的路径
1. OPEN=[], CLOSE=[]
2. 添加$node(p_s, parent = null, g = 0, h = h(p_s))$到 OPEN
3. While OPEN 非空:
4. n←出队 OPEN 中$f = g + h$最小的结点
5. 添加 n 到 CLOSE
6. If n在位置p_t:
7. Then 结束
8. For $n_i \in \{n$的邻接结点$\}$:
9. If n_i不在$M_{m\times n}$上 or n_i是障碍物 or n_i在 CLOSE 中:
10. Then continue
11. If n_i不在 OPEN 中:
12. Then 添加n_i到 OPEN
13. If n_i在 OPEN 中 and $g(n_i)$>$g(n)$+从n到n_i的代价
14. Then 添加n_i到 OPEN

4.1.2. 单向 A*算法的 Java 实现

坐标和结点数据结构

坐标数据结构保存 x 值和 y 值，实现代码如下：

```
public class Coord {
    public int x;
    public int y;
```

```
//methods...
}
```

结点数据结构保存坐标、父节点坐标、g值和h值，实现代码如下：

```
public class Node implements Comparable<Node> {
    private Coord coord;
    private Coord parent;
    private double G;
    private double H;

    //methods...
}
```

$g(n)$和$h(n)$的定义

$g(n)$定义为移动代价和地形代价的和。其中，移动代价定义为，每次横向移动为1，斜向移动为$\sqrt{2}$；对于地形代价，定义地图上会有3种地形：平地、水面和沙漠，平地的地形代价为0，水面的地形代价为2，沙漠的地形代价为4。

$h(n)$定义为从当前结点到终点可能的最小移动代价，即：

$$h(n) = \sqrt{2} \times min(|x - x_t|, |y - y_t|) \\ +1 \times (max(|x - x_t|, |y - y_t|) - min(|x - x_t|, |y - y_t|))$$

通俗理解为，假设从当前结点到终点无障碍，则先尽可能走斜线，走到和终点在同一x轴或y轴上时，再走直线，使得到终点的移动代价最小。$h(n)$不考虑地形代价，因为理论上来说从当前结点到终点之间的地形在探索之前是不可见的。

OPEN表和CLOSE表实现

OPEN表实现。OPEN表需要具备的功能有：添加新结点、出队结点、删除结点、获取某个坐标处的结点、判断某个坐标处是否存在结点、判断表是否为空。由于每次迭代需要从OPEN表中取出$f(n)$最小的结点，因此为了算法的效率，使用优先队列来实现。另外，在扩展当前结点时需要判断邻接结点是否已经在OPEN表中，在优先队列中查找的复杂度为$O(n)$。为了使查找的效率提高到$O(1)$，另外使用一个**结点图**——和地图一样大小的二维数组来保存对应坐标处的结点信息。当添加或出队新结点的时候，除了在优先队列中插入或出队之外，还要在结点图上相应地更新标记。OPEN表的实现代码如下：

```
class OpenList{
    Queue<Node> openQueue; // 优先队列(升序)
    Node[][] openMap;

    public OpenList(int height, int width) {
        this.openQueue = new PriorityQueue<>();
        this.openMap = new Node[height][width];
    }
```

```
    void add(Node node) {
        if (this.contains(node.getCoord())){
            System.out.println("[OpenList.add()] Node in this coord
is already in openList!");
            return;
        }
        openQueue.add(node);
        this.openMap[node.getCoord().y][node.getCoord().x] = node;
    }

    Node poll(){
        cleanQueueHead();

        Node node = openQueue.poll();
        assert(openMap[node.getCoord().y][node.getCoord().x] !=
null);
        openMap[node.getCoord().y][node.getCoord().x] = null;
        return node;
    }

    Node peek(){
        cleanQueueHead();
        return openQueue.peek();
    }

    Node get(Coord coord){
        return openMap[coord.y][coord.x];
    }

    boolean contains(Coord coord){
        return this.openMap[coord.y][coord.x] != null;
    }

    void remove(Coord coord){
        //在 openQueue 上 remove 的复杂度为 O(n)，因此仅在 openMap 上作标
记，poll()或 isEmpty()时会在 openMap 上检查是否存在
        assert(openMap[coord.y][coord.x] != null);
        openMap[coord.y][coord.x] = null;
    }

    boolean isEmpty(){
        //除掉队列头实际已经被删除的元素
        cleanQueueHead();
        return this.openQueue.isEmpty();
    }

    void cleanQueueHead(){
        //除掉队列头实际已经被删除的元素
        while(!openQueue.isEmpty()
&& !contains(openQueue.peek().getCoord())){
```

```
            openQueue.remove();
        }
    }
}
```

CLOSE 表实现。CLOSE 表需要具备的功能有：添加新结点、获取某个坐标处的结点、得到结点的父节点。CLOSE 表用一个结点图实现，和 OPEN 表的结点图同理，它是一个和地图同样大小的二维数组。CLOSE 表的实现代码如下：

```
class CloseList{
    Node[][] closeMap;

    CloseList(int height, int width){
        closeMap = new Node[height][width];
    }

    boolean contains(Coord coord){
        return closeMap[coord.y][coord.x] != null;
    }

    void add(Node node){
        Coord coord = node.getCoord();
        closeMap[coord.y][coord.x] = node;
    }

    Node get(Coord coord){
        return closeMap[coord.y][coord.x];
    }

    Node getParent(Node node) {
        Coord parentCoord = node.getParent();
        return closeMap[parentCoord.y][parentCoord.x];
    }
}
```

单向 A*算法实现

单向 A*算法的代码实现如下：

```
public class AStarImpl implements AStar {

    OpenList openList;
    CloseList closeList;

    public AStarSearchResult search(MapInfo mapInfo) {
        if (mapInfo == null) return null;

        createOpenCloseList(mapInfo.getHeight(), mapInfo.getWidth());
        openList.add(new Node(mapInfo.getStartCoord(), null, 0.0,
getH(mapInfo.getStartCoord(), mapInfo.getEndCoord())));
```

```
        AStarSearchResult aStarSearchResult = new
AStarSearchResult();

        while(!openList.isEmpty()){
            AStarSingleStep stepInfo = new AStarSingleStep();
            boolean success = step(mapInfo, stepInfo);
            aStarSearchResult.addStep(stepInfo);

            if(success){
                aStarSearchResult.isSuccess = true;
                aStarSearchResult.finalCost =
getFinalCost(closeList.get(mapInfo.getEndCoord()));
                aStarSearchResult.setPath(getPath(mapInfo));
                return aStarSearchResult;
            }
        }
        //未找到终点
        return aStarSearchResult;
    }

    /**
     * 执行一步搜索
     * @param mapInfo mapInfo
     * @param stepInfo 将该步信息记录在此对象上
     * @return 是否成功
     */
    boolean step(MapInfo mapInfo, AStarSingleStep stepInfo){
        Node current = openList.poll();
        closeList.add(current);
        stepInfo.setCurrentNode(current);
        if(current.getCoord().equals(mapInfo.getEndCoord())){
            //成功找到终点
            stepInfo.setNeighbors(new ArrayList<Node>(0));
            return true;
        }
        List<Node> neighborNodes = extendNeighbors(mapInfo, current);
        stepInfo.setNeighbors(neighborNodes);
        stepInfo.setNext(openList.peek());
        if(openList.peek().getCoord().equals(current.getCoord())){
            System.out.println("Error!");
        }
        return false;
    }

    static class NeighborInfo {
        Coord coord;
        double moveCost;
        NeighborInfo(Coord coord, double cost) { this.coord = coord;
this.moveCost = cost; }
    }
```

```
    List<Node> extendNeighbors(MapInfo mapInfo, Node current) {
        List<Node> extended = new ArrayList<>();
        for(NeighborInfo neighborInfo:
getNeighborInfos(current.getCoord())) {
            Coord neighbor = neighborInfo.coord;
            if (!isCoordOnMap(mapInfo, neighbor)) continue;
            if (mapInfo.isBarrier(neighbor)) continue;
            if (closeList.contains(neighbor)) continue;
            //G=父节点 G+移动代价+地形代价
            double G = current.getG() + neighborInfo.moveCost +
mapInfo.getTerrainCost(neighbor);
            Node node = extendNeighbor(mapInfo, current,
neighborInfo.coord, G);
            if(node != null) extended.add(node);
        }
        return extended;
    }

    private ArrayList<NeighborInfo> getNeighborInfos(Coord coord){
        ArrayList<NeighborInfo> neighbors = new ArrayList<>(8);

        neighbors.add(new NeighborInfo(new Coord(coord.x - 1, coord.y
- 1), OBLIQUE_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x - 1, coord.y
+ 1), OBLIQUE_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x + 1, coord.y
- 1), OBLIQUE_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x + 1, coord.y
+ 1), OBLIQUE_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x - 1,
coord.y), DIRECT_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x + 1,
coord.y), DIRECT_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x, coord.y -
1), DIRECT_MOVE_COST));
        neighbors.add(new NeighborInfo(new Coord(coord.x, coord.y +
1), DIRECT_MOVE_COST));
        return neighbors;
    }

    private Node extendNeighbor(MapInfo mapInfo, Node current, Coord
neighbor, double g) {
        if (openList.contains(neighbor)){
            Node neighborNodeInOpen = openList.get(neighbor);
            if(g < neighborNodeInOpen.getG()){
                openList.remove(neighbor);
                Node node = new Node(neighbor, current.getCoord(), g,
getH(neighbor, mapInfo.getEndCoord()));
                openList.add(node);
```

```
                return node;
            }
            return null;
        }
        else {
            Node node = new Node(neighbor, current.getCoord(), g,
getH(neighbor, mapInfo.getEndCoord()));
            openList.add(node);
            return node;
        }
    }

    static double getH (Coord coord1, Coord coord2){
        int xDiff = Math.abs(coord1.x - coord2.x);
        int yDiff = Math.abs(coord1.y - coord2.y);
        return OBLIQUE_MOVE_COST * Math.min(xDiff, yDiff) +
                DIRECT_MOVE_COST * (Math.max(xDiff, yDiff) -
Math.min(xDiff, yDiff));
    }

    private boolean isCoordOnMap(MapInfo mapInfo, Coord coord){
        return coord.x >= 0 && coord.x < mapInfo.getWidth()
                && coord.y >= 0 && coord.y < mapInfo.getHeight();
    }

    private List<Node> getPath(MapInfo mapInfo){
        List<Node> path = new ArrayList<>();

        Coord coord = mapInfo.getEndCoord();
        Node node = closeList.get(coord);
        path.add(node);
        while (!node.getCoord().equals(mapInfo.getStartCoord())){
            node = closeList.getParent(node);
            path.add(node);
        }
        Collections.reverse(path);
        return path;
    }

    private static double getFinalCost(Node endNode){
        return endNode.getG();
    }

    void createOpenCloseList(int height, int width){
        openList = new OpenList(height, width);
        closeList = new CloseList(height, width);
    }

    @Override
    public AStarType getCode() {
```

```
        return AStarType.UNIDIRECTIONAL;
    }
}
```

4.1.3. 双向 A*算法实现

思路

双向 A 算法就是使用两个单向 A 算法，从起点和终点同时往对方方向进行搜索，即起点以终点为目标、终点以起点为目标。在单向 A*的基础上，需要作以下修改：

- 使用两个单向 A*算法，从起点和终点同时往对方方向进行搜索
- 终止条件改为两个 A*的 OPEN 列表有一个为空（说明起点和终点不可达），或某一方的当前结点位置正好是对方的 CLOSE 结点（说明找到了对方的“脚印”，沿着脚印就可以到达终点）。

终止条件

网上有很多介绍双向 A 搜索的文章，但关于双向 A 搜索成功找到路径的条件，网上的说法大多数都比较模糊。有些谈到细节的博主认为成功终止条件是两个 OPEN 表重叠（某一方要放入 OPEN 表的邻接结点正好也是另一方的 OPEN 结点）。但我个人认为，终止条件应该设置为两个 CLOSE 表重叠更为合理。因为 OPEN 表只是表示搜索下一步可以扩展的结点，但还没有扩展，结点的情况还未确定，有可能之后被更新；而 CLOSE 表表示搜索已经走过的结点，结点的情况已经确定。在具体实现中，我采用 CLOSE 表重叠的方案，采用另一种方式也可以，具体的结果应该差别不大。

实现

双向 A*搜索的代码实现如下：

```
public class BidirectionalAStarImpl implements AStar {
    private AStarImpl forwardAStar;
    private AStarImpl backwardAStar;

    public AStarSearchResult search(MapInfo mapInfo){
        if (mapInfo == null) return null;

        forwardAStar = new AStarImpl();
        backwardAStar = new AStarImpl();
        forwardAStar.createOpenCloseList(mapInfo.getHeight(),
mapInfo.getWidth());
        backwardAStar.createOpenCloseList(mapInfo.getHeight(),
mapInfo.getWidth());
        forwardAStar.openList.add(new Node(mapInfo.getStartCoord(),
                null, 0.0, AStarImpl.getH(mapInfo.getStartCoord(),
mapInfo.getEndCoord())));
        backwardAStar.openList.add(new Node(mapInfo.getEndCoord(),
```

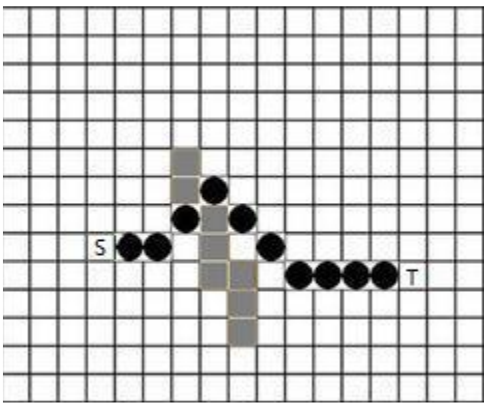
```
                null, 0.0, AStarImpl.getH(mapInfo.getEndCoord(),
mapInfo.getStartCoord())));
        MapInfo backwardMapInfo = getBackwardMapInfo(mapInfo);

        AStarSearchResult aStarSearchResult = new
AStarSearchResult(true);

        while(!forwardAStar.openList.isEmpty()
&& !forwardAStar.openList.isEmpty()){
            AStarSingleStep[] stepInfo = new AStarSingleStep[]{new
AStarSingleStep(), new AStarSingleStep()};
            Coord meetCoord = step(forwardAStar, backwardAStar,
mapInfo, stepInfo[0]);
            if(meetCoord != null){
                aStarSearchResult.addBidirectionalStep(stepInfo);
                aStarSearchResult.isSuccess = true;
                aStarSearchResult.finalCost =
getFinalCost(meetCoord);

aStarSearchResult.setBidirectionalPath(getPath(mapInfo, meetCoord,
forwardAStar));
                return aStarSearchResult;
            }

            meetCoord = step(backwardAStar, forwardAStar,
backwardMapInfo, stepInfo[1]);
            aStarSearchResult.addBidirectionalStep(stepInfo);
            if(meetCoord != null){
                aStarSearchResult.isSuccess = true;
                aStarSearchResult.finalCost =
getFinalCost(meetCoord);

aStarSearchResult.setBidirectionalPath(getPath(backwardMapInfo,
meetCoord, backwardAStar));
                return aStarSearchResult;
            }
        }
        //未找到终点
        return aStarSearchResult;
    }

    /**
     * 执行一步搜索
     * @param currentAStar 当前 AStar
     * @param anotherAStar 对面 AStar
     * @param mapInfo
     * @param stepInfo 将该步信息记录在此对象上
     * @return 成功找到解则返回相遇坐标，否则返回 null
     */
```

```
    private Coord step(AStarImpl currentAStar, AStarImpl
anotherAStar, MapInfo mapInfo, AStarSingleStep stepInfo){
        Node current = currentAStar.openList.poll();
        currentAStar.closeList.add(current);
        stepInfo.setCurrentNode(current);

        //如果当前结点在对面的 close 表中，成功找到解
        if(anotherAStar.closeList.contains(current.getCoord())){
            stepInfo.setNeighbors(new ArrayList<Node>(0));
            return current.getCoord();
        }

        List<Node> neighborNodes =
currentAStar.extendNeighbors(mapInfo, current);
        stepInfo.setNeighbors(neighborNodes);
        stepInfo.setNext(currentAStar.openList.peek());
        return null;
    }

    private List<List<Node>> getPath(MapInfo mapInfo, Coord
meetCoord, AStarImpl whoMetTheOtherFirst){
        List<Node> pathFromStart = new ArrayList<>();
        List<Node> pathFromEnd = new ArrayList<>();

        Node node = forwardAStar.closeList.get(meetCoord);
        if(whoMetTheOtherFirst == forwardAStar)
pathFromStart.add(node);
        while (!node.getCoord().equals(mapInfo.getStartCoord())){
            node = forwardAStar.closeList.getParent(node);
            pathFromStart.add(node);
        }

        node = backwardAStar.closeList.get(meetCoord);
        if(whoMetTheOtherFirst == backwardAStar)
pathFromStart.add(node);
        pathFromEnd.add(node);
        while (!node.getCoord().equals(mapInfo.getEndCoord())){
            node = backwardAStar.closeList.getParent(node);
            pathFromEnd.add(node);
        }
        Collections.reverse(pathFromStart);

        List<List<Node>> path = new ArrayList<>(2);
        path.add(pathFromStart);
        path.add(pathFromEnd);
        return path;
    }

    private static MapInfo getBackwardMapInfo(MapInfo mapInfo){
```

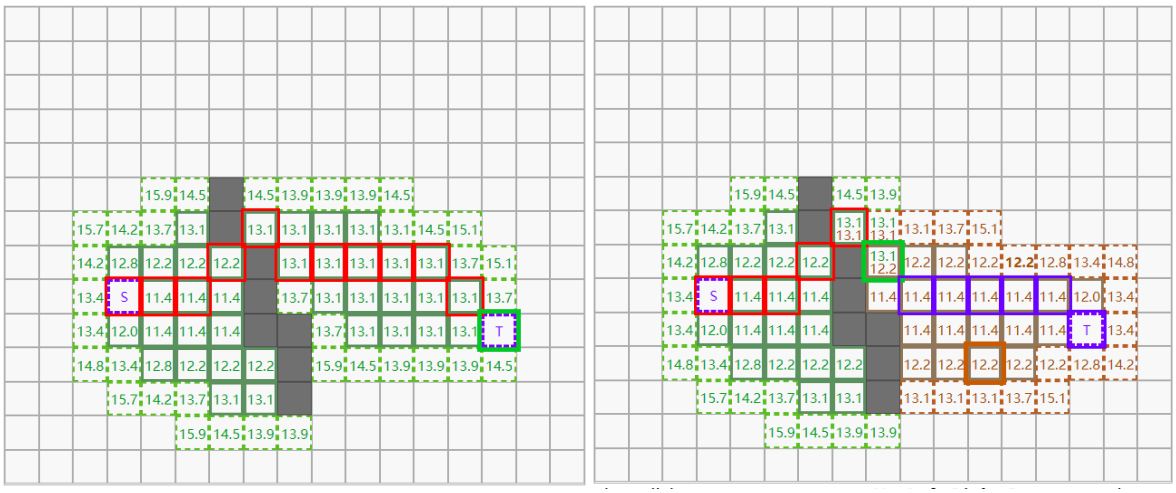
```
        return new MapInfo(mapInfo.getLabels(), mapInfo.getWidth(),
mapInfo.getHeight(),
                mapInfo.getEndCoord(), mapInfo.getStartCoord(),
mapInfo.getLabelsInfo());
    }

    @Override
    public AStarType getCode() {
        return AStarType.BIDIRECTIONAL;
    }

    private double getFinalCost(Coord meetCoord) {
        Node forwardNode = forwardAStar.closeList.get(meetCoord);
        Node backwardNode = backwardAStar.closeList.get(meetCoord);
        return forwardNode.getG() + backwardNode.getG();
    }
}
```

4.1.4. 输出测试

测试地图如下：



单向 A 算法和双向 A 算法的输出分别如下：



4.2. A*算法可视化

精灵王子Haposola 一大早从城堡中起床，要去沙漠中河流消失的地方寻找失落的宝藏。长路漫漫，处处险恶，精灵王子要尽快到达目标地点。

以图2为输入，采用单向和双向的A*算法，寻找地图上给出的起点和终点间的最小代价路径。规则和条件如第3节所述。

以适当的方式将地图可视化呈现，应该体现出方格、起点终点、障碍、地形等地图要素，并将算法计算出的最小代价路径在地图上体现出来。可以不完全遵守样例的形式（如颜色和以圆点表示的路径等），但以上要素应该完整地体现出来。

对于双向 A*算法，应当展示出双向搜索的路径（如从 S 点的搜索结果和从 T 点的搜索结果以不同的颜色表示）。

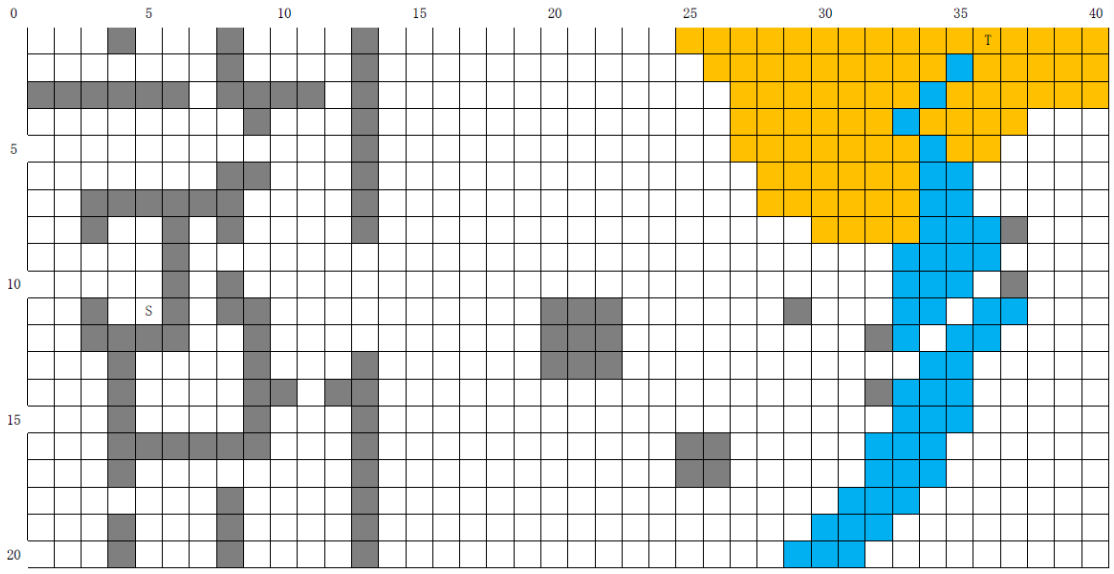


图 2

4.2.1. 可视化实现目标

本次实验计划实现的 A*可视化系统，除了能展示地图和最终路径，还要能以可交互动画形式展示搜索的整个过程，是本次实验的难点。具体地，可视化实现的目标为：

- 展示地图，能以不同颜色区分各种地形信息
- 显示最终路径
- 展示搜索过程，显示搜索中每次迭代的状态
- 动画可交互，用户能随意操作前进和后退

4.2.2. 系统概览

使用 HTML 页面来实现可视化。整个项目采用前后端分离的形式，分为前端和后端两个项目。前端通过接口往后端发送地图数据，请求结果；后端计算得到 A*搜索的结果和单步信息，一次性返回给前端；前端收到数据后以可交互动画的形式展示给用户。

前端采用的技术包括：

- HTML 展示静态界面
- CSS 美化界面
- Vue 实现交互式可视化
- Node.js 部署

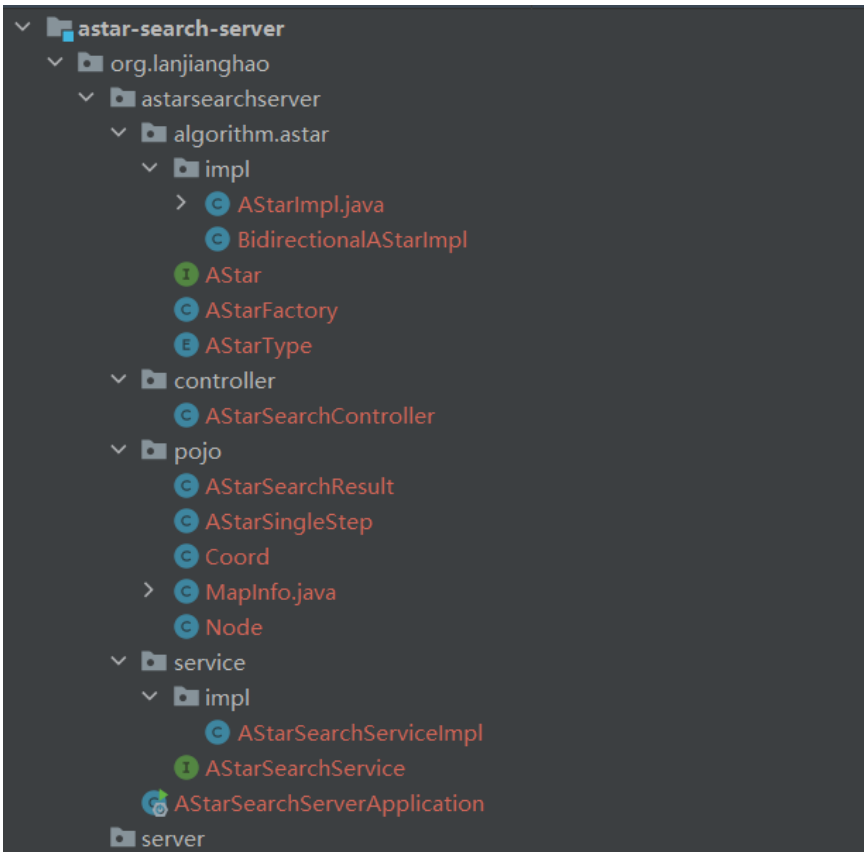
后端采用的技术包括：

- Maven 包管理
- Spring MVC 开发 Web 项目
- Spring Boot 简化配置和部署

4.2.3. 后端架构及实现

代码组织

代码目录组织如下：



服务流程

前端发送请求到服务器，服务器将请求交给 AStarSearchController 处理，AStarSearchController 使用 AStarSearchService 来提供服务，AStarSearchService 又通过 AStar 接口调用具体的 A*算法。AStarSearchController 得到结果后，最后以 Json 形式返回给前端。

结果数据

结果数据结构中包含以下信息：

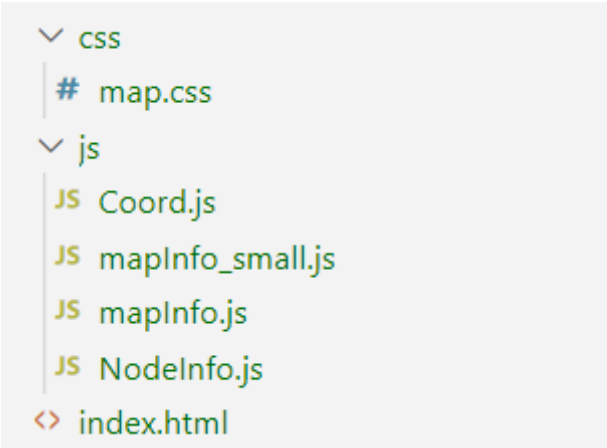
```
public class AStarSearchResult {
    //是否为双向搜索
    public boolean bidirectional;

    //是否找到路径
    public boolean isSuccess;

    //最终代价
    public double finalCost;

    //最终路径
    public List<List<Node>> path;

    //单步信息
    public List<AStarSingleStep[]> steps;
}
```

除了包括是否成功、最终代价、最终路径等信息外，还包括单步信息。单步信息是一个列表，列表中每一项保存着一步迭代的当前结点、扩展的邻接结点和下一次迭代要扩展的结点，这使得前端的交互式动画成为可能。

4.2.4. 前端架构及实现

代码组织

代码目录如下：



index 为前端界面，包括了 HTML 静态界面和 vue 交互代码；css 文件夹下存放 css 样式表，为静态界面提供美化；js 文件夹下定义了一些常用的类和地图信息。由于自己不擅长写前端，代码组织可能欠佳。

可视化界面实现

用一个 HTML 表格来显示地图：

```
<table id="map" border="1">
    <tr v-for="(row, y) in mapInfo.labels">
        <td v-for="(label, x) in row" :class="getCoordClasses(x, y,
label)">
            <!-- {{getContent(x, y)}} -->
            <span v-if="isStart(x, y)" class="start-flag">S</span>
            <span v-else-if="isEnd(x, y)" class="end-flag">T</span>
            <span v-else>
                <div class="forward-f" v-if="isInForwardClose(x, y)
|| isInForwardOpen(x, y)">
                    {{getForwardF(x, y).toFixed(1)}}
                </div>
                <div class="backward-f" v-if="isInBackwardClose(x, y)
|| isInBackwardOpen(x, y)">
                    {{this.getBackwardF(x, y).toFixed(1)}}
                </div>
            </span>
        </td>
    </tr>
</table>
```

表格大量使用 Vue 的强大特性进行渲染，使得最终的界面有很强的逻辑表现力。

在地图下方，显示结果和一些操作按钮：

```
<div>
    Result status:
    <span v-if="result === null">(not start yet)</span>
    <span v-else-if="result.isSuccess === false"><b>fail</b></span>
    <span v-else><b>success</b>, final cost is
<span>{{ getFinalCost().toFixed(2) }}</span></span>
</div>
<div>Current step: <b>{{ currentStepIdx }}</b></div>

<!-- <br/> -->
<button @click="initialize" :disabled="result ===
null">Initialize</button>
<button @click="getResult">Search</button>
<br />
<button
@click="toBegining" :disabled="!hasPrevStep()">&lt;&lt;</button>
<button
@click="decrementStep" :disabled="!hasPrevStep()">&lt;</button>
<button
@click="incrementStep" :disabled="!hasNextStep()">&gt;</button>
<button @click="toEnd" :disabled="!hasNextStep()">&gt;&gt;</button>
```

结点的不同状态要显示为不同的样式。例如，不同的地形有不同的背景色、当前结点会显示一个外框线、OPEN 结点会显示虚线边框等。实现方式为，为多种状态定义不同的 class 属性，并为每种 class 属性设置不同的样式。之后，通过 Vue 动态地改变结点的 class 属性，就可以修改其样式。比如，为不同的地形设置不同的背景色：

```
#map td.ground{
    background-color: #F8F8F8;
}

#map td.barrier{
    background-color: #707070;
    color: white;
}

#map td.river{
    background-color: rgba(47, 151, 255, 0.547);
    color: white;
}

#map td.desert{
    background-color: #F0E68C;
}
```

可视化逻辑实现

初始地图显示。初始地图用一张二维数组来表示，数组中每一个元素是一个整数标签，用来表示不同的地形。地图数据结构还包括起点和终点，分别用一个坐标来表示。初始地图数据结构代码如下：

```
var mapInfo = {
    labels: labels,    //an 2d array
    width: 40,
    height: 20,
    startCoord: new Coord(4, 10),
    endCoord: new Coord(35, 0),
    labelInfos:
        {
            1: new LabelInfo('ground', false, 0),
            0: new LabelInfo('barrier', true, 0),
            2: new LabelInfo('river', false, 2),
            3: new LabelInfo('desert', false, 4),
        }
}
```

获取结果数据。用 axios 向后端发送异步请求，获取结果数据。发送请求、获取结果的代码如下：

```
getResult() {
    if (this.result !== null) this.initialize()
```

```
    let that = this
    axios({
        method: 'post',
        url: '<http://localhost:8080/astar/unidirectional/search>',
        // url: '<http://localhost:8080/astar/bidirectional/search>',
        data: mapInfo
    }).then(function (response) {
        that.result = response.data
        that.initNodeInfoMap(that.mapInfo.height, that.mapInfo.width)
        that.renderCurrent(0)
        that.setPathNodes()
    })
},
```

可交互式动画。获取结果后，前端可以展示算法每一次迭代的状态，由用户任意操作前进和后退。为实现这个功能，先在 Vue 中声明一个整数类型的响应式状态 currentStepIdx，代表当前步数。当用户点击前进按钮或按下键盘 Right 键时，currentStepIdx 加 1；当用户点击后退按钮或按下键盘 Left 键时，currentStepIdx 减 1：

```
data() {
    return {
        currentStepIdx: 0,
        //...
    }
},
methods: {
    incrementStep() {
        if (this.hasNextStep()) this.currentStepIdx++
    },
    decrementStep() {
        if (this.hasPrevStep()) this.currentStepIdx--
    },
    toBegining() {
        this.currentStepIdx = 0
    },
    toEnd() {
        if (this.result !== null) {
            this.currentStepIdx = this.result.steps.length - 1
        }
        else {
            this.currentStepIdx = 0
        }
    },

    //...
}
```

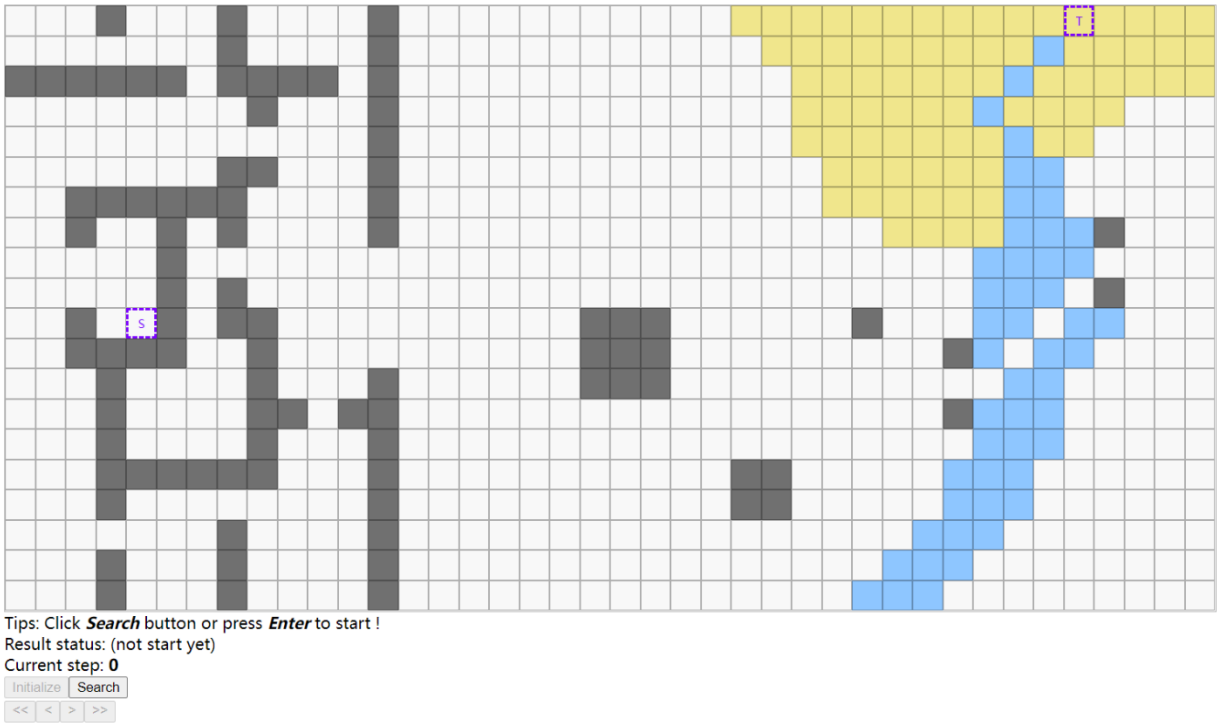
实现了步数 currentStepIdx 的更新后，再通过一个监听器来监听 currentStepIdx 的状态变化。当 currentStepIdx 变化时，更新地图状态：

```
watch: {
    // 每当 currentStepIdx 改变时，这个函数就会执行
    currentStepIdx(newIdx, oldIdx) {
        if (this.result === null) return
        if (newIdx != 0) {
            this.renderStepUpdate(newIdx, oldIdx)
        }
        else {
            this.initNodeInfoMap(this.mapInfo.height,
this.mapInfo.width)
            this.renderCurrent(0)
            this.setPathNodes()
        }
        if (newIdx === this.result.steps.length - 1 &&
this.result.isSuccess) {
            this.showPath = true
        }
        else {
            this.showPath = false
        }
    }
},
```

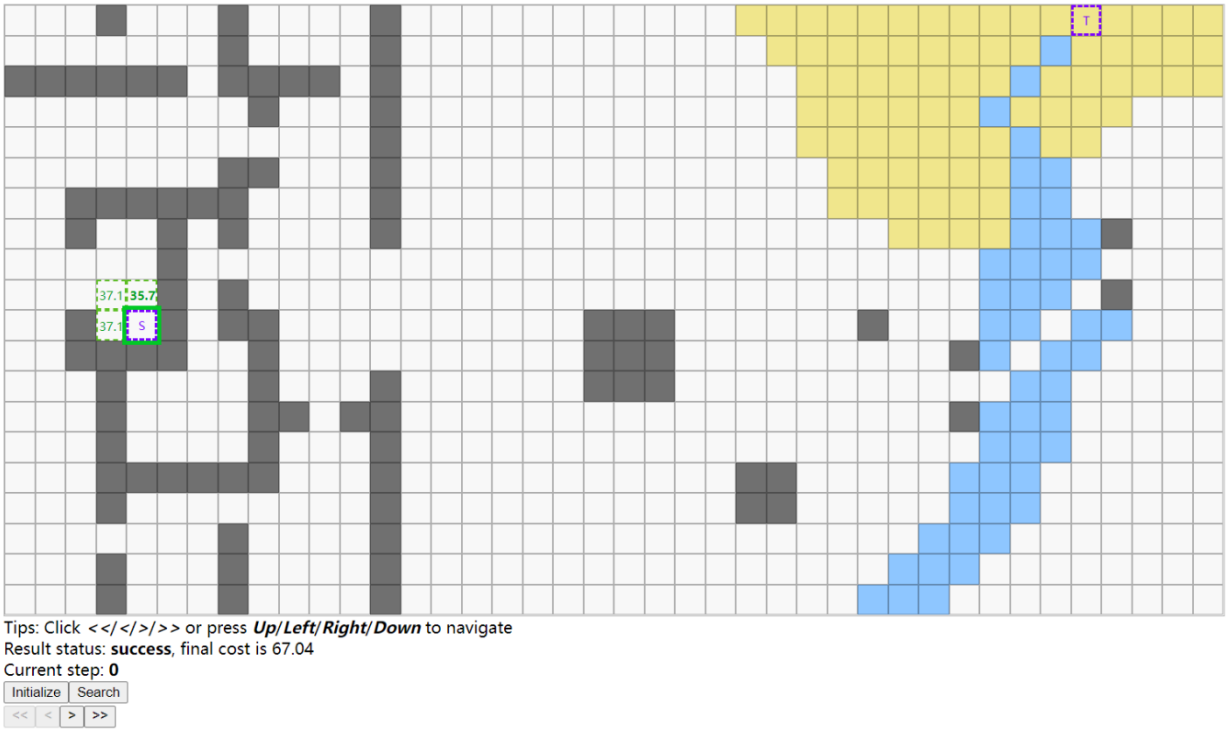
关于前端和后端的更多实现细节请看代码，正文中将不再介绍。

4.2.5. 可视化结果

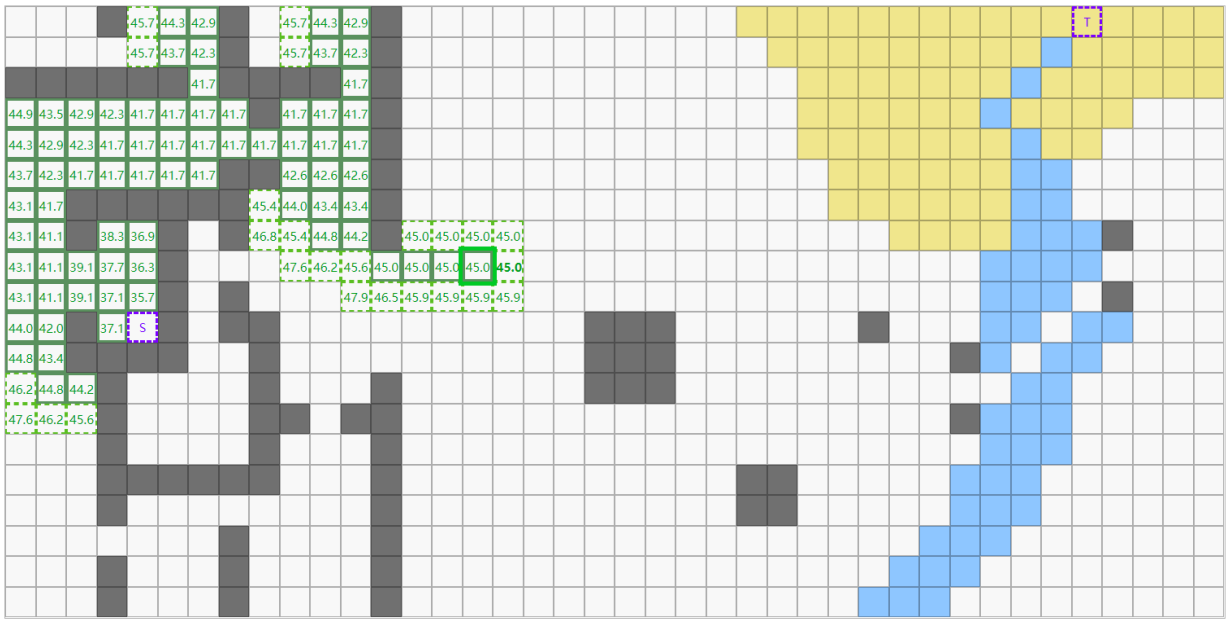
初始界面：



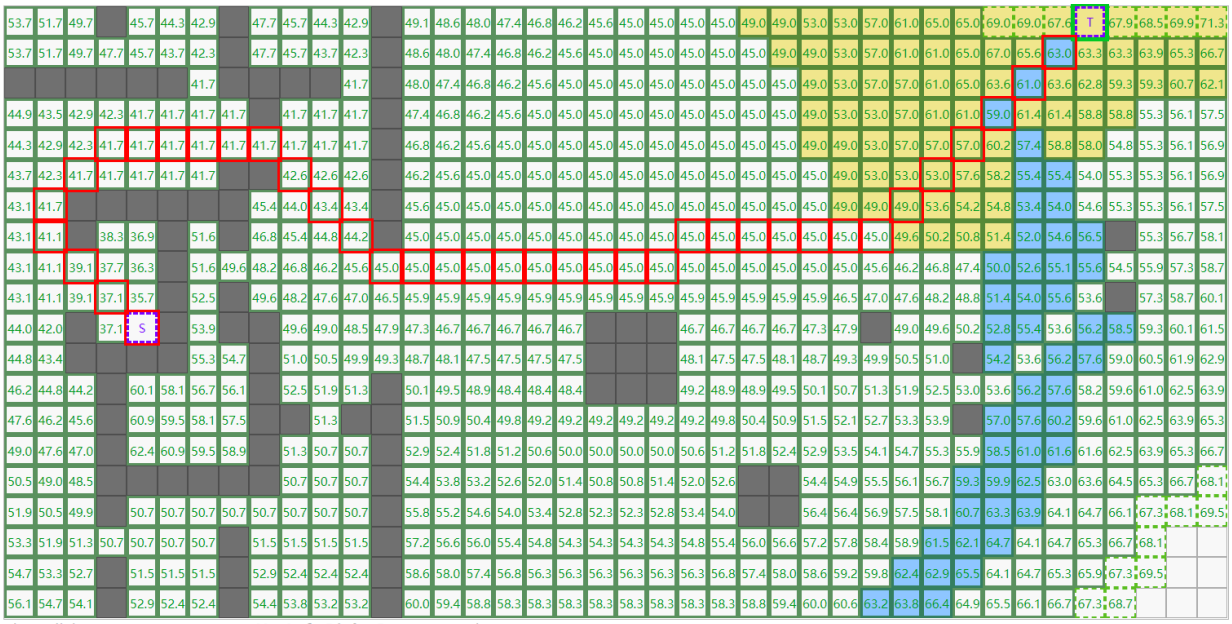
点击 Search 按钮或按下键盘 Enter 键得到结果：



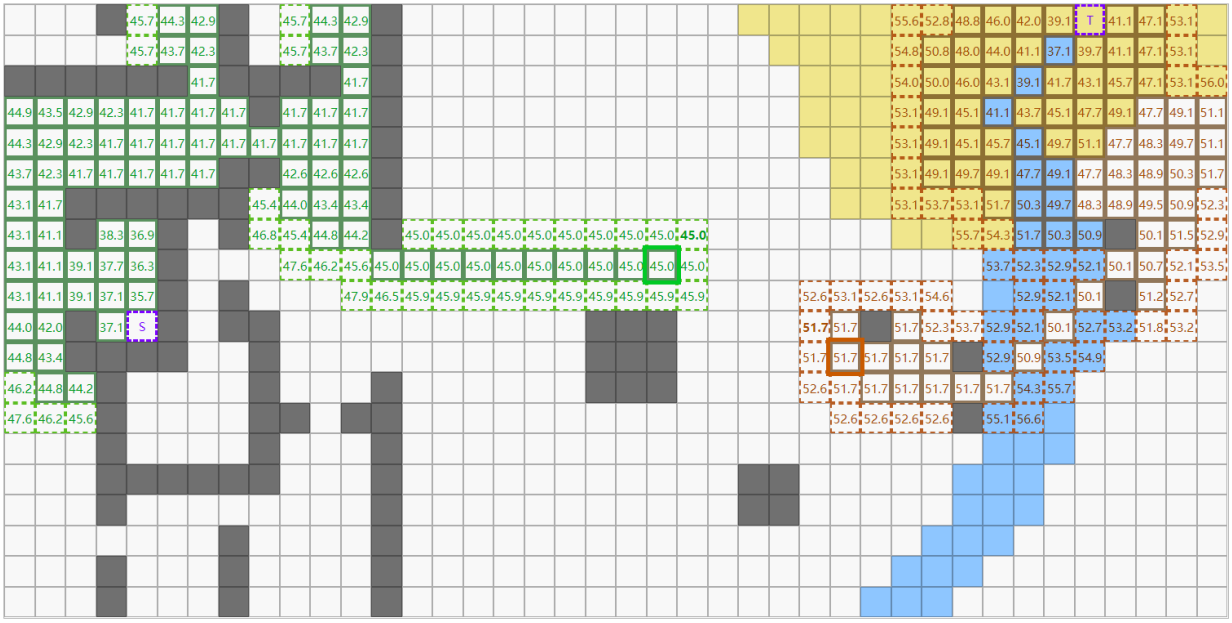
点击下方的四个按钮或键盘的四个方向键，可以调整前进或后退步骤，实现了步骤级别的精细的可视化：



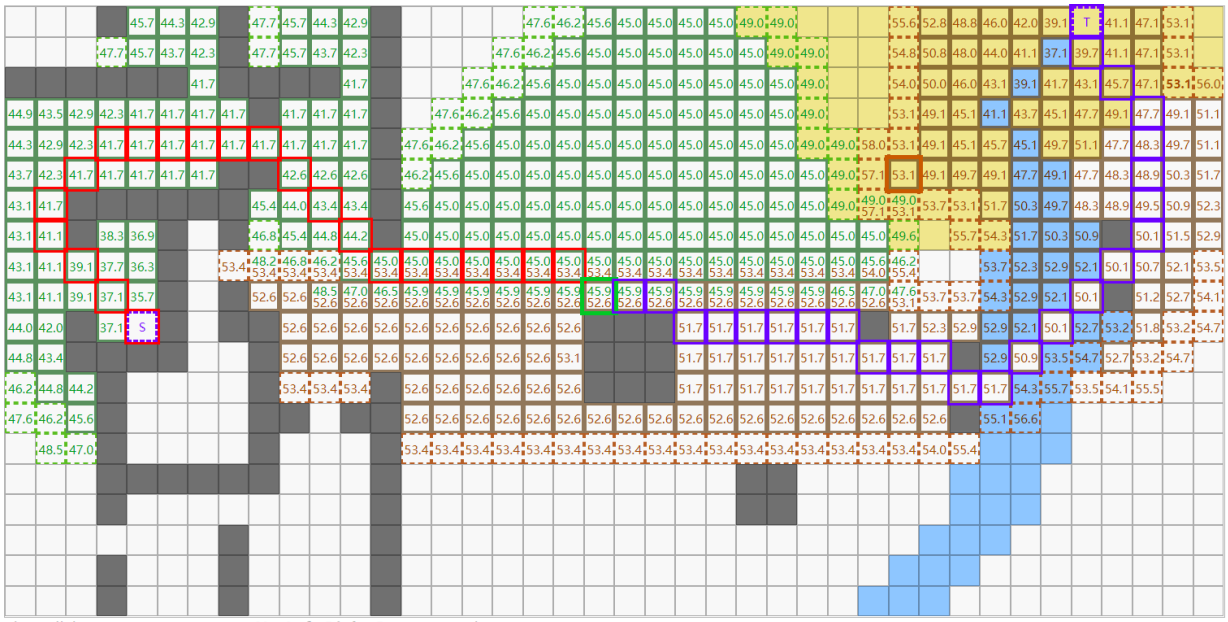
到达最后一步时，会显示最终路径：



双向 A*搜索的单步可视化：



双向 A*搜索的最终路径：

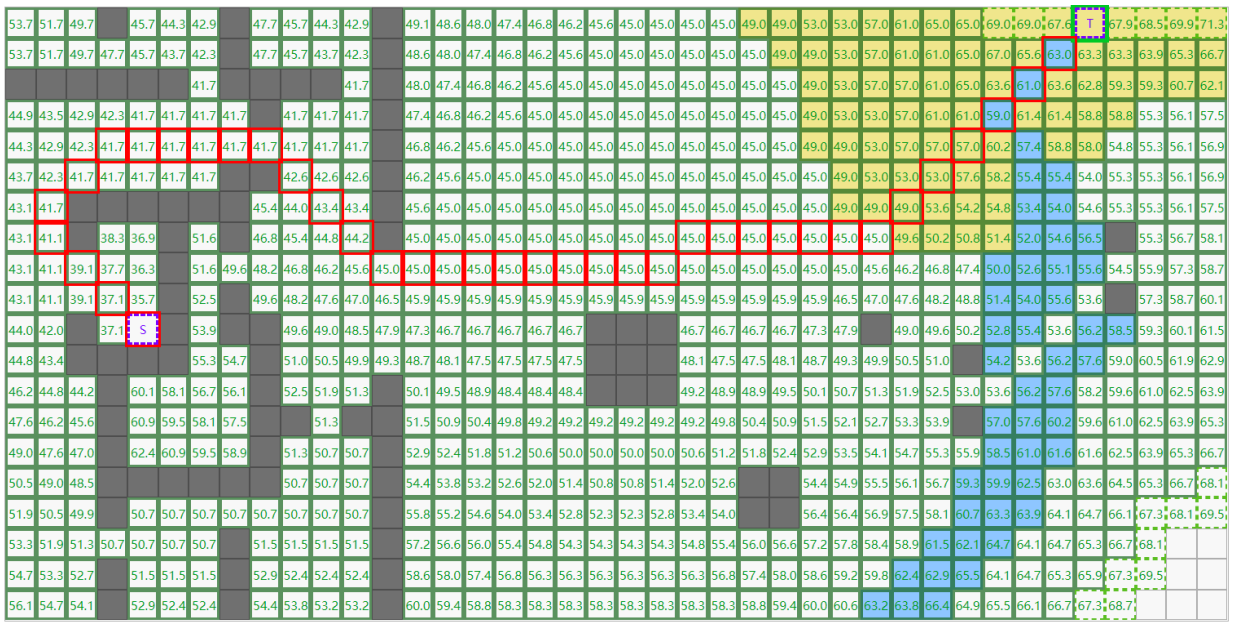


Tips: Click <</</>/>> or press ***Up*/*Left*/*Right*/*Down*** to navigate
Result status: **success**, final cost is 63.28
Current step: **182**
Initialize | Search
<< | < | > | >>

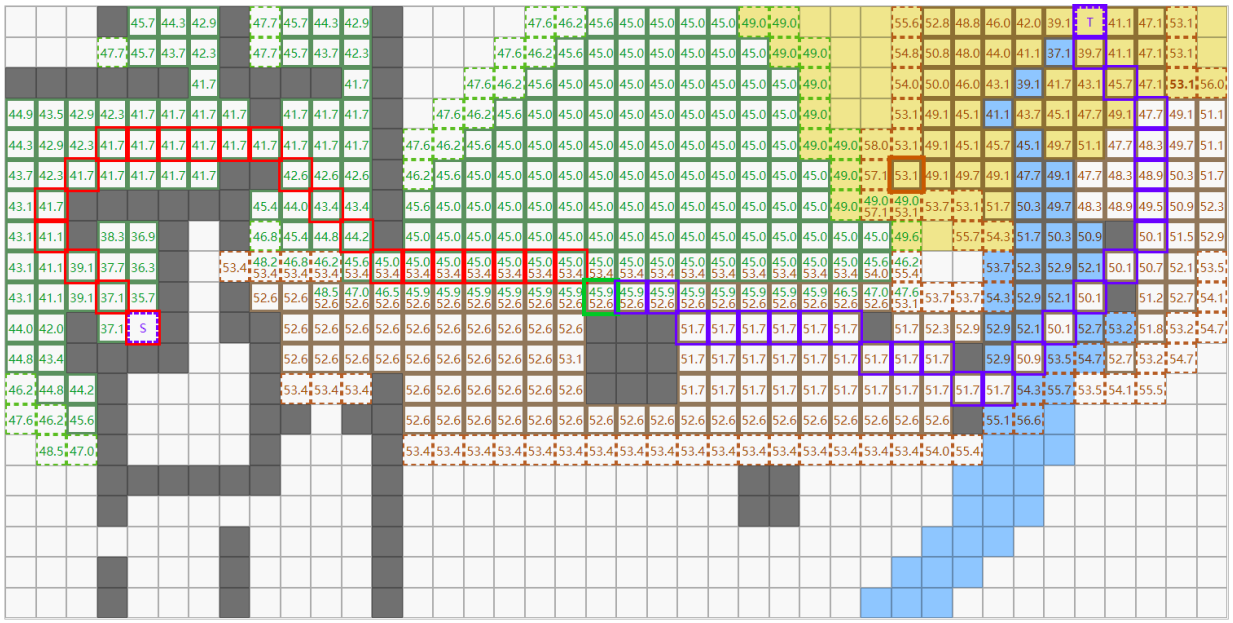
4.3. 结果与分析

4. 3. 1. 结果

单向 A*搜索最终代价为 67.04，迭代次数为 686，路径如下：



双向 A*搜索最终代价为 63.28，迭代次数为 183，结果如下：



4.3.2. 结果分析

从可视化图中可以看到，单向 A 搜索几乎走遍了整张地图。虽然使用了启发函数，但其对地形代价无法有效地估计，因此在前期，搜索近乎径直向终点前进。但到了沙漠和水面附近，由于代价瞬间增大，A*不再径直前进，而是往其它的方向继续探索，几乎遍历完了整张地图。而且，算法最终找到的路径也比双向搜索稍差。

双向 A 搜索从起点和终点同时开始搜索，有效地降低了搜索空间。从搜索图中可以看到，走过的路径相比单向搜索减少了很多，迭代次数从 686 降低到 183。因此，双向 A*搜索能有效地提高效率。而且，最后也得到了比单向搜索更优的路径。

5. 心得与总结

本次实验是要实现单向和双向的 A*搜索算法，实际代价除了移动代价外，还有地形代价。在实现双向 A*算法时，终止条件是我思考了很久的一个问题。网上关于双向 A*的很多博客都介绍地比较模糊。有的指出 OPEN 结点重叠时，成功找到路径，算法终止；而我认为，应该让 CLOSE 重叠时才算成功，因为 CLOSE 才表示搜索真正到达的结点。当然，两种方式可能差别不是很大。

本次实验做的比较工程，在实现基础算法之后，我把重心放到了结果的可视化上。我的目标是不但要能展示地图和最终路径，还要能展示搜索的整个过程。为此，我复习和补充了前端和后端的相关知识，例如复习了 Spring 框架，学习了 Vue。前端的逻辑比我最初想象的要复杂，由于前端代码管理能力较差，每添加一个新功能，都需要调试很久，这是我之后要锻炼的地方。相反，我的后端代码得益于良好的设计模式和不断地重构，能够进行方便地维护。比如，Service 需要调用不同的 A*算法（单向和双向），我就把 A*算法定义为一个接口，然后分别编写两个实现类。为了能利用 Spring 对 A*算法的 Bean 进行管理，我实现了一个 A*工厂类，在 Service 中自动装配这个工厂对象，然后能够通过这个工厂对象调用不同的 A*示例。最后，我对可视化的期望目标得以成功实现。