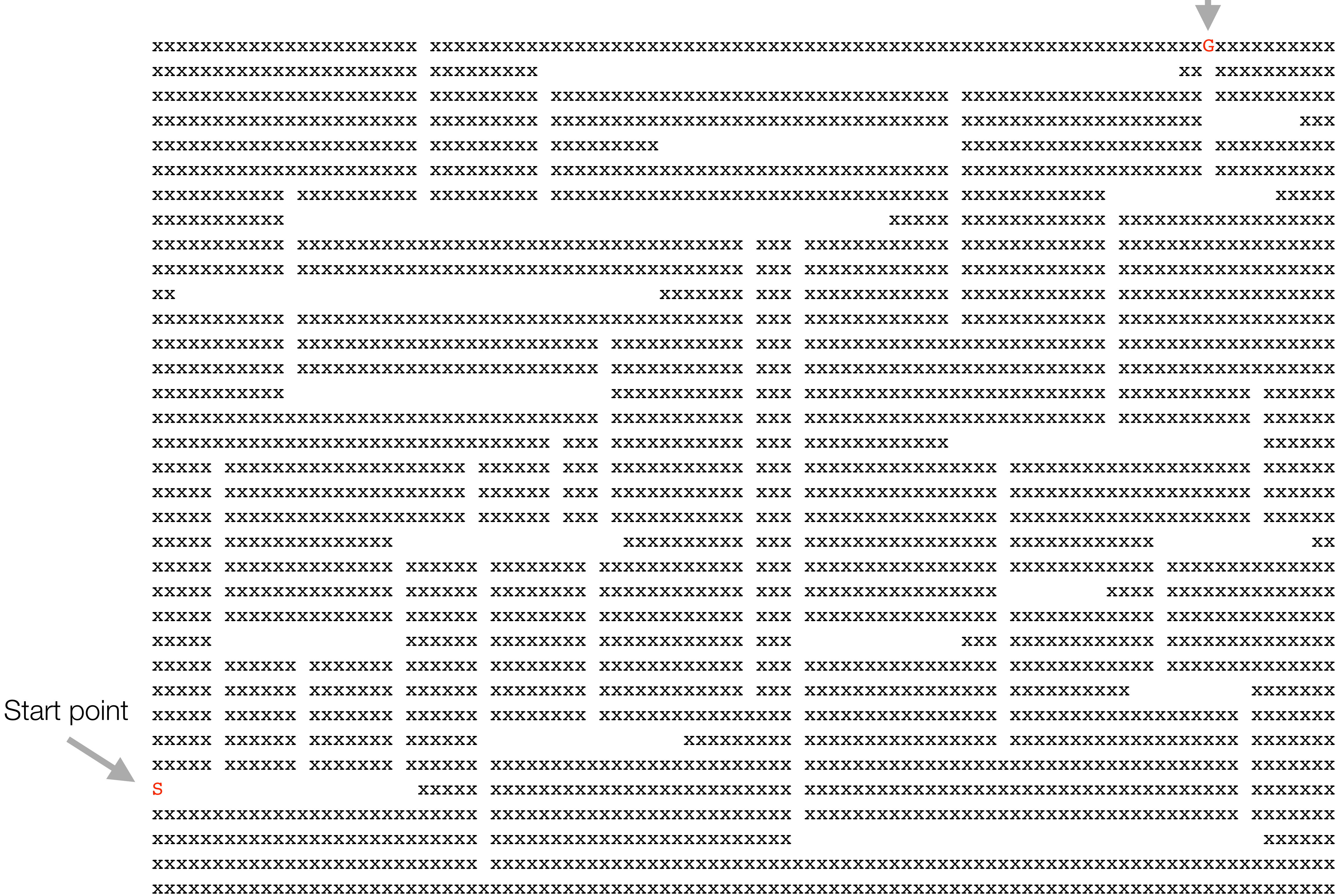
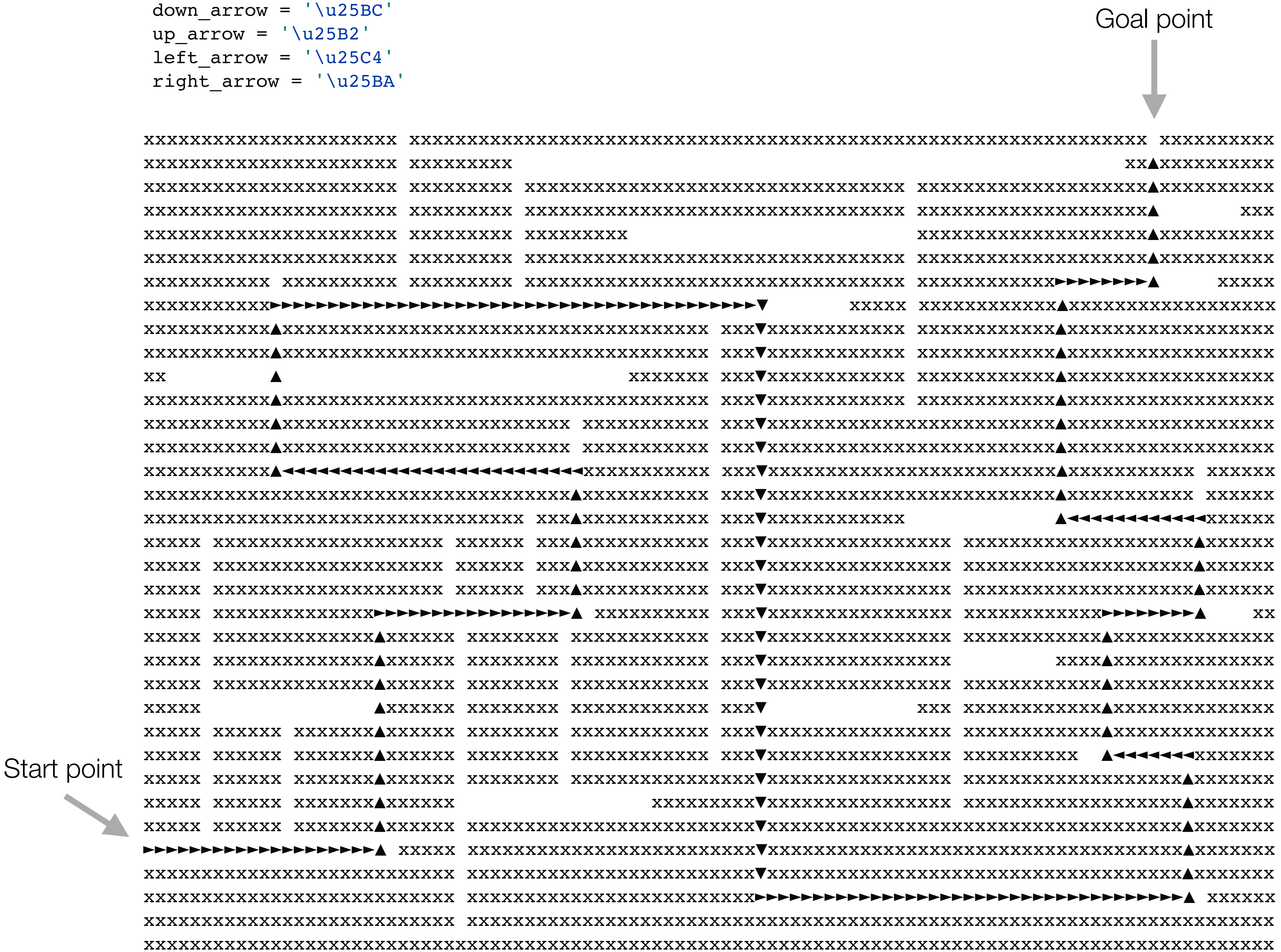
人工智能实验 1 ——迷宫寻路

UESTC · 2021 fall

```python
# 迷宫文件中，x表示墙，空格表示路
blocked_tile = 'X'
free_tile = ' '
```



```python
# 要打印到.txt文件中的Unicode字符（箭头）
down_arrow = '\u25BC'
up_arrow = '\u25B2'
left_arrow = '\u25C4'
right_arrow = '\u25BA'
```



搜索问题

- 一个搜索问题包含：
  - 一个状态空间：当前所处的位置
  - 在每一个状态里，一个可允许的动作集合：NSEW
  - 一个转换模型：移动到下一个位置
  - 一个步骤成本函数：1
  - 一个开始状态，和一个目标到达测试：S，G
- 一个解决方案是一序列动作（一个规划），从开始状态到一个目标状态

```python
if __name__ == '__main__':

    maze, start_location, goal_location = load_maze(maze_file_loc)

    start_time = datetime.now()
    path_trace, Number_Of_iterations = DFS_search(maze, start_location, goal_location)
    time_consumed = datetime.now() - start_time

    if path_trace is not None:
        print('Path = ' + str(path_trace))
        Create_Solution_Map(maze, path_trace)
        print('\nNumber of iterations = ' + str(Number_Of_iterations))
        print('Path length = ' + str(len(path_trace)))
        print('Time consumed = ', time_consumed)
    else:
        print('No path Found - ' + str(Number_Of_iterations))
        input('Press enter')
```

```python
# 深度优先搜索
def DFS_search(maze, start_location, goal_location):
    # 当前位置
    current_location = start_location
    # 访问过的位置
    visited_locations = list()
    # 记录路径，最终会通过这个dict中的记录来生成路径
    traceBack_dict = {'START': current_location}
    # 迭代的次数
    Number_Of_iterations = 0

    # 通过当前位置和已访问过的位置，找出下一步行动的全部可能位置
    next_Locations = Next_State_Generator(maze, current_location, visited_locations)
    # 要用deepcopy，确保新生成一个available_location对象
    available_location = copy.deepcopy(next_Locations)
    visited_locations.append(current_location)

    # 在traceBack_dict中记录，是从current_location到达的locationX
    for locationX in next_Locations:
        traceBack_dict.update({str(locationX): current_location})

    while available_location and current_location != goal_location:...

    if current_location == goal_location:...
    else:
        return None, Number_Of_iterations
```

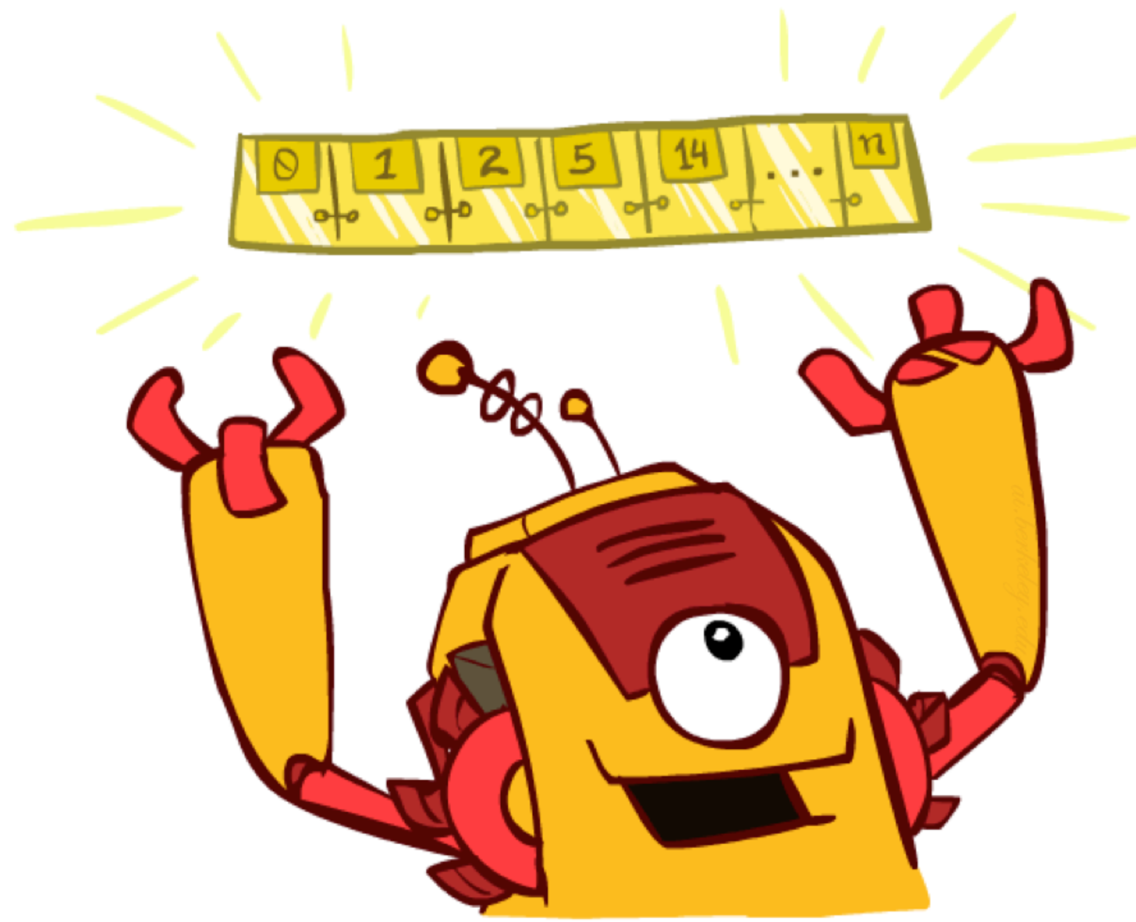
```python
while available_location and current_location != goal_location:
    # current_location移动到可移动的第一个位置
    current_location = available_location[0]
    # 删去已被移动的位置
    available_location = available_location[1:]
    next_Locations = Next_State_Generator(maze, current_location, visited_locations)
    for locationX in next_Locations:
        traceBack_dict.update({str(locationX): current_location})
        # 插入在表头，所以是深度优先。
        available_location.insert(0, locationX)
        # 如果插入在表尾，那就是广度优先。
        # available_location.append(locationX)
    visited_locations.append(current_location)
    Number_Of_iterations = Number_Of_iterations + 1
```

```python
if current_location == goal_location:
    path_trace = list()
    # 通过traceBack_dict反向寻找构建path_trace
    while current_location != start_location:
        path_trace.append(current_location)
        current_location = traceBack_dict[str(current_location)]
    path_trace = path_trace + [start_location]
    path_trace.reverse()
    return path_trace, Number_Of_iterations
else:
    return None, Number_Of_iterations
```

```python
# 生成下一个可能的路径
def Next_State_Generator(maze, current_location, visited_locations):
    max_vertical = len(maze) - 1
    max_horizontal = len(maze[0]) - 1

    allowed_location_1 = None
    allowed_location_2 = None
    allowed_location_3 = None
    allowed_location_4 = None
    next_states = list()

    # UP
    # Check if UP location is inside the maze
    if (current_location[0] - 1 >= 0):
        # Check if it is possible to move UP
        if (maze[current_location[0] - 1][current_location[1]] == free_tile):
            # check if the path was already visited
            if ([current_location[0] - 1, current_location[1]] not in visited_locations):
                allowed_location_1 = [current_location[0] - 1, current_location[1]]
                next_states.append(allowed_location_1)

    # DOWN ...
    # LEFT ...
    # RIGHT ...

    return next_states
```

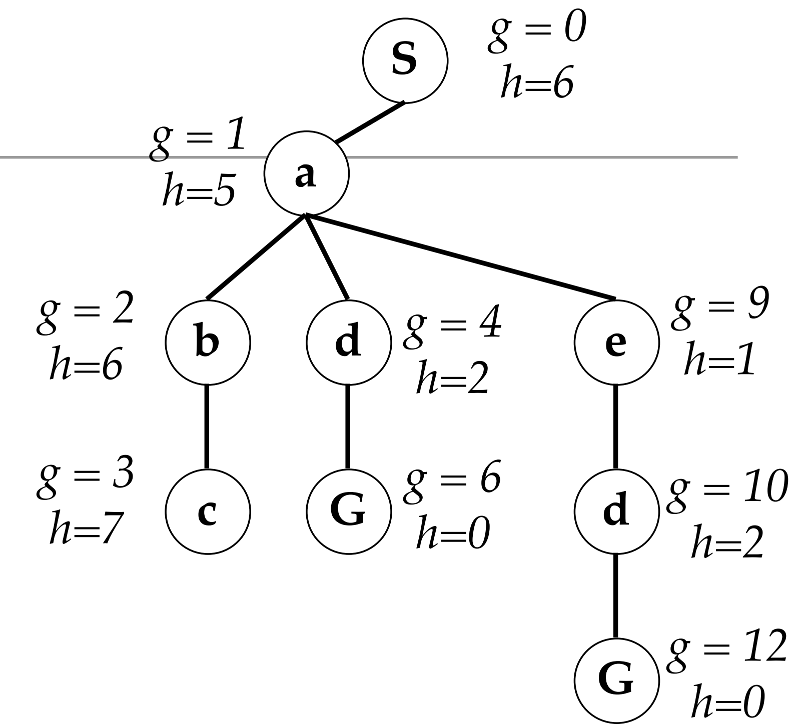
都是一个队列

- 除了边缘策略，所有这些搜索算法是相同的
  - 从概念上讲，所有的边缘都是优先队列(即集合与附加的节点优先级)
  - 实际上,DFS和BFS可以用栈和队列，避免log (n)的排序开销
  - 甚至可以同一套代码来实现，只需要一个参数来确定出队列的方式

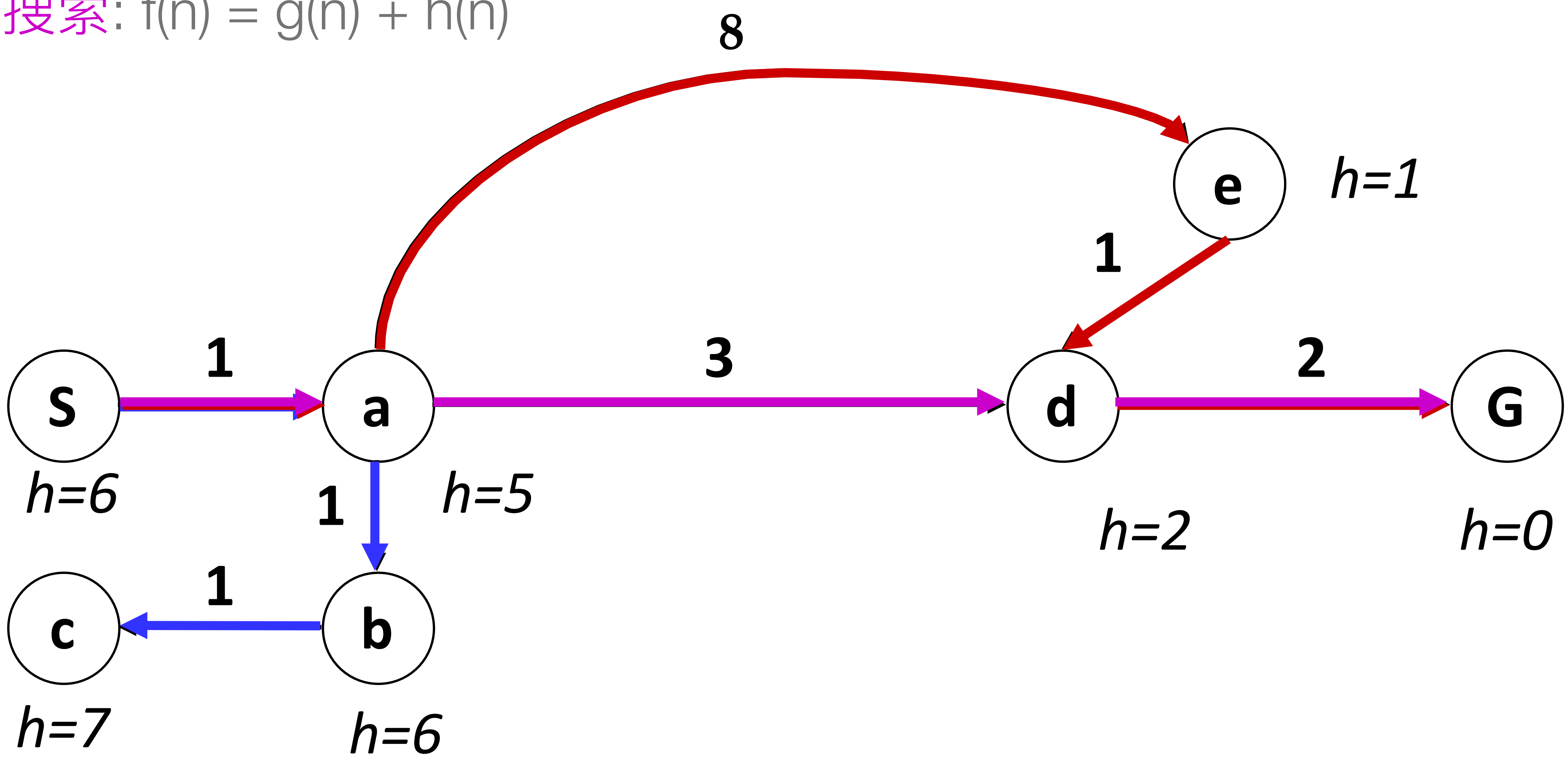


```python
while available_location and current_location != goal_location:
    # current_location移动到可移动的第一个位置
    current_location = available_location[0]
    # 删去已被移动的位置
    available_location = available_location[1:]
    next_Locations = Next_State_Generator(maze, current_location, visited_locations)
    for locationX in next_Locations:
        traceBack_dict.update({str(locationX): current_location})
        # 插入在表头，所以是深度优先。
        available_location.insert(0, locationX)
        # 如果插入在表尾，那就是广度优先。
        # available_location.append(locationX)
        # 如果插入在在其他位置呢?
    visited_locations.append(current_location)
    Number_Of_iterations = Number_Of_iterations + 1
```

结合统一搜索和贪婪搜索

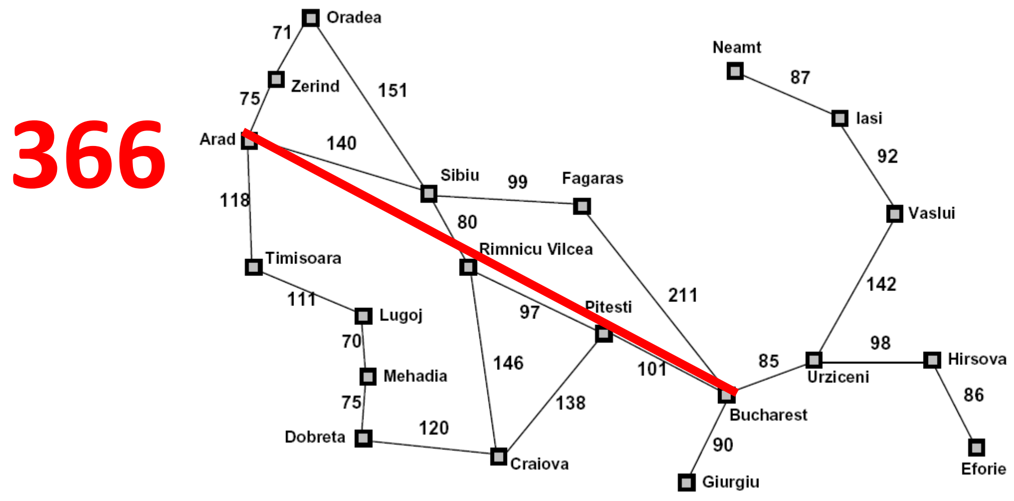


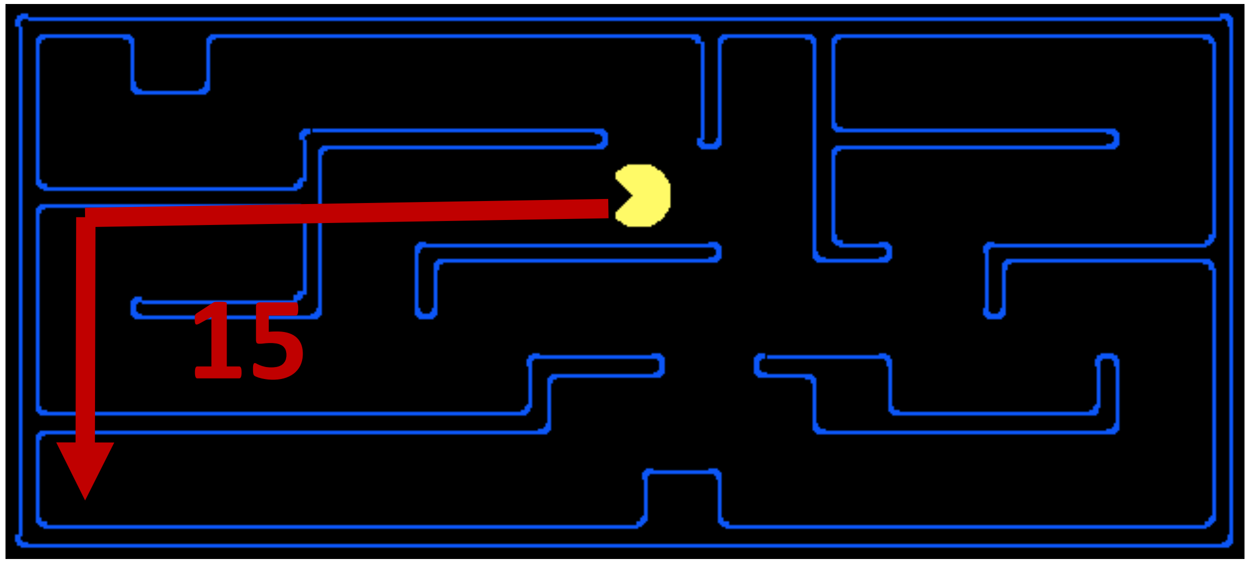
- UCS: 把路径成本排序, 即来程的成本 g(n)
- Greedy: 排序按照与目标的临近性, 即前程成本 h(n)
- A* 搜索: f(n) = g(n) + h(n)



创建可接纳的启发式函数

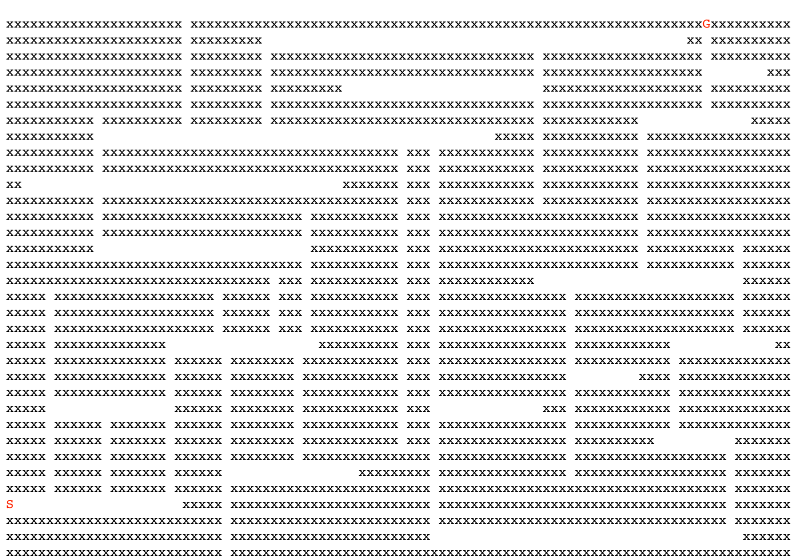
- 在求解很难的搜索问题时，大部分的工作是找到可接纳的启发式函 数。
- 可接纳的启发式函数信息，通常是对应的松弛问题(relaxed problems)的解，解除对行动的限制。



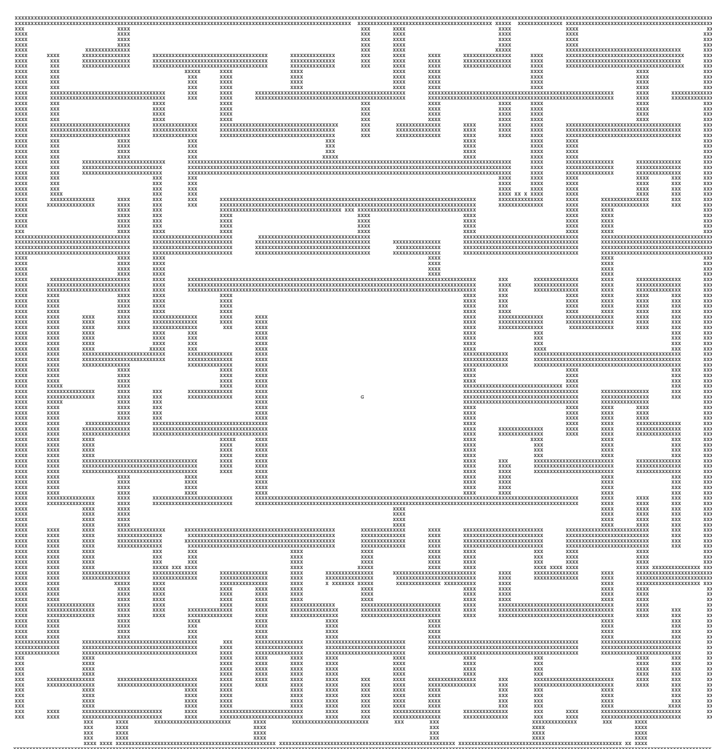


实验要求

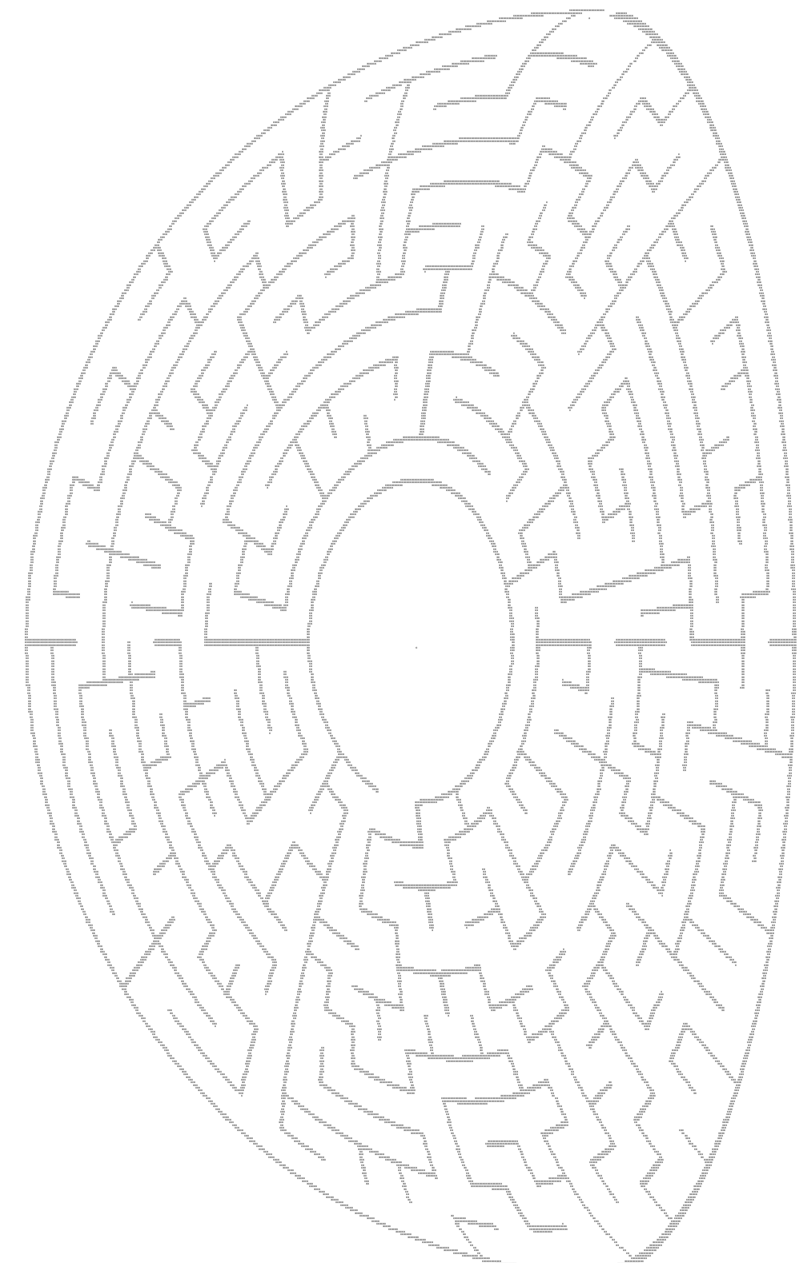
- 用A* 算法实现迷宫的寻路
- 设计合理的启发式函数
- 尽可能快地解决：



Maze



Maze_Hard



Maze_Very_Hard