Dynamic Programming Strikes Back

February 18, 2025

1 Introduction

In this paper, we introduct `DPhyp`. Experiments will shwo that it is highly superior to existing approaches.

2 Hypergraphs

2.1 Definitions

Definition 2.1 (Hypergraph)**.** A **hypergraph** is a pair $H = (V, E)$ s.t.

1. V is a non-empty set of nodes
2. E is a set of hyperedges, where a **hyperedge** is an unordered pair (u, v) of non-empty subsets of V ($u \subset V$ and $v \subset V$) with the additional condition that $u \cap v = \emptyset$

We call any non-empty subset of V a **hypernode**. We assume that the nodes in V are totally ordered via an (arbitrary) relation $\prec$. The ordering on nodes is important for our algorithm.

A hyperedge (u, v) is **simple** if $|u| = |v| = 1$. A hypergraph is **simple** if all its hyperedges are simple.

In our context, the nodes of hypergraphs are relations and the edges are abstractions of join predicates. Consider a join predicate of the form

$$R_1.a + R_2.b + R_3.c = R_4.d + R_5.e + R_6.f$$

This predicate will result in a hyperedge $(\{R_1, R_2, R_3\}, \{R_4, R_5, R_6\})$.

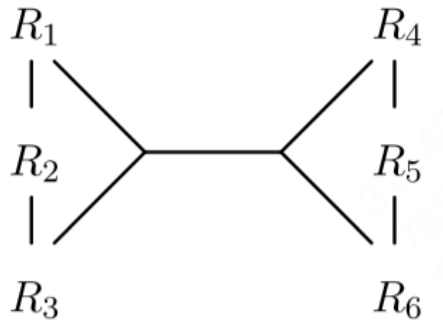

Figure 1: Sample hypergraph

Fig 1 contains an example of a hypergraph. The set V of nodes is $V = \{R_1, \dots, R_6\}$. Concerning the node ordering, we assume that $R_i \prec R_j \Leftrightarrow i < j$. There are simple edges $(\{R_1\}, \{R_2\})$, $(\{R_2\}, \{R_3\})$, $(\{R_4\}, \{R_5\})$ nad $(\{R_5\}, \{R_6\})$.

Note that is possible to rewrite the above complex join predicate. For example, it is equivalent to

$$R_1.a + R_2.b = R_4.d + R_5.e + R_6.f - R_3.c$$

This leads to a hyperedge $(\{R_1, R_2\}, \{R_3, R_4, R_5, R_6\})$.

Definition 2.2 (Subgraph)**.** Let $H = (V, E)$ be a hypergraph and $V' \subseteq V$ a subset of nodes. The **node induced subgraph** $G|_{V'}$ of G is defined as $G|_{V'} = (V', E')$ with $E' = \{(u, v) \mid (u, v) \in E, u \subseteq V', v \subseteq V'\}$. The node ordering on V' is the restriction of the node ordering of V.

Definition 2.3 (Connected)**.** Let $H = (V, E)$ be a hypergraph. H is connected if $|V| = 1$ or if there exists a partitioning V', V'' of V and a hyperedge $(u, v) \in E$ s.t. $u \subseteq V'$, $v \subseteq V''$ and both $G|_{V'}$ and $G|_{V''}$ are connected.

Let $H = (V, E)$ is a hypergraph and $V' \subseteq V$ is a subset of the nodes s.t. the node-induced subgraph $G|_{V'}$ is connected, then we call V' a **connected subgraph** or **csg** for short. The number of connected subgraphs is important for dynamic programming: it directly corresponds to the number of entries in the dynamic programming table. If a node set $V'' \subseteq (V \setminus V')$ induces a connected subgraph $G|_{V''}$, we call V'' a **connected complement** of V' or **cmp** for short.

Within this paper, we will assume that all hypergraphs are connected. This way, we can make sure that no cross products are needed. However,

when dealing with hypergraphs, this condition can easily be assured by adding according hyperedges: for every pair of connected components, we can add a hyperedge whose hypernodes contain exactly the relations of the connected components. By considering these hyperedges as $\bowtie$ operators with selectivity 1, we get an equivalent connected hypergraph

2.2 Csg-cmp-pair

Definition 2.4 (Csg-cmp-pair)**.** Let $H = (V, E)$ be a hypergraph and S_1, S_2 two subsets of V s.t. $S_1 \subseteq V$ and $S_2 \subseteq (V \setminus S_1)$ are a connected subgraph and a connected complement. If there further exists a hyperedge $(u, v) \in E$ s.t. $u \subseteq S_1$ and $v \subseteq S_2$, we call (S_1, S_2) a **csg-cmp-pair**.

We will restrict the enumeration of csg-cmp-pairs to those (S_1, S_2) which satisfy the condition that $\min(S_1) \prec \min(S_2)$.

Obviously, in order to be correct, any dynamic programming algorithm has to consider all csg-cmp-pairs. Further, only these have to be considered. Thus, the minimal number of cost function calls of any dynamic programming algorithm is exactly the number of csg-cmp-pairs for a given hypergraph. Note that the number of connected subgraphs is far smaller than the number of csg-cmp-pairs.

The problem now is to enumerate the csg-cmp-pairs efficiently and in an order acceptable for dynamic programming. The latter can be expressed more specifically. Before enumerating a csg-cmp-pair (S_1, S_2), all csg-cmp-pairs S_1', S_2' with $S_1' \subseteq S$ and $S_2' \subseteq S_2$ have to be enumerated.

2.3 Neighborhood

The main idea to generate csg-cmp-pairs is to incrementally expand connected subgraphs by considering new nodes in the **neighborhood** of a subgraph. Informally, the neighborhood $N(S)$ under an exclusion set X consists of all nodes reachable from S that are not in X. We derive an exact definition below

When choosing subsets of the neighborhood for inclusion, we have to treat a hypernode as a single instance: either all of its nodes are inside an enumerated subset or none of them. Since we want to use the fast subset enumeration procedure introduced by Vance and Maier, we must have a single bit representing a hypernode and also single bits for relations occurring in simple edges. Since these may overlap, we are constained to choose one unique representative of every hypernode occuring in a hyperedge. We choose the node $\min(S)$.

Let S be a current set, which we want to expand by adding further relations. Consider a hyperedge (u, v) with $u \subseteq S$. Then we will add $\min(v)$ to the neighborhood of S. However, we have to make sure that the missing elements of v, i.e., $v \setminus \min(v)$, are also contained in any set emitted. We thus define

$$\overline{\min}(S) = S \setminus \min(S)$$

We define the set of non-subsumed hyperedges as the minimal subset $E \downarrow$ of E s.t. for all $(u, v) \in E$ there exists a hyperedge $(u', v') \in E \downarrow$ with $u' \subseteq u$ and $v' \subseteq v$. Additionally, we make sure that none of the nodes of a hypernode are contained in a set X, which is to be excluded from neighborhood considerations. We thus define a set containing the **interesting hypernodes** for given sets S and X. We do so in two steps.

1. Collect the potentially interesting hypernodes into a set $E \downarrow' (S, X)$ and then minimize this set to eliminate subsumed hypernodes:

$$E \downarrow' (S, X) = \{v \mid (u, v) \in E, u \subseteq S, v \cap S = \emptyset, v \cap X = \emptyset\}$$

2. Define $E \downarrow (S, X)$ to be the minimal set of hypernodes s.t. for all $v \in E \downarrow' (S, X)$ there exists a hypernode v' in $E \downarrow (S, X)$ s.t. $v' \subseteq v$.

We now define the **neighborhood** of a hypernode S, given a set of excluded nodes X, to be:

$$\mathcal{N}(S, X) = \bigcup_{v \in E\downarrow(S,X)} \min(v)$$

For hypergraph in Fig 1 and with $X = S = \{R_1, R_2, R_3\}$, we have

$$E \downarrow' (S, X) = \{\{R_4, R_5, R_6\}\}$$
$$E \downarrow (S, X) = \{\{R_4, R_5, R_6\}\}$$
$$\mathcal{N}(S, X) = \{R_4\}$$

3 The algorithm

High level:

1. Constructs ccps (csg-cmp-pair) by enumerating connected subgraphs from an increasing part of the query graph
2. both the primary connected subgraphs and its connected complement are created by recursive graph traversals

3. during traversal, some nodes are **forbidden** to avoid creating duplicates

4. connected subgraphs are increased by following edges to neighboring nodes. For this purpose hyperedges are interpretaed as $n : 1$ edges, leading from n of one side to one canonical nodes of the other side.

Summarizing the above, the algorithm traverses the graph in a fixed order and recursively produces larger connected sub- graphs.

We give the implementation of our join ordering algorithm for hypergraphs by means of pseudocode for member functions of a class `DPhyp`.

The whole algorithm is distributed over five subroutines.

- The top-level routine `Solve` initializes the dynamic programming table with access plans for single relations and then calls `EmitCsg` and `EnumerateCsgRec` for each set containing exactly one relation.

- The member function `EnumerateCsgRec` is responsible for enumerating connected subgraphs. It does so by calculating the neighborhood and iterating over each of its subset.

- For each such subset S_1 , `EnumerateCsgRec` calls `EmitCsg`. This member function is responsible for finding suitable complements.

- `EmitCsg` does so by calling `EnumerateCmpRec`, which recursively enumerates the complements S_2 for the connected subgraph S_1 found before.

- The pair (S_1, S_2) is a csg-cmp-pair. For every such pair, `EmitCsgCmp` is called. Its main responsibility is to consider a plan built up from the plans for S_1 and S_2 .