RAIRO-Theor. Inf. Appl. 58 (2024) 18
https://doi.org/10.1051/ita/2024015

RAIRO - Theoretical Informatics and Applications
www.rairo-ita.org

ENUMERATING MINIMUM FEEDBACK VERTEX SETS IN DIRECTED GRAPHS WITH UNION-CAT TREES

Moussa Abdenbi*, Alexandre Blondin Massé, Alain Goupil, Mélodie Lapointe and Martin Lavoie

Abstract. The problem of finding a minimum feedback vertex set (MFVS) in a directed graph has been known to be NP-hard for around 40 years: It is one of the problems listed in Karp's famous 1972 paper. Several strategies to solve the MFVS problem, both exact and approximate, have been proposed. In particular, in 2000, Lin and Jou presented an exact algorithm based on eight graph contraction operators whose complexity is polynomial for a particular class of graphs called DOME-contractible graphs. This paper proposes two contributions. First, we introduce a data structure called union-cat tree that provides, in some cases, a compact representation of a family of constant size subsets of a given finite set. Secondly, we extend Lin and Jou's algorithm to compute the set of all MFVSs of any directed graph.

Mathematics Subject Classification. 05C05, 05C20, 05C30.

Received December 15, 2022. Accepted October 15, 2024.

1. Introduction

The *minimum feedback vertex set problem* is a well-known problem in graph theory, both for directed graphs and simple graphs. The directed version was first introduced by Karp in his famous paper listing important and challenging combinatorial NP-hard problems [1]: Given a directed graph $G = (V, A)$, identify a subset of vertices $U \subseteq V$ of minimum size such that the digraph obtained by removing U from G is acyclic, *i.e.* does not contain any circuit. Polynomial time algorithms have been proposed for special classes of graphs, such as cyclically reducible graphs [2], interval graphs [3] and permutations graphs [4]. In 1988, Levy and Low introduced the class of *contractible graphs*, strictly containing the classes of interval graphs and of cyclically reducible graphs, for which one can compute a MFVS in polynomial time [5]. Their ideas were improved in 2000 by Lin and Jou [6] who introduced three additional operators, yielding a larger class of contractible digraphs for which MFVS can be computed in polynomial time as well. An interesting property of the approach based on graph contraction is that the eight operators defined in [5] and [6] can be successfully combined with other methods, such as integer linear programming [7], meta-heuristics [8, 9], binary decision diagrams (BDD) [10] and fixed-parameter theory [11].

Another interesting problem related to the MFVS problem is that of efficiently *enumerating* all MFVS for a given graph. Few results have been proposed to date. In 2008, Formin et al. proposed another algorithm to compute a MFVS for undirected graphs in $O(1.7548^n)$ by transforming the problem into that of finding

Keywords and phrases: Directed graphs, Feedback vertex set, Union-cat tree.

* Corresponding author: abdenbi.moussa@uqam.ca

maximum induced subforests [12]. Also, in [13], Formin et al. provide an algorithm for enumerating all minimal (*inclusion-wise*, not *cardinality-wise*) feedback vertex sets in polynomial time. To our knowledge, this is so far the only nontrivial algorithm related to the enumeration of MFVSs.

Our paper proposes two contributions to the enumeration of MFVSs in directed graphs. The first is the introduction of a data structure, called *union-cat tree*, which provides a compact representation of a family of sets having the same cardinality. The second is an algorithm that computes the union-cat tree representing the set of all MFVSs of any directed graph.

This document is structured as follows. Section 2 introduces the pertinent definitions and notation. In Section 3, we motivate and formally define the union-cat tree data structure along with some basic operations. The algorithm building the union-cat tree is described in Section 4. Section 5 explains our specific motivation for designing an algorithm that computes the set of MFVSs of a directed graph. We conclude with a performance analysis of our algorithm in Section 6. In Section 7, we discuss further perspectives and propose a conjecture.

It is worth mentioning that this work is a full version of a 2-page extended abstract and communication – reporting preliminary results – that was presented at the Bordeaux Graph Workshop in 2012 [14].

2. Definitions and notations

A *directed graph*, or digraph, is an ordered pair $G = (V, A)$, where V is a finite set of *vertices* and $A \subseteq V \times V$ is a finite set of *arcs*.

Given a vertex $u \in V$, $N^-(u) = \{v \in V \mid (v, u) \in A\}$ is called the set of *predecessors* of u and $N^+(u) = \{v \in V \mid (u, v)\}$ the set of *successors* of u. The set of *neighbors of* u is $N(u) = N^-(u) \cup N^+(u)$. Similarly, $A^-(u) = \{(v, u) \mid v \in N^-(u)\}$ is the set of *incoming arcs* of u and $A^+(u) = \{(u, v) \mid v \in N^+(u)\}$ is the set of *outgoing arcs*) from u in G and the set of *arcs incident to* u is $A(u) = A^-(u) \cup A^+(u)$. The *in-degree* of u is given by $\deg^-(u) = |N^-(u)|$ and the *out-degree* of u is similarly defined by $\deg^+(u) = |N^+(u)|$. We say that $H = (V_H, A_H)$ is a *subgraph* of $G = (V, A)$, if $V_H \subseteq V$ and $A_H \subseteq \{(u, v) \in A \mid u, v \in V_H\}$. Given $U \subseteq V$, the *subgraph induced by* U *in* G, denoted by $G[U]$, is the subgraph of G with set of vertices U and set of arcs $A \cap (U \times U)$.

A *path* in G is any sequence $(u_0, u_1, \ldots, u_n)$ of vertices of G such that $(u_i, u_{i+1}) \in A$, for $i = 0, 1, \ldots, n-1$. The integer n is called the *length of the path*. In particular, if $n = 0$, it is called an *empty path based at* u_0. A *circuit* is a nonempty path starting and ending with the same vertex. A digraph is called *acyclic* (or *circuit-free*) if it does not contain any circuit.

A *feedback vertex set* is a set $U \subseteq V$ such that $G[V - U]$ is acyclic or, equivalently, if for any circuit c of G, at least one vertex of c belongs to U. A *minimum feedback vertex set* (MFVS) is a feedback vertex set of G having minimum cardinality. The problem of finding an MFVS in a directed graph is known to be NP-hard since 1972 [1]. The set of all MFVSs of a given digraph G is denoted by $\mathrm{MFVS}(G)$.

Given $u, v \in G$, we write $u \to v$ if there exists a path (possibly empty) from u to v, and $u \leftrightarrow v$ if $u \to v$ and $v \to u$. It is easy to prove that $u \leftrightarrow v$ is an equivalence relation. The subgraphs induced by each equivalence class of the relation $\leftrightarrow$ are called the *strongly connected components* of G. A *clique* of G is a set of vertices $U \subseteq V$ such that $G[U]$ is a complete graph without loops, *i.e.* for every $u, v \in U$ such that $u \neq v$ (i) $(u, u) \notin A$ and (ii) $(u, v) \in A$.

3. Union-cat trees

When treating large sets of subsets, it rapidly becomes impracticable to store each of the subsets in memory explicitly. Therefore, one might resort to data structures that compactly represent the subsets implicitly.

Some commonly used data structure are the so-called *binary decision diagrams (BDD)*, the *reduced ordered binary decision diagrams (ROBDD)* [15] and its variants, such as the *zero-suppressed BDD (ZDD)* [16]. The idea can be summarized as follows. Let S be some finite set and $\mathcal{F}$ a family of subsets of S. Then $\mathcal{F}$ can be represented by a boolean function $f : \{0, 1\}^{|S|} \to \{0, 1\}$ whose evaluation indicates whether some given subset

$A \subseteq S$ belongs to $\mathcal{F}$ or not, *i.e.* $f(A) = 1$ if and only if $A \in \mathcal{F}$. BDDs and ZDDs have been extensively studied: Donald Knuth gave two celebrated lectures about the subject [17, 18], in which he mentioned that Bryant's paper [15] was the most-cited in the field for several years. BDD and ZDD have many applications, ranging from genetics [19] to data mining [20] and game theory [21]. BDDs and ZDDs are also very useful in combinatorial problems when one wishes to represent all solutions to a given problem [16]. In particular, several enumeration algorithms rely on the classical branch-and-cut technique, according to whether some well-chosen variable is included or excluded from the solution [22].

However, binary branching is not always adaptated, so that BDDs/ZDDs are sometimes less convenient, especially when an exhaustive enumeration of the elements of the family is not possible. As an example, in [6], the authors propose a branch-and-cut algorithm with two branching rules: (1) When the digraph is not connected, according to each of its strongly connected components and (2) When all other rules have been exhausted, the branching is done according to whether some well-chosen vertex should be included or excluded from the solution. While BDDs/ZDDs naturally supports Rule (2), Rule (1) is less convenient to handle.

Before formally defining the union-cat tree data structure, we need additional definitions and notation. Let S be a finite set. Any set $\mathcal{F} \subseteq 2^S$ is called a *family* on S, *i.e.* $\mathcal{F}$ is a collection of subsets of S. All usual set theoretic operations are naturally defined for families. Given two families $\mathcal{F}$ and $\mathcal{G}$, the *catenation* of $\mathcal{F}$ and $\mathcal{G}$ is defined by

$$\mathcal{F} * \mathcal{G} = \{E \cup F \mid E \in \mathcal{F}, F \in \mathcal{G}\}.$$

The word *catenation* is borrowed from the operation of catenation of regular expressions. It is straightforward to see that the catenation operation is both associative and commutative, and has the empty family as a unit. More generally, if $\mathcal{F}_1$, $\mathcal{F}_2$, ..., $\mathcal{F}_n$ are families on S, then

$$\overset{n}{\underset{i=1}{\circledast}} \mathcal{F}_i = \left\{ \bigcup_{i=1}^{n} E_i \;\middle|\; E_i \in \mathcal{F}_i, \text{for } i = 1, 2, \ldots, n \right\}.$$

Union-cat trees are then defined by the following graph structure in Definition 3.1, which is consolidated with the maps from Definition 3.3 to make it concise and efficient.

Definition 3.1. Let S be a finite set, T a rooted tree and V the vertices of T. For any $v \in V$, denote by v.children the (possibly empty) set of all children nodes of v. Then T is called a *union-cat tree* (UCT) if there exists a map

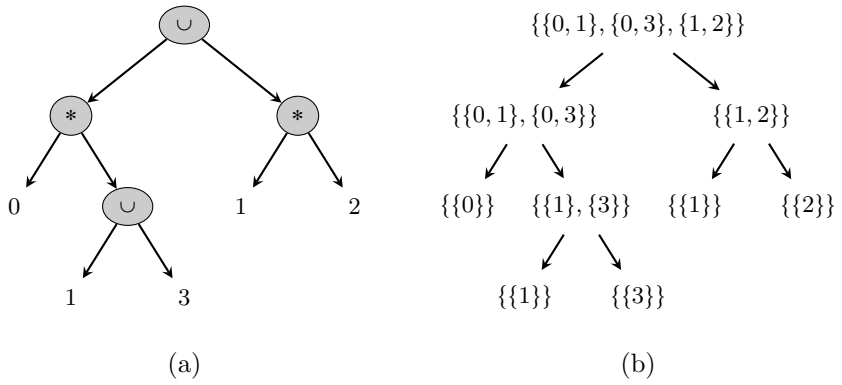
$$\textsc{Content} : V \to S \cup \{*, \cup\}$$

such that for every node $v \in V$ we have,

(i) If v.children $= \varnothing$ (v is a leaf), then $\textsc{Content}(v) \in S$;
(ii) If v.children $\neq \varnothing$ (v is an internal node), then
- either $\textsc{Content}(v) = *$ and then v is called a *catenation node*,
- or $\textsc{Content}(v) = \cup$ and then v is called a *union node*.

Given a union-cat tree T, we can associate a unique family $\mathcal{F}$, computed recursively as follows. The basic case is a leaf v of T, which is represented by the set $\{\{\textsc{Content}(v)\}\}$. For each internal node v of T, we apply the operations $\cup$ or $*$ of each internal node v on all v.children. This construction is illustrated in the next example.

Example 3.2. Let $S = \{0, 1, 2, 3\}$ and $\mathcal{F} = \{\{0, 1\}, \{0, 3\}, \{1, 2\}\}$. The union-cat tree T of Figure 1 (a) provides a representation of $\mathcal{F}$. The Figure 1 (b) shows the recursive steps in the construction of $\mathcal{F}$. We start with representing the leaf nodes of T, v with the corresponding sets $\{\{\textsc{Content}(v)\}\}$ and we apply recursively the operations in internal nodes. The union node with leaves 1 and 3 is replaced by $\{\{1\}, \{3\}\} = \{\{1\}\} \cup \{\{3\}\}$.



(a) (b)

FIGURE 1. (a) A union-cat tree T representing the family $\mathcal{F} = \{\{0,1\},\{0,3\},\{1,2\}\}$ on the set $S = \{0,1,2,3\}$. (b) The same union-cat tree with vertex labels given by the recursive construction of the family $\mathcal{F}$.

Then the left catenation node is replaced by $\{\{0,1\},\{0,3\}\} = \{\{0\}\} * \{\{1\},\{3\}\}$. The right catenation node is replaced by $\{\{1,2\}\} = \{\{1\} * \{2\}\}$. Finally, we get the family $\mathcal{F}$ by replacing the root union node of T by $\{\{0,1\},\{0,3\},\{1,2\}\} = \{\{0,1\},\{0,3\}\} \cup \{\{1,2\}\}$.

Note that the current definition of a UCT could allow redundancies, *i.e.* some members of the family $\mathcal{F}$ can be obtained from different branches. To avoid these, we introduce the following maps, which also serve to formalize the recursive construction process described in Example 3.2.

Definition 3.3. Let S be a finite set, T a union-cat tree on S. Then T is a *non ambiguous representation* of a family $\mathcal{F}$ on S, if there exist two maps $\textsc{Values} : V \to 2^S$ and $\textsc{Family} : V \to 2^{2^S}$ such that the following conditions are verified,

(i) For each $v \in V$,

$$\textsc{Values}(v) = \begin{cases} \{\textsc{Content}(v)\}, & \text{if } v \text{ is a leaf;} \\ \bigcup_{w \in v.\text{children}} \textsc{Values}(w) & \text{if } v \text{ is an internal node.} \end{cases}$$

(ii) For each $v \in V$,

$$\textsc{Family}(v) = \begin{cases} \{\{\textsc{Content}(v)\}\}, & \text{if } v \text{ is a leaf;} \\ \bigcup_{w \in v.\text{children}} \textsc{Family}(w) & \text{if } v \text{ is a union node;} \\ *_{w \in v.\text{children}} \textsc{Family}(w) & \text{if } v \text{ is a catenation node.} \end{cases}$$

(iii) For each union node $v \in V$ such that $w, w' \in v.\text{children}$, the condition $\textsc{Family}(w) \cap \textsc{Family}(w') \neq \varnothing$ implies $w = w'$;

(iv) For each catenation node $v \in V$ such that $w, w' \in v.\text{children}$, the condition $\textsc{Values}(w) \cap \textsc{Values}(w') \neq \varnothing$ implies $w = w'$.

Roughly speaking, constraint (iii) of Definition 3.3 guarantees that the union operation is always disjoint – so that no subset is represented more than once, while condition (iv) ensures that any catenation is done on disjoint families. From now on, we refer to the *UCT data structure* (or union-cat tree) as a UCT enhanced with the maps Values and Family from Definition 3.3.

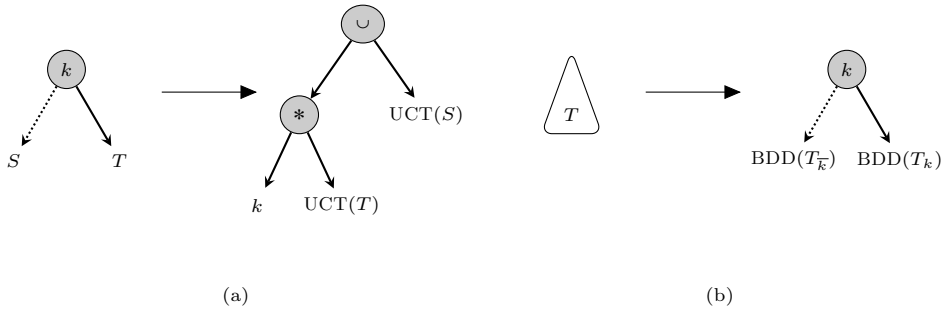


FIGURE 2. (a) Transforming a BDD/ROBDD/ZDD into a UCT. It suffices to apply UCT on both child nodes S and T and then combine them as in the picture above. (b) Transforming a UCT into a BDD. At each step, some variable k is chosen and then BDD is called on $T_{\overline{k}} = \textsc{Subfamily}(T, S - \{k\})$ and $T_k = \textsc{Subfamily}(T, \{k\})$.

Clearly, union-cat trees are not always adapted for representing some families, but they turn out to be quite compact when computing large families with particular structures, such as when catenation nodes partition the values in large enough groups. It should be noted that union-cat trees are very similar to another data structure known as *and/or trees*, which are mostly used in the artificial intelligence community [23] and as an exploration technique of large game tree space [24].

Basic and advanced operations can be defined recursively on the union-cat tree data structure. In many cases, they consist in a depth-first traversal with different actions depending on the type of node visited. To make the content of this paper easier to read, we present some of them in the Appendix A.

Before concluding this section, it is worth mentioning that union-cat trees and BDDs/ZDDs can be naturally converted from and to each other. Both conversions are depicted in Figure 2. To transform a BDD into a UCT, it suffices to use union and catenation nodes so that the value k is included (in the left subtree) or excluded (in the right subtree). This is clearly done by a single traversal of the BDD, and the nodes are inserted locally, so that the overall complexity is $\mathcal{O}(n)$. Conversely, it is possible to transform a UCT into a BDD. It suffices to use the SUBFAMILY modifier (see Appendix A) to partition the current tree according to whether some value k is included or not. The complexity is $\mathcal{O}(n^2)$, since SUBFAMILY takes $\mathcal{O}(n)$ and it is applied at most twice for each value. In the case where the UCT is "well-balanced", it is reasonable to expect a complexity closer to $\mathcal{O}(n \log n)$.

4. Enumeration algorithm

This section is devoted to the description of an algorithm that computes the union-cat tree representing the set MFVS(G) of any directed graph G. First, we need to introduce useful definitions.

Definition 4.1. Let $G = (V, A)$ be a digraph and $u \in V$. We say that u is

(i) *MFVS-essential* (or simply *essential*) if it belongs to every minimum feedback vertex set of G;
(ii) *MFVS-useless* (or simply *useless*) if it does not belong to any minimum feedback vertex set of G.

Essential and useless vertices can be contracted leaving the set of MFVSs unchanged. For example, vertices having a loop are essential, while vertices with no successor or no predecessor are useless.

The next two natural relations on vertices prove to be useful.

Definition 4.2. Let $G = (V, A)$ be a digraph and $u, v \in V$. We say that

(i) u *is dominated by* v, written $u \preceq v$, if for any $U \in \text{MFVS}(G)$, $u \in U$ implies $(U - \{u\}) \cup \{v\} \in \text{MFVS}(G)$;
(ii) u *and* v *are equivalent*, denoted by $u \equiv v$, if $u \preceq v$ and $v \preceq u$.

Let $U \in \text{MFVS}(G)$. It is easy to see that if $u \equiv v$, then $u \in U$ implies $v \notin U$, *i.e.* both vertices cannot belong to U. Indeed, arguing by contradiction, if $u, v \in U$, since $(U - \{u\}) \cup \{v\} = U - \{u\}$ is a MFVS with cardinality one less than U, we have a contradiction with the minimality of U.

It is also immediate that $\equiv$ is an equivalence relation. The interest of equivalent vertex pairs is that they can be merged and we only need to keep track of the equivalence classes when building the union-cat tree representing all MFVS. More precisely, for any $U \in \text{MFVS}(G)$, where G is any digraph, if $u \in U$ and $u \equiv v$ then, we can replace u in U by v. Finally, further properties linking the relation $\preceq$ with the concepts of essential and useless vertices can speed up Algorithm 1 significantly in some cases.

Lemma 4.3. *Let $G = (V, A)$ be a digraph and $u, v \in V$.*

(i) *If u has a unique successor v, then $u \preceq v$;*
(ii) *If v has a unique predecessor u, then $v \preceq u$;*
(iii) *If v has a unique predecessor u and u has a unique successor v, then $u \equiv v$.*

Proof. (i) Let $U \in \text{MFVS}(G)$ such that $u \in U$. Since every circuit passing through u also passes through v, then we can replace u by v in U, so that $(U - \{u\}) \cup \{v\}$ is a $\text{MFVS}(G)$. Hence $u \preceq v$.
(ii) Symmetric argument to (i).
(iii) Follows from (i) and (ii). □

Finally, we can infer that some vertices are useless without having to explicitly verify it:

Lemma 4.4. *Let $G = (V, A)$ be a digraph and $u, v \in V$.*

(i) *If $u \preceq v$ and v is essential, then u is useless.*
(ii) *If $u \preceq v$ and v is useless, then u is useless.*

Proof. (i) Arguing by contradiction, suppose that there exists $U \in \text{MFVS}(G)$ such that $u \in U$. But v is essential so that $v \in U$ as well. Since every circuit c passing through u also passes through v, this implies that $U - \{u\}$ is an FVS of G, contradicting the minimality of U.
(ii) Arguing by contradiction, suppose that there exists $U \in \text{MFVS}(G)$ such that $u \in U$. By definition of $\preceq$, this implies that $(U - \{u\}) \cup \{v\} \in \text{MFVS}(G)$, contradicting the assumption that v is useless. □

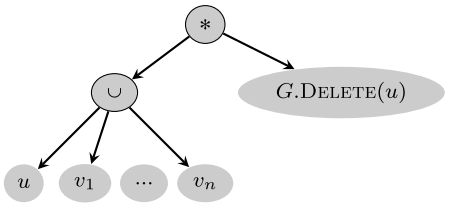
Finally, we note that some arcs can be removed without modifying the set of MFVS appearing in a given directed graph, such as *acyclic arcs*, *i.e.* arcs that do not belong to any circuit. The following definition provides a generalization of this type of arcs.

Definition 4.5. Let $G = (V, A)$ be a digraph, $(u, v) \in A$ an arc and C a circuit in G with vertex set U.

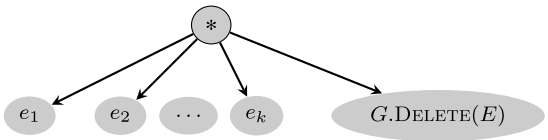
(i) C is said to be *minimal* if there does not exist a circuit C' in G with vertex set U' such that $U' \subset U$;
(ii) (u, v) is said to be *superfluous* if it does not belong to any minimal circuit.

In order to verify that a subset of vertices is indeed a FVS, it suffices to check that it has a nonempty intersection with every *minimal* circuit. In other words, if a set U of vertices has a nonempty intersection with every minimal circuit, then it covers every circuit. Indeed if C is a circuit that is not minimal, it contains a minimal circuit C' and $U \cap C$ is nonempty because $U \cap C'$ is nonempty. Therefore when looking for a FVS, one can disregard all arcs and vertices that do not belong to at least one minimal circuit. Note that arcs identified by the contractions Pie and Dome from [6] and Dome++ and Allcycles from [25], are superflous arcs.

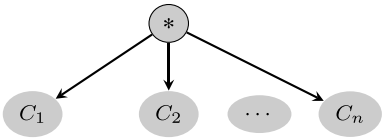
Before describing our enumeration algorithm we need some basic graph operations. Let $G = (V, A)$ be a directed graph, U a subset of V, and $u, v \in U$. Then $G.\text{Delete}(u)$ is the graph $(V - \{u\}, A - A(u))$. Moreover, if $(u, v) \in A$, then $G.\text{Merge}(u, v)$ is the graph with set of vertices $V - \{v\}$ and with set of arcs $(A \cup \{(w, u) \mid w \in N^-(v) - \{u\}\} \cup \{(u, w) \mid w \in N^+(v)\}) - A(v)$. Finally, $G.\text{Vanish}(u)$ is the digraph with set of vertices $V - \{u\}$ and set of arcs $(A \cup \{(v, w) \mid v \in N^-(u), w \in N^+(u)\}) - A(u)$. This last operation consists of deleting the vertex u while preserving the links between its predecessors and successors.



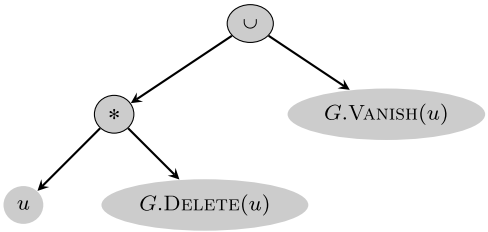
(A) Each time the NEWLEAF(u) function is called, if the equivalence class of u only contains u, then we create a leaf node containing u; otherwise, we create the above subtree, where $v_1, \ldots, v_n$ are the other vertices equivalent to u.



(B) If the set of essential vertices E is nonempty, i.e. $E = \{e_1, e_2, \ldots, e_k\}$, we can add them to the tree and remove E from the graph.



(C) Whenever there is more than one connected component, we divide the problem into smaller pieces.



(D) Otherwise, we branch according to some well-chosen vertex.

FIGURE 3. The different cases encountered when computing the union-cat tree.

We are now ready to describe our enumeration algorithm. It takes as input any directed graph and returns its set of minimum feedback vertex sets represented as a union-cat tree. The strategy goes as follows (see Figure 3):

1. Identify equivalent vertex pairs and merge them (it is important to keep track of equivalent vertices for future steps, see Figure 3a);
2. Identify useless and essential vertices by taking into account Lemmas 4.3 and 4.4 (for speed purposes);
3. Make useless vertices vanish;
4. Apply contractions identifying superfluous arcs;
5. Remove essential vertices and add them to the union-cat tree (Fig. 3b);
6. If the digraph has more than one connected components, split the problem into these (Fig. 3c);
7. Otherwise, choose some vertex (*e.g.* the one having maximum degree) and branch according to whether the vertex is included in or excluded from the solution (Fig. 3d).

Algorithm 1 Computing the union-cat tree of all MFVSs from any digraph

```
1:  function GetTree(G : digraph) : union-cat tree
2:      Merge all equivalent vertex pairs
3:      Make all useless vertices vanish
4:      Reduce digraph by removing superfluous arcs
5:      Let E be the set of essential vertices
6:      x ← NewNode()
7:      if |E| > 0 then
8:                                          ▷ Essential vertices may be removed
9:          x.content ← ∗
10:         for u ∈ E do
11:             Append NewLeaf(u) to x.children
12:         G′ ← G.Delete(E)
13:         Append GetTree(G′) to x.children
14:     else
15:         C ← ConnectedComponents(G)
16:         if |C| > 1 then
17:                                          ▷ Branch according to the CC's
18:             x.content ← ∗
19:             for C ∈ C do
20:                 Append GetTree(C) to x.children
21:         else
22:                                          ▷ Branch according to some vertex
23:             Let u be any vertex
24:             x.content ← ∪
25:             y ← EmptyNode()
26:             y.content ← ∗
27:             y.children ← {NewLeaf(u)}
28:             G′ ← G.Delete(u)
29:             Append GetTree(G′) to y.children
30:             G′ ← G.Vanish(u)
31:             Append GetTree(G′) to x.children
32:     return x
```

The complete pseudocode appears in Algorithm 1 and an output example is depicted in Figure 6. We assume that there is a data structure (such as disjoint sets) keeping track of equivalent vertices, which are inserted in the tree whenever the NewLeaf(u) function is called (Lines 11 and 27).

5. Motivation

The minimum feedback vertex set problem appears in numerous real-life problems, ranging from computer-aided design [26] and scan design [27] to Bayesian inference [28]. More recently, it turned out to be important in cognitive science, providing a natural formalization of the *symbol grounding problem* [29].

More precisely, the symbol grounding problem consists in identifying the smallest sets of *primitive* words of meaning already known and such that we are then able to define all remaining words out of this minimal set. For this purpose, we represent dictionaries as directed graphs in which vertices stand for words and arcs (u, v) indicate that word u appears in the definition of word v. An example of such a dictionary-graph is shown in Figure 4. Roughly speaking, if one knows a set of words U that is a feedback vertex set in some

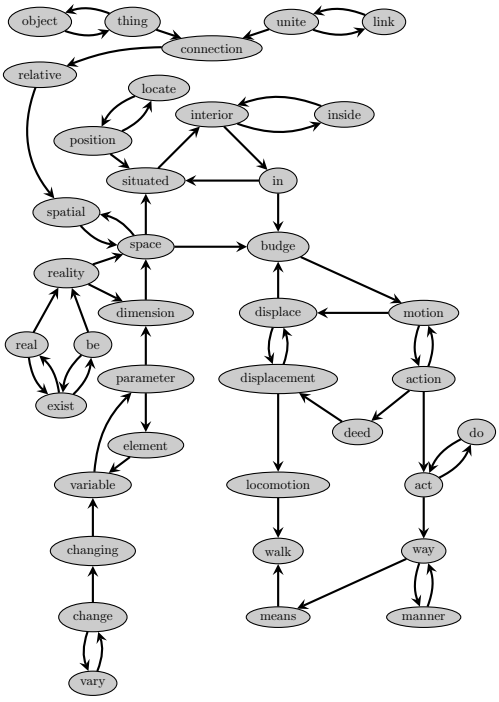


FIGURE 4. A dictionary graph created from an anonymous player in the dictionary game.

given dictionary-graph, then it means that the "circularity" in the definition graph is "broken", when the set U is removed. Thus allowing one to learn all remaining words through definition alone (on condition that the meanings of the words in the FVS are already known by some other means). MFVSs are the smallest such sets of grounding words. Although the problem of finding MFVS is known to be NP-hard for general digraphs [1], we were able to compute several MFVSs for small full dictionaries of around 100 000 words, Longman's Dictionary of Contemporary English [30] and Cambridge's International Dictionary of English [31]. These MFVS words turn out to have distinctive psycholinguistic characteristics: they are learned at a younger age, have more concrete meaning and are more frequently used [32]. However, the problem of computing only one MFVS of larger dictionaries such as Merriam-Webster and WordNet [33] (the corresponding graph definition has more than 132 500 vertices and 700 000 edges) is at present out of reach. Moreover, even small dictionaries (less than 80 000 words) were too big to generate all their MFVSs, and the ones we generated were not independent (in the probabilistic sense).

In order to generate smaller dictionaries for analysis purpose, we have designed an online one-player game, called "The dictionary game", in which the participant is asked to build a dictionary [34]. The game requires a starting word defined by the player, then the player must define the words used in the initial definition and so on, until all the words used in all definitions have been defined. The goal is to finish the game using the smallest number of words. It turns out that all dictionaries generated so far have only a few hundreds words and are hence amenable to a new method of enumerating all MFVS which we describe in the next section. For dictionaries of this size we are able to generate all MFVSs and a *uniform random sample* of MFVSs for psycholinguistic analysis, free of any bias from the algorithm design.

6. PERFORMANCE

To evaluate the performance of our algorithm, we use contractions from [5, 6]. More precisely, we use IN1 and OUT1 to identify vertices to merge, IN0 and OUT0 to identify useless vertices, LOOP to identify essential

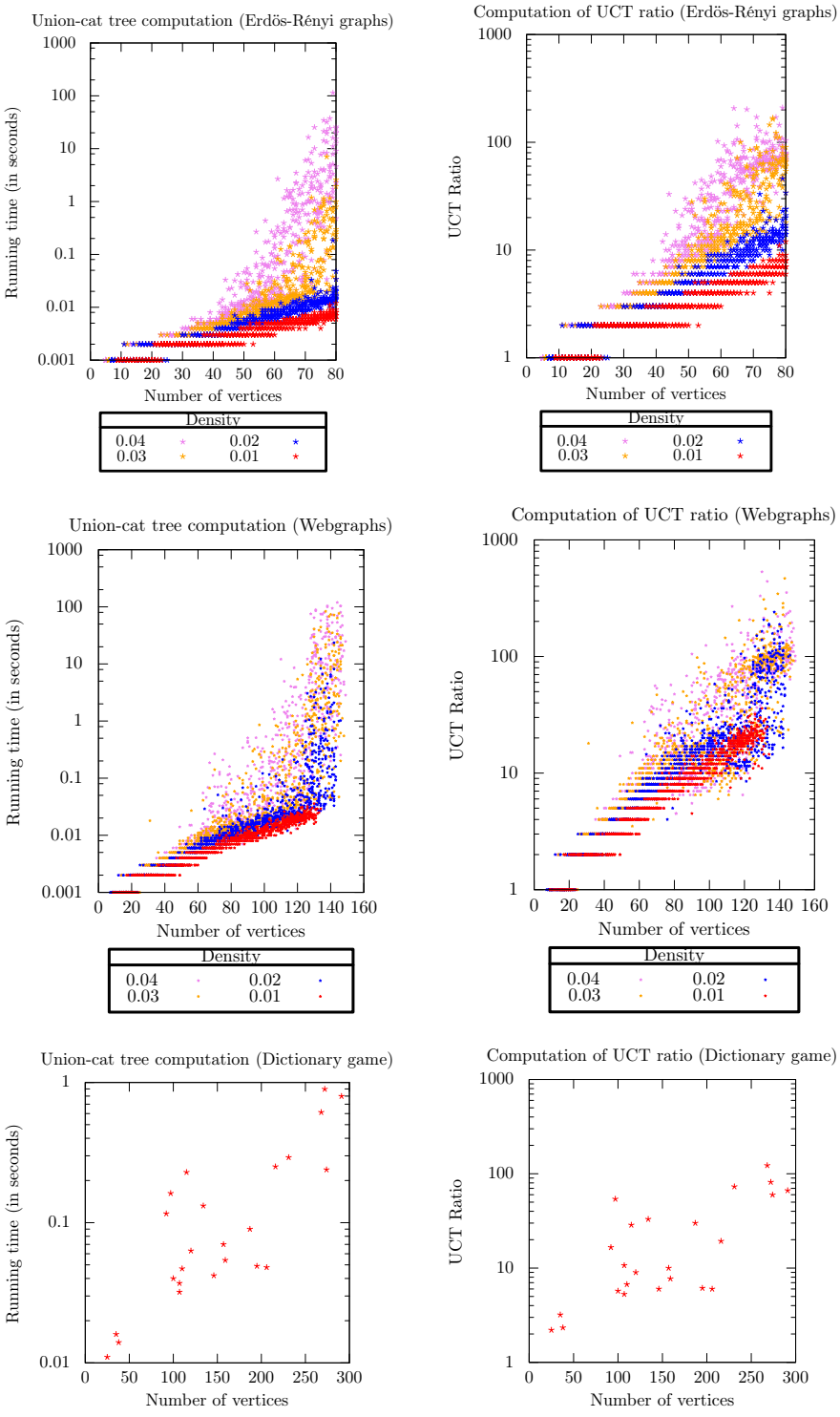


FIGURE 5. Running time and UCT ratio observed when running Algorithm 1 on random Erdős-Rényi graphs, random webgraphs and 24 dictionaries produced in the game by anonymous participants.

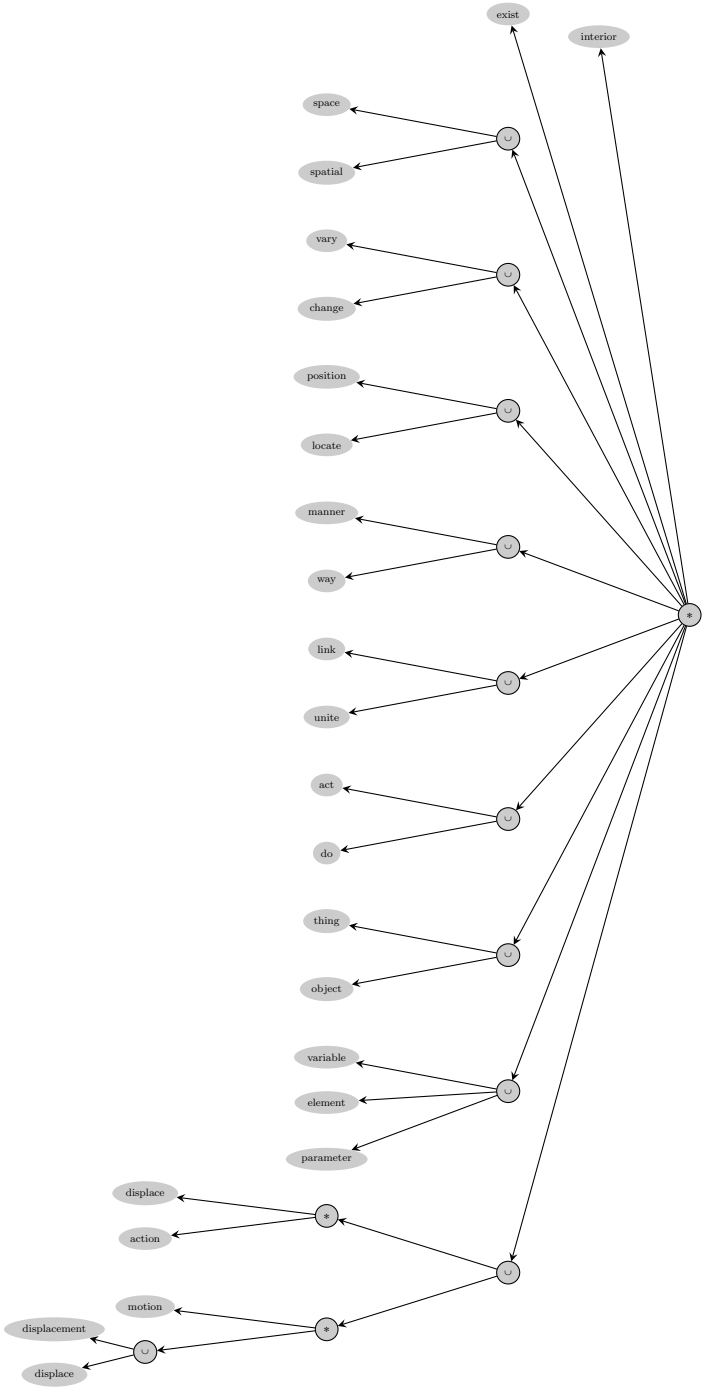


FIGURE 6. An example of output produced by Algorithm 1 from a dictionary of 38 words. It uses 38 nodes (and therefore 37 arcs) to represent a set of 1152 MFVSs each containing 12 vertices.

vertices and Pie and Dome to identify superfluous arcs. We have randomly generated graph samples from two families of directed graphs: Erdős-Rényi [35] and webgraphs [36]. The algorithm was also tested on the dictionaries obtained in the dictionary game described in Section 5.

Erdős-Rényi graphs [35] are generated as follows. We fix the number of vertices n and the expected density $p \in [0,1]$. Then, for every possible arc (u,v), we include it with probability p. Webgraphs are graphs with a structure similar to that of the Web. Many models have been studied; we used a publicly available Python generator [36] that implements the R-MAT algorithm [37].

In the case of Erdős-Rényi graphs, we generated 10 random instances for each size between 1 and 80 and density between 0.01 and 0.04. For the webgraphs, we could extend the number of vertices up to 150 with the same densities. The graphics describing the running time appear in the left column of Figure 5. The three graphics in the right column indicate the ratio of time used to compute the union-cat tree over the time used to compute one MFVS with Lin and Jou's algorithm (called *UCT ratio*). All growths appear exponential, except for the dictionary graphs; the difference deserves to be explored.

Unsurprisingly, the performance for Erdős-Rényi graphs is not as good as that for webgraphs. This is probably explained by the fact that the eight reduction operators apply more often on the latter than on the former. In the case of the 24 dictionaries produced in the dictionary game, computing one MFVS took at most 1 second and the UCT ratio is a bit more than 100, even if the graphs have up to 300 vertices. These results are consistent since all dictionaries are completely or almost-completely contractible.

7. Concluding remarks

We conclude by suggesting two avenues worth exploring. First, as pointed out above, it seems that the union-cat data structure is novel and can be used in many other combinatorial problems. A thorough study comparing it to other BDD-like data structures, as well as applications to other problems should be carried out.

Finally, computer exploration combined with the two lower graphics of Figure 5 and the DOME-contractible concept from [6] lead us to formulate the following conjecture:

Conjecture 7.1. *Let $G = (V, A)$ be a DOME-contractible digraph. Then there exists a polynomial size union-cat tree representing* MFVS(G).

As a first step, one could prove (or disprove) Conjecture 7.1 for contractible graphs instead of DOME-contractible graphs.

Appendix A.

We present some basic operations on the Union-Cat tree data structure. Obviously, there exist many other queries and modifiers that could be defined on UCTs, such as complementation, union, intersection and difference.

Queries

Queries are operations used to retrieve information from a data structure without modifying it. Given some family $\mathcal{F}$ on a set S, we might be interested in:

(1) Computing the cardinality of the family (Card($\mathcal{F}$));
(2) Checking whether a given set E belongs to the family, written $E \in F$;
(3) Choosing a random set with uniform probability (Random($\mathcal{F}$));

All these operators are naturally extended to the *nodes* of some given UCT. For instance, to compute Card($\mathcal{F}$), it suffices to define recursively Card on leaf nodes, catenation nodes and union nodes and then Card($\mathcal{F}$) =

$\textsc{Card}(r)$, where r is the root of T:

$$\textsc{Card}(v) = \begin{cases} 1 & \text{if } v \text{ is a leaf;} \\ \sum_{w \in v.\text{children}} \textsc{Card}(w) & \text{if } v \text{ is a union node;} \\ \prod_{w \in v.\text{children}} \textsc{Card}(w) & \text{if } v \text{ is a catenation node.} \end{cases}$$

Similarly, checking whether some set $E \in \mathcal{F}$ is done as follows, for every node v:

1. If v is a leaf, then return true if and only if $E = \{v\}$;
2. If v is a catenation node, then for each child node w, check if $E \cap \textsc{Values}(w)$ belongs to the family induced by the subtree rooted at w (if it is the case, return true, otherwise, return false);
3. If v is a union node, then return true if and only if E belongs to at least one family induced by a child node of v.

To extract a random set with uniform probability, it suffices to follow these steps on each node v:

1. If v is a leaf, then return $\{\textsc{Content}(v)\}$;
2. If v is a catenation node, then return $\bigcup_{w \in v.\text{children}} \textsc{Random}(w)$;
3. If v is a union node, then choose a child node w, with probability equal to $\textsc{Card}(w)/\textsc{Card}(v)$, for each child and then return $\textsc{Random}(w)$.

The complexity of each of these operations is linear, with respect to the size of the UCT:

Theorem A.1. *Let T be some union-cat tree with n nodes. Then the operators* Card, $\in$ *and* Random *have complexity $\mathcal{O}(n)$.*

Proof. Card performs a single traversal of the tree, which is trivially $\mathcal{O}(n)$. The operation $\in$ is also linear, provided that $\textsc{Values}(v)$ is known for every node v of T. Clearly, this information can either be stored directly in the data structure, or preprocessed in linear time, so that the overall complexity is $\mathcal{O}(n)$ as well. Random extraction is also linear, since it performs one traversal of the tree and relies on Card, which is $\mathcal{O}(n)$. □

Modifiers

Similarly to queries, one might define convenient operators that modify the structure of any UCT. We focus on two of them:

(1) Flattening the tree representing the family ($\textsc{Flatten}(v)$, where v is any node);
(2) Binarizing the tree representing the family ($\textsc{Binarize}(v)$, where v is any node);
(3) Extracting a subfamily whose sets exclude some value ($\textsc{Subfamily}(v, K)$, where v is any node and K is any subset of values in S).

The purpose of flattening a tree might be seen as a way of compressing the data when memory space economy is relevant. It is also a way of obtaining a data structure where the types of node alternate between *union* and *catenation* at each level. More precisely, flattening is as follows:

1. If some internal node v has only one child w, then v is deleted and replaced by w;
2. If some internal node v has a child w of the same type, *i.e.* such that $\textsc{Content}(v) = \textsc{Content}(w)$, then one can move all children of w so that they become children of v and then w can be removed.

Flattening is naturally implemented in a bottom-up manner.

Conversely, one might prefer to have a binary tree by allowing at most two children per node, thus simplifying the implementation of all operators. This is done by a simple top-bottom traversal. Indeed, on one hand, if the current node has one child, then it can be removed. On the other hand, if it has more than two children, then it suffices to insert as many intermediate nodes of the same type as needed.

As for all previous operations, extracting a subfamily with allowed/forbidden values can be done without enumerating explicitly the complete family. It suffices to apply those rules, for each node v (assuming that K are the allowed values):

1. If v is a leaf, then delete it if and only if it does not belong to K;
2. If v is a union node, then apply SUBFAMILY(w, K) to each child node w of v. If one of them is an empty tree, remove it. It all of them are empty, then delete v. Otherwise return a subtree whose root content is $\cup$ and whose child nodes are those computed recursively;
3. If v is a catenation node, then apply SUBFAMILY(w, K) to each child node w of v. If any of them is empty, then delete v, otherwise return a subtree whose root content is $*$ and whose child nodes are those computed recursively.

As for queries, all operators are linear:

Theorem A.2. *Let T be any UCT with n nodes. Then the operators* FLATTEN, BINARIZE *and* SUBFAMILY *have complexity $\Theta(n)$.*

Proof. In all three cases, each node is visited once and modifications are done locally in constant time (deletion/assignment of pointers or references), except BINARIZE, which can create many nodes at one step. However, the total number of nodes that are added cannot be more than the number of nodes that are in the original tree. □

ACKNOWLEDGEMENTS

This work was supported by NSERC and FQRNT.

REFERENCES

[1] R.M. Karp, Reducibility among combinatorial problems, in Complexity of Computer Computations, edited by R.E. Miller and J.W. Thatcher. Plenum Press (1972) 85–103.
[2] C. Wang, E.L. Lloyd and M.L. Soffa, Feedback vertex sets and cyclically reducible graphs. *J. ACM* **32** (1985) 296–313.
[3] C.L. Lu and C.Y. Tang, A linear-time algorithm for the weighted feedback vertex problem on interval graphs. *Inform. Process. Lett.* **61** (1997) 107–111.
[4] Y.D Liang. On the feedback vertex set problem in permutation graphs. *Inform. Process. Lett.* **52** (1994) 123–129.
[5] H. Levy and D.W. Low, A contraction algorithm for finding small cycle cutsets. *J. Algorithms* **9** (1988) 470–493.
[6] H.M. Lin and J.Y. Jou, On computing the minimum feedback vertex set of a directed graph by contraction operations. *IEEE Trans. CAD Integrated Circuits Syst.* **19** (2000) 295–307.
[7] S.T. Chakradhar, A. Balakrishnan and V.D. Agrawal, An exact algorithm for selecting partial scan flip-flops, in *Proceedings of the 31st Design Automation Conference* (1994) 81–86.
[8] P. Galinier, E. Lemamou and W. Bouzidi, Applying local search to the feedback vertex set problem. *J. Heuristics* (2013), in press.
[9] P.M. Pardalos, T. Qian and M.G.C. Resende, A greedy randomized adaptive search procedure for feedback vertex set. *J. Combin. Optim.* **2** (1999) 399–412.
[10] R. Ashar and S. Malik, Implicit computation of minimum-cost feedback vertex sets for partial scan and other applications. *Proceedings of the 31st Design Automation Conference* (1994) 77–80.
[11] R. Fleischer, X. Wu and L. Yuan, Experimental Study of FPT Algorithms for the Directed Feedback Vertex Set Problem, Algorithms – ESA 2009, Lecture Notes in Computer Science, edited by A. Fiat and P. Sanders. Springer Berlin Heidelberg (2009) 611–622.
[12] F. Formin, S. Gaspers, A. Pyatkin and I. Razgon, On the minimum feedback vertex set problem: Exact and enumeration algorithms. *Algorithmica* **52** (2008) 293–307.
[13] F. Formin, P. Heggernes, D. Kratsch, C. Papadopoulos and Y. Villanger, Enumerating minimal subset feedback vertex sets, in *Proceedings WADS'11 of the 12th International Conference on Algorithms and Data Structures* (2011) 399–410.

[14] M. Lapointe, A. Blondin Massé, P. Galinier, M. Lord and O. Marcotte, Enumerating minimum feedback vertex sets in directed graphs, in *Bordeaux Graph Workshop*, November 21st–24th, Bordeaux, France (2012) 101–102.

[15] R.E. Bryant, Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Comput.* **C-35** (1986) 677–691.

[16] S.-i. Minato, Zero-suppressed bdds for set manipulation in combinatorial problems, in *Proceedings of the 30th international Design Automation Conference* (1993) 272–277.

[17] D.E. Knuth, Fun with BDDs. Stanford University Lecture. (2008) http://www.youtube.com/watch?v=axUgEAgrSB8.

[18] D.E. Knuth, *Fun with ZDDs.* Stanford University Lecture. (2008) http://www.youtube.com/watch?v=axUgEAgrSB8.

[19] S. Yoon, C. Nardini, L. Benini and G. De Micheli Discovering coherent biclusters from gene expression data using zero-suppressed binary decision diagrams. *IEEE/ACM Trans. Comput. Biol. Bioinformatics* **2** (2005) 339–354.

[20] E. Loekito and J. Bailey, Fast mining of high dimensional expressive contrast patterns using zero-suppressed binary decision diagrams, in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD'06* (2006) 307–316.

[21] O. Schröer and I. Wegener, The theory of zero-suppressed bdds and the number of knight's tours. *Formal Methods Syst. Des.* **13** (1998) 235–253.

[22] B. Becker, M. Behle, F. Eisenbrand and R. Wimmer, Bdds in a branch and cut framework, in *Experimental and Efficient Algorithms, Proceedings of the 4th International Workshop on Efficient and Experimental Algorithms (WEA'05).* Vol. 3503 of *Lecture Notes in Computer Science* (2005) 452–463.

[23] G.F. Luger and W.A. Stubblefield, Artificial Intelligence – Structures and Strategies for Complex Problem Solving, 2nd edn. Benjamin/Cummings (1993).

[24] V. Kumar and L.N. Kanal, Parallel branch-and-bound formulations for and/or tree search. *Pattern Anal. Mach. Intell. IEEE Trans.* **PAMI-6** (1984) 768–778.

[25] R. Kiesel and A. Schidler, A Dynamic MaxSAT-based Approach to Directed Feedback Vertex Sets, arXiv:2211.06109 (2022).

[26] G.S. Smith and R. Walford, The identification of a minimal feedback vertex set of a directed graph. *IEEE Trans. Circuits Syst.* **22** (1975) 9–15.

[27] E. Trischler, Incomplete scan design with an automatic test generation methodology. *Proc. Int. Test Conf.* **1980** 153–162.

[28] R. Bar-Yehuda, D. Geiger, J. Naor and R.M. Roth, Approximation algorithms for the feedback vertex set problem with applications to constraint satisfaction and Bayesian inference. *SIAM J. Comput.* **27** (1998) 942–959.

[29] S. Harnad, The symbol grounding problem. *Physica D: Nonlinear Phenomena* **42** (1990) 335–346.

[30] B. Boguraev and T. Briscoe, editors, *Computational Lexicography for Natural Language Processing.* Longman, London (1989).

[31] B. Boguraev and J. Pustejovsky, editors, *Corpus Processing for Lexical Acquisition.* MIT Press, Cambridge, MA (1996).

[32] O. Picard, M. Lord, A. Blondin Massé, O. Marcotte and S. Harnad, Hidden structure and function in the lexicon, in *10th International Workshop on Natural Language Processing and Cognitive Science – NLPCS* (2013), submitted.

[33] C. Fellbaum, WordNet: An Electronic Lexical Database. MIT Press, Cambridge (1998).

[34] A. Blondin Massé *et al.*, The dictionary game. Université du Québec à Montréal (2022) http://lexis.uqam.ca:8080/dictGame/index.jsp#no-back-button.

[35] P. Erdős and A. Rényi, On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.* **5** (1960) 17–61.

[36] S. Sathe, Python Web Graph Generator (2008) http://pywebgraph.sourceforge.net/.
[37] D. Chakrabarti, Y. Zhan and C. Faloutsos, R-MAT: a recursive model for graph mining. *SIAM Int. Conf. Data Mining* (2004).