4.4 Yen's Algorithm Applied to Hypergraphs

With the fundamental problems presented in the state space approach, we need to devise a new approach that can handle larger instances.

The fundamental realization was: What if we apply Yen's Algorithm directly on the HoSP instead of on a derived graph?

This efficient approach to find the K best synthesis plans in the HoSP will be described below.

Recall that the problem of finding the K best synthesis plans in the HoSP, is equivalent to finding the K synthesis plans with the lowest cost, with the cost functions defined in 3.1.

4.4.1 Motivation

The state space approach for finding the K best synthesis plans, have several problems, i.e. with regard to uniqueness of the solutions generated. The problem imply that the number of shortest paths, in the state space, necessary to find K unique synthesis plans, often exceeds K by a factor 2 in real life examples.

As earlier described this leads to a large overhead which is undesirable. Especially because you often choose a large K. An efficient method that only generates unique plans is therefore desired. The algorithm given in [37] by Nielsen *et al.*, is an algorithm for finding the K shortest hyperpaths in a B-hypergraph.

Since the HoSP is a B-hypergraph, and a hyperpath in the HoSP describes a unique synthesis plan, the algorithm in [37], can be applied directly to our model.

The algorithm is based upon Yen's Algorithm, but with a different branching step. Like in Yen's Algorithm we need an algorithm for finding the shortest hyperpath. The algorithm we use is the dynamic programming approach described earlier in this thesis. The dynamic programming approach, as shown earlier, runs in $\mathcal{O}(|V| + |E|)$.

The algorithm will be described in detail below, but the basic idea is the same as Yen's algorithm. Initially the shortest hyperpath is found, next the algorithm recursively considers all hyperpaths which deviates from one of the shortest hyperpaths found so far. One of the considered hyperpaths is the shortest, and is added to the set of already found shortest hyperpaths. This continues until K hyperpaths have been found.

4.4.2 Theory

Theory and definitions required for understanding how the algorithm works is presented below.

From HoSP to *st* Hypergraph

Since the algorithm by [37], finds the K shortest *st* hyperpaths in a hypergraph. We will present a simple transformation from our HoSP to a *st* hypergraph, see Algorithm 8. Note t is the root in the HoSP.

Algorithm 8 Transformation of HoSP.

```
1: function TransformHosp(Hosp)
2:     Let H be a new hypergraph
3:     H.V ← Hosp.V
4:     H.E ← Hosp.E
5:     add S to H
6:     for all u ∈ Hosp.Input do
7:         add edge from S to u in H
8:     end for
9:     return H
10: end function
```

Note that the transformation is not necessary for an actual implementation, since the only transformation done is adding a virtual starting node to all input compounds.

B-Hyperedges and B-Hypergraphs

Recall that a directed hypergraph is a pair $H = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the set of nodes, and $\mathcal{E}$ is the set of directed hyperedges.

A hyperedge is an ordered pair $e = (\text{Tail}(e), \text{Head}(e))$ of subsets of $\mathcal{V}$.

Often $\text{Tail}(e) \cap \text{Head}(e) = \emptyset$ is an assumption. A hyperedge is called a backward hyperedge (B-edge) if $|\text{Head}(e)| = 1$.

If all hyperedge in $\mathcal{H}$ are B-edges, then $\mathcal{H}$ is a B-hypergraph.

Ordering

A valid ordering in $\mathcal{H}$, is a topological ordering of the nodes

$$\mathcal{V} = \{u_1, u_2, ..., u_n\}$$

such that, for any $e \in \mathcal{E} : (u_j \in \text{Tail}(e)) \vee (\text{Head}(e) = u_i) \Rightarrow j \leq i$.

Hyperpath

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

1. $\mathcal{E}_\pi \subseteq \mathcal{E}$
2. $s, t \in \mathcal{V}_\pi = \cup_{e \in \mathcal{E}_\pi}(\text{Tail}(e) \cup \{\text{Head}(e)\})$
3. $u \in \mathcal{V}_\pi \setminus \{s\} \Rightarrow$ is connected to s in $\mathcal{H}_\pi$

If there exist such a hyperpath from s to t in $\mathcal{H}$, we say s is *hyperconnected* to t.

If s is hyperconnected to t in a B-hypergraph via the hyperpath π_{st}, then s is also hyperconnected to all other nodes $u \in \mathcal{V}_\pi$ via hyperpath π_{su} with the node set $\mathcal{V}_u \subseteq \mathcal{V}_\pi$ and hyperedgess $\mathcal{E}_u \subseteq \mathcal{E}_\pi$.

We say π_{su} is a *subhyperpath* of π_{st}.

Since B-hypergraphs only has B-hyperedges, condition 3 guarantees, that for any node $u \in \mathcal{V}_\pi \setminus \{s\}$ there exist a hyperedge $e \in \mathcal{E}_\pi$ where $\text{Head}(e) = \{u\}$.

Because of minimality there exists exactly one such hyperedge, hence $e = p(u)$ is the predecessor hyperedge for the node u.

The predecessor function $p : \mathcal{V}_\pi \to \mathcal{E}_\pi$ defines the hyperpath π_{st}.

Condition 1 guarantees that there exists at least one hyperedge $e \in E_\pi$ with $u \in \text{Tail}(e)$. It follows that there exists an ut-hyperpath.

4.4.3 Finding the K shortest hyperpaths

The definition of the K shortest hyperpaths, is defined in the following way.

Let $H = (V, E)$ be a directed B-hypergraph, $s \in \mathcal{V}$, and $t \in V$, where node s and t is connected. The set of all hyperpaths from s to t is denoted $\mathcal{P} = \{\pi | \pi \text{is a hyperpath from } s \text{ to } t\}$.

Since π_{st} is a hyperpath and per definition acyclic. Hence there exists a valid ordering of the nodes.

$$\{s, u_1, u_2, ..., u_q = t\}$$

Since we have a valid ordering of the nodes, we can easily create a ordering of the hyperedges with the predecessor function.

$$\{p(u_1), p(u_2), ..., p(u_q)\}$$

This ordering of the hyperedges is used to partition the remaining hyperpaths into q disjoint subsets $\mathcal{P}^i, 1 \leq i \leq q$ in the following way:

- $\mathcal{P}^q$ is all st hyperpaths which does not contain the hyperedge $p(u_q)$
- For each hyperpath $\mathcal{P}^i$ where $1 \leq i \leq q$ is the set of all st hyperpaths which contains the hyperedges $p(u_{i+1}), p(u_{i+2}), ..., p(u_q)$, but not the hyperedge $p(u_i)$.

In other words, each set $\mathcal{P}^i$ consists of all the st hyperpaths that have the same $q - i$ hyperedges as π_{st}, and deviates at exactly the ith hyperedge.

All hyperpaths, except π_{st}, must be contained in one of the defined sets.

Idea

The idea behind the algorithm is the following:

- Given the shortest hyperpath π_{st} in H, create the partition $\mathcal{P} \setminus \{\pi_{st}\}$.
- The second shortest hyperpath in H, is the shortest hyperpath in $\mathcal{P} \setminus \{\pi_{st}\}$, hence it must be the shortest hyperpath in one of the sets $\mathcal{P}^i, 1 \leq i \leq q$ which $\mathcal{P}$ represents, lets call the set $\mathcal{P}^j$.
- The shortest hyperpath π_2 in $\mathcal{P}^j$ is used to create the partitions $\mathcal{P}^{j,m}, 1 \leq m \leq |\pi_2|$.
- The third shortest hyperpath in $\mathcal{H}$ will be in $\mathcal{P} \setminus \{\pi_{st}, \pi_2\}$, hence one of the partitions $\mathcal{P}^i, i = 1, ..., j-1, , j+1, ..., q$, or $\mathcal{P}^{j,m}, 1 \leq m \leq |\pi_2|$.
- Continue partitioning until K shortest hyperpaths are found.

Subhypergraphs

Storing $\mathcal{P}$ as sets of hyperpaths requires precomputation of all hyperpaths, making the whole algorithm unnecessary. Instead a set of hyperpaths is represented as a subhypergraph of $\mathcal{H}$.

$\mathcal{H}$ is the representation for all *st* hyperpaths in $\mathcal{H}$, hence a representation of $\mathcal{P}$.

Each $\mathcal{P}^i$ is represented by a hypergraph $\mathcal{H}^i = (\mathcal{V}^i, \mathcal{E}^i)$, which is derived in the following way:

- $\mathcal{V}^i = \mathcal{V}$, i.e all the nodes is included.
- The hyperedges $\mathcal{E}^i \subseteq \mathcal{E}$ in the following way:
  - The hyperedge $p(u_i)$ is removed from $\mathcal{H}^i$
  - The hyperedges $p(u_{i+1}), p(u_{i+2}), ..., p(u_q)$ is fixed in $\mathcal{H}^i$. Formally $u_j, j \geq i : \mathrm{BS}(u_j) = \{p(u_j)\}$, which means that all other ingoing edges to the last $q - i$ nodes will be removed.

The method for creating, the hypergraph $\hat{H}$ representation of a set of hyperpaths P' along the shortest path $\hat{\pi} \in \hat{P}$, is called `BackwardsBranching`.

Algorithm 9 Backward Branching method for a B-hypergraph.

Require: A valid ordering of the nodes $\{s, u_1, u_2, ..., u_q = t\}$.

```
1: function BackwardsBranching(Ĥ)
2:     B ← ∅
3:     for i = 1 to q do
4:         Let Ĥ^i be a new hypergraph
5:         Ĥ^i.V = Ĥ.V
6:         //Remove the i'th hyperedge from Ĥ^i
7:         Ĥ^i.E = Ĥ.E \ {π̂.p(u_i)}
8:         //Fix the backtree of Ĥ^i
9:         for j = i + 1 to q do
10:            Ĥ^i.BS(u_j) = {π̂.p(u_j)}
11:        end for
12:        B = B ∪ {Ĥ^i}
13:    end for
14:    return B
15: end function
```

The algorithm for finding the K shortest hyperpaths `YenHyp`, maintains a priority list L of tuples $(\hat{H}, \hat{\pi})$, where $\hat{H}$ is the hypergraph representation of a set of *st* hyperpaths in $\mathcal{H}$. The list L is sorted with respect to the cost of π.

In each iteration of `YenHyp` the pair (H', π') with the smallest cost is popped off the list, π' is outputted. Next H' is partitioned along π' using `BackwardsBranching`, the shortest path is found in each of the new partitions. All new tuples is inserted into L. The algorithm terminates when the K hyperpaths are found.

Algorithm 10 K Shortest Paths Algorithm in B Hypergraph.

```
1: function YenHyp(H, s, t, K)
2:     Let L be a new priority queue
3:     Let A be new list
4:     π ← ShortestPath(H, s, t)
5:     Insert (H, π) into L
6:     for k = 1 to K do
7:         if L = ∅ then
8:             Break
9:         end if
10:        (H', π') ← L.pop
11:        Insert π' into A
12:        for all H^i in BackwardsBranching(H', π') do
13:            π^i ← ShortestPath(H^i, s, t)
14:            if π^i is complete then
15:                Insert (H^i, π^i) into L
16:            end if
17:        end for
18:    end for
19:    return A
20: end function
```

Note that an algorithm ShortestPath is used to find the shortest hyperpath in a hypergraph. As shown earlier this can be done effectively with dynamic programming, which runs in $\mathcal{O}(|\mathcal{V}| + |\mathcal{E}|)$.

`YenHyp` makes K iterations, in each iteration the length of a hyperpath determines the number of calls to ShortestPath, the worst case for the length of a hyperpath is $\mathcal{O}(|\mathcal{V}|)$. Hence the running time for `YenHyp` becomes:

$$\mathcal{O}(\mathcal{K} \cdot |\mathcal{V}| \cdot (|\mathcal{V}| + |\mathcal{E}|))$$

4.4.4 Example

We will show a small example of how the algorithm works by applying it to the hypergraph shown in Figure 44. The weight will remain the same throughout the example, but will only be depicted in the first figure. Hyperedges depicted as red and dashed are deleted.

The example will cover an entire iteration of the algorithm.

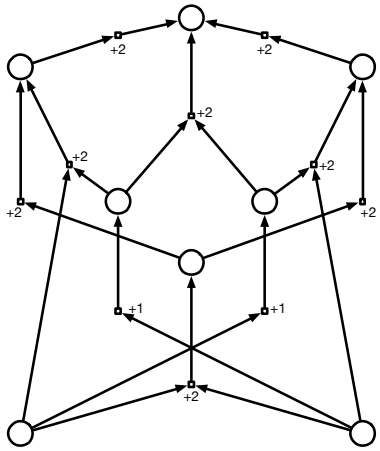


Figure 44: The hypergraph $\mathcal{H}$ with weights.

A method for finding the shortest path in the hypergraph $\mathcal{H}$ is applied, and the shortest path with a cost of 4, is shown in Figure 45.

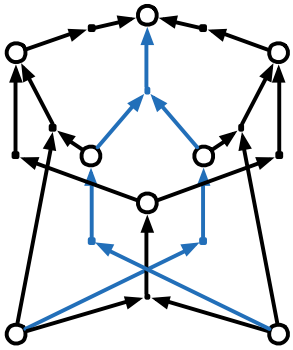
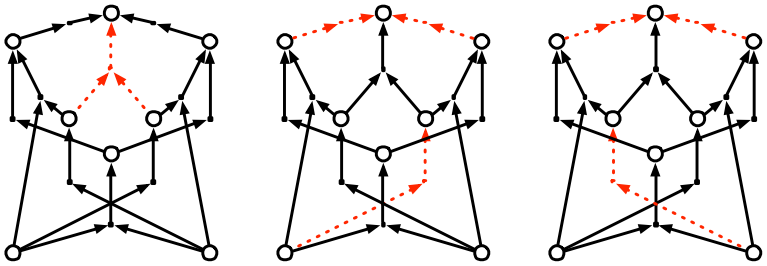


Figure 45: The best path π_1 found in $\mathcal{H}$.

The function `backwardsBranching` is applied on $(\mathcal{H}, \pi_1)$, which yields the hypergraphs $\mathcal{H}^1$, $\mathcal{H}^2$ and $\mathcal{H}^3$.

The shortest path is found in each of the hypergraphs. In the hypergraph $\mathcal{H}^1$ the shortest path is denoted π_2. In the hypergraphs $\mathcal{H}^2$ and $\mathcal{H}^3$ there are no paths.



$\mathcal{H}^1 : \{\pi_2\}$ $\mathcal{H}^2 : \emptyset$ $\mathcal{H}^3 : \emptyset$

The second shortest path is π_2 in $\mathcal{H}^1$ is depicted in Figure 46. All the red edges are deleted from the hypergraphs.

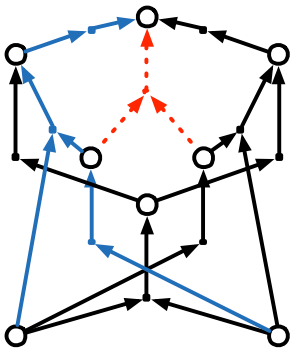
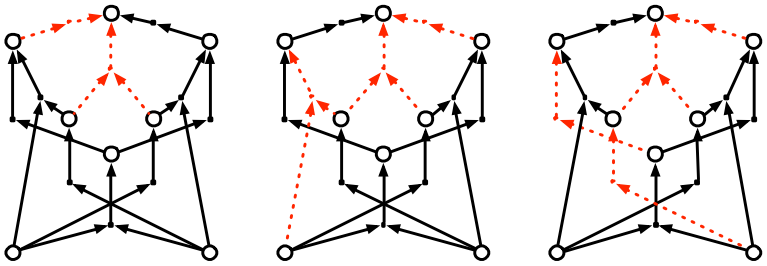


Figure 46: The second best path π_2 found in $\mathcal{H}^1$.

The algorithm will continue by calling `backwardsBranching` on $(\mathcal{H}^1, \pi_2)$, which would yield 3 new hypergraphs $\mathcal{H}^{11}$, $\mathcal{H}^{12}$ and $\mathcal{H}^{13}$.

Again the algorithm for finding the shortest path is applied to the hypergraphs. The path π_3 is found in $\mathcal{H}^{11}$ and π_4 is found in $\mathcal{H}^{12}$.



$\mathcal{H}^{11} : \{\pi_3\}$ $\mathcal{H}^{12} : \{\pi_4\}$ $\mathcal{H}^{13} : \emptyset$

The path π_3 is the third best path in $\mathcal{H}$. We cannot determine if π_4 is the forth best plan at this point. First we must apply `BackwardsBranching`

along π_3, this is left to the reader. See Figure 47 and Figure 48 for π_3 and π_4 respectively.

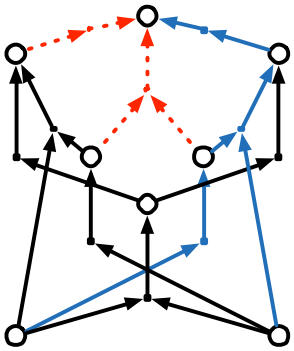


Figure 47: Third best path π_3 in $\mathcal{H}^{11}$.

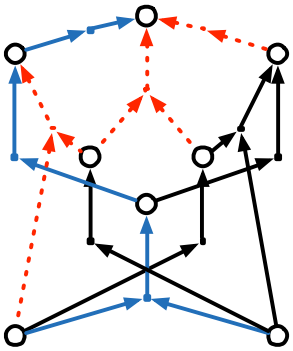


Figure 48: Shortest Path π_4 found in $\mathcal{H}^{12}$.

4.5 Results

In this section we will compare how robust the best plans are, when they are rated with another cost function. Phrased differently, we measure the cumulative distribution of two cost functions.

Comparing different cost functions

The first example is the molecule estrone, with the bondset specified in [23]. The HoSP is constructed with HoSP-Edge, and all plans are outputted.

We now plot how many of the x best plans with respect to Starting Material Weight (Weight) there also is the x best plans with respect to External Path Length (External) and vice versa for various x.

Two lists are created. One that is sorted with respect to Weight (list 1), and one which is sorted with respect to External (list 2).

First list 1 is clustered and each cluster is internally sorted with respect to External. The same thing is done for list two with respect to Weight.

For a given cluster of size x we check how many of the plans contained in the cluster is within the best x in the other list. An example could be that from the best 500 with respect to Weight, 400 of those is contained within the best 500 with respect to External. This would give the point $(500, 400)$.

The further away the points are from the line $x = y$, the less equal the two cost functions are.