An Algorithm for Generating All Connected Subgraphs with k Vertices of a Graph

Shant Karakashian[1] Berthe Y. Choueiry[1] Stephen G. Hartke[2]

[1]Constraint Systems Laboratory
Department of Computer Science & Engineering

[2]Department of Mathematics

University of Nebraska–Lincoln
Email: {shantk|choueiry}@cse.unl.edu, hartke@math.unl.edu

UNL-CSE-2013-0005

June 10, 2013

Abstract

In this paper, we introduce a new algorithm ConSubg(k, G) for computing all the connected subgraphs of a fixed size k of a graph G. ConSubg exploits the structure of the graph to prevent the generation of disconnected subgraphs, and is thus particularly advantageous for large sparse graphs. The two main features of our approach are the construction of a combination tree and the definition of an operator $\otimes_t$ applied to the nodes of the tree that allow us to generate without duplication the connected subgraphs. We describe and analyze ConSubg(k, G), then demonstrate its effectiveness on random graphs with a fixed degree, scale-free networks, and dual graphs of constraint satisfaction problems.

Contents

1 Introduction

Generating all subgraphs of a graph G that satisfy a given property has been studied in many contexts, for example, finding all maximum or maximal cliques [8, 1, 23, 47, 52, 4, 3, 36, 30, 35, 32, 43, 46, 9, 18, 12, 13], cycles [37, 56, 45, 53, 44, 22, 33, 5, 31, 39, 54, 6, 40, 19], and spanning trees [16, 14, 25, 34, 26, 41, 48, 42, 24]. In this paper, we describe an algorithm for generating all connected subgraphs with a fixed number of vertices. As far as we are aware, identifying all connected subgraphs has not been studied before. However, this problem is a crucial step in enforcing higher consistency levels in Constraint Processing [15, 49, 50, 51, 11, 29, 28].

This combinatorial problem is computationally challenging in practice because the number of connected subgraphs with k vertices in a graph of n vertices may be as large as $\binom{n}{k}$. However, in sparse graphs, the number of connected subgraphs is significantly smaller than the number of k subsets of vertices. Thus, it is important to design an algorithm that exploits the structure of the graph when generating the connected subgraphs.

In this paper, we propose, discuss, and evaluate ConSubg, an algorithm for this purpose. The two main features of our approach are the construction of a combination tree T and the definition of an operator $\otimes_t$. The combination tree T rooted at a vertex $v \in G$ has the property that the depth-first tree rooted at v of every G', where G' is a connected subgraph induced on G by at most k vertices including v, is isomorphic to a subgraph of T rooted at v. The operator $\otimes_t$ generates from T, without duplication, all connected subgraphs of G of size k including v. We evaluate it empirically on randomly generated graphs, scale-free graphs commonly used to model social networks, and graphs derived from constraint satisfaction problems. We use the simple example of Figure 1 throughout the paper to illustrate the operation of ConSubg. For example, the connected subgraphs of size $k = 4$ for the graph shown in Figure 1 are:

$$\textsc{ConSubg}(k, G) = \{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}. \quad (1)$$

Note that $\{b, c, d, e\}$ is not a connected subgraph of G and is thus excluded from the result.

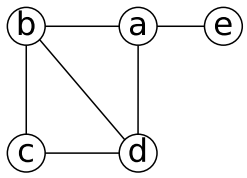


Figure 1: Simple graph.

Definition 1.1. Given a graph $G = (V, E)$ and a constant k, ConSubg(k, G)

returns all sets of V' vertices where $V' \subseteq V$, $|V'| = k$, and the subgraph G' of G induced by V' is connected. We call such sets of k vertices k-*ConnVertices*.

The graph in Figure 1 represents the dual graph of a Constraint Satisfaction Problem (CSP) [10]. A vertex in this graph represents a constraint, defined as a relation on a set of variables, which is the scope of the constraint. An edge connects two vertices whose scopes overlap (i.e., share a variable). Enforcing the consistency properties R(i,k)C [11] and R($*$,k)C [29] on this CSP requires computing all connected subgraphs of size k.[1]

This paper is structured as follows. Section 2 reviews alternative approaches. Section 3 constitutes the bulk of this paper: It discusses in great detail ConSubg and its various components, introducing data structures that we designed for this purpose and discussing the complexity, soundness and completeness of the constituent components of ConSubg. Section 4 proposes to improve the performance of the algorithm by memoization. Section 5 provides a proof of the correctness of the algorithm. Section 6 demonstrates the practical usefulness of our algorithm by comparing its performance on randomly generated graphs, scale-free networks, and constraint satisfaction problems. Finally, Section 7 concludes this paper.

2 Alternative approaches

A straightforward algorithm for implementing ConSubg is to, first, generate all k combinations of the V vertices of the graph G, then to remove those combinations that are not connected subgraphs of G. A simple algorithm for generating such a combination consists of k nested loops, which we call the 'brute-force algorithm' and denote BF-ConSubg. BF-ConSubg (Algorithm 1), where $s = |V|$, enumerates all possible combinations of k vertices storing only those that correspond to connected subgraphs.

BF-ConSubg fails to exploit the connectivity of the graph: it may generate many subgraphs that are not connected and have to be discarded, which is wasteful of computing resources. In contrast, ConSubg exploits the connectivity of the graph and generates only connected subgraphs. At the risk of significantly oversimplifying it, ConSubg operates as follows:

1. It considers an arbitrary node in the graph as a 'root' node.
2. It restricts itself to the nodes of a distance k from this root node.
3. It generates all k-*ConnVertices* that include the root node.
4. It removes the root node from the graph.
5. Finally, it iteratively applies the above process to the remaining nodes of the graph.

[1] The graph in Figure 1 can also represent the constraint network of a binary CSP where a vertex represent a variable and an edge represent a binary constraint. The connected subgraphs of size k are useful for enforcing the consistency property k-consistency [15].

Algorithm 1: BF-ConSubg(k, G)

Input: k, G
Output: A list of all k-*ConnVertices* of G

```
1  pos: a vector of the vertices of G
2  list ← ∅
3  for i_1 ← pos[1] to pos[s − k + 1] do
4  |  for i_2 ← pos[i_1] to pos[s − k + 2] do
5  |  |  for i_3 ← pos[i_2] to pos[s − k + 3] do
6  |  |  |  ...
7  |  |  |  for i_k ← pos[i_{k−1}] to pos[s] do
8  |  |  |  |  if (i_1, i_2, ..., i_k) forms a connected subgraph of G then
9  |  |  |  |  |  Push((i_1, i_2, i_3, ..., i_k), list)
10 |  |  |  |  end
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return list
```

The strength of ConSubg stems from the particular structures and processes implemented in the above mentioned Steps 2 and 3. In order to show that the effectiveness of our approach is not limited to the above 'decomposition' strategy but that we do exploit the topology of the graph in a much stronger sense, we modify the brute-force algorithm BF-ConSubg (Algorithm 1) to apply it in a localized manner similarly to the above-listed strategy, yielding LBF-ConSubg (Algorithm 2).

Algorithm 2: LBF-ConSubg(k, G).

Input: k, G
Output: A list of all k-*ConnVertices* of G

```
1  list ← ∅
2  queue ← Vertices(G)
3  foreach v ∈ queue do
4  |  G' ← the subgraph of G induced by vertices within distance k from v
5  |  list ← list ∪ BF-ConSubg(k, G')
6  |  Remove(v, G)
7  end
8  return list
```

While the worst-case complexity of all algorithms remains exponential in k (because the number of k-*ConnVertices* may be exponential in k), we conduct, in Section 6, an extensive empirical evaluation to compare the performance of ConSubg, BF-ConSubg and LBF-ConSubg on various types of graphs, and

empirically establish the advantages of ConSubg.

3 Description of the algorithm

For the sake of clarity and readability and to facilitate the analysis, we decompose the presentation of our algorithm into components shown in Table 1. After the presentation of each component of the algorithm, we illustrate its operation on the simple example of Figure 1. When applicable, we also discuss the complexity, soundness, and completeness of the proposed component.

Table 1: A quick reference table to the proposed algorithms.

Algorithm	**Pseudocode**	**Calls algorithm(s)**	**Section**
ConSubg	Algorithm 3	CombinationsWithV	Section 3.1
CombinationsWithV	Algorithm 4	CombinationTree CombinationsFromTree	Section 3.1
CombinationTree	Algorithm 5	BuildTree	Section 3.2
BuildTree	Algorithm 6	Self	Section 3.2
CombinationsFromTree	Algorithm 7	Self k-combinations k-compositions	Section 3.3

3.1 ConSubg and CombinationsWithV

ConSubg (Algorithm 3) takes as input an integer k and a graph G and returns all lists of k vertices inducing connected subgraphs of G. Starting from an arbitrary node, it calls CombinationsWithV (Algorithm 4) on a vertex of G to generate all k-*ConnVertices* that include that vertex. Then, it removes the vertex from the graph and repeats the same operation on each of the remaining vertices in the graph.

Algorithm 3: ConSubg(k, G).

Input: k, G
Output: A list of all k-*ConnVertices* of G

```
1 list ← ∅
2 queue ← Vertices(G)
3 foreach v ∈ queue do
4 |   list ← list ∪ CombinationsWithV(v, k, G)
5 |   Remove(v, G)
6 end
7 return list
```

CombinationsWithV (Algorithm 4) calls:

- CombinationTree (Algorithm 5), which builds a combination tree rooted at the vertex given as input, and
- CombinationsFromTree (Algorithm 7), which operates on the generated combination tree to compute the set of *k-ConnVertices*.

Algorithm 4: CombinationsWithV(v, k, G).

Input: v, k, G
Output: A list of all *k-ConnVertices* of G that include vertex v
1 *tree* ←CombinationTree(v, k, G)
2 *ncombs* ← CombinationsFromTree$(tree, k)$
3 **return** Labels$(ncombs)$

Illustrating the execution of ConSubg and CombinationsWithV: Below, we discuss the application of ConSubg (Algorithm 3) with $k = 4$ to the graph of Figure 2. The queue is initialized in Line 2 to $\{a, b, c, d, e\}$, which is the list of vertices of the graph. Calling CombinationsWithV (Algorithm 4) with a and $k = 4$ on G returns the list of all sought *k-ConnVertices* that include the vertex a. Thus, a can be removed from G (Line 5) for all subsequent calls to CombinationsWithV. The process is repeated on the remaining vertices (i.e., b, c, d, and e).

CombinationsWithV receives as input a vertex, the combination size, and the graph. In Line 1, it generates a special tree structure, which we call *combination tree* and discuss in Section 3.2. The algorithm uses the tree in Line 2 to collect the sought *k-ConnVertices* that include the vertex given as input. In the following sections, we describe how the tree is built for the graph in Figure 2 with the selected node a and combination size 4.

3.2 Building the combination tree

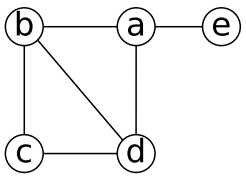
In this section, we study the process of building a combination tree. We introduce the algorithms, illustrate their application to a simple example, discuss their complexity, and establish their soundness and completeness.

CombinationTree (Algorithm 5) calls BuildTree (Algorithm 6). Together, these two algorithms yield a tree structure that we call the *combination tree*. We refer to the vertices of the combination tree as "nodes" in order to distinguish them from the vertices of the graph. The combination tree, rooted at a node n, is of maximum depth k. The node n corresponds to the graph vertex given as input, and each node n_t in the tree corresponds to some vertex of G, denoted Vertex(n_t). Two or more nodes in the generated tree may correspond to the same vertex in G. Further, any two nodes that are connected in the tree correspond to two connected vertices in the graph G. Figure 3 shows the tree generated by calling CombinationTree with the parameters a, $k = 4$, and the graph of Figure 2.

Algorithm 5: CombinationTree(v, k, G).

Input: v, k, G
Output: The root of a combination tree

```
1 root ← a new tree node corresponding to v
2 for i ← 0 to (k − 1) do list[i] ← ∅
3 list[0] ← {v}
4 BuildTree(root, 1, G, k)
5 return root
```



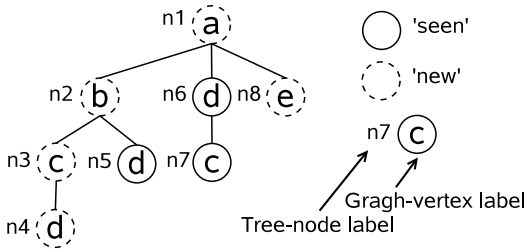


Figure 2: Simple graph. Figure 3: Combination tree for a, $k = 4$, and Fig. 2.

BuildTree proceeds in a depth-first manner. For each node n_t at depth l in the tree such that $l < k$, it adds, as children to n_t, all nodes n'_t that satisfy the following two conditions:

Condition 1: Vertex$(n'_t) \in$ Neighbors(Vertex(n_t)).

Condition 2: The vertex of n'_t is not the vertex of an ancestor, sibling, or a sibling of any ancestor of n_t.

Notably, BuildTree may visit a given vertex of the graph more than once, which occurs when the vertex can be reached through an alternative path from the root. The goal of Condition 2 is to:

1. Limit the size of the generated tree by pruning subtrees as argued in Proposition 3.6, and

2. Guarantee the existence of a subtree elsewhere in the combination tree that contains the vertices of the pruned subtree.

Indeed, Condition 2 above yields the following two propositions:

Proposition 3.1. *No two siblings of a tree node in the combination tree correspond to the same vertex of the graph.*

Proof. Follows directly from Condition 2. □

Proposition 3.2. *The maximum branching factor of the combination tree is bounded by the degree d of the graph.*

Proof. Given that the number of vertices in the graph is bounded and given Proposition 3.1, each tree node has a bounded number of children. □

In order to generate a tree that satisfies the two above-listed conditions, each node n_t in the tree maintains:

1. A list of the vertices of the ancestors of n_t in the tree, and
2. A list of the vertices of the siblings of the ancestors of n_t generated *before* the node n_t itself was generated.

A child for n_t is generated only when the corresponding vertex does not appear in the list of n_t. When the condition is not met, we say that the subtree rooted at this child is *omitted*.[2] When adding n'_t to the tree, the following operations are performed in sequence:

1. The vertex corresponding to n'_t is added to the list of n_t.
2. The list of n'_t is a copy of the list of n_t.

The pseudocode of BuildTree (Algorithm 6) uses two marking functions: MarkV for graph vertices and MarkN for tree nodes:

1. MarkV is used to mark a vertex of the graph as 'visited.' We assume that all graph vertices are initially marked as 'unvisited.'
2. MarkN is used to mark a node in the tree as 'new,' thus indicating that the corresponding graph vertex has not yet been encountered. Otherwise, the tree node is marked as 'seen' indicating that there already exists, in the tree, another node corresponding to the same graph vertex.

3.2.1 Illustrating the execution of CombinationTree

Below, we illustrate the generation of the tree shown in Figure 5, obtained by applying CombinationTree (Algorithm 5) on the vertex a, $k = 4$, and the graph of Figure 4. Line 1 of Algorithm 5 generates the root of the tree, $n1$, to correspond to the vertex a. Lines 2 and 3 initialize the vector array $list[]$. Line 4 calls BuildTree (Algorithm 6) with the two parameters $n1$ and 1 (for the tree depth) to build the children of the root.

In Line 1 of Algorithm 6, the list of 'ancestors' is copied from that of the parent. Thus, we have $list[1] = \{a\}$. Then, the subtrees corresponding to each of the neighbors of a (i.e., b, d and e) are built, see Figure 5.

First the vertex b is considered. Because $b \notin list[1]=\{a\}$, a node $n2$ corresponding vertex to b is added as a child to the root. The vertex b is added to $list[1]$ (i.e., $list[1]=\{a,b\}$) for the sake of the descendants of $n2$. $n2$ is marked as 'new' because b was not visited before. The vertex b is marked as 'visited.'

[2]This terminology is used in several of the proofs below.

Algorithm 6: BUILDTREE($n_t, depth, G, k$)

```
   Input: n_t, depth, G, k
 1 list[depth] ← list[depth − 1]
 2 foreach v' ∈ NEIGHBORS(VERTEX(n_t)) do
 3 |   if v' ∉ list[depth] then
 4 |   |   add n'_t as a child to n_t with VERTEX(n'_t) = v'
 5 |   |   list[depth] ← list[depth] ∪ {v'}
 6 |   |   if MARKV(v') ≠ visited then
 7 |   |   |   MARKN(n'_t) ← new
 8 |   |   |   MARKV(v') ← visited
 9 |   |   else
10 |   |   |   MARKN(n'_t) ← seen
11 |   |   end
12 |   |   if depth + 1 ≤ k then BUILDTREE(n'_t, depth + 1, G, k)
13 |   end
14 end
```

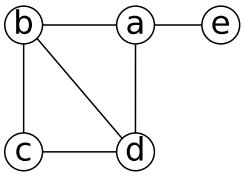


Figure 4: Simple example.

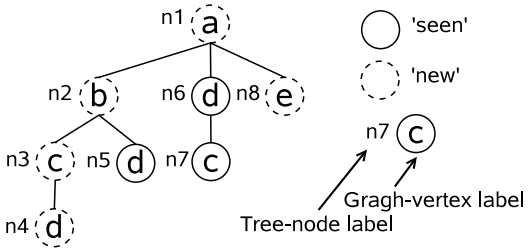


Figure 5: Combination tree rooted at vertex a with $k = 4$ for the graph in Figure 4.

Then, Line 12 calls Algorithm 6 recursively to generate the children of the node $n2$ corresponding to vertex b.

In the new recursive call to Algorithm 6, the set of ancestors at depth 2 is set to $\{a, b\}$ (i.e., $list[2]$={a,b}). Vertices a, c, and d are adjacent to vertex b. Because $a \in list[2]$, it is skipped. The node $n3$ is created for vertex c and added as a child of node $n2$. Then, the node $n3$ and the vertex c are appropriately marked as 'new' and 'visited,' respectively. Now, $list[2] = \{a, b, c\}$. The recursive call generates a child $n4$ for $n3$, where $n4$ corresponds to vertex d.

At this point, we have $depth = 3$. The condition in Line 12 is not satisfied, which ends the recursion. Back to node $n2$ at the previous level in the recursion, the second neighbor d of b is considered. The list of ancestors is $list[2] = \{a, b, c\}$ and $d \notin list[2]$. Therefore, a tree node $n5$ corresponding to the vertex d is added as a child of $n2$. The list of ancestors at this level $list[2]$ is updated to $\{a, b, c, d\}$. Because vertex d was visited in a previous recursive call, the node $n5$ is marked as 'seen.' Similarly the rest of the nodes are added to the tree resulting in the

tree shown in Figure 5.

3.2.2 Complexity of CombinationTree and BuildTree

We make the following observations about the combination tree. The depth of the generated tree is $(k-1)$.

If the maximum degree of the graph is d, the size of the list at *depth*=1 can be at most $2d$, and the size of the list at *depth*=$(k-1)$ inheriting from the ancestors is bounded by $\mathcal{O}(d \cdot k)$.

Because Algorithms 5 and 6 proceed in a depth-first manner, only the lists along the current path are stored. Thus, the space complexity of the lists is $\mathcal{O}(d \cdot k^2)$. These lists are stored in an $1 \times k$ array indexed by the depth of the node in the tree.

Proposition 3.3 (Complexity of CombinationTree and BuildTree.)**.** *The number of nodes in the tree is $\mathcal{O}(d^{(k-1)})$ assuming that the maximum degree of G is d. Thus, the time and space complexity of* CombinationTree *and* BuildTree *is $\mathcal{O}(d^{(k-1)})$.*

3.2.3 Soundness and completeness of combination trees

Below, we prove that:

1. The combination trees generated by ConSubG partition the set of all *k-ConnVertices* of the graph.
2. BuildTree terminates.
3. All connected subgraphs of size k including a given vertex are 'represented' in the combination tree built for this vertex.

Proposition 3.4 (Partitioning of combinations)**.** *No k-ConnVertices set can be extracted from two different combination trees generated by Algorithm 5.*

Proof. Every *k-ConnVertices* set extracted by CombinationsFromTree from the combination tree includes the vertex of the root of the tree. Moreover, once a combination tree has been processed, the vertex of the root is removed from the graph. Hence, the same combination cannot be extracted from subsequent combination trees. □

Proposition 3.5. *Let T be the combination tree generated by applying* CombinationTree *on v and G. For every connected subgraph G' induced on G by at most k vertices including v, the depth-first tree of G' rooted at v is isomorphic to a subgraph of T rooted at v. Moreover, every node in T is necessary for this property to hold.*

Proof. Let T be the combination tree rooted at v resulting from applying BuildTree on G, and let G' be an induced connected subgraph of G of at most k vertices including v. Let T' a depth-first traversal of G' rooted at v.

We prove that T' is isomorphic to a subgraph of T. Because T visits G in a depth-first manner without skipping already visited vertices except those violating Condition 2, a subgraph isomorphic to T' exists in T unless pruned by Condition 2. We next show that even after the application of Condition 2, there exists in T a subgraph T'' of T that is isomorphic to T'.

Consider a node n_p of T such that (1) $\textsc{Vertex}(n_p) \in G'$, (2) the vertices of the ancestors of n_p in T are in G', and (3) n_p is pruned by Condition 2. We show that the path from the root of T to n_p cannot be isomorphic to a path in T', but that there exists a path in T from the root to a node n'_p such that $\textsc{Vertex}(n_p)=\textsc{Vertex}(n'_p)$ that is isomorphic to a path in T'. Because n_p is pruned by Condition 2, then a node n'_p where $\textsc{Vertex}(n_p)=\textsc{Vertex}(n'_p)$ must exist in T where the ancestors of n'_p are all in G' (by Condition 2). Consequently, there are two paths p and p' in T where (1) p is the path from the root of T to n_p, (2) p' is the path from the root to n'_p, and (3) the vertices of the nodes in p and p' are all in G'. Thus, there must exist two paths in G' from v to v' that are isomorphic to p and p'. Further, only one of those two paths in G' appears in T', which is our depth-first traversal of G'. A path isomorphic to p cannot appear in T' because of the canonical ordering of the vertices is used to build the trees. Thus, there exists a path in T' that is isomorphic to p', and p' must be isomorphic to a path in T'. As a conclusion, the pruning by Condition 2 will maintain in T a tree isomorphic to T'.

Now, we prove that every node in T is necessary for the above property to hold. Consider a node $n \in T$, and let p be the path in T from the root to n. Let G' be the subgraph in G induced by the vertices of the nodes in p. Given the canonical ordering of the graph vertices, p is isomorphic to the depth-first tree of G'. Because no two siblings in T have the same vertex label, p is the only subgraph of T isomorphic to the depth-first tree of G'. Therefore, if n was removed from T, there will not be a subgraph of T that is isomorphic to the depth-first tree of G', and the above property will be lost. □

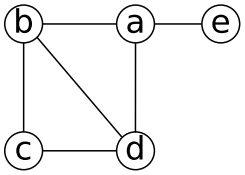


Figure 6: Simple example.

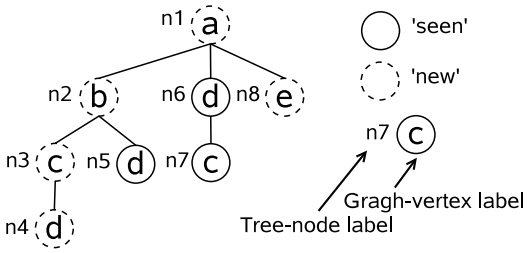


Figure 7: The tree rooted at vertex a for $k = 4$ for the graph in Figure 6.

Proposition 3.6. CombinationTree *terminates.*

Proof. BuildTree (Algorithm 6) traverses the graph in a depth-first manner without skipping already visited vertices. Thus, the termination of BuildTree

is a legitimate concern. The algorithm stops proceeding down a path under two conditions:

1. The condition in Line 12, which guarantees that the length of the 'current' path is always smaller than or equal to k, e.g. node $n4$ in Figure 7. Thus, the depth of the tree generated by Algorithm 6 is never larger than k.

2. The condition in Line 3 fails, which enforces Condition 2 of Section 3.2. Proposition 3.2 guarantees that the branching factor of the tree generated by Algorithm 6 is bounded.

Consequently, the size of the tree generated by BUILDTREE is bounded, and BUILDTREE terminates.

□

3.3 Extracting k-*ConnVertices* from a combination tree

COMBINATIONSFROMTREE (Algorithm 7) is recursive and calls itself at Line 11. It also calls the functions k-COMBINATIONS and k-COMPOSITIONS, and uses a new set operator $\otimes_t$.

- k-COMBINATIONS(i,s) generates all combinations of size i of the elements of a set s. We assume that each element in the generated set is ordered. For example,

  $$k\text{-COMBINATIONS}(2, \{n2, n6, n8\}) = \{\{\text{n2,n6}\},\{\text{n2,n8}\},\{\text{n6,n8}\}\}.$$

  BF-CONSUBG (after removing Line 8) is an obvious implementation for k-COMBINATIONS. Other implementations are reported in [38, 2]. Ours is described in [27].

- k-COMPOSITIONS generates all strings of length $size$ on the integer interval $[1,(Sum - size + 1)]$ such that the sum of the elements of a string is equal to Sum. For example,

  $$k\text{-COMPOSITIONS}(3, 4) = \{\{1, 1, 2\}, \{1, 2, 1\}, \{2, 1, 1\}\}.$$

  Because every element in the generated set is a string, the element is considered to be ordered. A recursive algorithm for k-COMPOSITIONS is attributed to Knuth [55]. Implementations are reported in [38, 2]. Our implementation is tree based and described in [27].

- The binary operator $\otimes_t$ operates on sets of sets and is discussed in Section 3.3.1.

Below, we formally define and analyze the operator $\otimes_t$, provide the pseudocode of 3.3, illustrate its execution on our running example, and discuss the implementation of the operator $\otimes_t$.

3.3.1 Defining of the $\otimes_t$ operator

We introduce the following definition for an operator that operates on two sets:

Definition 3.7 (UnionProduct)**.** We define the binary operator UnionProduct, denoted $\otimes$, as the operator that combines two sets of sets as follows:

$$S_1 \otimes S_2 = \{\ x \mid (x = s_1 \cup s_2) \wedge (s_1 \in S_1) \wedge (s_2 \in S_2)\ \} \tag{2}$$

UnionProduct is a cross-product-like operator in which two elements are combined by union instead of forming the usual tuple.

We refine the UnionProduct operator into a binary operator denoted $\otimes_t$, which we use in CombinationsFromTree (Line 15 of Algorithm 7). $\otimes_t$ operates on two sets of sets of nodes from a combination tree as follows:

$$S_1 \otimes_t S_2 = \begin{cases} \emptyset, \text{ if } S_1 = \emptyset \\ S_1, \text{ if } S_2 = \emptyset \\ \{x \mid (x = s_1 \cup s_2) \wedge (s_1 \in S_1) \wedge (s_2 \in S_2) \\ \quad \wedge (\forall i \in s_1, j \in s_2, \textsc{Vertex}(i) \neq \textsc{Vertex}(j)) \\ \quad \wedge ((\exists j \in s_2\, \textsc{MarkN}(j) = \text{`new'}) \vee (\forall i \in s_1, j \in s_2, l \in \textsc{Children}(i), \\ \qquad\qquad \textsc{Vertex}(j) \neq \textsc{Vertex}(l)))\}, \text{ otherwise}. \end{cases} \tag{3}$$

Let us explain the meaning of the two conditions in Expressions (3). The first condition is:

$$\forall i \in s_1, j \in s_2, \textsc{Vertex}(i) \neq \textsc{Vertex}(j). \tag{4}$$

This condition guarantees that no two nodes in an element of $S_1 \otimes_t S_2$ correspond to the same graph vertex. The goal is to guarantee that every element of $S_1 \otimes_t S_2$ has only nodes corresponding to distinct graph vertices. The second condition is the disjunction of the two following conditions:

$$\exists j \in s_2\, \textsc{MarkN}(j) = \text{`}new\text{'} \tag{5}$$

$$\forall i \in s_1, j \in s_2, l \in \textsc{Children}(i), \textsc{Vertex}(j) \neq \textsc{Vertex}(l). \tag{6}$$

The condition in Expression (5) guarantees that an element is added to $S_1 \otimes_t S_2$ when at least one of the tree nodes in s_2 is 'new,' that is, it corresponds to a vertex that had not been encountered before. The condition in Expression (5) is thus to ensure that elements not encountered before are included in $S_1 \otimes_t S_2$.

The intuition behind the condition in Expression (6) is as follows. When a tree node $j \in s_2$ corresponds to the same vertex as a child of a tree node $i \in s_1$, then the set of vertex labels obtained from the subtree rooted at j can also be obtained from the subtree rooted at i and from subtrees rooted at siblings, parents, and siblings of parents of i. Hence, $s_1 \cup s_2$ is omitted from $S_1 \otimes_t S_2$.

Note that, while the operator $\otimes$ is commutative, the operator $\otimes_t$, by definition, is associative but not commutative.

Proposition 3.8 (Time complexity of $S_1 \otimes_t S_2$.)**.** *The time complexity of $S_1 \otimes_t S_2$ is $\mathcal{O}(|S_1| \cdot |S_2| \cdot |s_1| \cdot |s_2|)$, where $|s_1|$ and $|s_2|$ are the sizes of the largest elements of $|S_1|$ and $|S_2|$ respectively.*

3.3.2 Pseudocode of CombinationsFromTree

CombinationsFromTree (Algorithm 7) takes as parameters a combination tree and a combination size k. It returns combinations of nodes of the tree that:

1. Include the root of the tree and
2. Correspond to the connected subgraphs of size k of the original graph.

Algorithm 7: CombinationsFromTree($tree, k$)

Input: $tree, k$
Output: A list of sets of nodes of the tree including the root node

```
1  t ← root_tree
2  lnodesets ← ∅
3  if k = 1 then return {t}
4  for i ← 1 to Min(|Children(t)|, (k − 1)) do
5      foreach NodeComb ∈ k-combinations(i,Children(t)) do
6          foreach string ∈ k-compositions(i, (k − 1)) do
7              fail ← false
8              for pos ← 1 to i do
9                  stRoot ← element in position pos in NodeComb
10                 size ← element in position pos in string
11                 S[pos] ←CombinationsFromTree(stRoot, size)
12                 if S[pos] = ∅ then fail ← true; break
13             end
14             if fail then continue
15             foreach combProduct in S[1] ⊗_t ··· ⊗_t S[i] do
16                 lnodesets ← lnodesets ∪ {combProduct ∪ {t}}
17             end
18         end
19     end
20 end
21 return lnodesets
```

3.3.3 Illustrating the execution of CombinationsFromTree

Consider the graph in Figure 8 and its corresponding combination tree shown in Figure 9. The tree is passed to Algorithm 7 with $k = 4$, yielding:

$$\{\{n1, n2, n3, n4\}, \{n1, n2, n3, n8\}, \{n1, n2, n5, n8\}, \{n1, n6, n7, n8\}\}, \quad (7)$$

which is mapped in a straightforward manner to yield the following combinations of vertices:

$$\{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}. \quad (8)$$

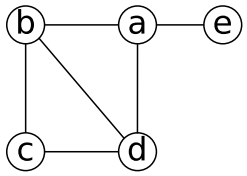


Figure 8: Simple graph.

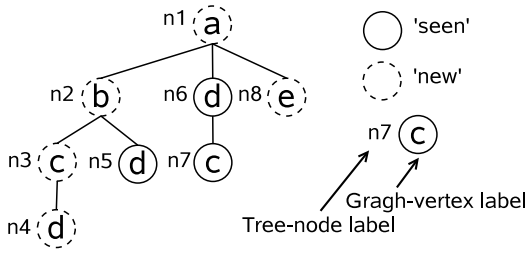


Figure 9: The tree rooted at vertex a for $k = 4$ for the graph in Figure 8.

Below, we explain step by step how CombinationsFromTree reaches the result in Expression (7). The call to CombinationsFromTree($n1$,4) yields three iterations for i=1, 2, and 3 in Line 4. Thus, in Line 5, *NodeComb* iterates over the elements of the following sets:

- For $i = 1$, k-combinations(1,$\{n2, n6, n8\}$)= $\{\{n2\}, \{n6\}, \{n8\}\}$,
- For $i = 2$, k-combinations(2,$\{n2, n6, n8\}$)= $\{\{n2, n6\}, \{n2, n8\}, \{n6, n8\}\}$, and
- For $i = 3$, k-combinations(3,$\{n2, n6, n8\}$)= $\{\{n2, n6, n8\}\}$.

At Line 6, *string* iterates over the elements of the following sets:

- For $i = 1$, k-compositions(1,3)=$\{\{3\}\}$,
- For $i = 2$, k-compositions(2,3)=$\{\{1,2\},\{2,1\}\}$, and
- For $i = 3$, k-compositions(3,3)=$\{\{1,1,1\}\}$.

In order to continue our illustration of the operation of CombinationsFromTree, we introduce the following definition:

Definition 3.9 (Configuration)**.** We define a *configuration* to be a set of 3-tuple $\langle j, N_j, C_j \rangle$ where:

1. j is a positive integer denoting the number of subtrees of the combinations tree to consider,
2. N_j is an ordered set of size j ($|N_j| = j$) of tree nodes that have the same parent in the combination tree, and
3. C_j is an ordered set of positive integers ($|C_j| = j$). Each integer in C_j specifies the size of the combination of tree nodes to be extracted from the subtree rooted at the node at the same position in N_j.

Examples of configurations in Figure 9 are

$$\langle 1, \{n2\}, \{3\}\rangle, \langle 2, \{n2, n6\}, \{1, 2\}\rangle, \langle 3, \{n2, n6, n8\}, \{1, 1, 1\}\rangle. \tag{9}$$

The three nested loops from Line 4 to Line 16 generate all configurations for the children of a given root (Lines 4, 5, and 6), generate the combinations of tree nodes from each configuration (Line 11), then combine the resulting combinations within each configuration (Line 15). Below, we illustrate this process for i=1, 2, and 3.

For $i = 1$**,** $NodeComb \in \{\{\text{n2}\}, \{\text{n6}\}, \{\text{n8}\}\}$, and $string \in \{\{3\}\}$. We have three configurations at this point:

$$\langle 1, \{n2\}, \{3\}\rangle, \langle 1, \{n6\}, \{3\}\rangle, \langle 1, \{n8\}, \{3\}\rangle. \tag{10}$$

For pos=1, Line 11 calls CombinationsFromTree on each node appearing in a configuration as follows:

- CombinationsFromTree($n2$,3) returns $S[1]= \{\{n2, n3, n4\}\}$.
- CombinationsFromTree($n6$,3) returns $S[1]= \emptyset$.
- CombinationsFromTree($n8$,3) returns $S[1]= \emptyset$.

Given that each configuration has only one node, the operator $\otimes_t$ is not applied. Given that the first configuration yields one element and the second and third configurations yield empty results, Line 15 is called only once for $i = 1$, yielding:

$$combProduct \quad = \quad \{\{n2, n3, n4\}\} \tag{11}$$

For $i = 2$**,** $NodeComb \in \{\{n2, n6\}, \{n2, n8\}, \{n6, n8\}\}$, and $string \in \{\{1,2\}, \{2,1\}\}$. We have six configurations at this point:

$$\begin{array}{l} \langle 2, \{n2, n6\}, \{1, 2\}\rangle, \langle 2, \{n2, n6\}, \{2, 1\}\rangle, \\ \langle 2, \{n2, n8\}, \{1, 2\}\rangle, \langle 2, \{n2, n8\}, \{2, 1\}\rangle, \\ \langle 2, \{n6, n8\}, \{1, 2\}\rangle, \langle 2, \{n6, n8\}, \{2, 1\}\rangle. \end{array} \tag{12}$$

Line 11 calls CombinationsFromTree on each node appearing in a configuration as follows:

1. For configuration $\langle 2, \{n2, n6\}, \{1, 2\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n2$,1) returns $S[1]$=$\{\{n2\}\}$.
   - pos=2, CombinationsFromTree($n6$,2) returns $S[2]$=$\{\{n6$,$n7\}\}$.
2. For configuration $\langle 2, \{n2, n6\}, \{2, 1\}\rangle$, we have the following calls:
   - pos=1, CombinationsFromTree($n2$,2) returns $S[1]$=$\{\{n2$,$n3\}$, $\{n2,\ n5\}\}$.
   - pos=2, CombinationsFromTree($n6$,1) returns $S[2]$=$\{\{n6\}\}$.

3. For configuration $\langle 2, \{n2, n8\}, \{1, 2\}\rangle$, we have the following calls:
   - *pos*=1, CombinationsFromTree($n2$,1) returns $S[1]$=$\{\{n2\}\}$.
   - *pos*=2, CombinationsFromTree($n8$,2) returns $S[2]$= $\emptyset$.
4. For configuration $\langle 2, \{n2, n8\}, \{2, 1\}\rangle$, we have the following calls:
   - *pos*=1, CombinationsFromTree($n2$,2) returns $S[1]$=$\{\{n2$,$n3\}$, $\{n2$,$n5\}\}$.
   - *pos*=2, CombinationsFromTree($n8$,1) returns $S[2]$= $\{\{n8\}\}$.
5. For configuration $\langle 2, \{n6, n8\}, \{1, 2\}\rangle$, we have the following calls:
   - *pos*=1, CombinationsFromTree($n6$,1) returns $S[1]$=$\{\{n6\}\}$.
   - *pos*=2, CombinationsFromTree($n8$,2) returns $S[2]$= $\emptyset$.
6. For configuration $\langle 2, \{n6, n8\}, \{2, 1\}\rangle$, we have the following calls:
   - *pos*=1, CombinationsFromTree($n6$,2) returns $S[1]$=$\{\{n6$,$n7\}\}$.
   - *pos*=2, CombinationsFromTree($n8$,1) returns $S[2]$= $\{\{n8\}\}$.

Hence, Line 15 is called only four times because two of the above results for $pos = 2$ are empty:

$$\begin{aligned} combProduct &= \{\{n2\}\} \otimes_t \{\{n6, n7\}\} \\ &= \emptyset \qquad (13) \\ combProduct &= \{\{n2, n3\}, \{n2, n5\}\} \otimes_t \{\{n6\}\} \\ &= \emptyset \qquad (14) \\ combProduct &= \{\{n2, n3\}, \{n2, n5\}\} \otimes_t \{\{n8\}\} \\ &= \{\{n2, n3, n8\}, \{n2, n5, n8\}\} \qquad (15) \\ combProduct &= \{\{n6, n7\}\} \otimes_t \{\{n8\}\} \\ &= \{\{n6, n7, n8\}\} \qquad (16) \end{aligned}$$

For $i = 3$**,** $NodeComb \in \{\{n2, n6, n8\}\}$ and $string \in \{\{1, 1, 1\}\}$. We have one configuration at this point: $\langle 3, \{n2, n6, n8\}, \{1, 1, 1\}\rangle$. Line 11 calls CombinationsFromTree on each node in this unique configuration with the corresponding results:

- *pos*=1, CombinationsFromTree is called with ($n2$,1), which returns $S[1]$= $\{\{n2\}\}$.
- *pos*=2, CombinationsFromTree is called with ($n6$,1), which returns $S[2]$= $\{\{n6\}\}$.
- *pos*=3, CombinationsFromTree is called with ($n8$,1), which returns $S[3]$= $\{\{n8\}\}$.

Hence, Line 15 is called only once to combine the results of the above calls, yielding:

$$\begin{aligned} combProduct &= (\{\{n2\}\} \otimes_t \{\{n6\}\}) \otimes_t \{\{n8\}\} \\ &= \emptyset \qquad (17) \end{aligned}$$

At the end, adding the root node n_1, we have

$$lnodesets = \{\{n1, n2, n3, n4\}, \{n1, n2, n3, n8\}, \{n1, n2, n5, n8\}, \{n1, n6, n7, n8\}\}.$$

Thus, the set of combinations of four vertices extracted from the combination tree of Figure 9 is $\{\{a, b, c, d\}, \{a, b, c, e\}, \{a, b, d, e\}, \{a, d, c, e\}\}$.

3.3.4 Implementation of the $\otimes_t$ operator

In general, the process of computing $S_1 \otimes_t S_2$, where S_1 and S_2 are sets of sets, can be executed in two steps:

1. The computation of $S_1 \otimes S_2$ as specified in Expression (2), and
2. The removal from the resulting set of those elements that do not satisfy the conditions of $\otimes_t$ specified in Expression (3).

We propose to compute $S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$ by modeling the problem as a Constraint Satisfaction Problem (CSP) [10]. A CSP, $\mathcal{P}=(\mathcal{V}, \mathcal{D}, \mathcal{C})$, is fully defined by specifying the set of variables $\mathcal{V}$, the set of their respective domains $\mathcal{D}$, and the set of constraints $\mathcal{C}$ that restrict the allowed combinations of values to variables. A solution to a CSP is an assignment of a value to each variable such that all constraints are simultaneously satisfied. In general, the task is to find one or all solutions to the CSP. CSPs are commonly used to model combinatorial problems and solve using advanced search techniques and constraint propagation algorithms [10]. We model the execution of the operator $\otimes_t$ over a sequence of sets $S_1, S_2, \ldots, S_n$ as a CSP as follows. A variable V_i of the CSP is (the 'name' of) the set $S_i \in \{S_1, S_2, \ldots, S_n\}$. The domain of the variable V_i is the definition of the corresponding set. A binary constraint is applied to every two variables V_i and V_j such that $i < j$. It constrains the acceptable combinations of values for V_i and V_j to satisfy the conditions[3] specified by Expression (3). We solve the CSP using exhaustive backtrack search [10], which yields all solutions to the problem, thus the set $S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$. The ordering of the variables in the search is fixed and static, and follows that of the sets S_i. We use a standard partial-lookahead technique to improve the performance of the search known as *forward checking* (FC) [17]. In summary, backtrack search with FC operates as follows:

1. A variable V_i is assigned a value from its domain. Initially $i = 1$.
2. All variables $V_{j>i}$ are 'revised' given V_i. To revise a variable V_j given a variable V_i, we remove all values in the domain of V_j that do not have at least one consistent value in the domain of V_i.

The search process is repeated by assigning any unassigned variable, until all the variables have been assigned. When all the variables are assigned, the assignment is a solution to the CSP, and consequently a valid element of

[3] These conditions are discussed in the proof of Theorem 3.11.

$S_1 \otimes_t S_2 \otimes_t \ldots \otimes_t S_n$. If, at some point during search, the domain of any of the unassigned variables is empty or after a solution is found, we backtrack chronologically to consider alternative assignments to the variables. The process ends when all the values for the first variable have been considered.

3.3.5 Completeness and soundness of CombinationsFromTree

We first establish that CombinationsFromTree generates only connected subgraphs, then use this result to prove its soundness and completeness.

Proposition 3.10. *Every combination of tree nodes generated by* CombinationsFromTree *induces a connected subgraph of the combination tree and corresponds to a set of vertices that induce a connected subgraph in the graph.*

Proof. We first prove that every combination generated by CombinationsFromTree induces a connected subgraph in the combination tree. Then, we prove that the vertices corresponding to the generated combination of tree nodes form a connected subgraph of the original graph.

The proof is by induction. When Algorithm 7 is called on a tree of depth zero, the only combination returned is the root, which induces a connected subgraph in the tree. Thus, we established the base case. We form the inductive hypothesis as follows: all generated combinations from tree of depth $(d-1)$ are connected. Then, we state and prove the inductive step: If all generated combinations from tree of depth $(d-1)$ are connected, then the combinations returned from the tree of depth d are also connected. When Algorithm 7 is called on a tree of depth d rooted at *root*, it is recursively called on the children of the *root*. Each combination generated from the tree is formed of combinations of nodes obtained from calls to Algorithm 7 on subtrees of depth at most $(d-1)$. Each of those subtrees are rooted at a child of *root*, hence each combination returned from the subtrees includes a child of *root*, and is connected. When these combinations are combined, and *root* is added to them, the result is a combination of nodes that induces a connected subgraph in the combination tree.

An edge between two nodes in the combination tree exists when the graph vertices to which the tree nodes correspond are adjacent. Thus, every edge in the combination tree corresponds to an edge in the original graph. Consequently, the set of vertices corresponding to a combination of connected tree nodes are also connected in the graph. Thus, the proof holds by the principle of mathematical induction. □

Theorem 3.11 (CombinationsFromTree is sound and complete.)**.** *Given a combination tree generated from a graph G with vertex v and the parameter k,* CombinationsFromTree *generates all unique k-ConnVertices sets that include the vertex v.*

Proof. First, we prove that CombinationsFromTree (Algorithm 7) generates all k-*ConnVertices* sets that include v. Consider a combination of k vertices of

G that includes vertex v and induces a connected subgraph of G. Consider also the combination tree T generated from G with vertex v and the parameter k. Proposition 3.5 insures that, for every connected subgraph G' induced on G by at most k vertices including v, the depth-first tree of G' rooted at v is isomorphic to a subgraph of T rooted at v. Therefore, considering all possible connected subgraphs of T of size k rooted at v guarantees that all connected subgraphs in G that include vertex v are considered.

The argument now shifts to showing that all possible induced subtrees in the combination tree including the root are considered in Line 15 of Algorithm 7. Let *root* be the root of the combination tree considered. The three loops in Lines 4, 5, and 6 ensure that Algorithm 7 systematically enumerates all the configurations[4] that lead to combinations of size $(k-1)$. Using these configurations, Algorithm 7 is recursively called in Line 11 on subtrees rooted at the children of *root*, and then the results are passed to the operator $\otimes_t$ to generate combinations of tree nodes of size $(k-1)$. The task is thus now to prove that Line 15 produces all *k-ConnVertices* sets.

The operator $\otimes_t$ is applied to the sets $S[pos]$, where *pos* varies from one to the number of subtrees in the considered configuration. The sets $S[pos]$ are produced by Algorithm 7 (see Line 11) from subtrees t_i rooted at children of *root*. Further, each element in $S[pos]$ is a combination of tree nodes and induces a connected subgraph in the tree by Proposition 3.10. Each element in the set produced at Line 11 by the application of the $\otimes_t$ operator is a set of tree nodes of size $(k-1)$. The set produced at Line 15 is, by the definition[5] of operator $\otimes_t$, a subset of the cross-product-like operation of the sets to which it is applied. The elements that are not removed from the 'complete' cross-product are those that verify the conditions specified in Expression (3). The task is now to prove that:

- The elements 'ruled out' by the Expression (3) yield combinations of tree nodes that are already in the set (i.e., they are 'duplicate' elements).
- No 'duplicate' elements are present in the resulting set.

Let us return to the application of the operator $\otimes_t$ in Algorithm 7. Let $S[pos]$ and $S[pos+1]$ be two sets of tree-nodes combinations obtained from recursive calls to Algorithm 7 and to which Algorithm 7 applies the operator $\otimes_t$. At the lowest level of the recursive calls, the tree roots given as arguments to CombinationsFromTree in Line 11 of Algorithm 7 are single tree nodes. In Line 15, the result from subtrees that have the same parent are combined using the operation $\otimes_t$. Hence, the Expression (6) must hold for every pair of nodes in an element of $S[pos] \otimes_t S[pos+1]$,

Let *comb* and *comb'* be two combinations generated from a combination tree such that the set of tree nodes in *comb* is different from that in *comb'*, but such that *comb* and *comb'* correspond to the same set of graph vertices. Given a combination of tree nodes, which is element of $S[pos] \otimes_t S[pos+1]$,

[4] See Definition 3.9.
[5] See Definition 3.7.

we showed in Proposition 3.10 that the nodes in the combination are connected in the tree. By construction, the root of the combination tree is one of the nodes in the combination. Hence, both *comb* and *comb*′ induce connected sets of nodes in the tree, and include the root node of the combination tree. Given that $comb \neq comb'$, there must necessarily exist a tree node n that has a child n_c, such that $n \in comb$, $n \in comb'$, $n_c \in comb$ and $n_c \notin comb'$. Because *comb* and *comb*′ both correspond to the same set of graph vertices, there must be a node $n'_c \in comb'$ such that VERTEX(n'_c) =VERTEX(n_c), which is impossible because it violates Expression (6). Indeed, we have the nodes $n, n'_c \in comb'$ such that $\exists n_c \in$CHILDREN(n) such that VERTEX(n_c)=VERTEX(n'_c). Because of this impossibility, we conclude that no two combinations of tree nodes in $S[pos] \otimes_t S[pos+1]$ can correspond to the same set of graph vertices. In conclusion, no two elements in $S[pos] \otimes_t S[pos+1]$ are the same. □

4 Memoization

At a node n at depth *depth* in the combination tree, COMBINATIONSFROMTREE (Algorithm 7) recursively calls itself (Line 11) at most the following number of times:

$$\sum_{i=1}^{k-depth-1} \left(id^i (k - depth - i - 1)^{(i-1)} \right), \tag{18}$$

where d denotes the degree of the graph. At each call, the arguments passed to COMBINATIONSFROMTREE are a child of n and a value of *size* ranging from 1 to $(k - depth - 1)$. Hence, there are at most $d \cdot (k - depth - 1)$ distinct calls that can be made from a single node at *depth* in the combination tree.

To avoid executing the redundant calls COMBINATIONSFROMTREE, the first time Algorithm 7 is called on a tree node with a given combination size, the result is stored in the node. The next time the call is made on the same node with the same combination size, the stored result is retrieved and used, which avoids re-executing the call. Hence we store at most $(k - depth - 1)$ sets of *k-ConnVertices* sets at each node. These *k-ConnVertices* sets are stored in an array indexed by the size of the combination.

Likewise, the results of the calls to k-COMPOSITIONS are also memoized in a data structure that is global to CONSUBG (Algorithm 3). The former memoization (i.e., in COMBINATIONSFROMTREE) proved to be extremely effective in reducing running time. The latter (i.e., in k-COMPOSITIONS) was also quite effective but to a lesser extent than the former. Neither introduced running-time overhead. The memory overhead of the former dominates that of the latter.

5 Analysis of CONSUBG

Because there can be an exponential number in k of connected subgraphs with k vertices, the worst-case time and space complexities remain prohibitive. For

this reason, we forego a formal complexity analysis and rely on an empirical evaluation of our algorithm for classes of graphs of interest.

Below, we prove the correctness of our algorithm.

Theorem 5.1 (Soundness and completeness of ConSubg)**.** ConSubg *(Algorithm 3) generates all unique* k-ConnVertices *sets from the combination trees.*

Proof. Let v be the first vertex considered in Algorithm 3. Algorithm 7 returns all unique combinations of vertices including v, as established by Theorem 3.11. Therefore, any k-*ConnVertices* set that includes v is generated from the combination tree with a root node labeled with vertex v.

After removing vertex v from the graph, Algorithm 3 repeats the same process for a vertex v' chosen arbitrarily from the graph. All k-*ConnVertices* sets starting at v' in the updated graph are generated in a similar manner. Thus, all k-*ConnVertices* sets in the graph that include v' are generated either when Algorithm 3 processes v (and those combinations would thus include v) or when it processes v'. Hence, no k-*ConnVertices* set that includes v' can be missed. Finally, all k-*ConnVertices* sets for each of the remaining graph vertices are generated in a similar manner. Indeed, the k-*ConnVertices* sets for a given vertex are generated by Algorithm 3 at any point either before the vertex is considered or when it is processed (at the latest).

Proposition 3.4 asserts that no k-*ConnVertices* set can be generated from two distinct combination trees. Theorem 3.11 guarantees that all k-*ConnVertices* sets generated from a combination tree are unique. Therefore, all k-*ConnVertices* sets generated by Algorithm 3 are also unique. □

6 Empirical evaluations

Below, we compare the performance of the proposed algorithm, ConSubg, to that of the brute-force algorithm, BF-ConSubg, and its localized variant, LBF-ConSubg. We measured the CPU time of executing the algorithms on graphs with a fixed degree (Section 6.1), scale-free graphs (Section 6.2), and graphs of constraint satisfaction benchmarks (Section 6.3). For random graphs (i.e., fixed degree and scale-free graphs), we generated 30 instances per data point. In all cases, we averaged the results on the instances that completed within one hour of CPU time. A missing data point corresponds to an experiment that did not terminate within the one-hour time limit.

6.1 Graphs of a fixed degree

Because ConSubg is designed to exploit the structure of the graph, one would expect that its performance would be worse on graphs where all vertices have the same degree (i.e., graphs lacking structure). For this purpose, we wrote a generator to generate connected random graphs where all vertices have the same degree. Given the number of vertices of a graph and the degree of the graph (which is a constant less than the number of vertices), we first determine

the number of edges in the graph using the hand-shaking theorem. Then, we repeat the following steps until each edge is connected to two vertices. We select an edge that has not been connected to any vertices. Then, we select two random vertices, and connect them with the edge only if the two vertices are not already adjacent and if the degree of each of them is less than the specified degree entered as input. If the resulting graph is not connected, we discard it and repeat the process.

To test our algorithms on the graphs generated as described above, we conducted the experiments summarized in Table 2. In those experiments, we in-

Table 2: Experiments on random graphs of a fixed degree.

Experiment	$\|V\|$	**Degree**	**Size of subgraphs**	**Figure**
I	100	10	$k = 3, 4, 5, 6, 7$	Figure 10
	100	40	$k = 3, 4, 5, 6$	Figure 11
II	{100,150,...,900}	10	$k = 4$	Figure 12
	{100,150,...,900}	40	$k = 4$	Figure 13
III	100	{5,10,...,40}	$k = 4$	Figure 14
	300	{5,10,...,40}	$k = 4$	Figure 15
	400	{5,10,...,40}	$k = 4$	Figure 16

vestigated the effect of increasing the size of the subgraph on sparse and dense graphs of 100 vertices (Experiment I), the effect of increasing the number of vertices for a fixed size of the subgraph on sparse and dense graphs (Experiment II), and the effect of increasing the density of the graph for a fixed subgraph size and on graphs with 100, 300, and 400 vertices.

Experiment I Figure 10 shows the performance of the three algorithms on sparse graphs for increasing combination values k. On those graphs, ConSubg clearly outperforms BF-ConSubg and LBF-ConSubg. Notably, BF-ConSubg and LBF-ConSubg fail to terminate within the CPU time limit for $k = 7$ while ConSubg succeeds. Figure 11 shows the only experiment where ConSubg fails to terminate because of the memory limitation while BF-ConSubg and LBF-ConSubg do. This situation occurs for $k = 6$. Note that the graph density in this experiment is 40.40%. Clearly, ConSubg fails to handle large values of k on dense graphs because of its memory requirements. However, note that large dense graphs are not of much use in practice because they have a prohibitively large number of connected subgraphs, which are challenging to store and operate on even if we were able to generate them.

Experiment II Figures 12 and 13 show that our new algorithm ConSubg vastly outperforms the brute-force algorithm BF-ConSubg and its localized version LBF-ConSubg as the size of the network increases. Indeed, on graphs of degree 10, BF-ConSubg and LBF-ConSubg cannot handle graphs beyond

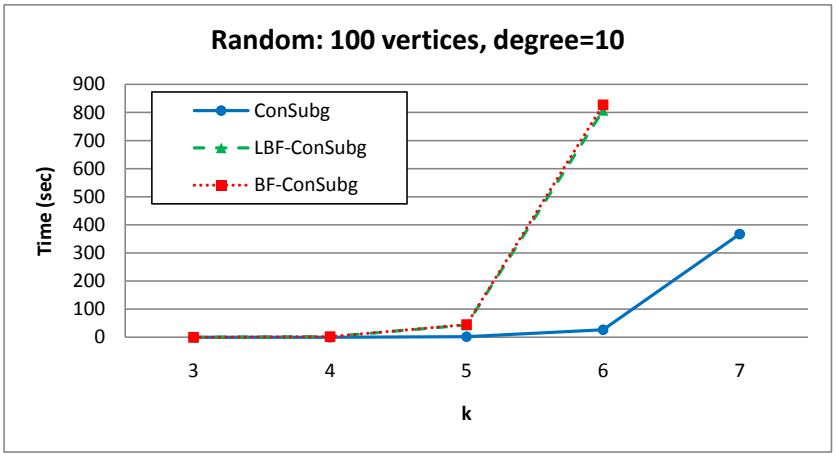


Figure 10: Increasing k with $|V|$=100 and of degree 10.

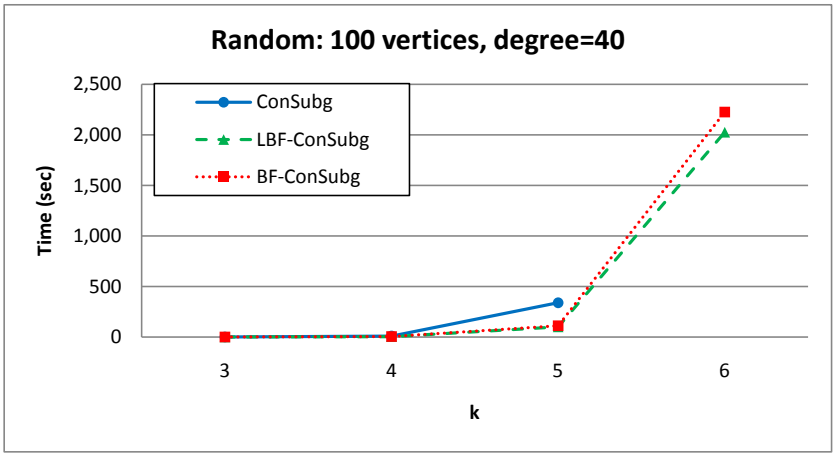


Figure 11: Increasing k on graphs with $|V|$=100 and of degree 40.

650 vertices. On graphs of degree 40, they both stop at graphs with 600 vertices. ConSubg easily scales to larger graphs in both cases and its cost remains relatively negligible.

Experiment III Figures 14, 15, and 16 show the performance of the algorithms as the degree of the graph grows on graphs with 100, 300, and 500 vertices respectively and for a fixed combination size $k = 4$. Here again, we see that the performance of our new algorithm ConSubg deteriorates as the density of the graph increases, again reaffirming that ConSubg is not suitable for dense

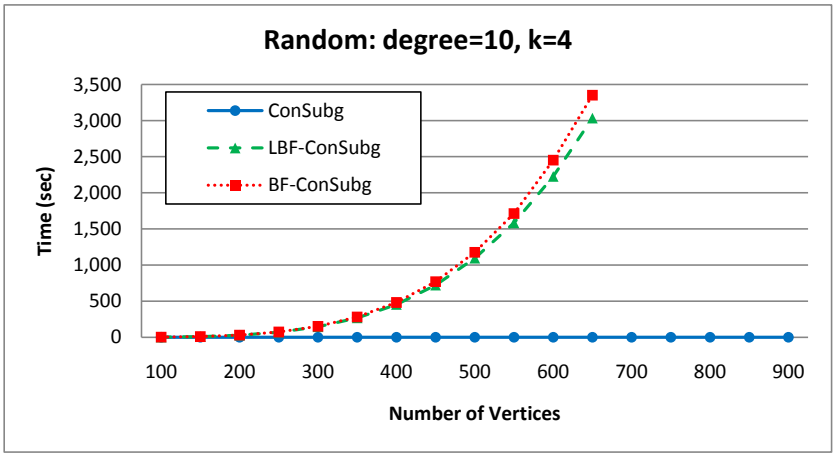


Figure 12: Increasing the number of vertices for $k = 4$ and graphs of degree 10.

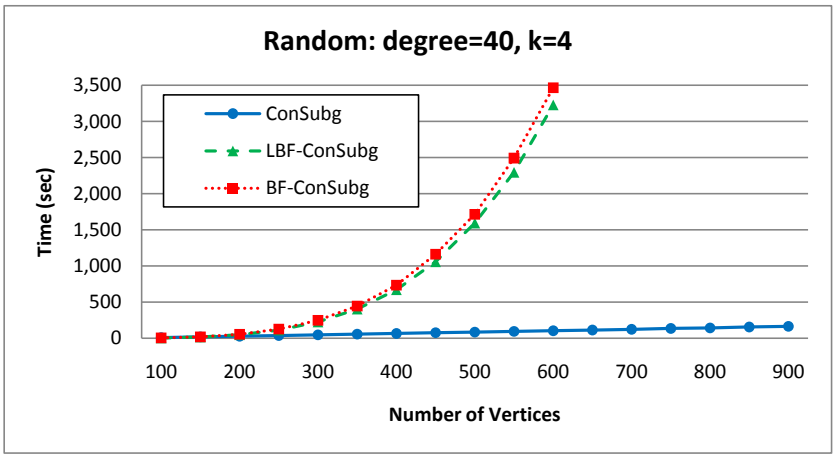


Figure 13: Increasing the number of vertices for $k = 4$ and graphs of degree 40.

graphs (see Figure 14). However, as the number of vertices increases, the brute-force algorithms BF-ConSubg and its localized version LBF-ConSubg are an order of magnitude more costly than ConSubg (see Figures 15 and 16). The main drawback of BF-ConSubg and LBF-ConSubg is that they generate many subgraphs that are not connected and, thus, must be discarded after they are generated, which ConSubg is designed to not do. Incidentally, in Figures 14, 15, and 16 the number of combinations generated by LBF-ConSubg and BF-ConSubg is constant for all degree values. However the corresponding curves present a slight positive slope. This slight slope can be attributed to the cost of testing the connectivity of the generated combinations.

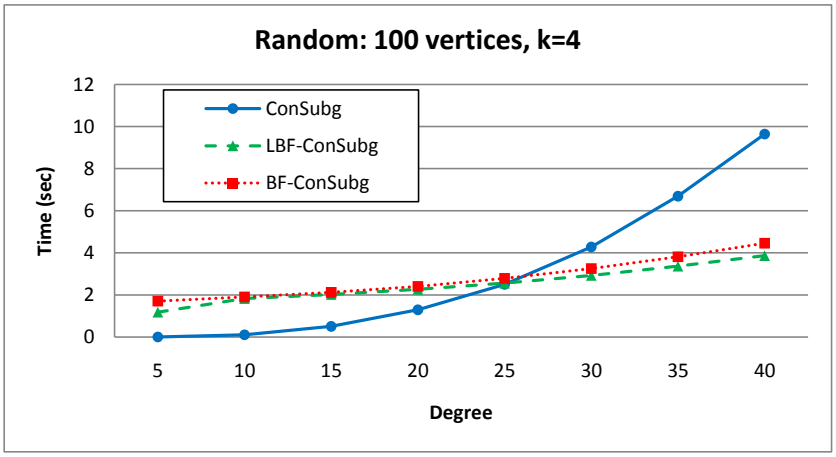


Figure 14: Increasing the degree of the vertices for $|V|$=100 and $k = 4$.

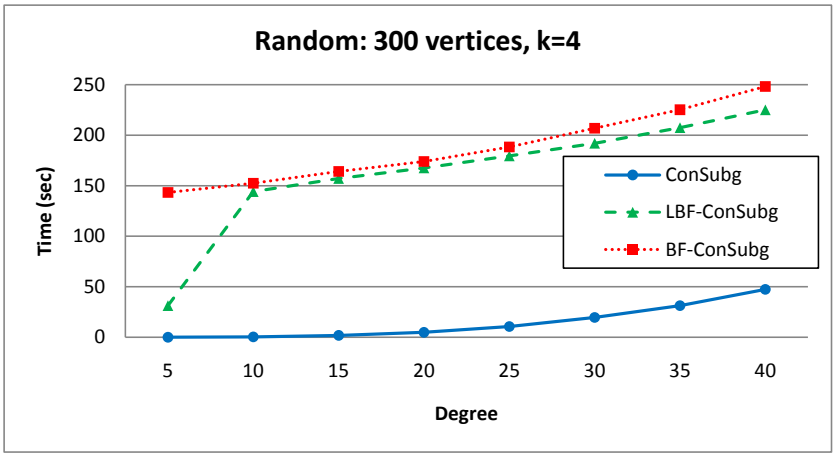


Figure 15: Increasing the degree of the vertices for $|V|$=300 and $k = 4$.

In summary and in all our experiments on randomly generated graphs of a fixed degree, ConSubg usually and largely outperforms LBF-ConSubg, which always outperforms BF-ConSubg. In particular:

1. The performances of LBF-ConSubg and BF-ConSubg are notably similar, while the localized version is always slightly quicker than, or at least as quick as, the original brute-force algorithm.

2. As the density of the graph increases, the likelihood that a given combination of k vertices induces a connected subgraph increases, and the benefit of exploiting the structure of the graph obviously decreases. At some

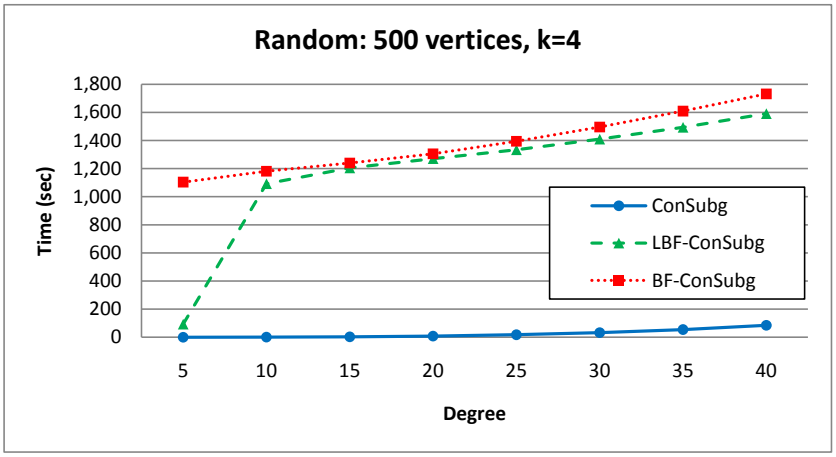


Figure 16: Increasing the degree of the vertices for $|V|$=500 and $k = 4$.

point, the cost of building the data structures necessary for ConSubg becomes detrimental. Note that, of all the experiments we conducted, this problem is visible only in the experiments shown in Figures 11 and 16 where the graph density is 40.40%, which is considered a high-density graph in practice.

6.2 Scale-free graphs

Scale-free graphs are commonly thought to model social networks and have received an increased attention in recent years. To Generate Scale-Free Networks, we used the procedure `scale_free_graph` from the open-source software `NetworkX`.[6] The procedure is based on the model proposed in [7]. We chose the default parameters for `scale_free_graph` (alpha=0.41, beta=0.54, gamma=0.05, delta_in=0.2, and delta_out=0) to generate the directed graph, and used the procedure `to_undirected` in NetworkX to obtain the corresponding undirected graph. We generated undirected graphs of 100, 200, ..., 900 vertices.

Increasing the number of vertices Figure 17 shows the CPU time needed to generate all subgraphs of size four (i.e., $k = 4$) by each of the three algorithms compared. We see that both our algorithms ConSubg and LBF-ConSubg scale significantly better with increasing number of vertices than the brute-force algorithm BF-ConSubg. Also, ConSubg clearly outperforms LBF-ConSubg.

[6] `NetworkX 1.3` http://networkx.lanl.gov/.

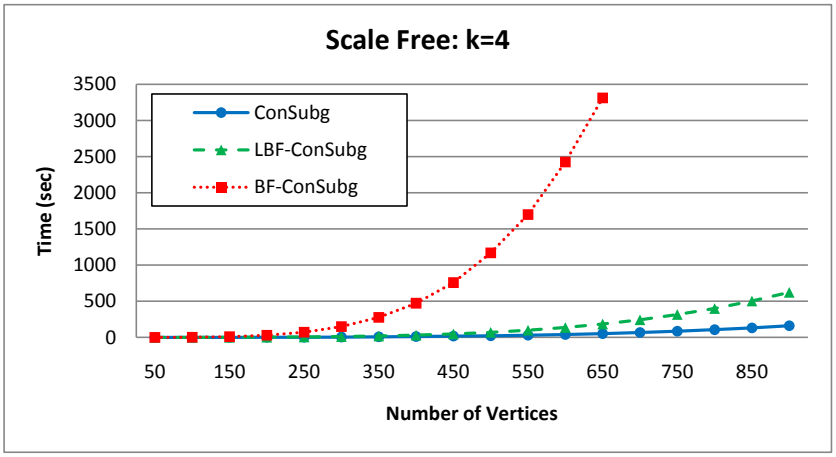


Figure 17: Increasing the number of vertices with $k = 4$ in scale-free networks.

Increasing k, the combination size Figure 18 compares the performance of the three algorithms on scale-free networks of 100 vertices as k grows. The brute-force algorithm, BF-ConSubg, does not terminate within the time limit of one hour for $k = 7$, and the performance of its localized version, LBF-ConSubg, is an order of magnitude worse than that of ConSubg.

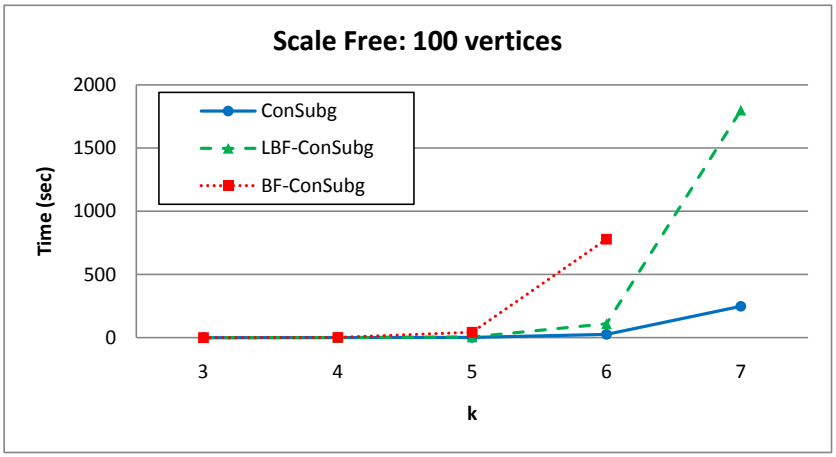


Figure 18: Increasing k in scale-free networks of 100 vertices.

In summary, ConSubg clearly outperforms its competitors on scale-free graphs, which are of practical importance. Interestingly, the performance of the localized variant of the brute-force algorithm, LBF-ConSubg, is significantly better than that of the original algorithm, albeit it is not as good as that of

ConSubg. Thus, while localization helps, it does not take full advantage of the problem structure.

6.3 CSP graphs

We examined the benchmarks of constraint satisfaction problems (CSPs) used in the 2009 Constraint Solver Competition,[7] and considered the dual graphs of 1689 CSP instances. Given that our algorithm is best suited for sparse graphs, it is appropriate to report the density of those benchmarks. 56.5% of the dual graphs of the 1689 benchmark instances have density less than or equal to 15%. The dual graphs can be reformulated, without loss of information, into equivalent graphs by removing redundant edges [10]. We applied the algorithm proposed in [20, 21] to remove redundant edges. The resulting minimal dual graphs that have density less than or equal to 11% constitute 79.7% of all tested instances. Consequently, it is fair to say that most benchmark problems, including the most challenging ones, have sparse dual graphs.

We executed ConSubg, LBF-ConSubg, and BF-ConSubg with $k = 5$ on the dual graphs of those 1689 instances after removing redundancies. Each experiment on a single instance was limited to a one hour. Table 3 shows a summary of the results. Below we relate some observations:

Table 3: Summary of results on 1689 CSP benchmark instances.

Number of instances	ConSubg	LBF-ConSubg	BF-ConSubg
Completed	1633	1602	918
Not completed	56	87	771
Algorithm performs best	1296	35	0
Completed by no algorithm	21		
Missed by only ConSubg	35		

- ConSubg is clearly the champion, both in terms of the number of instances it solves (1633 for ConSubg versus 1602 for LBF-ConSubg and 918 for BF-ConSubg) and the number of instances on which it performs as good as, or better than, the other two algorithms (1296 for ConSubg versus 35 for LBF-ConSubg and 0 for BF-ConSubg).

- ConSubg failed to complete the 56 instances because it ran out of memory space well before the one-hour time limit imposed on the experiments. However, LBF-ConSubg and BF-ConSubg failed to complete 87 and 771 instances respectively only because of the time limitation.

- ConSubg does not terminate within one hour processing time on 35 instances that were completed by both LBF-ConSubg and BF-ConSubg. A quick examination of those 35 instances shows that they all come from

[7] http://www.cril.univ-artois.fr/CSC09/benchs/CSC09.tar.

a single problem class (called `bddSmall`). They have high density (average 31.59%) and relatively few vertices (exactly 133 vertices), which means that most of the combinations enumerated by LBF-ConSubG and BF-ConSubG are connected. Such problems are not suited for ConSubG, which is intended for large problems with small density where LBF-ConSubG and BF-ConSubG would fail. Further, those 35 instances yield a *huge* number of *k-ConnVertices* (from 65,848,590 to 102,891,308), which is one to two orders of magnitude the number of *k-ConnVertices* of all 1654 other instances. Thus, that constraint propagation algorithms intended to be applied on such problems become totally impractical. In conclusion, this class of problems is not relevant to the techniques targeted by our approach.

- Excluding the 35 `bddSmall` instances discussed above, we notice that the set of 21 instances not completed by ConSubG is a strict subset of the set of 87 instances not completed by LBF-ConSubG, which is in turn a strict subset of the 771 instances not completed by BF-ConSubG. Thus, except for the 35 `bddSmall` instances, ConSubG terminates on more instances than LBF-ConSubG, which terminates on more instances than BF-ConSubG.

Tables 4, 5, and 6 provide condensed information on 1617 instances pertaining to 123 classes of problems (the remaining 72 instances tested were too simple to be reported). For each class, the tables provide the number of instances, the average number of vertices of the dual graphs after removing redundant edges and the average density of the resulting graphs. The tables provide also the average CPU time and the number of instances solved by ConSubG, LBF-ConSubG, and BF-ConSubG. The average CPU time is computed over the number of completed instances by the algorithm. Finally, those tables give the average number of connected subgraphs of size 5. Entries shown in boldface in the table correspond to the best values found. The dash character (-) indicates that the algorithm did not terminate on any instance in the class. ConSubG runs out of memory, and LBF-ConSubG and BF-ConSubG run out of time.

Tables 4 and 5 show instances where ConSubG clearly outperforms the other two algorithms, frequently solving instances that resisted other algorithms and always reducing the CPU by often *several orders of magnitude*.

Table 6 shows seven problem classes where the performance of ConSubG was the least spectacular. As one can clearly see, the graphs of those instances have relatively few vertices but high density. However, except for the class `bddSmall`, ConSubG solves all instances solved by the other algorithms and CPU time does not exceed half a second.

7 Conclusion

In this paper, we proposed a new algorithm, ConSubG, for computing all connected subgraphs of a graph that have a fixed size. This problem is particularly

Table 4: Results of experiments on CSP benchmarks for $k = 5$ (Part 1).

Benchmark	#Instances	#Vertices (average)	Density % (average)	ConSub Time [ms] (average)	ConSub #Instances solved	LBF Time [ms] (average)	LBF #Instances solved	BF Time [ms] (average)	BF #Instances solved	#Combinations (average)
aim-100	24	262.58	1.82	**126.67**	**24**	292,679.38	16	740,320.63	16	42,677.21
aim-200	24	532.75	0.94	**419.17**	**24**	1,765,608.57	7	-	0	134,159.83
aim-50	24	129.58	3.55	**41.67**	**24**	229,633.33	**24**	99,642.50	20	14,076.46
allIntervalSeries	14	563.43	2.58	**172.86**	**14**	16,051.43	**14**	387,881.25	8	33,688.64
BH-4-4	10	431.00	0.86	**110.00**	**10**	22,413.00	**10**	-	0	26,673.60
BH-4-7	15	1,261.00	0.30	**417.33**	**15**	89,696.67	**15**	-	0	89,585.80
bqwh-15-106	10	592.30	0.62	**129.00**	**10**	247,296.00	**10**	-	0	30,528.40
bqwh-18-141	10	876.90	0.42	**207.00**	**10**	485,233.00	**10**	-	0	47,419.70
chessbdColor	6	405.67	3.04	**1,486.67**	**6**	896,425.00	4	859,603.33	3	369,164.67
coloring	11	198.73	3.34	**39.09**	**11**	468,465.45	**11**	12,216.67	9	9,603.55
composed-25-1-2	6	224.00	1.66	**50.00**	**6**	14,581.67	**6**	2,347,030.00	**6**	12,398.83
composed-25-1-25	5	247.00	1.52	**58.00**	**5**	23,394.00	**5**	-	0	14,528.00
composed-25-1-40	5	262.00	1.44	**62.00**	**5**	31,174.00	**5**	-	0	15,795.80
composed-25-1-80	6	302.00	1.26	**66.67**	**6**	40,040.00	**6**	-	0	19,014.67
composed-25-10-20	5	620.00	0.59	**144.00**	**5**	328,602.00	**5**	-	0	34,134.20
composed-75-1-2	5	624.00	0.60	**156.00**	**5**	1,317,028.00	**5**	-	0	41,653.00
composed-75-1-25	5	647.00	0.58	**168.00**	**5**	1,383,910.00	**5**	-	0	43,806.40
composed-75-1-40	5	662.00	0.57	**170.00**	**5**	1,475,066.00	**5**	-	0	45,147.60
composed-75-1-80	5	702.00	0.54	**184.00**	**5**	1,754,266.00	**5**	-	0	48,667.80
dag-half	15	56.00	21.68	**1,595.33**	**15**	2,490.67	**15**	2,576.00	**15**	343,818.73
driver	2	2,136.00	0.82	**845.00**	**2**	1,430.00	1	1,993,930.00	1	136,263.50
dubois	13	65.38	5.47	**3.08**	**13**	14.62	**13**	103,950.77	**13**	597.85
ehi-85	5	4,108.40	0.09	**1,740.00**	**5**	-	0	-	0	310,019.40
ehi-90	5	4,368.00	0.09	**1,910.00**	**5**	-	0	-	0	329,943.00
frb30-15	5	225.40	1.65	**50.00**	**5**	52,640.00	**5**	1,696,837.50	4	13,743.00
frb35-17	5	312.00	1.15	**70.00**	**5**	179,202.00	**5**	-	0	20,030.80
frb40-19	5	371.80	0.97	**86.00**	**5**	232,160.00	**5**	-	0	24,209.20
frb45-21	5	436.60	0.83	**100.00**	**5**	348,554.00	**5**	-	0	28,827.00
frb50-23	5	480.40	0.77	**120.00**	**5**	479,054.00	**5**	-	0	32,679.80
frb53-24	5	540.40	0.67	**134.00**	**5**	605,546.00	**5**	-	0	36,977.20
frb56-25	5	558.80	0.66	**148.00**	**5**	691,146.00	**5**	-	0	38,599.20
frb59-26	5	596.60	0.62	**164.00**	**5**	868,088.00	**5**	-	0	41,594.80
geom	10	422.80	0.90	**99.00**	**10**	73,799.00	**10**	-	0	24,598.40
golombRlrArity3	11	751.00	0.90	**233.64**	**11**	39,200.00	**11**	180,920.00	1	47,821.55
golombRlrArity4	5	238.40	1.39	**136.00**	**5**	517,702.00	**5**	792,390.00	3	46,421.60
hanoi	5	46.60	13.43	**0.00**	**5**	**0.00**	**5**	24,966.00	**5**	42.60
haystacks	5	1,539.20	0.38	**454.00**	**5**	11,812.00	**5**	-	0	65,296.60
jobShop-e0ddr1	10	265.00	1.37	**54.00**	**10**	17,356.00	**10**	-	0	11,570.00
jobShop-e0ddr2	10	265.00	1.37	**51.00**	**10**	15,673.00	**10**	-	0	11,555.70
jobShop-enddr1	10	265.00	1.37	**51.00**	**10**	17,593.00	**10**	-	0	11,570.00
jobShop-enddr2	6	265.00	1.37	**50.00**	**6**	14,650.00	**6**	-	0	11,571.50
jobShop-ewddr2	10	265.00	1.37	**48.00**	**10**	15,849.00	**10**	-	0	11,555.70
js-taillard-15	10	1,785.00	0.21	**505.00**	**10**	234,190.00	**10**	-	0	88,542.80
js-taillard-20	10	4,180.00	0.09	**1,677.00**	**10**	519,587.00	**10**	-	0	220,239.10
js-taillard-20-15	10	3,130.00	0.12	**1,113.00**	**10**	357,110.00	**10**	-	0	165,069.40
knights	10	52.00	18.46	**9.00**	**10**	266.00	**10**	114.44	9	2,009.00
langford	4	380.75	3.21	**97.50**	**4**	3,905.00	**4**	110.00	1	19,725.50
langford2	14	446.71	2.94	**122.14**	**14**	4,650.71	**14**	327,998.33	6	23,874.36
langford3	11	948.82	1.09	**300.00**	**11**	11,165.45	**11**	44,445.00	2	53,292.36
langford4	10	999.00	1.10	**321.00**	**10**	11,833.00	**10**	86,395.00	2	56,382.60
lexHerald	10	487.90	8.34	8,895.00	**10**	110.00	4	115.00	4	2,502,517.60
lexPuzzle	14	289.00	7.87	**2,211.54**	**13**	414,564.00	10	106,602.22	9	657,299.38
modifiedRenault	40	151.58	1.78	**16.50**	**40**	8,893.00	**40**	323,745.25	**40**	6,036.60
nengfa	2	976.50	1.40	**145.00**	**2**	33,065.00	**2**	217,290.00	1	40,707.00
ogdPuzzle	15	263.60	14.88	**2,164.29**	**14**	365,969.09	11	88,025.00	10	598,629.64
os-gp	15	1,000.00	0.38	**270.67**	**15**	108,932.67	**15**	-	0	52,697.00
os-taillard-10	10	900.00	0.42	**236.00**	**10**	256,556.00	**10**	-	0	51,909.80
os-taillard-15	10	3,150.00	0.12	**1,151.00**	**10**	901,699.00	**10**	-	0	191,352.40

Results continue in next table.

Table 5: Results of experiments on CSP benchmarks for $k = 5$ (Part 2).

Benchmark				**ConSub**		**LBF**		**BF**		
	#Instances	**#Vertices (average)**	**Density % (average)**	**Time [ms] (average)**	**#Instances solved**	**Time [ms] (average)**	**#Instances solved**	**Time [ms] (average)**	**#Instances solved**	**#Combinations (average)**
os-taillard-4	10	48.00	7.09	**7.00**	**10**	388.00	**10**	914.00	**10**	1,563.90
os-taillard-5	10	100.00	3.54	**18.00**	**10**	4,112.00	**10**	39,267.00	**10**	4,203.00
os-taillard-7	10	294.00	1.25	**60.00**	**10**	48,247.00	**10**	-	0	15,300.80
pigeons	10	309.20	3.90	**79.00**	**10**	3,756.00	**10**	88,726.00	5	16,210.60
pret	8	70.00	5.36	**2.50**	**8**	35.00	**8**	19,972.50	**8**	497.00
primes-10	15	44.00	11.01	**28.00**	**15**	2,839.33	**15**	3,246.00	**15**	9,520.73
primes-15	16	46.25	10.39	**80.63**	**16**	2,403.13	**16**	3,223.75	**16**	25,388.38
primes-20	15	48.00	10.81	**103.33**	**15**	3,601.33	**15**	4,612.00	**15**	32,392.33
primes-25	15	48.00	11.81	**99.33**	**15**	2,928.67	**15**	3,838.67	**15**	31,125.33
primes-30	15	60.00	8.96	**121.33**	**15**	5,952.67	**15**	6,198.00	**15**	38,786.40
QCP-10	15	822.00	0.46	**213.33**	**15**	281,187.33	**15**	-	0	47,558.87
QCP-15	15	2,519.27	0.15	**853.33**	**15**	1,306,854.67	**15**	-	0	155,672.87
queenAttacking	6	723.50	2.39	**235.00**	**6**	17,148.33	**6**	65,610.00	2	40,884.00
queens	6	141.17	12.25	**31.67**	**6**	971.67	**6**	212,408.00	5	6,536.67
queensKnights	8	426.38	3.59	**127.50**	**8**	4,282.50	**8**	29,032.00	5	22,682.25
QWH-10	10	756.00	0.49	**195.00**	**10**	309,081.00	**10**	-	0	43,646.00
QWH-15	10	2,324.00	0.16	**760.00**	**10**	1,324,365.00	**10**	-	0	142,604.00
ramsey3	8	794.63	1.35	**1,188.75**	**8**	468,151.43	7	101,260.00	1	287,281.25
ramsey4	1	2,300.00	0.25	**4,000.00**	**1**	-	0	-	0	921,557.00
rand-2-23	10	253.00	1.52	**60.00**	**10**	1,877.00	**10**	-	0	12,194.00
rand-2-24	10	276.00	1.39	**66.00**	**10**	2,127.00	**10**	-	0	13,466.00
rand-2-25	10	300.00	1.28	**68.00**	**10**	2,351.00	**10**	-	0	14,801.00
rand-2-26	10	325.00	1.19	**79.00**	**10**	2,610.00	**10**	-	0	16,199.00
rand-2-27	10	351.00	1.10	**83.00**	**10**	2,872.00	**10**	-	0	17,660.00
rand-2-30-15	20	220.90	1.70	**48.50**	**20**	45,228.00	**20**	2,184,997.00	**20**	13,641.25
rand-2-30-15-fcd	20	221.55	1.69	**50.50**	**20**	48,451.00	**20**	2,223,963.50	**20**	13,700.70
rand-2-40-19	25	337.88	1.12	**83.60**	**25**	148,037.20	**25**	-	0	22,288.68
rand-2-40-19-fcd	25	338.60	1.12	**79.60**	**25**	157,371.60	**25**	-	0	22,361.60
rand-2-50-23	25	467.44	0.81	**121.20**	**25**	374,346.40	**25**	-	0	32,173.88
rand-2-50-23-fcd	25	466.72	0.81	**117.60**	**25**	370,328.80	**25**	-	0	32,116.40
rand-3-20-20	25	58.44	6.07	**13.20**	**25**	2,000.80	**25**	2,563.20	**25**	4,122.32
rand-3-20-20-fcd	25	58.72	6.02	**11.20**	**25**	2,045.60	**25**	2,619.60	**25**	4,109.84
rand-3-24-24	25	74.44	4.95	**20.00**	**25**	6,536.80	**25**	8,828.00	**25**	6,766.32
rand-3-24-24-fcd	25	74.76	4.95	**18.80**	**25**	6,652.00	**25**	8,943.60	**25**	7,066.68
rand-3-28-28	30	93.00	4.08	**30.67**	**30**	19,161.00	**30**	27,041.67	**30**	10,964.03
rand-3-28-28-fcd	29	93.00	4.08	**31.38**	**29**	19,083.10	**29**	27,315.17	**29**	10,855.48
renault	2	123.50	2.06	**15.00**	**2**	2,530.00	**2**	121,715.00	**2**	3,918.00
rlfapGraphs	10	2,638.00	0.18	**767.00**	**10**	480,292.00	**10**	-	0	125,167.30
rlfapGraphsMod	10	2,680.20	0.11	**774.00**	**10**	468,671.00	**10**	-	0	115,128.20
rlfapScens	10	3,702.60	0.12	**1,238.00**	**10**	458,117.00	**10**	-	0	175,810.00
rlfapScens11	10	4,103.00	0.09	**1,343.00**	**10**	440,730.00	**10**	-	0	188,087.00
rlfapScensMod	10	1,975.10	0.33	**607.00**	**10**	207,821.00	**10**	-	0	86,347.10
schurrLemma	9	375.22	3.64	**168.89**	**9**	622,366.67	**9**	325,518.33	6	44,669.67
ssa	7	1,505.71	0.22	**135.71**	**7**	48,061.43	**7**	694,430.00	1	28,497.00
subs	9	385.00	1.05	**92.22**	**9**	10,845.56	**9**	2,307,540.00	1	19,848.78
super-queens	5	211.80	4.93	**54.00**	**5**	9,212.00	**5**	493,927.50	4	13,722.20
tightness0.1	15	752.07	0.52	**214.00**	**15**	11,058.67	**15**	-	0	42,869.33
tightness0.2	15	414.00	0.92	**106.00**	**15**	119,200.67	**15**	-	0	27,847.67
tightness0.35	15	250.00	1.48	**54.00**	**15**	133,288.00	**15**	-	0	15,384.33
tightness0.5	15	180.00	1.99	**32.00**	**15**	67,550.00	**15**	768,524.00	**15**	9,522.47
tightness0.65	15	135.00	2.54	**20.00**	**15**	26,266.00	**15**	178,738.67	**15**	5,952.00
tightness0.8	25	103.00	3.16	**10.40**	**25**	7,806.80	**25**	45,634.40	**25**	3,498.24
tightness0.9	25	84.00	3.67	**7.20**	**25**	2,733.60	**25**	16,226.80	**25**	2,192.64
TSP-20	15	230.00	1.59	**45.33**	**15**	1,572.00	**15**	2,643,796.67	**15**	10,168.00
TSP-25	15	350.00	1.06	**78.67**	**15**	2,800.67	**15**	-	0	16,613.00
ukPuzzle	13	234.00	13.69	**1,497.50**	**12**	76,992.00	10	95,790.00	10	456,936.25
varDimacs	9	810.56	0.95	**113.33**	**9**	116,662.22	**9**	821,695.00	2	29,054.44
wordsPuzzle	14	253.21	14.20	**2,252.86**	**14**	352,576.00	10	17,054.44	9	631,028.36
Tally	*1377*				*1374*		*1316*		*639*	

Table 6: Results of experiments on CSP benchmarks for $k = 5$ (Part 3).

Benchmark	#Instances	#Vertices (average)	Density % (average)	ConSub Time [ms] (average)	ConSub #Instances solved	LBF Time [ms] (average)	LBF #Instances solved	BF Time [ms] (average)	BF #Instances solved	#Combinations (average)
bddSmall	35	133.00	31.59	-	0	**386,073.43**	**35**	435,296.00	**35**	80,665,957.60
dag-rand	15	16.00	94.17	66.67	**15**	**4.00**	**15**	4.67	**15**	4,367.13
ogdVg	45	21.62	51.85	460.22	**45**	**59.56**	**45**	64.67	45	65,490.60
rand-8-20-5	20	18.00	52.58	41.00	**20**	**7.00**	**20**	**7.00**	**20**	6,549.05
ukVg	45	21.20	51.91	425.78	**45**	**56.22**	**45**	61.11	**45**	61,197.53
lexVg	40	21.35	51.92	427.00	**40**	**56.50**	**40**	60.50	**40**	60,771.28
wordsVg	40	21.53	52.32	413.50	**40**	**54.75**	**40**	57.50	40	58,599.68
Tally	*240*				*205*		*240*		*240*	

important for enforcing high levels of consistency on Constraint Satisfaction Problems. We compared the performance of ConSubg to that a brute-force algorithm, BF-ConSubg, that generates all subgraphs then discards those that are not connected and also to a localized version of the brute-force algorithm, LBF-ConSubg, which we also proposed. We showed that ConSubg outperforms all other algorithms on structured graphs but is not suited for dense graphs when k is relatively large.

Our contributions are: (1) the posing of the problem of generating all connected subgraphs of a graph that have a fixed size, (2) the identification of an application where it is needed, and (3) the design and evaluation of a new algorithm for solving it. We are currently investigating how to reduce, or eliminate, the memory requirements while maintaining the processing time within practical limits.

Acknowledgments

Experiments were conducted on the equipment of the Holland Computing Center at UNL. This research was supported by the following grants from the National Science Foundation RI-111795, 0811250, and DMS-0914815.

References

[1] E. A. Akkoyunlu. The Enumeration of Maximal Cliques of Large Graphs. *SIAM J. Comput.*, 2:1–6, 1973.

[2] Jörg Arndt. *Matters Computational: Ideas, Algorithms, Source Code*, chapter Compositions. Academic Press, London, UK, 2010.

[3] L. Babel. Finding Maximum Cliques in Arbitrary and in Special Graphs. *Computing*, 46(4):321–341, 1991.

[4] L. Babel and G. Tinhofer. A Branch and Bound Algorithm for the Maximum Clique Problem. *Z. Oper. Res.*, 34(3):207–217, 1990.

[5] Eric T. Bax. Algorithms to Count Paths and Cycles. *Inform. Process. Lett.*, 52(5):249–252, 1994.

[6] Etienne Birmelé, Rui Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, Gustavo Sacomoto, and Marie-France Sagot. Optimal Listing of Cycles and st-Paths in Undirected Graphs. *Arxiv preprint arXiv:1205.2766*, 2012.

[7] Béla Bollobás, Christian Borgs, Jennifer Chayes, and Oliver Riordan. Directed Scale-Free Graphs. In *Proceedings of the 14^{th} Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 03)*, pages 132–139. Society for Industrial and Applied Mathematics, 2003.

[8] C. Bron and J. Kerbosch. Algorithm 457: Finding All Cliques of an Undirected Graph. *Comm. ACM*, 16(9):575–577, 1973.

[9] F. Cazals and C. Karande. A Note on the Problem of Reporting Maximal Cliques. *Theoret. Comput. Sci.*, 407(1-3):564–568, 2008.

[10] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[11] Rina Dechter and Peter van Beek. Local and Global Relational Consistency. *Theor. Comput. Sci.*, 173(1):283–308, 1997.

[12] David Eppstein, Maarten Lffler, and Darren Strash. Listing All Maximal Cliques in Sparse Graphs in Near-Optimal Time. In Otfried Cheong, Kyung-Yong Chwa, and Kunsoo Park, editors, *Algorithms and Computation*, volume 6506 of *Lecture Notes in Computer Science*, pages 403–414. Springer Berlin Heidelberg, 2010.

[13] David Eppstein and Darren Strash. Listing All Maximal Cliques in Large Sparse Real-World Graphs. In *Proceedings of the 10th International Conference on Experimental Algorithms*, SEA'11, pages 364–375, Berlin, Heidelberg, 2011. Springer-Verlag.

[14] Da Zhong Fang. Enumeration of All Spanning Trees of an Undirected Graph. *J. Tianjin Univ.*, (4):108–116, 1988.

[15] Eugene C. Freuder. Synthesizing Constraint Expressions. *Communications of the ACM*, 21 (11):958–966, 1978.

[16] Harold N. Gabow and Eugene W. Myers. Finding All Spanning Trees of Directed and Undirected Graphs. *SIAM J. Comput.*, 7(3):280–287, 1978.

[17] Robert M. Haralick and Gordon L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[18] William Hendrix, Matthew C. Schmidt, Paul Breimyer, and Nagiza F. Samatova. Theoretical underpinnings for maximal clique enumeration on perturbed graphs. *Theoret. Comput. Sci.*, 411(26-28):2520–2536, 2010.

[19] Chính T. Hoàng, Marcin Kamiński, Joe Sawada, and R. Sritharan. Finding and Listing Induced Paths and Cycles. *Discrete Appl. Math.*, 161(4-5):633–641, 2013.

[20] P. Janssen, Philippe Jégou, B. Nougier, and M.C. Vilarem. A Filtering Process for General Constraint-Satisfaction Problems: Achieving Pairwise-Consistency Using an Associated Binary Representation. In *IEEE Workshop on Tools for AI*, pages 420–427, 1989.

[21] Philippe Jégou and Marie-Catherine Vilarem. On Some Partial Line Graphs of a Hypergraph and the Associated Matroid. *Discrete Mathematics*, 111(1-3):333–344, 1993.

[22] Donald B. Johnson. Finding all the Elementary Circuits of a Directed Graph. *SIAM J. Comput.*, 4:77–84, 1975.

[23] H. C. Johnston. Cliques of a Graph–Variations on the Bron-Kerbosch Algorithm. *Internat. J. Comput. Information Sci.*, 5(3):209–238, 1976.

[24] S. Kapoor and H. Ramesh. An Algorithm for Enumerating All Spanning Trees of a Directed Graph. *Algorithmica*, 27(2):120–130, 2000.

[25] Sanjiv Kapoor and H. Ramesh. Algorithms for Generating All Spanning Trees of Undirected, Directed and Weighted Graphs. In *Algorithms and data structures (Ottawa, ON, 1991)*, volume 519 of *Lecture Notes in Comput. Sci.*, pages 461–472. Springer, Berlin, 1991.

[26] Sanjiv Kapoor and H. Ramesh. Algorithms for Enumerating All Spanning Trees of Undirected and Weighted Graphs. *SIAM J. Comput.*, 24(2):247–265, 1995.

[27] Shant Karakashian and Berthe Y. Choueiry. Tree-Based Algorithms for Computing k-Combinations and k-Compositions. Technical Report TR-UNL-CSE-2010-0009, Constraint Systems Laboratory, University of Nebraska-Lincoln, Lincoln, NE, 2010.

[28] Shant Karakashian, Robert Woodward, and Berthe Y. Choueiry. Improving the Performance of Consistency Algorithms by Localizing and Bolstering Propagation in a Tree Decomposition. In *Proceedings of the 27^{th} Conference on Artificial Intelligence (AAAI 2013)*, page 8 pages, 2013.

[29] Shant Karakashian, Robert J. Woodward, Christopher Reeson, Berthe Y. Choueiry, and Christian Bessiere. A First Practical Algorithm for High Levels of Relational Consistency. In *Proceedings of the 24^{th} Conference on Artificial Intelligence (AAAI 10)*, pages 101–107, 2010.

[30] Ina Koch. Fundamental Study: Enumerating All Connected Maximal Common Subgraphs in Two Graphs. *Theoretical Comp. Sc.*, 250(1–2):1–30, 2001.

[31] Hongbo Liu and Jiaxin Wang. A New Way to Enumerate Cycles in Graph. In *Proceedings of the Advanced Int'l Conference on Telecommunications and Int'l Conference on Internet and Web Applications and Services*, AICT-ICIW '06, pages 57–, Washington, DC, USA, 2006. IEEE Computer Society.

[32] Kazuhisa Makino and Takeaki Uno. New Algorithms for Enumerating All Maximal Cliques. In *Algorithm theory—SWAT 2004*, volume 3111 of *Lecture Notes in Comput. Sci.*, pages 260–272. Springer, Berlin, 2004.

[33] Prabhaker Mateti and Narsingh Deo. On Algorithms for Enumerating All Circuits of a Graph. *SIAM J. Comput.*, 5(1):90–99, 1976.

[34] Ladislav Novak, Žarko Karadžić, and Dragan M. Acketa. A Space Optimal Algorithm for Enumerating Spanning Trees of a Connected Graph. *Zb. Rad. Prirod.-Mat. Fak. Ser. Mat.*, 21(1):39–56, 1991.

[35] Patric R. J. Östergård. A Fast Algorithm for the Maximum Clique Problem. *Discrete Appl. Math.*, 120(1-3):197–207, 2002. Sixth Twente Workshop on Graphs and Combinatorial Optimization (Enschede, 1999).

[36] Panos M. Pardalos and Jue Xue. The Maximum Clique Problem. *J. Global Optim.*, 4(3):301–328, 1994.

[37] J. Ponstein. Self-Avoiding Paths and the Adjacency Matrix of a Graph. *SIAM J. Appl. Math.*, 14:600–609, 1966.

[38] Frank Ruskey. Combinatorial Generation. Unpublished manuscript from Citeseer, 2010.

[39] K. Sankar and A.V. Sarad. A Time and Memory Efficient Way to Enumerate Cycles in a Graph. In *International Conference on Intelligent and Advanced Systems (ICIAS 2007)*, pages 498–500, 2007.

[40] René Schott and G. Stacey Staples. Complexity of Counting Cycles Using Zeons. *Computers & Mathematics with Applications*, 62(4):1828 – 1837, 2011.

[41] Akiyoshi Shioura and Akihisa Tamura. PEfficiently Scanning All Spanning Trees of an Undirected Graph. *J. Oper. Res. Soc. Japan*, 38(3):331–344, 1995.

[42] Akiyoshi Shioura, Akihisa Tamura, and Takeaki Uno. An Optimal Algorithm for Scanning All Spanning Trees of Undirected Graphs. *SIAM J. Comput.*, 26(3):678–692, 1997.

[43] Volker Stix. Finding all maximal cliques in dynamic graphs. *Comput. Optim. Appl.*, 27(2):173–186, 2004.

[44] Robert Tarjan. Enumeration of the Elementary Circuits of a Directed Graph. *SIAM J. Comput.*, 2:211–216, 1973.

[45] James C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Comm. ACM*, 13:722–726, 1970.

[46] Etsuji Tomita, Akira Tanaka, and Haruhisa Takahashi. The Worst-Case Time Complexity for Generating All Maximal Cliques and Computational Experiments. *Theoret. Comput. Sci.*, 363(1):28–42, 2006.

[47] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM J. Comput.*, 6(3):505–517, 1977.

[48] Takeaki Uno. An Algorithm for Enumerating All Directed Spanning Trees in a Directed Graph. In *Algorithms and computation (Osaka, 1996)*, volume 1178 of *Lecture Notes in Comput. Sci.*, pages 166–173. Springer, Berlin, 1996.

[49] Peter van Beek and Rina Dechter. Constraint Tightness versus Global Consistency. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR 1994)*, pages 572–582, 1994.

[50] Peter van Beek and Rina Dechter. On the Minimality and Global Consistency of Row-Convex Constraint Networks. *Journal of the ACM (JACM)*, 42 (3):543–561, 1995.

[51] Peter van Beek and Rina Dechter. Constraint Tightness and Looseness versus Local and Global Consistency. *Journal of the ACM (JACM)*, 44 (4):549–566, 1997.

[52] Jie Sheng Wang. A General Algorithm for Enumerating Some Subgraphs of a Simple Graph. *Chinese J. Comput.*, 9(1):37–43, 1986.

[53] Herbert Weinblatt. A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph. *J. Assoc. Comput. Mach.*, 19:43–56, 1972.

[54] Marcel Wild. Generating All Cycles, Chordless Cycles, and HAmiltonian Cycles with the Principle of Exclusion. *J. Discrete Algorithms*, 6(1):93–102, 2008.

[55] Herbert S. Wilf. *Combinatorial Algorithms: An Update*. SIAM CBMS-NSF 55, 1989.

[56] S. S. Yau. Generation of All Hamiltonian Circuits, Paths, and Centers of a Graph, and Related Problems. *IEEE Trans. Circuit Theory*, CT-14:79–81, 1967.