Finding the K shortest hyperpaths: algorithms and applications

Lars Relund Nielsen* Kim Allan Andersen

Department of Operations Research

University of Aarhus

Ny Munkegade, building 530

DK-8000 Aarhus C

Denmark

Daniele Pretolani

Dipartimento di Matematica e Fisica

Università di Camerino

Via Madonna delle Carceri

I-62032 Camerino (MC) – Italy

(e-mail: daniele.pretolani@unicam.it)

Abstract

The K shortest paths problem has been extensively studied for many years. Efficient methods have been devised, and many practical applications are known. Shortest hyperpath models have been proposed for several problems in different areas, for example in relation with routing in dynamic networks. However, the K shortest hyperpaths problem has not yet been investigated.

In this paper we present procedures for finding the K shortest hyperpaths in a directed hypergraph. This is done by extending existing algorithms for K shortest loopless paths. Computational experiments on the proposed procedures are performed, and applications in transportation, planning and combinatorial optimization are discussed.

Keywords: Network programming, Directed hypergraphs, K shortest hyperpaths, K shortest paths.

*Corresponding author (e-mail: relund@imf.au.dk)

1 Introduction

One classical problem encountered in the analysis of networks is the ranking of paths in nondecreasing order of length, known as K *shortest paths*. As early as 1959 attention was drawn to this problem [11]. Usually, two different situations are distinguished.

In the general (*unrestricted*) problem the paths are allowed to be *looping*, i.e. to contain cycles. Several techniques, based e.g. on dynamic programming or sophisticated data structures, have been applied to this problem, obtaining algorithms that are fast from a practical as well as theoretical point of view; see for example the recent results in [4, 13].

The *restricted* problem where only *loopless* paths are accepted is considered to be harder to solve. In practice, solution methods proposed so far are based on the branching approach by Yen [22], later discussed by Lawler [14] in the more general framework of finding the K best solutions to a discrete optimization problem.

The applications of the K shortest paths problem are numerous. First, practical problems often include constraints which are hard to specify formally or hard to optimize. Here an optimal solution can be found by enumerating suboptimal paths until a path satisfying the hard constraints is found. Second, by computing more than one shortest path, one can to a certain extent determine how sensitive the optimal solution is to variations of the parameters in the model. Last but not least, the K shortest path problem often appears as a subproblem within algorithms for bicriteria shortest path problems, see for example [10, 3]. A complete survey of the existing literature on K shortest paths does not fall into the scope of this paper; the interested reader is referred to the work of Eppstein [4].

Directed hypergraphs are an extension of directed graphs, and have often been used in several areas as a modelling and algorithmic tool. A technical as well as historical introduction to directed hypergraphs has been given by Gallo *et al.* [6]. Hyperpaths in hypergraphs are a nontrivial extension of directed paths whose expressive power allows us to deal with more complex situations. In fact, several applications of shortest hyperpath methods are known, see among others [1, 5, 8, 18, 20]. In particular, a shortest hyperpath model has been proposed for routing problems in discrete dynamic networks, that have recently attracted a growing attention [16, 17, 20].

Similar to what is discussed above for K shortest paths, applications and solution methods based on shortest hyperpaths would take advantage of the availability of alternate optimal or sub-optimal solutions. However, to the authors' knowledge, no one considered the problem of finding the K shortest hyperpaths.

In this paper we propose and test some algorithms for the K shortest hyperpaths problem. Since hyperpaths in our context are acyclic, we extend to directed hypergraphs Yen's method for loopless paths; as we shall see, this extension is not straightforward. Moreover, we discuss in detail some relevant applications of K shortest hyperpaths algorithms.

The paper is organized as follows. Directed hypergraphs are introduced in Section 2. In Section 3 different procedures to find the K shortest hyperpaths are developed. Computational results are reported in Section 4, while applications are considered in Section 5. Finally, we summarize original contributions and topics for further research in Section 6.

2 Directed Hypergraphs

A *directed hypergraph* is a pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = (v_1, ..., v_n)$ is the set of nodes, and $\mathcal{E} = (e_1, ..., e_m)$ is the set of *hyperarcs*. A hyperarc $e \in \mathcal{E}$ is a pair $e = (T(e), h(e))$, where $T(e) \subset \mathcal{V}$ denotes the *tail* nodes and $h(e) \in \mathcal{V} \setminus T(e)$ denotes the *head* node.
In this paper we only consider hypergraphs where each hyperarc has one node in its head. These hypergraphs are referred to as *B-graphs* in [6], where more general classes of hypergraphs are introduced. Note that B-graphs are in some sense equivalent to *F-graphs*, where hyperarcs have one node in the tail and possibly more than one node in the head; in particular, the properties of F-graphs stated in [6] translate into analogous B-graphs properties.
The *cardinality* of a hyperarc e is the number of nodes it contains, i.e. $|e| = |T(e)| + 1$. If $|e| = 2$, hyperarc e is an *arc*. The *size* of $\mathcal{H}$ is the sum of the cardinalities of its hyperarcs:

$$size(\mathcal{H}) = \sum_{e \in \mathcal{E}} |e| \,.$$

We denote by

$$FS(u) = \{e \in \mathcal{E} \mid u \in T(e)\}$$
$$BS(u) = \{e \in \mathcal{E} \mid u \in h(e)\}$$

the *forward star* and the *backward star* of node u, respectively. A hypergraph $\tilde{\mathcal{H}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ is a *subhypergraph* of $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, if $\tilde{\mathcal{V}} \subseteq \mathcal{V}$ and $\tilde{\mathcal{E}} \subseteq \mathcal{E}$. This is written $\tilde{\mathcal{H}} \subseteq \mathcal{H}$ or we say that $\tilde{\mathcal{H}}$ is *contained* in $\mathcal{H}$.
A *path* P_{st} in a hypergraph $\mathcal{H}$ is a sequence of nodes and hyperarcs in $\mathcal{H}$:

$$P_{st} = (s = v_1, e_1, v_2, e_2, ..., e_q, v_{q+1} = t)$$

where, for $i = 1, \ldots, q$, $v_i \in T(e_i)$ and $v_{i+1} = h(e_i)$. A node v is connected to node u if a path P_{uv} exists in $\mathcal{H}$. A *cycle* is a path P_{st}, where $t \in T(e_1)$. This is in particular true if $t = s$. If $\mathcal{H}$ contains no cycles, it is *acyclic*.

Definition 1 Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a hypergraph. A *valid ordering* in $\mathcal{H}$ is a topological ordering of the nodes

$$V = \{u_1, u_2, \ldots, u_n\}$$

such that, for any $e \in \mathcal{E} : (u_j \in T(e)) \wedge (h(e) = u_i) \Rightarrow j < i$.

Notice that, in a valid ordering any node $u_j \in T(e)$ precedes node $h(e)$. The next theorem is a generalization of a similar result for acyclic directed graphs, see [21].

Theorem 1 $\mathcal{H}$ acyclic $\iff$ A valid ordering of the nodes in $\mathcal{H}$ is possible.

Theorem 1 is proven (for F-graphs) in [6], where an $O(size(\mathcal{H}))$ algorithm finding a valid ordering of the nodes in an acyclic hypergraph is presented. It should be noticed that a valid ordering in general is not unique, which is also the case for acyclic directed graphs.

2.1 Hyperpaths and hypertrees

Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. A *hyperpath* π_{st} of *origin* s and *destination* t, is an *acyclic minimal hypergraph* (with respect to deletion of nodes and hyperarcs) $\mathcal{H}_\pi = (\mathcal{V}_\pi, \mathcal{E}_\pi)$ satisfying the following conditions:

1. $\mathcal{E}_\pi \subseteq \mathcal{E}$
2. $s, t \in \mathcal{V}_\pi = \bigcup_{e \in \mathcal{E}_\pi} \big(T(e) \cup \{h(e)\}\big)$
3. $u \in \mathcal{V}_\pi \setminus \{s\} \Rightarrow u$ is connected to s in $\mathcal{H}_\pi$.

Note that condition 3 implies that, for each $u \in \mathcal{V}_\pi \setminus \{s\}$, there exists a hyperarc $e \in \mathcal{E}_\pi$, such that $h(e) = u$. It follows from minimality that e is unique; hyperarc e is the *predecessor* of u in π_{st}. Conversely, condition 3 can be replaced by condition 4 below, where $\mathcal{N} = \mathcal{V}_\pi \setminus \{s\}$. Minimality also implies that, for any node $u \in \mathcal{V}_\pi \setminus \{t\}$, there is a u-t path in π_{st}.
We say that node t is *hyperconnected* to s in $\mathcal{H}$ if there exists in $\mathcal{H}$ a hyperpath π_{st}.
Let $s \in \mathcal{V}$ be a given *root* node and let $\mathcal{N} \subseteq \mathcal{V} \setminus \{s\}$ be a set of nodes hyperconnected to s. Then a *directed hypertree* is the union of hyperpaths from s to all nodes in $\mathcal{N}$.

Definition 2 A *directed hypertree* with *root node* s is an *acyclic* hypergraph $\mathcal{T}_s = (\{s\} \cup \mathcal{N}, \mathcal{E}_\mathcal{T})$ with $s \notin \mathcal{N}$ such that:

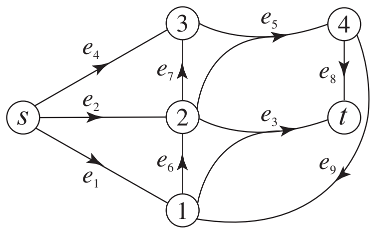
4. $BS\,(s) = \emptyset; \quad |BS(v)| = 1 \quad \forall v \in \mathcal{N}$.

A hypertree $\mathcal{T}_s = (\{s\} \cup \mathcal{N}, \mathcal{E}_\mathcal{T})$ in a hypergraph $\mathcal{H}$ is defined by a *predecessor function* $p : \mathcal{V} \to \mathcal{E}$; for each $u \in \mathcal{N}$, $p(u)$ is the unique hyperarc in $\mathcal{T}_s$ which has node u as the head. A *sub-hypertree* (or simply a *subtree*) of a hypertree $\mathcal{T}_s$ is a hypertree contained in $\mathcal{T}_s$. Note that any hyperpath is a hypertree, in particular it can be defined by a predecessor function. Moreover, different hypertrees can share the same hyperpath π_{st} as a subtree.
Procedure *B-Visit*, given in [6], finds a hypertree rooted at a given node s containing each node u hyperconnected to s in a hypergraph $\mathcal{H}$. Note that B-visit finds only one of potentially many hypertrees in $\mathcal{H}$. The overall complexity of B-Visit is $O(size(\mathcal{H}))$.
Below we list some elementary properties of hypertrees that mirror similar properties for trees in directed graphs. Let $V = \{s = u_1, u_2, \ldots, u_q\}$ be a valid ordering for the hypertree $\mathcal{T}_s = (\{s\} \cup \mathcal{N}, \mathcal{E}_\mathcal{T})$, defined by the predecessor function p. For $1 \le i \le q$, let us define the hypergraph

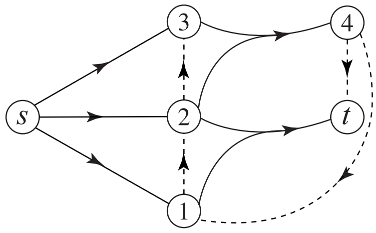
$$\mathcal{T}_s^i = \big(\{u_1, \ldots, u_i\}, \{p(u_2), \ldots, p(u_i)\}\big).$$

It is easy to see that $\mathcal{T}_s^i$ is a hypertree, thus a subtree of $\mathcal{T}_s$. Now suppose we change the predecessor $p(u) = e$ for a given $u \in \mathcal{N}$, setting $p(u) = \bar{e}$, where $\bar{e} \in \mathcal{E} \setminus \{e\}$. The following holds true for the resulting function p:

Property 1 p defines a hypertree if and only if the nodes in $T(\bar{e})$ belong to $\mathcal{T}_s$ and precede u in a valid ordering of $\mathcal{T}_s$.



(a) A hypergraph $\mathcal{H}$ with a cycle.



(b) A hypertree in $\mathcal{H}$.

Figure 1: The running example hypergraph.

Note that if $\mathcal{H}$ is acyclic, then p defines a hypertree if and only if the nodes in $T(\bar{e})$ belong to $\mathcal{T}_s$.

Example 1 A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is shown in Figure 1(a). Below we give two hyperpaths in $\mathcal{H}$, namely a hyperpath from s to t and a hyperpath from s to 4.

$$\pi_{st} = \big(\{s, 1, 2, t\}, \{e_1, e_2, e_3\}\big) \quad \pi_{s4} = \big(\{s, 2, 3, 4\}, \{e_2, e_4, e_5\}\big).$$

The hypergraph $\mathcal{H}$ becomes acyclic when hyperarc e_9 is deleted and has a unique valid ordering, namely $V = (\{s, 1, 2, 3, 4, t\}$.
A hypertree $\mathcal{T}_s$ in $\mathcal{H}$ is shown with solid lines in Figure 1(b). It is the union of the two hyperpaths given above. Several valid orderings for $\mathcal{T}_s$ exist; one of them is $V = (\{s, 1, 2, t, 3, 4\}$. According to V, the subtree $\mathcal{T}_s^4$ of $\mathcal{T}_s$ corresponds to hyperpath π_{st} above.
■

Next we point out some relevant differences between paths in directed graphs and hyperpaths in hypergraphs. Assume that a path P_{st} from node s to node t in a directed graph $G = (N, A)$ is known:

$$P_{st} = (s = u_1, a_1, u_2, a_2, \ldots, a_q, u_{q+1} = t).$$

Clearly, P_{st} is the concatenation of a subpath P_{si} from node s to node u_i and a subpath P_{it} from node u_i to node t, where

$$\begin{aligned} P_{si} &= (s = u_1, a_1, u_2, a_2, \ldots, a_{i-1}, u_i) \\ P_{it} &= (u_i, a_i, \ldots, a_q, u_{q+1} = t). \end{aligned}$$

In other words, for each $1 \leq i \leq q+1$ we can *split* path P_{st} into the subpaths P_{si} and P_{it}. Unfortunately, this need not be so for hyperpaths. In general a hyperpath π_{st} is not the concatenation of two hyperpaths π_{su} and π_{ut}; consider e.g. π_{st} in Example 1. However, we can define a "splitting" operation on hyperpaths as follows. Let $\mathcal{H}_\pi = (\mathcal{V}_\pi, \mathcal{E}_\pi)$ denote the hyperpath π_{st}, and let $V = \{u_1 = s, u_2, \ldots, u_{q+1} = t\}$ be a valid ordering. Recall that π_{st} is a hypertree $\mathcal{T}_s$ defined by a predecessor function p. For each $1 \leq i \leq q$, denote by $\pi_{st} = (\tau^i, \eta^i)$ the *splitting around* u_i of π_{st}, where:

- $\tau^i = (\mathcal{V}_\tau, \mathcal{E}_\tau)$, with $\mathcal{V}_\tau = \{u_1, \ldots, u_i\}$ and $\mathcal{E}_\tau = \{p(u_2), \ldots, p(u_i)\}$;
- $\eta^i = (\mathcal{V}_\eta, \mathcal{E}_\eta)$, with $\mathcal{E}_\eta = \{p(u_{i+1}), \ldots, p(t)\}$ and $\mathcal{V}_\eta = \bigcup_{e \in \mathcal{E}_\eta} T(e) \cup \{h(e)\}$;
- $\mathcal{H}_\pi = (\mathcal{V}_\tau \cup \mathcal{V}_\eta, \mathcal{E}_\tau \cup \mathcal{E}_\eta)$

Clearly, τ^i is the subtree $\mathcal{T}_s^i$ of $\mathcal{T}_s = \pi_{st}$. On the contrary, η^i is not a hypertree in general; we call η^i an *end-tree*. Observe that for any node $u \neq t$ in η^i there is a u-t path in η^i. The above splitting operation will be exploited in our algorithms for K-shortest hyperpaths.

2.2 Weighted hypergraphs and shortest hyperpaths

A *weighted hypergraph* is a hypergraph where each hyperarc e is assigned a real weight $w(e)$. In this paper we shall assume that all weights are non-negative. Given a hyperpath π_{st}, a weighting function W_π is a node function assigning weights $W_\pi(u)$ to all nodes in π_{st}. The weight of hyperpath π_{st} is $W_\pi(t)$. We shall restrict ourselves to *additive weighting functions*, defined by the recursive equations:

$$W_\pi(u) = \begin{cases} w(p(u)) + F(p(u)) & u \in \mathcal{V}_\pi \setminus \{s\} \\ 0 & u = s \end{cases} \tag{1}$$

where $F(e)$ is a nondecreasing function of the weights of the nodes in $T(e)$. We shall consider two particular weighting functions, namely the *distance* and the *value*, both of which have been studied in detail (see e.g. [6, 12]).
The *distance* function is obtained by defining $F(e)$ as follows:

$$F(e) = \max_{v \in T(e)} \{W_\pi(v)\}$$

and the *value* function is obtained as follows:

$$F(e) = \sum_{v \in T(e)} a_e(v)\, W_\pi(v)$$

where $a_e(v)$ is a nonnegative multiplier defined for each hyperarc e and node $v \in T(e)$. With respect to the value function there are two interesting cases which may arise:

- If $a_e(v) = 1,\ \forall\ e \in \mathcal{E},\ \forall\ v \in T(e)$, then the weighting function is called the *sum* function.
- If $\sum_{v \in T(e)} a_e(v) = 1,\ \forall\ e \in \mathcal{E}$, then the weighting function is called the *mean* function.

The distance (the value) of a hyperpath π_{st} is the weight of π_{st} with respect to the distance (the value) weighting function. Trivially, for each hyperpath the distance is a lower bound on the sum.

The *shortest hyperpath problem* consists in finding the minimum weight hyperpaths (with respect to a particular weighting function) from an origin s to all nodes in $\mathcal{H}$ hyperconnected to s. The result is a *shortest hypertree* $\mathcal{T}_s$ containing minimum weight hyperpaths to all hyperconnected nodes.

Example 1 (continued) Consider again the hypergraph in Figure 1(a), and suppose all edge weights are equal to 1. In this case, Figure 1(b) shows the shortest hypertree with respect to the sum as well as to the distance weighting functions. ∎

Finding the shortest hypertree has been shown in [6] to be equivalent to finding a solution to *Bellman's generalized equations*

$$W(v) = \begin{cases} 0 & v = s \\ \min\limits_{e \in BS(v)} \{w(e) + F(e)\} & v \in \mathcal{V} \setminus \{s\} \end{cases}$$

provided that the weighting function is additive, the weights are nonnegative and that all cycles are *nondecreasing.* Sufficient conditions ensuring that each cycle

$$C = \{v_1, e_1, v_2, e_2, \ldots, v_r, e_r, v_1\}$$

is nondecreasing have been given in [6]:

1. In the case of a distance function, the weights must satisfy:

$$\sum_{i=1}^{r} w(e_i) \geq 0$$

which is the normal nonnegativity condition from standard digraphs.

2. In the case of a value function, the multipliers must satisfy:

$$\prod_{i=1}^{r} a_{e_i}(v_i) \geq 1$$

which is the gainfree condition from Jeroslow *et al.* [12].

In our case, the condition given in 1 is obviously true, because all weights are non-negative. The condition given in 2 is true if all multipliers are at least 1 (the sum case) and also if the hypergraph is acyclic. In the following we shall consider the *sum* and *distance* functions, as well as the *mean* function restricted to acyclic hypergraphs.
We next give an example showing that if the condition given in (2) is not satisfied, then a solution to Bellman's generalized equations may not correspond to a shortest hypertree (with respect to the value function).

Initialization: Set $W(u) = \infty\ \forall u \in \mathcal{V}$, $k_j = 0\ \forall e_j \in \mathcal{E}$, $Q = \{s\}$ and $W(s) = 0$

1 **while** $(Q \neq \emptyset)$ **do**
2   select and remove $u \in Q$ such that $W(u) = \min\{W(x) | x \in Q\}$
3   **for** $(e_j \in FS(u))$ **do**
4       $k_j := k_j + 1$
5   **if** $(k_j = |T(e_j)|)$ **then**
6          $v := h(e_j)$
7   **if** $(W(v) > w(e_j) + F(e_j))$ **then**
8   **if** $(v \notin Q)$ **then** $Q := Q \cup \{v\}$
9   **end if**
10             $W(v) := w(e_j) + F(e_j)$, $p(v) := e_j$
11   **end if**
12   **end if**
13   **end for**
14 **end while**

Procedure 1: Shortest hypertree (SBT-Dijkstra)

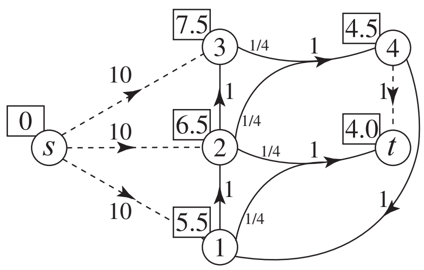


Figure 2: A solution to Bellman's generalized equations

Example 2 In Figure 2 the weights and the multipliers are shown next to each hyperarc. The value W of each node is shown next to it in a small box. The solution, shown with solid lines, satisfies Bellman's generalized equations. Evidently, the solution does not correspond to a (shortest) hypertree. ■

A general procedure (called *SBT*) for finding a shortest hypertree was proposed in [6]. Procedure 1 is a particular version of *SBT*, called *SBT-Dijkstra* in [6]. This procedure is a simple generalization of Dijkstra's algorithm for shortest paths. Indeed, when a node u is removed from the candidate set Q, $W(u)$ is the minimum weight of all hyperpaths from s to u. The condition in line 5 ensures that each hyperarc e_j is processed only once after the minimum weights for its tail nodes have been determined. If the priority queue Q is implemented as a d-heap, the complexity becomes $O(m \log n + \mathit{size}(\mathcal{H}))$.

It must be remarked that the order in which nodes are removed from Q in Procedure 1 gives a valid ordering for the nodes in the shortest hypertree. We assume that this valid ordering is used in the K shortest hyperpath algorithms described in the next section.

For the particular case where the hypergraph is acyclic, a simpler and faster procedure

Initialization: Set $W(s) := 0$, $W(v_i) = \infty$ $i = 1, ..., n$

for ($i = 1$ **to** n) **do**
  for ($e \in BS(v_i)$) **do**
    if ($W(v_i) > w(e) + F(e)$) **then** $W(v_i) := w(e) + F(e)$, $p(v_i) := e$
  end for
end for

Procedure 2: Shortest hypertree in acyclic hypergraphs (SBT-acyclic)

exists [7]. In Procedure 2, nodes are processed according to a valid ordering

$$V = (v_1 = s, v_2, ..., v_n).$$

Clearly, when a node v_i is processed, the shortest hyperpaths to all the nodes preceding v_i in V are known, thus $F(e)$ can be computed for all the hyperarcs in $BS(v_i)$. In this case, no priority queues are needed and the overall complexity is $O(size(\mathcal{H}))$.
The K *shortest hyperpaths* problem addressed in this paper is as follows: given a hypergraph $\mathcal{H}$, an *origin* node s and a *destination* node t, generate the K shortest s-t hyperpaths in $\mathcal{H}$ in nondecreasing order of weight; hyperpaths with the same weight can be generated in arbitrary order. Obviously, a problem is characterized by the chosen weighting function. Here we do not consider the problem of storing and retrieving the generated hyperpaths, which are generically "sent to output". Efficient data structures for representing the K shortest paths in graphs have often been analyzed, see e.g. [4, 13].

3 Finding the K shortest hyperpaths

In this section we describe algorithms for finding the K shortest hyperpaths in a hypergraph. Our algorithms extend the K shortest loopless paths procedure by Yen [22]; therefore we briefly recall this procedure first. The extension to the hyperpath case is then discussed in detail and some improvements are proposed.
In general terms, Yen's algorithm is an implicit enumeration method, where the set of solutions is partitioned into smaller sets by recursively applying a *branching* step. Given a graph $G = (N, A)$, and two nodes $s, t \in N$, denote by $\mathcal{P}$ the set of paths from s to t in G. Assume that a shortest s-t path P_{st} is known, where

$$P_{st} = (s = u_1, a_1, u_2, a_2, \ldots, a_q, u_{q+1} = t).$$

In the branching step, the set $\mathcal{P} \setminus \{P_{st}\}$ is partitioned into q subsets $\mathcal{P}^i$, $1 \leq i \leq q$. Each set $\mathcal{P}^i$ contains the *deviations from* P_{st} *at* i, that is: each s-t path in $\mathcal{P}^i$ is the concatenation of P_{si} (the subpath of P_{st} from s to u_i) and a path from u_i to t not containing arc a_i. Note that some of the sets $\mathcal{P}^i$ may be empty.
The shortest s-t path in each subset $\mathcal{P}^i$ can be found by a standard SPT procedure. Indeed, it suffices to find a shortest path from u_i to t in a subgraph G^i, obtained from G by deleting each node u_j in P_{si} except u_i, and deleting arc a_i as well. In practice, each set $\mathcal{P}^i$ can

be represented by a pair (P_{si}, G^i). Yen's algorithm maintains a list of such pairs (P', G'), where P' is a path from s to a node $u \neq t$, and a shortest s-t path P'_{st} is defined by the concatenation of P' and the shortest u-t path in G'. Initially, the list contains a single pair with $P' = (s)$ and $G' = G$; here, (s) is a path containing the single node s. At step k, the shortest P'_{st} in the list is ranked as the k^{th} shortest path. A branching step is then applied on P'_{st}, obtaining a set of pairs that replace (P', G') in the list. The algorithm terminates when the K shortest paths are found, or when the list is empty.

Note that, at the beginning of step $k+1$, the list of pairs represents a partition of $\mathcal{P} \setminus \{P^1, \ldots, P^k\}$, where $\{P^1, \ldots, P^k\}$ are the previously found k shortest paths. When a pair (P', G') is inserted in the list, the optimal path P'_{st} must be computed. Since $O(n)$ pairs can be generated at each branching step, Yen's algorithms must solve $O(Kn)$ SPT problems.

Finally, we remark that the commonly adopted "forward branching" described above can be replaced by a "backward branching", where path P_{st} is processed from t to s. In the latter case, each path in the set $\mathcal{P}^i$ would be the concatenation of a path from s to u_{i+1}, not containing arc a_i, and the subpath of P_{st} from u_{i+1} to t. From a theoretical as well as practical point of view, the two approaches are equivalent; as we shall see, this symmetry does no longer hold when hypergraphs are considered.

3.1 Branching on Hyperpaths

In order to extend Yen's algorithm to hypergraphs, we need to devise a suitable branching rule, i.e. a partition technique so that finding the best hyperpath in each subset is easy, in particular solved by procedure SBT.

Consider a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and two nodes $s, t \in \mathcal{V}$. In the following we let Π denote the set of hyperpaths from s to t in $\mathcal{H}$. We assume that a shortest s-t hyperpath $\pi_{st} = (\mathcal{V}_\pi, \mathcal{E}_\pi)$ is known, and is defined by a predecessor function $p: \; \mathcal{V}_\pi \setminus \{s\} \rightarrow \mathcal{E}$; moreover, a valid ordering:

$$V = (s = u_1, u_2, \ldots, u_q, u_{q+1} = t)$$

for the nodes in π_{st} is at hand.

Clearly, a direct application of Yen's branching technique is not possible. However, we may still follow a forward branching approach, where we split a hyperpath around each node u_j, according to the given valid ordering. More formally, we may partition the set $\Pi \setminus \{\pi_{st}\}$ into q subsets Π^i, $1 \leq i \leq q$; each s-t hyperpath in Π^i is the concatenation of τ^i (the subtree of π_{st} spanning the first i nodes) and an end-tree η not containing hyperarc $p(u_{i+1})$.

We stress the fact that the shortest hyperpath π^i in Π^i must contain the whole subtree τ^i; in order to find π^i, we must find the "best" end-tree η for τ^i. This is not the same as finding a shortest hypertree $\mathcal{T}$ containing τ^i as a subtree: indeed, the s-t hyperpath in $\mathcal{T}$ does not necessarily contain τ^i. In fact, finding the best η is a *constrained hyperpath* problem, which turns out to be difficult.

Assume that we are given a weighted hypergraph $\mathcal{H}$, a hypertree $\mathcal{T}$ rooted at s in $\mathcal{H}$, and a node t of $\mathcal{H}$ not in $\mathcal{T}$. The *Subtree Constrained Hyperpath problem* (SCH) consists in

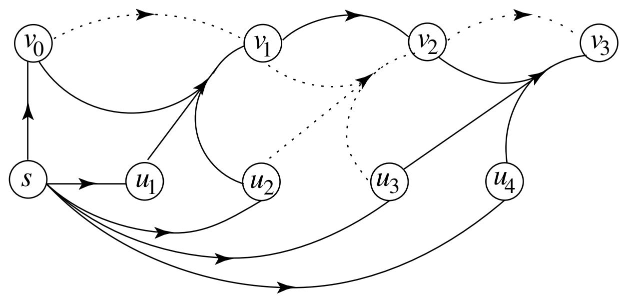


Figure 3: Hypergraph $\mathcal{H}^C$

finding a shortest s-t hyperpath containing $\mathcal{T}$ as a subtree. We show that this problem is NP-hard, also if $\mathcal{T}$ only contains arcs in $FS(s)$. We consider the distance function here, a simpler construction can be given for the value. We provide a reduction from the *Set Covering* problem (SC), which is well-known to be strongly NP-hard. An instance of (SC) is defined by a family $\mathcal{F} = \{F_1, F_2, \ldots, F_n\}$ of subsets of $\{1, 2, \ldots, m\}$, where each F_i has a cost c_i. The problem is to find a subset $\mathcal{C} \subseteq \{1, 2, \ldots, n\}$ with minimum cost $c(\mathcal{C}) = \sum_{j \in \mathcal{C}} c_j$ and such that

$$\{1, 2, \ldots, m\} = \bigcup_{j \in \mathcal{C}} F_j.$$

Theorem 2 Problem (SCH) for the distance function is NP-hard in the strong sense.

Proof Given an instance of (SC), define an instance of (SCH) as follows. Let $\mathcal{H}^C = (\mathcal{V}^C, \mathcal{E}^C)$ be a weighted hypergraph where

- $\mathcal{V}^C = \{s\} \cup \{u_i : 1 \leq i \leq m\} \cup \{v_j : 0 \leq j \leq n\}$;
- $\mathcal{E}^C = FS(s) \cup \{e^a_j : 1 \leq j \leq n\} \cup \{e^h_j : 1 \leq j \leq n\}$;

here $FS(s)$ contains an arc from s to each node u_i and an arc $\big(\{s\}, \{v_0\}\big)$; moreover, for each $1 \leq j \leq n$:

- $e^a_j = \big(\{v_{j-1}\}, v_j\big)$;
- $e^h_j = \big(\{u_i : i \in F_j\} \cup \{v_{j-1}\}, v_j\big)$.

The cost of each hyperarc e^h_j is c_j, arcs have zero costs. Finally, let the tree $\mathcal{T}$ contain the arcs in $FS(s)$, and choose the destination node $t = v_n$.
Figure 3 shows the hypergraph $\mathcal{H}^C$ for an (SC) instance where $m = 4$, $n = 3$, $F_1 = \{1, 2\}$, $F_2 = \{2, 3\}$ and $F_3 = \{3, 4\}$. The optimal hyperpath is represented by solid lines.
Observe that, in any feasible s-t hyperpath $\pi = (\mathcal{V}_\pi, \mathcal{E}_\pi)$ it is $\mathcal{V}_\pi = \mathcal{V}^C$, and for each $j > 0$, the predecessor $p(v_j)$ is either e^h_j or e^a_j. Therefore π is univocally defined by the set

$$\mathcal{C} = \{j : e^h_j \in \mathcal{E}_\pi\},$$

and the distance of each node v_j, $j > 0$, is given by the distance of v_{j-1} plus the cost of $p(v_j)$; the distance of π is thus $c(\mathcal{C}) = \sum_{j \in \mathcal{C}} c_j$. Moreover, in a feasible π, each node u_i must belong to the tail of some hyperarc e^h_j with $j \in \mathcal{C}$. Therefore, hyperpath π is feasible for (SCH) if and only if the corresponding $\mathcal{C}$ is a feasible solution for (SC). Since the cost of π is $c(\mathcal{C})$, we conclude that (SC) reduces to solving the above instance of (SCH), and the thesis follows. ■

Theorem 2 has an immediate negative impact on forward branching:

Corollary 1 Finding the shortest hyperpath π^i in Π^i is an NP-hard problem.

We shall therefore follow a different approach, in particular one based on *backward branching*. Consider again the shortest hyperpath π_{st} and the valid ordering V. We partition the set $\Pi \setminus \{\pi_{st}\}$ into q subsets Π^i, $1 \leq i \leq q$ as follows:

- s-t hyperpaths in Π^q do not contain hyperarc $p(u_{q+1})$, that is $p(t)$;
- for $1 \leq i < q$, s-t hyperpaths in Π^i contain the end-tree η^{i+1}, and do not contain hyperarc $p(u_{i+1})$.

Note that a hyperpath in Π^i, $i < q$, must contain hyperarcs $p(u_j)$, $i + 1 < j \leq q + 1$, however, it is not required to be the concatenation of η^{i+1} with a hypertree rooted at s (see Example 3). In this case, finding a shortest hyperpath $\pi^i \in \Pi^i$ reduces to solving a shortest hypertree problem on a hypergraph $\mathcal{H}^i$, obtained from $\mathcal{H}$ as follows:

- for each node u_j, $i + 1 < j \leq q + 1$, remove each hyperarc in $BS(u_j)$ except $p(u_j)$;
- remove hyperarc $p(u_{i+1})$ from $BS(u_{i+1})$.

We say that in $\mathcal{H}^i$ each hyperarc $p(u_j)$, $i + 1 < j \leq q + 1$, is *fixed*, while $p(u_{i+1})$ is *deleted*. The following property can be easily proved:

Property 2 Each hyperpath in Π^i is also an s-t hyperpath in $\mathcal{H}^i$; conversely, each s-t hyperpath in $\mathcal{H}^i$ belongs to Π^i.

As a consequence, each set Π^i is represented by the corresponding hypergraph $\mathcal{H}^i$. A (backward) *branching operation on* π_{st} returns the set of hypergraphs $\mathcal{B}(\mathcal{H}) = \{\mathcal{H}^i : 1 \leq i \leq q\}$, representing the partition $\{\Pi^i : 1 \leq i \leq q\}$ of $\Pi \setminus \{\pi_{st}\}$. Recall that a branching operation is related to an underlying valid ordering V; a different order may result in a different set of hypergraphs.
Adopting the backward branching technique, we can extend Yen's algorithm to hypergraphs. The algorithm maintains a list L of *subproblems*. Each subproblem r is represented by the pair $(\mathcal{H}_r, \pi_r)$, where π_r is a shortest s-t hyperpath in $\mathcal{H}_r$. Note that we assume t hyperconnected to s in each subproblem. Initially, L contains the pair $(\mathcal{H}, \pi_{st})$. In iteration k we remove from L a pair $(\mathcal{H}_r, \pi_r)$, such that π_r has minimum weight among the subproblems in L; π_r is the k^{th} shortest hyperpath. Then backward branching is applied to π_r,

and for each sub-hypergraph $\mathcal{H}_r^i \in \mathcal{B}(\mathcal{H}_r)$, the shortest hypertree rooted at s is computed; if t is hyperconnected to s in $\mathcal{H}_r^i$, the pair $(\mathcal{H}_r^i, \pi_r^i)$ is inserted into L, where π_r^i denotes the shortest s-t hyperpath in $\mathcal{H}_r^i$. Otherwise, $\mathcal{H}_r^i$ is discarded. The algorithm terminates when L becomes empty or at the end of iteration K.
Procedure *Yen*, given below, formally describes our K-shortest hyperpath algorithm. Here $W(\pi)$ denotes the weight of hyperpath π, and we assume $W(\pi_r^i) = +\infty$, if t is not hyperconnected to s in $\mathcal{H}_r^i$.

Procedure *Yen*$(\mathcal{H}, s, t, K)$

Step 0 $L = \{(\mathcal{H}, \pi_{st})\}$; $k = 1$;

Step 1 if $L = \emptyset$ then STOP; otherwise, let $L = L \setminus \{r\}$ where

$$\pi_r = \arg\min_{l \in L} W(\pi_l);$$

Step 2 OUTPUT π_r; $k = k + 1$; if $k > K$ then STOP;

Step 3 for each $\mathcal{H}_r^i$ in $\mathcal{B}(\mathcal{H}_r)$ do:

(a) apply procedure $SBT(s, \mathcal{H}_r^i)$;

(b) if $W(\pi_r^i) < +\infty$ then $L = L \cup \{(\mathcal{H}_r^i, \pi_r^i)\}$;

go to Step 1.

As happens for graphs, at most n subproblems are generated in each branching step and Procedure *Yen* must solve $O(Kn)$ shortest hyperpath problems in the worst case. In practice, however, it is not always necessary to process all the hypergraphs in $\mathcal{B}(\mathcal{H}_r)$. Consider the case where, in order to obtain $\mathcal{H}_r^i$, hyperarc $p(u)$ must be deleted and $BS(u) = \{p(u)\}$. This happens, in particular, if $p(u)$ has been fixed in a previous branching step. Due to the structure of backward branching, either $u = t$ or u belongs to the tail of a fixed hyperarc in $\mathcal{H}_r^i$. It follows that neither u nor t is hyperconnected to s in $\mathcal{H}_r^i$. In this case, we may assume that $\mathcal{H}_r^i$ is not generated by the algorithm.

Example 3 Assume that we want to find the $K = 3$ shortest hyperpaths, with respect to the sum function, in the hypergraph of Figure 1(a), where we assume unit weights on the hyperarcs. The minimal hypertree has been emphasized in Figure 1(b). The shortest hyperpath from s to t, with weight 3, is the one given in Example 1, that is $\pi_{st} = (\{t, 1, 2, s\}, \{e_1, e_2, e_3\})$.
A valid ordering of the nodes in π_{st} is $V = \{s, 1, 2, t\}$. Applying backward branching, three subhypergraphs $\mathcal{H}^3$, $\mathcal{H}^2$ and $\mathcal{H}^1$ are created from $\mathcal{H}$, as shown in Figure 4; $\mathcal{H}^3$ is obtained by deleting $p(t) = e_3$; $\mathcal{H}^2$ is created by fixing e_3 and deleting e_2; $\mathcal{H}^1$ is created by fixing e_3 and e_2, and by deleting e_1. Subhypergraphs $\mathcal{H}^3$, $\mathcal{H}^2$ and $\mathcal{H}^1$ are shown in Figures 5(a)-5(c) with minimal hypertrees emphasized; the shortest hyperpath weight appears close to t. In each figure the fixed hyperarcs are marked with thick lines.

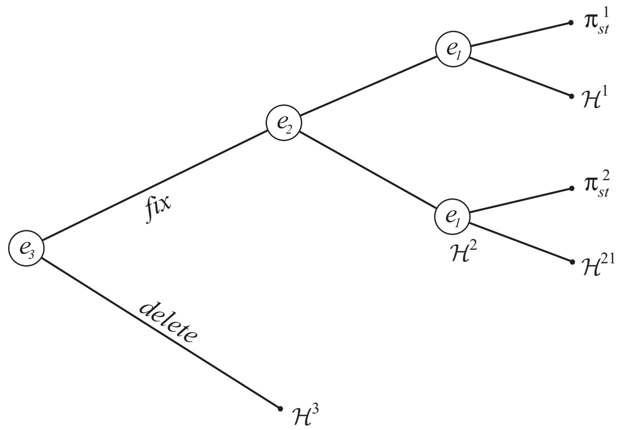
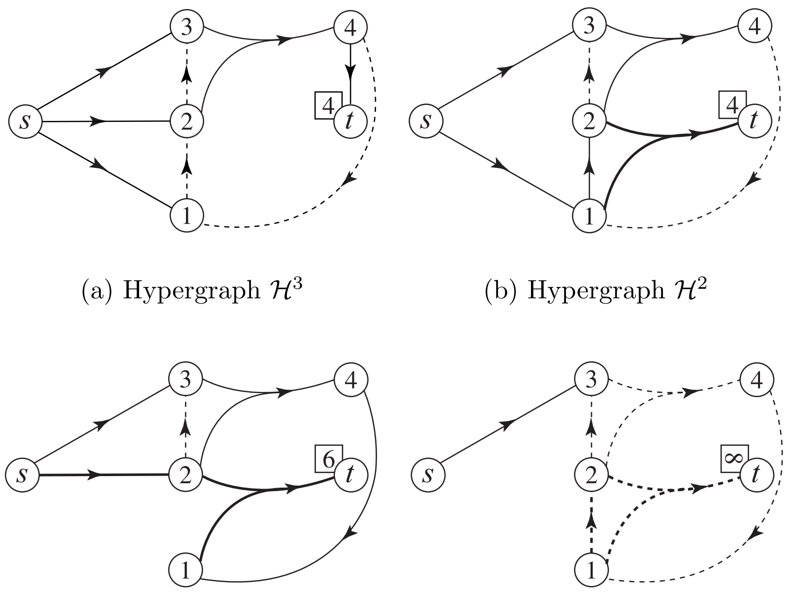


Figure 4: The branching tree of $\mathcal{H}$.



(a) Hypergraph $\mathcal{H}^3$ (b) Hypergraph $\mathcal{H}^2$

(c) Hypergraph $\mathcal{H}^1$ (d) Hypergraph $\mathcal{H}^{21}$

Figure 5: Subhypergraphs generated during the procedure.

Observe that arc $e_6 = (\{1\}, 2)$ has been removed from $\mathcal{H}^1$, since $e_2 = (\{s\}, 2)$ has been fixed. Moreover, the optimal s-t hyperpath in $\mathcal{H}^1$ is not the concatenation of a hypertree and the end-tree defined by e_3 and e_2.
Assume that we select the shortest hyperpath π^2_{st} in $\mathcal{H}^2$ next. Again, $V = \{s, 1, 2, t\}$ is a valid ordering. Since $BS(t)$ and $BS(2)$ have cardinality one, only hyperarc $p(1) = e_1$ can be deleted. This gives subhypergraph $\mathcal{H}^{21}$, shown in Figure 5(d), where no hyperpath

from s to t exists. The third shortest hyperpath thus becomes the hyperpath in $\mathcal{H}^3$, with weight 4.

3.2 An Improved Algorithm

The main drawback of Yen's algorithm for K-shortest paths is that an SPT problem must be solved for each G^i generated by branching. The number of SPT problems to solve is therefore much larger than K, and possibly proportional to Kn. The efficiency of the branching phase can be improved by applying *reoptimization* techniques (see e.g. [15]). These techniques cannot be directly extended to hypergraphs, and are not discussed here. Instead, we propose an improved version of procedure *Yen*, based on a new strategy. Our goal is to *delay* the computation of shortest hypertrees. In particular, hypertrees are computed when a subproblem is *selected* from the list L; the selection order is based on a tight lower bound on the shortest s-t hyperpath weight. The selected subproblems provide a superset of the K shortest hyperpaths. Clearly, this technique is effective if the number of selections is small (e.g. close to K) compared to the total size of L, which is $O(Kn)$.
In order to compute a tight lower bound on hyperpath weight, we take advantage of some properties that are known to hold for graphs, and can be extended to hypergraphs. Let the predecessor function p define the shortest hypertree $\mathcal{T}_s$ in hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. Moreover, denote by W the vector of minimum weights, i.e. $W(u)$ is the minimum weight of an s-u hyperpath, and let $F(W, e)$ denote the value of the weighting function F on hyperarc e with respect to the weights W. For example, for the distance function we would have:

$$F(W, e) = \max_{v \in T(e)} \{W(v)\}.$$

Given node $v \neq s$, suppose that $p(v)$ is removed from $\mathcal{H}$, obtaining a sub-hypergraph $\mathcal{H}'$; let $BS'(v) \neq \emptyset$ be the backward star of v in $\mathcal{H}'$. Compute the value $\overline{W}(v)$ as follows:

$$\overline{W}(v) = \min_{e \in BS'(v)} \{F(W, e) + w(e)\}$$

and let $\overline{p}$ be the predecessor function obtained from p by setting:

$$\overline{p}(v) = \arg \min_{e \in BS'(v)} \{F(W, e) + w(e)\}.$$

Recall that $\overline{p}$ may define a hypertree, but this is not necessarily true.

Theorem 3 The value $\overline{W}(v)$ is a lower bound on the minimum weight $W'(v)$ of an s-v hyperpath in $\mathcal{H}'$. Moreover, if $\overline{p}$ defines a hypertree, $\overline{W}(v) = W'(v)$.

Proof Denote by W' the vector of minimum weights in $\mathcal{H}'$; clearly, $W' \geq W$, and thus $F(W', e) \geq F(W, e)$ for each e in $\mathcal{H}'$. This implies the first claim:

$$\overline{W}(v) = \min_{e \in BS'(v)} \{F(W, e) + w(e)\} \leq \min_{e \in BS'(v)} \{F(W', e) + w(e)\} = W'(v).$$

To prove the second claim, assume that $\overline{p}$ defines a hypertree $\mathcal{T}'$, and let

$$V' = (s = u_1, u_2, \ldots, v = u_i, \ldots, t = u_n)$$

be a valid ordering for $\mathcal{T}'$. Since $p(u_j) = \overline{p}(u_j)$ for each $j < i$, the subtree $\mathcal{T}^{i-1}$ of $\mathcal{T}'$ is a subtree of $\mathcal{T}$; it follows that $W'(u) = W(u)$ for each node u in $\mathcal{T}^{i-1}$. Thus we have $F(W', \overline{p}(v)) = F(W, \overline{p}(v))$, which implies $W'(v) \leq \overline{W}(v)$, and the claim follows. ■

Let $V = (s = u_1, u_2, \ldots, v = u_{i+1}, \ldots, t = u_{q+1})$ be a valid ordering for the shortest hyperpath π_{st} in $\mathcal{H}$, and consider the sub-hypergraph $\mathcal{H}^i$ in the branching step. Recall that $p(v)$ is deleted in $\mathcal{H}^i$. As follows from Theorem 3, $\overline{W}(v)$ is a lower bound for the minimum weight of an s-v hyperpath in $\mathcal{H}^i$. We can extend the node function $\overline{W}$ to all the nodes in π_{st} as follows:

1. let $\overline{W}(u_j) = W(u_j)$ for $1 \leq j \leq i$;
2. for $i + 1 < j \leq q + 1$, let $\overline{W}(u_j) = F(\overline{W}, p(u_j)) + w(p(u_j))$.

Defining $\overline{p}$ according to $\overline{W}(v)$ as above; the following theorem holds true.

Theorem 4 The value $\overline{W}(t)$ is a valid lower bound on the minimum weight $W^i(t)$ of an s-t hyperpath in $\mathcal{H}^i$. Moreover, if $\overline{p}$ defines a hypertree, $\overline{W}(t) = W^i(t)$.

Proof Denote by W^i the vector of minimum weights in $\mathcal{H}^i$, and consider the splitting of π_{st} around v, i.e. $\pi_{st} = (\tau^{i+1}, \eta^{i+1})$, according to the ordering V. Since the shortest s-t hyperpath in $\mathcal{H}^i$ contains η^{i+1}, $W^i(t)$ is computed iteratively similar to $\overline{W}(t)$, that is:

- for $i + 1 < j \leq q + 1$, let $W^i(u_j) = F(W^i, p(u_j)) + w(p(u_j))$.

The first claim then follows since, for each u_j in τ^{i+1}, we have $\overline{W}(u_j) \leq W^i(u_j)$ by Theorem 3. Moreover, if $\overline{p}$ defines a hypertree, it is $\overline{W}(u_j) = W^i(u_j)$ for each u_j in τ^{i+1}, and the second claim follows. ■

In light of Theorem 4, we shall use the value $\overline{W}(t)$ as a lower bound on the shortest hyperpath weight in $\mathcal{H}^i$. Note that we need the minimum weights for all the nodes in $\mathcal{H}$ in order to compute $\overline{W}(t)$; the computation itself takes time $O(size(\mathcal{H}))$ for each $\mathcal{H}^i \in \mathcal{B}(\mathcal{H})$ in the worst case, but is expected to be quite fast in practice.
In our modified Yen's algorithm, a subproblem can be selected at most twice: the first time to compute the minimum hypertree, and (possibly) the second time to output the shortest s-t hyperpath and perform branching. Each subproblem r is represented by the triple $(\mathcal{H}_r, LB_r, \mathcal{T}_r)$. Here, $\mathcal{T}_r$ denotes a shortest hypertree in $\mathcal{H}_r$, if such hypertree is known, or *nil* otherwise; LB_r is a *finite* lower bound on the weight of the minimum s-t hyperpath π_r in $\mathcal{H}_r$. Note that we cannot assume t hyperconnected to s in $\mathcal{H}_r$ when we create r. Initially, L contains the pair $(\mathcal{H}, W(\pi_{st}), \mathcal{T}_s)$. In each iteration, we remove from L a subproblem r with minimum value LB_r; r is then processed according to $\mathcal{T}_r$.

$\mathcal{T}_r =$ ***nil*:** we compute a shortest hypertree in $\mathcal{H}_r$ and assign it to $\mathcal{T}_r$; if t is not connected to s in $\mathcal{H}_r$, we discard r; otherwise, we assign to LB_r the weight of the shortest s-t hyperpath π_r in $\mathcal{H}_p$, and reinsert r in L.

$\mathcal{T}_r \neq$ ***nil*:** we output π_r as the next hyperpath, and we proceed to branching.

In the branching phase, we compute the lower bound $\overline{W}_r^i = \overline{W}(t)$ for each hypergraph $\mathcal{H}_r^i$ in $\mathcal{B}(\mathcal{H}_r)$; if $\overline{W}_r^i$ is finite, we insert in L the subproblem $(\mathcal{H}_r^i, \overline{W}_r^i, nil)$. Procedure *LBYen*, given below, describes the improved algorithm. By $\mathcal{T}_r = SBT(s, \mathcal{H}_r)$ we mean that the shortest hypertree in $\mathcal{H}_r$ is stored in $\mathcal{T}_r$.

Procedure *LBYen*$(\mathcal{H}, s, t, K)$

Step 0 $L = \{(\mathcal{H}, W(\pi_{st}), \mathcal{T}_s)\}$; $k = 1$;

Step 1 if $L = \emptyset$ then STOP; otherwise, let $L = L \setminus \{r\}$ where

$$r = \arg\min_{l \in L} LB_l;$$

Step 2 if $\mathcal{T}_r \neq nil$ go to Step 4;

Step 3 set $\mathcal{T}_r = SBT(s, \mathcal{H}_r)$; if $W(\pi_r) < \infty$ then set $LB_r = W(\pi_r)$ and $L = L \cup \{r\}$; go to Step 1;

Step 4 OUTPUT π_r; $k = k + 1$; if $k > K$ then STOP;

Step 5 for each $\mathcal{H}_r^i$ in $\mathcal{B}(\mathcal{H}_r)$ do:

(a) compute the lower bound $\overline{W}_r^i$;

(b) if $\overline{W}_r^i < +\infty$ then $L = L \cup \{(\mathcal{H}_r^i, \overline{W}_r^i, nil)\}$;

go to Step 1.

It must be remarked that in actual implementations of procedure *LBYen* the reinsertion of r in Step 3 is not always necessary. In fact, reinsertion is necessary only if the weight of π_r is greater than the minimum lower bound in L, i.e.:

$$W(\pi_r) > \min_{l \in L} LB_l.$$

In the other cases, in particular if the lower bound $\overline{W}(t)$ previously computed for subproblem r gives the true weight of π_r, we can proceed to Step 4 without reinsertion. As shown in the computational results, the number of actual reinsertions performed by the procedure is quite low.

Example 3 (continued) Assume that procedure *LBYen* is used in Example 3. The lower bound $\overline{W}(t)$ for subhypergraphs $\mathcal{H}^3$, $\mathcal{H}^2$ and $\mathcal{H}^1$ is equal to the actual weight. Again, assume we select $\mathcal{H}^2$ next. In order to compute $\overline{W}(t)$ for subhypergraph $\mathcal{H}^{21}$, we must compute $\overline{W}(1)$ first; the backward star of node 1 in $\mathcal{H}^{21}$ contains the single arc $e_9 = (\{4\}, 1)$, which gives $\overline{W}(1) = 5$. We thus obtain $\overline{W}(2) = 6$ and $\overline{W}(t) = 12$, which is a weak lower bound on infinity. Subproblem $(\mathcal{H}^{21}, 12, nil)$ is inserted in L and, if selected later, it will be discarded in Step 3. ■

3.3 Acyclic Hypergraphs

The K shortest path problem in acyclic graphs is computationally much easier, since algorithms for the unrestricted problem can be used in this case. Up to a certain extent, this situation extends to acyclic hypergraphs. Here we shall devise a specialized procedure where only one shortest hypertree computation is needed. This resembles the approach used in Eppstein's algorithm for unrestricted K shortest paths. In our case, however, the computation of hyperpath and weights involves more complex operations.

According to Theorem 4, the lower bound $\overline{W}(t)$ always computes the actual hyperpath weights in an acyclic hypergraph. This implies that no reinsertions are performed by Procedure *LBYen*. Next we show that it is not necessary to find a shortest hypertree in each subproblem in order to compute $\overline{W}(t)$.

Consider an acyclic hypergraph $\mathcal{H}$, where

$$V = (u_1 = s, u_2, \ldots, u_{n-1}, u_n = t)$$

is a valid ordering of the nodes; the shortest hypertree $\mathcal{T}_s$, defined by the predecessor p, contains the shortest hyperpath π_{st}. In the branching step we process the nodes according to the ordering V, which of course induces a valid ordering for the nodes in π_{st}.

For notational convenience, assume that sub-hypergraph $\mathcal{H}'$ is obtained from $\mathcal{H}$ by deleting hyperarc $p(u_i)$ and possibly by fixing the hyperarcs in an end-tree η^i. In order to obtain the value $\overline{W}(t)$ in $\mathcal{H}'$, we must compute the value $\overline{W}(u_i)$. Note that, in this phase, we only consider nodes that precede u_i in V; for these nodes, the shortest hypertree and the shortest hyperpath weights in $\mathcal{H}'$ are the same as in $\mathcal{H}$.

Since we process the nodes in π_{st} according to V, the predecessor hyperarc is fixed in $\mathcal{H}'$ for each node u_h in η^i with $h > i$. Moreover, a shortest s-t hyperpath π'_{st} in $\mathcal{H}'$ contains η^i. As a consequence, when branching on π'_{st}, we delete the predecessor of a node u_j only if $j \leq i$. As before, in order to compute $\overline{W}(u_j)$ we only need to consider nodes that precede u_i in V. Clearly, the same argument can be applied to any hypergraph obtained from $\mathcal{H}'$ by recursively applying the branching step.

In conclusion, the value $\overline{W}(t)$ and the shortest hyperpath in each subhypergraph can be computed using the shortest hypertree and hyperpath weights for $\mathcal{H}$. Therefore we can devise a specialized version of Procedure *Yen*, referred to as *AYen*, where we do not apply procedure *SBT* in Step 3(a). Instead, we compute the weight $\overline{W}(u_i)$ and the predecessor $\overline{p}(u_i)$ as in procedure *LBYen*. In this way, we easily obtain the shortest hyperpath π^i_r used in Step 3(b) to create a new subproblem.

4 Computational Results

In this section we test the procedures described in Section 3. The procedures have been implemented in C++ and run on a 700 MHz PIII computer with 512MB RAM using a Linux operating system. The programs have been compiled using the GNU C++ compiler (version 2.96) with optimize option -O.
In our implementation, we use a 2-heap for the set of candidates Q in Procedure 1. The branching tree representing the list L (see Figure 4) is implemented as a dynamic binary tree; all the information related to the subproblems is associated with nodes in the tree, while a heap of tree node pointers is used for the selection phase. In fact, at the end of the procedure the branching tree provides a representation of the K shortest hyperpaths. Anyway, we do not perform any actual "output" operation on the generated hyperpaths.

Class	1	2	3	4	5	6	7	8	9	10
Nodes	100	300	500	800	1000	1000	3000	5000	8000	10000
Arcs	400	1200	2000	3200	4000	2000	6000	10000	16000	20000
Harcs	5000	15000	25000	40000	50000	4000	12000	20000	32000	40000

Table 1: randomly generated test problems.

Ten classes of randomly generated hypergraphs were considered, as shown in Table 1. The generated hypergraphs can be divided into two groups. Hypergraphs in class 1-5 have fewer nodes and are dense: the number of arcs is $4n$ and the number of "true hyperarcs" is $50n$. This gives an average number of 54 hyperarcs in the backward star of a node. Hypergraphs in class 5-10 have more nodes and are sparse, the number of arcs is $2n$ and the number of "true hyperarcs" is $4n$ resulting in an average number of 6 hyperarcs in the backward star of a node.
For all classes, the size of each true hyperarc is randomly generated with a uniform distribution in the interval $[3, 5]$. The weights for each arc is between 500 and 1000, and for each hyperarc it is between 1 and 100. This choice has been made to favor hyperpaths with many true hyperarcs; nevertheless, the percentage of arcs in the generated hyperpaths tends to be relevant, in particular for sparse hypergraphs.
For each class in Table 1, five general and five acyclic hypergraphs were generated. Note that, in the acyclic case, nodes were numbered according to a valid ordering. We shall consider the general and acyclic case separately. As discussed earlier, we shall consider the sum and distance weighting functions, and the mean function restricted to acyclic hypergraphs.

4.1 General hypergraphs

Our goal here is to evaluate and compare the behaviour of procedures *Yen* and *LBYen*. Results are reported in Table 2, where we set $K = 500$. Note that each row in Table 2

				Sum function						Distance function					
Class	Nodes	Arcs	Harcs	Cardinality	Arc %	BTsize	Reinsert	CPU - LB	CPU - Yen	Cardinality	Arc %	BTsize	Reinsert	CPU - LB	CPU - Yen
1	100	400	5000	12,0	44,4	4314	12	1,7	15,1	43,5	20,9	10009	60	2,2	31,3
2	300	1200	15000	12,9	50,4	4794	3	8,0	78,3	55,7	34,6	13147	7	9,4	220,0
3	500	2000	25000	13,1	52,4	5026	6	14,5	147,6	66,4	39,8	14010	44	17,8	427,0
4	800	3200	40000	12,4	54,2	4811	3	24,4	234,0	62,9	46,2	11668	26	29,0	606,5
5	1000	4000	50000	11,9	54,4	4557	3	31,4	284,2	86,2	45,4	16590	11	36,4	1113,4
6	1000	2000	4000	11,2	79,8	3469	4	2,4	14,6	52,9	70,4	7820	6	2,8	35,8
7	3000	6000	12000	10,1	78,7	3467	2	11,1	68,1	55,4	72,3	8974	4	12,5	189,4
8	5000	10000	20000	9,6	78,7	3408	0	20,4	121,9	47,3	71,3	7972	0	22,6	301,8
9	8000	16000	32000	9,0	77,7	3282	0	34,9	196,9	41,7	72,7	6899	0	38,7	445,3
10	10000	20000	40000	8,5	78,1	3200	1	45,0	243,7	39,1	71,2	6832	3	50,0	563,4

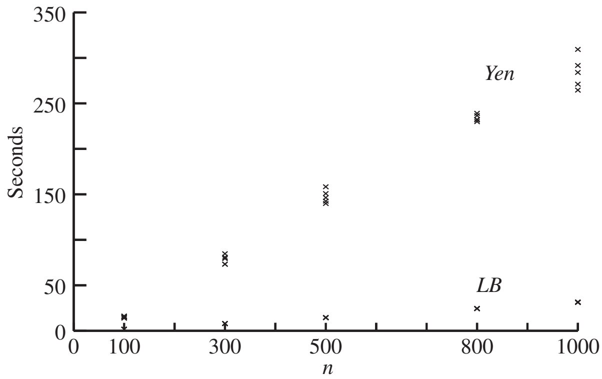
Table 2: Sum and distance functions, $K = 500$ (general hypergraphs).

contains the average results over the five hypergraphs generated for each class. Column "Cardinality" contains the average size of the generated hyperpaths, i.e. the number of arcs plus the number of true hyperarcs, while column "Arc %" contains the average percentage of arcs. Column "Btsize" gives the total number of subhypergraphs in the branching tree for procedure *LBYen*. The branching tree for procedure *Yen* is approximately the same. In column "Reinsert" we give the number of subproblems reinserted in the list L by procedure *LBYen*. Finally, columns "CPU - LB" and "CPU - Yen" contain the CPU time, reported in seconds, for procedures *LBYen* and *Yen*, respectively.

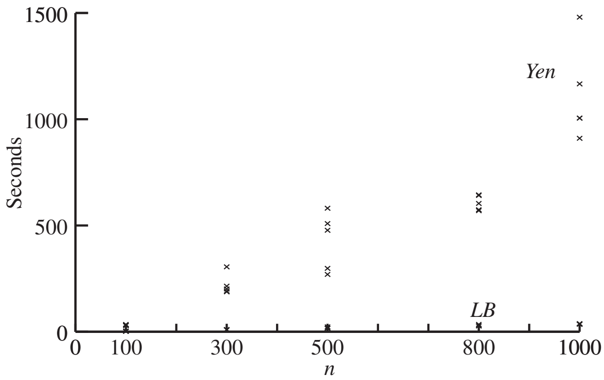
Table 2 shows that the average hyperpath cardinality increases, while the arc percentage decreases, when the distance function is compared with the sum function. This behaviour is to be expected since the sum function is quite sensitive to changes in cardinality and hyperarc percentage. This is not true for the distance which is equal the maximum length of a path contained in the hyperpath. Furthermore, the lower cost assigned to hyperarcs makes hyperpaths with many hyperarcs preferable for the distance. Indeed, for the distance function on dense hypergraphs, the cardinality tends to increase (though slowly) with the number of nodes n. On the contrary for the sum function, and for both functions on sparse hypergraphs, the cardinality does not increase with n.

The above observations also explain the behaviour of the branching tree size, which is expected to be proportional to hyperpath cardinality, more precisely, roughly equal to $K(\text{Cardinality} - f)$, where f is the average number of fixed hyperarcs in the generated hyperpaths.

If we compare the CPU times, we see that procedure *LBYen* outperforms procedure *Yen*. The results confirm that CPU time is roughly proportional to the number of times the



(a) Sum function

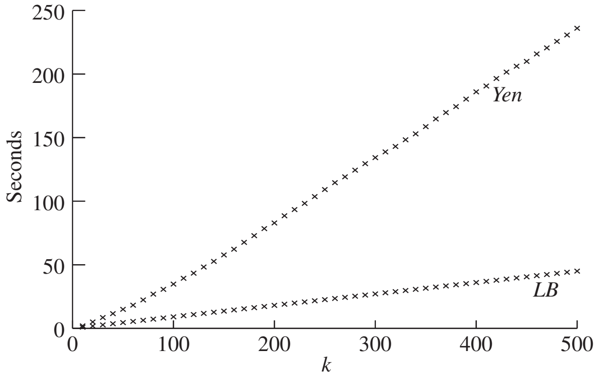


(b) Distance function

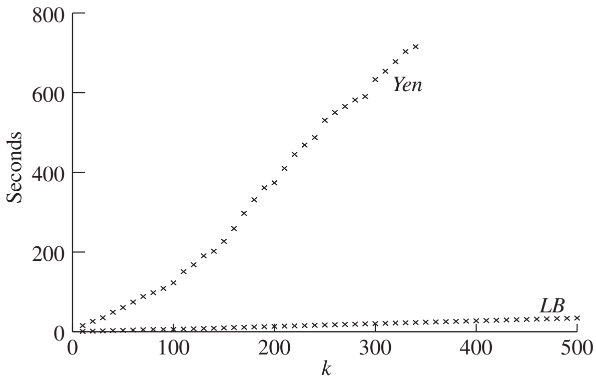
Figure 6: CPU times for dense hypergraphs of increasing size.

SBT procedure is used. Procedure *Yen* solves a shortest hyperpath problem for each subproblem, while *LBYen* does so only for selected subproblems, i.e. K times plus the number of reinsertions. It must be remarked that the actual number of reinsertions in Procedure *LBYen* is quite low (12% of K in the worst case) implying that the lower bound is mostly tight. This number tends to be higher for the distance function on dense hypergraphs, according to the fact that the branching tree size is higher.

Even though the branching tree size is roughly constant, CPU times increase with n, due to the fact that the SBT procedures are applied to larger hypergraphs. In order to investigate this behaviour more deeply, we plotted the CPU time against the number of nodes n for each generated hypergraph. The results for dense hypergraphs are reported in Figure 6(a) and 6(b) for the sum and the distance weighting functions, respectively. Both figures show a linear dependence in n, however, for higher n the behaviour of procedure *Yen* tends to be less stable, and this is particularly true for the distance function. On the contrary,



(a) Sparse hypergraph (sum)



(b) Dense hypergraphs (distance)

Figure 7: CPU time per hyperpath.

procedure *LBYen* is not only faster but seemingly more stable than *Yen*. We omit the plots for the sparse hypergraphs that show a similar linear dependence and an even more stable behaviour.

Finally, we tried to evaluate the computational effort required by each generated hyperpath. To this aim, we recorded the elapsed CPU time every ten generated hyperpaths, for a particular hypergraph and weighting function. Figure 7(a) refers to a large sparse hypergraph (class 10) and to the sum function. Figure 7(b) refers to a large dense hypergraph (class 5) and to the distance. A linear dependence is obvious in both cases, meaning that the computational effort required to generate one single hyperpath is approximatively constant for the first K hyperpaths. Again, the sum function seems to be more stable. As before, the results obtained for other combinations of hypergraph and weighting function show a similar dependence and a more stable behaviour.

				Sum function					Distance function				
Class	Nodes	Arcs	Harcs	Cardinality	Arc %	BTsize	CPU - LB	CPU - AYen	Cardinality	Arc %	BTsize	CPU - LB	CPU - AYen
1	100	400	5000	10,6	47,9	1530	1,7	0,2	24,7	35,0	2436	1,8	0,2
2	300	1200	15000	10,7	53,3	1478	8,1	1,3	35,8	43,7	3179	8,6	1,4
3	500	2000	25000	10,7	54,6	1524	14,9	2,4	37,1	49,5	3702	15,7	2,4
4	800	3200	40000	10,5	55,2	1547	25,2	3,9	39,2	52,0	3376	27,7	3,9
5	1000	4000	50000	9,8	56,1	1396	32,2	4,5	36,7	53,6	3380	36,0	4,9
6	1000	2000	4000	11,0	78,7	1567	1,4	0,4	32,0	73,5	2529	1,5	0,4
7	3000	6000	12000	9,6	77,4	1592	7,1	2,0	28,0	74,3	2113	6,8	2,1
8	5000	10000	20000	8,5	77,0	1453	12,7	3,6	23,9	73,4	2082	12,2	3,8
9	8000	16000	32000	7,9	76,8	1434	21,6	6,6	18,1	74,1	2070	21,0	6,2
10	10000	20000	40000	7,7	75,8	1453	27,3	7,9	21,0	71,6	1947	27,2	7,9

Table 3: Sum and distance functions, $k = 500$ (acyclic hypergraphs).

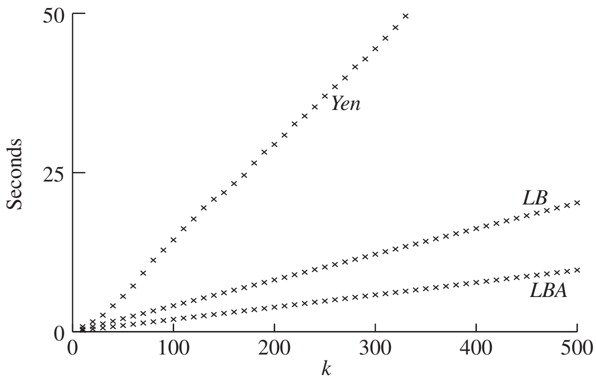


Figure 8: Cost per hyperpath, class 10 (distance).

4.2 Acyclic hypergraphs

The results for the sum and distance functions are reported in Table 3. Since procedure *Yen* is slower, here we only compare the CPU times for procedures *LBYen* and *AYen*. Recall that no reinsertions are performed by *LBYen* in this case.
Most of the observations made for general hypergraphs apply to acyclic hypergraphs too. In this case, however, the difference between sum and distance is reduced. In general, both the hyperpath cardinality and branching tree size are not influenced by the hypergraph size.

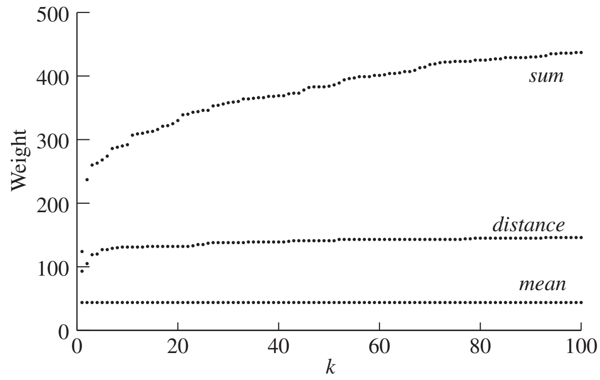
Class	Nodes	Arcs	Harcs	Cardinality	Arc %	BTsize	CPU - AYen
1	100	400	5000	32,79	31,87	2189	0,16
2	300	1200	15000	40,68	46,97	1399	1,25
3	500	2000	25000	58,50	50,27	2487	2,34
4	800	3200	40000	76,59	50,68	2253	4,35
5	1000	4000	50000	72,55	53,22	1956	5,42
6	1000	2000	4000	56,97	73,21	3372	0,47
7	3000	6000	12000	52,79	75,81	3111	2,57
8	5000	10000	20000	47,13	76,73	2472	4,19
9	8000	16000	32000	49,43	75,68	2566	7,16
10	10000	20000	40000	50,65	74,04	2231	8,78

Table 4: Mean function, $k = 500$ (acyclic hypergraphs)

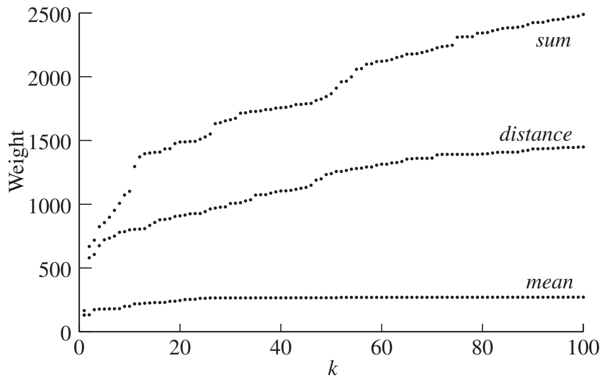
Comparing the CPU times for *LBYen* and *AYen* shows that the latter performs best as expected. It must be remarked that the CPU time for *AYen* increases almost linearly with n. Since *AYen* finds a minimal hypertree only once, this behaviour does not depend on the SBT procedure. In fact, it is due to the representation of sub-problems in the branching tree. Indeed, in the current implementation, each generated sub-hypergraph involves an $O(m)$ computational cost. Using a more sophisticated implementation, this cost may be made proportional to the cardinality of the hyperpath. Clearly, this might dramatically reduce the CPU times for *AYen* which (according to the results in Table 3) would be almost independent of the hypergraph size.

Like for general hypergraphs, we made plots similar to the ones in Figure 6 and 7. They actually reveal quite a similar behaviour. Here, we only report one of them in Figure 8.

In Table 4 we report the results of procedure *AYen* for the mean function. As expected, the average hyperpath cardinality grows considerably compared to the sum function, which results in a larger branching tree size. However, the hyperpath cardinality increases at least by a factor three, while the branching tree size is at most doubled. This difference can be explained since many of the sub-hypergraphs generated in the branching phase are infeasible, that is they do not contain an s-t hyperpath. This situation is detected in procedure *AYen* which does not actually generate the subproblem. Besides the above observations, the behaviour of *AYen* for the mean and sum functions is similar.



(a) Dense hypergraph (class 5)



(b) Sparse hypergraph (class 10)

Figure 9: The shortest 100 weights for different weighting functions.

4.3 The weight of suboptimal hyperpaths

In order to give a better intuition of the behaviour of our K shortest hyperpath procedures, we turn our attention to the weights of the generated hyperpaths. In Figure 9 we plot the weights of the 100 shortest hyperpaths found in a couple of hypergraphs, for the sum, distance and mean weighting functions. Again, a dense hyperpath and a sparse one are considered. It is apparent that the increase in the weight is highest for the sum function and lowest for the mean. As expected, in all cases the increase is higher in the sparse case. In particular, the increase in the mean function for the dense hypergraph is negligible, implying that many alternate shortest or nearly shortest hyperpaths exist. For both the sum and the distance functions, the rate of increase is quite impressive at the beginning. However, it decreases rapidly afterwards.

5 Applications

Shortest hyperpath models have been proposed for several problems in different areas. For example, applications to some classical problems in Computer Science are cited by Ausiello *et al.* [1]. Relevant applications are known in transportation and in combinatorial optimization. In this section we shall consider in detail two of them, namely, optimal routing problems in dynamic networks, and a separation problem arising in a branch-and-cut method for maximum Horn satisfiability. Bicriteria shortest hyperpath algorithms will be discussed too.

Two other applications deserve to be mentioned. The first is related to the hypergraph model for *transit networks* proposed by Nguyen and Pallottino [18, 6]. Transit networks consist of a set of *bus lines* connected to *stop nodes* where passengers board or unboard buses. In the hypergraph model, a hyperarc represents the set of *attractive* bus lines for a passenger waiting at a stop node; a shortest hyperpath represents a set of attractive origin-destination routes. The hypergraph model is embedded within a *traffic assignment* model, based on Wardrop's equilibrium, where passengers are assumed to travel along their shortest available hyperpaths. In the context of iterative methods for traffic assignment, it could be computationally useful to identify alternate optimal hyperpaths. More general, the possibility of exploiting ϵ-suboptimal hyperpaths might be taken into account.

The second application is related to *minimum makespan assembly problems.* As shown in [8], an assembly line can be represented by a suitable hypergraph, where each hyperarc represents a machine operation linking two or more *subassemblies* together. A hyperpath thus represents a particular *assembly plan.* Assuming that each operation has a cost as well as an execution time, shortest hyperpaths with respect to the sum and distance weighting functions give assembly plans with minimum total cost or minimum execution time (with an unlimited number of machines), respectively. Observe that a "good" assembly plan should represent a trade-off between execution time and cost; clearly, this is related to the aforementioned bicriteria shortest hyperpath problems. In general, scheduling a given assembly plan on a fixed number of machines in order to minimize its *makespan* is a hard problem; approximated methods are discussed in [8]. A possible approach for refining these methods would be to generate several candidate assembly plans; an "optimal" plan would then be chosen according to its approximated minimum makespan scheduling, possibly taking into account other objectives.

5.1 Random time dependent shortest paths

We consider *random time dependent* dynamic networks, where the travel time through an arc is a random variable whose distribution depends on the departure time. In particular, we concentrate on *discrete* dynamic networks *(RTDN)*, where departure times are integers in a finite interval. These networks have recently attracted a growing attention, related to applications such as hazardous material transportation or packet routing in congested communication networks (see e.g. [16, 17]). In these contexts, it is relevant to provide alternate solutions to allow for real-time routing decisions. Moreover, bicriteria shortest

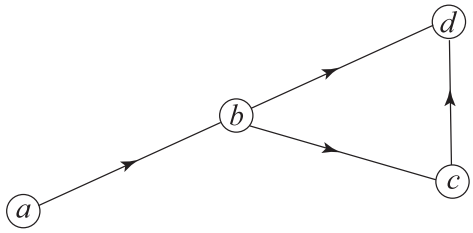


Figure 10: The topological network G.

$(i,j),t$	$(a,b)\,,0$	$(b,c)\,,1$	$(b,c)\,,2$	$(b,d)\,,1$	$(b,d)\,,2$	$(c,d)\,,2$	$(c,d)\,,3$
$I\left((i,j),t\right)$	$\{1,2\}$	$\{2\}$	$\{3\}$	$\{3\}$	$\{5\}$	$\{3,4\}$	$\{4,6\}$
p_{ijt}	$\left\{\frac{1}{3},\frac{2}{3}\right\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\{1\}$	$\left\{\frac{1}{4},\frac{3}{4}\right\}$	$\left\{\frac{3}{4},\frac{1}{4}\right\}$

Table 5: Input parameters for the RTDN.

hyperpaths (discussed later) can provide an efficient trade-off between different objectives, such as cost/exposure or time/reliability.
Hall [9] introduced the problem of finding the minimum expected travel time (*MET*) through a dynamic network. He pointed out that the best route in a dynamic network does not necessarily correspond to an origin-destination path. Instead, a *strategy* must be found that assigns optimal successors to a node as a function of time. As shown in [20], directed hypergraphs can be used to model discrete dynamic networks, and the MET problem can be reduced to solving a suitable shortest hyperpath problem. A deep computational analysis of hypergraph algorithms for the MET problem can be found in [16].
In addition, the hypergraph model shows a high degree of flexibility. Optimal strategies under different objectives, such as min-max travel time, min expected cost and min-max cost, can be found by using suitable weights and weighting functions (see [20]). Additional features, like (hard or weak) time windows, can be easily introduced. We illustrate the model by means of the following example.
Consider the *topological network* $G = (N, A)$ in Figure 10 with destination node d. Let the travel time distribution be given in Table 5. Here a pair $((i,j)\,,t)$ corresponds to a possible leaving time t from node i along arc $(i,j)\,;$ $I\left((i,j)\,,t\right)$ are possible arrival times at node j and p_{ijt} the corresponding probabilities. If we e.g. leave node c at time 2 along arc $(c,d)\,,$ we will arrive at node d at time 3 with probability $\frac{1}{4}$ or at time 4 with probability $\frac{3}{4}$.
Given G and Table 5, a *time expanded hypergraph* $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ can be created as follows. Create a node i_t for each possible arrival time t at node i. For each pair $((i,j)\,,t)$ create a hyperarc $e = \big(\{j_h : h \in I((i,j),t)\}, i_t\big)$. Finally, a dummy node s and dummy arcs $\big(\{s\}, d_t\big)$ are created. The time expanded hypergraph $\mathcal{H}$ is shown in Figure 11. Note that the time expanded hypergraph is acyclic and the orientation of the hyperarcs is opposite

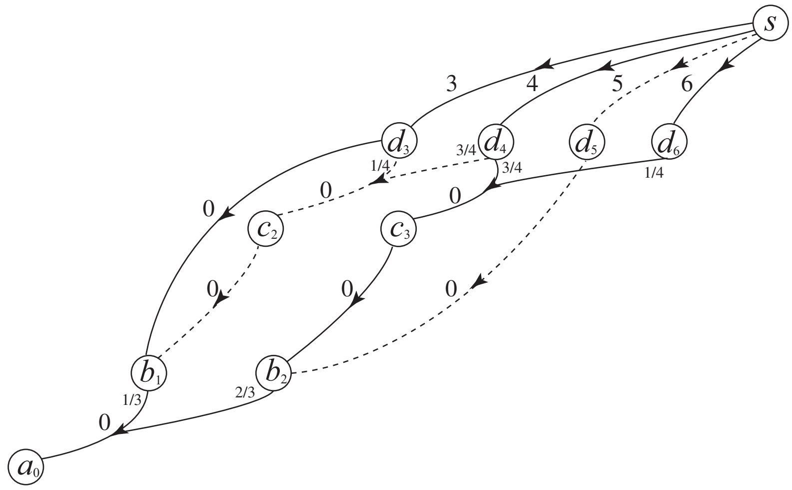


Figure 11: The time expanded hypergraph $\mathcal{H}$.

to the orientation of arcs in G. In addition the size of $\mathcal{H}$ is proportional to the size of the input data.

As shown in [20], each strategy is represented uniquely by a hypertree $\mathcal{H}_s$, that is by a predecessor function p. The best strategy for the MET problem, for a traveller leaving a at time zero, is shown in Figure 11 emphasized. Here we have $p(b_1) = (\{d_3\}, b_1)$ and $p(b_2) = (\{c_3\}, b_2)$, meaning that, when leaving from b at time 1, we go directly to d; while at time 2, we go to c first.

Now assign weight t to each dummy arc $\big(\{s\}, d_t\big)$ and zero weight to all other hyperarcs; assign multiplier p_{ijh} to tail node j_h in each hyperarc $e = (\{j_h : \ h \in I((i,j),t)\}, i_t)$. In this case, for a given strategy, the mean of a hyperpath π from s to a node i_t gives the expected arrival time at d when leaving i at time t. Moreover, the distance of π gives the maximum possible arrival time at d.

Therefore, a strategy minimizing the expected travel time (the maximum possible travel time) can be found by solving a shortest hypertree problem with respect to the mean (the distance) function. Clearly, the K best strategies can be found by finding the K shortest hyperpaths on the time expanded hypergraph.

Suppose we want to find the three best strategies to the MET problem in the hypergraph of Figure 11. The best hyperpath has a mean $W(a_0) = 4$. The second best strategy gives $W(a_0) = 4.25$ with $p(b_1) = (\{c_2\}, b_1)$ and $p(b_2) = (\{c_3\}, b_2)$; the third best strategy gives $W(a_0) = 4.\overline{3}$ with $p(b_1) = (\{d_3\}, b_1)$ and $p(b_2) = (\{d_5\}, b_2)$. Observe that the third hyperpath has a distance equal to five, while the first two have distance six.

5.2 The separation problem for Max Horn SAT

Propositional satisfiability problems represent a most relevant research area in Combinatorial Optimization and Computational Logic. Directed hypergraphs proved to be a quite effective tool in this area, both from a theoretical and a practical point of view, see e.g. [6, 5]. For example, the *maximum satisfiability* problem for Horn formulas (Max Horn SAT) turns out to be equivalent [5] to the problem of finding a *minimum cut* in a hypergraph (MCH). Since Max Horn SAT is NP-hard, also MCH is, opposed to the well-known minimum cut problem in graphs.
Several formulations of MCH have been investigated in [5]; in particular, one formulation is as follows. Define the *cost* of hyperpath π as the sum of the weights of its hyperarcs. Problem MCH asks to assign 0-1 weights to the hyperarcs so that the cost of each s-t hyperpath is at least one, and the sum of the assigned weights is minimum. This leads to the following ILP problem:

$$P(\Pi) = \begin{cases} \min \sum_{e \in \mathcal{E}} x(e) & \\ c_\pi^T x \geq 1, & \forall \pi \in \Pi; \\ x \in \{0,1\}^m & \end{cases}$$

where $c_\pi \in \{0,1\}^m$ is the *characteristic vector* of hyperpath π, i.e. $c_\pi^T x$ gives the cost of π. In practice, since $P(\Pi)$ may have an exponential number of constraints, a relaxation $P(\overline{\Pi})$ is considered, where only a subset $\overline{\Pi} \subset \Pi$ of hyperpaths are included. A suitable set $\overline{\Pi}$ can be built by means of a *row generation* procedure. For a given $\overline{\Pi} \subset \Pi$, let $\overline{P}(\overline{\Pi})$ be the linear relaxation of $P(\overline{\Pi})$, and let $\overline{x}$ be the optimal solution of $\overline{P}(\overline{\Pi})$. The *separation problem* finds an s-t hyperpath $\pi \in \Pi \setminus \overline{\Pi}$ whit cost less than one (if any), where the hyperarcs are assigned the weights $w = \overline{x}$. The violated constraint $c_\pi^T x \geq 1$ is then added to $\overline{\Pi}$. Based on the row generation procedure, a *branch and cut* algorithm for MCH has been devised in [5].
Observe that the cost of a hyperpath is not an additive weighting function. Indeed, it has been proved in [2] that the problem of finding a minimum cost s-t hyperpath is strongly NP-hard. As proved in [5], the separation problem for $\overline{P}(\overline{\Pi})$ is equivalent to the minimum cost hyperpath problem (in decisional version), and is NP-complete. A quite simple approximate separation heuristic is adopted in [5]. Even though this approximation performs well for some classes of instances, more sophisticated techniques seem to be necessary to improve the effectiveness of the algorithm. In particular, it is crucial to separate several violated constraints in the same cut generation step. To this aim, K-shortest hyperpaths procedures can be used.
Let us consider the sum weighting function first. It is easy to see that, for each hyperpath, the sum gives an upper bound on the cost. Therefore a hyperpath with a sum less than one provides a violated constraint; informally this defines a (possibly empty) set of cuts that can be generated "easily". In addition, it may be sensible to consider some of the hyperpaths with sum equal to or slightly larger than one, since some of them may have a cost less than one. Indeed, a hyperpath with a low cost is likely (though not required!) to have a low sum too.

Consider now the distance weighting function. Clearly, for each hyperpath, the distance gives a lower bound on the cost. Therefore only hyperpaths with a distance less than one need to be considered, i.e. they define a superset of the violated constraints. Clearly, hyperpaths with a low distance are likely (though not required!) to have a cost less than one.

In conclusion, the sum and the distance functions show different properties that can be exploited to devise an exact as well as an approximate cut generation procedure. These functions may also be combined, e.g. enumerating hyperpaths by sum, but dropping sub-hypergraphs where the minimum distance is not less than one.

5.2.1 Approximating hard shortest hyperpath problems

Besides the specific application to cut generation discussed above, it is apparent that a K shortest hyperpaths procedure can be used to find a minimum cost s-t hyperpath. More precisely, we rank hyperpaths by distance, keeping track of the minimum cost hyperpath generated in the process. The procedure terminates as soon as the distance of the next ranked hyperpath is greater than or equal to the minimum cost found so far. Here we exploit the fact that the distance is a lower bound on the cost.

The same approach can be used to solve the shortest hyperpath problem for any "difficult" weighting function, besides cost. To this aim, procedure *LBYen* can be modified as follows:

- in Steps 2 and 3, find hyperpath π_r by means of a heuristic procedure, and proceed to branching;
- in Step 5(a), replace the lower bound $\overline{W}(t)$ with an available lower bound.

It must be remarked that in Step 5(a) we need a lower bound on the optimal hyperpath weight which is not necessarily provided by or related to a tractable weighting function. Clearly, the effectiveness of the approach depends on the tightness of the available lower bound.

5.3 Bicriteria shortest hyperpaths

Algorithms based on K shortest paths procedures have been proposed for *bicriteria* shortest path problems. Assume that two *criteria*, e.g. time and cost, are associated with each arc of a graph. In general, there does not exist a path that is optimal for both criteria. Instead, a decision maker would be interested in finding *efficient* paths, that is solutions where the cost (respectively, time) criterion cannot be improved without getting a worse time (respectively, cost).

A possible approach to these problems is based on a two–phases method. Here first phase finds a subset of the efficient paths which are relatively easy to find ("supported" paths). In the second phase, a K shortest path procedure is used to find (some of) the remaining efficient paths. This approach has been proposed by Handler and Zang [10] for the *constrained* shortest path problem, i.e. finding a minimum cost path whose total time does not

exceed a given limit. Later, it has been extended to the more general problem of listing all the efficient paths, or a particular subset of them (see e.g. [3]).
Clearly, bicriteria problems can be extended to hypergraphs, and solved by a two-phases approach. In this case, K shortest hyperpath procedures would be used in the second phase. As discussed earlier, bicriteria shortest hyperpaths may be quite relevant in relation with dynamic networks. These problems have recently been investigated in [19], and will be the subject of a forthcoming paper.

6 Conclusions

In this paper, we introduced and investigated the K shortest hyperpath problem in directed hypergraphs. Even though several hyperpath models have been proposed in the literature, this problem has not yet been considered. The main contributions of this paper can be summarized as follows.
First, we pointed out the lack of symmetry between the graph and hypergraph case, which prevents the "forward branching" approach from being used.
Second, we extended to hypergraphs the method of Yen, and we proposed an algorithmic improvement that turned out to be quite effective in computational experiments. We also pointed out that acyclic hypergraphs are an easier case, as happens for graphs, and we devised a quite fast specialized procedure.
Finally, we showed that K shortest hyperpaths algorithms have several potential applications in different relevant areas.
We believe that the results in this paper provide a stimulating starting point for further research, in particular for the analysis of bicriteria shortest hyperpath problems.
The specialized method developed in procedure *LBYen* deserves some further comments. Our approach was inspired by some techniques for K shortest paths that cannot be directly extended to hypergraphs. This motivated us to introduce the innovative strategy used in *LBYen*. To the best of our knowledge, a variant of Yen's algorithm based on a similar strategy for digraphs has not yet been proposed in the literature, and may turn out to be effective also for K shortest paths algorithms. In this case, research in the more complex setting of shortest hyperpaths would have a positive feedback on the more "classical" area of shortest paths.

References

[1] G. Ausiello, P. G. Franciosa, and D. Frigioni. Directed hypergraphs: Problems, algorithmic results, and a novel decremental approach. In *Lecture Notes in Computer Science 2202*, pages 312–328. Springer Verlag, 2001.

[2] G. Ausiello, G. F. Italiano, and U. Nanni. Optimal traversal of directed hypergraphs. Technical Report TR–92–073, International Computer Science Institute, Berkeley, CA, September 1992.

[3] J. M. Coutinho-Rodrigues, J. C. N. Climaco, and J. R. Current. An interactive bi-objective shortest path approach: Searching for unsupported nondominated solutions. *Computers and Operations Research*, 26:789–798, 1999.

[4] D. Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):653–674 (electronic), 1999.

[5] G. Gallo, C. Gentile, D. Pretolani, and G. Rago. Max Horn SAT and the minimum cut problem in directed hypergraphs. *Mathematical Programming*, 80:213–237, 1998.

[6] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen. Directed hypergraphs and applications. *Discrete Applied Mathematics*, 42:177–201, 1993.

[7] G. Gallo and S. Pallottino. Hypergraph models and algorithms for the assembly problem. Technical Report 6, Dipartimento di Informatica, Università di Pisa, March 1992.

[8] G. Gallo and M. G. Scutellà. Minimum makespan assembly plans. Technical Report 10, Dipartimento di Informatica, Università di Pisa, September 1998.

[9] R. W. Hall. The fastest path through a network with random time-dependent travel times. *Transportation Science*, 20(3):182–188, 1986.

[10] G. Y. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10:293–310, 1980.

[11] W. Hoffman and R. Pavley. A method for the solution of the N'th best path problem. *Journal of the Association for Computing Machinery*, 6:506 – 514, 1959.

[12] R. G. Jeroslow, K. Martin, R. L. Rardin, and J. Wang. Gainfree Leontief substitution flow problems. *Mathematical Programming*, 57:375–414, 1992.

[13] V. M. Jiménez and A. Marzal. Computing the k shortest paths: A new algorithm and an experimental comparison. *Lecture Notes in Computer Science*, 1668:15 – 29, 1999.

[14] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path. *Management Science*, 18(7):401–405, March 1972.

[15] E. Q. V Martins and M. M. B. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. Technical report, Centro de Informatica e Sistemas, 2000. Available at http://www.mat.uc.pt/∼eqvm/.

[16] E. D. Miller-Hooks. Adaptive least-expected time paths in stochastic, time-varying transportation and data networks. *Networks*, 37(1):35–52, 2000.

[17] E. D. Miller-Hooks and H. S. Mahmassani. Optimal routing of hazardous materials in stochastic, time-varying transportation networks. *Transportation Research Record*, 1645:143–151, 1998.

[18] S. Nguyen and S. Pallottino. Equilibrium traffic assignment for large scale transit networks. *European Journal of Operational Research*, 37:176–186, 1988.

[19] L. R. Nielsen, K. A. Andersen, and D. Pretolani. Bicriteria shortest hyperpaths (biSBT). Available at http://www.imf.au.dk/∼relund/, September 2001.

[20] D. Pretolani. A directed hypergraph model for random time dependent shortest paths. *European Journal of Operational Research*, 123:315–324, June 2000.

[21] K. Thulasiraman and M.N.S. Swamy. *Graphs: Theory and Algorithms.* Wiley - Interscience, 1992.

[22] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971.