Order in which Subtrees are generated

The ordering in which subtrees are generated does not matter as long as the following condition is not violated:

> *Let S be a subset of $\{R_1, \ldots, R_n\}$. Then, before a join tree for S can be generated, the join trees for all relevant subsets of S must already be available.*

- *relevant* means that they are valid subproblems by the algorithm
- usually this means connected (no cross products)

Generation in Integer Order

000	$\{\}$
001	$\{R_1\}$
010	$\{R_2\}$
011	$\{R_1, R_2\}$
100	$\{R_3\}$
101	$\{R_1, R_3\}$
110	$\{R_2, R_3\}$
111	$\{R_1, R_2, R_3\}$

- can be done very efficiently
- set representation is just a number

Generating Linear Trees (3)

DPsubLinear(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal left-deep (right-deep, zig-zag) join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall 1 < i \leq 2^n - 1$ **ascending** $\{$
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for $\forall R_j \in S$ $\{$
    if $\neg$cross products $\wedge \neg S \setminus \{R_j\}$ connected to R_j **continue**
    $p_1 = B[S \setminus \{R_j\}], p_2 = B[\{R_j\}]$
    if $p_1 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  $\}$
$\}$
return $B[\{R_1, \ldots, R_n\}]$

Generating Bushy Trees

- a bushy tree T with $|T| > 1$ has the form $T_1 \bowtie T_2$, with $|T| = |T_1| + |T_2|$
- if T is optimal, both T_1 and T_2 must be optimal too
- basic strategy: find the optimal T by joining all pairs of optimal T_1 and T_2

Generating Bushy Trees (2)

DPsize(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall 1 < s \leq n$ **ascending** $\{$
  for $\forall S_1, S_2 \subset R : |S_1| + |S_2| = s$ $\{$
    if ($\neg$cross products $\wedge \neg S_1$ connected to S_2) $\vee$ ($S_1 \cap S_2 \neq \emptyset$) **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S_1 \cup S_2] = \epsilon \vee C(B[S_1 \cup S_2]) > C(P)$
      $B[S_1 \cup S_2] = P$
  $\}$
$\}$
return $B[\{R_1, \ldots, R_n\}]$

Generating Bushy Trees (3)

DPsub(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall 1 < i \leq 2^n - 1$ **ascending** $\{$
  $S = \{R_j \in R | (\lfloor i/2^{j-1} \rfloor \, mod \, 2) = 1\}$
  for $\forall S_1 \subset S, S_2 = S \setminus S_1$ $\{$
    if $\neg$cross products $\wedge \neg S_1$ connected to S_2 **continue**
    $p_1 = B[S_1], p_2 = B[S_2]$
    if $p_1 = \epsilon \vee p_2 = \epsilon$ **continue**
    $P =$ CreateJoinTree(p_1, p_2);
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  $\}$
$\}$
return $B[\{R_1, \ldots, R_n\}]$

Efficient Subset Generation

If we use integers as set representation, we can enumerate all subsets of S as follows:

$S_1 = S\&(-S)$
do $\{$
  $S_2 = S - S_1$
  // Do something with S_1 and S_2
  $S_1 = S\&(S_1 - S)$
$\}$ **while** $(S_1 != S)$

- enumerates all subsets except $\emptyset$ and S itself
- very fast

Remarks

- DPsize/DPsizeLinear does not really test for $p_1 = \epsilon$
- it keeps a list of plans for a given size
- candidates can be found very fast
- ensures polynomial time in some cases (we will look at it again)
- DPsub/DPsubLinear is faster if the problem is not polynomial, though

Memoization

- top-down formulation of dynamic programming
- recursive generation of join trees
- memoize already generated join trees to avoid duplicate work
- easier code
- sometimes more efficient (more knowledge, allows for pruning)
- but usually slower than dynamic programming

Memoization (2)

Memoization(R)
Input: a set of relations $R = \{R_1, \ldots, R_n\}$ to be joined
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
MemoizationRec(B, R)
return $B[\{R_1, \ldots, R_n\}]$

- initializes the DP table and triggers the recursive search
- main work done during recursion

Memoization (3)

MemoizationRec(B,S)
Input: a DP table B and a set of relations S to be joined
Output: an optimal bushy join tree for the subproblem
if $B[S] = \epsilon$ {
  for $\forall S_1 \subset S, S_2 = S \setminus S$
    p_1 =MemoizationRec(B, S_1), p_2 =MemoizationRec(B, S_2)
    P=CreateJoinTree(p_1, p_2)
    if $B[S] = \epsilon \vee C(B[S]) > C(P)$ $B[S] = P$
  }
}
return $B[S]$

- checks for connectedness omitted

Dynamic Programming - Connected Subgraphs

- DP a very versatile strategy
- common usage scenario: bushy, no cross produts
- DPsize and DPsub support it, of course, but not optimal
- enumeration order does not consider the query graph
- many pairs have to be pruned due to conectedness
- especially bad for DPsub

Solution: consider the query graph structure during DP enumeration [5]

Asymptotic Search Space

DPsize:

- organize DP by the size of the join tree
- problem: only few DP slots, many pairs considered

good algorithm for chains, very bad for cliques:

	chains	cycles	stars	cliques
pairs	$O(n^4)$	$O(n^4)$	$O(4^n)$	$O(4^n)$

DPsub:

- organize DP by the set of relations involved
- problem: always 2^n DP slots, fixed enumeration

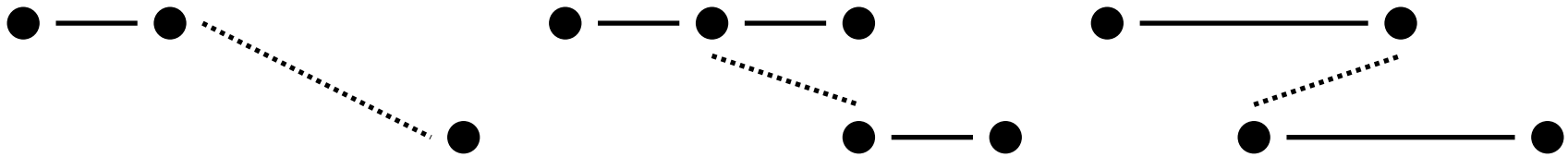
good algorithm for cliques, but adapts badly:

	chains	cycles	stars	cliques
pairs	$O(2^n)$	$O(n2^n)$	$O(3^n)$	$O(3^n)$

Observation

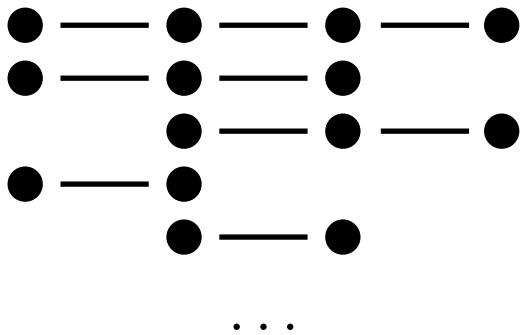
DPsize and DPsub generate many pairs that are pruned anyway (connectedness, overlap).

Typical pruned pairs (chain with 4 relations):



not connected | not disjoint | invalid subproblems

last example ⇒ every join partner must be a connected subgraph:



Graph Theoretic Approach

- reformulation as graph theoretic problem:
- enumerate all connected subgraphs of the query graph
- for each subgraph enumerate all other connected subgraphs that are disjoint but connected to it
- each connected subgraph - complement pair (ccp) can be joined
- enumerate them suitable for DP $\Rightarrow$ DP algorithm

algorithm adapts naturally to the graph structure:

	chains	cycles	stars	cliques
pairs	$O(n^3)$	$O(n^3)$	$O(n2^n)$	$O(3^n)$

Lohman et al: #ccp is a lower bound for all DP enumeration algorithms

DP Algorithm using Connected Subgraphs

If we can efficiently enumerate all connected subgraphs/connected complement pairs, the resulting DP algorithm is:

DPccp(R)
Input: a connected query graph with relations $R = \{R_0, \ldots, R_{n-1}\}$
Output: an optimal bushy join tree
$B =$ an empty DP table $2^R \rightarrow$ join tree
for $\forall R_i \in R$
  $B[\{R_i\}] = R_i$
for $\forall$ csg-cmp-pairs (S_1, S_2), $S = S_1 \cup S_2$ {
  $p_1 = B[S_1]$, $p_2 = B[S_2]$
  $P =$ CreateJoinTree(p_1, p_2);
  if $B[S] = \epsilon \vee C(B[S]) > C(P)$
    $B[S] = P$
}
return $B[\{R_0, \ldots, R_{n-1}\}]$

The main problem is enumerating the pairs.

Effect on Search Space

Absolute number of generated pairs

	Chain			Star		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	20	84	73	32	130	110
10	165	3,962	1,135	2,304	38,342	57,888
15	560	130,798	5,628	114,688	9,533,170	57,305,929
20	1,330	4,193,840	17,545	4,980,736	2,323,474,358	59,892,991,338
	Cycle			Clique		
n	DPccp	DPsub	DPsize	DPccp	DPsub	DPsize
2	1	2	1	1	2	1
5	40	140	120	90	180	280
10	405	11,062	2,225	28,501	57,002	306,991
15	1,470	523,836	11,760	7,141,686	14,283,372	307,173,877
20	3,610	22,019,294	37,900	1,742,343,625	3,484,687,250	309,338,182,241

Enumerating Connected Subgraphs

- two steps: enumerate all connected subgraphs, enumerate disjoint but connected subgraphs for a given one $\Rightarrow$ pairs
- enumerate all pairs, enumerate no duplicates, enumerate for DP
- if (a, b) is enumerated, do not enumerate (b, a)
- requires total ordering of connected subgraphs
- preparation: label nodes breadth-first from 0 to $n - 1$

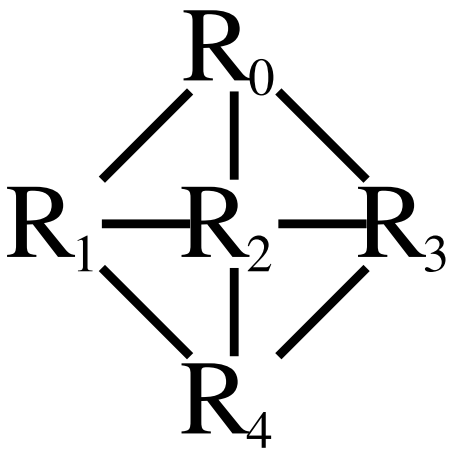
Preliminaries, given query graph $G = (V, E)$:

$$\begin{aligned} V &= \{v_0, \ldots, v_{n-1}\} \\ \mathcal{N}(V') &= \{v' | v \in V' \wedge (v, v') \in E\} \\ \mathcal{B}_i &= \{v_j | j \leq i\} \end{aligned}$$

Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

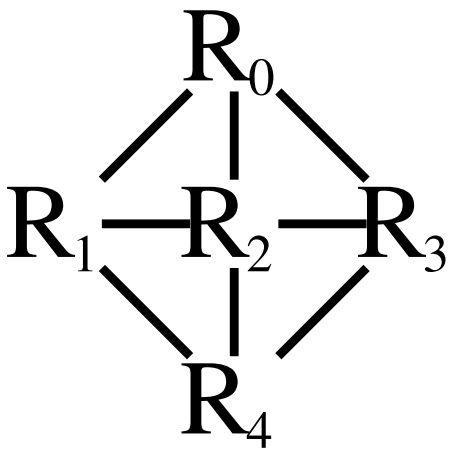


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Choose all nodes as enumeration start node once

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

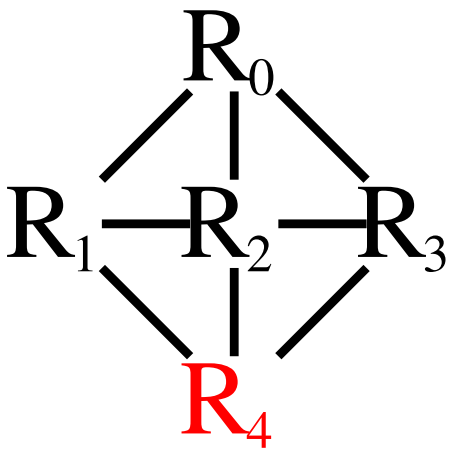


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

First emit only the node itself as subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

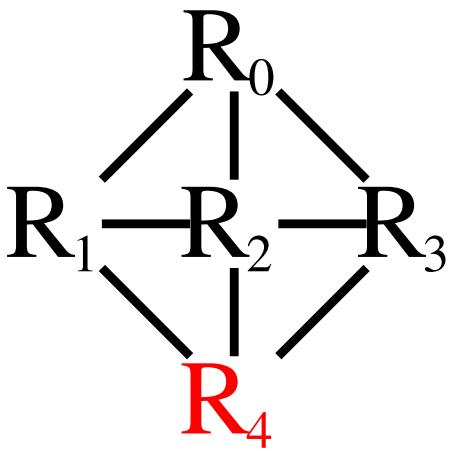


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Then enlarge the subgraph recursively

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

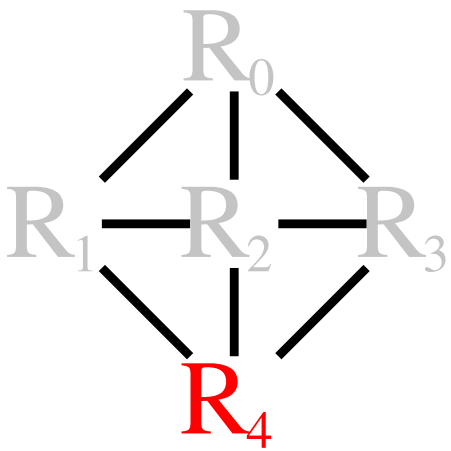


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Prohibit nodes with smaller labels. Thus the set of valid nodes increases over time

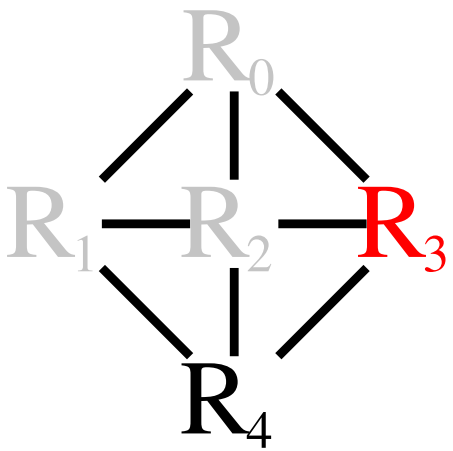
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

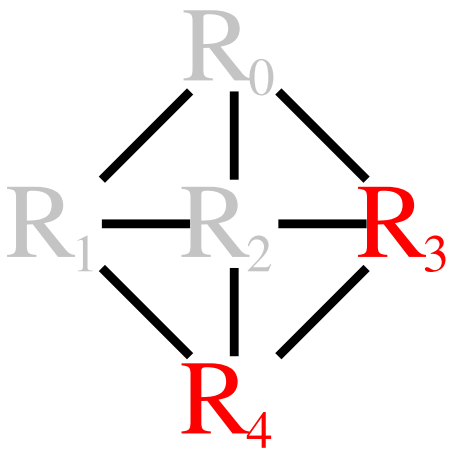
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

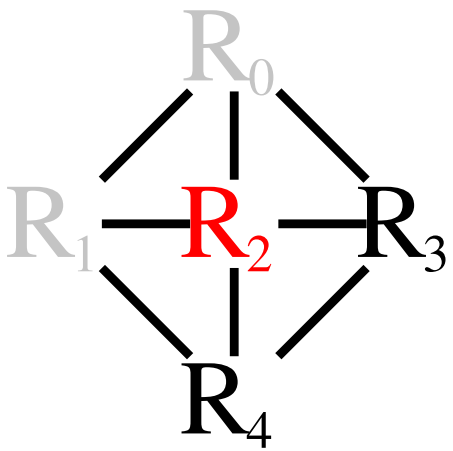
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}



Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

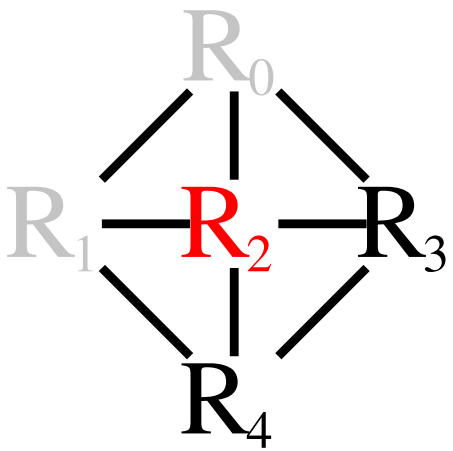


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

In each recursion, find all neighboring nodes that are not prohibited

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

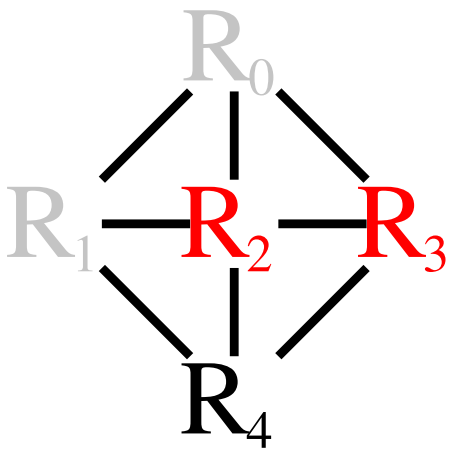


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Add all combinations to the subgraph and emit the new subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

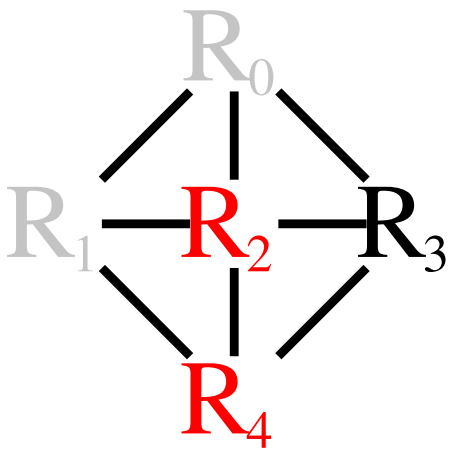


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

Add all combinations to the subgraph and emit the new subgraph

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

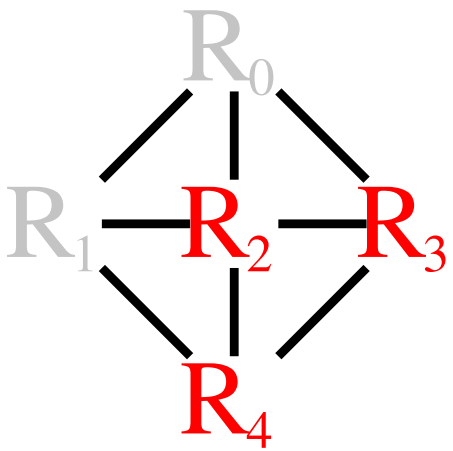


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

Add all combinations to the subgraph and emit the new subgraph

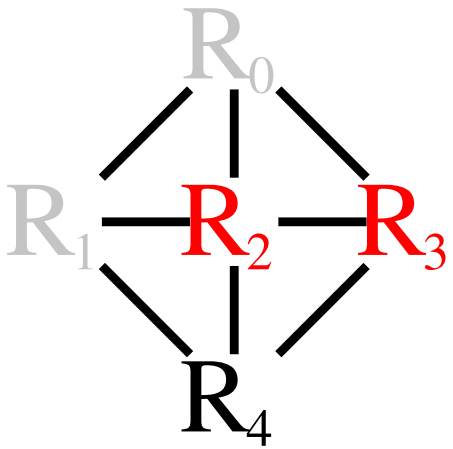


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

Then, add all combinations to the subgraph and increase recursively

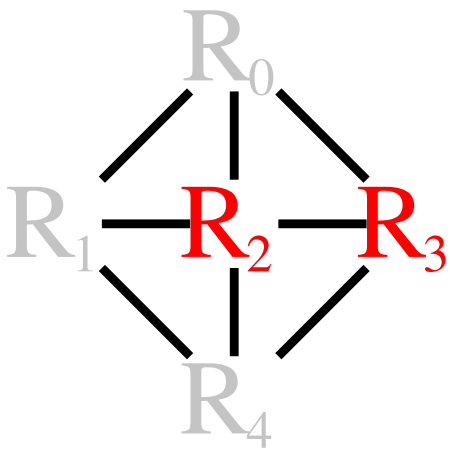


Enumerating Connected Subgraphs (2)

EnumerateCsg(G)
for all $i \in [n-1, \ldots, 0]$ **descending** {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $\mathcal{B}_i$);
}

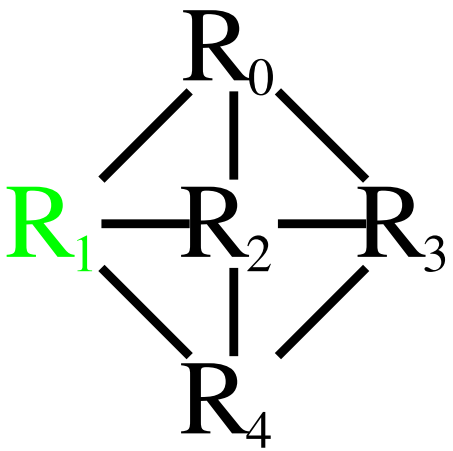
EnumerateCsgRec(G, S, X)
$N = \mathcal{N}(S) \setminus X$;
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    emit $(S \cup S')$;
}
for all $S' \subseteq N$, $S' \neq \emptyset$, enumerate subsets first {
    EnumerateCsgRec(G, $(S \cup S')$, $(X \cup N)$);
}

The neighborhood is prohibited during recursion, preventing duplicates



Enumerating Complementary Subgraphs

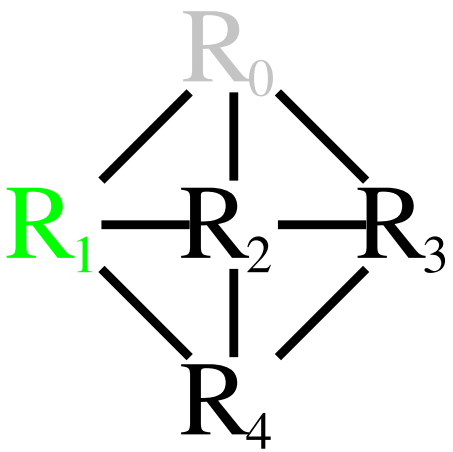
EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

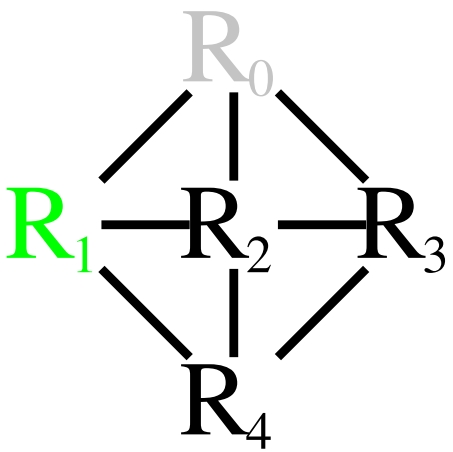
Prohibit all nodes that will be start nodes later on and the primary subgraph



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

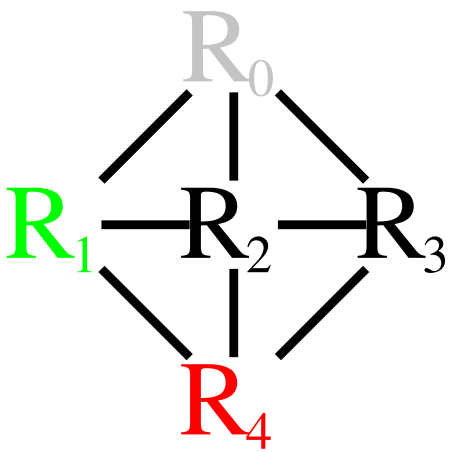
Find all neighboring nodes that are not prohibited



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

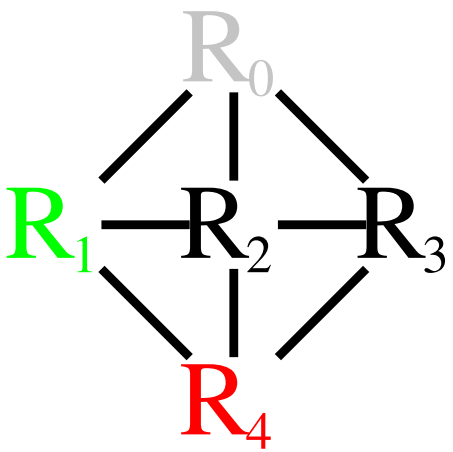
Consider each of the nodes



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

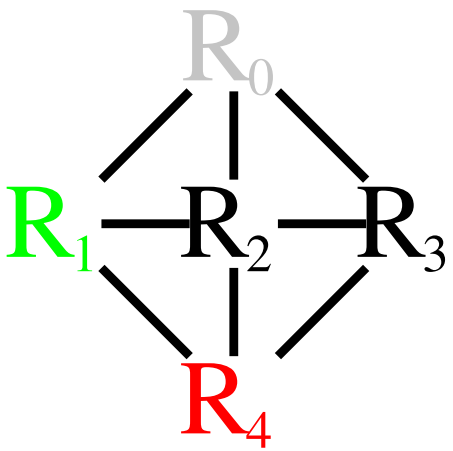
Choose the node as complementary subgraph and emit it



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

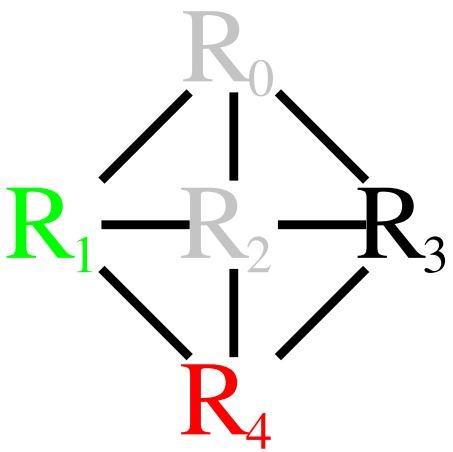
Recursively increase the subgraph re-using EnumerateCsgRec



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

Again prohibit nodes with a smaller label to prevent duplicates



Enumerating Complementary Subgraphs

EnumerateCmp(G,S_1)
$X = \mathcal{B}_{\min(S_1)} \cup S_1$;
$N = \mathcal{N}(S_1) \setminus X$;
for all ($v_i \in N$ by descending i) {
    emit $\{v_i\}$;
    EnumerateCsgRec(G, $\{v_i\}$, $X \cup (\mathcal{B}_i \cap N)$);
}

- EnumerateCsg+EnumerateCmp produce all ccp
- resulting algorithm DPccp considers exactly #ccp pairs
- which is the lower bound for all DP enumeration algorithms

Remarks

- DPsize is good for chains, DPsub for cliques
- implementation of DPccp is more involved
- each enumeration step must be fast (ideally $O(1)$, at most $O(n)$, where n is the number of relations)
- but benefits are huge
- DPccg adopts to query graph structure
- considers minimal number of pairs
- especially for "in-between queries" (e.g. stars) much faster