Lab 2

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 实验问题

寻路问题。以图 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√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 的所有移动的代价的总和;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√2+1(移动)+0(地形)，而 路径 D→E→F 的代价为 2(移动)+6(地形)。

3 实验步骤

3.1 单向A*算法

主要思想

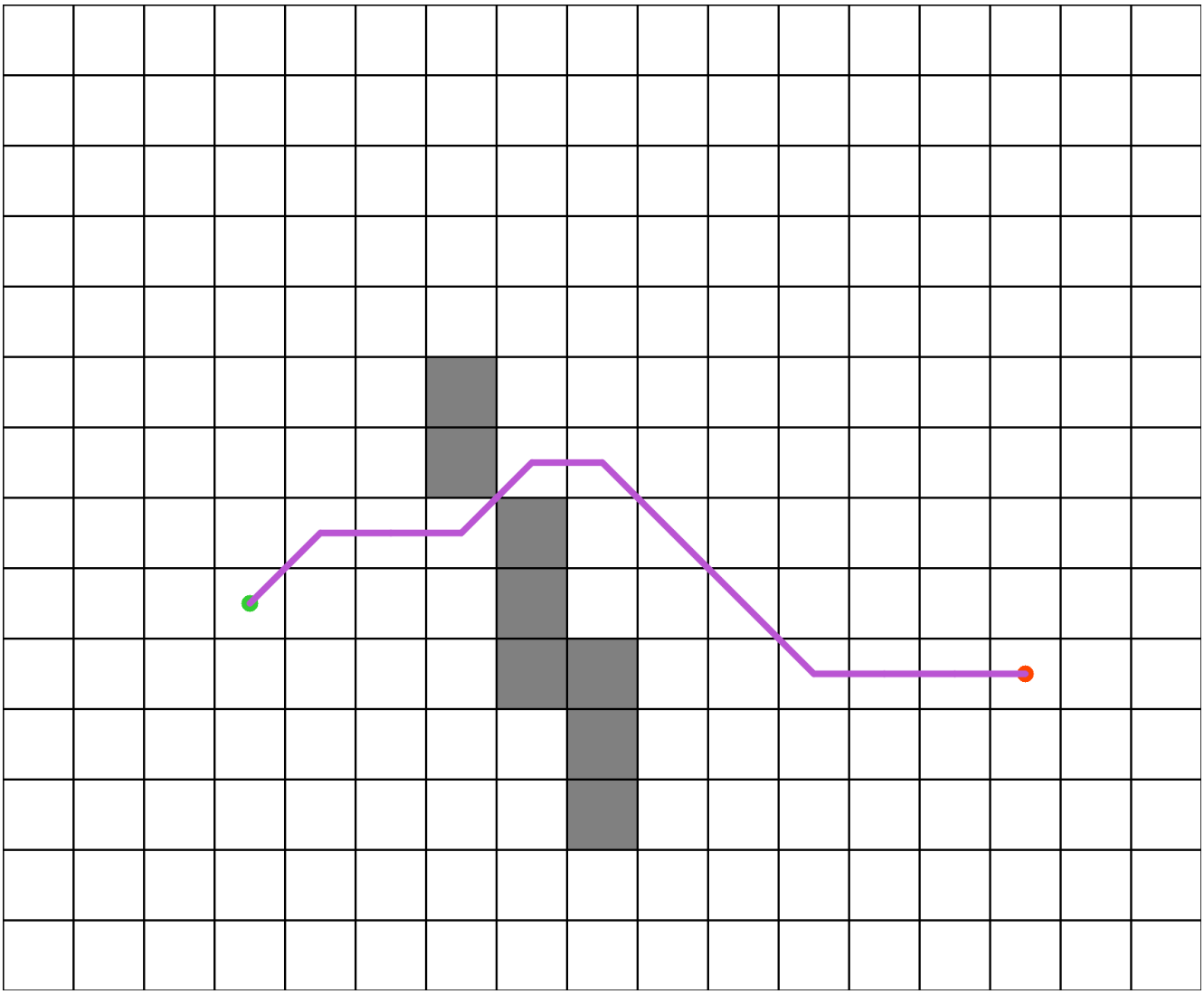
A*算法可以看作Dijkstra算法的延伸，只是在优先队列中的优先级用g(n)+h(n)表示，g(n)表示n到源点到最短距离，h(n)表示n到目标点的估计距离。当h(n)等于0时，即退化为Dijkstra算法。这个算法避免了像Dijkstra算法那样遍历全图，而是趋向选择距离目标点更近的方向，直到到达目标点。在本实验中，h(n)为n与目标点的欧式距离。

```
// 欧式距离
auto h = [](const Location& a, const Location& b) {
  return sqrt(pow(a.first - b.first, 2) + pow(a.second - b.second, 2));
};
```

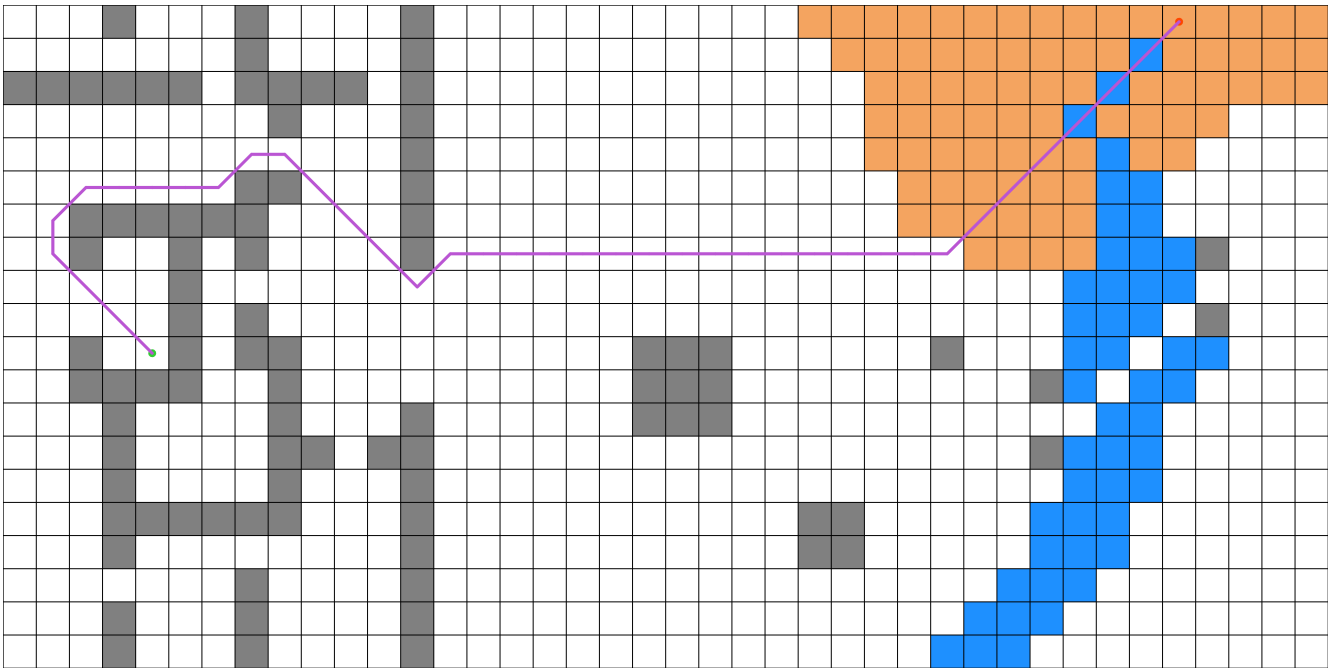
```
// 更新优先级
if (dist[p.first][p.second] + weight < dist[q.first][q.second]) {
  dist[q.first][q.second] = dist[p.first][p.second] + weight;
  edge[q.first][q.second] = p;
  pq.push({dist[q.first][q.second] + h(q, dst), q});
}
```

结果可视化

- 测试用例1



- 测试用例2

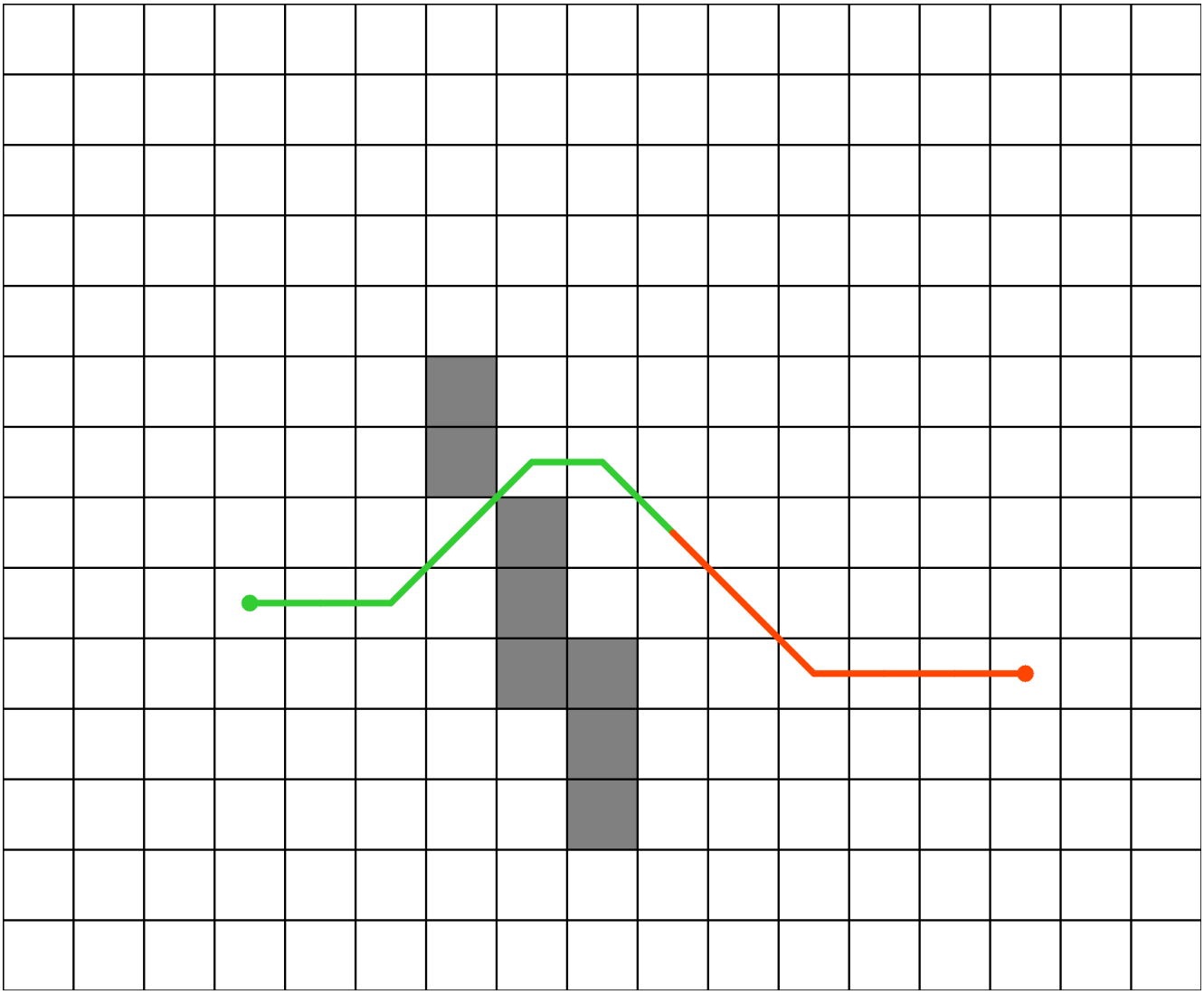


3.2 双向A*算法

主要思想

单向A*算法可以看作从起点开始扩展生成树，直到遇到终点停止，时间复杂度是b^d，b是每个节点的平均邻接点的个数，d是搜索的深度。双向A*算法则是同时从起点和终点开始搜索，直到两个生成树交叉重叠为止，时间复杂度为$2b^{d/2}$。双向A*实现的关键在于需要存储搜索过程中的生成树，以判断生成树是否重叠。在本实验中，使用到的启发式函数h和单向A*一致。

- 测试用例1



- 测试用例2