南京農業大學

实 验 报 告

（2020 /2021 学年　第 2 学期）

课程名称：算法设计与分析

班　　级：信息 192

姓　　名：朱雅丽

学　　号：19119206

指导老师：谢小军

南京农业大学人工智能学院

实 验 报 告

实验名称	基于多种算法的超图最小顶点覆盖问题求解	实验时间	2021.7.10-7.15

一、 实验目的和要求

实验目的：

通过研究超图的最小顶点覆盖问题，了解超图理论相关知识，掌握顶点覆盖问题的相关算法，融会贯通使用课堂所学精确算法与近似算法，掌握算法性能分析能力。

实验要求：

1、利用所学知识设计至少两种方法求解超图的最小顶点覆盖

2、理论分析其最坏时间复杂度

3、实验分析所提出算法的效率

二、实验环境(实验设备)

硬件： CPU lntel(R) Core(TM) i5-8265U CPU @1.60GHz1.80GHz

内存 8GB

软件： 操作系统 Windows 10 Professional 2004

语言 C++ 11

编译器 gcc version 8.1.0 (x86_64-posix-sjlj-rev0)

三、实验原理及内容

1 超图最小顶点覆盖问题的描述

1.1 超图

超图是图论中简单图的泛化形式，二者最大的差别在于简单图中每一条边最多可以连接两个顶点，而在超图中边可以连接两个以上的顶点。在超图理论中，将这种能够连接两个以上顶点的边称为超边。如果每个超边连接相同数量的顶点，则称该超图为k-一致超图。2-一致超图即为我们常知的简单图。

超图可以表示为$H=(V,E)$。

$V=\{v_1,v_2,\dots v_n\}$，表示顶点的集合；

$E=\{e_1,e_2,\dots e_m\}$，其中$e_i=\{v_j,\dots v_k\}(i=1,2,\dots m)(j,k=1,2,\dots n)$，$E$是顶点集合$V$的子集簇，需满足条件：

1）e_i不为空（$i=1,2,\dots m$）；

2）e_i的并集为V；

在普通图中，顶点度（Degree）的定义为包含该顶点的边的个数。在超图中，顶点的度的定义也类似，用公式可如下表示：$d(v)=\sum_{\{e\in E,v\in V\}}e$。超图中的边称为超边，对于超边$e\in E$而言，超边的度的定义为超边上包含的顶点的数目：$\delta(e)=|e|$。

1.2 最小顶点覆盖问题

简单图中顶点覆盖的定义如下：给定一个表$G=(V,E)$, 其中V表示顶点集，E表示边集；给出一个整数k，如果存在顶点的某个子集S，其满足S中顶点个数小于k,并且对E中每条边(u,v),都有u在S中，或者v在S中，也就是S中的顶点覆盖了边集E。

最小顶点覆盖即不给出k,而让算法判断并找出所有顶点覆盖中最小的那个。

超图中的最小顶点覆盖与简单图类似，给出一个超图$H=(V,E)$，在顶点集合V中找到一个最小的顶点子集S，对于E中每条边e_i所包含的顶点中，一定存在v_j属于 S。则S称为的H最小顶点覆盖。也即S中的顶点覆盖边集E，公式定义如下：

$$\forall e_i\in E,\exists v_j\in e_i\ and\ v_j\in S$$

最小顶点覆盖问题是顶点覆盖问题中最常见、研究最多及应用最广的一种，也是组合优化中典型NP-Hard问题，一般而言很难在多项式时间内求得精确解，因此目前求解这个问题的算法多为近似算法。

1.3 实验算法说明

本次实验中使用了回溯法、贪心法、退火法三种算法求解该问题，回溯法作为精确算法的代表，用于比较另外两种近似算法的准确程度。

由于回溯法的时间性能仍然是指数级别，无法在老师给出的大数据集上运行出结果，因此随机生成了几个小规模的超图用于测试比较。

2 基于回溯法的超图最小顶点覆盖算法及分析

2.1 实验原理

回溯法是一种基于搜索的算法设计技术，是一种通用的求解问题的方法，它可以看作是穷举搜索，即蛮力法的改进，所以回溯法得到的结果是精确的。回溯法可以系统地搜索到问题的某个或者全部解，故而适用于求解满足某些约束条件的组合优化问题。

回溯法将搜索空间（问题的解空间）看作一定的结构，通常作为树结构，一个解即对应于树中的一片树叶。算法从树的根节点出发，按照某种策略（深度优先、宽度有限、或者宽-深结合等方法）依序遍历一棵树，尝试所有可能到达的结点。当无法前行时（即不满足约束函数时），就进行剪枝，并回退一步或若干步，以此来避免无用搜索，改善算法的运行时间。

2.2 实验内容

对于超图的最小顶点覆盖问题，解空间是顶点集合的全部子集，而回溯法的思想就是从全集出发，通过逐渐删去元素来构建子集，按照深度优先的思想依次遍历每个子集。

剪枝的思想是依据约束条件：满足顶点覆盖，来进行剪枝，减少无用的回溯。即如果前一个顶点子集不满足顶点覆盖，那从这个子集中删去一个元素形成的子集一定也不满足，通过这样可以减少回溯的次数，避免对每个子集进行判断。

利用回溯法求解超图最小顶点覆盖问题的算法描述如表 1 所示。

表 1 回溯法求解超图最小顶点覆盖问题

算法 1 $BT\_MVC(V,v)$	
Input:	顶点集 V, 从顶点集中删去的顶点 v；v 初始为 0，MVC 初始为 V
Output:	最小顶点覆盖集 MVC
1	*for all* $w \in V$ *such that* $w \geqslant v$ *do*
2	$if\ V-\{w\}\ is\ a\ vertex\ cover$
3	$if\ V\ has\ less\ elements\ than\ MVC$
4	$\mathrm{MVC} \leftarrow \mathrm{V}-\{\mathrm{w}\}$
5	$BT\_MVC(V-\{w\},w)$

2.3 复杂性分析

回溯法的实质仍然是一种穷举搜索法，最坏情况下需要遍历所有可能的解，即顶点集合的全部子集，每次都需判断是否满足顶点覆盖，所以该算法的时间复杂度为$O(2^N)$。

算法生成递归树的最大深度为N，每次函数调用占用空间为$O(N)$，所以该算法的空间复杂度为$O(N^2)$。

2.4 算法运行测试

图 1 显示了回溯法运行时间随问题规模的变化曲线，可以看出随着问题规模的增大，算法的运行时间是以指数形式递增的。

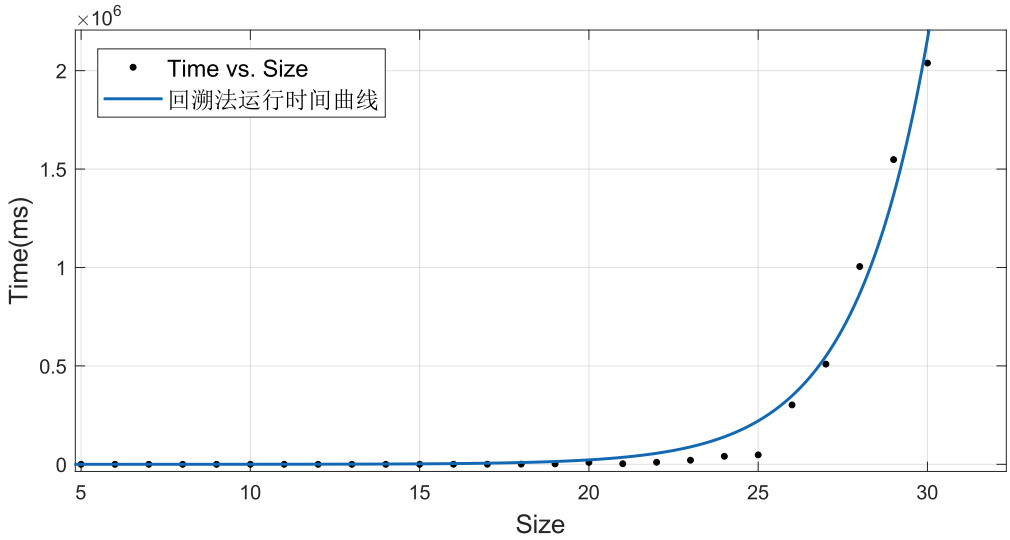

图1 回溯法运行时间随问题规模变化曲线

3 基于贪心法的超图最小顶点覆盖算法及分析

3.1 实验原理

贪心算法是指在对问题进行求解时，在每一步选择中都采取最好或者最优(即最有利)的选择，从而希望能够导致结果是最好或者最优的算法。贪心算法可以用来解决最优化问题，但通常只能得到一个局部最优解。

用贪心法求解问题，首先需要考虑贪心策略。在用贪心法解决最小顶点覆盖问题中，贪心策略是每次都寻找覆盖的边最多的顶点，即度数最大的顶点。

3.2 实验内容

基于贪心策略，该算法计算每个顶点的度（连接的边的数目），每次都选择度数最大的边加入到顶点覆盖集(MVC)中，只要边集没有被完全覆盖，就不断进行选择。最终得到一个近似最优解。

简单的贪心法如上所述，由于没有在顶点加入MVC的同时对边集进行处理，可能导致前一个顶点加入后与后一个顶点加入后，覆盖的边的条数没有太大变化（即存在无效边），从而使得结果的准确度大大降低。

考虑对简单贪心法进行改进，每次向MVC中加入一个顶点后，将该顶点连接的边从边集中删去，将该顶点也从顶点集中删去，各顶点的当前度数也将随之改变。这样可以保证每次加入到MVC的顶点都是剩下的顶点中覆盖有效边最多的。

利用贪心法求解超图的最小顶点覆盖问题算法描述如表2所示。

表2 回溯法求解超图最小顶点覆盖问题

算法 2 $Greedy\_MVC(V,E)$
Input: 顶点集 V, 边集 E;
Output: 最小顶点覆盖集 MVC
//MVC 初始为空，F(v)表示顶点 v 覆盖的边的集合
1 $while\ (E \neq \emptyset)\ do$
2 $choose\ vertex\ v \in V:\ \forall z\ \in\ V \deg(z) \leq \deg(v)$;
3 $V\ \leftarrow\ V - \{v\}$
4 $E\ \leftarrow E - F(v)$
5 $MVC\ \leftarrow\ MVC\ \cup \{v\}$

3.3 复杂性分析

最坏情况下，算法需要依次遍历每个顶点，同时每次需要计算顶点所覆盖的边的集合时间复杂度近似为$O(N^2)$(该复杂度与具体的数据集有关，本实验所用数据集边数等于顶点数目,故近似为$O(N^2)$)，因此总的时间复杂度为$O(N^3)$。由于需要大小$O(N)$的数组存储最终结果，空间复杂度为$O(N)$。

3.4 算法运行测试

图 2 显示了贪心法运行时间随问题规模的变化曲线，可以看出随着问题规模的增大，算法的运行时间是多项式递增的。

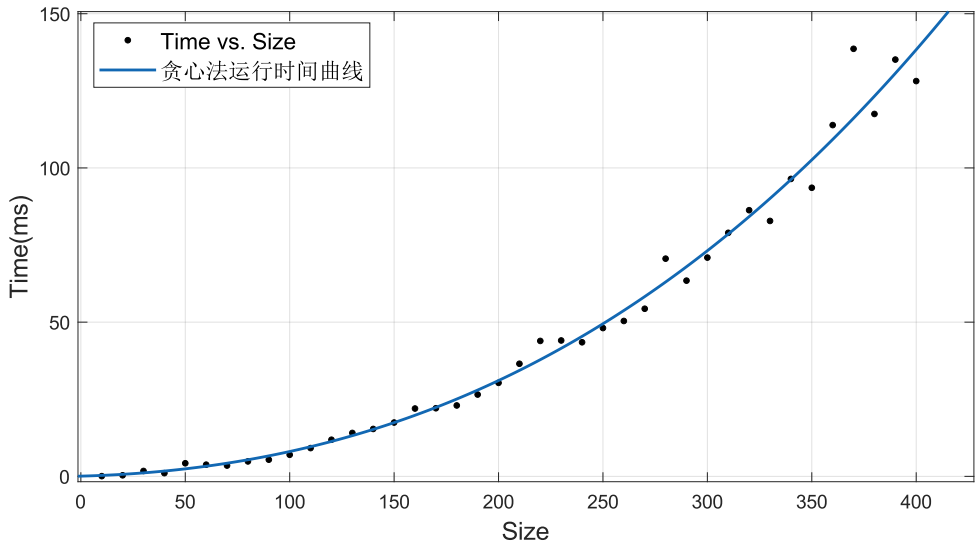

图2 贪心法运行时间随问题规模变化曲线

4 基于模拟退火法的超图最小顶点覆盖算法及分析

4.1 实验原理

模拟退火是一种概率算法，能够很好地求解最优化问题，在一定条件下可以得到问题的最优解。模拟退火算法的思想来源于固体物质的退火原理。固体物质的退火通常包含3个阶段：(1)加温阶段：将物体加热到临界温度以上某一温度；(2)保温阶段：物体与外界交换热量，使得物体全部软化;(3)冷却阶段：物体以大于临界冷却速度的冷速快速冷却，并随着温度的下降，冷却速度减缓。在凝结点附近温度下降得足够慢，物体趋向于最低能量的基态，从而获得最优的强度。

模拟退火算法就是从这种自然的退火过程中收到启发而得出的。在问题的初始阶段，先以某种办法迅速靠近最优解，极大减小搜索空间，等到达问题的最优解附近再仔细搜索，这就极大降低了获得问题最优解或近似最优解的成本。

模拟退火算法的思想可以描述如下：

表3 模拟退火算法

算法 3 Simulated Annealing
//S_i 表示当前状态，S_{i+1} 表示下一状态
//E(s)为在状态 s 时的能量
//r 为退火系数，控制降温速度,0<r<1
//T 为系统温度，初始为一个较高值
//T_{min} 为温度下限，当 T 达到 T_{min} 搜索停止
1 $while\ T > T_{min}$
2 $\Delta E \leftarrow E(S_{i+1}) - E(S_i)$
3 $if\ \Delta E > 0$ //新解为更优解
4 $S_i \leftarrow S_{i+1}$ //接受新解
5 else
6 if exp $(dE/T) > random(0,1)$
7 $S_i \leftarrow S_{i+1}$ //以概率exp (dE/T)接受新解
8 T← $r * T$

4.2 实验内容

根据模拟退火算法思想，针对超图的顶点覆盖问题，可以得出如表4所示的算法描述。

表 4 模拟退火求解超图最小顶点覆盖问题算法

算法 4 SA_MVC(V, E, T_0,T_{min}, r, L)
Input: 顶点集 V，边集 E，初始温度 T_0, 温度下限 T_{min}，退火系数 r, 迭代次数 L
Output: 最小顶点覆盖集 MVC
//bestSol 保存当前最优解，MVC 初始为 V，Deg(V)计算顶点集 V 所覆盖边的数目
1 $while\ T > T_{min}\ AND\ bestSol\ is\ not\ a\ cover$

```
    for k ← 1 to L
        v← random(0, N)   //随机选取顶点 v
        if v not in bestSol
            if Deg(bestSol ∪ {v}) > Deg(bestSol)
                bestSol ← bestSol ∪ {v}
        else
            dE ← Deg(bestSol − {v}) − Deg(bestSol)
            if dE ≥ 0 or exp (dE / T) > random(0,1)
                bestSol ← bestSol − {v}
        if bestSol is a cover AND |bestSol| < |MVC|
            MVC ← bestSol
    T ← T * r
return MVC
```

4.3 复杂性分析

时间复杂度具体与所设置的初始温度T_0，退火系数r，温度下限T_{min}，迭代次数L有关，与Deg(V)算法的复杂度有关(该复杂度取决于具体的数据集，本实验所用数据集边数等于顶点数目,故近似为$O(N^2)$)，也取决于一定的概率因素，可近似表示为$O(log_r \frac{T_{min}}{T} * L * N^2)$。若将L设置为$\sqrt{N}$的倍数，则时间复杂度为$O(N^{5/2})$；

只需要额外的几个大小为N的数组保存解，空间复杂度为$O(N)$。

4.4 算法运行测试

图 3 显示了模拟退火算法运行时间随问题规模的变化曲线，可以看出随着问题规模的增大，算法的运行时间是多项式递增的。

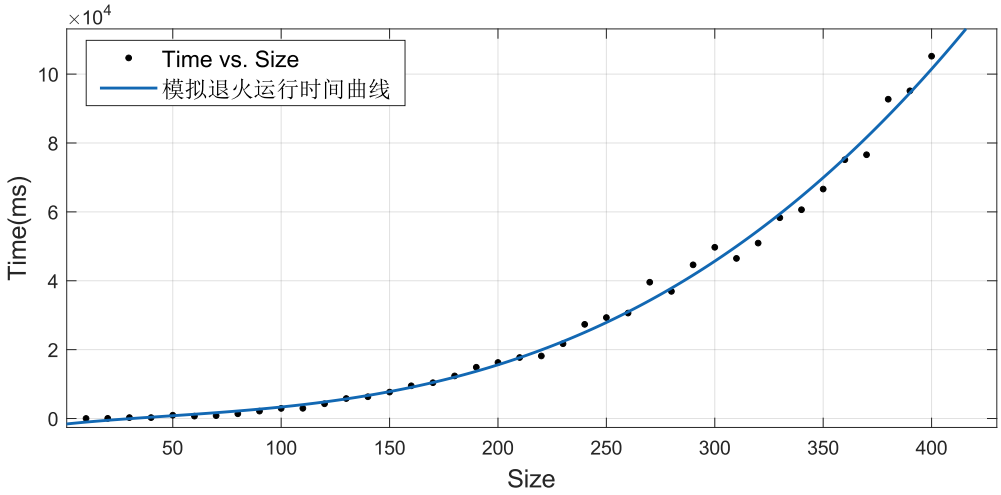

图3 模拟退火运行时间随问题规模变化曲线

5 仿真实验分析

表 5 将各算法的时间和空间复杂度作了比较，表 6 将不同规模下运行时间放在一起进行比较，从而可以对各算法的运行效率有一个直观的了解和比对。

各问题规模下每个算法运行 10 次取平均值。

表 5 超图最小顶点覆盖问题的不同算法时间复杂度和空间复杂度比较

算法	时间复杂度	空间复杂度
回溯法	$O(2^N)$	$O(N^2)$
贪心法	$O(N^3)$	$O(N)$
模拟退火算法	$O(log_r(T_{min}/T_0) * L * \ N^2)$	$O(N)$

注：模拟退火各参数取决于初始设置，当 Markov 链长 L 取$\sqrt{N}$的倍数时，算法的时间复杂度为$O(N^{5/2})$

表 6 不同问题规模下各算法运行时间比较(ms)

规模	回溯法	贪心法	模拟退火法
5	0.0774	0.0200	2.5340
6	0.1174	0.0377	3.7640
7	0.1973	0.0502	7.5663
8	0.4741	0.0662	6.7652
9	1.5864	0.0755	8.5255
10	4.0479	0.0725	13.550
11	7.5383	0.1101	29.692
12	18.246	0.1356	21.953
13	22.481	0.2313	17.450
14	55.546	0.1963	27.671
15	103.55	0.2671	21.080
16	365.02	0.3132	18.2943
17	556.29	0.2468	37.1869
18	1016.42	0.2303	60.7876
19	2018.70	0.3041	37.849
20	9721.29	0.3504	42.950
21	2953.23	0.4074	44.375
22	10468.2	0.7778	88.296
23	20839.1	0.8652	39.069
24	40757.8	1.1998	48.196
25	47915.4	0.9969	76.566
26	301785	1.1559	98.058
27	508992	1.3186	117.83
28	1004780	1.3828	112.315
29	1548120	1.3826	190.919

30	2038570	1.75419	265.84
50	NA	4.2503	961.71
100	NA	21.499	4478.2
200	NA	87.044	22967
300	NA	171.85	66269
400	NA	273.09	88919

可以看到，回溯法在问题规模达到50时，就很难运行出结果，无法应用到实际问题中。同等问题规模下，贪心法的运行时间最短，模拟退火算法的运行时间长于贪心法，且呈现摇摆的趋势，体现随机算法的特点。

由于近似算法只能得到局部最优解，所以除了运行时间外，与全局最优解的差距也应当作为衡量其性能优劣的重要指标，有必要对其所得到的近似最优解与全局最优解的差异程度进行评估。考虑用回溯所得解法作为精确对照，来衡量两近似算法与最优解的差距，从而比较其准确程度。

由于回溯法的最坏情况下时间复杂度为指数级，因此精准的比较测试只选取了顶点个数从 5 到 30 的小规模数据集。表 7 是用模拟退火算法运行 10 次所得到的最差、平均、及最好的近似最小顶点覆盖个数，将所得最优近似解与回溯法所得到最优解对比，得出其差距。表 8 将贪心法所得局部最优解与回溯法所得最优解进行比较，得出其差距。

表 9 是在问题规模稍大时，数据规模从 50 到 400 之间，将退火法与贪心法所得最优解相互比较，得出两者的准确性差异。

表 7 模拟退火算法运行 10 次所得解与最优解比较

问题规模	最差	平均	最好	最优解	相差
5	2	2	2	2	0
10	3	3	3	3	0
15	3	3	3	3	0
20	5	5	5	5	0
25	10	10	10	7	3
30	10	10	10	9	1

表 8 贪心法所得局部最优解与实际最优解比较

问题规模	局部最优解	最优解	相差
5	2	2	0
10	3	3	0
15	4	3	1
20	5	5	0
25	8	7	1
30	9	9	0

表 9 贪心法和退火法所得最优解相比较

问题规模	贪心法	退火法	相差
50	10	10	0
100	24	25	-1
200	56	57	-1
300	69	77	-8
400	258	258	0

从表中可以看出，在问题规模较小时，模拟退火算法几乎总能得到全局最优解。而贪心法还存在一些差异，所以当数据量较小时，模拟退火算法可以作为一个较好的近似。

而当数据量大时，可以看出贪心法与退火算法相比有更大优势，得出的顶点覆盖集更小，且运行时间也比退火算法大大减少。这与实际上随机算法的优势不相符，可能是由于我的水平不够写出来的退火策略不够好。

6 代码

回溯法：

```
#include <iostream>
#include <fstream>
#include<sstream>
#include <random>
#include <vector>
#include <Algorithm>
#include <chrono>

using namespace std;
using namespace chrono;
std::random_device rd;

vector<int> V; //顶点集
int sizeV ;   //顶点个数
vector<vector <int>> E; //边集
int sizeE ;  //边个数
vector<int> MVC; //保存最小顶点覆盖集

void init(string path)
{
   ifstream infile (path);
   string str;
   while(getline(infile, str))
   {
```

```
        istringstream input(str);
        vector<int> e;
        int v;
        while(input >> v)
            e.push_back(v);
        E.push_back(e);
        sizeE += 1;
        sizeV += 1;
    }
    for (int i = 0; i < sizeV;i++)
    {
        V.push_back(i);
        MVC.push_back(i);  //顶点覆盖集初始化为所有顶点
    }
}

//判断顶点集 V 是否是边集 E 的一个覆盖
bool IS_VC(vector<int> &V)
{
    vector<int> Esign(sizeE, 0);  //用向量取值 0 或 1 代表是否这条边是否被覆盖，用 0 初始化
    for (int j = 0; j < (int)V.size(); j++)   //遍历 V 中每个顶点
    {
        for (int i = 0; i < (int)E.size(); i++)    //遍历每条边
        {
            if (count(E[i].begin(), E[i].end(), V[j]))  //如果 V 中顶点出现在第 i 条边中
            {
                Esign[i] = 1;    //则将这条边的 Esign 设为 1
            }
        }
    }

    for (auto i : Esign)
    {
        if (i == 0)
        {
            return false;
        }
    }
    return true;
}

//回溯法求最小顶点覆盖
void BT(vector<int> &V, int v) //V 是顶点集,v 是要被删去的顶点,初始设为 0，即从顶点全集开始
{
```

```
    for (auto w : V)  //循环找到顶点集合 V 删去 v 个顶点的所有顶点子集
    {
        if (w >= v)
        {
            vector<int> C = V;  //将 V 复制给 C
            C.erase(remove(C.begin(), C.end(), w), C.end());  //从 C 中将 w 删去，得到新的 C 顶点子集
            //剪枝
            if (IS_VC(C))  //如果这个顶点子集 C 是顶点覆盖
            {
                if (C.size() < MVC.size())  //且规模小于之前找到的最小顶点覆盖集
                {
                    MVC = C;  //则更新最小顶点覆盖集
                }
                BT(C, w);  //回溯
            }
        }
    }
}


int main()
{
    init("C:\\Users\\YaliZhu\\Desktop\\50.txt");
    auto start = steady_clock::now();  //计时开始
    BT(V, 0);

    auto end = steady_clock::now();  //计时结束
    auto duration = duration_cast<nanoseconds>(end - start);
    double timeElapse = (double)duration.count() / 1000000.0;  //时长

    for (auto v : MVC)
        cout << v << ' ';
    cout << endl;
    cout << "Vertex num:" << MVC.size() << endl;
    cout << "The run time is "
         << timeElapse
         << "ms" << endl;

    return 0;
}
```

贪心法：

```
#include <iostream>
```

```
#include <fstream>
#include<sstream>
#include <random>
#include <vector>
#include <chrono>

using namespace std;
using namespace chrono;
std::random_device rd;

vector<int> V; //顶点集
int sizeV ;   //顶点个数
vector<vector <int>> E; //边集
int sizeE ;  //边个数

void init(string path)
{
   ifstream infile (path);
   string str;
   while(getline(infile, str))
   {
      istringstream input(str);
      vector<int> e;
      int v;
      while(input >> v)
         e.push_back(v);
      E.push_back(e);
      sizeE += 1;
      sizeV += 1;
   }
   for (int i = 0; i < sizeV;i++)
   {
      V.push_back(i);
   }
}

//遍历边集和 E，计算顶点 v 的度数
int Deg_v(int v)
{
   int deg = 0;
   for (auto e : E)
   {
      for (auto vi : e)
      {
```

```
            if (vi == v)
            {
                deg++;
                break;
            }
        }
    }
    return deg;
}

// 找度数最大的顶点，其位置索引即顶点名
int FindMax(vector<int> &degV)
{
    int MaxIndex = 0, MaxNum = 0;
    for (int i = 0; i < (int)degV.size(); i++)
    {
        if (degV[i] > MaxNum)
        {
            MaxNum = degV[i];
            MaxIndex = i;
        }
    }
    return MaxIndex;
}

//计算边集和中被覆盖的边的条数
int cover_E(vector<int> &MVC)
{
    int num = 0;
    for (auto i : MVC)
    {
        if (i == 1)
            num++;
    }
    return num;
}

vector<int> Greedy()
{
    vector<int> MVC(sizeV, 0);  //最小顶点覆盖集合，其中每个
    vector<int> degV(sizeV, 0); //保存每个顶点的度数
    vector<int> Esign(sizeE, 1);  //标志某条边是否被覆盖，若被覆盖取值为 0
    for (int i = 1; i <= sizeV; i++)  //计算每个顶点的度
    {
```

```
        degV[i - 1] = Deg_v(i);
    }

    //只要还有边没有被覆盖，就一直循环
    while (cover_E(Esign) != 0)
    {
        int index = FindMax(degV);  //找到度数最大的顶点
        degV[index] = 0;            //将其度数置为 0
        MVC[index] = 1;             //顶点覆盖集合中置为 1，标志该顶点选中

        //在边集合中，将该顶点连接的边标志为已覆盖，并将这些边中包含的顶点的度减 1
        for (int i = 0; i < sizeE; i++)
        {
            if (Esign[i] != 0)
                for (auto j : E[i])
                {
                    if (j == index + 1)   //如果 E[i]边中有这个度数最大的顶点
                    {
                        Esign[i] = 0;     //则将该边覆盖
                        for (auto v : E[i])  //同时边里面每个顶点度数减 1
                        {
                            if (degV[v - 1] >= 1)
                                degV[v - 1] -= 1;
                        }
                        break;
                    }
                }
        }
    }
    return MVC;
}

int main()
{
    init("C:\\Users\\YaliZhu\\Desktop\\50.txt");

    auto start = steady_clock::now();

    vector<int> MVC = Greedy();

    auto end = steady_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    double timeElapse = (double)duration.count() / 1000000.0;
```

```
    int Vnum = 0;
    for (int i = 0; i < (int)MVC.size(); i++)
    {
        if (MVC[i] == 1)
        {
            cout << i + 1 << " ";
            Vnum++;
        }
    }
    cout << endl;
    cout << "Vertex num:" << Vnum << endl;
    cout << "The run time is "
         << timeElapse
         << "ms" << endl;
    return 0;
}
```

退火法：

```
#include <Algorithm>
#include <chrono>
#include <iostream>
#include <fstream>
#include <sstream>
#include <random>
#include <vector>

using namespace std;
using namespace chrono;
std::random_device rd;

vector<int> V; //顶点集
int sizeV ;   //顶点个数
vector<vector <int>> E; //边集
int sizeE ;  //边个数

// 读入数据
void init(string path)
{
    ifstream infile (path);
    string str;
    while(getline(infile, str))
```

```
    {
        istringstream input(str);
        vector<int> e;
        int v;
        while(input >> v)
            e.push_back(v);
        E.push_back(e);
        sizeE += 1;
        sizeV += 1;
    }
    V = vector<int>(sizeV, 1);  //顶点集，用取值为 0 或 1 表示顶点是否存在
}


 //计算顶点 v 的度数
int Deg_v(int v)
{
    int deg = 0;
    for (auto e : E)
    {
        for (auto v1 : e)
        {
            if (v1 == v)
            {
                deg++;
                break;
            }
        }
    }
    return deg;
}

 //计算顶点集 V 总共覆盖的边的条数
int Deg_V(vector<int> &V)
{
    int deg = 0;
    vector<int> Esign(E.size(), 0); //标志边是否被覆盖的指示向量
    for (int k = 0; k < (int)V.size(); k++)
    {
        if (V[k] == 1)
            for (int i = 0; i < (int)E.size(); i++)
            {
                for (auto v1 : E[i])
                {
```

```
                if (v1 == k + 1)
                {
                    if (Esign[i] == 0)
                        Esign[i] = 1;
                    break;
                }
            }
        }
    }
    for (auto i : Esign)
    {
        if (i == 1)
            deg++;
    }
    return deg;
}

//计算顶点集 V 包含的顶点个数
int nV(vector<int> &V)
{
    int num = 0;
    for (auto i : V)
    {
        if (i == 1)
            num++;
    }
    return num;
}

vector<int> SA()
{
    int i, dD;

    float T0 = 100; //控制参数 t 的初值
    float T = T0;
    float r = 0.8f; //退火系数
    float T_min = 30;
    int L = 20*sqrt(sizeV); // Markov 链长度，每轮迭代次数

    vector<int> mvc(sizeV, 1);    //保存最终解
    vector<int> bestSol(sizeV, 0); //随机初始化解

    vector<int> tempSol(sizeV, 0); //
```

```
while (T > T_min || Deg_V(bestSol) < sizeE) //停止准则
{
    for (int k = 0; k < L; k++)
    {
        i = rd() % sizeV; //随机选取第 i 个顶点(实际为第 i+1 个)

        if (bestSol[i] == 0) //若 i 不在顶点集中
        {
            tempSol = bestSol;
            tempSol[i] = 1;
            if (Deg_V(tempSol) > Deg_V(bestSol)) //如果加入后的解覆盖的边数目更多
                bestSol[i] = 1;                  //将 i 加入顶点集中
        }
        else //i 在顶点集中
        {
            tempSol = bestSol;
            tempSol[i] = 0;
            dD = Deg_V(tempSol) - Deg_V(bestSol);

            if (dD >= 0 || (exp(dD / T) > (double)(rd() / (double)rd.max()))) //大于等于 0 或以 exp(dP/T)的
接受概率接受新解
            {
                bestSol[i] = 0;
                //bestSol[j] = 1;
            }
        }

        if (Deg_V(bestSol) == sizeE)
        {
            //if (Deg_V(bestSol) == sizeE)
                cout << "Find a cover:";
            for (int i = 0; i < (int)bestSol.size(); i++)
            {
                if (bestSol[i] == 1)
                    cout << i + 1 << " ";
            }
            if (nV(bestSol) <= nV(mvc))
            {
                mvc = bestSol;
            }
            cout << endl;
        }
    }
    T = T * r;
```

```
    }
    return mvc;
}


int main()
{
    init("C:\\Users\\YaliZhu\\Desktop\\10.txt");
    auto start = steady_clock::now();

    vector<int> bestSol = SA();

    auto end = steady_clock::now();
    auto duration = duration_cast<nanoseconds>(end - start);
    double timeElapse = (double)duration.count() / 1000000.0;

    cout << "The minimum vertex cover is:";
    int Vnum = 0;
    for (int i = 0; i < (int)bestSol.size(); i++)
    {
        if (bestSol[i] == 1)
        {
            cout << i + 1 << " ";
            Vnum++;
        }
    }
    cout << endl;
    cout<<"Vertex num:"<<Vnum << endl;
    cout << "The run time is "
        << timeElapse
        << "ms" << endl;
    return 0;
}
```

生成超图数据集（Python）：

```
import random

nv=5

def main():
    n=1
    hg=[]
    while True:
```

```
        while(n<=nv):
            dege=random.randint(2,4)
            list1=random.sample(range(1,nv+1),dege)
            hg.append(list1)
            n+=1
        if (len(list(set([i for item in hg for i in item])))==nv):
            return hg
        else:
            hg=[]


if __name__ == '__main__':
    hg=main()
    for i in hg:
        for j in i:
            print(j,end=' ')
        print('\n',end='')
    with open('C:\\Users\\YaliZhu\\Desktop\\'+str(nv)+'.txt','w',encoding='utf8') as f:
        for i in hg:
            for j in i:
                print(j,end=' ',file=f)
            print('\n',end='',file=f)
```