算法可视化报告

算法可视化实验报告

1. 问题描述和思路

本实验旨在通过可视化方式展示多种经典搜索算法的执行过程。我选择了一个迷宫寻路场景作为实验背景，在这个场景中：

- 起点和终点分别位于迷宫的左上角和右下角
- 黑色方块代表障碍物，白色方块代表可通行区域（如果有障碍物的话）
- 算法需要在避开障碍物的同时，找到从起点到终点的最优路径

直观地比较不同搜索算法的特点：

- DFS会沿着一条路径深入探索，可能找到非最优解
- BFS会逐层扩展，保证找到最短路径
- UCS考虑路径代价，找到代价最小的路径
- A*结合了路径代价和启发式估计，能更高效地找到最优解
- GS只考虑启发式估计，可能找到非最优解
- AC3用于解决约束满足问题
- Minimax用于博弈树搜索
- 剪枝化搜索通过剪枝提高搜索效率

2. 实验过程

2.1 可视化框架设计

我先考虑使用Python的Pygame库构建可视化界面，主要包含以下组件：

1. 网格系统：将迷宫划分为n*m的网格
2. 节点类：每个网格单元都是一个节点，包含位置、颜色、邻居等信息
3. 动画系统：实时显示算法的搜索过程

但是在后续实现的过程中，发现pygame和matpolit这类实时生成的动画可能会在一些场景下降低算法的效率（比如一些很深的递归，DFS）
虽然也可以使用迭代的方法去避免这个问题，但是后续的算法大部分还是使用了GIF图像来

存储每个算法的路径（使用了imageio)
比如这个：

```
def draw_graph(domains, step, removed_arc=None):
    G = nx.Graph()
    G.add_edges_from(edges)
    pos = nx.spring_layout(G, seed=42)

    color_map = {'R': 'red', 'G': 'green', 'B': 'blue'}

    fig, ax = plt.subplots()
    plt.title(f'Step {step}', fontsize=14)

    # 节点显示 domain
    labels = {node: ','.join(domains[node]) for node in G.nodes()}
    nx.draw_networkx_labels(G, pos, labels, font_size=10)

    # 节点颜色为灰（固定）
    nx.draw(G, pos, with_labels=False, node_size=1200, node_color="lightgray", ax=ax)

    # 高亮当前处理的弧
    if removed_arc:
        nx.draw_networkx_edges(G, pos, edgelist=[removed_arc], width=2.5,
edge_color="red")

    filename = f"ac3_step_{step}.png"
    plt.savefig(filename)
    plt.close()
    return filename
```

但是这个时候还是一堆静态图像，然后使用以下函数生成GIF:

```
def make_gif(frames, output='ac3_visual.gif'):
    images = [imageio.imread(frame) for frame in frames]
    imageio.mimsave(output, images, fps=1)
    for frame in frames:
        os.remove(frame)
```

2.2 算法实现

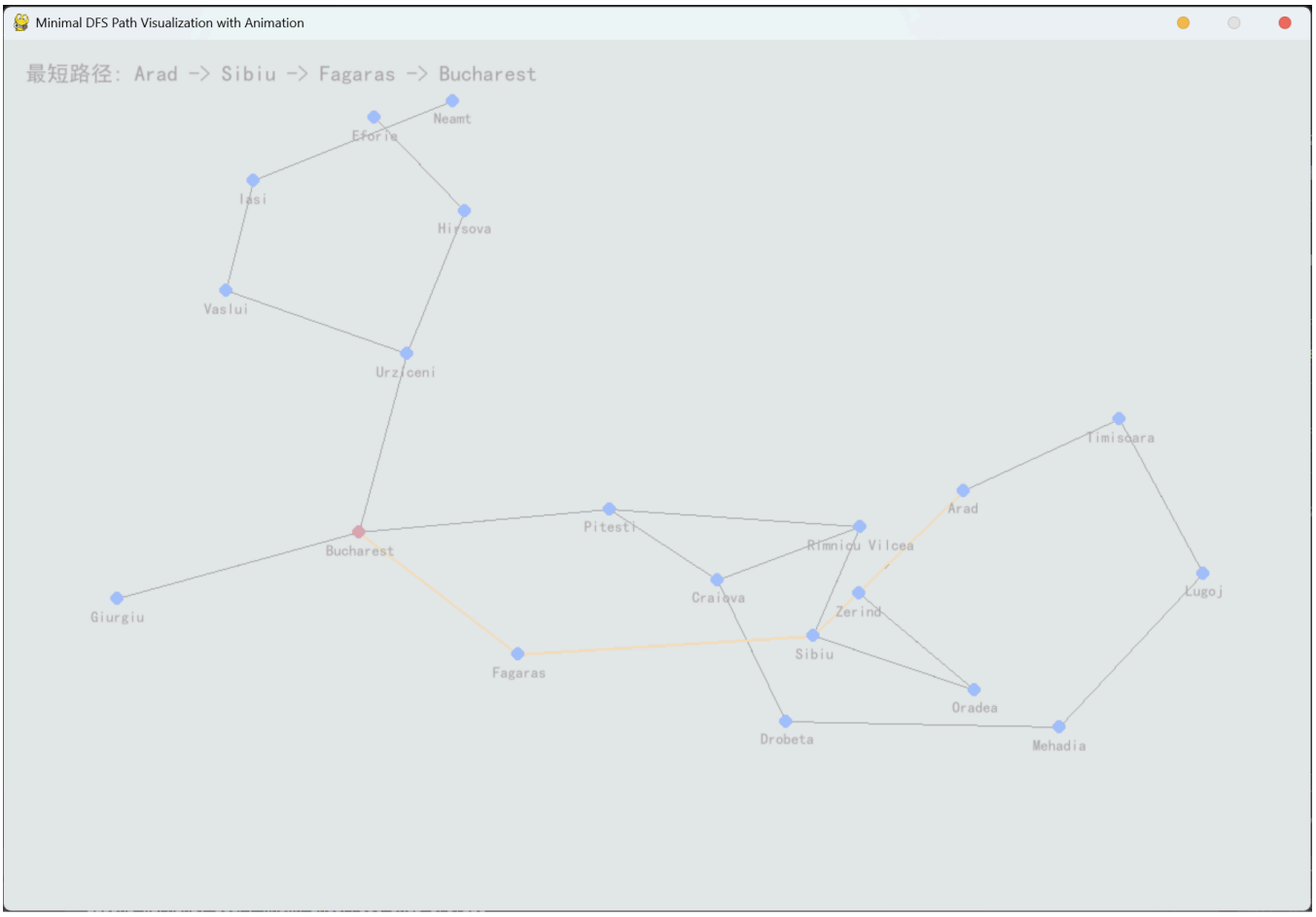
2.2.1 深度优先搜索(DFS)

递归实现

DFS的实现思路如下:

1. 数据结构设计:
   - 使用递归栈来实现深度优先的遍历顺序
   - 使用visited集合记录已访问节点
   - 使用邻接表或邻接矩阵表示图结构
2. 核心步骤:
   - 从起点开始,将当前节点标记为已访问
   - 递归访问当前节点的未访问邻居节点
   - 当无法继续访问时,回溯到上一个节点
   - 直到所有可达节点都被访问过

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(f"访问节点: {start}")
    for next_node in graph[start] - visited:
        dfs(graph, next_node, visited)
    return visited
```



2.2.2 广度优先搜索(BFS)

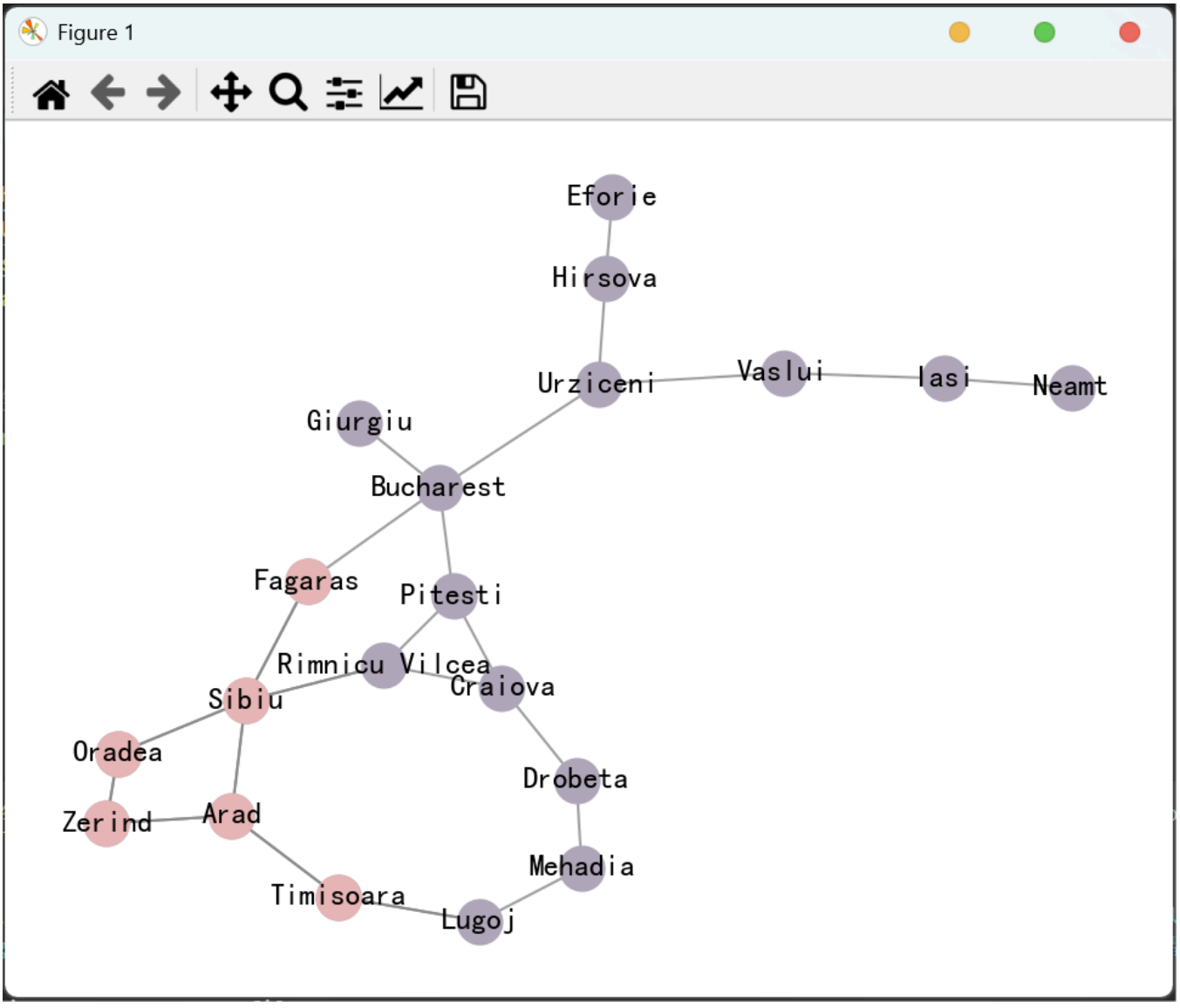
BFS的实现思路如下:

1. 数据结构设计:
   - 使用队列(Queue)来实现广度优先的遍历顺序
   - 使用visited集合记录已访问节点
   - 使用邻接表或邻接矩阵表示图结构
2. 核心步骤:
   - 将起点加入队列并标记为已访问
   - 从队列中取出一个节点,访问其所有未访问的邻居节点
   - 将未访问的邻居节点加入队列并标记为已访问
   - 重复上述过程直到队列为空
3. 实现特点:
   - 按层次顺序遍历,先访问距离起点近的节点
   - 能找到最短路径(边权重相同时)
   - 空间复杂度较高,需要存储整个队列

```
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(f"访问节点: {vertex}")
        for next_node in graph[vertex] - visited:
            visited.add(next_node)
            queue.append(next_node)
```



2.2.3 一致代价搜索(UCS)

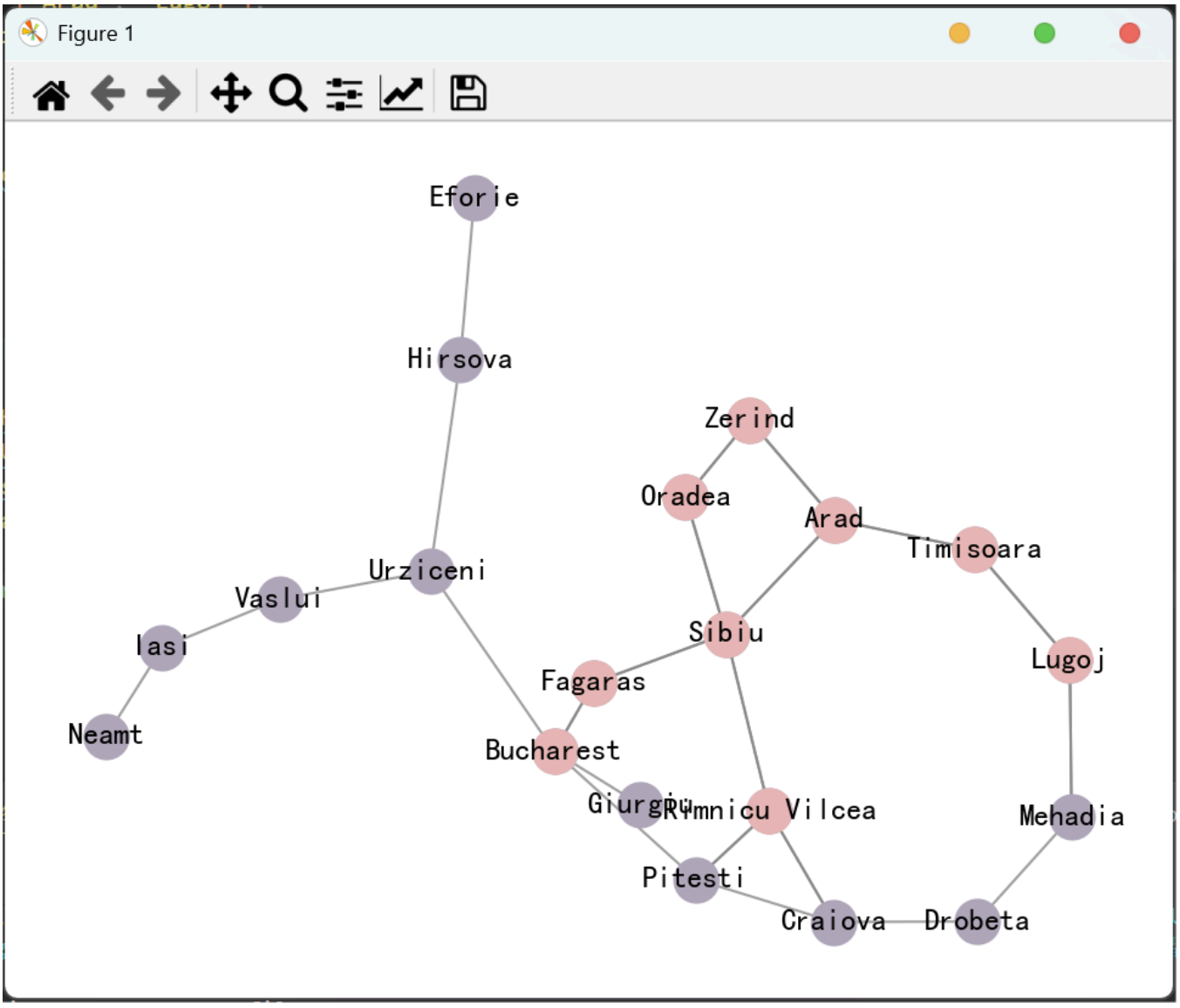
UCS的实现思路如下:

1. 数据结构设计:
   - 使用优先队列(PriorityQueue)存储待访问节点
   - 使用字典记录到达每个节点的最小代价
   - 使用字典记录最优路径的前驱节点
2. 核心步骤:
   - 将起点加入优先队列,初始代价为0
   - 每次从队列中取出代价最小的节点访问
   - 更新其邻居节点的代价,如果找到更小代价则更新
   - 将更新后的邻居节点加入优先队列
   - 重复上述过程直到找到目标节点
3. 实现特点:
   - 按照路径代价顺序进行搜索
   - 保证找到最小代价路径
   - 适用于带权图的最短路径搜索
   - 时间复杂度取决于优先队列的实现

```
import heapq

def ucs(graph, start, goal):
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {start: None}
    cost_so_far = {start: 0}

    while frontier:
        current_cost, current = heapq.heappop(frontier)
        if current == goal:
            break
        for next_node, cost in graph[current].items():
            new_cost = current_cost + cost
            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                heapq.heappush(frontier, (new_cost, next_node))
                came_from[next_node] = current
```



]]

2.2.4 A星搜索

A*搜索算法的实现思路如下:

1. 数据结构设计:
   - 使用优先队列存储待访问节点
   - 使用字典记录g值(从起点到当前节点的实际代价)
   - 使用字典记录f值(g值加上启发式估计值h)
   - 使用字典记录路径的前驱节点
2. 核心步骤:
   - 将起点加入优先队列,初始f值为启发式估计值
   - 每次从队列中取出f值最小的节点访问
   - 如果是目标节点则搜索结束
   - 否则扩展当前节点的所有邻居节点

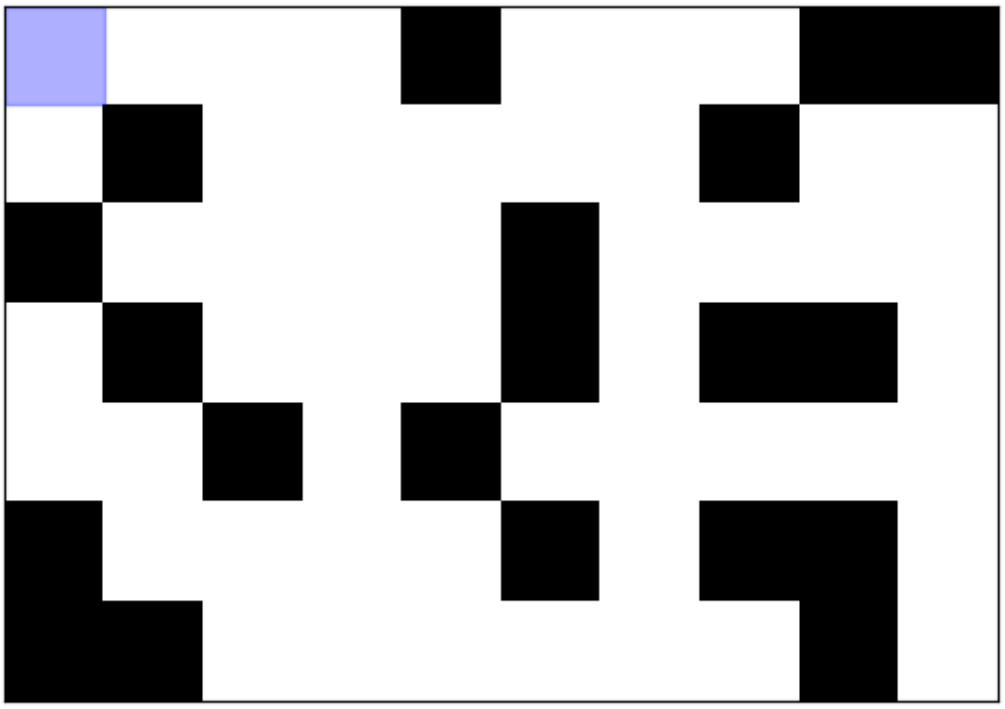
- 计算邻居节点的g值和f值
- 如果找到更优路径则更新相关值
- 将邻居节点加入优先队列
- 重复上述过程直到找到目标或队列为空

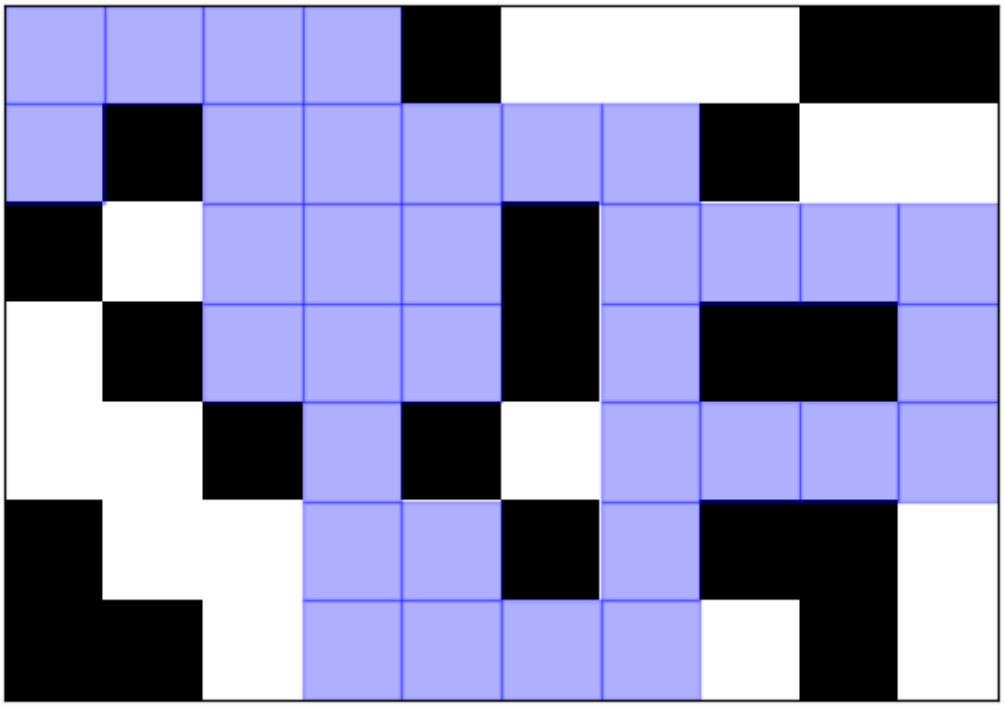
3. 实现特点:
   - 结合了最佳优先搜索和启发式搜索
   - 启发式函数需要满足可采纳性
   - 保证找到最优解
   - 搜索效率高于普通的广度优先搜索
   - 时间复杂度取决于启发式函数的选择
4. 启发式函数设计:
   - 常用曼哈顿距离或欧几里得距离
   - 需要保证不会高估实际代价
   - 启发式函数越接近实际代价,搜索效率越高
   - 可以根据具体问题设计特定的启发式函数

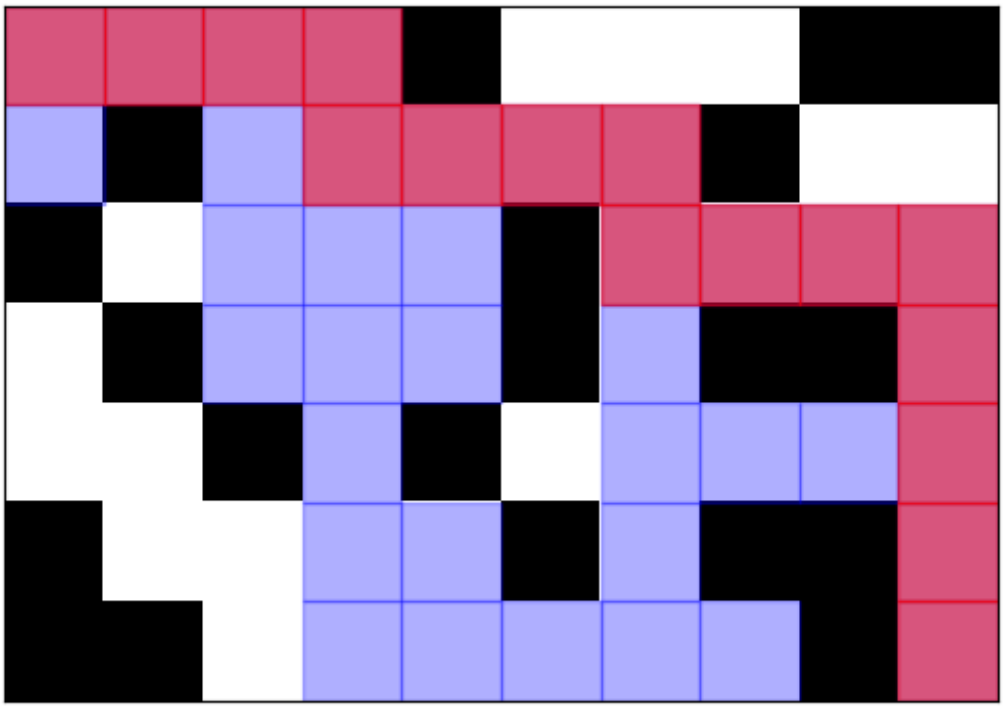
```
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star_search(graph, start, goal):
    frontier = []
    heapq.heappush(frontier, (0, start))
    came_from = {start: None}
    cost_so_far = {start: 0}

    while frontier:
        current = heapq.heappop(frontier)[1]
        if current == goal:
            break
        for next_node in graph[current]:
            new_cost = cost_so_far[current] + graph[current][next_node]
            if next_node not in cost_so_far or new_cost < cost_so_far[next_node]:
                cost_so_far[next_node] = new_cost
                priority = new_cost + heuristic(goal, next_node)
                heapq.heappush(frontier, (priority, next_node))
                came_from[next_node] = current
```







2.2.5 贪婪搜索(GS)

贪婪搜索(GS)的实现思路如下:

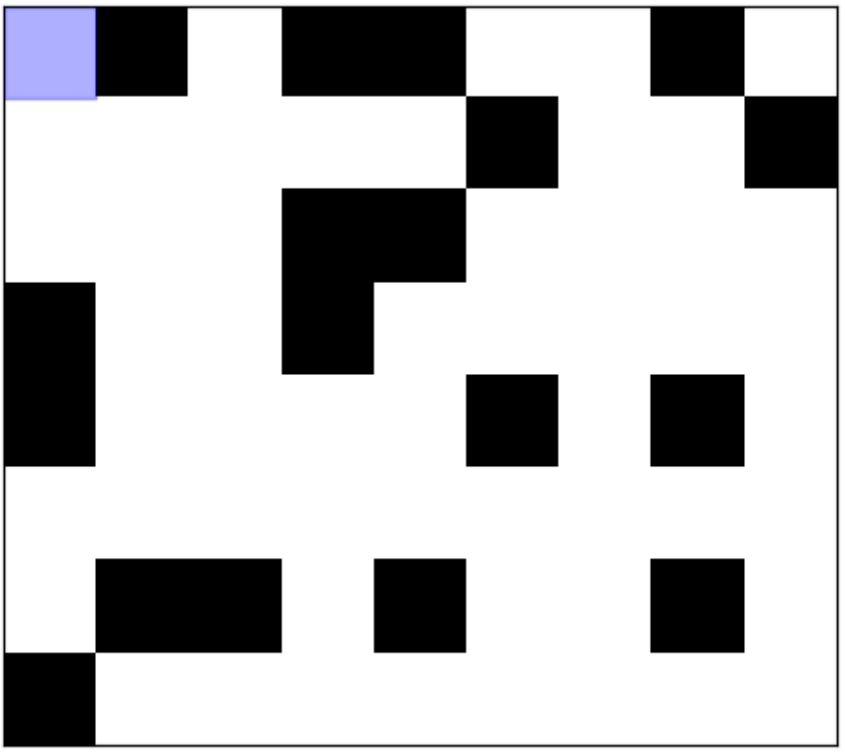
1. 数据结构设计:
   - 使用优先队列存储待访问节点
   - 使用字典记录已访问节点的前驱节点
   - 使用启发式函数估计节点到目标的距离
2. 核心步骤:
   - 将起点加入优先队列,优先级为启发式估计值
   - 每次从队列中取出启发式估计值最小的节点访问
   - 如果是目标节点则搜索结束
   - 否则将当前节点的未访问邻居节点加入队列
   - 邻居节点的优先级为其启发式估计值
   - 重复上述过程直到找到目标或队列为空
3. 实现特点:
   - 只考虑启发式估计,不考虑实际路径代价

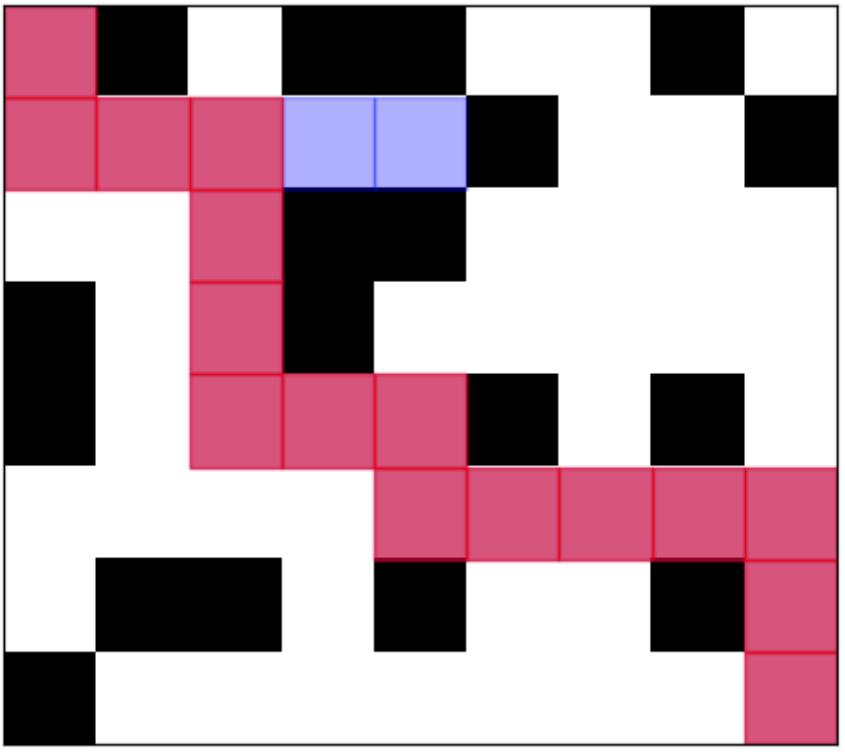
- 搜索速度快但不保证找到最优解
- 容易陷入局部最优
- 适用于对解的质量要求不高但要求快速得到解的场景
- 时间复杂度主要取决于启发式函数

4. 启发式函数设计:
   - 常用曼哈顿距离或欧几里得距离
   - 启发式函数的设计直接影响算法效果
   - 可以根据具体问题特点设计专门的启发式函数
   - 不需要满足可采纳性要求

```
def greedy_search(graph, start, goal, heuristic):
    frontier = []
    heapq.heappush(frontier, (heuristic(start, goal), start))
    came_from = {start: None}

    while frontier:
        current = heapq.heappop(frontier)[1]
        if current == goal:
            break
        for next_node in graph[current]:
            if next_node not in came_from:
                priority = heuristic(next_node, goal)
                heapq.heappush(frontier, (priority, next_node))
                came_from[next_node] = current
```





2.2.6 弧相容算法(AC3)

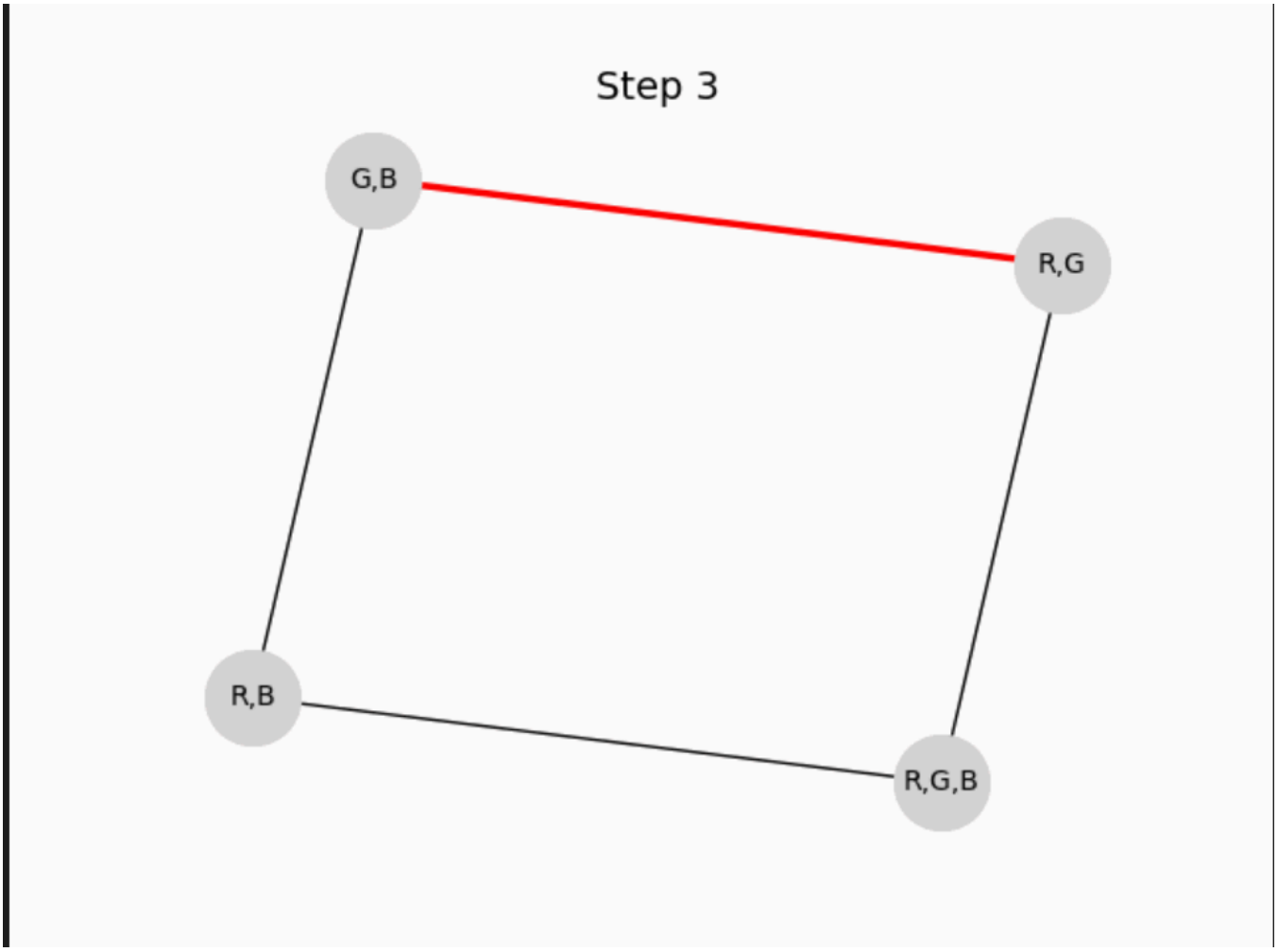
弧相容算法(AC3)的实现思路如下:

1. 数据结构设计:
   - 使用队列存储待检查的弧(变量对)
   - 使用字典存储每个变量的取值域
   - 使用邻接表表示变量之间的约束关系
2. 核心步骤:
   - 初始化队列,加入所有相邻变量对形成的弧
   - 每次从队列中取出一条弧(Xi,Xj)
   - 检查Xi的取值域是否需要修改以满足与Xj的约束
   - 如果Xi的取值域被修改,将所有与Xi相关的弧重新加入队列
   - 重复上述过程直到队列为空或某个变量的取值域为空
3. 实现特点:
   - 通过消除不满足约束的取值来缩小问题规模
   - 可以作为其他搜索算法的预处理步骤
   - 不一定能完全解决问题,但可以大幅减少搜索空间
   - 时间复杂度为$O(ed^3)$,其中e为弧数,d为最大取值域大小
   - 空间复杂度为O(e)用于存储队列
4. 约束检查:
   - revise函数检查一条弧是否满足约束
   - 对Xi的每个取值x,检查是否存在Xj的取值y使约束成立
   - 如果不存在则从Xi的取值域中删除x
   - 返回是否修改了Xi的取值域

```
def ac3(csp):
    queue = [(Xi, Xk) for Xi in csp.variables for Xk in csp.neighbors[Xi]]
    while queue:
        (Xi, Xj) = queue.pop()
        if revise(csp, Xi, Xj):
            if len(csp.domains[Xi]) == 0:
                return False
            for Xk in csp.neighbors[Xi]:
                if Xk != Xj:
                    queue.append((Xk, Xi))
```

```
    return True

def revise(csp, Xi, Xj):
    revised = False
    for x in csp.domains[Xi]:
        if not any(csp.constraints(Xi, x, Xj, y) for y in csp.domains[Xj]):
            csp.domains[Xi].remove(x)
            revised = True
    return revised
```



2.2.7 极小化极大算法(Minimax)

极小化极大算法(Minimax)的实现思路如下:

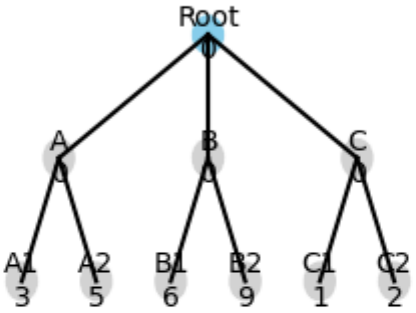
1. 数据结构设计:
   - 使用树结构表示博弈状态
   - 每个节点包含当前状态评估值
   - 使用递归实现极大极小层的交替
2. 核心步骤:

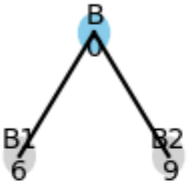
- 从根节点开始,交替执行极大层和极小层
- 极大层选择子节点中的最大值
- 极小层选择子节点中的最小值
- 到达叶子节点或指定深度时返回评估值
- 逐层向上传播选择的值

3. 实现特点:
   - 假设对手总是采取最优策略
   - 通过递归实现决策树的遍历
   - 适用于双人零和博弈
   - 时间复杂度为O(b^m),其中b为分支因子,m为最大深度
   - 可以通过Alpha-Beta剪枝优化性能
4. 评估函数:
   - 对叶子节点进行评估
   - 评估值反映当前状态对MAX方的有利程度
   - 评估函数的设计直接影响算法效果
   - 常用特征包括:位置、数量、机动性等

```
def minimax(node, depth, maximizing_player):
    if depth == 0 or node.is_terminal():
        return node.evaluate()

    if maximizing_player:
        value = float('-inf')
        for child in node.get_children():
            value = max(value, minimax(child, depth - 1, False))
        return value
    else:
        value = float('inf')
        for child in node.get_children():
            value = min(value, minimax(child, depth - 1, True))
        return value
```


Root
0
A
0
B
0
C
0
A1
3
A2
5
B1
6
B2
9
C1
1
C2
2


B
0
B1
6
B2
9

2.2.8 剪枝化搜索

剪枝化搜索的实现思路如下:

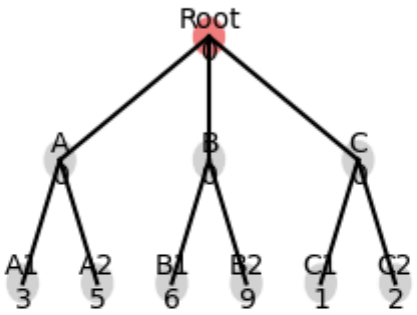
1. 数据结构设计:
   - 使用Alpha-Beta值记录搜索边界
   - 使用递归实现极大极小层的交替
   - 维护当前最优解的上下界
2. 核心步骤:
   - 从根节点开始,交替执行极大层和极小层
   - 维护Alpha值(已知的最大值)和Beta值(已知的最小值)
   - 当Beta<=Alpha时剪枝,不再搜索当前节点的其他子节点
   - 极大层更新Alpha值,极小层更新Beta值
   - 到达叶子节点或指定深度时返回评估值
3. 实现特点:
   - 在Minimax基础上增加剪枝操作
   - 不影响最终结果的正确性

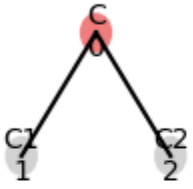
- 显著减少搜索空间
- 时间复杂度最优情况可降至O(b^(m/2))
- 剪枝效果与节点访问顺序相关

4. 优化策略:
   - 优先搜索可能导致剪枝的节点
   - 根据历史信息对节点排序
   - 迭代加深搜索
   - 结合置换表缓存中间结果
   - 动态调整搜索深度

```
def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or node.is_terminal():
        return node.evaluate()

    if maximizing_player:
        value = float('-inf')
        for child in node.get_children():
            value = max(value, alpha_beta_pruning(child, depth - 1, alpha, beta, False))
            alpha = max(alpha, value)
            if beta <= alpha:
                break
        return value
    else:
        value = float('inf')
        for child in node.get_children():
            value = min(value, alpha_beta_pruning(child, depth - 1, alpha, beta, True))
            beta = min(beta, value)
            if beta <= alpha:
                break
        return value
```


Root
0
A
0
B
0
C
0
A1
3
A2
5
B1
6
B2
9
C1
1
C2
2


C
0
C1
1
C2
2

3. 实验结果

3.1 算法特点分析

1. DFS：
   - 优点：实现简单，内存占用小
   - 缺点：可能找到非最优解，在复杂迷宫中效率低
   - 适用场景：需要快速找到任意解的问题
2. BFS：
   - 优点：保证找到最短路径
   - 缺点：需要存储所有已访问节点，内存占用大
   - 适用场景：需要找到最短路径的问题
3. UCS：
   - 优点：考虑路径代价，找到代价最小的路径
   - 缺点：需要存储所有节点的代价信息
   - 适用场景：需要考虑路径代价的问题
4. A*：

- 优点：结合了路径代价和启发式估计，效率高
- 缺点：启发函数的选择影响算法性能
- 适用场景：需要高效找到最优解的问题

5. GS：
   - 优点：实现简单，运行速度快
   - 缺点：可能找到非最优解
   - 适用场景：需要快速找到可行解的问题
6. AC3：
   - 优点：能有效减少搜索空间
   - 缺点：实现复杂，运行时间较长
   - 适用场景：约束满足问题
7. Minimax：
   - 优点：能找到最优策略
   - 缺点：搜索空间大，运行时间长
   - 适用场景：博弈问题
8. 剪枝化搜索：
   - 优点：通过剪枝减少搜索空间
   - 缺点：剪枝条件的选择影响算法性能
   - 适用场景：需要优化搜索效率的问题

4. 结论

通过本次实验，我实现了多种经典搜索算法的可视化展示，并对其性能进行了分析。可视化展示帮助我更好地理解了这些算法的执行过程和特点。实验结果表明：

1. 不同算法适用于不同场景，需要根据具体问题选择合适的算法
2. 可视化展示能够直观地展示算法的执行过程，有助于理解算法原理