Path Problems in Dynamically Orientable Graphs

William Klostermeyer
Department of Statistics and Computer Science
West Virginia University
Morgantown, West Virginia 26506-6330
e-mail: wfk@cs.wvu.edu

Abstract

Motivated by a problem in railroad systems, a variant of the *k*-server problem in graphs is studied in which edges are oriented as they are traversed. The offline version of the problem is shown to be NP-complete, in sharp contrast to the polynomial-time solvable offline *k*-server problem. Approximation algorithms and algorithms for restricted classes of graphs are discussed.

1. Introduction

The *k*-server problem is concerned with constructing paths (walks) for each of *k* mobile *servers* to request points in a metric space. Formally, the *k*-server problem is as follows: a sequence $\sigma=\{r_1, r_2, ..., r_q\}$ of requests are made at points in a metric space M in which there are *k* servers to satisfy these requests. Requests must be served in the order they are made. To serve a request, at least one server must move to the requested point. The cost incurred by an algorithm is the total distance moved by its servers in serving σ. A server problem is called *offline* if all requests in σ are known in advance and *online* if the request are revealed one at a time. Results on server problems, primarily the online version, can be found in [1-7].

In this paper we assume the servers are trains travelling in a railroad system. Due to their weight, trains create a tremendous amount of friction as they move. Essentially, the friction heats the track to the point where it is gradually pushed in the direction of travel [8]. Consequently, it is desirable to have an equal number of trains travel in either direction along each piece of track, else the quality of the track deteriorates.

Using a graph to model the rail system, an edge between vertices u and v may be in one of three states: *neutral*, $<u, v>$, or $<v, u>$. That is, the edge may be undirected, or directed in either direction. All edges are initially undirected. When a train traverses an edge in one direction, say from u to v, the edge changes state: an undirected edge becomes a directed edge $<v, u>$, meaning the edge may only be traversed from v to u and a directed edge $<u, v>$ becomes neutral, meaning it may be traversed in either direction. An edge of the form $<v, u>$ may not be traversed from u to v until it is *neutralized* by a train travelling in the opposite direction. Such graphs are called *dynamically orientable graphs (dogs)*.

Let $G=(V, E)$ be a dog with $|V|=n$, $|E|=m$. For simplicity let each edge have weight equal to one, using weighted graphs does not affect the results. We are interested in studying the *k*-server problem in dogs. We term this the *k-train problem.* As usual, the objective is to construct a set of paths for the *k* trains so that the sum of the lengths of the paths is minimum over all such path sets that serve the requests in the given order.

In this paper, we study the offline k-train problem. The problem is shown to be NP-complete for $k \geq 1$, in sharp contrast to the offline k-server problem, which can be optimally solved in polynomial time for any k using dynamic programming techniques [1, 6]. Approximation algorithms and algorithms for some restricted classes of graphs are discussed.

2. NP-completeness of the Offline k-train Problem

The offline k-train problem is shown to be NP-complete for $k=1$. The decision version of the 1-train problem is formulated as, follows:

Instance: *Dog* G=(V, E), request sequence $\sigma=\{r_1, r_2, ..., r_q\}$, vertex v_1 which is the initial location of the train, and an integer d.

Question: Does there exist a path for the train:

$$(v_1, ..., r_1, ..., r_2 ..., r_{q-1}, ..., r_q)$$

such that the total distance travelled is at most d and the train traverses edges only in the proper direction? The length of the path is the number of edges traversed.

The decision version of the offline k-train problem can be formulated similarly.

Theorem 1. The 1-train problem is NP-complete.

Proof: The 1-train problem is easily seen to be in NP, since, given a initial vertex and a request sequence, it is easy to "trace" a given path in polynomial time to verify if it traverses the edges legally, if the total distance travelled it at most d, and if the requests are served in order.

To show 1-train is NP-hard, we reduce 3-SAT to 1-train. Recall that a 3-SAT instance consists of a set of boolean variables $U=\{u_1, u_2, ..., u_f\}$ and a set of clauses $C=\{c_1, c_2, ..., c_g\}$ over those variables [9]. The 3-SAT problem is to determine if there exists a truth assignment which satisfies C. The reduction is as follows. For each variable u_i construct the following subgraph, called a *variable subgraph*:

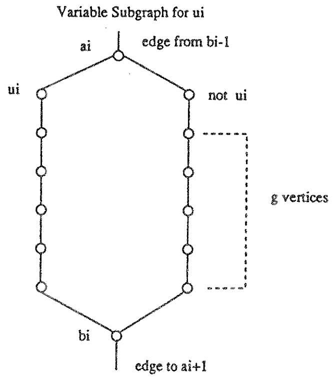
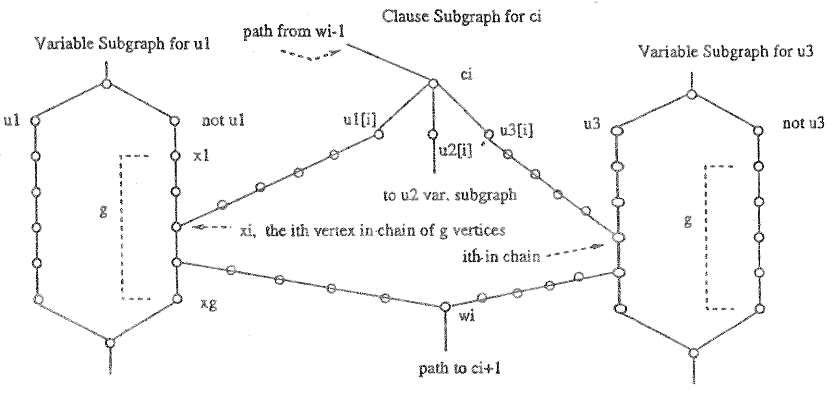


Figure 1. Variable Subgraph

Connect the variable subgraphs by adding edges (b_{i-1}, a_i), for all $i > 1$.

For each clause $c_i = (u_1$ OR u_2 OR $u_3)$ construct the following *clause* subgraph, noting its connection to the u_1, u_2, and u_3 variable subgraphs:



Note: Assumes u1 appears in ci as non-negated literal and u3 is negated literal in ci
u2 connection not shown in Figure

Figure 2. Clause Subgraph

We also add edges (b_f, b_{f1}), (b_{f1}, b_{f2}), (b_{f2}, b_{f3}), (b_{f3}, b_{f4}), (b_{f4}, b_{f5}), (b_{f5}, c_1) to connect the last variable subgraph to the first clause subgraph and edges of the form (w_i, w_{i1}), (w_{i1}, w_{i2}), (w_{i2}, w_{i3}), (w_{i3}, w_{i4}), (w_{i4}, c_{i+1}), for all $i < g$ to connect consecutive clause subgraphs.

Define $\sigma = \{a_1, b_1, a_2, b_2, ..., a_f, b_f, c_1, w_1, c_2, w_2, ..., c_g, w_g\}$, let a_1 be the initial vertex for the train, and set $d = (g+3)*f + 17*g$. The reduction can be seen to take only polynomial time. We now claim C is satisfiable if and only if there exists a path of length d or less that serves all the requests in the given order.

"→" Suppose C is satisfiable. Then there is a legal truth assignment (i.e. each variable in U is assigned either "true" or "false") that satisfies C. Consider the variables that are assigned true. Suppose u_1 is such a variable. Then a request at b_1 can be served by passing from a_1 (the initial location of the train) to b_1 via vertex u_1. Likewise, if we had assigned u_1 the value "false," we would serve the request at b_1 by passing from a_1 to b_1 via vertex $\neg u_1$. The rest of the first $2f$ requests are served in a similar manner, based on the truth assignment of the corresponding variable. That is, if u_i is "true," we serve request b_i by passing through vertex u_i and through vertex $\neg u_i$ otherwise. The total cost of serving these requests is $(g+3)*f - 1$, since each variable subgraph is of length $g+2$ (i.e. the distance from a_i to b_i) and there are $f-1$ (b_{i-1}, a_i) edges connecting variable subgraphs to one another.

We then move from b_f to vertex c_1 at a cost of six. Since C is satisfiable, each

clause contains a literal which is a *witness*, i.e. negated variable which is assigned "false" or a non-negated variable which is assigned "true". Suppose u_j is a witness for clause c_1. The request at w_1 is served by travelling from c_1 to $u_j[1]$ and then via a path of length five into the variable subgraph associated with variable u_j. Supposing variable u_j appears in clause c_i as a non-negated literal, this path leads us to the right-hand column of vertices in the variable subgraph. In order to proceed to request site w_1 at minimum cost, we need to traverse exactly one edge on this right-hand column of vertices in a top-to-bottom direction (else the train must take a more circuitous path, clearly spoiling any attempt at finding a complete path of length at most d--this is detailed in the "only if" part below). Note that the edges on this right-hand column can only be traversed by the train if the train previously went down the left-hand column on its way to serving b_1. In other words, if u_j appears in clause c_i as a non-negated literal, it can only satisfy the clause if it is assigned "true." Once the train traverses the edge in the variable subgraph, it may move unhindered to w_1 at a cost of five. We proceed in a likewise fashion with if u_j is a witness for clause c_1 and appears in clause c_1 as a negated literal--we shall pass through an edge in the left-hand column of vertices in the variable subgraph instead on our way to request w_1. The cost of moving from c_1 to w_1 is twelve. The train then moves at a cost of five to the request at vertex c_2. The request at w_2 is served in a similar way, by using a witness for clause c_2 and passing through an edge in the variable subgraph associated with that witness. Summing the costs for serving the requests in this way gives a cost of at most d since the first $2f+1$ requests cost $(g+3)f + 5$ to serve and the remaining request cost seventeen each, except the last which costs twelve (as the train stops when it reaches w_g).
"$\leftarrow$" Suppose there exists a path of length d or less that serves all the requests. We derive a truth assignment as follows. If the train served request b_i by passing through vertex u_i, we assign u_i "true," otherwise assign u_i "false." As claimed above, any path that satisfies all the requests must do so by traversing exactly one edge in a variable subgraph each time it is going from a c_i vertex to the request at vertex w_i. Otherwise, the path would have to move through one or more edges in this variable subgraph in a bottom-to-top direction (rather than the desired top-to-bottom direction). Doing so would force the train to visit vertices in another clause subgraph, say c_j, in addition to c_i, before serving the request at w_i. But this means the path travelled from c_i to w_i has either length greater than twelve or was not traversable, as is the case with certain instances when $j < i$. By traversable, we mean the edges are oriented (or neutral) so that the path can be legally traversed. Since there are consecutive requests at c_i and w_i, for $1 \leq i \leq g$, in order to achieve a cost of at most d, we must travel from c_i to w_i, for $1 \leq i \leq g$, at a cost of at most twelve. Since twelve is the length of the shortest path between c_i and w_i, it must be that the path contains no c_i to w_i traversals of cost more than twelve.

Continuing with the argument, we claim that the first vertex encountered on the train's path from c_i to w_i, say $u_j[i]$, is the "witness" for clause c_i. Because of the length of the path and the fact that it serves all the requests, the train must have passed through exactly one edge in the variable subgraph corresponding to variable u_j on its path from c_i to w_i. Then u_j is assigned "true" (assuming it is a non-negated literal) by our assignment, since the train traversed the u_j subgraph by going from a_j

to b_j via vertex $\neg u_j$. Therefore C has a legal, satisfying assignment.□

Theorem 1 also implies the NP-completeness for all $k > 1$ as the same reduction can be used by initially locating $k-1$ of the trains on isolated vertices.

3. Approximation Algorithms

As the offline k-train problems are NP-complete for any k, it is believed that no polynomial time algorithm can find the optimal solution for all instances. Therefore, it is of interest to develop an approximation algorithm. Recall from [9] that an approximation algorithm is a polynomial time algorithm whose goal is to find a near-optimal solution. The *performance ratio* of an approximation algorithm is essentially the maximum, over all problem instances, of the ratio of the value of the approximate solution to that of the optimal.

We shall describe a 2-competitive algorithm for the online version of the 1-train problem. Some terminology is needed first, however. An online algorithm A is said to be *c-competitive* if $C_A(\sigma) \leq c{*}C_{opt}(\sigma) + a$ for some constant a for all request sequences σ, where $C_A(\sigma)$ is the cost incurred by A for request sequence σ, and $C_{opt}(\sigma)$ is the optimal cost for any algorithm, including one that knows σ in advance. The quantity c is called the competitive ratio of algorithm A. Thus competitive analysis compares the cost incurred by an online algorithm to that incurred by an optimal off-line *adversary*. Therefore, assuming $a=0$, a c-competitive online algorithm also serves as an approximation algorithm with performance ratio c provided the algorithm runs in polynomial time. If $a > 0$, we must adjust the performance ratio somewhat. The competitive algorithm we give has $a=0$.

The online algorithm is as follows. Let v_0 be the initial location of the train and v its current location. Observe that it is possible for a 1-train algorithm to behave so that, at any step, the directed edges in G form a directed path P from v to v_0. That is, the only directed edges in G are those on path P. A 1-train algorithm can maintain this invariant by prudent use of backtracking: instead of neutralizing an edge (u, v) which would create disconnected directed paths, the train can backtrack along P until either (i) it reaches u again and then traverse (u, v) safely or (ii) it reaches a vertex w on P from which it can traverse an undirected/neutral path to v.

Our algorithm behaves as follows, under the constraint that the train always maintains a directed path P from its current location to its initial vertex and no other edges in G are directed except those on P. Upon receiving a request r, let w be the vertex in P having the shortest *traversable* path to r among all the vertices in P. By traversable, we mean the edges are oriented (or neutral) so that the path can be legally traversed. We use the term "sequence of edges" when discussing a path that may or may not be traversable. If more than one such w exists, choose the nearest to v. To serve the request, *backtrack* (if necessary, as w may equal v) along P to w and proceed to r.

We claim the path length from w to r is no greater than the length of the shortest sequence of edges from v to r. Suppose otherwise. Then there is a sequence of edges $S=(v, ..., v_a, v_b, ..., r)$ which is shorter than $(w, ..., r)$ where the edge between v_a and v_b is oriented as $<v_b, v_a>$. Then P is the directed path $(v, ..., v_b, v_a, ..., v_0)$. Consider the proper subsequence of S, $(v_b, ..., r)$. Since v_b is on P, we may set w equal to v_b and have therefore constructed a shorter path than S.

Furthermore, this path from w to r consists entirely of neutral edges. This latter is true because P is a path and not, for example, a directed tree. In this manner, the 1-train algorithm maintains its path P by traversing only neutral edges as "shortest path" edges and traversing directed edges only for purposes of backtracking.

Fact. The online algorithm has a competitive ratio of two.

Proof: Follows since the algorithm only traverses edges in a path no longer than the shortest path to each request and that each such edge is traversed at most twice.□

It is easy to show that no online 1-train algorithm can have a competitive ratio better than 2 [7]. Another algorithm for the 1-train problem is as follows: traverse the path to the request that minimizes the length of the path plus the change in the number of oriented edges in the graph. This may be proved 2-competitive by using a potential argument wherein the number of oriented edges is defined to be the potential function. The details are omitted. However, simulations indicate that this algorithm is somewhat better in practice than the directed path algorithm [11].

It is a simple matter to convert the online algorithm into an algorithm which takes σ as input and runs in time $O((n + m)*q)$. Hence the following result.

Theorem 2. A solution to the offline 1-train problem can computed in polynomial time with performance ratio two.

Proof: Follows from the fact that the online 1-train algorithm has competitive ratio two.□

We note that a 4-competitive online algorithm for the 2-train problem in complete graphs is given in [7], implying the existence of an approximation algorithm with performance ratio four. However, as we shall see in the next section, we do not know if complete graphs simplify the problem to the point of being polynomial-time solvable or not.

4. Restricted Graph Classes

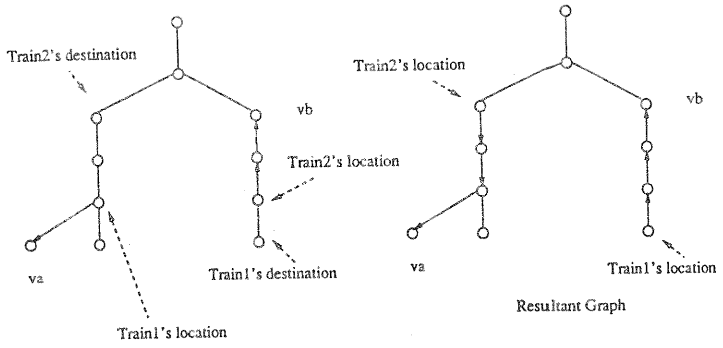
A common technique for coping with NP-complete problems is to construct polynomial time algorithms that behave optimally on restricted classes of inputs. For example, the reader will observe that the naive algorithm is optimal for the offline *1*-train algorithm when the input graph is a tree. In this case each request is served by traversing the unique path between the two vertices. It is not possible to encounter an edge oriented in the "wrong" direction, as trees are acyclic.

In this section we give an optimal algorithm for the 2-train problem in trees, 1-train problem in rings, and discuss the surprising difficulty of the 1-train problem in complete graphs.

4.1 Two Trains in a Tree

We show how to construct a dynamic programming algorithm to optimally solve the 2-train problem in a tree. Let $T=(V, E)$ be a tree with $|V|=n$ and let $\sigma=\{r_1, r_2, ..., r_q\}$. Construct a table of dimension $q \, X \, n^2$ for use with the algorithm. The offline 2-server algorithm of [1, 6] computes the optimal manner of serving the first m requests so that the servers reside on vertices v_i, v_j after serving request m. A key step in the optimal offline 2-server algorithm is computing the cost of moving from a previous pair of vertices (called a *configuration*) to configuration v_i, v_j. A modification of that algorithm is needed since the distance between two configurations may be complicated by the existence of oriented edges.

Suppose v_a and v_b are the train's initial locations. *table[m, i, j]* holds the optimal cost of serving the first m request so that the servers reside on vertices v_i, v_j after serving request m. Initially, *table[a, b, 0]* = 0 and all other entries are infinite. The algorithm iterates from one to q, computing each row of n^2 entries for each request. Assume row m has been computed. Then *table[m+1, i, j]* is computed as follows for request r_{m+1}. If $r_{m+1} \notin \{i, j\}$ then *table[m+1, i, j]* = ∞. Otherwise, *table[m+1, i, j]* = MIN(*table[m, x, y]* + *dist((x, y), (i, j))*), over all $1 \leq x, y, \leq n$. *dist((x, y), (i, j))* is computed in O(n) time by constructing the tree that results from having $train_1$ at v_a and $train_2$ at v_b initially and moving the trains to vertices x and y respectively. This may require that we first send $train_1$ to x and then send $train_2$ to y or vice versa. Once this is done, we attempt to send $train_1$ to vertex v_i. If $train_1$ has a traversable path to v_i, we count the distance it travelled and then send $train_2$ to v_j. If $train_1$ does not have a traversable path to v_i, we first move $train_2$ across the oriented edges so that $train_1$ may pass on to v_i. $Train_2$ then moves to v_j, if possible. The distance travelled by the two trains in moving to configuration v_i, v_j is the *dist* value returned. An example is shown in Figure 3 and Figure 4 shows an example when *dist((x, y), (i, j))* = ∞.



dist=11

Figure 3. *dist* Computation Example

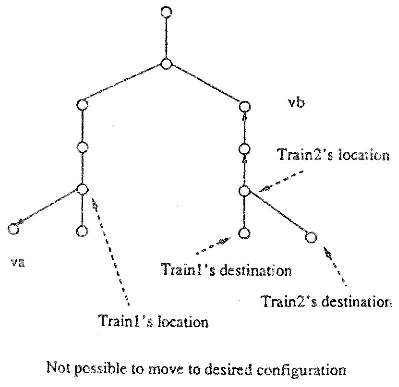


Figure 4. Another *dist* Computation Example

The naive implementation of the algorithm runs in O(qn^5) time and is easily modified to produce the optimal paths, rather than just the optimal cost. The reader may also observe that the algorithm can be significantly optimized by only considering those configurations which contain the requested vertex at each step. In this case, the number of different configurations that need to be considered at each step is O(n), rather than O(n^2). Similar dynamic programming algorithms can be conceived for the k-train problem, but the number of states is exponential in k, leading to impractical algorithms.

4.2 One Train in a Ring

Rings (or simple cycles) are a common restricted graph class for routing and transportation problems, see for example [10]. We give a linear time algorithm to find the optimal solution to the 1-train problem in a ring. This particular result is of interest because of lower bound of two on the competitive ratio for the online problem in rings [7]. Thus rings are a class of graphs for which the offline version of the problem is in some sense "easier" than the corresponding online version.

Let v_0 be the initial location of the train. Assume the n vertices in the ring are numbered from v_0 to v_{n-1} in a clockwise order. The algorithm is based on the observation that at any time there is a directed path from the current location of the train to v_0. At request r_i, we compute the optimal manner of serving the first i requests so that this directed path is directed in a clockwise direction or a counterclockwise direction. This is done as follows. Let $cost_1[i]$ give the optimal cost of serving the first i requests so that there is a clockwise directed path from r_i to v_0 and cost$_2$[i] give the optimal cost of serving the first i requests so that there is a counterclockwise directed path from r_i to v_0. Compute $cost_1[i+1] := MIN(cost_1[i] + dist_{cc}(r_i, r_{i+1}), cost_2[i] + dist_c(r_i, r_{i+1}))$, and compute $cost_2[i+1] := MIN(cost_1[i] + dist_{cc}(r_i, r_{i+1}), cost_2[i] + dist_c(r_i, r_{i+1}))$, where $dist_c(r_i, r_{i+1})$ is the clockwise distance from r_i to r_{i+1} and $dist_{cc}(r_i, r_{i+1})$ is the counterclockwise distance from r_i to r_{i+1}. The $dist(v_a, v_b)$ values are computed as follows for $cost_1$:

$$dist_c(v_a, v_b) = |a - b|$$

$$dist_{cc}(v_a, v_b) = a - b \text{ if } a \geq b \qquad dist_{cc}(v_a, v_b) = a + n - b \text{ if } a < b$$

The $dist_{cc}(v_a, v_b)$ value is as such because the train must first go to v_0 and then proceed to v_b, due to edge orientations. The $dist(v_a, v_b)$ values are computed as follows for $cost_2$:

$$dist_{cc}(v_a, v_b) = |a - b|$$

$$dist_c(v_a, v_b) = b - a \text{ if } a \leq b \qquad dist_c(v_a, v_b) = a + n - b \text{ if } a > b$$

This algorithm requires O(q) time to compute the optimal path for a sequence of length q. The algorithm's optimality is easily proved.

4.3 One Train in a Complete Graph

Complete graphs are a particularly interesting case of the problem, due in part to the following fact.

Fact 3. In the k-train problem in a complete graph, suppose there is a train at vertex u and no train at vertex v. Then there exists a (traversable) path of length at most two from u to v.

Proof: Suppose otherwise. Then (u, v) is oriented as $<v, u>$. The outdegree of u plus the number of undirected edges incident to u is at least as large as the indegree of u, since there is a train at u. Likewise, the indegree of v plus the number of undirected edges incident to v is at least as large as the outdegree of v, since there is no train at v. Vertex u must have a path of length one to at least $\lceil (n-1)/2 \rceil$ other vertices. If there is no path of length two from u to v, it must be that each vertex w which can be reached from u by a path of length one has an edge of the form $<v, w>$. This implies v has outdegree at least $\lceil (n-1)/2 \rceil + 1$. Suppose n is even. Then v has outdegree at least $n/2 + 1$. Since v has degree $n-1$, it must be that the indegree of v plus the number of undirected edges incident to v is at most $n/2 - 2$. But this cannot be, since any vertex's indegree and outdegree can differ by a most one. The case when n is odd follows in a similar fashion.□

At first glance, an optimal 1-train algorithm would simply behave in a greedy fashion: serving requests at cost one if possible and cost two otherwise. This approach is doomed, however. For example, suppose we have a four vertex complete graph where the train is initially at v_1. Now consider $\sigma=\{v_2, v_3, v_4, v_1, v_2\}$. Using the greedy method, we must clearly serve the last request (at v_2) at cost two. Suppose we extend σ to be $\sigma=\{v_2, v_3, v_4, v_1, v_2, v_3\}$. Notice there are two possible ways to serve the second v_2 request: either via v_3 or v_4. In this case, it is obvious that moving from v_1 to v_3 and then to v_2 enables us to serve the final request at v_3 at a cost of one. Whereas if we had moved from v_1 to v_4 and then to v_2, the final request at v_3 would cost two. This example is rather simple, but in general it is not so obvious which path has the most beneficial effect on future requests, even though we know the future! That is to say, there is a combinatorial explosion of possible ways to serve each request: each time we traverse edge (u, v), we affect future requests at u and v, as well as future paths that may wish to use edge (u, v) en route to a request.

Finding a polynomial time optimal algorithm for the 1-train problem in a complete graph is further complicated by the following, non-intuitive result.

Fact 4. There exists a request sequence where the optimal solution, *Opt*, serves certain individual requests at costs greater than two, and every solution that serves each request at cost at most two has total cost greater than *Opt*.

Proof: The following example demonstrates this fact. Consider a complete graph on six vertices: a, b, c, d, e, f. Let the train initially be a vertex a and let $\sigma=\{a, d, f, c, d, b, e, c, a, d, e, b, d, e, c, a\}$. $Opt = (a, d, f, c, d, b, e, c, \underline{a, c, e, d}, e, b, d, e, c, a)$. The underlined requests indicated where *Opt* has served request d by traversing a path of length three. The reason this path of length three is useful is that the subsequent requests can each be served at cost one. The last requests, such as the final request at a, cannot be served at cost one by any algorithm unless that algorithm has neutralized edge (c, a) previously; which is necessitated by the eighth and ninth requests in σ, which are at c and a. An examination of all possible alternate solutions that do not use paths of length three confirms that none are better *Opt*. Some of these solutions incur the exact same cost as *Opt* on this sequence, but leave their graph in a different configuration (i.e. a different set of edge orientations) than *Opt*. It is then easy to extend σ to create a sequence σ' which exploits the differences in configurations so as to require the other solution to incur a greater cost than *Opt* on σ'.□

The reason Fact 4 makes a polynomial-time solution harder to obtain is that we cannot bound the search at any step to, say, explore paths of length two only. In conclusion, we leave open the complexity of the 1-train problem in complete graphs, as well as the k-train problem in complete graphs. Finding polynomial-time algorithms or proving NP-completeness seems quite difficult.

References

1. M. Chrobak, H. Karloff, T. Payne,and S. Vishwanathan, "New Results on Serevr Problems," *SIAM J. Disc. Math*, 11 (1991), pp. 172-181
2. M. Chrobak and L. Larmore, "Generosity Helps, or an 11-Competitive Algorithms for Three Servers," *Proc. Third ACM-SIAM Symp. on Disc. Alg.*, 1992, pp. 196-202
3. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young, "Competitive Paging Algorithms," *Journal of Algorithms*, 12 (1991), pp. 685-699
4. A. Fiat, Y. Rabani, and Y. Ravid, "Competitive *k*-Server Algorithms," *Proc. 31st Symp. on Foundations of Computer Science*, 1990, pp. 452-469
5. E. Koutsoupias and C. Papadimitriou, "On the *k*-server Conjecture," Proc. 26th ACM Symp. on Theory of Computing, 1994, pp. 507-511
6. M. Manasse, L. McGeoch, and D. Sleator, "Competitive Algorithms for Server Problems," *Journal of Algorithms*, 11 (1990), pp. 208-230
7. W. Klostermeyer, "The *k*-server Problem in Dynamically Orientable Graphs," Technical Report 95-1, Dept. Stat. and Computer Science, West Virginia University
8. C. Brad Boyse, *CSX Corporation*, Jacksonville, Florida, personal communication, 1992
9. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979
10. G. Frederickson, "A Note on the Complexity of a Simple Transportation Problem," *SIAM. J. Comput.*, vol. 22 (1993), pp. 57-61
11. B. Schneider, "Approximation Algorithms for the *k*-train Problem," Masters Report, Dept. Stat. and Computer Science, West Virginia University, May 1995

(Received 28/3/95)