

**软件可靠性课程实验报告**

**题目:**基于Petri网的客机维修排故专家系统

**院系:**计算机科学与技术学院/软件学院

**专业:**软件工程

**姓名:**陈梓鹏

**学号:**162230217

**2024年12月7日**

目录

[1.引言 3](#_Toc185770456)

[1.1编写目的 3](#_Toc185770457)

[2.基于体系结构的Petri网模型 3](#_Toc185770458)

[2.1 模型背景 3](#_Toc185770459)

[2.2 算法原理 3](#_Toc185770460)

[2.3 算法步骤 5](#_Toc185770461)

[2.4 算法流程 5](#_Toc185770462)

[2.5 算法实例分析 6](#_Toc185770463)

[2.6 算法实现 8](#_Toc185770464)

[2.7 算法结果分析 17](#_Toc185770465)

[参 考 文 献 17](#_Toc185770466)

# 1.引言

## 1.1编写目的

随着航空工业的迅速发展，客机的数量和复杂性不断增加，客机的维修和故障排除工作变得更加繁重且复杂。传统的维修排故方法通常依赖人工经验和手动诊断，既效率低下，又容易受到人为错误的影响。因此，开发一种高效、准确的维修排故专家系统成为了航空公司和维修机构的迫切需求。

本系统的编写目的在于利用Petri网模型的优势，通过对客机维修流程和故障诊断过程进行建模，开发一个智能化、自动化的客机维修排故专家系统，以提高故障排查效率、减少人为失误，并提升整体维修质量。

# 2.基于体系结构的Petri网模型

## 2.1 模型背景

Petri网（Petri Net）是一种图形化的数学建模语言，能够描述系统的并发、同步、资源共享等行为。它特别适用于复杂系统的建模和分析，广泛应用于生产过程、通信系统以及计算机网络等领域。在软件可靠性分析中，Petri网能够有效地捕捉系统组件之间的相互作用和故障传播机制，帮助分析系统的故障状态和可靠性表现。

基于体系结构的Petri网模型（Architecture-based Petri Net Model）结合了软件体系结构的分层性和Petri网的状态转换机制。通过对软件系统的体系结构进行建模，可以从宏观上反映出系统的结构特性，如模块之间的依赖关系、各个组件的交互方式以及数据流动等。Petri网模型则进一步提供了系统中各个组件在不同状态下的行为表现，能够精确描述各个组件发生故障时如何影响整个系统的稳定性。

## 2.2 算法原理

Petri网是一种网状信息流模型，其中包括库所（Place）和变迁（Transition）两种节点，并且用有向弧连接起来，在以库所和变迁为节点的有向二分图的基础上加上表示状态信息的令牌（token），并按一定的引发规则使得事件驱动状态演变，从而反映系统的动态运行过程。

Petri 网不仅仅是一种可以用网状图形表示的数学对象，它的本质是尊重自然规律的现实中存在的物理模型。通过Petri 网描述的异步并发现象反映了事物间的依赖关系。下面是有关Petri网的一些基本概念。

定义：（Petri网结构）给定三元组，该三元组是Petri网结构当且仅当：

其中*P*和*T*分别称为*N*的库所集合和变迁集合；*F*为有向边集合，它是由元素*P*和*T*组成的有序偶集合，表示网的流（*Flow*）关系；分别为F的定义域和值域，集合是Petri网元素的集合。

Petri网结构中，库所表示条件、资源、等待队列和信道等，一般用圆圈（⭕）表示；变迁表示事件、动作、语句执行和消息发送/接受等，一般采用矩形表示。通常在Petri网的图形表示中，用带黑点的圆圈（）表示标记。一个变迁（事件）有一定数量的输入和输出库所，分别代表事件的前置条件和后置条件，库所中的令牌代表可以使用的资源数量或数据。

一个典型的Petri网结构如图1所示。

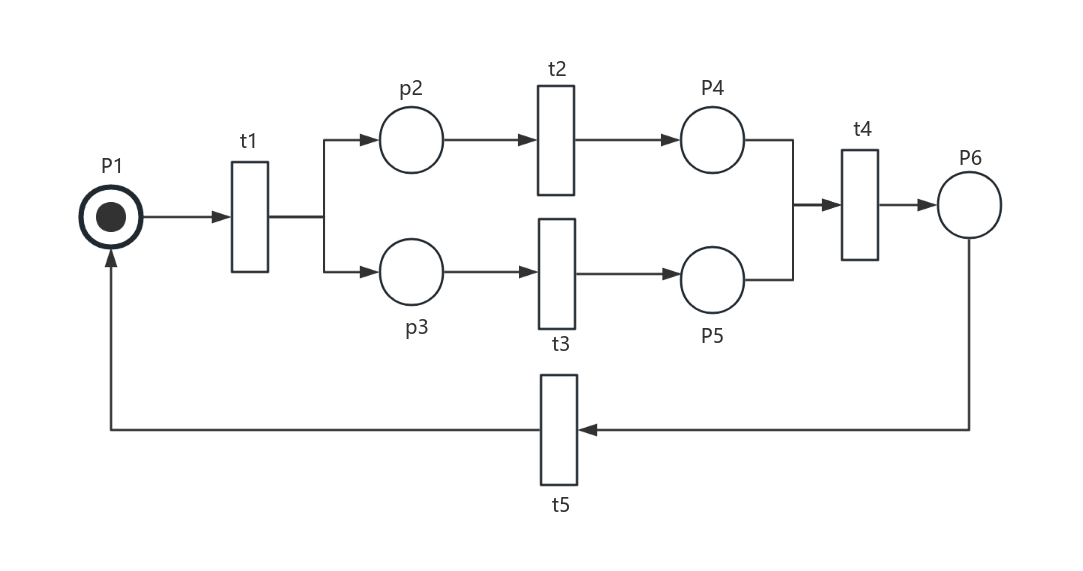


图1.典型Petri网结构图

定义：（加权SPAN）加权SPAN是一个八元序偶*（）*，其中*P*为库所集合，*T*为变迁集合，全函数*H*：为起始库所集合，终止库所集合，为变迁概率集合，为组件和连接件的可靠性度量域，，，为迁移过程的可靠性量域，通常集合，，

在加权*SAPN*=上，若路径*PW=...*满足*：*

则称该路径为加权*SAPN*的一个运行。

在基于*SA*的可靠性建模中，一般把组件和连接件作为黑盒看待，因此必须有一个评估软件组件和连接件可靠性的方法，下面定义可靠性函数。

定义：（可靠度函数）设*Interface（C）*表示*SA*中组件*C*的接口数，设*Interface（L）*表示*SA*中连接件*L*的接口数，则*SA*中组件和连接件的可靠度分别为：

其中,可以通过测试多个有不同接口的不同组件和连接件得出相应的系数，迁移过程的可靠性由组件与连接件之间的耦合程度决定，耦合度越弱，则迁移过程的可靠性越高，反之，则迁移过程的可靠度越低。

## 2.3 算法步骤

根据上述的定义和可靠性映射，假设起始组件为附加组件，它不做任何动作，可靠性为100%，则评估*SAPN*的可靠性步骤如下：

步骤1：根据*SA*模型建立加权*SAPN*模型。

步骤2：根据加权*SAPN*的迁移概率生成测试路径*PW*，若在加权*SAPN*中存在循坏测试路径，则该路径不重复计算，并求出测试路径*PW*的迁移概率。在加权*SAPN*中，测试路径就是运行，可以通过广度优先搜索（*BFS*）算法求出从初始点终止点的测试路径，测试路径*PW*的迁移概率可以计算如下：

步骤3：计算测试路径*PW*的可靠性。

假设测试路径*PW*为*...，*那么该测试路径的可靠性可以计算如下：

其中是*SA*中组件的可靠度，是*SA*中组件的可靠度，为迁移过程的可靠度。

步骤4：*SA*的可靠性计算如下：

上述中的,表示沿路径的迁移概率，由于在测试路径的可靠性估计上加了迁移过程的可靠性，所以通过上述计算方法得出的*SA*可靠性是一个系统动态运行时的可靠性，能够很好地反应系统运行状态下软件体系结构的可靠性。

## 2.4 算法流程

算法流程步骤如图2所示。

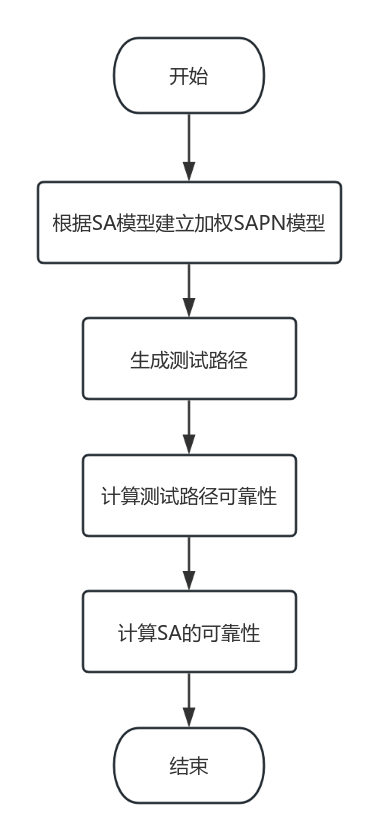


图2.基于SAPN的可靠性建模步骤

## 2.5 算法实例分析

下面以客机维修排故专家系统为例，基于Petri网对其体系结构的可靠性进行分析评估。客机维修排故专家系统的软件体系结构如图3所示。

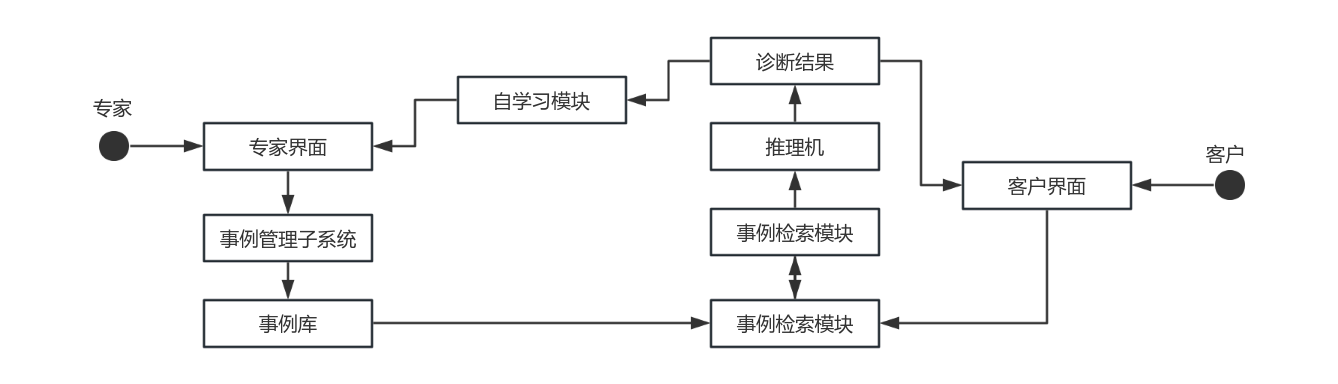


图3.维修排故专家系统体系结构

在该系统的*SA*模型中，系统总功能实现首先由专家点开始，人机界面与民航客机维修排故专家系统进行交互，通过一系列功能模块的共同协作完成对飞机故障的分析，最后将处理结果传输给客户。

利用SA模型对该系统进行抽象描述，假设专家系统功能的起始点为S，客户为系统功能的终止点记为EN，专家界面、事例库管理子系统、事例库、事例检索模块、动态数据库、推理机、诊断结果、自学习模块、客户界面等组件分别记作：按此顺序它们之间的连接件分别记作：，诊断结果、事例检索模块与客户界面之间的连接件分别基座。

假设系统的组件、连接件和迁移过程的可靠度已知，迁移概率为已知。那么该*SA*模型的可靠性可以通过对*SA*加权*SAPN*模型进行建模，利用*BFS*方法寻找运行路径，计算各个运行路径的可靠性，最后求出整个*SA*的可靠性，其具体分析评估可以按以下步骤进行：

步骤1：对图7.5的*SA*模型用加权SAPN模型进行建模，其加权SAPN模型如7.6所示。

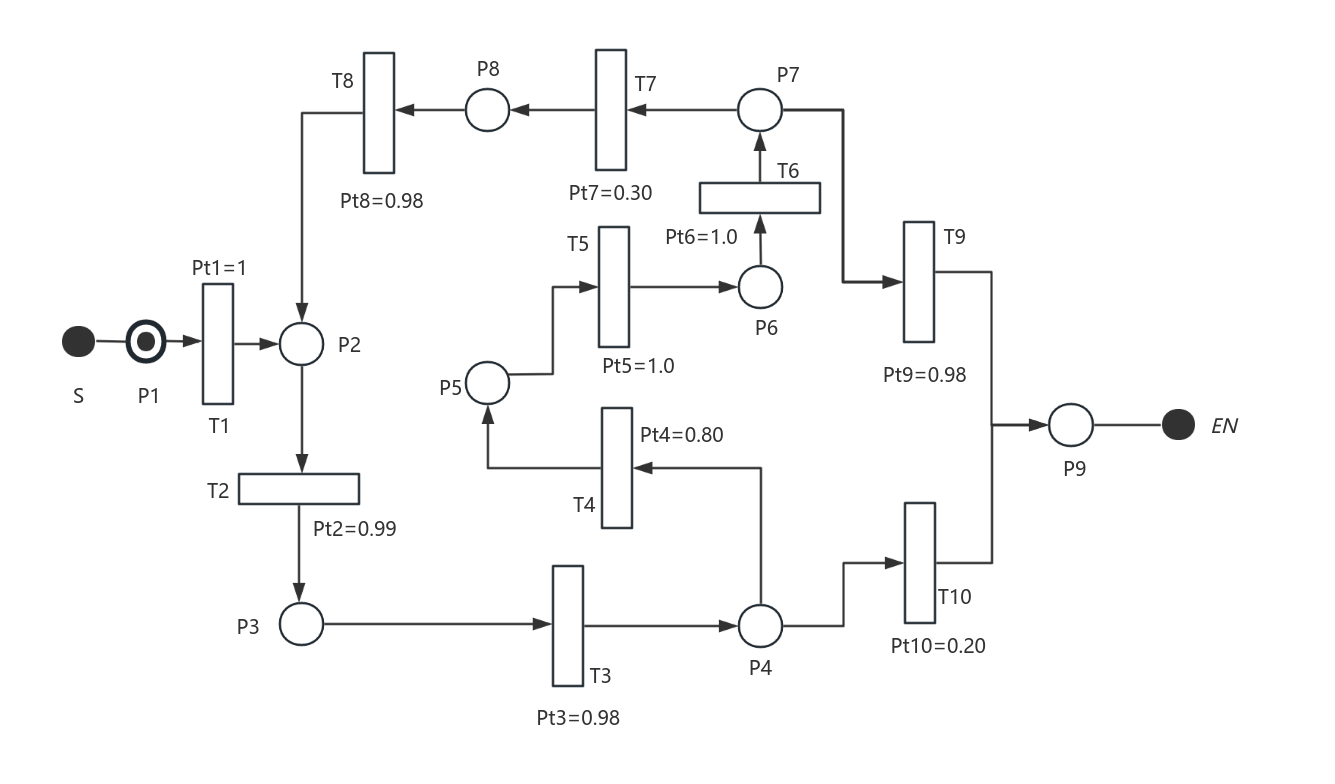


图4.维修排故专家系统加权SAPN模型

步骤2：根据加权SAPN的迁移概率生成测试路径。利用BFS算法可计算出从S到EN的路径有：

1. S->C1->T1->C2->T2->C3->T3->C4->T10->C9->EN
2. S->C1->T1->C2->T2->C3->T3->C4->T4->C5->T5->C6->T6->C7->T9->C9->EN
3. S->C1->T1->C2->T2->C3->T3->C4->T4->C5->T5->C6->T6->C7->T7->C8->T8->C2->T2->C3->T3->C4->T10->C9->EN
4. S->C1->T1->C2->T2->C3->T3->C4->T4->C5->T5->C6->T6->C7->T7->C8->T8->C2->T2->C3->T3->C4->T4->C5->T5->C6->T6->C7->T9->C9->EN

总共有4条测试路径。其迁移概率分别为：

1. =0.194;
2. =0.760;
3. =0.044;
4. =0.174.

步骤3：根据BFS算法求出的测试路径，计算测试路径的可靠性。假设从组件到的可靠度依次为

连接件到的可靠度依次为

迁移过程到的可靠度依次为

则各测试路径的可靠度分别为：

.

.

.

.

步骤4：利用上述计算结果带入公式，计算该系统*SA*的可靠度为：

通过上述计算可知，整个系统*SA*的可靠性不小于可靠性最差的运行路径，不大于可靠性最大的运行路径。这说明系统的*SA*的可靠性不仅与*SA*的各个组件和连接件的可靠性有关，还与系统运行时的测试路径、路径的迁移概率和迁移过程的可靠性相关，即系统的可靠性与系统SA架构有密切的关系，好的*SA*架构能够大幅度提高软件系统的可靠性。

## 2.6 算法实现

数据结构定义：

*// Petri网库所和变迁的表示*

struct Node

{

    std::string name;

    double reliability1; *// 节点可靠度*

    double reliability2; *// 连接件可靠度*

    double probability; *// 转移概率*

};

*// 定义边*

struct Edge

{

    std::string from, to;

};

std::map<std::string, Node> dic; *// 名称与点的映射*

*// Petri网类*

class PetriNet

{

public:

    std::vector<Node> nodes; *// 点集*

    std::vector<Edge> edges; *// 边集*

std::map<std::string, std::vector<std::string>> adjacencyList;

*// 邻接矩阵*

    std::map<std::string, int> vis;*// BFS访问记录*

    std::vector<std::vector<Node>> paths; *// 路径*

    std::vector<double> proofpath;*// 路径转移概率*

    std::vector<double> reofpath;*// 路径可靠度*

}

添加顶点：

根据SA模型建立加权SAPN模型：

    void addNode(std::string *name*, double *reliability1*, double *reliability2*, double *probability*)

    {

*// name代表node名称 reliability1表示库所或变迁可靠性 reliability1连接件可靠性 probability表示转移概率*

        if (*name*[0] == 'C')

        {

            nodes.push\_back({*name*, *reliability1*, 1, 1});

            dic[*name*] = nodes.back();

        }

        else

        {

            nodes.push\_back({*name*, *reliability1*, *reliability2*, *probability*});

            dic[*name*] = nodes.back();

        }

    }

添加边：

    void addEdge(std::string *from*, std::string *to*)

    {

        adjacencyList[*from*].push\_back(*to*);

    }

根据SA模型建立加权SAPN模型

    PetriNet Net;

    Net.addNode("C1", 1, 1, 1);

    Net.addNode("C2", 0.99, 1, 1);

    Net.addNode("C3", 0.98, 1, 1);

    Net.addNode("C4", 1, 1, 1);

    Net.addNode("C5", 0.99, 1, 1);

    Net.addNode("C6", 0.99, 1, 1);

    Net.addNode("C7", 1, 1, 1);

    Net.addNode("C8", 0.98, 1, 1);

    Net.addNode("C9", 1, 1, 1);

    Net.addNode("T1", 1, 0.99, 1);

    Net.addNode("T2", 0.99, 1, 0.99);

    Net.addNode("T3", 1, 1, 0.98);

    Net.addNode("T4", 0.98, 0.98, 0.80);

    Net.addNode("T5", 0.99, 1, 1.0);

    Net.addNode("T6", 1, 0.99, 1.0);

    Net.addNode("T7", 0.98, 0.99, 0.30);

    Net.addNode("T8", 0.98, 1, 0.98);

    Net.addNode("T9", 0.99, 0.98, 0.98);

    Net.addNode("T10", 1, 1, 0.20);

    Net.addEdge("C1", "T1");

    Net.addEdge("T1", "C2");

    Net.addEdge("C2", "T2");

    Net.addEdge("T2", "C3");

    Net.addEdge("C3", "T3");

    Net.addEdge("T3", "C4");

    Net.addEdge("C4", "T4");

    Net.addEdge("C4", "T10");

    Net.addEdge("T10", "C9");

    Net.addEdge("T4", "C5");

    Net.addEdge("C5", "T5");

    Net.addEdge("T5", "C6");

    Net.addEdge("C6", "T6");

    Net.addEdge("T6", "C7");

    Net.addEdge("C7", "T9");

    Net.addEdge("T9", "C9");

    Net.addEdge("C7", "T7");

    Net.addEdge("T7", "C8");

    Net.addEdge("C8", "T8");

    Net.addEdge("T8", "C2");

根据加权SAPN的迁移概率生成测试路径。利用BFS算法

*// BFS算法生成所有路径*

    void generatePaths(Node *start*, Node *end*)

    {

        std::vector<Node> s = {*start*};

        std::queue<std::vector<Node>> q;

        q.push(s);

        while (q.size())

        {

            auto t = q.front();

            q.pop();

            auto node = t.back();

            if (node.name == *end*.name)

            {

                paths.push\_back(t);

                continue;

            }

            for (auto &x : adjacencyList[node.name])

            {

                if (x != *end*.name && vis[x] >= 2)

                    continue;

                vis[x]++;

                std::vector<Node> nn = t;

                nn.push\_back(dic[x]);

                q.emplace(nn);

            }

        }

    }

计算路径迁移概率：

*// 计算路径概率*

    void CalculatePathProbability()

    {

        std::cout << "总共有" << paths.size() << "条测试路径,其迁移概率分别为:" << std::endl;

        int cnt = 0;

        for (auto &path : paths)

        {

            double pro = 1.0;

            for (auto &x : path)

                pro \*= x.probability;

            std::cout << "路径" << cnt++ << "的迁移概率为" << pro << std::endl;

            proofpath.push\_back(pro);

        }

        std::cout << std::endl;

    }

计算测试路径的可靠性：

*// 计算路径可靠度*

    void CalculatePathReliability()

    {

        std::cout << "总共有" << paths.size() << "条测试路径,其可靠度分别为:" << std::endl;

        int cnt = 0;

        for (auto &path : paths)

        {

            double pro = 1.0;

            for (auto &x : path)

                pro \*= x.reliability1 \* x.reliability2;

            std::cout << "路径" << cnt++ << "的可靠度为" << pro << std::endl;

            reofpath.push\_back(pro);

        }

        std::cout << std::endl;

    }

利用上述计算结果，计算该系统SA的可靠度

    void CalculateSystemReliability()

    {

        double sum\_Pro = 0;

        double sum\_weighted\_R = 0;

        for (int i = 0; i < paths.size(); ++i)

        {

            sum\_Pro += proofpath[i];

            sum\_weighted\_R += proofpath[i] \* reofpath[i];

        }

        std::cout << "计算该系统SA的可靠度为:";

        std::cout << sum\_weighted\_R / sum\_Pro << std::endl;

    }

完整代码如下：

#include <bits/stdc++.h>

*// Petri网库所和变迁的表示*

struct Node

{

    std::string name;

    double reliability1; *// 节点可靠度*

    double reliability2; *// 连接件可靠度*

    double probability; *// 转移概率*

};

*// 定义边*

struct Edge

{

    std::string from, to;

};

std::map<std::string, Node> dic; *// 名称与点的映射*

*// Petri网类*

class PetriNet

{

public:

    std::vector<Node> nodes; *// 点集*

    std::vector<Edge> edges; *// 边集*

    std::map<std::string, std::vector<std::string>> adjacencyList; *// 邻接矩阵*

    std::map<std::string, int> vis; *// BFS访问记录*

    std::vector<std::vector<Node>> paths; *// 路径*

    std::vector<double> proofpath; *// 路径转移概率*

    std::vector<double> reofpath; *// 路径可靠度*

    void addNode(std::string *name*, double *reliability1*, double *reliability2*, double *probability*)

    {

*// name代表node名称 reliability1表示库所或变迁可靠性 reliability1连接件可靠性 probability表示转移概率*

        if (*name*[0] == 'C')

        {

            nodes.push\_back({*name*, *reliability1*, 1, 1});

            dic[*name*] = nodes.back();

        }

        else

        {

            nodes.push\_back({*name*, *reliability1*, *reliability2*, *probability*});

            dic[*name*] = nodes.back();

        }

    }

    void addEdge(std::string *from*, std::string *to*)

    {

        adjacencyList[*from*].push\_back(*to*);

    }

*// BFS算法生成所有路径*

    void generatePaths(Node *start*, Node *end*)

    {

        std::vector<Node> s = {*start*};

        std::queue<std::vector<Node>> q;

        q.push(s);

        while (q.size())

        {

            auto t = q.front();

            q.pop();

            auto node = t.back();

            if (node.name == *end*.name)

            {

                paths.push\_back(t);

                continue;

            }

            for (auto &x : adjacencyList[node.name])

            {

                if (x != *end*.name && vis[x] >= 2)

                    continue;

                vis[x]++;

                std::vector<Node> nn = t;

                nn.push\_back(dic[x]);

                q.emplace(nn);

            }

        }

    }

    void showPaths()

    {

        std::cout << "利用BFS算法可计算出从S到EN的路径有:" << std::endl;

        for (int i = 0; i < paths.size(); ++i)

        {

            std::cout << "S->";

            for (int j = 0; j < paths[i].size(); ++j)

            {

                std::cout << paths[i][j].name << "->";

            }

            std::cout << "EN" << std::endl;

        }

        std::cout << std::endl;

    }

*// 计算路径概率*

    void CalculatePathProbability()

    {

        std::cout << "总共有" << paths.size() << "条测试路径,其迁移概率分别为:" << std::endl;

        int cnt = 0;

        for (auto &path : paths)

        {

            double pro = 1.0;

            for (auto &x : path)

                pro \*= x.probability;

            std::cout << "路径" << cnt++ << "的迁移概率为" << pro << std::endl;

            proofpath.push\_back(pro);

        }

        std::cout << std::endl;

    }

*// 计算路径可靠度*

    void CalculatePathReliability()

    {

        std::cout << "总共有" << paths.size() << "条测试路径,其可靠度分别为:" << std::endl;

        int cnt = 0;

        for (auto &path : paths)

        {

            double pro = 1.0;

            for (auto &x : path)

                pro \*= x.reliability1 \* x.reliability2;

            std::cout << "路径" << cnt++ << "的可靠度为" << pro << std::endl;

            reofpath.push\_back(pro);

        }

        std::cout << std::endl;

    }

    void CalculateSystemReliability()

    {

        double sum\_Pro = 0;

        double sum\_weighted\_R = 0;

        for (int i = 0; i < paths.size(); ++i)

        {

            sum\_Pro += proofpath[i];

            sum\_weighted\_R += proofpath[i] \* reofpath[i];

        }

        std::cout << "计算该系统SA的可靠度为:";

        std::cout << sum\_weighted\_R / sum\_Pro << std::endl;

    }

};

int main(void)

{

    PetriNet Net;

    Net.addNode("C1", 1, 1, 1);

    Net.addNode("C2", 0.99, 1, 1);

    Net.addNode("C3", 0.98, 1, 1);

    Net.addNode("C4", 1, 1, 1);

    Net.addNode("C5", 0.99, 1, 1);

    Net.addNode("C6", 0.99, 1, 1);

    Net.addNode("C7", 1, 1, 1);

    Net.addNode("C8", 0.98, 1, 1);

    Net.addNode("C9", 1, 1, 1);

    Net.addNode("T1", 1, 0.99, 1);

    Net.addNode("T2", 0.99, 1, 0.99);

    Net.addNode("T3", 1, 1, 0.98);

    Net.addNode("T4", 0.98, 0.98, 0.80);

    Net.addNode("T5", 0.99, 1, 1.0);

    Net.addNode("T6", 1, 0.99, 1.0);

    Net.addNode("T7", 0.98, 0.99, 0.30);

    Net.addNode("T8", 0.98, 1, 0.98);

    Net.addNode("T9", 0.99, 0.98, 0.98);

    Net.addNode("T10", 1, 1, 0.20);

    Net.addEdge("C1", "T1");

    Net.addEdge("T1", "C2");

    Net.addEdge("C2", "T2");

    Net.addEdge("T2", "C3");

    Net.addEdge("C3", "T3");

    Net.addEdge("T3", "C4");

    Net.addEdge("C4", "T4");

    Net.addEdge("C4", "T10");

    Net.addEdge("T10", "C9");

    Net.addEdge("T4", "C5");

    Net.addEdge("C5", "T5");

    Net.addEdge("T5", "C6");

    Net.addEdge("C6", "T6");

    Net.addEdge("T6", "C7");

    Net.addEdge("C7", "T9");

    Net.addEdge("T9", "C9");

    Net.addEdge("C7", "T7");

    Net.addEdge("T7", "C8");

    Net.addEdge("C8", "T8");

    Net.addEdge("T8", "C2");

    Net.generatePaths(dic["C1"], dic["C9"]);

    Net.showPaths();

    Net.CalculatePathProbability();

    Net.CalculatePathReliability();

    Net.CalculateSystemReliability();

    system("pause");

    return 0;

}

## 2.7 算法结果分析

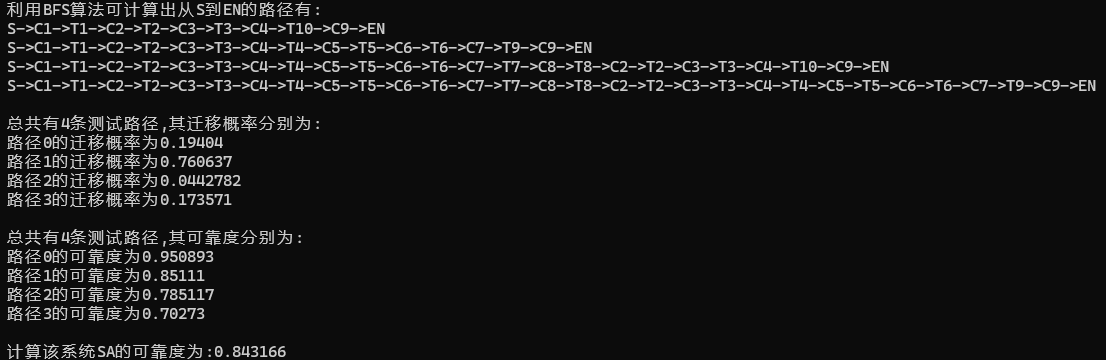


图5.算法运行结果

从输出结果可以看出，BFS算法搜索路径、路径迁移概率、路径可靠度、系统SA的可靠度运算均正确，验证了算法的正确性。

参 考 文 献

[1] (中)张德平编著;软件系统可靠性分析基础与实践（Fundamentals and Practice of Software System Reliablity Analysis）清华大学出版社

[2] 袁崇义.Petri网原理与应用[M].电子工业出版社,2005.