湖南大學

HUNAN UNIVERSITY

人工智能实验一报告
搜索算法求解问题

目录

一、实验目的

1. 掌握有信息搜索策略的算法思想；
2. 能够编程实现搜索算法；
3. 应用 A*搜索算法求解罗马尼亚问题。

二、实验描述（实验原理）

A*算法是一种高效的启发式搜索算法，通过结合实际代价(g)和估算代价(h)来寻找最优路径。其核心方法是在搜索过程中优先探索总代价 f(n) = g(n) + h(n) 最小的节点，这使得算法能够在保证最优解的同时显著减少搜索空间。

基本思想：

1.启发式评估：A*算法利用一个启发式函数来评估从当前节点到目标节点的估计成本。这个函数的值通常是根据问题的特性来设计的，比如在地图上可能使用直线距离或欧几里得距离。

2.代价估算：每个节点都有一个代价估算值，即从起始节点通过当前节点到达目标节点的总成本，这个总成本由两部分组成：从起始节点到当前节点的实际成本（g 值）和从当前节点到目标节点的估计成本（h 值）。

3.优先队列：A*算法使用一个优先队列来存储待探索的节点，节点根据其代价估算值进行排序，优先队列通常是一个最小堆，以确保每次都能选取代价最低的节点进行扩展。

4.路径重建：当算法找到目标节点时，通过回溯每个节点的父节点来重建路径。

5.算法终止：当目标节点被添加到优先队列并被探索时，算法终止，并输出从起始节点到目标节点的最短路径。

6.效率：A*算法在许多情况下比纯粹的深度优先或广度优先搜索更高效，因为它利用启发式信息来引导搜索过程，减少需要探索的节点数量。

7.完备性与最优性：如果启发式函数是可接受的（即它不会高估到目标的实际代价），A 算法是完备的，即总能找到解决方案（如果存在的话）。如果启发式函数也是一致的（即任何时候都满足三角不等式），A 算法能保证找到最优解。

数据集处理：

使用罗马尼亚问题的书中描述：

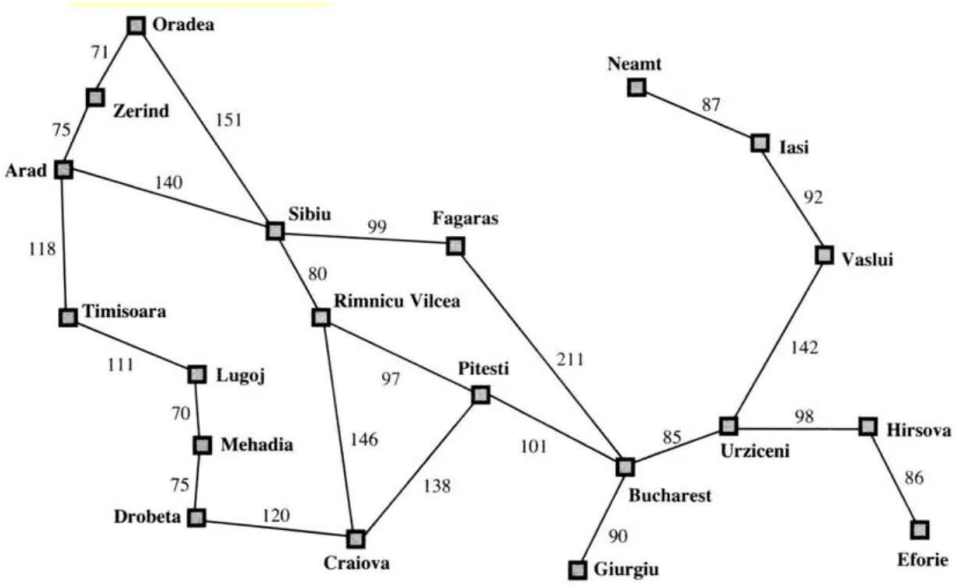


图 3.2　部分罗马尼亚地图的简化版

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

图 3.22　h_{SLD}的值——到 Bucharest 的直线距离

算法的基本思想是维护一个优先队列（frontier），用于存储待探索的节点，这些节点按照总代价（f 值）进行排序。初始时，将起点节点加入优先队列。在搜索过程中，每次从优先队列中取出 f 值最小的节点（current_node），检查是否达到目标节点。如果是，则通过回溯父节点（parent）来重建路径。如果不是，将当前节点的邻居节点加入优先队列，同时更新到达这些邻居节点的代价（cost_so_far）和父节点（came_from）。

在罗马尼亚问题中，A*算法在搜索过程中使用f(n) = g(n) + h(n)来引导搜索方向，h(n)由直线距离确定，这使得它在许多情况下比纯粹的深度优先或广度优先搜索更高效，也具有完备性和最优性。

三、实验及结果分析

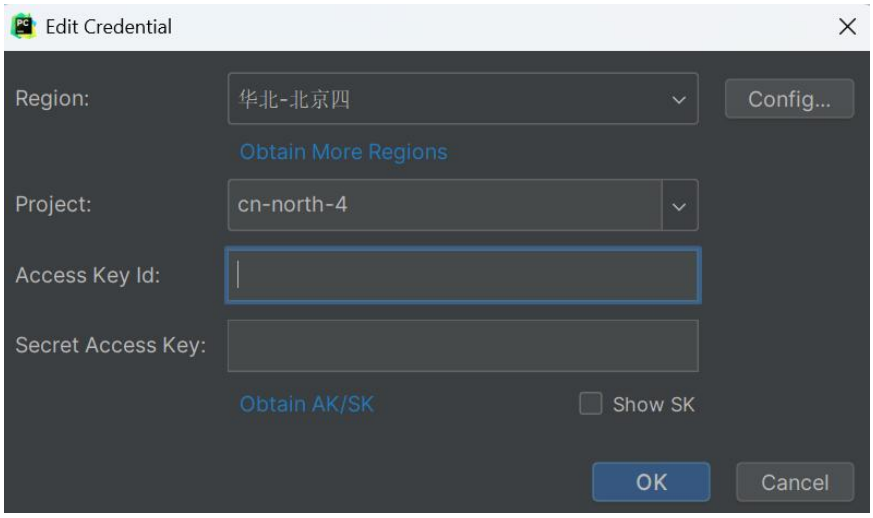
（1）开发语言及运行环境

语言：python。

运行环境：使用pycharm的ssh远程连接（需要使用pycharm专业版2019.2-2023.2之间，我下载的版本是2023.1.6）。

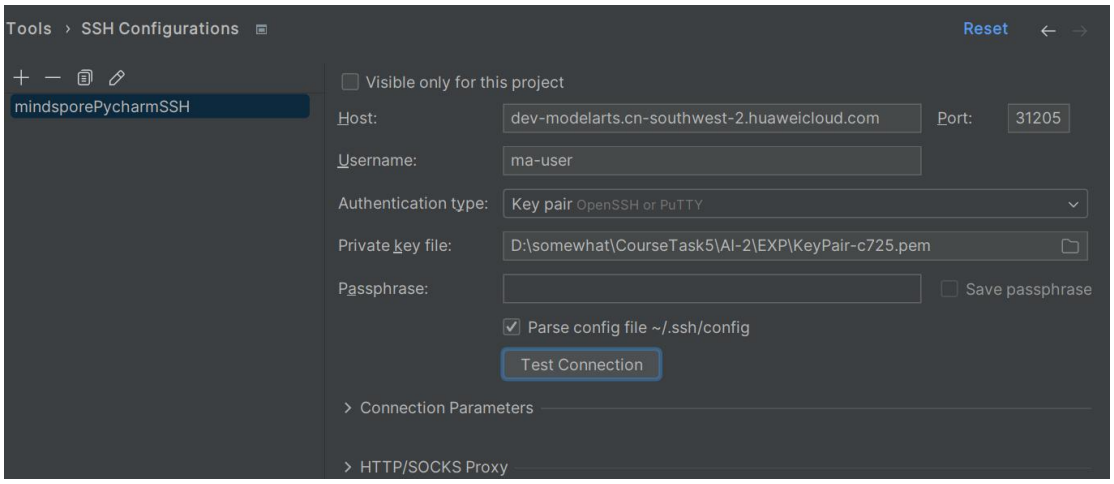
本来想使用 ModelArts 插件，但发现可选地区没有控制台所在地区“西南-贵阳一”，点击获取地区的 gitee 网址中也没有，同时发现控制台显示的 notebook 接入中没有 pycharm 的选项，只有 vscode 的，有可能是太新了还没有被覆盖。于是按照使用 PyCharm 手动连接 Notebook_AI 开发平台 ModelArts_华为云(huaweicloud.com)该文档手动配置 ssh 连接。

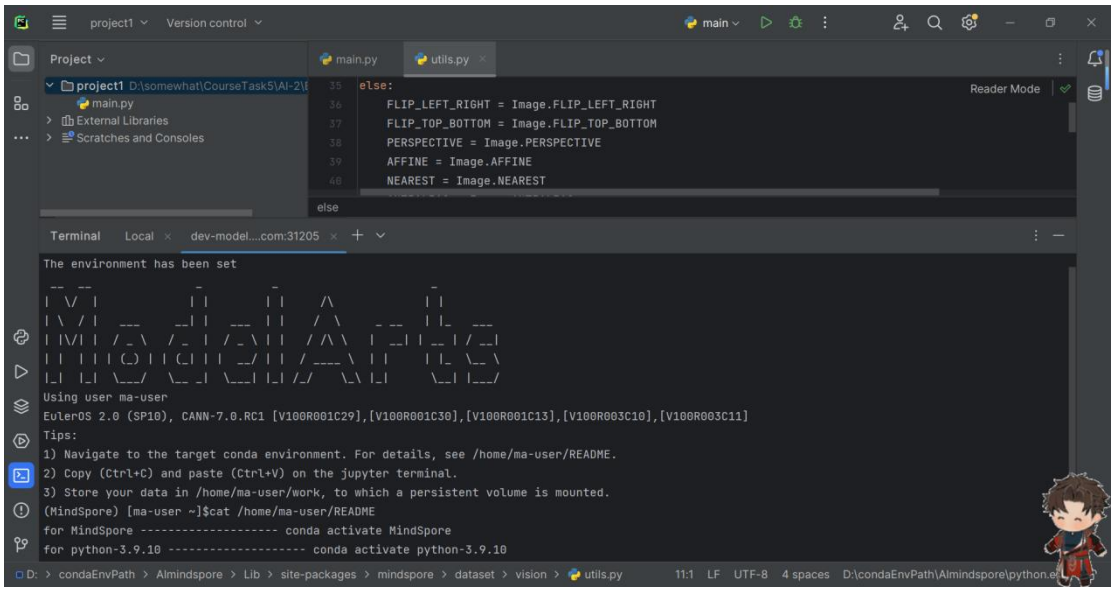
ModelArts:



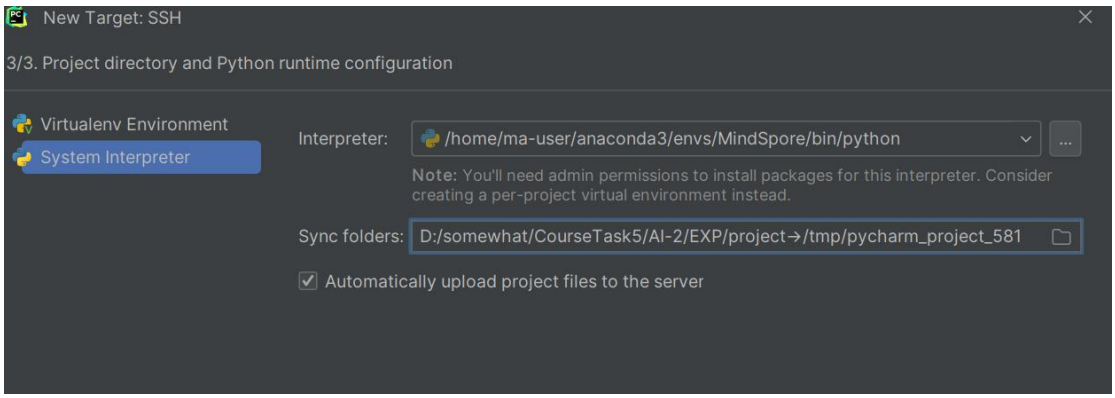
Obtain More Regions/gitee:

SSH 配置：





配置云上 Python Interpreter（这个配起来有点怪，我的版本显示的界面和教程并不完全一样，而且最开始配的时候经常出错（我怀疑是当时控制台那边有点卡），还是使用它教程给的存放路径会好一点，使用的 system Interpreter）：



配置完成后本地文件夹中的改变会自动在远程项目文件夹下更新，而环境使用的是连接ssh后使用which python得到的环境，即，使用的是远程环境。

环境：/home/ma-user/anaconda3/envs/MindSpore/bin/python

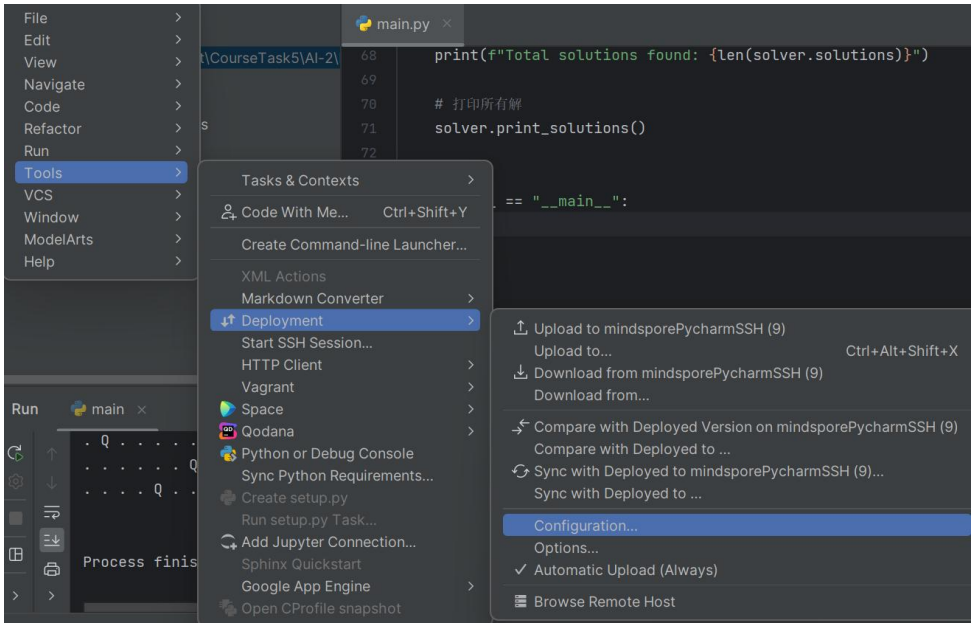
路径：/home/ma-user/work/project[n]

注意：每创建一个新project都要重新配一下云上Python Interpreter这一步（SSH 配置不用）。我做第二个实验的时候虽然连接成功了，但是文件不会自动同步（我上一个的传输文件范围设置的是所有project），运行报错。后来搜到：

（3）找到刚刚配置好的，添加即可，点击 `Next`。

注意注意注意：每远程调试一个新的项目，一定要重新创建一个链接并根据新创建的链接重新配置一个新的解释器，不然会报错！（即每远程调试一个新的项目，就重复一遍以上的步骤。）

也看到自动传输的配置这里是每一次连接都会创建一个新的mindsporePycharmSSH（用序号区分），可能它的设计就是要每新建一个工程就要配置一下。



以下是**使用本机环境**的配置步骤，如果用远程SSH用的是远程的环境。

第一个实验要配一下环境。conda下建立环境AImindspore（使用-p设定位置为D盘，并使用-add增加路径使文件名识别为环境名）。Python（>=3.7.5），我使用的3.9。

```
(D:\condaEnvPath\AImindspore) PS C:\Users\12915> pip install https://ms-release.obs.cn-north-4
.myhuaweicloud.com/2.0.0a0/MindSpore/cpu/x86_64/mindspore-2.0.0a0-cp39-cp39-win_amd64.whl --tr
usted-host ms-release.obs.cn-north-4.myhuaweicloud.com -i https://pypi.tuna.tsinghua.edu.cn/si
mple
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
Collecting mindspore==2.0.0a0
  Downloading https://ms-release.obs.cn-north-4.myhuaweicloud.com/2.0.0a0/MindSpore/cpu/x86_64
/mindspore-2.0.0a0-cp39-cp39-win_amd64.whl (66.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 66.6/66.6 MB 3.5 MB/s eta 0:00:00
```

安装mindspore包。

报错：AttributeError: module PIL.Image has no attribute ANTIALIAS（这里忘记截图了）

当前版本：Name: mindspore，Version: 2.0.0a0。Name: pillow，Version: 11.0.0。

高版本pillow与之不兼容，将pillow降级即可。

```
(D:\condaEnvPath\AImindspore) PS C:\Users\12915> conda install pillow=8.3.2
```

（2）实验内容及步骤

① 创建搜索树

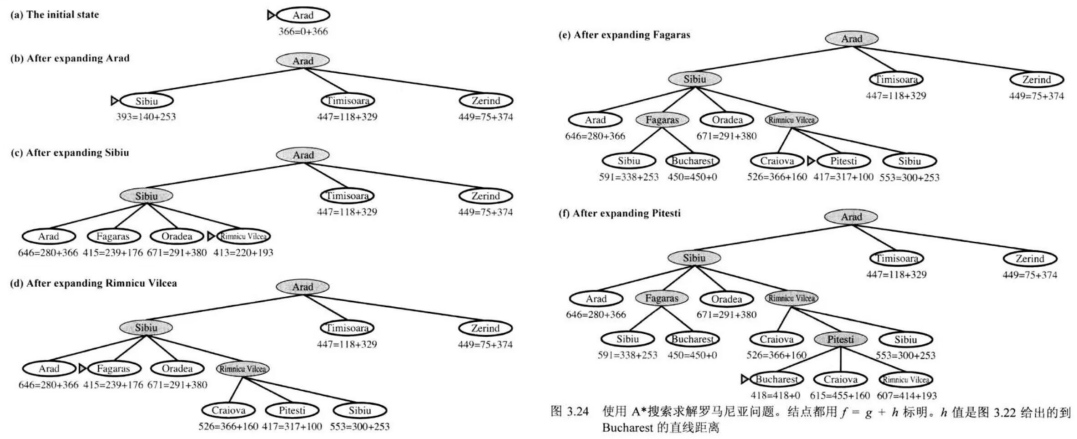


图 3.24 使用 A*搜索求解罗马尼亚问题。结点都用 $f = g + h$ 标明。h 值是图 3.22 给出的到 Bucharest 的直线距离

② 实现 A*搜索算法 && 使用编写的搜索算法代码求解罗马尼亚问题

1. 数据结构定义

SearchNode 类代表搜索树中的节点，包含当前节点状态、父节点、g: 从起点到当前节点的实际代价、h: 启发式估算代价、f: 总代价(g+h)。

RomaniaMap 类封装了地图信息，graph 存储城市间的连接和距离，heuristics 存储每个城市到目标的估算距离。

2. A*搜索的核心实现（astar_search）

初始化起始节点，优先队列(heapq)管理待探索节点，使用了一个 came_from 字典记录最优路径，cost_so_far 记录到达每个节点的最小代价。

搜索过程：heapq.heappop(frontier)每次从 frontier 选择代价最小节点，遍历当前节点的邻居，计算新路径代价，如果找到更优路径则更新，到达目标后重建并返回路径。

路径重建：从目标节点反向追溯到起点，生成完整路径。

3. main()

初始化、设置起点和终点、调用函数 astar_search 来搜索路径并输出。

具体代码内容见附录部分。

③ 分析算法的时间复杂度

A*算法的理论时间复杂度在最坏情况下为 O(b^d)，其中 b 为每个节点的平均分支数，d 为解的深度。

书中提到：

> 复杂度的结论严重依赖于对状态空间所做的假设。最简单的模型是只有一个目标状态的状态空间，本质上是树及行动是可逆的。（八数码问题满足第一、第三个假设。）在这种情况下，A*的时间复杂度在最大绝对误差下是指数级的，为 $O(b^{\Delta})$。考虑每步骤代价均为常量，我们可以把这记为 $O(b^{\varepsilon d})$，其中 d 是解所在深度。考虑绝大多数实用的启发式，绝对误差至少是路径代价 h^*的一部分，所以ε是常量或者递增的并且时间复杂度随 d 呈指数级增长。我们还可以看到更精确的启发式的作用：$O(b^{\varepsilon d}) = O((b^{\varepsilon})^d)$，所以有效的分支因子（下节会给出形式化定义）为 b^{ε}。

对于本实验中实现的 astar_search **时间复杂度**为：

每次循环操作中：

heapq.heappop(): O(log n)。

遍历邻居节点: O(k)，k 为邻居数量。

heapq.heappush(): O(log n)。

总体的时间复杂度：

最坏情况：O(b^d)。

平均情况：O(d)。

空间复杂度为：

frontier 队列：O(b^d)。

came_from 字典：O(b^d)。

cost_so_far 字典：O(b^d)。

（3）根据实验数据集，按实验要求给出相应的结果（截图）

原问题：(Arad->Bucharest)

```
从 Arad 到 Bucharest 的最短路径:
完整路径: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest']
Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest

时间复杂度分析:
最坏情况时间复杂度: O(b^d)
其中b是分支因子，d是解的深度
空间复杂度: O(b^d)
```

随机抽取一个问题：(Arad->Iasi)

```
从 Arad 到 Iasi 的最短路径:
完整路径: ['Arad', 'Sibiu', 'Rimnicu Vilcea', 'Pitesti', 'Bucharest', 'Urziceni', 'Vaslui', 'Iasi']
Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest -> Urziceni -> Vaslui -> Iasi

时间复杂度分析:
最坏情况时间复杂度: O(b^d)
其中b是分支因子，d是解的深度
空间复杂度: O(b^d)
```

（4）对实验结果进行简要分析

这里可以看出给出的路径是正确的，在问题 Arad->Iasi 这种贪婪有限最佳搜索这种会死循环的的情况也能给出最佳的路径。

四、思考题

（1）宽度优先搜索，深度优先搜索，一致代价搜索，迭代加深的深度优先搜索算法哪种方法最优？

答：

宽度优先搜索（BFS）：从根节点开始，逐层遍历节点，直到找到目标节点或遍历完所有节点。适合在无权图中找到从起点到目标节点的最短路径。保证找到的路径是最短的，但在大规模图中可能需要很大的空间来存储所有节点。

深度优先搜索（DFS）：从根节点开始，尽可能深地搜索树的分支。适合在树形或图状结构中搜索特定节点或路径。空间复杂度较低，只需要存储当前路径，但不保证找到最短路径，可能需要回溯。

一致代价搜索（UCS）：类似于 BFS，但是适用于有权图，确保找到的路径是最优的。在有权图中寻找最优路径。可以找到最优解，在某些情况下，可能不如 A*算法高效。

迭代加深的深度优先搜索（IDDFS）：是 DFS 的变种，通过限制搜索深度来模拟 BFS 的行为，通过迭代增加深度限制来找到目标。适合在树或图中寻找最短路径，尤其是在空间受限的情况下。结合了 DFS 和 BFS 的优点，空间效率较高。可能不如 BFS 或 A*算法在某些情况下高效。

这四种算法各自有不同的特点和适用场景。没有一种算法是绝对最优的，选择哪种算法取决于具体的问题和条件。

（2）贪婪最佳优先搜索和 A*搜索那种方法最优？

答：

贪婪最佳优先搜索：根据启发式函数h(n)所提供的信息，每次选择看起来最有希望的节点进行扩展，但是它不能保证找到最优解，因为它没有考虑到节点到目标的真实代价。

A*搜索算法：通过综合考虑节点的实际代价g(n)和启发式函数h(n)的估计值，保证了在每一步都能选择到最优的节点进行扩展，从而保证找到最优解。

A*搜索在大多数情况下是更优的选择，它考虑了实际代价，在保证找到解决方案的同时，还能够找到最优解。贪婪最佳优先搜索则在特定情况下，如对速度有极高要求且对最优性要求不高时，可能是一个可行的选择。

且书中提到:

最后一个观察到的事实是，在这类最优算法中——从根结点开始扩展搜索解路径的算法——A*算法对于任何给定的一致的启发式函数都是**效率最优**的。就是说，没有其他的最优算法能保证扩展的结点少于 A*算法（除了在 $f(n) = C^*$ 的结点上做文章）。这是因为如果算法不扩展所有 $f(n) < C^*$ 的结点，那么就很有可能会漏掉最优解。

令人满意的是，A*搜索在所有此类算法中是完备的、最优的也是效率最优的。然而，这并不意味着 A*算法是我们所需要的答案。难点在于，对于相当多的问题而言，在搜索空间中处于目标等值线内的结点数量仍然以解路径的长度呈指数级增长。对这个结论的分析超出了本书的范围，但仍有如下基本结论。对于那些每步骤代价为常量的问题，时间复杂度的增长是最优解所在深度 d 的函数，这可以通过启发式的绝对错误和相对错误来分析。绝对误差定义为 $\Delta \equiv h^* - h$，其中 h^* 是从根结点到目标结点的实际代价，相对误差定义为 $\varepsilon \equiv (h^* - h)/h^*$。

（3）分析比较无信息搜索策略和有信息搜索策略。

答：

无信息搜索策略和有信息搜索策略是指搜索算法是否利用额外的信息来指导搜索方向：

无信息搜索策略，如深度优先搜索（DFS）、宽度优先搜索（BFS）和一致代价搜索（UCS），只利用当前节点的信息进行搜索，不考虑节点到目标的距离或代价，因此可能需要更多的搜索步骤来找到解。

有信息搜索策略，如A*搜索算法和贪婪最佳优先搜索，利用启发式函数提供的额外信息（如节点到目标的估计距离）来指导搜索方向，从而更快地找到解。有信息搜索策略通常能更快地找到最优解，但是需要在空间和时间上付出更多的代价来计算和存储启发式函数的值。

信息使用：无信息搜索不使用任何领域特定信息，而有信息搜索利用启发式信息来指导搜索。

效率：有信息搜索通常更高效，因为它可以减少搜索空间，更快地找到解决方案。

完备性：无信息搜索策略在有限搜索空间中是完备的，而有信息搜索的完备性取决于启发式函数的设计。

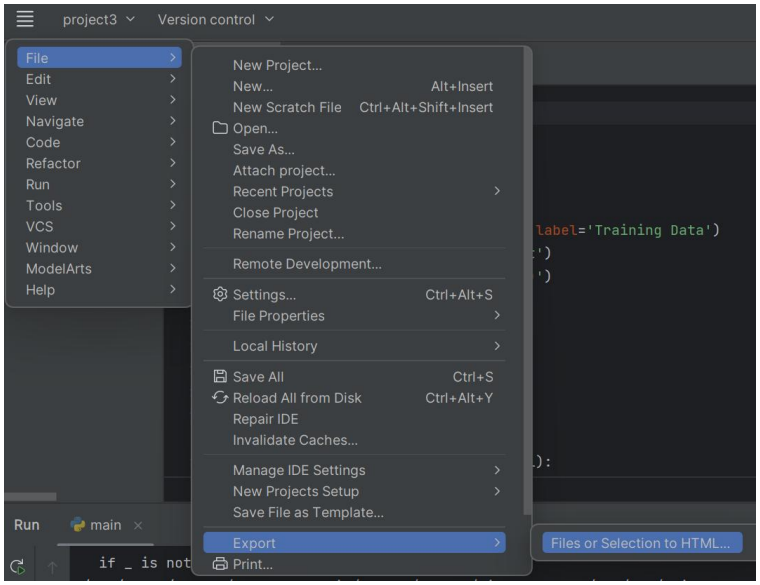
五、实验心得

这个远程环境不是很好配，主要是太新了，那个插件还没适配好，其实不用远程连接也能用，直接在华为的控制台那个环境里写就行。但是我一开始的时候已经做一步写一步在实验报告上了，而且还是更想用熟悉的环境，所以还是硬着头皮配了下去。还是有用的，最起码 anaconda 的虚拟和这些乱七八糟的配置到底代表着什么更清晰了，数据挖掘实验当时做这里的时候一头雾水。

用的时候会把我的设备带得很卡，有时候卡到我开任务管理器把它关掉都要卡半天，尤其是运行的时候，不知道是我设备的问题还是远程连接的问题，还是华为那边控制台的问题。

而且很奇怪，我不知道这个自动上传文件是怎么判定的，如果修改文件 ctrl-s 是可以成功上传的，但是加入 txt 文件不会自动上传，后来我干脆就把数据集放进来之后再开连接，就可以了。

一个小技巧：



这里可以输出 HTML 文件，用浏览器打开后右键打印可以输出 pdf 文件，这样就可以得到一份好看的代码 pdf，输出时还可以选择背景，选中可以控制只打印一段。然后将 pdf 转图片，在本实验报告结尾插入分割符（下一页分节符），并插入图片，就可以得到我现在的效果。如果使用了目录的话需要额外设置一下图片为‘T’，否则会得到奇怪的效果。

六、程序文件名清单

main.py：源代码

main.py.pdf：源代码 pdf 版

七、附录

代码如下。

main.py

```python
import mindspore
import mindspore.nn as nn
import heapq


class SearchNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g  # 从起点到当前节点的代价
        self.h = h  # 启发式估计代价
        self.f = g + h  # 总代价

    def __lt__(self, other):
        return self.f < other.f


class RomaniaMap:
    def __init__(self):
        # 定义罗马尼亚地图的图结构
        self.graph = {
            'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
            'Zerind': {'Arad': 75, 'Oradea': 71},
            'Oradea': {'Zerind': 71, 'Sibiu': 151},
            'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
            'Timisoara': {'Arad': 118, 'Lugoj': 111},
            'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
            'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
            'Drobeta': {'Mehadia': 75, 'Craiova': 120},
            'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
            'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
            'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
            'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
            'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
            'Giurgiu': {'Bucharest': 90},
            'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
            'Hirsova': {'Urziceni': 98, 'Eforie': 86},
            'Eforie': {'Hirsova': 86},
            'Vaslui': {'Urziceni': 142, 'Iasi': 92},
            'Iasi': {'Vaslui': 92, 'Neamt': 87},
            'Neamt': {'Iasi': 87}
        }

        # 启发式估计（直线距离）
        self.heuristics = {
            'Arad': 366, 'Bucharest': 0, 'Craiova': 160, 'Drobeta': 242,
            'Eforie': 161, 'Fagaras': 176, 'Giurgiu': 77, 'Hirsova': 151,
            'Iasi': 226, 'Lugoj': 244, 'Mehadia': 241, 'Neamt': 234,
            'Oradea': 380, 'Pitesti': 100, 'Rimnicu Vilcea': 193,
            'Sibiu': 253, 'Timisoara': 329, 'Urziceni': 80, 'Vaslui': 199,
            'Zerind': 374
        }

    def heuristic(self, state):
        return self.heuristics.get(state, float('inf'))

    def get_neighbors(self, state):
        return self.graph.get(state, {})


def astar_search(map_instance, start, goal):
    start_node = SearchNode(start, g=0, h=map_instance.heuristic(start))
    frontier = []
    heapq.heappush(frontier, start_node)

    # 使用字典来跟踪最佳路径
```

```python
    came_from = {start: None}

    # 记录到每个节点的最小代价
    cost_so_far = {start: 0}

    while frontier:
        current_node = heapq.heappop(frontier)

        if current_node.state == goal:
            # 重建路径
            path = []
            current = current_node.state
            while current is not None:
                path.append(current)
                current = came_from.get(current)
            return path[::-1]

        for neighbor, cost in map_instance.get_neighbors(current_node.state).items():
            new_cost = cost_so_far[current_node.state] + cost

            # 如果邻居节点之前未访问，或找到了更优路径
            if (neighbor not in cost_so_far or
                    new_cost < cost_so_far[neighbor]):
                cost_so_far[neighbor] = new_cost
                h_cost = map_instance.heuristic(neighbor)
                priority = new_cost + h_cost

                neighbor_node = SearchNode(
                    neighbor,
                    parent=current_node,
                    g=new_cost,
                    h=h_cost
                )

                heapq.heappush(frontier, neighbor_node)
                came_from[neighbor] = current_node.state

    return None


# 主程序
def main():
    # 创建罗马尼亚地图
    romania_map = RomaniaMap()

    # 设置起点和终点
    start = 'Arad'
    goal = 'Iasi'

    # 执行A*搜索
    path = astar_search(romania_map, start, goal)

    # 输出结果
    print(f"从 {start} 到 {goal} 的最短路径:")

    # 调试：打印完整路径信息
    print("完整路径:", path)

    # 输出路径
    print(" -> ".join(path))

    # 时间复杂度分析
    print("\n时间复杂度分析:")
    print("最坏情况时间复杂度: O(b^d)")
    print("其中b是分支因子，d是解的深度")
    print("空间复杂度: O(b^d)")
```

```python
if __name__ == "__main__":
    main()
```