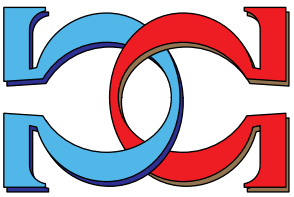
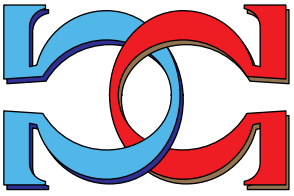
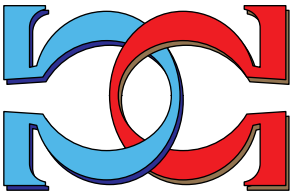


CDMTCS
Research
Report
Series



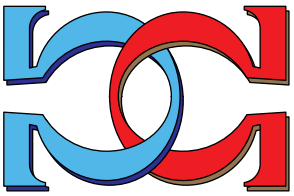
Searching for Spanning k-Caterpillars and k-Trees



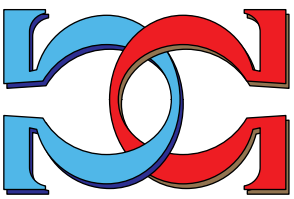
Michael J. Dinneen

and

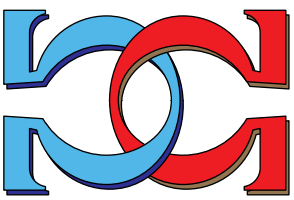
Masoud Khosravani



Department of Computer Science,
University of Auckland,
Auckland, New Zealand



CDMTCS-336
September 2008



Centre for Discrete Mathematics and
Theoretical Computer Science

Searching for Spanning k-Caterpillars and k-Trees

Michael J. Dinneen* and Masoud Khosravani†‡
Department of Computer Science
The University of Auckland
Auckland, New Zealand

Abstract

We consider the problems of finding spanning k-caterpillars and k-trees in graphs. We first show that the problem of whether a graph has a spanning k-caterpillar is $\mathcal{NP}$-complete, for all $k \geq 1$. Then we give a linear time algorithm for finding a spanning 1-caterpillar in a graph with treewidth k. Also, as a generalized versions of the depth-first search and the breadth-first search algorithms, we introduce the k-tree search (KTS) algorithm and we use it in a heuristic algorithm for finding a large k-caterpillar in a graph.

1 Introduction

One can consider k-trees as generalization of trees and they are built from a complete graph on k vertices by repeated addition of vertices, such that each new vertex is just connected to a k-clique [13, 1]. To build a k-caterpillar we need to restrict the process of attaching a new vertex to k vertices of one of those *last* k-cliques that has been formed (for more formal definitions see Section 2). Having a spanning tree is considered as a measure of reliability in a network and one may need to find a spanning k-tree or k-caterpillar as a more reliable substructure in a network; since a network with such substructure would be immune to $k-1$ nodes or $k-1$ links failures. For results concerning spanning 2-trees see Farley [7]. Recently, Tan and Zhang [14] used 1-caterpillars to solve some problems concerning the Consecutive Ones Property problem. They also showed that the Spanning 1-Caterpillar problem in graphs with maximum degree 3 is $\mathcal{NP}$-complete.

Bern [2] showed that for all $k \geq 2$ the problem of whether a graph has a spanning k-tree is $\mathcal{NP}$-complete. He also gave an approximation algorithm for finding a minimum spanning k-tree in a weighted graph, using an idea of Farley [7]. Cai and Maffraye [5] proved that the problem remains $\mathcal{NP}$-complete even when it is restricted to split graphs,

*mjd@cs.auckland.ac.nz
†masoud@cs.auckland.ac.nz
‡Research supported by the Computer Science Department Doctoral Scholarship

graphs with maximum degree $3k + 2$, and planar graphs (for $k = 2$). Later, Cai [4] presented an approximation algorithm for finding a minimum spanning 2-tree in graphs whose edge weights satisfy the triangle inequality and graphs that are complete Euclidean graphs on a set of points in the plane.

The natural question that arises is whether the problem of finding a spanning k-caterpillar is easier than finding a spanning k-tree. In Section 2 we give a negative answer to this question by showing that the problem remains $\mathcal{NP}$-complete even when it is restricted to searching for a spanning k-caterpillar in a graph, for any $k \geq 1$. To cope with the hardness of the problem we present an algorithm for finding a spanning 1-caterpillar in a graph with bounded treewidth in Section 3. Then in Section 4 we present and analyze the performance of a heuristic algorithm that finds a large k-tree in a given graph and we modify the algorithm to search for a large k-caterpillar in a graph.

2 Preliminaries

In this paper we suppose that all graphs are undirected and they have no multiple edges or loops. If $G = (V, E)$ is a graph and $U \subseteq V$ is any set of vertices, $G[U]$ denotes the induced subgraph on U. We also use the same notation to refer to the induced subgraph $G[H]$ on a subgraph H. For each $H \subseteq G$ we denote its *neighborhood* by

$$N(H) = \{v \mid \exists u \in H, (u, v) \in E \text{ and } v \notin H\}.$$

The *closed neighborhood* of H is denoted as $N[H] = N(H) \cup H$. A graph G is a *k-tree* if G is a k-clique or G is obtained recursively from a k-tree G' by attaching a new vertex to an induced k-clique of G'. The first k-clique in this process is called the *base* of a k-tree. Each vertex of degree k in a k-tree is called a *k-leaf*. A *partial k-tree* is any subgraph of a k-tree.

Partial k-trees are also referred as graphs with bounded treewidth k. The concept of treewidth became widely known by Robertson and Seymour's work on graph minors [12]. Although, before that, it was appeared by different names in literatures, see Halin [9] and Rose [13]. Here we use the *constructive* definition of partial k-trees. Since, as we later show, it is closer to our k-parse data structure for representing graphs of bounded treewidth. For an up-to-date survey on treewidth refer to Bodlaender [3].

A *simplicial vertex* of a graph is a vertex whose neighborhood induces a clique. An ordering of the vertices $\sigma = [v_1, \ldots, v_n]$ is called a *perfect elimination scheme* if for every $1 \leq i \leq n$, v_i is a simplicial vertex in $G[v_i, \ldots, v_n]$. If v_i is a vertex in a perfect elimination scheme then we refer to each clique in $N(v_i) \cap \{v_i, \ldots, v_n\}$ as a parent of v_i. We say two k-cliques are *smooth* neighbors if the induced subgraph of their union has a perfect elimination scheme. Note that each k-tree has a perfect elimination scheme.

A *k-boundaried graph* is a pair (G, ∂) of a graph $G = (V, E)$ and an injective function ∂ from $\{0, \ldots, k\}$ to V. The image of ∂ is the set of *boundaried vertices* and is denoted by $Im(\partial)$. When it is clear from the context, we abuse the notation and refer to $Im(\partial)$ as ∂. A graph G is a *k-caterpillar* if it is (1) a k-clique; or (2) a k-boundaried $(k+1)$-clique; or (3) a k-boundaried graph (G', ∂') that results from attaching a new vertex v to k vertices of $Im(\partial)$ of a k-caterpillar (G, ∂), such that $Im(\partial') = N[v]$. It is easy to see that each

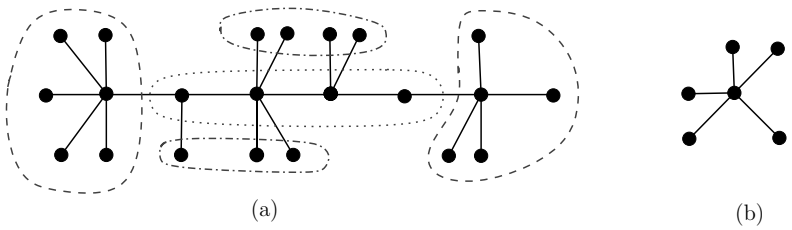


Figure 1: (a) A 1-caterpillar with heads (dashed), leaves (dash-dotted), and spine (dotted), (b) a 1-star.

k-caterpillar is also a k-tree. Some authors use the term k-path instead of k-caterpillar but since k-path also refers to a different concept in the Mathematics and Computer Science literature, we use the term k-caterpillar to avoid any confusion, for example see [8, 10].

If one deletes all k-leaves of a k-caterpillar the remaining (nonempty) subgraph is called a *k-spine*. A k-star is a k-caterpillar that has a k-clique as its k-spine. If a k-caterpillar G is not a k-star, then its k-spine can be considered as an alternating sequence of distinct k-cliques and $(k+1)$-cliques $(e_0, t_1, \ldots, t_n, e_n)$. Where e_is are k-cliques and t_is are $(k+1)$-cliques, and the sequence starts and ends with k-cliques, e_0 and e_n, see Proskurowski [11]. Each k-caterpillar has two *heads*. Each head is a subgraph induced by the union of e_j, for $j = 0, n$, and the k-leaves attached to it. See Figure 1.

A *spanning k-tree* (*k-caterpillar*) of a graph is a k-tree (k-caterpillar) that composed of all the vertices of the graph.

Here we introduce the k-parse data structure for representing partial k-trees and k-caterpillars. We first show how a k-caterpillar can be represented by a string of single or paired boundary values. For more detailed description refer to Dinneen [6].

The partial k-caterpillars can be generated by strings of (unary) operators from the following *operator set* $\Sigma_k = V_k \cup E_k$:

$$V_k = \{ \textcircled{0}, \ldots, \textcircled{k} \} \quad \text{and} \quad E_k = \{ \boxed{i\ j} \mid 0 \le i < j \le k \}.$$

To generate partial k-trees, an additional (binary) operator $\oplus$, called *circle plus*, is added to Σ_k. The semantics of these operators on boundaried graphs G and H of boundary size at most $k+1$ are as follows:

$G\ \textcircled{i}$	Add an isolated vertex to the graph G, and label it as the new boundary vertex i.
$G\ \boxed{i\ j}$	Add an edge between boundaried vertices i and j of G (ignore if operation causes a multi-edge).
$G \oplus H$	Take the disjoint union of G and H except that equal-labeled boundary vertices of G and H are identified.

It is syntactically incorrect to use the operator $\boxed{i\ j}$ without being preceded by both $\textcircled{i}$ and $\textcircled{j}$, and the operator $\oplus$ must be applied to graphs with the same boundary ∂. A

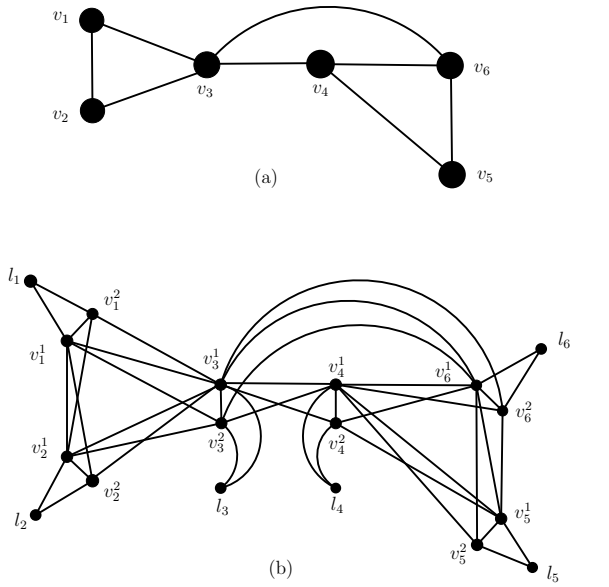


Figure 2: (a) A graph G and (b) its spanning 2-caterpillar instance.

graph described by a string (tree, if $\oplus$ is used) of these operators is called a *k-parse*, and has an implicit labeled boundary ∂ of at most $k+1$ vertices. By convention, a k-parse begins with the operator string $[\,⓪, ①, \ldots, ⓚ\,]$ which represents the edgeless graph of order $k+1$. Throughout this paper, we refer to a k-parse and the graph it represents interchangeably. Let $G = (g_0, g_1, \ldots, g_n)$ be a k-parse and $Z = (z_0, z_1, \ldots, z_m)$ be any sequence of operators over Σ_k. The *concatenation* $(\cdot)$ of G and Z is defined as

$$G \cdot Z = (g_0, g_1, \ldots, g_n, z_0, z_1, \ldots, z_m).$$

(For the treewidth case, G and Z are viewed as two connected subtree factors of a parse tree $G \cdot Z$ instead of two parts of a sequence of operators.)

As we noted earlier, Bern [2] showed that, for a fix k, finding a spanning k-tree in a graph is $\mathcal{NP}$-complete. In the following theorem we prove that when the problem is restricted to finding a spanning k-caterpillar, it still remains intractable. Let us state the problem formally.

Problem: Spanning k-Caterpillar
Instance: A graph $G = (V, E)$,
Question: Does G contain a spanning k-caterpillar?

Theorem 1 *For each $k \geq 1$, the Spanning k-Caterpillar problem is $\mathcal{NP}$-complete.*

Proof: We prove the theorem by transforming the Hamiltonian path problem to the spanning k-caterpillar problem. Let $G = (V, E)$ be an instance of the Hamiltonian path problem. We construct a graph $\tilde{G} = (\tilde{V}, \tilde{E})$ such that $|\tilde{V}| = (k+1)|V|$ and

$|\tilde{E}| = (k/2)(k+1)(|E|+|V|)$. To each $v \in V$ we assign a k-clique $\tilde{K}_v$ that we label its vertices by $\{\tilde{v}^1, \ldots, \tilde{v}^k\}$. For each two adjacent vertices v and w in G we assign $k(k+1)/2$ edges in $\tilde{G}$ by connecting each $\tilde{v}$ to $\{\tilde{w}^1, \ldots, \tilde{w}^{(k-i+1)}\}$, $1 \leq i \leq k$. It is easy to see that by this process each $\tilde{w}^j$ is also adjacent to $\{\tilde{v}^1, \ldots, \tilde{v}^{(k-j+1)}\}$, $1 \leq j \leq k$. Then for each k-clique $\tilde{K}_v$ in $\tilde{G}$ we add a new vertex $\tilde{l}_v$ and we connect it to all vertices in $\tilde{K}_v$. In Figure 2 the process is shown for a graph G when $k = 2$.

Now let $G = (V, E)$, $|V| = n$, be a graph with a Hamiltonian path, $P = v_1, v_2, \ldots, v_n$. We show $\tilde{G}$ has a spanning k-caterpillar, too. To construct an spanning k-caterpillar for $\tilde{G}$, we first choose $\tilde{K}_{v_1}$ as the base and connect $\tilde{l}_{v_1}$ to all its vertices. Then for each fixed i, $i = 2, \ldots, n$, we choose $\tilde{v}_i^j$, $j = 1, \ldots, k$ from $\tilde{K}_{v_i}$ and we attach it to the vertices $\{\tilde{v}_{i-1}^1, \ldots, \tilde{v}_{i-1}^{(k-j+1)}\}$ in $\tilde{K}_{v_{i-1}}$; note that by this process the set $\{\tilde{v}_i^1, \ldots, \tilde{v}_i^j, \tilde{v}_{i-1}^1, \ldots, \tilde{v}_{i-1}^{(k-j+1)}\}$ forms a $(k+1)$-clique and they are the vertices of the boundary value set. Then we attach $\tilde{l}_{v_i}$ to $\tilde{K}_{v_i}$ and continue the process for $i+1$, $i < n$.

Now consider a graph G whose corresponding graph $\tilde{G}$ has a spanning k-caterpillar, $\tilde{H}$. The construction of $\tilde{H}$ from a base K imposes an ordering on the vertices of $\tilde{G}$, especially on the k-leaves. Let $\{l_{v_{i_1}}, \ldots, l_{v_{i_n}}\}$ be the order of the k-leaves in the construction of $\tilde{H}$. We show that $P = v_{i_1}, \ldots, v_{i_n}$ is a Hamiltonian path for G. Let l_{v_i} and $l_{v_{i+1}}$, $i = 1, \ldots, n-1$ be two consecutive k-leaves and also let $\tilde{K}_{v_i} = N(l_{v_i})$ and $\tilde{K}_{v_{i+1}} = N(l_{v_{i+1}})$. Because of our method for constructing $\tilde{G}$ from G, we just need to prove that there is at least an edge between $\tilde{K}_{v_i}$ and $\tilde{K}_{v_{i+1}}$. As l_{v_i} is just attached to $\tilde{K}_{v_i}$, in ordering of $\tilde{H}$ it appears exactly after the last vertex of it, also it is the same for $l_{v_{i+1}}$ and $\tilde{K}_{v_{i+1}}$. So the vertices of $\tilde{K}_{v_{i+1}}$ should be attached to $\tilde{K}_{v_i}$. □

3 Finding a spanning 1-caterpillar in a partial k-tree

In this section we show that the problem of finding a spanning 1-caterpillar in a partial k-tree with n vertices has an algorithm in $O(5^{k+1}B_{k+1}^2 n)$, where B_{k+1} is the $(k+1)$th Bell number. Throughout this section we suppose that G is represented as a k-parse $G = (g_0, \ldots, g_m)$.

We use a forest of at most $k+1$ different 1-caterpillars as a partial solution, each has at least one vertex in the boundary set $\partial = \{0, \ldots, k\}$. The main point is *to code* the information of each partial solution in a state vector $S = (A, B)$. We define the set A as

$$A = \{(b_0, L_{b_0}), \ldots, (b_r, L_{b_r})\},$$

where in (b_i, L_{b_i}), $0 \leq i \leq r \leq k$, each b_i represents a distict boundary value from ∂. Each L_{b_i} is a label from the set $\{H, S, C, I\}$, where H, S, C, and I are characteristics of boundary vertices in a partial solution. They stand respectively for *head, spine, center* (of a k-star), and *isolated vertex*. The set B is a partition set of ∂. If any two boundary vertices belong to the same element of B, then they belong to the same connected component of a partial solution that is represented by B.

Note that we do not consider any label for leaves. Since each leaf that appears as a neighbor of a spine has no role in extending a partial solution. Those leaves that belong to a head appear with an H label in a state vector. A leaf that is a boundary vertex and is not part of a head just appears as a member in an element of a partition set B.

In accordance with our dynamic programming approach, we use a table T with rows indexed by state vectors and columns indexed by k-parse operators from g_k to g_m. We initialize all entries of T to the value *false*. Due to our convention for the first $k+1$ operators we have $g_i = \textcircled{i}$, $0 \le i \le k$. So at the first step we assign the value *true* to $T((A,B), g_k)$, where $A = \{(0,I), \ldots, (k,I)\}$ and $B = \{\{0\}, \{1\}, \ldots, \{k\}\}$. In the next steps we compute entries of g_p column by using the following rules and the values for g_{p-1}, $k < p \le m$.

Vertex operator $\textcircled{i}$: In a vertex operation a boundary vertex i is replaced by an isolated vertex. Let $T((A,B), g_{p-1})$ be a true entry of T and let S_i be the element of B that contains i.

If $S_i - \{i\}$ is not empty, we set the entry $T((A',B'), g_p)$ to *true* where

$$A' = (A - \{(i, L_i)\}) \cup \{(i, I)\}$$

and

$$B' = (B - \{S_i\}) \cup \{\{S_i - i\}, \{i\}\}.$$

Otherwise, $T((A,B), g_{p-1})$ produces no entry *true* for the new column g_p.

Edge operator $\boxed{i\ j}$: When g_p is an edge operator, we initially set $T((A,B), g_p) = T((A,B), g_{p-1})$. Let S_i and S_j be the elements of B that contain i and j, respectively. Then if $S_i \neq S_j$ and $T((A,B), g_{p-1}) = \mathit{true}$, we set $T((A',B'), g_p)$ to *true* using the following rules:

1. if $L_i = H$ and $L_j = H$ if there is no $k \in S_i, S_j$ such that $(k, C) \in A$ then $A' = (A - \{(i,H),(j,H)\}) \cup \{(i,S),(j,S)\}$, otherwise $A' = (A - \{(i,H),(j,H),(k,C)\}) \cup \{(i,S),(j,S),(k,H)\}$,
2. if $L_i = H$ and $L_j = C$ then $A' = (A - \{(i,H),(j,C)\}) \cup \{(i,S),(j,H)\}$,
3. if $L_i = H$ and $L_j = I$ then $A' = (A - \{(j,I)\}) \cup \{(j,H)\}$, also add $A' = (A - \{(i,H),(j,I)\}) \cup \{(i,S),(j,H)\}$,
4. if $L_i = C$ and $L_j = C$ then $A' = (A - \{(i,C),(j,C)\}) \cup \{(i,H),(j,H)\}$,
5. if $L_i = C$ and $L_j = I$ then $A' = (A - \{(j,I)\}) \cup \{(j,H)\}$,
6. if $L_i = S$ and $L_j = I$ then $A' = A$,
7. if $L_i = I$ and $L_j = I$ then $A' = (A - \{(i,I),(j,I)\}) \cup \{(i,C),(j,H)\}$, also $A' = (A - \{(i,I),(j,I)\}) \cup \{(i,H),(j,C)\}$.

In all cases we set

$$B' = (B - \{S_i, S_j\}) \cup \{S_i \cup S_j\}.$$

When $L_i \in \{H, C, S\}$ and $L_j = S$ then the resulted graph of joining i and j is not a 1-caterpillar and is discarded.

Boundary Join Operator $G \oplus G'$: We suppose the graphs G and G' are represented by $G = (g_0, \ldots, g_p)$ and $G' = (g'_0, \ldots, g'_q)$, respectively. We use the last columns of the tables T and T' to fill the first column of the table T'' for $G'' = G \oplus G'$. Let

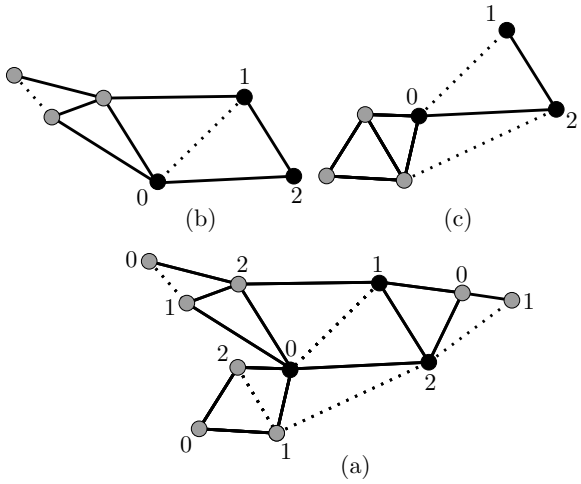


Figure 3: (a) A graph G, (b) a subgraph H and (c) another subgraph H'.

$T((A,B),g_p)$ and $T'((A',B'),g'_q)$ be a pair of true entries in T and T'. If there are two different boundary values i and j such that $S_i = S_j$ and $S'_i = S'_j$ then unifying the boundary vertices produces a cycle and the resulted graph is not acceptable as a partial solution. So we consider only those pairs of entries in T and T' such that if $S_i = S_j$ then $S'_i \neq S'_j$ and vice versa, for $i \neq j$ and $i, j \in \partial$.

The value of a row (A'', B'') of the first column of T'' is set to *true* by using the values of (A,B) and (A',B'). If $(i, L_i) \in A$ and $(i, L'_i) \in A'$ then we add (i, L''_i) to A'' due to the following rules:

1. If $L_i = S$ and $L'_i \in \{H, I, C\}$ then $L''_i = S$.
2. If $L_i = H$ and $L'_i \in \{I, C\}$ then $L''_i = H$, but if $L'_i = H$ then $L''_i = S$.
3. If $L_i = C$ and $L'_i \in \{C, I\}$ then $L''_i = C$.
4. If $L_i = I$ and $L'_i = I$ then $L''_i = I$.
5. If $L_i = I$ and there is no $(i, L'_i) \in A'$ then discard (i, L''_i).
6. If L_i and L'_i do not belong to this list, $A'' = B'' = \emptyset$.

Also we have

$$B'' = \{S_i \cup S'_i \mid i \in \partial, S_i \in B, S'_i \in B'\}.$$

The final answer is yes if there is at least one entry *true* in the last column such that its associated row is indexed by a vector (A,B), where $B = \partial$.

3.1 Illustrating the algorithm

Before proving the correctness of the algorithm, let us clarify the process by an example. In Figure 3 we depict a graph G that is the result of the following 2-parse

$$G = (H \oplus H') \cdot (\, ⓪, \boxed{0\ 1}, \boxed{0\ 2}, \ ①, \boxed{0\ 1}).$$

The subgraphs $H = (h_0, \ldots, h_{12})$ and $H' = (h'_0, \ldots, h'_{11})$ are partial 2-caterpillars and they are represented as 2-parses

$$H = (\ ⓪,\ ①,\ ②, \boxed{0\ 2}, \boxed{1\ 2},\ ⓪, \boxed{0\ 1}, \boxed{0\ 2},\ ①, \boxed{1\ 2},\ ②, \boxed{0\ 2}, \boxed{1\ 2})$$

and

$$H' = (\ ⓪,\ ①,\ ②, \boxed{0\ 1}, \boxed{0\ 2},\ ⓪, \boxed{0\ 1}, \boxed{0\ 2},\ ②, \boxed{0\ 2},\ ①, \boxed{1\ 2}).$$

When the algorithm is applied to H the true entries of the first two columns and the last two columns of the table are given next.

column h_2: $(\{(0, I), (1, I), (2, I)\}, \{\{0\}, \{1\}, \{2\}\})$,

column h_3: $(\{(0, H), (1, I), (2, C)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, C), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$, also true entries of the column h_2.

...

column h_{10}: $(\{(0, H), (1, I), (2, I)\}, \{\{1\}, \{0\}, \{2\}\})$,
$(\{(0, I), (1, H), (2, I)\}, \{\{0\}, \{1\}, \{2\}\})$,
$(\{(0, H), (1, H), (2, I)\}, \{\{0, 1\}, \{2\}\})$,
$(\{(0, H), (2, I)\}, \{\{0, 1\}\{2\}\}\})$.

column h_{11}: $(\{(0, H), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, S), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, H), (1, H), (2, C)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, C), (1, H), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, H), (1, H), (2, H)\}, \{\{0, 1, 2\}\})$,
$(\{(0, S), (1, H), (2, H), \{\{0, 1, 2\}\})$,
$(\{(0, H), (2, H)\}, \{\{0, 1, 2\}\})$,
$(\{(0, S), (2, H)\}, \{\{0, 1, 2\}\})$, also true entries of the column h_{10}.

Finally the last two column for H' are given next.

column h'_9: $(\{(0, C), (1, I), (2, I)\}, \{\{0\}, \{2\}, \{1\}\})$,
$(\{(0, H), (1, I), (2, I)\}, \{\{0\}, \{2\}, \{1\}\})$,
$(\{(0, H), (1, I), (2, I)\}, \{\{0\}, \{2\}, \{1\}\})$,
$(\{(0, H), (1, I), (2, I)\}, \{\{0\}, \{1\}, \{2\}\})$,
$(\{(1, I), (2, S)\}, \{\{0\}, \{1\}, \{2\}\})$,
$(\{(0, S), (1, I), (2, I)\}, \{\{0\}, \{1\}, \{2\}\})$.

colum h'_{11}: $(\{(0, H), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, S), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
$(\{(0, S), (1, I)\}, \{\{0, 2\}, \{1\}\})$, also true entries of the column h'_9.

Now we use the rule concerning the boundary join operation to the entry

$$(\{(0, H), (1, I), (2, I)\}, \{\{1\}, \{0\}, \{2\}\}),$$

from column h_9 and to the entry

$$(\{(0,H),(1,I),(2,H)\},\{\{0,2\},\{1\}\}),$$

from column h'_{11}. The resulting entry is

$$(\{(0,S),(1,I),(2,H)\},\{\{0,2\},\{1\}\}).$$

We next apply the 2-parse operators from the extension to this entry (from the first column of the table of $H \oplus H'$) and we have:

1. ⓪ : $(\{(0,I),(1,I),(2,H)\},\{\{0\},\{1\},\{2\}\})$,
2. $\boxed{0\ 1}$: $(\{(0,H),(1,C),(2,H)\},\{\{0,1\},\{2\}\}),(\{(0,C),(1,H),(2,H)\},\{\{0,1\},\{2\}\})$,
3. $\boxed{0\ 2}$: $(\{(0,S),(1,C),(2,S)\},\{\{0,1,2\}\}),(\{(0,H),(1,H),(2,S)\},\{\{0,1,2\}\})$,
4. ① : $(\{(0,S),(1,I),(2,S)\},\{\{0,2\},\{1\}\}),(\{(0,H),(1,I),(2,S)\},\{\{0,2\},\{1\}\})$,
5. $\boxed{0\ 1}$: $(\{(0,S),(2,S)\},\{\{0,1,2\}\}),(\{(0,H),(1,H),(2,S)\},\{\{0,1,2\}\})$, $(\{(0,S),(1,H),(2,S)\},\{\{0,1,2\}\})$.

As it is seen from the result of the last operation, the graph G has a spanning 1-caterpillar.

3.2 Correctness of the algorithm

In this remaining part of the section we provide the next three lemmas that alongside with Theorem 5 will show the correctness of our spanning 1-caterpillar algorithm.

Lemma 2 *Let $G = (g_0, \ldots, g_m)$ be a partial k-tree and also let T be the table produced by the algorithm. If $T((A,B), g_m)$ is a true entry in the last column of T such that $B = \partial$, then the graph G has a spanning 1-caterpillar.*

Proof: We show that each true entry in the column p, $p \leq m$ of T relates to a partial solution that has the following properties:

1. the partial solution is a forest of 1-caterpillars,
2. the partial solution covers all vertices in $(g_0, \ldots, g_p)$.

The conditions are satisfied for the only true entry in the first column of T. Since in the first step we set the element $T((A,B), g_k)$ to *true*, where $A = \{(0,I), \ldots, (k,I)\}$ and $B = \{\{0\},\{1\},\ldots,\{k\}\}$.

Now suppose that the conditions hold for each true entry in a column $p-1$, $k \leq p-1 < m$. We show that they also hold for each true entry in the column p. Due to our rule for a vertex operation the lemma is true when $T((A',B'), g_p)$ is the resulted entry *true* from $T((A,B), g_p)$ by a vertex operation. When g_p is an edge operation Property 2 is trivially hold, since no new vertex is added. Also each rule for an edge operation maintains the first property. The same argument is applicable when g_p is a boundary join operation.

Since Properties 1 and 2 hold for g_m and also since $B = \partial$, we conclude that the partial solution corresponds to $T((A,B), g_m)$ is a spanning 1-caterpillar for G. □

Lemma 3 *If $G = (g_0, \ldots, g_m)$ is a partial k-caterpillar that has a spanning 1-caterpillar then the last column of the table T that results from the algorithm has a true entry $T((A, B), g_m)$ with $B = \partial$.*

Proof: Without loss of generality we suppose that T is a spanning 1-caterpillar of G such that each leaf of it is attached to a spine vertex with the smallest index in the k-parse representation. We prove a stronger claim by showing that such spanning 1-caterpillar appears as a partial solution represented by an entry *true* in the last column.

We prove the statement by an inductive argument on the number of vertices of the spine of T. If T has only one vertex on its spine then it is a 1-star and the vertex that appears on the center of T takes a unique boundary value in the k-parse, otherwise it fails to attach to all vertices of G. So the center of the star always appears on each boundary and we attach it to a vertex u when u appears on a boundary by a vertex operation.

Now assume the lemma is valid for any k-parse that has a spanning 1-caterpillar with less than p vertices on its spine, $p \geq 2$. Let T be a spanning 1-caterpillar of G that has p vertices on its spine. Suppose v is the spine vertex that is created by g_f, the vertex operation that assumes the largest index among all vertex operators corresponding to spine vertices of T. We delete v and all its leaves in T and if v is not a head we connect its two neighbors on the spine by adding an edge. The resulted graph H has a 1-caterpillar D with $p - 1$ vertices on its spine. We can consider the k-parse representation of H as $(g_0, \ldots, g_{f-1})$, when v is a head, or $(g_0, \ldots, g_{f-1}).g_e$, where g_e is the edge operation corresponds to attaching the neighbors of v on the spine. Because of our inductive assumption, the last column of the table of the algorithm, when applied to H, has a true entry that its partial solution is D. Note that the table of H is the same as the table produced by the algorithm when applied to G in the column $f-1$. If v is not a head in T we just discard the edge operation g_e to allow the neighbors of v on the spine to appear as heads in the partial solution. Now as v stays as a boundary vertex during $g_f, \ldots, g_m$, the leaves of T can be attached to v by their appropriate edge operation. □

Lemma 4 *Let $G = H \oplus H'$, where H and H' are partial k-caterpillars. If G has a spanning 1-caterpillar then the column of the table T, that results from applying the algorithm to G, has a true entry $T((A, B), H \oplus H')$ with $B = \partial$.*

Proof: If C is a spanning 1-caterpillar in $G = H \oplus H'$, then $H \cap C$ and $H' \cap C$ are forests of 1-caterpillars that span H and H', respectively. To connenct the (spanning) forsests of 1-caterpillars in H, we first direct the edges on the spine of C from one head to the other. Then we walk along the path on the spine. Once we leave H (by entering to a non boundary vertex of H') and return to it (by entering to a non boundsry vertex of H), we add an edge between the two consecutive visited boundary vertices. By the same method we connect components of $H' \cap C$. Note that since we connect the connected components via their heads, the resulted graphs are spanning 1-caterpillar of H and H'. We consider the new edges as extensions of the k-parses of H and H'.

Now we apply the algorithm to the extended k-parses of H and H'. By Lemma 3 we know that the last column of each table has a true entry. Since the extensions are done by adding edges, there are also true entries on the last columns of the tables associated

to k-parse representations of H and H'. In particalar, there are true entries that their partial solutions are associated to $H \cap C$ and $H' \cap C$, so joining them by an $\oplus$ operator produces a true entry that has C as its partial solution. □

Theorem 5 *The algorithm solves the spanning 1-caterpillar problem in $O(5^{k+1}B_{k+1}n)$ for a partial k-caterpillar and in $O(5^{k+1}B_{k+1}^2n)$ for a partial k-tree with n vertices; where B_{k+1} is the $(k+1)$th Bell number.*

Proof: Note that each k-parse G has a representation as (a) $G = G' \cdot H$, or (b) $G = G' \oplus G''$, where G' and G'' are partial k-trees and H is a sequence of vertex and edge operators. The correctness of the algorithm follows from an inductive argument on the number of operators as in (a) and (b). The validity of the base case is the result of Lemmas 3 and 4. For the induction step one just need to use the same technique as Lemma 4 to reduce a problem to the cases with less number of operations.

To solve the problem for a partial k-caterpillar, the algorithm uses a table that has $O(5^{k+1}B_{k+1}n)$ entries. In the case when the graph is a partial k-tree, the algorithm processes each boundary join operation by comparing all pairs of entries in the last two columns of the joined graphs. So it takes $O(5^{k+1}B_{k+1}^2n)$ steps. □

4 Depth-first and breadth-first search for k-trees

There is an efficient algorithm for recognizing a k-tree; that is choosing consecutively k-leaves to remove vertices from a graph in any order the same as a perfect elimination scheme. If the process fails to find a k-leaf at some step then the graph is not a k-tree. As simply seen, this process is not applicable for finding a *hidden* k-tree in a graph.

The depth-first search and the breadth-first search algorithms are used as subroutines in many graph algorithms. Since k-trees are considered as generalization of trees, one may ask how we can generalize these algorithms for finding a hidden k-tree in a graph? We try to answer this question by introducing a heuristic algorithm that, if it is implemented in a proper way, resembles the foregoing algorithms. We refer to this algorithm as the k-tree search algorithm (KTS).

Note that in a graph that has a spanning k-tree, each vertex is attached to at least one k-clique. We use this trivial fact for designing a heuristic algorithm to extract k-trees of a graph G. To save information during the process of our algorithm we use a list L to save the order of vertices in a reverse order of their appearance in a perfect elimination scheme. We also use a set S (with sompe priorty structure) to save k-cliques that appear during the search process.

We first choose an arbitrary vertex v of the graph. Then we find a k-clique K in $G[N(v)]$ and save K in S. We also add the vertices in K as the first k vertices to L and assign the empty set as the parent of all vertices.

We repeat the following steps until S becomes empty. We remove a k-clique K from S and add each vertex $u \in K$ to L if it is not added yet. For each $w \in \bigcap_{x \in K} N(x)$ if w is not in L, we update its parent by considering K as the new parent, otherwise we do not change the parent of w. We then save all k-cliques comprising w and each set of $k-1$ vertices of K in S.

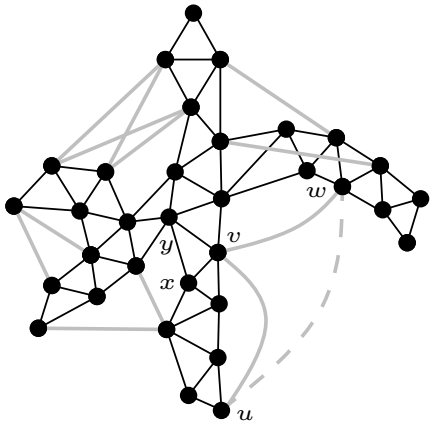


Figure 4: A graph with a spanning 2-tree.

Note that if one implements S (as a dynamic set) by a queue the resulting k-trees extend in breadth fashion, and if S is implemented by a stack the resulting k-trees extend in depth fashion. Sometimes, due to improper choices of vertices or k-cliques, an output of the algorithm is a forest of k-trees and it does not cover all vertices of a graph. For example in Figure 4 if we first choose the vertex v and then the edge (u, w) as a 2-clique, the first resulting 2-tree is $G[\{v, u, w\}]$. This cause the output of the algorithm consists of at least four connected components.

As a solution to such problems we propose to use the concept of neighborhood-density to guide the search in a proper way. In a graph G the *neighborhood-density* of two subgraphs H and H' of G is defined as

$$d(H, H') = |N(H) \cap N(H')|.$$

This concept can be used in the algorithm by this way: In the first step choose the k-clique K such that $d(v, K)$ gets the maximum value among all k-cliques that are neighbors of v. Also in the iterative steps of the algorithm, when a k-clique is chosen from the set S, allow the next k-clique to be chosen if it is built from a vertex $w \in \bigcap_{x \in K} N(x)$ and a set of $k-1$ vertices of K such that

$$d(w, K) = max\{d(u, K) \mid u \in \bigcap_{x \in K} N(x)\}.$$

Now if one uses the guided version of the algorithm for the graph in Figure 4, the first 2-clique is (x, y) rather than (u, v). Choosing the next 2-cliques by the same way, produces a larger k-tree in comparison with the unguided version of the KTS algorithm.

We use the KTS algorithm as a subroutine to find a large k-caterpillar. We refer to this as the k-caterpillar search algorithm (KCS). To this end we use a function ω that is defined, on a path P, as

$$\omega(P) = \left| \bigcup_{v \in V(P)} N[v] \right|.$$

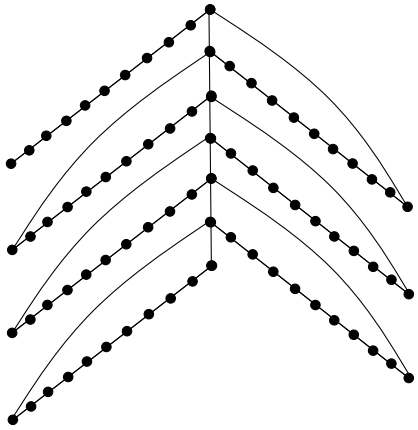


Figure 5: A somewhat difficult graph to find a spanning 1-caterpillar.

The function ω computes the number of vertices in a k-path comprising the number of k-leaves that are attached to it. The KCS algorithm first uses the k-tree search algorithm to find a forest of k-trees in a graph. Then for each pair of k-leaves, it finds a k-caterpillar that has the largest number of vertices. This is done by computing the weight function ω for each k-caterpillar between a pair of k-leaves.

It is worth to mention that the result of this algorithm depends on the result of the k-tree search algorithm. For example see Figure 5. The graph is shown as though the final result of applying the depth-first search algorithm; with back-edges that are not part of the tree. As one can easily check by choosing the path with the largest *weight*, a small portion of the total number of vertices is covered.

5 Conclusion and further work

We have primarily focused on the graph problem of deciding whether a graph contains a spanning 1-caterpillar or not. This problem, although closely related to the classic spanning tree problem (i.e., "Is the graph connected?"), turns out to be just as hard as deciding if a graph contains an Hamiltonian path. To cope with this apparent intractability we provide a new linear-time algorithm to decide this problem when input graphs are of bounded treewidth. Our explicit dynamic programming rules for this algorithm are quite simple but are also frugal in the amount of state information that needs to be maintained.

In this paper we also propose two simple algorithmic heuristics that extend the standard graph searching techniques of breadth-first and depth-first search to find spanning 'forests' of k-trees and/or k-caterpillars. With the goal of finding large spanning k-caterpillars, our experimental work with these simple heuristics seems promising. However, we would like to know if there exists a polynomial-time approximation algorithm for finding the largest 1-caterpillar in a graph. There are other related optimization problems that are interesting like minimizing or maximizing the spine length over all possible spanning k-caterpillars (assuming at least one exists).

A natural extension to the work given in this paper is to study graphs with edge weights. We know that finding minimum-weighted spanning trees is easy by very simple greedy algorithms. Since we have shown that even deciding if a spanning k-caterpillar exists is $\mathcal{NP}$-hard it is easy to see that it is also hard to decide if an edge-weighted graph contains a spanning k-caterpillar of total weight at most W. [Reduce any unweighted graph G of order at least k to a copy of graph G with edge weights 1 and use $W = \binom{k}{2} + (|G| - k)k$.] Similarly, finding good algorithms for finding minimum edge-weighted spanning k-trees would be of interest.

References

[1] S. Arnborg and A. Proskurowski. Characterization and recognition of partial k-trees. In *Proceedings of the sixteenth Southeastern international conference on combinatorics, graph theory and computing (Boca Raton), Florida, 1985*, volume 47, pages 69–75, 1985.

[2] M. W. Bern. *Network design problems: Steiner trees and spanning k-trees.* PhD thesis, University of California, Berkely, 1987.

[3] H. Bodlaender. Treewidth: structure and algorithms. In *Structural information and communication complexity*, volume 4474 of *Lecture Notes in Comput. Sci.*, pages 11–25, Berlin, 2007. Springer.

[4] L. Cai. On spanning 2-tree in a graph. *Discrete Appl. Math.*, 74:203–216, 1997.

[5] L. Cai and F. Muffray. On the spanning k-tree problem. *Discrete Appl. Math.*, 44:139–156, 1993.

[6] M. J. Dinneen. Practical enumeration methods for graphs of bounded pathwidth and treewidth. Technical Report CDMTCS-055, Centre for Discrete Mathematics and Theoretical Computer Science, University of Auckland, New Zealand, September 1997.

[7] A. M. Farley. Network immune to isolated failures. *Networks*, 11:255–268, 1981.

[8] A. Gupta and N. Nishimura. Characterizing the complexity of subgraph isomorphism for graphs of bounded path-width. In *STACS 96 (Grenoble, 1996)*, volume 1046 of *Lecture Notes in Comput. Sci.*, pages 453–464, Berlin, 1996. Springer.

[9] R. Halin. S-functions for graphs. *J. Geometry*, 8(1-2):171–186, 1976.

[10] K. Kawarabayashi and B. Mohar. Some recent progress and applications in graph minor theory. *Graphs Combin.*, 23(1):1–46, 2007.

[11] A. Proskurowski. Separating subgraphs in k-trees: cables and caterpillars. *Discrete Math.*, 49(3):275–285, 1984.

[12] N. Robertson and P.D. Seymour. Graph minors II, algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.

[13] D. J. Rose. On simple characterizations of k-trees. *Discrete Math.*, 7:317–322, 1974.

[14] J. Tan and L. Zhang. The consecutive ones submatrix problem for sparse matrices. *Algorithmica*, 48(3):287–299, 2007.

A A detailed example

In this section we give the details of applying the algorithm in Section 3 to the graph depicted in Figure 3. We just show the true entries that are produced by each operator. For the result of applying the final $\oplus$ operator see the main text.

1. The results of the algorithm, when applied to

$$H = (\textcircled{0}, \textcircled{1}, \textcircled{2}, \boxed{0\ 2}, \boxed{1\ 2}, \textcircled{0}, \boxed{0\ 1}, \boxed{0\ 2}, \textcircled{1}, \boxed{1\ 2}, \textcircled{2}, \boxed{0\ 2}, \boxed{1\ 2}).$$

- $h_2 : (\{(0, I), (1, I), (2, I)\}, \{\{0\}, \{1\}, \{2\}\})$
- $h_3 : (\{(0, H), (1, I), (2, C)\}, \{\{0, 2\}, \{1\}\})$,
  $(\{(0, C), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$, also h_2.
- $h_4 : (\{(0, H), (1, H), (2, C)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, C), (1, H), (2, H)\}, \{0, 1, 2\}\})$,
  $(\{(0, C), (1, H), (2, S)\}, \{0, 1, 2\}\})$,
  $(\{(0, I), (1, H), (2, C)\}, \{\{0\}, \{1, 2\}\})$,
  $(\{(0, I), (1, C), (2, H)\}, \{\{0\}, \{1, 2\}\})$, also h_3.
- $h_5 : (\{(0, I), (1, H), (2, C)\}, \{\{0\}, \{1, 2\}\})$,
  $(\{(0, I), (1, H), (2, H)\}, \{\{0\}, \{1, 2\}\})$,
  $(\{(0, I), (1, H), (2, S)\}, \{0\}, \{1, 2\}\})$.
- $h_6 : (\{(0, H), (1, H), (2, C)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, S), (2, C)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, H), (2, H)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, S), (2, H)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, H), (2, S)\}, \{0, 1, 2\}\})$,
  $(\{(0, H), (1, S), (2, S)\}, \{0, 1, 2\}\})$, also h_5.
- $h_7 : (\{(0, H), (1, H), (2, C)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, H), (2, H)\}, \{\{0, 1, 2\}\})$,
  $(\{(0, H), (1, H), (2, S)\}, \{\{0, 1, 2\}\})$,
  $(\{(1, H), (2, S)\}, \{0, 1, 2\}\})$, also h_6.
- $h_8 : (\{(0, I), (1, I), (2, C)\}, \{\{0\}, \{1\}, \{2\}\})$,
  $(\{(0, I), (1, I), (2, H)\}, \{\{0\}, \{1\}, \{2\}\})$,
  $(\{(0, H), (1, I), (2, C)\}, \{\{0, 2\}, \{1\}\})$,
  $(\{(0, H), (1, I), (2, H)\}, \{\{0, 2\}, \{1\}\})$,
  $(\{(0, H), (1, I), (2, S)\}, \{\{0, 2\}, \{1\}\})$,
  $(\{(1, I), (2, S)\}, \{\{0, 2\}, \{1\}\})$.

- $h_9 : (\{(0,I),(1,H),(2,C)\},\{\{0\},\{1,2\}\})$,
  $(\{(0,I),(1,H),(2,H)\},\{\{0\},\{1,2\}\})$,
  $(\{(0,I),(1,H),(2,S)\},\{\{0\},\{1,2\}\})$,
  $(\{(0,H),(1,H),(2,C)\},\{\{1,0,2\}\})$,
  $(\{(0,H),(1,H),(2,H)\},\{\{1,0,2\}\})$,
  $(\{(0,H),(1,H),(2,S)\},\{\{1,0,2\}\})$,
  $(\{(0,H),(2,S)\},\{\{1,0,2\}\})$,
  $(\{(2,S)\},\{0,1,2\}\})$, also h_8.
- $h_{10} : (\{(0,H),(1,I),(2,I)\},\{\{1\},\{0\},\{2\}\})$,
  $(\{(0,I),(1,H),(2,I)\},\{\{0\},\{1,\},\{2\}\})$,
  $(\{(0,H),(1,H),(2,I)\},\{\{0,1\},\{2\}\})$,
  $(\{(0,H),(2,I)\},\{\{0,1\}\{2\}\})$.
- $h_{11} : (\{(0,H),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,S),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,H),(1,H),(2,C)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,C),(1,H),(2,H)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,H),(1,H),(2,H)\},\{\{0,1,2\}\})$,
  $(\{(0,S),(1,H),(2,H)\},\{\{0,1,2\}\})$,
  $(\{(0,H),(2,H)\},\{\{0,1,2\}\})$,
  $(\{(0,S),(2,H)\},\{\{0,1,2\}\})$, also h_{10}.

2. The results of the algorithm, when applied to

$$H' = (\,⓪\,,\ ①\,,\ ②\,,\boxed{0\ 1},\boxed{0\ 2},\ ⓪\,,\boxed{0\ 1},\boxed{0\ 2},\ ②\,,\boxed{0\ 2},\ ①\,,\boxed{1\ 2})\,.$$

- $h'_2 : (\{(0,I),(1,I),(2,I)\},\{\{0\},\{1\},\{2\}\})$.
- $h'_3 : (\{(0,H),(1,C),(2,I)\},\{\{0,1\},\{2\}\})$,
  $(\{(0,C),(1,H),(2,I)\},\{\{0,1\},\{2\}\})$, also h'_2.
- $h'_4 : (\{(0,H),(1,I),(2,C)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,C),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
  $(\{(0,H),(1,C),(2,H)\},\{\{0,1,2\}\})$,
  $(\{(0,S),(1,C),(2,H)\},\{\{0,1,2\}\})$,
  $(\{(0,C),(1,H),(2,H)\},\{\{0,1,2\}\})$, also h'_3.
- $h'_5 : (\{(0,I),(1,C),(2,I)\},\{\{0\},\{1\},\{2\}\})$,
  $(\{(0,I),(1,H),(2,I)\},\{\{0\},\{1\},\{2\}\})$,
  $(\{(0,I),(1,I),(2,C)\},\{\{0\},\{1\},\{2\}\})$,
  $(\{(0,I),(1,I),(2,H)\},\{\{0\},\{1\},\{2\}\})$,
  $(\{(0,I),(1,C),(2,H)\},\{\{0\},\{1,2\}\})$,
  $(\{(0,I),(1,H),(2,H)\},\{\{0\},\{1,2\}\})$.
- $h'_6 : (\{(0,H),(1,C),(2,I)\},\{\{0,1\},\{2\}\})$,
  $(\{(0,H),(1,H),(2,I)\},\{\{0,1\},\{2\}\})$,
  $(\{(0,H),(1,S),(2,I)\},\{\{0,1\},\{2\}\})$,
  $(\{(0,H),(1,C),(2,C)\},\{\{0,1\},\{2\}\})$,

$(\{(0,C),(1,H),(2,C)\},\{\{0,1\},\{2\}\})$,
$(\{(0,H),(1,C),(2,H)\},\{\{0,1\},\{2\}\})$,
$(\{(0,C),(1,H),(2,H)\},\{\{0,1\},\{2\}\})$,
$(\{(0,H),(1,C),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,H),(1,H),(2,H)\},\{\{0,1,2\}\})$
$(\{(0,H),(1,S),(2,H)\},\{\{0,1,2\}\})$, also h'_5.

- $h'_7 : (\{(0,H),(1,C),(2,C)\},\{\{0,2\},\{1\}\})$,
$(\{(0,C),(1,C),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,H),(2,C)\},\{\{0,2\},\{1\}\})$,
$(\{(0,C),(1,H),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,I),(2,C)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,I),(2,S)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,C),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,H),(1,C),(2,S)\},\{\{0,1,2\}\})$,
$(\{(1,C),(2,S)\},\{\{0,1,2\}\})$,
$(\{(0,H),(1,H),(2,H)\},\{\{0,1,2\}\})$
$(\{(0,H),(1,H),(2,S)\},\{\{0,1,2\}\})$
$(\{(0,S),(1,C),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,S),(1,H),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,H),(1,S),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,S),(1,S),(2,H)\},\{\{0,1,2\}\})$,
$(\{(0,S),(1,C),(2,S)\},\{\{0,1,2\}\})$, also h'_6.
- $h'_8 : \{(0,I),(1,I),(2,H),\{\{0\},\{1\},\{2\}\}$,
$\{(0,H),(1,I),(2,I),\{\{0\},\{1\},\{2\}\}$,
$\{(0,H),(1,I),(2,C),\{\{0\},\{1\},\{2\}\}$,
$\{(0,H),(1,I),(2,H),\{\{0\},\{1\},\{2\}\}$,
$\{(0,C),(1,I),(2,H),\{\{0,2\},\{1\}\}$,
$\{(0,H),(1,I),(2,H),\{\{0,2\},\{1\}\}$,
$\{(0,H),(1,I),(2,S),\{\{0,2\},\{1\}\}$,
$(\{(0,H),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,H),(1,I),(2,S)\},\{\{0,2\},\{1\}\})$,
$(\{(1,I),(2,S)\},\{\{0,2\},\{1\}\})$,
$(\{(0,S),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,S),(1,I),(2,S)\},\{\{0,2\},\{1\}\})$.
- $h'_9 : \{(0,C),(1,I),(2,I),\{\{0\},\{2\},\{1\}\}$,
$\{(0,H),(1,I),(2,I),\{\{0\},\{2\},\{1\}\}$,
$\{(0,H),(1,I),(2,I),\{\{0\},\{2\},\{1\}\}$,
$(\{(0,H),(1,I),(2,I)\},\{\{0\},\{1\},\{2\}\})$,
$(\{(1,I),(2,S)\},\{\{0\},\{1\},\{2\}\})$,
$(\{(0,S),(1,I),(2,I)\},\{\{0\},\{1\},\{2\}\})$.
- $h'_{10} : (\{(0,H),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,S),(1,I),(2,H)\},\{\{0,2\},\{1\}\})$,
$(\{(0,S),(1,I)\},\{\{0,2\},\{1\}\})$, also h'_9.