Project 2: A* 算法

1 A* 算法原理与实现

A* 算法是一种启发式搜索算法，结合了 Dijkstra 算法 (UCS) 和最佳优先搜索（贪心）的优点。其核心是通过评估函数来指导搜索方向：

$$f(n) = g(n) + h(n) \tag{1}$$

其中 $g(n)$ 表示从起始节点到当前节点 n 的实际代价，$h(n)$ 表示从节点 n 到目标节点的估计代价（启发函数）。

常用的启发函数包括：

Manhattan 距离： 在只允许横平竖直移动时

$$h(n) = |x_1 - x_2| + |y_1 - y_2| \tag{2}$$

对角距离： 在允许沿对角线移动时

$$h(n) = D_1 \times \max(|dx|, |dy|) + (D_2 - D_1) \times \min(|dx|, |dy|) \tag{3}$$

其中 D_2 时沿对角线移动的代价，D_1 是横竖移动的代价

Euclidean 距离： 在允许任意移动时

$$h(n) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{4}$$

当启发为 0 时，则退化为 Dijkstra 算法。

1.1 可容性 (Admissible) 与树搜索:

启发函数不应高估实际代价，即：

$$\forall n : h(n) \leq h^*(n) \tag{5}$$

其中 $h^*(n)$ 是从节点 n 到目标的实际最小代价。对于树搜索来说，即使存在重复节点，只要启发函数满足可容性，A* 算法仍然能保证最优，只是可能多次访问同一状态的不同路径，增加了搜索空间。

1.2 一致性 (Consistent) 与图搜索:

启发函数满足三角不等式，即：

$$\forall n, n' : h(n) \leq c(n, n') + h(n') \tag{6}$$

其中 $c(n, n')$ 是节点 n 到 n' 的代价。一致性蕴含可容性，是更强的条件。在图搜索中需要一致性才能保证最优性，并且用一个 closed 表来避免重复访问。在这个过程中，

- 第一次访问节点时就找到了到达该节点的最优路径
- $f(n)$ 值沿着搜索路径单调非减
- 不需要重新打开已关闭的节点（closed 表中的节点不会被重新考虑）

如果存在启发高估了实际代价，就不能保证结果的最优性，此时表现会类似于最佳优先搜索。

1.3 重要性质

- **完备性：** 当分支因子有限且所有边的代价都大于某个小的正数 ϵ 时，算法是完备的。
- **最优性：**
  - 树搜索：启发函数满足可容性即可
  - 图搜索：需要启发函数满足一致性
- **时间复杂度：** 最坏情况下为 $O(b^d)$，其中 b 是分支因子，d 是解的深度。实际性能强烈依赖于启发函数的质量和搜索策略的选择。
- **空间复杂度：**
  - 树搜索：不强制需要 closed 表，但可能重复访问节点，体现时间-空间权衡
  - 图搜索：需要额外的 closed 表空间

1.4 算法实现

A* 算法的实现可以分为三个重要步骤：

1.4.1 准备阶段

创建待探索列表 (openSet)，初始只包含起点和已探索列表 (closedSet)，初始为空，并给起点设置初始值。

*

1.4.2 探索过程

每次从待探索列表中选择预估总距离最小的点作为当前节点，如果当前节点就是目标，那么成功，否则将当前节点移到已探索列表并查看其所有邻居节点

1.4.3 对每个邻居节点的处理

如果邻居节点已经在已探索列表中，就跳过它（因为先前已经找到了到达该点的最佳路径），然后计算从起点经过当前节点到达这个邻居的距离。如果这个邻居是新发现的（不在待探索列表中），就把它加入待探索列表；如果找到了到达这个邻居的更短路径，就更新它的距离信息。之后更新这个邻居的预估总距离。

最后，如果待探索列表变空还没找到目标，说明没有可行路径，返回失败。形式化描述如算法 1.

Algorithm 1 A* 算法（图搜索）

openSet ← {startNode}
closedSet ← {}
g[startNode] ← 0
f[startNode] ← h(startNode)
while openSet 不为空 **do**
  current ← openSet 中 f 值最小的节点
  if current = goalNode **then**
    return 重建路径 ()
  end if
  从 openSet 中移除 current
  将 current 添加到 closedSet {避免重复访问}
  for each neighbor of current **do**
    if neighbor 在 closedSet 中 **then**
      goto nextNeighbor
    end if
    tentative_g ← g[current] + dist(current, neighbor)
    if neighbor 不在 openSet **then**
      将 neighbor 添加到 openSet
    else if tentative_g ≥g[neighbor] **then**
      goto nextNeighbor
    end if
    g[neighbor] ← tentative_g
    f[neighbor] ← g[neighbor] + h(neighbor)
    nextNeighbor:
  end for
end while
return failure

或用流程图描述：

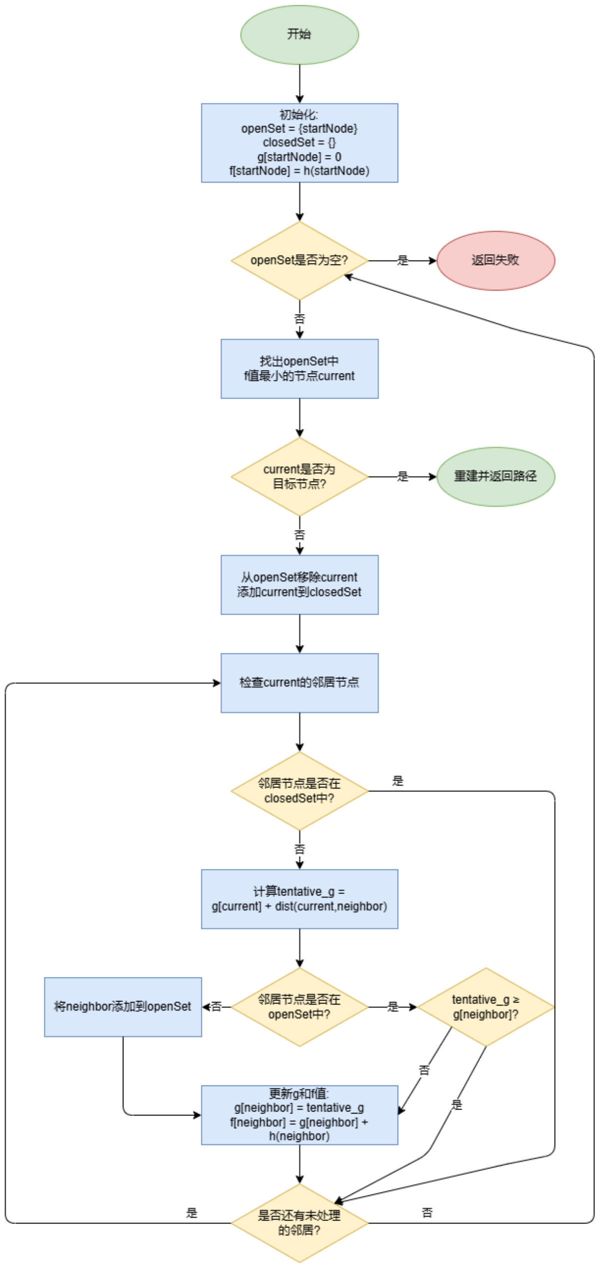

图 1 A* Flowchart

2 部署说明

为了对算法过程进行可视化展示，我采用了 React 框架搭建了一个简易网页，因此算法部分使用 JavaScript 来实现。在项目目录中，使用

```
npm install
```

来安装 package.json 中的依赖。使用

```
npm start
```

来运行项目。在网页首页可以选择进入哪一题

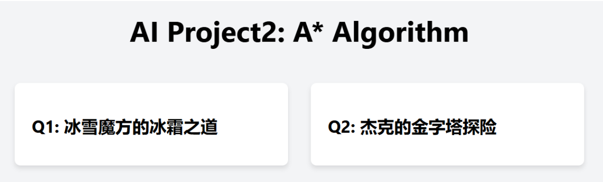

图 2 Main Page

由于 JavaScript 没有内置的优先队列实现，我使用了 @datastructures-js/priority-queue 包。

3 Q1：八数码问题

3.1 启发函数设计

第一个问题的启发函数选取比较直观。我们只能横竖滑动空冰块，可以把每一次移动的代价视为 1，设当前状态为 n，每个数字块的坐标为 (x_i, y_i)，而在目标状态 n^* 中为 (x_i^*, y_i^*)。定义启发函数 $h(n)$ 为所有数字块的 Manhattan 距离之和：

$$h(n) = \sum_{i=1}^{8} \left(|x_i - x_i^*| + |y_i - y_i^*|\right)$$

对于一致性，我们考虑任意两个相邻状态，则有 $c(n, n') = 1$（即一次滑动的代价），并且由于只有一个数字块的位置改变，因此 Manhattan 距离之和至多减少或增加 1，故满足 $h(n) \le c(n, n') + h(n')$，由此可以归纳出任意两个状态满足一致性。如果更简单一些，我们甚至可以直接用不在目标位置上的冰块数目（即它们的汉明距离）作为启发

$$h(n) = \sum_{i=1}^{8} \delta\left(x_i \neq x_i^* \text{ OR } y_i \neq y_i^*\right)$$

但这个的精度显然要低一些，因为他对距离的估计更保守了。

在代码上，如果我们用一个字符串来存储冰块的状态，我们可以遍历当前状态的每一个位置，如果不是空冰块，找到在目标状态中这个数字的位置，分别计算它们在哪一行哪一列，根据差值计算出 Manhattan 距离，累加所有数字的距离即为最终的启发值。

```
// 计算曼哈顿距离作为启发式函数
calculateHeuristic() {
  let distance = 0;
  for (let i = 0; i < 9; i++) {
    if (this.state[i] !== '0') {
      const currentPos = i;
      const targetPos = GOAL_STATE.
          indexOf(this.state[i]);
      // 计算当前位置和目标位置的行列坐标
      const currentRow = Math.floor(
          currentPos / 3);
      const currentCol = currentPos % 3;
      const targetRow = Math.floor(
          targetPos / 3);
      const targetCol = targetPos % 3;
      // 计算曼哈顿距离
      distance += Math.abs(currentRow -
          targetRow) + Math.abs(
          currentCol - targetCol);
    }
  }
  return distance;
}
```

3.2 代码实现

代码中有很多可视化逻辑的部分，我们在这里只着重说明算法逻辑部分。为了方便起见，我们设计一个状态类，

```
class PuzzleState {
  constructor(state, gScore = 0, previous
      = null, swapIndex = null) {
    this.state = state; // 当前状态字符串
    this.gScore = gScore; // 从初始状态到
        当前状态的实际代价
    this.hScore = this.calculateHeuristic
        (); // 启发式评估值
    this.fScore = this.gScore + this.
        hScore; // 总评估值 f = g + h
  }
```

它有两个核心功能，一个是计算当前状态的启发，第二个是获得当前状态的邻居。前者我们刚才已经介绍过，对于后者，我们要找到空白块的位置，然后遍历每一种移动方式，筛选出合理的（移动后坐标合法），加入加入 neighbors

```
// 获取所有可能的相邻状态
getNeighbors() {
```

```
  const neighbors = [];
  const zeroPos = this.state.indexOf('0');
  const zeroRow = Math.floor(zeroPos / 3);
  const zeroCol = zeroPos % 3;
  const moves = [[-1, 0], [1, 0], [0, -1], [0, 1]];

  moves.forEach(([dx, dy], index) => {
    const newRow = zeroRow + dx;
    const newCol = zeroCol + dy;
    if (newRow >= 0 && newRow < 3 && newCol >= 0 && newCol < 3) {
      // 后续处理，加入neighbors
    }
  });
  return neighbors;
}
```

在 A* 算法中，我们首先要准备好一个 openset 的最小优先队列，因为我们总是要取出里面 f 值最小的，它的实现可以实现 $O(1)$ 的复杂度，以及一个 closedSet 保存访问过的状态

```
const start = new PuzzleState(initialState);
const openSet = new MinPriorityQueue((state) => state.fScore);
openSet.enqueue(start);
const closedSet = new Set();
```

然后我们每次从 openSet 中取出一个，如果已经是目标状态就进行路径重建并返回，如果不是，则把这个状态加入以访问节点并找到它的邻居

```
while (!openSet.isEmpty()) {
  const current = openSet.dequeue();

  if (current.state === GOAL_STATE) {
    // 路径重建
  }

  closedSet.add(current.state);

  // 探索相邻状态
  const neighbors = current.getNeighbors();
}
```

对于每一个邻居，如果已经在 closedSet 里面我们直接跳过，

```
  for (const { state: neighbor, direction } of neighbors) {
    if (closedSet.has(neighbor.state))
      continue;
  }
```

否则我们检查是否已经在 openSet 里，如果在，就检查我们是否发现了一个更好的路径，如果是则更新，如果不在 openSet 里则新加入

```
    const existingNode = openSet.toArray().find((node) => node.state === neighbor.state);
    if (existingNode) {
      if (neighbor.fScore < existingNode.fScore) {
        openSet.remove(existingNode);
        openSet.enqueue(neighbor);
      }
    } else {
      openSet.enqueue(neighbor);
    }
```

通过以上步骤，我们就实现了算法 1 中所描述的 A* 算法流程。

3.3 结果展示

以给的实例样例来看：

图 3 Example Case, Step 1

在每一步中，我们可以清晰看到当前状态，OpenSet 中的状态，每个状态的代价拆解以及状态变化历史。这里粉色块表示空白块是和哪个块交换才得到这个状态的。具体来说，测试 1 中，从初始状态，有向左（和 5 换）和向下（和 2 换）两种可能操作，我们选择了总代价更低的前者，注意此时它们的紫色数字，也就是 g 值都是 1. 然后

图 4 Example Case, Step 2

然后我们的当前状态变成了刚才选择的状态，这时候在执行下一步的时候，我们的 OpenSet 里不仅有现在这个状态的邻居（它们的 g 是 2），还有先前未选择的（g 是 1）。还需要注意，这里是没有空白块向右移这种操作的，因为这样的话就回到了刚才的状态，而我们不希望重复。然后继续选择总代价最小的。

图 5 Example Case, Fin

此时我们已经找到了目标状态，成功完成。然后我们来看一下其余的测试样例。前两个比较简单，一开始有三个邻居，其中总代价最小的已经是最终答案了（绿色的 h 值，即 Manhattan 距离为 0），都只需要 1 步

图 6 Test Case 1

图 7 Test Case 2

第三个测试和示例是类似的，需要两步

图 8 Test Case 3

第四个也只需要一步

图 9 Test Case 4

最后一个稍微复杂一些，一开始有三个邻居，我们选择向上移动

图 10 Test Case 5, Step 1

然后未选择的两个仍保留在 openSet 里，新增一个当前状态向右的 (g 为 2)

图 11 Test Case 5, Step 2

然后 openSet 中再增加两个新邻居，此时它们的 g 应该是 3

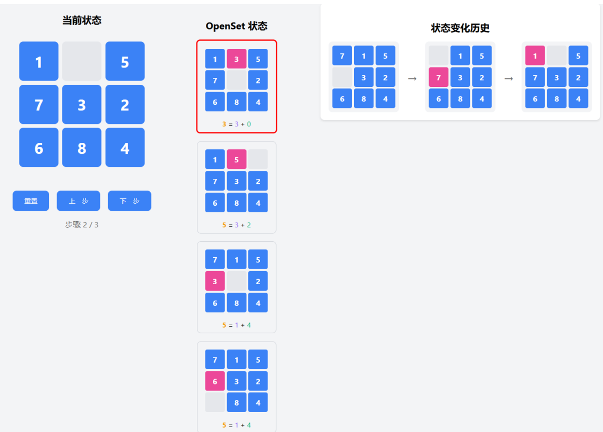

图 12 Test Case 5, Step 3

这里最小的就是我们最后答案，因此总共需要三步。

4 Q2: K 最短路径问题

4.1 启发函数设计

这个问题里面启发函数的设计比较困难，因为我们在测试样例 1 中可以看到，存在 5 为终点，3 到 5 距离为 1 的这种通道，此时如果我们直接用差来作为启发函数的话，显然会存在高估的问题，导致不可容。那么此时我们可能会想直接去修正它，保证启发小于等于 1，例如 $\frac{D_{max}-D_n}{D_{max}-1}$ 这种归一化的距离，满足条件。但其实仔细思考的话，这个题目里的层数只是一个幌子，因为层本身不连接，是输入提供的通道让他们连接，实际上的距离和层数的差本质上没有关系。另一种更粗暴的策略就是对于除了目标点之外的启发都采用 1 来估计，也满足可容性，但必然很低效。因此，这两种启发都是不合理的，我们需要寻找新的启发：

- 采用当前位置到所有能直接去的地方的距离的最小值，这个距离必然小于等于从当前位置到终点的距离。如果我们回到一致性的定义，$h(n) \leq c(n, n') + h(n')$，这个启发很直接，因为他就是直接控制 $h(n) \leq c(n, n')$，并且启发都是非负的。
- 先运行一遍 Dijkstra 算法，用终点作为起点，从而得到终点到其他点的真实距离作为启发。这种想法的合理性在于，由于这是一个 K 最短路径问题，而不只是给出一条最短路径，当数据量非常大，K 非常多的时候，跑一遍 Dijkstra 算法的开销在整体中占的比例可能很低，但它带来了最精准的启发，或许可以显著加快 A* 算法。并且它肯定是满足一致性的。

那么我们可能会去思考，这两种启发到底哪种更好呢？我们先来进行代码实现。

4.2 代码实现

由于需要比较性能和画图等，这里我使用了 Python。我们设计一个 Pathfinder 类，它需要记录总共有多少层，和层之间连接的图（这个图是单向的），实际上我们管理的是邻接表

```python
class PathFinder:
    def __init__(self, n, edges):
        self.n = n
        self.graph = defaultdict(list)
        for x, y, d in edges:
            self.graph[x].append((y, d))
```

然后我们实现第一种启发，在这里我稍微精细了一下，如果当前点不能直接到达终点，我们的启

发是当前点所有能直接去的地方的距离 +1，因为去到那个地方后去终点至少还需要 1；如果可以直接到达终点，就返回直接到终点的距离和刚才所说的距离里面较小的那个。本质上就是在 $h(n) \leq c(n, n') + h(n')$ 中，如果 n' 是终点，那么 $h(n') = 0$，否则 $h(n') \geq 1$。这里如果已经到达终点，启发应该是 0；如果没到终点却无路可走了，说明走了错误的路，启发是无穷。

```
def get_min_edge(self, node):
    if node == self.n:
        return 0
    if not self.graph[node]:
        return float('inf')
    min_dist = min([weight + 1 for _, weight in self.graph[node]])
    for next_node, weight in self.graph[node]:
        if next_node == self.n:
            return min(min_dist, weight)
    return min_dist
```

然后我们实现第二种启发，以终点为起点做 Dijkstra 算法，注意这里我们要改成只能从下往上走，算法描述如算法2，流程图描述如图13。

如果采用这种启发，我们在 A* 算法中执行一开始先运行 Dijkstra 并保存，这样之后要获取启发的时候直接读取即可。在 A* 算法里，我们还是初始化一个优先队列和访问过的节点集合，Python 种我们可以用内置的 heapq 中的最小堆。此外这里有一个不一样的地方就是，我们需要找到 K 条最短路径，而不是一条，所以我们用一个列表保存找到的路径。

```
paths = []
pq = [(heuristic(1), 0, 1, [1])] # (estimated_total, current_distance, node, path)
seen = set()
```

相应的，循环条件除了优先队列非空之外，还要加上找到的路径少于 K 个，然后每次弹出总代价最小的，如果此时节点就是终点，因为我们要找到 K 条路径，我们不能直接结束，而是需要继续下一轮

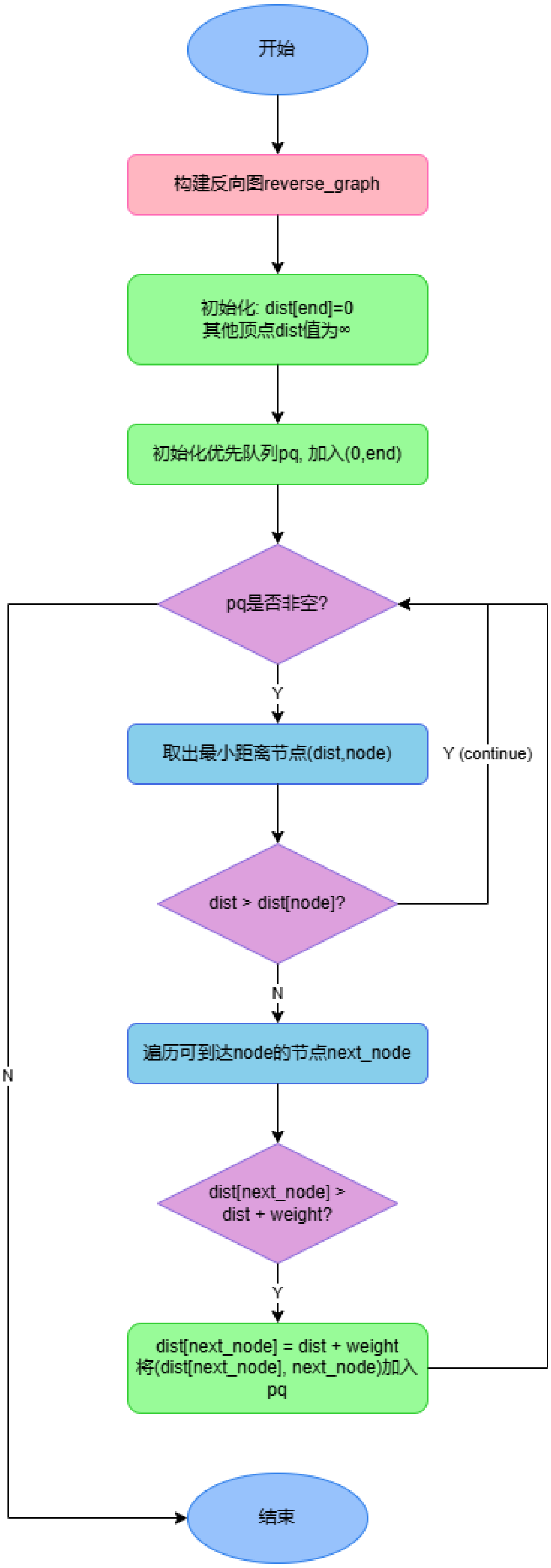

图 13 Dijkstra, Flowchart

Algorithm 2 反向 Dijkstra 算法

1: 构建反向图 reverse_graph
2: 初始化：$dist[end] = 0$，其他顶点的 $dist$ 值为 ∞
3: 初始化优先队列 pq，加入 $(0, end)$
4: **while** pq 非空 **do**
5:     $(dist, node) =$ 从 pq 中取出最小距离的节点
6:     **if** $dist > dist[node]$ **then**
7:         continue {惰性删除，跳过过期值}
8:     **end if**
9:     **for** 每个可以到达 $node$ 的节点 $next\_node$ **do**
10:         **if** $dist[next\_node] > dist + weight$ **then**
11:             $dist[next\_node] = dist + weight$
12:             将 $(dist[next\_node], next\_node)$ 加入 pq
13:         **end if**
14:     **end for**
15: **end while**

循环，然后把这个路径加入已访问的路径

```
while pq and len(paths) < k:
    _, current_dist, node, path = heapq.heappop(pq)

    if node == self.n:
        paths.append((current_dist, path))
        continue

    path_tuple = tuple(path)
    seen.add(path_tuple)
```

在正常情况下，我们要找到当前 node 能去到哪些 node，如果是新路径把这些新邻居加入优先队列。这里有一个小区别就是因为我们要找到 K 条路径，所以此时我们没有先前的更新到某一个节点的更小距离那一步，而是把所有可能路径（不重复）都加入优先队列。

```
for next_node, weight in self.graph[node]:
    if next_node > node: # 只能向下移动
        new_path = path + [next_node]
        if tuple(new_path) in seen:
            continue
        new_dist = current_dist + weight
        estimated_total = new_dist + heuristic(next_node)
        heapq.heappush(pq, (estimated_total, new_dist, next_node, new_path))
```

当然这里要注意，如果循环结束后找到的路径不足 K 个，我们需要按要求补全-1

```
while len(paths) < k:
    paths.append((-1, []))
```

通过上述步骤，我们就成功解决了这个问题。

4.3 结果展示

我同样制作了一个简易的 React 网页，基于 JavaScript 的代码实现，由于逻辑一样，我就不重复展示代码了。简单起见，我采用了第一种启发。我们来详细跟踪一下示例，理解它的流程，验证我们算法的正确性：

这里所有的边实质上是单向的，即只能从小数字往大数字，为了美观我省略了。浅蓝色的节点包表示路径历史节点，深蓝色则为当前节点，路径用蓝色的线连接。

首先从 1 开始，这里队列中有两种可能选择，分别是 1 到 2 和 1 到 3。两者的 g 分别为 1 和 4，而 h 根据我们的约定，2 的启发是 4，因为他只能往 4 走，这段代价是 3，还需要再加 1。3 的则为 1，因为他有一条直接通往 5 的路线并且为 1。这里我们可以体会到我们刚才的精细的启发的优势，假设 3 到 5 的边变成 2，它还是会一定选择这条边，因为 3 到 4 这条边的代价虽然也是 2，但是我们对于不是直接到终点的还会再加 1

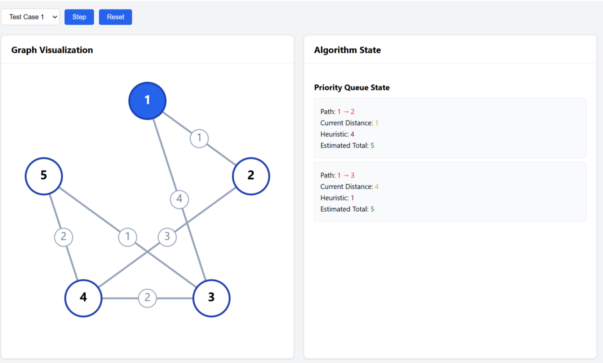

图 14 Step1

这里两者 f 一样，程序选择了前者。走到 2 后会在队列里增加 1,2,4 这条路径，但总代价更大了，所以接下来要回退到刚才的 13

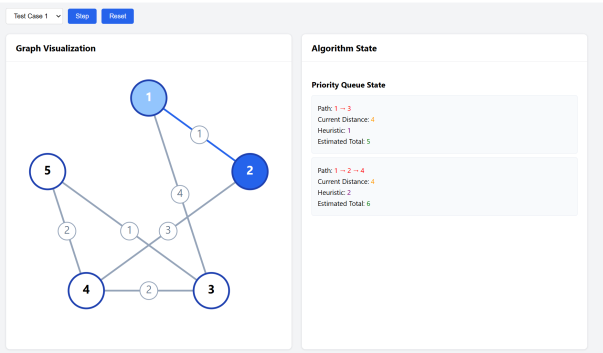

图 15 Step2

此时新增 134 和 135 两条路径

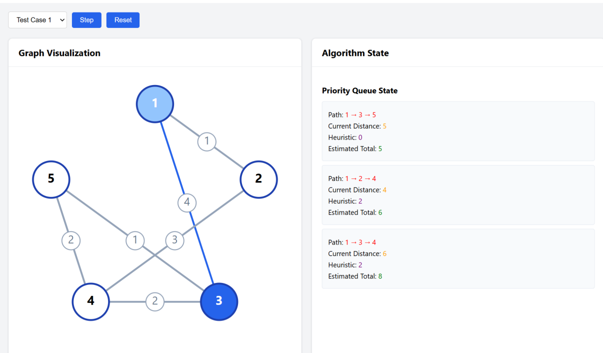

图 16 Step3

注意在 A* 算法里，虽然我们此时找到了终点，它的启发是 0，但是我们需要的是保证这条路径在最小队列里面最优先，并实际去访问它，弹出队列，然后重构路径

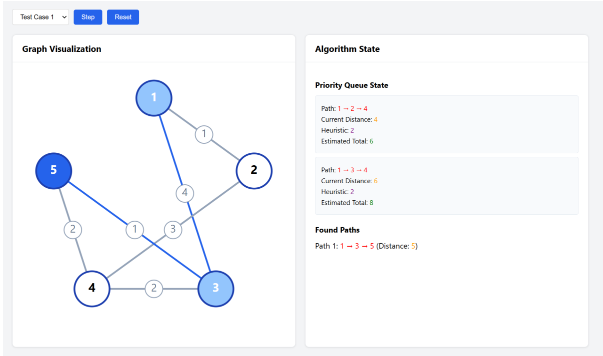

图 17 Step4

这时候根据优先队列，我们需要回到 124 这个状态，新增 1,2,4,5 路径

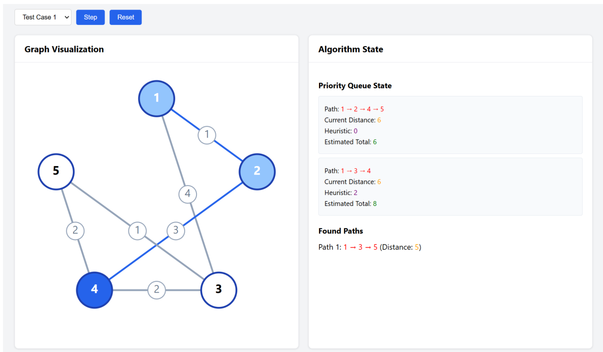

图 18 Step5

由此我们找到了第二短的路线

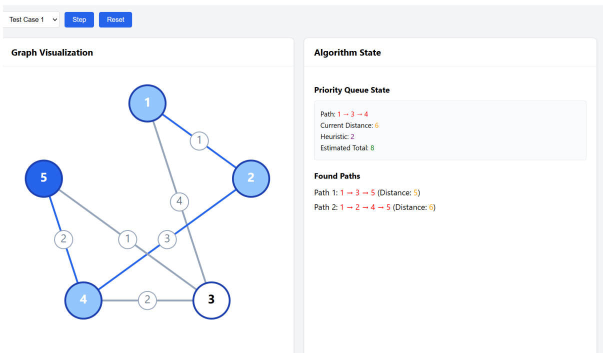

图 19 Step6

然后处理优先队列剩余内容

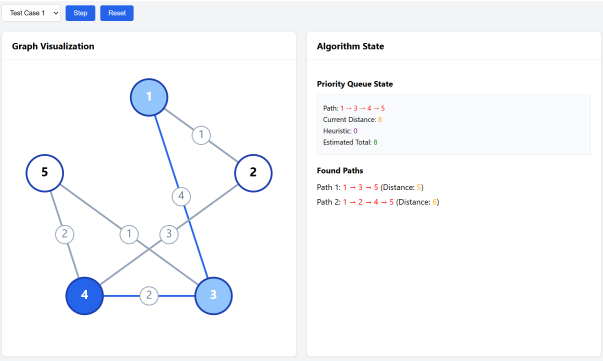

图 20 Step7

即可找到三个路线

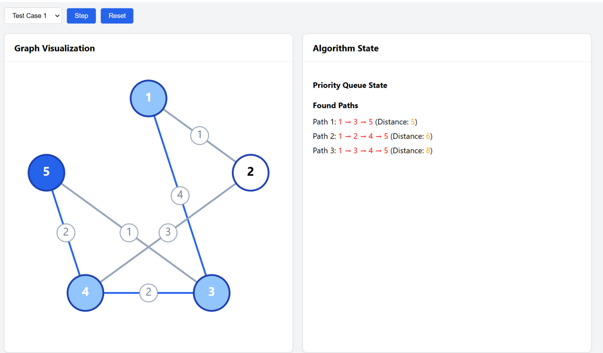

图 21 Step8

这里有一个比较巧合的事情，因为总共只有三个路径，所以我们同时达成了清空优先队列和找到 K 条路径。其他测试样例类似，我在此就不重复分析了。不过有一个情况需要指出，例如在第一个测试样例中：

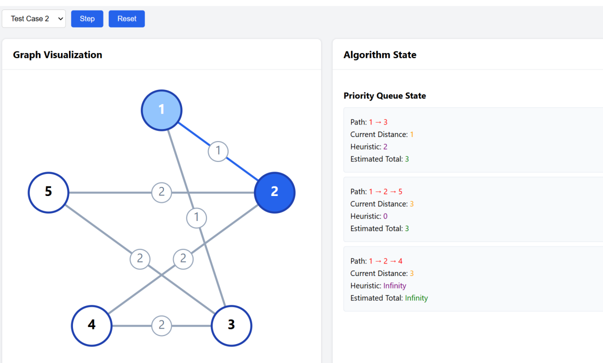

图 22 Special Case, 1

这里我们发现出现了 1,2,4 这个路径，但 4 接下来五路可走了，它的启发是无穷大，相应地总代价也是无穷。最后会出现优先队列只剩下堆积在底部的这些思路，然后程序会一个个处理，把他们弹出优先队列，但由于它们也没有邻居，所以只是清理优先队列而已。当全部清空后程序就会终止：

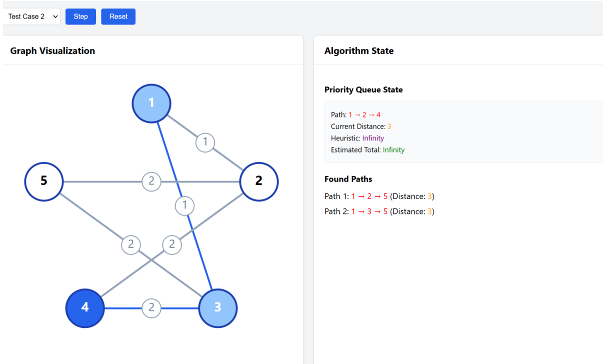

图 23 Special Case, 2

当然更高效的做法是不把这种路径加入队列，不过我们这里只是为了演示我们逻辑的正确性。

对于其他测试样例，我就直接给出 Python 版本的运行结果，采用两种启发可以得到相同的结果：

```
Test case: n=5, m=6, k=4
Heuristic 1 results: [3, 3, -1, -1]
Heuristic 2 results: [3, 3, -1, -1]


Test case: n=6, m=9, k=4
Heuristic 1 results: [4, 5, 6, 7]
Heuristic 2 results: [4, 5, 6, 7]


Test case: n=7, m=12, k=6
Heuristic 1 results: [5, 5, 6, 6, 7, 7]
Heuristic 2 results: [5, 5, 6, 6, 7, 7]


Test case: n=5, m=8, k=7
Heuristic 1 results: [4, 4, 5, -1, -1, -1, -1]
Heuristic 2 results: [4, 4, 5, -1, -1, -1, -1]


Test case: n=6, m=10, k=8
Heuristic 1 results: [5, 5, 6, 6, 6, 8, -1, -1]
Heuristic 2 results: [5, 5, 6, 6, 6, 8, -1, -1]
```

图 24 Results

为了展示紧凑，我直接把结果打印成一行列表了，没有分行打印。此外程序中还保存了路径，如果需要也可以展示，如对于示例：

```
Test case: n=5, m=6, k=3
Heuristic 1 results: [5, 6, 8]
Path: [[1, 3, 5], [1, 2, 4, 5], [1, 3, 4, 5]]
Heuristic 2 results: [5, 6, 8]
Path: [[1, 3, 5], [1, 2, 4, 5], [1, 3, 4, 5]]
```

图 25 Paths

4.4 性能比较

我们设计一组简单的实验比较两种启发算法的性能，N 从 2^{15} 到 2^{20} 次方，M 为 N 的两倍，K 则为 $N//20$，对于每一种配置，随机生成 100 组数据，保证所有边都是从上往下，边距离的最大值是 N 的一半，记录两种启发在这 6 种数据量，每组重复 100 次实验下的均值和标准差，额外记录第二种时里面运行那次 Dijkstra 算法花费了多久，并作图（横轴采用了对数尺度）：

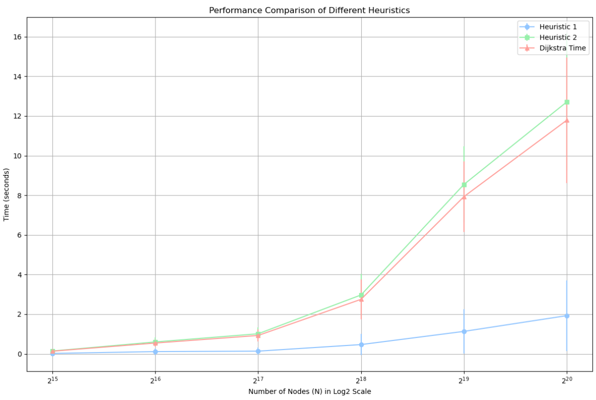

图 26 Performance Comparison

在运行 100 次平均后，我们发现：

- 对于启发 2，运行 Dijkstra 算法需要的时间占据了 A* 算法需要的总时间的很大比例，这个其实和我们一开始设想的不大一样，我们可能会觉得数据量大的时候一次 Dijkstra 算法的时间占比会变小，但实际上数据量大的时候 Dijkstra 算法也会很耗时
- 对于启发 1，我们可以在图上看到，它的效果明显更好，平均值要低很多。

我们推测，这可能是由于，当数据量很大的时候运行 Dijkstra 算法得到了所有点到终点的距离，但是这里面很多可能是不必要的，但另一方面我们也无法预测哪些是需要的。因此，虽然我们从图上可以看出来，启发 2 减去 Dijkstra 的时间，也就是真正 A* 的时间是小于启发 1 的，更精准的启发确实加快了 A*，但得到这个启发的代价实在太大。相比较而言，启发一虽然对于真实距离的逼近没有那么精准，但他的计算简单，综合效率更高。

但这个实验仅仅只是给出了一种工况下的结果，实际运行效果可能受具体的数据影响。例如我们可能会推测，有些情况下，启发 1 的碰运气的成分更大，得到的结果可能会更不稳定等。因此要比较清楚它们的性能，还需要结合具体的数据。

5 补充问题

在 Q1 里，虽然我们没有涉及，但这个问题存在一个一般性的可解性讨论。我们可以把它写成一维的形式，如 1 2 3 4 5 6 7 8 0，根据线性代数的知识可以计算这个排列的逆序数，即出现较大的数在较小的数前面的次数，可以通过数学方法证明，只有始末状态排列的逆序数奇偶性相同，才可达，否则无解。

在 Q2 中有一些细节问题需要考虑，在给的示例样例中，出现了 5 1 5，即 5 到 1 的路径，但是根据题目描述的话，应当只允许下坡的路，并且后面的五个样例中并没有这个情况，因此我直接将这一个路径删除了。此外，在启发函数中，我默认了每条路径代价至少为 1，没有 0 代价的路线。