Practical Exhaustive Generation of Small Multiway Cuts in Sparse Graphs*

Petr Hliněný and Ondřej Slámečka

Faculty of Informatics, Masaryk University
Botanická 68a, 602 00 Brno, Czech Republic
{hlineny, xslameck}@fi.muni.cz

Abstract. We propose a new algorithm for practically feasible exhaustive generation of small multiway cuts in sparse graphs. The purpose of the algorithm is to support a complete analysis of critical combinations of road disruptions in real-world road networks. Our algorithm elaborates on a simple underlying idea from matroid theory – that a circuit-cocircuit intersection cannot have cardinality one (here cocircuits are the generated cuts). We evaluate the practical performance of the algorithm on real-world road networks, and propose algorithmic improvements based on the technique of generation by a canonical construction path.

1 Introduction

In the area of real-world road network planning and management, one of the vital tasks is to identify potential vulnerabilities of the network in advance. One of the most critical such vulnerabilities is the possibility of a complete break-up of the network as a result of simultaneous disruptions of several roads. In graph theory terms, this corresponds to finding minimal cuts in the network graph (here we consider *edge cuts* by default). However, not every graph cut corresponds to a major disintegration of the whole network; e.g., a cut may just isolate one or several unimportant road intersections (or small villages) and the rest of the network remains fully functional. In fact, one can easily imagine that most small cuts in a real-world network are of the latter (unimportant) kind.

There exist various rather complicated measures of severity of a network break-up, taking into an account the numbers of inhabitants and the economic importance of the nodes which get disconnected from each other, as well as the number of components (cells) into which the network is broken up. See [1] for further references. Notice that, in particular, we have to consider also cuts which separate the network into more than two components (called *multiway cuts*). In a nutshell, research shows that efficient identification of all severe network break-ups does not seem possible without first exhaustively generating all the minimal multiway cuts with small number of edges in the given network.

In our paper we focus right on this task. If we fix integers k, m, then the *task of generating* all the minimal k-way cuts consisting of at most m edges is, in theory, solvable in polynomial time by brute force testing all combinations of at most m

* Research supported by the Czech Science Foundation, project 14-03501S.

edges of the network. However, experiments carried out over networks of around 1000 edges in [1] clearly show that such a brute force approach is practically feasible, even on parallel machines, only for $m \leq 4$. To support the analysis of road network break-ups caused by more than 4 simultaneous disruptions, we proposed a new approach to an exhaustive small cut generation whose simplified heuristic version has already been implemented and successfully used in [1].

The underlying idea of our proposed approach is very natural and simple: suppose we construct a (to-be) cut X iteratively, and there exists a cycle C such that C intersects X in exactly one edge—then another edge of C must belong to the resulting minimal cut extending X. Consequently, the next iteration can choose only from the edges of C instead of the whole network. Since in real-world road networks one can usually find an abundance of short cycles everywhere (where "short" typically means 4 or 5), this approach can dramatically reduce the search space and the runtime of the algorithm.

Here we provide a theoretical background for this new *Circuit-cocircuit algorithm scheme* in terms of matroid theory, which seamlessly integrates generation of minimal k-way cuts for all values of k into the one scheme. We further elaborate the algorithm towards the so-called canonical generation which provides additional important speed-up. We also report on the results of practical computational experiments carried out with different versions of our algorithm.

Related research. Computing a *minimum* two-terminal cut in a graph is a well-known easy application of network flow theory. However, the seemingly similar problem of counting the minimum cuts in a graph is #P-complete [2] (i.e., equivalent to #SAT). Notice that there is a crucial difference between the terms *minimum* and *minimal* cut—where "minimum" means of smallest possible cardinality and "minimal" cuts are those for which no proper subset of them is a cut again (while their cardinality may be arbitrarily high). In our task, if we set m equal to the minimum cut size in the graph (instead of fixing it to a small value beforehand), we hence get that our generation problem is #P-hard in general. Fortunately, experiments with real-world road networks show that their particular case is often computationally much simpler.

Concerning k-terminal cuts for $k > 2$, already computing a minimum three-terminal cut in a graph is an APX-hard problem [3]. Consequently, things do not get any easier with generating multiway cuts. Besides obvious brute force attempts, not much has been published in literature about exhaustive generation of small cuts in graphs. One remarkable exception is the work of Reinelt and Wenger [4], who elaborated on the classical so-called "cactus representation" of Dinitz et al. [5] to provide a practically efficient algorithm for generation of all minimum multiway cuts in a graph. However, the problem with [4] and previous related papers is that they all compute "minimum" cuts, but in our case we have to generate also all the larger minimal cuts (up to a cardinality bound m) in addition to the minimum (in terms of cardinality) ones.

Paper organization. In Section 2 we give a brief introduction to the necessary theoretical concepts. After that we state the abstract Circuit-cocircuit algorithm

for matroids (Alg. 3.2) and illustrate a simple use of it for generating minimal 2-cuts in a graph (Alg. 3.7). The full power of the Circuit-cocircuit meta-algorithm shows up in Section 4 where we apply it to exhaustively generate all minimal k-way cuts in a graph (Alg. 4.5). Section 5 then outlines a further improvement of the algorithm using the so-called canonical generation. Due to space restrictions in the conference paper, some parts have to be skipped from the main paper body, and those are included in the Appendix.

2 Preliminaries

We mostly follow standard terminology of graph theory. The vertex set of a graph G is referred to as $V(G)$ and the edge set as $E(G)$. In the paper we pay a particular attention to the following graph terms.

An *edge-cut* in a graph G is a set of edges $X \subseteq E(G)$ such that $G \setminus X$ (the subgraph of G obtained by deleting the edges X) has more connected components than G has. A *k-way edge-cut* in a graph G is a set of edges $X \subseteq E(G)$ such that $G \setminus X$ has at least k connected components. Note that in connected graphs, an edge-cut coincides with a 2-way edge-cut, while in a disconnected graph this assertion fails (the empty set is then a 2-way edge-cut).

Definition 2.1 (Bond). *We call a* bond *any minimal edge-cut in a graph, and a* k-bond *any minimal k-way edge-cut in a graph (minimality is considered with respect to set inclusion).*

A graph is a *tree* if it is connected but deleting any of its edges disconnects it. In other words, a tree contains no cycles. A graph is a *forest* if each of its connected components is a tree. If G is a graph and $F \subseteq G$ is a tree (forest) such that $V(F) = V(G)$, then F is a *spanning tree (forest)* of G.

It turns out that the most suitable framework for an abstract description of our proposed algorithm is that of matroid theory. We follow Oxley [6] in matroid terminology, and we give a brief introduction (with examples) next.

Definition 2.2 (Matroid). *A* matroid *is a pair $M = (E, \mathcal{B})$ where $E = E(M)$ is the finite ground set of M (elements of M), and $\mathcal{B} \subseteq 2^E$ is a nonempty collection of* bases *of M, no two of which are in an inclusion. Moreover, matroid bases must satisfy the "exchange axiom"; if $B_1, B_2 \in \mathcal{B}$ and $x \in B_1 \setminus B_2$, then there is $y \in B_2 \setminus B_1$ such that $(B_1 \setminus \{x\}) \cup \{y\} \in \mathcal{B}$.*

The following terminology is used in matroid theory. Subsets of bases are called *independent sets*, and the remaining sets are *dependent*. Minimal sets not contained in a basis (i.e., dependent sets) are called *circuits*, and maximal sets not containing any basis are called *hyperplanes*.

Example 2.3. Let $A = \{a_1, \ldots, a_n\}$ be a finite set of vectors. If $\mathcal{B}$ is the set of all maximal independent subsets of A, then $M = (A, \mathcal{B})$ is a matroid, called the *vector matroid* of A. The independent sets of M are precisely the linearly independent subsets of A.

Example 2.4. If G is a connected graph, then its *cycle matroid* on the ground set $E(G)$ is as follows: bases are the (edge sets of the) spanning trees of G, independent sets are the (edge sets of) forests in G, circuits are the usual cycles in G, and hyperplanes are the set complements of bonds in G.

For a matroid $M = (E, \mathcal{B})$, the matroid on the same ground set E and with the (complementary) bases $B^* = \{E \setminus B : B \in \mathcal{B}\}$ is called the *dual matroid of* M and denoted by M^*. The circuits of M^* are called *cocircuits of* M.

Example 2.5. Let G be a planar graph and M the cycle matroid of G. Then M^* is the cycle matroid of the geometric dual of G.

Claim 2.6 (folklore, see [6]). *Let M be a matroid.*

a) If B is a basis of M and $e \in E \setminus B$, then $B \cup \{e\}$ contains precisely one circuit (through e).
b) If H is a hyperplane of M and $e \in E \setminus H$, then $H \cup \{e\}$ contains a basis B of M and $e \in B$.
c) A set $X \subseteq E$ is a cocircuit of M iff $E \setminus X$ is a hyperplane of M.
d) Cocircuits of M are precisely the minimal sets intersecting every basis of M.

Claim 2.7 (cf. Example 2.4). *Let M be the cycle matroid of a graph G. Then the cocircuits of M are precisely the bonds of G.* □

3 The Circuit-Cocircuit Meta-algorithm

In view of Claim 2.7, it is possible to formulate the problem of generating all bonds of a graph as generating all the cocircuits of its cycle matroid. This approach might seem restrictive at the first sight as it does not directly capture generation of k-bonds for $k > 2$, but precisely the opposite is true: we will later show that k-bonds are the cocircuits under a suitably adjusted definition of the cycle matroid of a graph.

It is quite natural to see that a cycle and a bond in a graph cannot intersect in precisely one edge. A generalization of this observation is one of the fundamental claims in matroid theory (note, however, that a matroid circuit and a cocircuit may intersect in 3 or 5, etc, elements...):

Proposition 3.1 (folklore, see [6]). *If C is a circuit and X is a cocircuit in a matroid M, then $|C \cap X| \neq 1$.*

With Proposition 3.1 at hand, we may simply proceed as follows: start with any element of M in X, find a circuit C such that $|C \cap X| = 1$, and then for each element $c \in C \setminus X$ try to add c to X and recurse. The recursion proceeds as long as $E \setminus X$ contains a hyperplane of M, cf. Claim 2.6 c). The full pseudocode is given in Algorithm 3.2.

Theorem 3.3. *Algorithm 3.2 generates all the cocircuits of size $\leq m$ in a matroid M (with repetition – the same cocircuit may be generated several times).*

Algorithm 3.2 Abstract Circuit-Cocircuit Meta-algorithm

Input: Matroid $M = (E, \mathcal{B})$ and an integer $m \in \mathbb{N}$ (a cocircuit size bound)
Output: All cocircuits of M with size $\leq m$

```
 1: B ← an arbitrary basis in 𝓑
 2: for all b ∈ B do
 3:     X ← {b}
 4:     GenCocircuits(X)
 5: end for
 6: procedure GenCocircuits(X)
 7:     if E \ X contains no hyperplane of M or |X| > m then
 8:         return ⊥                                   ▷ this branch fails
 9:     end if
10:     Find any circuit C ⊆ E such that |C ∩ X| = 1
11:     if such C doesn't exist then
12:         output X                                   ▷ X is a cocircuit
13:     else
14:         D ← C \ X
15:         for all c ∈ D do
16:             GenCocircuits(X ∪ {c})
17:         end for
18:     end if
19: end procedure
```

The proof follows rather straightforwardly (though not shortly) from Claim 2.6, but due to space restrictions it is skipped here.

Remark 3.4. Note that Algorithm 3.2 makes some nondeterministic steps – the choices (of B, C) on lines 1, 10 and also the ordering (of D) on line 15. Theorem 3.3 asserts that for any particular implementation of these steps, the algorithm remains correct. We exploit this fact mainly with the choice of C on line 10, where we aim to minimize $|C|$. If we are (mostly) able to choose C "very small", bounded by a constant such as 5 or 6, then we get a dramatic runtime speed-up over the basic brute force approach trying all $\leq m$-elements subsets of E. Indeed, this is the typical case for the cycle matroids of real-world road networks.

Remark 3.5. There is one weakness of Algorithm 3.2 which is common to many iterative/recursive combinatorial generation algorithms—the same object (here a cocircuit or a bond) is generated many times in different orders of its elements. While there is no easy general remedy for this common problem, we will provide a practically working fast resolution in Section 5.

3.1 Generating 2-Bonds in a Graph

To better explain Algorithm 3.2 and its use, we now present a sample implementation for generating all the 2-bonds in a connected graph. The main task of our implementation is to realize line 7—to be able to efficiently test whether $E \setminus X$ contains a hyperplane of M. This is based on the following claim:

Lemma 3.6. *Let G be a connected graph, $M = (E, \mathcal{B})$ its cycle matroid and $Y \subseteq E = E(G)$. The set $E \setminus Y$ contains a hyperplane of M if, and only if, the vertices incident to the edges of Y can be coloured red and blue, such that each edge of Y gets two colours and there exist two disjoint trees T_r and T_b in $G \setminus Y$ such that the tree T_r (T_b) connects all the red (blue, resp.) vertices of Y.*

Proof. ($\Rightarrow$) If $E \setminus Y$ contains a hyperplane of M, then there exists a cocircuit $X \supseteq Y$ by Claim 2.6 c). Since X is a 2-bond in G, $G \setminus X$ has precisely two connected components (as otherwise X would not be minimal). Colouring the ends of Y in one component red and in the other blue finishes the argument.

($\Leftarrow$) Let $R = V(T_r)$ and $B = V(T_b)$ be the vertex sets of the assumed two trees in $G \setminus Y$. Let $U \subseteq V(G)$ be the set reachable from R in $G \setminus B$ and $X \subseteq E(G)$ be the edges having precisely one end in U. Then $Y \subseteq X$ by the definition, and X is a cut in G separating R from B. Moreover, X is minimal, and so X is a 2-bond and $E \setminus X$ is a hyperplane of M which is contained in $E \setminus Y$. □

In regard of Lemma 3.6, we choose the following implementation of the hyperplane test on line 7. During the progress of the algorithm, each edge e chosen to be added to X gets the colours *red* and *blue* at its ends, such that this choice is consistent (wrt. edges already in X) and fulfills the next conditions (Alg. 3.7). This implementation results in the following algorithm:

Algorithm 3.7 (Circuit-Cocircuit algorithm for 2-bonds in a graph). We specify Algorithm 3.2 with the following points:

(1) Let M of Algorithm 3.2 be the cycle matroid of an input graph G.
(2) With respect to implementation of line 7, the first edge added to X on line 3 gets the colours red/blue arbitrarily. Let, subsequently, $V(X) = V_r \cup V_b$ where V_r are the red ends of X and V_b the blue ends. We actively maintain only a *red tree* T_r interconnecting V_r (as expected by Lemma 3.6), while a *blue tree* is implicit – the two trees are not treated symmetrically: see further Algorithm 4.5 for details of building and maintaining T_r.
(3) Instead of a cycle C on line 10, we explicitly look for a (shortest) path $P \subseteq G \setminus X$ such that one end of P is $u_r \in V_r$ and the other end is $u_b \in V_b$. Note that, for any $f \in X$ with one end u_b, there is a cycle C formed by P, f and the unique path in T_r from u_r to f such that $C \cap X = \{f\}$, as expected by Algorithm 3.2, but we do not explicitly invoke C in our implementation.
(4) On line 14, we set $D \leftarrow E(P)$ (which is a subset of the implicit circuit C).

Proposition 3.8. *Algorithm 3.7 generates all the 2-bonds of size $\leq m$ in a connected graph G (with possible repetition).*

The proof nearly immediately follows from Theorem 3.3 and Claim 2.7, but there is one catch: the set D computed on line 14 may be a strict subset of $C \setminus X$ expected in Algorithm 3.2. We can show that for every 2-bond X_0 of G, at least one of the computation paths leading to X_0 is not affected by this deficiency. Again, due to space restrictions a full proof is skipped here.

4 k-Way Cycle Matroid and Generating k-Bonds

As mentioned before, Algorithm 3.2 can be used for generating k-bonds of a graph for any $k \geq 2$. We just have to extend the definition of a cycle matroid so that cocircuits within the new definition are precisely the k-bonds.

Definition 4.1 (k-way cycle matroid). *Let G be a graph of less than $k \geq 2$ components. The* k-way cycle matroid *of G is a matroid on the ground set $E(G)$, such that its bases are the edge sets of the spanning forests of G consisting of $k-1$ trees. The bases, circuits, cocircuits, hyperplanes of the k-way cycle matroid are also called the* k-way bases, circuits, cocircuits, hyperplanes *of G.*

From this definition one can easily conclude some basic properties.

Claim 4.2. *Let G be a graph consisting of less than $k \geq 2$ components.*
The k-way cocircuits of G are precisely the k-bonds of G.
The k-way circuits of G are of two types, type-C *and* type-F*:*

- *type-C circuits are the graph cycles in G.*
- *type-F circuits, also called* spanning circuits*, for $k \geq 3$, are the spanning forests of G that are formed by $k-2$ trees.*

Now, by Theorem 3.3, every implementation of Algorithm 3.2 for the k-way cycle matroid of a graph G generates all the k-bonds of G. Although, working with the circuits of Claim 4.2 is somehow intricate. We thus restrict our attention to a special variant of Algorithm 3.2 which has several advantages.

- First, this variant is compatible with and extends Algorithm 3.7.
- Second, it coincides with the natural naive approach to generating k-bonds: find a 2-cut, choose one of its sides and recursively find a 2-cut of this side, and so on until k parts are generated. In other words, we also prove that such a naive approach is indeed correct (if properly implemented).

This special variant is defined as follows:

Definition 4.3 (Stepwise Circuit-Cocircuit implementation scheme). *We call an implementation of Algorithm 3.2 stepwise if, for every set $X = X_0$, $|X_0| = l$, generated by the algorithm the following holds:*

1. *X_0 is an ordered sequence $(c_1, c_2, \ldots, c_l)$, where c_i has been added to X_0 at the level $i-1$ of recursion, and*
2. *there exists a mapping $s : \{1, 2, \ldots, k\} \to \{0, 1, \ldots, l\}$ such that $s(1) = 0$, $s(k) = l$ and, for each $j \in \{2, \ldots, k-1\}$, the set $\{c_1, \ldots, c_{s(j)}\} \subsetneq X_0$ forms a j-bond in G.*

For j, $1 \leq j < k$, we call the j-th stage of the algorithm the steps the algorithm does at the levels $s(j), s(j)+1, \ldots, s(j+1)-1$ of recursion. In other words, the algorithm in its j-th stage selects the elements $c_{s(j)+1}, \ldots, c_{s(j+1)}$.

Before proceeding into details of the stepwise implementations, we first show that the definition indeed makes sense. A proof is again left for the Appendix.

Algorithm 4.5 One stage of a stepwise implementation

Input: A conn. graph G, param. $j, k, m \in \mathbb{N}, j < k, m \geq 1$, and a j-bond $Y_1 \subseteq E(G)$
Output: A collection of $(j+1)$-bonds such that for each k-bond Y, $Y_1 \subseteq Y \subseteq E(G)$, $|Y| \leq m$, some subset of Y is among the generated $(j+1)$-bonds

if $j = 1$ **then** ▷ $Y_1 = \emptyset$: select a k-way basis
  $F \leftarrow$ an arb. spanning forest of $k-1$ trees
else ▷ $Y_1 \neq \emptyset$: select a type-F circuit
  $F \leftarrow$ an arb. spanning forest of $k-2$ trees and $|F \cap Y_1| = 1$
end if
for all $d = \{u, v\} \in F \setminus Y_1$ **do**
  GenStage$(j, Y_1, X = \{d\}, V_r = \{u\}, V_b = \{v\}, T_r = \{u\})$
end for.
procedure GenStage(j, Y, X, V_r, V_b, T_r)
  Let $G_1 \subseteq G$ be the component of $G \setminus Y$ containing X
  if $|Y \cup X| > m - k + j + 1$ **then**
    return $\perp$ ▷ no way to get a k-bond of size $\leq m$
  end if
  if there does not exist a connected subgraph
    $T_b \subseteq (G_1 \setminus V(T_r)) \setminus X$ such that $V_b \subsetneq V(T_b)$ **then**
    return $\perp$ ▷ the "no hyperplane" condition
  end if
  $P \leftarrow$ a minimal path in G_1 from $V(T_r)$ to V_b
  if such P does not exist **then**
    output $Y \cup X$ ▷ $Y \cup X$ is a $j+1$-bond
  else
    for all $c \in P$ **do** ▷ add c to X and update T_r
      Let u be the vertex in $c = \{u, v\}$ which is closer to T_r
      Let P_u be the component of $P - c$ which contains u
      GenStage$(j, Y, X \cup \{c\}, V_r \cup \{u\}, V_b \cup \{v\}, T_r \cup P_u)$
    end for
  end if
end procedure

Proposition 4.4. *A stepwise implementation of Algorithm 3.2 is possible. Precisely, for every $k \geq 2$ there exists a stepwise implementation generating all the k-bonds in a given connected graph.*

One can, moreover, easily show (with a formal proof in the Appendix) that a "transition" from the j-th stage to $(j+1)$-st one in a stepwise implementation really means to construct a 2-bond in one of the parts of the previous j-bond. A desired consequence is that we can decompose the stepwise algorithm computation into these stages such that, in each stage, we simply invoke Algorithm 3.7.

These findings directly lead to a stepwise algorithm whose one stage is shown in pseudocode in Algorithm 4.5. Validity of this new algorithm then, in turn, follows immediately from the following statement describing its one stage output.

Theorem 4.6. *Let G be a graph, j, k, m integers such that $j < k, m \geq 1$ and $Y_1 \subseteq E(G)$ a j-bond in G. Algorithm 4.5 generates a set $\mathcal{S}$ of $(j+1)$-bonds such*

that for each k-bond Y, $Y_1 \subseteq Y \subseteq E(G)$, $|Y| \leq m$, some subset of Y is among the generated $(j+1)$-bonds in $\mathcal{S}$.

The proof follows from Proposition 3.8 via the previous claims.

5 Canonical Generation

We now return to Remark 3.5; addressing the problem that one bond X_0 is typically generated many times by our circuit-cocircuit algorithm, each time with a different permutation of its elements. While such a repetition can be easily removed by a post-processing, it costs running time. Ideally, our algorithm should for each X_0 "guess" one computation path leading to X_0 and immediately dismiss all the other attempts, as early as possible in the generation process. But how can this be done? This is not at all an easy question since, for example, we have to ensure that (nearly) every two bonds X_0, X_1 sharing many elements also share a long prefix of the guessed computation path, and so on. Most importantly, the guessed computation path of X_0 must be compatible with Algorithm 3.2, i.e., each next element of X_0 on the path must be from the circuit C on line 10 of the algorithm, which is not a priori clear how to achieve.

There exists a sophisticated technique of *generation by a canonical construction path* by McKay [7], outlined next. Since we cannot fit the details of this technique and its application to our case into the restricted conference paper, we stay on a very informal level in the main text body and leave more details for the Appendix.

In our case, a computation path of a bond X_0 in G is simply encoded by a permutation $\vec{X}_0$ of the elements of X_0. The definition of a canonical form $\vec{X}_0$ of X_0 respects the stepwise generation framework as follows:

I) The permutation $\vec{X}_0$ refines the order of the stages in some stepwise computation path leading to X_0 (cf. Definition 4.3).

II) There is an arbitrary bijection $\iota : E(G) \to \{1, \ldots, |E(G)|\}$ indexing the edges of G. The starting edges of the stages in $\vec{X}_0$ are each ι-minimal within its stage, and they are altogether strictly ordered by ι (first-to-last stage).

III) Within each stage, the corresponding sub-permutation of $\vec{X}_0$ (except the starting edge) is determined by the shortest path P selection and the red/blue tree mechanism of Alg. 3.7; see also the appropriate parts of Alg. 4.5.
Two additional details are important for a successful implementation of this point. First, the red and blue sides of the hyperplane test are uniquely decided with the first edge of the stage based on a fixed vertex indexing of G. Second, the unit lengths of edges of G are slightly perturbed to achieve uniqueness of the shortest path P selection.

Concerning the canonical implementation of bond generation, point I) and parts of III) of the scheme are already embedded in Algorithm 4.5, and the rest of III) is rather straightforward to add. The biggest runtime savings come from implementing point II). At the beginning of each stage, the starting edge

is selected from an ι-minimal basis (or type-F circuit) among its edges of ι-value higher than that of the previous stage. Then, the remaining edges of this stage are restricted only to those candidates of higher ι-value than the starting edge.

Although the presented scheme is not *truly canonical* since one k-bond X_0 can still be generated in more than one canonical form, it is implementation-wise very easy and provides great speed-up for the algorithm; see the next section.

6 Evaluation

In this section we present the outcomes of measurements performed with implementations of our algorithms on the road networks of the regions of Czech republic: the Zlín Region (723 vertices, 974 edges) and the Olomouc Region (1454 vertices, 2066 edges). Measurements using the larger road network of Central Bohemian Region (4114 vertices, 5964 edges) gave similar results.

We have implemented the core algorithm of Section 4 which generates same bonds multiple times (i.e., without canonical generation), and the improved algorithm of canonical generation from Section 5. For the running time evaluation we used a computer with 16 GB RAM and the Intel Core i7-3770 CPU @ 3.40GHz. The source code was compiled with `gcc` 4.8.2.

The measurement results are summarized in the tables below. To start, Tables 1 and 2 show the overall runtimes where the entries marked '-' did not finish before the time limit. Tables 3 and 4 show the improvement, in terms of runtime, of the canonical generation algorithm from Section 5 over the ordinary algorithm from Section 4. The improvement achieved by preventing repeated generation of the same bonds is up to 15× in the experiments. This runtime improvement well correlates with the average multiplicity of repeatedly generated bonds by the ordinary algorithm in Table 5. Although the approach of Section 5 does not completely prevent repeated generation of the same bonds, the percentage of "leftover" multiply generated bonds is truly marginal and hence negligible for practical computations; see Table 6.

To demonstrate superiority of the circuit-cocircuit algorithm over the brute-force approach trying all m-tuples of edges for k-bonds, we include Table 7. The table summarizes the distribution of lengths of the path P (Algorithm 4.5, line 18), which represent the degrees of branching of the circuit-cocircuit algorithm inside each stage. While the brute-force approach would result in a quite bad running time of order $\mathcal{O}\big(|E(G)|^m/m!\big)$, the nature of Algorithm 4.5 together with the experimental data in Table 7 suggest that the running time can be, roughly,

$$\mathcal{O}\big(|V(G)|^k \cdot \beta^{m-k}\big), \tag{1}$$

where the auxiliary constant β stands for a typical bound on the length of the path P and can be guessed as $\beta \approx 5$.

Comparing to Tables 1 and 2, one can see quite a good match in the runtime dependence on $(m-k)$ in (1), while the dependence on k seems overshadowed by other aspects of the algorithm for the small experimental values of k, m.

Table 1. Running time of an implementation of the canonical generation in seconds. Zlín Region

k \ m	2	3	4	5	6	7	8
2	0.0	0.1	1	2	9	42	210
3	0.6	2.8	13	53	223	986	4604
4		29.5	198	1018	4771	21269	-
5			1156	9885	56847	-	-

Table 2. Running time of an implementation of the canonical generation in seconds. Olomouc Region

k \ m	2	3	4	5	6	7	8
2	0.1	0.3	1	5	16	69	305
3	3.0	10.4	61	235	921	3482	13342
4		158.3	781	6008	-	-	-
5			6205	43242	-	-	-

Table 3. Ratio of running times without and with canonical generation. Zlín (top) and Olomouc (bottom) Region

k \ m	2	3	4	5	6	7
2	1.00	2.00	2.45	3.60	4.3	1.34
3	3.28	3.51	6.05	8.83	12.11	14.90
4		8.40	11.32	15.73	-	-

k \ m	2	3	4	5	6	7
2	1.14	1.93	1.91	2.18	4.30	5.06
3	1.97	3.47	4.49	6.69	8.31	9.41
4		5.96	10.16	15.53	-	-

Table 4. Ratio of the numbers of generated bonds without and with canonical generation. Zlín (top) and Olomouc (bottom) Region

k \ m	2	3	4	5	6	7
2	1.43	1.97	2.67	3.52	4.42	1.15
3	2.00	3.08	4.27	5.99	7.94	10.09
4		6.00	9.50	13.38	-	-

k \ m	2	3	4	5	6	7
2	1.32	1.93	2.53	3.26	4.04	4.78
3	2.00	2.84	4.03	5.50	7.41	9.59
4		6.00	8.79	12.37	-	-

Table 5. The average multiplicity of (repeatedly) generated bonds in the non-canonical generation algorithm. Zlín (top) and Olomouc (bottom)

k \ m	3	4	5	6	7
3	3.112	4.309	6.092	8.152	10.465
4		9.706	13.642	-	-

k \ m	3	4	5	6	7
3	2.840	4.038	5.536	7.491	9.739
4		8.817	12.432	-	-

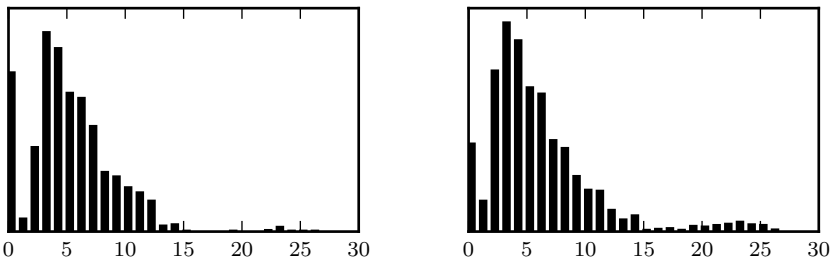
Table 6. The percentage of repeatedly generated bonds in the canonical generation algorithm. Zlín (top) and Olomouc (bottom) Region

k \ m	3	4	5	6	7
3	0.972%	0.814%	1.618%	2.664%	3.715%
4		2.177%	1.950%	3.462%	-

k \ m	3	4	5	6	7
3	0.156%	0.253%	0.649%	1.049%	1.568%
4		0.352%	0.541%	-	-

Lastly, we would like to comment on a possible *parallelization* of the new algorithm. This is actually very easy: each time when adding a new edge to the bond X, one may simply run all the computation branches in parallel, without

Table 7. The distribution of lengths of the path P from Algorithm 4.5. Results of the computation on Zlín Region, $k = 3, m = 6$; on the left showing the second level of recursion of `GenStage`, on the right the fifth level (the algorithm occasionally uses even longer paths in later `GenStage` calls).



any need for synchronization or communication between the branches. Furthermore, especially in the canonical generation case, no costly final post-processing of the generated bonds is needed.

7 Conclusion

We have presented a new "Circuit-Cocircuit" algorithm for exhaustive generation of cocircuits in a matroid, with a practical application to finding all the minimal k-way cuts in a graph. We have further elaborated on the algorithm to achieve an almost canonical generation process, which significantly speeds-up the algorithm by early removal of duplicate computation branches. This theoretical work has been complemented by an implementation and extensive practical evaluations of the algorithm on real-world data. The source code of our implementation is available at `https://github.com/OndrejSlamecka/mincuts`.

In a conclusion, our implementation solves the problem of finding all small multiway cuts correctly as well as quickly (given the high theoretical complexity of the problem) and with very low memory usage, thus demonstrating the feasibility of this algorithm for practical computations, e.g., in road network planning and management. In particular, the algorithm performs significantly better than the brute-force algorithm on real-world networks. Our algorithm will help to improve the results of [1] (where only a simplified heuristic version of the Circuit-Cocircuit algorithm, without canonicity, was implemented).

Our main suggestions for future work are as follows. The main theoretical question is whether there exists a method of truly canonical generation which does not require costly explicit isomorphism checks. On the implementation side, profiling shows that the algorithm spends most of time in the `shortestPath` procedure—finding a good CPU-aware implementation [8] of this procedure would benefit the running time.

References

1. Bíl, M., Vodák, R., Hliněný, P., Svoboda, T., Rebok, T.: "A Novel Method for Rapid Identification of Road Links Causing Network Break-Up". Submitted to EJOR 2015
2. Provan, J.S., Ball, M.O.: The Complexity of Counting Cuts and of Computing the Probability that a Graph is Connected. SIAM J. Comput. 12(4), 777–788 (1983)
3. Dahlhaus, E., Johnson, D.S., Papadimitriou, C.H., Seymour, P.D., Yannakakis, M.: The Complexity of Multiterminal Cuts. SIAM J. Comput. 23(4), 864–894 (1994)
4. Reinelt, G., Wenger, K.M.: Generating partitions of a graph into a fixed number of minimum weight cuts. Discrete Optimization 7(1-2), 1–12 (2010)
5. Dinic, E., Karzanov, A., Lomonosov, M.: A structure of the system of all minimum cuts of a graph. In: Studies in Discrete Optimization, A.A. Fridman ed. Pp. 290–306. Nauka, Moscow (in Russian) (1976)
6. Oxley, J.: *Matroid Theory*. Oxford University Press (2006)
7. McKay, B.D.: Isomorph-free Exhaustive Generation. J. Algorithms 26(2), 306–324 (1998)
8. Chhugani, J., Satish, N., Kim, C., Sewall, J., Dubey, P.: Fast and Efficient Graph Traversal Algorithm for CPUs: Maximizing Single-Node Efficiency. In: 2012 IEEE 26th International Parallel and Distributed Processing Symposium, IEEE (2012)

Appendix

A Supplements for Section 3

Proof of Theorem 3.3

Proof. First, we show that any set X_0 returned by the algorithm is a cocircuit and $|X_0| \leq m$. By the condition on line 7, the set $E \setminus X_0$ contains a hyperplane H of M and $|X_0| \leq m$. If $H = E \setminus X_0$ then we are done by Claim 2.6 c). Now suppose $H \neq E \setminus X_0$ and choose $e \in E \setminus (X_0 \cup H)$. According to Claim 2.6 b) there is a basis $B_0 \subseteq H \cup \{e\}$. Choose $a \in X_0$ and let C_0 be the circuit contained in $B_0 \cup \{a\}$ due to Claim 2.6 a). Then $C_0 \cap X_0 = \{a\}$ contradicts the condition on line 11 and thus H must be equal to $E \setminus X_0$.

Second, we show that any cocircuit X_1, $|X_1| \leq m$, is returned. Let $Z_1 \subseteq X_1$ be the largest subset of X_1 such that GenCocircuits(Z_1) has been called. By Claim 2.6 d), $X_1 \cap B$ is nonempty for the basis B selected on line 1 and thus Z_1 is defined and non-empty. If $Z_1 = X_1$ then X_1 will be returned since it is a cocircuit and no circuit C is found on line 10. Suppose $Z_1 \neq X_1$ and choose $d \in X_1 \setminus Z_1$. By Claim 2.6 b) there is a basis $B_1 \subseteq H_1 \cup \{d\}$ (where $H_1 := E \setminus X_1$ is a hyperplane of M). For any $b \in Z_1$, let C_1 be the circuit contained in $B_1 \cup \{b\}$ by Claim 2.6 a). Then $C_1 \cap Z_1 = \{b\}$, and so there exists a circuit C_2 to be found by the algorithm on line 10 whilst $X = Z_1$ (note that it may be $C_2 \neq C_1$). Since $|C_2 \cap X_1| \neq 1$, there exists $d' \in C_2 \cap X_1$ such that $d' \notin Z_1$, and GenCocircuits($Z_1 \cup \{d'\}$) is eventually called, contradicting the maximality of Z_1. □

Proof of Proposition 3.8

Proof. For the purpose of this proof, we imagine an *extended* version of Algorithm 3.7, in which we set $D \leftarrow E(C) \setminus X$ on line 14 where C any one of the cycles implicitly associated with the path P found by Algorithm 3.7 (3). Recall that $P = C \setminus (T_r \cup X)$. This extended algorithm is an instance of Algorithm 3.2 and so it generates all the 2-bonds of size $\leq m$ by Theorem 3.3.

Let $\vec{X_0} = (c_1, c_2, \ldots, c_l)$ be a permutation of a generated 2-bond $X_0 = X$ such that the edge c_i has been added to X at the level $(i-1)$ of recursion. Let T_r^i be the value of the red-tree variable T_r in the *extended algorithm*, at the moment when c_i has been added to X. It is now easy to see that $\vec{X_0}$ is generated by Algorithm 3.7 if, and only if, in the extended algorithm it hols $c_i \notin E(T_r^i)$ for all $i = 2, \ldots, l$. For every 2-bond X_0 generated by the extended algorithm, such a permutation $\vec{X_0}$ indeed does exist—it results from the computation branch which, from the path P at each recursion level selects the edge of $E(P) \cap (X_0 \setminus X)$ closest to the end in V_r. Therefore, X_0 is generated by Algorithm 3.7. □

B Supplements for Section 4

Proof of Proposition 4.4. The proof follows, with respect to Theorem 3.3, immediately from the following lemma (whose rich technical proof will be also used in subsequent claims):

Lemma B.1. *Let G be a connected graph and $X \subseteq E(G)$ a k-bond, $k \geq 2$. For any i-bond $Z \subset X$ of G, $i < k$, there exists an $(i+1)$-bond Y of G such that $X \supseteq Y \supset Z$.*

Proof. Let $A = X \setminus Z$, $e \in A$ and $G'' = G \setminus X$. Then $G'' \cup \{e\}$ has $k-1$ components, since X is a minimal k-way cut. Choose maximal $A^1 \subseteq A$ with $e \in A^1$, such that $G'' \cup A^1$ has $k-1$ components. In the same manner inductively construct $A^2, \ldots, A^{k-i-1}$, where A^{k-i-1} is a maximal subset of A such that $G'' \cup A^{k-i-1}$ has i components. This constructions yields $Z = X \setminus A^{k-i-1}$.

Now consider $G' = G \setminus Z$, a graph of i components. Let $B := A^{k-i-1} \setminus A^{k-i-2}$ and by G'_0 denote the component of G' containing B (note that $B \cap Z = \emptyset$). It remains to prove that $Y := Z \cup B$ is a desired $(i+1)$-bond. Indeed, $G \setminus Y$ has $i+1$ components by the definition of A^{k-i-2}, and by maximality of A^{k-i-2}, for any $f \in B$, the graph $G \setminus (Y \setminus \{f\})$ has the same i components as G' does. For any $f \in Z$, the graph $G' \cup \{f\} = G \setminus (Z \setminus \{f\})$ has $i-1$ components since Z is an i-bond, and so $G \setminus (Y \setminus \{f\})$ has again at most i components. □

Properties of stepwise implementation scheme. We will also use the following two technical lemmas which follow by similar arguments as Lemma B.1.

Lemma B.2. *Let G be a connected graph and $X \subseteq E(G)$ any k-bond, and for j, $1 \leq j \leq k$, denote by X^j the set $\{c_1, \ldots, c_{s(j)}\}$ (w.r.t. Definition 4.3) and by G^j the graph $G \setminus X^j$. For j, $1 \leq j < k$, there is a connected component G^j_0 of G^j such that $Z^j = X^{j+1} \setminus X^j \subseteq E(G^j_0)$ and Z^j is a 2-bond in G^j_0.*

Proof. The set B from the proof of Lemma B.1 is a 2-bond in G^j_0. □

Lemma B.3. *For each stage j, $1 < j < k$, of a stepwise implementation of Algorithm 3.2 for k-bonds, w.r.t. Definition 4.3, the following holds:*

a) the set C chosen on line 10 at the level $s(j)$ of recursion has to be a type-F (spanning) circuit,
b) at the levels i, $s(j)+1 \leq i < s(j+1)$ of recursion, the set C on line 10 can always be chosen as a type-C circuit.

Proof (B.3 a). In each stage j, the edge the algorithm selects is $c_{s(j)+1}$ from C which is a k-way circuit in G. Since the set $\{c_1, \ldots, c_{s(j)}\}$ forms a j-bond in G, the value of C at the level $s(j)$ cannot be a type-C circuit and thus it has to be a type-F circuit by Claim 4.2.

(B.3 b). Let X_0 be any k-bond (to be generated by the algorithm). Then the associated set Z^j by Lemma B.2 for $X = X_0$ is a 2-bond in a component G^j_0 of G^j. At the level i it holds that $Z^j \cap X \neq \emptyset \neq Z^j \setminus X$. Hence there exists a cycle in G^j_0 intersecting X in precisely one edge, and this cycle (its edges) can be chosen as C in the algorithm. □

C Supplements for Section 5

C.1 Generation by a Canonical Construction Path

Imagine that we would like to generate some *labeled* objects $\vec{X}$, and there is an equivalence relation $\simeq$ on the set of these objects. We can then apply *generation by a canonical construction path* [7] to generate precisely one object from each equivalence class of $\simeq$.

For each equivalence class $\mathcal{C}$ of $\simeq$ we (carefully) choose in advance a unique *canonical representative*. We denote the canonical representative $\mathrm{can}(\vec{X})$ for some $\vec{X} \in \mathcal{C}$.

At each iteration of the generating procedure we start with a partial solution $\vec{Y}$ and we *check* whether this augmentation of $\vec{Y}$ is a prefix of $\mathrm{can}(\vec{X})$, where $\vec{X}$ is some complete solution made from $\vec{Y}$. Note that this check is allowed to have false positives (that is allow augmentation which does not lead to the canonical solution) but no false negatives.

At the end of the generating procedure there is a complete solution $\vec{X} = \mathrm{can}(\vec{X})$ and the algorithm outputs it.

The definition of can() has to be compatible with the original generating routine so it is feasible to test X against this definition often in the algorithm and so it does not miss any solution. On the other hand one wants to choose can() such that it prunes the search tree of the generating procedure[1] to the utmost possible measure.

C.2 Canonical Form of k-bonds

We apply the idea of generation by a canonical construction path to Algorithm 4.5. Firstly, we will define a unique ordering of the elements in each stage of the stepwise generation procedure (for each generated bond) – Def. C.3. Secondly, we will define a canonical ordering of a stepwise partition (Def. C.1) based on an arbitrary indexing of the edges, which will then determine a canonical form of a bond – Def. C.5.

Before continuing we recall the notation of Definition 4.3: Let G be a connected graph and $X \subseteq E(G)$ a k-bond of size l. Consider a mapping $s : \{1, 2, \ldots, k\} \to \{0, 1, \ldots, l\}$ such that $s(1) = 0$, $s(k) = l$. For each $j \in \{1, \ldots, k\}$, denote by Z^j the set $\{c_{s(j)+1}, \ldots, c_{s(j+1)}\}$, by G^j the set $G \setminus \{c_1, \ldots, c_{s(j)}\}$ and by G_0^j the connected component of G^j such that $Z^j \subseteq E(G_0^j)$.

Definition C.1 (Stepwise partition). *Let Z be a k-bond. An ordered partition $(Z^1, \ldots, Z^{k-1})$ of Z is called a stepwise partition if there exists a stepwise implementation (confer Definition 4.3) of Algorithm 3.2, such that the elements of Z^j are added to X in its j-th stage.*

[1] A tree whose root marks the start of the algorithm, inner nodes contain partial solutions and leaves mark the ends of computation branches (the output structures).

There is a one to one correspondence between a stepwise partition and the index mapping s shown above. Hence, to simplify the text, we will freely use one of these two to refer to the other.

Remark C.2. Observe that not every ordering of $\{Z^1, \ldots, Z^{k-1}\}$ is a valid stepwise partition by Definition C.1. On the other hand, by Proposition 4.4, there exists a valid stepwise partition for each k-bond.

Using Lemma B.2 and the previous definitions, we give:

Definition C.3 (Canonical form of a stepwise partition). *Let G be a connected graph, $X \subseteq E(G)$ a k-bond with $|X| = l$, $\mathcal{Z}$ a stepwise partition of X and s the corresponding index mapping dependent on $\mathcal{Z}$. Furthermore let $\iota : E(G) \to \{1, \ldots, |E(G)|\}$, $\nu : V(G) \to \{1, \ldots, |V(G)|\}$ be arbitrary bijections (indexing the edges and vertices) and let $\lambda : E(G) \to \mathbb{N}$ be an arbitrary mapping. The canonical form of $\mathcal{Z}$, $can(\mathcal{Z}) := \vec{X}$, is a unique permutation $\vec{X} = (c_1, c_2, \ldots, c_l)$ of X which refines the ordered partition $\mathcal{Z}$ and satisfies the following, for each $j \in \{1, \ldots, k-1\}$:*

a) Let u, v be the ends of $c_{s(j)+1} \in Z^j$. The vertices of edges in Z_j can be bipartitioned as $V(Z^j) = V_r^j \cup V_b^j$, such that each edge $uv \in Z^j$ has one end in V_b^j and the other in V_r^j, and it holds $u \in V_r^j$ if and only if $\nu(u) < \nu(v)$.

b) The edge $c_{s(j)+1}$ satisfies $\iota(c_{s(j)+1}) = \min\{\iota(c_i) : s(j) < i \leq s(k)\}$.

c) For each $i \in \{s(j)+2, \ldots, s(j+1)\}$, let $Y_{j,i} := \{c_{s(j)+1}, \ldots, c_i\} \subseteq Z^j$. Denote by $\mathcal{P}_{j,i}$ the set of all paths $P \subseteq G_0^j \setminus Y_{j,i}$ having one end in $V(Y_{j,i}) \cap V_r^j$ and the other in $V(Y_{j,i}) \cap V_b^j$. Let $P_{j,i}$ be the unique path in $\mathcal{P}_{j,i}$ lexicographically minimizing the triplet $\langle |E(P)|, \mathrm{len}_\lambda(P), \vec{\iota}(P) \rangle$, where $\mathrm{len}_\lambda(P) = \sum_{e \in E(P)} \lambda(e)$ and $\vec{\iota}(P) = (\iota(e_1), \iota(e_2), \ldots, \iota(e_{|E(P)|}))$.
The edge c_i is the edge of $E(P) \cap Z^j$ closest to the end of P in V_r^j.

To simplify our notation we introduce functions μ and τ as follows.

Definition C.4. *Let ι be an arbitrary bijection (confer Definition C.3), Z be a k-bond with stepwise partition $(Z^1, \ldots, Z^{k-1})$, s be the associated index mapping and $1 \leq j \leq k-1$. We set*

$$\mu(Z^j) := \min\{\iota(c) : c \in Z^j\}.$$

For Y being a prefix of Z and h the largest index such that $c_{s(h)+1}$ belongs to Y;

$$\tau(Y) := \begin{cases} -\infty & \text{if } Y = \emptyset \\ c_{s(h)+1} & \text{otherwise.} \end{cases}$$

Now we are going to define a canonical form of a bond, which deserves a deeper explanation in advance. Let X be an arbitrary k-bond. While for each fixed stepwise partition $\mathcal{Z}$ (which is ordered) of X, we get a unique associated

canonical form $can(\mathcal{Z}) = \vec{X}$ by Definition C.3, different stepwise partitions of X may exist and they obviously give different forms $\vec{X}$.

In this situation, one should perhaps try to define a unique canonical stepwise partition $\mathcal{Z}$ of X (and then resort to Def. C.3). However, this would be rather complicated and it appears easier and faster in practical applications, to slightly relax the requirement of uniqueness of $\mathcal{Z}$ and to have a "canonical form" definition for a k-bond which is not strictly unique. Such a definition follows, and we also refer to further Example C.7.

Definition C.5 (Canonical form of a bond). *Let G be a connected graph and $X \subseteq E(G)$ be a k-bond. A permutation $\vec{X}$ is a* canonical form *of X if there exists a stepwise partition $\mathcal{Z} = (Z^1, \ldots, Z^{k-1})$ of X, such that $\mu(Z^1) < \mu(Z^2) < \ldots < \mu(Z^{k-1})$ and $\vec{X} = can(Z)$.*

Lemma C.6. *Let G be a connected graph and $X \subseteq E(G)$ be a k-bond. For any ι, ν, λ as in Definition C.3, there exists a canonical form $\vec{X}$ of X.*

Proof. We proceed by induction on the number of stages. First we observe that the edge $c_1 = c_{s(1)+1} \in \vec{X}$ satisfying $\iota(c_{s(1)+1}) = min\{\iota(c_i) : s(1) < i \leq s(k)\}$, lies in the ι-minimum k-way basis, that is in the ι-minimum spanning forest $F \subseteq G$ consisting of $k-1$ trees.

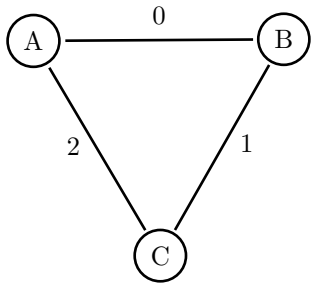
For a contradiction suppose that $c_{s(1)+1}$ does not lie in F. Because $\vec{X}$ is a cocircuit of G, it intersects every basis of G including the ι-minimal one F, by Proposition 2.6. Say there is an element e in the intersection of $\vec{X}$ with F. Then $F \cup \{c_{s(1)+1}\}$ contains a circuit through e. The set $F' = F \cup \{c_{s(1)+1}\} \setminus \{e\}$ is a basis such that $\sum_{e \in F'} \iota(e) < \sum_{e \in F} \iota(e)$, contradicting minimality.

By Lemma B.3 the remaining edges of the first stage can be selected from type-C circuits. By the same argument as in Algorithm 3.7(3), these type-C circuits are in a correspondence with the paths of $\mathcal{P}_{j,i}$ as in Definition C.3 c). Hence the edges of Z^1 can be ordered to satisfy this point of the definition.

For the induction step consider the j-th stage and a j-bond $Y \subseteq \vec{X}$. We want to show that the edge $c_{s(j)+1} \in \vec{X}$ lies in the ι-minimum corresponding type-F circuit F'. Since $|F' \cap Y| = 1$, it suffices to show that the edge $c_{s(j)+1}$ lies in the ι-minimum spanning forest $F = F' \setminus Y$ on $k-1$ trees (incidentally, this is again a k-way basis) such that and $Y \cap S = \emptyset$. To observe this we can use a very similar argument as in the base case.

The remaining edges of the j-th stage are ordered in the same way as in the first stage. This finishes the induction step. □

Example C.7. Let $X = \{0, 1, 2\}$ be edges as in the figure. The canonical form of X is not unique. Let $k = 3$, $m = 3$. In stage 1 the algorithm generates bonds $(0, 1)$ and $(0, 2)$. In stage 2 the bond $(0, 1)$ can be extended to $(0, 1, 2)$ and the bond $(0, 2)$ can be extended to $(0, 2, 1)$, which are both in a canonical form.



C.3 Canonical Generation within Algorithm 4.5

Definitions of a canonical bond can be incorporated into Algorithm 4.5 as follows (using also some ideas from the proof of Lemma C.6).

- At the beginning of each stage the edge c which is being added to X has to satisfy $\iota(c) > \tau(Y)$. Note that as a result the spanning minimum forest F has to be recomputed at the beginning of each stage
- Within each stage, after the first edge has been chosen, then all the edges c' being added to X have to satisfy $\iota(c') > \tau(Y)$.
- The path P is chosen on line 18 such that it satisfies the condition of Definition C.3 c).
- The choice of edge $c \in E(P) \cap Z^j$, such that it is the edge in $E(P) \cap Z^j$ closest to V_r (Def. C.3 c)), is already "embedded" in the algorithm by iterating through the edges of P in order "red $\rightarrow$ blue".

Directly from the above definitions and claims, one can conclude:

Lemma C.8. *Let G be a connected graph. If an implementation of Algorithm 4.5 satisfies the conditions of Definition C.3, namely*

- *the spanning forest selected at the beginning of each stage in* `ExtendBond` *is chosen such that it is the unique minimal one with respect to edge weights ι,*
- *each path P is chosen such that it lexicographically minimizes the triplet $(|E(P)|, \mathrm{len}_\lambda(P), \vec{\iota}(P))$,*

then every generated partition $\mathcal{Z}$ of each k-bond $X \subseteq E(G)$ is in its canonical form $can(\mathcal{Z})$. □

We hence can modify Algorithm 4.5 according to Lemma C.8, which results in a "canonical algorithm" whose pseudocode is presented in Algorithm C.9.

Summarizing the previous findings, we finally obtain:

Theorem C.10. *Let G be a connected graph and $k \geq 2$, $m \geq 1$ integers. Recursive application of Algorithm C.9 returns every k-bond $X \subseteq E(G)$, $|X| \leq m$, at least once and in a canonical form.*

Algorithm C.9 One canonical stage of stepwise implementation

Input: A conn. graph G, param. $j, k, m \in \mathbb{N}, j < k, m \geq 1$, and a j-bond $Y_1 \subseteq E(G)$
Output: A collection of $(j+1)$-bonds as in Algorithm 4.5, such that each one is in the unique canonical form extending Y_1

if $j = 1$ **then**
  $F \leftarrow$ a ι-minimum spanning forest of $k-1$ trees
else
  $F \leftarrow$ a ι-minimum spanning forest of $k-2$ trees, $|F \cap Y_1| = 1$
end if
for all $d = \{u, v\} \in F \setminus \{e \in E(G) \mid \iota(e) \leq \tau(Y_1)\}$ **do**
  if $\nu(v) < \nu(u)$ **then** $(u, v) \leftarrow (v, u)$ **fi**
  GenStage$(j, Y_1, X = \{d\}, V_r = \{u\}, V_b = \{v\}, T_r = \{u\})$
end for.

procedure GenStage(j, Y, X, V_r, V_b, T_r)
  Let $G_1 \subseteq G$ be the conn. component of $G \setminus Y$ containing X
  if $|Y \cup X| > m - k + j + 1$ **then**
    return $\perp$ ▷ no way to get a k-bond of size $\leq m$
  end if
  if there does not exist a connected subgraph
    $T_b \subseteq (G_1 \setminus V(T_r)) \setminus X$ such that $V_b \subseteq V(T_b)$ **then**
    return $\perp$ ▷ the "no hyperplane" condition
  end if
  $P' \leftarrow$ a path $P' \subseteq G_1$ connecting some $r \in V_r$ and $b \in V_b$
    and minimizing $\langle |E(P)|, \mathrm{len}_\lambda(P), \vec{\iota}(P) \rangle$
  $P \leftarrow P' \setminus T_r$
  if such P does not exist **then**
    output $Y \cup X$ ▷ $Y \cup X$ is a $j+1$-bond
  else
    for all $c \in P$ **do**
      if $\iota(c) < \tau(Y \cup X)$ **then**
        continue
      end if
      Let u be the vertex in $c = \{u, v\}$ which is closer to T_r
      Let P_u be the component of $P - c$ which contains u
      GenStage$(j, Y, X \cup \{c\}, V_r \cup \{u\}, V_b \cup \{v\}, T_r \cup P_u)$
    end for
  end if
end procedure

Algorithm C.10 Canonical stepwise Circuit-Cocircuit algorithm

Input: A connected graph G, parameters $k, m \in \mathbb{N}, m \geq 1$
Output: All k-way bonds of G in a canonical form and with $\leq m$ edges

```
ExtendBond(j = 1, Y = ∅, V_r = ∅, V_b = ∅, T_r = ∅, G_b = ∅)
procedure ExtendBond(j, Y, V_r, V_b, T_r, G_b)
    F ← ι-minimum spanning forest F ⊆ (G \ Y) on k − 1 trees
    for all d = {u, v} ∈ F \ {e ∈ E(G) | ι(e) < τ(Y)} do
        if ν(v) < ν(u) then (u, v) ← (v, u) fi
        GenStage(j, Y, {d}, {u}, {v}, {u}, {v})
    end for
end procedure
procedure GenStage(j, Y, X, V_r, V_b, T_r, G_b)
    Let G_1 ⊆ G be the conn. component of G \ Y containing X
    if |Y ∪ X| > m − k + j + 1 then
        return ⊥                          ▷ no way to get a k-bond of size ≤ m
    end if
    P ← a path P ⊆ G_1 connecting some r ∈ V(T_r) and b ∈ V_b
          minimizing ⟨|E(P)|, len_λ(P), ῑ(P)⟩
    if such P does not exist then
        if j = k − 1 then output Y ∪ X                          ▷ X is a k-bond
        else ExtendBond(j + 1, Y ∪ X, ∅, ∅, ∅, ∅) fi
    else
        for all c = {u, v} ∈ P do
            Let u be the vertex in {u, v} which is closer to T_r
            Let P_u be the component of P − c which contains u
            Let P_v be the component of P − c which contains v
            if G_b \ {c} is disconnected then
                G_b ← a new subgraph interconnecting V_b
                if such G_b cannot be found then
                    return ⊥                  ▷ the "no hyperplane" condition
                end if
            end if
            if ι(c) < τ(Y ∪ X) then
                continue
            end if
            GenStage(j, Y, X ∪ {c}, V_r ∪ {u}, V_b ∪ {v}, T_r ∪ P_u, G_b ∪ P_v)
        end for
    end if
end procedure
```

C.4 The Final Algorithm

See Algorithm C.10 for the overall pseudocode which results from Algorithm C.9 after applying certain heuristic improvements (which do not affect validity).

D Supplements for Section 6

The tool `callgrind`[2] was used to measure the functions where the program spends the most time. Over 80 % of the running time in computation of Zlín with $k = 5, m = 4$ and Olomouc with $k = 6, m = 3$ is spent in the `shortestPath` procedure. When computing Zlín with $k = 8, m = 2$ the `shortestPath` procedure accounts for 54 % of the running time and finding a new blue subgraph accounts for 41 %. The memory consumption of the program is marginal.

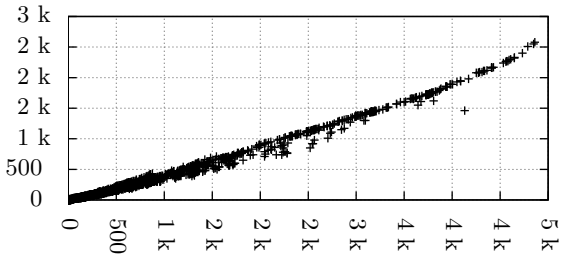


Fig. 8. Central Bohemian Region, $k = 3, m = 4$. Each point represents a branch of computation started with a single edge in X. The running time [ms] is on the y axis, the number of generated bonds is on the x axis. This suggests an easy speedup: parallelizing the `for` cycle at the beginning of the first stage

[2] `http://valgrind.org/docs/manual/cl-manual.html`