

南京农业大学

实 验 报 告

(2023 /2024 学年 第 2 学期)

课程名称: 算法设计和分析

班 级: 人工智能 221

姓 名: 叶俊泽

学 号: 11522105

联系方式: 18860965895

南京农业大学人工智能学院

实 验 报 告

实验名称	超图最小顶点覆盖的研究	实验时间	6.30-7.7
<p>一、 实验目的和要求</p> <p>实验目的：</p> <p>本实验的主要目的是对超图最小顶点覆盖问题进行求解，比较和分析不同算法在解决该问题时的性能和效果。具体而言，通过使用贪心算法、模拟退火算法、爬山法和分支定界法四种算法，测试它们在不同规模的数据集上的运行时间、解的质量以及算法的收敛性。通过实验，我们希望达到以下目标：</p> <p>掌握和理解超图最小顶点覆盖问题的基本概念和求解方法。</p> <p>熟悉贪心算法、模拟退火算法、爬山法和分支定界法的基本原理和实现。</p> <p>比较不同算法在解决超图最小顶点覆盖问题时的效率和效果，分析其优缺点。</p> <p>提供实验数据和结果，为进一步研究和改进算法提供参考。</p> <p>实验要求：</p>			
<p>二、实验环境(实验设备)</p> <p>硬件：</p> <p>LAPTOP-HRVLI054</p> <p>处理器 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz 3.11 GHz</p> <p>机带 RAM 16.0 GB (15.8 GB 可用)</p> <p>系统类型 64 位操作系统，基于 x64 的处理器</p> <p>笔和触控 为 10 触摸点提供触控支持</p> <p>硬盘 512GB</p> <p>软件：</p> <p>PyCharm（专业版）</p> <p>Python 3.11 exe</p>			

三、实验原理及内容

实验原理：

1. 贪心算法

贪心算法通过每一步选择当前状态下的最优解，从而希望最终能达到全局最优解的目标。其核心思想是**局部最优解能够推动整体问题的解决，通过不断做出当前看似最优的选择，以期在每一步都能朝着最优解前进**。当解决最小顶点覆盖问题时，选择每次覆盖能够覆盖最多未覆盖顶点的超边，并选择其中度数最高的顶点加入覆盖集合。这种策略基于局部最优选择，期望通过每一步的最优决策，最终达到覆盖所有超边的最小顶点集合。

2. 模拟退火算法

模拟退火算法是一种启发式优化算法，基于模拟金属退火过程中的物理现象而命名。

基本思想：

初始解的选择：通过随机或启发式方法选择一个初始解作为当前解。

目标函数：定义一个目标函数，用于评估当前解的质量。在最小顶点覆盖问题中，目标是**最小化所选顶点的数量，同时确保覆盖所有超边**。

迭代优化过程：

在每个温度 T 下，通过接受新解（即使它比当前解更差）的概率来探索解空间。随机选择一个操作（例如添加或移除顶点），计算新解的目标函数值。如果新解比当前解更好（目标函数值更小），则接受新解。如果新解比当前解更差，则以一定概率接受新解。这个概率由 Metropolis 准则决定，即根据当前温度和目标函数值的差异来计算。

温度更新：

在每个温度下的迭代结束后，降低温度

T

温度降低的速率由冷却率

α

控制。当温度足够低（接近零）时，算法收敛并输出当前找到的最优解或者近似最优解。

在超图最小顶点覆盖问题中，**模拟退火算法通过随机添加或移除顶点，并根据当前温度接受劣解的概率，来搜索最小的顶点集合，以覆盖所有超边**。通过温度降低的过程，算法逐步优化，最终得到一个接近最优的解。

3. 爬山法

爬山法是一种迭代改进算法，用于从一个初始解出发，**通过不断地移动到当前解的邻域解来寻找局部最优解。其核心思想是每次选择能带来最优增益的解，直到没有更优的解可选为止。**

1. 初始化

选择一个初始顶点集作为当前解。这个初始解可以通过随机选择、贪心策略或其他启发式方法生成。

2. 邻域定义

邻域是当前解通过局部变动生成的新解。对于超图最小顶点覆盖问题，邻域可以通过以下方式定义：

添加一个顶点：向当前顶点集中添加一个新的顶点。移除一个顶点：从当前顶点集中移除一个顶点。替换一个顶点：从当前顶点集中移除一个顶点，并添加一个新的顶点。

3. 评价函数

定义一个评价函数来衡量当前顶点集的质量。对于超图最小顶点覆盖问题，评价函数可以是覆盖的超边数或顶点集的大小。目标是最大化覆盖的超边数或最小化顶点集的大小。

4. 邻域搜索和选择最优解

在当前解的邻域中，选择评价函数值最优的解。如果该解优于当前解，则将其作为新的当前解。

5. 迭代过程

重复邻域搜索和选择最优解的过程，直到满足某个停止条件：局部最优停止：当邻域中没有比当前解更优的解时，算法停止。其他条件：如达到最大迭代次数、运行时间限制或评价函数值达到某个阈值。

6. 返回结果

算法结束时的当前解即为爬山法找到的局部最优解。

4. 分支定界法

分支定界法是一种系统搜索算法，用于解决组合优化问题。其基本思路是构建一个搜索树，每个节点代表一个部分解，通过分支和剪枝的方法逐步逼近最优解。

上界（Upper Bound）：当前找到的一个可行解的代价，作为整个搜索过程的初步上界。上界提供了一个参考点，如果当前部分解的上界已经达到了我们的期望或已知的最优解，我们可以提前结束搜索或停止探索该分支，以节省计算资源。

下界（Lower Bound）：当前部分解的最优估计代价，若该下界高于上界，则该分支及其子分支可以被剪枝。即在搜索过程中，当发现当前部分解的下界已经高于当前找到的最优解（即上界），那么可以判定该分支及其所有子分支不可能产生更优的解，从而可以安全地剪掉这些分支，减少搜索空间。

搜索过程：

分支生成：

从初始状态开始，生成可能的分支。每次可以考虑添加或移除一个顶点，生成新的可能解。

剪枝条件：

对于每个生成的分支，计算其下界。如果下界比当前已知的上界还要高，则这个分支及其所有子分支可以被剪枝掉，因为它们不可能产生更优的解。如果下界比当前的上界低或相等，则继续探索这个分支。结合启发式方法（如贪心算法）来确定初始解和下界，以及利用剪枝策略来减少搜索空间，从而提高算法的效率。

迭代搜索：不断地生成新的分支，计算它们的下界，根据剪枝条件来决定是否继续搜索或剪枝。

终止条件：当所有可能的分支都被探索完毕或者达到了停止条件（例如找到了一个比当前上界更优的解），算法结束。

实验内容：

所有的算法都输入两个数据集进行测试，分别是 circuit_3.txt, 和 cryg10000.txt。每一行表示一个超边连接的顶点集，其中 circuit_3 顶点数：12127，cryg10000 顶点数：10000。

实验指标均为：

最小顶点覆盖集合大小：评估贪心算法求解的覆盖效果。

算法运行时间：记录每次求解的时间，评估算法的效率和性能。

比较不同数据集的表现：分析不同数据集在算法执行过程中的区别和影响。

1. 贪心算法

1.1 实验步骤

实现贪心算法解决超图最小顶点覆盖问题的算法。

编写代码读取数据集，并计算每个顶点的度数。

```
def read_hypergraph_data(file_path):
    hyperedges = []
    vertices_degrees = {}
    with open(file_path, 'r') as file:
        for line in file:
            line = line.strip()
            if line:
                vertices = list(map(int, line.split()))
                hyperedges.append(vertices)
                for vertex in vertices:
                    if vertex in vertices_degrees:
                        vertices_degrees[vertex] += 1
                    else:
                        vertices_degrees[vertex] = 1
    return hyperedges, vertices_degrees
```

这个方法在接下来所有的算法中都和重复使用的！

实现贪心策略：选择能够覆盖最多未覆盖顶点的超边，并选择其中度数最高的顶点加入覆盖集合。

1.2 实验过程

数据预处理：

从文件中读取超图数据，并构建超边集合。

统计每个顶点的度数，作为贪心算法选择策略的依据。

```
# 将超边按照未覆盖顶点最多的顺序排序
sorted_hyperedges = sorted(hyperedges, key=lambda x: -max(vertices_degrees[v]
for v in x))
```

贪心算法求解：

根据贪心策略，选择并计算最小顶点覆盖集合。

记录算法运行时间，并输出最小顶点覆盖集合的结果。

```
# 如果当前超边没有被覆盖，则选择当前超边中度数最高的顶点加入覆盖集合
max_degree_vertex = max(edge, key=lambda v: vertices_degrees[v])
vertex_cover.add(max_degree_vertex)
```

1.3 复杂度分析

1.3.1 数据预处理阶段：

读取数据集：假设超图有： V 个顶点和 E 条超边。

读取超图数据的时间复杂度为：

$$O(V+E)$$

因为需要遍历每条超边和每个顶点。

计算顶点度数：

计算每个顶点的度数的时间复杂度为

$$O(E)$$

因为需要遍历每条超边中的顶点。

总结：数据预处理阶段的时间复杂度为

$$O(V+E)。$$

1.3.2 贪心算法求解阶段：

排序超边：对超边进行排序，每次排序的时间复杂度为

$$O(E\log E)$$

其中 E 是超边的数量。

选择顶点：在每次迭代中，选择能够覆盖最多未覆盖顶点的超边，加入覆盖集合。

每次选择顶点的操作时间复杂度为

$$O(E)$$

因为需要遍历每条超边的顶点。

总结：贪心算法求解阶段的时间复杂度主要由排序操作决定，

$$O(E\log E)。$$

总体时间复杂度分析：

综上所述，总体贪心算法的时间复杂度为： $O(V+E+E\log E)$ 。然而，在大多数情况下，我们会简化为 $O(E\log E)$ ，因为 $E\log E$ 项对于超图问题来说通常是主导因素，而 V 的影响相对较小。

贪心算法在处理超图最小顶点覆盖问题时，时间复杂度主要由排序超边的操作决定，并且在不同的数据规模下，算法的效率可以通过优化排序和选择策略来进一步改善。

2. 模拟退火算法

2.1 Metropolis 准则

Metropolis 准则包含以下步骤：

计算目标函数差异：假设当前解的目标函数值为

$$f(\text{current})$$

新提议的解（操作后的解）的目标函数值为

$$f(\text{new})$$

计算能量差：计算当前解到新解的能量差，通常表示为

$$\Delta E = f(\text{new}) - f(\text{current})$$

决定接受概率：根据以下规则决定是否接受新解：

如果 $\Delta E < 0$ （新解比当前解更好），则始终接受新解。

如果 $\Delta E \geq 0$ （新解比当前解差或相等），则根据一定的概率接受新解，这个概率由下式给出：

$$P(\text{accept}) = e^{-\Delta E / T}$$

其中， T 是当前的温度。这个公式表明，当温度 T 较高时，接受差解的概率较高；随着温度的降低，接受差解的概率也随之减小。

随机决定，生成一个随机数 r 从均匀分布中取值 $[0, 1)$ ，如果

$$r \leq P(\text{accept})$$

则接受新解；否则保持当前解不变。

Metropolis 准则确保了在模拟退火算法的迭代过程中，即使遇到比当前解更差的解，也有一定概率接受它，以避免陷入局部最优解。随着算法温度 T 的降低，接受劣解的概率减小，算法更趋向于在温度逐步降低的过程中收敛到更优的解。

2.2 实验过程

2.2.2 初始化参数

`current_solution`（当前解的集合）和 `best_solution`（最优解的集合）都初始化为空集合。初始温度 `current_temperature` 设置为 `initial_temperature`，如 100。冷却率 `cool_rating` 设为：0.99。截止温度 `stopping_temperature` 设为：26。

2.2.3 计算覆盖超边数

计算覆盖超边数的主要目的是**评估当前解的质量**，即确定给定顶点集 `vertex_set` 能覆盖多少条超边。这是模拟退火算法评估和选择解的重要步骤。这就相当于理论中的 $f(x)$ 这个函数。

在模拟退火算法中，我们需要知道当前解 `current_solution` 能覆盖多少条超边。覆盖的超边数越多，解的质量越高。根据覆盖超边数，我们可以计算当前解和最佳解的质量差异。在 Metropolis 准则中，这个差异用于决定是否接受新解，即使新解的质量不如当前最佳解（允许一定的概率接受更差的解，以跳出局部最优）。通过计算当前解的覆盖超边数，可以判断是否更新最佳解 `best_solution`。如果当前解的覆盖数更多，则更新最佳解。

2.2.3 循环迭代

温度条件判断：

主循环在温度 `current_temperature` 大于停止温度 `stopping_temperature` 时进行迭代。通过逐步降低温度，模拟退火算法探索解空间，趋向于找到最优解。

迭代生成新解：

每次温度下，进行若干次（1000 次）操作，通过随机选择顶点添加或移除，生成新解 `current_solution`。

每次操作生成新解 `new_solution`：

50% 概率随机选择一个顶点添加到 `current_solution` 中，生成新解 `new_solution`。

50% 概率从 `current_solution` 中随机移除一个顶点（前提是 `current_solution` 非空），生成新解 `new_solution`。

```
for _ in range(100):
    # Perturb current solution (e.g., add or remove a vertex)
    # Here, we randomly choose to add or remove a vertex
    if random.random() < 0.5:
        # Add a random vertex to the current solution
        candidate_vertex = random.choice(list(vertices_degrees.keys()))
        current_solution.add(candidate_vertex)
    else:
        # Remove a random vertex from the current solution (if not empty)
        if current_solution:
            vertex_to_remove = random.choice(list(current_solution))
            current_solution.remove(vertex_to_remove)
```

计算覆盖超边数:

计算当前解 `current_solution` 和最佳解 `best_solution` 能覆盖的超边数 `current_cost` 和 `best_cost`, 评估解的质量。

根据 Metropolis 准则接受新解:

根据 Metropolis 准则, 决定是否接受新解并更新最佳解 `best_solution`。这一步允许在一定概率下接受较差的解, 以避免陷入局部最优。

如果当前解 `new_solution` 的覆盖超边数 `current_cost` 大于最佳解 `best_cost`, 则接受当前解 `new_solution`, 更新当前解 `current_solution`。

如果当前解 `new_solution` 的覆盖超边数 `current_cost` 小于或等于最佳解 `best_cost`, 则根据 Metropolis 准则, 以概率 $\exp((\text{current_cost} - \text{best_cost}) / \text{current_temperature})$ 决定是否接受当前解 `new_solution`。

如果接受当前解 `new_solution`, 则更新当前解 `current_solution`。

如果当前解 `new_solution` 的覆盖超边数 `current_cost` 大于最佳解 `best_cost`, 则更新最佳解 `best_solution`。

```
if current_cost > best_cost or random.random() < math.exp((best_cost -
current_cost) / current_temperature):
    best_solution = current_solution.copy()
# Cooling schedule: reduce temperature
current_temperature *= cooling_rate
```

2.2.4 实验复杂度分析

模拟退火算法的时间复杂度分析主要集中在以下几个方面: 生成新解的过程、计算覆盖超边数的过程、以及温度下降的过程。

生成新解的过程:

每次迭代中, 模拟退火算法会生成若干个新解。生成新解的过程包括随机选择顶点进行添加或移除。拷贝当前解 `current_solution`: 时间复杂度为:

$$O(V)$$

其中 V 是顶点数。

随机选择顶点并添加到解中或从解中移除: 时间复杂度为 :

$$O(1)$$

总的来说, 生成一个新解的时间复杂度为 : $O(V)$ 。

计算覆盖超边数的过程

每次生成一个新解后, 需要计算该解能覆盖的超边数。遍历所有超边: 时间复杂度为 :

$$O(E)$$

其中 E 是超边数。

检查每条超边是否至少有一个顶点在 `vertex_set` 中: 最坏情况下, 时间复杂度为 :

$$O(V)$$

因此, 计算覆盖超边数的总时间复杂度为 $O(E \cdot V)$ 。

主循环及温度下降的过程:

外层循环 (温度下降的过程): 设温度下降的次数为 T , 即从初始温度到停止温度需要的迭代次数。

内层循环（每次温度下的迭代次数）：设为常数 k （如 100 次）。

总体时间复杂度

综合以上分析，模拟退火算法的总体时间复杂度为：

$$O(T \cdot k \cdot (V + E \cdot V))$$

由于 k 是常数，我们可以简化为：

$$O(T \cdot (V + E \cdot V))$$

在最坏情况下，当 E 大于 V 时，时间复杂度主要受 $E \cdot V$ 项影响，因此总体时间复杂度可以表示为：

$$O(T \cdot E \cdot V)$$

T ：温度下降的次数，取决于初始温度、停止温度和冷却率。

V ：顶点数。

E ：超边数。

模拟退火算法的时间复杂度表明其运行时间主要取决于温度下降的次数以及计算覆盖超边数的复杂度，而生成新解的过程相对次要。通过调整温度参数，可以在运行时间和解的质量之间进行权衡。

3. 爬山法

3.1 初始化

设置初始解：使用贪心算法 `greedy_minimum_vertex_cover` 获取初始解 `current_solution`。

初始化变量：

`best_solution`：保存当前最优解。

`best_cost`：当前最优解覆盖的超边数。

`improved`：标志变量，用于判断是否继续迭代。

`iterations`：迭代计数器，用于控制最大迭代次数。设置为 1000 次

3.2 主循环

生成邻域解：

对每个顶点 `vertex`：创建当前解的副本 `neighbor_solution`。如果当前解包含该顶点，则移除它；否则，添加它。计算邻域解 `neighbor_solution` 的覆盖超边数 `neighbor_cost`。

```
for vertex in list(vertices_degrees.keys()):
    neighbor_solution = current_solution.copy()
    if vertex in current_solution:
        # 如果当前解中包含该顶点，则移除它
        neighbor_solution.remove(vertex)
    else:
        # 否则，添加该顶点
        neighbor_solution.add(vertex)
    neighbor_cost = calculate_covered_hyperedges(neighbor_solution)
```

更新最佳解:

如果邻域解的覆盖超边数 `neighbor_cost` 大于当前最佳解 `best_cost`, 则更新最佳解 `best_solution` 并设置 `improved` 为 `True`。复制邻域解为新的当前解 `current_solution`。

对比此解法和模拟退火法, 爬山法是一种贪心算法, 每次迭代选择最优邻域解, **只接受改进解, 这使其易于实现且计算速度快, 但容易陷入局部最优解**。模拟退火法则通过引入温度参数, 允许在高温时接受较差解以跳出局部最优, **随着温度降低逐渐只接受更优解, 这增加了找到全局最优解的概率, 但其算法复杂度较高且运行时间较长**。此外, 模拟退火法对初始温度、冷却速率等参数的依赖性更强, 需要进行调整和优化。例如:

参数设置不当:

初始温度: 如果初始温度设置过低, 算法一开始就不能有效地探索解空间, 容易陷入局部最优。

冷却速率: 如果冷却速率过快, 温度下降过快, 使得算法在早期阶段就失去了接受较差解的能力, 导致算法过早收敛。

爬山法和贪心法本质上是一样的, 只是贪心策略不同, 导致时间复杂度有巨大的不同。而且, 无论是爬山法还是模拟退火法, 初始化解都可以使用贪心法作为初始解。

3.3 复杂度分析

初始化:

创建初始解的时间复杂度是

$$O(n)$$

其中 n 是顶点的数量。

主循环:

每次迭代需要检查所有邻域解的覆盖超边数, 即对每个顶点, 添加或移除一个顶点并计算新解的覆盖超边数。

对每个顶点, 计算覆盖超边数的复杂度为

$$O(m)$$

其中 m 是超边的数量, 因为需要检查每个超边是否被当前解覆盖。

邻域搜索:

对每个顶点进行邻域搜索的复杂度为

$$O(n \cdot m)$$

因为需要遍历所有顶点并计算每个顶点的覆盖超边数。

终止条件:

通常, 爬山法会在没有邻域解能进一步优化时停止。因此, 最坏情况下, 算法可能需要遍历所有可能的解, 这种情况的复杂度是指数级的:

$$O(2^n)$$

然而, 实际中爬山法往往在局部最优解处提前终止, 所以平均情况下的复杂度要低得多。

总的时间复杂度:

综合以上分析, 爬山法的时间复杂度取决于: 爬山法的时间复杂度在最坏情况下可能是指数级的 $O(2^n)$, 但在实际应用中, 通过限制迭代次数和适当的初始解选择, 时间复杂度可以简化为:

$$O(k \cdot n \cdot m)$$

其中 k 是迭代次数。因此, 在一般情况下, 爬山法的时间复杂度是多项式时间复杂度 $O(n \cdot m)$ 。

4. 分支定界法

4.1 定义节点类

定义节点类 `BranchBoundNode`，用来表示搜索树中的每个节点状态。具体来说，在解决最小顶点覆盖问题时，每个节点代表一个可能的解（即顶点覆盖集合）。节点的状态包括当前已选取的顶点，当前的覆盖超边数，当前的上界估计等。分支定界法会通过扩展这些节点来探索更多的解空间，并在每一步根据当前状态的上界信息进行剪枝，以减少搜索空间。

其中，在节点类中，`bound` 属性和 `cost` 属性是最重要的。`cost` 表示当前节点的顶点覆盖解集合的大小，即顶点的数量。在最小顶点覆盖问题中，我们希望找到的解集合越小越好，因此 `cost` 可以用来比较不同节点之间的解的优劣。

`bound` 是当前节点的一个上界估计值。它用于帮助算法进行剪枝操作，即提前终止对某些子树的搜索，以节省计算资源。

```
class BranchBoundNode:
    def __init__(self, solution=None, cost=float('inf'), bound=float('inf'),
depth=0):
        self.solution = solution # 当前节点的解（顶点覆盖集合）
        self.cost = cost # 当前节点解的代价（覆盖的超边数）
        self.bound = bound # 当前节点的上界（用于剪枝的参考值）
        self.depth = depth # 当前节点在搜索树中的深度
```

4.2 搜索节点

这个部分也是这个算法中最核心的部分。

循环处理节点：使用一个循环来遍历 `nodes` 列表中的每个节点，直到所有节点都被处理完毕。

剪枝操作：对于当前节点 `current_node`，首先检查其 `bound` 是否小于当前找到的最优解 `best_cost`。如果是，则跳过该节点，这是一种剪枝操作，因为该节点及其子节点不可能产生更好的解。

扩展操作：根据当前节点的深度 `depth` 选择一个顶点进行扩展。顶点通常是从一个顶点集合中选择的，可以选择未覆盖的顶点或者已覆盖的顶点进行扩展。

生成新节点：包含顶点的情况：生成一个新节点 `included_node`，这个节点表示当前节点解集合中包含了选定顶点的情况。更新新节点的解集合、成本和上界（如果适用），然后将其加入到 `nodes` 列表中，以便进一步探索。

不包含顶点的情况：生成一个新节点 `excluded_node`，这个节点表示当前节点解集合不包含选定顶点的情况。同样，更新新节点的解集合、成本和上界，并将其加入到 `nodes` 列表中。

更新最优解：在每次扩展过程中，如果新生成的节点的解集合比当前找到的最优解更优，则更新 `best_solution` 和 `best_cost`。

终止条件：当所有节点都被处理完毕（即 `nodes` 列表为空），算法结束。此时，`best_solution` 包含了找到的最优解。

```
while nodes:
    # 取出当前节点
    current_node = nodes.pop(0)
    # 如果当前节点的上界小于当前最优解的代价，则剪枝，继续下一个节点
    if current_node.bound < best_cost:
```

```

        continue
# 如果当前节点的深度小于顶点度数的长度，则继续扩展
if current_node.depth < len(vertices_degrees):
    # 获取当前深度对应的顶点
    vertex = list(vertices_degrees.keys())[current_node.depth]
    # 探索包含该顶点的情况
    included_solution = current_node.solution.copy()
    included_solution.add(vertex) # 将该顶点加入当前解中
    included_bound = calculate_covered_hyperedges(included_solution,
hyperedges) # 计算新解的覆盖超边数
    included_node = BranchBoundNode(solution=included_solution,
cost=len(included_solution),
                                bound=included_bound,
depth=current_node.depth + 1)
    # 更新最优解和最优代价
    if included_node.cost < best_cost:
        best_solution = included_node.solution
        best_cost = included_node.cost
    # 如果新解的上界大于等于当前最优代价，则将其加入节点列表继续扩展
    if included_node.bound >= best_cost:
        nodes.append(included_node)
    # 探索不包含该顶点的情况
    excluded_solution = current_node.solution.copy()
    excluded_bound = calculate_covered_hyperedges(excluded_solution,
hyperedges) # 计算不包含该顶点的覆盖超边数
    excluded_node = BranchBoundNode(solution=excluded_solution,
cost=len(excluded_solution),
                                bound=excluded_bound,
depth=current_node.depth + 1)
    # 更新最优解和最优代价
    if excluded_node.cost < best_cost:
        best_solution = excluded_node.solution
        best_cost = excluded_node.cost
    # 如果新解的上界大于等于当前最优代价，则将其加入节点列表继续扩展
    if excluded_node.bound >= best_cost:
        nodes.append(excluded_node)

```

4.3 时间复杂度分析

分支定界法的时间复杂度分析比较复杂，取决于具体问题、节点数、剪枝效率等因素。

最坏情况：

在最坏情况下，分支定界法会生成所有可能的解空间，类似于穷举搜索。对于一个有 n 个顶点的超图，可能的顶点覆盖解空间是 2^n 个。

最坏情况下的时间复杂度为 $O(2^n)$ 。

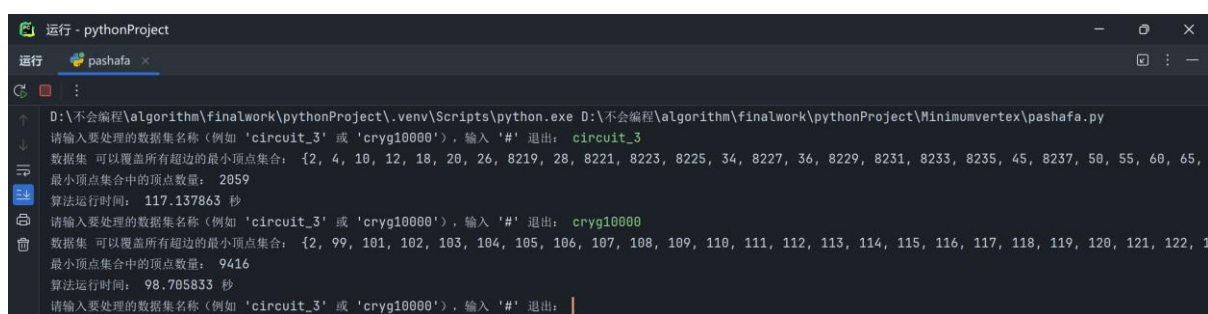
实际情况：

分支定界法通过剪枝大大减少了需要搜索的节点数，因此实际时间复杂度通常远低于最坏情况。剪枝效率越高，搜索的节点越少，时间复杂度越低。

实验结果：

数据集\算法	贪心算法	模拟退火	爬山法	分支定界法
circuit_3	运行时间： 0.0151s 顶点集大小： 2059	运行时间： 2219.5071s 顶点集大小： 2059	运行时间： 117.1378s 顶点集大小： 2059	运行时间： 30min 无法得出解 顶点集大小：
cryg10000	运行时间： 0.0167s 顶点集大小： 9416	运行时间： 2560.0060s 顶点集大小： 9416	运行时间： 98.7058s 顶点集大小： 9416	运行时间： 30min 无法得出解 顶点集大小：

同时，我们对比所有可以在一定时间内得出结果的算法，比较其得到的顶点集的顶点，几乎完全一致。其中可能有数据集本身的问题，但至少可以说明，这几种算法在一定规模的数据集上，可以达到一个较好的近似解，甚至得出最优解。



根据实验结果，贪心算法运行速度最快，在处理 circuit_3 和 cryg10000 数据集时，分别只需约 0.0151 秒和 0.0167 秒，但得到的顶点集大小均为 2059 和 9416，但得到的解可能不是最优；模拟退火算法在时间上消耗较大，分别需约 2219.5071 秒和 2560.0060 秒，但能找到相同的解，表明其质量较好；爬山法在运行时间和解的质量之间取得了一定平衡；分支定界法虽然理论上能找到最优解，但在大规模数据集上耗时过长，30 分钟内无法得出解，实际应用受限。

总体而言，贪心算法适合快速求解，模拟退火和爬山法适合在时间允许的情况下求更优解，分支定界法在大数据集上不适用。

参考文献：

- [1] A Simulated Annealing Algorithm for Spatial Straightness Error Evaluation[J]. Altarazi, Safwan A;Mandahawi, Nabeel F.IIE Annual Conference. Proceedings,2008
- [2] Deep learning for fake news detection: A comprehensive survey[J]. Hu Linmei;Wei Siqu;Zhao Ziwang;Wu Bin.AI Open,2022
- [3] Do Sentence Interactions Matter? Leveraging Sentence Level Representations for Fake News Classification.[J]. Vaibhav Vaibhav;Raghuram Mandyam Annasamy;Eduard Hovy.CoRR,2019
- [4] Twitter rumour detection in the health domain[J]. Rosa Sicilia;;Stella Lo Giudice;;Yulong Pei;;Mykola

Pechenizkiy;;Paolo Soda.Expert Systems With Applications,2018

[5] Rumor Detection over Varying Time Windows.[J]. Sejeong Kwon;;Meeyoung Cha;;Kyomin Jung.PLoS ONE,2017

[6] Very Deep Convolutional Networks for Large-Scale Image Recognition.[J]. Karen Simonyan;Andrew Zisserman.CoRR,2014

[7]Recurrent Neural Network Regularization.[J]. Wojciech Zaremba;Ilya Sutskever;Oriol Vinyals.CoRR,2014

[8] 基于混沌模拟退火粒子群优化算法的电动汽车充电站选址与定容[J]. 艾欣;李一铮;王坤宇;胡俊杰. 电力自动化设备, 2018 (09)

[9] 模拟退火算法的应用[J]. 周佳莉. 西部皮革, 2019

[10] 基于改进模拟退火粒子群算法的立体车库结构优化[J]. 洪晴岚;宋燕利;冯维. 武汉理工大学学报, 2020

[11] 基于快速模拟退火的案例检索模型研究[J]. 赵卫东, 盛昭瀚. 管理工程学报, 2001

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

超图理论是非常重要且应用广泛的理论。在本次实验中，我使用了贪心算法、模拟退火算法、爬山法和分支定界法这四种不同的算法，来解决超图的最小顶点覆盖问题。通过在 `circuit_3` 和 `cryg10000` 数据集上的实验分析，我深入理解了不同算法在求解效率和解质量上的综合表现。

超图的最小顶点覆盖问题旨在寻找一个最小的顶点集合，使得该集合中的顶点能够覆盖超图中的所有超边。这一问题在网络安全、资源分配以及生物信息学等领域具有广泛的应用。在本次实验，我深刻体会到不同算法在解决同一问题时的优缺点。贪心算法尽管实现简单且快速，但解的质量有限；模拟退火算法虽然耗时较长，但能够找到较优的解；爬山法在解的质量和运行时间上取得了较好的平衡；而分支定界法尽管理论上能够找到最优解，但在大规模数据集上表现欠佳。不同算法各有优劣，适用于不同的应用场景。

同时，实验中也有一些不足需要在日后加以完善。

1. 可以尝试结合多种算法的优点，例如将**贪心算法与模拟退火算法结合**，以提高求解速度和解的质量。
2. 在实验中合理选择数据结构和优化代码实现，可以进一步提高算法的运行效率。
3. 进一步学习和研究其他优化算法，如遗传算法、蚁群算法等，探索更多解决超图最小顶点覆盖问题的可能性。

这次实验，我深刻认识到了算法选择的重要性，并且学会了根据问题的特点和规模选择合适的算法来解决问题。这对我以后在解决实际问题时选择合适的算法提供了重要的参考和指导。这些知识和经验将为我今后的研究和工作奠定坚实的基础。

