Highly Efficient Spare Capacity Planning for Generalized Link Restoration

Srinivasan Krishnamurthy*, R. Chandrasekaran*, S. Venkatesan* and Milind Dawande†
*Department of Computer Science †School of Management
The University of Texas at Dallas, Richardson, TX 75083–0688
Email: ksrini@student.utdallas.edu, {chandra, venky, milind}@utdallas.edu

***Abstract*— We consider the problem of spare capacity allocation in mesh networks for link restoration. Only single link failures are considered and restored traffic is not split across multiple paths. In our model, links that are not part of the original transport network can also be used for restoration. This model, which is a generalized version, is different from the traditional model studied in literature. We present two heuristics to solve this problem and compare their performance with known lower bounds and the optimal solution. In numerous simulation experiments on randomly generated graphs containing up to 100 nodes, the heuristics often produced solutions within 3% of the lower bound for sparse graphs. The Integer Linear Programming solutions suggest that the heuristic solutions are closer to optimal than indicated by the deviation from the lower bounds. The results are optimal or near optimal (within 0.5% of the optimum values) for small graphs containing up to 8 nodes and 18 edges.**

I. Introduction

Restoration in mesh networks involves switching the disrupted traffic to alternate path(s). For restoration to be possible, sufficient spares must exist on the links that constitute the alternate path(s). Mesh-based restoration can be performed using Link Restoration or Path Restoration schemes. In Link Restoration [2], [5], [10], [12], traffic on the failed link is rerouted over a replacement path connecting the end-points of the failed link. In Path Restoration [4], [6], [8], each path disrupted by the link failure is considered separately and rerouted over an alternate path between the source and destination of that path. Link Restoration is easy to implement since only the two end nodes of the failed link are affected and the rerouting is localized but expensive in spare capacity [9]. In Path Restoration rerouting is done over a large area but is economical in spares. A comparative study of the two approaches is presented in [9]. Various other survivability schemes are discussed in [13].

We consider spare capacity assignment for link restoration to handle single link failures. Spares are on a separate restoration network. We assume that links that are not part of original network can be used. They may be links that have zero working capacity or links that are unused because of changing traffic patterns. This is a generalized version of the Link Restoration problem often considered in literature. We also assume that traffic splitting [11] is not allowed. To the best of our knowledge, there is no previous work on this generalized link restoration problem.

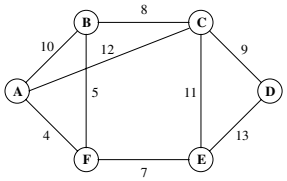


Fig. 1. Working Capacities on the links in Sample Network

II. System Model

A wired, telecom network consists of nodes inter-connected by an arbitrary number of point-to-point bi-directional links (copper/optical). Parallel links and self-loops are assumed to be absent. The working capacity of a link is the aggregate of traffic on multiple node-to-node connections that employ the link. The additional capacity made available on a link for restoration is the link's spare capacity. We have assumed a uniform cost model: the cost of one unit of spare capacity on all links is a constant. The network is modeled as an undirected graph $G(V, E)$ where V is the set of vertices (nodes), E is the set of edges (links) with working capacity $w_{kl} > 0 \; \forall \; (k, l) \in E$. Figure 1 shows working capacity requirements for a sample network. The link restoration problem is formally defined as:
Problem: Obtain an undirected restoration graph, $R(V, E_1)$ that serves as a link restoration network with capacity, $c_{ij} \; \forall \; (i, j) \in E_1$ such that R has minimum total capacity among all such networks. Unless otherwise stated, capacity refers to spare capacity. The terms "link" and "edge" have been used interchangeably in the rest of the paper.
Description/Assumptions: The failed link cannot be part of the restoration path for that link. Links that are not present in G are assumed to have zero working capacity in G. Note that if any of these links fail, we do not have to restore any traffic and the original network is fully operational. We expect that the total capacity of the restoration network would be minimized if we utilized the zero working capacity links. Hence, we refer to these links as *preferred* links while the links in G are referred to as *non-preferred* links.
Problem Status: We conjecture that the problem is NP-complete [3] based on that fact that if we are restricted to using only links with non-zero working capacities in the original graph, the problem is provably NP-complete [7]. The Integer Linear Program (ILP) formulation described in [7] may have an exponential number of constraints in $O(n^4)$ variables in the

worst case. Standard programs such as $CPLEX^{®}$ that are used to solve ILPs will take a substantial amount of time and may not be very useful in practice. Since the problem is potentially "hard", fast and efficient heuristics or approximation algorithms with performance guarantees are very helpful. In the following sections, we present two heuristics to solve the problem. Simulation results suggest that our heuristic solutions are close to optimal values.

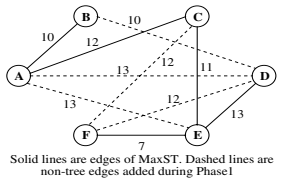
III. HEURISTIC 1

The first heuristic is based on the concept of Maximum Spanning Tree (MaxST)[3]. We define edge weights, $t_{ij}\ \forall\ (i,j) \in E$ to be equal to the link's working capacity ($t_{ij} = w_{ij}$). We then find the MaxST, $T(V, E_2)$ of G based on these edge weights. Note that if we set capacities of the edges in T to be equal to the corresponding edge weights ($c_{mn} = t_{mn}\ \forall\ (m,n) \in E_2$), T serves as a feasible (not necessarily optimal) restoration network for all links that are in G but not part of T. So, we only need additional restoration paths for edges in T.

The heuristic consists of two distinct phases, namely *Growing* Phase and *Pruning* Phase. During the *Growing* Phase, we find a feasible solution R' based on MaxST for G. We then reduce the total capacity of R' by decreasing excess capacities on some of the links in R' (without affecting its feasibility) during the *Pruning* Phase. The result is R. A trivial way to construct R' is as follows. For every edge $(m,n) \in T$, pick a node $p \in V$ such that $p \neq m, n$, add two edges (m,p) and (p,n) and assign them capacities equal to edge weight (working capacity) of (m,n) i.e. $c_{mp} = c_{pn} = t_{mn}$. Thus, at the end of the *Growing* Phase, R' will have a capacity three times that of T. We do not increase capacities during the *Pruning* Phase. As a result, we are guaranteed a final solution, R whose total capacity does not exceed three times the total capacity of T. Since MaxST is a lower bound to our problem [7], this heuristic has an upper bound of three times the lower bound in the worst case, making this a 3-approximation algorithm.

A. Pre-processing Steps and Algorithm outline

We first find the MaxST, T of the given graph. We then sort the working capacities (w_{ij}'s) of the edges in T in non-increasing order and store them with corresponding end nodes in a list, $treeList$. Starting with the first element in $treeList$, we add edges with sufficient capacity to T until all edges in $treeList$ have an alternate path. We call this augmented graph R'. Note that for an edge $(i,j) \in T$, there may exist an alternate path from i to j if the edges added earlier could be used for restoring the working capacity (WC) on that edge. If no such path exists, the edge may be restored by the addition of one or two edges. For example, in Figure 2, the alternate path for AC adds two edges CF and DF to R', while AB needs only one additional edge (BD). To select the most favorable edges to be added to R', we associate a cost with each edge. The cost of adding an edge (x,y) to R' is directly proportional to the working capacity w_{xy} on that edge. Also, observe that



Solid lines are edges of MaxST. Dashed lines are non-tree edges added during Phase1

Fig. 2. Restoration Network at the end of Growing Phase

an edge used for restoring a WC w_{ij} will be better utilized if the same edge can be used for restoring other large WCs associated with the nodes x or y. Hence, we define the cost, $C(x,y)$ of adding an edge (x,y) as

if $(w_{xy} > 0)$

$C(x,y) = w_{xy} + \delta * (w_{max} - (w_{xmax} + w_{ymax})/2)$

else

$C(x,y) = w_{xy} + \delta_1 * (w_{max} - (w_{xmax} + w_{ymax})/2)$

where w_{xy} is the WC of edge (x,y)

w_{max} is the maximum WC value

w_{xmax} is the next highest WC value $\leq w_{xy}$ at x

w_{ymax} is the next highest WC value $\leq w_{xy}$ at y

$0 \leq \delta \leq 1$

$\delta_1 = 0.0001$ (some small constant)

We always pick the minimum cost edge(s) to add to R'. The δ and δ_1 ensure that among two edges with the same WC, the edge that will potentially restore additional larger WCs in the future is given preference. However, for $w_{xy} > 0$, it is sometimes more important to consider the second factor rather than the actual WC on that edge. We iteratively increase the relative significance of the δ part of the cost by incrementing δ by a small quantity (≈ 0.1) between iterations. We pick the solution that gives the minimum value for total capacity of R.

The above cost assignment scheme also implies that adding two edges is given preference over adding one edge if the total cost of adding two edges is less than the cost of adding one edge. We found that this approach could be counter-productive in certain cases. So, we conducted experiments giving preference to adding one edge even if the cost as calculated above was higher than that of adding two edges. Note that, when an edge is added to R', it is assigned a capacity equal to WC of the edge being restored. Once all edges in $treeList$ are restored, the *Growing* phase terminates with a feasible solution R' to our problem. Figure 2 illustrates R' obtained for sample graph in Figure 1 for $\delta = 0$.

For the *Pruning* phase, we first sort the WCs in non-increasing order and store the corresponding edges in a list, $decWCList$. Edges that form the final restoration network, R are called *marked* edges. Initially, all edges in R' are *unmarked*. We then reassign edge weights. For an edge $(i,j) \in R'$, $t_{ij} = w_{ij}$ if $w_{ij} > 0$. If $w_{ij} = 0$, $t_{ij} = \rho$ where ρ is a small constant (say 0.001). We assign this non-zero weight for edges with zero WC to distinguish them from *marked* edges in the later stages of the algorithm. Starting with the first edge in $decWCList$, we find the shortest restoration (least weight) path with sufficient capacity using Dijkstra's Shortest Path Algorithm [3]. If the capacity, c_{xy} of any edge (x,y) on this

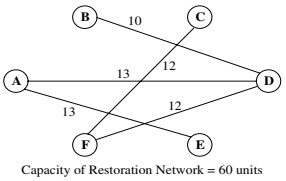


Fig. 3. Restoration Network at the end of Pruning Phase

path is greater than WC (w_{ij}) to be restored, we set $c_{xy} = w_{ij}$ and *mark* this edge to ensure that its capacity is not reduced in the future. In order to fully utilize the restoration paths for higher working capacities, we reset the weight of *marked* edges to zero. When all edges in $decWCList$ are restored, we have the required restoration network, R consisting of only the *marked* edges. The restoration network, R obtained at the end of the *Pruning* phase for the R' shown in Figure 2 is shown in Figure 3.

B. Formal Algorithm

Build Maximum Spanning Tree T of G
Create $treeList, decWCList$
Set $c_{ab} = t_{ab}$ for each edge $(a, b) \in T$
Set $c_{cd} = 0$ for each edge $(c, d) \notin T$
$\delta = 0$; $\rho = 0.001$; $minCap = \infty$; # Initialization
while ($\delta \leq 1$) loop
  $R' = T$; $index = 0$;
  while ($index < |V - 1|$) loop
    Get edge (i, j) at position $index$ of $treeList$
    Find least cost alternate path P_{ij} using cost function C
    for each edge $(k, l) \in P_{ij}$
      if ($c_{kl} < w_{ij}$) $c_{kl} = w_{ij}$;
    $index = index + 1$;
  end loop
  Set status of edges in R' to *unmarked*
  for each edge (m, n) in R'
    if ($w_{mn} > 0$) $t_{mn} = w_{mn}$;
    else $t_{mn} = \rho$;
  while ($decWCList \neq \emptyset$) loop
    Remove edge (p, q) at the head of $decWCList$
    Find shortest restoration path P_{pq} for (p, q)
    for each edge (x, y) in P_{pq}
      if ($c_{xy} \geq w_{mn}$ and (x, y) is *unmarked*) {
        Set status of (x, y) to *marked*
        $c_{xy} = w_{mn}$; $t_{xy} = 0$; }
  end loop
  $totCap =$ total capacity of R'
  if ($totCap < minCap$)
    { $minCap = totCap$; $R = R'$; }
  Increment δ
end loop
return R

IV. Heuristic 2

Heuristic 1 considers addition of a maximum of two edges at any stage. For high density graphs, since the number of *preferred* edges is very small, a substantial number of *non-preferred* edges can be part of the final restoration network, thereby increasing the total capacity of R. In high density graphs, it might be better to consider longer restoration paths if such paths could be found using *preferred* edges. Heuristic 2 has been developed with this goal.

A. Pre-processing Steps and Algorithm outline

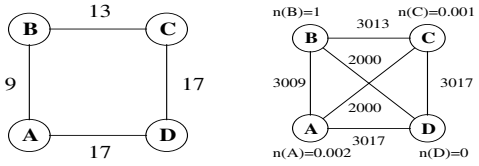
We maintain a sorted edge list based on WCs, $decWCList$ as in Heuristic 1. We also maintain a sorted list of the distinct WC values, $incWCList$. We then obtain a complete graph $R'(V, E_1)$ by including edges that are not in G. The capacity, c_{ij} is initialized to zero for all $(i, j) \in E_1$. We find the "best" restoration paths for all edges in $decWCList$ and assign capacities accordingly. We assign weights to edges and nodes in R' to make this process of finding restoration paths deterministic.

Each edge, $(i, j) \in E_1$ is assigned an edge weight, t_{ij} proportional to the WC of that link in G. The weight assignment is also done such that we do not find long restoration paths that nullify the gains made by using edges with small WCs in the original graph. The weight assigned to *preferred* edges is substantially lower than *non-preferred* edges. Even among *non-preferred* edges, the weight assigned to an edge is directly proportional to its WC. Further, when we consider x edges (*preferred* edges $(PREF)$ and *non-preferred* edges $(NPREF)$), the following is always true.

$$\begin{aligned} Wt\ (x\ PREF) &< Wt\ ((x-1)\ PREF\ and\ 1\ NPREF) \\ &< Wt\ ((x-2)\ PREF\ and\ 2\ NPREF) \\ &< \ldots \\ &< Wt\ (x\ NPREF) \end{aligned}$$

where $Wt(.)$ is a function that sums the edge weights of the edges passed as arguments. However, there seems to be no clear-cut preference between x *preferred* and y *non-preferred* edges when $x > y$.

For each node $v \in V$, its node weight n_v is calculated. The node weights are used to choose one restoration path among paths that seem to have the same incremental effect on the total capacity. The node weights are computed as follows: For the first edge in $decWCList$, we find all the links that have the same WC in $decWCList$. We then create a sorted list, $nodeWtList$ of the end nodes based on non-increasing order of the total traffic at each node. We then assign a node weight of zero to the first node in $nodeWtList$. The second node is assigned a weight of $(0+\rho)$ where ρ is a small value ($\rho << 1$). The third node is assigned a weight of $(0 + 2 \times \rho)$ and so on. Once all the nodes in $nodeWtList$ are assigned weights, we get the second highest WC, re-create the $nodeWtList$ based on this WC, and assign node weights to the end nodes starting from a base value of one in a similar manner. This process is continued until every node is assigned a node weight. As a result, when there are multiple paths between two end nodes of a link that needs to be restored, we use nodes that have high traffic as our intermediate nodes. High WCs will be restored



(a) Sample Graph (b) Weights

Fig. 4. Assigning Node and Edge Weights

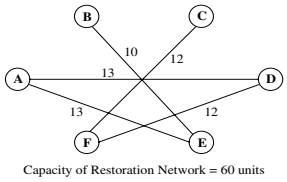


Fig. 5. Restoration Network produced by Heuristic 2

first and we hope to reuse the capacity on the corresponding restoration paths for restoring the WC under consideration. If there is a tie between nodes with the same highest flow, we prefer the node with the highest total flow since it is likely to have multiple paths going through it. This preferential ordering among nodes with same highest flow is ensured by ρ in the node weight assignment. Figure 4 shows the edge and node weight assignment for a given sample network.

Starting with the first edge in $decWCList$, we employ a modified version of the Dijkstra's Shortest Path algorithm to find the shortest (least weight) alternate path, P_{ij} for every edge, (i, j) in $decWCList$. Here, we also take node weights into consideration and update the weight of the shortest path as follows:

if $(d_v > d_u + n_u + t_{uv})$ then
$d_v = d_u + n_u + t_{uv}$
where d_u is the weight of shortest path to node u from i
d_v is the weight of shortest path to node v from i
n_u is the weight of node u
t_{uv} is the weight of link between nodes u and v

If the capacity of any edge (x, y) in P_{ij} is less than the WC, w_{ij}, we set $c_{xy} = w_{ij}$. The edge weights (t_{ij}'s) are reset to zero for all edges (x, y) in P_{ij}. This is done to ensure that the algorithm tries to use these edges for restoring subsequent working capacities. We also re-assign node weights using one of the following three schemes:

- Reset all node weights to zero ($n(x) = 0\ \forall x \in P_{ij}$).
- Reset the node weights of only the end nodes to zero ($n(i) = n(j) = 0$).
- Reset the node weights to zero or close to infinity ($>>$ maximum edge weight) depending on the next element in $decWCList$. If the next edge has end nodes that have not been a part of any restoration path determined so far, we set the node weights to a very high value. Otherwise, the node weights are reset to zero. This scheme of node re-assignment is based on the proposed third approach to compute the lower bound described in [7]. We observed that this approach gives the best results for most cases.

Considering the fully connected graph R' as the starting point for our heuristic often does not give the best solutions because the heuristic employs a large number of "expensive" *non-preferred* links. So, we use an iterative procedure to determine the "best" sub-graph, R' of the fully connected graph that will act as the initial framework for R. We define an upper bound, $threshold$, on the WC of the links to be considered as part of R'. Only links with WC $\leq threshold$ are added to R'. We first try to find a feasible solution considering only the *preferred* links implying $threshold$ is zero initially. Even if we find that all requirements are satisfied, this might not be the best solution as some of restoration paths may be very long. We then increase the threshold and consider all edges with WC value less than the new threshold to be part of R'. Thus, incrementing the $threshold$ in effect adds a few *non-preferred* links to R'. We again run the algorithm and find the total capacity needed for restoration. We repeat this procedure for all threshold values and pick the R with minimum total capacity.

The increment in $threshold$ can be in terms of the distinct working capacities in $incWCList$ or staggered over the working capacities (for large graphs). For large graphs (more than 15 nodes), we can terminate the iterative process after the solution stabilizes or goes into a steady increase mode. In practice, the network with minimum total capacity was generally obtained for low $threshold$ values. The reason we get very good solutions for small $threshold$ values, which generally imply longer paths since fewer edges are available, is that the shortest path need not necessarily be the best path. The shortest path may use links that have large WCs, and these links have to be protected by other restoration paths. For instance, in Figure 5, if we had used the edge BC to restore AB instead of BE, we would have needed a restoration path for the link BC itself and that would have increased the total capacity of R.

Figure 5 illustrates the heuristic solution obtained for the sample graph shown in Figure 1. Note that this restoration network has the same capacity as the restoration network produced by Heuristic 1. For the sake of comparison, the total capacity of R obtained using the ILP formulation described in [7] for this particular sample graph is equal to the total capacity obtained using the heuristics. The two lower bounds defined in [7] gave bounds of 52 and 53 respectively.

B. Formal Algorithm

$threshold = 0$; $minCap = \infty$; # Initialization
L1: while($threshold$ values not exhausted) loop
Assign edge and node weights
Add edge (i, j) to R' if $w_{ij} \geq threshold$
if(R' is not connected)
{Increment $threshold$ and goto L1}
while($decWCList$ is not empty) loop
Remove edge (p, q) at the head of $decWCList$

```
        Find shortest restoration path P_pq for (p, q)
        for each edge (x, y) ∈ P_pq {
            if(c_xy < w_pq)
                c_xy = w_pq;
            t_xy = 0;
            Re-assign node weights }
    end loop
    totCap = total capacity of R'
    if(totCap < minCap)
        { minCap = totCap; R = R'; }
    Increment threshold
  end loop
return R
```

V. Simulation Experiments

Simulation experiments were performed using randomly generated input graphs of varying size and density. The number of nodes, n varied from $n = 5$ to $n = 100$. The graphs were generated by using density as an input parameter. An edge was added between a pair of nodes with a non-zero probability, p equal to the density of the graph to be generated. The two heuristic solutions and two lower bounds described in [7] were computed. A small subset of the above-mentioned input graphs were also solved optimally using $CPLEX^{®}$. To compare the heuristics with the lower bounds and ILP solution, a parameter called $Relative\ Spare\ Capacity(RSC)$ was defined. The parameter is calculated as follows:

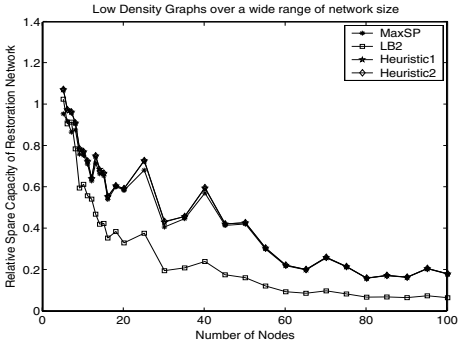
$$RSC = \frac{Total\ Capacity\ of\ Restoration\ Network}{Total\ Working\ Capacity\ of\ Original\ Network}$$

A. Heuristic Solutions vs Lower Bounds

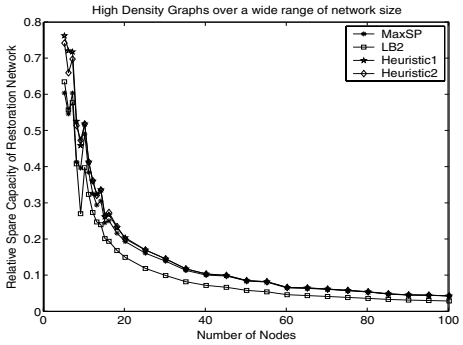
The performance of both heuristics improved as the size of the network increased. This could be attributed to the fact that as the size of the network increased, the number of *preferred* edges available for restoration increased and the heuristics performed better. We further noticed that the divergence from the lower bounds was smaller for low and medium density graphs than in high density graphs. Again, the obvious reason was the availability of more number of *preferred* edges in sparse graphs relative to the more dense graphs. Figure. 6(a) and Figure. 6(b) illustrate the relative capacity requirements as obtained by the two lower bounding algorithms and the heuristic solutions for low density and high density graphs respectively. Each point in the figures represents one corresponding experiment only. The maximum divergence from the lower bound for low density graphs was about 8%. The divergence was less than 3% for most of the low density graphs. The heuristic solutions diverged by a large margin for small, dense graphs with a maximum divergence of 24% for heuristic 2 and 28% for heuristic 1. For graphs with more than 8 nodes, the heuristics were rarely more than 10% away from the lower bounds. The divergence was less than 4% for graphs with more than 35 nodes.

B. Heuristic Solutions vs ILP Solution

The large divergence from the lower bounds for small, dense graphs could be due to two reasons:



(a) Low Density Graphs for $n = 5$ to $n = 100$

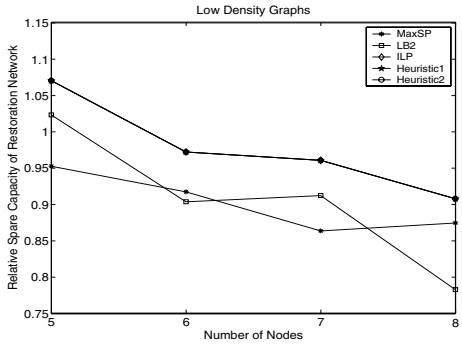


(b) High Density Graphs for $n = 5$ to $n = 100$

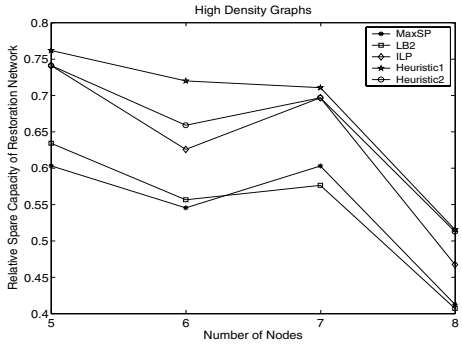
Fig. 6. Heuristics vs. Lower Bounds

1. The heuristics give poor solutions.
2. The lower bounds are poor i.e. not tight enough.

Simulation experiments were done to find the optimal solution for such graphs and determine the dominant reason among the two mentioned above. The optimal solution was generally found to be very close or equal to the heuristic solutions. The maximum divergence was about 10% for heuristic 2, while it was a more pronounced 15% for heuristic 1. The divergence was rarely more than 0.5% for low density graphs. This leads us to believe that the reason for large divergence from lower bounds is due to the lower bounds and not due to the heuristic solutions. In Figure. 7(a) (low density graphs), it can be observed that both heuristics give the optimal solution for all cases ($n = 5, 6, 7, 8$). It was also observed that the heuristic solutions, especially those of heuristic 2, are close to the corresponding optimal solutions even for high density graphs as illustrated in Figure. 7(b). Only small graphs were solved here because $CPLEX^{®}$ took a substantial time to solve the ILP formulation presented in [7] and the divergence from lower bounds was significantly high only for such graphs (as noted earlier). The ILP was solved using methods similar to column generation technique [1] used for standard Linear Programs. It took up to a week to solve certain ILPs (for just 8 nodes) on a $Pentium^{®}$ 4 based Linux machine with 1 GB of



(a) Low Density Graphs for $n = 5$ to $n = 8$



(b) High Density Graphs for $n = 5$ to $n = 8$

Fig. 7. Heuristics vs. Optimal Solution

memory. On the other hand, the heuristics completed execution in a few hours even for large, dense graphs (up to 100 nodes) indicating a tremendous gain in running time. Small graphs were solved within a few seconds.

C. Heuristic 1 vs Heuristic 2

Heuristic 2 consistently out-performed Heuristic 1 for high density graphs. However, for low and medium density graphs the two heuristics performed comparably. As explained earlier, Heuristic 1 performs poorly for high density graphs because of its limitation of choosing only up to two edge additions per WC. Heuristic 2 does not have this constraint and consequently gives better solutions. However, Heuristic 1 has a provable worst case upper bound for the solution while that is not the case with Heuristic 2.

VI. Conclusion

We have studied the Link Restoration problem in telecom networks. We were allowed to use links that were not part of the original graph for restoration with no traffic splitting. We have presented two heuristics to solve this problem. Simulation results suggest that the proposed heuristics give close to optimal solutions for small graphs. In practice, telecom networks are large and sparse. The two heuristic perform exceptionally well for such input graphs. Interestingly, the restoration network is very sparse even when we consider the original graph to be fully connected.

The upper bound of three times the Maximum Spanning Tree seems to be a very loose bound because of the relatively much smaller divergence obtained in simulation experiments. Future work could involve developing and proving tighter lower and upper bounds for the problem. To limit the apparent reliance of the heuristics on zero working capacity links, one can restrict the number of such links employed for restoration. Also, one can extend the heuristics to solve the Link Restoration problem where only the links of the original network can be used for restoration. This version of the problem is provably NP-Complete and a heuristic that gives a good solution to the problem would be valuable. Bounds incorporating these additional constraints have to be developed for an accurate evaluation of the heuristics.

Acknowledgment

The authors would like to thank the anonymous reviewers for their helpful suggestions and comments that greatly improved the quality of this paper.

References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, NJ, 1993.

[2] C. E. Chow, J. Bicknell, S. McCaughey, and S. Syed. A fast distributed network restoration algorithm. In *Proc. Twelfth Annual International Phoenix Conference on Computers and Communications*, pages 261–267, March 1993.

[3] T. H. Cormen, C. L. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[4] B. Doshi, S. Dravida, P. Harshavardhana, O. Hauser, and Y. Hang. Optical network design and restoration. *Bell Labs Technical Journal*, pages 58–84, January/March 1999.

[5] W. D. Grover, T. D. Bilodeau, and R. D. Venables. Near optimal spare capacity planning in a mesh restorable network. In *Proc. IEEE Global Telecomunications Conference (GLOBECOM'91)*, volume 3, pages 2007–2012, Phoenix, AZ, December 1991.

[6] R. R. Iraschko and W. D. Grover. A highly efficient path-restoration protocol for management of optical network transport integrity. *IEEE Journal on Selected Areas in Communications*, 18(5):779–794, May 2000.

[7] Srinivasan Krishnamurthy, R. Chandrasekaran, S. Venkatesan, and Milind Dawande. On lower bounds for link restoration. Technical Report UTDCS-03-03, The Univ. of Texas at Dallas, Richardson, TX, February 2003 (http://www.utdallas.edu/∼ksrini/Restoration/lowerBounds.pdf).

[8] K. Murakami and H. S. Kim. Comparative study on restoration schemes of survivable ATM networks. In *Proc. IEEE INFOCOM'97)*, volume 1, pages 345–352, April 1997.

[9] K. Murakami and H.S. Kim. Optimal capacity and flow assignments for self-healing ATM networks based on line and end-to-end restoration. *IEEE/ACM Trans. on Networking*, 6(2):207–221, April 1998.

[10] H. Sakauchi, Y. Nishimura, and S. Hasegawa. A self-healing network with an economical spare-channel assignment. In *Proc. IEEE Global Telecomunications Conference (GLOBECOM'90)*, volume 1, pages 438–443, San Diego, CA, December 1990.

[11] J. Veerasamy, S. Venkatesan, and J. C. Shah. Spare capacity assignment in telecom networks using path restoration. In *Proc. IEEE the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'95)*, pages 370–374, January 1995.

[12] C. H. Yang and S. Hasegawa. FITNESS: Failure immunization technology for network services survivability. In *Proc. IEEE Global Telecomunications Conference (GLOBECOM'87)*, volume 3, pages 1549–1554, Tokyo, Japan, November 1987.

[13] D. Zhou and S. Subramaniam. Survivability in optical networks. *IEEE Network*, 14(6):16–23, November/December 2000.