Available online at www.sciencedirect.com
SCIENCE DIRECT®
Computers & Operations Research 31 (2004) 499–513
computers & operations research
www.elsevier.com/locate/dsw

Finding the first K shortest paths in a time-window network

Yen-Liang Chen[a,*], Hsu-Hao Yang[b]

[a] *Department of Information Management, National Central University, Chung-Li, Taiwan 320, Republic of China*
[b] *Department of Industrial Engineering and Management, National Chinyi Institute of Technology, Taiping, Taiwan 411, Republic of China*

Received 1 June 2002; received in revised form 1 October 2002

Abstract

The time-constrained shortest path problem is an important generalization of the classical shortest path problem and has attracted much research interest in recent years. We consider a time-window network, where every node in the network has a list of pre-specified windows and departure from a node may take place only in these window periods. The objective of this paper is to find the first K shortest paths in a time-window network. An algorithm of time complexity of $O(Kr|V|^3)$ is developed to find the first K shortest paths, where $|V|$ is the numbers of nodes in the network and r represents the maximum number of windows associated with a node.

Scope and purpose

Time window has been a common form of time constraint considered in the literature. Basically, a time window is a time period, defined by the earliest and latest times, when a node is available for traveling through. There are many practical situations where time windows can be used to describe the time constraints associated with the nodes and arcs on a network. For example, a time window in a transportation network may be the time period that a service or transition facility is available for the traveler to pass through. Although there are many researches on the transportation problem in time-window networks, no previous researches study how to find the first K shortest paths in a time-window network; hence, this paper studies this extended problem. The extension has many potential applications in practice. For example, there may be some complex constraints associated with the nodes and/or arcs of the network, which are difficult to incorporate into the model formulation. A solution strategy is to temporarily ignore these constraints and obtain a list of paths in nondecreasing order in terms of their total times. The optimal solution can be found by identifying the first path in the list that satisfies the constraints. Another use of the result is a quick selection of an alternative path when encountering temporary disconnection of the active shortest path.

Keywords: Network; Time-window constraint; Shortest path; Simple path

* Corresponding author. Tel: +886-3-4267266; fax: +886-3-4254604.
E-mail address: ylchen@mgt.ncu.edu.tw (Y.-L. Chen).

0305-0548/04/$ - see front matter © 2003 Elsevier Ltd. All rights reserved.
doi:10.1016/S0305-0548(02)00230-7

1. Introduction

The classical shortest path problem is concerned with finding the path with the minimum time, distance, or cost from a source node to a destination through a connected network. It is an important problem because of its numerous applications and generalizations in transportations [1], communications [2], and many other areas. Several excellent reviews on the shortest path problem have been presented [3–5].

The time-constrained shortest path problem (TCSPP) is an important generalization of the shortest path problem and has attracted much research interest over the past years. The basic notion is to consider when a node, under time constraints, in the network can be visited. The time window has been a common form of time constraints considered in the TCSPP, which requires that a node can be visited only in a specified time interval [6–9]. Restated, a time window defines the earliest time and the latest time that the node is available. In principle, two types of time windows are available. The first is the hard time window, where, if one or more time-window constraints are not satisfied, then the solution becomes infeasible [7–9]. The second is the soft time window, where a cost penalty is incurred if the node's arrival is outside its time window [6].

This paper is interested in finding the first K shortest paths in a time-window network. Many potential applications in practice justify the extension. For example, it is not unusual that some extra constraints may have been present that pose great challenges to model the problem. One common solution strategy would be to temporarily ignore these constraints and rank a list of paths in nondecreasing order in terms of the total time, distance or cost. It then leaves the decision-maker to determine the optimal solution by identifying the first path in the list that satisfies the extra constraints.

Finding the first K shortest paths can be classified into two main categories. In the first category, only simple paths, i.e., paths without repeated nodes, are allowed. In the other one, looping paths, i.e., paths with repeated nodes and arcs, can be considered as solutions. Regardless of the nature of the network under consideration, the problem of finding simple paths appears to be harder than that of finding looping paths. Yen's algorithm [10] is a well known algorithm for finding the first K simple paths in a general network with $|V|$ nodes, which requires time $O(K|V|^3)$. If the network is undirected, Katoh et al. [11] improved the time bound associated with Yen's algorithm to $O(K(|A| + |V|\log|V|))$, where $|A|$ is the number of arcs. As for finding the first K looping paths, Fox [12] developed an algorithm with time complexity of $O(|V|^2 + K|V|\log|V|)$. Recently, Eppstein [13] used an implicit representation of paths to significantly improve the time bound further to $O(|A| + |V|\log|V| + K|V|)$.

Because of the time-window constraint, the paths enumerated in this paper are different from the conventional paths appeared in a network without time-window constraints. Let us consider a path route $(s, \mathrm{A}, \mathrm{D}, d)$ shown in Fig. 1, where the number on an arc is the travel time and there are several windows associated with the nodes. Traditionally, we view this route as a single path with total time 10, because we will arrive and leave node A at time 4, arrive node D at time 8, and finally arrive node d at time 10. In other words, the traditional researches assume that a traveler will leave a node once he is allowed to leave. For simplicity, we may call this as "no-extra-waiting" condition.

In practice, taking an extra wait on a node is not uncommon; hence, this paper will give up the traditional "no-extra-waiting" condition. Instead, we assume that once arriving a node we may

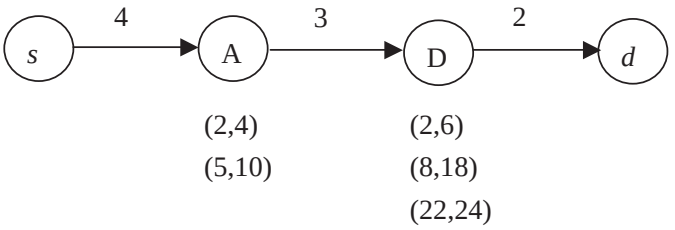

Fig. 1. A path route in a time-window network.

choose to stay on a node for a while rather than leave immediately. Specifically, when arriving a node, if the arrival time is not within a window, we have to wait for the next window. But if the arrival time is within a window, we have two choices: (1) leave right away, or (2) wait. In case that we choose to wait, we will wait until the next window and then make a wait-or-leave decision again. Repeatedly doing this way, we will finally leave in a window after bypassing a certain number of windows.

Let us use the path route $(s, \mathrm{A}, \mathrm{D}, d)$ shown in Fig. 1 for illustration. There are many possible paths associated with this path route because of different waiting times on each node. Let x_t represent that the departure time of node x in the path is t. Then, we have two possible paths in this case:

$$(s_0, \mathrm{A}_4, \mathrm{D}_8, d_{10}) \quad \text{and} \quad (s_0, \mathrm{A}_5, \mathrm{D}_8, d_{10}).$$

Apparently, they are different paths but with the same route. Therefore, the first K paths enumerated by our approach are by no means the same as those generated by the traditional approach, because path route and path represent different concepts in our approach. As a result, the objective of this paper is to find the first K shortest paths with waiting times in the network. The paper is organized as follows. In the next section, we first define the problem and then develop some auxiliary functions that will be used in the algorithm that follows. In Section 3, we propose the algorithm for finding the first K shortest paths in a time-window network and also analyze its time complexity. Finally, we give the conclusion and directions for future research in Section 4.

2. Problem definition and auxiliary functions

We begin with defining the problem in Section 2.1. In Section 2.2, we give some supporting functions that are used to develop the algorithms in Section 3.

2.1. Problem definition

Let $N = (V, A, WL, t, s, d)$ denote a time-window network, where V is the node set, A is the arc set without multiple arcs and self-loops, $t(u, v)$ is the travel time of arc $(u, v) \in A$ and s and d are the source and destination of the path. For each node $u \in V$, it is associated with a window-list $WL(u) = (w_{u,1}, w_{u,2}, \ldots, w_{u,r})$, where $w_{u,i}$ is the ith time window of node u for $i = 1$ to r. Each window $w_{u,i}$ is associated with a starting time $s_{u,i}$ and an ending time $e_{u,i}$, where $s_{u,i} < e_{u,i} < s_{u,i+1}$ for any $i \geqslant 1$. When arriving a node, if the arrival time is not in a window period, we must wait.

Otherwise, we can leave right away or wait. In case of waiting, we need to wait until the next window is coming; then, we have to make a wait-or-leave decision once again. Repeatedly doing this way, we may skip a number of windows before we finally leave a node.

In this paper, "path route" and "path" have different meanings. The former is just a listing of the nodes along the route, whereas the latter is a listing of the nodes as well as their departure times. In other words, a path route can be mapped to many corresponding paths. Let us call the arrival time to the destination node the total time of a path. Then, the goal of this paper is to enumerate the first K simple paths from the source to the destination in nondecreasing order in terms of their total times.

2.2. Some auxiliary functions

Since a path route may correspond to many different paths by including different waiting times, the first function we must have is, if we are given a path route P and a given total time T, how many paths can be mapped to this particular path route P and total time T. Let us use a simple example to explain what this function is for. For example, assume that our goal is to generate the first 30 paths. Then, we may first generate the first path P_1 with total time T_1, and assume that, by applying this function, we find that there are 15 paths that can be generated from P_1 and T_1. So, we first output the first 15 paths with the same route but different waiting times. After that, we then generate the 16th path in the network. Let P_{16} be this path and T_{16} be its total time. Suppose this time we find that there are 19 paths corresponding to P_{16} and T_{16}. Since this is a little more than what we require, we only need to output 15 of them as the solution. This way, we successfully generate the first 30 paths in the network.

Here, we will present how to find paths corresponding to path route P and total time T. Since a number of different paths may arise from the inclusion of waiting, the core of our approach is to construct a graph, named *related-graph*, that answers two questions: (1) how many of paths corresponding to P have a total time T, and (2) who these paths are. In this section, we design the function *related-pno*(P, T) to answer the first question, and *related-paths*(P, T) the second question. In addition, the *related-graph* serves to compute P's next total time (denoted *next*) later than T.

Before going into the details, we use an example to explain the ideas. Let us consider the path P shown in Fig. 1, and assume that $T = 10$. Our objective is to construct the related-graph shown in Fig. 2a so that we can find all paths corresponding to P with the same total time but different waiting times. The rationale behind this transformation is that every path from s to d in the constructed graph maps to a unique path coming from P and T, and vice versa.

For each node u along the path route P, there are two cases that we will leave node u for the next node. First is at the beginning time of a window of node u that is no later than T. Second is at the very moment that we arrive node u from its preceding node and then we decide to leave right away. Let the numbers inside the boxes denote the beginning times of those windows no later than T. Then, we will generate nodes A_2 and A_5 for node A and nodes D_2 and D_8 for node D. Besides, we also create node s_0 for node s and node d_{10} for node d because they are the source and destination, respectively. To identify the possible moments that we can leave a node because of the second case, we process nodes s, A, D, d sequentially. After leaving from node s, we arrive at node A at time 4. Since time 4 is in window $(2, 4)$ of node A, it is a possible leaving moment and we add A_4 into the graph without box. After that, we find that the possible times leaving node A are

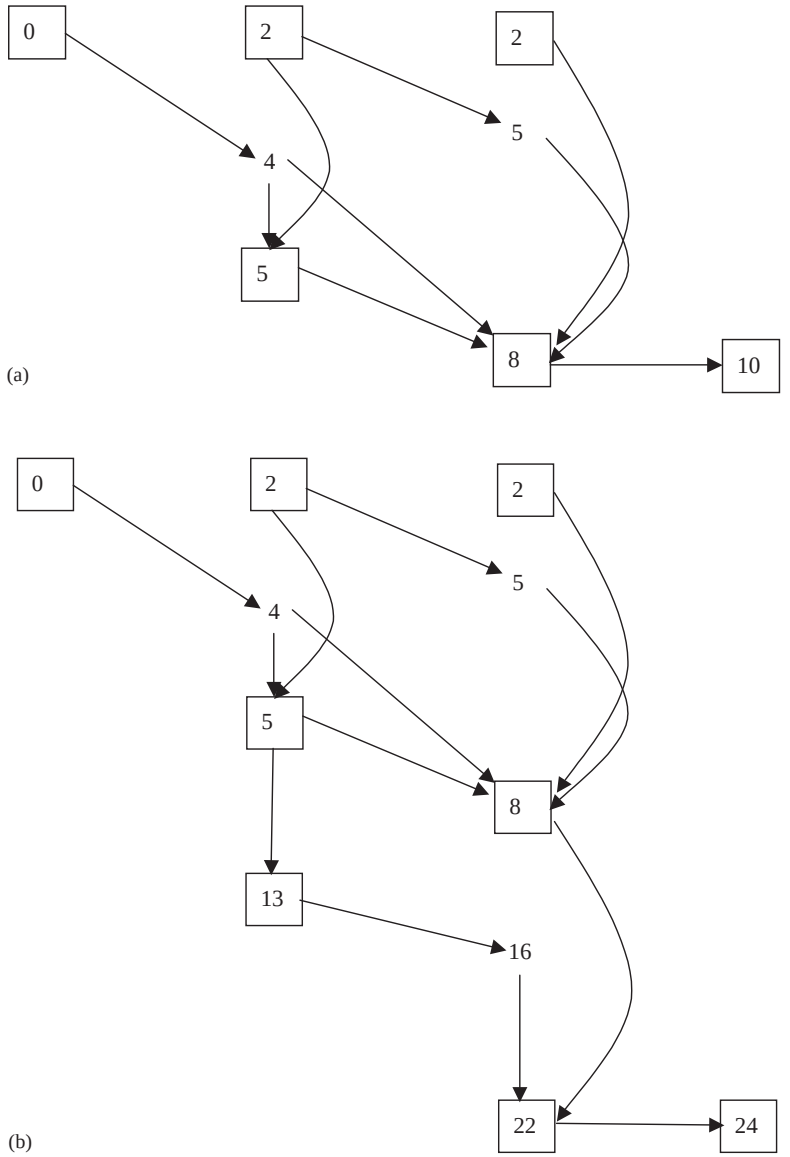

Fig. 2. (a) The related-graph constructed from path $P = (s, \mathrm{A}, \mathrm{D}, d)$ and $T = 10$. (b) The related-graph constructed for a modified case.

2, 4 and 5. Among all of them, only leaving from node A at time 2 will enter a window $(2, 6)$ of node D, so we add D_5 into node D without box. In all the other leaving times, we will be forced to wait until the next window has come. Treated this way, all the nodes in the constructed graph are shown either by numbers within boxes or otherwise. Finally, note that, if we decide to wait on a node, we will wait until the beginning time of the next window and then make a decision again. To describe this relationship in the constructed graph, we create an arc from a number, either with

box or otherwise, to the smallest larger number with box. That is why we create arcs (A_2, A_5) and (A_4, A_5) for node A. By the same reason, we create arcs (D_2, D_8) and (D_5, D_8) for node D.

In the constructed graph, there are two kinds of arcs. For example, arcs (A_2, A_5) and (A_4, A_5) represent waiting occurs on the same node. In contrast, arc (A_2, D_5) means that the earliest time to leave from node D is at time 5 if coming from node A at time 2.

In Fig. 2a, there may exist a number of paths from s_0 to d_{10}, path (s_0, A_4, D_8, d_{10}) is but one of them. Comparing with the other paths, say $(s_0, A_4, A_5, D_8, d_{10})$, we find that every path in the graph has the same path route and the same total time but differs from any other one in the waiting time. To find all the paths from s_0 to d_{10}, the function *related-paths*(P, T) can be done simply by a breadth-first search or a depth-first search. While searching, we also count the number of paths simultaneously so that the function *related-pno*(P, T) is solved by hitchhiking. For instance, by searching the graph in Fig. 2a, we find 2 paths, which are (s_0, A_4, D_8, d_{10}) and $(s_0, A_4, A_5, D_8, d_{10})$.

Now, we will describe how to construct the graph. Suppose we have a function *window-beginning* (u, T) that can output the beginning times of node u's windows which start no later than T. Let $P = (s = v^0, v^1, \ldots, v_i, \ldots, v^m = d)$. For each node v^i in P, calling the function *window-beginning*(v^i, T) will produce a list $ts(v^i) = (ts(v^i, 1), ts(v^i, 2), \ldots, ts(v^i, n^i))$, where $ts(v^i, j)$ is the beginning time of the jth window and n^i is the total number of windows with beginning times no later than T. Without loss of generality, assume that $n^0 = 1$, $ts(s) = (0)$ and we leave node s at time 0. Leaving from node v^i at time p, we will reach node v^{i+1} at time $p + t(v^i, v^{i+1})$. Then, there are two cases: (1) if the arrival time is in a window, we either leave from node v^{i+1} immediately or wait until a later window, or (2) if the arrival time is not in a window, we must wait. Either way, let q denote the earliest time we can leave from node v^{i+1} for node v^{i+2}. Recall that at time q, we still have the option to wait or leave.

With the preceding provision, we are ready to construct the related-graph by the algorithm presented below, where the meanings of nodes and arcs are as follows.

node v_q^i: make a leave-or-wait decision on v^i at time q,

arc (v_p^i, v_q^{i+1}): time q is the earliest time to leave from v^{i+1} if leaving from v^i at time p,

arc (v_p^{i+1}, v_q^{i+1}): wait at v^{i+1} from time p to time q,

arc (v_p^{m-1}, v_T^m): reach v^m at time T if leaving from v^{m-1} at time p.

We classify all nodes in this graph into two disjoint sets. For node v_q^i, if q is the beginning time of a window, it will belong to the set V_1^i. On the other hand, if q is not the beginning time of a window, then node v_q^i will be a member of the other set V_2^i. Similarly, all arcs are classified into two disjoint sets, where the arcs in A_1^{i+1} denote the waiting occurred on node v^{i+1} and A_2^{i+1} denotes the traveling from v^i to v^{i+1}.

Network Construction Algorithm

1. $s' = s_0$, $d' = v_T^m$, $A' = \phi$, $V' = V^0 = \{s_0\}$, and $V^m = \{v_T^m\}$.
2. For every node v^i from v^0 to node v^{m-2} in path P do

$$V_1^{i+1} = \{v_q^{i+1} \mid q \in ts(v^{i+1})\}$$

$$V_2^{i+1} = \{v_q^{i+1} \mid \exists v_p^i \text{ in } V^i \text{ such that } p + t(v^i, v^{i+1}) = q \text{ and } q \text{ is in a window}\}$$

$$V^{i+1} = V_1^{i+1} \cup V_2^{i+1}$$

$$V' = V' \cup V^{i+1}$$

$$A_1^{i+1} = \{(v_p^{i+1}, v_q^{i+1}) \mid v_p^{i+1} \in V^{i+1}, v_q^{i+1} \text{ is the node with the smallest subscript in } V_1^{i+1} \text{ satisfying } p \leqslant q\}$$

$$A_2^{i+1} = \{(v_p^{i}, v_q^{i+1}) \mid v_p^{i} \in V^{i}, v_q^{i+1} \text{ is the node with smallest subscript in } V^{i+1} \text{ satisfying } p + t(v^i, v^{i+1}) \leqslant q\}$$

$$A' = A' \cup A_1^{i+1} \cup A_2^{i+1}$$

3. $A^m = \{(v_p^{m-1}, v_T^m) \mid p + t(v^{m-1}, v^m) = T\}$.

Lemma 1. *The transformation requires time $O(|V|^2 r)$, where r is the maximum number of windows in a node and $|V|$ is the number of nodes in network N.*

Proof. See Appendix A. □

Having constructed the related-graph, the function *related-paths*(P, T) can be done simply by a breadth-first search or a depth-first search. While searching, we also count the number of paths simultaneously so that the function *related-pno*(P, T) is solved by hitchhiking. Since the depth-first search or the breadth-first search requires a time in a linear function of the network size, the total time for solving functions *related-paths*(P, T) and *related-pno*(P, T) can be done in time $O(|V|^2 r)$.

Finally, what remains is to find the next total time of path P that is later than T, which will be handled by the function *next*(P, T). It can be done by the following steps:

1. Find in $ts(v^{m-1})$ the smallest value larger than $T - t(v^{m-1}, v^m)$. Let it be T^{m-1}.
2. Use *Network Construction Algorithm* to build the graph for P with time $T^{m-1} + t(v^{m-1}, v^m)$.
3. Find in V^{m-1} the smallest leaving time value larger than $T - t(v^{m-1}, v^m)$. Let it be T'.
4. Return $T' + t(v^{m-1}, v^m)$ as the solution.

Let us use Fig. 2a for illustration. Step 1 sets T^{m-1} as 22, step 2 constructs a graph for P with total time $22 + 2 = 24$, step 3 finds T' as 22 and finally step 4 returns $22 + 2 = 24$ as the next total time. For comparison, let us add a new window $(13, 15)$ into node A. Then, if we leave node A at time 13, then we will arrive node D at time 16, which is within window $(8, 18)$ of node D. If we use this modified case to run the above algorithm, the execution will be like this: Step 1 sets T^{m-1} as 22, step 2 constructs a graph for P with total time $22 + 2 = 24$, step 3 finds T' as 16 and finally step 4 returns $16 + 2 = 18$ as the next total time. For illustration, we also draw the related-graph for this modified case in Fig. 2b.

The time complexity of function *next*(P, T) is $O(|V|^2 r)$, for the most time-consuming part is in step 2 that calls *Network Construction Algorithm.*

3. The framework of the algorithm

Before going into the detailed steps of the algorithm, we briefly describe the framework of our solution strategy as follows. Throughout the execution of the algorithm, we use a heap data structures, denoted as Q, to store a set of paths as well as their total times. With the support of this data structure, we can quickly find the path with the smallest total time, add a new path and delete an existing path. The algorithm iterates the following steps repeatedly until the number of paths enumerated exceeds K. Suppose the first $(r-1)$ paths have been generated, and our current job is to enumerate the rth path. To this end, we will select the path with the smallest total time from Q. Let P_r be the path found and T_r its total time. Suppose there are n_r paths having the same path route as P_r and total time T_r, where n_r is obtained by function *related-pno*(P_r, T_r). Then, we can output these n_r paths as the rth path, the $(r+1)$th path,..., the $(r+n_r-1)$th path, where these paths are generated by function *related-paths*(P_r, T_r). After outputting these paths, we will use the function *next*(P_r, T_r) to compute the next largest total time of path P_r. Let it be T. We then store path P_r and its new total time T into the heap data structure Q. (The reason why we need to do so is because a path route may appear multiple times in the first K paths but with different total times.) In addition, if the path route P_r is enumerated for the first time, then we will further partition all the paths into different exclusive sets. For example, let the route of P_r be (A, B, C, D, E). Then, we partition all the still-yet-to-be-found paths into four sets: those with suffix path (B, C, D, E) but without arc (A, B), those with suffix path (C, D, E) but without arc (B, C), those with suffix path (D, E) but without arc (C, D) and finally those without arc (D, E). We then find the shortest path for each of these four sets and then store all these four paths into Q. Repeatedly doing this way, we will finally obtain all the first K paths.

Now, we formally describe our solution strategy. Let $\boldsymbol{P}_c$ be the set of all the paths from s to d in N. Initially, we find the first shortest path $P_1 = (s = v^0, v^1, \ldots, v^m = d)$ with a total time T_1. Since there are *related-pno*(P_1, T_1) paths corresponding to P_1 with the same total time but different waiting times, the path we are searching next should be different from what have been found unless we have identified as many as K paths. For this reason, let $\boldsymbol{P}_c$-*related-paths*(P_1, T_1) be the set of paths containing all the paths in $\boldsymbol{P}_c$ but not those in *related-paths*(P_1, T_1). Define $\boldsymbol{P}_c^{(i)}$, $i = 1, 2, \ldots, m$, as the subset of the paths in $\boldsymbol{P}_c$-*related-paths*(P_1, T_1) that includes the subpath $(v^i, v^{i+1}, \ldots, v^m = d)$ but excludes the arc (v^{i-1}, v^i). In this context, we define that for $\boldsymbol{P}_c^{(i)}$, $(v^i, v^{i+1}, \ldots, v^m = d)$ is an *in-subpath* and (v^{i-1}, v^i) is an *out-arc*. It is trivial to verify that $\boldsymbol{P}_c$-*related-paths*(P_1, T_1) can be partitioned into m disjoint path subsets $\boldsymbol{P}_c^{(1)}, \boldsymbol{P}_c^{(2)}, \ldots, \boldsymbol{P}_c^{(m)}$.

Let $x = \textit{related-pno}(P_1, T_1) + 1$, and let P_x denote the xth shortest path with a total time T_x. Suppose P_x is in $\boldsymbol{P}_c^{(r)}$. We then partition $\boldsymbol{P}_c^{(r)}$-*related-paths*(P_x, T_x) into disjoint subsets in the same way we partition $\boldsymbol{P}_c$-*related-paths*(P_1, T_1). The subsets obtained from the partition of $\boldsymbol{P}_c^{(r)}$-*related-paths*(P_x, T_x), together with $\boldsymbol{P}_c^{(1)}, \boldsymbol{P}_c^{(2)}, \ldots, \boldsymbol{P}_c^{(r-1)}, P_c^{(r+1)}, \ldots, P_c^{(m)}$, constitute a partition of $\boldsymbol{P}_c$-{*related-paths*(P_1, T_1), *related-paths*(P_x, T_x)}. Next, let P_y be the shortest path found in the subsets that partition $\boldsymbol{P}_c$-{*related-paths*(P_1, T_1), *related-paths*(P_x, T_x)}, where $y = \textit{related-pno}(P_x, T_x) + \textit{related-pno}(P_1, T_1) + 1$. Following the same procedure allows us to generate the paths successively in nondecreasing order.

Recall that all the paths in the path set $\boldsymbol{P}_c^{(i)}$, for $1 \leqslant i \leqslant m$, contain an in-subpath from node v^i to d and exclude an out-arc (v^{i-1}, v^i). Suppose P_x is in $\boldsymbol{P}_c^{(r)}$, and let P_x be denoted by $(s =$

$u^0, u^1, \ldots, u^{r'} = v^r, v^{r+1}, \ldots, v^m = d$). The path set $\boldsymbol{P}_c^{(r)}$-*related-paths*(P_x, T_x) can be further partitioned into r' disjoint subsets. We define $\boldsymbol{P}_c^{(r,i)}$ as the subset of paths in $\boldsymbol{P}_c^{(r)}$-*related-paths*(P_x, T_x) with the in-subpath $(u^i, u^{i+1}, \ldots, u^{r'} = v^r)$ and without the out-arc (u^{i-1}, u^i). By including the original in-subpath and excluding the out-arc of $\boldsymbol{P}_c^{(r)}, \boldsymbol{P}_c^{(r,i)}$ is the subset of paths with in-subpath $(u^i, u^{i+1}, \ldots, u^{r'} = v^r, v^{r+1}, \ldots, v^m = d)$ and without out-arcs (u^{i-1}, u^i) and (v^{r-1}, v^r). Repeatedly applying the procedure, we have the following property.

Property 1. *Let* $\boldsymbol{P}$ *be a path subset in the partition of* $\boldsymbol{P}_c$-{*related-paths*(P_1, T_1), *related-paths*$(P_x, T_x), \ldots$}. *Then, all the paths in* $\boldsymbol{P}$ *contain an in-subpath from a certain node* u^* *to* d *and exclude an out-arc set.*

After dealing with the partition of path subset, our next objective is to find the shortest path from s to d in the path subset that includes a given in-subpath from a certain node u^* to d and excludes a given out-arc set. For ease of presentation, let P^{in} and A^{out} denote the in-subpath and out-arc set, respectively, and refer to the path satisfying the constraints P^{in} and A^{out} as the constrained path. The following section discusses how to find the shortest constrained path.

3.1. How to find the shortest constrained path

To find the path, we construct a network $N' = (V', A', s', d')$ so that the shortest path P^* from $s' = s$ to $d' = u^*$ in N' followed by P^{in} forms a shortest constrained path from s to d in N. The procedure of constructing N' is as follows.

1. All the arcs in P^{in} are removed from N, because P^* does not pass through these arcs.
2. All the arcs in A^{out} are removed from N.
3. Set $s' = s$ and $d' = u^*$.

At this point, we have several observations.

1. Since the constructed network N' is a time-window network rather than a conventional network, conventional shortest path algorithm (such as Dijkstra's) cannot be directly applied to the constructed network N'. However, after a small modification Dijsksta's algorithm [14] can be used to find the shortest path in a time window network. We briefly sketch the modified Dijkstra Algorithm: (1) two labels, the arrival time and the leaving time, are attached with each node. (2) In the beginning, all the nodes are in the set S. (3) Each cycle selects and removes the node with the minimal leaving time label from S. (4) The labels of the nodes adjacent to the selected node must be updated. For each adjacent node, we first determine the arrival time label. Then, we determine the leaving time label by using the time windows of this node as well as its arrival time label. (5) The algorithm stops if S becomes empty.
2. It is easy to see that the shortest path P^* from s' to d' in N' followed by P^{in} will form the shortest constrained path from s to d in N. Based on this observation, we have the following algorithm to find the shortest constrained path from s to d in network N subject to the conditions P^{in} and A^{out}.

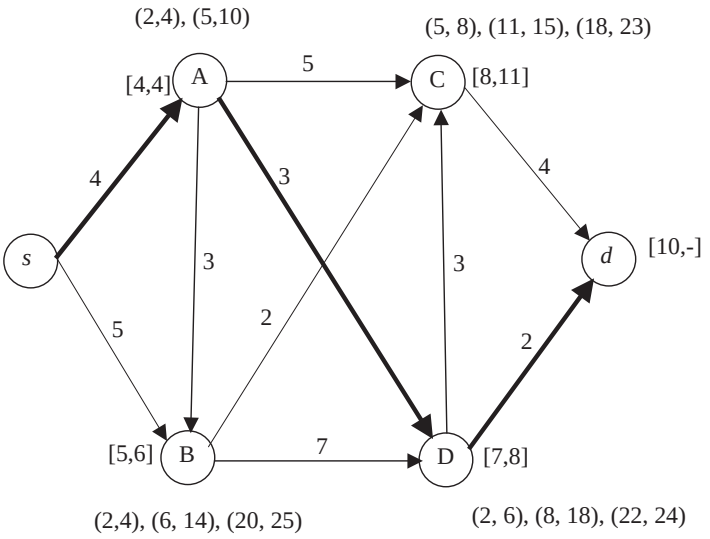

Fig. 3. The original time-window network and the first shortest path.

Shortest constrained path algorithm

1. Transform N into N'.
2. Apply the modified Dijkstra's algorithm to find the shortest path P^* from s to d' in N'.
3. Append P^{in} to the path P^* to obtain the shortest constrained path in N subject to the conditions P^{in} and A^{out}.

To illustrate the algorithm, we first consider the network shown in Fig. 3, where $P_1 = (s, \text{A}, \text{D}, d)$ with a total time $T_1 = 10$, and the numbers in the square bracket beside each node are the earliest arrival time and earliest leaving time. The set of paths P_c-*related-paths*(P_1, T_1) can be partitioned into 3 disjoint path subsets:

$P_c^{(1)}$: The paths with $P^{\text{in}} = (\text{A}, \text{D}, d)$ and without $A^{\text{out}} = (s, \text{A})$.
$P_c^{(2)}$: The paths with $P^{\text{in}} = (\text{D}, d)$ and without $A^{\text{out}} = (\text{A}, \text{D})$.
$P_c^{(3)}$: The paths without $A^{\text{out}} = (\text{D}, d)$.

Take $P_c^{(2)}$ as an example. After removing the arcs in A^{out} and P^{in}, the resulting network is shown in Fig. 4, where the shortest path is (s, B, D) with the arrival time = 13. Since the earliest time we can leave from node D is 13, the total time of the shortest constrained path $(s, \text{B}, \text{D}, d)$ is 15.

Lemma 2. *The time complexity of Shortest Constrained Path Algorithm is* $O(r|V|^2)$, *where* $|V|$ *is the number of nodes in the network and* r *is the number of windows in a node.*

Proof. See Appendix A. □

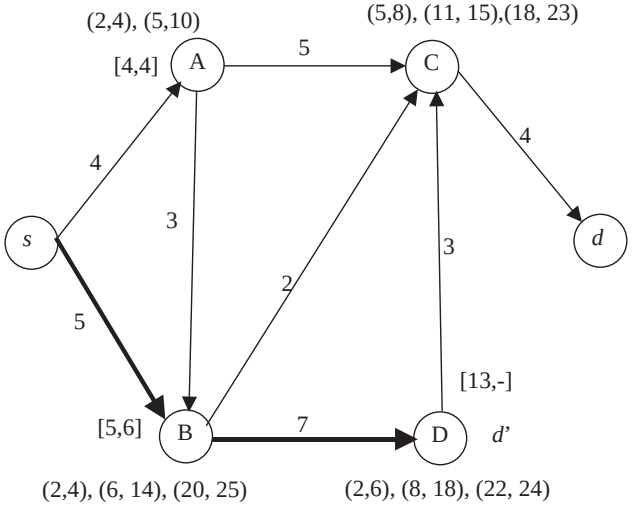

Fig. 4. Subset $P_c^{(2)}$ with $A^{\text{out}} = \{(\text{A}, \text{D})\}$ and $P^{\text{in}} = (\text{D}, d)$.

3.2. How to find the first k shortest paths

Implementing the solution procedure above may be exposed to the problem that a great deal of paths and their related constraints are generated after a number of iterations. To manage this problem, we use the heap structure of Fredman and Tarjan [15], where the times to find and remove the minimum element or to insert a new element are all $O(\log n)$ for a heap with n elements. Let each element in the heap Q represent an enumerated path and each element is associated with its total time, marking status (marked if it has been outputted before; unmarked otherwise), in-subpath and out-arc set. All the paths in Q can be classified into two kinds: (1) paths have been output previously but with smaller total times, and (2) paths have never been output before. While outputting a path of the first kind, since its route must had been partitioned, duplicate path sets with the same in-subpath and out-arc set will be created if it is partitioned again. To ensure that each element in the heap Q has different in-subpath and out-arc sets, we mark those of the first kind to prevent us from partitioning again. Partition is done only to a path without the mark. The following algorithm uses the heap structure to store the set Q for finding the first K shortest paths.

K shortest paths algorithm

1. Use the shortest path algorithm to find P_1 with the total time T_1. Set $\textit{in-subpath}(P_1) = \phi$ and $\textit{out-arc}(P_1) = \phi$, where ϕ represents an empty path or set. Store the element of P_1 into Q.
2. For $w = 1$ to K, execute the cycle from step 3 to step 13.
3. Select the shortest path P with total time T from Q, and remove P from Q.
4. Output the paths in $\textit{related-paths}(P, T)$ as the wth, …, $(w + \textit{related-pno}(P, T) - 1)$th paths.
5. Set $w = w + \textit{related-pno}(P, T) - 1$.
6. Set $T' = \textit{next}(P, T)$ and $P'' = P$.
7. Mark P'' and store the element of P'' with total time T'' into Q.

8. If P has been marked then go to the next cycle.
9. Let P' be the subpath of P satisfying $P = P' \oplus \textit{in-subpath}(P)$, where $\oplus$ is the operator for connecting two subpaths.
10. Let the number of arcs in P' be m, and $P'(i,j)$ denote the subpath from the ith arc to the jth arc of P'.
11. Partition the original path set of P into m disjoint path subsets. The in-subpath and out-arc set of the ith path subset, for $1 \leqslant i \leqslant m$, can be obtained by the out-arc set is $\{P'(i,i)\} \cup \textit{out-arc}(P)$ the in-subpath is $P'(i+1,m) \oplus \textit{in-sub path}(P)$.
12. For each path subset, find the shortest path using the *Shortest Constrained Path Algorithm.*
13. For each path subset, if there is a shortest path, then we store its elements into Q.

Lemma 3. *The time complexity of the K Shortest Paths algorithm is $O(Kr|V|^3)$, where r is the maximum number of windows in a node.*

Proof. See Appendix A. □

Example 1. Consider the network in Fig. 3. In step 1, we find the first path $P_1 = (s, \mathrm{A}, \mathrm{D}, d)$ with $T_1 = 10$. Path P_1 with $\textit{in-subpath}(P_1) = \phi$ and $\textit{out-arc}(P_1) = \phi$ are stored into the heap. In the first cycle of step 2, where $w = 1$, the path removed from the heap Q is P_1. By calling the function $\textit{related-paths}(P_1, 10)$, we find that there are two paths with the same total time but different waiting times, i.e., $(s_0, \mathrm{A}_4, \mathrm{D}_8, d_{10})$ and $(s_0, \mathrm{A}_5, \mathrm{D}_8, d_{10})$. These two paths are output as the first two shortest paths, and w is set from 1 to 2. Once finished, calling the function $\textit{next}(P_1, 10)$ produces the next total time of P_1 being 24. We then store the marked copy of path $(s, \mathrm{A}, \mathrm{D}, d)$ and its total time 24 into the heap Q again. Since path P_1 is unmarked, we need to partition its path subset into disjoint path subsets. Because $\textit{in-subpath}(P_1) = \phi$, we have $P' = (s, \mathrm{A}, \mathrm{D}, d)$ and $m = 3$. So, we partition the original path set of P into 3 path subsets as follows.

The first subset $P_c^{(1)}$ (Fig. 5) has $A^{\mathrm{out}} = \{(s, A)\}$ and $P^{\mathrm{in}} = (\mathrm{A}, \mathrm{D}, d)$.

The second subset $P_c^{(2)}$ (Fig. 4) has $A^{\mathrm{out}} = \{(\mathrm{A}, \mathrm{D})\}$ and $P^{\mathrm{in}} = (\mathrm{D}, d)$.

The third subset $P_c^{(3)}$ (Fig. 6) has $A^{\mathrm{out}} = \{(\mathrm{D}, d)\}$ and $P^{\mathrm{in}} = \phi$.

For each of these three subsets, we summarize their shortest constrained paths as follows.

There is no path in $P_c^{(1)}$.

The shortest constrained path in $P_c^{(2)}$ is $(s, \mathrm{B}, \mathrm{D}, d)$ with time 15.

The shortest constrained path in $P_c^{(3)}$ is $(s, \mathrm{B}, \mathrm{C}, d)$ with time 12.

Hence, path subsets $P_c^{(2)}$ and $P_c^{(3)}$ and related information are stored into the heap Q.

In the second cycle of step 2, where $w = 3$, the path $P = (s, \mathrm{B}, \mathrm{C}, d)$ in $P_c^{(3)}$ is removed from the heap Q and output as the third shortest path. Calling function $\textit{related-paths}(P, 12)$ produces no other paths with the same total time. Further, by calling function $\textit{next}(P, 12)$, the next total time of this path is 15. So, we insert marked path $(s, \mathrm{B}, \mathrm{C}, d)$ with the total time 15 into the heap Q. Since $\textit{in-subpath}(P) = \phi$ and $\textit{out-arc}(P) = (\mathrm{D}, d)$, we have $P' = (s, \mathrm{B}, \mathrm{C}, d)$ and $P_c^{(3)}$ can be further partitioned into three path subsets.

The first subset $P_c^{(3,1)}$ has $A^{\mathrm{out}} = \{(s, \mathrm{B}), (\mathrm{D}, d)\}$ and $P^{\mathrm{in}} = (\mathrm{B}, \mathrm{C}, d)$.

The second subset $P_c^{(3,2)}$ has $A^{\mathrm{out}} = \{(\mathrm{B}, \mathrm{C}), (\mathrm{D}, d)\}$ and $P^{\mathrm{in}} = (\mathrm{C}, d)$.

The third subset $P_c^{(3,3)}$ has $A^{\mathrm{out}} = \{(\mathrm{C}, d), (\mathrm{D}, d)\}$ and $P^{\mathrm{in}} = \phi$.

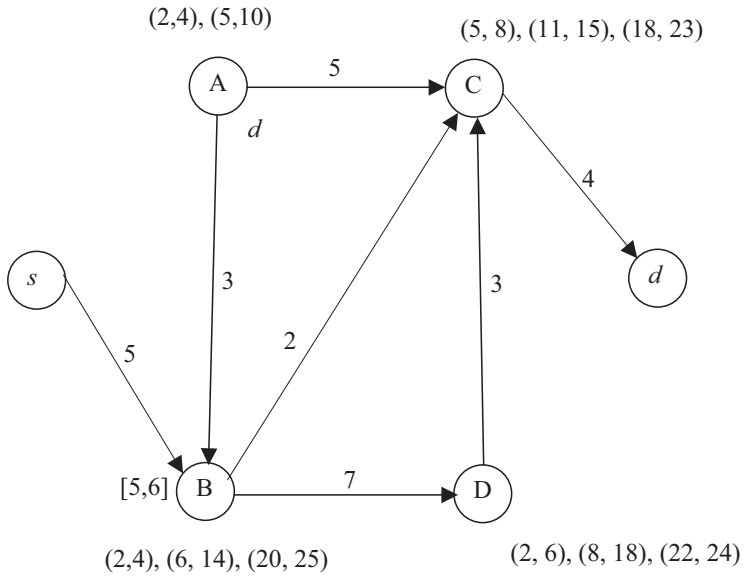

Fig. 5. Subset $P_c^{(1)}$ with $A^{\text{out}} = \{(s, A)\}$ and $P^{\text{in}} = (\text{A}, \text{D}, d)$.

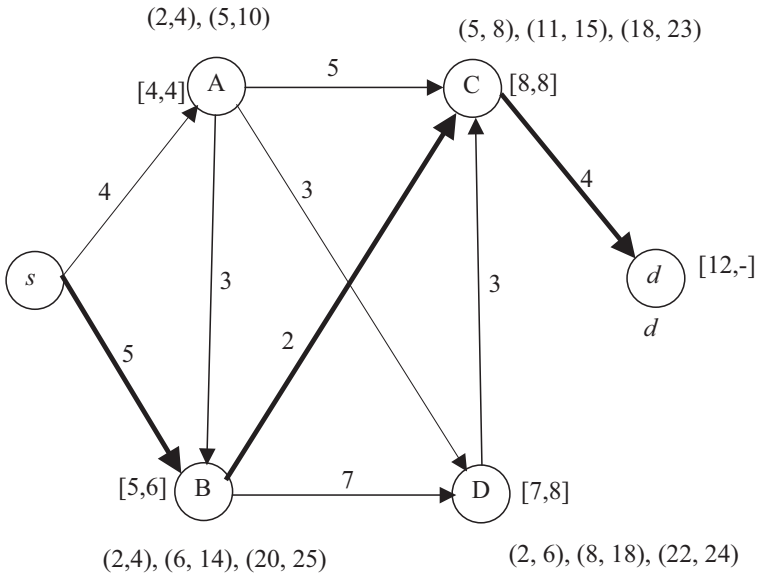

Fig. 6. Subset $P_c^{(3)}$ with $A^{\text{out}} = \{(\text{D}, d)\}$ and $P^{\text{in}} = \phi$.

Similarly, for each of these three subsets, we find their shortest constrained paths as follows.

The shortest constrained path in $P_c^{(3,1)}$ is $(s, \text{A}, \text{B}, \text{C}, \text{d})$ with total time 13.

The shortest constrained path in $P_c^{(3,2)}$ is $(s, \text{A}, \text{C}, d)$ with total time 13.

There is no path in $P_c^{(3,3)}$.

Continuously running step 2 of the *K Shortest Path Algorithm* K times, we will find all K shortest paths.

4. Conclusions

This paper studies finding the first K shortest paths in a time-window network, whose primary difference from the conventional models lies in the inclusion of waiting on a node. Traditionally, treatment of waiting on a node is either ignored or not a serious concern so that we can identify a path route uniquely. In the presence of waiting, however, a path route may correspond to a number of paths with different total times. Therefore, we search for the paths with waiting time consideration in this paper. The time-window network in this paper not only models the real-life situation but also acts as a promising platform for investigating the nature of waiting. The major contribution of the paper is that we have developed an efficient algorithm, with polynomial time complexity, for finding the first K shortest paths. No doubt, the efficiency of the algorithm renders it considerably attractive in terms of implementation for practical problems. Over the course of developing the algorithm, we use a related-graph that plays a vital role in searching the graph in a polynomial time. In perspective of gaining insights into analyzing the waiting of the network and applying the related-graph in broader contexts, we hope this study will be important steps upon which other future work can be based.

Acknowledgement

The research was supported in part by MOE Program for Promoting Academic Excellence of Universities under the Grant Number 91-H-FA07-1-4.

Appendix A

Proof of Lemma 1. The provision of $ts(v^i)$, i.e., invoking *window-beginning*(v^i, T), for each node v^i takes time $O(r)$. Thus the total time for preprocessing all the nodes in the path is $O(mr)$. For a simple path, $m \leqslant |V|$; hence, the time complexity is $O(|V|r)$.

Note that each of the node sets $V_1^1, V_1^2, \ldots, V_1^{m-1}$ contains at most r nodes and V_2^{i+1} has at most $|V^i|$ nodes. Since V^0 has only one node, V^1 has at most $r+1$ nodes and so does V_2^2. In turn, V^2 has at most $2r+1$ nodes and so does V_2^3. Repeatedly, node set V^i has at most $(i \times r)+1$ nodes. Therefore, the total number of nodes in V' is at most $O(m^2 r)$. As for the arc, each node v_p^i has at most two outgoing arcs, i.e., (v_p^i, v_q^{i+1}) and (v_p^i, v_q^i). So, the total number of arcs is also bounded from above by $O(m^2 r)$. Since a simple path satisfies the relation that $m \leqslant |V|$, the total time complexity is thus $O(|V|^2 r)$. □

Proof of Lemma 2. Since every arc and node will be examined and processed at most one time in transforming the network from N into N', the time for step 1 is $O(|A| + |V|)$. Step 2 can be done in time $O(r|V|^2)$ owing to the shortest path algorithm. The time for performing step 3 is negligible. Therefore, the total time complexity is $O(r|V|^2)$. □

Proof of Lemma 3. The algorithm executes the functions *related-paths*(P,T) and *next*(P,T) at most K times. By Lemma 1, they can be done in time $O(Kr|V|^2)$ and are not the dominant part of the algorithm. Instead, the most time-consuming part is the partition of the subset of paths containing the one most recently found into disjoint path subsets. Since the path is simple, there are at most $|V|$ nodes in the path; hence, there are at most $|V|$ disjoint subsets in each partition. For each disjoint subset, we find the shortest path by using the *Shortest Constrained Path Algorithm* that requires $O(r|V|^2)$. Hence, the total time for finding the next path is $O(r|V|^3)$. If K paths are enumerated, the total time becomes $O(Kr|V|^3)$. □

References

[1] Chen YL, Yang HH. Shortest paths in traffic-light networks. Transportation Research B 2000;34:241–53.
[2] Chen YL, Chin YH. The quickest path problem. Computers and Operations Research 1990;17:153–61.
[3] Ahuja RK, Magnanti TL, Orlin JB. Networks flow. Englewood Cliffs, NJ: Prentice-Hall, 1993.
[4] Zwick U. Exact and approximate distances in graphs—a survey. Proceedings of the 9th Annual European Symposium on Algorithms, Aarhus, Denmark, 2001, p. 33–48.
[5] Solomon M, Desrosiers J. Time window constrained routing and scheduling problems: a survey. Transactions of Science 1988;22:1–13.
[6] Balakrishnan N. Simple heuristics for the vehicle routing problem with soft time windows. JORS 1993;44:279–87.
[7] Bramel J, Simchilevi D. Probabilistic analyses and practical algorithms for the vehicle routing problem with time windows. Operations Research 1996;44:501–9.
[8] Kolen WJ, Rinnooy Kan AHG, Trienekens HWJM. Vehicle routing with time windows. Operations Research 1987;35:266–73.
[9] Russell RA. Hybrid heuristics for the vehicle-routing problem with time windows. Transactions of Science 1996;29:156–66.
[10] Yen JY. Finding the k shortest loopless paths in a network. Management Science 1971;17:712–16.
[11] Katoh N, Ibaraki T, Mine H. An efficient algorithm for k shortest simple paths. Networks 1982;12:411–27.
[12] Fox BL. Data structures and computer science techniques in operations research. Operations Research 1978;26: 686–717.
[13] Eppstein D. Finding the k shortest paths. SIAM Journal of Computing 1998;28:652–73.
[14] Dijkstra EW. A note on two problems in connection with graphs. Numerische Mathematik 1959;1:269–71.
[15] Fredman ML, Tarjan RE. Fibonacci heaps and their uses in improved network optimization algorithms. JACM 1987;34:596–615.

Yen-Liang Chen received the B.S. degree in industrial management from National Cheng Kung University, Tainan, Taiwan and the M.S. degree in industrial engineering from National Tsing Hua University, Hsinchu, Taiwan. He received his Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan. He is currently a professor in the Department of Information Management, National Central University, Chung-Li, Taiwan. His current research interests include operations research, data mining and data warehousing.

Hsu-Hao Yang received the B.S. degree in industrial management from National Cheng Kung University, Tainan, Taiwan and M.S. degree as well as Ph.D. degree in industrial engineering from University of Iowa, USA. He is currently a professor in the Department of Industrial Engineering and Management, National Chinyi Institute of Technology, Taiping, Taiwan. His research interests include operations research, logistics and supply chain management.