View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Creative Repository of Electro-Communications

Department of Communication Engineering and Informatics

Graduate School of Informatics and Engineering

The University of Electro-Communications, Tokyo, Japan

Preventive Start-time Routing considering Failure Characteristics

Student number: 1231028

Name: **KAPTCHOUANG** Stephane

Supervisor 1: **OKI** Eiji, Professor

Supervisor 2: **KISHI** Naoto, Professor

Submission date: 27 January 2014

専攻主任印	主指導教員印	指導教員印

Department of Communication Engineering and Informatics

Graduate School of Informatics and Engineering

The University of Electro-Communications, Tokyo, Japan

Preventive Start-time Routing considering Failure Characteristics

Student number: 1231028

Name: **KAPTCHOUANG** Stephane

Supervisor 1: **OKI** Eiji, Professor

Supervisor 2: **KISHI** Naoto, Professor

Submission date: 27 January 2014

Contents

Abstract

We propose a Preventive Start-time Routing that considers failure characteristics. The Preventive Start-time Routing (Optimization) scheme determines a suitable set of OSPF link weights at the start time that can handle any link failure scenario preventively. Previously a PSO (Preventive Start-time Optimization) scheme was designed to minimize the worst case congestion ratio in case of failure. That scheme considers any failure pattern to determine a link weight set that counters worst case failure. Unfortunately, under no failure, that link weight set leads to a high congestion ratio. Under no failure, a high congestion ratio would be a penalty that will be carried on and thus become a burden especially in networks with few failures.

In the first part of this work, we present a Preventive Start-time Optimization scheme that suppresses that penalty while reducing the worst congestion ratio by considering both failure and non failure scenarios. we call this scheme PSO-NP(Preventive Start-time Optimization with no Penalty). PSO-NP is simple and effective in finding a link weight set thats considers both failure and non failure scenarios. We expand PSO-NP into a General Preventive Start-time Optimization (GPSO) to find a link weight set that balances both the penalty under no failure and the congestion ratio under worst case failure. Simulation results show that PSO-NP achieves substantial congestion reduction for any failure case while suppressing the penalty in case of no failure in the network. In addition, GPSO as framework is effective in determining a suitable link weight set that considers the trade off between the penalty under non failure and the worst case congestion ratio reduction.

In the second part of our work, we propose a Preventive Start-time Optimization that considers link Failure Probability (PSO-FP). While PSO, PSO-NP and GPSO examine every failure pattern, more than 50% of link failures may sometimes be concentrated on only 3% of total links in the network. This factor should be taken into consideration when dealing with failure. To include this factor we propose a

Preventive Start-time Optimization that considers link Failure Probability. Simulation results show that PSO-FP is effective in finding a weight set that reduces the congestion ratio expectation value.

Acknowledgments

We would like to thank Eiji Oki, Naoto Kishi, Ihsen Aziz Ouédraogo, and Ravindra Sandaruwan Ranaweera for their contribution to our work.

CHAPTER 1

Introduction

Implementing the most appropriate set of routes ameliorates the network resource utilization rate and thus increase network throughput of Internet Protocol (IP) networks. Since it optimises resource assignment, additional traffic can be supported. It also reduces network congestion and increases robustness against network failure. One useful approach to enhancing routing performance is to minimise the maximum link utilization rate, also called the network congestion ratio, of all network links. minimising the worst-case network congestion ratio can increase the admissible traffic.

Open Shortest Path First (OSPF) [**2**] is a widely used routing protocol. OSPF is a link-state-based Interior Gateway Protocol (IGP) for IP networks. This means that it gathers all the topology characteristics to build a routing scheme. In OSPF, all packets are transmitted over shortest paths. These paths calculation is based on Dijkstra's algorithm. Dijkstra's algorithm uses link weight or link cost (metric) to determine the path with the lowest total cost between a source and a destination pair. Therefore based on link weights in the network, OSPF computes the shortest path from each originating node to all nodes in the same network, and shares it among nodes in the network. In other words, determining the optimal path based on shortest-path routing means determining the optimal link weights.

Several algorithms that compute a set of optimal link weights in OSPF-based networks were addressed in [**3, 4, 5, 6, 7**] under the condition that the network topology and traffic matrix are given. Fortz et al. presented a heuristic algorithm based on tabu search [**3, 4**]. Buriol et al. presented a genetic algorithm with a local improvement procedure [**6**]. A fast heuristic algorithm was also developed by Reichert

and Magedanz [**5**]. These Optimisation algorithms yield nearly optimal sets of link weights in a practical manner.

Under the condition that the network topology and traffic matrix are given, Start-time Optimisation (SO) determines the optimal set of link weights once at the beginning of network operation. This set minimises the congestion ratio under the given traffic matrix so as to maximise additional traffic. Unfortunately, SO is weak against network failures. For example, a link failure will trigger the rerouting of active paths, causing a surge of congestion in the network. Meaning that under a failure scenario SO-generated weight set is no longer optimal. This makes SO unsuitable for failure prone networks because, it does not take any link failure into consideration during the Optimisation process.

The weakness of SO can be overcome by computing a new optimal set of link weights whenever the topology is changed. This approach, called Run-time Optimisation (RO), provides the best routing performance after each link failure, but it makes the network unstable. When link weights are changed, the updated information is broadcast across the network. As routers learn the updated link weights, they recompute their shortest paths to update their routing tables. This leads to higher resource utilization while creating confusion [**8, 9**]. Meanwhile, IP packets may arrive out of order and the performances of Transport Control Protocol (TCP) connections are degraded [**3, 4**]. The more often are link weights changed, the more the network becomes unstable. This is because packets are sent back and forth between routers to achieve the divergent processes of updating routing tables and calculating the shortest paths based on the updated link weight set.

It seems reasonable to find a scheme that can determine, at start time, a set of link weights that avoids both unexpected network congestion and network instability, regardless of which link failure occurs. Moreover, 70% of link failures affect a single link at a time [**10, 11**]. It makes sense to focus on single failures that occur in the network. A scheme called Preventive Start-time Optimisation (PSO) was presented [**12**].

PSO determines, at the start time, a suitable set of link weights that can handle any single link failure scenario preventively. PSO considers any failure scenario and minimises the worst possible congestion ratio. In PSO scheme, the link that creates the worst case congestion has to be avoided in other to relax the network. For that reason, A higher link weight will be assigned to that link (critical link) so that in case of failure the portion of traffic sent to those link is reduced.

There are several related works on PSO [**13, 14, 15, 16**]. To determine a suitable set of link weights based on the PSO policy, there are two PSO-based algorithms, PSO-LC (limited rage of candidates) [**12**] and PSO-WC (wide range of candidates) [**13, 14**]. Although PSO-LC relaxes the worst-case congestion, it does not confirm the optimal worst-case performance. To pursue this optimality, PSO-WC upgrades the objective function of SO that determines the set of link weights at start time by considering all possible single link failures; its goal is to minimise the worst-case congestion. Numerical results showed that PSO-WC effectively relaxes the worst-case network congestion compared to SO, while it avoids the network instability caused by the run-time changes of link weights caused by RO. At the same time, PSO-WC yields performance superior to that of PSO-LC [**14**]. Ranaweera et al. presented a PSO policy for the hose model, where the exact traffic demand between each source and destination node pair does not need to be specified, to optimise the link weights against link failures [**15, 16**]. Their presented scheme for the hose model employs a heuristic algorithm to determine a suitable set of link weights to reduce the worst-case congestion for any single link failure.

However, the authors in [**12**] pointed out that any application of PSO's results will lead to a penalty in case of no failure, but they do not show how to decrease that penalty. If we consider a network where link failures rarely happen PSO might not be realistic. Using PSO when there is no failure means that we are avoiding would-be critical links even when they might not fail. So network resources are under-utilized and the quantity of data sent throughout the network become smaller. For a network where failures scarcely happen, the under-utilisation will be a burden

overall. Although PSO minimises the worst case congestion ratio, we are in fact carrying a burden compare to SO in networks less prone to link failures, because SO is optimal in case of no link failure. Now let us call this Preventive Start-time Optimisation scheme, a PSO-P (Preventive Start-time with Penalty).

A question arises: Is it possible to find a link weight set that eliminates that penalty while reducing the worst case congestion ratio?

The first part of the work answers this question by proposing a Preventive Start-time with No-Penalty (PSO-NP) [**1**]. PSO-NP generates a link weight set that completely suppresses the penalty and at the same time, reduces substantially the congestion ratio even for the worst case congestion scenario. PSO-NP scheme is based on SO. Under no failure scenario, SO generates many optimal sets but chooses only the first one. PSO-NP evaluates the performance of each of these sets under worst failure and chooses the one that reduces most the congestion ratio. We use two approaches to realise PSO-NP. The first one is an ILP based approach while the second is a heuristic algorithm. We expand PSO-NP into General Start-time Optimisation (GPSO) to find a weight that considers the trade-off between the penalty under no failure and the worst case congestion.

While PSO, PSO-NP and GPSO considers every failure patterns, some of them may not even happen. In fact, vulnerable links might be a very small number of links in the network. For example PSO considers only the worst case failures. For that reason, the network becomes more congested under no failure. That worst case failure may not even happen in reality. Therefore Considering failure probability when in Preventive Start-time Routing is required.

An another question arises: Is it possible to find a link weight set that reduces the expectable congestion ratio?

In the second part of our work we answer this question by proposing PSO-FP (Preventive Start-time Optimization that considers link failure probability). PSO-FP generates a link weight set that minimizes the congestion ratio expectation value when the failure probability of each link is given.

1.1. Network model

The network is represented as a directed graph $G(V, E)$, where V is the set of nodes and E is the set of links. $v \in V$, where $v = 1, 2, \cdots, N$, indicates an individual node, where N is the number of nodes in the network, or $N = |V|$. We consider only single link failure in this work, as the probability of multiple link failure at the same time is much less than that of single link failure. Let a link from node $i \in V$ to node $j \in V$ be denoted as $(i, j) \in E$. L is the number of links in the network, or $L = |E|$. F is the set of link failure indices l, where $l = 0, 1, 2, \ldots, L$ and $F = E \cup \{0\}$. The number of elements in F is $|F| = L + 1$. $l = 0$ indicates no link failure and $l(\neq 0)$ indicates the failure of $(i, j) = l(\neq 0)$. The network in which link $l(\neq 0)$ is considered failed is described as a directed graph $G_l(V, E_l)$. Since $l = 0$ indicates no link failure, $G_0(V, E_0) = G(V, E)$. u_{ij}, $(i, j) \in E$ represents the traffic flowing through link (i, j) and c_{ij} its capacity. If $(i, j) = l$, $c^l_{ij} = 0$. $W = \{w_{ij}\}$ is the link weight set of network G, where w_{ij} is the weight of (i, j). Let $\{1, \ldots, w_{max}\}$ be the set of values possibly taken as a link weights. $x^{pq}_{ij}(W, l)$ is the portion of traffic from node $p \in V$ to node $q \in V$ routed through $(i, j) \in E_l$. Note that $x^{pq}_{ij}(W, l)$ will be determined based on the shortest path routing when link weight set W is applied to the network. In this analysis, it is assumed that Equal-Cost Multi-Path (ECMP) routing is employed, where traffic is evenly split among equal-cost paths [**17**]. $x^{pq}_{ij}(W, l)$ is used to represent the load distribution variables under link weights set W. A traffic matrix T is defined by $T = \{d_{pq}\}$, where d_{pq} represents the traffic rate from node p to node q.

Let consider $E(W)$ the set of links on our transmitting paths when W is our link weight set. The network congestion ratio r refers to the maximum value of all link utilization ratios in the network. r is defined by,

$$r(W) = \max_{(i,j)\in E(W)} \frac{u_{ij}}{c_{ij}}, \tag{1.1}$$

where $0 \leq r(W) \leq 1$. maximising the additional traffic volume is equivalent to minimising $r(W)$ [**18**] by choosing the best weight set W. For any link $l \in F$ failed,

the congestion ratio is defined as:

$$r(W, l) = \max_{(i,j) \in E_l(W)} \frac{u_{ij}}{c_{ij}}, \tag{1.2}$$

and the worst case congestion as:

$$R(W) = \max_{l \in F} r(W, l). \tag{1.3}$$

Note that $r(W, 0)$ represents the congestion ratio in a network when there is no failure. For simplicity, let $r(W, l)$ be denoted as $r(l)$. $r(W, 0)$ is therefore $r(0)$. The paper is organized as follows. In chapter 2, we revisite previous work related to Preventive Start-time Optimisation. In chapter 3, we explain PSO-NP and evaluate its performance. We introduce GPSO in chapter 4 and we discuss PSO-FP in chapter 5. Finally, we conclude our discussion in chapter 6.

CHAPTER 2

Conventionnal Routing Scheme

2.1. SO: Start-time Optimisation

2.1.1. Overview. SO determines an optimal link weight set W that minimises $r(0)$ when link failure is not considered, enabling additional flow in the network. Let us call that set W_{SO}. It is expressed as:

$$W_{SO} = \arg\min_{W} r(0), \tag{2.1}$$

and its corresponding congestion ratio denoted as r_{SO} or $r_{SO}(0)$ to stress the fact that link failure is not considered.

2.1.2. application. Let us consider the illustrative network defined in Fig. 2.1, where all link capacity is set to 1 unit of traffic. Suppose that, we want to send 0.5 unit of traffic from node 1 to node 4 and another 0.5 unit from node 2 to node 4. Under a no link failure scenario, SO enables us to find an optimal link weight set that minimises congestion ratio in our network up to $r_{SO} = 0.375$.

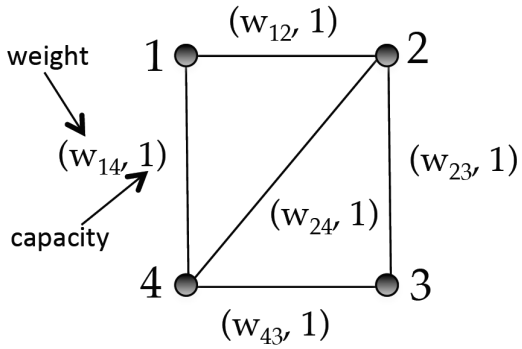

FIGURE 2.1. Illustrative sample network

Unfortunately, SO is not optimal under failure. Let us consider that a failure occurs. The failure that causes the highest congestion is called a worst case failure.

One possible worst case failure is a failure of link (2,4). if (2,4) fails, rerouting the traffic under the same link weight set determined by SO, creates a more congested network with a congestion ratio $\max_{l \in F} r_{SO}(l) = 0.75$. This is almost the double of the congestion ratio under no failure.

To tackle this drawback, PSO-P [**12**] was presented.

2.2. PSO-P: Preventive start-time Optimisation with Penalty

2.2.1. overview. PSO-P determines the best link weight set at the start time to minimise congestion ratio under any possible failure. The main achievement is the preservation of network stability, which is a major drawback for RO, because there are no running-time changes.

PSO-P considers any failure patterns and provides the best link weight set that reduces the worst congestion. Since PSO-P focuses only on finding a link weight set that minimises $R(W)$, a solution is expressed as:

$$W_{PSO-P} = \arg\min_{W} R(W). \tag{2.2}$$

2.2.2. application. In our illustrative network, under the same traffic demand, PSO-P optimal weight set reduces the worst case congestion up to $\max_{l \in F} r_{PSO-P}(l) = 0.5$ which is less than $\max_{l \in F} r_{SO}(l) = 1.0$.

More generally, worst case congestion in SO is higher or equal to that of PSO-P, which is expressed by:

$$R(W_{SO}) \geq R(W_{PSO-P)}. \tag{2.3}$$

In other words, PSO-P outperforms SO under the worst case congestion with a significant worst congestion ratio reduction. We define this reduction ratio of PSO-P as:

$$\alpha_{PSO-P} = \frac{\max_{l \in F} r(W_{SO}, l) - \max_{l \in F} r(W_{PSO-P}, l)}{\max_{l \in F} r(W_{SO}, l)}. \tag{2.4}$$

Note that from Eq. (1.3),

$$R(W_{SO}) = \max_{l \in F} r(W_{SO}, l), \tag{2.5}$$

$$R(W_{PSO-P}) = \max_{l \in F} r(W_{PSO}, l). \tag{2.6}$$

2.2.3. problem statement. The difficulty with PSO-P is that if the link weight set determined by PSO-P is used in a no failure scenario, the congestion ratio may be higher than that of SO.

For example, Fig. 2.1 illustrative network gives us, $r_{PSO-P}(0) = 0.5$ while $r_{SO}(0) = 0.375$. This means that, compared to SO we will have to cary a penalty when there is no failure.

In general,

$$r(W_{SO}, 0) \leq r(W_{PSO-P}, 0). \tag{2.7}$$

Therefore, PSO-P shows a penalty under no failure that we define as:

$$\beta_{PSO-P} = \frac{r(W_{PSO-P}, 0) - r(W_{SO}, 0)}{r(W_{SO}, 0)}. \tag{2.8}$$

β_{PSO-P} raises an issue for networks with relatively few link failures because the penalty is carried on and becomes a burden in the long run.

CHAPTER 3

PSO-NP: Preventive Start-time with No Penalty

3.1. Overview

PSO-NP determines a link weight set that completely suppresses the penalty and at the same time, reduces substantially the congestion ratio even for the worst case congestion scenario.

PSO-NP scheme is based on SO. Under no failure scenario, SO generates a set that minimises the congestion ratio. There is in fact, a possibility that many candidate sets with the same performance exist. SO chooses the first one among these sets. PSO-NP extends SO by evaluating the performance of each of these sets under worst failure to choose the one that reduces most the congestion ratio.

Since the solution weight set of PSO-NP, W_{PSO-NP} is generated through SO, under no failure PSO-NP and SO show the same congestion ratio:

$$r(W_{SO}, 0) = r(W_{PSO-NP}, 0). \tag{3.1}$$

Moreover, PSO-P is optimal under worst case failure. As a result,

$$R(W_{PSO-P}) \leq R(W_{PSO-NP}) \leq R(W_{SO}). \tag{3.2}$$

PSO-NP guarantees *zero* penalty under no failure while boosting protection under failure compared to the typical SO scheme.

In this work we use two approaches to realise PSO-NP. The first one is a direct mathematical approach while the second is a heuristic approach that focuses on both the reduction ratio and the penalty.

3.2. ILP approach

3.2.1. ILP formulation. The ILP formulation of PSO-NP is straightforward in minimising $r(0)$ while decreasing $\max_{l \in F} r(l)$. When the network topology and the traffic demand are known, an optimal weight link set at the start time can be determined using SO. In fact, the optimal link weight set returned by SO may not be unique. SO returns only one among these sets, the first one found. In PSO-NP, all the optimal link weight sets generated by SO are kept and we examine the performance of each optimal set under the worst failure scenario. The most suitable set will be the one showing the lowest worst congestion under any single failure scenario.

We extend the ILP formulation of SO [**19**] to PSO-NP, which is expressed as follows:

(3.3a) Objective $\min_{W} \quad r_{SO} + \epsilon \cdot r_{PSO}$

s.t. $$\sum_{j:(i,j)\in E_l} x_{ij}^{pq}(W,l) - \sum_{j:(i,j)\in E_l} x_{ji}^{pq}(W,l) = 1,$$

(3.3b) $$\forall p,q \in V, i = p, l \in F$$

$$\sum_{j:(i,j)\in E_l} x_{ij}^{pq}(W,l) - \sum_{j:(i,j)\in E_l} x_{ji}^{pq}(W,l) = 0,$$

(3.3c) $$\forall p,q \in V, i(\neq p,q), l \in F$$

$$\sum_{p,q\in V} d_{pq} x_{ij}^{pq}(W,0) \leq c_{ij}^{0} \cdot r_{SO},$$

(3.3d) $$\forall (i,j) \in E_0$$

$$\sum_{p,q\in V} t_{pq} x_{ij}^{pq}(W,l) \leq c_{ij}^{l} \cdot r_{PSO},$$

(3.3e) $$\forall (i,j) \in E_l, l \in F$$

$$0 \leq f_{pq}^{i}(l) - x_{ij}^{pq}(W,l) \leq 1 - \delta_{q}^{ij}(l), \forall p,q \in V,$$

(3.3f) $$(i,j) \in E_l, l \in F$$

$$x_{ij}^{pq}(W,l) \leq \delta_{q}^{ij}(l), \forall p,q \in V,$$

(3.3g) $$(i,j) \in E_l, l \in F$$

$$0 \leq \psi^{jq}(l) + w_{ij} - \psi^{iq}(l) \leq (1 - \delta_{q}^{ij}(l))U,$$

(3.3h) $$\forall q \in V, (i,j) \in E_l, l \in F$$

$$1 - \delta_{q}^{pq}(l) \leq \psi^{jq}(l) + q_{ij} \leq \psi^{iq}(l),$$

(3.3i) $$\forall q \in V, (i,j) \in E_l, l \in F$$

(3.3j) $$f_{pq}^{i}(l) \geq 0, \forall p,q,i \in V, l \in F$$

(3.3k) $$\delta_{q}^{ij}(l) \in \{0,1\}, \forall q \in V, (i,j) \in E_l, l \in F$$

(3.3l) $$1 \leq w_{ij} \leq w_{max}, \forall (i,j) \in E_l, l \in F.$$

The key decision variable of this ILP problem is the link weight set W. In the objective function, r_{SO} and r_{PSO} which are also decision variables represent $r(0)$and $\max_{l\in F} r(l)$ respectively. A constant value of ϵ is set so that $\frac{\epsilon \cdot r_{PSO}}{r_{SO}} << 1$ can be satisfied. ϵ is small enough to enable us to find the weight set that reduces most $\max_{l\in F} r(l)$ among the ones minimising r_{SO}. Other decision variables are $c_{ij}^{l}, f_{pq}^{i}(l), x_{ij}^{pq}(W,l), \delta_{q}^{ij}(l), \delta_{q}^{ij}$ and $\psi^{jq}(l)$. These variables are determined once W is calculated.

The constraints of Eqs. (3.3b)-(3.3l) are explained as follows. Eqs. (3.3b) and (3.3c) express the flow conservation constraints. Eq. (3.3d) expresses the capacity constraint for each link by using the network congestion ratio, r_{SO}, in case of no link failure. Eq. (3.3e) expresses the capacity constraint for each link by using the worst-case network congestion ratio, r_{PSO}, for any single link failure. Eqs. (3.3f) - (3.3i) indicate the constraints of the ECMP routing, where U is a given constant with a sufficiently large value. The details of the ECMP routing constraints are explained in [**19**]. Eqs. (3.3j)-(3.3l) give the types and ranges of the decision variables.

3.2.1.1. *Application of ILP approach.* Sample networks used are shown in Fig. 3.1. We consider the traffic matrix given.

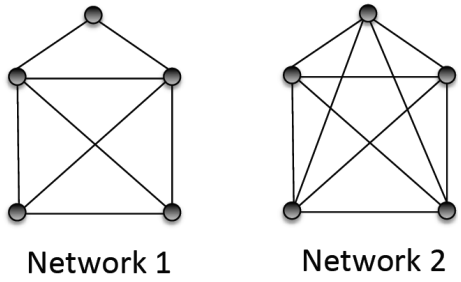

FIGURE 3.1. Sample Networks for ILP

We determine W_{PSO-NP} by solving the ILP problem defined in Eqs. (3.3a)-(3.3l). For $\epsilon = 0$, we obtain W_{SO}. Finally we change Eq. (3.3a) into:

$$\text{Objective} \quad \min_{W} \quad r_{PSO}, \tag{3.4}$$

to obtain W_{PSO-P}. From these three sets we deduce α_{SO}, α_{PSO-NP}, α_{PSO-P}, β_{SO}, β_{PSO-NP}, β_{PSO-P}.

Table 3.1 shows the comparison of PSO-P, SO, PSO-NP schemes for both the worst case failure and the no failure scenario. By definition both α_{SO} and β_{SO} are qual to zero. Note that the congestion ratios are normalized by that of SO under no failure scenario.

TABLE 3.1. Comparisons of PSO-NP and PSO-P performance in Network 1 based on ILP formulation

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.11	1.22	0.39	0.11
PSO-NP	1.00	1.22	0.39	0.00
SO	1.00	2.00	0.00	0.00

TABLE 3.2. Comparisons of PSO-NP and PSO-P performance in Network 2 based on ILP formulation

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.02	1.07	0.46	0.02
PSO-NP	1.00	1.51	0.23	0.00
SO	1.00	1.97	0.00	0.00

From Table 3.1, PSO-NP reduces the worst congestion ratio up 39%, equalling PSO-P performance. While PSO-P show a penalty of 11% under no failure case, PSO-NP keeps the penalty to *zero*. PSO-NP generated link weight set is therefore effective for both failure and non failure scenarios. In Table 3.2, PSO-NP still reduces worst case congestion ratio. Even though PSO-P shows a higher reduction ratio it still has a penalty. In this case, Network operators can select one of either depending on the quality of service they need to deliver. In any case, PSO-NP is certainly effective in a network with few failures, where a performance close to SO's is required when there is no failure in the network.

In the ILP approach, it is not possible to get the optimal solution when the network size becomes large. For this reason we present a heuristic algorithm to determine W_{PSO-NP} in larger networks.

3.3. Heuristic approach

3.3.1. Overview. Since under a no failure scenario SO is optimal and returns only one out multiple link weight set solution, let W_{SO}^k be a solution candidate generated by SO, where k is an index of each candidates. Let us call this set of weights candidates S. In a no failure case, all these sets present the same congestion ratio that is the minimal congestion ratio by definition of SO. In other terms $r(W_{SO}^k, 0) = r(W_{SO}, 0)$ for all $W_{SO}^k \in S$. Thus the penalty for all the members of S under no failure scenario would be such that:

$$\beta_k = \frac{r(W_{SO}^k, 0) - r(W_{SO}, 0)}{r(W_{SO}, 0)} = 0, \tag{3.5}$$

for all $W_{SO}^k \in S$. Since all elements of S show the lowest congestion ratio under no failure, the one showing the highest congestion ratio under worst case failure is by definition W_{PSO-NP}. Let

$$\alpha_k = \frac{\max_{l \in F} r(W_{SO}, l) - \max_{l \in F} r(W_{SO}^k, l)}{\max_{l \in F} r(W_{SO}, l)}, \tag{3.6}$$

be the reduction ratio of the worst case congestion of $W_{SO}^k \in S$. Under worst case failure, it is desirable to have a large value of α as it represents the worst congestion reduction ratio. Since W_{PSO-NP} is the element of S with the lowest congestion ratio under worst case failure, it is equivalent to the set with the largest α. Therefore,

$$W_{PSO-NP} \equiv \arg \max_{W_{SO}^k \in S} \alpha_{SO}^k. \tag{3.7}$$

Under worst case failure, W_{PSO-NP} reduction ratio compared to W_{SO} is defined as:

$$\alpha_{PSO-NP} = \frac{\max_{l \in F} r(W_{SO}, l) - \max_{l \in F} r(W_{PSO-NP}, l)}{\max_{l \in F} r(W_{SO}, l)}, \tag{3.8}$$

and its penalty under no failure,

$$\beta_{PSO-NP} = \frac{r(W_{PSO-NP}, 0) - r(W_{SO}, 0)}{r(W_{SO}, 0)}, \tag{3.9}$$

which is, of course, *zero*.

3.3.2. Procedure. Since PSO-NP selects one set among SO generated weight sets, the first step of our procedure is to implement SO and generate S. In our procedure, z represents a fixed increment value in the link weight searching procedure and S is initialized to $S = \emptyset$. w_{max} is set to 1000.

The procedure taken by PSO-NP to obtain W_{PSO-NP} is defined by Eq. (3.7) and divided into three steps as follows.

- Step 1: Get W_{SO} and collect all sets that generates the same $r(W_{SO}, 0)$. These sets are elements of S.
- Step 2: For each $W_{SO}^k \in S$, evaluate α_k, using Eq. (3.6).
- Step 3: Get W_{PSO-NP} by using Eq. (3.7).

Step 1 is divided into 4 steps.

In Step 1, r and I are variables used to respectively store the congestion ratio and count the number of generated weight sets. the variables are respectively initialized to $r = \infty$ and $I = 1$. The procedure in step 1 is as follow:

- Step 1.1:

  Generate new link weight set W_{init}.

  $W_{temp} \leftarrow W_{init}$.

  $flag \leftarrow true$.

- Step 1.2:

  While (flag)

  Run shortest path algorithm and find the link

  (i_0, j_0) with the highest congestion ratio r_0.

  if($r_0 = r$)

  add W_{temp} into S.

  if($r_0 < r$)

  $r \leftarrow r_0$.

  clear S.

add W_{temp} into S.

Update W_{temp} by replacing $w_{i_0j_0} \leftarrow w_{i_0j_0} + z$.

if $(w_{i_0j_0} + z > w_{max})$

$flag \leftarrow false.$

- Step 1.3:

Increment I.

if $(I \leq I_{max})$

go to step 1.1.

else

return $S = \{W^1_{SO}, W^2_{SO}, W^3_{SO}, \ldots\}$.

end.

3.3.3. Computation time complexity. The computation time complexity of PSO-NP is the total sum of the computation time complexity of steps 1, 2, and step 3.

The computation time complexity of step 1 is equal to that of SO. Under the SO Optimisation process, if we focus on determining one link weight, there are w_{max} combinations for each link weight. To determine L link weights, we do not decrement link weights and we only increment them at most Lw_{max} times to get the SO solution. Therefore, we have Lw_{max} weight-set combinations to determine L link weights for a given initial link weight set. For each weight set, we need to compute the network congestion ratio. Let $O(X)$ be the computation time complexity to find the congestion ratio for each weight set. Moreover, we need to compute all the initial I_{max} sets. Therefore, the computation complexity of SO is $O(Lw_{max}XI_{max})$.

Consider the computation time complexity of steps 2 and 3. To determine the worst case congestion ratio we need to examine all possible link failure patterns including the no failure scenario, which gives us a total of $L+1$ failure patterns. Therefore, the computation complexity to get the worst case congestion ratio is $O(LX)$ for a link weight set in S. We need this computation time complexity for every element of

S. Let S_{max} be the maximum number of elements in S. The total computation time complexity of steps 2 and 3 is $O(LXS_{max})$.

In all, the PSO-NP computation time complexity is expressed by,

$$O(Lw_{max}XI_{max} + LXS_{max}). \tag{3.10}$$

$O(X)$ can be evaluated by using the computation time complexity of Dijkstra's algorithm. In Dijkstra's algorithm, for each source node a shortest path tree is computed using a computation complexity of $O(L+N\log N)$. Since we use Dijkstra's algorithm to every node in the network to determine the congestion ratio after a single failure, $O(X) = O((L + N\log N) \times N) = O(NL + N^2 \log N)$.

By substituting $X = NL + N^2 \log N$ into Eq. (3.10), the PSO-NP computation time complexity is expressed by,

$$O((w_{max}I_{max} + S_{max})(NL^2 + N^2L\log N)). \tag{3.11}$$

w_{max}, I_{max}, and S_{max} are the parameters that can be controlled. To enhance the Optimisation accuracy, we set $w_{max}I_{max} > S_{max}$. In this case, the PSO-NP computation time complexity is expressed by $O(w_{max}I_{max}(NL^2 + N^2L\log N))$, which is the same computation time complexity of SO.

3.4. Performance evaluation and simulation environments

The performances of the PSO-P, PSO-NP and SO are compared via simulations for both failure and non-failure scenarios through our heuristic approach. For each scheme we evaluate the congestion ratio under worst case failure and non failure scenarios. Comparison metrics are respectively the reduction ratio of the worst congestion,

$$\alpha_X = \frac{\max_{l\in F} r(W_{SO}, l) - \max_{l\in F} r(W_X, l)}{\max_{l\in F} r(W_{SO}, l)}. \tag{3.12}$$

and the penalty under no failure,

$$\beta_X = \frac{r(W_X, 0) - r(W_{SO}, 0)}{r(W_{SO}, 0)}, \tag{3.13}$$

where X = PSO-P, PSO-NP and SO.

We use five sample networks, as shown in Fig. 5.1.

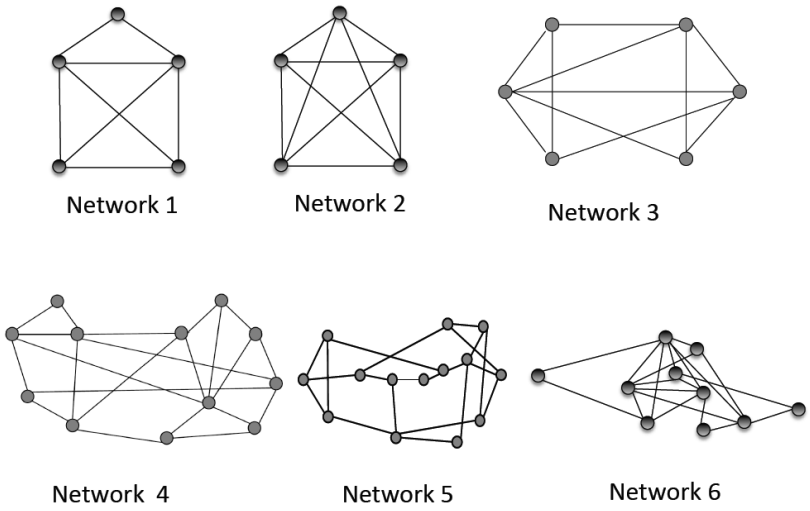

FIGURE 3.2. Examined network topologies.

Networks 1 and 2 are the examined topologies used in the ILP approach of PSO-NP. Networks 3 and 4, which mirror typical backbone networks are used to evaluate routing performance in [**20**]. Networks 5 is an Abiline network [**21**]. Finally, network 6 is a random network generated via BRITE [**22**]. The characteristics of networks considered in this work are shown in Table 5.1.

TABLE 3.3. characteristics of networks

Network type	No. of nodes	No. of links (bidirectional)	Aver. node degree
Network 1	5	8	1.60
Network 2	5	10	2.00
Network 3	6	11	1.83
Network 4	12	22	1.83
Network 5	14	21	1.50
Network 6	10	20	2.00

For the given networks, link capacities are randomly generated with uniform distribution in the range of ($10U_C$,$100U_C$), where U_C [Gbit/s] is given a constant integer

value. d_{pq} is also randomly generated with uniform distribution in the range of (0, $100U_D$), where U_D [Gbit/s] is a given constant integer value. U_D/U_C is determined so that we can get feasible solutions. z is set to 1 to evaluate the performance of a maximum number of weight sets. Tables 3.4, 3.5, 3.6, 3.7, 3.8 and 3.9 show respectively the results obtained for networks 1, 2, 3, 4, 5 and 6. As in the ILP approach, the congestion ratios are normalised by that of SO.

TABLE 3.4. Comparisons of PSO-NP and PSO-P performance in Network 1

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.11	1.22	0.39	0.11
PSO-NP	1.00	1.22	0.39	0.00
SO	1.00	2.00	0.00	0.00

TABLE 3.5. Comparisons of PSO-NP and PSO-P performance in Network 2

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.02	1.07	0.46	0.02
PSO-NP	1.00	1.51	0.23	0.00
SO	1.00	1.97	0.00	0.00

TABLE 3.6. Comparisons of PSO-NP and PSO-P performance in Network 3

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.42	1.47	0.45	0.42
PSO-NP	1.00	1.50	0.44	0.00
SO	1.00	2.68	0.00	0.00

TABLE 3.7. Comparisons of PSO-NP and PSO-P performance in Network 4

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.25	1.56	0.80	0.25
PSO-NP	1.00	2.46	0.68	0.00
SO	1.00	7.78	0.00	0.00

For both networks 1 and 2, the results obtained through the heuristic approach equal the the ones obtained through the ILP approach. This can be observed by comparing Tables 3.1 and 3.2 to Tables 3.4 and 3.5, respectively. In addition, From TABLE 3.6 and 3.9, we can state that PSO-NP reduces the worst case congestion ratio almost as much as PSO-P while suppressing the penalty which PSO-P does not.

TABLE 3.8. Comparisons of PSO-NP and PSO-P performance in Network 5

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.67	2.03	0.14	0.67
PSO-NP	1.00	2.37	0.00	0.00
SO	1.00	2.37	0.00	0.00

TABLE 3.9. Comparisons of PSO-NP and PSO-P performance in Network 6

	$r(0)$	$\max_{l \in F} r(l)$	α	β
PSO-P	1.85	1.83	0.66	0.85
PSO-NP	1.00	2.02	0.63	0.00
SO	1.00	5.41	0.00	0.00

PSO-P carries a penalty of 42% and 85% for networks 3 and 6 respectively when there is no failure. This shows the effectiveness of PSO-NP in reducing the worst congestion ratio while eliminating the penalty when there no failure. In Table 3.7, for network 4, the reduction ratio of PSO-NP is lower than PSO-P's but still has the advantage of keeping penalty to zero. For this case also, network operators can select one of either depending on the quality of service they need to deliver. Finally, in Table 3.8, for network 5, PSO-NP does not show any amelioration compared to SO. We study this case later in this paper.

For all the samples examined here, the penalty in case of no failure β, numerically confirmed our prediction as follows:

$$\beta_{SO}(=0) = \beta_{PSO-NP} \leq \beta_{PSO-P}, \tag{3.14}$$

where PSO-NP maintains the penalty to *zero*. For the worst case single link failure, we also compared the reduction ratio for both PSO-NP and PSO-P schemes. We observed:

$$\alpha_{SO}(=0) \leq \alpha_{PSO-NP} \leq \alpha_{PSO-P}, \tag{3.15}$$

For larger networks the effectiveness of PSO-NP depends on I_{max} values. For larger values of I_{max}, α_{PSO-NP} may be boosted and even match α_{PSO-P} because more initial weight sets means more possible elements of S. However, in smaller

networks such as network 1 and 2, the variety of initial weight sets does not impact the performance of PSO-NP. We examined the performance of PSO-NP for the values of I_{max} ranging from 1 to 10000 in Network 3, 4 5 and 6. The results are listed in Tables 3.10, 3.12 3.11 and 3.13.

TABLE 3.10. Dependency of PSO-NP over I_{max} in Network 3

I_{max}	α_{PSO-P}	α_{PSO-NP}	β_{PSO-P}	β_{PSO-NP}
1	0.18	0.09	0.28	0.00
3	0.18	0.09	0.28	0.00
5	0.18	0.09	0.28	0.00
20	0.27	0.12	0.27	0.00
100	0.44	0.32	0.16	0.00
1000	0.45	0.44	0.41	0.00
5000	0.45	0.44	0.41	0.00
10000	0.45	0.44	0.42	0.00

TABLE 3.11. Dependency of PSO-NP over I_{max} in Network 4

I_{max}	α_{PSO-P}	α_{PSO-NP}	β_{PSO-P}	β_{PSO-NP}
1	0.62	0.00	0.43	0.00
3	0.62	0.00	0.43	0.00
5	0.62	0.00	0.43	0.00
30	0.62	0.00	0.43	0.00
100	0.62	0.00	0.43	0.00
1000	0.58	0.00	0.47	0.00
5000	0.36	0.00	0.25	0.00
10000	0.80	0.68	0.25	0.00

TABLE 3.12. Dependency of PSO-NP over I_{max} in Network 5

I_{max}	α_{PSO-P}	α_{PSO-NP}	β_{PSO-P}	β_{PSO-NP}
1	0.003	0.00	0.72	0.00
3	0.003	0.00	0.72	0.00
5	0.003	0.00	0.72	0.00
30	0.003	0.00	0.72	0.00
100	0.003	0.00	0.62	0.00
1000	0.003	0.00	0.75	0.00
5000	0.003	0.00	0.72	0.00
10000	0.14	0.00	0.67	0.00

In all, PSO-NP heuristic algorithm eliminates PSO-P penalty while producing a considerable congestion ratio reduction under the worst case failure scenario. It may even reduce the worst congestion ratio as PSO-P does for some cases. In the examined cases, increasing the number of initial sets ameliorates our solution.

TABLE 3.13. Dependency of PSO-NP over I_{max} in Network 6

I_{max}	α_{PSO-P}	α_{PSO-NP}	β_{PSO-P}	β_{PSO-NP}
1	0.37	0.00	0.00	0.00
3	0.37	0.00	0.00	0.00
5	0.37	0.00	0.00	0.00
20	0.37	0.00	0.43	0.00
100	0.63	0.60	0.85	0.00
1000	0.69	0.60	0.85	0.00
5000	0.66	0.63	0.85	0.00
10000	0.66	0.63	0.85	0.00

The computation times of SO, PSO-NP, PSO-P are examined. The Simulation is performed by using a Linux-based computer equipped with a 2.3 GHz Intel Core i5 Processor and 4 GB memory. Computation time is evaluated in second (s) and shown in Table 5.6.

TABLE 3.14. Computation time comparison of SO, PSO-NP and PSO-P in seconds

Network type	SO	PSO-NP	PSO-P
Network 1	1620.22	1848.48	3892.85
Network 2	1685.79	1723.13	8568.55
Network 3	3157.54	3140.79	18026.84
Network 4	31603.00	32129.77	336610.75
Network 5	42136.04	43073.50	627295.30
Network 6	16291.27	16559.43	182786.63

We observe that PSO-NP computation time is close to that of SO as we expected. PSO-NP extends SO by evaluating SO generated sets worst congestion ratio.

PSO-P computation time is 2 to 10 times that of PSO-NP depending on the network size. Compared to PSO-P, PSO-NP reduces the worst congestion ratio in a shorter time while keeping the penalty to *zero*.

3.5. Variations of PSOs

PSO-NP basically keeps $r(0) = r_{SO}(0) = 1.00$ while reducing $\max_{l \in F} r(l)$ but may not be effective in some cases. In Table 3.8 for example PSO-NP and SO show the same result. There is no advantage of PSO-NP compared to SO under the worst case failure. Now, if we try to loosen the constraint under no failure, for example $r(0) = 1.001$ compared to $r_{SO}(0) = 1.0$, it is actually possible to get a substantial

decrease of $\max_{l \in F} r(l)$. In this case the penalty $\beta = 0.001$ and is almost negligible while α is boosted. Therefore, a limited or small enough penalty under no failure may yield a substantial reduction ratio under worst case failure. A preventive start-time that considers a limited penalty while increasing the reduction ratio should be examined. This scheme is called Preventive Start-time with Limited Penalty (PSO-LP).

PSO-LP can even be extended so as to consider the balance between $r(0)$ and $\max_{l \in F} r(l)$ through a General Preventive Start-time Optimisation(GPSO).

CHAPTER 4

GPSO: Generalized preventive Start-time Optimization

4.1. Overview

GPSO determines a suitable weight set that balances $r(0)$ and $\max_{l \in F} r(l)$ and enables network operators to select the link weight set that shows the best couple $(r(0), \max_{l \in F} r(l))$. As in network 6, PSO-NP may sometimes coincide with SO under worst case failure. For those cases, the reduction ratio α cannot be improved. Actually the reduction ratio improves if we loosen the penalty β under no failure. Therefore finding a link weight set that considers the balance of both α and β makes senses. We propose a way to find that set through a mathematical and heuristic approach.

4.2. GPSO mathematical formulation

GPSO is designed to give network operator the latitude to get suitable link weight set considering the balance between $r(0)$ and $\max_{l \in F} r(l)$ instead of focusing on one of them. GPSO ILP formulation derives from Eq. (3.3a). The formulation can be expressed by changing only the objective function of our PSO-NP ILP formulation in to:

$$\text{Objective} \quad \min_{W} \quad (1-\gamma) \cdot r_{SO} + \gamma \cdot r_{PSO}, \tag{4.1}$$

where $0 \leq \gamma \leq 1$. γ is a parameter defined by network operators. $\gamma = 0$ means that we do not consider worst case failure. In other terms only r_{SO} becomes our objective function. This shows that setting $\gamma = 0$ can enable us to recover SO. In addition, when γ is set to 1, the objective function becomes r_{PSO}, meaning that only the worst case congestion ratio is considered. GPSO becomes equivalent to PSO-P.

Finally, when γ increases from 0 towards 1, GPSO shifts progressively from PSO-NP, PSO-LP and PSO-P. This enables network operators to find the best couple $(r(0), \max_{l \in F} r(l))$ instead of just targeting on one of them as in SO and PSO-P.

In total GPSO is summarised as follows:

$$GPSO = \begin{cases} SO, & \text{if} \quad \gamma = 0 \\ PSO - NP, & \text{if} \quad 0 < \gamma \ll \frac{r_{SO}}{r_{PSO}} \\ PSO - LP, & \text{if} \quad \frac{r_{SO}}{r_{PSO}} < \gamma < 1 \\ PSO - P, & \text{if} \quad \gamma = 1 \end{cases}$$

Unfortunately, for larger topologies like network 5, the ILP approach is not able to give us a solution in a practical time. A heuristic formulation of GPSO is therefore required.

4.3. GPSO heuristic algorithm

In our heuristic algorithm we directly evaluate the balance between α and β because finding the weight set that gives the best $(r(0), \max_{l \in F} r(l))$ is equivalent to finding the one with the best (α, β). For that, we allow the congestion ratio under no failure $r(0)$ to have a range. That range is set as:

$$r_{SO}(0) \leq r(0) \leq r_{SO}(0) + \delta, \quad \delta > 0, \tag{4.2}$$

where δ is given. In other words, under no failure the penalty is no kept at *zero* like PSO-NP but bounded as:

$$0 \leq \beta \leq \beta_{upp}, \tag{4.3}$$

where

$$\beta_{upp} = \frac{\delta}{r_{SO}(0)} \tag{4.4}$$

is the upper bound penalty.

The upper bound penalty is the maximal allowable penalty when there is no failure in the network. For a given value of β_{upp}, GPSO finds the weight set that reduces most $\max_{l \in F} r(l)$ boosting α as a consequence. β_{PSO-P} is the highest possible penalty to pay under a no failure scenario compared to SO scheme. Therefore the range of β_{upp} is:

$$0 \leq \beta_{upp} \leq \beta_{PSO-P}, \tag{4.5}$$

Since β_{upp} represents the maximal allowable penalty under no failure in the network, the solution weight set of GPSO can be expressed as:

$$W_{GSO_{\beta_{upp}}} = \underset{0 \leq \beta \leq \beta_{upp}}{arg} \max_{W} \alpha(W). \tag{4.6}$$

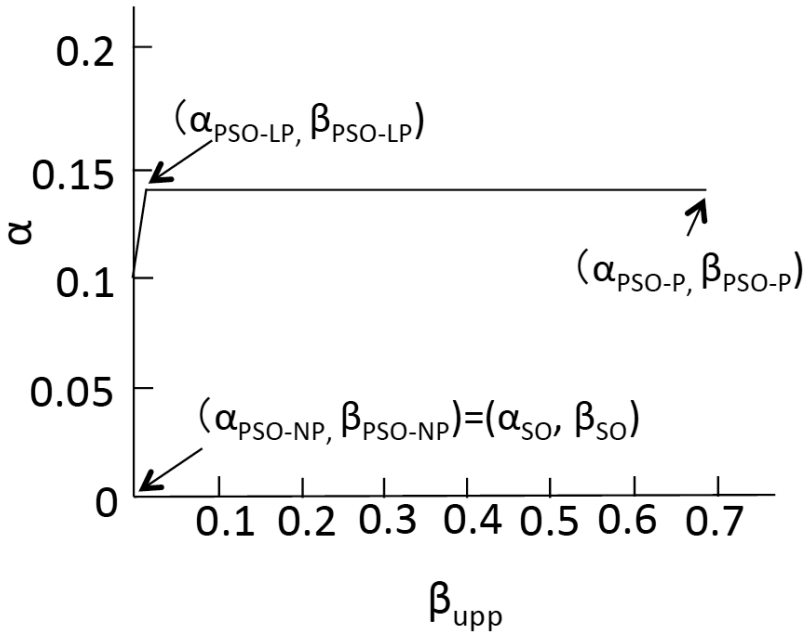

FIGURE 4.1. Worst congestion reduction ratio for given penalty under no failure

Now, if we shift the values of β_{upp} progressively from 0 to β_{PSO-P} we obtain respectively PSO-NP, PSO-LP an PSO-P scheme. In General, we have:

$$GPSO = \begin{cases} PSO - NP, & \text{if} \quad \beta_{upp} = 0 \\ PSO - LP, & \text{if} \quad 0 < \beta_{upp} < 1 \\ PSO - P, & \text{if} \quad \beta_{upp} = \beta_{PSO-P} \end{cases}$$

We examined the performance of GPSO in network 5. As shown in Fig. 4.1 for $\beta_{upp} = 0.0015$ we achieve a reduction equal to that of PSO-P showing that with negligible penalty maximal reduction can be achieved. In addition, for values of β_{upp} equal to 0 and β_{PSO-P} GPSO matches PSO-NP and PSO-P reduction ratio respectively. GPSO therefore generalizes the previous schemes and goes further to provide a more flexible weight set depending on upper bound penalty when there is no failure.

CHAPTER 5

PSO-FP: Preventive Start-time Optimization Considering link Failure probability

5.1. Overview

PSO-FP determines the link weight set that minimizes the worst congestion ratio based on the link failure probability.

When W is fixed, for $l \in E$, $(p_l,\ r(W,l))$ couples are known. PSO-FP aim is to calculate a set that minimizes the expectation value $\sum_{l\in E} P_l r(W,l)$. Therefore a mathematical formulation of a possible solution to our problem can be expressed as:

$$\min_{W} \quad \sum_{l\in E} p_l r(W,l). \tag{5.1}$$

Many weight set may satisfy this formulation of PSO-FP. Under no failure the congestion ratio should be as low as possible. As a result our Objective function is:

$$\text{Objective} \quad \min_{W} \quad \sum_{l\in E} p_l r(W,l) + \epsilon \cdot r(0). \tag{5.2}$$

Now, let

$$W_{PSO-FP} = \arg \quad \min_{W} \quad \sum_{l\in E} p_l r(W,l) + \epsilon \cdot r(0), \tag{5.3}$$

the solution set. We show below how to find the solution set.

5.2. Procedure

Our procedure used Simulated annealing search ,where a link is randomly selected and its weight changed. In our Procedure W_sol the final weight set solution. W_{init} represents a initial set use in the optimization procedure. We use I_{max} initial weight sets. *step* represents the number of times we select a link randomly to change its

weight. M_0 is used as a temporary variable to store our expected congestion ratio while M is the minimal expectable congestion ratio. M and *step* are initialized as $M = \infty$, step = E× V. The rest of our procedure is as follow:

while(I_{max}){
Generate an initial W_{init}
$W_{temp} \leftarrow W_{init}$
while(*step*){
flag $\leftarrow false$
Select randomly a link $(u, v) \in E$.
Update W_{temp} (W^{up}_{temp})by increasing or decreasing w_{uv} by a value z.
$M_0 \leftarrow average\_congestion(W_{temp})$.
if (M_0>M){
$M \leftarrow M_0$.
flag $\leftarrow true$
$W_{sol} \leftarrow W_{temp}$.
}
if(flag)}
$W_{temp} \leftarrow W^{up}_{temp}$.
step is renitialized.
}
else{
decrement *step*.
{
}
decrement I_{max}.
}

Now the function, $average\_congestion(W)$ gives the expected congestion ratio when the argument weight set is given. it is defined as follow:

double $average\_congestion$(weight set W){

initialize $m_0 = 0$.

for all $l \in F$ {

get G_l.

calculate $r(W, l)$.

$m_0 \leftarrow m_0 + r(W, l) \times P_l$.

}

return m_0.

}

5.3. Performance Evaluation & Simulation Environments

The performance of the PSO-FP scheme is compared to those of PSO and SO scheme via simulations for both failure and non-failure scenarios. For each scheme we evaluate the worst case congestion ratio, the congestion ratio under no failure scenario and the expected congestion ratio. For that, we use three comparison variables. Let X =PSO-FP, PSO or SO.The first comparison variable is the penalty β under no failure:

$$\beta_X = \frac{r(W_X, 0) - r(W_{SO}, 0)}{r(W_{SO}, 0)}. \tag{5.4}$$

The second comparison is δ. δ represents the risk of not using PSO under worst case failure and is defined as:

$$\delta_X = \frac{\max_{l \in F} r(W_X, l) - \max_{l \in F} r(W_{PSO}, l)}{\max_{l \in F} r(W_{PSO}, l)}. \tag{5.5}$$

The last comparison variable is γ. γ represents the gap between the expected congestion ratio given by PSO-FP and other schemes.

$$\gamma_X = \frac{M_X - M_{PSO-FP}}{M_{PSO-FP}}. \tag{5.6}$$

with $M_X = \sum_{l \in E} p_l r(W_X, l)$.

We use four sample networks, as shown in Fig. 5.1.

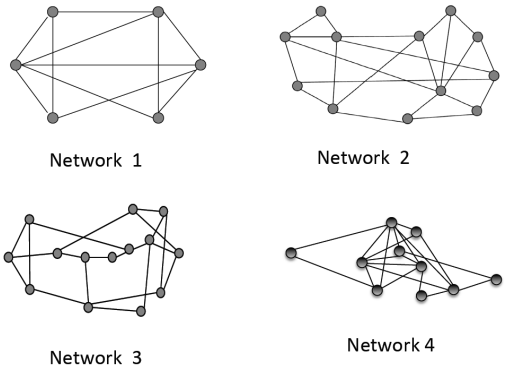

FIGURE 5.1. Examined network topologies.

Networks 1 and 2, which mirror typical backbone networks are used to evaluate routing performance in [**20**]. Networks 3 is an Abiline network [**21**]. Finally, network 4 is a random network generated via BRITE [**22**]. The characteristics of networks considered in this work are shown in Table 5.1.

TABLE 5.1. characteristics of networks

Network type	No. of nodes	No. of links (bidirectional)	Aver. node degree
Network 1	6	11	1.83
Network 2	12	22	1.83
Network 3	14	21	1.50
Network 4	10	20	2.00

For the given networks, link capacities are randomly generated with uniform distribution in the range of ($10U_C$,$100U_C$), where U_C [Gbit/s] is given a constant integer value. d_{pq} is also randomly generated with uniform distribution in the range of (0, $100U_D$), where U_D [Gbit/s] is a given constant integer value. U_D/U_C is determined so that we can get feasible solutions. z is set to 1 to evaluate the performance of a maximum number of weight sets. Also I_{max} and *step* are set respectively to 1000 and $E \times V$.

TABLE 5.2. Comparisons of PSO-FP and PSO performance in Network 1

	$r(0)$	$\max_{l \in F} r(l)$	M	β	δ	γ
SO	1.0	1.97	4.55	0.0	0.27	27.31
PSO	1.19	1.55	2.37	0.19	0.0	13.76
PSO-FP	1.34	2.02	0.16	0.34	0.30	0.0

TABLE 5.3. Comparisons of PSO-FP and PSO performance in Network 2

	$r(0)$	$\max_{l \in F} r(l)$	M	β	δ	γ
SO	1.0	1.85	0.90	0.0	0.09	4.13
PSO	1.26	1.70	0.91	0.26	0.0	4.20
PSO-FP	1.57	2.00	0.18	0.57	0.18	0.0

TABLE 5.4. Comparisons of PSO-FP and PSO performance in Network 3

	$r(0)$	$\max_{l \in F} r(l)$	M	β	δ	γ
SO	1.0	1.70	0.33	0.00	0.15	0.54
PSO	1.18	1.48	0.39	0.18	0.00	0.85
PSO-FP	1.43	1.78	0.21	0.43	0.20	0.00

TABLE 5.5. Comparisons of PSO-FP and PSO performance in Network 4

	$r(0)$	$\max_{l \in F} r(l)$	M	β	δ	γ
SO	1.0	2.05	0.62	0.00	0.00	1.92
PSO	1.28	2.05	0.32	0.28	0.00	0.50
PSO-FP	1.46	2.19	0.21	0.46	0.07	0.00

TABLE 5.2, 5.3, 5.4 and 5.5 show respectively the results obtained for the sample networks 1, 2, 3 and 4. We can say that under no failure case PSO-FP congestion ratio is close to that of SO while SO average congestion ratio is high compare to PSO-FP's. In addition, under worst case failure scenario, PSO-FP congestion ratio is closer to that of PSO while PSO average congestion ratio is greater than PSO-FP's.

The computation time of SO, PSO and PSO-FP is compared. and show in table 5.6. PSO-FP computation time is is much more lower than that of SO and PSO.

TABLE 5.6. SO, PSO and PSO-FP Computation time comparison in seconds

Network type	SO	PSO	PSO-FP
Network 1	610.58	2491.40	14.61
Network 2	6821.74	135362.71	889.47
Network 3	6612.07	103392.75	1818.56
Network 4	3119.18	31842.83	430.35

PSO also shows a congestion that nears both SO and PSO more than PSO and SO near itself. When link failure probability is known PSO-FP is a useful preventive scheme that enables us to determine a link weight set that minimizes the average of the worst possible congestion ratio in the network.

CHAPTER 6

Conclusion

Our objective is to determine, at the start time a link weight set that counters any link failure scenario while considering link failure characteristics. Previous studies have focused on finding a set of link weights that minimises the worst case congestion ratio under *only* a failure scenario. Applying these link weight sets in the network may cause a larger than normal congestion under a non-failure scenario. This issue was mentioned in [**12**] but was not addressed. We have addressed this problem because in networks with few failure, that burden would be carried all along and become troublesome.

We have proposed a scheme to eliminate that penalty while reducing the worst case congestion ratio. Our scheme is simple and based on SO (Start-time Optimisation) scheme. SO enables us to find the optimal link weight set that minimises the congestion ratio under a non-failure scenario. We also know that SO's solution is not unique and there are actually many sets calculated by SO that would produce the same result. The proposed scheme relies on choosing among the solutions of SO the set best prepared to deal with the worst case congestion in our network.

In our evaluation, we used two approaches, a direct ILP based mathematical approach and heuristic algorithm. We verified that both approaches matches in terms of performance for networks of simple size. For network of bigger size, only the heuristic algorithm was explored as the ILP formulation cannot be solved in a practical time. Still, PSO-NP is able to guarantee SO performance in case of no failure while significantly reducing the congestion ratio under the worst case failure. In some cases PSO-P performance under failure was even matched. PSO-NP is therefore applicable for networks with few failures because unlike previous schemes it does not carry any

penalty in case of no failure and can still manage to reduce notably the worst case congestion ratio under any single link failure scenario.

However, in some cases loosening the penalty to a negligible value could lead to even bigger reduction of the congestion ratio under worst case failure scenario. We explored the trade-off between the penalty when there is no failure and the reduction ratio of the worst case congestion through General Preventive Start-time Optimisation (GPSO). GPSO determines the weight set that minimises the worst case congestion when maximal allowable penalty (upper bound penalty) under no failure is given. We showed that GPSO includes previous start-time optimisation when the upper bound penalty is appropriately set. If the penalty upper bound if fixed to zero, GPSO results matches PSO-NP's. If the penalty is neglected, GPSO will match PSO-P. Overall, GPSO provides an efficient and flexible weight set that considers both failure and non failure scenarios.

In fact, failure distribution is not the same for every link in the network. Most link failures may in fact focus on a small number of links. Therefore a scheme that considers link failure distribution in the network has to be considered. We address this issue in the second part of our work. We propose a Preventive Start-time Optimization that considers link failure probability (PSO-FP). When Traffic demand and link failure probability are known PSO-FP computes a link weight set that minimizes the expectation value of the congestion ratio for all possible failures. Simulation results show that PSO-FP is much more closer to both SO and PSO than they are to it. For both no failure and worst scenario PSO-FP gives a congestion close to SO and PSO respectively whereas SO and PSO gives an expected congestion ratio sometimes 10 times that of PSO-FP. Overall PSO-FP is that proposes more flexibility in the assignment of link weight when link failure may concentrate on a few number of links in the network.

Publications

Curriculum Vitae

Kaotchouang Stephane is a master's student at the University of Electro-Communications. He received his Bachelor in Mathematics from the Faculty of science of the University of Yaounde 1, Cameroon in 2007. Then he will receive his Master of Engineering majoring in Information and Communications Systems in March 2014. His research focuses on network optimization, network failure and routing.

Journal Publications

- S. Kaptchouang, Ihsen Aziz Ouédraogo, and E. Oki, "Preventive Start-time Optimisation Considering Both Failure and Non-Failure Scenarios," *IET Networks*, (Under review).
- S. Kaptchouang, Ihsen Aziz Ouédraogo, and E. Oki,"Preventive Start-Time Optimization Considering Link Failure Probability," *IEEE Trans. Commun.*, (To be submitted).
- S. Kaptchouang, Ihsen Aziz Ouédraogo, and E. Oki,"Preventive Start-time Optimization Enhanced," *IEICE Commun. Express.*,(To be submitted).

Conference Proceeding Publications

- S. Kaptchouang, and E. Oki, "Enhancing Preventive Start-Time Optimization Considering Both Failure and Non-Failure Scenarios," *19th Asia-Pacific Conference on Communications (APCC 2013)*, Aug. 2013.
- S. Kaptchouang, and E. Oki, "Preventive Start-time Optimisation Considering Both Failure and Non-Failure Scenarios," *IEICE Technical report (PN2013-11)*, Page:39-43, Nov. 2013.

- S. Kaptchouang, Ihsen Aziz Ouédraogo, and E. Oki,"PSO-FP: Preventive Start-Time Optimization Considering Link Failure Probability," 6^{th} *Workshop on Photonics Network.*, Mar. 2014.

APPENDIX A

SO source code

```
    //SO source code, C++//
#include<iostream>
#include<stdio.h>
#include<limits.h>
#include<time.h>
#include<set>
#include<queue>
#include<algorithm>
#include<math.h>
using namespace std;
typedef pair<int,int>P;
typedef pair<P, double>PPD;
#define MAX_V 20
#define INF 11000
#define BORN 1000
int V, Imax, E;
int w = 0;
double p[MAX_V][MAX_V];
double T[MAX_V][MAX_V];
class Graph{
private:
    int cost[MAX_V][MAX_V];
    double capacity[MAX_V][MAX_V];
    double Y[MAX_V][MAX_V];
```

```
    double x[MAX_V][MAX_V];
    double U[MAX_V][MAX_V][MAX_V][MAX_V];
    double traffic[MAX_V][MAX_V];
    double c_ratio[MAX_V][MAX_V];
    bool unconnected[MAX_V][MAX_V];
    int d[MAX_V];
    bool used[MAX_V];
    int counter[MAX_V][MAX_V];
    int n[MAX_V];
    double x_int[MAX_V];
    int V;
    int prev[MAX_V][MAX_V];
    int next[MAX_V][MAX_V];
public
    void init_cost() {
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                if(unconnected[i][j]) continue;
                cost[i][j] = INF;
                capacity[i][j]=capacity[j][i]=0;
            }
        }
    }
    void init_unconnected(){
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                x[i][j]=0.0;
                p[i][j]=0.0;
                unconnected[i][j] = false;
```

```
            }
        }
    }
    void breakLink(int u, int v) {
        setCost(u,v,INF);
        setCost(v,u,INF);
    }
    void showGraph() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(cost[i][j] >= INF)
                    printf("INF ");
                else
                    printf("%2d " ,cost[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    }
    void setRandomCost() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(unconnected[i][j]){
                    setCost(i,j, rand( )%BORN+1);
                }
            }
        }
    }
    double getAveRatioBreak(){
```

```
        double sum=0.0;
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                sum=sum+p[p_max_link1.first.first][p_max_link1.first.second]*p_max_link1.second
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return sum;
    }
    PPD getMaxCRatioBreak(){
        PPD p_max_link=PPD(P(0,0),-1);
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                if(p_max_link1.second¿p_max_link.second){
                    p_max_link = p_max_link1;
                }
```

```
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return p_max_link;
}
void setCost(int u, int v, int pcost) {
        cost[u][v] = pcost;
}
void setGraphSize(int v) {
        V = v;
}
int size() {
        return V;
}
bool isConnected(int u, int v) {
        return cost[u][v] < INF;
}
void setConnected(int u, int v, int cost1,int capa, double prop) {
        cost[u][v] = cost1;
        if(w==0){
            cost[v][u]=cost1;
}
        capacity[u][v]=capacity[v][u]=capa ;
        p[u][v]=p[v][u]=prop;
        unconnected[u][v]=unconnected[v][u]=true;
}
void get_ecmp(int q ){
        for(int j=0;j<V;j++){
```

```
            if(prev[q][j]==-1) continue;
            next[j][q]=q; counter[q][j]++;
            if(counter[q][j]<2) n[j]++;
            get_ecmp(j);
        }
    }
    void cal_ecmp(int p){
        for(int i=0;i<V;i++){
            if(next[p][i]==-1) continue;
            x[p][next[p][i]]=x[p][next[p][i]]+x_int[p]/n[p];
            x_int[next[p][i]]=x[p][next[p][i]];
            cal_ecmp(next[p][i]);
        }
    }
    void dijkstra(int s) {
        fill(d, d + V, INF/2);
        fill(n, n + V, 0 );
        fill(x_int, x_int + V, 1.0);
        fill(used, used + V, false);
        for(int i=0;i<V;i++){
            fill(prev[i],prev[i]+V,-1);
            fill(counter[i], counter[i] + V, 0);
            fill(next[i],next[i]+V,-1);
            fill(x[i], x[i] + V, 0.0);
        }
        d[s] = 0;
        while(true) {
            int v = -1;
            for(int u = 0; u < V; u++) {
```

```
                if (!used[u] && (v == -1 —— d[u] < d[v])) v = u;
            }
            if(v == -1) break;
            used[v] = true;
            for(int u=0;u<V;u++) {
                if(u == v) continue;
                if( d[u] ¿ d[v] + cost[v][u]) {
                    fill(prev[u],prev[u]+V,-1);
                    d[u] = d[v] + cost[v][u];
                    prev[u][v] = v;
                }
                else if(d[u] == d[v] + cost[v][u]){
                    prev[u][v] = v;
                }
            }
        }
}
int getCost(int u, int v) {
        return cost[u][v];
}
void usedSet(int p, int q) {
        dijkstra(p);
        get_ecmp(q);
        cal_ecmp(p);
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(x[i][j])
                    U[p][q][i][j]=x[i][j];
            }
```

```
    }
    for(int i=0;i<V;i++)
        fill(x[i], x[i] + MAX_V, 0.0);
}
void operation_ecmp(){
    for (int p=0;p<V;p++){
        for(int q=0;q<V;q++){
            if(p!=q) {
                usedSet(p,q);
            }
        }
    }
}
PPD getcongestion() {
    for(int i=0;i<V;i++){
        fill(Y[i], Y[i] + MAX_V, 0.0);
    }
    PPD p_max_link=PPD(P(0,0),-1);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(i==j) continue;
            for(int p=0;p<V;p++){
                for(int q=0;q<V;q++){
                    if(p==q) continue;
                    Y[i][j]=Y[i][j]+T[p][q]*U[p][q][i][j];
                }
            }
        }
    }
```

```
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(!unconnected[i][j]) continue;
                double r = Y[i][j]/capacity[i][j];
                if(p_max_link.second<r){
                    p_max_link.second = r;
                    p_max_link.first.first=i;
                    p_max_link.first.second=j;
                }
            }
        }
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                for(int p=0;p<V;p++){
                    for(int q=0;q<V;q++){
                        U[q][p][i][j]=0.0;
                    }
                }
            }
        }
        return p_max_link;
    }
};
Graph g;
Graph prev_g;
Graph so_g;
PPD prev_link =PPD(P(0,0),INT_MAX);
void solve(){
    PPD p_max_link=PPD(P(0,0),-1);
```

```
    int C=Imax;
    while(C){
        bool changed = false;
        while(true){
            g.operation_ecmp();
            p_max_link= g.getcongestion();
            if(p_max_link.second<prev_link.second){
                p_max_link.second=prev_link.second;
                so_g =g;
            }
            int pcost=g.getCost(p_max_link.first.first,p_max_link.first.second);
            if(pcost>BORN-1){
                break;
            }
            g.setCost(p_max_link.first.first,p_max_link.first.second,pcost+1);
        }
        g.setRandomCost();
        C–;
    }
    so_g.operation_ecmp();
    printf("SO congestion no failure case =%1.7lf\n",so_g.getcongestion().second);
    printf("SO worst case congestio=%1.7lf\n",so_g.getMaxCRatioBreak().second);
    printf("SO expected congestion=%1.7lf\n",so_g.getAveRatioBreak());
    printf("\n");
}
int main(){
    clock_t start;
    start =clock();
    double sum=0.0;
```

```
    srand(1);
    cin>>Imax;
    cin>>V>>E;
    g.setGraphSize(V);
    g.init_unconnected();
    g.init_cost();
    for(int i=0;i<E;i++) {
        int u,j,cost,capacity;
        double prop;
        cin>>u>>j>>cost>>capacity>>prop;
        sum+=prop;
        g.setConnected(u,j,cost,capacity,prop);
    }
    w++;
    for (int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            cin>>T[i][j];
            p[i][j]=p[i][j]/sum;
        }
    }
    solve();
    start=(clock()-start);
    printf("Run time %lf\n",start/(double)CLOCKS_PER_SEC);
    printf("Imax= %3d\n",Imax);
    return 0;
}
```

APPENDIX B

PSO-NP source code

```
    //PSO-NP source code, C++//
#include<iostream>
#include<stdio.h>
#include<limits.h>
#include<time.h>
#include<set>
#include<queue>
#include<algorithm>
#include<math.h>
using namespace std;
typedef pair<int,int>P;
typedef pair<P, double>PPD;
#define MAX_V 20
#define INF 11000
#define BORN 1000
int V, Imax, E;
int w = 0;
double p[MAX_V][MAX_V];
double T[MAX_V][MAX_V];
class Graph{
private:
    int cost[MAX_V][MAX_V];
    double capacity[MAX_V][MAX_V];
    double Y[MAX_V][MAX_V];
```

```
    double x[MAX_V][MAX_V];
    double U[MAX_V][MAX_V][MAX_V][MAX_V];
    double traffic[MAX_V][MAX_V];
    double c_ratio[MAX_V][MAX_V];
    bool unconnected[MAX_V][MAX_V];
    int d[MAX_V];
    bool used[MAX_V];
    int counter[MAX_V][MAX_V];
    int n[MAX_V];
    double x_int[MAX_V];
    int V;
    int prev[MAX_V][MAX_V];
    int next[MAX_V][MAX_V];
public
    void init_cost() {
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                if(unconnected[i][j]) continue;
                cost[i][j] = INF;
                capacity[i][j]=capacity[j][i]=0;
            }
        }
    }
    void init_unconnected(){
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                x[i][j]=0.0;
                p[i][j]=0.0;
                unconnected[i][j] = false;
```

```
            }
        }
    }
    void breakLink(int u, int v) {
        setCost(u,v,INF);
        setCost(v,u,INF);
    }
    void showGraph() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(cost[i][j] >= INF)
                    printf("INF ");
                else
                    printf("%2d " ,cost[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    }
    void setRandomCost() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(unconnected[i][j]){
                    setCost(i,j, rand( )%BORN+1);
                }
            }
        }
    }
    double getAveRatioBreak(){
```

```
        double sum=0.0;
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                sum=sum+p[p_max_link1.first.first][p_max_link1.first.second]*p_max_link1.second
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return sum;
    }
    PPD getMaxCRatioBreak(){
        PPD p_max_link=PPD(P(0,0),-1);
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                if(p_max_link1.second¿p_max_link.second){
                    p_max_link = p_max_link1;
                }
```

```
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return p_max_link;
}
void setCost(int u, int v, int pcost) {
        cost[u][v] = pcost;
}
void setGraphSize(int v) {
        V = v;
}
int size() {
        return V;
}
bool isConnected(int u, int v) {
        return cost[u][v] < INF;
}
void setConnected(int u, int v, int cost1,int capa, double prop) {
        cost[u][v] = cost1;
        if(w==0){
            cost[v][u]=cost1;
}
        capacity[u][v]=capacity[v][u]=capa ;
        p[u][v]=p[v][u]=prop;
        unconnected[u][v]=unconnected[v][u]=true;
}
void get_ecmp(int q ){
        for(int j=0;j<V;j++){
```

```
            if(prev[q][j]==-1) continue;
            next[j][q]=q; counter[q][j]++;
            if(counter[q][j]<2) n[j]++;
            get_ecmp(j);
        }
    }
    void cal_ecmp(int p){
        for(int i=0;i<V;i++){
            if(next[p][i]==-1) continue;
            x[p][next[p][i]]=x[p][next[p][i]]+x_int[p]/n[p];
            x_int[next[p][i]]=x[p][next[p][i]];
            cal_ecmp(next[p][i]);
        }
    }
    void dijkstra(int s) {
        fill(d, d + V, INF/2);
        fill(n, n + V, 0 );
        fill(x_int, x_int + V, 1.0);
        fill(used, used + V, false);
        for(int i=0;i<V;i++){
            fill(prev[i],prev[i]+V,-1);
            fill(counter[i], counter[i] + V, 0);
            fill(next[i],next[i]+V,-1);
            fill(x[i], x[i] + V, 0.0);
        }
        d[s] = 0;
        while(true) {
            int v = -1;
            for(int u = 0; u < V; u++) {
```

```
                if (!used[u] && (v == -1 —— d[u] < d[v])) v = u;
            }
            if(v == -1) break;
            used[v] = true;
            for(int u=0;u<V;u++) {
                if(u == v) continue;
                if( d[u] ¿ d[v] + cost[v][u]) {
                    fill(prev[u],prev[u]+V,-1);
                    d[u] = d[v] + cost[v][u];
                    prev[u][v] = v;
                }
                else if(d[u] == d[v] + cost[v][u]){
                    prev[u][v] = v;
                }
            }
        }
}
int getCost(int u, int v) {
    return cost[u][v];
}
void usedSet(int p, int q) {
    dijkstra(p);
    get_ecmp(q);
    cal_ecmp(p);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(x[i][j])
                U[p][q][i][j]=x[i][j];
        }
```

```
    }
    for(int i=0;i<V;i++)
        fill(x[i], x[i] + MAX_V, 0.0);
}
void operation_ecmp(){
    for (int p=0;p<V;p++){
        for(int q=0;q<V;q++){
            if(p!=q) {
                usedSet(p,q);
            }
        }
    }
}
PPD getcongestion() {
    for(int i=0;i<V;i++){
        fill(Y[i], Y[i] + MAX_V, 0.0);
    }
    PPD p_max_link=PPD(P(0,0),-1);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(i==j) continue;
            for(int p=0;p<V;p++){
                for(int q=0;q<V;q++){
                    if(p==q) continue;
                    Y[i][j]=Y[i][j]+T[p][q]*U[p][q][i][j];
                }
            }
        }
    }
```

```
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(!unconnected[i][j]) continue;
                double r = Y[i][j]/capacity[i][j];
                if(p_max_link.second<r){
                    p_max_link.second = r;
                    p_max_link.first.first=i;
                    p_max_link.first.second=j;
                }
            }
        }
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                for(int p=0;p<V;p++){
                    for(int q=0;q<V;q++){
                        U[q][p][i][j]=0.0;
                    }
                }
            }
        }
        return p_max_link;
    }
};
Graph g;
Graph prev_g;
Graph so_g;
PPD prev_link =PPD(P(0,0),INT_MAX);
void solve(){
    PPD p_max_link=PPD(P(0,0),-1);
```

```
int C=Imax;
while(C){
    bool changed = false;
    while(true){
        g.operation_ecmp();
        p_max_link= g.getcongestion();
        if(p_max_link.second=prev_link.second){
            printf(" non failure c =%1.7lf\ worst congestion=%1.7lf\n",prev_link.second,
g.getMaxCRatioBreak().second);
        }
        if(p_max_link.second<prev_link.second){
            p_max_link.second=prev_link.second;
            so_g =g;
        }
        int pcost=g.getCost(p_max_link.first.first,p_max_link.first.second);
        if(pcost>BORN-1){
            break;
        }
        g.setCost(p_max_link.first.first,p_max_link.first.second,pcost+1);
    }
    g.setRandomCost();
    C–;
}
so_g.operation_ecmp();
printf("SO congestion no failure case =%1.7lf\n",so_g.getcongestion().second);
printf("SO worst case congestio=%1.7lf\n",so_g.getMaxCRatioBreak().second);
printf("SO expected congestion=%1.7lf\n",so_g.getAveRatioBreak());
```

```
    printf("\n");
}
int main(){
    clock_t start;
    start =clock();
    double sum=0.0;
    srand(1);
    cin>>Imax;
    cin>>V>>E;
    g.setGraphSize(V);
    g.init_unconnected();
    g.init_cost();
    for(int i=0;i<E;i++) {
        int u,j,cost,capacity;
        double prop;
        cin>>u>>j>>cost>>capacity>>prop;
        sum+=prop;
        g.setConnected(u,j,cost,capacity,prop);
    }
    w++;
    for (int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            cin>>T[i][j];
            p[i][j]=p[i][j]/sum;
        }
    }
    solve();
    start=(clock()-start);
    printf("Run time %lf\n",start/(double)CLOCKS_PER_SEC);
```

```
    printf("Imax= %3d\n",Imax);
    return 0;
}
```

APPENDIX C

PSO-P source code

```
//PSO-P Source code// ♯include<iostream>
♯include<stdio.h>
♯include<limits.h>
♯include<time.h>
♯include<set>
♯include<queue>
♯include<algorithm>
♯include<math.h>
using namespace std;
typedef pair<int,int>P;
typedef pair<P, double>PPD;
♯define MAX_V 20
♯define INF 11000
♯define BORN 1000
int V, Imax, E;
int w = 0;
double p[MAX_V][MAX_V];
double T[MAX_V][MAX_V];
class Graph{
private:
    int cost[MAX_V][MAX_V];
    int capacity[MAX_V][MAX_V];
    double Y[MAX_V][MAX_V];
    double x[MAX_V][MAX_V];
```

```
    double U[MAX_V][MAX_V][MAX_V][MAX_V];
    double traffic[MAX_V][MAX_V];
    double c_ratio[MAX_V][MAX_V];
    bool unconnected[MAX_V][MAX_V];
    int d[MAX_V];
    bool used[MAX_V];
    int n[MAX_V];
    int counter[MAX_V][MAX_V];
    int next[MAX_V][MAX_V];
    double x_int[MAX_V];
    int V;
    int prev[MAX_V][MAX_V];
public: void init_cost() {
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                if(unconnected[i][j]) continue;
                cost[i][j] = INF;
                capacity[i][j]=capacity[j][i]=0;
            }
        }
    }
    void init_unconnected(){
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                x[i][j]=0.0;
                p[i][j]=0.0;
                unconnected[i][j] = false;
            }
        }
```

```
}
void breakLink(int u, int v) {
    setCost(u,v,INF);
    setCost(v,u,INF);
} void showGraph() {
    for(int i=0;i<V;i++) {
        for(int j=0;j<V;j++) {
            if(cost[i][j] >= INF)
                printf("INF ");
            else
                printf("%2d " ,cost[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
void setRandomCost() {
    for(int i=0;i<V;i++) {
        for(int j=0;j<V;j++) {
            if(unconnected[i][j]){
                setCost(i,j, rand( )%BORN+1);
            }
        }
    }
}
double getAveRatioBreak(){
    double sum=0.0;
    for(int u=0;u<size();u++){
        for(int v=u+1;v<size();v++){
```

```
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                sum=sum+p[p_max_link1.first.first][p_max_link1.first.second]*p_max_link1.second
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return sum;
    }
    PPD getMaxCRatioBreak(){
        PPD p_max_link=PPD(P(0,0),-1);
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                if(p_max_link1.second¿p_max_link.second){
                    p_max_link = p_max_link1;
                }
                setCost(u,v,x);
                setCost(v,u,y);
            }
```

```
        }
        return p_max_link;
    }
    void setCost(int u, int v, int pcost) {
        cost[u][v] = pcost;
    }
    void setGraphSize(int v) {
        V = v;
    }
    int size() {
        return V;
    }
    bool isConnected(int u, int v) {
        return cost[u][v] < INF;
    }
    void setConnected(int u, int v, int cost1,int capa, double prop) {
        cost[u][v] = cost1;
        if(w==0){
            cost[v][u]=cost1;
    }
        capacity[u][v]=capacity[v][u]=capa ;
        p[u][v]=p[v][u]=prop;
        unconnected[u][v]=unconnected[v][u]=true;
    }
    void get_ecmp(int q ){
        for(int j=0;j<V;j++){
            if(prev[q][j]==-1) continue;
            next[j][q]=q; counter[q][j]++;
            if(counter[q][j]<2) n[j]++;
```

```
            get_ecmp(j);
        }
    }
    void cal_ecmp(int p){
        for(int i=0;i<V;i++){
            if(next[p][i]==-1) continue;
            x[p][next[p][i]]=x[p][next[p][i]]+x_int[p]/n[p];
            x_int[next[p][i]]=x[p][next[p][i]];
            cal_ecmp(next[p][i]);
        }
    }
    void dijkstra(int s) {
        fill(d, d + V, INF/2);
        fill(n, n + V, 0 );
        fill(x_int, x_int + V, 1.0);
        fill(used, used + V, false);
        for(int i=0;i<V;i++){
            fill(prev[i],prev[i]+V,-1);
            fill(counter[i], counter[i] + V, 0);
            fill(next[i],next[i]+V,-1);
            fill(x[i], x[i] + V, 0.0);
        }
        d[s] = 0;
        while(true) {
            int v = -1;
            for(int u = 0; u < V; u++) {
                if (!used[u] && (v == -1 —— d[u] < d[v])) v = u;
            }
            if(v == -1) break;
```

```
            used[v] = true;
            for(int u=0;u<V;u++) {
                if(u == v) continue;
                if( d[u] ¿ d[v] + cost[v][u]) {
                    fill(prev[u],prev[u]+V,-1);
                    d[u] = d[v] + cost[v][u];
                    prev[u][v] = v;
                }
                else if(d[u] == d[v] + cost[v][u]){
                    prev[u][v] = v;
                }
            }
        }
}
int getCost(int u, int v) {
        return cost[u][v];
}
void usedSet(int p, int q) {
        dijkstra(p);
        get_ecmp(q);
        cal_ecmp(p);
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(x[i][j])
                    U[p][q][i][j]=x[i][j];
            }
        }
        for(int i=0;i<V;i++)
            fill(x[i], x[i] + MAX_V, 0.0);
```

```
}
void operation_ecmp(){
    for (int p=0;p<V;p++){
        for(int q=0;q<V;q++){
            if(p!=q) {
                usedSet(p,q);
            }
        }
    }
}
PPD getcongestion() {
    for(int i=0;i<V;i++){
        fill(Y[i], Y[i] + MAX_V, 0.0);
    }
    PPD p_max_link=PPD(P(0,0),-1);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(i==j) continue;
            for(int p=0;p<V;p++){
                for(int q=0;q<V;q++){
                    if(p==q) continue;
                    Y[i][j]=Y[i][j]+T[p][q]*U[p][q][i][j];
                }
            }
        }
    }
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(!unconnected[i][j]) continue;
```

```
                double r = Y[i][j]/capacity[i][j];
                if(p_max_link.second<r){
                    p_max_link.second = r;
                    p_max_link.first.first=i;
                    p_max_link.first.second=j;
                }
            }
        }
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                for(int p=0;p<V;p++){
                    for(int q=0;q<V;q++){
                        U[q][p][i][j]=0.0;
                    }
                }
            }
        }
        return p_max_link;
    }
};
Graph g;
Graph prev_g;
Graph pso_g;
PPD prev_link =PPD(P(0,0),INT_MAX);
void solve(){

    PPD p_max_link=PPD(P(0,0),-1);
    int C=Imax;
    while(C){
```

```
        while(true){
            p_max_link= g.getMaxCRatioBreak();
            if(p_max_link<prev_link){
            prev_link.second=p_max_link.second;
            pso_g=g;
            int pcost=g.getCost(p_max_link.first.first,p_max_link.first.second);
            if(pcost>BORN-1){
                break;
            }
            g.setCost(p_max_link.first.first,p_max_link.first.second,pcost+1);
        }
        g.setRandomCost();
        C–;
    }
    printf("PSO-P worst congestion ratio = %1.7lf\n",pso_g.getMaxCRatioBreak().second);
    pso_g.operation_ecmp();
    printf("PSO-P congestion under no failure= %1.7lf\n",pso_g.getcongestion().second);
    printf("PSO-P expected congestion ratio = %1.7lf\n",pso_g.getAveRatioBreak());
    printf("\n");
}
int main(){
    clock_t start;
    start =clock();
    double sum=0.0;
    srand(1);
    cin>>Imax;
    cin>>V>>E;
    g.setGraphSize(V);
    g.init_unconnected();
```

```
g.init_cost();
for(int i=0;i<E;i++) {
    int u,j,cost,capacity;
    double prop;
    cin>>u>>j>>cost>>capacity>>prop;
    sum+=prop;
    g.setConnected(u,j,cost,capacity,prop);
}
w++;
for (int i=0;i<V;i++){
    for(int j=0;j<V;j++){
        cin>>T[i][j];
        p[i][j]=p[i][j]/sum;
    }
}
solve();
start=(clock()-start);
printf("Run time %lf\n",start/(double)CLOCKS_PER_SEC);
printf("Imax= %3d\n",Imax);
```

APPENDIX D

PSO-FP source code

```
//PSO-FP source code// ♯include<iostream>
♯include<stdio.h>
♯include<limits.h>
♯include<time.h>
♯include<set>
♯include<queue>
♯include<algorithm>
♯include<math.h>
using namespace std;
typedef pair<int,int>P;
typedef pair<P, double>PPD;
♯define MAX_V 20
♯define INF 11000
♯define BORN 1000
int V, Imax, E;
int w = 0;
double p[MAX_V][MAX_V];
double T[MAX_V][MAX_V];
class Graph{
private:
    int cost[MAX_V][MAX_V];
    double capacity[MAX_V][MAX_V];
    double Y[MAX_V][MAX_V];
    double x[MAX_V][MAX_V];
```

```
    double U[MAX_V][MAX_V][MAX_V][MAX_V];
    double traffic[MAX_V][MAX_V];
    double c_ratio[MAX_V][MAX_V];
    bool unconnected[MAX_V][MAX_V];
    int d[MAX_V];
    bool used[MAX_V];
    int counter[MAX_V][MAX_V];
    int n[MAX_V];
    double x_int[MAX_V];
    int V;
    int prev[MAX_V][MAX_V];
    int next[MAX_V][MAX_V];
public
    void init_cost() {
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                if(unconnected[i][j]) continue;
                cost[i][j] = INF;
                capacity[i][j]=capacity[j][i]=0;
            }
        }
    }
    void init_unconnected(){
        for(int i=0;i<MAX_V;i++) {
            for(int j=0;j<MAX_V;j++) {
                x[i][j]=0.0;
                p[i][j]=0.0;
                unconnected[i][j] = false;
            }
```

```
        }
    }
    void breakLink(int u, int v) {
        setCost(u,v,INF);
        setCost(v,u,INF);
    }
    void showGraph() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(cost[i][j] >= INF)
                    printf("INF ");
                else
                    printf("%2d " ,cost[i][j]);
            }
            printf("\n");
        }
        printf("\n");
    }
    void setRandomCost() {
        for(int i=0;i<V;i++) {
            for(int j=0;j<V;j++) {
                if(unconnected[i][j]){
                    setCost(i,j, rand( )%BORN+1);
                }
            }
        }
    }
    PPD getWorstRatio(){
        PPD p_max_link=PPD(P(0,0),-1);
```

```
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                if(p_max_link1.second¿p_max_link.second){
                    p_max_link = p_max_link1;
                }
                setCost(u,v,x);
                setCost(v,u,y);
            }
        }
        return p_max_link;
    }
    double getMaxCRatioBreak(){
        double sum=0.0;
        for(int u=0;u<size();u++){
            for(int v=u+1;v<size();v++){
                int x=getCost(u,v);
                int y=getCost(v,u);
                if(x == INF) continue;
                breakLink(u,v);
                operation_ecmp();
                PPD p_max_link1 = getcongestion();
                sum=sum+p[p_max_link1.first.first][p_max_link1.first.second]*p_max_link1.second
                setCost(u,v,x);
```

```
                setCost(v,u,y);
            }
        }
        return sum;
    }
    void setCost(int u, int v, int pcost) {
        cost[u][v] = pcost;
    }
    void setGraphSize(int v) {
        V = v;
    }
    int size() {
        return V;
    }
    bool isConnected(int u, int v) {
        return cost[u][v] < INF;
    }
    void setConnected(int u, int v, int cost1,int capa, double prop) {
        cost[u][v] = cost1;
        if(w==0){
            cost[v][u]=cost1;
    }
        capacity[u][v]=capacity[v][u]=capa ;
        p[u][v]=p[v][u]=prop;
        unconnected[u][v]=unconnected[v][u]=true;
    }
    void get_ecmp(int q ){
        for(int j=0;j<V;j++){
            if(prev[q][j]==-1) continue;
```

```
            next[j][q]=q; counter[q][j]++;
            if(counter[q][j]<2) n[j]++;
            get_ecmp(j);
        }
    }
    void cal_ecmp(int p){
        for(int i=0;i<V;i++){
            if(next[p][i]==-1) continue;
            x[p][next[p][i]]=x[p][next[p][i]]+x_int[p]/n[p];
            x_int[next[p][i]]=x[p][next[p][i]];
            cal_ecmp(next[p][i]);
        }
    }
    void dijkstra(int s) {
        fill(d, d + V, INF/2);
        fill(n, n + V, 0 );
        fill(x_int, x_int + V, 1.0);
        fill(used, used + V, false);
        for(int i=0;i<V;i++){
            fill(prev[i],prev[i]+V,-1);
            fill(counter[i], counter[i] + V, 0);
            fill(next[i],next[i]+V,-1);
            fill(x[i], x[i] + V, 0.0);
        }
        d[s] = 0;
        while(true) {
            int v = -1;
            for(int u = 0; u < V; u++) {
                if (!used[u] && (v == -1 —— d[u] < d[v])) v = u;
```

```
            }
            if(v == -1) break;
            used[v] = true;
            for(int u=0;u<V;u++) {
                if(u == v) continue;
                if( d[u] ¿ d[v] + cost[v][u]) {
                    fill(prev[u],prev[u]+V,-1);
                    d[u] = d[v] + cost[v][u];
                    prev[u][v] = v;
                }
                else if(d[u] == d[v] + cost[v][u]){
                    prev[u][v] = v;
                }
            }
        }
    }
    int getCost(int u, int v) {
        return cost[u][v];
    }
    void usedSet(int p, int q) {
        dijkstra(p);
        get_ecmp(q);
        cal_ecmp(p);
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                if(x[i][j])
                    U[p][q][i][j]=x[i][j];
            }
        }
```

```
    for(int i=0;i<V;i++)
        fill(x[i], x[i] + MAX_V, 0.0);
}
void operation_ecmp(){
    for (int p=0;p<V;p++){
        for(int q=0;q<V;q++){
            if(p!=q) {
                usedSet(p,q);
            }
        }
    }
}
PPD getcongestion() {
    for(int i=0;i<V;i++){
        fill(Y[i], Y[i] + MAX_V, 0.0);
    }
    PPD p_max_link=PPD(P(0,0),-1);
    for(int i=0;i<V;i++){
        for(int j=0;j<V;j++){
            if(i==j) continue;
            for(int p=0;p<V;p++){
                for(int q=0;q<V;q++){
                    if(p==q) continue;
                    Y[i][j]=Y[i][j]+T[p][q]*U[p][q][i][j];
                }
            }
        }
    }
    for(int i=0;i<V;i++){
```

```
            for(int j=0;j<V;j++){
                if(!unconnected[i][j]) continue;
                double r = Y[i][j]/capacity[i][j];
                if(p_max_link.second<r){
                    p_max_link.second = r;
                    p_max_link.first.first=i;
                    p_max_link.first.second=j;
                }
            }
        }
        for(int i=0;i<V;i++){
            for(int j=0;j<V;j++){
                for(int p=0;p<V;p++){
                    for(int q=0;q<V;q++){
                        U[q][p][i][j]=0.0;
                    }
                }
            }
        }
        return p_max_link;
    }
};
Graph g;
Graph prev_g;
Graph so_g;
void solve(){
    int C=0;
    int j=0;
    double sum = 0.0;
```

```
while(C<Imax){
    if(j==0){
        sum = g.getMaxCRatioBreak();
        j++;
        prev_g=g;
    }
    step = E*V;
    while(step){
        bool flag=false;
        int u,v;
        int a=0;
        do{
            u = rand() %V + 1;
            v = rand() %V + 1;
        }while(g.getCost(u,v)==INF);
        int pcost1 = g.getCost(u,v);
        int pcost = pcost1;
        if(pcost>=2){
            g.setCost(u,v,pcost-1);
            double t=g.getMaxCRatioBreak();
            if(t<sum){
                sum=t;
                a= g.getCost(u,v);
                flag =true;
                prev_g = g;
            }
        }
        if(pcost<=BORN-1){
            g.setCost(u,v,pcost+1);
```

```
                double t=g.getMaxCRatioBreak();
                if(t<sum){
                    sum=t;
                    a= g.getCost(u,v);
                    flag =true;
                    prev_g = g;
                }
            }
            if(flag){
                g.setCost(u,v,a);
                step = E*V;
            }
            g.setCost(u,v,pcost1);
            step–;
        }
        g.setRandomCost();
        C++;
    }
    prev_g.operation_ecmp();
    printf("PSO-FP congestion under no failure =%1.7lf\n",prev_g.getcongestion().second);
    printf("PSO-FP worst congestion ratio =%1.7lf\n",prev_g.getWorstRatio().second);
    printf("PSO-FP expected congestion ratio =%1.7lf\n",sum);
    printf("\n");
}
int main(){
    clock_t start;
    start =clock();
    double sum=0.0;
    srand(1);
```

```
cin>>Imax;
cin>>V>>E;
g.setGraphSize(V);
g.init_unconnected();
g.init_cost();
for(int i=0;i<E;i++) {
    int u,j,cost,capacity;
    double prop;
    cin>>u>>j>>cost>>capacity>>prop;
    sum+=prop;
    g.setConnected(u,j,cost,capacity,prop);
}
w++;
for (int i=0;i<V;i++){
    for(int j=0;j<V;j++){
        cin>>T[i][j];
        p[i][j]=p[i][j]/sum;
    }
}
solve();
start=(clock()-start);
printf("Run time %lf\n",start/(double)CLOCKS_PER_SEC);
printf("Imax= %3d\n",Imax);
```

Bibliography

[1] Kaptchouang, S., and Oki, E.: 'Enhancing Preventive Start-Time Optimization Considering Both Failure and Non-Failure Scenarios', 19th Asia-Pacific Conference on Communications (APCC 2013), Aug. 2013.

[2] Moy, J.: 'OSPF version 2', IETF RFC 1247, July 1991.

[3] Fortz, B., and Thorup, M.: 'optimizing OSPF/IS-IS weights in a changing world', IEEE Journal on Selected Areas in Communications, May 2002, 20, (4), pp: 756-767.

[4] Fortz, B., Rexford, J., and Thorup, M.: 'Traffic engineering with traditional IP protocols', IEEE Commun. Mag., Dec. 2002, 40, (10), pp: 118-124.

[5] Reichert, C., and Magedanz, T.: 'A fast heuristic for genetic algorithms in link weight Optimization', Lecture Notes in Computer Science, Oct. 2004, 3266, pp: 144-153.

[6] Buriol, L.S., Resende, M.G.C., Ribeiro, C.C., and Thorup, M.: 'A hybrid genetic algorithm for the weight setting problem in OSPF-IS-IS routing', Networks, Aug. 2005, 46, (1), pp: 36-56.

[7] Wang, Y., and Ito, M.R.: 'Dynamics of load sensitive adaptive routing', IEEE International Conference on Communications (ICC), May 2005.

[8] 'Reducing Link Failure and Topology Change Notification times in IS-IS Networks', online, http://www.cisco.com/en/US/docs/ios-xml/ios/iproute_isis/configuration/15-mt/irs-fscnt.html/.

[9] Brewer, E.: 'Lessons from giant-scale services', IEEE Internet computing, July 2001, 5, (4), pp: 46-55.

[10] Iannaccone, G., Chuah, C., Mortier, R., Bhattacharyya, S., and Diot, C.: 'Analysis of link failures in a large IP backbone', Proc. 2nd ACM SIGCOM Internet Measurement Workshop, Nov. 2002.

[11] Markopolou, A., Iannaccone, G., Bhattacharyya, S., and Chuah, C.: 'Characterization of link Failures in IP Backbone', IEEE INFOCOM, Mar. 2004.

[12] Kamrul, I.M., and Oki, E.: 'Optimization of OSPF Link Weights to Counter Network Failure', IEICE Trans. on Commun., July 2011, E94B, (7), pp: 1964-1972.

[13] Kamrul, I.M., and Oki, E.: 'Optimization of OSPF link weight to minimise worst-case network congestion against single-link failure', IEEE International Conference on Communications, June 2011, pp: 1-5.

[14] Kamrul, I.M., and Oki, E.: 'PSO: Preventive Start-time Optimization of OSPF Link Weights to Counter Network Failure', IEEE Commun. Letters, June 2010, 14, (6), pp: 581-583.

[15] Ranaweera, R.S., Kamrul, I.M., and Oki, E.: 'Preventive Start-Time Optimization of OSPF Link Weights against Link Failure for Hose Model', 18th Asia-Pacific Conference on Communications (APCC 2012), Oct. 2012.

[16] Ranaweera, R.S., Kamrul, I.M., and Oki, E.: 'Preventive Start-Time Optimization of OSPF Link Weights for Hose Model', IET Networks, available online, sep. 2013.

[17] Thaler, D., and Hopps, C.: 'Multipath Issues in Unicast and Multicast Next-Hop selection', IETF RFC 2991, Nov. 2000.

[18] Oki, E., and Iwaki, A.: 'Load-Balanced IP Routing Scheme Based on Shortest Paths in Hose Model', IEEE Trans. Commun., July 2010, 58, (7), pp: 2088-2096.

[19] Chu, J., and Lea, C.: 'Optimal link weights for IP-based networks supporting hose-model VPNs', IEEE/ACM Trans. on Networking, June 2009, 17, (3), pp: 778-788.

[20] Chu, J., and Lea, C.: 'Optimal link weights for maximizing QoS traffic', IEEE International Conference on Communications, June 2007, pp: 610-615.

[21] 'The Internet2 Network', online, http://www.internet2.edu/network/.

[22] Medina, A., Lakhina, A., Matta, I., and Byers, J.: BRITE: Boston University Representative Internet Topology Generator, Boston Univ., Boston, MA, Apr. 2001 [Online]. Available: http://www.cs.bu.edu/brite