常州工学院
CHANGZHOU INSTITUTE OF TECHNOLOGY

人工智能
实验报告

实验名称：	A*算法求解迷宫寻路问题实验
专业班级：	22 软件三
姓　　名：	杨熙承
学　　号：	22030531
指导教师：	刘文杰
实验成绩：	
批改日期：	

计 算 机 信 息 工 程 学 院

实验一 A*算法求解迷宫寻路问题实验

一、实验目的

熟悉和掌握 A*算法实现迷宫寻路功能，要求掌握启发式函数的编写以及各类启发式函数效果的比较。

二、实验内容：

迷宫问题，作为实验心理学中的一项经典课题，其核心在于求解从入口至出口的最优化路径，即在众多可能路径中筛选出最短的一条。在本实验中运用技术手段，特别是利用 matplotlib 库的数据可视化功能，来模拟并展示这一求解过程。

采用 matplotlib 库中的 Circle 函数来绘制迷宫的节点，Rectangle 函数则用于描绘节点间的连接边。随后，借助 matplotlib.pyplot 模块中的 ion()函数，以动态图形的形式，逐一呈现算法在搜索过程中的每一个关键阶段。

采用随机生成的深度优先搜索法作为基本框架。在迷宫构建的初始阶段，仅设定一个由固定大小节点组成的阵列，而节点间并无直接连接。随后，从预设的起点出发，算法将随机选择四个可能的方向进行探索。若某方向的相邻节点尚未被访问，则在该两点间生成一条边，并将搜索焦点转移至该新节点，重复上述过程直至所有节点均被遍历，此时算法宣告结束。

为进一步增加迷宫问题的挑战性，将随机在迷宫内部增添 k 条边，以此构造出包含多条可行路径的复杂迷宫结构，从而更全面地检验并优化求解算法。

三、实验预习和准备

学习 A*算法的基本概念，包括启发式搜索、曼哈顿距离作为启发式函数的选择，以及 A*算法如何综合启发式值和路径代价来找到最优解。启发式函数在 A*算法中起到了关键作用，我通过比较不同启发式函数（如欧几里得距离、曼哈顿距离）对于路径搜索的影响，理解了它们在不同场景中的效果差异。

在学习过程参考了若干资料，查阅了斯坦福大学网站上的一篇关于 A*算法的比较文章（https://theory.stanford.edu），深入了解了不同搜索算法的对比，包括 A*算法与其他

算法在路径搜索效率上的优劣势。同时，我还参考了 Red Blob Games 网站上的一篇图解（https://www.redblobgames.com），该图解生动地展示了 A*算法的工作过程，帮助我更直观地理解了算法的运行机制。此外，我观看了 YouTube 上的一段关于 A*算法的讲解视频（https://www.youtube.com/watch?v=T8mgXpW1_vc），通过视频的动态演示进一步加深了对 A*算法的理解。

参考《算法图解》这本书，这本书以浅显易懂的方式解释了各种经典算法的工作原理，特别是 A*算法如何在不同情况下找到最优路径。同时，为了更好地理解 A*算法的工作原理，我复习了广度优先搜索（BFS）和 Dijkstra 算法。广度优先搜索为我提供了路径搜索的基础知识，而 Dijkstra 算法帮助我理解了路径代价的计算和优先队列的使用，这些都是 A*算法的重要组成部分。

研究迷宫问题的随机生成方法，特别是如何通过深度优先搜索（DFS）来生成迷宫结构。通过随机生成的 DFS 方法，我实现了一个初始节点的迷宫生成框架。为了增加迷宫的复杂性，我在生成迷宫之后随机增加了若干条边，以创造更多可能的路径，从而增加 A*算法在搜索路径过程中的挑战性。

查阅了 matplotlib 库的相关资料，学习了如何使用 matplotlib 进行数据可视化。特别是掌握了如何利用 Circle 函数和 Rectangle 函数绘制迷宫中的节点和边，以及如何使用 ion()函数实现动态可视化，逐步呈现 A*算法的搜索过程。

四、实验过程

4.1. 迷宫生成

首先生成了一个随机迷宫。迷宫的大小为 20x20，起点和终点随机选择，障碍物的数量不超过整个网格的 25%。迷宫中的每个节点都可以向上下左右四个方向移动，路径代价通过曼哈顿距离来计算。

4.2. A*算法的实现

实验的核心是实现 A*算法来求解迷宫寻路问题。A*算法结合了广度优先搜索和 Dijkstra 算法的思想，通过启发式函数和代价函数找到最优路径。

g(n)函数表示从起点到当前节点 n 的实际代价，在本实验中，每个节点的移动代价都为 1，因此 g(n)为到当前节点的步数累计。h(n)函数表示从当前节点 n 到终点的估计代价，在本实验中使用曼哈顿距离作为启发式函数，即 h(n) = |x1 - x2| + |y1 - y2|，其中(x1,

y1)为当前节点坐标，(x2, y2)为终点坐标。f(n)函数，A*算法的总代价函数，f(n) = g(n) + h(n)，通过将实际代价与估计代价相加，A*算法可以在搜索过程中保持对路径的优化。

具体实现过程中，算法从起点开始，将起点加入优先队列，并设置初始代价为 0。每次从优先队列中取出代价最小的节点进行扩展，检查它的邻居节点，计算每个邻居的代价。如果发现更优的路径，就更新该节点的代价并将其加入队列。这个过程不断重复，直到找到终点或队列为空。

4.3. 启发式函数的选择和优化

在 A*算法中，启发式函数的选择非常关键。本实验中我选择了曼哈顿距离作为启发式函数，因为迷宫中的移动仅限于上下左右四个方向。此外，为了进一步优化算法，在多个节点代价相同且均为最小时，我选择优先探索距离终点最近的节点。这一策略通过在优先队列中加入启发式值 h(n)作为次级排序条件来实现。

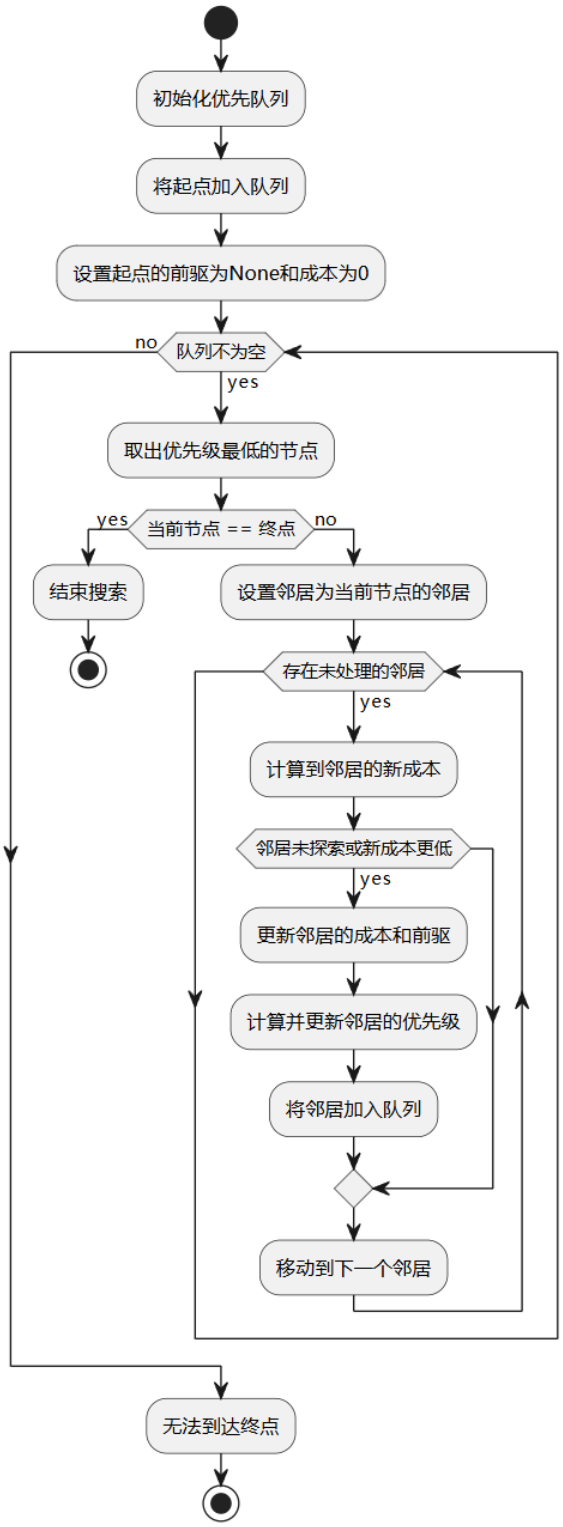
4.4. 可视化搜索过程

在实验中，我使用了 matplotlib 库来实现搜索过程的可视化。通过 ion()函数实现动态绘图，每次算法扩展节点时实时更新迷宫图，使得整个搜索过程更加直观和易于理解。被访问过的节点用蓝色表示，最终找到的路径用黄色表示，起点和终点分别用绿色和紫色标注。

4.5. 代码优化

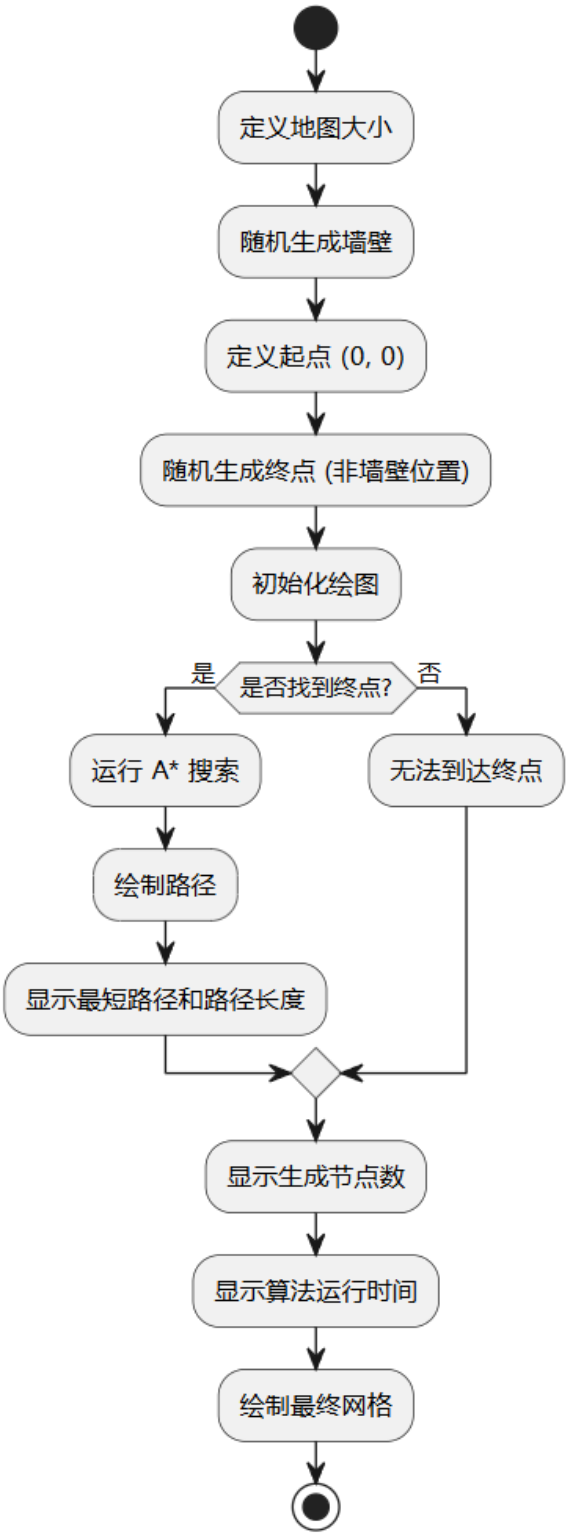
在初始实现的基础上，我对代码进行了优化。**当代价相同且最小时，我进一步优化选择了距离终点最近的节点，以加速搜索过程并减少无效路径的探索。**这种优化有效提高了 A*算法的效率，尤其是在迷宫复杂度较高的情况下。

附 1 算法原理图



附 2 算法框图

（此图详细展示整体程序实现中的每一步逻辑决）



附 3 核心程序清单

```
def a_star_search(start, goal):
    frontier = []  # 创建一个优先队列，存储待探索的节点
    heapq.heappush(frontier, (0, start))  # 将起点加入队列，优先级为 0
```

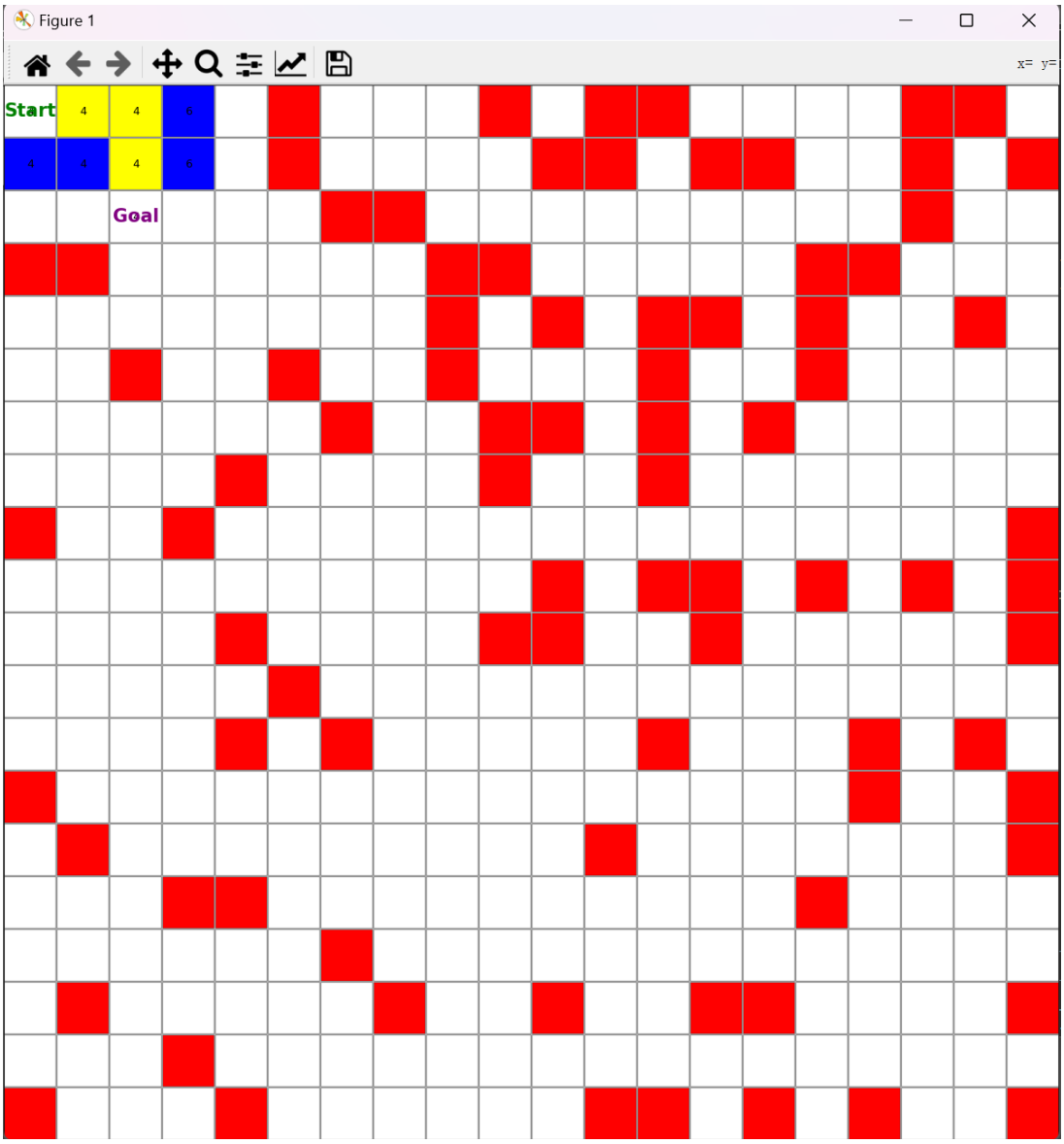
```
    came_from = {}  # 记录路径的字典，键为节点，值为前驱节点
    cost_so_far = {}  # 记录从起点到当前节点的成本
    came_from[start] = None  # 起点的前驱节点为 None
    cost_so_far[start] = 0  # 起点的成本为 0
    while frontier:  # 当队列不为空时，继续搜索
        _, current = heapq.heappop(frontier)  # 取出优先级最低的节点，只需要节点坐标
        if current == goal:  # 如果到达目标节点，结束搜索
            break  # 退出循环
        for neighbor in neighbors(current):  # 遍历当前节点的所有邻居
            new_cost = cost_so_far[current] + 1  # 计算从起点到邻居节点的成本（假设移动成本为 1）
            if neighbor not in cost_so_far or new_cost < cost_so_far[neighbor]:
            # 如果邻居未被探索过或找到更低成本的路径
                cost_so_far[neighbor] = new_cost  # 更新邻居节点的成本
                priority = new_cost + heuristic(goal, neighbor)
                # 计算邻居节点的优先级（成本 + 预估距离）
                heapq.heappush(frontier, (priority, neighbor))  # 将邻居节点加入队列
                came_from[neighbor] = current  # 记录邻居节点的前驱节点
    return came_from, cost_so_far  # 返回路径字典和成本字典
```

文件较多，放在个人远程管理仓库中：https://github.com/Naasi-LF/algorithm/tree/main/Astar

五、实验结果

5.1 示例 1

已知红色为障碍物，蓝色为途径的点，黄色为最佳路径。Start 为初始点，Goal 为终点，方格中的数字为代价函数的值，即 g(x,y)+h(x,y)



运行结果如下：

最短路径：[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)]

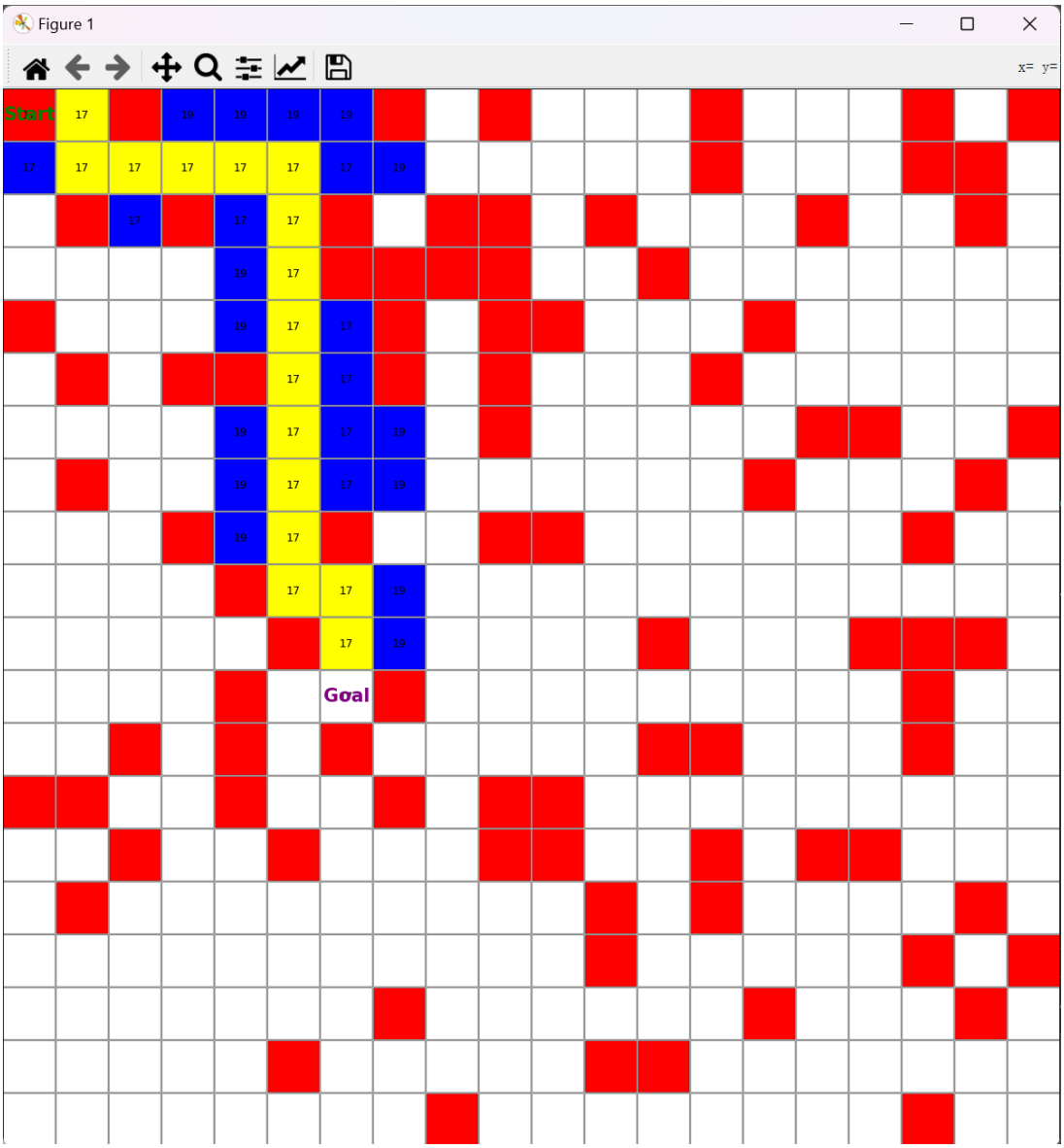
路径长度：5

生成的节点总数：9

算法运行时间：1.620049 秒

***注：**算法运行时间因过快无法看出差别，因此这里同时计算图像交互的时间来让时间的大小更加明显，更好体现算法的时间对比。

5.2 示例二



运行结果如下：

最短路径：

[(0, 0), (0, 1), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5), (8, 5), (9, 5), (9, 6), (10, 6), (11, 6)]

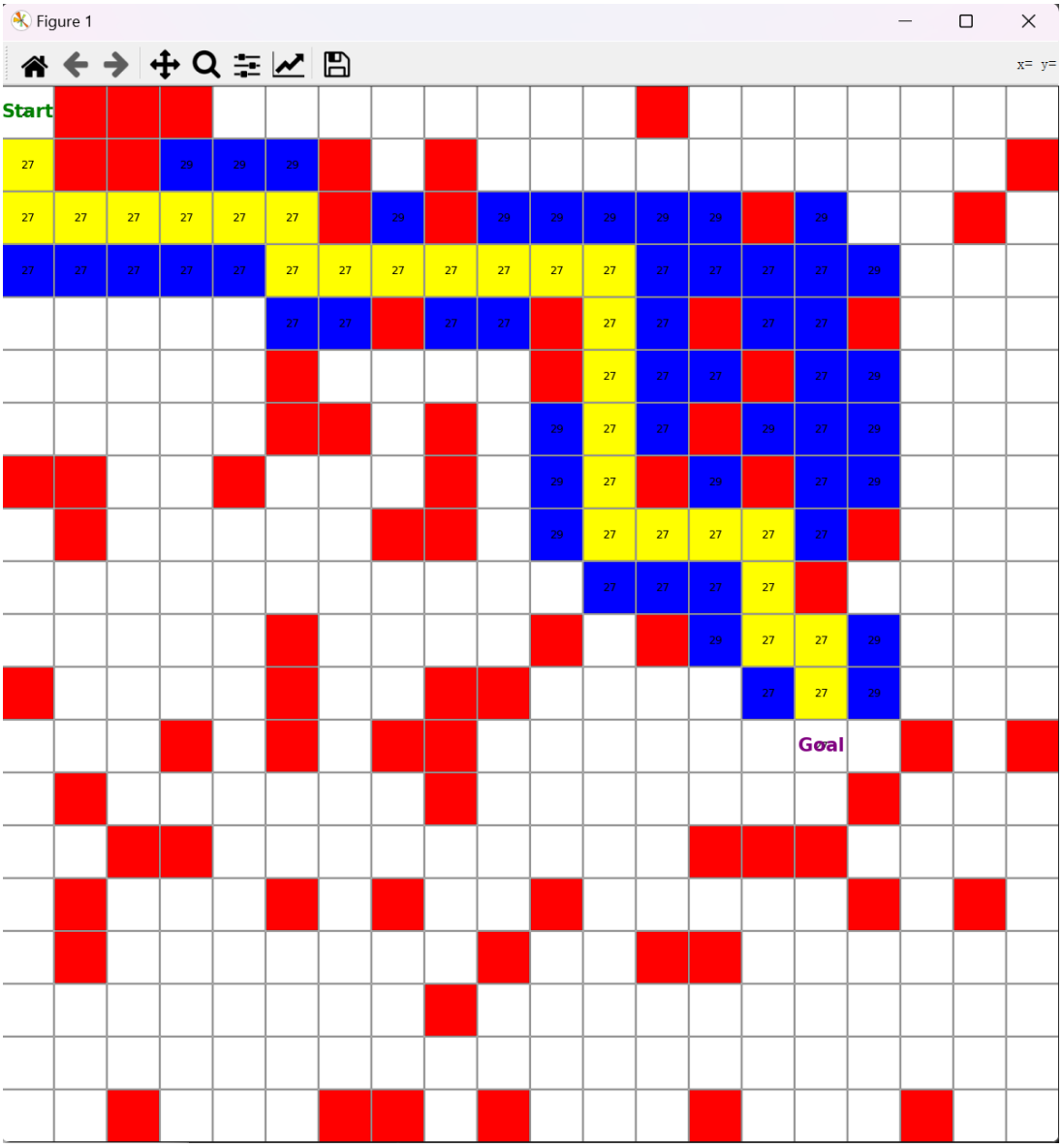
路径长度：18

生成的节点总数：42

算法运行时间：8.069112 秒

***注：**算法运行时间因过快无法看出差别，因此这里同时计算图像交互的时间来让时间的大小更加明显，更好体现算法的时间对比。

5.3 示例三



运行结果如下：

最短路径：

[(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (2, 5), (3, 5), (3, 6), (3, 7), (3, 8), (3, 9), (3, 10), (3, 11), (4, 11), (5, 11), (6, 11), (7, 11), (8, 11), (8, 12), (8, 13), (8, 14), (9, 14), (10, 14), (10, 15), (11, 15), (12, 15)]

路径长度：28

生成的节点总数：80

算法运行时间：15.047731 秒

***注：**算法运行时间因过快无法看出差别，因此这里同时计算图像交互的时间来让时间的大小更加明显，更好体现算法的时间对比。

六、实验结果分析与体会

本实验通过实现 A*算法解决了迷宫寻路问题，A*算法的效率非常高，能够快速找到从

起点到终点的最短路径。在实验中，本人对比了 A*算法与广度优先搜索（BFS）和 Dijkstra 算法的效率，发现 A*算法在同样的搜索条件下比 BFS 和 Dijkstra 算法更快，尤其是在障碍物较多的情况下，其搜索效率优势更为明显。此外，A*算法结合了路径代价与启发式估计，使得其不仅可以找到最优路径，还能减少搜索过程中的不必要扩展。

本人还对 A*算法进行了优化，特别是在多个节点的总代价相同的情况下，优先选择距离终点最近的节点，从而加速了搜索过程。这种优化在复杂迷宫中效果尤为显著，使得算法能够更快速地找到最优路径，减少了无效的节点扩展。总体来说，本实验不仅让我熟悉了 A*算法的实现过程，还体会到了启发式函数和优化策略在提高算法效率方面的重要性。

在实验过程中，通过对启发式函数的调整与优化，使得 A*算法在多个代价相同的节点中优先选择距离终点最近的节点，从而减少了无效的节点扩展，提高了搜索效率。这种优化在复杂迷宫中尤为明显，体现了启发式函数在 A*算法中的重要性。总体来说，本实验使我对 A*算法有了更加深刻的理解，也掌握了如何利用数据可视化工具来分析和展示算法的执行过程。