GRASP WITH PATH-RELINKING FOR NETWORK MIGRATION SCHEDULING

DIOGO V. ANDRADE AND MAURICIO G.C. RESENDE

ABSTRACT. Network migration scheduling is the problem where inter-nodal traffic from an outdated telecommunications network is to be migrated to a new network. Nodes are migrated, one at each time period, from the old to the new network. All traffic originating or terminating at given node in the old network is moved to a specific node in the new network. Routing is predetermined on both networks and therefore arc capacities are known. Traffic between nodes in the same network is routed in that network. When a node is migrated, one or more temporary arcs may need to be set up since the node migrated may be adjacent to more than one still active node in the old network. A temporary arc remains active until both nodes connected by the arc are migrated to the new network. In one version of the problem, one seeks an ordering of the migration of the nodes that minimizes the maximum sum of capacities of the temporary arcs. In another version, the objective is to minimize the sum of the total capacities of the temporary arcs over each period in the planning horizon. In this paper, we propose a hybrid heuristic which combines GRASP with path-relinking to find cost-efficient solutions to both versions of the network migration problem.

1. INTRODUCTION

Consider the problem where inter-nodal traffic from an outdated telecommunications network is to be migrated to a new network. Nodes are migrated, one at each time period, from the old to the new network. All traffic originating or terminating at given node in the old network is moved to a specific node in the new network. Routing is predetermined on both networks and therefore arc capacities are known. Traffic between nodes in the same network is routed in that network. Suppose that node y_o in the old network is migrated to node y_n in the new network. Let the capacity of arc (x_o, y_o) be c_o. Traffic between node x_o and node y_n must use a temporary arc (x_o, y_n) connecting the two nodes with capacity c_o. When a node is migrated, one or more temporary arcs may need to be set up since y_o may be adjacent to more than one still active node in the old network. A temporary arc remains active until both nodes connected by the arc are migrated to the new network. In one version of the *network migration scheduling problem*, one seeks an ordering of the migration of the nodes that minimizes the maximum sum of capacities of the temporary arcs. In another version, the objective is to minimize the sum of the total capacities of the temporary arcs over each period in the planning horizon.

These scheduling problems can be modeled as variants of the linear arrangement problem. A *linear arrangement* of a graph $G = (V, E)$, where V is the vertex set with $|V| = n$ and E is the edge set, is a one-to-one function $\pi : V \rightarrow \{1, 2, \ldots, n\}$. Consider an edge-weighted graph $G = (V, E, w)$, where w is the vector of real edge weights. For $1 \leq i < n$,

Date: October 31, 2006.

Key words and phrases. Network optimization, node migration, minimum cut, linear arrangement, GRASP, path-relinking, local search, metaheuristic, heuristic.

AT&T Labs Research Technical Report TD-6V4V9Z. This extended abstarct was submitted to INOC 2007.

let *cut* K_i between nodes $\pi^{-1}(i)$ and $\pi^{-1}(i+1)$ be the sum of the weights of all arcs with one endpoint in $\pi^{-1}(j)$ and the other in $\pi^{-1}(k)$ for all $j \leq i$ and all $k > i$. The value $K_{\max}$ of the largest cut is the *max-cut*, i.e. $K_{\max} = \max\{K_1, K_2, \ldots, K_{n-1}\}$. The value K_{sum} is the cut sum, i.e. $K_{sum} = \sum_{i=1}^{n-1} K_i$. There are two variants of linear arrangement problems. In the first type, the *minimum cut linear arrangement problem*, we seek to minimize the maximum cut. One is given an edge-weighted graph $G = (V, E, w)$ and seeks an arrangement π that minimizes $K_{\max} = \max\{K_1, K_2, \ldots, K_{n-1}\}$, where $K_i = \sum_{\{u,v\} \in E : \pi(u) \leq i \leq \pi(v)} w_{uv}$. In the second type, the *minimum linear arrangement problem*, one seeks to minimize the sum of the cuts. One is given an edge-weighted graph $G = (V, E, w)$ and seeks an arrangement π that minimizes $K_{sum} = \sum_{i=1}^{n-1} K_i = \sum_{(u,v) \in E} |\pi(u) - \pi(v)| \cdot w_{uv}$.

In this paper, we propose a hybrid heuristic which combines GRASP with path-relinking to find cost-efficient solutions to the above linear arrangement problems. In Section 2 we describe the hybrid heuristic. Construction and local search for the GRASP are presented in Sections 3 and 4, respectively. Concluding remarks are made in Section 5.

2. GRASP WITH PATH-RELINKING

A greedy randomized adaptive search procedure, or GRASP [3, 4, 8], is a metaheuristic for finding approximate solutions to combinatorial optimization problems of the form $\min\{f(x) \mid x \in X\}$, where f is the objective function and X is a discrete set of feasible solutions. It can be viewed of as a search method that applies local search repeatedly from many starting solutions. The starting solutions for local search are constructed with a randomized greedy procedure.

Path-relinking was originally proposed by Glover [5] as an intensification strategy exploring trajectories connecting elite solutions. Starting from one or more elite solutions, paths in the solution space leading toward other elite solutions are generated and explored in the search for better solutions. Figure 1 illustrates the pseudo-code of the path-relinking procedure applied to a pair of solutions x_s (starting solution) and x_t (target solution). The

```
procedure Path-relinking($x_s$,$x_t$)
1    Compute symmetric difference $\Delta(x_s, x_t)$
2    $f^* \leftarrow \min\{f(x_s), f(x_t)\}$
3    $x^* \leftarrow \mathrm{argmin}\{f(x_s), f(x_t)\}$
4    $x \leftarrow x_s$
5    while $\Delta(x, x_t) \neq \emptyset$ do
6        $m^* \leftarrow \mathrm{argmin}\{f(x \oplus m) : m \in \Delta(x, x_t)\}$
7        $\Delta(x \oplus m^*, x_t) \leftarrow \Delta(x, x_t) \setminus \{m^*\}$
8        $x \leftarrow x \oplus m^*$
9        if $f(x) < f^*$ then
10           $f^* \leftarrow f(x)$
11           $x^* \leftarrow x$
12       end if
13   end while
14   return($x^*$);
end Path-relinking;
```

FIGURE 1. Path-relinking.

procedure starts by computing the symmetric difference $\Delta(x_s, x_t)$ between the two solutions, i.e. the set of moves needed to reach x_t (target solution) from x_s (initial solution). A path of solutions is generated linking x_s and x_t. The best solution x^* in this path is returned by the algorithm. At each step, the procedure examines all moves $m \in \Delta(x, x_t)$ from the current solution x and selects the one which results in the least cost solution, i.e. the one which minimizes $f(x \oplus m)$, where $x \oplus m$ is the solution resulting from applying move m to solution x. The best move m^* is made, producing solution $x \oplus m^*$. The set of available moves is updated. If necessary, the best solution x^* is updated. The procedure terminates when x_t is reached, i.e. when $\Delta(x, x_t) = \emptyset$.

Path-relinking is a major enhancement [6] to the basic GRASP procedure, leading to significant improvements in solution time and quality [1, 2, 8, 9, 10]. Two basic strategies are usually adopted. In one, path-relinking is applied as a *local improvement* phase to each local optimum obtained after the local search phase. In the other, path-relinking is periodically applied as an *elite-set intensification* phase to some or all pairs of elite solutions. In general, combining local improvement with elite-set intensification results in the preferred strategy. In the context of local improvement, path-relinking is applied to pairs $\{x, y\}$ of solutions, where x is a locally optimal solution produced by each GRASP iteration after local search and y is one of a few elite solutions randomly chosen from a pool with a limited number `Max_Elite` of elite solutions found along the search.

The elite set, or pool, is originally empty. Any solution that is *best so far* is inserted in the pool. To maintain a pool of good but diverse solutions, each locally optimal solution obtained by local search is considered as a candidate to be inserted into the pool if it is sufficiently different from every other solution currently in the pool. If the pool already has `Max_Elite` solutions and the candidate is better than the worst of them, then a simple strategy is to have the former replace the latter. If the pool is not full, the candidate is simply inserted.

Elite set intensification is done on a series of pools. The initial pool P_1 is the elite set P obtained in the GRASP iterations. The pool counter is initialized $k \leftarrow 0$. At the k-th iteration, pairs of elements in pool P_k are combined using path-relinking. Each result of path-relinking is tested for membership in pool P_{k+1} following the same criteria used during the GRASP iterations. If a new best solution is produced, i.e. $\min\{f(x) \mid x \in P_{k+1}\} < \min\{f(x) \mid x \in P_k\}$, then $k \leftarrow k+1$ and a new iteration of elite-set intensification is done. Otherwise, intensification halts with $P \leftarrow P_{k+1}$ as the resulting elite set.

The pseudo-code in Figure 2 illustrates such a procedure.

3. GRASP CONSTRUCTION

We begin by describing the GRASP construction heuristic, a generalization of an idea presented by Petit [7]. Given a initial sequence with $k-1$ vertices, we augment the sequence by adding the k-th vertex to a position where the increase in the objective function is minimized. Our heuristic is a generalization in the sense that it chooses from any position in the current arrangement, while in Petit's case, only the head and the tail of the ordering are considered. The heuristic starts with an initial ordering of the vertices. This ordering is used to determine which vertex is the next to be inserted in the solution. Petit proposed four different strategies for this initial ordering: *normal*, *random*, *breadth-first search* and *depth-first search*. The last two with an option of being randomized, i.e. pick a random neighbor when performing the BFS or DFS. For our heuristic we chose randomized depth-first search, since the DFS uses the structure of the input graph to give an ordering where neighbor vertices are close to each other. This option not only takes advantage of the structure of the graph, but also provides some randomization to the initial ordering, that will result in a random greedy heuristic.

```
procedure GRASP-PR($i_{\max}$)
1   $P \leftarrow \emptyset$; $r \leftarrow 0$;
2   while stopping criteria not satisfied do
3       for $i = r+1,\ldots,r+i_{\max}$ do        /* GRASP iterations */
4           $x \leftarrow$ GreedyRandomizedConstruction();
5           $x \leftarrow$ LocalSearch($x$);
6           Update the elite set $P$ with $x$;
7           if $|P| > 0$ then        /* local improvement */
8               Choose, at random, pool solution $y \in P$ to relink with $x$;
9               Determine which ($x$ or $y$) is initial $x_s$ and which is target $x_t$;
10              $x_p \leftarrow$ PathRelinking($x_s,x_t$);
11              Update the elite set $P$ with $x_p$;
12          end if
13      end for
14      $k \leftarrow 0$;
15      do        /* elite-set intensification */
16          $k \leftarrow k+1$; $P_k \leftarrow P$; $P_{k+1} \leftarrow \emptyset$;
17          forsome $\{x,y\} \in P_k \times P_k$ do
18              $x_p \leftarrow$ PathRelinking($x,y$);
19              Update the elite set $P_{k+1}$ with $x_p$;
20          end forsome
21          $P \leftarrow P_{k+1}$;
22      until $\min\{f(x), x \in P_{k+1}\} \geq \min\{f(x), x \in P_k\}$;
23      $r \leftarrow r+1$;
24  end while
25  $x^* \leftarrow \mathrm{argmin}\{f(x), x \in P\}$
26  return($x^*$);
end GRASP-PR;
```

FIGURE 2. A basic GRASP with path-relinking heuristic for minimization.

We now show how the increase in the cuts is computed when a new vertex is inserted. Let $\{v_1,\ldots,v_{k-1}\}$ be the $k-1$ vertices currently in the solution, and $K_1,\ldots,K_{k-2}$ be the current cut values. Let x be the vertex inserted at position i, the new set of cut values $K' = \{K'_1,\ldots,K'_{k-1}\}$ can be computed as follows:

$$K'_j = \begin{cases} K_j + \sum_{a=1}^{j} w_{xv_a} & j = 1,\ldots,i-1 \\ K_{j-1} + \sum_{a=j}^{k-1} w_{xv_a} & j = i,\ldots,k-1. \end{cases}$$

The above expression results in an algorithm to compute the new cut values. At first sight, such algorithm would require $O(k^2)$ operations to compute all $k-1$ cuts for one given position i. However, this can be easily improved to $O(k)$ operations for each position i, by computing the updated cuts in the correct order, and by storing the value of the corresponding cumulative sums: $S_- = \sum_{a=1}^{j} w_{xv_a}$, $S_+ = \sum_{a=j}^{k-1} w_{xv_a}$. For example, to update the cuts positioned before i, i.e. $j = 1,\ldots,i-1$, it is convenient to start by computing K_1 ($j = 1$), since it suffices to add $S_- = w_{xv_1}$ to the current cut value. To update K_2, we update the cumulative sum $S_- \leftarrow S_- + w_{xv_2}$, and add it to the current cut value. Notice that by starting from $j = 1$, in each step we can update the current cumulative sum in $O(1)$ time, and therefore, recompute each cut value in $O(1)$. Analogously, for the cuts positioned after

i, i.e. $j = i, \dots, k-1$, we start by updating the *last* cut K_{k-1} and proceed decreasing j until we reach the i-th cut.

In principle, the procedure described above would take $O(k^2)$ time to compute the best position for x to be inserted, but this can be further improved using the following proposition.

Proposition 3.1. *Let x be the vertex to be inserted, $K^{(i)} = \{K_1^{(i)}, \dots, K_{k-1}^{(i)}\}$ be the updated cut values if x is inserted at position i, and $K^{(i+1)} = \{K_1^{(i+1)}, \dots, K_{k-1}^{(i+1)}\}$ be the updated cut values if x is inserted at position $i+1$. $K^{(i)}$ and $K^{(i+1)}$ are related as follows:*

$$K_j^{(i+1)} = \begin{cases} K_j^{(i)} - K_{j-1} - \sum_{a=j}^{k-1} w_{xv_a} + K_j + \sum_{a=1}^{j} w_{xv_a} & \textit{if } j = i \\ K_j^{(i)} & \textit{otherwise.} \end{cases}$$

Using Proposition 3.1 we can now find the best position to insert element x in $O(k)$. We start by setting $i = 1$ and finding the corresponding updated cut values $K^{(1)}$ and cumulative sums ($S_- = \sum_{a=1}^{0} w_{xv_a} = 0$ and $S_+ = \sum_{a=1}^{k-1} w_{xv_a}$) in $O(k)$ time. Now for the next position ($i+1 = 2$) it suffices to update the cut value $K_1^{(2)}$, and the cumulative sums S_- and S_+. Those steps can all be trivially done in $O(1)$:

$$\begin{array}{lll} S_- & \leftarrow & S_- + w_{xv_1} \\ K_1^{(2)} & \leftarrow & K_1^{(1)} - K_0 - S_+ + K_1 + S_- \\ S_+ & \leftarrow & S_+ - w_{xv_1} \\ i & \leftarrow & i+1 \end{array}$$

For simplicity, we assume $K_0 = 0$. Moreover, notice that the order in this update is important, otherwise w_{xv_1} could be counted twice (if we update S_+ before $K_1^{(2)}$) or not counted at all (if we update $K_1^{(2)}$ before S_-). We repeat this procedure until $i = k$, and every step (except $i = 1$) takes $O(1)$ time, so the overall complexity is $O(k)$.

The randomization provided in the initial ordering combined with the greedy choice of the position to insert the element gives us the greedy randomized heuristic.

4. Local search

We consider 2-swap neighborhoods used in a local search procedure. The construction procedure, described in Section 3, outputs an ordering $\pi(1), \pi(2), \dots, \pi(n)$, $n-1$ cut values $K_1, K_2, \dots, K_{n-1}$, and for each $v \in V$, forward weighted degrees $f(v) = \sum_{u \mid \pi(u) > \pi(v)} w_{vu}$ and backward weighted degrees $b(v) = \sum_{u \mid \pi(u) < \pi(v)} w_{uv}$. The 2-swap local search, in its full mode, examines all vertex swaps in the ordering, searching for an improvement in the objective function. In other words, it determines how the swap affects the cut values, and if it improves the objective function, then the swap is done. The following proposition shows how the cuts are affected.

Proposition 4.1. *Let x and y be the two nodes to be swapped ($x = \pi^{-1}(i), y = \pi^{-1}(j), i < j$), and $f(x), b(x)$ (resp. $f(y), b(y)$) be the forward and backward weighted degrees of x (resp. y)* before *the swap. Then, we have the following:*

(i) *Only the cuts $K_i, \dots, K_{j-1}$ will be affected.*

(ii) *Let $e_{i+1},\dots,e_j$ be the nodes between x and y. The variation of the cut value $\Delta K(i+a)$ $(a=0,\dots,j-i-1)$ is given by the following expression:*

$$\Delta K(i+a) = b(x)-f(x)+f(y)-b(y)+2w_{xy} + 2\textstyle\sum_{k=i+1}^{i+a} w_{xe_k} + 2\sum_{k=i+a+1}^{j-1} w_{ye_k}.$$

Using the expression given in Proposition 4.1, we can now estimate the execution time of recomputing the cut values given a swap (x,y) in terms of the distance between x and y in the permutation array. Let $k=j-i+1$. It takes $O(k)$ to compute each new cut value, and therefore $O(k^2)$ to determine if the swap improves the solution value. Such execution time can be considerably improved by realizing that $\Delta K(i+a+1)$ can be calculated as a function of $\Delta K(i+a)$. From Proposition 4.1 we can derive the relation

$$\Delta K(i+a+1) = \Delta K(i+a) + 2w_{xe_{i+a+1}} - 2w_{ye_{i+a+1}}.$$

Therefore, we can first compute $\Delta K(i)$ in $O(k)$ time, and then compute the each of the remaining cuts in $O(1)$ time, taking only $O(k)$ time to compute all new cut values. This procedure is illustrated in the pseudo-code in Figure 3.

```
procedure test-swap($i, j, \pi, \pi^{-1}, w, K$)
1   $x \leftarrow \pi^{-1}(i)$;
2   $y \leftarrow \pi^{-1}(j)$;
3   $K' \leftarrow K$;
4   $\Delta K(i) = b(x) - f(x) - b(y) + f(y) + 2w_{xy} + 2\sum_{k=i+1}^{j-1} w_{y\pi^{-1}(k)}$;
5   $K'(i) \leftarrow K(i) + \Delta K(i)$;
6   for $k = i+1,\dots,j-1$ do
7       $\Delta K(k) = \Delta K(k-1) + 2w_{x\pi^{-1}(k)} - 2w_{y\pi^{-1}(k)}$;
8       $K'(k) \leftarrow K(k) + \Delta K(k)$);
9   end for
10  return obj-func($K'$);
end test-swap;
```

FIGURE 3. `test-swap` procedure for the local search.

Once the swap (x,y) is done, we must recompute the weighted forward and backward degrees ($f(k)$ and $b(k)$). Notice that we only need to recompute the indices k such that $i \le k \le j$. The update goes as follows:

$$\begin{aligned}
f(x) &\leftarrow f(x) - w_{xy} - \textstyle\sum_{k=i+1}^{j-1} w_{x\pi^{-1}(k)}\\
b(x) &\leftarrow b(x) + w_{xy} + \textstyle\sum_{k=i+1}^{j-1} w_{x\pi^{-1}(k)}\\
f(y) &\leftarrow f(y) + w_{xy} + \textstyle\sum_{k=i+1}^{j-1} w_{y\pi^{-1}(k)}\\
b(y) &\leftarrow b(y) - w_{xy} - \textstyle\sum_{k=i+1}^{j-1} w_{y\pi^{-1}(k)}\\
f(\pi^{-1}(k)) &\leftarrow f(\pi^{-1}(k)) + w_{x\pi^{-1}(k)} - w_{y\pi^{-1}(k)}\\
b(\pi^{-1}(k)) &\leftarrow f(\pi^{-1}(k)) - w_{x\pi^{-1}(k)} + w_{y\pi^{-1}(k)}
\end{aligned}$$

Figure 4 shows the pseudo-code for the local search procedure. The `test-swap` procedure is illustrated in Figure 3. The procedure `update-vars` is not explicitly illustrated but is simple to implement from the formulas given above (that shows how to update f and b), and from the procedure `test-swap` (for the cuts).

```
procedure local-search($\pi$,$\pi^{-1}$,$w$,$K$,$K^*$,$nK^*$)
1   improved ← true;
2   while improved = true do
3       improved ← false;
4       for all pairs $(i,j) : 1 \le i \le nK^* < j \le n$ do
5           $K'$ ← test-swap($i,j,\pi,\pi^{-1},w,K$);
6           if $K' < K^*$ then
7               update-vars($i,j,\pi,\pi^{-1},f,b,K,K^*,nK^*$);
8               improved ← true;
9           end if;
10      end for;
11  end while;
12  return($\pi,K^*$);
end localsearch;
```

FIGURE 4. `local-search` procedure for MCLA problem.

5. CONCLUDING REMARKS

The heuristic presented in this extended abstract has been extensively tested on real and artificial test problems. In the extended version of this paper, we will present the computational experiments.

REFERENCES

[1] R.M. Aiex, M.G.C. Resende, P.M. Pardalos, and G. Toraldo. GRASP with path relinking for three-index assignment. *INFORMS J. on Computing*, 17:224–247, 2005.

[2] S.A. Canuto, M.G.C. Resende, and C.C. Ribeiro. Local search with perturbations for the prize-collecting Steiner tree problem in graphs. *Networks*, 38:50–58, 2001.

[3] T.A. Feo and M.G.C. Resende. A probabilistic heuristic for a computationally difficult set covering problem. *Operations Research Letters*, 8:67–71, 1989.

[4] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[5] F. Glover. Tabu search and adaptive memory programing – Advances, applications and challenges. In R.S. Barr, R.V. Helgason, and J.L. Kennington, editors, *Interfaces in Computer Science and Operations Research*, pages 1–75. Kluwer, 1996.

[6] M. Laguna and R. Martí. GRASP and path relinking for 2-layer straight line crossing minimization. *INFORMS Journal on Computing*, 11:44–52, 1999.

[7] J. Petit. Experiments on the minimum linear arrangement problem, 2001.

[8] M.G.C. Resende and C.C. Ribeiro. Greedy randomized adaptive search procedures. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 219–249. Kluwer Academic Publishers, 2003.

[9] M.G.C. Resende and C.C. Ribeiro. GRASP with path-relinking: Recent advances and applications. In T. Ibaraki, K. Nonobe, and M. Yagiura, editors, *Metaheuristics: Progress as real problem solvers*, pages 29–63. Springer Science+Business Media, 2005.

[10] M.G.C. Resende and R.F. Werneck. A hybrid heuristic for the p-median problem. *Journal of Heuristics*, 10:59–88, 2004.

(D.V. Andrade) RUTCOR, RUTGERS UNIVERSITY, PISCATAWAY, NJ 08854 USA.
E-mail address: `dandrade@rutcor.rutgers.edu`

(M.G.C. Resende) INTERNET AND NETWORK SYSTEMS RESEARCH CENTER, AT&T LABS RESEARCH, FLORHAM PARK, NJ 07932 USA.
E-mail address: `mgcr@research.att.com`