On Canonical Forms for Frequent Graph Mining

Christian Borgelt

School of Computer Science
Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, D-39106 Magdeburg, Germany

Email: borgelt@iws.cs.uni-magdeburg.de
http://fuzzy.cs.uni-magdeburg.de/~borgelt/

Overview

- **Canonical Form Pruning in Frequent Item Set Mining**
  - Searching the Subset Lattice / Types of Search Tree Pruning
  - Structural Pruning in Frequent Item Set Mining
- **Canonical Form Pruning in Frequent Graph Mining**
  - Constructing Spanning Trees (depth-first vs. breadth-first)
  - Edge Sorting Criteria (sort edges into insertion order)
  - Construction of Code Words
  - Restricted Extensions (rightmost vs. maximum source)
  - Checking for Canonical Form
  - Experimental Comparison (depth-first vs. breadth-first)
- **Combination with other Pruning Strategies**
  - Equivalent Sibling Pruning / Perfect Extension Pruning
- **Conclusions**

Brief Review: Frequent Item Set Mining

- Frequent item set mining is a method for **market basket analysis**.
- It aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops etc.
- More specifically:
  Find sets of products that are frequently bought together.
- **Formal problem statement:**

  Given: a set $I = \{i_1, \ldots, i_m\}$ of items (products, services, options etc.),
  a set $T = \{t_1, \ldots, t_n\}$ of transactions over I, i.e., $\forall t \in T : t \subseteq I$,
  a minimal support $s_{\text{rel}} \in (0, 1]$ or $s_{\text{abs}} \in (0, |T|]$.

  Desired: all **frequent item sets**, that is, all item sets r, such that
  $|\{t \in T \mid r \subseteq t\}| \geq s_{\text{rel}} \cdot |T|$ or $|\{t \in T \mid r \subseteq t\}| \geq s_{\text{abs}}$.

  Approach: search the **item subset lattice** top down.

Brief Review: Types of Frequent Item Sets

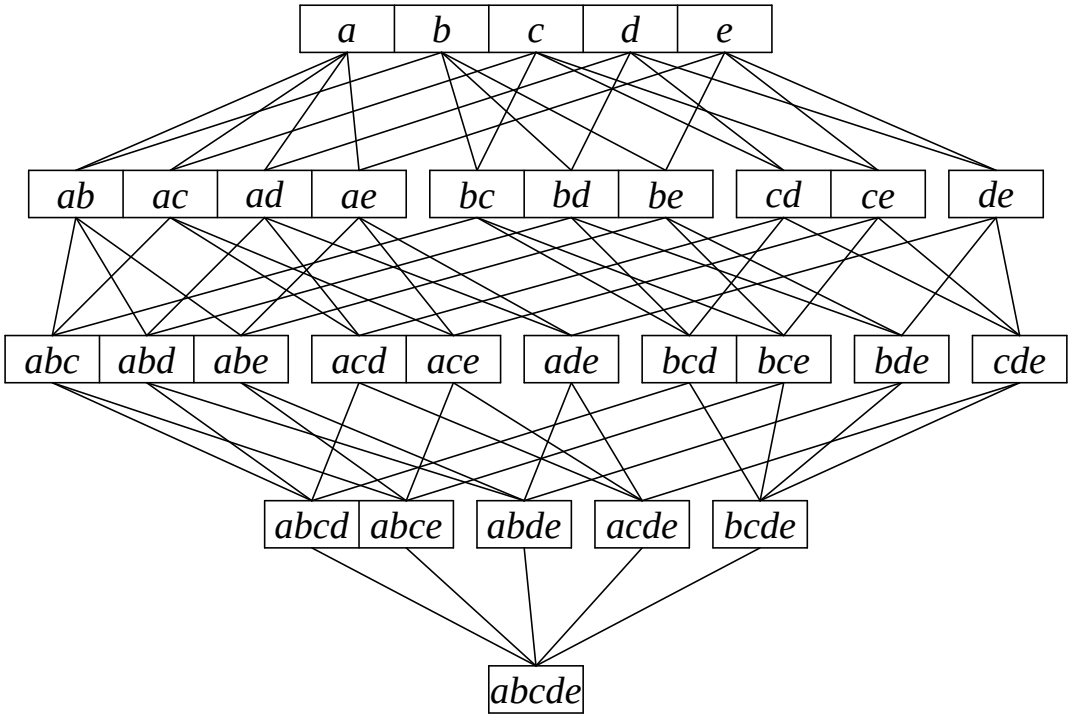
- **Free Item Set** (or simply *item set*)
  Any frequent item set (support is higher than the minimal support).
- **Closed Item Set** (marked with $^+$ in example below)
  A frequent item set is called *closed* if no superset has the same support.
- **Maximal Item Set** (marked with * in example below)
  A frequent item set is called *maximal* if no superset is frequent.

Simple Example:

1 item	2 items		3 items
$\{a\}^+$: 70%	$\{a,c\}^+$: 40%	$\{c,e\}^+$: 40%	$\{a,c,d\}^{+*}$: 30%
$\{b\}$: 30%	$\{a,d\}^+$: 50%	$\{d,e\}$: 40%	$\{a,c,e\}^{+*}$: 30%
$\{c\}^+$: 70%	$\{a,e\}^+$: 60%		$\{a,d,e\}^{+*}$: 40%
$\{d\}^+$: 60%	$\{b,c\}^{+*}$: 30%		
$\{e\}^+$: 70%	$\{c,d\}^+$: 40%		

Traversing the Subset Lattice

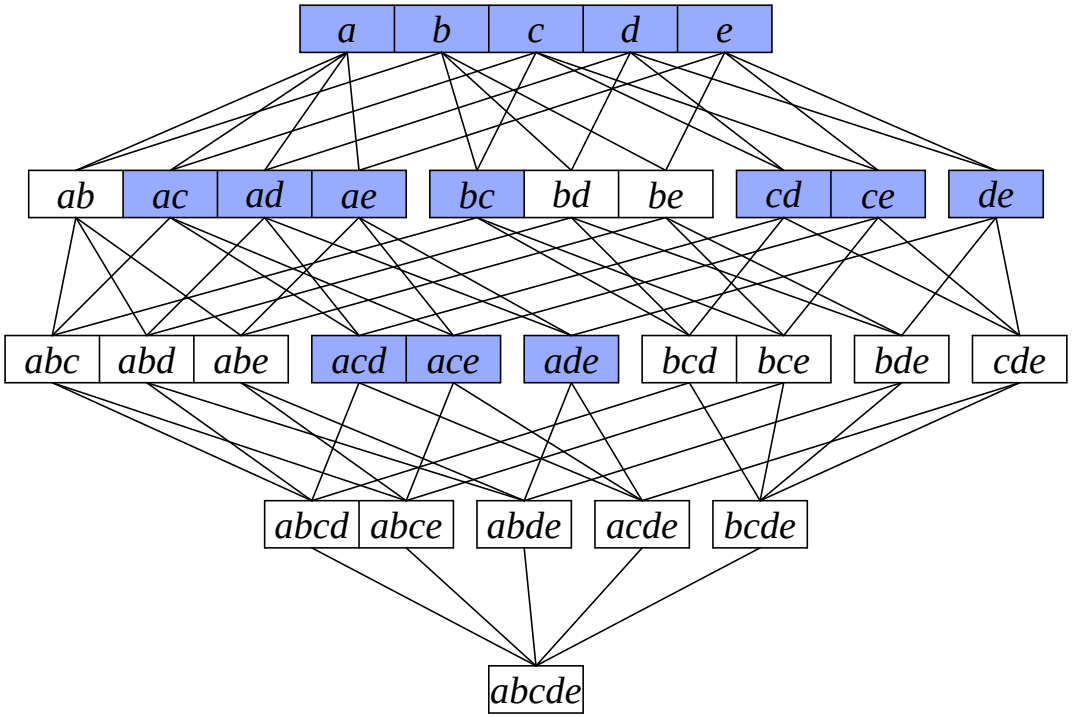
A subset lattice
for five items:



- **Apriori**
  - Breadth-first search (item sets of same size).
  - Subsets tests on transactions to find the support of item sets.
- **Eclat**
  - Depth-first search (item sets with same prefix).
  - Intersection of transaction lists to find the support of item sets.

Traversing the Subset Lattice

A subset lattice
for five items
(frequent item sets
colored blue):



- **Apriori**
  - Breadth-first search (item sets of same size).
  - Subsets tests on transactions to find the support of item sets.
- **Eclat**
  - Depth-first search (item sets with same prefix).
  - Intersection of transaction lists to find the support of item sets.

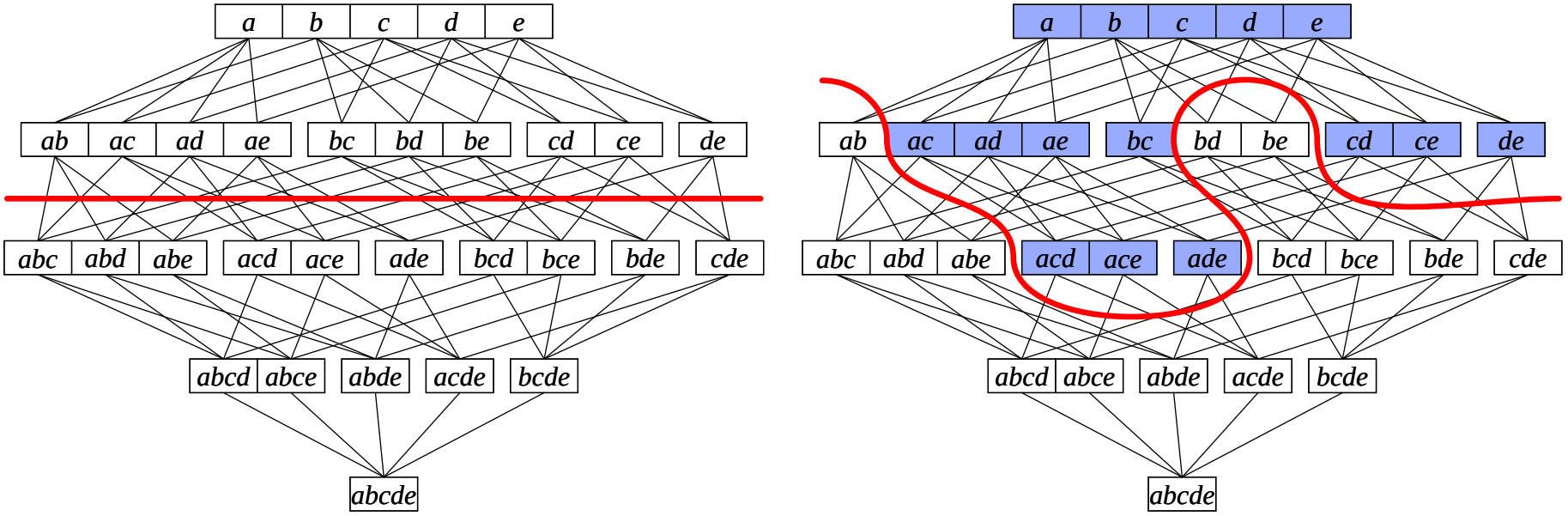
Pruning the Search

In applications the search trees tend to get very large, so we have to prune them.

- **Size Based Pruning:**
  - Prune the search tree if a certain depth is reached.
  - Restrict item sets to a certain size.
- **Support Based Pruning:**
  - No superset of an infrequent item set can be frequent.
  - No counters for item sets having an infrequent subset are needed.
- **Structural Pruning:**
  - Make sure that there is only one counter for each possible item set.
  - Explains the unbalanced structure of the full search tree.

Size-based and Support-based Pruning

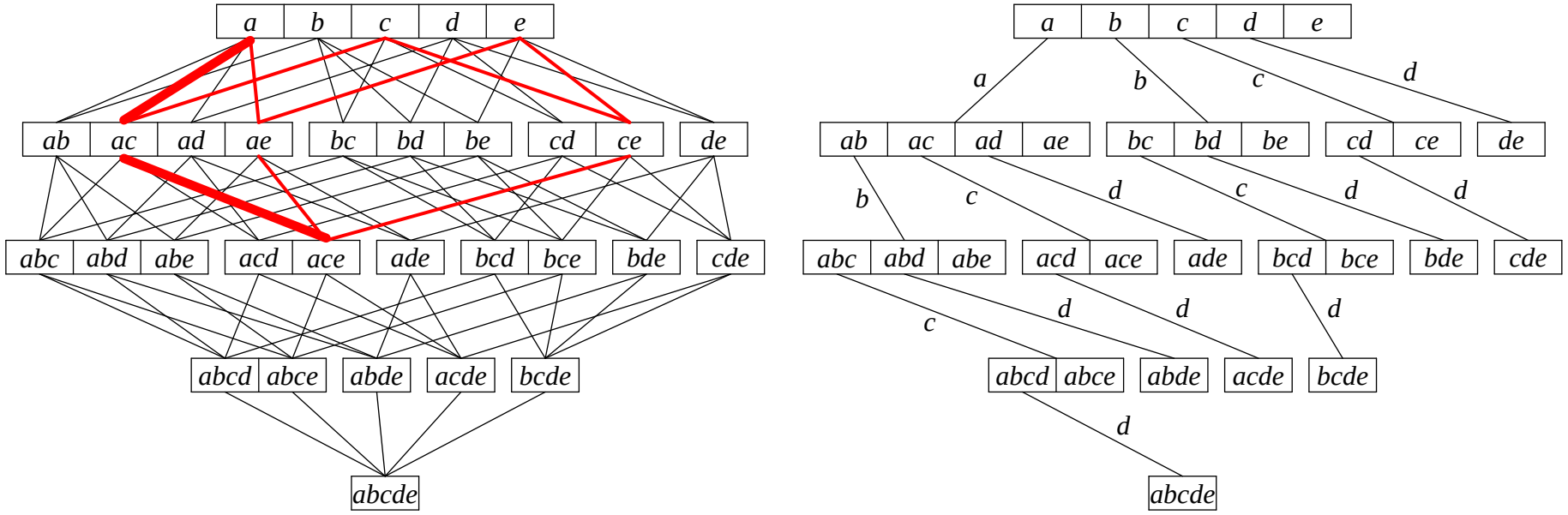
A subset lattice pruned with size-based and support-based pruning:



- **Size**
  - Prune the search tree if a certain depth is reached.
  - Restrict item sets to a certain size.
- **Support**
  - No superset of an infrequent item set can be frequent.
  - No counters for item sets with an infrequent subset are needed.

Pruning the Search

A subset lattice and the corresponding prefix tree for five items:



- **Structural**
  - Make sure that there is only one counter for each possible item set.
  - Approach: structure lattice as a prefix tree. In this prefix tree each item set appears only once.

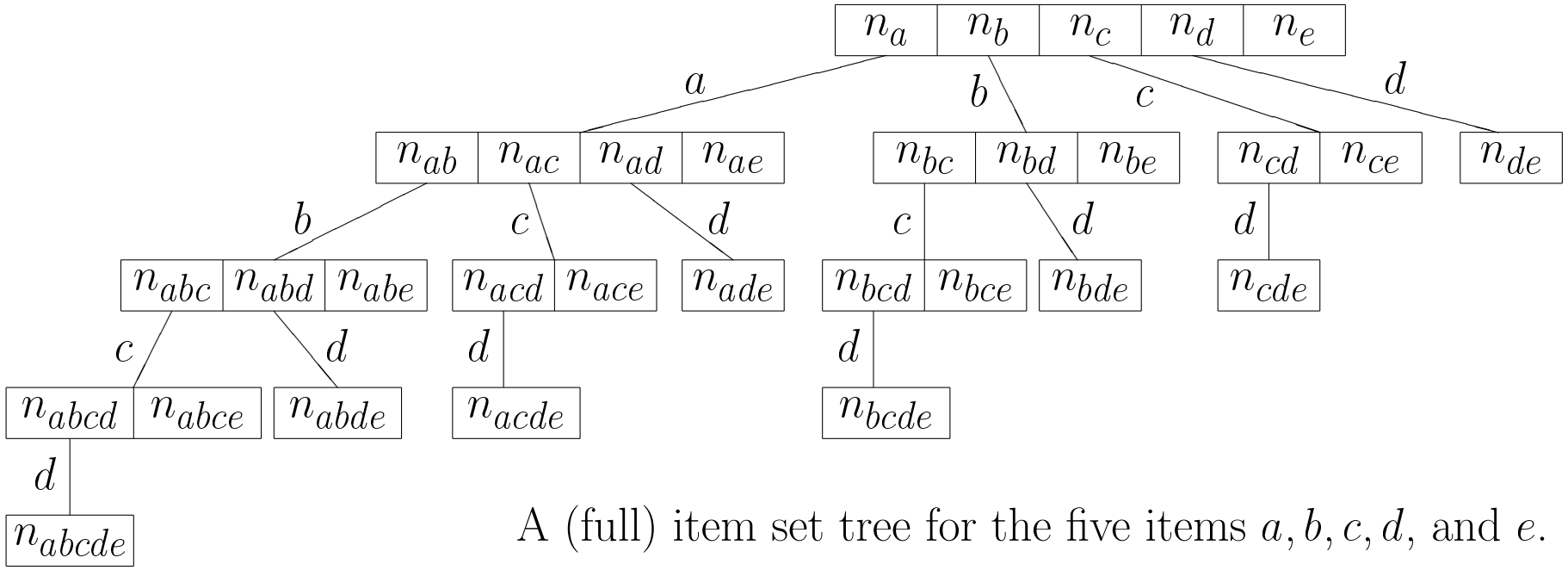
Structural Pruning for Item Sets: Canonical Form

- An item set can be written in several different ways.
  (The item set $\{a, c, e\}$ may be written as ace, aec, cae, cea, eac, and eca.)
  We say that these are different **code words** for the item set.
- Technically, the search in the subset lattice is carried out on code words.
  If in a search in the subset lattice we always follow *all* edges to supersets,
  we consider all possible code words, which leads to highly redundant search.
- We need not consider (and extend) all of these code words;
  it suffices to consider and extend one of them to traverse all supersets.
  The one we choose is called the **canonical code word** (canonical form).
- However, in order to be able to reach all possible item sets,
  the chosen canonical code words should have the **prefix property**:
  Any prefix of a canonical code word is a canonical code word itself.
- A possible choice is the **lexicographically smallest code word**;
  this is then the **canonical form** of the item set (the only extendable one).

Frequent Item Sets: Restricted Extensions

- In principle, with a canonical form for item sets, each canonical code word we meet is extended by appending all items not yet contained in it.
- It is then checked whether a resulting code word is canonical, and if it is, the support of the corresponding item set is determined. Infrequent item sets are, of course, discarded.
- However, of some such extensions we can tell immediately—that is, before actually appending the item—that the resulting code word id not canonical.
- The item to append must follow the last item in the code word (w.r.t. the global order of the items). This restricted way of extending item sets may be called **lexicographic extension**.
- This may appear to be a complex way to describe a simple pruning strategy, but it provides insights about canonical form pruning for frequent graph mining. Canonical forms for frequent graph mining can be derived in analogous ways.

Structural Pruning of Item Set Trees



A (full) item set tree for the five items a, b, c, d, and e.

- Based on a global order of the items (which can be arbitrary).
- The item sets counted in a node consist of
  - all items labeling the edges to the node and
  - one item following the last edge label.

Frequent Graph Mining: General Approach

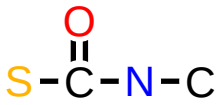
- Finding frequent item sets means to find
  sets of items that are contained in many transactions.
- Finding frequent substructures means to find
  graph fragments that are contained in many graphs
  in a given database of attributed graphs (user specifies minimum support).
- But: Graph structure of nodes and edges has to be taken into account.

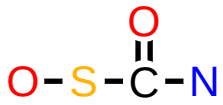
  $\Rightarrow$ Search semi-lattice of graph structures instead of subset lattice.
- Commonly the search is restricted to **connected substructures**.
- Preferred search strategy: **depth-first search**
  - Large number of small fragments $\Rightarrow$ very wide tree.
  - Embedding an attributed graph into another is costly.
- Find support by counting graphs in **lists of embeddings**.

Frequent Graph Mining: General Approach

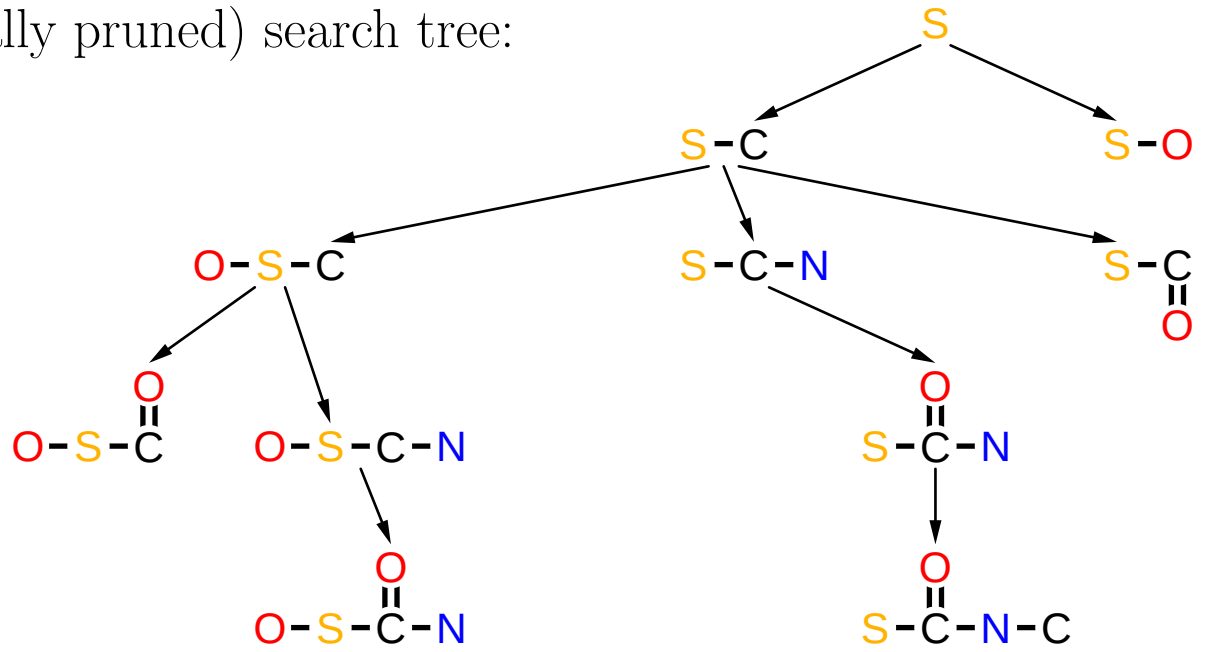
Example: Part of a search tree for a molecular database.

Three (fictitious) example molecules:





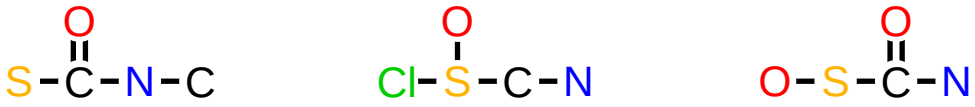
(Structurally pruned) search tree:



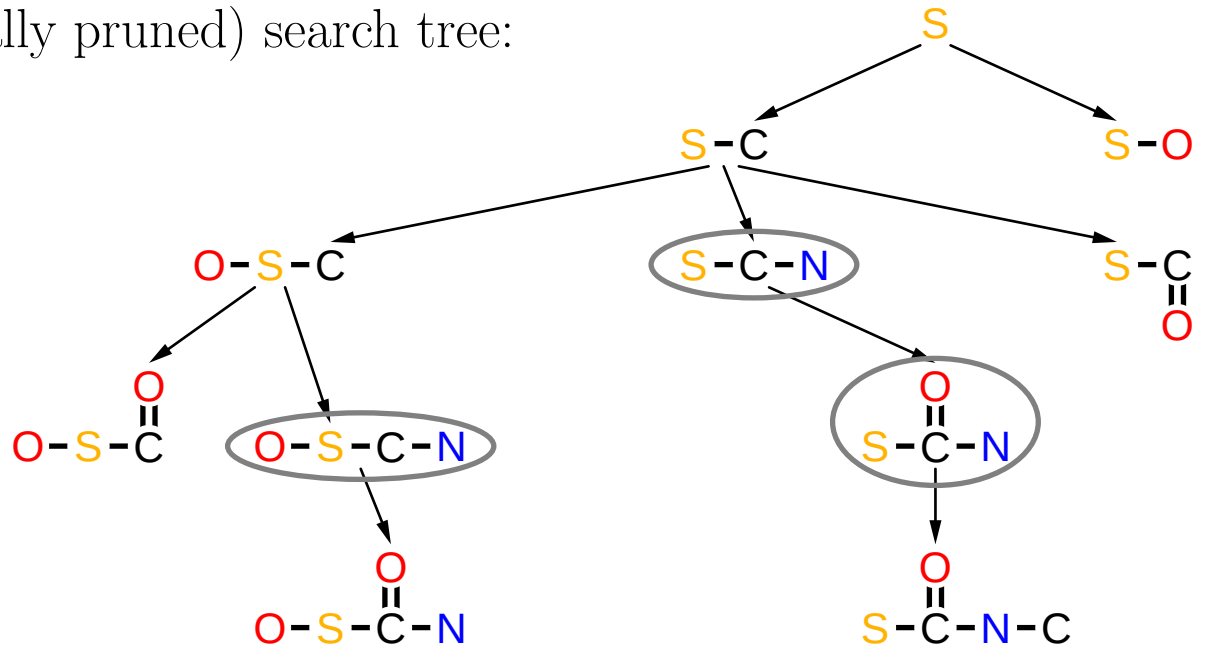
Frequent Graph Mining: Closed Fragments

A fragment F is called **closed** if no fragment that contains F as a proper substructure has the same support, i.e., is contained in the same number of graphs.

Three (fictitious) example molecules:

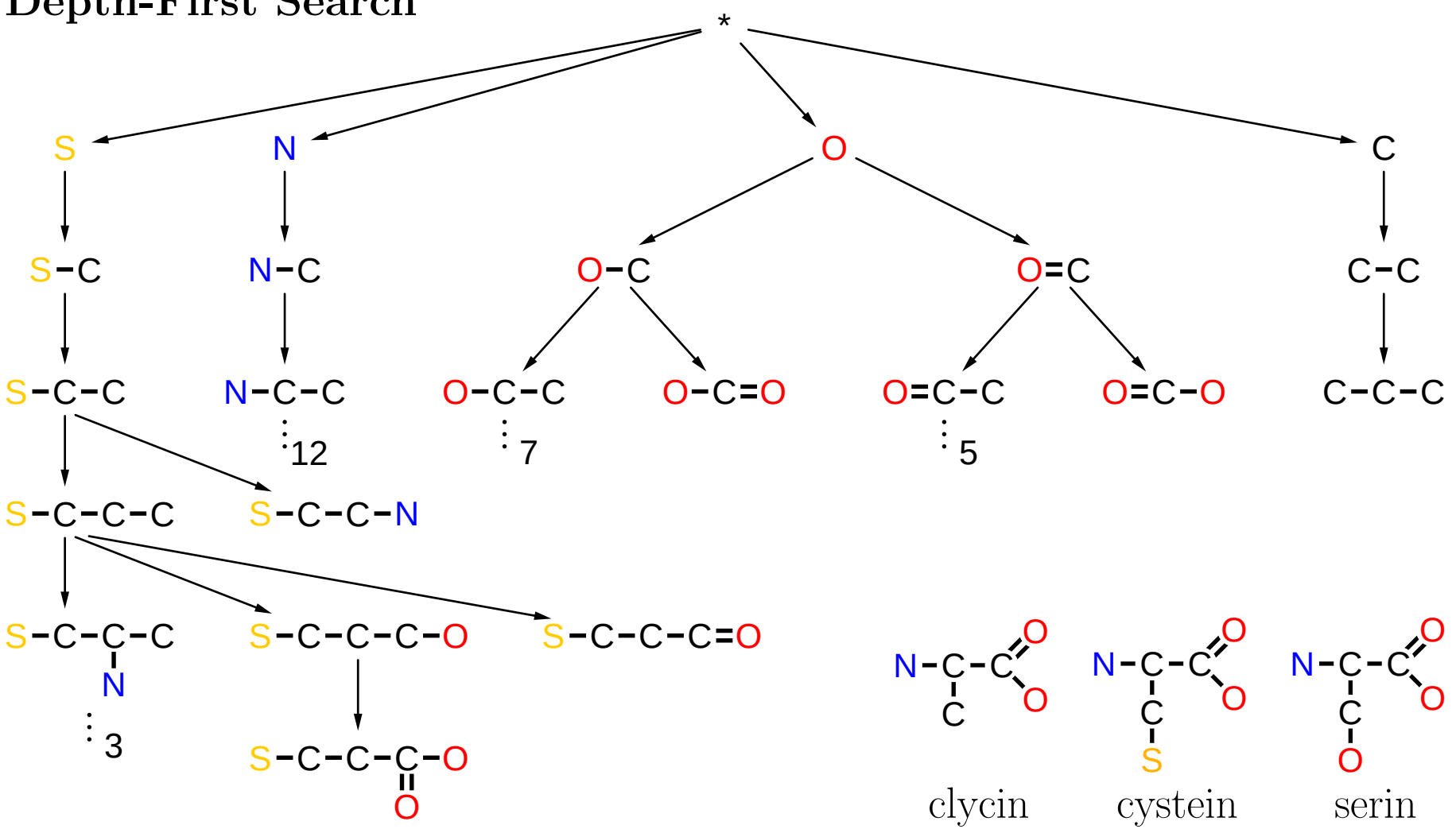


(Structurally pruned) search tree:



Searching without a Seed Atom

Depth-First Search



Canonical Forms of Graphs: General Idea

- Construct a **code word** that uniquely identifies an (attributed) graph up to isomorphism and symmetry (i.e. automorphism).
- **Basic idea:** The characters of the code word describe the edges of the graph.
- **Core problem:** Node and edge attributes can easily be incorporated into a code word, but how to describe the connection structure is not so obvious.
- The nodes of the graph must be numbered (endowed with unique labels), because we need to specify the source and the destination node of an edge.
- Each possible numbering of the nodes of the graph yields a code word, which is the concatenation of the sorted edge descriptions ("characters"). (Note that the graph can be reconstructed from such a code word.)
- The resulting list of code words is sorted lexicographically.
- The lexicographically smallest code word is the **canonical description.**

Canonical Forms: Constructing Spanning Trees

- For graph mining the canonical form should have the **prefix property**:

  Any prefix of a canonical code word is a canonical code word itself.

  (Because it guarantees that all possible graphs can be reached with it.)

- With a search restricted to connected substructures, we can ensure this by
  - systematically constructing a **spanning tree** of the graph, numbering the nodes in the order in which they are visited,
  - sorting the edge descriptions into the order in which the edges are added.

- The most common ways of constructing a spanning trees are
  - **depth-first search** $\Rightarrow$ canonical form of gSpan
  - **breadth-first search** $\Rightarrow$ canonical form of MoSS/MoFa

- An alternative way is to create all children of a node before proceeding in a depth-first manner (can be seen as a variant of depth-first search).

Canonical Forms: Edge Sorting Criteria

- The **edge description** consists of
  - the indices of the source and the destination atom
    (definition: the source of an edge is the node with the smaller index),
  - the attributes of the source and the destination atom,
  - the edge attribute.
- Sorting the edges into insertion order must be achieved
  by a **precedence order** on the describing elements of an edge.
- Order of individual elements (conjectures, but supported by experiments):
  - Node and edge attributes should be sorted according to their frequency.
  - Ascending order seems to be recommendable for the node attributes.
- **Simplification:** the source attribute is needed only for the first edge
  and thus can be split off from the list of edge descriptions.

Canonical Forms: Edge Sorting Criteria

- **Precedence Order for Depth-first Search:**
  - destination node index (ascending)
  - source node index (descending) ←
  - edge attribute (ascending)
  - destination node attribute (ascending)
- **Precedence Order for Breadth-first Search:**
  - source node index (ascending)
  - edge attribute (ascending)
  - destination node attribute (ascending)
  - destination node index (ascending)
- **Edges Closing Cycles:**

  Edges closing cycles may be distinguished from spanning tree edges, giving spanning tree edges absolute precedence over edges closing cycles.

Canonical Forms: Code Words

From these edge sorting criteria, the following code words result (regular expressions with non-terminal symbols):

- **Depth-First Search:** $a\,(i_d\,[n - i_s]\,b\,a)^m$
- **Breadth-First Search:** $a\,(i_s\,b\,a\,i_d)^m$

where
- n the number of nodes of the graph,
- m the number of edges of the graph,
- i_s index of the source node of an edge, $i_s \in \{1, \ldots, n\}$,
- i_d index of the destination node of an edge, $i_d \in \{1, \ldots, n\}$,
- a the attribute of a node,
- b the attribute of an edge.

The order of the elements describing an edge reflects the precedence order.

The expression in square brackets is one character, with the (numeric) value $n - i_s$. This serves the purpose that the edge descriptions may be sorted ascendingly w.r.t. all characters. Alternatively, $[n - i_s]$ by a i_s, which is sorted descendingly.

Searching with Canonical Forms

Principle of the Search Algorithm:

- **Base Loop:**
  - Traverse all possible node attributes, i.e.,
    the canonical code words of single node fragments.
  - Recursively process each code word that describes a frequent fragment.
- **Recursive Processing:**

  For a given (canonical) code word of a frequent fragment:
  - Generate all possible extensions by an edge (and a maybe a node).
    This is done by appending the edge description to the code word.
  - Check whether the extended code word is the **canonical form**
    of the fragment described by the extended code word
    (and whether the described fragment is frequent).

    If it is, process the extended code word recursively, otherwise discard it.

Checking for Canonical Form: Compare Prefixes

- **Base Loop:**
  - Traverse all nodes that have the same attribute as the current root node (first character of the code word; possible roots of spanning tree).
- **Recursive Processing:**
  - The recursive processing constructs alternative spanning trees and compare the code words resulting from it with the code word to check.
  - In each recursion step one edge is added to the spanning tree and its description is compared to the corresponding one in the code word to check.
  - If the new edge description is **larger**, the edge can be skipped (new code word is lexicographically larger).
  - If the new edge description is **smaller**, the code word is not canonical (new code word is lexicographically smaller).
  - If the new edge description is **equal**, the rest of the code word is processed recursively (code word prefixes are equal).

Checking for Canonical Form

```
function isCanonical (w: array of int, G: graph) : boolean;
var v : node;                          (* to traverse the nodes of the graph *)
    e : edge;                          (* to traverse the edges of the graph *)
    x : array of node;                 (* to collect the numbered nodes *)
begin
  forall v ∈ G.V do v.i := −1;         (* clear the node indices *)
  forall e ∈ G.E do e.i := −1;         (* clear the edge markers *)
  forall v ∈ G.V do begin              (* traverse the potential root nodes *)
    if v.a = w[0] then begin           (* if v is acceptable as a root node *)
      v.i := 1; x[0] := v;             (* number and record the root node *)
      if not rec(w, 1, x, 1, 0)        (* check the code word recursively and *)
      then return false;               (* abort if a smaller code word is found *)
      v.i := −1;                       (* clear the node index again *)
    end
  end
  return true;                         (* the code word is canonical *)
end
```

Checking for Canonical Form

```
function rec (w: array of int, k : int, x: array of node, n: int, i: int) : boolean;
        (* w: code word to be tested *)
        (* k: current position in code word *)
        (* x: array of already labeled/numbered nodes *)
        (* n: number of labeled/numbered nodes *)
        (* i: index of next extendable node to check; i < n *)
var d : node;                              (* node at the other end of an edge *)
    j : int;                               (* index of destination node *)
    u : boolean;                           (* flag for unnumbered destination node *)
    r : boolean;                           (* buffer for a recursion result *)
begin
   if k ≥ length(w) return true;           (* full code word has been generated *)
   while i < w[k] do begin                 (* check whether there is an edge with *)
      forall e incident to x[i] do         (* a source node having a smaller index *)
         if e.i < 0 then return false;
      i := i + 1;                          (* go to the next extendable node *)
   end
```

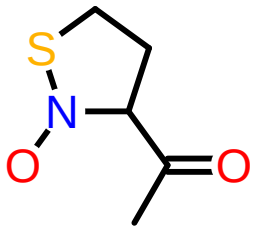
Checking for Canonical Form

```
forall e incident to x[i] (in sorted order) do begin
    if e.i < 0 then begin          (* traverse the unvisited incident edges *)
        if e.a < w[k+1] then return false;  (* check the *)
        if e.a > w[k+1] then return true;   (* edge attribute *)
        d := node incident to e other than x[i];
        if d.a < w[k+2] then return false;  (* check destination *)
        if d.a > w[k+2] then return true;   (* node attribute *)
        if d.i < 0 then j := n else j := d.i;
        if j < w[k+3] then return false;    (* check destination node index *)

        [...]                               (* check rest of code word recursively, *)
                                            (* because prefixes are equal *)
    end
end
return true;                                (* return that no smaller code word *)
                                            (* than w could be found *)
```

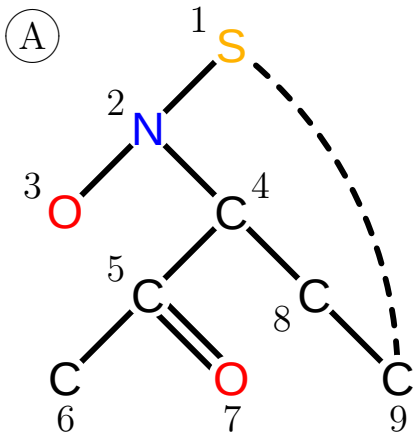
Checking for Canonical Form

```
forall e incident to x[i] (in sorted order) do begin
    if e.i < 0 then begin          (* traverse the unvisited incident edges *)

        [...]                      (* check the current edge *)

        if j = w[k+3] then begin   (* if edge descriptions are equal *)
            e.i := 1; u := d.i < 0;        (* mark edge and number node *)
            if u then begin d.i := j; x[n] := d; n := n + 1; end
            r := rec(w, k+4, x, n, i);          (* check recursively *)
            if u then begin d.i := -1; n := n - 1; end
            e.i := -1;                 (* unmark edge (and node) again *)
            if not r then return false;
        end                            (* evaluate the recursion result *)
    end
end
return true;                           (* return that no smaller code word *)
end                                    (* than w could be found *)
```

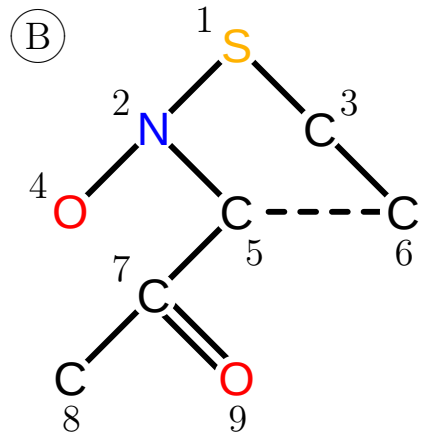
Canonical Forms: A Simple Example



example molecule



depth-first



breadth-first

Order of Elements: `S` $\prec$ `N` $\prec$ `O` $\prec$ `C` **Order of Bonds:** $-$ $\prec$ $=$

Code Words:

```
A:  S 28-N 37-O 47-C 55-C 64-C 74=O 85-C 91-C 98-S
       1    2    2    4    5    5    4    8    1

B:  S 1-N2 1-C3 2-O4 2-C5 3-C6 5-C6 5-C7 7-C8 7=O9
```

Canonical Forms: Restricted Extensions

Principle of the Search Algorithm up to now:

- Generate all possible extensions of a given (frequent) fragment by an edge (and a maybe node).
- Check whether the extended fragment is in canonical form (and frequent). If it is, process the extended fragment recursively, otherwise discard it.

Straightforward Improvement:

- For some extensions of the given (frequent) fragment it is easy to see that they are not in canonical form.
- The trick is to check whether a spanning tree **rooted at the same node** yields a code word that is smaller than the one describing the fragment.
- This immediately rules out extensions of certain nodes in the fragment as well as certain edges closing cycles.

Canonical Forms: Restricted Extensions

Depth-First Search: Rightmost Extension

- **Extendable Nodes:**
  - Only nodes on the **rightmost path** of the spanning tree may be extended.
  - If the source node of the new edge is not a leaf, the edge description must not precede the description of the downward edge on the path.

    (That is, the edge attribute must be no less than the edge attribute of the downward edge, and if it is equal, the attribute of its destination node must be no less than the attribute of the downward edge's destination node.)
- **Edges Closing Cycles:**
  - Edges closing cycles must start at an extendable node.
  - They must lead to the rightmost leaf (node at end of rightmost path).
  - The index of the source node must precede the index of the source node of any edge already incident to the rightmost leaf.

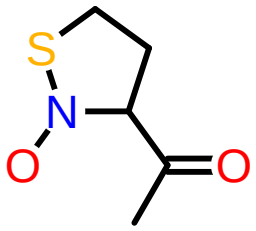
Canonical Forms: Restricted Extensions

Breadth-First Search: Maximum Source Extension

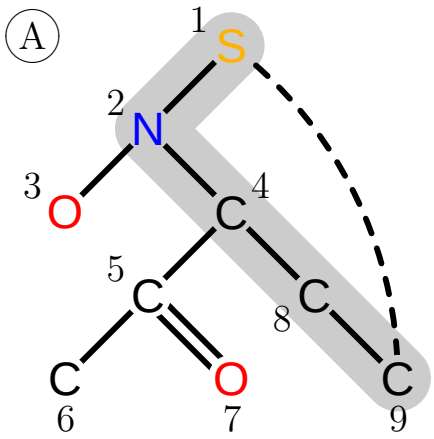
- **Extendable Nodes:**
  - Only nodes having an index no less than the **maximum source index** of an edge already in the fragment may be extended.
  - If the source of the new edge is the one having the maximum source index, it may be extended only by edges whose descriptions do not precede the description of any downward edge already incident to this node.

    (That is, the edge attribute must be no less, and if it is equal, the attribute of the destination node must be no less.)
- **Edges Closing Cycles:**
  - Edges closing cycles must start at an extendable node.
  - They must lead "forward",
    that is, to a node having a larger index than the extended node.

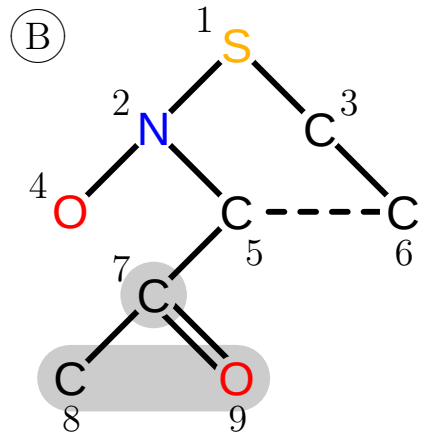
Restricted Extensions: A Simple Example



example molecule



depth-first



breadth-first

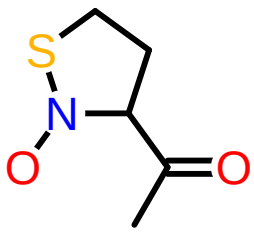
Extendable Nodes:

- A: nodes on the rightmost path, i.e., 1, 2, 4, 8, 9.
- B: nodes with an index no smaller than the maximum source, i.e., 7, 8, 9.

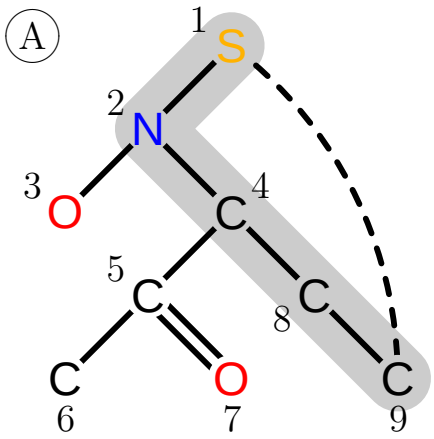
Edges Closing Cycles:

- A: none, because the existing cycle edge has minimum source.
- B: edge between nodes 8 and 9.

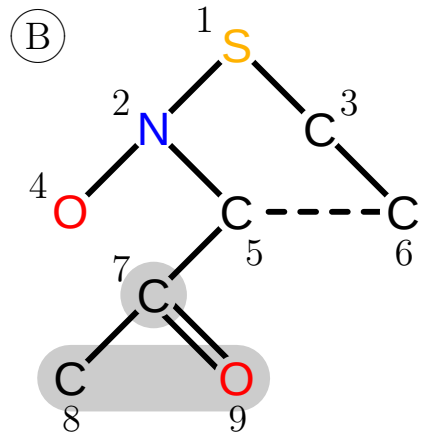
Restricted Extensions: A Simple Example



example molecule



depth-first



breadth-first

If other nodes are extended, a tree *with the same root* yields a smaller code word.

Examples:

```
A:  S 28-N 37-O 47-C 55-C 64-C 74=O 85-C 91-C 98-S 03-C
    S 28-N 37-C 43-C ···

B:  S 1-N2 1-C3 2-O4 2-C5 3-C6 5-C6 5-C7 7-C8 7=O9 4-C0
    S 1-N2 1-C3 2-O4 2-C5 3-C6 4-C7 ···
```

Canonical Forms: Comparison

Depth-First vs. Breadth-First Search Canonical Form

- With breadth-first search canonical form the extendable nodes are much easier to traverse, as they always have consecutive indices:

  One only has to store and update one number, namely the index of the maximum bond source, to describe the node range.

- Also the check for canonical form is slightly more complex (to program) for depth-first canonical form (maybe I did not find the best way, though).

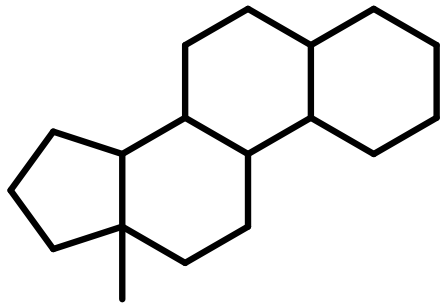
- The two canonical forms obviously lead to different branching factors, widths and depths of the search tree.

  However, it is not immediately clear, which form leads to the "better" (more efficient) structure of the search tree.

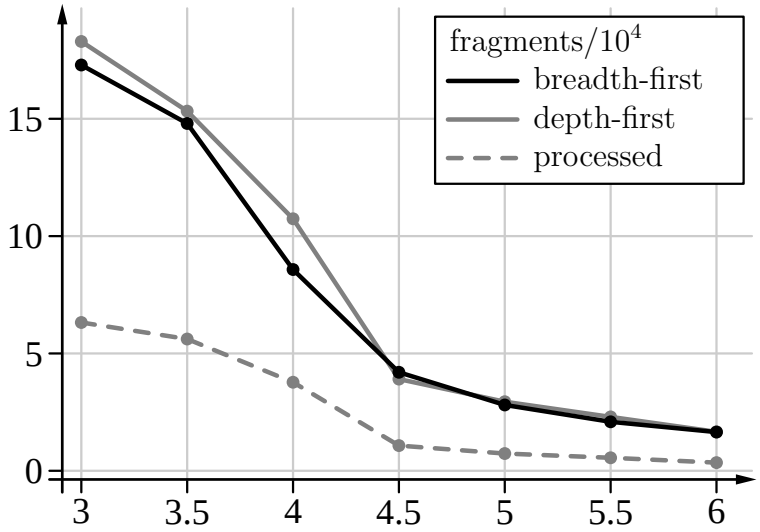
- The experimental results reported in the following indicate that it may depend on the data set which canonical form performs better.

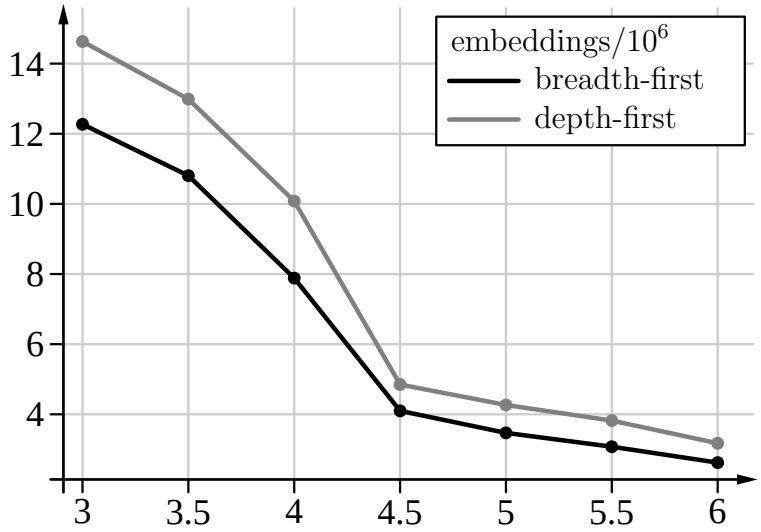
Experimental Results: Data Sets

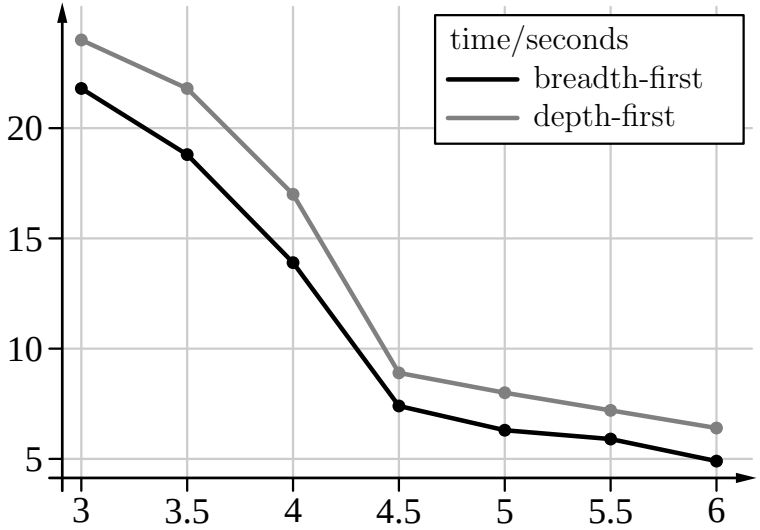
- **Index Chemicus — Subset of 1993**
  - 1293 molecules / 34431 atoms / 36594 bonds
  - Frequent fragments down to fairly low support values are trees (no rings).
  - Medium number of fragments and closed fragments.
- **Steroids**
  - 17 molecules / 401 atoms / 456 bonds
  - A large part of the frequent fragments contain one or more rings.
  - Huge number of fragments, still large number of closed fragments.



Experimental Results: IC93 Data Set

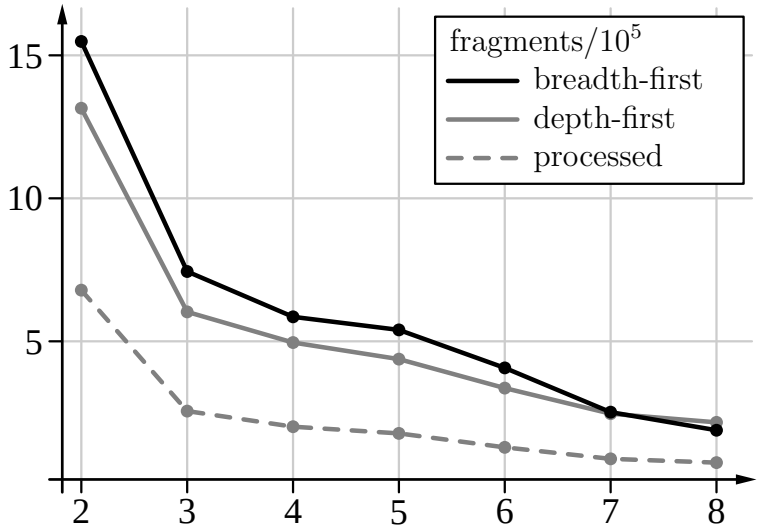


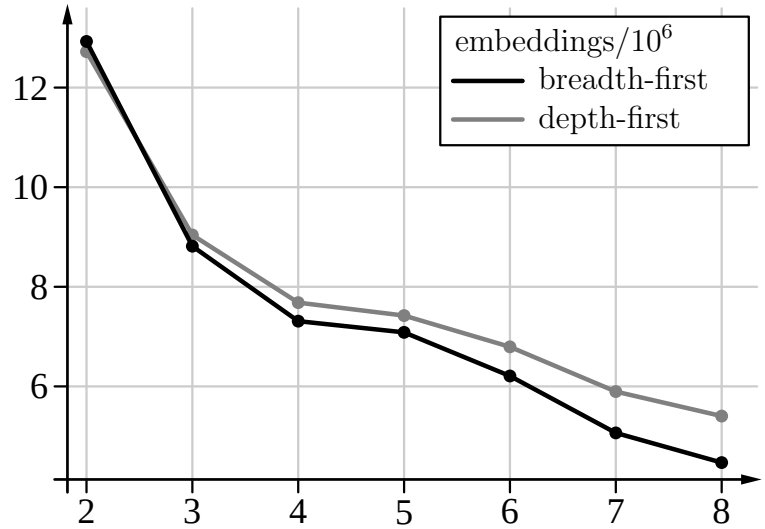


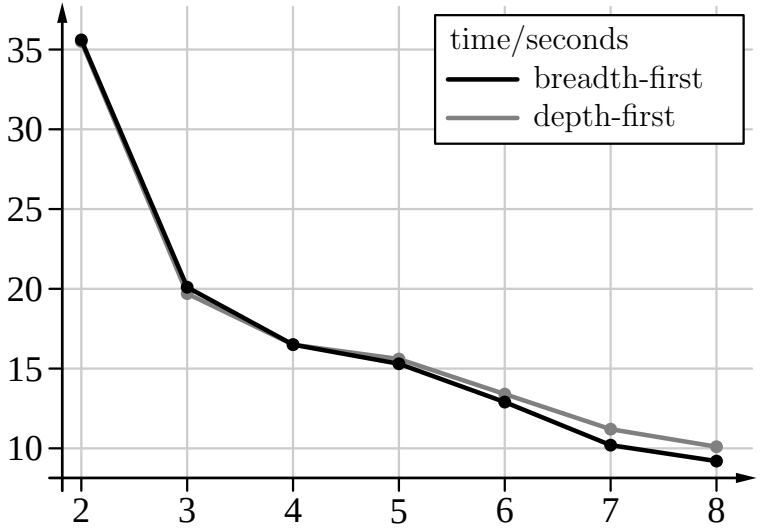


Experimental results on the IC93 data. The horizontal axis shows the minimal support in percent. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

Experimental Results: Steroids Data Set







Experimental results on the steroids data. The horizontal axis shows the absolute minimal support. The curves show the number of generated and processed fragments (top left), number of generated embeddings (top right), and the execution time in seconds (bottom left) for the two canonical forms/extension strategies.

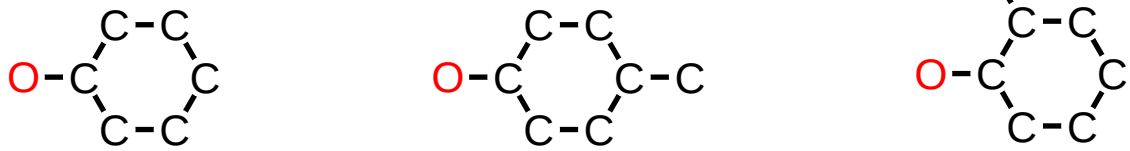
Alternative Test: Equivalent Siblings

- **Basic Idea:**
  - If the fragment to extend exhibits a certain symmetry, several extensions may be equivalent (in the sense that they describe the same fragment).
  - At most one of these sibling extensions can be in canonical form, namely the one *least restricting future extensions* (smallest code word).
  - Identify equivalent siblings and keep only the maximally extendable one.
- **Test Procedure for Equivalence:**
  - Get any molecule into which two sibling fragments to compare can be embedded. (If there is no such molecule, the siblings are not equivalent.)
  - Mark any embedding of the first fragment in the molecule.
  - Traverse all embeddings of the second fragment into the molecule and check whether all bonds of an embedding are marked. If there is such an embedding, the two fragments are equivalent.

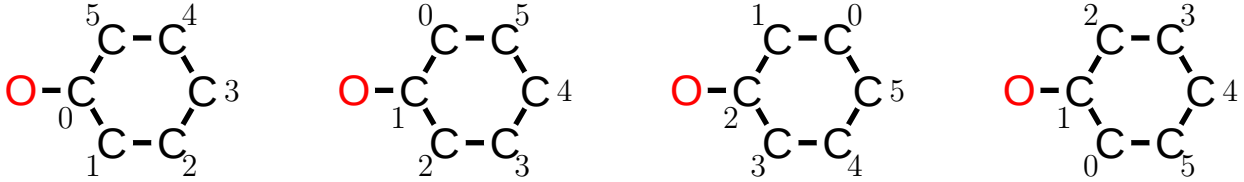
Alternative Test: Equivalent Siblings

**If siblings in the search tree are equivalent,
only the one with the least restrictions needs to be processed.**

Example: Mining phenol, p-cresol, and catechol.



Consider extensions of a benzene ring (twelve possible embeddings):



Only the fragment that **least restricts future extensions**
(i.e., that has the smallest code word) can be in canonical form.

Alternative Test: Equivalent Siblings

- **Test for Equivalent Siblings before Test for Canonical Form**
  - Traverse the sibling extensions and compare each pair.
  - Of two equivalent siblings remove the one that restricts future extensions more.
- **Advantages:**
  - Identifies some fragments that are non-canonical in a simple way.
  - Test of two siblings is at most linear in the number of bonds.
- **Disadvantages:**
  - Does not identify all non-canonical fragments, therefore a subsequent canonical form test is still needed.
  - Compares two sibling fragments, therefore it is quadratic in the number of siblings.

Alternative Test: Equivalent Siblings

The effectiveness of equivalent sibling pruning depends on the canonical form:

Mining the **IC93 data** with 4% minimal support

	depth-first	breadth-first
equivalent sibling pruning	156 (1.9%)	4195 (83.7%)
canonical form pruning	7988 (98.1%)	815 (16.3%)
total pruning	8144	5010
(closed) fragments found	2002	2002

Mining the **steroids data** with minimal support 6

	depth-first	breadth-first
equivalent sibling pruning	15327 (7.2%)	152562 (54.6%)
canonical form pruning	197449 (92.8%)	127026 (45.4%)
total pruning	212776	279588
(closed) fragments found	1420	1420

Alternative Test: Equivalent Siblings

Observations:

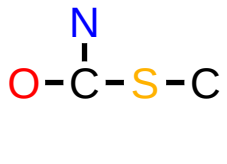
- Depth-first form generates more duplicate fragments the on IC93 data and fewer duplicate fragments on the steroids data (as seen before).
- There are only very few equivalent siblings with depth-first form on both the IC93 data and the steroids data.

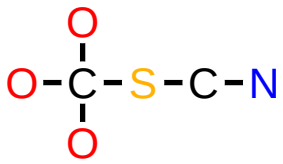
  (Conjecture: equivalent siblings result from "rotated" tree branches, which are less likely to be siblings with depth-first form.)
- With breadth-first form a large part of the fragments that are not in canonical form can be filtered out with equivalent.
- On the test IC93 data no difference in speed could be observed, presumably because pruning takes only a small part of the total time.
- On the steroids data, however, equivalent sibling pruning yields a slight speed-up for breadth-first form ($\sim 5\%$).

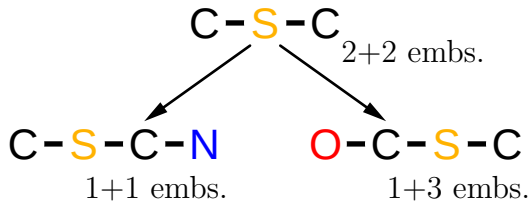
Perfect Extension Pruning

An extension of a fragment is called **perfect** if it is a bridge and can be applied to all embeddings of the fragment in the same way.

Examples of perfect and non-perfect extensions:







- **If a fragment allows for a perfect extension, siblings in the search tree can be pruned.**
- **Idea:** First grow the fragment to the biggest common substructure of the set of molecules considered in this branch of the search tree.
- **Presupposition:** Restriction to closed fragments. (Some non-closed fragments may be lost.)

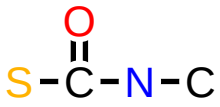
Perfect Extension Pruning

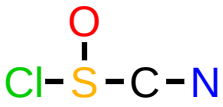
Checking for perfect extensions during the search:

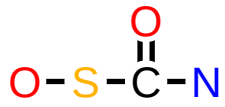
- Exploit simple relations of the number of embeddings and molecules.
- **Failing early:** An extension cannot be perfect if
  - the number of molecules referred to by the extended fragment differs from those referred to by its parent,
  - the number of embeddings of the extended fragment is not an integer multiple of the number of embeddings of its parent.
- **Succeeding early:** An extension is perfect if
  - the extended fragment refers to only one molecule,
  - the number of molecules referred to equals the number of embeddings.
- Only afterwards the somewhat costly **exact check** is carried out: Does each embedding lead to the same number of extended embeddings.

Perfect Extension Pruning

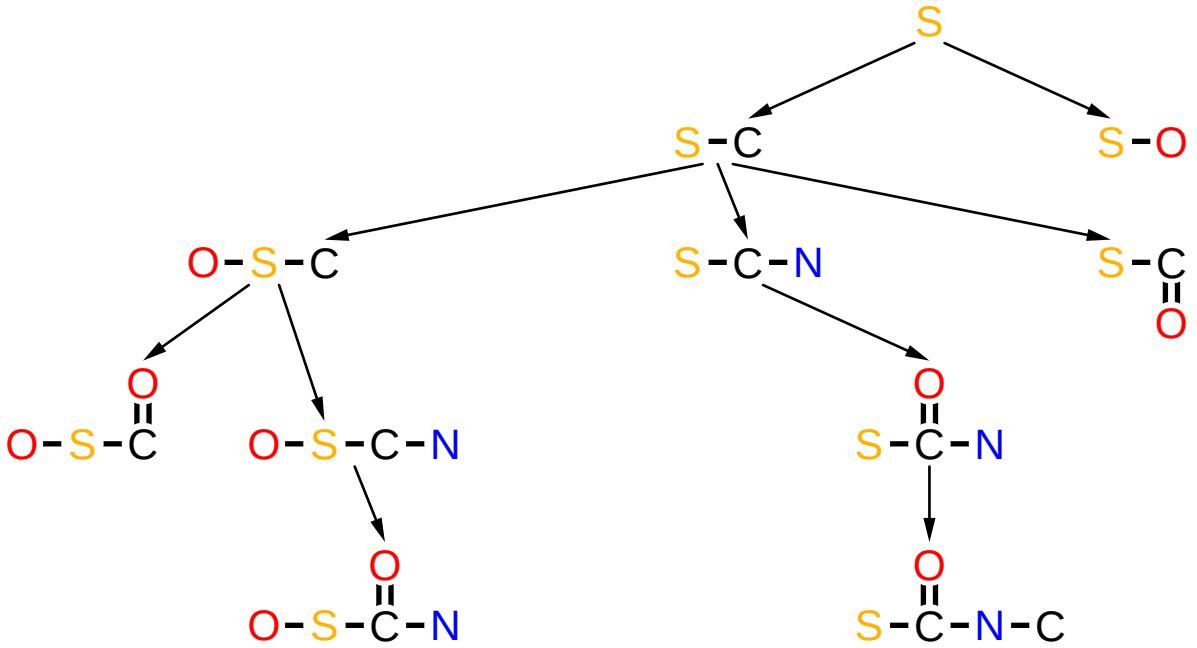
Three (fictitious) example molecules:





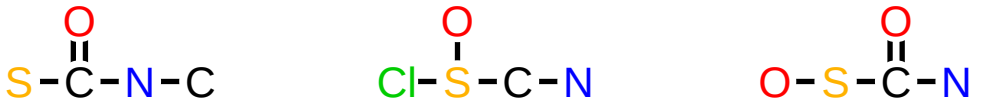


Search tree **without** perfect extension pruning:

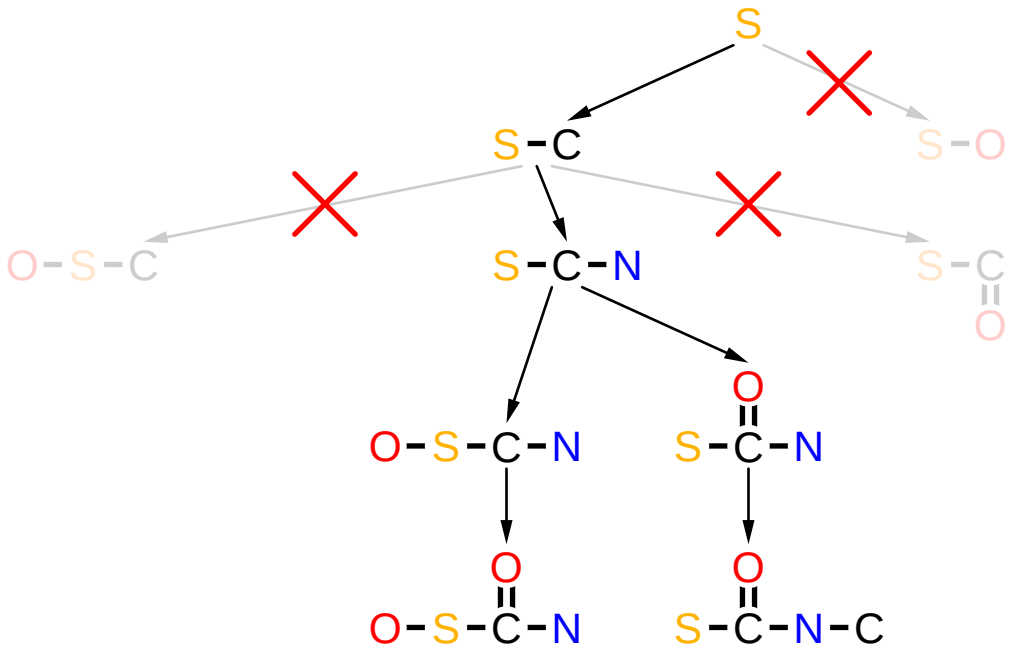


Perfect Extension Pruning

Three (fictitious) example molecules:



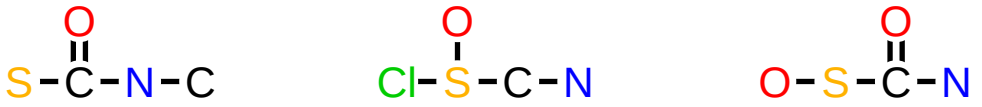
Search tree **with** perfect extension pruning (reseting extension information):



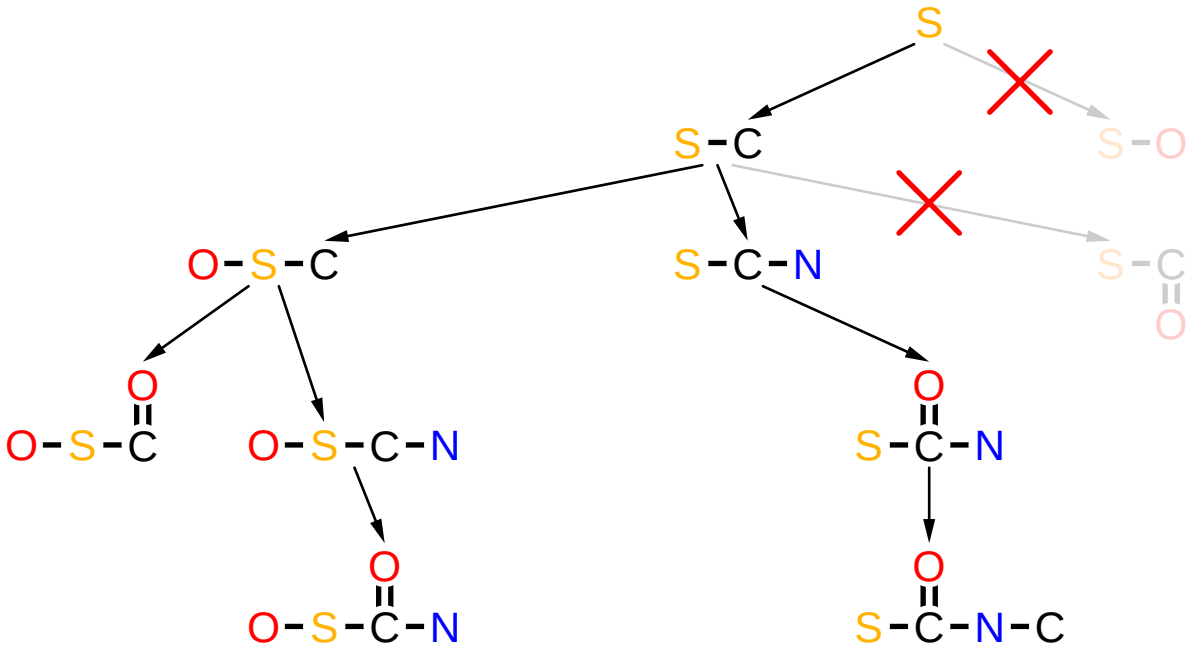
However, reseting the extension information interferes with canonical form pruning!

Perfect Extension Pruning

Three (fictitious) example molecules:



Search tree **with restricted** perfect extension pruning:



Only prune extensions to the right of the (left-most) perfect extension. (Order of the siblings: lexicographically by code word).

Perfect Extension Pruning and Canonical Form

- All siblings to the right of (i.e. with a code word larger than) a perfect extension can be pruned (sibling extensions are sorted by their code words).
- The reason is that no fragment in the search tree branches to the right of a perfect extension can be a closed fragment:

  It is always possible to add the edge of the perfect extension without reducing the number of supporting graphs.

  (Note that the perfect extension edge cannot be added in any of the branches to the right due to restricted extensions — rightmost or maximum source.)
- However, this restricted perfect extension pruning does not exploit the full power of perfect extensions.
- A better approach would be to keep perfect extensions separate and treat them in a specific way in the canonical form test. First investigations indicate that this may be easier to accomplish with a breadth-first than with a depth-first canonical form.

Conclusions

- All algorithms for frequent graph mining that add a bond (and maybe an atom) in each step, can be seen as building a spanning tree for each fragment.
- The way in which the spanning tree is built, produces a labeling/numbering of the nodes, which yields a specific way of describing the edges.
- Code words for a fragment are sorted edge descriptions (preceded by the node attribute of the root node).
- The lexicographically smallest code word is the canonical form.
- Each systematic way of constructing a spanning tree and each sorting order for the edge descriptions (having the prefix property) yields a canonical form.
- With this insight gSpan and MoSS/MoFa can be seen as two variants of the same basic scheme (depth-first vs. breadth-first spanning tree construction).

Software: `http://fuzzy.cs.uni-magdeburg.de/~borgelt/moss.html`