Intelligent $p$-Cycle Protection for Dynamic Multicast Sessions in WDM Networks

Taiming Feng, Lu Ruan, and Wensheng Zhang
Department of Computer Science
Iowa State University, Ames, IA 50011
email: \{taiming, ruan, wzhang\}@cs.iastate.edu

Abstract—In WDM networks, it is important to maintain the survivability of communication sessions when link failure occurs due to the high bandwidth provided by a fiber link. Link failures have more serious impact on multicast sessions than on unicast ones since a link used by a multicast session may carry traffic to multiple destinations. Thus, it is more critical to protect multicast sessions against single link failures. Researchers have proposed various protection schemes for multicast sessions, including tree-based, path-based, and segment-based schemes. Tree-based schemes suffer excessive use of network capacity while path-based and segment-based schemes require long restoration time. In this paper, we propose a new $p$-cycle based dynamic multicast protection scheme named I$\p$C, which achieves both fast restoration and high capacity efficiency. The main feature of I$\p$C is that it computes high efficiency $p$-cycles on-demand to protect dynamic multicast sessions as they arrive. Extensive simulations have been conducted to evaluate the proposed I$\p$C scheme, and the results show that it outperforms an existing $p$-cycle based scheme.

I. INTRODUCTION

In WDM optical networks, a link failure will affect a large number of communication sessions due to the huge bandwidth provided by a fiber. Therefore, it is important to protect communication sessions against link failures. Compared to unicast sessions, multicast sessions suffer more seriously from link failures because a link may carry traffic to multiple destinations rather than to a single destination. Hence, multicast sessions demand more effective and efficient protection against link failures.

In this paper, we focus on the problem of protecting dynamic multicast sessions in WDM networks. Upon the arrival of a multicast request, a unidirectional primary multicast tree is first computed that connects the source node to all the destination nodes. Then, backup resources are reserved for the primary tree to protect it against single link failures. Researchers have proposed various multicast tree protection schemes, including tree-based [1]–[4], segment-based [1], [5], and path-based [1] schemes. In tree-based schemes, a primary tree is protected by either a link-disjoint backup tree or an arc-disjoint backup tree. In the former case, if the primary tree uses link $u \rightarrow v$, then the backup tree can use neither link $u \rightarrow v$ nor link $v \rightarrow u$. In the latter case, however, the backup tree is allowed to use link $v \rightarrow u$, but not link $u \rightarrow v$. The drawbacks of tree-based schemes include excessive use of network resources and unavailability of link/arc-disjoint trees in some cases. In segment-based schemes, each segment in the primary tree is protected by a path that is link-disjoint with the segment. Here, a segment is defined as the sequence of edges from the source or a splitting node on the tree to a leaf node or a downstream splitting node [1]. In path-based schemes, each destination $d_l$ in the multicast session is protected by a backup path that is link-disjoint with the path from $s$ to $d_l$ on the primary tree. Segment-based and path-based schemes are more capacity efficient than tree-based schemes since a backup path can share capacity with the primary tree as well as with the other backup paths. However, these schemes require long restoration time since some nodes need to reconfigure their switches to set up the backup path when a link failure occurs. $p$-Cycle [6] is a promising link-based protection technique that achieves both fast restoration speed and high capacity efficiency. Failure restoration is extremely fast since $p$-cycles are preconfigured using the spare capacity in the network. When a link fails, the two end nodes simply switch to the protection path provided by the $p$-cycle. $p$-Cycles also achieve high capacity efficiency since a $p$-cycle can protect both on-cycle links and straddling links. Although many $p$-cycle based schemes have been proposed for unicast protection [7]–[11], applying $p$-cycles for multicast protection has been barely studied. To use $p$-cycles to protect a multicast tree against single link failures, every link on the multicast tree should be protected by a $p$-cycle. Meanwhile, the $p$-cycles used to protect the tree links should consume as few network resources as possible. This results in a challenging problem of finding a set of $p$-cycles that can protect all links on the multicast tree and use the minimum number of wavelength channels. This problem has been studied by Zhong et al. in [12], [13]. Specifically, they proposed Integer Linear Program based methods [12] to protect static multicast sessions and the dynamic $p$-Cycle (D$p$C) scheme [13], extended from [14], to protect dynamic multicast sessions. The D$p$C scheme chooses $p$-cycles from a set of pre-computed candidate cycles, which cannot adapt to dynamic incoming multicast requests. In addition, the D$p$C scheme prefers short cycles, which may not always be a good choice because longer cycles may introduce more straddling links and therefore provide better protection efficiency.

To address the aforementioned limitations of the D$p$C scheme, we propose an intelligent $p$-Cycle (I$\p$C) scheme to provide $p$-cycle protection for dynamic multicast sessions. When a multicast request arrives, a multicast tree is computed for it (using any known algorithm) and then the I$\p$C scheme is
used to compute a set of high efficiency $p$-cycles on-demand to protect the multicast tree. The proposed IpC schemes has the following attractive features.

- It provides fast restoration since pre-configured $p$-cycles are used to protect the multicast tree links.
- It makes efficient use of spare capacity since a set of high efficiency $p$-cycles are computed on demand to protect the multicast tree links.
- Both intra-session sharing and inter-session sharing are achieved since a $p$-cycle can provide protection to links belonging to not only the same multicast tree, but also different multicast trees.
- The capacity efficiency is further improved by combining the existing $p$-cycles whenever possible.
- Assuming sufficient capacity is available in the network, a set of $p$-cycles can always be found to protect any multicast tree as long as the network is 2-edge-connected. This is not true for tree-based, segment-based, and path-based protection schemes. (Note that segment-based and path-based schemes suffer from the trap topology problem where a backup path cannot be found for a tree segment or tree path even though the network is 2-edge-connected.)

In this paper, we assume each node has full wavelength conversion capability. This makes wavelength assignment trivial (the following attractive features.

Given a multicast tree $T$, our IpC algorithm, formally presented in Algorithm 1, is used to find a set $PC$ of $p$-cycles to protect $T$ so that every link in $ET$ is protected by some $p$-cycle in $PC$. The framework of the algorithm is as follows.

1. For every link in $ET$, there are two options to protect it: finding a new $p$-cycle for it, or extending an existing $p$-cycle in $PC$ to protect it. Hence, we can find at most $2*|ET|$ $p$-cycles for all links in $ET$.
2. Let $p$ be the $p$-cycle with the maximum ER among all the $p$-cycles found in (1). We add $p$ to $PC$ and remove all links in $ET$ that can be protected by $p$.
3. We combine $p$ with the other $p$-cycles in $PC$ to reduce the wavelength usage of the $p$-cycles.
4. If $ET$ becomes empty, $PC$ is returned; otherwise, the above steps are repeated.

Three algorithms are used by our IpC algorithm. Algorithm 2 and Algorithm 3 are used in Step (1) to compute a new $p$-cycle and an extended $p$-cycle to protect a link in $ET$, respectively. Algorithm 4 is used in Step (3) to combine $p$ with the other $p$-cycles in $PC$. In the following, we discuss the detail of Algorithm 1. We then describe Algorithm 2, Algorithm 3, and Algorithm 4 in the next three subsections.

<table>
<thead>
<tr>
<th>Algorithm 1 Find $p$-Cycles to Protect Multicast Tree $T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $PC = \phi$</td>
</tr>
<tr>
<td>2: Remove every link in $ET$ that can be protected by an existing $p$-cycle</td>
</tr>
<tr>
<td>3: while ($ET \neq \phi$) do</td>
</tr>
<tr>
<td>4: $Temp = \phi$</td>
</tr>
<tr>
<td>5: for every $e \in ET$ do</td>
</tr>
<tr>
<td>6: Find a new $p$-cycle $p_{new}$ for $e$ using Algorithm 2 and add $p_{new}$ to $Temp$</td>
</tr>
<tr>
<td>7: if $PC \neq \phi$ then</td>
</tr>
<tr>
<td>8: Find an extended $p$-cycle $p_{ext}$ for $e$ using Algorithm 3 and add $p_{ext}$ to $Temp$</td>
</tr>
<tr>
<td>9: end if</td>
</tr>
<tr>
<td>10: end for</td>
</tr>
<tr>
<td>11: if $Temp = \phi$ then</td>
</tr>
<tr>
<td>12: Return NULL</td>
</tr>
<tr>
<td>13: end if</td>
</tr>
<tr>
<td>14: Find $p$ in $Temp$ with the maximum ER and add $p$ to $PC$</td>
</tr>
<tr>
<td>15: if $p$ is extended from a $p$-cycle $p_i$ in $PC$ then</td>
</tr>
<tr>
<td>16: Remove $p_i$ from $PC$</td>
</tr>
<tr>
<td>17: end if</td>
</tr>
<tr>
<td>18: Remove the links in $EP(p)$ from $ET$</td>
</tr>
<tr>
<td>19: Update $PC$ based on $p$ using Algorithm 4</td>
</tr>
<tr>
<td>20: end while</td>
</tr>
<tr>
<td>21: Return $PC$</td>
</tr>
</tbody>
</table>

The purpose of Algorithm 1 is to find a set $PC$ of $p$-cycles such that every link in $ET$ is protected by some $p$-cycle in $PC$. Since some of the links in $ET$ may be protected by some existing $p$-cycles formed for existing multicast/unicast sessions, we first remove all links that can be protected by
reusing the existing p-cycles from $E_T$ (in line 2). Then we start the process of iteratively building p-cycles for every link $e$ in $E_T$. We use set $PC$ to store newly built p-cycles.

In line 6, we find a new p-cycle for link $e$ using Algorithm 2. Basically, Algorithm 2 finds a set of p-cycles that can protect $e$ and returns the p-cycle with the maximum ER. In lines 7-9, if $PC$ is not empty, we find an extended p-cycle for $e$ using Algorithm 3. Basically, Algorithm 3 finds the maximum ER p-cycle that is extended from a p-cycle in $PC$. After the for loop in lines 5-10 are executed, every link $e$ in $E_T$ has at most two candidate p-cycles, $p_{new}$ and $p_{ext}$. All these p-cycles are stored in set $Temp$.

In line 11-13, if the set $Temp$ is empty, then NULL is returned. This occurs when no p-cycles could be found due to lack of spare capacity in the network. As a result, $T$ cannot be protected.

In line 14-17, we choose p-cycle $p$ with the maximum ER from $Temp$ and add $p$ into $PC$. Furthermore, if $p$ is extended from a p-cycle in $PC$, we remove the original p-cycle from $PC$. Since $p$ may protect one or more links in $E_T$, we remove all these links from $E_T$ in line 18.

In line 19, Algorithm 4 is used to update $PC$ based on $p$. Specifically, Algorithm 4 combines $p$ with the other p-cycles in $PC$ to reduce the wavelength usage of the p-cycles without affecting the protection of the links in $E_T$.

When $E_T$ becomes empty, the algorithm returns $PC$, which contains a set of p-cycles that protect all the links in $E_T$.

B. Finding New p-Cycles

We now present Algorithm 2, which finds a new p-cycle for link $e = n_1 \rightarrow n_2$ in $E_T$. The new p-cycle contains link $n_2 \rightarrow n_1$ and therefore can protect $e$. The basic idea is to perform breadth-first searches from $n_1$ and $n_2$ at the same pace until these two searches arrive at some common node(s), indicating the finding of one or more p-cycles. Among the found p-cycles, the one with the maximum ER is returned.

The algorithm uses the following notations:

- $G_1$ and $G_2$: storing all nodes that have been reached by the breadth first searches from nodes $n_1$ and $n_2$, respectively. Initially, $G_1 = \{n_1\}$ and $G_2 = \{n_2\}$.
- $G_1'$ and $G_2'$: storing nodes which were added into $G_1$ and $G_2$ in the most recent step of the breadth first searches. Initially, $G_1' = \{n_1\}$ and $G_2' = \{n_2\}$.
- $PL_i$: node $n_i$'s parent list, storing the nodes through which $n_i$ is connected to $n_1$ or $n_2$. The first node in the list is called the primary parent and the other nodes in the list are called secondary parents.

The detail of the algorithm is explained as follows. To find a p-cycle that includes link $n_2 \rightarrow n_1$, there must be a free wavelength on this link. Line 2-4 check whether this condition is met. If not, NULL is returned to indicate that we cannot find a p-cycle to protect link $n_1 \rightarrow n_2$.

Before performing the Breadth First Searches (BFS), we remove link $(n_1, n_2)$ from $G$ in line 5 to make sure BFS does not consider this link.

Algorithm 2: Find a new p-cycle for link $e = n_1 \rightarrow n_2$

1: $G_1 = G_1 = \{n_1\}; G_2 = G_2 = \{n_2\}$
2: $PL_i = \phi$ for all node $i \in V$
3: if link $n_2 \rightarrow n_1$ has no free wavelength then
4: Return NULL
5: end if
6: for $\forall n_i \in G_1 \cap n_2$ do
7: Run breadth-first search for one step for $\forall n_i \in G_1$
8: Update $G_1$ and $G_1'$; Update $PL_j$ for $\forall n_j \in G_1$
9: Run breadth-first search for one step for $\forall n_i \in G_2$
10: Update $G_2$ and $G_2'$; Update $PL_j$ for $\forall n_j \in G_2$
11: until $G_1 \cap G_2 \neq \phi$ or $G_1' = \phi$ or $G_2' = \phi$
12: if $G_1 \cap G_2 = \phi$ then
13: Return NULL
14: end if
15: for $\forall n_j \in G_1 \cap G_2$ do
16: if $n_j$'s primary parent $\in G_1$ then
17: $tmpSet = G_2' - (G_1 \cap G_2)$
18: else
19: $tmpSet = G_1' - (G_1 \cap G_2)$
20: end if
21: for $\forall n_j \in tmpSet$ do
22: if link $(n_i, n_j) \in E$ then
23: $PL_i = PL_i \cup \{n_j\}$
24: end if
25: end for
26: end for
27: for $\forall n_i \in G_1 \cap G_2$ do
28: $Path_i = GetPath(n_i)$;
29: for $\forall secondary parent n_i' \in PL_i$ do
30: $Path_i' = \{n_i\} \cup GetPath(n_i')$
31: Combine $Path_i$ and $Path_i'$ to build cycle $C_i$
32: end for
33: end for
34: $p_{new} = the p-cycle with the maximum ER among all the p-cycles built
35: Return $p_{new}$
36: while $(n_i$'s parent list $PL \neq \phi$) do
37: Function GetPath($n_i$)
38: $Path = \{n_i\}$
39: $n_i = the primary parent in $PL_i$.
40: $Path = Path \cup \{n_i\}$
41: end while
42: Return $Path$
Lines 6-11 perform the BFS starting from $n_1$ and $n_2$, respectively, until (1) some node(s) is found to be in both $G_1$ and $G_2$ (i.e. $G_1 \cap G_2 \neq \emptyset$), which indicates at least one cycle has been found, or, (2) the BFS could not continue ($G_1' = \emptyset$ or $G_2' = \emptyset$), which means there is not enough spare capacity to create a p-cycle. During the BFS, we need to make sure that the link we use to reach a node has free capacity in the correct direction. Specifically, in line 7, for all $n_i \in G_1'$, to run BFS for one step to reach node $n_j$, we need to make sure there is free capacity in link $n_i \rightarrow n_j$. And in line 9, for all $n_i \in G_2'$, to run BFS for one step to reach node $n_j$, we need to make sure there is free capacity in link $n_j \rightarrow n_i$.

In line 12-14, NULL is returned if there is no common node between $G_1$ and $G_2$, which indicates no p-cycle could be found due to lack of spare capacity.

Lines 15-26 update the parent list for every $n_i \in G_1 \cap G_2$ as follows. If $n_i$’s primary parent is in $G_1$ (or $G_2$), then we consider every node $n_j$ in the set $G_2' - (G_1 \cap G_2)$ (or $G_1' - (G_1 \cap G_2)$). If there is an edge between $n_i$ and $n_j$ in $G$, then $n_j$ is added to $n_i$’s parent list. This is because the facts that $n_j$ has been reached in the most recent step of BFS from $n_2$ (or $n_1$) and there is an edge between $n_i$ and $n_j$ in $G$ indicate that there is a path from $n_i$ to $n_2$ (or $n_1$) via $n_j$. Note that whenever the BFS reaches a node $n$, $n$’s parent list is updated (in line 8 and line 10) to include the node via which $n$ is reached. This update occurs during the BFS and is different from the update in lines 15-26, which is done after the BFS stops. Specifically, in lines 15-26, $n_i$ is added to the parent list of $n_i$ because $n_i$ can be reached from $n_j$ using BFS, not because $n_i$ has been reached from $n_j$ using BFS. Due to the update done both during the BFS and after the BFS, every node $n_i \in G_1 \cap G_2$ has one primary parent via which it is connected to $n_1$ (or $n_2$), and a set of secondary parents via which it is connected to $n_2$ (or $n_1$).

In Lines 27-33, for every $n_i \in G_1 \cap G_2$, if its primary parent is in $G_1$ (or $G_2$), then we find its only path $Path_i$ to root $n_1$ (or $n_2$) via its primary parent and $|PL_i| - 1$ paths from $n_i$ to the other root $n_2$ (or $n_1$) via its secondary parents. After we get all these paths, we combine $Path_i$ with each of the other $|PL_i| - 1$ paths to form $|PL_i| - 1$ p-cycles. Note that all formed p-cycles have link $n_2 \rightarrow n_1$ as an on-cycle link. Finally, Lines 34-35 select the p-cycle $p_{new}$ with the maximum ER from all the formed p-cycles and return $p_{new}$.

Function $GetPath$ is used by Algorithm 2 and is defined in Lines 37-43. This function finds the path from the input node to one root ($n_1$ or $n_2$) by following the node’s primary parent step by step, and returns the path.

We illustrate Algorithm 2 using the example shown in Fig. 1. To find a new p-cycle for link $n_1 \rightarrow n_2$, we first remove link $(n_1, n_2)$ from the graph. Next, we perform BFS from $n_1$ and $n_2$ at the same pace. After one step of BFS, $G_1 = \{n_1, n_3\}$ and $G_2 = \{n_2, n_5, n_6, n_7\}$. After two steps of BFS, $G_1 = \{n_1, n_3, n_4\}$ and $G_2 = \{n_2, n_5, n_6, n_7, n_4, n_8\}$. We stop the BFS now since $G_1 \cap G_2 = \{n_4\} \neq \emptyset$. At this time, $PL_4 = \{n_3, n_4, n_5\}$, where $n_3$ is $n_4$’s primary parent and $n_5, n_6$ are $n_4$’s secondary parents. Next, $n_8$ is added to $PL_4$ according to lines 15-26. Thus, $n_8$ becomes the third secondary parent of $n_4$. Since $n_4$’s primary parent $n_3$ is in $G_1$, there is only one path $Path_4 = n_4 \rightarrow n_3 \rightarrow n_1$ from $n_4$ to $n_1$. On the other hand, since $n_4$ has three secondary parents ($n_5, n_6, n_8$), it has three paths to $n_2$, which are $Path_4' = n_4 \rightarrow n_5 \rightarrow n_2$, $Path_4'' = n_4 \rightarrow n_6 \rightarrow n_2$, and $Path_4''' = n_4 \rightarrow n_8 \rightarrow n_7 \rightarrow n_2$. Therefore, we can find three p-cycles for link $n_1 \rightarrow n_2$ by combining $Path_4$ with $Path_4', Path_4''$ and $Path_4'''$, respectively. The resulting p-cycles are $\{n_1 \rightarrow n_3 \rightarrow n_4 \rightarrow n_5 \rightarrow n_2 \rightarrow n_1\}$, $\{n_1 \rightarrow n_3 \rightarrow n_4 \rightarrow n_6 \rightarrow n_2 \rightarrow n_1\}$ and $\{n_1 \rightarrow n_3 \rightarrow n_4 \rightarrow n_8 \rightarrow n_7 \rightarrow n_2 \rightarrow n_1\}$. Among these three p-cycles, the one with the maximum ER will be returned by Algorithm 2.

C. Extending Existing p-Cycles

In this section, we present Algorithm 3, which finds a p-cycle that is extended from a p-cycle in $PC$ to protect a link $e = n_1 \rightarrow n_2$ in $ET$.

The basic idea of Algorithm 3 is as follows. For every p-cycle $p \in PC$, if $p$ can be extended to include link $n_2 \rightarrow n_1$ or include nodes $n_1$ and $n_2$, then extension is performed to produce a new p-cycle that can protect $e$, which is added to set $EPC$. After all p-cycles in $PC$ have been considered, the algorithm chooses from $EPC$ the p-cycle with the maximum ER and returns it.

Consider a p-cycle $p$ and a link $e$, we can extend $p$ to protect $e$ according to the following two cases.

Case I: One endnode of $e$ is already on $p$

The left graph in Fig. 2 shows an example, where link $e = n_1 \rightarrow n_2$ needs to be protected and one endnode of the link (i.e., $n_2$) is already on $p$. If $p$ can be extended to also include the other endnode of the link (i.e., $n_1$), then $e$ will be protected.
Algorithm 3 Find an extended p-cycle for link $e = n_1 \to n_2$

1: $EPC = \emptyset$;
2: for $(i \in p_j \in PC)$ do
3: \hspace{1em} for $(\forall n_k \text{ on } p_j)$ do
4: \hspace{2em} $u = n_k, v =$ next-hop node of $u$ on $p_j$;
5: \hspace{2em} while (no node on segment $u \to v$, excluding $u$ and $v$, belongs to the multicast tree) \& $(v \neq u)$ do
6: \hspace{3em} if neither endnode of $e$ is on $p_j$ then
7: \hspace{4em} $n_1 =$ the virtual node representing $e$
8: \hspace{4em} end if
9: \hspace{2em} Find Path$_1$: the shortest path from $u$ to $n_1$ that is link-disjoint with $p_j$
10: \hspace{2em} Find Path$_2$: the shortest path from $n_1$ to $v$ that is link-disjoint with $p_j$ and Path$_1$
11: \hspace{2em} Goto 16 if Path$_1$ or Path$_2$ cannot be found
12: \hspace{2em} Extend $p_j$ to cycle $p_{j \text{ ext}}$ by replacing segment $u \to v$ with the concatenation of Path$_1$ and Path$_2$
13: \hspace{2em} if $p_{j \text{ ext}}$ can protect $e$ then
14: \hspace{3em} Add $p_{j \text{ ext}}$ to $EPC$
15: \hspace{2em} end if
16: \hspace{2em} $v =$ the next-hop node of $v$ on $p_j$
17: \hspace{2em} end while
18: end for
19: end for
20: if $EPC = \emptyset$ then
21: \hspace{1em} Return NULL
22: end if
23: $p_{\text{ext}} =$ the p-cycle with the maximum ER among all p-cycles in $EPC$
24: Return $p_{\text{ext}}$

We extend $p$ in the following way. Selecting two nodes $u$ and $v$ on $p$ and two paths $u \to n_1$ and $n_1 \to v$ such that the following rules are followed. Then replacing segment $u \to v$ on $p$ with the concatenation of $u \to n_1$ and $n_1 \to v$.

(R1) Path $u \to n_1$ and path $n_1 \to v$ should be link-disjoint with each other, and also link-disjoint with $p$. Otherwise, replacing segment $u \to v$ with the concatenation of $u \to n_1$ and $n_1 \to v$ will not result in a p-cycle.

(R2) After the extension, link $n_1 \to n_2$ should not become an on-cycle link. Otherwise, if this link fails, an alternative path from $n_1$ to $n_2$ cannot be provided by the p-cycle. On the other hand, if link $n_1 \to n_2$ becomes a straddling link or link $n_2 \to n_1$ becomes an on-cycle link after the extension, then the extended p-cycle can protect link $e$.

(R3) There should be no multicast tree nodes appearing on segment $u \to v$. Otherwise, replacing this segment may cause some links on the multicast tree to lose protection.

In Algorithm 3, the for loop from line 3 to line 18 computes a set of p-cycles extended from an existing p-cycle $p_j \in PC$ to protect link $e = n_1 \to n_2$. Here, we consider every possible pair of $u$ and $v$ on $p_j$, one by one. Line 5 checks if there is any multicast tree node appearing on segment $u \to v$ to ensure rule (R3) is followed. Lines 9 and 10 check if the paths $u \to n_1$ and $n_1 \to v$ are link-disjoint, and if they are link-disjoint with $p_j$, to ensure rule (R1) is followed. Line 13 checks if rule (R2) is followed. All p-cycles extended from $p_j$ that have passed the above checks are put in set $EPC$.

**Case I: No endnode of $e$ is on $p$**

The right graph in Fig. 2 shows an example, where link $e = n_1 \to n_2$ needs to be protected and none of its endnodes is on $p$. We deal with this case by viewing $e$ as a virtual node and adding this virtual node into an existing p-cycle using the method described in Case I. Lines 6-8 in Algorithm 3 handles this case by setting $n_1$ to be the virtual node. In this case, the extended p-cycle must have link $n_2 \to n_1$ as an on-cycle link in order to protect $e$. That is, the direction of the extended p-cycle must be opposite to the direction of $e$.

In Algorithm 3, for every p-cycle $p_j \in PC$, a set of extended p-cycles are computed and put into $EPC$ based on the two cases described above. If no extended p-cycles can be found due to lack of spare capacity, then NULL is returned. This case is dealt with in lines 20-22. Otherwise, the p-cycle with the maximum ER among all p-cycles in $EPC$ is selected to protect $e$ and is returned in lines 23-24.

**D. Updating the p-Cycle Set PC**

In this section, we describe Algorithm 4, which is used by Algorithm 1 to update the p-cycle set PC after a p-cycle $p$ is added to PC. The update involves combining $p$ with the other p-cycles in $PC$ in a way that reduces the wavelength usage of the p-cycles while not affecting the existing protections of the links in $E_T$. The combining of the p-cycles continue repeatedly until no more combinations can be done.

Consider two p-cycles $p$ and $p_i$, we can combine them to create a new p-cycle according to the following two cases.

**Case I: $p$ and $p_i$ have one or more common edges**

In this case, $p$ and $p_i$ have one or more common edges with opposite directions. An example is shown in Fig. 3. In this example, $p = n_1 \to n_2 \to n_3 \to n_5 \cdots \to n_i \cdots \to n_m \to n_3 \to n_2 \to n_1$ and $p_i = n_3 \to n_5 \cdots \to n_j \cdots \to n_m \to n_1 \to n_k \cdots \to n_4 \to n_3$. If $p_i$ can protect all the tree edges that are protected by either $p$ or $p_i$ (i.e., $PE(p) \cup PE(p_i) \subseteq PE(P_c)$, then $p_i$ can provide the same protection with less wavelength usage. Since the new p-cycle $p_i$ is more efficient, we will add $p_i$ into $PC$ and remove $p$ and $p_i$ from $PC$.

**Case II: $p$ and $p_i$ have two common nodes**

When $p$ and $p_i$ have two common nodes, they can be combined to create a new p-cycle as shown in Fig. 4. In this Figure, $p = n_1 \to \ldots \to n_6 \to \ldots \to n_3 \to \ldots \to n_5 \cdots \to n_1$ and $p_i = n_3 \to \ldots \to n_2 \to n_1 \to \ldots \to n_4 \cdots \to n_1$. Two different new p-cycles can be obtained by combining $p$ and $p_i$. The first one is $p_{c1} = n_1 \to \ldots \to n_6 \to \ldots \to n_3 \to \ldots \to n_2 \to n_1$. The second one is $p_{c2} = n_1 \to \ldots \to n_k \to \ldots \to n_1 \to \ldots \to n_4 \to \ldots \to n_3 \to \ldots \to n_2 \to \ldots \to n_1$. If one of the two new p-cycles can
process until no more combinations could be done. replacing performed, we assign then it can provide the same protection as and we will add it into and wavelength usage. Since this new p-cycle is more efficient, we will add it into PC and remove p and pi from PC.

Algorithm 4: Update set PC based on the newly added p-cycle p

1: \(\text{Size}_{Be} = |PC|; \text{Size}_{Af} = 0\);
2: \(\text{while } \text{Size}_{Be} > \text{Size}_{Af} \text{ do}\)
3: \(\text{Size}_{Be} = |PC|, NC = \text{true}, i = 1\);
4: \(\text{while } NC \text{ and } i < |PC| \text{ do}\)
5: \(\text{if } p \text{ and } p_i \text{ have one or more common edges then}\)
6: \(p_c = \text{combination of } p \text{ and } p_i\)
7: \(\text{if } (PE(p) \cup PE(p_i) \subseteq PE(p_c)) \&\& \text{Simple}(p_c) \text{ then}\)
8: \(NC = \text{false};\)
9: \(\text{end if}\)
10: \(\text{end if}\)
11: \(\text{if } NC \text{ and } p \text{ and } p_i \text{ have two common nodes then}\)
12: \(p_{c1} = \text{First combination of } p \text{ and } p_i\)
13: \(p_{c2} = \text{Second combination of } p \text{ and } p_i\)
14: \(\text{if } (PE(p) \cup PE(p_i) \subseteq PE(p_{c1})) \&\& \text{Simple}(p_{c1}) \text{ then}\)
15: \(NC = \text{false};\)
16: \(p_c = p_{c1}\)
17: \(\text{else}\)
18: \(\text{if } (PE(p) \cup PE(p_i) \subseteq PE(p_{c2})) \&\& \text{Simple}(p_{c2}) \text{ then}\)
19: \(NC = \text{false};\)
20: \(p_c = p_{c2}\)
21: \(\text{end if}\)
22: \(\text{end if}\)
23: \(\text{end if}\)
24: \(\text{if } P_E \cap P_A \neq \emptyset \text{ then}\)
25: \(PC = PC - \{p\} - \{p_i\} + \{p_c\}\)
26: \(\text{end if}\)
27: \(i += 1;\)
28: \(\text{end while}\)
29: \(\text{Size}_{Af} = |PC|\)
30: \(p = p_c\)
31: \(\text{end while}\)
32: \(\text{Return } PC\)

In Algorithm 4, the while loop in lines 4-28 checks whether p-cycle p can be combined with another p-cycle pi in PC by considering the two cases described above. (Lines 5-10 consider Case I and lines 11-23 consider Case II.) We do not allow non-simple p-cycles where a link appears more than once on the p-cycle. So in line 7, 14, and 18, the combined p-cycle is checked to guarantee it is a simple p-cycle using function Simple(p). If a p-cycle pi can be found to combine with p, then the combined new p-cycle pc is added to PC, replacing p and pi (Lines 24-26). Once a combination is performed, we assign pc to p (in line 30) and repeat the above process until no more combinations could be done.

Fig. 3. Combining two p-cycles with one or more common edges.

Fig. 4. Combining two p-cycles with two common nodes.

E. Connection Release

When a multicast session terminates, we perform two steps to release the connection. First, we release the wavelengths used by the multicast tree T. Second, among those p-cycles protecting T, we tear down the ones that are no longer needed. (A p-cycle is no longer needed if all the multicast sessions protected by it have terminated.) Let pe be the p-cycle that protects link e on T. We check whether pe is still needed as follows. Suppose the set of all links protected by pe is PA(pe). We remove link e from set PA(pe). If PA(pe) becomes empty, pe is no longer needed and we tear it down. Otherwise, we keep pe since it is needed to protect some other multicast sessions.

III. NUMERICAL RESULTS

We conduct simulations to compare the performance of our proposed IpC scheme with another p-cycle based multicast protection scheme DpC [13], where each link has two pre-selected p-cycles. Two networks, the NSF network (Fig. 5) and the COST239 network (Fig. 6), are used in the simulations. The weight of each network link is set to 1. In each simulation run, a set of randomly generated multicast requests are loaded to the network to compare the performance of IpC and DpC. For each multicast request, the source node and the destination nodes are randomly selected. For NSF network, the number
of destination nodes are randomly generated in the range \([3, 6]\). For COST239 network, the number of destination nodes are randomly generated in the range \([2, 5]\). The algorithm for computing a multicast tree for a given multicast request is given in the appendix. The performance of the IpC and DpC are compared under two simulation settings: unlimited link capacity and limited link capacity. We also study the computation time of IpC for each multicast request under different traffic load. The simulation results are presented in the next three subsections.

A. Comparison of IpC and DpC under Unlimited Link Capacity

First, we compare the performance of the two algorithms when the network link capacity is set to infinity. In this case, all multicast requests can be satisfied. The performance metric we use is the total number of wavelength channels used by all the \(p\)-cycles for protecting the multicast sessions.

In Fig. 7, we show the performance of IpC and DpC under different traffic load for NSF network, where the traffic load varies from 1000 to 6000 multicast requests. The figure shows that IpC uses significantly less wavelength channels than DpC under all traffic loads. Specifically, IpC achieves a 24.5%-24.8% reduction in wavelength usage over DpC. The reason IpC performs better than DpC is two fold. First, IpC computes \(p\)-cycles on demand while DpC chooses \(p\)-cycles from pre-computed \(p\)-cycles. Second, IpC always selects high efficiency cycles while DpC uses short cycles which tend to have low efficiency since short cycles tend to have few straddling links.

In Fig. 8, we compare the performance of IpC and DpC for COST239 Network. The results again show that IpC is much more capacity efficient than DpC. The capacity saving of IpC over DpC ranges from 28.5% to 29.2%. For IpC, the number of wavelength channels used to protect the multicast trees is even less than that used by the multicast trees, which is not the case in NSF network. This is because COST239 network is denser than NSF network. Consequently, the \(p\)-cycles calculated by IpC have a higher probability of containing more straddling links which leads to better protection efficiency.

B. Comparison of IpC and DpC under Limited Link Capacity

Next, we compare the performance of the two algorithms when the capacity of each directed link in the network is set to 16. That is, every directed link supports 16 wavelength channels. In this case, some multicast requests may be blocked because either the multicast tree cannot be established or the \(p\)-cycles for protecting the tree links cannot be created due to lack of wavelengths. The performance metric we use is the reject ratio, which is defined as the number of rejected multicast requests to the total number of multicast requests.

In each simulation run, 5000 randomly generated multicast requests are loaded to the network and the reject ratio is computed at the end of the simulation run. The arrival of
multicast requests follows Poisson distribution with \( \lambda \) requests per second and the duration of the request is exponentially distributed with a mean of \( 1/\mu \). The traffic load measured in erlangs is \( \lambda/\mu \). For each traffic load, 10 simulations are conducted and the average reject ratio is plotted in Fig 9 and Fig 10.

In Fig 9, we compare the reject ratio of \( IpC \) and \( DpC \) under different traffic load in NSF network. The results show that \( IpC \) achieves lower reject ratio than \( DpC \) under all traffic loads. The reason \( IpC \) performs better than \( DpC \) is that \( IpC \) computes \( p \)-cycles on demand and prefers long \( p \)-cycles while \( DpC \) chooses \( p \)-cycles from short pre-computed \( p \)-cycles. When the capacity of the network link is limited, the long \( p \)-cycles used by \( IpC \) tend to spread the wavelength usage across the whole network. While the short \( p \)-cycles used by \( DpC \) tend to consume the wavelengths in areas of heavy traffic, which blocks future multicast requests. The maximum difference between the reject ratio of \( DpC \) and the reject ratio of \( IpC \) is 17.3%, which occurs at the load of 40 erlangs. The average difference between the two reject ratios is 10.9%.

In Fig 10, we compare the reject ratio of \( IpC \) and \( DpC \) under different traffic load in COST239 Network. Again, the performance of \( IpC \) is better than that of \( DpC \). In addition, the performance improvement of \( IpC \) over \( DpC \) is higher than that in NSF network. This is because the COST239 Network is denser so that there exists more high efficiency cycles which could be found by \( IpC \). Thus, the \( p \)-cycles selected by \( IpC \) will provide even larger advantage than the short \( p \)-cycles used by \( DpC \).

C. Computation Time of \( IpC \)

In this section, we study the computation time of \( IpC \) in milliseconds for each multicast request in both networks under different traffic loads. We use java language to implement the \( IpC \) on a computer with Intel 3.0GHZ CPU and 1.5GB of memory.

The results for NSF network is shown in Table I. The table shows the maximum and the mean computation time for a multicast request. It also shows the computation time for the first 10 multicast requests. As can be seen from the table, the first few requests generally have relatively large computation time. This is because in the beginning most links in a multicast tree cannot be protected by existing \( p \)-cycles since few \( p \)-cycle exists in the network. So we need to find new \( p \)-cycles to protect these tree links, which requires more time. As more \( p \)-cycles are created, more links in the new multicast request can be protected by the existing \( p \)-cycles. So the computation time of one request will generally decrease as the number of requests increases. For some requests, the computation time is 0 (i.e., less than 1 ms) because all or most links in the multicast tree can be protected by existing \( p \)-cycles.

The results for COST239 network is shown in Table II. The maximum and mean computation time for a multicast request and the computation time for the first 10 multicast requests are shown. Again we observe that the first few requests generally have relatively large computation time. Compared with the mean computation time in NSF network, the mean computation time in COST239 network is less because the average size of multicast requests in COST239 is smaller.

<table>
<thead>
<tr>
<th>Erlang</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>125</td>
<td>265</td>
<td>356</td>
<td>512</td>
<td>625</td>
<td>750</td>
<td>875</td>
<td>1000</td>
<td>1875</td>
<td>1888</td>
</tr>
<tr>
<td>Mean</td>
<td>125</td>
<td>265</td>
<td>356</td>
<td>512</td>
<td>625</td>
<td>750</td>
<td>875</td>
<td>887</td>
<td>1888</td>
<td>1401</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Traffic Load (Erlang)</th>
<th>( IpC )</th>
<th>( DpC )</th>
<th>( IpC )</th>
<th>( DpC )</th>
<th>( IpC )</th>
<th>( DpC )</th>
<th>( IpC )</th>
<th>( DpC )</th>
<th>( IpC )</th>
<th>( DpC )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.00</td>
<td>2.00</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
<td>0.10</td>
</tr>
<tr>
<td>1</td>
<td>1.99</td>
<td>1.99</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
<td>0.11</td>
</tr>
<tr>
<td>2</td>
<td>1.97</td>
<td>1.97</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>3</td>
<td>1.95</td>
<td>1.95</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>4</td>
<td>1.94</td>
<td>1.94</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td>5</td>
<td>1.92</td>
<td>1.92</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
<td>0.15</td>
</tr>
<tr>
<td>6</td>
<td>1.91</td>
<td>1.91</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
<td>0.16</td>
</tr>
<tr>
<td>7</td>
<td>1.90</td>
<td>1.90</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
<td>0.17</td>
</tr>
<tr>
<td>8</td>
<td>1.89</td>
<td>1.89</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
<td>0.18</td>
</tr>
<tr>
<td>9</td>
<td>1.88</td>
<td>1.88</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
<td>0.19</td>
</tr>
<tr>
<td>10</td>
<td>1.87</td>
<td>1.87</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
<td>0.20</td>
</tr>
</tbody>
</table>

IV. CONCLUSION

In this paper, we propose an Intelligent \( p \)-Cycle (\( IpC \)) scheme to provide \( p \)-cycle based protection for dynamic multicast sessions. The main feature of \( IpC \) is that it dynamically
computes high-efficiency p-cycles to protect multicast sessions as they arrive so that spare capacity is used efficiently. The capacity efficiency is further improved by reusing existing p-cycles to protect a new multicast session and combining existing p-cycles whenever possible. The numerical results show that IpC has significantly better performance than DpC, which is an existing p-cycle based protection scheme. In addition, IpC performs better in denser networks since denser networks contain more high efficiency cycles which could be utilized by IpC.

ACKNOWLEDGMENT

This work is supported by the National Science Foundation under award #ANI-0237592.

REFERENCES


APPENDIX

Algorithm 5 Find a Multicast Tree for a Multicast Session

In each round, we find the shortest path SP between any node in set T, and any node in set UTD. Then we add all nodes on SP to T, add all edges on SP to P, and remove the last node on SP from UTD. When UTD becomes empty, we have found a multicast tree for R, where the edges of the tree are stored in P.

<table>
<thead>
<tr>
<th>Erlang</th>
<th>10</th>
<th>20</th>
<th>30</th>
<th>40</th>
<th>50</th>
<th>60</th>
<th>70</th>
<th>80</th>
<th>90</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>125</td>
<td>172</td>
<td>109</td>
<td>109</td>
<td>187</td>
<td>141</td>
<td>219</td>
<td>125</td>
<td>109</td>
<td>125</td>
</tr>
<tr>
<td>Mean</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>125</td>
<td>65</td>
<td>109</td>
<td>109</td>
<td>187</td>
<td>141</td>
<td>219</td>
<td>125</td>
<td>78</td>
<td>125</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>172</td>
<td>32</td>
<td>32</td>
<td>16</td>
<td>47</td>
<td>16</td>
<td>63</td>
<td>94</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>62</td>
<td>98</td>
<td>43</td>
<td>42</td>
<td>109</td>
<td>63</td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td>47</td>
<td>63</td>
<td>32</td>
<td>94</td>
<td>15</td>
<td>15</td>
<td>78</td>
<td>16</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>31</td>
<td>0</td>
<td>62</td>
<td>62</td>
<td>0</td>
<td>0</td>
<td>31</td>
<td>15</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>110</td>
<td>47</td>
<td>0</td>
<td>63</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
<td>15</td>
<td>31</td>
<td>15</td>
<td>15</td>
<td>31</td>
<td>31</td>
<td>16</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>47</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>16</td>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>16</td>
<td>16</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>15</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>