2007

Survivability issues in WDM optical networks

Chang Liu
Iowa State University

Survivability issues in WDM optical networks

by

Chang Liu

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Iowa State University

Ames, Iowa

2007

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

WDM optical networks make it possible for the bandwidth of transport networks to reach a level on which any failures would cause tremendous data loss and affect a lot of users. Thus, survivability issues of WDM optical networks have attracted a lot of research work. Within the scope of this dissertation, two categories of problems are studied, one is survivable mapping from IP topology to WDM topology, the other is *p*-cycle protection schemes in WDM networks.

Survivable mapping problem can be described as routing IP links on the WDM topology such that the IP topology stays connected under any single link failure in the WDM topology. This problem has been proved to be NP-complete [1]. At first, this dissertation provides a heuristic algorithm to compute approximated solutions for input IP/WDM topologies as an approach to ease the hardness of it. Then, it examines the problem with a different view, to augment the IP topology so that a survivable mapping can be easily computed. This new perspective leads to an extended survivable mapping problem that is originally proposed and analyzed in this dissertation. In addition, this dissertation also presents some interesting open problems for the survivable mapping problem as future work.

Various protection schemes in WDM networks have been explored. This dissertation focuses on methods based on the *p*-cycle technology. *p*-Cycle protection inherits the merit of fast restoration from the link-based protection technology while yielding higher efficiency on spare capacity usage [2]. In this dissertation, we first propose an efficient heuristic algorithm that generates a small subset of candidate cycles that guarantee 100% restorability and help to achieve an efficient design. Then, we adapt *p*-cycle design to accommodate the protection of the failure of a shared risk link group (SRLG). At last, we discuss the problem of establishing survivable connections for dynamic traffic demands using flow *p*-cycle.

CHAPTER 1. INTRODUCTION

Wavelength-division multiplexing (WDM) is a technology that multiplexes multiple optical carrier signals on a single optical fibre by using different wavelengths (colours) of laser light to carry different signals. With the tremendous bandwidth that is enabled by this technology, WDM optical networks are becoming the backbone transport networks for the next generation Internet. Because of the nature of large bandwidth traffic transported by WDM optical networks, any failures such as a fibre cut would cause enormous data loss. Therefore, it is important to design WDM networks that are able to survive failures. Extensive research work has been engaged in this area. This dissertation will focus on two survivability issues in WDM networks which are survivable mapping problem and *p*-cycle protection.

1.1 Survivable Mapping Problem in IP-over-WDM Networks

The Internet Protocol (IP) is the foundation of packet switched internetwork. There is a growing consensus that the next generation Internet will employ an IP-over-WDM architecture [3], where IP routers are attached to a WDM optical network consisting of optical cross-connects (OXCs) interconnected by multi-wavelength optical fibers.

A network is survivable if it can automatically reroute the traffic around failure. Survivability can be achieved by either protection or restoration. In protection, one or more backup paths are computed and corresponding spare capacities are reserved in advance for every primary path (i.e., the path that carries traffic in absence of a failure). In restoration, one or more backup paths with sufficient spare capacities are searched after the primary path fails. Protection has fast recovery time and guarantees the success of recovery while restoration is more efficient in capacity utilization. Lightpath protection schemes [4][5][6][7][8][9] and lightpath restoration schemes [10][11][12] have been studied extensively for WDM optical networks. Although WDM layer survivability mechanisms can achieve fast and efficient recovery of lightpaths, they can not provide differentiated resilience to individual IP flows since WDM layer operates on lightpath granularity with each lightpath carrying aggregated IP traffic flows. In addition, IP router failures cannot be handled by the WDM layer. Thus, IP layer recovery is needed. Multi-Protocol Label Switching (MPLS) [13] is a promising technology to provide both QoS and resilience in IP networks [14]. Several MPLS-based protection and restoration schemes have been studied recently [15][16][17][18][19]. IP layer failure recovery is possible only if a failure does not disconnect the IP topology. This leads to the following *survivable mapping* problem: given a IP topology and a WDM topology, map the IP topology onto the WDM topology such that the IP topology remains connected in case of any single WDM link failure. This dissertation explores this problem from different perspectives and extends it to a new problem with interesting graph theoretical properties.

1.2 p-Cycle Protection in WDM Networks

As discussed in the previous section, WDM layer survivability is necessary in order to minimize the interruption duration resulted from failures happened in the WDM optical networks. As a promising protection scheme, p-cycle technology was proposed in [2]. It achieves both fast restoration and spare capacity efficiency, which is an intriguing feature that makes it an appealing research area in recent years. This dissertation addresses one of its fundamental problems and presents new p-cycle protection schemes.

1.3 Dissertation Structure and Contribution

For the survivable mapping problem in IP-over-WDM networks, Chapter 2 provides a general overview of the problem and introduces notations that will be used later in Chapter 3 through Chapter 5. In Chapter 3, a heuristic algorithm solving the survivable mapping problem is presented while Chapter 4 investigates a different approach to this problem by augmenting the logical topology. As a follow-up of this idea, Chapter 5 analyzes a new version of survivable mapping problem that allows adding new logical links.

For the *p*-cycle protection schemes on WDM networks, Chapter 6 introduces the concept of *p*-cycle protection and terminologies that will be used in future chapters. Chapter 7 gives a heuristic algorithm that efficiently generates candidate cycles for *p*-cycle design. Chapter 8 extends the original *p*-cycle design against single-link-failure assumption to supporting shared risk link group (SRLG) failure model which is more general and realistic in real world. Moreover, Chapter 9 proposes a strategy that deploys path-segment *p*-cycles for protection of dynamic traffic demands.

Finally, Chapter 10 concludes the dissertation with a summary and future work consideration.

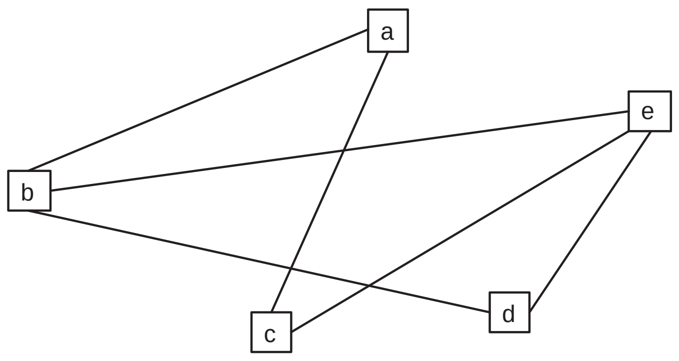
CHAPTER 2. OVERVIEW OF SURVIVABLE MAPPING PROBLEM

2.1 Introduction

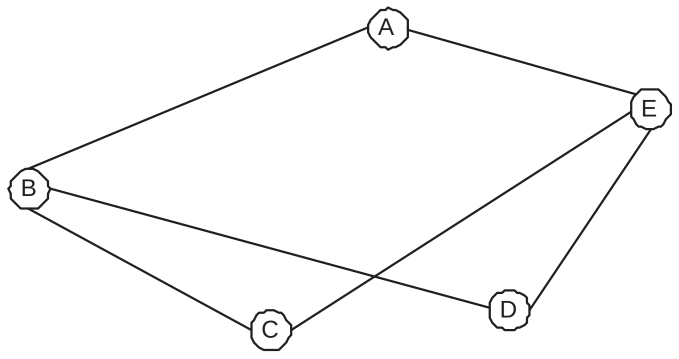
In IP-over-WDM architecture, IP routers are attached to the optical cross-connects (OXCs) in a WDM optical network and IP links are realized by lightpaths in the optical network. We refer to IP network topology, IP routers, and IP links as *logical topology*, *logical nodes*, and *logical links*, respectively. We refer to optical network topology, OXCs, and optical fibers as *physical topology*, *physical nodes*, and *physical links*, respectively. A logical topology is mapped on a physical topology by mapping each logical link to a path in the physical topology. (This reflects the fact that each IP link is realized by a lightpath in the optical network.) There are different ways to map a logical topology on a physical topology. For example, consider the logical and physical topologies shown in Figure 2.1. One mapping could map the logical link (a, c) to the physical path $< A - B - C >$, while another mapping could map (a, c) to the physical path $< A - E - C >$.

Consider the logical topology and the physical topology given in Figure 2.1. One possible mapping is to map (a, b) to $< A - B >$, (a, c) to $< A - B - C >$, (b, d) to $< B - D >$, (b, e) to $< B - A - E >$, (c, e) to $< C - E >$, and (d, e) to $< D - E >$. Notice that the failure of physical link (A, B) will cause logical links (a, b), (a, c), and (b, e) to fail simultaneously, leaving the logical topology disconnected. A possible solution to this problem is to map (a, b) to $A - E - D - B$ instead. Therefore, it is desirable to solve the following *survivable mapping problem*: given a logical topology and a physical topology, map the logical topology onto the physical topology such that the logical topology remains connected in case of any single physical link failure.

The survivable mapping problem has been studied in [1][20][21][22][23][24][25][26]. In [1],



(a). Logical Topology



(b). Physical Topology

Figure 2.1 A logical topology and a physical topology.

it is proved that determining whether a survivable mapping exists for a logical topology on a physical topology is NP-Complete. [20] gives a necessary and sufficient condition for a mapping to be survivable. Based on the condition, an integer linear program (ILP) formulation is given to solve the survivable mapping problem with the objective of minimizing the cost of the mapping. In [21], a mixed integer linear program (MILP) formulation for the survivable mapping problem is given. The MILP has faster computation speed than the ILP given in [20] since the number of constraints in the MILP grows as a polynomial with the size of the network while the number of constraints in the ILP grows exponentially with the size of the network. Various heuristic algorithms for the survivable mapping problem are proposed in

[22][23][24][25]. In [26], a formal analysis of a previously proposed heuristic algorithm named SMART [25] is given and the analysis shows that SMART can be used to derive practical methods for determining the existence or absence of a survivable mapping for large networks.

2.2 Terminology and Notation

Because fiber cut is the predominant form of failures in optical networks [27], we only consider single physical link failure here. Moreover, we also assume that sufficient capacity (wavelengths) is available on each physical link so that capacity constraints will not be considered.

Logical and physical topologies are represented by undirected graphs $G_l = (V_l, E_l)$ and $G_p = (V_p, E_p)$, respectively. Generally, $V_l \subseteq V_p$; however, we make the simplifying assumption that $V_l = V_p = V$ in this dissertation. Although logical and physical topologies are represented by undirected graphs, sometimes it is useful to treat an undirected edge $ij \in E_p$ as two directed edges (ij and ji) in opposite directions. We denote the set of directed edges obtained by replacing each undirected edge in E_p by two directed edges of opposite directions as E_p^d, where 'd' stands for 'directed'.

For a graph $G = (V, E)$ and $S \subset V$ $(S \neq \emptyset)$, the *edge cut of G defined by S*, denoted by $EC_G(S)$, is the set of edges in G with one end node in S and the other end node in $V - S$. Clearly, removing the edges in $EC_G(S)$ from G will disconnect G.

For $s, t \in V$ $(s \neq t)$, a path from s to t in G_p is denoted as $p_{st} = (s - \cdots - t)$. P_{st} denotes the set of all paths from s to t in G_p. A mapping from G_l to G_p is a function $M : E_l \longrightarrow \bigcup_{s,t \in V(s \neq t)} P_{st}$. That is, M maps each edge $st \in E_l$ to a path from s to t in G_p. The cost of mapping M, denoted by $cost(M)$, is the total wavelength channels used to map all the logical links in G_l. It can be computed as $cost(M) = \sum_{st \in E_l} |M(st)|$, where $|M(st)|$ is the hop count of $M(st)$.

$st \in E_l$ is a *reflective logical link* if there is a physical link between s and t in G_p. In other words, st is a reflective logical link if $st \in E_l \cap E_p$. A reflective logical link st is *reflectively-routed* under mapping M if $M(st) = (s - t)$, i.e., M maps st to the single-hop path between s and t

in G_p. M is a *reflectively-routed mapping* if all reflective logical links are reflectively-routed.

The *load set* of a physical link $ij \in E_p$ under mapping M, denoted as $L_M(ij)$, is the set of all logical links whose physical path traverses ij, i.e., $L_M(ij) = \{st \in E_l | M(st) \text{ traverses } ij\}$. The *remaining logical topology* upon the failure of $ij \in E_p$ under M, denoted as $G_l^M(ij)$, is the new logical topology obtained by removing all logical links in G_l whose physical path traverses ij. Thus, $G_l^M(ij) = (V, E_l - L_M(ij))$. A physical link $ij \in E_p$ is called a *critical link* under M if $G_l^M(ij)$ is not connected. If this is the case, a logical link $st \in L_M(ij)$ is labeled as a *bridge link of* ij under M if s and t belong to two difference connected components of $G_l^M(ij)$. Moreover, the set of all bridge links of ij under M are denoted as $B_M(ij)$.

M is a *survivable mapping* from G_l to G_p if $G_l^M(ij)$ is connected for all $ij \in E_p$. In other words, M is a survivable mapping if there is no critical link in G_p under M.

G_l is a *survivable logical topology* on G_p if there exists a survivable mapping from G_l to G_p. The cost of a survivable logical topology G_l, denoted by $cost(G_l)$, is the minimal cost of all survivable mappings from G_l to G_p. A survivable mapping M from G_l to G_p is called a *minimal cost survivable mapping* if $cost(M) = cost(G_l)$.

We say $G_1 = (V_1, E_1)$ *contains* $G_2 = (V_2, E_2)$ if $V_1 = V_2$ and $E_2 \subseteq E_1$. Given a logical topology G_l and a physical topology G_p, a *minimal cost survivable logical topology that contains* G_l on G_p is a survivable logical topology G_l' on G_p such that G_l' contains G_l and $cost(G_l')$ is minimized. We denote the minimized cost as MIN-COST$_{G_l}$, then $cost(G_l') = \text{MIN-COST}_{G_l} = \min cost(G)$, where the min is taken over all G such that G contains G_l and G is a survivable logical topology on G_p.

CHAPTER 3. MAP-AND-FIX HEURISTIC

MAP-and-FIX is a polynomial time heuristic algorithm for the survivable mapping problem. The idea is based on the observation that although it is hard to obtain a survivable mapping in a single attempt, it is possible to obtain a mapping that is *close* to survivable by carefully selecting the routing of the logical links. By close, we mean that the non-survivable mapping can be turned into a survivable one by rerouting only a small number of logical links. This algorithm first computes a mapping that is likely to be survivable or close to survivable; if the mapping is not survivable, then we identify a set of troublesome logical links and reroute them to transform the mapping into a survivable one.

This chapter is organized as follows. In section 3.1, we present the Map-and-Fix algorithm. In section 3.2, numerical results on the performance of the proposed algorithm are provided. Finally, section 3.3 gives a conclusion.

3.1 The MAP-and-FIX Algorithm

3.1.1 Outline

First, we compute a mapping using the Simple-Mapping (SM) algorithm, which maps each logical link to the shortest path in G_p. Next, we check whether the mapping is survivable. If yes, the mapping is returned. Note that this mapping is optimal since it minimizes the total cost of the logical links. If the mapping obtained by SM is not survivable, we compute a new mapping using the Load-Based-Mapping (LBM) algorithm. The new mapping is then checked for survivability. If it is survivable, then it is returned. Otherwise, we use the FIX algorithm to try to convert the mapping to a survivable one by rerouting some logical links. We will present the LBM algorithm and the FIX algorithm in the following two sections. The steps of

MAP-and-FIX is given bellow.

```
MAP-and-FIX($G_l, G_p$)
1.    Compute a mapping $M_1$ using SM;
2.    **if** $M_1$ is survivable **then**
3.        **return** $M_1$;
4.    **else**
5.        Compute a mapping $M_2$ using LBM;
6.    **if** $M_2$ is survivable **then**
7.        **return** $M_2$;
8.    **else**
9.        Modify $M_2$ using FIX;
```

3.1.2 The Load-Based-Mapping Algorithm

The key idea in LBM is to avoid using physical links with high load when routing a logical link because the higher is the load on a physical link, the more likely the link is critical. This is achieved by assigning each physical link a cost based on its current load. Specifically, the cost function of a physical link x is $l(x) + 1$ where $l(x)$ is the number of logical links that traverse x up to this point and the "+1" term ensures that a physical link not used by any logical link will receive a cost of 1. The LBM algorithm routes the logical links sequentially and use Dijkstra's algorithm to compute the least-cost path for each logical link. The load-based cost function is also used by the FIX algorithm to compute the routes for the logical links chosen to be rerouted.

3.1.3 The FIX Algorithm

The input of FIX is a non-survivable mapping M that maps G_l onto G_p. The goal of the algorithm is to "fix" M (i.e., transform M to a survivable mapping) by rerouting some logical links. The fix is done in iterations. In each iteration, we reroute a certain number of bridge links for each critical link in G_p as follows. For each critical link x, we compute the

the number of components in $G_l^M(x)$. If $G_l^M(x)$ has m components ($m > 1$ since x is critical), we pick $m-1$ bridge links of x to reroute. The goal is to ensure that after a bridge link is rerouted, the number of components in $G_l^M(x)$ is reduced by one. We maintain a set called $reroute\_candidates(x)$ from which we pick $m-1$ bridge links to reroute. Initially, this set is set to $B_M(x)$ i.e., it contains all the bridge links of x. We then randomly pick a link l from $reroute\_candidates(x)$ and reroute it so that it does not use x. This will reduce the number of components in $G_l^M(x)$ by 1. Before we pick the next bridge link to reroute, we update $reroute\_candidates(x)$ by removing l and the *siblings* of l from it. Suppose the two endpoints of l are in components C_1 and C_2 in $G_l^M(x)$ before l is rerouted, then the siblings of l, denoted by $siblings(l)$, is the set of links in $reroute\_candidates(x)$ that also have their endpoints in C_1 and C_2. The reason we remove $siblings(l)$ from $reroute\_candidates(x)$ after l is rerouted is that once l is rerouted, the failure of x will not disconnect C_1 and C_2 and therefore rerouting any link in $siblings(l)$ will not decrease the number of components in $G_l^M(x)$. The update of $reroute\_candidates(x)$ guarantees that when we pick a link in $reroute\_candidates(x)$ to reroute, the number of components in $G_l - L_M(x)$ will be reduced by 1. After $m-1$ links are rerouted, $G_l^M(x)$ will have only one component and thus x is no longer critical.

Note that the fix of one critical link may create new critical link(s). Therefore, after we have fixed all the critical links by rerouting some of their bridge links, we check whether the new mapping is survivable. If so, the mapping is returned. Otherwise, we discard the mapping and carry out another iteration to fix M (i.e., pick another set of bridge links for each critical link to reroute). The process is repeated until a survivable mapping is found or the maximum number of iterations is reached. The maximum number of iterations is set to $K * c$ where K is a constant and c is the number of critical links in M. The random selection of to-be-rerouted bridge links coupled with multiple iterations can increase the probability of finding a survivable mapping.

Special handling is needed when we reroute a bridge link e to fix critical link x because e may belong to the bridge-set of some other critical links. Let $critical(e)$ denote the set of critical links that contain e in their bridge-sets. Let $num\_reroute(x)$ be the total number of bridge links of x that needs to be rerouted to fix x ($num\_reroute(x)$ equals the number of components in $G_l^M(x)$ minus 1). Let $to\_reroute(x)$ denote the number of bridge links of x yet to be rerouted to fix x. $to\_reroute(x)$ is initialized to $num\_reroute(x)$ and is decreased by one whenever a bridge link of x is rerouted. x is said *done* when $to\_reroute(x)$ reaches 0. Let $done\_critical\_links$ be a set containing all the done critical links. Suppose we are currently processing critical link x and have chosen bridge link e of x to reroute. When we compute a new route for e, we do not allow it to use any link in $critical(e) - done\_critical\_links$. Thus, the rerouting of e not only contributes to the fix of x, but also contributes to the fix of all other critical links in $critical(e) - done\_critical\_links$. Therefore, $to\_reroute(y)$ should decrease by 1 for all $y \in critical(e) - done\_critical\_links$ after e is rerouted. In addition, for every critical link $y \in critical(e) - done\_critical\_links$, $reroute\_candidates(y)$ should be updated by removing e and $siblings(e)$ from it.

The pseudo-code of the FIX algorithm is given bellow.

```
FIX($G_l$, $G_p$, $M$)
$M$: inout parameter
$G_l, G_p$: in parameter
1.    **let** $c$ = the number of critical links in $G_p$;
2.    **let** $max\_iteration = K * c$;
3.    **let** $num\_iteration = 0$;
4.    **let** $S$ = set of all critical links in $G_p$;
5.    **for** each $x \in S$ **do**
6.        **let** $m$ = number of components in $G_l^M(x)$;
7.        **let** $num\_reroute(x) = m - 1$;
8.    **let** $done\_critical\_links = \emptyset$;
9.    **for** each $x \in S$ **do**
10.       **let** $to\_reroute(x) = num\_reroute(x)$;
11.       **let** $reroute\_candidates(x) = B_M(x)$;
12.   **while** $S - done\_critical\_links \neq \emptyset$ **do**
```

```
13.        Randomly pick a link $x$ from $S - done\_critical\_links$;
14.        while $to\_reroute(x) \neq 0$ do
15.            Randomly pick a link $e$ from $reroute\_candidates(x)$;
16.            Reroute $e$ so that it does NOT use any link in $critical(e) - done\_critical\_links$;
17.            for each $b \in critical(e) - done\_critical\_links$ do
18.                let $to\_reroute(b) = to\_reroute(b) - 1$;
19.                Remove $e$ and $siblings(e)$ from $reroute\_candidates(b)$;
20.                if $to\_reroute(b) = 0$ then
21.                    Add $b$ to $done\_critical\_links$;
22.    let $num\_iteration = num\_iteration + 1$;
23.    if the new mapping is survivable then
24.        return $M$ = the new mapping;
25.    if $num\_iteration < max\_iteration$ then
26.        goto 8
27.    else Print "Survivable mapping not found.";
```

Line 1–7 initialize some variables. Line 8–22 is the iteration that computes a new mapping by fixing M. Following the initializations in line 8–11 is a while loop (line 12–21) that fixes M by rerouting a certain number of bridge links for each critical link. The rerouting reduces the number of components in $G_l^M(x)$ to 1 for every critical link x. Line 22 increases the iteration counter. Line 23–24 return the new mapping if it is survivable. Line 25–27 determine whether another iteration is needed.

3.1.4 Running Time Analysis

Let $G_p = (V, E_p)$ and $G_l = (V, E_l)$ represent the physical topology and the logical topology respectively. The running time of MAP-and-FIX is the sum of the running times of SM, LBM, survivability check, and FIX.

SM has complexity $O(|E_l||V|^2)$ since it takes $O(|V|^2)$ to find the shortest path in G_p for each logical link.

In LBM, the cost of every physical link must be computed first when routing a logical link. Thus, the time to route a logical link is $O(|E_p| + |V|^2)$, which is $O(|V|^2)$ since $|E_p| = O(|V|^2)$. There are $|E_l|$ logical links to route, so LBM has complexity $O(|E_l||V|^2)$.

To determine if a mapping is survivable, we check whether $G_l^M(x)$ is connected using breadth first search for each physical link x. Since breadth first search takes $O(|V| + |E_l|)$, the total time for survivability check is $O(|E_p|(|V| + |E_l|))$.

The running time of FIX is determined by the time taken in one iteration and the number of iterations performed. The running time for one iteration is dominated by the rerouting time for fixing all critical links. At most $O(|E_l|)$ links will be rerouted, so the running time of one iteration is $O(|E_l||V|^2)$. There are at most $K * |L|$ iterations, where K is a constant and L is the set of all critical links in the input mapping. So the running time of FIX is $O(|L||E_l||V|^2)$, which is $O(|E_p||E_l||V|^2)$ since $|L|$ is $O(|E_p|)$.

Summing up the running times of SM, LBM, survivability check, and FIX gives the running time of MAP-and-FIX, which is $O(|E_p||E_l||V|^2)$.

3.2 Numerical Results

To evaluate the performance of MAP-and-FIX, we conduct simulations on various logical and physical topologies. Two physical topologies are used. One is the 14-node 21-link NSFNET shown in Figure 3.1 (a), the other is a randomly generated 10-node 15-link topology shown in Figure 3.1 (b). Both physical topologies have average node degree 3. For the NSFNET physical topology, two groups of 14-node 21-link logical topologies are mapped onto it. The first group consists of 100 3-regular (every node has degree 3) topologies. The second group consists of 100 arbitrary topologies. For the 10-node 15-link random physical topology, a group of 100 10-node 17-edge arbitrary logical topologies are mapped onto it. All logical topologies are randomly generated and 2-connected. All simulations are run on a Sun Ultra 10 workstation with a 440MHz CPU, 256MB RAM, 4GB virtual memory. CPLEX8.1 is used as the ILP solver.

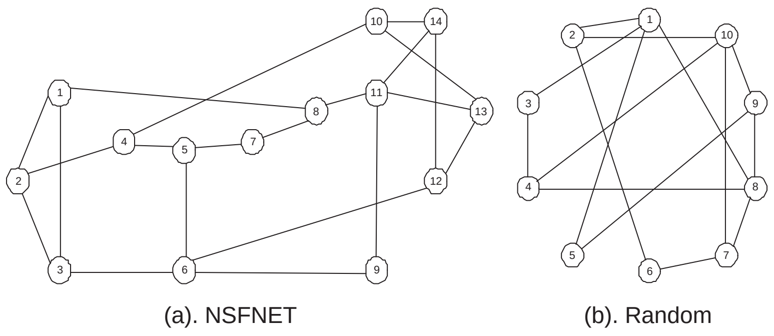


Figure 3.1 (a). NSFNET: a 14-node 21-link physical topology. (b). Random: a 10-node 15-link random physical topology.

3.2.1 Performance Comparison of MAP-and-FIX, ILP, and SM

For each group of 100 logical topologies and the corresponding physical topology, ILP [20], SM, and MAP-and-FIX are used to solve the survivable mapping problem. The number of solutions found, the average cost of the solutions, and the average runtime are shown in Table 3.1. Here the cost of a solution is the sum over all logical links' costs (recall that the cost of a logical link is the hop count of its physical path). In the FIX procedure, the maximum number of iterations is set to 10 times the number of critical links, i.e., $K = 10$.

Table 3.1 Performance Comparison of MAP-and-FIX, ILP, and SM

Topologies (Logical over Physical)	Algorithm	# of solutions	Ave cost	Ave runtime (sec)
14-node 3-regular over NSFNET	ILP	100	45.91	677.352
	SM	17	43.00	0.068
	MAP-and-FIX	100	47.98	0.137
14-node 21-link over NSFNET	ILP	99	46.71	692.499
	SM	6	42.5	0.068
	MAP-and-FIX	86	48.53	0.169
10-node 17-link over Random	ILP	100	31.78	1.906
	SM	28	30.79	0.041
	MAP-and-FIX	99	32.78	0.074

As shown in Table 3.1, MAP-and-FIX outperforms SM considerably in terms of the number of solutions found. This demonstrates the effectiveness of LBM and FIX used in MAP-and-FIX. In addition, the success ratio of MAP-and-FIX (i.e., the ratio of the number of solutions found by MAP-and-FIX to the number of solutions found by ILP) is very high. For 14-node 21-link arbitrary logical topologies, MAP-and-FIX has a success ratio of 86.9% (86/99), which is not as good as the success ratio for 14-node 3-regular logical topologies (100%) even though both groups of logical topologies have the same number of nodes and links. This can be explained by the fact that almost all r-regular graphs are r-connected [28]. Since almost all 3-regular topologies are 3-connected and 3-connected topologies have better connectivity than arbitrary 2-connected topologies, it is easier to find survivable mappings for 3-regular topologies than for arbitrary 2-connected topologies. The success ratio of the 10-node 17-link arbitrary logical topologies is 99%, which is about 12% higher than that of the 14-node 21-link arbitrary logical topologies. The reason is that 10-node 17-link logical topologies (average node degree 3.4) are denser than 14-node 21-link logical topologies (average node degree 3.0), and the denser is a logical topology, the easier can a survivable mapping be found.

To evaluate the capability of MAP-and-FIX to find optimal solutions, we compare the average cost of solutions found by MAP-and-FIX and by ILP for those logical topologies that MAP-and-FIX can find a solution. For 14-node 3-regular logical topologies, MAP-and-FIX found solutions for all 100 logical topologies with an average cost 2.07 (or 4.5%) higher than ILP solutions. Moreover, 31 solutions obtained by MAP-and-FIX are optimal. For 14-node 21-link arbitrary logical topologies, the average extra-cost over ILP solutions is 2.30 (or 5.0%) for the 86 solutions, out of which 16 are optimal. For 10-node 17-link logical topologies, the average extra-cost is 1.04 (or 3.3%), and 42 out of 99 solutions found by MAP-and-FIX are optimal. These results show that the solutions found by MAP-and-FIX are very close to optimal.

As of runtime, we can see that MAP-and-FIX runs much faster than ILP. The speed-up is $677.352/0.137 \approx 4944$ for the 14-node 3-regular logical topologies, $692.499/0.169 \approx 4098$ for the 14-node 21-link logical topologies, and $1.906/0.074 \approx 26$ for the 10-node 17-link logical topologies.

3.2.2 Effectiveness of Different Procedures in MAP-and-FIX

Three procedures–SM, LBM, and FIX–are used in MAP-and-FIX. To find out the effectiveness of these procedures, the number of solutions found by each of them are shown in Table 3.2. For example, among the 86 solutions for the 14-node 21-link arbitrary logical topologies, 6 are obtained by SM, 10 more are obtained by applying LBM, and 70 are obtained by FIX after SM and LBM fail to find a survivable mapping. The combination of these three procedures leads to good overall performance.

Table 3.2 Number of solutions obtained by different procedures in MAP-and-FIX

Topologies		By SP	By LBM	By FIX
Logical	Physical			
14-node 3-regular	NSFNET	17	23	60
14-node 21-link arbitrary	NSFNET	6	10	70
10-node 17-link arbitrary	Random	28	27	44

To study the runtime of FIX, the number of iterations and the number of reroutings per iteration are recorded for each logical topology whose solution is obtained by FIX. The average number of iterations and the average reroutings per iteration for the three simulation scenarios are shown in Table 3.3. The averages are taken over all solutions that are obtained by FIX, i.e., solutions obtained by SM and LBM are not counted. Table 3.3 shows that no more than 2 iterations on average are needed to fix a non-survivable mapping. And on average, no more than 3 reroutings are needed in one iteration. These results demonstrate that the mappings obtained by LBM are indeed *close* to survivable since only a small number of logical links needs to be rerouted to fix the mapping.

Table 3.3 Performance of FIX

Topologies		Average number of iterations	Average rerouting per iteration
Logical	Physical		
14-node 3-regular	NSFNET	1.12	1.48
14-node 21-link	NSFNET	1.66	2.20
10-node 17-link	Random	1.23	1.20

To see how close are the non-survivable mappings obtained by LBM to survivable, we classify them into four types. A mapping is type A if it has only one critical link and the failure of the critical link will partition the logical topology into exactly two components. A mapping is type B if it has more than one critical links and the failure of each critical link will partition the logical topology into exactly two components. A mapping is type C if it has only one critical link and the failure of the critical link will partition the logical topology into more than two components. A mapping is type D if it has more than one critical links and the failure of at least one of the critical links will partition the logical topology into more than two components. Intuitively, the more logical links are rerouted to fix a critical link, the more likely that a new critical link will be created, which makes the new mapping non-survivable. Thus, type A and B are easy to fix because only one logical link needs to be rerouted to fix a critical link and type C and D are hard to fix because at least one logical link needs to be fixed by rerouting more than one logical link. Table 3.4 shows how many non-survivable mappings belong to each of the four types and how many of them can be fixed by FIX. A table entry a/b in the column labeled by X means that there are a total of b type X non-survivable mappings, out of which a can be converted to a survivable mapping by FIX. Note that the sum of the a values in a row equals the total number of mappings fixed by FIX for the corresponding logical and physical topologies. The table shows that for all three groups of logical/physical topologies, most of the non-survivable mappings are of type A and B and most of the type A and B mappings can be fixed by FIX. This confirmed that most non-survivable mappings obtained by LBM are easy to fix and FIX is very effective in fixing type A and type B mappings.

Table 3.4 Number of fixed/total non-survivable mappings for the four types.

Topologies		A	B	C	D
Logical	Physical				
14-node 3-regular	NSFNET	35/35	23/23	1/1	1/1
14-node 21-link arbitrary	NSFNET	20/21	45/51	0/0	5/12
10-node 17-link arbitrary	Random	32/33	11/11	1/1	0/0

3.3 Conclusion

This chapter provides an effective polynomial time heuristic algorithm, MAP-and-FIX, for the survivable mapping problem. The algorithm first uses the Simple-Mapping algorithm and the Load-Based-Mapping algorithm to find a mapping for the given logical topology and physical topology. If both fail to produce a survivable mapping, the FIX algorithm is used to reroute a set of logical links for each critical link in the non-survivable mapping. The fix is done in iterations until a survivable mapping is found or the maximum number of iterations is reached. Simulation results show that MAP-and-FIX runs much faster than the ILP [20] and can find a survivable mapping with high probability if such a mapping exists. The effectiveness of MAP-and-FIX lies in the fact that most of the non-survivable mappings obtained by LBM are close to survivable so that only a small number reroutings are needed to fix the mapping. In addition, the solutions obtained by MAP-and-FIX are very close to optimal in terms of the solution cost.

CHAPTER 4. LOGICAL TOPOLOGY AUGMENTATION

In this chapter, a new approach to tackle the inherent hardness of the survivable mapping problem is introduced. Instead of trying to find a survivable mapping for the given logical topology on the physical topology, we augment the logical topology to an extent such that a survivable mapping can be polynomial-time computed. In other words, we make the survivable mapping problem polynomial-time solvable by selectively introducing new logical links. As the theoretical foundation, we will prove that as long as the intersection of the logical and the physical topologies is 2-edge-connected, any shortest path mapping from the logical topology to the physical topology is survivable. Then a natural optimization is to minimize the number of edges added to make the intersection of the logical and physical topologies 2-edge-connected. This graph augmentation problem is NP-hard [29]. We provide an integer linear programming (ILP) formulation to find its optimal solution. Moreover, to make the scheme a complete polynomial-time solution, we offer a heuristic for the augmentation problem as well.

This chapter is organized as follows. In section 4.1, we give a theorem about the condition when any reflectively-routed mapping is survivable. In section 4.2, we first present the general framework of our scheme, and then describe an ILP formulation that solves the graph augmentation problem optimally. As an alternative, we also give a heuristic to the augmentation problem. Simulations for our scheme as well as the performance comparison between the ILP and the heuristic are discussed in section 4.3. Finally, a conclusion is given in section 4.4.

4.1 Theoretical Foundation

4.1.1 When is any reflectively-routed mapping survivable?

Our goal is to augment the logical topology such that a survivable mapping could be found easily. A natural direction is to consider shortest-path mappings. So, it would be helpful to know under what condition is **any** shortest-path mapping survivable. The reason why we emphasize **any** shortest-path mapping be survivable, instead of just some of them is that for each logical link, there might be multiple shortest paths in the physical topology. Combinatorially, this could blow up the number of different shortest-path mappings. If not all of them are survivable, the situation will be complex when we are trying to pick survivable ones out of them, which compromises the simplicity of the potential shortest-path mapping algorithms.

By definition, shortest-path mappings are reflectively-routed. So any reflectively-routed mapping being survivable implies any shortest-path mapping being survivable. In the following theorem, we give a condition under which any reflectively-routed mapping is survivable given G_l and G_p.

Theorem 4.1 *Given $G_l = (V, E_l)$ and $G_p = (V, E_p)$, let $G = (V, E_l \cap E_p)$. If G is 2-edge-connected, then any reflectively-routed mapping from G_l to G_p is survivable.*

Proof: For any reflectively-routed mapping M from G_l to G_p, by definition, $\forall st \in E_l \cap E_p$, $M(st) = (s - t)$. Under this mapping, in case of any single physical link $ij \in E_p$ failure, among logical links in E_l, at least those in $E(G) - \{ij\}$ will stay in the remaining logical topology because none of their physical paths traverses ij. And since G is 2-edge-connected, logical links in $E(G) - \{ij\}$ compose a connected graph. In other words, any single link failure in the physical topology does NOT disconnect G_l under mapping M. Therefore M is a survivable mapping from G_l to G_p. ▮

4.2 Algorithms

4.2.1 General framework

The procedure of the proposed scheme is described by the following pseudo-code.

```
Easy_Survivable_Mapping($G_l$, $G_p$, $M$)
$G_l$, $M$: inout
$G_p$: in
1.    let $G = (V, E_l \cap E_p)$;
2.    let $G_h = (V, E_p - E(G))$;
3.    let $E$ =Augment($G$, $G_h$);
4.    let $G_l = (V, E_l \cup E)$;
5.    Compute a shortest-path mapping $M$ from $G_l$ to $G_p$;
```

In this procedure, we first compute the intersection (G) of the logical and physical topologies. Then the edges that can be used for augmenting the logical topology together with all vertices form G_h. In line 3, the function **Augment** returns the edges in G_h needed to add into G to make G 2-edge-connected. If G is already 2-edge-connected, function **Augment** just returns an empty set. By **Theorem 4.1**, we know that the mapping M computed in line 5 is survivable. Unfortunately, the problem of augmenting a graph to 2-edge-connected using a given set of edges while minimizing the number of edges added is NP-hard [29]. We will provide an ILP formulation solving the problem optimally and give a heuristic algorithm for function **Augment**.

4.2.2 An ILP formulation

The ILP formulation has the following variables:

- x^{st}: takes value 1 if st is included in the result logical topology, 0 otherwise, $\forall st \in E_p$.

$$\text{Minimize} \sum_{st \in E_p} x^{st}$$

Subject to:

(a). 2-edge-connectivity constraints:

$$\sum_{\substack{(s\in S\wedge t\in V-S)\\ \vee \\ (s\in V-S\wedge t\in S)}} x^{st} \geq 2,\ \forall S \subset V.$$

(b). Deletion of existing logical links is not allowed:

$$x^{st} = 1,\ \forall st \in E_l \cap E_p.$$

(c). Integer constraints:

$$x^{st} \in \{0, 1\},\ \forall st \in E_p - E_l.$$

The 2-edge-connectivity constraints in (a) guarantee that the number of links in each edge cut is at least 2, i.e., the solution constitutes a 2-edge-connected spanning subgraph of G_p. The constraints in (b) make sure that existing logical links in the intersection topology are kept. The objective function minimizes the number of edges in the result logical topology, hence the number of edges added to the logical topology is minimized.

4.2.3 A Heuristic

The heuristic algorithm for function **Augment** is described as follows. First, we find all 2-edge-connected components in G. Since no edge inside a 2-edge-connected component needs to be considered, a 2-edge-connected component will be treated as a single vertex for augmentation. So, contraction according to 2-edge-connected components in G (i,e, contracting the vertex set in each 2-edge-connected component in G) are done on G and correspondingly on G_h. (To contract a vertex set $X \subseteq V(G)$ means that all vertices in X are replaced by a new vertex x. Edges with both endpoints in X are gone. Edges incident to one vertex in X becomes edges incident to x. Note that contraction can result in parallel edges. The default setting is to drop parallel edges after contraction.) After the contraction, G becomes a forest with one or more trees. The augmentation of G consists of 2 steps: first, "Augment1" adds edges to G to make it a spanning tree using edges in G_h; then, "Augment2" continues to add edges to G until it becomes 2-edge-connected. Finally, the edge set that contains the added edges during the two steps is returned. The pseudo-code of the heuristic algorithm is given below.

```
Augment_Heuristic($G$, $G_h$)
$G$, $G_h$: in
1.    **let** $C$ = all 2-edge-connected components in $G$;
2.    Contract components in $C$ on $G$ and $G_h$*;
3.    **let** $E_1$ = Augment1($G$, $G_h$);
4.    **let** $E_2$ = Augment2($G$, $G_h$);
5.    **return** $E_1 \cup E_2$;
*: For the contraction on $G_h$, keep resulted parallel edges.
```

Augment1 adds edges into G one by one, each new edge decreases the number of trees in G by one. So it always returns an edge set with size $|V(G)| - |E(G)| - 1$. However, the shape of the resulted tree has strong impact for the next step Augment2 and a tree with less leaves (the degree one vertices) is more desirable for Augment2 in which leaves cost edges to be added. (In Augment2, we will add leaf-incident edges to eliminate leaves.) To minimize the number of resulted leaves after adding an edge to G, we want to maximize the length of the longest path in the resulted tree. Thus, in Augment1, we add edges one by one, each time making a greedy choice. That is, we always add an edge such that the length of the longest path of the resulted tree is maximized. To facilitate this goal, we define a weight for each candidate edge in $E(G_h)$ to be the length of the longest path in the tree resulted from adding this edge. For $uv \in E(G_h)$ where u, v are in different trees in G, the weight of uv turns out to be $\text{height}(T_u) + \text{height}(T_v) + 1$ where T_u/T_v is the tree that is rooted at u/v, and $\text{height}(T_u)/\text{height}(T_v)$ is the length of the longest path from the root u/v to a leaf in T_u/T_v. An iteration of Augment1 is illustrated in Figure 4.1. Note that after adding $x - y$ to G, the weights of the edges in G_h are updated.

```
Augment1($G$, $G_h$)
$G$, $G_h$: inout
```

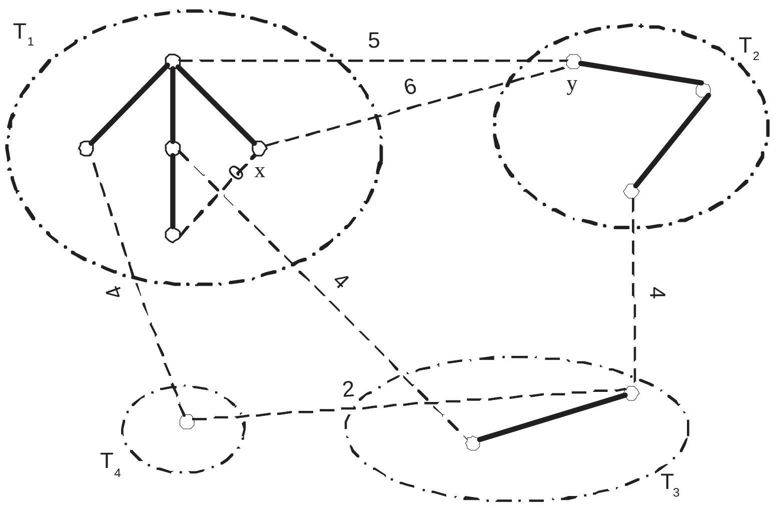
```
1.  let $A = \emptyset$;
2.  while $G$ is not connected do
3.      for each $uv \in E(G_h)$ do
4.          if $u$, $v$ are in different trees of $G$ then
5.              let weight$(u, v) =$ height$(T_u) +$ height$(T_v) + 1$;
6.          else
7.              let weight$(u, v) = 0$;
8.      Pick an edge $xy \in E(G_h)$ with the maximum weight;
9.      let $G = G \cup \{xy\}$;
10.     let $E(G_h) = E(G_h) - \{xy\}$;
11.     let $A = A \cup \{xy^*\}$;
12. return $A$;
```

*: For the returning set A, the edge in the original G_h
(before contraction) corresponding to xy should be used.

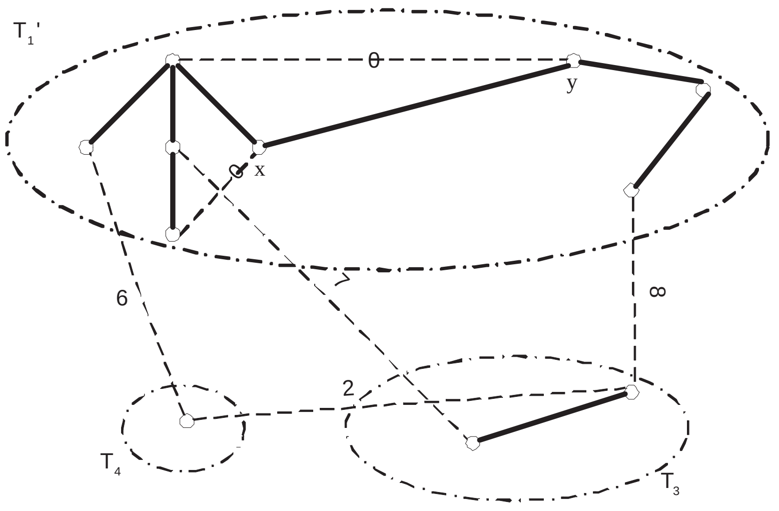
Augment2 finds out edges needed to augment a tree to a 2-edge-connected graph. This algorithm is due to Even et al in [29]. The idea is to always add an edge incident to a leaf such that the unique resulted cycle contains the most vertices. To achieve this objective, we define a weight for each candidate edge $uv \in E(G_h)$ (at least one of u and v is a leaf) to be the length of the unique path between u and v in the tree, which is denoted by $P_G(u, v)$. Whenever $uv \in E(G_h)$ is selected to be added to G, we contract the vertices on path $P_G(u, v)$ before we proceed to the next iteration. The procedure terminates when G is contracted to one vertex, which means that the added edges have made it 2-edge-connected.

Augment2(G, G_h)
G, G_h: in

```
1.  let $A = \emptyset$;
2.  while $|V(G)| > 1$ do
3.      for each $uv \in E(G_h)$ do
```



(a). Before adding x-y



(b). After adding x-y

Figure 4.1 An iteration of Augment1. Solid lines are current edges in G, and dashed lines with weights are edges in G_h.

```
4.            if at least one of $u$, $v$ is a leaf in $G$ then
5.                let weight$(u,v)$ = length$(P_G(u,v)^*)$;
6.            else
7.                let weight$(u,v) = 0$;
8.        Pick an edge $xy \in E(G_h)$ with the maximum weight;
9.        let $A = A \cup \{xy^{**}\}$;
10.       Contract $V(P_G(x,y))$ on $G$ and $G_h$;
11.   return $A$;
```

*: Note that throughout Augment2, we never really add edges into G. $P_G(u,v)$ is always valid since G is always a tree.

**: For the returning set A, the edge in the original G_h (before all contractions) corresponding to xy should be used.

4.3 Numerical Results

To evaluate the effectiveness of the strategy proposed in this chapter, we run simulations on 100 pairs of 14-node, 21-link random logical and physical topologies, all of which are 2-edge-connected. The minimal cost survivable mappings are computed using the ILP given in [20] that is referred to as "OPT" later in this chapter. (The cost of a mapping is defined as the total length of the physical paths for all logical links.) The algorithm "Easy_Survivable_Mapping" will be denoted by "ESM_ILP" and "ESM_HEURISTIC" depending on the choice of its augmentation algorithm. All simulations are run on a Sun Ultra 10 workstation equipped with a single 440MHz CPU, 256MB RAM, and 4GB virtual memory. CPLEX8.1 is used to solve the ILP formulations.

4.3.1 OPT vs. ESM_HEURISTIC

As shown in Table 4.1, among the 100 pairs of logical/physical topologies, 7 of them don't have survivable mappings (note that a survivable mapping may not exist for an arbitrary pair of logical/physical topologies). OPT can do nothing but output "infeasible" for those 7 pairs,

Table 4.1 Performance comparison of OPT and ESM_HEURISTIC

Mapping Algorithm	OPT	ESM_HEURISTIC
Number of non-survivable logical/physical pairs	7	0
Avg. cost of survivable mappings	50.01	59.22
Avg. running time (sec)	2.897	0.012

Table 4.2 Performance comparison of ESM_ILP and ESM_HEURISTIC

Augmentation Algorithm	ESM_ILP	ESM_HEURISTIC
Avg. number of added edges	10.98	11.96
Avg. running time (sec)	1.001	0.010

however, ESM_HEURISTIC can provide solutions by augmenting the logical topologies.

Table 4.1 also shows that the speed-up of ESM_HEURISTIC over OPT is $2.897/0.012 \approx 241$; on the other hand, ESM_HEURISTIC pays an average of 9.21 extra cost for the mappings. It can be verified that as the size of logical/physical topologies grows, the expected running time difference between OPT and ESM_HEURISTIC increases dramatically since ESM_HEURISTIC is a completely polynomial-time algorithm.

4.3.2 ESM_ILP vs. ESM_HEURISTIC

ESM_ILP and ESM_HEURISTIC (excluding the computation of the shortest-path mapping) are run to compare the performance of the ILP and the heuristic for the augmentation problem. Table 4.2 shows that the heuristic runs much faster than the ILP. It can also be verified that as the size of logical/physical topologies goes up, this gap increases exponentially. Meanwhile, the average number of added edges output by the heuristic is close to the optimal results from the ILP.

Recall that the average extra cost produced by ESM_HEURISTIC is about 9.21 over OPT, while the average number of edges added is 11.96. This discrepancy can be explained as follows. The minimal cost survivable mapping from a given logical topology (without augmentation) to a physical topology might not be a shortest-path mapping because of the survivability constraints. However, after the augmentation, any shortest-path mapping will be survivable for the resulted logical topology. Thus, given a logical topology and a physical topology, it is always true that the extra cost resulted from ESM_HEURISTIC(ESM_ILP) over OPT is less than or equal to the number of edges added computed by ESM_HEURISTIC(ESM_ILP).

4.4 Conclusion

This chapter presents a new approach to the survivable mapping problem. We proved that if the intersection of the given logical and physical topology is 2-edge-connected, then any reflectively-routed mapping (hence any shortest-path mapping) is survivable. Based on this result, we proposed the Easy_Survivable_Mapping algorithm that augments the given logical topology until its intersection with the given physical topology becomes 2-edge-connected. To solve the NP-hard 2-edge-connectivity augmentation problem within Easy_Survivable_Mapping, we give an ILP formulation to obtain optimal solution and a heuristic algorithm to make Easy_Survivable_Mapping a polynomial-time algorithm. Simulation results show that our scheme can find survivable mappings extremely fast while incuring additional mapping costs. In addition, our scheme addresses the situation when there does not exist a survivable mapping for the given logical and physical topologies. Furthermore, the performance comparison between the ILP and the heuristic for the 2-edge-connectivity augmentation problem demonstrates the encouraging effectiveness and efficiency of the heuristic.

CHAPTER 5. NEW SURVIVABLE MAPPING PROBLEM

5.1 Introduction

5.1.1 Background

While the original survivable mapping problem does not allow the given logical topology to be changed when finding a survivable mapping for it, we note that it is sometimes beneficial to add logical links to the given logical topology for two reasons. First, if the given logical topology does not have a survivable mapping on the given physical topology, adding some logical links to the given logical topology will enable a survivable mapping to be obtained. Second, even if a survivable mapping for the given logical topology exists, adding some logical links to the given logical topology may reduce the minimal survivable mapping cost. As in [20], we define the cost of a mapping as the total number of wavelength channels used to map all the logical links in the logical topology. Since a logical link (i.e., a lightpath) uses one wavelength channel on each link along its physical path, the cost of a logical link equals the number of hops in its physical path and the cost of a mapping equals the total cost of all logical links in the logical topology. To see the benefit of adding logical links to a logical topology in reducing the minimal survivable mapping cost, consider the logical and physical topologies shown in Figure 2.1. The left table in Figure 5.1 shows the minimal cost survivable mapping for the logical and physical topologies, which has a cost of 10. If we add link (a, e) to the logical topology, a survivable mapping with a cost of 9 can be obtained, as shown in the right table in Figure 5.1. Due to the benefits of adding logical links in a logical topology, we propose a new version of the survivable mapping problem and study the problem in this chapter. The new survivable mapping problem is the following: Given a logical topology G_l and a physical topology G_p, compute a logical topology G'_l by adding 0 or more logical links to G_l such that G'_l has a survivable mapping on G_p and the cost of the survivable mapping is minimized.

A minimal cost survivable mapping
for the original logical topology

Logical link	Physical path
(a, b)	A-E-D-B
(a, c)	A-B-C
(b, d)	B-D
(b, e)	B-A-E
(c, e)	C-E
(d, e)	D-E
Cost of mapping	10

A minimal cost survivable mapping
after adding logical link (a, e)

Logical link	Physical path
(a, b)	*A-B*
(a, c)	A-B-C
(a, e)	**A-E**
(b, d)	B-D
(b, e)	B-A-E
(c, e)	C-E
(d, e)	D-E
Cost of mapping	9

Figure 5.1 Minimal cost survivable mapping for the logical topology in Figure 2.1(a) before and after adding the logical link (a, e). Note that the logical link (a, b) is mapped differently in the original logical topology and the new logical topology.

The idea of adding logical links to a logical topology to enable a survivable mapping has been explored in [26]. An algorithm is given in [26] to identify a good logical link to add to the given logical topology and simulation results show that adding one logical link can enable a survivable mapping in most cases. A drawback of the algorithm is that it does not guarantee to enable a survivable mapping since only one logical link is added to the logical topology. In [30], we propose a method to add logical links to a given logical topology so that the resulting logical topology has a survivable mapping on the given physical topology and any shortest path mapping of the resulting logical topology on the physical topology is survivable. (A shortest path mapping maps every logical link to the shortest physical path, which is polynomial-time computable.) Clearly, the method can be used to enable a survivable mapping for a logical topology if it does not have one. However, the cost of the survivable mapping for the resulting logical topology may not be minimized because the resulting logical topology is chosen such that a survivable mapping can be computed in polynomial time.

5.1.2 Problem Definition

We now give the formal definition of the new survivable mapping problem (NSM).

NSM: Given a logical topology $G_l = (V, E_l)$ and a 2-edge-connected physical topology $G_p = (V, E_p)$, compute a minimal cost survivable logical topology G'_l that contains G_l on G_p and a mapping M from G'_l to G_p such that $cost(M) = \text{MIN-COST}_{G_l}$.

Note that physical topologies are required to be 2-edge-connected in practice so that any single physical link failure does not disconnect the physical topology. (A graph is 2-edge-connected if the minimal number of edges whose removal disconnects the graph is 2.)

The rest of this chapter is organized as follows. In section 5.2, we prove that a solution to the new survivable mapping problem always exists and give a straightforward ILP formulation to solve the problem. In section 5.3, we first present a theoretical result about the new survivable mapping problem. Based on the theoretical result, we then provide an improved ILP formulation for the problem and give an NP-hardness proof of the problem. Simulation results are discussed in section 5.4. Finally, a conclusion is given in section 5.5.

5.2 A Straightforward ILP Formulation

First, we prove that a solution to NSM always exists.

Theorem 5.1 *Given a logical topology $G_l = (V, E_l)$ and a 2-edge-connected physical topology $G_p = (V, E_p)$, there exists a survivable logical topology that contains G_l on G_p.*

Proof: Let $G'_l = (V, E_l \cup E_p)$. Clearly, G'_l contains G_l. We next prove that G'_l is a survivable logical topology on G_p by showing that there exists a survivable mapping from G'_l to G_p. Let M be a reflectively-routed mapping from G'_l to G_p. For any $ij \in E_p$, $G_l'^M(ij)$ contains all links in $E_p - \{ij\}$. Since G_p is 2-edge-connected, $G_l'^M(ij)$ must be connected. Thus, M is a survivable mapping from G'_l to G_p. ∎

We now present a straightforward ILP formulation (referred to as ILP1) that solves NSM. Let K_n denote the undirected complete graph on the vertex set V, where $n = |V|$. ILP1 considers all edges in $E(K_n)$ as candidate edges to be included in the resulting logical topology, where $E(K_n)$ is the edge set of K_n.

Variables to be solved:

- f_{ij}^{st}: takes value 1 if logical link st is mapped to a path that contains physical link ij, 0 otherwise.

- x^{st}: takes value 1 if st is included in the resulting logical topology, 0 otherwise.

Objective function:

$$\text{Minimize} \sum_{\substack{ij\in E_p^d \\ st\in E(K_n)}} f_{ij}^{st}$$

Subject to:

(a). Flow conservation constraints:

$$\sum_{\substack{j \text{ s.t.} \\ ij\in E_p^d}} f_{ij}^{st} - \sum_{\substack{j \text{ s.t.} \\ ji\in E_p^d}} f_{ji}^{st} = \begin{cases} x^{st} & \text{if } s = i \\ -x^{st} & \text{if } t = i \\ 0 & \text{otherwise} \end{cases},$$

$$\forall i \in V,\ \forall st \in E(K_n).$$

(b). Survivability constraints:

$$\sum_{\substack{(s\in S\wedge t\in V-S) \\ \vee \\ (s\in V-S\wedge t\in S)}} f_{ij}^{st} + f_{ji}^{st} < \sum_{\substack{(s\in S\wedge t\in V-S) \\ \vee \\ (s\in V-S\wedge t\in S)}} x^{st},$$

$$\forall ij \in E_p,\ \forall S \subset V.$$

(c). Logical links in the given logical topology must be kept:

$$x^{st} = 1,\ \forall st \in E_l.$$

(d). Integer constraints:

$$f_{ij}^{st} \in \{0,1\},\ \forall ij \in E_p^d,\ \forall st \in E(K_n).$$

$$x^{st} \in \{0,1\},\ \forall st \in E(K_n).$$

The flow conservation constraints in (a) ensure that a logical link is mapped to a physical path only if it is included in the resulting logical topology, i.e., $x^{st} = 1$. In the survivability constraints in (b), the right hand side is the number of edges in the edge cut $EC_{G_l'}(S)$ and the left hand side is the number of edges in $EC_{G_l'}(S)$ that are mapped to a physical path containing

$ij \in E_p$ in either direction, which equals $|l_M(ij) \cap EC_{G_l'}(S)|$, where G_l' is the resulting logical topology and M is the resulting mapping from G_l' to G_p. It is proved in [20] that M is survivable if and only if $|l_M(ij) \cap EC_{G_l'}(S)| < |EC_{G_l'}(S)|$ for all $ij \in E_p$ and all $S \subset V$. Therefore, the constraints in (b) ensure that the resulting mapping is survivable. Constraints in (c) ensure that the logical links in the given logical topology must stay in the resulting logical topology.

5.3 A Theorem and Its Applications

ILP1 provides a straightforward method for solving NSM, which considers all links not in G_l as candidate links to be added to G_l. In this section, we present a theorem which shows that we can obtain a solution to NSM by adding only reflective logical links to G_l, and the resulting logical topology has a reflectively-routed survivable mapping that achieves the minimal cost. We also give two applications of the theorem: an improved ILP for NSM and an NP-hardness proof for NSM.

5.3.1 A Theorem

Theorem 5.2 *Given a logical topology $G_l = (V, E_l)$ and a 2-edge-connected physical topology $G_p = (V, E_p)$, there exists an edge set $E'' \subseteq E_p - E_l$ such that $G_l'' = (V, E_l \cup E'')$ is a minimal cost survivable logical topology that contains G_l on G_p. Moreover, there is a reflectively-routed survivable mapping M'' from G_l'' to G_p such that cost$(M'') =$ MIN-COST$_{G_l}$.*

Proof: See Appendix. ▮

The theorem shows that given a logical topology G_l and a 2-edge-connected physical topology G_p, it is always possible to obtain a minimal cost survivable logical topology that contains G_l on G_p by adding only reflective logical links to G_l. Furthermore, the resulting logical topology has a reflectively-routed survivable mapping that achieves the minimal cost.

5.3.2 An Improved ILP Formulation

Theorem 5.2 can be used to improve ILP1 in two ways. First, the candidate logical links to be included in the resulting logical topology can be confined to links in $E_l \cup E_p$ instead of links in $E(K_n)$. This helps reduce the number of variables in ILP1. Second, the existence of the minimal cost reflectively-routed survivable mapping for the resulting logical topology makes the mapping job easier since the physical paths for those reflective logical links can be determined right away (they are reflectively-routed). The improved ILP, referred to as ILP2, is given as follows.

$$\text{Minimize} \sum_{\substack{ij \in E_p^d \\ st \in E_l \cup E_p}} f_{ij}^{st}$$

Subject to:

(a). Flow conservation constraints:

$$\sum_{\substack{j \text{ s.t.} \\ ij \in E_p^d}} f_{ij}^{st} - \sum_{\substack{j \text{ s.t.} \\ ji \in E_p^d}} f_{ji}^{st} = \begin{cases} 1 & \text{if } s = i \\ -1 & \text{if } t = i \\ 0 & \text{otherwise} \end{cases},$$

$$\forall i \in V, \ \forall st \in E_l - E_p.$$

(a'). Reflectively-routed constraints:

$$f_{st}^{st} = x^{st}, \ \forall st \in E_p. \tag{5.1}$$

$$f_{ij}^{st} = 0, \ \forall st \in E_p, \ ij \in E_p^d \text{ and } (i \neq s \vee j \neq t). \tag{5.2}$$

(b). Survivability contraints: Same as those in ILP1.

(c). Logical links in the given logical topology must be kept: Same as those in ILP1.

(d). Integer constraints:

$$f_{ij}^{st} \in \{0, 1\}, \ \forall ij \in E_p^d, \ \forall st \in E_l \cup E_p.$$

$$x^{st} \in \{0, 1\}, \ \forall st \in E_l \cup E_p.$$

The flow conservation constraints in (a) are only used for logical links in $E_l - E_p$ because other logical links are reflective and will be reflectively-routed. The constraints in (a') ensure that reflective logical links are reflectively-routed. Note that the existence of a resulting survivable logical topology and the corresponding reflectively-routed survivable mapping is guaranteed by Theorem 5.2.

Compared with ILP1, ILP2 has fewer variables and fewer flow conservation constraints. As a result, ILP2 runs faster than ILP1, as will be shown in section 5.4.

5.3.3 NP-hardness of NSM

With the help of Theorem 5.2, we can prove that NSM is NP-hard by showing that M2ECSS is polynomial-time reducible to NSM, where M2ECSS stands for the Minimum 2-Edge-Connected Spanning Subgraph problem that has been proved to be NP-hard [31].

For the purpose of the proof, we define the decision problem of M2ECSS and NSM as follows.

M2ECSS: Given a graph G and a positive integer k, determine whether G has a 2-edge-connected spanning subgraph containing at most k edges.

NSM: Given a logical topology G_l, a physical topology G_p, and a positive integer c, determine whether there is a survivable logical topology G_l' that contains G_l on G_p such that the cost of G_l' is at most c.

Theorem 5.3 *NSM is NP-hard.*

Proof: We show that M2ECSS is polynomial-time reducible to NSM.

Given an instance $\langle G, k\rangle$ of M2ECSS, we construct an instance $\langle G_l, G_p, c\rangle$ of NSM as follows:

- Let G_l be a graph with the same vertex set as G and no edges, which is denoted as $G_\emptyset$.
- Let $G_p = G$.
- Let $c = k$.

Clearly, the construction is polynomial-time computable.

Next, we show that G has a 2-edge-connected spanning subgraph containing at most k edges if and only if there is a survivable logical topology G_l' that contains $G_\emptyset$ on G such that the cost of G_l' is at most k.

Suppose G_{sub} is a 2-edge-connected spanning subgraph of G and $|E(G_{sub})| \leq k$. Consider G_{sub} as a logical topology and G as a physical topology, then all links in G_{sub} are reflective. Let M be the reflectively-routed mapping from G_{sub} to G, then $cost(M) = |E(G_{sub})| \leq k$. Under M, any single link failure in G will affect at most one link in G_{sub}. Since G_{sub} is 2-edge-connected, the failure will not disconnect G_{sub}. Therefore, M is a survivable mapping. Hence, G_{sub} is a survivable logical topology that contains $G_\emptyset$ on G such that $cost(G_{sub}) = cost(M) \leq k$.

Suppose there is a survivable logical topology that contains $G_\emptyset$ on G such that its cost is at most k, then we have MIN-COST$_{G_\emptyset} \leq k$. By Theorem 5.2, we can obtain a minimal cost survivable logical topology that contains $G_\emptyset$ on G (denoted as G_{min}) by adding only reflective logical links to $G_\emptyset$. Thus, G_{min} is a spanning subgraph of G. Let M be the reflectively-routed mapping from G_{min} to G. By Theorem 5.2, M is a survivable mapping that achieves the minimal cost, i.e., $cost(M) = |E(G_{min})| =$ MIN-COST$_{G_\emptyset} \leq k$. Therefore, G_{min} is a spanning subgraph of G with at most k edges. Also, G_{min} must be 2-edge-connected because the reflectively-routed mapping M from G_{min} to G is survivable. So, G_{min} is a 2-edge-connected spanning subgraph of G with at most k edges. ∎

5.4 Numerical Results

5.4.1 Simulation Settings

We use two physical topologies (shown in Figure 5.2) in the simulations. The first one is the 14-node 21-link NSFNET and the second one is a 12-node 18-link random graph (referred to as RANDOM). Both physical topologies are 2-edge-connected. For each physical topology, two groups of 2-edge-connected logical topologies, referred to as GROUP1 and GROUP2, are used. For NSFNET, GROUP1 consists of 100 14-node 17-link random topologies and GROUP2 consists of 100 14-node 21-link random topologies. For RANDOM, GROUP1 consists of 100 12-node 15-link random topologies and GROUP2 consists of 100 12-node 18-link random topologies. All simulations are run on a Sun Ultra 10 workstation with a 440MHz CPU, 256MB RAM, and 4GB virtual memory. CPLEX8.1 is used as the ILP solver.

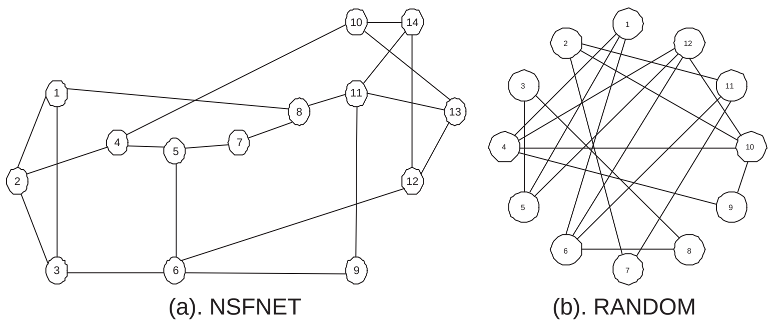


Figure 5.2 Physical topologies used in the simulations.

5.4.2 Significance of the New Survivable Mapping Problem

As discussed in section 5.1, the new survivable mapping problem provides two benefits by allowing logical link addition to the given logical topology. First, a survivable mapping can be obtained for a non-survivable logical topology. Second, the minimal survivable mapping cost may be reduced for a survivable logical topology. To see these benefits, we run ILP2 and the ILP provided in [20] (denoted as ILP_ORIG) on the physical and logical topologies described in the previous subsection. Note that ILP2 solves the new survivable mapping problem that allows adding new logical links to the given logical topology while ILP_ORIG solves the original survivable mapping problem that does not allow the given logical topology to be changed. Thus, given a pair of logical and physical topologies, ILP2 can always find a minimal cost survivable logical topology that contains the given logical topology and the corresponding minimal cost survivable mapping; on the other hand, ILP_ORIG can obtain a minimal cost survivable mapping for the pair only if the given logical topology is survivable. Although both ILP1 and ILP2 solve the new survivable mapping problem, we use ILP2 in the simulations since it runs faster than ILP1.

Table 5.1 shows the improvement made by ILP2 over ILP_ORIG. For GROUP1 over NSFNET, 43 out of 100 logical topologies are not survivable (i.e., ILP_ORIG can't obtain a survivable mapping). However, ILP2 can transform these non-survivable logical topologies into survivable ones by adding new logical links. Among the 57 survivable logical topologies, ILP2 obtains lower cost than ILP_ORIG for 50 (about 88%) of them. That is, 50 surviv-

Table 5.1 Improvement of ILP2 over ILP_ORIG

Physical topology	NSFNET		RANDOM	
Logical topology	GROUP1	GROUP2	GROUP1	GROUP2
# non-survivable logical topologies fixed by ILP2	43	1	53	10
# survivable logical topologies improved by ILP2	50	42	28	28
Maximum(Average) cost saving ratio* among improved survivable logical topologies	20.8% (7.2%)	12.0% (3.5%)	17.4% (7.0%)	10.4% (3.7%)

$*$: cost saving ratio is defined as $\frac{\text{cost computed by ILP\_ORIG} - \text{cost computed by ILP2}}{\text{cost computed by ILP\_ORIG}}$.

able logical topologies can achieve lower survivable mapping cost by adding new logical links. Moreover, among the 50 improved logical topologies, the maximum/average cost saving ratio is 20.8%/7.2%. For GROUP2 over NSFNET, 1 logical topology is not survivable, for which ILP_ORIG can't find a solution while ILP2 can. Among the 99 survivable logical topologies, 42 (about 42%) can achive lower survivable mapping cost by adding new logical links and the maximum/average cost saving ratio is 12.0%/3.5%. These results show that 1) GROUP1 has a larger number of non-survivable logical topologies than GROUP2, 2) among the survivable logical topologies, the percentage of improved ones is larger in GROUP1 than in GROUP2, and 3) the maximum/average cost saving ratio among the improved survivable logical topologies is larger in GROUP1 than in GROUP2. Thus, the overall improvement on GROUP1 is more than on GROUP2. This suggests that the new survivable mapping problem exhibits more significance on sparser logical topologies than on denser ones. This is intuitive because denser logical topologies are generally closer to survivable, and the room to reduce the survivable mapping cost is generally smaller in denser logical topologies. For RANDOM, the results in Table 5.1 also show the benefits of adding logical links to the given logical topology in enabling a survivable mapping and reducing minimal survivable mapping cost. Again, the overall improvement on GROUP1 is more than on GROUP2 since the logical topologies in GROUP1 are sparser than the logical topologies in GROUP2.

5.4.3 Running Time Comparison Between ILP1 and ILP2

To evaluate the running time improvement made by ILP2 over ILP1, we run ILP1 and ILP2 on GROUP1 over NSFNET and GROUP1 over RANDOM. For NSFNET, the average running time of ILP2 over all the 100 logical topologies in GROUP1 is 3505 seconds (about 1 hour), and the running time of ILP1 for a randomly selected logical topology in GROUP1 is 128593 seconds (about 35 hours and 43 minutes). (We did not run ILP1 for all the 100 logical topologies in GROUP1 due to its long running time.) For RANDOM, the average running time over the 100 logical topologies in GROUP1 taken by ILP1 and ILP2 are 544 seconds and 28 seconds respectively. The average speedup of ILP2 over ILP1 is 544sec/28sec $\approx$ 20. For both physical topologies, ILP2 runs much faster than ILP1. As explained in section 5.3-B, this is because ILP2 has fewer variables and fewer flow conservation constraints than ILP1.

5.5 Conclusion

In this chapter, we proposed the following new survivable mapping problem: given a physical topology and a logical topology, compute a minimal cost survivable logical topology that contains the given logical topology and the corresponding minimal cost survivable mapping. The problem is significant for two reasons: 1) If the given logical topology is not survivable, we can add logical links to it to make it survivable; 2) If the given logical topology is survivable, we may reduce the minimal survivable mapping cost by adding logical links to it. We proved that a solution to the new survivable mapping problem always exists and provided a straightforward ILP formulation (ILP1) to solve the problem. Furthermore, we proved that we can find a solution to the new survivable mapping problem by only adding reflective logical links to the given logical topology, and the resulting logical topology has a reflectively-routed survivable mapping that achieves the minimal cost. Based on this result, we developed an improved ILP formulation (ILP2) that solves the new survivable mapping problem more efficiently and proved that the new survivable mapping problem is NP-hard. The benefits of adding logical links to a logical topology in enabling a survivable mapping and reducing minimal survivable mapping cost are demonstrated through simulations. Simulation results also show that ILP2 achieves significant speedup over ILP1.

Appendix

We prove Theorem 5.2 in this section. First, we prove the following lemma.

Lemma *Given a physical topology $G_p = (V, E_p)$, for any survivable logical topology $G_l = (V, E_l)$ on G_p, there exists a set $E' \subseteq E_p - E_l$ such that $G'_l = (V, E_l \cup E')$ has a survivable reflectively-routed mapping M' and $cost(M') \leq cost(G_l)$.*

Proof: Let M be a minimal cost survivable mapping from G_l to G_p. If M is reflectively-routed, just let $E' = \emptyset$. Then $G'_l = G_l$ has a survivable reflectively-routed mapping $M' = M$ and $cost(M') = cost(G_l)$.

If M is not reflectively-routed, we call the procedure TRANSFORM(G_l, G_p, M) to transform M and G_l so that the resulting G_l is obtained by adding links in $E_p - E(G_l^{old})$ to G_l^{old} and the resulting M is a survivable reflectively-routed mapping from G_l to G_p with $cost(M) \leq cost(G_l^{old})$, where G_l^{old} and M^{old} denote the old logical topology and the old mapping inputted to the procedure. The pseudo-code of the procedure TRANSFORM is given below. The correctness proof of the procedure follows the pseudo-code.

```
TRANSFORM($G_l$, $G_p$, $M$)
$G_l$, $M$: inout parameter
$G_p$: in parameter
1.    for each non-reflectively-routed $st \in E_l \cap E_p$ do
2.        let $M(st) = (s - t)$;
3.        REMOVE_CRITICAL_LINK($G_l$, $G_p$, $M$, $st$);
```

```
REMOVE_CRITICAL_LINK($G_l$, $G_p$, $M$, $st$)
$G_l$, $M$: inout parameter
$G_p$, $st$: in parameter
1.    Pick $s' \in V(C_1)$, $t' \in V(C_2)^*$ such that $s't' \in E_p - E_l$;
```

```
2.    if such $s't'$ exists then
3.        let $E_l = E_l \cup \{s't'\}$; $M(s't') = (s' - t')$;
4.    else
5.        Pick $x \in V(C_1)$, $y \in V(C_2)$ such that
          $xy \in E_l \cap E_p$ and $xy \neq st$;
6.        let $M(xy) = (x - y)$;
7.        REMOVE_CRITICAL_LINK($G_l$, $G_p$, $M$, $xy$);
```

*: As stated in **Claim 1** (given later), upon entering procedure REMOVE_CRITICAL_LINK, M is a non-survivable mapping with $st \in E_p$ being the only critical link, whose failure will disconnect G_l into two connected components C_1 and C_2.

We have the following observations about TRANSFORM.

Observation 1: In the resulting G_l, all links in G_l^{old} are kept and the newly added links are all from $E_p - E(G_l^{old})$.

It is easy to verify that there is no logical link deletion anywhere in TRANSFORM and in REMOVE_CRITICAL_LINK. The only place where logical link addition occurs is at line 3 of REMOVE_CRITICAL_LINK and the added logical link belongs to $E_p - E(G_l^{old})$.

Observation 2: The cost of M never increases.

Every time before REMOVE_CRITICAL_LINK is called (at line 2 of TRANSFORM or at line 6 of REMOVE_CRITICAL_LINK), a reflective logical link is rerouted from a multi-hop physical path to a single-hop physical path, which decreases the cost of M by at least 1. On the other hand, within REMOVE_CRITICAL_LINK, at most one logical link is added (at line 3). The added logical link is a reflective logical link and is reflectively-routed, which increases the cost of M by 1. Thus, the cost of M never increases.

In TRANSFORM, if M is a survivable mapping from G_l to G_p at the beginning of an iteration of the for loop, then we have the following two claims.

Claim 1: After line 2 is executed and before REMOVE_CRITICAL_LINK is called, M is a non-survivable mapping from G_l to G_p with $st \in E_p$ being the only critical link. The failure of st will disconnect G_l into two connected components with one component containing s and the other component containing t.

Claim 2: REMOVE_CRITICAL_LINK always returns. Moreover, when it returns, M is a survivable mapping from G_l to G_p.

If Claim 1 and Claim 2 hold, then each iteration of the for loop in TRANSFORM eliminates at least one non-reflectively-routed reflective logical link and end up with a survivable mapping M from G_l to G_p with no new non-reflectively-routed reflective logical link being introduced. (REMOVE_CRITICAL_LINK only adds reflectively-routed reflective logical links to G_l at line 3.) Since there is a finite number of non-reflectively-routed reflective logical links, TRANSFORM always terminates with a survivable reflectively-routed mapping M from G_l to G_p. Together with Observation 1 and Observation 2, we know that TRANSFORM returns G_l and M such that G_l is obtained by adding links in $E_p - E(G_l^{old})$ to G_l^{old}, and M is a reflectively-routed survivable mapping from G_l to G_p with $cost(M) \leq cost(G_l^{old})$. Thus, if we can prove Claim 1 and Claim 2, the proof of Lemma is done. In the following, we give the proofs for Claim 1 and Claim 2.

Proof of Claim 1: Let M^{before}/M^{after} denote the mapping before/after line 2 of TRANSFORM is executed. Assume M^{after} is survivable. We have $cost(M^{after}) < cost(M^{before})$ since st is rerouted from a multi-hop physical path to a single-hop physical path at line 2 of TRANSFORM. This contradicts the fact that M^{before} is a minimal cost survivable mapping from G_l to G_p. Therefore, M^{after} must be non-survivable. Furthermore, $st \in E_p$ must be the only critical link under M^{after} since $st \in E_p$ is the only physical link whose load set expands because of the reroute. (The reroute causes $st \in E_l$ to be included in the load set of $st \in E_p$.) Since $st \in E_p$ becomes a critical link under M^{after} and $st \in E_l$ is the only new logical link added to the load set of $st \in E_p$ due to the reroute, $st \in E_l$ must be a bridge in $G_l^{M^{before}}(st)$. Thus, under M^{after}, the failure of $st \in E_p$ will disconnect G_l into two connected components, one containing s and the other containing t.

(End of Proof of Claim 1)

Proof of Claim 2: The correctness of Claim 2 is based on the following four facts about the procedure REMOVE_CRITICAL_LINK. Each fact is followed by a proof.

Fact 1: At line 5, it is always possible to find x and y that meet the condition. And xy found in line 5 is a non-reflectively-routed reflective logical link.

To enter line 5, we must have

$$\forall s' \in V(C_1), t' \in V(C_2), s't' \in E_p \Rightarrow s't' \in E_l \qquad (*)$$

Because G_p is 2-edge-connected, the edge cut $EC_{G_p}(V(C_1))$ must contain at least one more physical link $xy \neq st$ besides st. By (*), xy must also be in E_l. Therefore, it is always possible to find $x \in V(C_1)$ and $y \in V(C_2)$ such that $xy \in E_l \cap E_p$ and $xy \neq st$.

Since $st \in E_p$ is a critical link when line 5 is executed, we have $EC_{G_l}(V(C_1)) \subseteq l_M(st)$. Since xy is in $EC_{G_l}(V(C_1))$, xy is also in $l_M(st)$. This means that $xy \in E_l \cap E_p$ is routed on $st(\neq xy) \in E_p$. So xy is a non-reflectively-routed reflective logical link.

Fact 2: After line 6 is executed, st is not a critical link. Moreover, xy is the only critical link whose failure will disconnect G_l into two connected components, one containing x and the other containing y.

After line 6 is executed, $xy \in E_l$ is no longer in the load set of $st \in E_p$. And because $x \in V(C_1)$ and $y \in V(C_2)$, the failure of $st \in E_p$ will not disconnect G_l now. So st is not a critical link. However, the mapping becomes non-survivable after line 6 is executed. Let M^{before}/M^{after} denote the mapping before/after line 6 is executed. Assume M^{after} is survivable. We have $cost(M^{after}) < cost(M^{before})$ since xy is rerouted from a multi-hop physical path to a single-hop physical path at line 6. This contradicts the fact that M^{before} is a minimal cost survivable mapping from G_l to G_p. Therefore, M^{after} must be non-survivable. Also, $xy \in E_p$ must be the only critical link under M^{after} because $xy \in E_p$ is the only physical link whose load set expands because of the reroute. (The reroute causes $xy \in E_l$ to be included in the load set of $xy \in E_p$.) Since $xy \in E_p$ becomes a critical link under M^{after} and $xy \in E_l$ is the only new logical link added to the load set of $xy \in E_p$ due to the reroute, $xy \in E_l$ must be a bridge in $G_l^{M^{before}}(xy)$. Thus, under M^{after}, the failure of $xy \in E_p$ will disconnect G_l into two connected components, one containing x and the other containing y.

Fact 3: After line 3 is executed, $s't' \neq st$ and neither $st \in E_p$ nor $s't' \in E_p$ is a critical link.

Before line 3 is executed, $st \in E_l$ and $s't' \notin E_l$, so $s't' \neq st$. After line 3 is executed, the newly added logical link $s't'$ is not routed on st, so the failure of $st \in E_p$ will not affect $s't' \in E_l$. As a result, the remaining logical topology upon the failure of $st \in E_p$ will be connected with $s't' \in E_l$ being a bridge between C_1 and C_2. Thus, $st \in E_p$ is no longer a critical link. As of $s't' \in E_p$, a new logical link $s't' \in E_l$ is added to the load set of $s't' \in E_p$ after line 3 is executed. Assume that $s't' \in E_p$ is critical now, it must be critical also before $s't' \in E_l$ is added to G_l, which contradicts the fact that $st \in E_p$ is the only critical link at that time (this fact is shown by Claim 1 if REMOVE_CRITICAL_LINK is called from line 3 of TRANSFORM, and by Fact 2 if REMOVE_CRITICAL_LINK is called from line 7 of itself). Thus, $s't' \in E_p$ is not a critical link after $s't' \in E_l$ is added to G_l.

Fact 4: After $st \in E_p$ becomes non-critical in REMOVE_CRITICAL_LINK, it will never become critical again till the end of TRANSFORM. Also, for each logical link $s't'$ added to G_l at line 3 of REMOVE_CRITICAL_LINK, the corresponding physical link $s't'$ will never become critical again either.

After $st \in E_p$ becomes non-critical in REMOVE_CRITICAL_LINK, $st \in E_l$ is reflectively-routed. The load set of $st \in E_p$ will never include other logical links till the end of TRANSFORM because

(1) All new logical links added at line 3 of REMOVE_CRITICAL_LINK will be reflectively-routed.

(2) We only reroute non-reflectively-routed reflective logical links to make them reflectively-routed (at line 2 of TRANSFORM and at line 6 of REMOVE_CRITICAL_LINK).

So $st \in E_p$ will never become critical again.

Because of the same reasons, $s't' \in E_p$ will never become critical again either.

During the execution of REMOVE_CRITICAL_LINK, if the "then" branch is entered, Fact 3 tells us that $st \in E_p$ will become non-critical, and for the newly added logical link $s't' \in E_l$, the corresponding $s't' \in E_p$ is not critical either. On the other hand, if the "else" branch is entered, Fact 1 and Fact 2 tell us that st will become non-critical and another physical link xy will become critical. Thus, REMOVE_CRITICAL_LINK always eliminates one critical link (st) and may introduce another critical link (xy). Fact 4 guarantees that once st becomes non-critical, it will never become critical again. Also, for each logical link $s't'$ added to the logical topology, the corresponding $s't' \in E_p$ will never become critical again either. Since we have a finite number of physical links that are potential critical links, REMOVE_CRITICAL_LINK will always return with no critical link existing in G_p. Therefore, when REMOVE_CRITICAL_LINK returns, M is a survivable mapping from G_l to G_p.

(End of Proof of Claim 2)

As argued earlier, Observations 1 and 2 together with Claims 1 and 2 prove Lemma. ∎

We now give the proof of Theorem 5.2.

Theorem 5.2 *Given a logical topology $G_l = (V, E_l)$ and a 2-edge-connected physical topology $G_p = (V, E_p)$, there exists an edge set $E'' \subseteq E_p - E_l$ such that $G''_l = (V, E_l \cup E'')$ is a minimal cost survivable logical topology that contains G_l on G_p. Moreover, there is a reflectively-routed survivable mapping M'' from G''_l to G_p such that $cost(M'') = MIN\text{-}COST_{G_l}$.*

Proof: Let $G = (V, E)$ be a minimal cost survivable logical topology that contains G_l on G_p. Let M be a minimal cost survivable mapping from G to G_p. Then $cost(M) = cost(G) =$ MIN-COST$_{G_l}$.

CASE I: All logical links in $E - E_l$ are reflective, i.e., $E - E_l \subseteq E_p - E_l$.

If M is a reflectively-routed mapping, then $E'' = E - E_l$, $G''_l = G$, and $M'' = M$ are the edge set, the logical topology, and the mapping we are looking for.

If M is not a reflectively-routed mapping from G to G_p, then by Lemma, there exists $E' \subseteq E_p - E$ such that $G' = (V, E \cup E')$ has a survivable reflectively-routed mapping M' from G' to G_p and $cost(M') = cost(M)$. (It is impossible to get $cost(M') < cost(M)$ since $cost(M) =$ MIN-COST$_{G_l}$.) Then $E'' = (E \cup E') - E_l$, $G''_l = G'$, and $M'' = M'$ are the edge set, the logical topology, and the mapping we are looking for.

CASE II: At least one logical link in $E - E_l$ is non-reflective, i.e., $\exists st \in E - E_l$ such that $st \notin E_p$.

In this case, we call the procedure PURIFY(G_l, G_p, G, M) to transform G and M so that the resulting G is a minimal cost survivable logical topology that contains G_l on G_p and

$E - E_l \subseteq E_p - E_l$. And the resulting M is a reflectively-routed survivable mapping from G to G_p with $cost(M) =$ MIN-COST$_{G_l}$. The pseudo-code of PURIFY is given below. The correctness proof of PURIFY follows the pseudo-code.

```
PURIFY($G_l$, $G_p$, $G$, $M$)
$G_l$, $G_p$: in parameter
$G$, $M$: inout parameter
1.    if $M$ is not reflectively-routed then
2.        Find $G' = (V, E \cup E')$ and $M'$ such that
          $E' \subseteq E_p - E$ and $M'$ is a reflectively-routed
          survivable mapping from $G'$ to $G_p$
          and $cost(M') = cost(M)$;
3.        let $G = G'$; $M = M'$;
4.    for each $st \in (E - E_l) - E_p$ do
5.        let $E = E - \{st\}$;
6.        for each $ij \in M(st)$ do
7.            let $E = E \cup \{ij\}$;
8.            let $M(ij) = (i - j)$;
```

We have the following observations about PURIFY.

Observation 3: When PURIFY returns, G contains all logical links in G_l and all logical links in $E - E_l$ are reflective.

In PURIFY, logical link removal only occurs at line 5, where $st \in (E - E_l) - E_p$ is removed. Thus, all logical links in G_l are kept in G. In the for loop from line 4 to line 8, each non-reflective logical link $st \in E - E_l$ is removed and replaced by a set of reflective logical links corresponding to the physical links in $M(st)$. Therefore, all logical links in $E - E_l$ are reflective.

Observation 4: The cost of M never increases or decreases.

Clearly, the cost of M cannot decrease because $cost(M) =$ MIN-COST$_{G_l}$ when M is inputted to PURIFY.

We now show that the cost of M never increases. If the mapping M inputted to PURIFY is not a reflectively-routed mapping, then line 2 is executed. By Lemma, we can successfully find G' and M' at line 2 and the cost of the mapping does not increase. In each iteration of the for loop at line 4, on the one hand, $st \in (E - E_l) - E_p$ is removed from G, which decreases the cost of M by $|M(st)|$; on the other hand, at most $|M(st)|$ reflectively-routed reflective logical links are added to G, which increases the cost of M by at most $|M(st)|$. Thus, the cost of M does not increase in the for loop. Overall, the cost of M never increases in PURIFY.

Within one iteration of the for loop at line 4 in PURIFY, we use G^{before}/M^{before} and G^{after}/M^{after} to denote the logical topology/mapping *before* removing st and *after* adding ij's in $M(st)$ and mapping them reflectively. For each iteration of the for loop, if M^{before} is a survivable mapping, then we have the following two claims.

Claim 3: $\forall ij \in M^{before}(st)$, ij is not a critical link under M^{after}.

Claim 4: $\forall ij \in E_p - M^{before}(st)$, ij is not a critical link under M^{after}.

If Claim 3 and Claim 4 hold, each iteration of the for loop at line 4 eliminates exactly one non-reflective logical link in $E - E_l$ without breaking the survivability of the mapping or introducing any new non-reflective logical links. Since we have a finite number of non-reflective logical links in $E - E_l$, PURIFY always terminates with M being a survivable mapping from G to G_p. In addition, M is a reflectively-routed mapping when PURIFY terminates. This is because M is a reflectively-routed mapping before the for loop is executed and all the logical links added in the for loop are reflective and are reflectively-routed. By Observation 3 and Observation 4, the logical topology $G = (V, E)$ returned by PURIFY is a minimal cost survivable logical topology that contains G_l on G_p and $E - E_l \subseteq E_p - E_l$, and the mapping M returned by PURIFY is a reflectively-routed survivable mapping from G to G_p with $cost(M) = \text{MIN-COST}_{G_l}$. So if we can prove Claim 3 and Claim 4, the proof of Theorem 5.2 is done.

In the following, we give the proofs for Claim 3 and Claim 4.

Proof of Claim 3: For all $ij \in M^{before}(st)$, we have $l_{M^{after}}(ij) = (l_{M^{before}}(ij) - \{st\}) \cup \{ij\}$. Assume that $ij \in E_p$ is critical under M^{after}, then the failure of $ij \in E_p$ will disconnect

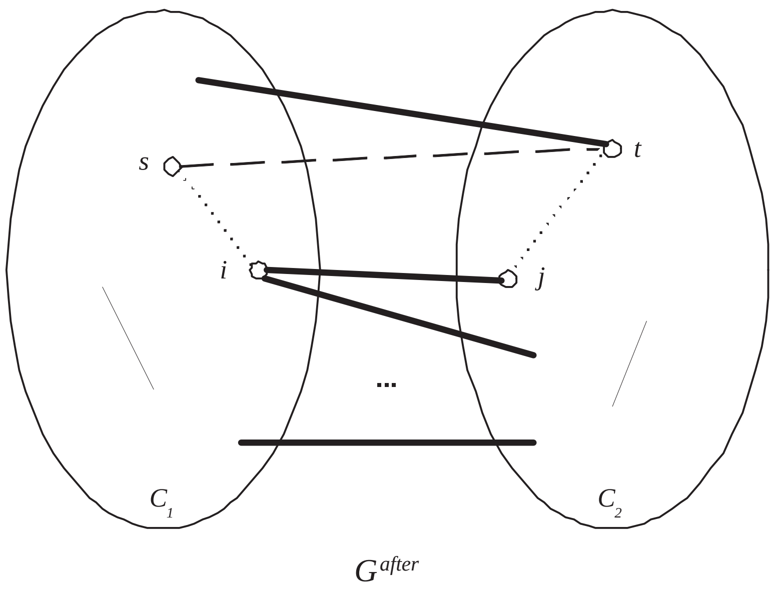


Figure 5.3 Illustration for the Proof of Claim 3. Solid (thin and thick) lines are logical links in $l_{M^{after}(ij)}$. Thick solid lines are logical links in $EC_{G^{after}}(V(C_1))$. The dashed line is $st \in E(G^{before})$. The dotted lines denote the *logical* path from s to i and the *logical* path from j to t in G^{after}. This diagram shows that if the removal of $l_{M^{after}}(ij)$ disconnects G^{after}, then the removal of $l_{M^{before}}(ij) = (l_{M^{after}}(ij) - \{ij\}) \cup \{st\}$ would also disconnect G^{before}.

the G^{after} into two connected components C_1 and C_2, one containing i and the other containing j. This is because $ij \in E_p$ is not critical under M^{before} and $ij \in E(G^{after})$ is the only logical link that is contained in $l_{M_{after}}(ij)$ but not in $l_{M_{before}}(ij)$.

On the other hand, all logical links in $E(G^{after})$ except ij along the *logical* path corresponding to $M^{before}(st)$ are reflectively-routed, so these logical links will not be affected by the failure of $ij \in E_p$ under M^{after}. In other words, only $ij \in E(G^{after})$ is broken on the *logical* path corresponding to $M^{before}(st)$ when $ij \in E_p$ fails. Without loss of generality, suppose $s, i \in V(C_1)$ and $t, j \in V(C_2)$. As can be seen from Figure 5.3, if $ij \in E_p$ is critical under M^{after}, $ij \in E_p$ must be critical under M^{before} because removing $l_{M^{before}}(ij) = (l_{M^{after}}(ij) - \{ij\}) \cup \{st\}$ from G^{before} would disconnect G^{before}. This contradicts the fact that M^{before} is survivable. Thus, $ij \in E_p$ is not critical under M^{after}.

(End of Proof of Claim 3)

Proof of Claim 4: Consider $ij \in E_p - M^{before}(st)$. Assume ij is critical under M^{after}. Since the load set of ij under M^{after} is the same as that under M^{before}, the only possible reason that makes ij critical under M^{after} is the loss of $st \in E(G^{before})$ in G^{after}. So, the failure of ij will disconnect G^{after} into two connected components, one containing s and the other containing t. However, this is impossible because there exists a path in G^{after} from s to t when ij fails since all the logical links along the logical path corresponding to $M^{before}(st)$ are reflectively-routed under M^{after} and therefore not affected by the failure of ij. Thus, ij is not a critical link under M^{after}.

(End of Proof of Claim 4)

As argued earlier, Observations 3 and 4 together with Claims 3 and 4 prove Theorem 5.2.

∎

CHAPTER 6. OVERVIEW OF P-CYCLE DESIGN

p-Cycle is a promising approach for survivable WDM network design because of its ability to achieve ring-like recovery speed while maintaining the capacity efficiency of a mesh-restorable network [2]. A p-cycle is a pre-configured cycle formed out of the spare capacity in the network, which occupies one unit of spare capacity on each on-cycle span (from this chapter, a "span" is equivalent to a "link"). Like a self-healing ring, a p-cycle provides one restoration path for every on-cycle span; unlike a self-healing ring, a p-cycle also provides two restoration paths for every *straddling span* – a span whose two end nodes are on the cycle but itself is not on the cycle. As shown in Figure 6.1, $a - b - c - d - f - a$ is a p-cycle. For the on-cycle span $a - b$, the p-cycle provides one restoration path $a - f - d - c - b$. For the straddling span $f - b$, the p-cycle provides two restoration paths: $f - a - b - c$ and $f - d - c$. Thus, a p-cycle can protect one unit of working capacity on every on-cycle span and protect two units of working capacity on every straddling span.

p-Cycle design is considered as a promising protection scheme since it combines the advantages of the link-based protection and path-based protection. In link-based protection, fast restoration is achieved because when a link fails, both end nodes will detect the failure immediately and initiate the restoration of the traffic on this link onto a pre-computed back-up path right away. While in path protection, the backup paths are configured between source destination node pairs. When a link failure happens, both end nodes of this link will have to notify each affected flow's source and/or destination so that they can start to redirect their traffic onto corresponding backup paths. This results in longer restoration time compared to link-based protection. On the other hand, path-based protection is more cost-effective than link-based protection in terms of spare capacity consumption [32]. p-Cycle achieves fast restoration like

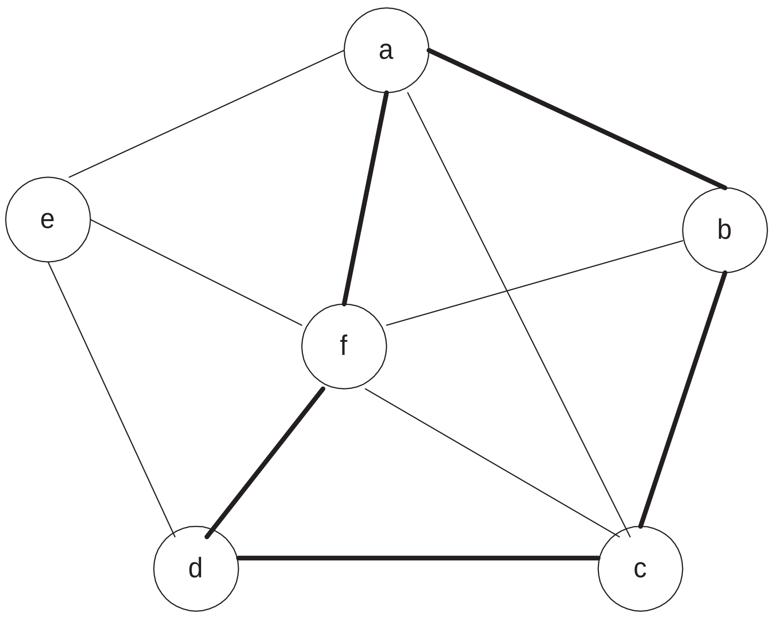


Figure 6.1 A p-cycle example

link-based protection since the end nodes of a failed link would reroute the interrupted traffic along the protection path(s) provided by pre-configured *p*-cycles right after the failure is detected. Moreover, *p*-cycle protection also yields efficient spare capacity usage because *p*-cycle offers two restoration paths for a straddling span.

CHAPTER 7. FINDING GOOD CANDIDATE CYCLES

7.1 Optimal p-Cycle Protection Design and Heuristics

The problem of finding an optimal set of p-cycles to protect a given set of traffic demands is a basic design issue. Two versions of the optimization problem have been studied: non-joint version and joint version. In the non-joint version, working capacity on every span is given (i.e., routing of the working paths for the demands are known) and the objective is to find a set of p-cycles to protect the working capacity so that the total spare capacity used by the p-cycles is minimized [2][33][34][35]. In the joint version, the routing of the working paths for the demands and the p-cycles are computed jointly so that the total capacity (i.e., working capacity plus spare capacity) is minimized [36][37][38]. A common approach for solving the optimization problems is Integer Linear Programming (ILP). In this approach, a set of candidate p-cycles is precomputed and supplied to an ILP formulated to find the optimal set of p-cycles out of the candidate p-cycles. The ILP will give the optimal solution when the candidate p-cycle set includes all the cycles in the network. However, since the number of cycles in a network grows exponentially with the network size, various methods have been proposed to reduce the size of the candidate p-cycle set. One method is to limit the maximal length of the candidate p-cycles[2][33]. Another method defines a metric called *a priori efficiency (AE)* for each cycle. $AE(p) = \frac{\sum_{i \in S} X_{p,i}}{\sum_{i \in S, X_{p,i}=1} c_i}$ where S is the set of spans in the network, c_i is the cost of a unit capacity on span i, and $X_{p,i}$ is the number of restoration paths that cycle p can provide for span i. $X_{p,i} = 1$ if i is an on-cycle span, $X_{p,i} = 2$ if i is a straddling span, $X_{p,i} = 0$ otherwise [36]. Assuming $c_i = 1$ for all $i \in S$, then the AE metric of the p-cycle shown in Figure 6.1 is $\frac{2*3+5}{5} = 2.2$ since it has 3 straddling spans and 5 on-cycle spans. then the first k cycles with the highest AE score are chosen as the candidate p-cycles where k is an adjustable parameter[36]. While both methods can reduce the number of candidate p-cycles, they still require the enumeration of all cycles in the network. To address this problem, an algorithm called SLA that generates a set of candidate p-cycles without enumerating all cycles was proposed in [39]. The idea is to generate one p-cycle for each span in the network so that the span is a straddling span of the p-cycle.

On the other hand, since the running time of ILP for large networks is prohibitively long, a heuristic algorithm for the non-joint version of the p-cycle optimization problem called CIDA was proposed in [34]. In CIDA, a set of candidate p-cycles is computed first, then one p-cycle is chosen iteratively from the candidate p-cycle set and placed in the network to reduce the unprotected working capacity until all working capacities are protected. In each iteration, the p-cycle with the highest *actual efficiency* is selected. The actual efficiency for a p-cycle p is defined as $E_w(p) = \frac{\sum_{i \in S} w_i * X_{p,i}}{\sum_{i \in S, X_{p,i}=1} c_i}$ where w_i is the amount of unprotected working capacity on span i. Unlike a priori efficiency, actual efficiency depends not only on the number of on-cycle and straddling spans, but also on the unprotected working capacity of those spans. Three algorithms (SP-Add, Expand, and Grow) were proposed in [34] to generate candidate p-cycles for CIDA. All three algorithms start with the set of primary cycles generated by SLA and then create more cycles from each primary cycle by replacing an on-cycle span by a path between the end nodes of the span, which converts the on-cycle span to a straddling one. The p-cycles generated by the three algorithms have higher average a priori efficiency than the p-cycles generated by SLA, which lead to better performance when used by CIDA. On the other hand, the three algorithms generate more candidate p-cycles than SLA does: while SLA generates only $O(m)$ p-cycles, SP-Add and Expand generate $O(m^2)$ p-cycles and Grow generates $O(m^2n)$ p-cycles where m and n are the number of spans and nodes in the network respectively.

For both the ILP approach and the heuristic approach, the quality of the solution depends on the candidate p-cycle set used. A good candidate p-cycle set should contain a small number of candidate p-cycles while give good performance (i.e., produce near optimal solutions) when used by ILP or the heuristic algorithm. Among existing candidate p-cycle generation algorithms, SLA generates the least number of p-cycles ($O(m)$). However, the candidate p-cycles generated by SLA do not bring good performance since they usually contain no more than one straddling span and thus have low a priori efficiency [34]. In this chapter, we propose a new algorithm that generates $O(m)$ candidate p-cycles with good performance. Unlike existing algorithms that generate a fixed number of candidate p-cycles, the new algorithm can control the number of candidate p-cycles generated by adjusting an input parameter, which in term affect the performance of the generated candidate p-cycles.

This chapter is organized as follows. In section 7.2, we describe the new cycle generation algorithm. We study the performance of the algorithm in section 7.3. A conclusion is given in section 7.4.

7.2 The Candidate Cycle Generation Algorithm

7.2.1 Design Considerations

Our candidate cycle generation algorithm is designed based on three considerations.

1) In order for the network to survive any single span failure, the candidate cycles must be able to protect all spans in the network. That is, each span must be an on-cycle span or a straddling span of some candidate cycle.

2) Since we would like to minimize the spare capacity required for protecting a given working capacity distribution, the candidate cycle set should contain cycles with high efficiency. (In the rest of the chapter, efficiency means a priori efficiency.)

3) Short candidate cycles are needed in order to achieve good performance. The reason is the following. After we apply some high efficiency p-cycles to protect the working capacities, we may be left with only a few spans with unprotected working capacities. In this case, a p-cycle with high efficiency would be a bad choice since many of the on-cycle spans and/or straddling spans on this cycle would have no working capacity left unprotected so that the potential benefits of them would be wasted. Take the example in Figure 6.1, suppose after applying a number of high efficiency p-cycles to protect the working capacities only two units of working capacities are left unprotected, one on span $b-f$ and one on span $a-f$. Now it is not efficient to use the p-cycle $a-b-c-d-f-a$ to protect the unprotected working

capacities although it has high efficiency because it will use five units of spare capacity just for protecting two units of working capacity. In this case, the short p-cycle $a - b - f$ (with an efficiency of 1) is the best choice since it can protect two units of working capacity with only three units of spare capacity. This example shows that it is not always desirable to use high efficiency p-cycles to protect the working capacities. In general, when most of the spans contain unprotected working capacities, p-cycles with high efficiency are preferred since the benefit of the on-cycle and straddling spans can be well utilized; when the unprotected working capacity of most spans have dropped to 0, short p-cycles are preferred since they can reduce the unprotected working capacities with low spare capacity consumption. Thus, a good candidate cycle set should contain both high efficiency cycles and short cycles.

Based on the above considerations, our algorithm is designed to generate a combination of high efficiency cycles and short cycles so that every span in the network is protected by at least one high efficiency cycle and one short cycle. The algorithm consists of two steps: in the first step, it generates a set of high efficiency cycles and in the second step, it computes a set of short cycles.

7.2.2 Notations

Before describing the two steps in detail, we first introduce some notations. A network is modeled as an undirected graph $G = (V, E)$ where each vertex $v \in V$ represents a network node and each undirected edge $e \in E$ represents a network span. An undirected edge consists of two directed edges. We use unordered pair (u, v) to represent an undirected edge between u and v and ordered pair $\langle u, v \rangle$ to represent a directed edge from u to v. Thus an undirected edge (u, v) consists of two directed edges $\langle u, v \rangle$ and $\langle v, u \rangle$. The set of neighbors of a vertex v is denoted as $N(v)$, i.e., $N(v) = \{u | (u, v) \in E\}$. A DFS path from s to v, written as $s \longrightarrow v$, is the path from s to v taken by a Depth First Search (DFS) traversal starting from s. We assign weights to the directed edges in the graph before conducting a DFS search and the weight of a directed edge $\langle u, v \rangle$ is denoted as $weight(u, v)$.

7.2.3 Step One: Generating High Efficiency Cycles

The key component of step one is the Weighted DFS-Based Cycle Search algorithm implemented as a procedure WDCS(s, v, k). The procedure finds k cycles starting from s as the root and traversing the DFS path $s \longrightarrow v$ if k is smaller than the total number of cycles traversing the DFS path $s \longrightarrow v$, otherwise the procedure will find all cycles traversing the DFS path $s \longrightarrow v$. WDCS is based on Johnson's cycle enumeration algorithm [40], which uses DFS to generate all cycles in a graph. A procedure Cycle(s, v) based on Johnson's algorithm was provided by Grover in [41], which finds all cycles starting from s as the root and traversing the DFS path $s \longrightarrow v$. The key difference between our procedure WDCS(s, v, k) and the procedure Cycle(s, v) is that in WDCS the order of cycle search is controlled by assigning weights to the directed edges so that high efficiency cycles are likely to be found early in the search. In addition, the number of cycles generated by WDCS is controlled by the input parameter k. As a special case, when $k = \infty$, the procedure will find all cycles traversing the DFS path $s \longrightarrow v$.

The pseudocode of WDCS is given below.

```
bool WDCS($s$, $v$, $k$)
1.   bool flag := false;
2.   avail($v$) := false;
3.   time_stamp($v$) := counter; counter++;
4.   while (number_cycles_found $< k$)
5.       find $w \in N(v)$ such that avail($w$) $=$ true and
         $weight(v, w) = \max_u weight(v, u)$;
6.       if (such $w$ exists) then
7.           if ($w = s$) then // a cycle is found
8.               cycle := stack contents followed by $s$;
9.               output cycle;
10.              flag := true;
11.              number_cycles_found++;
```

```
12.             else // extend the searching path
13.                 push $w$ onto the stack;
14.                 flag := flag $OR$ WDCS($s$, $w$, $k$);
15.         else // the searching path hits a dead end
16.             break;
17.     if (number_cycles_found $< k$) then
18.         if (flag = true) then
19.             Unmark($v$)
20.         else
21.             for each $x \in N(v)$ do
22.                 if (time_stamp($x$) $<$ time_stamp($v$)) then
                        // $x$ is a DFS ancestor of $v$
23.                     B($x$) := B($x$) $\cup \{v\}$
24.     pop $v$ from the stack;
25.     return flag;
```

```
void Unmark($u$)
1.      avail($u$) := true;
2.      for each $t \in$ B($u$) do
3.          Unmark($t$);
4.      B($u$) := $\emptyset$
```

Before WDCS(s, v, k) is called by another procedure, the following procedure is used to initialize the necessary data structures and variables.

```
void init($s$, $v$)
1.      empty the stack;
2.      push $s$ onto the stack;
```

```
3.    push $v$ onto the stack;
4.    counter := 0; number_cycles_found := 0;
5.    for each $u \in V$ do
6.        avail($u$) := true;
7.        B($u$) := $\emptyset$;
8.        time_stamp($u$) = $\infty$;
```

The basic cycle search strategy used by WDCS is the following. The DFS starts from the root s. First, v is included into the DFS path. Whenever a vertex is added to the DFS path, it is marked unavailable and pushed onto the stack that contains the current DFS path. We keep extending the DFS path until the root s is reached, which means that a cycle is found, or, there is no available vertex to extend the path, which means that a cycle can't be found. Then we back up one vertex by popping the top vertex off the stack and go for another search from the vertex preceding the popped vertex. If the previous search succeeded in finding a cycle, the popped vertex as well as its marked DFS descendants should be unmarked as available for exploring new cycles; if the previous search failed to find a cycle, the popped vertex as well as its DFS descendants should stay unavailable to avoid future search running into the unsuccessful path again. For more details about the DFS-based cycle search algorithm, please refer to [40] and [41].

We now describe the new features we added to the basic DFS-based cycle search algorithm. In the basic algorithm, when extending the search path from a node v, the neighbors of v are explored in arbitrary order. Thus there is no control over the order of the cycles generated. In WDCS, we added some intelligence into the DFS search so that cycles with high efficiency are likely to appear early in the DFS search. As shown in line 5 of WDCS(s, v, k), instead of picking any available neighbor of v to extend the searching path, we choose an available neighbor w such that $\langle v, w \rangle$ has the highest weight among all outgoing edges incident to v. This makes it possible for us to have control over the order by which vertices are included into the search path (hence the order by which cycles are found) by properly setting the edge weights. The following three rules are used to set the weights of the directed edges.

1) Since long cycles tend to include more straddling spans and therefore result in high efficiency, we would like to avoid going back to the root vertex during the DFS search whenever possible. This is achieved by assigning a small weight to the directed edges that end at the root, i.e. we set $weight(u, s) = \epsilon$ for all $u \in N(s)$ where $0 < \epsilon < 1$ and s is the root.

2) When choosing an available vertex among the neighbors of the current searching path endpoint, we would like to pick the neighbor with the highest degree. The reason is that if this high degree vertex is included into the searching path that results in a cycle, the cycle will have high probability of including more straddling edges. This strategy can be achieved by setting the weight of $\langle u, v\rangle$ as the degree of v for all $\langle u, v\rangle$ where $v \neq s$.

3) According to rule 2, 2-degree vertices are not desirable because they do not introduce straddling edges directly into a cycle and they will not be selected to extend a searching path whenever another higher degree vertex is available. However, we note that 2-degree vertices are actually desirable in some cases. Consider the graph shown in Figure 7.1. According to rule 1 and rule 2, the edge weights would be set as shown in Figure 7.1 (a) ($\epsilon = 0.9$) and the cycle found by WDCS(r, v_1, 1) would be $r - v_1 - v_2 - r$ with an efficiency of 1. However, the cycle $r - v_1 - v - v_2 - r$ has a higher efficiency of 1.5, which is better than the cycle $r - v_1 - v_2 - r$. In order for the DFS search to discover the higher efficiency cycle first, we need special treatment for the 2-degree vertex v. We observe that when v is included into the searching path from v_1, the only way for the search to continue is to extend the path to v_2 from v. So when the current searching path endpoint is v_1, whether the 2-degree vertex v is a desirable vertex to be included in the path depends on the desirability (i.e., degree) of v_2 since the path $v_1 - v - v_2$ can be viewed as a single edge $\langle v_1, v_2\rangle$. Thus we set $weight(v_1, v) = degree(v_2)$. Now both $\langle v_1, v\rangle$ and $\langle v_1, v_2\rangle$ have a weight of 3. To break the tie, we give higher priority to $\langle v_1, v\rangle$ by adding a small $\epsilon'(0 < \epsilon' < 1)$ to its weight. With this special treatment of the 2-degree vertex, the weights of the edges are shown in Figure 7.1 (b). In this case, v would be chosen over v_2 when extending the searching path from v_1. Thus, the higher efficiency cycle $r - v_1 - v - v_2 - r$ would be found by WDCS(r, v_1, 1). The special treatment of 2-degree vertex can be generalized to

handle 2-degree chains. We call a path from u to w a 2-degree chain if both u and w have degree greater than 2 and all the other intermediate vertices on the path have degree 2. Such a path can be regarded as one edge from u to w by ignoring all the intermediate 2-degree vertices, so we set the weight of all edges on the path to be the degree of w and for the first edge on the path an extra ϵ' is added to its weight. Note that when we include a 2-degree chain into a cycle, we may introduce one more straddling edge into the cycle while at the same time increase the cycle length and therefore the cycle cost. As a result, the efficiency of the cycle may or may not be improved. However, we observe that real world transport networks usually only contain short 2-degree chains (i.e., only one or two 2-degree vertices are on the chain), so the benefit of introducing one more straddling edge generally outweighs the negative effect of increased cycle length.

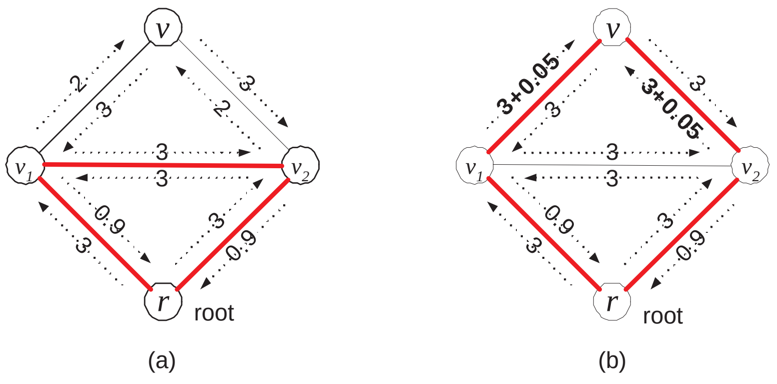


Figure 7.1 Illustration of the special handling of the degree 2 vertex. Numbers above the dotted lines are weights. Here we let $\epsilon = 0.9$ and $\epsilon' = 0.05$. (a) Without special handling of 2-degree vertex. (b) With special handling of 2-degree vertex.

The following is the pseudo-code of the procedure Set_Weights, which sets the edge weights based on the three rules.

```
void Set_Weights($s$) //$s$ is the root of DFS cycle search
1.    for each $\langle u, v \rangle \in E$ do
2.        if $(v \neq s)$ then
3.            $weight(u, v) := degree(v)$
```

```
4.           else
5.               $weight(u, v) := \epsilon$ $(0 < \epsilon < 1)$
6.       for each 2-degree chain in $G$ do
7.           set the weight of all edges on the chain to the degree
             of the last vertex on the chain
8.           add $\epsilon'$ to the weight of the first edge on the chain;
```

To generate a set of high efficiency cycles that can protect every edge in the graph, we define a procedure Efficient_Cycles(k) that calls WDCS $2|E|$ times to generate $2k|E|$ cycles in total. Specifically, WDCS is called for every vertex $u \in V$ and every neighbor v of u to generate k cycles. Thus, every directed edge $\langle u, v\rangle$ is used once to start the search for k cycles and exactly $2k|E|$ cycles are generated. The pseudo-code of the procedure is given below.

```
void Efficient_Cycles($k$)
1.       for each $u \in V$ do
2.           Set_Weights($u$);
3.           for each $v \in N(u)$ do
4.               init($u$, $v$);
5.               WDCS($u$, $v$, $k$);
```

Note that k is an input parameter for the procedure, which is a constant. So the number of high efficiency cycles generated in this step is $O(|E|)$. Also, the number of *unique* cycles generated by the procedure may be smaller than $2k|E|$ because of the existence of duplicate cycles.

7.2.4 Step Two: Generating Short Cycles

As discussed earlier in this section, short cycles are needed to deal with sparse working capacity situation efficiently. To this end, we compute two short cycles for each edge in the graph, one has the edge as an on-cycle edge, the other has the edge as a straddling edge. For each edge e, we compute the two short cycles as follows. We find the shortest path between the endpoints of e in $G - e$ and combine the path with e to create a cycle. Note that this is the shortest cycle that can protect e and e is on-cycle. To generate a short cycle that has e as a straddling edge, we first find the shortest path P_1 between the two endpoints of e in $G - e$. We then tries to find the shortest path P_2 between the two endpoints of e in $G - e$ that is node-disjoint with P_1. If P_2 can be found, then it is combined with P_1 to form a cycle. Otherwise, no cycle exists that has e as a straddling edge. Clearly, for each edge in G, the algorithm generates exactly one short cycle with the edge being on-cycle and at most one short cycle with the edge being straddling, so the total number of short cycles generated in this step is at most $2|E|$.

Since both step 1 and step 2 generate $O(|E|)$ cycles, a total of $O(|E|)$ cycles are generated by our cycle generation algorithm.

7.3 Numerical Results

We used two test networks shown in Figure 7.2 to evaluate the performance of our cycle generation algorithm. Network1 is a 11-node 23-span network taken from the website of [41] and Network2 is a 28-node 45-span network taken from [34].

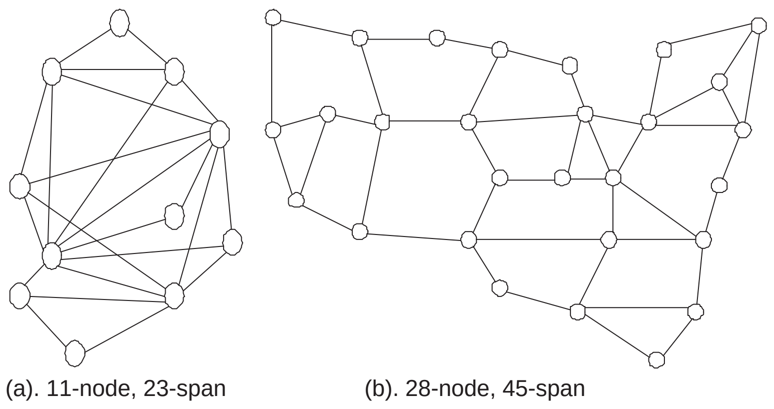


Figure 7.2 Two test networks: (a) Network1; (b) Network2.

For each test network, two groups of demand sets are used. One is the uniform demand set that contains one demand for each unordered source-destination pair; the other is 10 random demand sets of which each random demand set contains 150 demands (for Network1) and 1000 demands (for Network2) respectively. In all demand sets, each demand requests for one unit of capacity. For each combination of a test network and a demand set, working capacities on the network spans are obtained by routing each demand over the shortest path.

7.3.1 Spare Capacity Efficiency When $k = 1$

We first evaluate the performance of our cycle generation algorithm with k set to 1 so that the least number of candidate cycles are generated. After the candidate cycles are generated, we feed them to the heuristic algorithm CIDA provided in [34] and the ILP formulation given in [41] to compute a set of p-cycles to protect the working capacities and record the spare capacity required. To obtain the minimal spare capacity required by the optimal p-cycle configuration, we also run the ILP with all cycles in the network supplied to it. The results for the two test networks are shown in Table 7.1 and Table 7.2. In these tables, "CIDA on CAND" means CIDA with candidate cycles generated by our algorithm, "ILP on CAND" means ILP with candidate cycles generated by our algorithm, and "ILP on ALL" means ILP with all cycles in the network, which gives the optimal solution. For each of these three cases, the number of candidate cycles and the percentage difference from the optimal spare capacity are shown.

Table 7.1 Comparison of spare capacity efficiency for Network1

	Uniform demands			150 random demands		
	CIDA on CAND	ILP on CAND	ILP on ALL	CIDA on CAND	ILP on CAND	ILP on ALL
# Candidate Cycles	49	49	307	49	49	307
%Diff	16.3%	12.8%	0%	15.3%	13.8%	0%

Note that in Table 7.1 and 7.2 (as well as in the next section), the percentage difference for random demands is the average percentage difference of the corresponding group of 10 random demand sets.

Table 7.2 Comparison of spare capacity efficiency for Network2

	Uniform demands			1000 random demands		
	CIDA on CAND	ILP on CAND	ILP on ALL	CIDA on CAND	ILP on CAND	ILP on ALL
# Candidate Cycles	92	92	7321	92	92	7321
%Diff	21.9%	17.0%	0%	21.8%	17.4%	0%

The quality of the candidate cycles generated by our algorithm can be judged by %Diff of ILP on CAND. For Network1, the %Diff is 12.8% and 13.8% for uniform demands and random demands respectively. This is quite promising because only 49 out of the 307 cycles in the network are used as candidate cycles. For Network2, the %Diff is 17.0% and 17.4% for uniform demands and random demands respectively. Here only 92 out of the 7321 cycles in the network are used as candidate cycles. These results show that our algorithm can generate a small number of candidate cycles with good performance when k is set to 1.

7.3.2 Effect of k on Spare Capacity Efficiency

Figure 7.3 shows how %Diff of ILP on CAND changes as we change the value of k and Figure 7.4 shows how the number of candidate cycles generated by our algorithm grows as k is increased. As can be seen from the figures, as k increases, the number of candidate cycles increases and %Diff decreases. This is expected because increasing k value will make more candidate cycles to be generated, which in turn leads to better ILP solution. For Network1, when k is increased to 5, the number of candidate cycles is 130 (42.3% of all cycles), the spare capacity used is optimal for the uniform demand set and within 0.8% from the optimal for random demand sets. For Network2, in order to get a solution within 1% from optimal, a much larger k is needed: when k is increased to 43 (not shown in Figure 7.3 and Figure 7.4), the number of candidate cycles is 1714 (23.4% of all cycles), and the spare capacity used is within 0.7% and 0.6% from optimal for the uniform demand set and random demand set respectively. These results show that with our algorithm, almost optimal spare capacity consumption can be achieved with much less candidate cycles than all cycles in the network. In addition, by adjusting the k value, we can obtain a tradeoff between the number of candidate cycles and the spare capacity efficiency of the p-cycle network design.

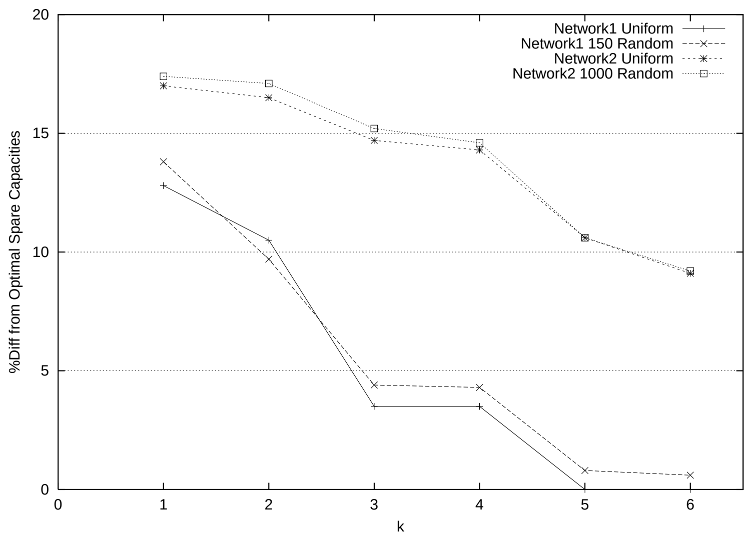


Figure 7.3 Effect of k on Spare Capacity Efficiency of ILP on CAND.

7.3.3 Efficiency of Cycles Found in Step 1 ($k = 1$)

Step 1 of our cycle generation algorithm is designed to find cycles with high efficiency. To check how well this objective is achieved, we compare the highest efficiency and the average efficiency of the cycles generated in Step 1 with the best efficiency of all cycles in the network. For Network1, the highest/average efficiency of cycles found in Step 1 is 3.0/2.18 while the best efficiency of all cycles is 3.29. For Network2, the highest/average efficiency of the cycles found in Step 1 is 2.07/1.58 while the best efficiency of all cycles is 2.22. These results show that the highest efficiency of the cycles found in Step 1 is close to the best efficiency of all cycles in the network. Therefore, our algorithm is effective in finding high efficiency cycles.

7.4 Conclusion

We proposed a cycle generation algorithm that can find good candidate cycles for use by ILP or a heuristic algorithm to find the optimal p-cycles to protect a given working capacity

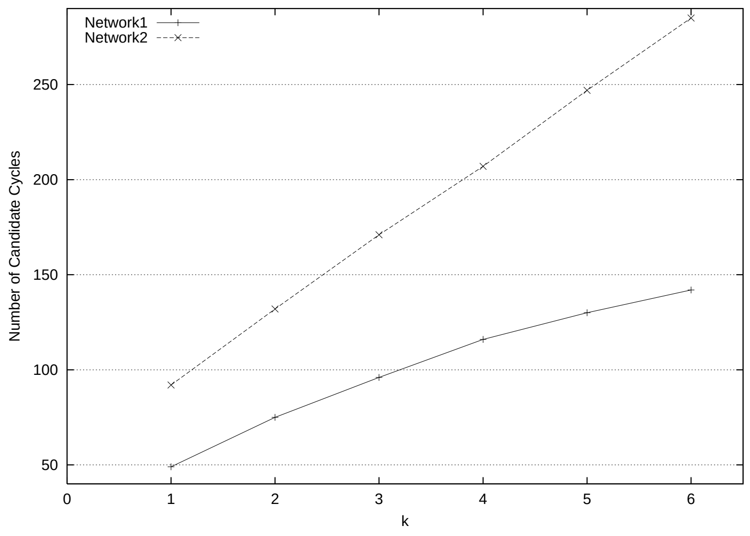


Figure 7.4 Effect of k on the Number of Candidate Cycles.

distribution. The algorithm consists of two steps, which generates $O(|E|)$ high efficiency cycles and $O(|E|)$ short cycles respectively. The key component in step 1 is the Weighted DFS-based Cycle Search (WDCS) algorithm that can generate high efficiency cycles early in the DFS search by properly setting the edge weights in the graph and always using the edge with the highest weight to extend the searching path. The problem of sparse working capacity distribution is addressed in step 2 by including short cycles into the candidate cycle set. Test results show that the candidate cycles generated by our algorithm lead to good spare capacity efficiency when used by ILP and the heuristic algorithm CIDA. In addition, by tuning the input parameter k, our algorithm provides tradeoff between the number of candidate cycles and the performance of the candidate cycles.

CHAPTER 8. P-CYCLE DESIGN ON WDM NETWORK WITH SRLG'S

A shared risk link group (SRLG) is a set of links that share a common resource whose failure will cause the failure of all links in it [42]. For instance, multiple fiber links laid out in a common conduit in WDM networks can be viewed as an SRLG because the conduit cut will result in the failure of all fibers in it. In general, a network contains a set of SRLGs that can be pre-determined according to the resource sharing relationship. An example SRLG set for the network in Figure 6.1 may be $\{\{ab\}, \{ac, ae, af\}, \{ac, bc, ef\}, \{cd, df\}, \{bf, cf, de, ef\}\}$. If the SRLG $\{ac, bc, ef\}$ fails, the links ac, bc, and ef all fail. A well-studied path-based protection scheme for the single-SRLG failure model is SRLG-diverse routing, which finds a pair of SRLG-disjoint paths (primary path and backup path) between the source and destination nodes of a connection. SRLG-disjoint means that no single SRLG failure will break both the primary and the backup paths simultaneously. The NP-completeness proof of the SRLG-diverse routing problem as well as an ILP solution are given in [43][44]. Heuristic approaches to this problem are proposed in [45][46]. Some recent works [47][48][49] explored p-cycle design for dual link failures. In this chapter, we will discuss a new protection strategy for the single-SRLG failure model, that is, to apply p-cycle network design on networks with SRLGs.

This chapter is organized as follows. In Section 8.1, we study the protection a p-cycle can offer upon an SRLG failure. In Section 8.2, we describe the p-cycle design problem and give an ILP formulation that solves it optimally. In Section 8.3, we provide an algorithm for generating a subset of all cycles that can protect any single SRLG failure. In Section 8.4, we introduce the concept of *SRLG-independent restorability* to address the SRLG failure detection issue that affects fast restoration. In Section 8.5, we present some simulation results on two networks with randomly generated SRLG sets. Finally, a conclusion is given in Section 8.6.

8.1 p-Cycle Protection upon an SRLG Failure

In the single link failure model, the protection that can be provided by a p-cycle for a link depends on their relationship. Specifically, if the link is on-cycle, then the p-cycle can offer one restoration path in case the link fails; if the link straddles the p-cycle, then two restoration paths are provided by the p-cycle when the link is broken; otherwise the p-cycle cannot protect the link. Upon an SRLG failure, all links in the SRLG are gone. To restore such a failure, every failed link must be taken care of by some p-cycles. Meanwhile, since multiple links may fail in case of an SRLG failure, if two or more failed links happen to be on the same p-cycle, then the p-cycle is broken, which makes the situation more complicated than in the single link failure model. Figure 8.1 illustrates the possible relationships between an SRLG failure and a p-cycle. Part (a) shows the case where the p-cycle remains a cycle after the SRLG failure. Two restoration paths can be provided for links $s-t$, $u-v$, and $t-x$ respectively given enough copies of the p-cycle. However, no restoration path can be provided for link $x-y$ since y is not on the p-cycle. In part (b), the p-cycle is broken into a path from s to t after the SRLG failure. One restoration path can be provided for links $s-t$, $u-v$, and $t-x$ respectively given enough copies of the p-cycle. The p-cycle provides no restoration path for link $y-z$ because y and z are not on the p-cycle. In part (c), the p-cycle is broken into two segments $s\cdots u$ and $v\cdots t$. No restoration path is available for links $s-t$, $u-v$, and $x-z$ since their two end nodes are on different segments. However, the p-cycle can supply one restoration path for link $x-y$ since both x and y are on the segment $v\cdots t$. To compute the protection a p-cycle can provide for a link upon an SRLG failure, we define a function called CYCLE_LINK_SRLG. CYCLE_LINK_SRLG takes a cycle i, a link j, and an SRLG k as inputs, and returns the number of restoration paths that can be provided for link j by cycle i in case of the failure of SRLG k. The pseudo-code of function CYCLE_LINK_SRLG is given below.

```
int CYCLE_LINK_SRLG(cycle $i$, link $j$, SRLG $k$)
1. **if** $j \notin k$ **then**
```

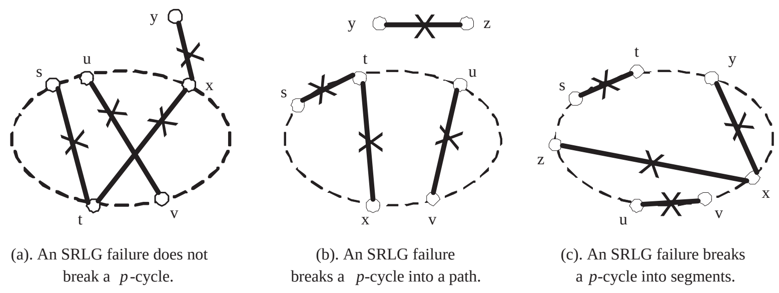

y
u
s
x
t
v
(a). An SRLG failure does not break a p-cycle.
y
z
t
u
s
x
v
(b). An SRLG failure breaks a p-cycle into a path.
t
y
s
z
x
u
v
(c). An SRLG failure breaks a p-cycle into segments.

Figure 8.1 Relationship between an SRLG failure and a p-cycle (Solid lines are links in the failed SRLG. Dashed-line ellipses represent p-cycles.)

```
2.      return 0; // The link does not belong to the SRLG and
        therefore does not fail.
3.  if link $j$'s end nodes are not both on cycle $i$ then
4.      return 0; // The link cannot be protected by the cycle.
5.  Remove links in SRLG $k$ from cycle $i$.
6.  if cycle $i$ remains a cycle then
7.      return 2;
8.  else // Cycle $i$ is broken into one or more segments.
9.      if link $j$'s end nodes are on the same segment then
10.         return 1;
11.     else
12.         return 0;
```

8.2 An ILP for Optimal p-Cycle Design

8.2.1 Problem Description

We consider the following p-Cycle design problem: given a network represented by a graph $G = (V, L)$, a set of SRLGs in G, a set of distinct candidate p-cycles in G, and the working capacity on each link in G, compute a set of p-cycles that minimizes the total cost of spare capacity required to achieve 100% restoration in case of a single SRLG failure.

To guarantee the existence of a solution to this problem, the following conditions are assumed:

- The network is two-edge-connected so that each link can be protected by at least one cycle.
- The failure of any single SRLG does not disconnect the network.
- In case of any single SRLG failure, for each link in this SRLG, at least one cycle exists in the candidate p-cycle set such that the cycle can provide at least one restoration path for it.
- There is enough capacity on each link in the network.

8.2.2 ILP Formulation

Sets: (input)

L: The set of all links.

P: The set of candidate p-cycles.

R: The set of SRLGs.

Parameters: (input or pre-computed)

w_j: The working capacity on link j.

c_j: The cost of one unit of spare capacity on link j.

p_{ij}: 1 if link j is on cycle i, 0 otherwise.

b_{jk}: 1 if link j is in SRLG k, 0 otherwise.

x_{ijk}: The number of restoration paths for link j that can be provided by cycle i in case SRLG k fails. This value can take 0, 1, or 2, which is pre-computed by the function CYCLE_LINK_SRLG(i, j, k).

Variables: (to be determined)

s_j: The spare capacity on link j.

n_i: The number of copies of cycle i needed in the p-cycle design.

n_{ik}: The number of copies of cycle i needed in case SRLG k fails.

n_{ijk}: The number of copies of cycle i needed for link j in case SRLG k fails.

$$\text{Minimize } \sum_{j \in L} c_j \cdot s_j$$

Subject to:

$$s_j = \sum_{i \in P} p_{ij} \cdot n_i \quad \forall j \in L \tag{8.1}$$

$$b_{jk} \cdot w_j \leq \sum_{i \in P} x_{ijk} \cdot n_{ijk} \quad \forall j \in L, \ \forall k \in R \tag{8.2}$$

$$n_{ik} = \sum_{j \in L} b_{jk} \cdot n_{ijk} \quad \forall i \in P, \ \forall k \in R \tag{8.3}$$

$$n_i \geq n_{ik} \quad \forall i \in P, \ \forall k \in R \tag{8.4}$$

Constraints in (8.1) reflect the relationship between the spare capacities and the result p-cycle design. Specifically, the spare capacity on link j will be the total number of p-cycles that traverse it. Constraints in (8.2) guarantee that if link j is in SRLG k (i.e., $b_{jk} = 1$), its working capacity is protected in case SRLG k fails. And if link j is not affected by SRLG k's failure (i.e., $b_{jk} = 0$), this constraint can be ignored since the left hand side is 0. This enforces that when an SRLG fails, only the links that belong to it need to have their working capacities restored by p-cycles. Constraints in (8.3) ensure that when SRLG k fails, all links in k should be restored. So the number of copies of cycle i needed for SRLG k's failure is the sum of the number of copies of cycle i needed for all links in SRLG k. Constraint (8.4) is equivalent to $n_i = \max_{k \in R} n_{ik}$. This equation reflects the fact that under the single SRLG failure assumption, the number of copies of cycle i needed is dominated by the maximum requirement over all single SRLG failures.

This ILP will be referred to as ILP1 in the rest of this chapter.

8.3 Generation of Candidate Cycles

As in the single link failure model, the set of candidate p-cycles must contain all cycles in the network in order for the ILP to obtain the optimal p-cycle design. This requirement blows up the complexity of the p-cycle design since the number of cycles in a network grows exponentially with the network size. To overcome this difficulty, we give an algorithm for generating a small subset of all cycles as the candidate p-cycle set such that a p-cycle design can be found to fully survive any single SRLG failure in the network given enough spare capacities. The algorithm works as follows. For each SRLG, first remove all links in it from the network graph. Then for each pair of end nodes of a removed link, find its shortest path as well as two node-disjoint shortest paths (if exist) in the remaining graph. The shortest path is combined with the removed link to form a cycle that contains the removed link as on-cycle link, and the two node-disjoint shortest paths (if exist) are combined to form a cycle on which the removed link straddles. The distinct cycles generated are collected into the candidate p-cycle set. FIND_BASIC_CYCLES is a function implementing the above algorithm and the pseudocode of it is given below.

```
FIND_BASIC_CYCLES(network $G$, SRLG set $R$)
1.  $P = \emptyset$;
2.  for each SRLG $k \in R$ do
3.      let $G' = (V, L - k)$;
4.      for each link $l = (u, v) \in k$ do
5.          Compute the shortest path between $u$ and $v$ in $G'$.
6.          let $c$ = the cycle formed by the shortest path and $l$;
7.          let $P = P \cup \{c\}$;
8.          Compute two node-disjoint shortest paths
            between $u$ and $v$ in $G'$.
9.          if such a pair of node-disjoint paths exist then
10.             let $c$ be the cycle formed by the two paths;
```

```
11.            let $P = P \cup \{c\}$;
12. return $P$;
```

Given a two-edge-connected network and an SRLG set such that any single SRLG failure does not disconnect the network, the algorithm can always find a candidate p-cycle set that can provide 100% restorability in case of any single SRLG failure given enough spare capacities. The reason is that in case of any SRLG failure, for each affected link, it is always possible to find a shortest path between the two end nodes of the link because the network is still connected. This guarantees that when an SRLG fails, each link in the SRLG has at least one cycle that can provide a restoration path for it.

For each link $j \in L$, suppose t_j is the number of SRLGs that contains j. Let $t = \max_{j \in L} t_j$. Then the number of distinct cycles generated by FIND_BASIC_CYCLES is $O(tm)$ where m is the number of links in the network.

8.4 SRLG-Independent Restorability

8.4.1 Impact of SRLG Failure Detection Problem on Restoration Speed

An important feature of p-cycle survivable network design with the single link failure model is fast restoration. When a link fails, its end nodes can detect the failure immediately and start restoration right away using the pre-configured p-cycles. But the single SRLG failure model changes the situation. Upon an SRLG failure, for each failed link, although its end nodes can detect the link failure immediately as in the single link failure model, they may not be able to start restoration at that moment. The reason is that in order to figure out which pre-configured p-cycles should be used to restore the link, the end nodes of the link need to know which SRLG has failed. Unless this can be inferred directly from the knowledge of the SRLG set, a signaling protocol is needed to enable all involved nodes to find out which SRLG has failed. For example, in Figure 8.2, when SRLG $g_1 = \{ab, ac\}$ fails, node c can detect the failure of link ac instantly; however, since ac also belongs to another SRLG g_2, c cannot tell

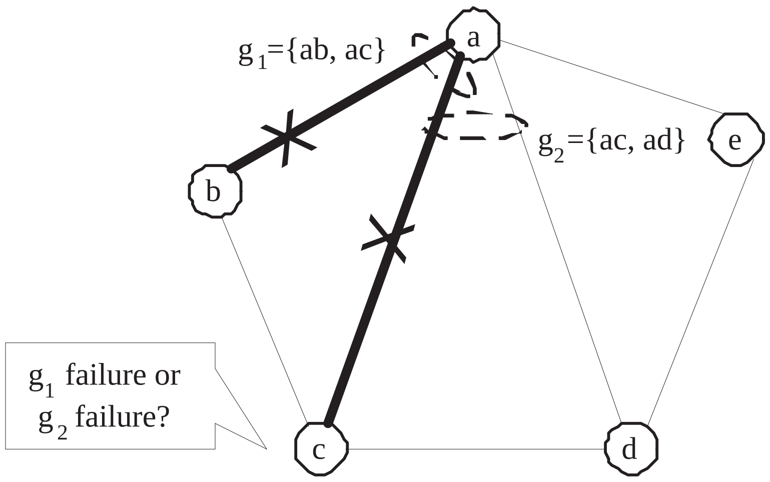


Figure 8.2 SRLG Failure Detection Problem

whether the failed SRLG is g_1 or g_2 until it gets more failure information from the network.

8.4.2 A Solution with SRLG-Independent Restorability

When an SRLG failure happens, for each affected link, we want its end nodes to start the restoration of the failed link before they find out which SRLG is down. To achieve this, we introduce the concept of *SRLG-independent restorability.* The idea can be illustrated by Figure 8.3 in which we try to restore the traffic on link $a-c$ right after node a and c detect the failure of link $a-c$. Note that at this moment, node c does not know whether the failed SRLG is g_1 or g_2. Consider two p-cycles, $c_1 = a-b-c-d-e-a$ and $c_2 = a-b-c-a$. It can be seen that c_1 can provide the failed link $a-c$ with either one restoration path $(a-e-d-c)$ if g_1 fails or two restoration paths $(a-b-c$ and $a-e-d-c)$ if g_2 fails. To accommodate the worst case scenario, we consider that c_1 can provide only one restoration path for link $a-c$ and can be used to restore traffic on link $a-c$ regardless whether g_1 or g_2 has failed. On the other hand, although c_2 can be used to restore link $a-c$ in case of g_2 failure, it cannot be used to restore link $a-c$ in case of g_1 failure. Therefore, we cannot use c_2 to restore link $a-c$ immediately after a and c detect the link failure. We say that link $a-c$ is *SRLG-independently restorable* by cycle c_1 but not by cycle c_2, i.e., c_1 can be used to restore link $a-c$ no matter which SRLG that contains $a-c$ has failed.

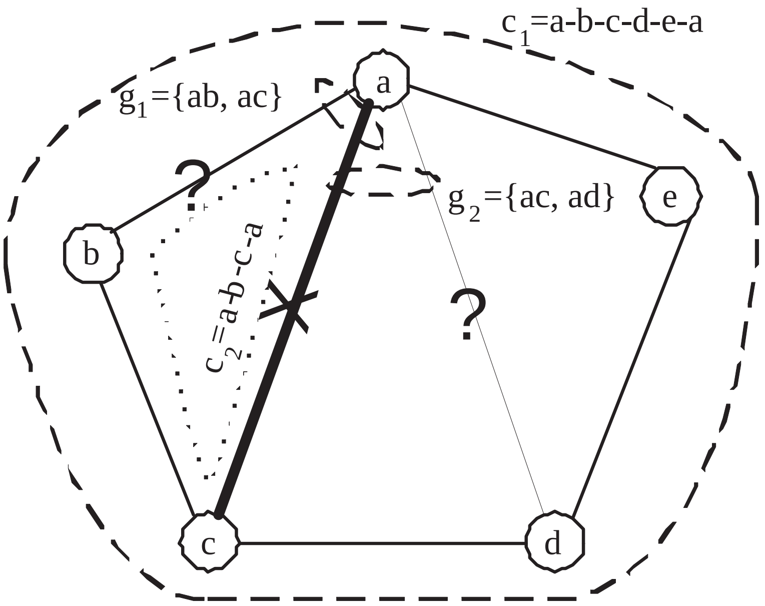


Figure 8.3 SRLG Failure Detection Problem

For convenience, the SRLG-independent restorability for a given a network $G = (V, L)$ with respect to an SRLG set R is referred to as *R-independent restorability.* The formal definition of R-independent restorability is given as follows.

Definition *R-independent restorability parameter x'_{ij}* is the number of restoration paths for link j that can be provided by cycle i in case of a failure of any SRLG $k \in R$ that contains j. And $x'_{ij} = \min_{k \in R_j} x_{ijk}$ where $R_j = \{g \in R : j \in g\}$ and x_{ijk} is computed by the CYCLE_LINK_SRLG function defined in Section 8.1.

Definition Link $j \in L$ is *R-independently restorable by cycle i* if $x'_{ij} > 0$. And link $j \in L$ is *R-independently restorable* if there exists a cycle i in G such that j is R-independently restorable by i. Moreover, network G is *R-independently restorable* if all links in G are R-independently restorable.

To compute an optimal p-cycle design with SRLG-independent restorability, constraint (8.2) of ILP1 needs to be revised as follows.

$$b_{jk} \cdot w_j \leq \sum_{i \in P} x'_{ij} \cdot n_{ijk} \quad \forall j \in L,\ \forall k \in R \tag{8.2'}$$

The new ILP is referred to as ILP2. Note that ILP2 may not be able to find a solution for a problem instance for which ILP1 can find a solution because some link(s) in G may not be SRLG-independently restorable. To obtain the optimal p-cycle design with the presence of non-SRLG-independently restorable links, we can apply constraint (8.2') for those links that are SRLG-independently restorable and apply constraint (8.2) for those links that are not SRLG-independently restorable. Upon an SRLG failure, the affected SRLG-independently restorable links can be restored immediately; however, the non-SRLG-independently restorable links involved cannot be restored until the end nodes of the failed links use a signaling protocol to find out which SRLG actually failed.

8.4.3 Hardness of Generating Candidate Cycles with SRLG-Independent Restorability

In Section 8.3, we give a polynomial time algorithm to generate a small candidate p-cycle set to guarantee 100% restorability so that the enumeration of all cycles is avoided. The algorithm guarantees that when an SRLG fails, each link in the SRLG has at least one candidate cycle that can provide a restoration path for it. A natural question to ask is whether a similar approach can be taken with regard to SRLG-independent restorability, that is, whether it is possible to find in polynomial time a small subset of all cycles as the candidate p-cycle set such that when an SRLG fails, each link in the SRLG is R-independently restorable by at least one candidate cycle.

In this section, we show that it is NP-hard to find a cycle for a given link such that the link is SRLG-independently restorable by the cycle.

We start with the well-studied SRLG-diverse routing problem, which can be described as follows. For a network $G = (V, L)$ and an SRLG set R, between a given pair of nodes $u, v \in V$, find two paths p_1 and p_2 such that no SRLG failure breaks both paths simultaneously, i.e., $\forall g \in R$, $E(p_1) \cap g = \emptyset \vee E(p_2) \cap g = \emptyset$ must hold where $E(p_1)/E(p_2)$ denotes the link set of path p_1/p_2.

On the other hand, for any link $j \in L$ with end nodes u and v, a cycle i that could potentially protect j must contain both u and v. So cycle i can be viewed as two node disjoint paths between u and v. Without loss of generality, we call these two paths p_1 and p_2. We have the following alternative definition for SRLG-independent restorability.

Definition Link $j = u - v \in L$ is *R-independently restorable by cycle* i in G if and only if $u, v \in V(i)$ and $\forall k \in R_j$, $E(p_1) \cap k = \emptyset \vee E(p_2) \cap k = \emptyset$ where $V(i) = \{\text{all nodes on cycle } i\}$ and $R_j = \{g \in R : j \in g\}$.

It can be seen that to generate a cycle that can SRLG-independently restore a link is essentially to find two **node disjoint** SRLG-diverse paths between the end nodes of the link. Here we need two paths to be node disjoint to guarantee they form a cycle. With a simple modification to the NP-hardness proof for the SRLG-diverse routing problem provided in [44], we can prove that it is NP-hard to generate a cycle that can SRLG-independently restore a given link by reduction from the 3-SAT problem. For the complete proof, please refer to the Appendix of this chapter. Because of this result, no effort should be spent on finding a polynomial time candidate cycle generation algorithm for a given network and its SRLG set with SRLG-independent restorability consideration.

8.5 Numerical Results

8.5.1 Settings

Two networks shown in Figure 8.4 are used for simulations. Network 1 is an 11-node 23-link network taken from the website of [41] and Network 2 is a 15-node 28-link Metropolitan network.

For each network, two demand sets are used. One is the uniform demand set that contains one demand for each unordered source-destination pair. The other is a set that contains 150 random demands for Network 1 and 300 random demands for Network 2, respectively. In all demand sets, each demand requests for one unit of capacity. For each combination of a test network and a demand set, working capacities on the network links are obtained by routing each demand over the shortest path. The cost of each unit of spare capacity on a link is set to one, i.e., $c_j = 1$ for all $j \in L$.

In practice, SRLG sets are known *a priori*. In our simulations, they are randomly generated.

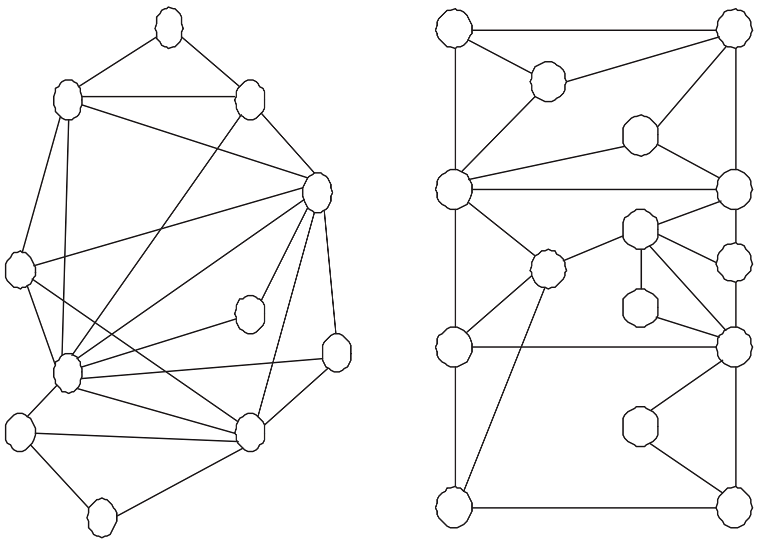


Figure 8.4 Test networks.

To facilitate this, we define the term "r-SRLG set" where r is a positive integer. An r-SRLG set has the property that each SRLG in the set contains at most r links. For each network, we randomly generate r-SRLG sets ($2 \leq r \leq 4$) conforming to the following rules for each SRLG set:

- Any single SRLG failure does not disconnect the network. This is necessary to guarantee a feasible p-cycle design.
- Each link in the network belongs to at least one SRLG. That is, there is no risk-free link in the network.
- Each SRLG is not a subset of another SRLG. Note that as long as an SRLG failure is restorable, a failure of any subset of it is also restorable without requiring more spare capacities.

Note that a network with single link failure has a 1-SRLG set.

All simulations are run on a Sun Ultra 10 workstation equipped with a single 440 MHz CPU, 256 MB RAM, and 4 GB virtual memory. CPLEX8.1 is used as the solver for ILP formulations.

Table 8.1 Optimal p-cycle design results

	Network 1		Network 2	
	Uniform	150 random	Uniform	300 random
# candidate cycles	307	307	976	976
Working Capacity	96	244	228	679
Spare Capacity (1-SRLG set)	86	214	136	400
Spare Capacity (2-SRLG set)	122	302	260	763
Spare Capacity (3-SRLG set)	133	315	354	1035
Spare Capacity (4-SRLG set)	165	392	388	1174

8.5.2 *p*-Cycle Design without Considering SRLG-Independent Restorability

To compute the optimal p-cycle design, we solve ILP1 for both test networks and all cycles are used as candidate p-cycles for ILP1. The number of candidate cycles, total working capacity on all links, and total spare capacity required under different simulation settings are shown in Table 8.1. It can be seen that when a SRLG set has larger SRLGs (i.e., larger r values), the total spare capacity needed becomes larger since more working capacity is affected by a single SRLG failure. On the other hand, this trend seems to be more dramatic for Network 2 than for Network 1. This can be explained by the fact that the average working capacity per link in Network 2 is higher than that in Network 1, so the amount of affected working capacity grows faster in Network 2 than in Network 1 as the size of SRLGs in the SRLG set becomes larger.

To evaluate the effect of using the basic candidate cycle set generated by the algorithm FIND_BASIC_CYCLES given in Section 8.3 as the candidate p-cycle set for ILP1, we compare the spare capacity requirement and running time of ILP1 with all cycles and with the basic candidate cycles in Tables 8.2 and 8.3. Table 8.2 shows the number of cycles generated for the basic candidate cycle set, the total spare capacity required for ILP1 with all cycles, and the total spare capacity required for ILP1 with the basic candidate cycles for various simulation settings. For ILP1 with basic candidate cycles, we also list the percentage of extra spare

Table 8.2 Spare capacity comparison between all cycles and basic cycles (the percentages over the optimum are shown in basic cycles entries)

r	Network 1					Network 2				
	#cycles	Uniform		150 random		#cycles	Uniform		300 random	
	BASIC	ALL	BASIC	ALL	BASIC	BASIC	ALL	BASIC	ALL	BASIC
1	27	86	136 58%	214	329 54%	30	136	223 64%	400	651 63%
2	39	122	175 43%	302	445 47%	41	260	341 31%	763	1008 32%
3	33	133	188 41%	315	444 41%	42	354	433 22%	1035	1269 23%
4	38	165	251 52%	392	588 50%	49	388	466 20%	1174	1404 20%

capacity required over the optimal solution in the parentheses. Table 8.3 gives the running time comparison between ILP1 with all cycles and ILP1 with basic candidate cycles. In algorithm FIND_BASIC_CYCLES, we used a two-step algorithm to find two node-disjoint shortest paths for a pair of nodes. That is, we find a shortest path first, then remove all intermediate nodes along this path and try to find a shortest path in the remaining graph.

As shown in Table 8.2, the spare capacity obtained by ILP1 with basic candidate cycles is always greater than the optimal value. This is expected because FIND_BASIC_CYCLES generates a small subset of all cycles. We notice that for both test networks, the spare capacity over usage with single link failure (1-SRLG set) is larger than with 2, 3, 4-SRLG sets. In other words, the negative effect of using a small basic candidate cycle set on an r-SRLG set ($r > 1$) is less than on the single link failure model. This can be explained by the following two facts. Firstly, under the single link failure model, a link can always be protected by a cycle as long as its end nodes are on the cycle; while for an r-SRLG ($r > 1$) set, we need an extra requirement that the SRLG failure should not break the cycle. This makes the number of cycles that can potentially protect a link become smaller for an r-SRLG set ($r > 1$) compared to 1-SRLG set. Secondly, the simulation results show that FIND_BASIC_CYCLES generates more cycles for 2, 3, 4-SRLG sets than for 1-SRLG sets. Hence, the loss of candidate cycles in the basic candidate cycle sets is less significant for 2, 3, 4-SRLG sets than for 1-SRLG sets.

Table 8.3 Running time (second) comparison between all cycles and basic cycles

r	Network 1					Network 2				
	#cycles	Uniform		150 random		#cycles	Uniform		300 random	
	BASIC	ALL	BASIC	ALL	BASIC	BASIC	ALL	BASIC	ALL	BASIC
1	27	3.2	0.2	3.5	0.3	30	21.1	0.4	23.6	0.4
2	39	15.7	0.8	110	0.8	41	237	1.0	270	1.0
3	33	6.7	0.3	7.8	0.4	42	1184	0.9	922	0.9
3	38	50.0	0.5	40.0	0.5	49	2420	1.2	940	1.3

On the other hand, Table 8.3 shows that ILP1 with basic candidate cycle set runs significantly faster than ILP1 with all cycles because the basic candidate cycle set contains much fewer cycles and as a result the corresponding ILP formulation has much fewer variables and constraints. It can be seen from Table 8.3 that the speedup ranges from 13 to 132 for Network 1, and from 60 to 1945 for Network 2. Of course, the performance gain in running time is achieved at the cost of sacrificing spare capacity efficiency, as shown in Table 8.2.

8.5.3 p-Cycle Design with SRLG-Independent Restorability

To evaluate the impact of SRLG-independent restorability on the optimal p-cycle design, we also run ILP2 with all cycles as the candidate cycles. The comparison between the total spare capacity required with and without SRLG-independent restorability (computed by ILP2 and ILP1 respectively) is shown in Table 8.4. Since for 1-SRLG set (single link failure model), the results are the same no matter SRLG-independent restorability is considered or not, we omit the results for 1-SRLG set in the table.

As expected, the introduction of SRLG-independent restorability results in more spare capacity requirement because x'_{ij} is a more restrictive restorability parameter than x_{ijk} is. However, the extra spare capacity required by ILP2 over ILP1 is relatively small – between 5.7% and 10.7% for Network 1, and between 12.3% and 16.4% for Network 2.

Notice that for 4-SRLG set, ILP2 fails to find a solution for both networks, which means that there is at least one non-SRLG-independently restorable link in both networks. To find out how often such an undesirable situation happens and how many non-SRLG-restorable links

Table 8.4 Optimal p-cycle design with and without SRLG-independent restorability

Spare Capacity	Network 1				Network 2			
	Uniform		150 random		Uniform		300 random	
	ILP1	ILP2	ILP1	ILP2	ILP1	ILP2	ILP1	ILP2
2-SRLG set	122	135	302	331	260	292	763	872
3-SRLG set	133	142	315	333	354	412	1035	1200
4-SRLG set	165	N/A	392	N/A	388	N/A	1174	N/A

Table 8.5 Non-SRLG-independently restorable cases

	Network 1		Network 2	
	# bad cases (out of 100)	Avg. # bad links among bad cases	# bad cases (out of 100)	Avg. # bad links among bad cases
$r = 2$	11	1.18	1	1.00
$r = 3$	31	1.19	29	1.14
$r = 4$	46	1.61	63	1.87

exist in each case, we generate 100 r-SRLG ($2 \leq r \leq 4$) set for each network. Corresponding results are shown in Table 8.5 in which "bad" means non-SRLG-independently restorable.

As shown in Table 8.5, as r increases, the probability that we run into a non-SRLG-independently restorable situation gets higher for both networks. Meanwhile, in those non-SRLG-independently restorable cases, the number of non-SRLG-independently restorable links is low. Actually, most non-SRLG-independently restorable cases are caused by only one or two non-SRLG-independently restorable links. This means that when an SRLG failure occurs, it is very likely that most broken links can be restored immediately. Those affected non-SRLG-independently restorable links can be restored after their end nodes find out which SRLG has failed using a signaling protocol.

8.6 Conclusion

In this chapter, we extend the p-cycle survivable network design from the single link failure model to the single SRLG failure model. An ILP formulation is provided to compute a p-cycle design with minimum spare capacity requirement for an input network, its SRLG set, and its working capacities on the network links such that 100% restorability can be guaranteed in case of any single SRLG failure. To avoid the enumeration of all cycles in the input network, we propose a polynomial time algorithm called FIND_BASIC_CYCLES to generate a basic candidate p-cycle set of size $O(tm)$ (m is the number of links in the network and $t = \max_{j \in L} t_j$ where t_j is the number of SRLGs to which link j belongs). Given enough spare capacity, such a candidate p-cycle set can be used by the ILP to compute a p-cycle design that guarantees 100% restorability. Using the basic candidate p-cycle set can significantly reduce the time to compute an ILP solution while compromising the spare capacity optimality of the ILP solution due to the reduced number of candidate cycles. This trade-off is confirmed by our simulation results.

The SRLG failure detection issue undermines fast restoration, which is a key merit of p-cycle survivable network design with the single link failure model. We propose the concept of SRLG-independent restorability to solve this problem. The idea is to redefine the restorability parameter such that a broken link can be restored immediately by a p-cycle before its end nodes find out which SRLG has failed. We provide a revised ILP to compute an optimal p-cycle design with SRLG-independent restorability. Simulation results show that the additional spare capacity required by SRLG-independent restorability is reasonable. Moreover, we prove that it is NP-hard to compute a candidate p-cycle set to ensure 100% SRLG-independent restorability in case of any single SRLG failure.

Appendix

SRLG-independent restorability problem (SRLG-I-R) is defined as follows. For a network $G = (V, L)$, an SRLG set $R \subseteq \mathcal{P}(L)$, and a link $j \in L$, is there a cycle i in G such that j can be SRLG-independently restored by i, i.e., both end nodes of j are on cycle i and $\forall k \in R_j$, $E(p_1) \cap k = \emptyset \vee E(p_2) \cap k = \emptyset$ where $R_j = \{g \in R : j \in g\}$ and p_1 and p_2 are two paths between the end nodes of j on cycle i?

Theorem 8.1 *SRLG-I-R is NP-hard.*

Proof: We will prove that 3-SAT $\leq_p$ SRLG-I-R.

Given a conjunction normal form formula with n boolean variables $x_1, x_2, \cdots, x_n$ and m clauses $C_1, C_2, \cdots, C_m$, each of which is a disjunction of three literals where each literal takes the form of x_q or $\overline{x_q}$ $(1 \leq q \leq n)$ and the three literals have different subscriptions, we construct an instance of the SRLG-I-R problem as follows.

1) Construct a network.

- For each variable x_q $(1 \leq q \leq n)$, build a node q. In addition, a node 0 is added. Add two parallel links labeled x_q and $\overline{x_q}$ between nodes $q-1$ and q $(1 \leq q \leq n)$.
- For each clause C_h $(1 \leq h \leq m)$, build a node h'. In addition, a node $0'$ is added. For each clause $C_h = l_1 \vee l_2 \vee l_3$ $(1 \leq h \leq m)$, add three parallel links labeled $C_h l_1$, $C_h l_2$, and $C_h l_3$ between nodes $(h-1)'$ and h'.
- Add two extra nodes s and t. Add links $s-t$, $s-0$, $s-0'$, $n-t$, and $m'-t$.

2) Define an SRLG set.

For each literal l, we define an SRLG that contains l and $C_h\bar{l}$ if $\bar{l}$ appears in clause C_h $(1 \leq h \leq m)$. In addition, each SRLG contains link $s-t$.

3) Let j be $s-t$, i.e., we need to determine whether link $s-t$ is SRLG-independently restorable by a cycle in the network.

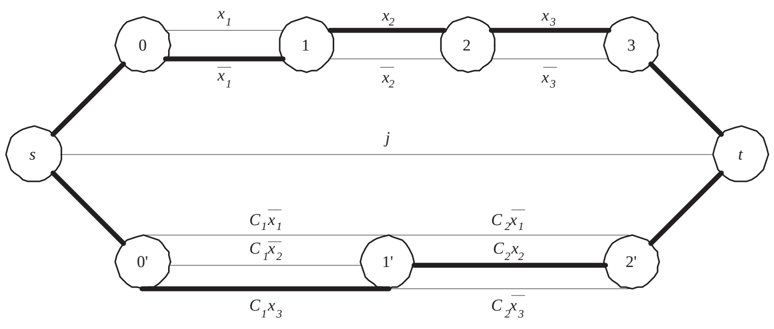
Figure 8.5 shows an example that illustrates the construction. Note that parallel links in the network are treated as different links when they are used to form a cycle. It can be seen that there are three groups of cycles in the constructed network.

(a) Cycles with the form $s-0-1-2-\cdots-n-t-m'-(m-1)'-\cdots-0'-s$.

(b) "Top" cycles with the form $s-0-1-2-\cdots-n-t-s$.

(c) "Bottom" cycles with the form $s-t-m'-(m-1)'-\cdots-0'-s$.

It is easy to verify that no cycle in group (b) or group (c) can provide SRLG-independent restorability for link $s-t$ because of the way we define the SRLG set. Therefore, only cycles in group (a) can possibly offer SRLG-independent restorability for link $s-t$. And whether



SRLGs on the network:

$g_1 = \{x_1, C_1\overline{x_1}, C_2\overline{x_1}, j\}, g_2 = \{\overline{x_1}, j\},$

$g_3 = \{x_2, C_1\overline{x_2}, j\}, g_4 = \{\overline{x_2}, C_2x_2, j\},$

$g_5 = \{x_3, C_2\overline{x_3}, j\}, g_6 = \{\overline{x_3}, C_1x_3, j\}$

Figure 8.5 The network as well as the SRLG set derived from the formula $(\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$. Thick solid lines denote an SRLG-independently restorable cycle for link j corresponding to the assignment $x_1 = 0$, $x_2 = 1$, $x_3 = 1$.

a cycle in group (a) can provide SRLG-independent restorability for link $s - t$ depends on whether the cycle's top and bottom paths contain any links that belong to the same SRLG.

We now show that link $j = s - t$ is SRLG-independently restorable if and only if the given 3-CNF formula is satisfiable.

Suppose j is SRLG-independently restorable, then there exists a cycle i in group (a) that can always provide either its top or bottom path between s and t as the restoration path for j no matter which SRLG fails. We obtain an assignment to the given 3-CNF formula based on cycle i as follows. If i's top path traverses link x_q, then x_q is assigned value 1. If i's top path traverses $\overline{x_q}$, then x_q is assigned value 0. Such an assignment must satisfy each clause $C_h = l_1 \vee l_2 \vee l_3$ $(1 \leq h \leq m)$. The reason is the following. Without loss of generality, assume the bottom path of i traverses C_hl_1 between nodes $(h-1)'$ and h', then i's top path cannot traverse $\overline{l_1}$ because if so, the SRLG-independent restorability would be violated since there is an SRLG that contains both $\overline{l_1}$ and C_hl_1 and the failure of this SRLG will break both the top and the bottom paths. Therefore, i's top path must traverse l_1. According to the assignment rule, the literal l_1 is assigned value 1, which makes the clause C_h 1. Thus, the given 3-CNF formula is satisfiable.

Suppose the given 3-CNF formula is satisfiable, then it has a satisfying assignment. Based on the assignment, we construct a cycle using the following rules. If x_q is $1/0$ in the assignment, then let the top path of the cycle traverse link $x_q/\overline{x_q}$ and include all links with labels $C_h x_q/C_h\overline{x_q}$ into the bottom path. Note that this may result in multiple links for a bottom path segment. If this happens, just arbitrarily pick one link out of them for the segment. Since the assignment satisfies the 3-CNF formula, a valid bottom path (i.e. each segment contains one link) can be built from s to t. Furthermore, the bottom path together with the top path form a cycle that offers SRLG-independent restorability for link j because any single SRLG failure will not break both the top path and the bottom path simultaneously. ∎

CHAPTER 9. PATH-SEGMENT P-CYCLE DESIGN FOR DYNAMIC TRAFFIC

Early works on p-cycle mainly focus on the p-cycle network design problem under the static traffic model where a set of demands to be supported by the network is known in advance. Recently, the problem of using p-cycles to protect dynamic traffic has been studied in [50][51]. In [52], the concept of protected working capacity envelope (PWCE) [53] is applied to p-cycle networks to support dynamic traffic.

On the other hand, the original concept of a p-cycle protecting a span that is either on the p-cycle or straddling the p-cycle is also expanded. In [54], span protecting p-cycle (hereafter referred to as *span p-cycle*) is extended to path-segment protecting p-cycle (hereafter referred to as *flow p-cycle*) that can protect a segment (consisting one or more continuous spans) of the working path. An ILP model is given in [54] to solve the problem of using flow p-cycles to protect a given set of demands with minimum spare capacity requirement. It was shown in [54] that network designs using flow p-cycles can yield significant reduction in spare capacity requirement compared with network designs using span p-cycles.

In this chapter, a dynamic service provisioning algorithm that uses flow p-cycles to provide service protection is developed. When a demand arrives at the network, we need to compute a working path for the demand and configure flow p-cycles to protect the demand's working capacities on the working path. The goal is to minimize the total working and spare capacities required to set up the demand. When a demand leaves the network, we need to reclaim the working capacities and possibly some spare capacities used by the demand.

This chapter is organized as follows. In section 9.1, we review the concept of flow p-cycle defined in [54] and give a function to compute a cycle's protection capability for a working path

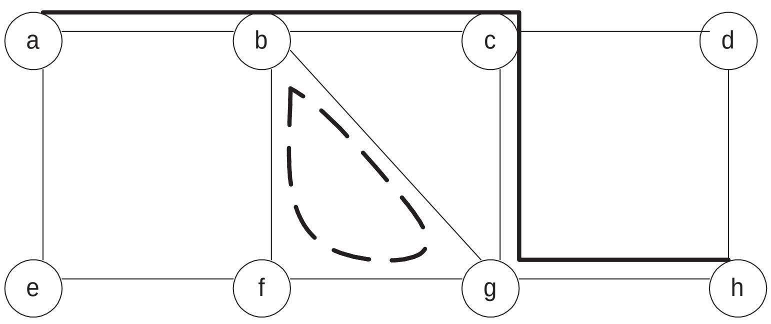


Figure 9.1 A flow p-cycle example

upon a span failure. In section 9.2, we present an ILP formulation to compute the optimal setup for a dynamic demand. We also present a procedure to compute the reclaimable resources when a demand departs the network. Simulation results are discussed in section 9.3. Finally, a conclusion is given in section 9.4.

9.1 Flow p-Cycle Concept

The concept of flow p-cycle (or path-segment protecting p-cycle) is introduced in [54]. The difference between a flow p-cycle and a conventional span p-cycle is illustrated in Figure 9.1. The figure shows a working path $p =< a-b-c-g-h >$ and a p-cycle $c =< b-g-f-b >$. If c is regarded as a span p-cycle, then it cannot provide protection for span $b-c$ or span $c-g$ on p since neither of the two spans is an on-cycle or straddling span of c. On the other hand, segment $< b-c-g >$ of p can be considered to straddle cycle c. Thus, c can be considered as a flow p-cycle that provides two protection paths ($< b-g >$ and $< b-f-g >$) for segment $< b-c-g >$ of p in case span $b-c$ or span $c-g$ fails. Essentially, the concept of flow p-cycle extends the protectability of p-cycle from a span to a path segment.

Given a cycle i, a span j, and a working path p, we can compute the number of protection paths cycle i can provide for path p in case span j fails using the following CYCLE_SPAN_PATH function.

```
int CYCLE_SPAN_PATH(cycle $i$, span $j$, path $p$)
1. **if** $j \notin p$ **then**
```

```
2.      return 0;
Suppose the source and destination of $p$ are $s$ and $t$,
and the end nodes of $j$ are $u$ and $v$.
Without loss of generality, assume $u$ is closer to $s$
and $v$ is closer to $t$.
The segment $< s - \cdots - u >$ is denoted as $S_1$.
The segment $< v - \cdots - t >$ is denoted as $S_2$.
3. let $N_1 = V(S_1) \cap V(i)$, $N_2 = V(S_2) \cap V(i)$;
4. if $N_1 = \emptyset$ or $N_2 = \emptyset$ then
5.      return 0;
6. Pick $u' \in N_1$ such that $u'$ is the closest to $u$ along $S_1$;
7. Pick $v' \in N_2$ such that $v'$ is the closest to $v$ along $S_2$;
8. if $u' = u$ and $v' = v$ then
9.      if $u - v$ is an on-cycle span of cycle $i$ then
10.          return 1;
11.return 2;
```

Line 1-2 consider the case that j is not contained in p. In this case, 0 is returned since the failure of j will not affect p and therefore no protection path is needed for p. In line 3, $V(S_1)$ and $V(S_2)$ denote the set of nodes on S_1 and S_2 respectively, and $V(i)$ denotes the set of nodes on i. Line 4-5 deal with the case that j is not in a segment of p that intersects i at both ends. In this case, 0 is returned since cycle i cannot provide a protection path for p in case j fails. Line 6-11 cover the case that j is in a segment of p that intersects i at both ends. Two subcases are considered here. In the first subcase, j is a span on cycle i, so i can provide one protection path for j in case j fails. In the second subcase, j is contained in a segment of p that straddles i (the segment may contain 0 or more spans other than j), so i can provide two protection paths for the segment in case j fails.

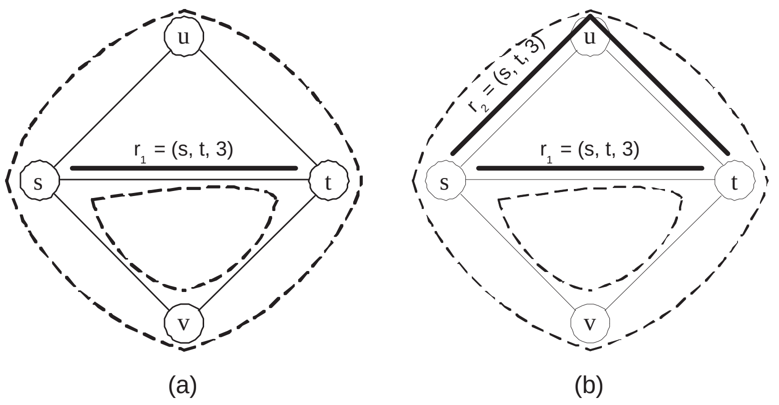


Figure 9.2 A demand is denoted as a 3-tuple (source, destination, capacity request). The thick solid line represents the working path for a demand and the dashed line represents a *p*-cycle. (a) Total capacity required to set up r_1 is 10. (b) Total capacity required to set up r_2 is 6 if we reuse the existing *p*-cycles.

9.2 Dynamic Service Provisioning Using Flow *p*-Cycles

9.2.1 Demand Setup

We consider dynamic provisioning of demands with protection against single span failures using flow *p*-cycles. When a demand with capacity requirement of d arrives, we need to take the following steps to set up the demand: a) Choose a working path. b) Set up d working lightpaths on the chosen working path. c) Configure flow *p*-cycles to protect the demand's working capacities on the working path. Our goal is to minimize the total cost of working and spare capacities required to set up the demand. If the network cannot accommodate the demand due to insufficient available capacity, the demand will be rejected.

The key to reduce the total capacity requirement of the current demand is to reuse the existing *p*-cycles (i.e., *p*-cycles created to protect the existing demands) to protect the current demand. Consider the example shown in Figure 9.2. In (a), a demand r_1 comes with source s, destination t, and 3 units of capacity request. The optimal way to set it up is to use working path $< s - t >$, and create a *p*-cycle $< s - u - t - v - s >$ and a *p*-cycle $< s - t - v - s >$ to protect the working capacities of the demand. The first *p*-cycle can provide two restoration paths in case span $s - t$ fails and the second *p*-cycle can provide one restoration path in case span $s-t$ fails. Thus, the two p-cycles together can protect 3 units of working capacity on the working path. Suppose after r_1 is set up, another demand r_2 also asking for 3 units of capacity between s and t arrives. If we use $< s-t >$ as its working path (not shown in the figure), the most efficient way to protect r_2 is to add a p-cycle $< s-u-t-v-s >$ and a p-cycle $< s-t-v-s >$ (or $< s-t-u-s >$). By this way, the total capacity required to set up r_2 is 10, which is the sum of 3 units of working capacity, 4 units of spare capacity taken by p-cycle $< s-u-t-v-s >$, and 3 units of spare capacity taken by p-cycle $< s-t-v-s >$ (or $< s-t-u-s >$). Figure 9.2 (b) offers a better solution, which uses $< s-u-t >$ as the working path. In this solution, no new p-cycle needs to be deployed since the existing p-cycle $< s-u-t-v-s >$ can provide one restoration path ($< s-v-t >$) for r_2 and the other existing p-cycle $< s-t-v-s >$ can provide two restoration paths ($< s-t >$ and $< s-v-t >$) for r_2 in case span $s-u$ or $u-t$ fails. r_2 can reuse the p-cycles deployed to protect r_1 because r_1's working path and r_2's working path will not fail simultaneously under the single span failure assumption. In this solution, the total capacity required to set up r_2 is 6, which equals the total working capacity on the working path $< s-u-t >$.

In the following, we provide an ILP formulation to solve the following dynamic demand provisioning problem. Given a demand that requires d units of capacity, compute a working path and a set of p-cycles to protect the demand's working capacity on the working path so that the total working and spare capacity required by the demand is minimized. The ILP computes the optimal solution by reusing the existing p-cycles to protect the current demand.

Sets: (input or pre-computed)

S: The set of all spans in the network.

P: The set of candidate cycles.

Q: The set of candidate working paths for the current demand.

D_j: The set of existing (i.e., already admitted) demands whose working paths traverse span j.

Parameters: (input or pre-computed)

d: The capacity units required by the current demand.

c_j: The cost of one unit of capacity on span j.

t_j: The residual capacity on span j.

p_{ij}: 1 if span j is on cycle i, 0 otherwise.

b_{jk}: 1 if span j is on the k^{th} candidate working path for the current demand, 0 otherwise.

x_{ijk}: The number of restoration paths that can be provided by cycle i in case span j fails if the k^{th} candidate working path is chosen for the current demand. This value can be 0, 1, or 2, which is pre-computed by the function CYCLE_SPAN_PATH(i, j, p_k) where p_k denotes the k^{th} candidate working path.

N_i: The number of unit-capacity copies of cycle i that have been deployed before the current demand arrives.

n^r_{ij}: The number of unit-capacity copies of cycle i configured for the restoration of an existing demand r in case span j fails.

Integer variables: (to be solved)

$w_j \geq 0$: The working capacity on span j required by the current demand.

$s_j \geq 0$: The spare capacity on span j required by p-cycles to be deployed for the current demand.

$\gamma_k \in \{0, 1\}$: Takes value 1 if the k^{th} candidate working path is chosen for the current demand, 0 otherwise.

$n_i \geq 0$: The number of unit-capacity copies of cycle i need to be deployed for the current demand.

$n_{ij} \geq 0$: The number of unit-capacity copies of cycle i needed for the restoration of the current demand in case span j fails.

$$\text{Minimize} \sum_{j \in S} c_j \cdot (w_j + s_j)$$

Subject to:

$$\sum_{k \in Q} \gamma_k = 1 \tag{9.1}$$

$$w_j = \sum_{k \in Q} \gamma_k \cdot b_{jk} \cdot d \qquad \forall j \in S \tag{9.2}$$

$$s_j = \sum_{i \in P} p_{ij} \cdot n_i \qquad \forall j \in S \tag{9.3}$$

$$\sum_{i \in P} x_{ijk} \cdot n_{ij} \geq \gamma_k \cdot b_{jk} \cdot d \qquad \forall k \in Q,\ \forall j \in S \tag{9.4}$$

$$N_i + n_i \geq n_{ij} + \sum_{r \in D_j} n_{ij}^r \qquad \forall i \in P,\ \forall j \in S \tag{9.5}$$

$$w_j + s_j \leq t_j \qquad \forall j \in S \tag{9.6}$$

The objective is to minimize the cost of working and spare capacity required by the current demand. Constraint (9.1) ensures that exactly one candidate working path is chosen for the current demand. Constraints in (9.2) compute the working capacity required by the current demand on each span. Constraints in (9.3) compute the spare capacity required by the current demand on each span. Specifically, the spare capacity required on span j equals the number of p-cycles to be deployed for the current demand that traverse j. Constraints in (9.4) guarantee that if span j fails and j is on the working path of the current demand, there are enough p-cycles to restore the current demand's working capacity on j. Constraints in (9.5) guarantee that the newly deployed copies of cycle i for the current demand together with the already deployed copies of cycle i for the existing demands should be enough to satisfy the requirement for copies of cycle i to restore the current demand as well as all the existing demands upon any single span failure. Finally, constraints in (9.6) ensure that the residual capacity on each span is sufficient to support the working and spare capacities required by the current demand.

When a demand arrives at the network, the ILP is solved for the demand. If a solution cannot be found for the ILP, then the demand is rejected. If a solution is found, then the demand can be satisfied by setting up d lightpaths on the working path and configuring the p-cycles for demand protection. When a demand is satisfied, the corresponding sets and parameters need to be updated accordingly. Specifically, assume r is the satisfied demand, the following need to be done. $D_j = D_j \cup \{r\}$ for all span j on the working path of r.

$t_j = t_j - (w_j + s_j)$ for all span j that needs to allocate working or spare capacity. $N_i = N_i + n_i$ for all newly deployed cycle i. Finally, n^r_{ij} should be set to the value of n_{ij}.

9.2.2 Demand Teardown

When an admitted demand r departs the network, we need to collect the resources taken by it. Since the working capacities are dedicated to a demand, the working capacities along r's working path can be reclaimed. In terms of the spare capacities taken by the p-cycles that protects r, the situation is more complicated because the p-cycles used for protecting r might be shared by other demands. Using Figure 9.2 (b) as an example, suppose after r_2 is set up, r_1 is leaving the network. We can collect the working capacities along r_1's working path $< s-t >$. However, the p-cycles $< s - u - t - v - s >$ and $< s - v - t - s >$ must be kept to protect the remaining demand r_2. Therefore, when tearing down a demand, only those p-cycles not used by any other demand can be torn down and their occupied spare capacities can be returned to the network. The pseudo-code describing the teardown procedure is given below. In the pseudo-code, D denotes the current set of demands in the network (including r, the demand to be torn down) and N_i denotes the number of copies of cycle i that are currently deployed in the network.

```
TEAR_DOWN(demand $r$)
1. let $D = D - \{r\}$;
2. Release the working capacities along $r$'s working path.
3. let $C = \{$cycles used for protecting demand $r\}$;
4. for each cycle $i \in C$ do
5.     let $N'_i = \max_{j \in S} \sum_{d \in D_j} n^d_{ij}$;
6.     if $N_i - N'_i > 0$ then
7.         Release the spare capacities taken by
           $(N_i - N'_i)$ copies of cycle $i$.
```

In line 5, N'_i is the number of copies of cycle i needed to protect all demands against any single span failure after r departs the network. So the number of copies of cycle i that can be torn down is $(N_i - N'_i)$. Note that in line 5, it is not necessary to compute $\sum_{d \in D_j} n^d_{ij}$ for each j for the following reason. If span j is on the working path of r, then $D_j = D_j - \{r\}$ after demand r's departure; otherwise D_j is unchanged. The computation of N'_i can be accelerated with an auxiliary data structure $N_{ij} = \sum_{d \in D_j} n^d_{ij}$ for each cycle i and each span j because N_{ij} has to be re-computed only if j is on the working path of r.

9.3 Numerical Results

9.3.1 Simulation Settings

The 14-node, 21-span NSFNET shown in Figure 9.3 is used for simulations. Each span in the network is set to have 64 wavelengths. The arrival of demands follows Poisson distribution with arrival rate $\lambda = 2$ demands per unit time and the demand holding time is exponentially distributed with mean $1/\mu \in \{4, 5, 6, 7, 8\}$. Therefore, the traffic load in Erlang is $\lambda/\mu \in \{8, 10, 12, 14, 16\}$. For each traffic load, ten groups of 1000 demands are loaded to the network. Demands are uniformly distributed among all node pairs and the capacity units requested by a demand is chosen from $\{1, 2, 3, 4\}$ with equal probability. Each group of demands are set up and torn down according to their arrival and departure sequence and the *total revenue* of all demands is measured. Then the average total revenue of the ten groups is calculated for each traffic load. The *revenue* of a demand is defined as the product of its capacity units requested, the hop count of its shortest path in the network, and its holding time if the demand is satisfied, otherwise the revenue is 0. (The total revenue metric is similar to the total earning metric proposed in [55].)

Two candidate cycle sets are used for the ILP. The first set contains all cycles in the network. The second set contains cycles obtained by the algorithm presented in [56] which computes $O(m)$ (m is the number of spans in a network) cycles that provide protection for all spans in the network. For NSFNET, the first set contains 139 cycles and the second set contains 34 cycles. Candidate working paths for each node pair are pre-computed using k-shortest paths

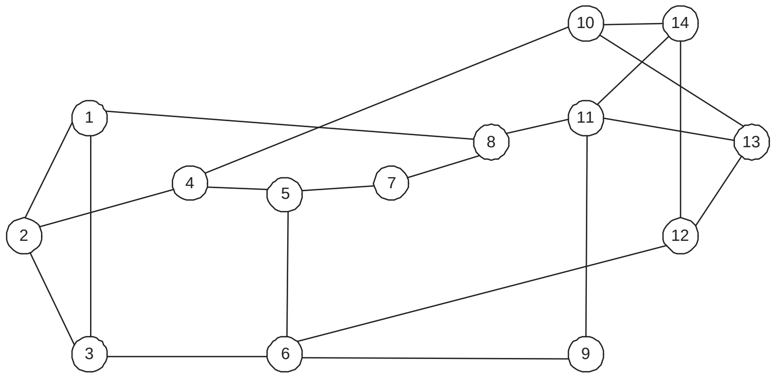


Figure 9.3 NSFNET

algorithm provided in [57]. In our simulations, the value of k is adjusted from 1 to 6.

All simulations are run on a Sun Ultra 10 workstation equipped with a single 440 MHz CPU, 256 MB RAM, and 4 GB virtual memory. CPLEX8.1 is used as the solver for ILP.

9.3.2 Flow p-Cycle vs. Span p-cycle

Since span p-cycle is a special case of flow p-cycle, the ILP for setting up a demand given in section 9.2 can be easily adapted to work for span p-cycle by changing the computation of x_{ijk} to $x'_{ijk} = x_{ij} \cdot b_{jk}$ where x_{ij} is the span-cycle relationship parameter, i.e., $x_{ij} = 2$ if j is a straddling span of cycle i, $x_{ij} = 1$ if j is on cycle i, $x_{ij} = 0$ otherwise. As for demand teardown procedure, nothing needs to be changed. The performance comparison between flow p-cycle and span p-cycle is shown in Figure 9.4. The figure shows the average total revenue at various loads when k (number of candidate working paths) is set to 3.

It can be seen that flow p-cycle always yields higher revenue than span p-cycle. The improvement ranges from 3% to 25% for all cycles case, and from 3% to 13% for limited cycles case. In addition, as the load gets higher, the advantage of flow p-cycle over span p-cycle gets larger.

9.3.3 On the Number of Candidate Working Paths k

Figure 9.5 shows the relationship between average total revenue and k value when load = 12. For flow p-cycle case, we expect that providing more candidate working paths for each

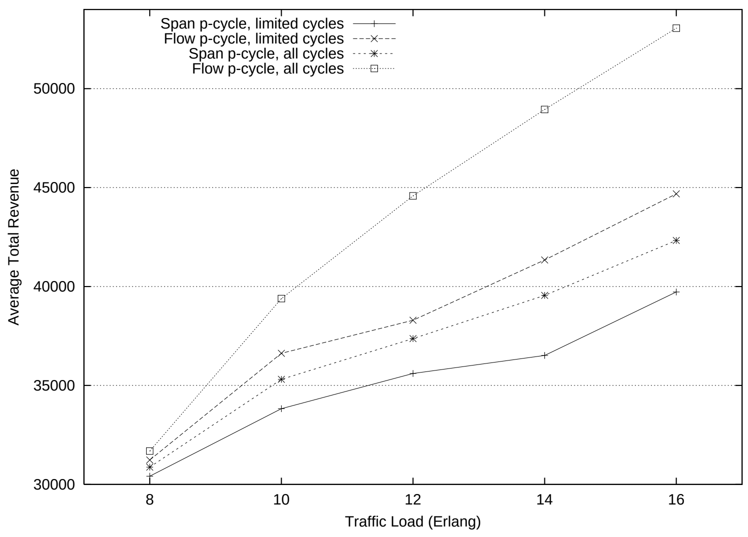


Figure 9.4 Relationship between average total revenue and traffic load when $k = 3$.

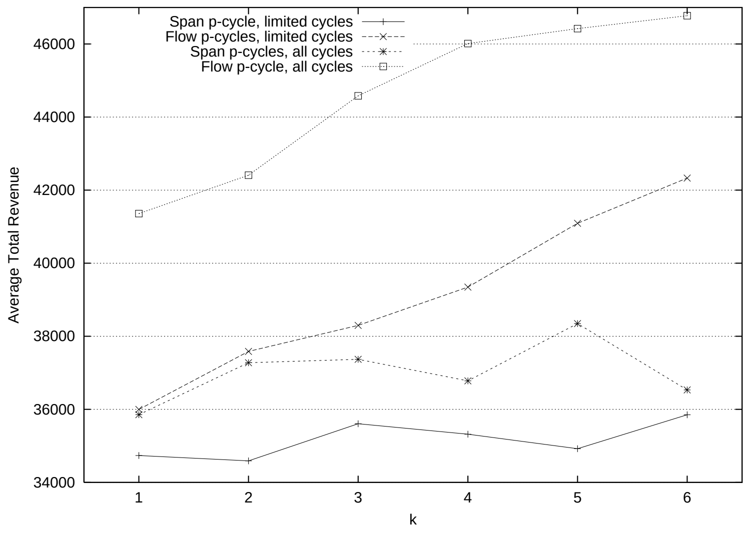


Figure 9.5 Relationship between average total revenue and k value when load = 12.

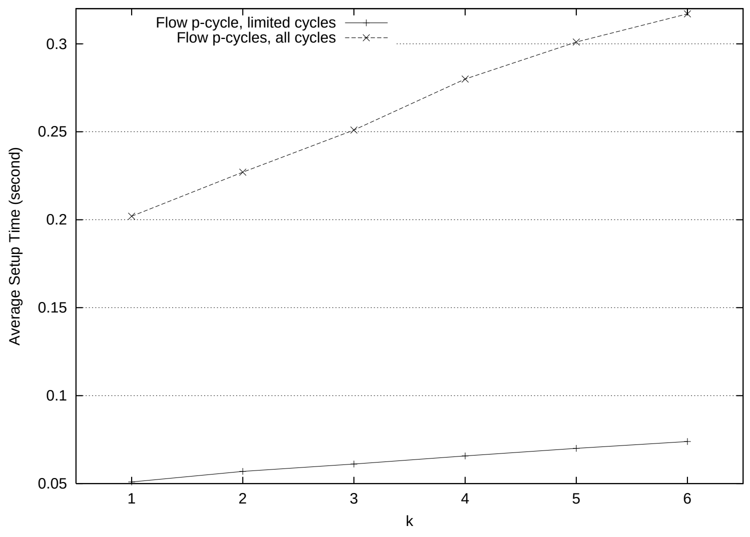


Figure 9.6 Relationship between average demand setup time and k value when load = 12.

demand would help increase the total revenue since the network would have a better chance to accommodate a new demand more efficiently. Consider the flow p-cycle reuse example shown in Figure 9.2. If only one candidate working path is provided for each demand, p-cycle reuse cannot be achieved. On the other hand, we can benefit from p-cycle reuse by providing more candidate working paths as shown in Figure 9.2. In Figure 9.5, the top two curves for flow p-cycle show the trend that total revenue increases as k increases. However, for span p-cycle, this trend is not exhibited. To explain this result, notice that when flow p-cycle is used, candidate working paths other than the shortest path provide opportunities to take advantage of the straddling relationship between working path segments and flow p-cycles. However, such a benefit does not exist for span p-cycles that only provide protection to spans. Therefore, span p-cycles may not benefit from multiple choices of working paths as flow p-cycles do.

9.3.4 All Cycles vs. Limited Cycles

It can be observed from Figure 9.4 and Figure 9.5 that using all cycles always results in better performance than using limited cycles for both flow p-cycle and span p-cycle cases. On the other hand, as shown in Figure 9.6, the price of using all cycles is longer time to compute the setup for a demand because more candidate cycles result in more integer variables (n_i, n_{ij})

and more constraints (9.5) in the ILP. Similar tradeoff also applies to the number of candidate paths (k value) because as k gets larger, more integer variables (γ_k) and more constraints (9.4) are introduced.

9.4 Conclusion

We propose a dynamic service provisioning algorithm using flow p-cycle to protect demand against single span failures. When a demand arrives at the network, we use an ILP formulation to compute a working path and a set of flow p-cycles to protect the demand with the goal of minimizing the total working capacity and spare capacity consumption. When an existing demand leaves the network, its working capacities and releasable spare capacities are reclaimed. Simulations are run on various settings to evaluate the performance of the proposed scheme. The results show that flow p-cycle outperforms span p-cycle to a considerable extent for dynamic traffic. Furthermore, trade-off between total revenue and demand setup time can be achieved by tunable parameters such as the number of candidate cycles and the number of candidate working paths.

CHAPTER 10. CONCLUSION AND FUTURE WORK

10.1 Conclusion

WDM optical networks are becoming the backbone transport network for the next generation Internet. Survivability issues in WDM optical networks are important since failures would result in huge data loss. This dissertation studies this field from two perspectives, one is the survivable mapping from IP layer to WDM layer, the other is *p*-cycle protection schemes on WDM networks.

For the survivable mapping problem, this dissertation offers an efficient heuristic approximation algorithm, MAP-and-FIX, to break down its hardness. Moreover, it also discovers a new strategy, logical topology augmentation. The idea of selectively adding some logical links so that a survivable mapping can be easily computed is unique among existing survivable mapping problem solutions. Following this approach, this dissertation originates a new survivable mapping problem that minimizes the survivable mapping cost with the flexibility of adding new links into the given logical topology. The significance as well as certain interesting graph theoretical properties of this new problem is deliberated. Overall, this dissertation exhibits a new perspective for the survivable mapping problem.

On the other hand, this dissertation also delves in WDM layer *p*-cycle protection design problems. The first challenge settled by this dissertation is how to avoid using all cycles in the network without sacrificing too much efficiency in the result *p*-cycle design. Then we apply *p*-cycle design onto a more realistic failure model, single SRLG failure model that eliminates the single link failure assumption. In addition, using path-segment *p*-cycle (also known as flow *p*-cycle) to offer survivability for dynamic traffic demands is also presented in this dissertation. Our proposed schemes make it more practical to apply *p*-cycle protection in WDM networks.

10.2 Future Work

For the survivable mapping problem, some intriguing problems are also observed besides the results shown in this dissertation. And we believe that the answers to them would also have profound implication on related graph theory algorithms. Specifically, two open problems are proposed:

1. Given a logical topology and a physical topology, if there is a survivable mapping, is there always a reflectively-routed survivable mapping?

Should we have a positive answer to this question, it would be helpful to simplify the computation of a survivable mapping because the routes of reflective links would be determined right away. Although our simulations suggest a positive answer to the question, this is still an open problem waiting for a proof.

2. What would happen if the intersection of the logical topology and physical topology is a tree?

In Chapter 4, we proved a theorem saying that as long as the intersection of the logical and physical topology is 2-edge-connected, any reflectively-routed mapping would be survivable. Since the condition of 2-edge-connectivity is quite restrictive, we would like to know what will happen if we relax the condition to a spanning tree. At this point, we do not know whether the survivable mapping problem is NP-hard or not if the intersection is a tree. No matter what the answer is, it is significant. If survivable mapping problem is NP-hard even under this constraint, it would mean that 2-edge-connectivity might be the bottom line for the logical topology augmentation approach. Otherwise, if survivable mapping problem is polynomial time solvable under this constraint, it would mean that the relaxation works and we get an improved solution over the one proposed in Chapter 4.

For the p-cycle protection in WDM networks, we believe that more research efforts should be devoted into applying p-cycle on survivable dynamic traffic provisioning because as the price of bandwidth supported by WDM networks goes down, WDM networks service providers should see more and more dynamic traffic requests coming in. Cost-effective resources management will be of great interests for both service providers and customers.

BIBLIOGRAPHY

[1] A. Sen, B. Hao, and B.H. Shen, "Survivable routing in WDM networks", *Seventh International Symposium on Computers and Communications*, pp. 726-731, July 2002.

[2] W. D. Grover and D. Stamatelakis, "Cycle-oriented distributed preconfiguration: ring-like speed with mesh-like capacity for self-planning network restoration", *Proc. of IEEE ICC '98*, pp. 537-543, Jun. 1998.

[3] J.Y. Wei, "Advances in the management and control of optical Internet", *IEEE Journal on Selected Areas in Communications*, Vol. 20, No. 4, pp. 768-785, May 2002.

[4] L. Ruan and H. Luo, "Dynamic routing of restorable lightpaths: a tradeoff between capacity efficiency and resource information requirement", *Proceedings of the 7th IFIP Working Conference on Optical Network Design and Modeling*, pp. 537-548, Budapest, Hungary, February 2003.

[5] H. Luo and L. Ruan, "Load balancing heuristics for dynamic establishment of restorable lightpaths", *Proceedings of the Eleventh International Conference on Computer Communications and Networks*, pp. 472-477, Miami, Florida, October 2002.

[6] G. Mohan, C. S. R. Murthy, and A. K. Somani, "Efficient algorithms for routing dependable connections in WDM optical networks", *IEEE/ACM Transactions on Networking*, Vol. 9, No. 10, pp. 553-566, October 2001.

[7] B. T. Doshi, S. Dravida, P. Harshavardhana, O. Hauser and Y. Wang, "Optical network design and restoration", *Bell Labs Technical Journal*, Vol. 4, No. 1, pp. 58-84, Jan.-Mar. 1999.

[8] S. Ramamurthy and B. Mukherjee, "Survivable WDM Mesh Networks, Part I-Protection", *Proc. IEEE INFOCOM'99*, pp. 744-751, 1999.

[9] S. Sengupta and R. Ramamurthy, "Capacity efficient distributed routing of mesh-restored lightpaths in optical networks", *Proc. IEEE GLOBECOM '01*, pp. 2129-2133, 2001.

[10] J. Wang, L. Sahasrabuddhe, and B. Mukherjee, "Path vs. subpath vs. link restoration for fault management in IP-over-WDM networks: performance comparisons using GMPLS control signaling", *IEEE Communications Magazine*, Vol. 40, No. 11, pp. 80-87, Nov. 2002.

[11] S. Ramamurthy and B. Mukherjee, "Survivable WDM Mesh Networks, Part II-Restoration", *Proc. IEEE ICC'99*, pp. 2023-2030, 1999.

[12] R. Iraschko and W. Grover, "A highly efficient path-restoration protocol for management of optical network transport integrity", *IEEE Journal on Selected Areas in Communications*, Vol. 18, No. 5, pp. 779-794, May 2000.

[13] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol label switching architecture", RFC 3031, Jan. 2001.

[14] A. Autenrieth and A. Kirstadter, "RD-QoS - the integrated provisioning of resilience and QoS in MPLS-based networks", *IEEE International Conference on Communications 2002*, pages 1174-1178, 2002.

[15] M. Kodialam and T. V. Lakshman, "Dynamic routing of bandwidth guaranteed tunnels with restoration", *Proc. of IEEE INFOCOM'00*, pp. 902-911, 2000.

[16] D. Zhou and T.-H. Lai, "Efficient resource allocation in self-healing multiprotocol label switching mesh networks", *In Proc. of IEEE GLOBECOM'01*, pp. 2671-2675, 2001.

[17] D. Colle, P. Van Heuven, C. Develder, S. Van den Berghe, I. Lievens, M. Pickavet, and P. Demeester, "MPLS recovery mechanisms for IP-over-WDM networks", *Photonic Network Communications*, Vol. 3, No. 1-2, pp. 23-40, January-June 2001.

[18] J. L. Marzo, E. Calle, C. Scoglio, and T. Anjali, "Adding QoS protection in order to enhance MPLS QoS routing", *ICC '03*, pp. 1973-1977, May 2003.

[19] D. Colle, S. De Maesschalck, C. Develder, P. Van Heuven, A. Groebbens, J. Cheyns, I. Lievens, M. Pickavet, P. Lagasse, and P. Demeester, " Data-centric optical networks and their survivability", *IEEE Journal on Selected Areas in Communications*, Vol. 20, No. 1, pp. 6 -20, January 2002.

[20] E. Modiano and A. Narula-Tam, "Survivable routing of logical topologies in WDM networks", *INFOCOM 2001*, pp. 348-357, April 2001.

[21] Q. Deng, G. Sasaki, and C.-F. Su, "Survivable IP over WDM: a mathematical programming problem formulation," Proc. of 40th Allerton Conference on Communication, Control and Computing, Monticello, IL, Oct. 2002.

[22] O. Crochat and J.-Y. Le Boudec, "Design protection for WDM optical networks", *IEEE Journal on Selected Areas in Communications*, Vol. 16, No. 7, pp. 1158-1165, Sept. 1998.

[23] O. Crochat, J.-Y. Le Boudec, and O. Gerstel, "Protection interoperability for WDM optical networks", *IEEE/ACM Transactions on Networking*, Vol. 8, No. 3, pp. 384-395, June 2000.

[24] G. Sasaki, C.-F. Su, and D. Blight, "Simple layout algorithms to maintain network connectivity under faults", *38th Annual Allerton Conference on Communication, Control, and Computing*, pp. 1266-1273, Sept. 2000.

[25] M. Kurant and P. Thiran, "Survivable mapping algorithm by ring trimming (SMART) for large IP-over-WDM networks," *Poceedings of BroadNets 2004*, pp. 44-53, San Jose, CA, Oct. 2004.

[26] M. Kurant and P. Thiran, "On survivable routing of mesh topologies in IP-over-WDM networks," *Proc. of IEEE INFOCOM'05*, pp. 1106-1116, March 2005.

[27] L. Sahasrabuddhe, S. Ramamurthy, and B. Mukherjee, "Fault management in IP-over-WDM networks: WDM protection versus IP restoration", *IEEE Journal on Selected Areas in Communications*, Vol. 20, No. 1, pp. 21-33, Jan. 2002.

[28] N. C. Wormald, "Generating random regular graphs", *Journal of Algorithms*, Vol. 5, pp. 247-280, 1984.

[29] G. Even, J. Feldman, G. Kortsarz, and Z. Nutov, "A 3/2-approximation for augmenting edge-connectivity of a graph from 1 to 2 using a subset of a given edge set", *Proc. RANDOM-APPROX '01*, pp. 90-101, 2001.

[30] C. Liu and L. Ruan, "Logical topology augmentation for survivable mapping in IP-over-WDM networks," *Proc. of IEEE Globecom'05*, pp. 1885-1889, St. Louis, MO, Nov/Dec. 2005.

[31] P. Kelsen and V. Ramachandran, "On finding minimal two-connected subgraphs, " *Journal of Algorithms,* Vol. 18, No. 1, pp. 1-49, January 1995.

[32] R. Iraschko, M. MacGregor, and W. Grover, "Optimal capacity placement for path restoration in STM or ATM mesh-survivable networks", *IEEE/ACM Transaction on Networking*, Vol. 6, No. 3, pp. 326-336, Jun. 1998.

[33] D. A. Schupke, C. G. Gruber, and A. Autenrieth, "Optimal configuration of *p*-cycles in WDM networks", *Proc. of IEEE ICC '02*, pp. 2761-2765, Apr. 2002.

[34] J. Doucette, D. He, W. D. Grover, and O. Yang, "Algorithmic approaches for efficient enumeration of candidate *p*-cycles and capacitated *p*-cycle network design", *Proc. of the Fourth International Workshop on the Design of Reliable Communication Networks (DRCN 2003)*, pp. 212-220, Oct. 2003.

[35] D. A. Schupke, "An ILP for optimal *p*-cycle selection without cycle enumeration", *Proc. of the Eighth Working Conference on Optical Network Design and Modelling (ONDM)*, Feb. 2004.

[36] W. D. Grover and J. E. Doucette, "Advances in optical network design with p-cycles: joint optimization and pre-selection of candidate p-cycles", *Proc. of the IEEE-LEOS Summer Topical Meeting on All Optical Networking*, pp. WA2-49-WA2-50, Jul. 2002.

[37] C. G. Gruber, "Resilient networks with non-simple p-cycles", *Proc. of International Conference on Telecommunications (ICT 2003)*, Feb. 2003.

[38] C. Mauz, "p-Cycle protection in wavelength routed networks", *Proceedings of the Seventh Working Conference on Optical Network Design and Modelling (ONDM'03)*, Feb. 2003.

[39] H. Zhang and O. Yang, "Finding protection cycles in DWDM networks", *Proc. of IEEE ICC '02*, pp. 2756-2760, Apr./May 2002.

[40] D. B. Johnson, "Finding all the elementary circuits of a directed graph", *SIAM J. Comput.* Vol. 4, No. 1, pp. 77-84, 1975.

[41] W. D. Grover, *Mesh-based Survivable Networks: Options and Strategies for Optical, MPLS, SONET and ATM Networking*, Prentice Hall PTR, Upper Saddle River, New Jersey, 2003.

[42] D. Papadimitriou *et al.*, "Inference of shared risk link groups", Internet Draft, Work in Progress, Nov. 2001.

[43] J. Q. Hu, "Diverse routing in optical mesh networks", *IEEE Transactions on Communications*, vol. 51, no. 3, pp. 489-494, Mar. 2003.

[44] G. Ellinas *et al.*, "Routing and restoration architectures in mesh optical networks", *Optical Network Magazine*, vol. 4, no. 1, pp. 91-106, Jan./Feb. 2003.

[45] A. Todimala and B. Ramamurthy, "An iterative heuristic for SRLG diverse routing in WDM mesh networks", *Proc. of IEEE ICCCN '04*, pp. 199-204, Oct. 2004.

[46] D. Xu, Y. Xiong, C. Qiao, and G. Li, "Trap avoidance and protection schemes in networks with shared risk link groups", *IEEE/OSA Journal of Lightwave Technology*, vol. 21, no. 11, pp. 2683-2693, Nov. 2003.

[47] D. A. Schupke, "The tradeoff between the number of deployed *p*-cycles and the survivability to dual fiber duct failures", *Proc. of IEEE ICC '03*, vol. 2, pp. 1428-1432, May 2003.

[48] D. A. Schupke, "Multiple failure survivability in WDM networks with *p*-cycles", *Proc. of IEEE International Symposium on Circuits and Systems (ISCAS) '03*, vol. 3, pp. 866-869, May 2003.

[49] D. A. Schupke, W. D. Grover, and M. Clouqueur, "Strategies for enhanced dual failure restorability with static or reconfigurable *p*-Cycle networks", *Proc. of IEEE ICC '04*, vol. 3, pp. 1628-1633, Jun. 2004.

[50] L. Ruan and F. Tang, "Dynamic establishment of restorable connections using *p*-cycle protection in WDM networks", *Proc. of Broadnets '05*, pp. 147-154, Oct. 2005.

[51] W. He, J. Fang, and A. K. Somani, "A *p*-cycle based survivable design for dynamic traffic in WDM networks", *Proc. of IEEE Globecom '05*, pp. 1869-1873, Nov./Dec. 2005.

[52] G. Shen and W.D. Grover, "Design and performance of protected working capacity envelopes based on *p*-cycles for dynamic provisioning of survivable services", *OSA J. Optical Networking*, vol.4, no.7, pp. 361-390, Jul. 2005.

[53] W. D. Grover, "The protected working capacity envelope concept: an alternative paradigm for automated service provisioning", *IEEE Communications Magazine*, pp. 62-69, Jan. 2004.

[54] G. Shen and W. D. Grover, "Extending the *p*-cycle concept to path-segment protection", *Proc. IEEE International Conference on Communications*, pp. 1314-1319, May 2003.

[55] C. Qiao and D. Xu, "Distributed Partial Information Management (DPIM) Schemes for Survivable Networks - Part I", *Proc. of IEEE INFOCOM'02*, pp. 302 - 311, Jun. 2002.

[56] C. Liu and L. Ruan, "Finding good candidate cycles for efficient *p*-cycle network design", *Proc. of IEEE ICCCN '04*, pp. 321-326, Oct. 2004.

[57] N. Katoh, T. Ibaraki, and H. Mine, “An efficent algorithm for k shortest simple paths”, *Networks*, vol. 12, pp. 411-427, 1982.