University of Southern Denmark

Department of Mathematics
and Computer Science

Master Thesis – 31/12-2017

Computational Synthesis Planning Using Big Data

Computational Syntese planlægning ved hjælp af big data

Author:
Henrik Schulz

Supervisors:
Daniel Merkle

Abstract

Resumé

Contents

1 Acknowledgements

2 Introduction

kkk [2]

2.1 Overview

Hvad indeholder hver sektion??

3 Preliminaries

This section contains definitions that will be used throughout this paper. It is assumed that the reader have a basic understanding of graph theory.

Hypergraphs
A directed hypergraph h is a set of V of vertices and a set E of hyperedges, where each hyperedge $e = (T(e), H(e))$ is an ordered pair of non-empty multisets of vertices. The set $T(e)$ is denoted as the tail of the hyperedge and $H(e)$ is the head. If $|H(e)| = 1$ then the hyperedge is denoted as a B-hyperedge. If all edges in the hypergraph is B-hyperedges, then the graph is denoted a B-hypergraph. This paper will only consider hypergraphs that are B-hypergraphs. A hypergraph $H' = (V', E')$ is a subhypergraph of $H = (V, E)$ if $V' \subset V$ and $E' \subset E$.[3]

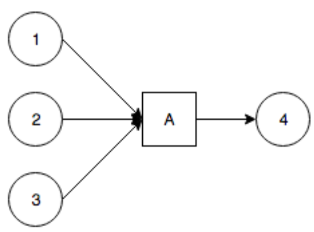

Fig. 1: Example of hyperedge A. $T(A) = \{1, 2, 3\}$, $H(A) = \{4\}$

Hyperpaths
A path P_{st} from s to t in a B-hypergraph is a sequence $P_{st} = \langle e_1, e_2, e_3, ..., e_q \rangle$ of B-hyperedges such that $s \in T(e_1)$ and $t = H(e_q)$ and $H(e_i) \in T(e_{i+1})$ for $i = 1..q-1$. Its length $|P_{st}|$ is the number q of hyperedges. If $t \in T(e_1)$, then P_{st} is a cycle. A hypergraph is acyclic if it does not contain any cycles. [3]

4 Finding The K-Best Synthetic Plans

This section describes how to find the K-Best paths of a hypergraph modifying an algorithm made by Jin Y. Yen. The section starts out by describing the work flow of the algorithm, and then proceeds to deal with the alterations that are needed to run the algorithm on a hypergraph.

4.1 Yen's Algorithm

Yen's algorithm is an algorithm that computes the K-shortest paths for a graph with non-negative edges. It was publish in 1971 and uses any shortest path algorithm to find the best path and then proceeds to find the $K - 1$ deviation of the best path. [4]
It starts out by finding the best path using a shortest path algorithm. Once the best path have been found it uses the path to find all the potential second best paths by fixing and removing edges in the graph.
By using the same first vertex as the original path but removing the first edge, it forces the shortest path algorithm to take another route through the graph and thereby creating a potential second best path. This is added to the list of potential paths and the algorithm can continue to derive other paths from the best path. By fixing the first edge in the previous best plan, Yen's algorithm forces the shortest path algorithm to take the first edge which it now shares with the best path. However, now the algorithm have removed the second edge from the original path and once again forces the shortest path algorithm to find alternative routes. This process is then repeated until we reach the next to last vertex in the best path.
By sorting the list of potential paths, it has the second best path at the start of the list and it can add it to the final list of best path. The algorithm then repeats on the second best path to find the third best path. This is done until all K-best path have been found or there are no more paths to find.

4.2 Yen's Algorithm On Hypergraphs

We use the principles from Yen's algorithm to make our own algorithm that will work on hypergraphs. To handle the problem of generating all derived paths from our best path in our hypergraph, we use a method called Backwards-Branching. [3] [5] [6]

Algorithm 1: Backwards Branching for B-Hypergraph

```
function Back−Branch(H,π)
    B=∅
    for i = 1 to q do
        Let H^i be a new hypergraph
        H^i.V = H.V
        // Remove hyperarc from H
        H^i.E = H.E \{π.p(v_i)}
        // Fix Back tree
        for j = i+1 to q do
            H^i.BS(vj) = \{π.p(v_j)\}
        B = B ∪ \{H^i\}
    return B
```

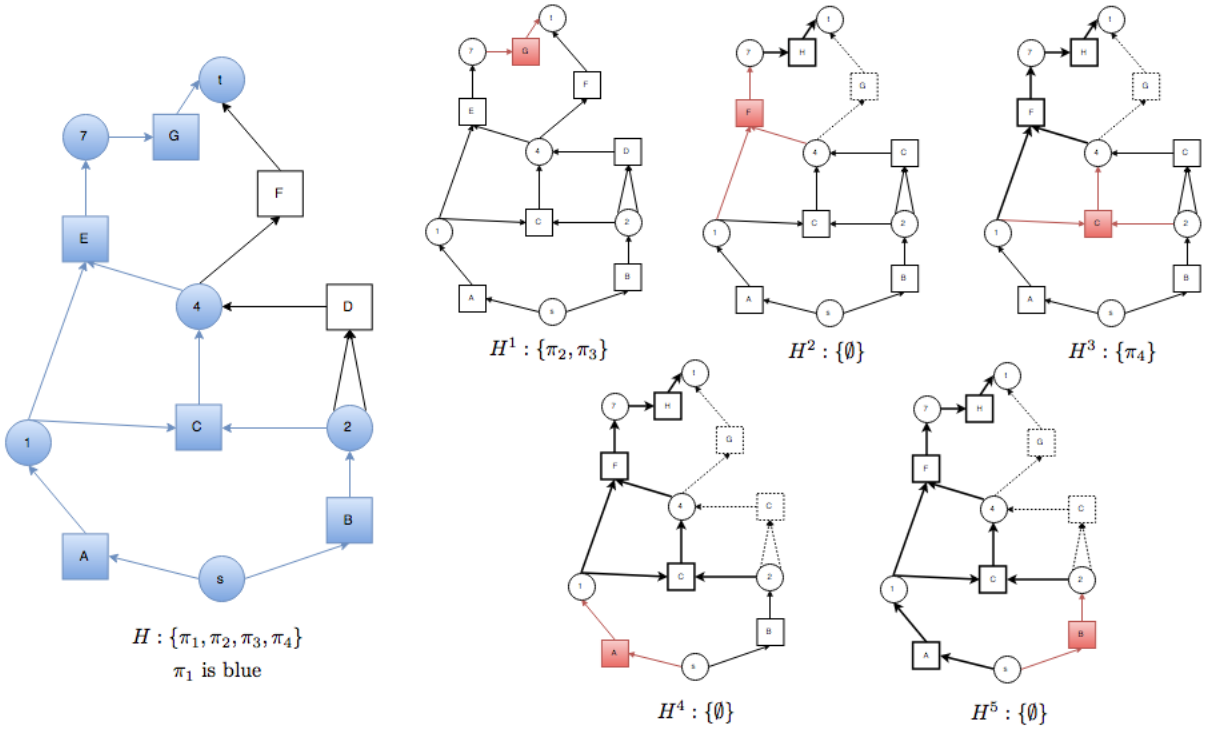

Fig. 2: An example of branching for hypergraphs. H is the original hypergraph. The vertices and hyperedges marked with blue are part of the best path. The rest of the figure illustrates the backwards branching. Each of the smaller figures shows a hypergraph H^i and how it is created from H. Dotted hyperedges and red heyperedges are not part of the hypergraph, but have been deleted due to branching. Hyperedges in bold are fixed hyperedges. When hyperedges are fixed it leads to other hyperedges being deleted (dotted). The caption beneath each hypergraph represents the possible paths that are available in the given hypergraph.

However, this algorithm have a problem when working on a larger hypergraph. It demands that each time we make alterations on the hypergraph we have to make a copy, H^i, of the graph, H, with the exception of the hyperedges that is removed when fixing the back tree and removing $\pi.p(v_i)$.
This could easily work for smaller graphs, but if we use this on the hypergraph that we generate from the beilstein database, we would have to copy a graph of multiple GigaBytes.
To handle this problem I came up with the idea of creating an overlay for the graph instead of copying it. The overlay would work as an transparent on top of the original graph, stating which edges still were accessible. This is done by creating a *vector*<*bool*> which has a length of R, where R is the number of reactions. Normally a reaction would contain at least 24 bytes of data:

- 2x ints of 4 bytes each

- 1x double of 8 bytes
- 1x *vector<int>* head of length one of at least 4 bytes
- 1x *vector<int>* tail of length N (number of educts) of at least 4 bytes

This can be reduced dramaticly by using the *vector<bool>*, since c++ only uses 1 bit per boolean in the vector instead of the regular 1 byte per boolean.[7] This means if working on a hypergraph with 40 million reactions, we would be able to create an overlay using 5 MB of space per alternated graph, instead of copying a hypergraph were the reactions alone uses at least 960 MB per copy. As the figure below shows, we never change or remove anything on the hypergraph. We simply create the following overlay:

Reaction	A	B	C	D
Usable	true	true	true	false
Bit Representation	1	1	1	0

And then when trying to use an edge, we ask: "Does overlay at reaction A exist?". If yes, you can use it. If no, the edge have been "removed", and therefore cannot be used.

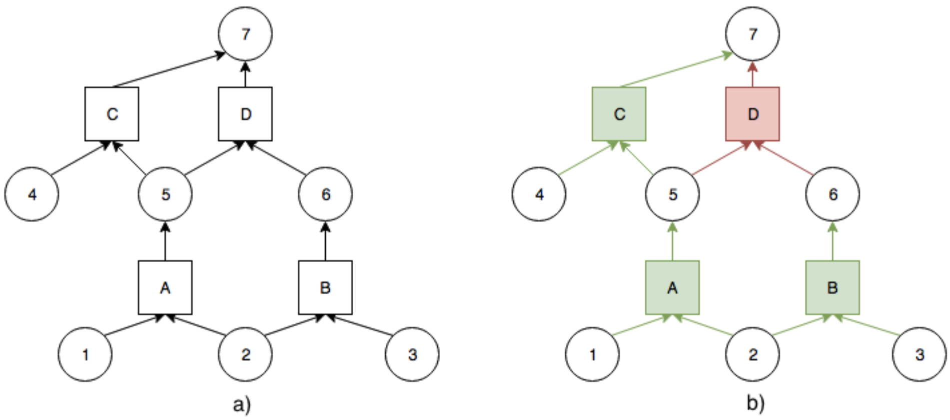

Fig. 3: a) The original hypergraph. b) The original hypergraph, but using the overlay. If the reaction is green it is still usable. If red then it have been "removed" from the hypergraph.

This of course means that the algorithm for back-branch have to be changed accordingly. Instead of the hypergraph as input we now give it an overlay. This overlay is changed so it fits with the new layout of the graph. Instead of deleting hyperedges in the copy, we now simply changes the boolean at the index of the hyperedge.id. True if we should add the hyperedge and false when we want to "remove" a hyperedge.

Algorithm 2: Backwards Branching for B-Hypergraph using overlay

function BackwardsBranching(π, Overlay)

```
    List B = ∅
    // q = Path length
    for i = 1 to q do
        //remove i'th hyperedge from Path in overlay
        Set Overlay[π[i]] to false
        //fix the backtree
        for j = i downto 1  do
            vertex C <− π[j].head
            for each hyperedge into C
                Set Overlay[reaction.id] to false
            Set Overlay[π[j]] to true
        endfor
        B = B ∪ {Overlay}
    endfor
    return B
endfunction
```

Now that we have the branching in place are we able to construct an algorithm that are similar to Yen's algorithm but can run on hypergraphs. As input it takes a start vertex, a goal vertex, and an integer K, where K is the number of best paths we want to find.
It creates a heap, L, and a list, A, which will contain the K-best paths once the algorithm is finished. It then finds the best path using a shortest path algorithm and inserts it into the heap. Inside the loop it extracts the best path found from the heap and performs a backward branching, and finds all possible paths in the branches. If there is a path from s to t, then this path is added to the heap. The algorithm either terminates if the heaps is empty (No more paths available) or once it have found the K-best paths.

Algorithm 3: K-Shortest Paths Algorithm in B-Hypergraph

```
function YenHyp(s, t, K)
    L = new heap with elements (overlay, π)
    A = List of shortest paths
    //(Graph is default overlay (all true))
    π = shortestPath(Graph, s,t)
    Insert (Graph, π) into L
    for k = 1 to K do
        if L = ∅
            Break
        endif
        (Overlay′, π′) = L.pop
        add π′ to A
        for all Overlay^i in BackwardBranching((Overlay′,π′)) do
            π^i = shortestPath(Overlay^i, s, t)
            if π^i is complete
                Insert( H^i, π^i) into L
            endif
        endfor
    endfor
    return A
```

```
endfunction
```

5 Shortest Path

This section describes two different approaches to the shortest path problem in a hypergraph. A dynamic approach proposed by Carsten Grønbjerg Lützen and Daniel Fentz Johansen [6] and a Dijkstra inspired approach proposed by Lars Relund Nielsen, Kim Allan Andersen and Daniele Pretolani [5].

5.1 Dynamic Approach

kkk [6]
kkkk [3]

5.1.1 Approach

Algorithm 4: Dynamic programming for finding the best path

```
function Min(V)
    if(V) is starting material
        return Cost of V
    endif
    mincost <− inf
    for all e ∈ BS(V) do
        cost <− cost of e
        for all u ∈ Tail(e) do
            cost <− cost + Min(u)
        endfor
        if mincost ≤ cost
            mincost <− cost
            V.minedge <− e
        endif
    endfor
    return mincost
endfunction
```

5.1.2 Testing

5.1.3 Problems

5.2 Nielsens Algorithm

k [5]

5.2.1 Approach

Algorithm 5: Dynamic programming for finding the best path

Initialization: Set W(u) = inf $\forall u \in V$, $k_j = 0 \forall e_j \in E$,$Q = \{s\}$ and $W(s) = 0$
function SBT−Dijkstra
    while $(Q = \emptyset)$ **do**
        select and remove $u \in Q$ such that $W(u) = min\{W(x)|x \in Q\}$
        for $(e_j \in FS(u))$ **do**
            $k_j < -kj + 1$
            if $(k_j = |T(e_j)|)$
                v <− h(e_j)
                if (W(v) > w(e_j) + F(e_j))
                    if $(v \notin Q$
                        Q <− Q ∪ \{v\}
                    endif
                W(v) <− w(e_j)+ F(e_j)
                p(v) <− e_j
                endif
            endif
        endfor
    endwhile
endfunction

5.2.2 Testing

5.2.3 Problems

5.2.4 Optimizing

6 Work With Beilstein Data

6.1 The Graph

6.2 Testing

6.2.1 Strychnine

6.2.2 Compound 2

6.2.3 Compound 3

kkkk [1]

7 Konklusion

Books

[1] R. W. Hoffmann, *Elements of Synthesis Planning*. Springer Berlin Heidelberg, 2009.

Articles

[2] S. Szymkuc, E. P. Gajewska, T. Klucznik, K. Molga, P. Dittwald, M. Startek, M. Bajczyk, and B. A. Grzybowski, “Computer-assisted synthetic planning: The end of the beginning”, *Angewandte Chemie International Edition*, no. 55, pp. 5904–5937, 2016.

[3] R. Fagerberg, C. Flamm, R. Kianian, D. Merkle, and P. F. Stadler, “Finding the K best synthesis plans”, 2017, Unpublished Article.

[4] J. Y. Yen, “Finding the K shortest loopless paths in a network”, *Management Science*, vol. 17, no. 11, pp. 712–716, Jul. 1971.

[5] L. R. Nielsen, K. A. Andersen, and D. Pretolani, “Finding the K shortest hyperpaths: Algorithms and applications”, 2002.

Other

[6] C. G. Lützen and D. F. Johansen, “A computational and mathematical approach to synthesis planning”, Master’s thesis, University of Southern Denmark, 2015.

[7] Sep. 2017. [Online]. Available: `http://en.cppreference.com/w/cpp/container/vector_bool`.

8 Appendiks