Study of the article: " An O(k2n2) Algorithm to Find a k -partition in a k -connected Graph "

Ludovic Hofer, Lambert Thibaud

Study of the article: "An $O(k^2n^2)$ Algorithm to Find a k-partition in a k-connected Graph"

Article authors: Ma Jun and Ma Shaoan
Publisher: Journal of Computer Science and Technology, 1994

Authors:

- Ludovic Hofer
- Thibaud Lambert

Directed by:

- Olivier Baudon
- Julien Bensmail

Second year of Master in Computer Science

october 2013 - january 2014

Contents

1 Introduction

The goal of the project was to study in depth the article [MM94], which, although pretends to answer positively to a difficult problem, surprisingly did not attract a lot of attention in the litterature. More precisely, in [MM94] is proposed a polynomial-time algorithm for constructing a special kind of vertex-partition of any k-connected graph which necessarily exists according to [Győ78, Lov77]. According to a private communication from E. Győri mentioned in [iNRN97], the algorithm from [MM94] should not be correct. Our goal was hence to verify if the algorithm was correct and if he had the claimed complexity. In order to have an entire comprehension of this article, we had have to search, read and understand several related articles.

Our very first task was to understand the algorithm. For this purpose, we have had to implement it in Java, using a framework which contains the implementation of several classic graph algorithms. We also have had to implement a k-connected graph generator in order to test both the validity and the complexity of the proposed algorithm. This allowed us to test our implementation.

Our implementation has pointed out the fact that specific executions of the algorithm lead to an endless loop. With those counter-examples, we prove that the algorithm proposed in [MM94] has not the expected complexity and also is not correct. We propose slight modifications to this algorithm, ensuring that instead of falling in an endless loop, the algorithm will end with a fail state. This change makes this algorithm easier to use as a base for a new Las-Vegas algorithm.

2 Definitions

In this section, definitions about graph theory are presented. Most of the following definitions are taken from or inspired by Wikipedia, Wikibook and Diestel's Graph Theory book [Die05].

2.1 Types of graphs

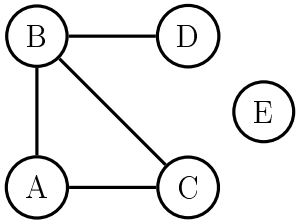
Undirected simple graph: An undirected simple graph G is a mathematical object composed of a set V of vertices and a set E of edges and is noted $G = (V, E)$.

An edge is an unordered pair $\{u, v\}$ of vertices and is sometimes also denoted uv for short.

A graph is used to model relation between objects. The vertices represent the objects and the edges represent the relations.

Thereafter if it is not stated otherwise, by "a graph", we implicitely refer to an undirected simple graphs.

Figure 1: An undirected simple graph with 5 vertices and 4 edges



$$V = \{A, B, C, D, E\}$$
$$E = \Big\{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}\Big\}$$

Graph size: The size of a graph $G = (V, E)$ is its number of edges, i.e. $|E|$.

Graph order: The order of a graph $G = (V, E)$ is its number of vertices, i.e. $|V|$.

Adjacency: In a graph $G = (V, E)$, two vertices, u and v, are adjacent if $uv \in E$.

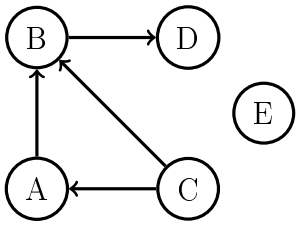
Vertex degree: The degree of a vertex v in a graph is equal to the number of vertices adjacent to it and is noted $d(v)$.

Acyclic graph: A graph $G = (V, E)$ is acyclic if for every pair $\{u, v\}$ of vertices in V, there is no more than one path[1] from u to v.

Directed graph: A directed graph is a graph whose edges are defined as ordered pairs (u, v). The edges of a directed graph are called arcs. u is called the head of the arc and v is the tail of the arc.

[1] See 2.2

Figure 2: A directed simple graph with 5 vertices and 4 edges

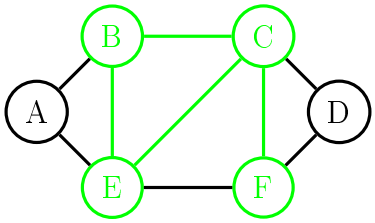


$$V = \{A, B, C, D, E\}$$
$$E = \Big\{(A, B), (C, A), (C, B), (B, D)\Big\}$$

Subgraph: Given a graph $G = (V, E)$ and a graph $G' = (V', E')$, then G' is a subgraph of G if and only if $V \subseteq V'$ and $E \subseteq E'$.

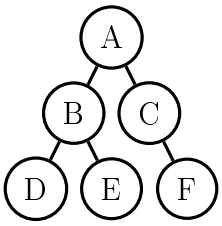
Induced subgraph: Given a graph $G = (V, E)$ and a set of vertices $V' \subseteq V$, the subgraph G' of G induced by V' is the subgraph of G whose vertex set is V' and whose edges are those of G whose two ends belong to V', it is denoted $G[V']$.

Figure 3: A graph and one of its subgraphs (in green)



Tree: A tree is an acyclic connected[2] graph.

Figure 4: A tree



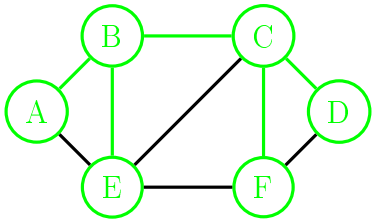
Root: In a tree, one vertex can be distinguished from the other. It is called the root of the tree.

[2]See 2.4

Descendant: Let r be the root of a tree. Given two vertices v and w, if v lies on the unique path between w and r, then w is a descendant of v.

Spanning tree: A tree T is a spanning tree of a graph G if it includes every vertex of G and is a subgraph of G.

Figure 5: A graph and one of its spanning trees (in green)



Complete graph: If all the vertices of G are pairwise adjacent, then G is complete. A complete graph on n vertices is denoted K_n.

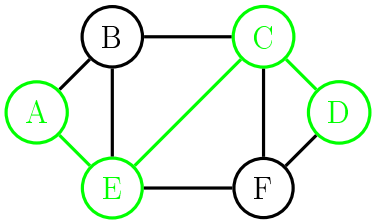
2.2 Paths

Path: A path is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \ldots, x_k\} \quad E = \{x_0x_1, x_1x_2, \ldots, x_{k-1}x_k\}$$

where the x_i are all distinct. The vertices x_0 and x_k are linked by P and are called its ends; the vertices $x_1, \ldots, x_{k-1}$ are the inner vertices of P. The number of edges of path a is its length, and the path of length k is denoted by P_k. Note that k is allowed to be zero; thus, $P_0 = K_1$.

Figure 6: A graph and one of its paths of length 4 (in green)



Vertex-disjoint path: Two paths P and Q are disjoint if and only if they (possibly) share their ends but have no common inner vertices.

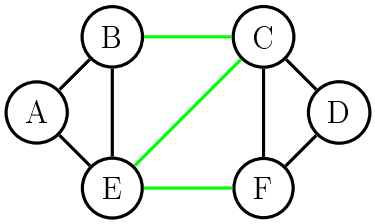
2.3 Cuts

Let $G = (V, E)$ be a graph.

Cut: A cut $C = (S, T)$ is a partition of V into two subsets S and T.

Cut-set: The cut-set of a cut $C = (S, T)$ is the set of edge defined as $\{uv \in E | u \in S \wedge v \in T\}$.

Figure 7: A graph and one of its cut-sets (in green)



Cut size: The size of a cut is the number of edges in the cut-set.

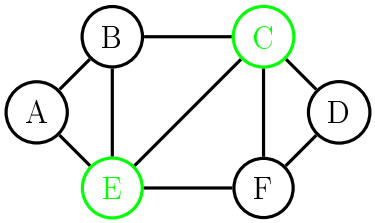
Articulation set: An articulation set C of a graph $G = (V, E)$ is a subset of V such that the subgraph of G induced by $V \setminus C$ is disconnected.

2.4 Connectivity

Connectedness: A graph G is connected if for every pair $\{u, v\}$ of vertices, there is a path from u to v. Otherwise, the graph is disconnected.

k-connectedness: A graph is k-connected if deleting any set of fewer than k vertices does not disconnect the graph.

Figure 8: A 2-connected graph and one of its articulation sets (in green)



Connectivity: The connectivity of a graph is the largest k for which it is k-connected.

Menger's Theorem: The Menger's Theorem is an important theorem concerning connectivity. It states the following fact:

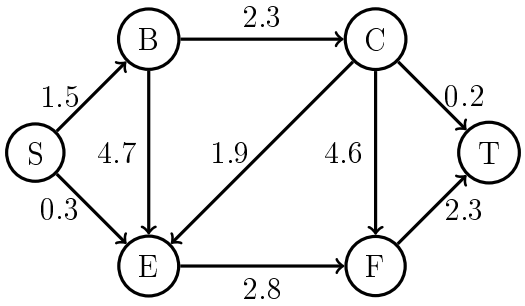
Let G be a undirected graph and u and v two distinct vertices. Then the size of the minimum vertex-cut for u and v is equal to the maximum number of vertex-disjoint paths from u to v.

2.5 Flows

Flow network: A flow network is a directed graph $G = (V, E)$ whose edges have a non-negative capacity, i.e. $\forall (u, v) \in E(G), c(u, v) \geq 0$.

Flow: Let G be an oriented graph, which has two specific vertices: a source s and a sink t. A flow is a function which associates to each edge $(u, v) \in E$ a real: $f : V \times V \to \mathbb{R}$ so that the following properties are met:

Figure 9: A flow network



- Capacity constraints: $\forall (u,v) \in E, f(u,v) \leq c(u,v)$
- Symetry: $\forall (u,v) \in E, f(u,v) = -f(v,u)$
- Flow conservation: $\forall u \in V - \{s,t\}, \sum_{v \in V} f(u,v) = 0$

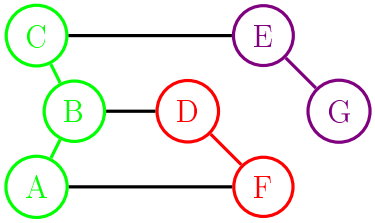
Maximum-flow Minimum-cut Theorem: The max-flow min-cut theorem [FD55] states that in a flow network, the maximum amount of flow passing from the source node to the sink node is equal to the capacity of the cut which has the minimum capacity.

2.6 Partitions

k-partition: Let $G = (V, E)$ be a graph. $\{V_1, ..., V_k\}$ is a k-partition of G, if and only if:

- $\forall i, G[V_i]$ is connected
- $\sum_{i=0}^{k} |V_i| = |V|$
- $\forall i, j \in \{1, \ldots, k\}^2, i \neq j, V_i \cap V_j = \emptyset$

Figure 10: A 3-partition (in red, green and puple) of a graph



3 Context: k-partion of k-connected graphs

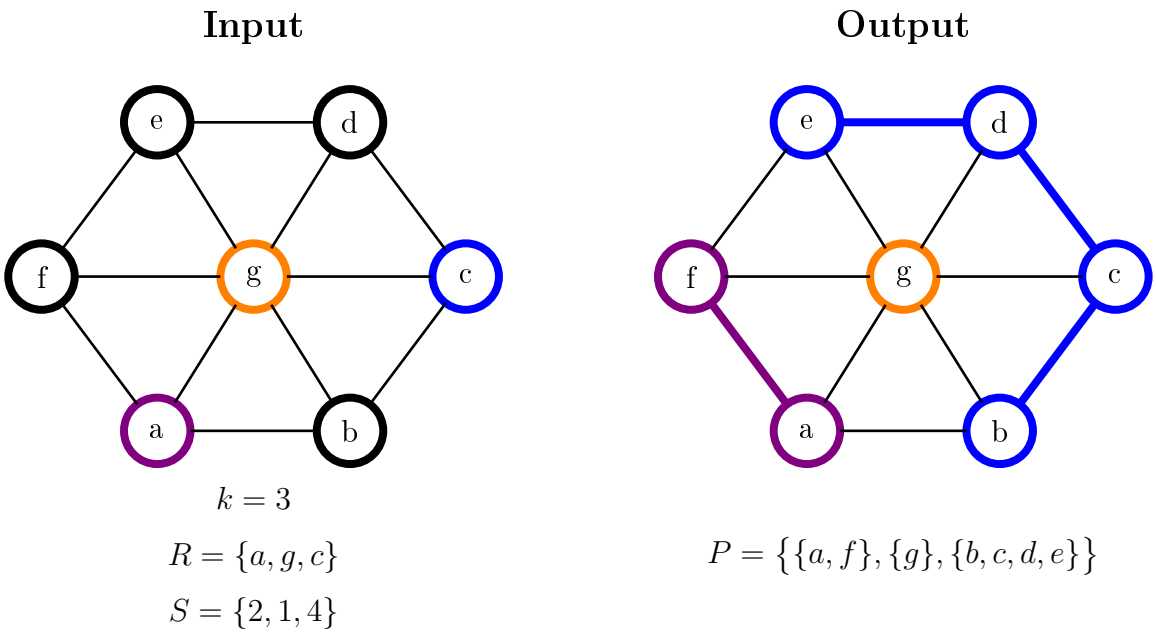
Győri-Lovász Theorem [Győ78, Lov77] states that k-connected graphs are k-partitionable for all $k \geq 1$. However the two proofs of this theorem are not constructive. We are interested in polynomial algorithms which are capable of finding k-partitions in k-connected graphs.

3.1 Problem description

The algorithm proposed in the article [MM94] claims to be able to find a k-partition in a k-connected graph in a polynomial time, and this no matter what is k. The complexity of the claimed algorithm for a graph $G = (V, E)$ with $|V| = n$ is $O(k^2n^2)$.

The k-partition problem which the article pretends to resolve can be formally defined as follows. Given a k-connected graph $G = (V, E)$, a set of k vertices $R = \{r_1, r_2, \ldots, r_k\}$ and k integers $S = \{s_1, s_2, \ldots, s_i\}$ summing up to the order of G. Find a partition $P = \{p_1, p_2, \ldots, p_k\}$ of V such that for any i from 1 to k, p_i contains r_i and $|p_i| = s_i$. An illustrated example is presented in Figure 11.

Figure 11: Input and output of the k-partition problem



3.2 Algorithm description

3.2.1 Principles

The main idea of the algorithm proposed in [MM94] is to expand different trees from the roots given as input. The specificities of this expansion is that it is both *progressive* and *balanced*.

It is *progressive* because the algorithm iterates over the different components. Each component will try to acquire free vertices (vertices which do not belong to a component) and if there is no free vertices among its neighborhood, it will try to steal a vertex from another component.

The algorithm is *balanced* because a component can only steal a vertex from a component which has a lower ratio of the number of wished vertices over the number of vertices currently owned by the component.

In order to avoid endless loops engendered by two components stealing the same vertex one after the other, a memory of the components to which a vertex has belonged is maintained.

A proof of the complexity of this algorithm is given in the article [MM94]. This proof can be summarized as follows.

1. The number of iterations of the expansion is $O(kn)$.
2. Every step of an iteration is $O(m)$.
3. An existing algorithm with complexity $O(m)$ allows to reduce the number of edges in a k-connected graph to a number of edge $O(kn)$ [NI92].

From those three properties, the authors affirm that the complexity of the algorithm is $O(k^2n^2)$. But, even if the second possibility can be verified and if the validity of the third one does nott influence the fact that this algorithm is polynomial, the proof of the first property seems to be wrong since we exhibit a counter-example in section 4.3.2, showing that the algorithm may loop endlessly.

3.2.2 Used data

As it was mentioned previously, the algorithm of [MM94] partition the graph into trees. Thoses trees needs to be stored, so both vertices and edges of each tree must be memorized and updated through the execution. The tree rooted at $v = r_i$ is denoted T_v.

The history of the trees to which a vertex has belonged is stored in a set of vertices. An *HashMap* with vertices as keys and a set of vertices as values can then be used to obtain the history of any vertex. The data are stored in a structure called treeNode and the set of roots whose tree has owned the vertex v is denoted treeNode[v]. It is important to note that the history of some vertices can be reseted in some specific cases (see section 3.2.4).

The feature ensuring that the expansion is balanced is called the p-value. Each tree has its own p-value which is defined as the number of vertices wished for the tree divided by the number of vertices currently in the tree. A tree which has a p-value of 1 is a component which has the right amount of vertices. When iterating over such a tree, nothing will be done, because it does not need to acquire more vertices.

3.2.3 Acquiring vertices

When iterating over a tree which needs to gain vertices, the first step is to compute the neighborhood of the tree, i.e. the vertices of the graph which are neighbors to one (or more) of the vertices of the tree but are not in it. If among this neighborhood, there is at least one free vertex, i.e. a vertex which does not belong to any tree. The tree will acquire all the free vertices of this neighborhood, except if there are more free vertices than required. In that case, the tree acquires enough vertices so that its order reached the desired size, but no more. The vertices to acquire are chosen randomly among the available ones.

3.2.4 Stealing vertices

If there is not any free vertex in the neighborhood of the tree, then it tries to steal a vertex from another tree. Several rules defines whether a vertex can be stolen or not.

- The root of a tree (i.e. the r_i's) cannot be stolen.
- A tree T_v cannot steal a vertex u if T_v belongs to treeNode[u].
- A tree cannot steal a vertex from another tree which has a lower p-value.

Once those three rules have been applied to remove some vertices of the neighborhood, if there is no vertex left, then the tree doest not steal any other vertex during this iteration. On the other side, if there are still multiple vertices remaining, then one vertex is chosen as the vertex to steal. The following procedure is applied.

1. Keep only the vertices which have the smallest p-value of the set.
2. Among this new set, keep only the vertices which have the smallest degree inside the tree.
3. If there are still multiple vertices, pick one randomly.

If the vertex v which must be stolen has degree greater than 1 in its tree, removing it disconnects the tree. To avoid this kind of situation, the subtree rooted at v is cut and only v is stolen. All the other vertices u of the subtree have their history cleaned and are considered again as free vertices, i.e. treeNode[u] $= \emptyset$.

3.3 State of the art: existing algorithms

In this section, we present algorithms related to k-connectivity and k-partitioning.

3.3.1 Computing k-connectivity

As we have seen in Section 2.4, the connectivity of a graph is equal to the size of its minimal vertex cut. The max-flow min-cut Theorem states that the maximum flow between to vertices is equal to the minimum cut to disconnect these two vertices. But if we use flow algorithm directly, do not compute the connectivity, since it will be possible to have two different flows passing through the same vertex.

So first the graph needs to be transformed. For this purpose, we need a capacity on the vertices. To do so each vertex is divided into two new vertices. Each incoming edge is attached to the first vertex and each leaving edge to the second vertex. Another edge is added between the two new vertices. The capacity of all edges of the graph is set to 1.

Then to find the connectivity, we compute the maximum flow between each pair of nodes and we keep only the lowest value. This value is the connectivity.

3.3.2 k-partitioning

In this section, we focus on the different existing k-partitioning algorithms.

First of all, since it has been proven that for some kinds of graphs finding a k-partition is an NP-hard problem [BF06], finding a partition in a graph is NP-hard in general.

Data: $G = (V, E)$ a graph
Result: k the connectivity
$min = \infty$;
$G^{'} = TransformNode(G)$;
forall $u, v \in V, u \neq v, (u, v) \notin E$ **do**
    $f = maximumFlow(G^{'}, u, v)$;
    if $f < min$ **then**
        $min = f$;
    end
end
return min;

Algorithm 1: Computing the connectivity of a graph

As we mentioned earlier, it has been proven that, for a k-connected graph G, a k-partition always exists [Győ78, Lov77]. The algorithm from [MM94] apart, no polynomial general algorithm, i.e. for all values of k, have been proposed. Polynomial algorithms have been proposed for $k = 2$[STN90] and $k = 3$[WK94]. More recently, it has been proven in [iNRN97] that if G is a planar graph and if the roots of the partition are located on the same face of a plane embedding of G, there is a linear-time algorithm for $k = 4$.

4 Work

4.1 Tools

4.1.1 Generating k-connected random graphs

In order to test our implementation of the algorithm proposed in [MM94], k-connected graphs had to be generated. We have implemented two methods, although the graphs produced are not uniformly randomised.

The first method is based on the vertex degree. To be k-connected, a graph needs to have vertices of degree at least k. For each vertex, edges are added until the vertex has degree at least k. Once this procedure achieved, then possibly the reduced graph is k-connected.

Since this first method is not deterministic, we have implemented another method which produces a k connected graph by creating several complete components and linking them together. The algorithm is described in Algorithm 2.

Data: k the connectivity,
N the number of components,
n the size of components
Result: A k-connected graph
$components = \emptyset$;
$edges = \emptyset$;
for *i = 1 to N* **do**
    $components = components \cup newCompleteComponent(n)$
end
for *i = 1 to N-1* **do**
    for *j = 1 to k* **do**
        $edges = edges \cup (components[i][j], components[i+1][j])$
    end
end
$g = newGraph(components, edges)$;
return g;

Algorithm 2: k-connected graph generator

First, the complete components are generated. Then k edges are added between the first and the second components, k edges are added between the second and the third ones and so on.

The edges between every pair of components are not randomly selected, otherwise it would be easy to diconnect the graph by removing less than k vertices. So we procede in the following way for adding the edges. Suppose that the vertices of each component are numbered from 1 to the order of the component. The edges between the vertex 1 of the first component and the vertex 1 of the second component are selected. The same process is repeated for the second vertices to the k-th vertices and then for the other components. At the end, the resulting graph is k-connected.

4.1.2 Calculating a sparse spanning subgraph

In order to reduce the execution time of the main algorithm, we had to implement another algorithm described in [NI92].

This algorithm is called `FOREST` and takes a graph $G = (V, E)$ as a parameter. The output is slightly more complicated since it does not simply return the sparse spanning subgraph. Instead, the algorithm returns an array A of set of edges. The size of the array returned is $|E(G)|$. The graph induced by the set of edges $\bigcup_{i=1}^{k} E_i$ is k-connected, if the connectivity of G is greater or equal to k.

The main difficulty about the implementation of this algorithm was the use of a specific data structure that allows an access to the maximum, an insertion and a deletion of an element with a constant amortized cost. Even if such a structure does not exist for any kind of utilisation, it is possible to create one which has all the required properties for this algorithm, since the utilisation is very specific and since an element has always an integer associated value that can grow only from 1 at each step.

Tests have been created and run in order to prove that the result of our implementation of the `FOREST` algorithm was corresponding to the main objective:

$$\forall k, k \leq K(G), K(V, \bigcup_{i=1}^{k} E_i) = k.$$

This test has been validated on simple graphs, but it has also been seen that our implementation did not produce the expected result if the input is a multigraph. We checked this property by following the algorithm on small multigraphs and this showed us that the problem was not coming from the implementation, but rather from the algorithm. Since we are working on simple graphs, this was not really a problem for our tests.

4.2 Implementing the algorithm

In order to facilitate the comprehension of the algorithm implementation and the debugging, we choose to present the different parts of the algorithm. While we were implementing it, we also noted that some lines where misplaced, breaking the coherency of the algorithm.

In order to have the same complexity as the one proved in the paper, we sticked to the same data structures. In order to accept graphs where vertices are not only integers, we added an HashMap, allowing an access in constant time to the index of a vertex.

In the library we were using, spanning graphs were not designed to be modified. We created our own basic class which was handling trees with the needed methods like cutting a tree on a specific vertex.

4.3 Criticism

4.3.1 Finding a counter-example

Once the main issues of the implementation of the algorithm were solved, we had still some issues with the algorithm falling in endless loops. Our investigations on this specific problem ended with the fact that the problem was coming from the algorithm and not from the implementation. This was ensured by verifying the validity of the execution of our algorithm step by step on these specific problematic instances.

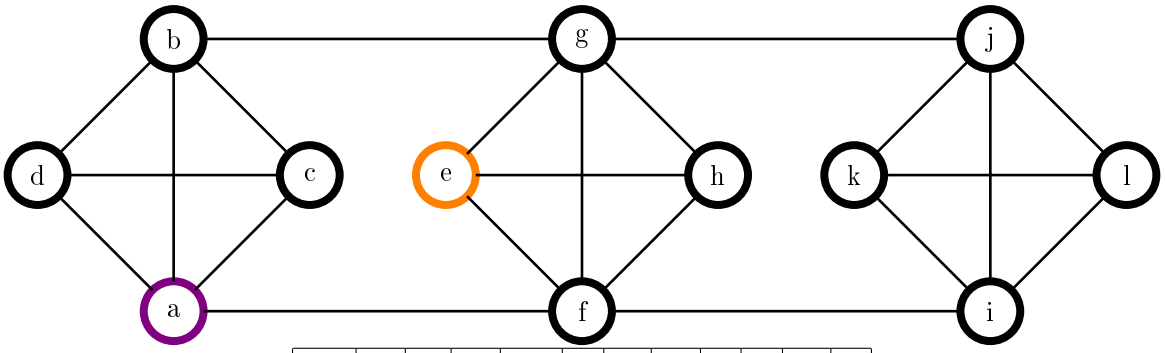
4.3.2 Detailed example of a failed execution

We here present an example of an execution which fails with the algorithm described in [MM94]. The input of the algorithm is as follows :

- G is the graph drawn in Figure 12,
- $k = 2$,
- roots $= \{e, a\}$,
- partitionSize $= \{5, 7\}$.

In order to illustrate the status of the algorithm, edges which are included in a tree have a stronger thickness, while both edges and vertices are colored according to the tree they belong to. Additionally, an array is presented under each graph, showing the content of the tree-node array which is used by the algorithm to choose which action has to be executed.

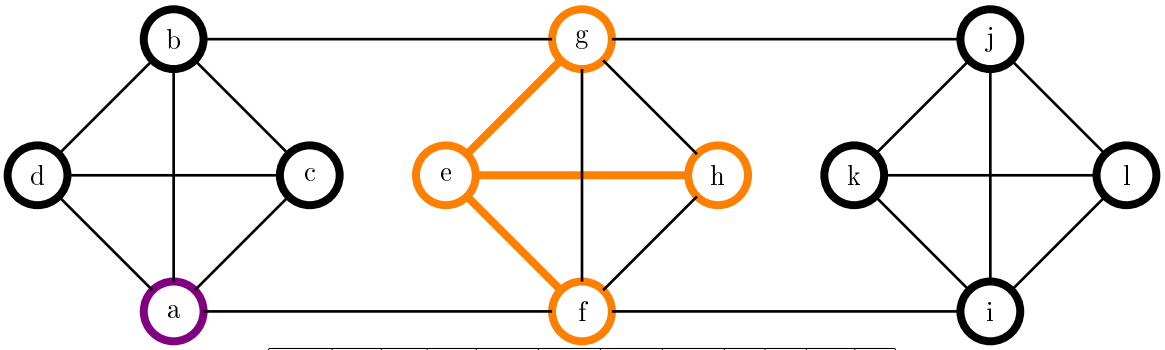
Figure 12: Failed Execution : Initial graph



a	b	c	d	e	f	g	h	i	j	k	l
T_a				T_e							

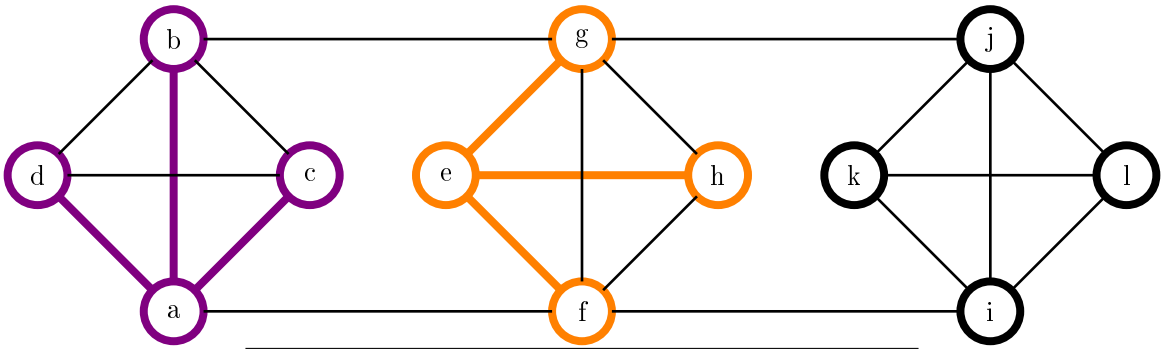
During the first two steps, each partition needs to acquire more vertices, so each one grows by adding all the available neighborhood.

Figure 13: Failed Execution : Step 1



a	b	c	d	e	f	g	h	i	j	k	l
T_a				T_e	T_e	T_e	T_e				

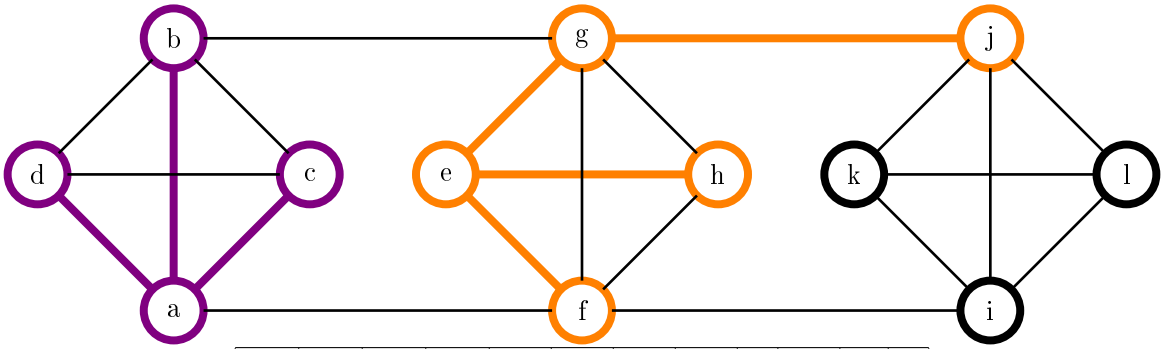
Figure 14: Failed Execution : Step 2



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e	T_e	T_e				

At step 3, two vertices are available to increase the size of the current working tree: i and j. Since it needs only one of the two vertices, say j, is chosen.

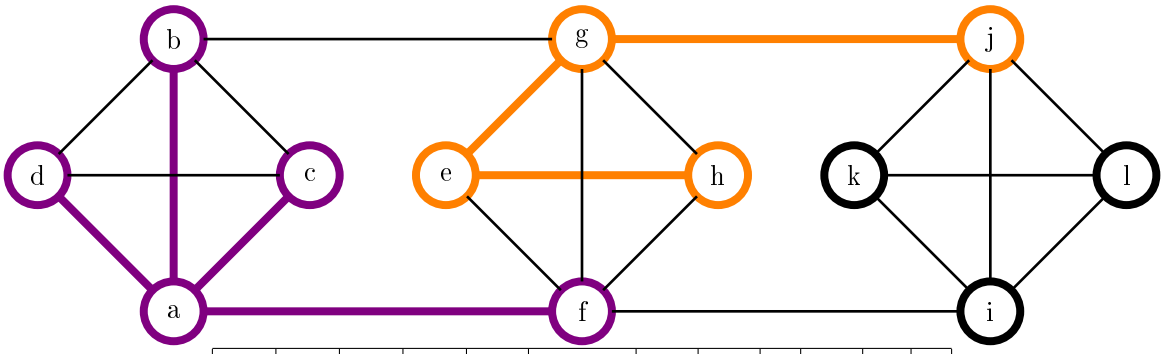
Figure 15: Failed Execution : Step 3



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e	T_e	T_e		T_e		

At step 4, the tree rooted at a cannot be simply extended, since every adjacent vertex belongs to the other tree. The adjacent free vertices are f and g. Both respect the conditions necessary to allow the swapping. According to the algorithm, the vertex to swap must have the lowest degree in the concerned tree among the swappable candidates. It cannot be g since it has a higher internal degree than f in the concerned tree. The swapped vertex is then f.

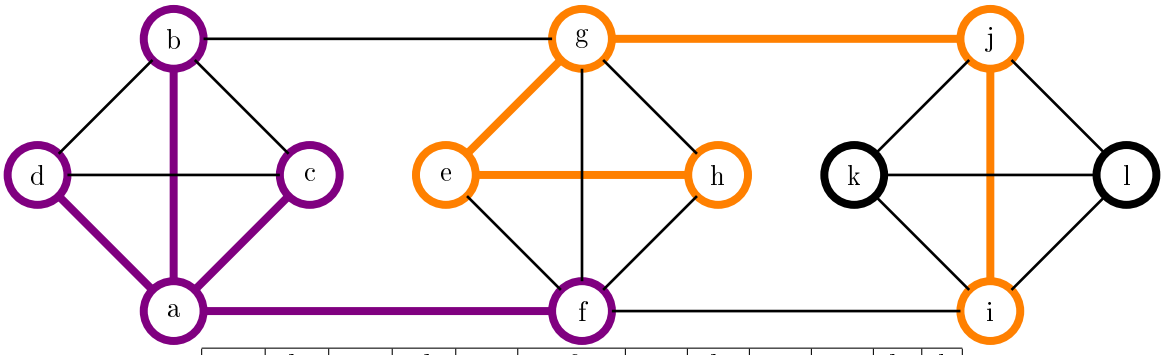
Figure 16: Failed Execution : Step 4



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e, T_a	T_e	T_e		T_e		

At step 5, the tree rooted at e must get one vertex back, since the other tree has taken one of its vertices. The two free candidates are i and k. Since only one is missing, only one must be added. In our case, i is added.

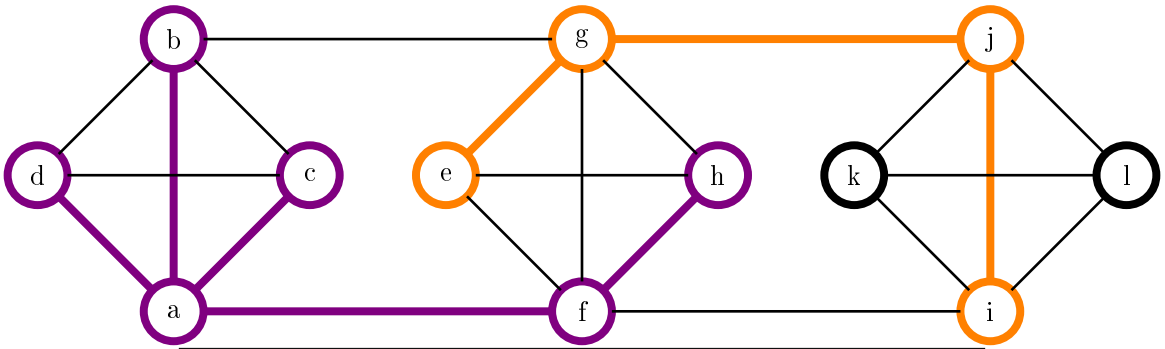
Figure 17: Failed Execution : Step 5



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e, T_a	T_e	T_e	T_e	T_e		

At step 6, the tree rooted at a has no available adjacent vertex, it then has to swap a vertex among the neighborhood: e,g,h,i. e is not an option since it is the root of the other tree. Since g has an internal degree of 2, it cannot be choosen since h and i have an internal degree of 1. The swapped vertex is then h or i. Since nothing in the algorithm determines which one must be chosen, h might be swapped at that step.

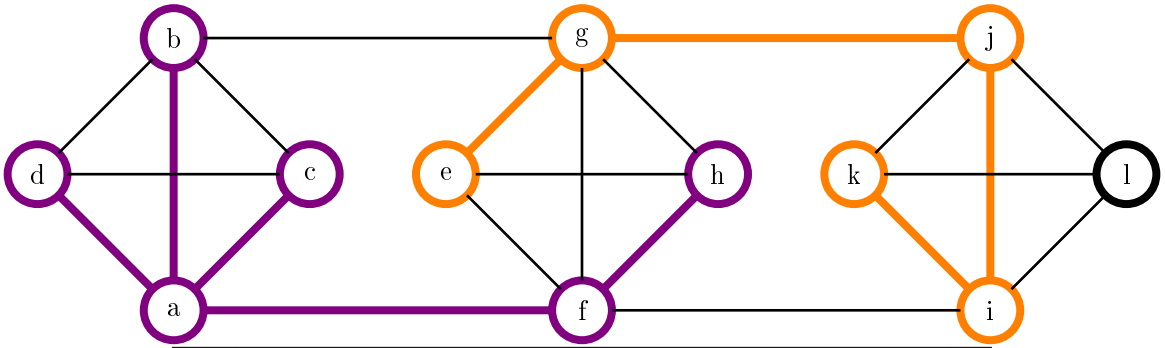
Figure 18: Failed Execution : Step 6



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e, T_a	T_e	T_e, T_a	T_e	T_e		

At step 7, the tree rooted at e must add a vertex among k and l, the vertex k can be freely added with the edge $\{i, k\}$.

Figure 19: Failed Execution : Step 7

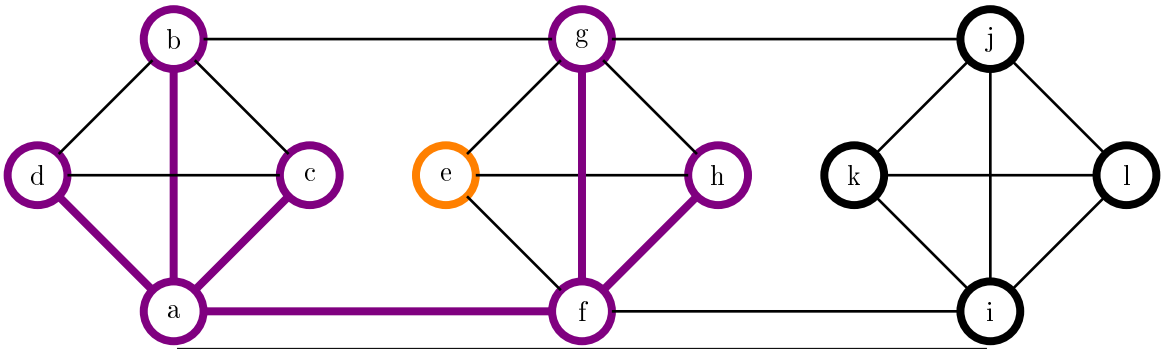


a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e, T_a	T_e	T_e, T_a	T_e	T_e	T_e	

At step 8, the tree rooted at a must take a last vertex but, since it has no available adjacent vertex, it must then take a vertex from the tree rooted at e. The swappable vertices are g and i. Since they both have the same internal degree in the concerned tree, g can be choosen as a valid choice for the extension.

By swapping with g, the subtree rooted at g must be cut from the tree rooted at e. All the removed vertices (i,j,k) are then set back as unexplored, but not the swapped vertex g.

Figure 20: Failed Execution : Step 8



a	b	c	d	e	f	g	h	i	j	k	l
T_a	T_a	T_a	T_a	T_e	T_e, T_a	T_e, T_a	T_e, T_a				

After step 8, we are in a situation where the algorithm is stucked in a dead-loop. The tree rooted at a has the right amount of vertices, and steps concerning this tree will not produce any modification. The tree rooted at e cannot expand anymore, because all the adjacent vertices have already belong to him[3] and have not been reset[4].

4.3.3 Counter-example in general case

In order to ensure that $k = 2$ is not a specific case for which the algorithm presented in [MM94] can loop endlessly, we present a proof that an execution can loop endlessly for any k.

The proof is by induction. Since we have already exhibited a counter-example for $k = 2$, we only need to prove the induction step.

Let G be a k-connected graph for which the algorithm has a probability of looping endlessly strictly greater than 0 for the entry $\{r_1, r_2, \ldots, r_k\}$ and $\{n_1, n_2, \ldots, n_k\}$. Then a graph G' can be found such that G' is $k+1$-connected and there exists an entry for which the algorithm has a probability of looping endlessly strictly greater than 0.

Proof: If we create G' by adding a new vertex v connected to every vertex of G, i.e. $V(G') = V(G) \cup v$ and $E(G') = E(G) \cup \bigcup_{u \in V(G)} \{u, v\}$, it is obvious that G' is $k+1$-connected, since every minimal vertex cut of G needs to have one additional vertex in order to be a cut of G'.

An execution of the algorithm with G', $\{r_1, r_2, \ldots, r_k, v\}$, $\{n_1, n_2, \ldots, n_k, 1\}$ as input has a probability of looping endlessly strictly greater than 0, as the introduced vertex v has no influence since the associated tree is full since the beginning and the new edges are not useable.

4.4 Improvements

4.4.1 Vertex selection condition

To select the vertex to swap between two trees, the article algorithm bases its choices on the vertex degree. As we have seen before, the selection of specific vertices can lead to an endless

[3] See line 28 of the algorithm
[4] See line 47 of the algorithm

loop. Choosing the vertex with the lowest degree can cut a long branch of the tree, whereas select another vertex with a greater degree removes less vertices to the tree.

That is why, we have modified the vertex selection condition. Rather than basing the selection condition on the vertex degree, we base it on the number of descendants. So, when a vertex is swapped the minimum number of vertices is removed from the tree.

This modification allows to avoid the endless loop for the counter-example presented above. However there are still some executions on other graphs which are falling into endless loops.

4.4.2 Detecting endless loops

A counter-example which shows that the algorithm can fall in endless loops have been exhibited. This raises one question: is it possible to detect when there is an endless loop? An improvement has been added to the algorithm in order to detect such situations. To do so, we save the last root index where we have added or swapped a vertex. If the saved index is equal to the current root index and no vertex have been added or swap then it means that the algorithm has entered in an endless loop. Then the execution is stopped.

5 Conclusion

Through this study, we have demonstrated that the algorithm proposed in [MM94] is not correct, for any k. The private communication from E. Győri mentioned in [iNRN97] has been proved by exhibiting a counter-example, which we then generalized to show that there is no value of k for which the proposed algorithm is successful.

The counter-example was found by implementing the algorithm and running it on randomly generated graphs. Therefore, the tools implemented can be used to test new approaches similar to the algorithm or needing the generation of k-connected graphs.

Since we proved that the algorithm proposed in [MM94] is not correct, there are still no polynomial solutions solving the k-partition problem for any k. So at the moment, only the specific algorithms for particular values of k mentioned in Section 3.3.2 are known to exhibit solutions in polynomial time.

We proposed a modification which allows to avoid endless loops, informing of a failure if the execution leads to an endless loops. This specific modification could allow to create a Las Vegas algorithm if the probability that the algorithm returns successfully the solution is equal or greater than $\frac{1}{2}$. Proving this would not bring a polynomial solution to the k-partition problem, but it would already prove that the k-partition problem is in ZPP.

References

[BF06] D. Barth and H Fournier. A degree bound on decomposable trees. *Discrete Mathematics*, 306(5):469–477, 2006.

[Die05] R. Diestel. *Graph Theory : 3rd Edition*. Springer-Verlag Heidelberg, 2005. Graduate Texts in Mathematics **173**.

[FD55] D. R. Fulkerson and G. B. Dantzig. Computation of maximal flows in networks. *Naval Research Logistics Quarterly*, 2(4):277–283, 1955.

[Győ78] E. Győri. *On division of graphs to connected subgraphs*, pages 485 – 494. North-Holland Publ. Comp, Amsterdam ; Oxford ; New York, 1978.

[iNRN97] Shin ichi Nakano, Md.Saidur Rahman, and Takao Nishizeki. A linear-time algorithm for four-partitioning four-connected planar graphs. *Information Processing Letters*, 62(6):315 – 322, 1997.

[Lov77] L. Lovász. A homology theory for spanning trees of a graph. *Acta Mathematica Hungarica*, 30(3):241–251, 1977.

[MM94] J. Ma and S. Ma. An $o(k^2n^2)$ algorithm to find a k-partition in a k-connected graph. *Journal of Computer Science and Technology*, 9:86–91, 1994.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of a k-connected graph. *Algorithmica*, 7(1-6):583–596, 1992.

[STN90] Hitoshi Suzuki, Naomi Takahashi, and Takao Nishizeki. A linear algorithm for bipartition of biconnected graphs. *Information Processing Letters*, 33(5):227 – 231, 1990.

[WK94] Koichi Wada and Kimio Kawaguchi. Efficient algorithms for tripartitioning triconnected graphs and 3-edge-connected graphs. In Jan Leeuwen, editor, *Graph-Theoretic Concepts in Computer Science*, volume 790 of *Lecture Notes in Computer Science*, pages 132–143. Springer Berlin Heidelberg, 1994.