Implementing A Constraint-based Shortest Path First Algorithm in Intelligent Optical Networks

MAXIMIZING CARRIER PERFORMANCE

Table of Contents

1.0 Introduction

Traditional SONET based optical networks are by design not only constructed in ring topology but also provisioned and managed via the management plane or manual process since early designers do not anticipate heavy demands for those high-capacity optical networks. As the primary means to carry traffic today in the carrier networks, such optical networks have increased in size significantly due to the exponential growth of Internet over the last decade. This has in turn made huge impact on the carriers in terms of both added network operation costs and newly encountered difficulties in managing large networks. The solution lies in intelligent control and management, as evidenced by the widely adoption of Generalized Multi-Protocol Label Switching (GMPLS) in the industry in recent years. The GMPLS protocol suites extend the IP routing and signalling paradigm into optical networks by taking into account attributes associated with optical networks.

Under the GMPLS framework, traffic-engineering extensions have been added to the IGP protocols such as OSPF or IS-IS so that optical resources represented as traffic-engineering link attributes can be distributed across the network. Each network element then is able to construct a traffic-engineering database (TED) using the information received from the IGP. This database not only provides the network-wide inventory but also is critical in determining an optical path from one node to another. One of the key applications in GMPLS enabled optical networks is to intelligently route the traffic in such a way that it both accommodates service requests and adapts to the current status of the network to optimize resource utilization. Optical paths can be automatically set up and torn down using dynamic signaling protocols such as RSVP. Since optical paths are typically created subject to multiple constraints,the algorithm to determine an optimal path under such constraints, known as CSPF in the literature, becomes a vital part of GMPLS design and implementation. By its name, CSPF – Constraint-based Shortest Path First is an extension to the traditional shortest-path (SPF) algorithm with a set of constraints attached. The implementation of an effective and efficient CSPF algorithm is the subject of this paper.

There are several key differences between CSPF and traditional SPF. From the complexity point of view, traditional SPF only has a polynomial-time complexity and thus it is a relatively much easier problem to solve. Among several algorithms known in the literature, the Dijkstra algorithm is the most well known in solving the SPF problem. CSPF on the other hand is an intrinsically harder problem because of the introduction of multiple constraints. With only one additive constraint, for example, it is proved to be an NP-complete problem. Thus a heuristic may be used to look for a near-optimal solution.

In addition, traditional SPF used in IGP routing protocols such as OSPF and IS-IS computes a hop-by-hop forwarding table for the best-effort traffic using the link-state information. The path taken is completely determined by the destination. The computation result is consulted in making a packet forwarding decision. In contrast, for CSPF applications in SONET-based optical networks, the goal is to establish dedicated TDM optical paths or connections between nodes. Each path is dynamically requested, and may have dependencies on other paths that are in progress for protection and restoration purpose. The path taken could be different each time when the number or values of the constraints associated with the path request change.

Another aspect is related to traffic engineering. Since traditional SPF is for destination-based routing, the paths to a destination are determined by the SPF calculation and stay the same regardless of the load on individual links. As a result, links that are most used by sources to reach a destination tend to get congested the most while bandwidths on other links may be left under-utilized. Since CSPF is essentially designed with traffic engineering in mind, it can adapt to the network condition and select paths to route around overloaded portion of the network at the cost of increased complexity in dealing with multiple constraints.

Finally, in GMPLS networks, it's crucial to provide a protection/restoration mechanism for fast service restoration in the presence of a network failure. CSPF plays a key role in selection of

restoration paths to achieve mesh protection, as SONET-based optical networks migrate from ring topology to mesh topology for the benefits of increased resource utilization[1].

This paper is organized as follows: In section II, we describe the major constraints that are currently proposed in GMPLS networks and demonstrate challenges in handling those complex constraints during the path computation. In section III, we present a brief overview of some candidate CSPF algorithms that we have considered in our implementation and explain the rationale why a particular algorithm is selected to meet our design requirements and objectives. In section IV, we describe the algorithm and explain how it can be modified to overcome the aforementioned design issues.

2.0 Constraints in GMPLS Networks

In this section, we discuss the constraints typically considered in the GMPLS networks. In general, there are two categories of constraints: link constraints and path constraints. The link constraints or attributes are those associated with an optical link in the network. Those constraints are dynamically distributed by the IGP routing protocol as previously. They could change over time during the lifetime of the link. Listed below are some of the most common link constraints currently considered in the GMPLS framework.

- Interface switching capability: In the context of GMPLS, optical links are considered to be logical interfaces, each interface interconnected by two network elements or nodes. Each interface may have different switching capabilities, such as packet switch capable or TDM switch capable. This constraint requires that all the links on the path have the same interface switching capability. Each path must initiate and terminate at the same level of GMPLS LSP hierarchy.
- Bandwidth: Each interface dynamically advertises the amount of bandwidth available at any given time.
- Protection type: The link protection type represents the protection capability that exists for a link. Those types defined include: Extra Traffic, Unprotected, Shared, Dedicated 1:1, Dedicated 1+1 and Enhanced.
- Traffic engineering (TE) metric: This constraint is related to the cost of the link if it is used for a path. The cost of the path is the sum of TE metrics on all the links along the path. Strictly speaking, this is not a constraint but an object function that CSPF aims to minimize.
- Shared Risk Link Group (SRLG): A set of links can form a SRLG group if they share a resource whose failure may affect all the links in the group. Each link or interface can associate with one or more SRLG groups. The constraint that two paths are SRLG-disjoint can be used to describe protection/restoration requirement. It is worth noting that even though SRLG is defined at the link level, it can be applied to the node level with little effort. For example, if we define the entire links incident to a node as a single SRLG group, SRLG-disjointness constraint effectively becomes a node constraint, which can be used for computing a node-disjoint backup path.
- Link resource class: Links can be associated with some resource class attributes (also called link colour) that can be used to instruct CSPF to either include or exclude them from the path to achieve policy enforcement.

The path constraints on the other hand are concerned with path level requirements. In addition to the same constraints defined at the link level, which are equally applied to the path level, there are a few path-specific constraints. They are defined below:

- Hop count limit: Specifies the upper bound on the number of hops the path can take.
- Loose explicit route: Specifies some intermediate nodes or links that the path has to traverse. This is very useful in several cases where, for example, the operator of a network needs to assert some degree of control over which trail the path has to follow.

[1] We will use GMPLS networks and intelligent optical networks interchangeably in this paper.

- Protection and restoration: According to [15], there are three levels of recovery scheme in terms of recovery scope:
  - Path level recovery: a complete disjoint path is resorted upon any failure on the primary path.
  - Segment level recovery: a path that spans a failed segment of the primary path is used for recovery.
  - Span level recovery: only a path that spans a failed link is used for recovery.

  Note that we can view the path level and span level recovery as special cases of segment level recovery, since either the whole path or a single link can be treated as a segment.
- Priority and pre-emption: These attributes facilitate a way to specify relative importance of paths during set-up or pre-emption. The differential treatment of paths is important to provide different levels of service when the resource is limited.

As can be seen, the number of constraints is large and could change over time. If each constraint appears to require a special treatment, any solution to the problem does not seem to be scalable. From mathematical standpoint, however, there exists a degree of commonality among those constraints that can classify most of them into two categories:

- Boolean, which requires a particular parameter over each link along the path to satisfy a certain condition. Interface switching capability, bandwidth, protection type, and link resource class, are examples of boolean constraints. This kind of constraints is relatively easy to deal with since we can simply prune the links that do not meet the constraints from the path computation.

- Additive, which requires the sum of a particular parameter over the links along the path to be within certain limit. Among the constraints described above, only hop limit and TE metric are additive constraints. This kind of constraints is the major reason that sets CSPF apart from traditional SPF and has to be addressed by any viable CSPF implementation.

There are some important exceptions to this classification. Although SRLG can be considered as a boolean link attribute, in the context of protection and restoration, additional consideration has to be given. In the case that a protection path and a working path need to be SRLG-disjoint, this constraint cannot be simply treated in the same way as an ordinary boolean constraint. It concerns essentially with the relation between two paths. Unfortunately, the problem of finding two or more shortest paths between a pair of nodes in a network is shown to be also NP-hard in the literature. One way to get around this problem is to treat the working and protection path request in two separate and independent steps. In the first step, a feasible path is calculated as the working path. Then in the second step, by using the SRLG of the first path as an extra boolean constraint (i.e. eliminating all links that share SRLGs with those in the working path), we can calculate the second path as the protection path. The problem with this approach is that it may fail to find a solution even if one does exist. This can be best illustrated by the following example.

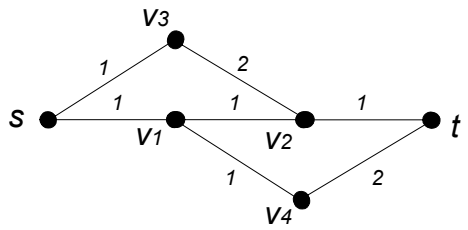

Fig. 1. Multiple Path Calculation

Here the labels on the links represent TE metrics. We are asked to compute two node-disjoint paths from node *s* to node *t*. If one shortest-path is computed at a time, we will get the first path (s, v_1, v_2, t). Then when we try to compute the second one, we will find there is no feasible path between *s* and *t* anymore and hence fail to meet the request. However there is clearly a solution to the problem: (s, v_3, v_2, t) and (s, v_1, v_4, t).

Another exception is loose explicit route path constraint. This constraint cannot be simply treated as boolean because it is not tied to any attribute of a single link. So pruning is not possible in this case. One may think that we could treat each segment (we call the part of the path between two consecutive nodes in the explicit route a segment) of the explicit route one by one. Each segment as a sub-problem could be solved separately and the solutions could be concatenated into one final path. This approach becomes very difficult to proceed in the presence of additive constraints, since we have to divide the additive constraints among different segments. If the constraint is non-discrete, it might not be possible to divide such constraint. Even if the constraint is discrete, it's far from simple how to determine the division without enumerating all the possibilities in order to reach a best solution. If multiple additive constraints are present, the complexity of combinations increases dramatically. This presents another serious challenge to the CSPF design and implementation.

In this paper, we present and describe an implementation of CSPF that attempts to resolve the challenging issues mentioned above. One additional design objective of this paper is to find a CSPF solution that can be computed in real time. This is a very practical requirement in future GMPLS networks, where each path has to be selected within a reasonably amount of time. Secondly, we make no algorithm assumption that traffic matrix between pairs of nodes in the network is available. Furthermore, if this kind of information does become available, it can be incorporated into our implementation by formulating it as some additional constraints after pre-processing [2].

3.0 CSPF Algorithm Selection

As discussed previously, the key to the CSPF design is to select an algorithm that can meet the multiple constraints discussed in the previous section. There are several criteria that influence the selection of a CSPF algorithm. They are:

- Efficacy, which concerns the ability of the algorithm to find the best possible solution. In case that the best or optimal solution is impractical to find, how close is the proposed solution to the best?
- Efficiency, which is related to the computational complexity of the algorithm. Can the algorithm find a solution within an acceptable time for a given set of practical constraints and reasonably network size?
- Capability of constraint treatment, which is concerned with the ability and flexibility of an algorithm to handle diverse set of constraints. How many (additive) constraints and what kind can the algorithm handle? Is the algorithm flexible enough so that new constraints can be easily incorporated?

Several algorithms have been proposed in the literature in the context of QoS routing. Here we briefly overview the ideas behind them and give a comparison based on the selection criteria outlined above. There are basically two major techniques used in the design of these algorithms:

The first category of techniques is based on the extension of the best-known Dijkstra algorithm or its variants that are widely used in constraint based routing, to handle more than one constraint. Some algorithms in this category are also extended in various ways to accommodate the additive constraints, which are the major obstacles in constraint based routing. These approaches are similar in that they compute shortest paths using Dijkstra algorithm based on metrics derived from the constraints and then utilize the result as a look-ahead pointer in further facilitating the searching and pruning process. An example of such algorithms in this category is

H_MCOP [3], which approximates the original problem by aggregating multiple additive constraints into one non-linear objective function and constructs the look-ahead pointer based on a linear combination of constraints. The algorithm is approximate by nature and is not suitable if one or more protection or backup path is required. Another algorithm example in this category is the so-called A*Prune algorithm [1], which will be explained in the next section.

The second category of techniques shares the common assumption that we have some prior knowledge of traffic profile of the network and applies some form of flow-based algorithms to select the path and allocate the network resource in an efficient way. MIRA [4] for example, assumes the knowledge of traffic load of ingress-egress pairs in the network. To decide the preference in selecting a path, it uses the amount of max-flow between other pairs reduced by the path as a cost measure in order to avoid creation of bottlenecks. The algorithm focuses on optimizing traffic distribution across the network but suffers the limitation of not being able to handle additive constraints. Another flow-based routing technique, called Profile-Based Routing [2], converts the traffic profile into a multi-commodity network flow problem and by solving it, pre-allocates the network resource into several classes. This step can be performed offline. Then as each path request arrives, it is classified and routed only within the resources allocated to its class. In this way, as long as the traffic profile is accurate and the requests are conforming to the classification, a feasible solution can be found. This algorithm can achieve very good network utilization. The offline step can be used by other algorithms such as A*Prune before servicing individual path requests.

A*Prune algorithm, proposed in [1], is capable of finding optimal solutions for multiple additive path constraints. Our implementation shows that the algorithm best suits our needs except for its computational complexity. However in [1], the authors claim that for networks containing hundreds of nodes with reasonable density, the experimental result is comparable to the current best-known approximation algorithms in term of run time. In early deployment of GMPLS networks where CSPF usually operates within one OSPF traffic engineering area, this does not appear to be a major problem in such network typically with no more than several hundred nodes. In practice, an effective CSPF implementation not only has to have a reasonable computational complexity but also has to be flexible enough to handle a variety of practical constraints encountered in real networks. A*Prune is among the few that can claim this property. As will be shown in the next section, the algorithm can be easily modified to solve protection/backup path computation and explicit route handling, which most of the other algorithms fail to address. All these considerations make A*Prune the algorithm of our choice.

4.0 CSPF Implementation Using A*Prune Algorithm

4.1 A*Prune Algorithm

Now we give a description of A*Prune algorithm. In the remaining of the section, we describe how to extend this algorithm to solve some of the issues raised in section II.

Figure 2 shows a skeleton pseudo-code of A*Prune algorithm which, combined with the extensions discussed later in the section, serves as the basis of our implementation in the next generation optical systems.

Input:
G = *(V, E),* a graph with node set *V* and edge set *E*.
(*s*, *t*): a node pair with source *s* and destination *t*.
K: number of paths to be found.
R_a: number of additive constraints.
R_b: number of Boolean constraints.
$C_a(i)$*:* the *ith* additive constraints.
$C_b(i)$*:* the *ith* Boolean constraints.
$w_k(e)$*:* weight related to *kth* constraint associated to link $e \in E$.
m(e): TE metric of link $e \in E$.

Pre-computation:
For $\forall v \in V$, and $\forall r \in (1,2,...,R_a)$, compute:
$D_r(v,t)$: length of Dijkstra path from v to t associated with rth additive constraint.

Initialization:
```
k = 0;                          // number of feasible paths found so far.
W_r(p(s,s)) = 0; 1 ≤ r ≤ R_a.   // path's current constraint value,   // used for additive constraint
                                   pruning
M(p(s,s)) = 0;                  // path's TE metric, objective function
pathHeap = {p(s,s)};
CSP_list = {};
```

Expanding and Pruning:
```
while (k ≤ K and pathHeap != ϕ) {
  p(s,u) = extract_min(pathHeap);
  if (u == t) {
    insert p(s,u) into CSP_list;  .
    k = k+1;      // Find a feasible path, add it to the list
    continue;     // and continue with next path
  }
  foreach u's outgoing edge e=(u,v) {    // Go through each     // outgoing link...
    if (v ∈ p(s,u)) {             // Check loop existence and prune // looping paths.
      continue;
    }
    for (i =1; i ≤ R_b; i++) {        // Boolean constraint pruning.
      if (test(C_b(i), (u,v)) == fail) {
        break;
      }
    }
    if (i ≤ R_b) {
      continue;
    }
    p(s,v) = append(p(s,u), (u,v));       // Expand the path.
    for (i=1; i≤R_a;i++) {        // Update the additive constraints.
      W_i(p(s,v)) = W_i(p(s,u))+w_i(u,v);
    }
    M(p(s,v)) = M(p(s,u)) + m(u,v);      // Update the TE metric
    for (r=1; r≤R_a; r++) {             // Additive constraint pruning.
      if (W_r(p(s,v)) + D_r(v,t) > C_r) {
        break;
      }
    }
    if (r ≤ R_a) {
      continue;
    }
    insert_heap(pathHeap, p(s,v));   // Add the new path into // path Heap.
  }
}
```

Fig. 2. A*Prune Algorithm

The algorithm consists of two major steps: pre-computation and path expanding/pruning. To deal with the additive constraints, it performs a pre-computation of their associated Dijkstra

distances from node *v* to *t*, $D_r(v, t)$. This pre-computation can be done in the background and stored for use by multiple path computations as long as the topology remains unchanged.

In expanding/pruning stage, A*Prune algorithm, instead of maintaining a heap of nodes as in Dijkstra algorithm, keeps a heap of paths all starting from the source node *s*. The algorithm maintains a heap data structure *pathHeap* that holds feasible partial paths found up to now. The heap initially contains the path *p(s, s)*, which consists of the single node *s*. Looking at the shortest path *p(s, u)* contained in the heap, the algorithm adds the path to the list of feasible paths *CSP_list* if the path already reaches destination node *t*. Otherwise it attempts to expand the path using each of *u*'s outgoing links *(u, v)* to generate a path *p(s, v)*. It prunes the path either when a loop is detected or a boolean constraint is violated. For an additive constraint, it combines the associated Dijkstra distance to node *t* - $D_r(v, t)$ obtained in pre-computation and the path's constraint value $W_r(p(s, v))$ to have an estimate of the projected distance and compares it against the constraint C_r. If the projected distance is greater than the constraint, any path expanded beyond this point will violate the constraint such that *p(s, v)* can be safely pruned or removed from further consideration. This process continues until sufficient number of paths has been found for the objective. Of course, it is possible that it may take too long to find a feasible path or an appropriate path subject to constraints may not exist. Although the algorithm guarantees to terminate, a limit or bound of iterations is typically imposed so that it can simply abort further operation. In practice, this happens usually because a feasible path does not exist.

4.2 Handling Recovery Path Calculation

There are some algorithms that are developed to address the protection/restoration problem [5][6][7]. However, they either suffer the inability of finding a feasible solution or the limitation on the constraints that they can handle.

Given that A*Prune algorithm can calculate an unlimited number of constraint-based paths, we can modify the algorithm to select a pair of feasible working (primary)/protection (backup) paths among those feasible paths by introducing a post-processing step. Every time a feasible path is achieved, it is then compared against each feasible path already calculated to see if there is a pair of paths satisfying, for example, the SRLG-disjoint property. The algorithm can terminate if such a pair has been found. Otherwise, it saves the new path and continues the process. This technique can be generalized to the case that requires computing more than one protection path. The problem of choosing among a set of feasible paths *k* SRLG-disjoint paths can be reduced into a graph problem as follows:

> A vertex in the graph denotes each feasible path. There is an edge or link between two vertices if the two corresponding paths are SRLG-disjoint. The problem then becomes to find a *k*-clique in the resulting graph.

For a small *k* and a small number of feasible paths under consideration, the problem is manageable.

Although path level recovery from source to destination is desirable in that it is relatively simple for restoration, its major disadvantage compared with segment or span recovery is its difficulty and less likelihood of finding a backup path. In the presence of additive constraints, it might be impossible to find such a path although with segment recovery, a solution does exist. For illustrative purpose, let's consider the following example.

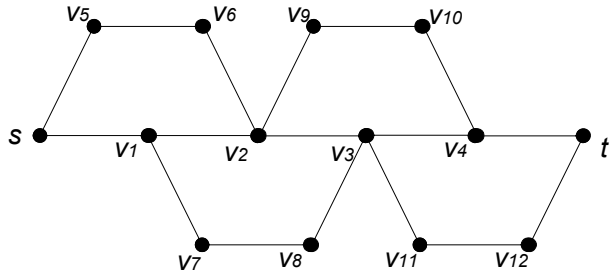

Fig. 3. Segment Level Recovery

In Figure 3, the objective is to find a protected path from *s* to *t* with a hop count limit of *6*. It is easy to see no two disjoint paths from *s* to *t* satisfy the requirement. However, using segment level recovery, we can select the primary path as ($s, v_1, v_2, v_3, v_4, v_5, t$), and use paths ($s, v_5, v_6, v_2$), ($v_1, v_7, v_8, v_3$), ($v_2, v_9, v_{10}, v_4$) and ($v_3, v_{11}, v_{12}, t$) to protect segment ($s, v_1, v_2$), ($v_1, v_2, v_3$), ($v_2, v_3, v_4$) and ($v_3, v_4, t$) respectively.

For segment level recovery, however, the task of CSPF is to find a set of paths between pairs of nodes along the primary path so that, all the paths are disjoint from the primary path; each link/node is protected by one of them. By pruning the nodes and links within the segment in the primary path, our implementation uses A*Prune algorithm to find a constrained-base path to protect the segment. Note that each additive constraint needs to be adjusted to accommodate the difference between requested constraint value and the actual value of the primary path. The remaining issue is how to select the segments that protect the primary path so that the total cost of restoration paths is minimized. The selection process also depends on whether link/node protection, or other SRLG-disjointness is desired. We can use the technique in [7] that applies the Dijsktra algorithm on an auxiliary graph to obtain an approximate solution.

4.3 Loose Explicit Route Issues

Normally CSPF computes the complete path from source to destination. However the user may specify some intermediate nodes or links of the path. As shown previously, this constraint cannot be fit into additive or boolean category. Treating each segment of the path separately also fails to address additive constraints. There seems to be no apparent solution for most of the proposed algorithms. The A* Prune algorithm, however, with a little modification, can solve this problem in a natural way.

Our solution in this case is to treat the constraint as a path-level boolean constraint instead of link level one. Before a new path is added into the path heap, a check whether the path violates the explicit route constraint is performed done and the path will be pruned if it reaches a node/link in the explicit route without passing through the one that precedes it. This ensures that the paths obtained in the end follow the loose explicit route in addition to satisfying other constraints.

5.0 Conclusion

As illustrated in this paper, CSPF is a critical and challenging design issue in the GMPLS enabled optical networks. It is often considered as the key vendor implementation differentiator in building next generation intelligent optical systems. This paper not only presents an overview of the CSPF deign issues related to such optical networks but also described a particular design and implementation of such an algorithm.

There are several areas of this work that can be extended for future study. First of all, the worst-case complexity of the A*Prune Algorithm appears to be high, and requires further simulation and testing under a reasonably sized network environment. Secondly, there is still room for improvement in handling protection/restoration. One nice property of our implementation is that it generates optimal partial paths in the searching process. These paths

may be exploited to provide more efficient implementation for segment level recovery. Finally, the list of optical constraints used in this paper is not necessarily exhaustive and will continue to develop, for example, how to deal with the path pre-emption.

6.0 References

[1] G. Liu and K. G. Ramakrishnan, "A*Prune: An Algorithm for Finding K Shortest Paths Subject to Multiple Constraints," *INFOCOM 2001.*

[2] S. Suri, M. Waldvogel, D. Bauer and P. R. Warkhede, "Profile-Based Routing and Traffic Engineering," *Computer Communications, 2002.*

[3] T. Korkmaz and M. Krunz, "Multi-Constrained Optimal Path Selection," *INFOCOM 2001.*

[4] K. Kar, M. Kodialam and T.V. Lakshman, **"**Minimum Interference Routing of Bandwidth Guaranteed Tunnels with MPLS Traffic Engineering Applications," *IEEE Journal on Selected Areas in Communications,* vol. 18, No. 12, December 2000.

[5] G. Li, D. Wang, C. Kalmanek and R. Doverspike, "Efficient Distributed Path Selection for Shared Restoration Connections," *INFOCOM 2002*.

[6] K. Kar, M. Kodialam and T.V. Lakshman, "Routing Restorable Bandwidth Guaranteed Connections using Maximum 2-Route Flows," *INFOCOM 2002.*

[7] Y. Bejerano, Y. Breitbart, A. Orda, R. Rastogi and A. Sprintson, "Algorithms for Computing QoS Paths with Restoration," *INFOCOM 2003*.

[8] R. Hassin, "Approximation Schemes for the Restricted Shortest Path Problem," *Mathematics of Operations Research*, 17(1):36-42, Feburary 1992.

[9] M.R. Garey and D.S. Johnson, "Computers and Intactability," Freeman, San Francisco, 1979.

[10] J. Suurballe, "Disjoint Path in Networks," *Networks*, 4:125-145, 1974.

[11] K. Kompella and Y. Rekhter, "Routing Extensions in Support of Generalized MPLS," Draft-ietf-ccamp-gmpls-routing-05.txt, August 2002.

[12] D. Awduche, J. Malcolm, J. Agogbua, M. O'Dell and J. McManus, "Requirement for Traffic Engineering Over MPLS," RFC 2702, September 1999.

[13] E. Mannie, "Generalized Multi-Protocol Label Switching Architecture," draft-ietf-ccamp-gmpls-architecture-05.txt, March 2003.

[14] J. Lang and B. Rajagopalan, "Generalized MPLS Recovery Functional Specification," draft-ietf-ccamp-gmpls-recovery-functional-00.txt, January 2003.

[15] E. Mannie et al., "Recovery (Protection and Restoration) Terminology for GMPLS," draft-ietf-ccamp-gmpls-recovery-terminoloty-01.txt", November 2003.

7.0 Revision History

Revision	Date	Author	Comments
0.1	5/05/03	Biao Gao, Yibin Yang and Charles Chen	First Draft.