# 基于图论模型的(n,K)网络的容错性问题研究

**摘要**

本文针对ｋ元ｎ维冒泡排序网络故障检测问题，建立基于故障检测矩阵的故障诊断算法，实现对k元n维冒泡排序网络结构下专用的故障检测与分析手段，并且基于二分图建模出了集合覆盖模型，并证明了问题3和问题4在满足一种特殊情况时可以把问题转化成一个精确求解问题，进而求出了子网络数量以及需要破坏的节点数量，后续提出对问题规模分类成精确覆盖问题和集合覆盖问题，针对这两个问题给出了具体的算法用来求可能解。

针对问题一和问题二，根据附件3和附件4，我们可以知晓任意一个节点收到故障指认和发出故障指认的数目，根据这两个因素我们可以发现了四个可以确定某些节点是否为故障节点或者正常节点的简易的验证规则。根据已经确定的节点可以通过简单的DFS知道未确定节点的状态，可以证明整个过程时间复杂度为O(n²)。

针对问题三和问题四，我们把顶点集合和子网络集合分别当作一个部集，建模出了二分图的模型，要求选择一个顶点集合的最小子集保证子网络集合每个点都有至少一条边相连，通过“Hall婚配定理”可以证明在给出的问题的规模下这其实可以是一个精确覆盖问题，即存在一个二分图的最大匹配保证所有的子网络都被覆盖。

针对问题五，我们根据证明的结论可以实现对问题的规模分类，分类成精确覆盖问题或者贪心问题，从而选择针对于集合覆盖问题的贪心算法和针对于精确覆盖的二分图最大匹配问题进行求解。

**关键词**：网络拓扑；故障诊断；二分图最大匹配；Hall婚配定理；精确覆盖。

# 一、问题的背景与重述

## 1.1问题背景

并行分布式多处理器系统的基础拓扑通常以图为数学模型，这类系统由多个处理器组成，它们通过通信网络相互连接，共同完成计算任务。在并行分布式系统中，处理器之间的连接式可以用图来表示，其中顶点代表处理器，边代表它们之间的通信联系。不同的网络拓扑结构会影响系统的性能和可靠性。网络作为多处理器系统的主要候选拓扑之一而得到广泛关注。

一个多处理机系统在运行过程中一些处理器（顶点）发生故障是不可避免的，因此系统会经常启用自诊断测试生成测试报告用于识别故障顶点。为了识别故障处理器，系统会进行自诊断测试。每个处理器会测试与之相邻的处理器，并报告其状态。然而，如果测试处理器本身存在故障，其报告可能不可靠。

问题要求根据自诊断测试报告，识别出网络中的故障顶点，这涉及到对测试数据的分析和推理，需要设计有效的算法来处理不确定和矛盾的信息。除了识别故障顶点，问题还考察了对网络结构的理解和分析能力。参赛者需要确定网络中包含的不同子网络数量，以及如何通过破坏最少数量的顶点来破坏所有子网络，这涉及到图论和网络分析的知识。

在这样的背景下，需要进行数学建模和分析，以确定系统中的故障处理器，并评估不同的容错策略对系统性能的影响。解决这些问题对于提高并行分布式系统的可靠性和稳定性具有重要意义，尤其是在面对大规模、高复杂度的计算任务时。

## **1.2问题重述**

问题一：附件3是网络的一次自诊断测试所产生的报告。依据经验，已知在这个网络中故障顶点个数不会超过5，请根据报告求出这个网络中的所有故障。

问题二：附件4是网络的一次自诊断测试所产生的报告。依据经验，已知在这个网络中故障顶点个数不会超过15，请根据报告求出这个网络中的所有故障。

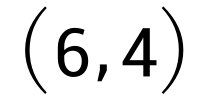
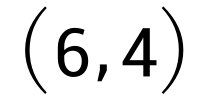
问题三：网络包含多少个不同的子网络? 要破坏网络中的所有子网络，至少需要破坏多少个顶点，如何破坏？

问题四：网络包含多少个不同的子网络? 要破坏网络中的所有子网络，至少需要破坏多少个顶点，如何破坏？

问题五:要通过破坏尽可能少的顶点达到破坏网络中的所有子网络的目的，有什么好的破坏点的策略？

# 二、 问题分析

## 2**.1 问题一的分析**

附件1是网络的邻接矩阵。附件3是网络的一次自诊断测试所产生的报告。根据顶点报告机制以及故障顶点个数限制，如果一个顶点 u 无故障，则它准确报告与之相邻的顶点的故障状态，如果顶点 u 有故障，则其报告不可信，可能随机报告相邻顶点的故障状态。通过以上约束，我们可以设计一种基于故障检测矩阵的故障诊断算法，用于识别可能的故障顶点集合。具体验证规则包括如下：

对附件3进行读入获得故障检测矩阵M。其中**M(i,j)**意思为i对j的检测结果，0代表i与j不相邻，1代表i认为j无故障，-1代表i认为j故障

做以下四种分类:

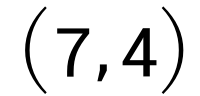
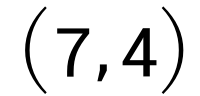
1. 顶点u报告某个邻居节点故障，并且某个邻居也报告u故障。
2. 顶点u报告某个邻居节点故障，但是全部邻居报告u无故障。(有前提)
3. 顶点u报告所有邻居节点无故障，但是某个邻居报告u故障。
4. 顶点u报告所有邻居节点无故障，并且全部邻居报告u无故障。(有前提)

后续文章中可以证明，以上分类对应了自身或邻居节点的相关故障检测矩阵性质，可以帮助我们很快确认一些节点的状态是故障的还是非故障的。

根据已经确定的节点的状态我们可以遍历没有确定状态的节点，根据其邻居的状态确认自身状态。

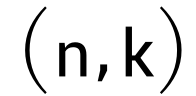
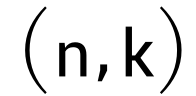
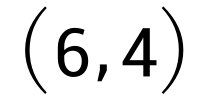
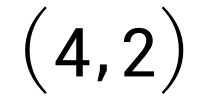
结果为{1426,3416,4216,5416,6412}。

## 2.2 问题二的分析

附件 1 是网络的邻接矩阵。附件4是网络的一次自诊断测试所产生的报告。可以采取与问题一相同的策略。

{2361,2465,2643,2763,3741,4523,4527,4756,5162,5716,6153,7653}是这个题的答案，总共为12个故障点。

## 2.3 问题三的分析

问题三要求根据网络的层次结构，计算网络中由顶点子集诱导的子图是子网络，具体要求是计算网络中包含的所有子网络数量，并且找到最小的破坏顶点数量，使得所有顶点被破坏。

关于求子网络数量这个过程，一个子网络是由两个数列，p数列和b数列定义的，根据p数列和b数列的数量根据乘法原理可以知道子网络的数量。

我们把顶点集合和子网络集合分别当作一个部集，可以按照顶点和子网络的包含关系建模出一个二分图的模型，题目要求选择一个顶点集合的最小子集保证子网络集合每个点都有至少一条边相连，但是发现通过“Hall婚配定理”可以证明特殊情况下这个问题可以是一个精确覆盖问题，即存在一个二分图的最大匹配保证所有的子网络顶点都被覆盖且只被覆盖一次。发现问题三满足特殊情况的要求，因此问题三可以是一个精确覆盖问题。

问题三中子网络数量为90，认为至少需要破坏30个点就可以使得全部子网络被破坏。

## 2.4 问题四的分析

问题四与问题三要求一致，问题三已经建立了一个正确的模型，问题四依然满足这个模型，可以手算出来。

子网络数量为142,800，认为至少需要破坏4080个顶点可以处理所有子网络。

## 2.5 问题五的分析

对于问题三中分析出模型进行了推广，可以把一些集合覆盖问题特殊化成精确覆盖问题，对于特殊问题和一般问题分开求解即可。

# 模型假设

（1）假设所有处理器之间的消息传递是可靠的，没有消息丢失或错误传递的情况发生。

（2）假设每个处理器独立地进行测试，并且其测试结果不会受到其他处理器状态或测试结果的影响。

（3）假设处理器的故障状态在整个自诊断测试期间保持不变，即处理器不会在测试过程中从无故障变为故障或从故障变为无故障。

（4）假设测试图是连通的，即任意两个处理器之间总是可以通过一系列相邻处理器相连，从而确保所有处理器的状态能够被检测到。

（5）假设如果处理器 A 能够测试处理器 B，那么处理器 B 也能够测试处理器 A。这一假设符合无向图模型中的邻接矩阵定义。

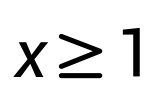
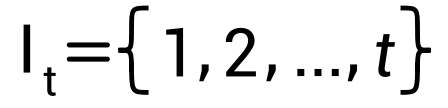
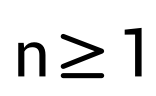
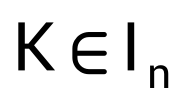
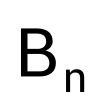
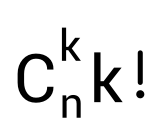
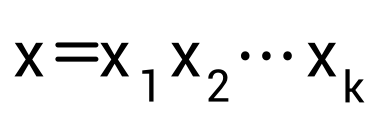
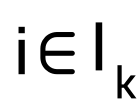
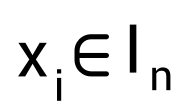
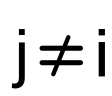
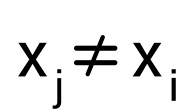
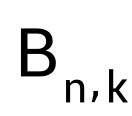
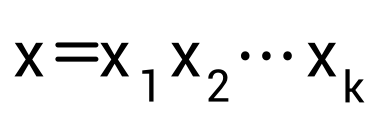
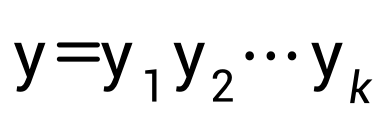
（6）假设自诊断测试过程不受外界环境因素（如温度、电磁干扰等）的影响，所有 测试在相同条件下进行。

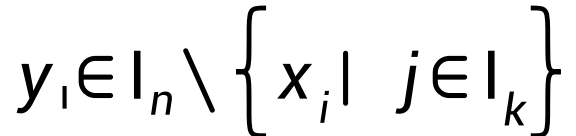
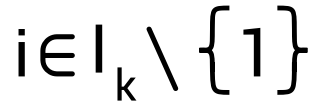
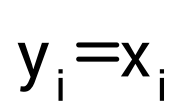
# 四、符号说明

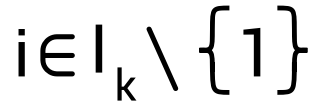
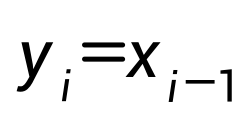
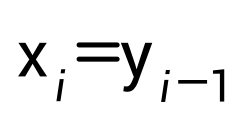
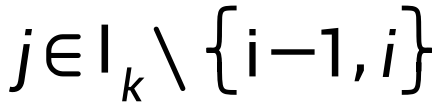
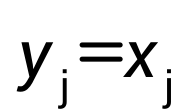
# 五、模型的建立与求解

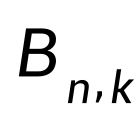
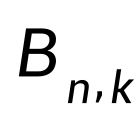
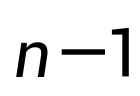
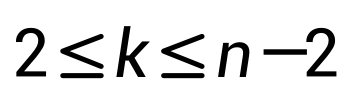
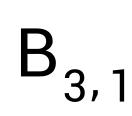
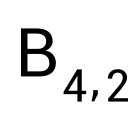
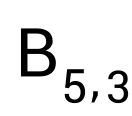
## 5.1 基于故障检测矩阵的故障诊断算法与问题1求解

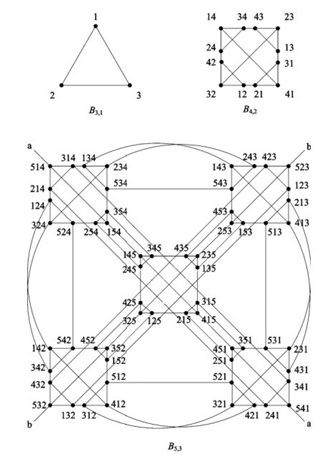
### 5.1.1 ｋ元ｎ维冒泡排序网络的原理

查阅文献可知latexmath元latexmath维冒泡排序网络是一种用于排序的网络结构，适用于任意整数。我们记。对于任意给定的和，latexmath元latexmath维冒泡排序网络是一个具有个节点的无向图。其中，每个节点的形式如下：，满足对于任意的，都有，且对于任意的，都有。[[[1]](#endnote-0)] 中的两个不同节点和是相邻的，当且仅当满足以下两个条件之一：

1. ,并且对于对任意的均有成立

（2）存在一个整数，使得，,并且对于任意的，都有

当节点latexmath和latexmath因满足条件（1）而相邻时，称连接它们的边为 1-边；因满足条件（2）而相邻时，称连接它们的边为i-边。由上述定义可知，是 正则且节点对称的。当latexmath取０和１时，latexmath元latexmath维冒泡排序网络分别约简为ｎ个点的完全图网络和传统的latexmath维冒泡排序网络[[[2]](#endnote-1)]。在无特殊说明的情况下latexmath元latexmath维冒泡排序网络均满足。图 1 给出了网络,,。

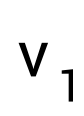
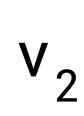
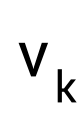


**图1 ｋ元ｎ维冒泡排序网络实例**

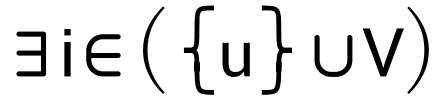
可以知道这个图的一个重要性质，就是每个点之间都是自同构的，也都是可以互相替换的。这个性质十分重要，可以知道每个节点的度数都是相同的。

### 5.1.2 基于点的性质确定点的状态

我们希望有一些方法直接确认一些点的性质，利用已经确定的点的性质去反推不确定的节点的性质。

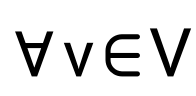
我们规定一个状态函数S(u)表示节点的性质，如S(u)=1,则认为u节点是非故障的,如果S(u)=-1,则表示u节点是故障的，如果S(u)=0,则表示节点的状态不确定。同时规定u的邻居节点用V={,,...,}集合表示。

我们发现根据节点收到-1和发出-1的数量可以进行节点性质的确认。

1. 顶点u报告某个邻居节点故障，并且某个邻居也报告u故障。那么节点u本身和节点u的邻居节点，一定有满足S(i)=-1，但是无法确定具体是哪几个节点，因此他们的状态函数都赋值为0。
2. 在当前节点和所有邻居节点至少有一个节点不为故障点的前提下，顶点u报告某个邻居节点故障，但是所有邻居报告u无故障。那么可以知道S(u)=1。 证明:在当前节点和所有邻居节点至少有一个节点不为故障点的前提下，假设u是坏节点并且u没有报告故障，那么就知道u周围没有故障点，但是由于具体情况是当前顶点u和邻居i至少有一个非故障点，这是矛盾的，可证点u必为非故障节点。
3. 顶点u报告所有邻居节点无故障，但是某个邻居报告u故障。那么S(u)=-1。

证明:假设当前节点是非故障的，还收到举报，那么邻居必然有一个是故障的，所以当前节点应该会报告有一个邻居有故障，但是没有报告，所以当前节点是非故障的假设不成立，所以当前节点是故障的。

1. 在当前节点和所有邻居节点至少有一个节点不为故障点的前提下，

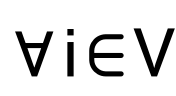
点u报告所有邻居节点无故障，并且全部邻居报告u无故障。那么S(u)=1，并且对于有S(v)=1。

证明:假设当前节点u是故障点，在约束前提下必然会收到一次举报，但是没有收到举报，所以点u不能是故障点。而该点又没有认为其他点是故障点，因此该点和邻居都是非故障的。

### 5.1.3 深度优先搜索

在上文的辅助下，有部分节点的性质可以被确定了，因此可以从所有状态函数S(u)=0的点开始向外搜索。

规则如下:

1. 如果当前节点u的邻居v的状态函数S(v)=0，那么再对v继续进行深度优先搜索。在回溯之后可以确定v的状态。
2. 如果找到当前节点u的邻居v的状态函数是S(v)=1，那么可以根据M(v,u)的值确定u的状态，之后进行回溯。
3. 如果找到当前节点u的的状态都是S(i)=-1,那么可以根据M(u,i)是否全部等于-1，判断节点是否是故障点。

### 5.1.4 问题1结果与模型分析

(1)结果分析

问题1的每个节点的度数是5,最大故障点数目为5，因此满足约束的前提，当前节点u和所有邻居节点i至少存在一个非故障点。因此是可以直接使用的。

获得结果为{1426,3416,4216,5416,6412}，经过验证，符合题意。

(2)时间复杂度和有效性分析

本模型处理处理一个节点的收到的-1和发出的-1的数量的过程需要遍历M矩阵，时间复杂度是O(n²)。

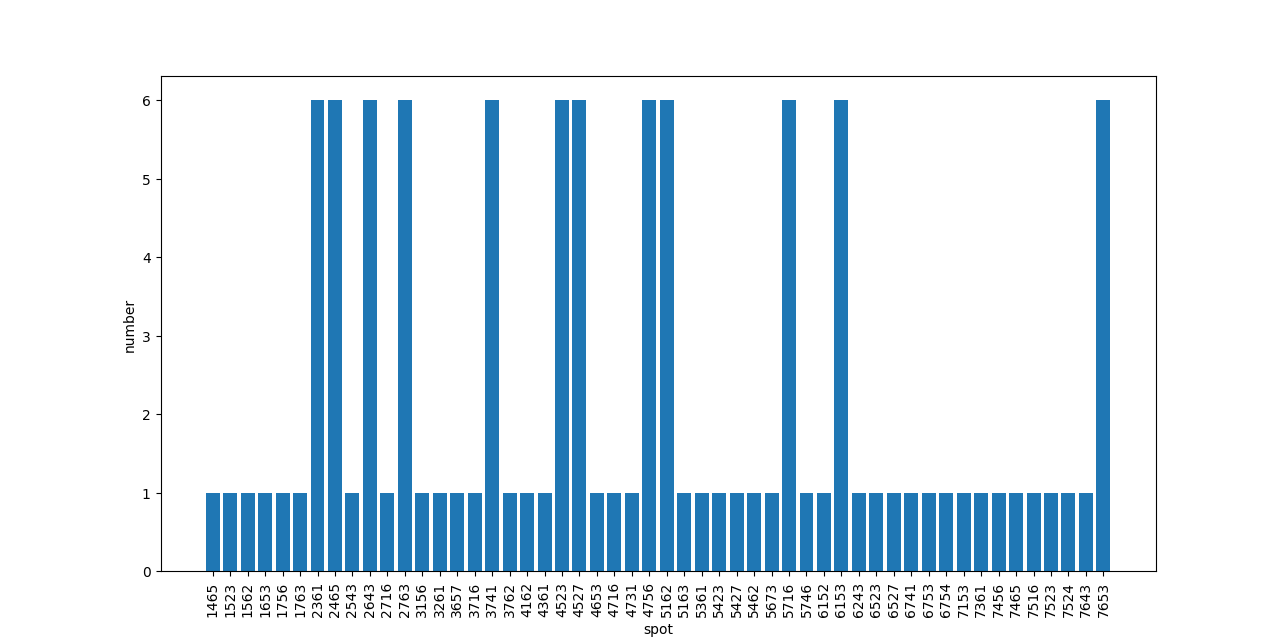
DFS的时间复杂度是O(n+m)。

总体复杂度就是O(n²)。

## 5.2 问题二节点故障诊断模型的建立与求解

### 5.2.1 对整个(7，4)网络的预处理

在5.2.1中由于每个顶点度数为6，但是故障点最大限为15，因此不能直接保证当前顶点和邻居节点之间至少一个非故障点，因此要根据数据结果进行分析，大致确定可能故障点的分布，找出不能使用规则(2)和规则(4)的节点然后再处理确定的点。

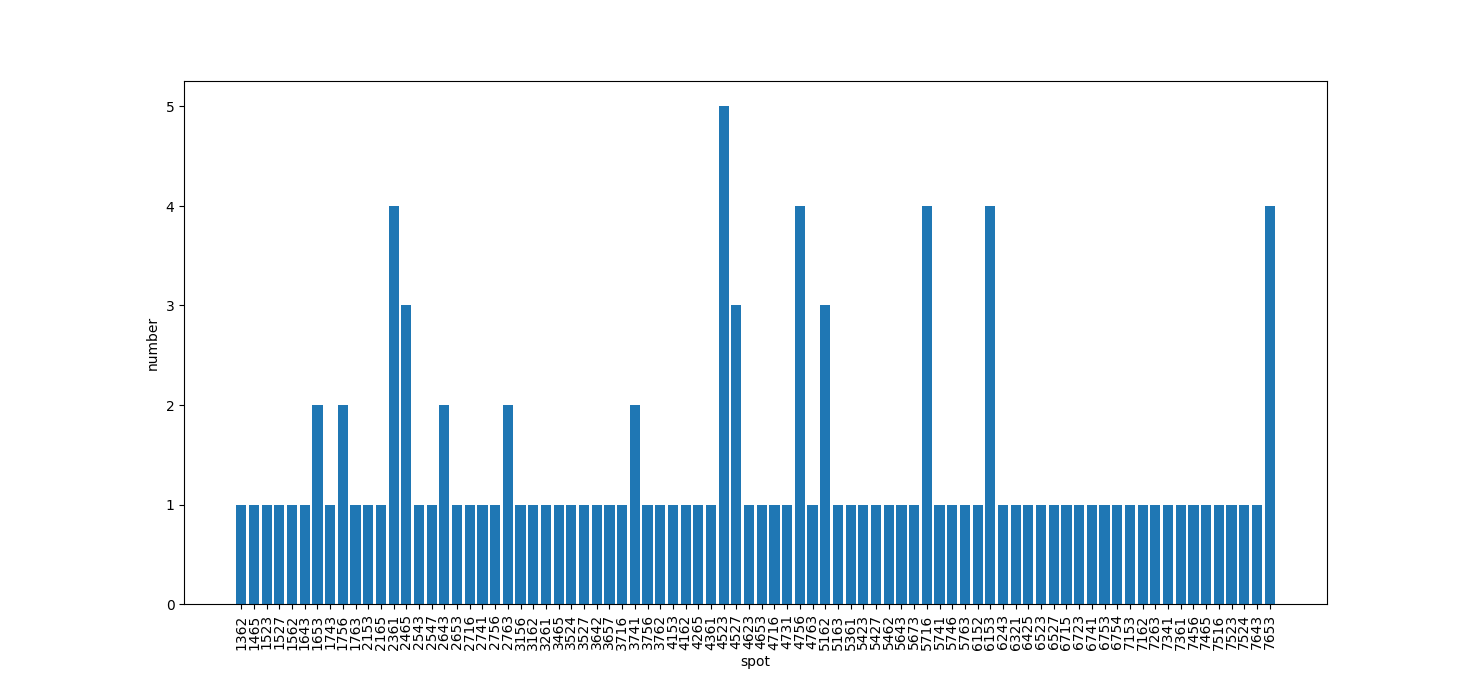
我们可以先统计出每个节点被指认为故障节点的数量。

**图2 收到故障指认的节点的名称和数量**

图中共12个点被所有邻居指认为故障点，如果当前节点u被所有邻居指认为故障点，假设u的邻居存在一个非故障点，因此有一个非故障点指认u为故障点，所以u是故障点成立。假设u的邻居不存在非故障点，而所有邻居都认为u是故障点，如果u认为所有的邻居都是故障的那么满足假设条件，u点一定是个非故障点，反之u是个故障点。

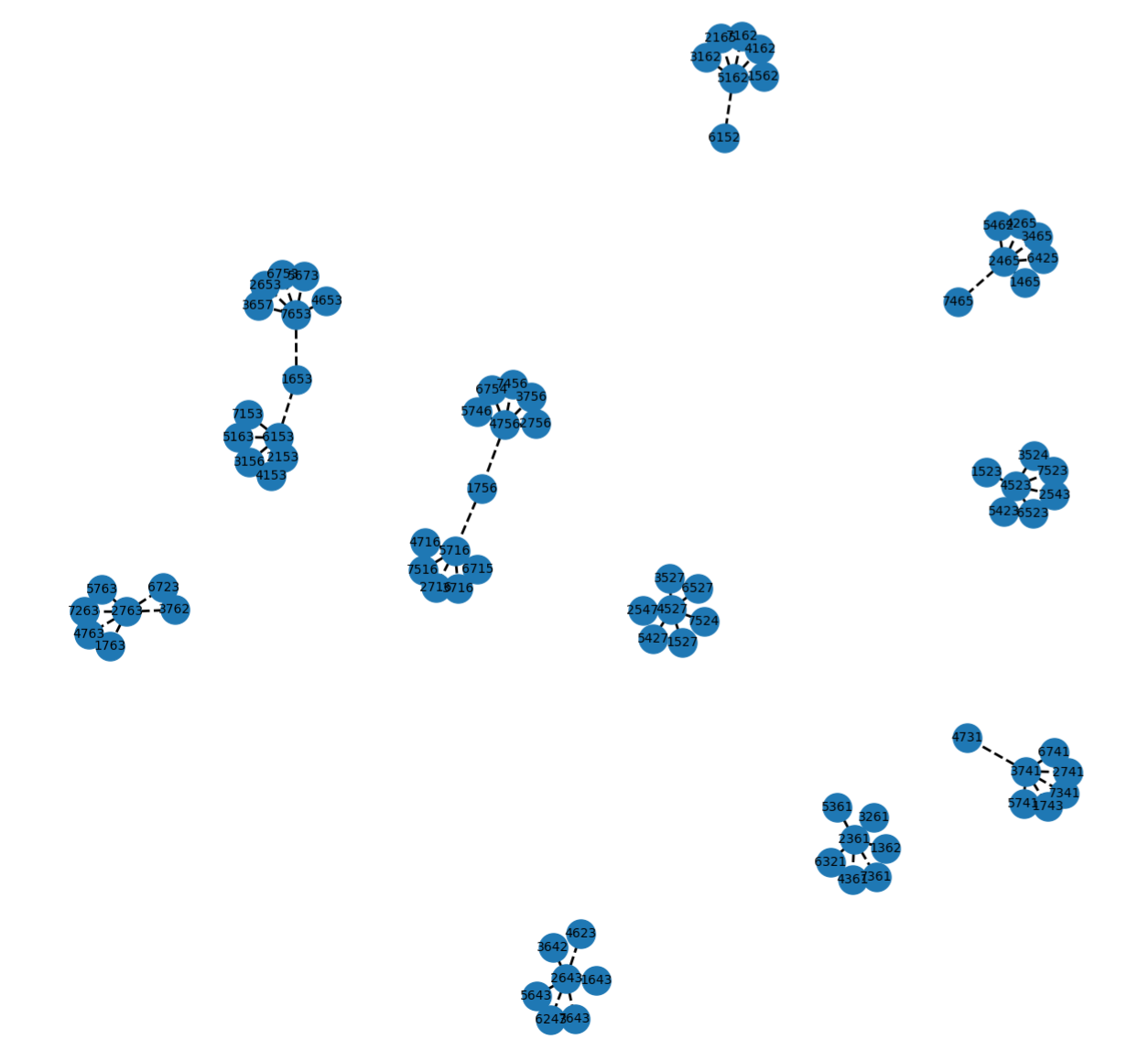
整个假设的意思就是如果这个点收到了6个故障点指认又发出了6个故障点指认，那么这个点的状态无法确定是故障点还是非故障点，否则这个点就一定是个故障点。

因此只需要统计这12个点指认的故障点数量，根据每个点指认的故障点数量是否等于度数6即可判断，如果不等于6那么就知道当前点u一定是个故障点，否则是一个故障点。



**图3 发出故障指认的节点的名称和数量**

根据对比，发现这些节点都是故障点，对这些故障点可视化生成点诱导子图。



**图4 故障点的点诱导子图**

发现联通块数量是10个，根据抽屉原理知晓最多也就是三个节点在一个点的菊花图内，由于还有可能存在的3个故障点未发现，因此最多一个节点的菊花图内最多存在6个故障节点，也就是当前节点u和邻居节点V之间至少存在一个节点保证属于非故障点。这符合我们四条性质的前提条件。

因此所有节点都可以使用四条性质去确认结果。

### 5.2.2 问题2结果与模型分析

结果为{2361,2465,2643,2763,3741,4523,4527,4756,5162,5716,6153,7653}，总共12个点，经检验该结果正确。

对于本模型，由于题目特殊性质，导致没有展现不满足前提的条件下，该模型如何使用，其实只需要经过类似于5.2.1的预处理对可能不满足前提的节点以及邻居节点标记状态为不确定状态就好，对满足前提的节点正常使用四条性质即可。只要能保证找到确定状态为非故障的节点就可以用来DFS求解。

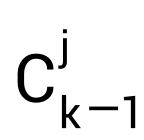
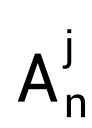
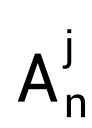
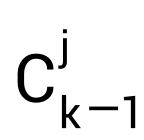
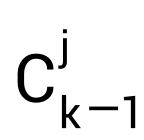
时间复杂度O(n²)。

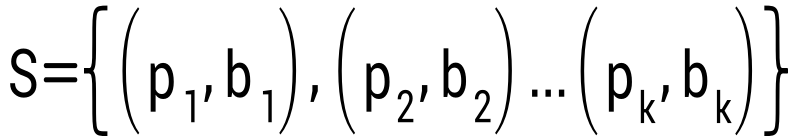
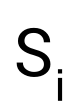
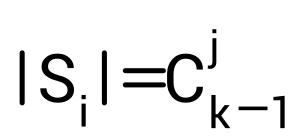
## 5.3 二部图集合覆盖模型的建立与问题3求解

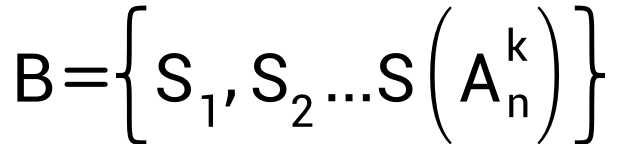
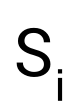
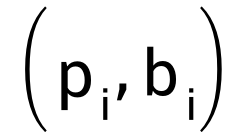
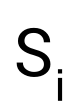
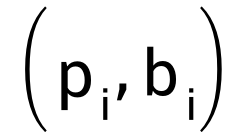
### 5.3.1 建模成集合覆盖问题

对于问题三四五，我们可以对问题进行转化，每个顶点同时存在于多个子网络中，破坏掉该顶点那么其对应的子网络也会被破坏掉，每个顶点对应的子网络都是全部子网络的子集，因此问题选择最少的顶点使得所有子网络被破坏可以转化成选择最少的子集数量使得所有集合被覆盖问题。

### 5.3.2 二分图建模集合覆盖问题

每个子网络都由数列p和数列b确定，对于数列p， 是个不同的数且,对于数列b， 是中的个不同的数。数列p的数量是,数量b的数量是，子网络的数量就是。每个顶点对应的子网络的数量就是序列p的数量。这样子网络的数量就可以确定了。

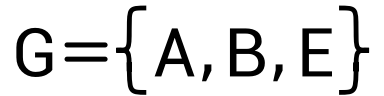
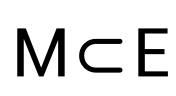
令集合包含了对所有子网络的描述，那么每个顶点都对应了S的一个子集，其中。

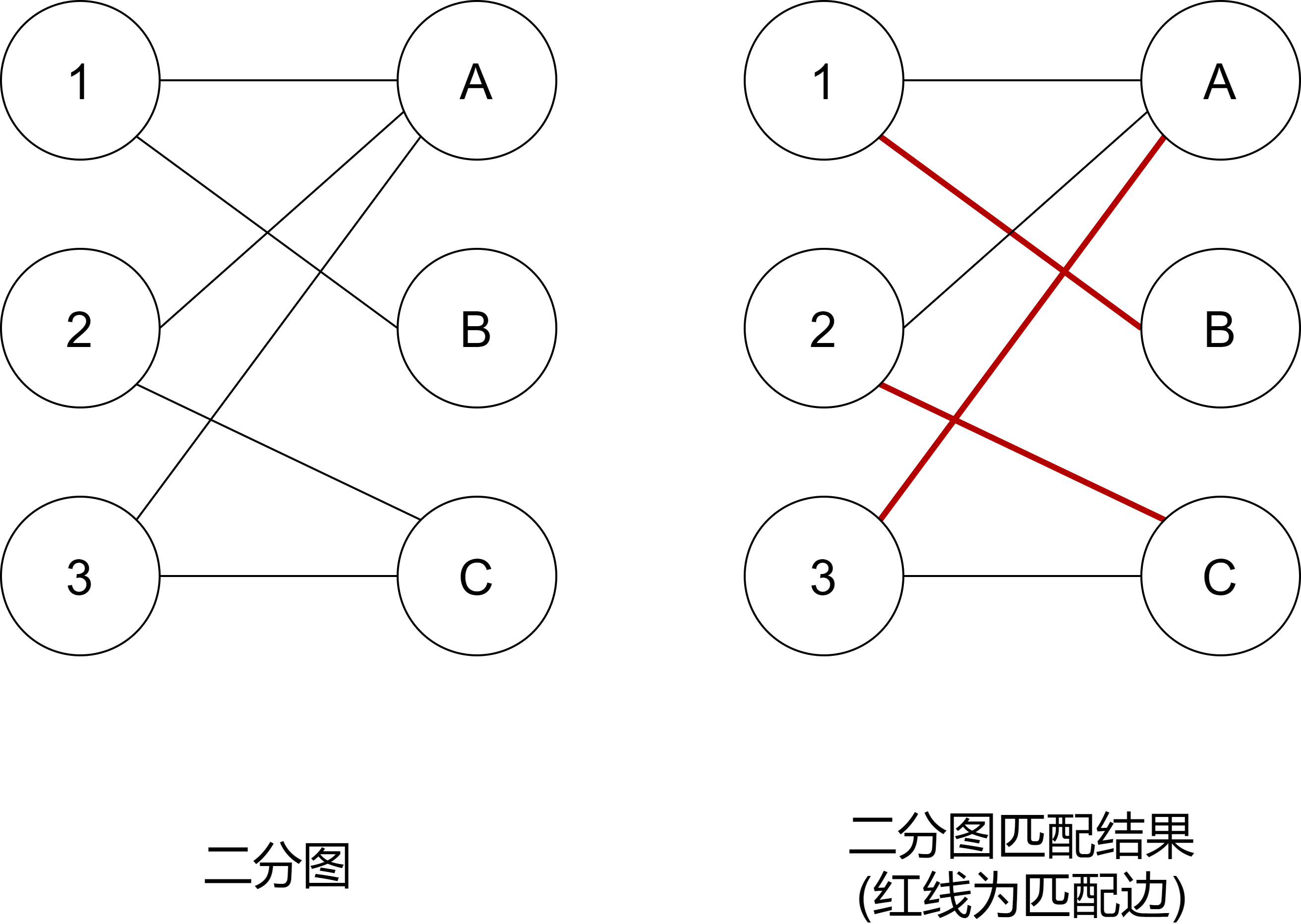
设集合为所有顶点的集合，每个顶点的描述是集合S的一个子集,令集合B为一个部级，集合S为另一个部级，如果顶点所对应的子集包含了子网络那就为子集和对应的子网络连上线。最后构成了一个二部图。

问题三这个集合覆盖问题可以表述为需要选择B中的几个点并生成一个点诱导子图使得S中全部的点均被选中。

### 5.3.3 hall婚配定理与二部图的完备匹配

二部图的完备匹配的概念是从匹配的概念上延伸出来的。

设二部图，匹配的意思就是保留所有的点不变，选择一个边集的子集M使得同时保证每个点的度不超过1。



**图5 二分图匹配的介绍**

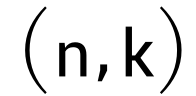
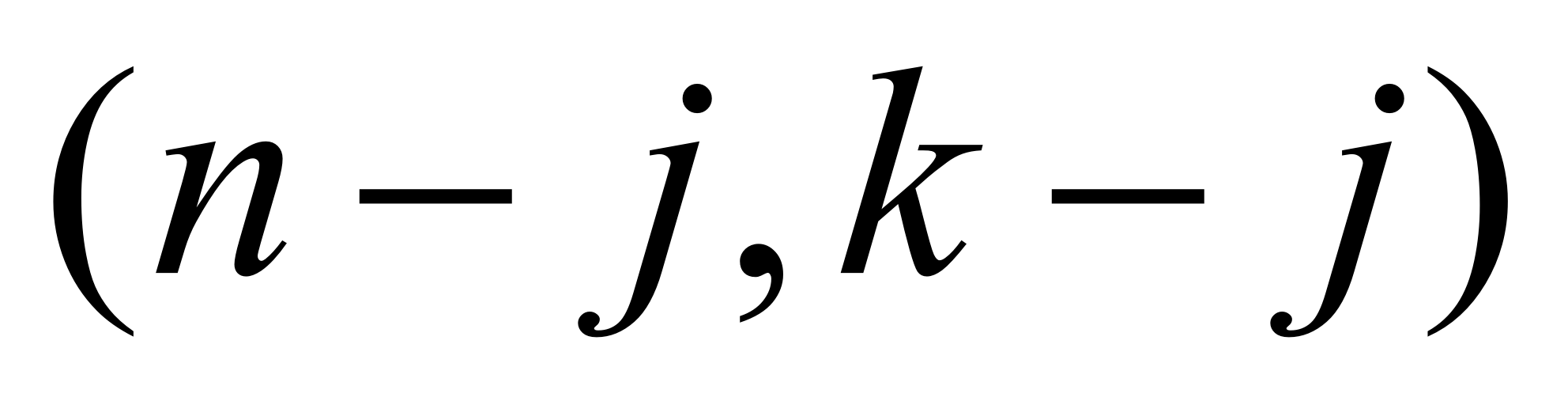
完备匹配的意思则是，存在一个匹配M存在，并且保证所有的B点的度都为1,图五中右图描述的匹配也是一个完备匹配。

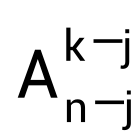
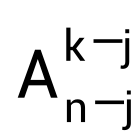
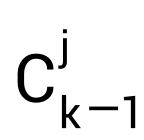
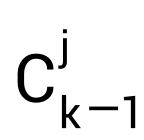
那么如何判断一个二分图存在完备匹配呢？

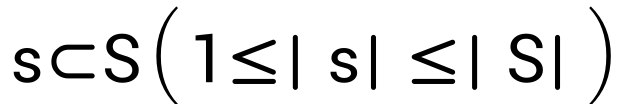
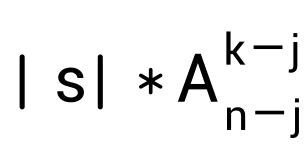
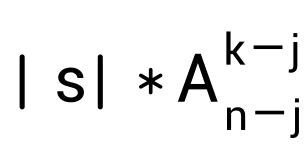
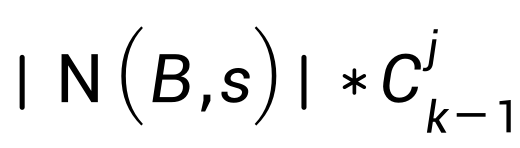
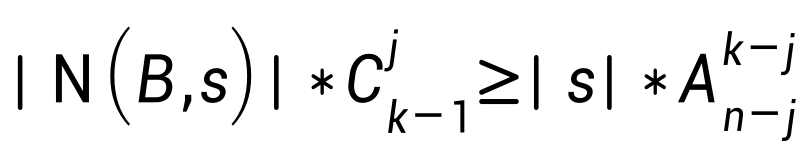
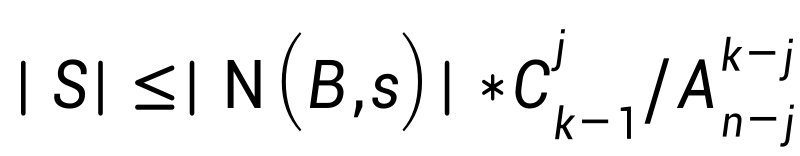
那就要用到hall婚配定理。此定理使用于组合问题中；[二部图](https://baike.baidu.com/item/%E4%BA%8C%E9%83%A8%E5%9B%BE/0?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%9C%8D%E5%B0%94%E5%AE%9A%E7%90%86/_blank)G中的两部分顶点组成的集合分别为集合X,集合Y,图G中有一组无公共点的边即一个匹配M使得一端恰好为组成X的点的[充分必要条件](https://baike.baidu.com/item/%E5%85%85%E5%88%86%E5%BF%85%E8%A6%81%E6%9D%A1%E4%BB%B6/10943559?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%9C%8D%E5%B0%94%E5%AE%9A%E7%90%86/_blank)是：X中的任意k个点至少与Y中的k个点相邻。（1≤k≤m）本定理是[二分图匹配](https://baike.baidu.com/item/%E4%BA%8C%E5%88%86%E5%9B%BE%E5%8C%B9%E9%85%8D/0?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%9C%8D%E5%B0%94%E5%AE%9A%E7%90%86/_blank)问题中[匈牙利算法](https://baike.baidu.com/item/%E5%8C%88%E7%89%99%E5%88%A9%E7%AE%97%E6%B3%95/0?fromModule=lemma_inlink" \t "https://baike.baidu.com/item/%E9%9C%8D%E5%B0%94%E5%AE%9A%E7%90%86/_blank)的基础。[[[3]](#endnote-2)]

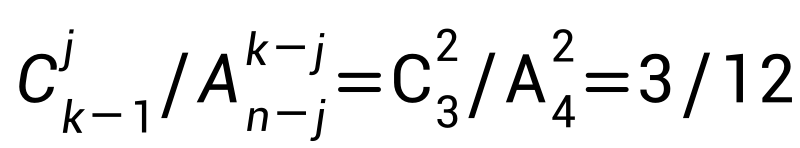
Hall婚配定理的意思是只要可以证明对于集合B的任意k个点所连接到的集合A中的点的数量大于等于k即可使得存在一个完备匹配，使得B中的点全部被选中且只有每个点只有条边相连。

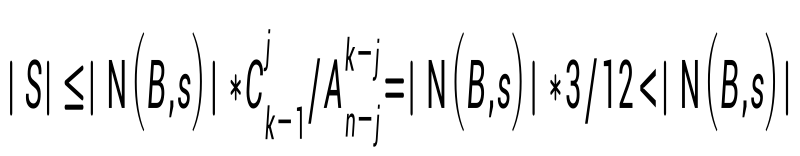
### 5.3.4 问题3的完备匹配证明

在5.3.1中，已经把问题三描述成一个二部图G了，G={S,B,E}。并且已经把集合覆盖问题转换成了一个二部图中的选择问题，接下来我希望证明题意中的P网络以及对应的P子网络之间构成的集合覆盖问题可以转化成一个二分图的完备匹配问题。

已知集合S中每个点的度为即为一个子网络可以包含个顶点，同时每个点的度为，即为一个顶点属于个子网络中。

假设选择任意集合S的子集s，,现在知道s产生的边的数量一定是,根据握手定理得知集合B中与s产生连接到点的集合**N(B,s)**与s之间有个边相连，但是**N(B,s)**向集合S发出的边的数量是，N**(B,s)**向集合S发出的边的数量一定是大于等于**N(B,s)**与s之间的边的数量的，即为。那么就有。

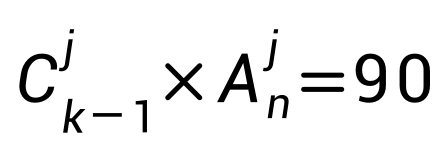
在问题三中**k=4,n=6,j=2**,可知。

因此。

则证毕，问题三中的集合覆盖问题可以转换成一个精确覆盖问题求解。

### 5.3.4 问题3的解

子网络数量:

由5.3.2的公式可知子网络的数量就是。

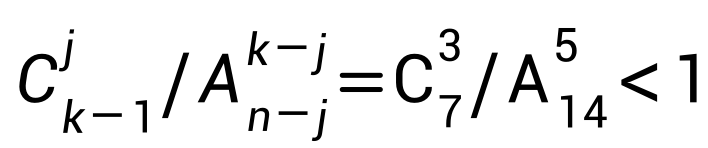
破坏节点数目:

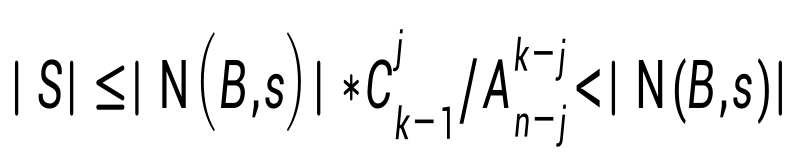
问题三现在已经可转化成一个精确覆盖问题求解了，由于每个顶点都对应了3个子网络，子网络个数是90，已知可以达成一个精确覆盖，那么只需要选择30个顶点就可以实现精确覆盖了，也就是至少破坏30个节点即可破坏所有子网络。

## 5.4 问题4精确覆盖问题的证明与问题四的解

### 5.4.1 问题4的建模和完备匹配证明

对问题4中采取和问题3一样的二分图建模。

问题4中k=8,n=17,j=3，可知。

因此。可以根据上述结论知道，问题4依然满足完备匹配。

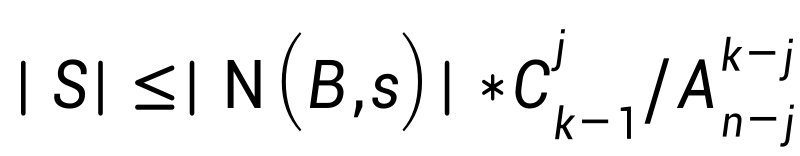
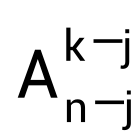
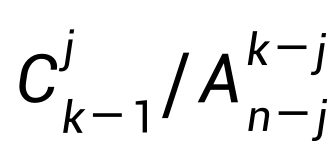
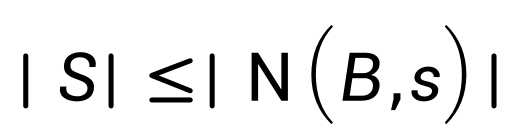
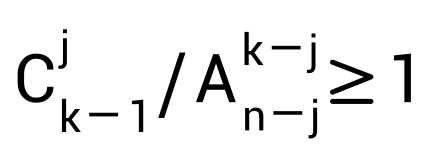
### 5.4.1 问题4的解

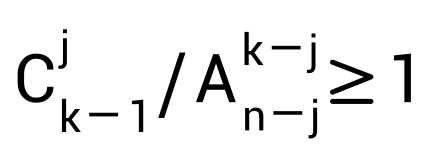
问题4中子网络的数量等于142,800。

由于满足精确覆盖问题，每个顶点对应了35个子网络，因此结果是142,800/35=4080个顶点，只需要破坏4080个顶点就可以破坏掉所有子网络。

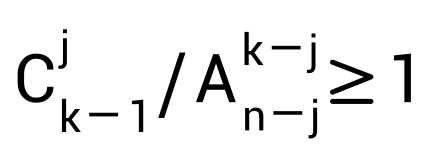
## 5.5 二分图上求解完备匹配问题

### 5.5.1 对(n,k)网络以及其对应的(n-j,k-j)子网络进行分类

对于5.3.2中的公式，希望通过的大小判断是否存在,如果满足那就可以说明问题可以达成精确匹配。

直观的看,k足够小一些的时候是直接成立的，否则那就是不成立。

### 5.5.2 二分图最大匹配求具体方案

假设成立，那么这个求破坏方案的问题就可以变成一个求二分图完备匹配的问题，在这个问题里面求二分图完备匹配可以直接使用二分图最大匹配的求法，可以保证求出的最大匹配是个完备匹配。

求二分图最大匹配常常选用最大流或匈牙利算法，前者效率更高，出于简单的目的，这里只简单介绍一下匈牙利算法，时间复杂度**O(|V³|)**。

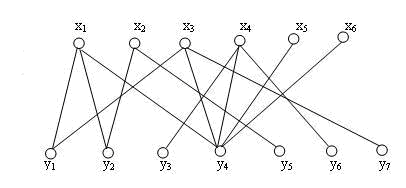
先介绍一下两个概念:

**交替路：**从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交替路。

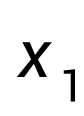
**增广路**：从一个未匹配点出发，走交替路，如果途径另一个未匹配点（出发的点不算），则这条交替路称为增广路（agumenting path）。

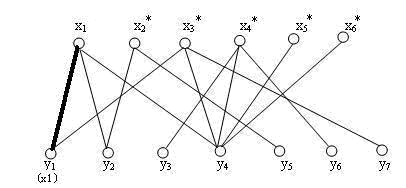
匈牙利算法与增广路的性质相关，增广路中的匹配边总是比未匹配边多一条，所以如果我们放弃一条增广路中的匹配边，选取未匹配边作为匹配边，则匹配的数量就会增加。匈牙利算法就是在不断寻找增广路，如果找不到增广路，就说明达到了最大匹配。

以下是一个匈牙利算法的示例:

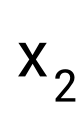


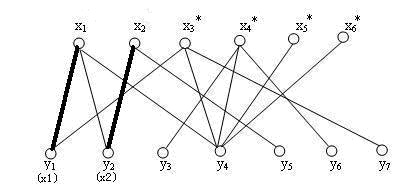
**图6 初始二分图**

选中第一个点找第一跟连线

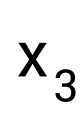
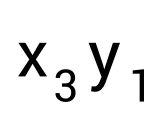
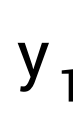
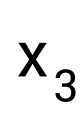
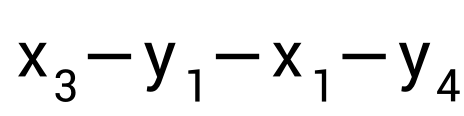
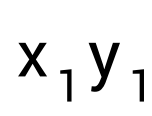
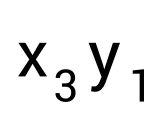
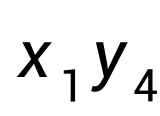


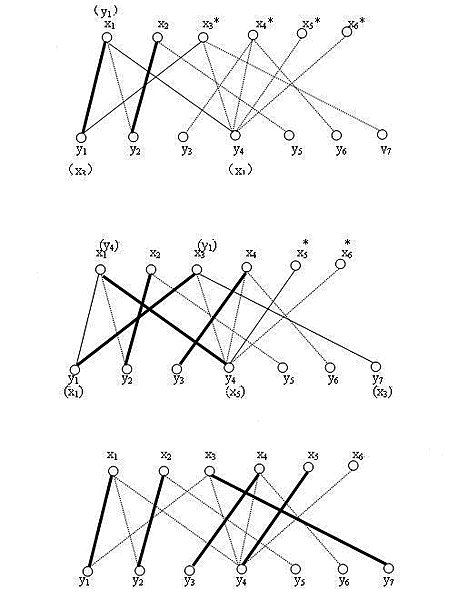
**图7 二分图第一次匹配**

选中第二个点找第二跟连线

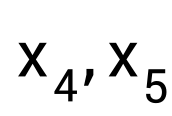


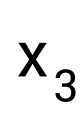
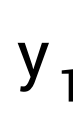
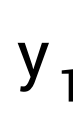
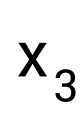
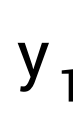
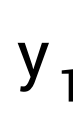
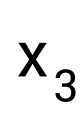
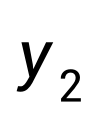
**图8 二分图第二次匹配**

发现的第一条边发现已经被人占了，找出出发的的交错路径，把交错路中已在匹配上的边从匹配中去掉，剩余的边 加到匹配中去

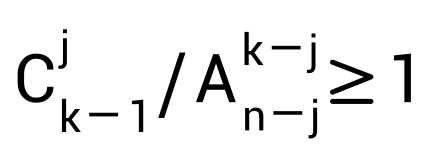


**图9 二分图增广路匹配**

 同理加入。

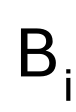
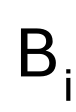
匈牙利算法可以深度有限或者广度优先，刚才的示例是深度优先，即找,已经有匹配，则找交错路。若是广度优先，应为：找,有匹配，找。

### 5.5.3 集合覆盖问题的求解具体方案

假设不成立，那这个问题只能建模成一个集合覆盖问题了，可以使用线性规划求解但是效率太慢资源消耗太大，使用贪心更合适，虽然不是最优解，但是也是较好的解了。

本题中对集合覆盖的贪心策略是这样的。

定义一个答案集合Answer。

遍历集合**B**中的顶点，找到覆盖了最多没有被破坏的子网络的顶点，就是选择被破坏的顶点添加到Answer中。

直到所有子网络都被选择破坏就获得了具体破坏顶点的集合Answer。

# 模型和算法的优缺点与改进

## 6.1模型算法的优点

故障检测模型:时间复杂度优秀，当故障点数量远小于结点数量时前提成立的条件下可以通过一次对矩阵的遍历大幅减少DFS的规模，可以确定精确解。

精确覆盖模型:把集合覆盖问题特殊化，在处理特殊规模问题的时候可以在与贪心算法资源消耗相当的情况下求出最优解。

## 6.2模型算法的缺点

1. 故障检测模型受故障点规模影响大，如果有较大的故障点规模，就需要对所有可能疑似故障点的联通性进行分析，对不满足前提条件的节点以及邻居节点进行排除才能对剩余节点继续使用准则进行节点的确定性分析，对前提条件的分析十分困难。
2. 对问题34，由于该问题规模特别，可以变成精确覆盖问题求解，但是若遇到一些规模之外的算法就只能使用贪心算法了，虽然效率高但是不能获得最优解。

## 6.3模型算法的改进

1. 在问题故障检测模型中，改进对每个节点是否满足前提条件的判断，使得该算法泛化性更强。
2. 对集合覆盖问题的算法的分类的结论进行改进，希望实现集合覆盖和精确覆盖的有机结合，提高对不能转化成集合覆盖问题尝试利用精确覆盖的求解思路进行加速。

# 参考文献

1. [] 杨玉星,邱亚娜.k 元 n 维冒泡排序网络的子网排除[J].计算机学报,2017,44(11):264- 267. [↑](#endnote-ref-0)
2. [] 李金强. 网络拓扑结构的可靠性分析[D].福建师范大学,2019. [↑](#endnote-ref-1)
3. []程钊.图论中若干重要定理的历史注记[J].数学的实践与认识,2013,43(01):261-268.

   **附录**

   1. import pandas as pd
   2. import numpy as np
   3. import matplotlib.pyplot as plt
   4. *#import itertools*
   5. *#读取*
   6. doc\_path=r'C:\Users\lenovo\Desktop\A题\附件3. S6,4报告.xlsx'
   7. doc\_path\_1=r'C:\Users\lenovo\Desktop\A题\附件1. S6,4邻接矩阵.xlsx'
   8. *#一部分初始化数据*
   9. the\_max\_bad\_num=5     #题目要求不超过
   10. the\_degree=5            #图的度数
   11. #doc\_path=r'C:\Users\lenovo\Desktop\A题\附件4. s7.4报告.xlsx'
   12. *#doc\_path\_1=r'C:\Users\lenovo\Desktop\A题\附件2. S7,4邻接矩阵.xlsx'*
   13. df = pd.read\_excel(doc\_path)
   14. *#获得映射和逆映射*
   15. dic=list(df.columns)
   16. dic.pop(0)
   17. dic\_inverse={value: index for index, value in enumerate(dic)}
   18. row\_num=len(dic)    #行数量
   19. #存成故障邻接矩阵
   20. data\_range = df.loc[:row\_num,dic[0]:dic[-1]]
   21. np\_array = data\_range.to\_numpy()
   22. #获取邻居邻接表
   23. df\_2= pd.read\_excel(doc\_path\_1)
   24. nodes = df\_2.columns[1:].values
   25. adjacency\_data = df\_2.iloc[0:, 1:].values
   26. adjacency\_list = {node: [] for node in nodes}
   27. for i, node\_from in enumerate(nodes):
   28. for j, node\_to in enumerate(nodes):
   29. if adjacency\_data[i, j] > 0:
   30. adjacency\_list[node\_from].append(node\_to)
   31. #统计发出-1的点以及发出的个数
   32. test = [0]\*row\_num
   33. out = [[] for \_ in range(row\_num)]
   34. graph={}
   35. for i in range(len(np\_array)):#行向量
   36. count = 0
   37. for j in range(len(np\_array)):#列向量
   38. if np\_array[i][j] == -1:
   39. count += 1
   40. *#test[i] = count*
   41. out[i].append(dic[j])
   42. test[i]=count
   43. for i in range(row\_num):
   44. if test[i]!=0:
   45. sub\_graph={dic[i]:out[i]}
   46. graph.update(sub\_graph)
   47. *#print(graph)*
   48. #统计接收-1的点以及发出的个数
   49. test\_1 = [0]\*row\_num
   50. receive = [[] for \_ in range(row\_num)]
   51. graph\_1={}
   52. for i in range(len(np\_array)):#列向量
   53. count = 0
   54. for j in range(len(np\_array)):#行向量
   55. if np\_array[j][i] == -1:        #!
   56. count += 1
   57. receive[i].append(dic[j])
   58. test\_1[i]=count
   59. for i in range(row\_num):
   60. if test\_1[i]!=0:
   61. sub\_graph={dic[i]:receive[i]}
   62. graph\_1.update(sub\_graph)
   63. *#print(graph\_1)*
   64. #初始化状态字典
   65. status\_dict = {}
   66. for i in range(row\_num):
   67. status\_dict.update({dic[i]:0})
   68. *#获得发出-1和接收-1的点的列表*
   69. key\_list\_out = []
   70. for key in graph:
   71. key\_list\_out.append(key)
   72. key\_list\_receive = []
   73. for key in graph\_1:
   74. key\_list\_receive.append(key)
   75. *#四条准则*
   76. for i in range(row\_num):
   77. if (dic[i] in key\_list\_out) and (dic[i] not in key\_list\_receive):
   78. status\_dict[dic[i]]=1
   79. if (dic[i] not in key\_list\_out) and (dic[i] in key\_list\_receive):
   80. status\_dict[dic[i]]=-1
   81. if (dic[i] not in key\_list\_out) and (dic[i] not in key\_list\_receive):
   82. status\_dict[dic[i]]=1
   83. neibor\_list=adjacency\_list[dic[i]]
   84. for neibor in neibor\_list:
   85. status\_dict[neibor]=1
   86. #对0号元素根据邻居的状态更新自身,加一个判断联通块的预处理
   87. vis=[0]\*row\_num
   88. def DFS(inde):  #键值,极端情况下自身是1还是0
   89. if(vis[dic\_inverse[inde]]):
   90. return
   91. vis[dic\_inverse[inde]]=1
   92. minus\_1\_number=0        #统计负一的个数
   93. for neibor in adjacency\_list[inde]:
   94. if(status\_dict[neibor]==0):
   95. DFS(neibor)
   96. *#不能加elif，必须是if*
   97. if(status\_dict[neibor]==1):
   98. if(np\_array[dic\_inverse[neibor]][dic\_inverse[inde]]==-1):
   99. status\_dict[inde]=-1
   100. return
   101. elif(np\_array[dic\_inverse[neibor]][dic\_inverse[inde]]==1):
   102. status\_dict[inde]=1
   103. return
   104. if(np\_array[dic\_inverse[inde]][dic\_inverse[neibor]]==-1):
   105. minus\_1\_number+=1
   106. *#走到这里邻居一定是全为-1*
   107. if(minus\_1\_number==the\_degree):     #当前点认为自己周围全是坏的点
   108. status\_dict[inde]=1
   109. else:
   110. status\_dict[inde]=-1
   111. return
   112. for i in status\_dict:   #键
   113. if status\_dict[i]==0:
   114. DFS(i)
   115. vis=[0]\*row\_num
   116. #结果输出的位置
   117. answer\_num=0
   118. for i in status\_dict:   #键
   119. if status\_dict[i]<=0:
   120. print(f'{i}  {status\_dict[i]}')
   121. answer\_num+=1
   122. if(answer\_num<=the\_max\_bad\_num):
   123. print("legal")

   [↑](#endnote-ref-2)