SAPIENZA
UNIVERSITÀ DI ROMA

Polynomially Recognising Graphs Where Saturating Flows Are Always Maximum

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Corso di laurea in Informatica
Cattedra di Progettazione di Algoritmi

Candidato
Federica Granese
n° matricola 1615552

Federica Granese

Responsabile
Daniele Gorla

Daniele Gorla

A/A 2016/2017

- SAM: Se faccio ancora un passo, non sarò mai stato così lontano da casa mia.
- FRODO: Forza, Sam. Ricorda cosa dceva Bilbo: "È pericoloso, ...
- BILBO: "... Frodo, uscire dalla porta. Ti metti in strada, e se non dirigi bene i piedi, non si sa dove puoi finire spazzato via".
(Il Signore degli Anelli - La Compagnia dell'Anello)

Abstract

This dissertation considers network models where information items flow from a source vertex to a sink vertex, specifically standard flows networks with capacity on edges. Starting from the characterization of graph topologies ensuring that every saturating flow under every capacity-to-edge assignment is maximum in [7], the dissertation provides a polynomial-time algorithm for checking this property. Such a property is called *edge-weakness*. A notion strongly related to edge-weakness is that of *minimal edge separator* (or *mes*): by examining if there exists a path in the graph touching twice a mes, it is possible to conclude that such a graph is edge-weak. Since there is no need to examine all the minimal edge separators in the graph, the dissertation also provides an algorithm for building a *chain* of mes iteratively. Such an algorithm comes out from that of the enumeration of *all* minimal edge separators in the graph (which is always reported in this dissertation). Finally, the dissertation shows the implementations of the algorithms.

Contents

Contents **v**

List of Algorithms **vi**

1 Introduction **1**

2 Basic Definitions on the Model **7**
2.1 Background on Graph Theory 7
2.2 Flow Networks 8
2.3 Edge-Weakness 11

3 Checking Edge-Weakness **13**
3.1 Preliminaries 13
3.2 Efficient enumeration of all minimal edge-separators in a graph . . . 14
3.2.1 Definition and characterization of minimal edge-separator . . 14
3.2.2 Definition and characterization of successor and predecessor of a minimal edge-separator 16
3.2.3 Enumerating all minimal edge-separators 18
3.2.4 An Algorithm for enumerating all minimal edge-separators . 20
3.3 Edge-Weakness 25
3.3.1 Characterization of Edge-Weakness 25
3.3.2 Edges set-coverage and complete chain of mes 26
3.3.3 Correctness of Checking Edge-Weakness 28
3.3.4 Polynomial-time Algorithm to checking Edge-Weakness . . . 33

4 Implementation **39**
4.1 Code Organization 39
4.2 Source Code 40
4.2.1 Algorithm 1 40
4.2.2 Algorithm 2, Algorithm 3, Algorithm 4 44
4.2.3 Graph Viewer 54

4.2.4 Example of Graph Viewer 61
4.3 Remarks 63

5 Conclusion 67

Bibliography 71

List of Algorithms

1 Generating all minimal edge-separators 20
2 Checking Edge-Weakness 34
3 Generating an e-minimal mes smaller (w.r.t. $\sqsubset$) than X 34
4 Generating next minimal edge-separator 35

Chapter 1

Introduction

Flow networks lie between several research fields, including applied mathematics, computer science, engineering, management, operational research and so on (see [16][20][4]). The field has a rich and long tradition, tracing its roots back to the work of Gustav Kirchhof and other early pioneers of electrical engineering and mechanics who first systematically analyzed electrical circuits. Specifically, a *flow network* is a directed graph with a chosen pair of vertices called *source* and *sink* (resp. s and t). In the standard model of flow networks, edges are endowed with capacities and a flow is possible only if it does not exceed the capacity of all edges it passes through. Moreover, a flow saturates a network if, for every path from s to t, there exists at least an edge having capacity equal to the amount of flow passing on it.

In studying flow networks one of the main issues is trying to send as much flow as possible between s and t without exceeding the capacity of any edge. Such a problem is called *maximum flow*. Examples [16][13] of the maximum flow problem include determining the maximum amount of cars in a road network, messages in a telecommunication network, and electricity in an electrical network. For calculating a maximum flow in a net, sophisticated algorithms are needed. These algorithms are of two types [16][12] [9]:

1. *Augmenting path algorithms*, that incrementally augment the flow along paths from the source node to the sink node.

2. *Preflow-push algorithms*, that incrementally relieve the flow from nodes with excesses by sending flow from the node forward toward the sink node or backward toward the source node.

Clearly, every maximum flow is a saturing flow. The opposite is not true. For this purpose, consider the example of Figure 1.1 (a): the maximum amount of information units that can pass through the net is 3 (as shown by paths s v_1 v_4 t,

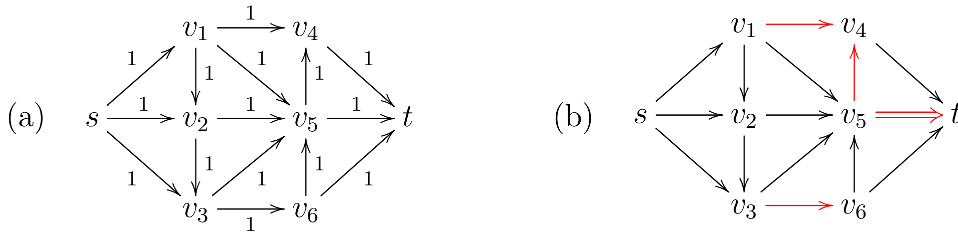


Figure 1.1: (a) A flow network with edges labeled with their capacities. (b) The graph under the network. The red edges form a mes, the not dashed edges form an edge-critical path, the edge with the double arc is the critical edge. Clearly, the graph is edge-weak.

$s\ v_2\ v_5\ t$ and $s\ v_3\ v_6\ t$). However, in a distributed scenario, due to partial knowledge of the net, the flow could take the path $s\ v_3\ v_6\ v_5\ t$ and the net would be saturated by exchanging two information units only (in other words, such a net has a maximum flow of value 3, but admits a saturating flow of value 2). Hence, there exist flow networks that, under some capacity-to-edge assignment, can have a non-maximum saturating flow. The graphs underlying such a type of flow networks are called *edge-weak*. An algorithm [7] that non-deterministically chooses paths and saturates them always calculates a maximum flow if and only if the graph is not edge-weak.

Recognizing if a graph is edge-weak by checking all possible capacity assignements is not a decidable problem. Indeed, since for each edge it is possible to assign any real, the capacity functions for a flow network are infinite.

To avoid this issue, edge-weakness is addressed in a different way and the focus is on the topology of the graph. Indeed, recalling from [7], a graph is edge-weak if and only if there exists a *minimal edge separator* and a path touching it at least twice. A minimal edge separator (or *mes*) is a minimal set of edges whose removal from the graph disconnects s from t. A mes is a bit different from a *cutset* (that is always a set of edges). Indeed, a cutset comes from a bipartition of the vertex set of the graph, whereas a mes leads to a bipartition of the vertex set of the graph. In both cases, each edge of the set has one endpoint in one vertex subset and the other endpoint in the other vertex subset; however, if a cutset separates the graph in two connected components, a mes separates the graph in two special subgraphs. If G is the graph and X is the mes, then such subgraphs are called G_s^X and G_t^X. G_s^X is the subgraph containing all vertices reachable from s, while G_t^X is the subgraph containing all vertices reaching t.

The characterization of edge-weakness in those new terms is close to that of *weakness*. Weakness [6][8] is a form of inefficiency in *depletable channels*. A depletable channel is like a flow network, except flow networks have edges endowed

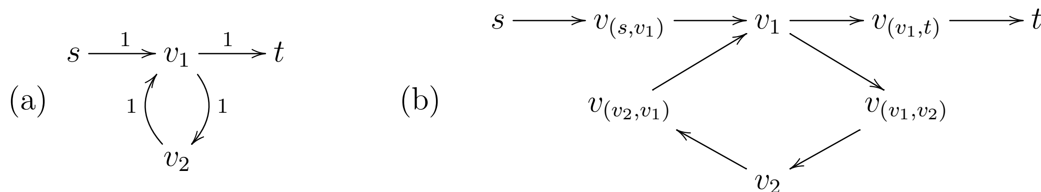


Figure 1.2: A flow network (a) and its edge-expansion (b). The 'expanded edges' are labeled with v_e where e is the corresponding edge in the original graph.

with capacities and depletable channels have vertices endowed with charges; the function allocating charges is always called flow. The underlying graph of a depletable channel is called *weak* if there exists a charge assignment to vertices such that the resulting depletable channel admits a non-maximum saturating flow.

Anyway, weakness and edge-weakness do not coincide. In particular the standard translation from edge-capacitated to vertex-capacitated networks can not be used to reason about edge-weakness in terms of weakness. Let *edge expansion* denote the translation of a network to the corresponding channel. Such a channel is obtained by adding, for each edge in the network, a vertex (called *expanded edge*) endowed with the capacity of the original edge; the original vertices are endowed with infinite charge. On one hand, every edge-weak graph is edge-expanded to a weak graph (each mes becomes an mvs in the translation); on the other hand, a non edge-weak graph can be edge-expanded in a weak graph. For this purpose, consider the flow network in Figure 1.2 (a) with its corresponding channel transformation depicted in Figure 1.2 (b). The graph underlying the network is not edge-weak but the graph underlying its edge-expansion is weak. This happens because not all the mvs in the edge-expanded graph have their counterpart in the original graphs. Indeed, the mvs $\{v_1\}$ in Figure 1.2 (b) (which causes weakness) has not a corresponding mes in Figure 1.2 (a). Thus, if by executing the algorithm for weakness on an edge-exapanded graph we obtain that such a graph is not weak then the original graph is also not edge-weak; but if such a graph is weak then there is no way to determine if the original graph is edge-weak or not (see Figure 1.2). Hence, if in many cases the models with capacities on vertices and those with capacities on edges are interchangeable, in this context such an interchange is not possible.

In [6] the authors characterize weakness in terms of the existence of a path that passes at least twice through a minimal vertex separator, which is a minimal set of vertices whose removal disconnects s and t. A trivial algorithm [6] to determine if a graph is weak is to generate all mvs and, for each of them, check if there exists a walk that touches the mvs twice. Unfortunately, the number of mvs in a graph can be exponential in the number of vertices [14]. However, the authors show that

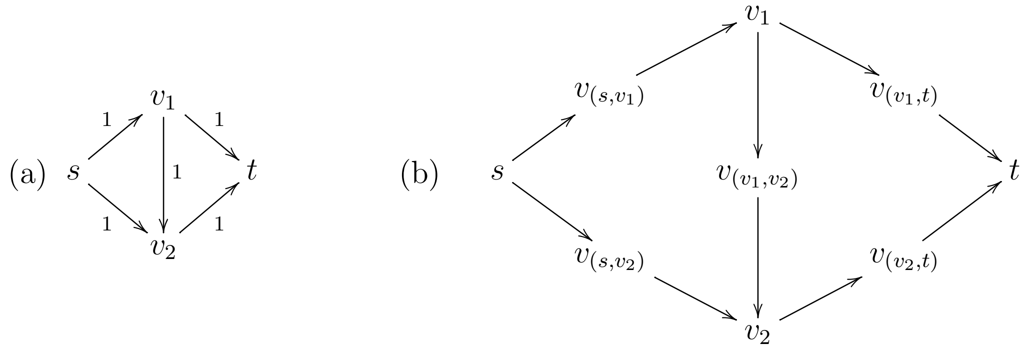


Figure 1.3: A flow network (a) and its edge-expansion (b) to show the dramatic increase of mvs.

it is enough to examine at most a chain of mvs. The method for building a chain of mvs relies on the study of the enumeration of all minimal vertex separator in undirected graphs [18]. In [18] the authors give an algorithm which uses a greedy approach and enumerates all minimal vertex separators via a level-by-level adjacent vertex replacement scheme, where the separators at each level are generated by replacing every vertex of each separator in the previous level with a set of its adjacent vertices, thus avoiding expanding all previously generated separators and making the search considerably more efficient. Moreover, the authors in [6] give a polynomial-time algorithm for checking weakness.

As previously described each minimal edge separator in a graph becomes a minimal vertex separator in the edge-expaded graph. For this purpose, let G and G' be the graphs in Figure 1.3 resp. (a) and (b). Each mes in G matches with the mvs in G' formed by the expansion of the edges of the mes. For example, the mes $\{(s, v_1), (s, v_2)\}$ in G matches with the mvs $\{v_{(s,v_1)}, v_{(s,v_2)}\}$ in G', the mes $\{(s, v_2), (v_1, v_2), (v_1, t)\}$ in G matches with the mvs $\{v_{(s,v_2)}, v_{(v_1,v_2)}, v_{(v_1,t)}\}$, the mes $\{(s, v_1), (v_2, t)\}$ in G matches with the mvs $\{v_{(s,v_1)}, v_{(v_2,t)}\}$ in G' and the mes $\{(v_1, t), (v_2, t)\}$ in G matches with the mvs $\{v_{(v_1,t)}, v_{(v_2,t)}\}$ in G'. Anyway, finding this mes with the enumeration of the mvs in the expanded graph is computationally heavy: from a graph with n vertices we move to one with $n + m$ vertices. The problem is that the number of mvs in a graph is exponential in the number of the graph's vertices [6]; hence, this increase from $O(n)$ to $O(n^2)$ is dramatic (indeed the number of mes in G is 4 whereas the number of mvs in G' is 12).

A problem similar to the one above is that of the enumeration of all cutsets in graphs (e.g. [19][3][15]). For example in [19] the authors constructs two efficient algorithms to enumerate all minimal st-cutset separating two specified vertices s and t in undirected graphs. Both the algorithms have time complexity $O((n + m)(\mu + 1))$, where μ denotes the number of st-cutsets in a given graph. Notice

that these publications consider only undirected graphs.

Hence, the notion of minimal edge separator is different from that of minimal vertex separator and from that of cutset. Indeed, an mvs (like a cutset) leads to connected components while a mes leads to the special subgraphs described above. Moreover also the enumeration of all mes is a bit different from that of all mvs. Indeed, as described below, it is not enough do a nodes-to-edges transformation for obtaining the corresponding mes enumeration.

Starting from the edges outgoing from s, the generation of all mes proceeds via edges substitution. If G is the graph under consideration and X is a minimal edge separator, for generating a new mes, the successor of a minimal edge separator with respect to any of its edges is defined. Such a set is computed by replacing the edge under consideration with its immediate successors (that are the edges outgoing from the last endpoint of the edge) and by removing all edges in the set under construction that have not one endpoint G_s^X and the other in G_t^X. This process is done for all edges in all mes and it ends when no new mes is generated (other than one consisting of the incoming edges in t). To ensure that all mes are enumerated, a level is associated to each minimal edge separator.

Following the notion of successor of a minimal edge separator, it is possible to build a *chain* of mes. A sequence of mes $X_0 \dots X_n$ is a chain if, for all i s.t. $0 \leq i < n$, $X_i \sqsubset X_{i+1}$. The order relation $\sqsubset$ is *edge set-coverage*. This relation compares two mes. If X and X' are two minimal edge separators, than X covers X' if every path from every edge in X' to t passes through X. Such a chain is *complete* if between X_i and X_{i+1} there are no other mes.

Like weakness, also edge-weakness can be detected checking only a chain of mes. If X is the first mes of a chain (i.e. the set of the edges outgoing from s), testing edge-weakness is done by generating the successor of X and by checking if such successor is *edge-critical*. A mes is edge-critical if there exists an *edge-critical path*, i.e. a path containing two edges of the same mes. Figure 1.1 (b) provides an example of edge-critical path. If the successor of X is edge-critical, the graph is edge-weak. If the successor of X is not edge-critical, then it is required a way to check if any of its edges is critical. For this purpose, for each edge e in the successor of X, the *e-minimal* mes is computed. An e-minimal mes is a mes s.t. e does not belong to none of the mes covering it. Hence, by choosing one of the edges of the e-minimal mes, a predecessor (that is like a successor but in backwards direction) of the successor of X is computed. If such predecessor is critical the cause is that e is critical (since, if e is critical, all e-minimal mes are e-critical) thus the graph is edge-weak; otherwise a new mes in the chain is generated and this process is repeated for the new mes.

Thanks to the implementation of the algorithm on edge-weakness, some tests on graphs have been conducted. The results of such tests show that it is possible

to note a linear behavior in the number of vertices and edges of the graph under consideration: with the increase of the edges in respect of vertices, the probability that a graph is edge-weak increases. Such a probability is 0 if the number of vertices and edges is the same and it becomes certain with the maximum amount of edges.

This dissertation is structured as follows. The second chapter provides a background to the main topic of this dissertation describing some basic definitions and properties of graphs. Then, the focus is moved on flow network and on the maximum flow problem. Finally the chapter ends with an introduction to edge-weakness. The third chapter is divided into three parts. The first one is preliminary. The second one is on the enumeration of all minimal edge-separators in graphs and the third one is on edge-weak graphs. For each section, it is shown the algorithm related to the problem handled by the same section. Chapter 4 provides the implementation of the algorithms in Chapter 3 and it shows the results of the tests conducted on the graphs.

Chapter 2

Basic Definitions on the Model

The aim of this chapter is to provide a background to the main topic of this dissertation. Therefore the following sections describe some basic definitions and properties of graphs which are substantial in the discussion. For this part, the references are [10] for *Graph Theory*, [11] and [16] for *Flow Networks*. Since edge-weakness is a problem whose origin lies in flow networks, to flow networks is dedicated more attention. This chapter ends with an introduction to edge-weakness, which is thoroughly discussed in Chapter 3.

2.1 Background on Graph Theory

A *directed graph* (or *di-graph*) G is a pair (V, E), where V is a finite set and E is a binary relation on V. V is the vertex set of G and its elements are called vertices. E is the edge set of G and its elements are called edges.

If (u, v) is an edge of a directed graph $G = (V, E)$, it is said that (u, v) is incident to u and v, or that it is an outgoing edge from u and an incoming edge in v. The set of the outgoing edges of a vertex v in G are denoted $out(v) = \{(v, a) \in E \mid a \in V\}$ and the set of incoming edges of a vertex v in G are denoted $in(v) = \{(a, v) \in E \mid a \in V\}$. A *source* (respectively, *sink*) is a vertex which is only incident with outgoing edges (respectively, incoming edges). Instead, a vertex with both incoming and outgoing edges is called an intermediate vertex.

In directed graphs, vertices have both outdegrees and indegrees. The *outdegree* of a vertex is the number of edges leading away from that vertex, and the *indegree* of a vertex is the number of edges leading to that vertex. Let v a vertex of di-graph $G = (V, E)$, the outdegree of v is denoted $deg^+(v) = |out(v)|$ and indegree of v is denoted $deg^-(v) = |in(v)|$.

A *directed path* [17] (also called *dipath*) is a path with the added restriction that all edges of the path must be directed in the same direction. A *path* from a

vertex a to a vertex b is a sequence of vertices $(v_0, v_1, \ldots, v_k)$ where $v_0 = a$ and $v_k = b$, s.t. (v_{i-1}, v_i) belongs to E for all $1 \leq i \leq k-1$. In such a case, the path has length k and it *contains* (or *passes through*) the vertices $v_0, v_1, \ldots, v_k$ and the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$. An a-b dipath is denoted $a \rightsquigarrow b$. Moreover, if $a = b$ then the dipath is a cycle. A di-graph is defined acyclic if it contains no cycles.

The *distance* between two vertices a and b in a di-graph is defined as the number of edges in a shortest directed path from a to b and it is denoted as $d(a, b)$.

A di-graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$; in such a case one writes $G' \subseteq G$. Hence, a subgraph of a di-graph is a subset of a di-graph's edges (and associated vertices) that constitutes a di-graph. A *spanning subgraph* [5] of G is a subgraph obtained by edge deletions only. If X is the set of deleted edges, this subgraph of G is denoted $G \setminus X$. In other words $G \setminus X$ is a subgraph whose vertex set is the entire vertex set of G and whose edge set is a subset of the edge set of G.

A di-graph $G = (V, E)$ is *strongly connected* if every pair of its vertices is strongly connected, that is if in G there exists at least one dipath between each pair of vertices; otherwise the graph is disconnected.

An *st-cut* is a is a partition of vertex set into two parts X and $\bar{X}$ defined with respect to two distinguished vertices s and t, such that $s \in X$ and $t \in \bar{X}$. Each cut defines a *cutset* that is the set of edges that have one endpoint in X and the other endpoint in $\bar{X}$.

2.2 Flow Networks

In this section we use the following convention: if φ is a function on edges (nodes) and X is a set of edges (nodes), then $\varphi(X) = \sum_{x \in X} \varphi(x)$.

A *Flow Network* is a directed graph $G = (V, E, s, t, c)$, where s and t are two special vertices respectively called *source* and *sink*, and c is a capacity function, $c : V^2 \to \mathbb{R}^+$, associating to every edge $(a, b) \in E$ a non-negative capacity $c(a, b)$, and a zero capacity to every $(a, b) \notin E$. The next definition describes the notion of *flow* in flow networks:

Definition 2.2.1 (Flow [11]). A flow on G is a function $f : V^2 \to \mathbb{R}^+$ satisfying the following three constraints:

1. *Capacity:* $f(a, b) \leq c(a, b) \quad \forall (a, b) \in V \times V$
2. *Conservation:* $f(in(a)) = f(out(a)) \quad \forall a \in V \setminus \{s, t\}$
3. *Antisymmetry:* $f(a, b) = -f(b, a) \quad \forall (a, b) \in V \times V$

The main issue in flow networks is knowing the amount of information that can pass from source to sink. To this end the value of flow is defined.

Definition 2.2.2 (Value of flow [11]). Given a flow f, $|f|$ denotes the amount of flow outgoing from the source: $|f| \triangleq f(out(s))$.

Moreover, it is well known that $f(out(s)) = f(in(t))$.

If every edge in the network has a capacity equal to zero, then nothing can pass on the network. In such a case the net has a zero flow.

Definition 2.2.3 (Zero flow [11]). A zero flow is a function $f_0 : V^2 \rightarrow \mathbb{R}^+$ s.t. $f_0(e) = 0$, $\forall e \in E$.

Diametrically opposed to zero flow is the maximum flow.

Definition 2.2.4 (Maximum flow [11]). A maximum flow f^* is a flow s.t. $|f^*| \triangleq max_f|f|$.

Since networks are pervasive, they arise in numerous application settings and in many forms. Surely, given a flow network one wants to monitor the objects passing; and surely, one wants to transfer these objects, without exceed the capacity of the edges in the network. For this purpose, it is defined the residual capacity of edges.

Definition 2.2.5 (Residual capacity [11]). The residual capacity $r(e)$ of an edge e w.r.t. a flow f is $r(e) \triangleq c(e) - f(e)$.

Definition 2.2.6 (Capacity of a path [11]). Given a path p in a network G, the capacity of p, $c(p)$, is the minimum capacity of its edges: $c(p) \triangleq min_{e \in p} c(e)$.

Definition 2.2.7 (Capacity of an st-cut [11]). Given an st-cut $(X, \bar{X})$ in a network G, the capacity of $(X, \bar{X})$ is: $c(X, \bar{X}) \triangleq c(x, \bar{x})$.

The maximum flow problem is one of the standard problems in flow networks. Such a problem arises in a wide variety of situations and in several forms. Indeed, the maximum flow problem occurs as a subproblem in the solution of other network issues and it also arises in a number of combinatorial applications. In particular, in this chapter the maximum flow problem is mentioned because it represents the starting point to study edge-weakness.

Definition 2.2.8 (Maximum flow problem [16]). In a capacited network, the maximum flow problem is the problem of sending as much flow as possible between two special vertices, a source vertex s and a sink vertex t, without exceeding the capacity of any edge.

To solving the maximum flow problem there are a number of algorithms. In [16] it is written that these algorithms are of two types: augmenting path algorithms and preflow-push algorithms.

Augmenting path algorithms are essentially based on the observation that whenever the network contains an augmenting path, it is possible to send additional flow from the source to the sink. An augmenting path is a path constructed by repeatedly finding a path of positive capacity from a source to a sink and then adding it to the flow. The algorithm proceeds by identifying augmenting paths and augmenting flows on these paths until the network contains no such path [16].

The correctness of augmenting path algorithms rests on the renowned *max-flow min-cut theorem* of network flows, that establishes an important correspondence between flows and cuts in networks.

Theorem 2.2.1 ([16])**.** *The maximum value of the flow, from a source node s to a sink node t, in a capacitated network, is equal to the minimum capacity among all st-cuts.*

As described in [11] the amount of flow passing through every cut is equal to the amount of flow passing from source vertex to sink vertex. Furthermore, as stated in Theorem 14.3 (always in [11]), a flow f is a maximum flow if and only if there exists a cut such that its capacity is $|f|$.

The downside of augmenting path algorithms is the computational limitation. In general in worst-case the computational complexity is $O(nmC)$, where $C = max_{e \in E} c(e)$. Moreover, for problems with irrational capacity data, augmenting path algorithms might not find an optimal solution.

Examples of algorithms employing the method of augmenting paths are the algorithm of *Ford and Fulkerson* and the algorithm of *Edmonds and Karp* whose pseudocodes are in [11].

Preflow-push algorithms work more efficiently than augmenting path algorithms, by augmenting flows not along a whole path but along single edges. In such a case it is defined the preflow as function, f, which satisfies the *capacity* constraint of flow (see Definition 2.2.1) and for which $excess_f(v) \geq 0$ for all $v \in V \setminus \{s, t\}$, where $excess_f(v) = f(in(v)) - f(out(v))$. A vertex v s.t. $excess_f(v) > 0$ is called *active*.

The basic operations of all preflow-push algorithms are *flow push* and *relabel*. The first operation is a function, *push*, which adds a value $q = min\{excess_f(v), c((v, w))\}$ to $f((v, w))$, if $excess_f(v) > 0$, $h(w) < h(v)$ and $(v, w) \in E$, where h is a function which assigns a non-negative integer to all v in V. The latter is a function, *relabel*, which increases by one $h(v)$ if $excess_f(v) > 0$, for all w s.t

$(v, w) \in E$, $h(w) \geq h(v)$. Hence, in general preflow-push algorithms terminate when the network contains no active node.

Examples of preflow-push algorithms are in [16].

2.3 Edge-Weakness

As described in the previous section, flow networks arise in a wide variety of situations. An example of such a type of situations are channels for transmitting informations. Here the network is modelled as a di-graph with two special vertices called source and sink, resp. s and t. At a given time, an amount of informations passes from s to t [7].

Ideally, one wants to transfer the maximum amount of informations from source to sink. However, in a distributed scenario, due to partial knowledge of the net, the maximum transfer of informations may not happen. Clearly, this is a form of inefficiency, since not all possible flow is delivered.

Definition 2.3.1 (Edge-Weak[6]). A graph is edge-weak if there exists a capacity assignment to edges such that the resulting flow network admits a non-maximum saturating flow.

The problem of Edge-Weakness in terms of Definition 2.3.1 is algorithmically undecidable. Indeed, since for each edge it is possible to assign any real, the capacity functions for a flow network are endeless and to check if the graph is edge-weak each of these capacity functions should be assigned to the net under consideration. For this purpose edge-weakness is addressed in a different way [6][7].

Anticipating the notion of *minimal edge-separator*:

Definition 2.3.2 (Edge separator, minimal edge-separator). An *edge separator* is any $X \subseteq E$ such that $G \backslash X$ is a graph where there is no path between s and t. An *edge-separator* is *minimal* if it does not properly contain any *edge-separator* and it is referred to as *mes*.

Edge-weakness is characterised as follows in [7]:

Theorem 2.3.1 ([7]). *G is edge-weak if and only if there exists a mes X and a path p touching it at least twice.*

Consider the network in Figure 2.1 (a); here the maximum amount of informations passing at a given time is 2. However, such a net admits a saturing flow of value 1 by sending all the flow along the path $p = s\ v_1\ v_2\ t$. Consider now the graph under the network as depicted in Figure 2.1 (b). The edges belonging to $\{(s, v_1), (v_2, t)\}$ constitute a minimal edge-separator. Since by Theorem 2.3.1 p touches those edges twice, the graph is edge-weak.

Figure 2.1: (a) A network with capacity assignment proving its weakness and (b) the edge-weak graph under network

Theorem 2.3.1, which graph-theoretically characterises the notion of edge-weakness, leads to the implementation of an algorithm to checking edge-weakness in polynomial time without having to use the notion of capacity function.

The purpose of the next chapter is to deepen this connection between edge-weakness and minimal edge-separators and to show how the theory legitimises the correctness of such polynomial algorithm.

Chapter 3

Checking Edge-Weakness

The chapter is divided into three parts. The first one is preliminary. The second one is on the enumeration of all minimal edge-separators in graphs and the third one is on edge-weak graphs.

For each section, it is shown the algorithm related to the problem handled by the same section. Such algorithms are presented with their pseudocodes, see the next chapter for their implementation.

3.1 Preliminaries

In this chapter we only consider directed simple st-graphs, that are directed graphs without self-loops and parallel edges, with a fixed source vertex s and sink vertex t. In the considered graphs each node belongs to at least one st-path. Throughout the sections, reference is made to a graph $G = (V, E)$ s.t. $|V| = n$, $|E| = m$. An additional assumption is that the source vertex has no incoming edges and the sink vertex has no outgoing edges e.g. $in(s) = out(t) = \emptyset$.

Before dealing with the main issue, it is necessary to state some definitions to which reference is made in subsequent sections.

Definition 3.1.1 (Immediate successors of an edge e). The immediate successors of $e = (a, b) \in E$ is the set of edges denoted by $next(e) \triangleq \{(b, v) \in E \mid v \in V\}$.

Definition 3.1.2 (Immediate predecessors of an edge e). The immediate predecessors of $e = (a, b) \in E$ is the set of edges denoted by $prev(e) \triangleq \{(v, a) \in E \mid v \in V\}$.

Definition 3.1.3 (Distance between source node and an edge). If $e = (x, y) \in E$, then $h(e) \triangleq d(s, x)$ is the distance from s to e.

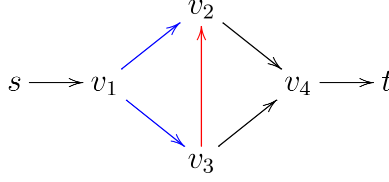


Figure 3.1: A simple graph to show the difference between edge-separator and minimal edge-separator. The set containing blue edges is a minimal edge-separator; the set containing coloured arrows is only an edge-separator.

3.2 Efficient enumeration of all minimal edge-separators in a graph

As written in the introduction of this chapter, [18] describes an efficient algorithm for enumerating all minimal a-b separators separating two given non-adjacent vertices a and b in an undirected connected simple graph. The algorithm requires $O(n^3 R_{a,b})$ time for solving this problem, where $R_{a,b}$ is the number of minimal a-b separators.

Choosing a and b as s and t respectively, via such an algorithm it is possible to enumerate all minimal vertex separators in graphs.

A problem similar to the one above is that of the enumeration of all minimal edge separators.

3.2.1 Definition and characterization of minimal edge-separator

An edge-separator, as Definition 2.3.2 sets out, is a set of edges of G for which all paths between s and t must pass. Such a set is minimal if the removal of any of its edges does not make it an edge-separator.

Figure 3.1 depicts a simple example of edge-separator and minimal edge-separator.

As mentioned in Chapter 3.1, this dissertation only considers directed graphs. Therefore, it is not possible to state that an edge-separator (or a minimal edge-separator) separates G in at least two disjoint connected components (and definitely not in at least two disjoint strongly connected components). Thus, let $X \subset E$ be a generic set of edges; then G_s^X and G_t^X are two subgraphs of $G \setminus X$ s.t. $G_s^X = \{(V_{X,s}, E_{X,s}) \mid \forall s \overset{p}{\rightsquigarrow} w \in G \setminus X,\, \forall v \in p,\, \forall e \in p \,.\, v \in V_{X,s} \wedge e \in E_{X,s}\}$ and $G_t^X = \{(V_{X,t}, E_{X,t}) \mid \forall w \overset{p}{\rightsquigarrow} t \in G \setminus X,\, \forall v \in p,\, \forall e \in p \,.\, v \in V_{X,t} \wedge e \in E_{X,t}\}$.

What follows is necessary to prove Lemma 3.2.2.

Lemma 3.2.1. *If X is a minimal edge-separator, then $V_{X,s}$ and $V_{X,t}$ are a bipartition of V.*

Proof. Looking for a contradiction, assume that X is a minimal edge-separator and $v \in V$ then two cases are possible:

(i) If $v \notin V_{X,s}$ and $v \notin V_{X,t}$ then in X there must be two edges (x, v) and (v, y) contradicting minimality of X.

(ii) If $v \in V_{X,s}$ and $v \in V_{X,t}$ then X would not be a separator. Indeed, $\exists s \rightsquigarrow v \in G_s^X$ and also $\exists v \rightsquigarrow t \in G_t^X$. Hence $\exists s \rightsquigarrow v \rightsquigarrow t \in G \setminus X$.

Thus for every vertex v in V, $v \in V_{X,s} \dot{\vee} v \in V_{X,t}$ □

Definition 3.2.1 (Isolated sets of edges). For any $X \subset E$, it is possible to define the isolated sets of edges of X, denoted by $\mathcal{I}_t(X)$ and $\mathcal{I}_s(X)$, as follows: $\mathcal{I}_t(X) \triangleq \{e \in X \mid next(e) \cap E_{X,t} = \emptyset\}$ and $\mathcal{I}_s(X) \triangleq \{e \in X \mid prev(e) \cap E_{X,s} = \emptyset\}$.

Now it is possible to characterise the notion of minimal edge-separator:

Lemma 3.2.2. *Let X be an edge-separator of the $st-$graph $G = (V, E)$. Then X is a minimal edge-separator of G if and only if every edge in X has the first end in G_s^X and the last end in G_t^X.*

Proof. (Only if) Looking for a contradiction, assume that X is a minimal edge-separator and in X there exists an edge $e = (a, b)$ that has not the first end in G_s^X and the last end in G_t^X. Then three cases are possible:

(i) If $a \in G_s^X$ and $b \notin G_t^X$ then all paths from s to t, passing through e, must pass also through another edge in X since by Lemma 3.2.1, $b \in G_s^X$ (since it does not belong to G_t^X). Hence deleting e from X would still yield a separator (against minimality of X).

(ii) If $a \notin G_s^X$ and $b \in G_t^X$ the same of point (i) happens.

(iii) If $a \notin G_s^X$ and $b \notin G_t^X$ then all paths from s to t, passing through e, must pass also at least through another two edges in X since $a \notin G_s^X$ and $b \notin G_t^X$. Hence deleting e from X would still yield a separator (against minimality of X).

(If) Looking for a contradiction, assume that X is not a minimal edge-separator. Then there exists an edge e which can be removed from X. Since by hypothesis e has the first end in G_s^X and the last end in G_t^X, in G there exists a path from s to t not passing through $X \setminus \{e\}$. Hence X cannot be an edge-separator without e. □

3.2.2 Definition and characterization of successor and predecessor of a minimal edge-separator

Let $X = out(s)$, from Lemma 3.2.2 it is clear that X is a minimal edge-separator. A new minimal edge-separator can be generated from X by replacing any edge e in X with all its immediate successors and deleting all edges in the isolated set of edges $\mathcal{I}_t$ (the same can be done in backwards from $X = in(t)$). Hence, each X may generate at most $|X|$ new minimal-edge separators. This leads to reason in an incremental manner to enumerate all mes in G. To this aim, it is defined the successor of a minimal edge-separator and, similarly, the predecessor of a minimal edge-separator.

Definition 3.2.2 (Successor of a minimal edge-separator with respect to any of its edges). The successor X_e of a mes X with respect to an edge $e \in X$, is the mes defined by the following equation:

$$X_e \triangleq ((X \setminus \{e\}) \cup next(e)) \setminus \mathcal{I}_t((X \setminus \{e\}) \cup next(e)) \tag{3.1}$$

Lemma 3.2.3. *Let X be a minimal edge-separator and $e \in X$. If $e \notin in(t)$ then X_e, defined by the equation* (3.1)*, is a minimal edge-separator and $X_e \neq X$.*

Proof. Since by hypothesis X is a minimal edge-separator then also $X' = (X \setminus \{e\}) \cup next(e)$, for any $e \in X$ s.t. $e \notin in(t)$, is an edge-separator. Indeed by construction X' is made up from X's edges (except for e) and e's immediate successors. To make X' minimal, it is necessary to remove from X' those edges that have not the first end in $G_s^{X'}$ and the last end in $G_t^{X'}$ (by Lemma 3.2.2), de facto these edges are $\mathcal{I}_t((X \setminus \{e\}) \cup next(e))$. □

Definition 3.2.3 (Predecessor of a minimal edge-separator with respect to any of its edges). The predecessor X^e of a mes X with respect to an edge $e \in X$, is the mes defined by the following equation:

$$X^e \triangleq ((X \setminus \{e\}) \cup prev(e)) \setminus \mathcal{I}_s((X \setminus \{e\}) \cup prev(e)) \tag{3.2}$$

Lemma 3.2.4. *Let X be a minimal edge-separator and any $e \in X$. If $e \notin out(s)$ then X_e, defined by equation* (3.2)*, is a minimal edge-separator and $X^e \neq X$.*

Proof. The same as in *Proof* of Lemma 3.2.3. □

Consider the graph depicted in Figure 3.1. Let X be the mes $\{(v_1, v_2), (v_3, v_2), (v_3, v_4)\}$, the successor of X with respect to the edge $e = (v_1, v_2)$ is the mes $X_e = \{(v_2, v_4), (v_3, v_4)\}$. Indeed, $next(e) = \{(v_2, v_4)\}$, thus $(X \setminus \{e\}) \cup next(e) = \{(v_3, v_2), (v_3, v_4), (v_2, v_4)\}$ and $\mathcal{I}_t((X \setminus \{e\}) \cup next(e)) = \mathcal{I}_t(\{(v_3, v_2), (v_3, v_4), (v_2, v_4)\}) = \{(v_3, v_2)\}$. Hence $X_e = \{(v_3, v_2), (v_3, v_4), (v_2, v_4)\} \setminus \{(v_3, v_2)\} = \{(v_2, v_4), (v_3, v_4)\}$.

Similarly, $X^e = ((X \setminus \{e\}) \cup prev(e)) \setminus \mathcal{I}_s((X \setminus \{e\}) \cup prev(e)) = ((\{(v_1, v_2), (v_3, v_2), (v_3, v_4)\} \setminus \{(v_1, v_2)\}) \cup \{(s, v_1)\}) \setminus \mathcal{I}_s(((\{(v_1, v_2), (v_3, v_2), (v_3, v_4)\} \setminus \{(v_1, v_2)\}) \cup \{(s, v_1)\})) = \{(v_3, v_2), (v_3, v_4), (s, v_1)\} \setminus \{(v_3, v_2), (v_3, v_4)\} = \{(s, v_1)\}$.

When $e \in in(t)$, since e cannot be replaced with its immediate successors (because edges entering into t have no successors), the removal of the edges in X_e from G cannot block paths from s to t via e. The same reasoning is made for X^e if $e \in out(s)$. Thus X_e (or X^e) is not an edge-separator. So:

Lemma 3.2.5. *Let X be a minimal edge-separator and $e \in X \cap in(t)$; then X_e is not an edge-separator.*

Proof. If $e \in in(t)$ then $next(e) = \emptyset$, so formula (3.1) is reduced to $(X \setminus \{e\}) \setminus \mathcal{I}_t(X \setminus \{e\})$. Since X is a mes then, because of Lemma 3.2.2, $\exists s \rightsquigarrow a \in G$ and $\exists b \rightsquigarrow t \in G$, $a \in V_{X,s}$ and $b \in V_{X,t}$, where $e = (a, b)$. Hence $\exists s \rightsquigarrow a \rightarrow b \rightsquigarrow t \in G \setminus (X \setminus \{e\})$; this means that $X \setminus \{e\}$ is not an edge-separator thus also X_e is not an edge-separator (since $X_e \subseteq X \setminus \{e\}$). □

Lemma 3.2.6. *Let X be a minimal edge-separator and $e \in X \cap out(s)$; then X^e is not an edge-separator.*

Proof. The same as in *Proof* of Lemma 3.2.5. □

The next lemma shows that formule (3.1) and (3.2) are inverses of each other. Indeed, by applying (3.2) to a mes X and then by applying (3.1) to the result, we obtain X. Moreover, the following lemma is crucial to prove Theorem 3.2.1:

Lemma 3.2.7. *$\forall X$ minimal edge-separator $\forall e \in X$ and $\forall f \in prev(e)$, it holds that $(X^e)_f = X$.*

Proof. Let $e = (a, b) \in X$ and $out(a) = A \uplus B$, where $A = out(a) \cap X$ and $B = out(a) \setminus X$. Because of formula (3.2), the predecessor of X with respect to e is $X^e = ((X \setminus \{e\}) \cup prev(e)) \setminus \mathcal{I}_s((X \setminus \{e\}) \cup prev(e))$. So from $X \cup prev(e)$ must be removed those edges that have not immediate predecessors in $G_s^{(X \setminus \{e\}) \cup prev(e)}$. These edges are nothing more than the successors of the predecessors of e in X. Since $prev(e) = \{(w, a) \in E \mid w \in V\}$ the edges to eliminate are A.

Moreover only these edges are deleted. Indeed, looking for a contradiction let $e' = (a', b') \in X \setminus A$ (so $a' \neq a$) be an edge to remove, then X would not be minimal. Indeed, the only cause to remove e' from X is that, by adding $prev(e) = in(a)$ to X, every path passing through e' passes also through $prev(e)$. Since $a' \neq a$, X has to contain some edges linking $prev(e)$ to e' against minimality of X (because $X \setminus \{e\}$ would still be a separator). Hence $X^e = ((X \setminus \{e\}) \cup prev(e)) \setminus A$.

Let $f = (c, a) \in prev(e)$. Compute $(X^e)_f = [(X^e \setminus \{f\}) \cup next(f)] \setminus \mathcal{I}_t([(X^e \setminus \{f\}) \cup next(f)])$. Let $X' = [(X^e \setminus \{f\}) \cup out(a)]$, since $next(f) = out(a)$ then

$(X^e)_f = X' \setminus \mathcal{I}_t(X')$. Clearly, by construction, $((X \setminus \{e\}) \cup prev(e)) \setminus \{f\} \subseteq X'$ and so $[((X \setminus \{e\}) \cup prev(e)) \setminus \{f\}] \setminus \mathcal{I}_t(X') \subseteq X' \setminus \mathcal{I}_t(X')$.

Since X is a minimal edge-separator, it must be $\forall e' \in B$, $\forall p$ path s.t. $e' \in p$ $\exists e'' \in X$ s.t. $e'' \in p$. Indeed, looking for a contradiction suppose $\exists e' = (a, b') \in B$ s.t. $e' \notin \mathcal{I}_t(X')$. This means that $b' \in G_t^{X'}$ i.e. $\exists b' \rightsquigarrow t \in G \setminus X'$, because of Definition 2.3.2. Given that $X' = (((X \setminus \{e\}) \cup prev(e)) \setminus \{f\}) \cup B$ then $(b' \rightsquigarrow t) \cap X = \emptyset$. In addition, because $a \in G_s^X$ then $\exists s \rightsquigarrow a \in G \setminus X$ (always by Definition 2.3.2). Hence $s \rightsquigarrow a \rightarrow b' \rightsquigarrow t \in G \setminus X$, contradicting the assumption that X is a mes. So it must be $B \subset \mathcal{I}_t(X')$. Moreover, because $e \in out(a)$ then $e \in X'$ and $prev(e) \subset \mathcal{I}_t(X')$.

Furthermore $\mathcal{I}_t(X')$ contains only the edges of $prev(e) \cup B$. Indeed, looking for a contradiction suppose $\exists e' \in X$ s.t. $e' \in \mathcal{I}_t(X')$. Since $X' = [(X^e \setminus \{f\}) \cup out(a)]$, the only reason to remove e' from X' is that, by adding $out(a)$ to $X^e \setminus \{f\}$, every path passing through e' passes also through $out(a)$. Since $e' \notin B$, X has to contain some edges linking A to e' against minimality of X (because $X \setminus \{e\}$ would still be a separator).

Hence because $\mathcal{I}_t(X') = prev(e) \cup B$, $[((X \setminus \{e\}) \cup prev(e)) \setminus \{f\}] \setminus \mathcal{I}_t(X') = X$. Since X is a mes and even $(X^e)_f$ is a mes (by Lemma 3.2.3 and Lemma 3.2.4) then $X \not\subset (X^e)_f$ so it must be $X = (X^e)_f$. □

Consider the graph depicted in Figure 3.1. Let X be the mes $\{(v_1, v_2), (v_3, v_2), (v_3, v_4)\}$, $e = (v_3, v_2)$ and so $X^e = \{(v_1, v_2), (v_1, v_3)\}$. Let $f = (v_1, v_3) \in prev(e)$, then $(X^e)_f = [(\{(v_1, v_2), (v_1, v_3)\} \setminus \{(v_1, v_3)\}) \cup \{(v_3, v_2), (v_3, v_4)\}] \setminus \emptyset = \{(v_1, v_2), (v_3, v_2), (v_3, v_4)\} = X$.

3.2.3 Enumerating all minimal edge-separators

The approach explained in the previous section shows that starting from $out(s)$ it is possible to generate each minimal edge-separator. This process continues until every edge, in the preexisting mes, does not belong to $in(t)$.

It makes sense to introduce the concept of *level* of minimal edge-separator. Level 0 contains only one separator $out(s)$; the following levels contain the minimal edge-separators generated from some other mes in the same level or in the previous one.

What follows provides an inductive definition of level of a minimal edge-separator:

Definition 3.2.4 (Level of a minimal edge-separator). The level of a minimal edge-separator is defined as follows:

$lev(out(s)) = 0$

$$lev(X_e) = \begin{cases} lev(X) + 1 & \text{if } e \in X \text{ and } h(e) = lev(X) \\ lev(X) & \text{if } e \in X \text{ and } h(e) < lev(X) \end{cases}$$

From Definition 3.2.4 it seems clear that the notion of level of a mes X is really bound to the notion of maximum distance between s and the edges in X. Indeed:

Lemma 3.2.8. *If $lev(X) = k$, then $h(e) \leq k$, $\forall e \in X$.*

Proof. The proof proceeds by induction on k.

(i) If $k = 0$ then $lev(X)$=0. Hence, since $X = out(s)$ then $h(e) = 0$, $\forall e \in X$.

(ii) If $k > 0$ then $lev(X) > 0$. Hence let $X = Y_{e'}$, for some $e' \in Y$. Two cases are possible:

1. If $e \in Y$, then by induction $h(e) \leq lev(Y) \leq lev(X)$.
2. If $e \notin Y$, then $e \in next(e')$. Two cases are possible:
   a) If $h(e') < lev(Y)$ then $h(e) \leq lev(Y) \leq lev(X)$.
   b) If $h(e') = lev(Y)$ then $h(e) = lev(Y) + 1 = lev(X)$. □

Always refering the graph in Figure 3.1 and its mes $X = \{(v_1, v_2),\ (v_1, v_3)\}$ having $lev(X) = 1$, then it is possible to compute the mes $X_{(v_1,v_3)}$ s.t. $X_{(v_1,v_3)} = \{(v_1, v_2),\ (v_3, v_2),\ (v_3, v_4)\}$. From Definition 3.2.4 the level of $X_{(v_1,v_3)}$ is 2, since $h((v_1, v_3) = 1 = lev(X)$. Moreover Lemma 3.2.8 is respected given that $h((v_1, v_2)) = 1$, $h((v_3, v_2)) = 2$ and $h((v_3, v_4)) = 2$.

Because to every minimal edge-separator can be assigned a level, it makes sense to partition minimal edge-separators according to their level.

Definition 3.2.5 (Clustering of minimal edge-separator according to level). L_i is the set of minimal edge-separator at level i i.e., $L_i \triangleq \{X \subseteq E \mid X$ is a mes $\wedge\ lev(X) = i\}$.

Finally, starting from Definition 3.2.5 the following theorem illustrates that every minimal edge-separator can be placed in some set of mes in which every minimal edge-seprator has the same level. This means that the union of all L_i, $0 \leq i \leq d(t) - 1$, is equal to the set of all mes in G.

Theorem 3.2.1. *If X is a mes, then $\exists i$ s.t. $X \in L_i$.*

Proof. For any minimal edge-separator X in G, it is possible to compute a value $h(X)$. Clearly $h(X)$ grows in the transition from X to X_e, $\forall e \in X$.

The proof proceeds by induction on $h(X)$.

(i) If $h(X) = 0$ then $\forall e \in X$, $h(e) = 0$. Hence $X \subseteq out(s)$. Since by assumption X is a mes, then X must contain *all* edges of $out(s)$; this means that $X = out(s)$. Therefore, because of Definition 3.2.4 and Definition 3.2.5, $X \in L_0$.

(ii) If $h(X) > 0$ then there exists $e \in X$ s.t. $h(e) > 0$. Because of Lemma 3.2.7 $X = (X^e)_f$, for some $f \in prev(e)$. Let $Y = X^e$, because $X = Y_f$ then $h(Y) < h(X)$; thus by induction $Y \in L_i$ for some i s.t. $0 \leq i \leq d(t) - 1$. Hence the level of X may be calculated as in Definition 3.2.4. Indeed, by Lemma 3.2.8 if $h(f) = lev(Y)$ then $X \in L_{i+1}$ otherwise $X \in L_i$. □

3.2.4 An Algorithm for enumerating all minimal edge-separators

Based on the approach described above, the algorithm for generating all minimal edge-separators is presented below.

The algorithm enumerates all minimal edge-separators according to Theorem 3.2.1. Each minimal edge-separator is generated correctly by Lemma 3.2.3. Starting from $out(s)$, all edges of all minimal edge-separators are considered: for each mes it is determined the successor with respect to everyone of its edges. The minimal edge-separators are distinct since duplicates are excluded by Step 11. The generation proceeds until the mes in the last level contains $in(t)$.

Algorithm 1 Generating all minimal edge-separators

Input: An st-graph $G = (V, E)$
function $st{-}minimal\ edge\text{-}separators(G)$

1: $j \leftarrow 0$;
2: $L_j \leftarrow \{out(s)$;
3: **while** $L_j \neq \emptyset$ **do**
4: $\quad L_{j+1} \leftarrow \emptyset$;
5: $\quad$ **for all** $X \in L_j$ **do**
6: $\quad\quad$ **for all** $e \in X \setminus in(t)$ **do**
7: $\quad\quad\quad$ Compute the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$;
8: $\quad\quad\quad$ **if** $G_t^{(X \setminus \{e\}) \cup next(e)} \neq \emptyset$ **then**
9: $\quad\quad\quad\quad$ Compute $\mathcal{I}_t((X \setminus \{e\}) \cup next(e))$;
10: $\quad\quad\quad\quad$ $X' = ((X \setminus \{e\}) \cup next(e)) \setminus \mathcal{I}_t((X \setminus \{e\}) \cup next(e))$;
11: $\quad\quad\quad\quad$ **if** $X' \notin \bigcup_{i=0}^{j+1} L_i$ **then**
12: $\quad\quad\quad\quad\quad$ **if** $h(e) = j$ **then**
13: $\quad\quad\quad\quad\quad\quad$ $L_{j+1} \leftarrow L_{j+1} \cup \{X'\}$;
14: $\quad\quad\quad\quad\quad$ **else**
15: $\quad\quad\quad\quad\quad\quad$ $L_j \leftarrow L_j \cup \{X'\}$;
16: $\quad$ $j \leftarrow j + 1$;

The correctness of this algorithm is proved by the results in the section above; its complexity is now discussed.

Analysis

In Step 2 the determination of $out(s)$ takes time $O(deg^+(s))$ hence $O(n)$.

In Step 7 one needs to compute $G_t^{(X\setminus\{e\})\cup next(e)}$ which can be done by building the incidence matrix with $q < n$ rows and $p < m$ columns.

Step 9 takes time $O(m)$: if the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ is rapresented using the incidence matrix, then checking if the edge e has an immediate successor in $G_t^{(X\setminus\{e\})\cup next(e)}$ can be done by scrolling one row of the matrix.

The total number of edge-separators is $O(2^m)$; if the union of L_j, $0 \leq j \leq d(t) - 1$, consists of a minimal size expansion tree, then Step 11 costs $O(m^2)$ as in Lemma 5 in [18].

The loop at Step 6 is executed at most m times. Therefore, let μ the number of minimal edge-separators in G, then function *minimal edge-separators* costs $O(m^3\mu)$, or alternatively $O(n^6\mu)$ (since the maximum number of edges in a simple directed graph is $n(n-1)/2$).

An example of Algorithm 1 execution

Let G be the graph in Figure 3.2 (a).

- Initialization and 1st iteration of *while*:

1. In the beginning $j \leftarrow 0$ hence $L_j = L_0 \leftarrow \{out(s)\} = \{\{(s, v_1), (s, v_2), (s, v_3)\}\}$.

2. Since L_0 contains only one mes that is different to *in*(t), the *while* is executed.

   a) For $e = (s, v_1)$ it is computed the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ depicted in Figure 3.2 (b).

   b) $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\}$ is produced.

   c) Since X' is not present in $L_{j+1} = L_1$ and $h(e) = 0$, X' is added to L_1.

   d) For $e = (s, v_2)$ it is computed the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ depicted in Figure 3.2 (c).

   e) $X' \leftarrow \{(v_2, v_3), (v_2, v_5), (s, v_1), (s, v_3)\}$ is produced.

   f) Since X' is not present in $L_{j+1} = L_1$ and $h(e) = 0$, X' is added to L_1.

   g) For $e = (s, v_3)$ it is computed the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ depicted in Figure 3.2 (d).

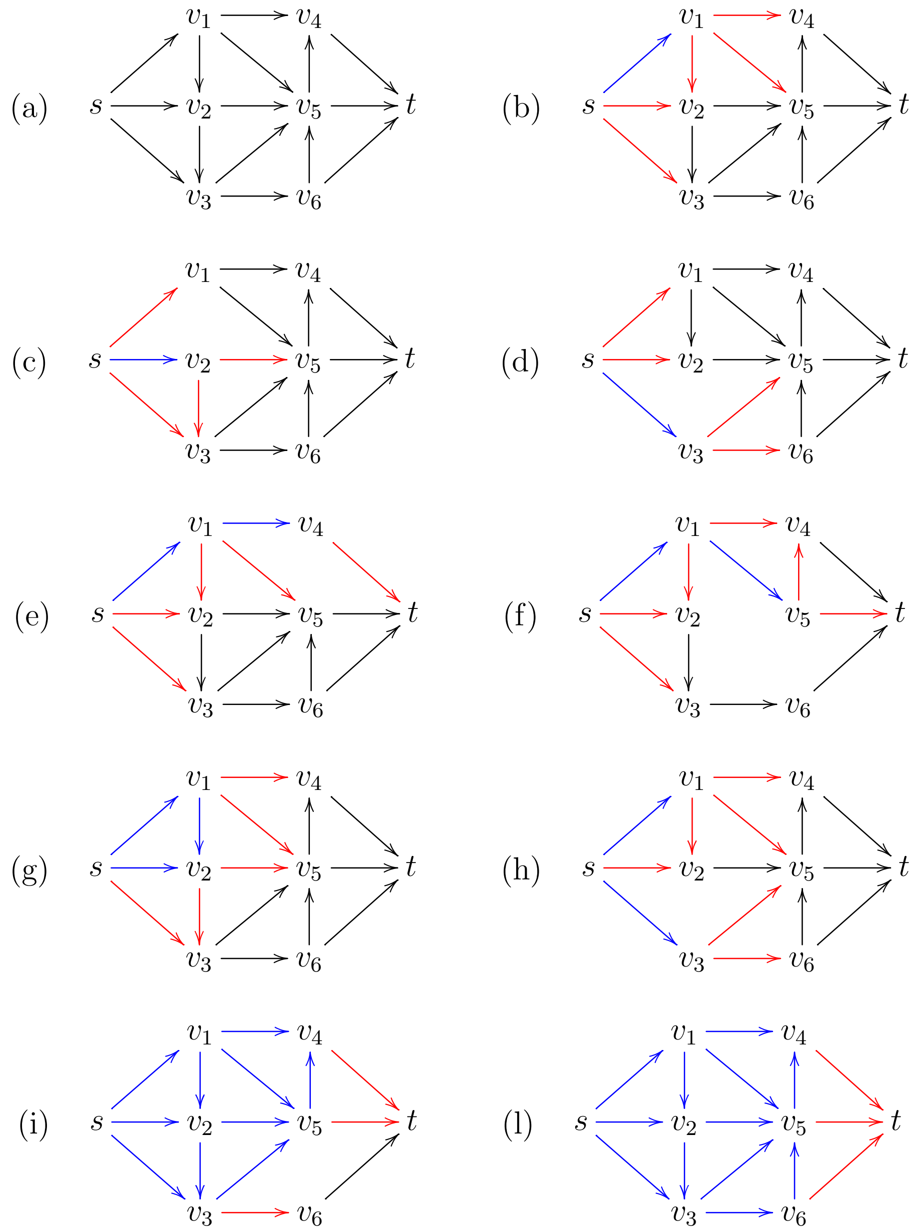


Figure 3.2: Execution of Algorithm 1 on graph (a). In blue the edges of G_s^X, in black the edges of G_t^X, in red the edges of the mes. The edges not in G_s^X, G_t^X or in the mes are not shown.

h) $X' \leftarrow \{(v_3, v_5), (v_3, v_6), (s, v_1), (s, v_2)\}$ is produced.

i) Since X' is not present in $L_{j+1} = L_1$ and $h(e) = 0$, X' is added to L_1.

j) Now L_1 contains three mes: $\{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\}$, $\{(v_2, v_3), (v_2, v_5), (s, v_1), (s, v_3)\}$ and $\{(v_3, v_5), (v_3, v_6), (s, v_1), (s, v_2)\}$. The value of j is increased.

- 2nd iteration of *while*:
Since L_1 contains a mes different to $in(t)$, *while* is executed.

1. For $X = \{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\}$:

a) For $e = (v_1, v_4)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (e).

b) $X' \leftarrow \{(v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3), (v_4, t)\}$ is produced.

c) Since X' is not present in $L_{j+1} = L_2$ and $h(e) = 1$, X' is added to L_2.

d) For $e = (v_1, v_5)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (f).

e) $X' \leftarrow \{(v_1, v_4), (v_1, v_2), (s, v_2), (s, v_3), (v_5, v_4), (v_5, t)\}$ is produced.

f) Since X' is not present in $L_{j+1} = L_2$ and $h(e) = 1$, X' is added to L_2.

g) For $e = (v_1, v_2)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (g).

h) $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (s, v_3), (v_2, v_3), (v_2, v_5)\}$ is produced.

i) Since X' is not present in $L_{j+1} = L_2$ and $h(e) = 1$, X' is added to L_2.

j) For $e = (s, v_2)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (g).

k) $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (s, v_3), (v_2, v_3), (v_2, v_5)\}$ is produced. Note that (v_1, v_2) is not in X' because its immediate successors are in X'.

l) Since X' is already present in $L_{j+1} = L_2$, it is not added.

m) For $e = (s, v_3)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (h).

n) $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (v_3, v_6), (v_3, v_5)\}$ is produced.

o) Since $h(e) = 0$ and X' is not present in $L_j = L_1$, X' is added to L_1.

2. The same procedure is applied for the other mes in L_1.

3. At the end: $L_1 = \{\{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\},$
$\{(v_2, v_3), (v_2, v_5), (s, v_1), (s, v_3)\},$
$\{(v_3, v_5), (v_3, v_6), (s, v_1), (s, v_2)\},$
$\{(s, v_2), (v_1, v_2), (v_1, v_4), (v_1, v_5), (v_3, v_5), (v_3, v_6)\},$
$\{(s, v_3), (v_2, v_3), (v_2, v_5), (v_1, v_4), (v_1, v_5)\}\}$

$L_2 = \{\{(v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3), (v_4, t)\},$
$\{(v_1, v_4), (v_1, v_2), (s, v_2), (s, v_3), (v_5, v_4), (v_5, t)\},$
$\{(v_2, v_3), (s, v_1), (s, v_3), (v_5, v_4), (v_5, t)\}$
$\{(v_2, v_5), (s, v_1), (v_3, v_6), (v_3, v_5)\}$
$\{(v_3, v_5), (s, v_1), (s, v_2), (v_6, v_5), (v_6, t)\}$
$\{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (v_3, v_5), (v_6, v_5), (v_6, t)\},$
$\{(v_5, v_4), (s, v_1), (v_3, v_6), (v_5, t)\}$
$\{(s, v_3), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_2, v_5)\}$
$\{(s, v_2), (v_1, v_2), (v_1, v_5), (v_3, v_5), (v_3, v_6), (v_4, t)\}$
$\{(v_1, v_4), (v_1, v_5), (v_3, v_5), (v_3, v_6), (v_2, v_5)\}$
$\{(v_1, v_4), (v_5, v_4), (v_3, v_6), (v_5, t)\}\}.$

- 3th iteration of *while*:
Since L_2 contains a mes different to *in*(t), *while* is executed.

1. For $X = \{(v_1, v_4), (v_5, v_4), (v_5, t), (v_3, v_6)\}$:

   a) For $e = (v_1, v_4)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (i).

   b) $X' \leftarrow \{(v_4, t), (v_5, t), (v_3, v_6)\}$ is produced.

   c) Since $h(e) = 1$ and X' is not present in $L_j = L_2$, X' is added to L_2.

   d) For $e = (v_5, v_4)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (i).

   e) $X' \leftarrow \{(v_4, t), (v_5, t), (v_3, v_6)\}$ is produced.

   f) Since X' is already present in $L_j = L_2$, X' is not added.

2. For $X = \{(v_3, v_6), (v_4, t), (v_5, t)\}$:

a) For $e = (v_3, v_6)$ it is computed the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ depicted in Figure 3.2 (l).

b) $X' \leftarrow \{(v_4, t), (v_5, t), (v_6, t)\}$ is produced.

c) Since X' is not present in $L_j = L_2$, X' is added to L_2.

3. At the end: $L_2 = \{\{(v_1, v_4), (v_1, v_5), (v_2, v_5), (v_2, v_3), (s, v_3)\},$
$\{(v_1, v_2), (v_4, t), (v_1, v_5), (s, v_3), (s, v_2)\},$
$\{(v_1, v_2), (v_5, v_4), (v_5, t), (v_1, v_4), (s, v_3), (s, v_2)\},$
$\{(s, v_1), (v_6, t), (s, v_2), (v_6, v_5), (v_3, v_5)\},$
$\{(s, v_1), (v_5, v_4), (v_5, t), (v_3, v_6)\},$
$\{(s, v_1), (v_5, v_4), (v_5, t), (v_2, v_3), (s, v_3)\},$
$\{(v_1, v_4), (v_1, v_5), (v_3, v_6), (v_2, v_5), (v_3, v_5)\},$
$\{(v_1, v_2), (v_4, t), (v_1, v_5), (v_3, v_6), (s, v_2), (v_3, v_5)\},$
$\{(v_5, v_4), (v_5, t), (v_3, v_6), (v_1, v_4)\},$
$\{(v_1, v_2), (v_6, t), (v_1, v_4), (v_1, v_5), (s, v_2), (v_6, v_5), (v_3, v_5)\},$
$\{(s, v_1), (v_6, t), (v_2, v_5), (v_6, v_5), (v_3, v_5)\},$
$\{(v_4, t), (v_1, v_5), (v_2, v_3), (v_2, v_5), (s, v_3)\},$
$\{(v_5, v_4), (v_5, t), (v_1, v_4), (v_2, v_3), (s, v_3)\},$
$\{(v_1, v_2), (v_4, t), (v_5, t), (s, v_3), (s, v_2)\},$
$\{(v_4, t), (v_1, v_5), (v_3, v_6), (v_2, v_5), (v_3, v_5)\},$
$\{(v_6, t), (v_1, v_4), (v_1, v_5), (v_2, v_5), (v_6, v_5), (v_3, v_5)\},$
$\{(v_1, v_2), (v_4, t), (v_6, t), (v_1, v_5), (s, v_2), (v_6, v_5), (v_3, v_5)\},$
$\{(v_5, v_4), (v_5, t), (v_6, t), (v_1, v_4)\},$
$\{(v_4, t), (v_6, t), (v_1, v_5), (v_2, v_5), (v_6, v_5), (v_3, v_5)\},$
$\{(v_4, t), (v_5, t), (v_6, t)\},$
$\{(s, v_1), (v_5, v_4), (v_5, t), (v_6, t)\},$
$\{(v_4, t), (v_5, t), (v_3, v_6)\},$
$\{(s, v_3), (v_5, t), (v_2, v_3), (v_4, t)\}\}$
Hence, $\{\{(v_4, t), (v_5, t), (v_6, t)\}\} = in(t) \subseteq L_2$. Thus since $L_3 = \emptyset$ the loop ends and the algorithm terminates.

3.3 Edge-Weakness

3.3.1 Characterization of Edge-Weakness

As described in Chapter 2 the downside of Definition 2.3.1 is that it describes the problem of edge-weakness in an undecidable way.

Thanks to Theorem 2.3.1 in [7], edge-weakness can be detected without recourse to an algorithm considering capacity-to-edge assignment. So, stemming

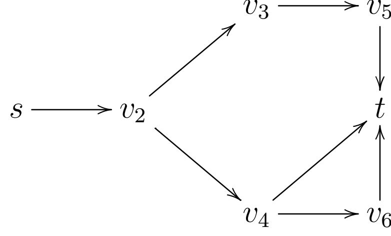


Figure 3.3: A simple graph to show a complete chain of mes

from Theorem 2.3.1, a trivial algorithm to determine if a graph is edge-weak is to generate all mes and, for each of them, check if there exists a path touching the mes twice.

Despite the number of mes in a graph can be exponential, the following sections prove that it is enough to examine at most a complete chain of mes that is linear in the size of the graph.

By Theorem 2.3.1 it makes sense to define a terminology to identify these main ingredients. For this purpose the following definition sets out the notion of *edge-critical path*, *critical edge* and *critical mes*.

Definition 3.3.1 (Edge-critical path, critical edge, critical mes). A path $a \rightsquigarrow b$ in G is *edge-critical* if there exists a mes X such that $(a, v_1), (v_2, b) \in X$ where $(a, v_1), (v_2, b)$ are both edges in $a \rightsquigarrow b$. In such a case $(v_2, b) = e$ is a *critical edge* and X is a *critical* mes (or *e-critical* mes).

Consider the graph in Figure 2.1 (b) and its path $p = s\ v_1\ v_2\ t$. Clearly p is an edge-critical path because it touches the mes $X = \{(s, v_1), (v_2, t)\}$ twice, (v_2, t) is a critical edge and X is a critical mes.

3.3.2 Edges set-coverage and complete chain of mes

Consider the relations defined below:

Definition 3.3.2 (Edge-coverage). An edge e is covered by a set of edges H, written $e \sqsubseteq H$, if all paths $v \rightsquigarrow t$ with e as first edge in the path, touch at least an edge $e' \in H$.

Definition 3.3.3 (Edges set-coverage). A set of edges H' is covered by a set of edges H, written $H' \sqsubseteq H$, if $e \sqsubseteq H, \forall e \in H'$.

Definition 3.3.4 (Edge-precedence). A set of edges H precedes an edge e, written $H \preceq e$, if all paths $s \rightsquigarrow v$ with e as last edge in the path, touch at least an edge $e' \in H$.

Definition 3.3.5 (Edges set-precedence). A set of edges H precedes a set of edges H', written $H \preceq H'$, if $H \preceq e, \forall e \in H'$.

Lemma 3.3.1. *The set of mes is a partially order set w.r.t $\sqsubseteq$ and $\preceq$.*

Proof. Clearly $\sqsubseteq$ is reflexive. The relation is antisymmetric; looking for a contradiction, let X and X' two mes and assume that if $X \sqsubseteq X' \wedge X' \sqsubseteq X$ then $X \neq X'$. By contradiction there exists an edge e s.t. $e \in X \setminus X'$.
Consider those two propositions:

Prop. 1: if $e \in X \cap in(t) \wedge X \sqsubseteq Y$ then $e \in Y$.

Prop. 2: if X is a mes then $\forall e \in X$ $\exists p$ s.t. p is a simple st-path and $p \cap X = \{e\}$.

Obviously, Prop. 1 is true since if $e \in in(t)$ then $e = (v, t)$, $v \in V$. Hence $e \sqsubseteq Y$ only if $e \in Y$. Moreover, Prop. 2 is true since, otherwise, X would not be a minimal edge-separator but only an edge-separator.
By Prop. 1, $e \notin in(t)$. Because of Prop. 2, there exists a simple st-path p touching X only in e. Let $p' = e \rightsquigarrow t$ where $e' \in p$, $\forall e' \in p'$. Since $X \sqsubseteq X'$, p' passes through X'. Let $e' \in X' \cap p'$ then: $e' \neq e$ because $e \notin X'$; $e' \notin in(t)$ otherwise, by Prop. 1, $e' \in X$. Let $p'' = e' \rightsquigarrow t$ where $e'' \in p'$, $\forall e'' \in p''$. Since $X' \sqsubseteq X$, p'' passes through X. Let $e'' \in X \cap p''$ then:

(i) If $e'' = e$ then p would not be simple.

(ii) If $e'' \neq e$ then p would not touch X only in e.

Finally the relation is also transitive: let $X \sqsubseteq X'$ and $X' \sqsubseteq X''$. Thus $e \sqsubseteq X'$, $\forall e \in X$, and $e' \sqsubseteq X''$, $\forall e' \in X'$; hence $e \sqsubseteq X''$, $\forall e \in X$, i.e. $X \sqsubseteq X''$. The same is for $\preceq$. ☐

For example consider the graph in Figure 3.3. Let $A = \{(s, v_2)\}$ then A is trivially covered by itself. Let $B = \{(v_2, v_3), (v_2, v_4)\}$ and $C = \{(v_4, t), (v_5, t), (v_6, t)\}$ then $A \sqsubseteq B$ and $B \sqsubseteq C$ but also $A \sqsubseteq C$. Clearly $C \not\sqsubseteq B$ and also $C \not\sqsubseteq A$.

Always by referring the graph in Figure 3.3. Let $A = \{(v_4, t), (v_5, t), (v_6, t)\}$ then A is trivially preceded by itself. Let $B = \{(v_3, v_5), (v_4, t), (v_6, t)\}$ and $C = \{(s, v_2)\}$ then $B \preceq A$ and $C \preceq B$ but also $C \preceq A$. Clearly $B \not\preceq C$ and also $A \not\preceq C$.

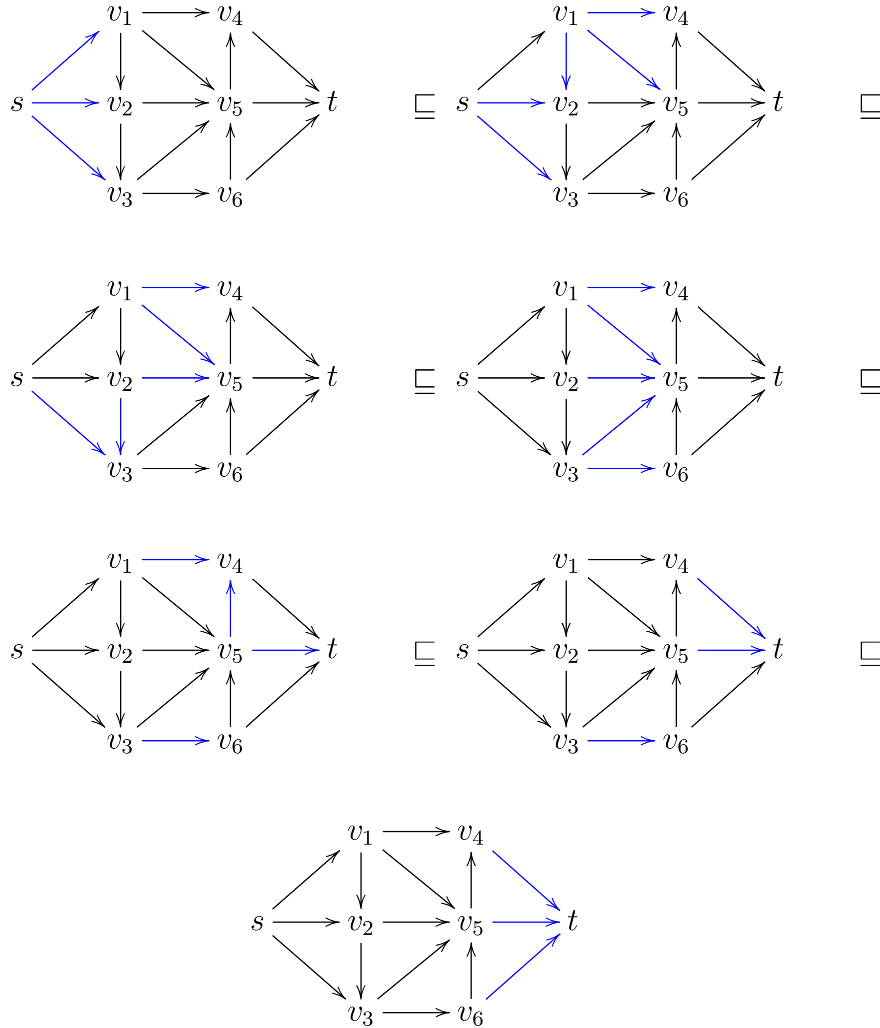
Thanks to relation in Definition 3.3.3, it is possible to define a chain of minimal edge-separators with minimum element $out(s)$ and maximum element $in(t)$.

Definition 3.3.6 (Complete chain of mes). A sequence of mes $X_0, X_1, \ldots, X_n$ is a chain if $X_i \sqsubseteq X_{i+1}, \forall i \in \{0, \ldots, n-1\}$. The chain is complete if $X_0 = out(s), X_n = in(t)$ and if $X_i \sqsubseteq X \sqsubseteq X_{i+1}$ then either $X = X_i$ or $X = X_{i+1}$.

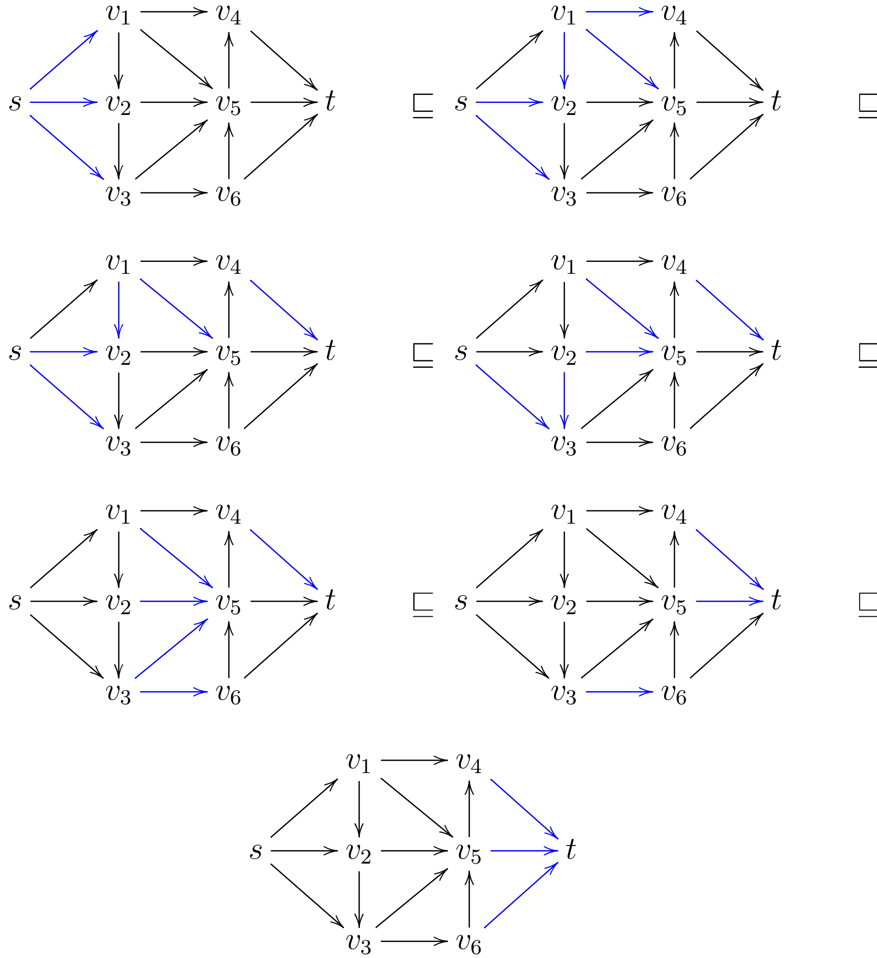
Some examples of complete chains of minimal edge-separators are in the next section.

3.3.3 Correctness of Checking Edge-Weakness

In [6] a specific critical node could not appear in a complete chain of mvs; the same happens with mes and critical edges. For this purpose, let G be the graph in Figure 3.2 (a) and C a complete chain of G:



Since in G there exists a path $p = v_3\ v_6\ v_5\ v_4$ the edge (v_5, v_4) is a critical edge in $\{(v_1, v_4), (v_3, v_6), (v_5, v_4), (v_5, t)\}$. However consider C':



The critical edge (v_5, v_4) does not belong to any mes in C' but at the same time C' contains (v_4, t) that is a critical edge.

Naturally, because of Theorem 2.3.1 G is edge-weak.

Again like [6], the above observation leads to Theorem 3.3.1. To prove such a theorem it is necessary the following lemma:

Lemma 3.3.2. *If $e \notin H$ and either $H \preceq e$ or $e \sqsubseteq H$, then $H \cup \{e\} \nsubseteq X$ for every mes X.*

Proof. Looking for a contradiction, assume that in a graph G there is an edge $e = (a, b)$ such that $H \preceq e$ and a mes X such that $H \cup \{e\} \subseteq X$. Since X is a separator, all paths from s to t cross X in some edge of X. Consider a path $s \rightsquigarrow a \to b \rightsquigarrow t$; since X contains H and $H \preceq e$, if the e is removed from X, X what we obtain would still be a separator. Indeed all paths $s \rightsquigarrow b$ ending in e would pass through H. Hence X cannot be a minimal edge-separator. The same happens if $e \sqsubseteq H$. $\square$

Theorem 3.3.1. *If the graph is edge-weak, then in every complete chain of mes $X_0, X_1, \dots, X_n$ there exists at least a X_i that contains a critical edge.*

Proof. Looking for a contradiction, assume that in a graph there is an e-critical mes X, where $e = (a, b)$, but there exists a complete chain $X_0, X_1, \dots, X_n$ such that every X_i does not contain any critical edge.

Trivially, $e \not\sqsubseteq X_0$ and $e \sqsubseteq X_n$ (this happens for every $e \in E \setminus out(s)$); thus let i be an index such that $e \not\sqsubseteq X_i$ and $e \sqsubseteq X_{i+1}$, where X_{i+1} is generated through any edge that belongs X_i. Trivially $e \notin X_i$ and $e \notin X_{i+1}$.

Let $prev^*(e) = \{(u, z) \in E \mid \exists s \rightsquigarrow a \rightarrow b \text{ s.t. } (u, z) \in s \rightsquigarrow a \rightarrow b\}$ be the set of many-steps predecessors of e, and let $P = prev^*(e) \cap X_i$ be the set of many-steps predecessors of e in X_i.

(i) By hypothesis $e \not\sqsubseteq X_i$ this implies that $X_i \preceq e$. Indeed since X_i is a mes all paths from the source node to the target node must cross X_i. So either every path from s to b, passing through e, finds X_i or every path from a to t, passing through e, finds X_i.
Since $X_i \preceq e$ and P is built by removing from X_i those edges that do not appear in the paths from s to b passing through e, $P \preceq e$; thus trivially $P \neq \emptyset$.

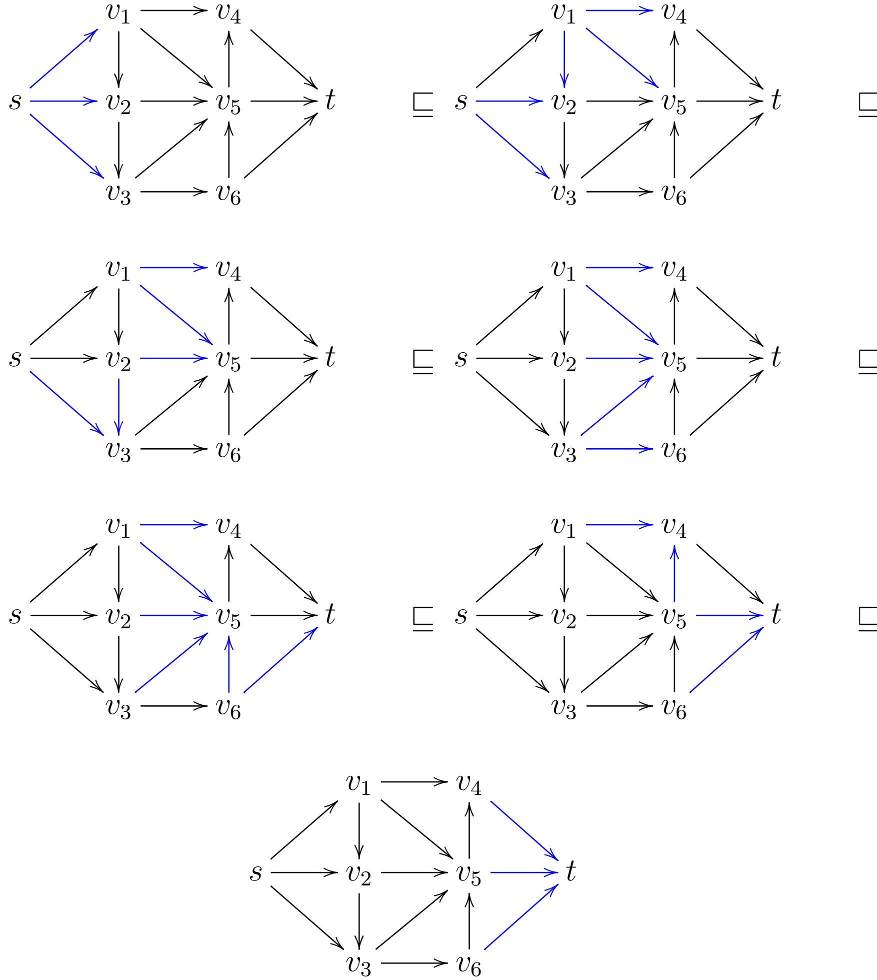
(ii) If some $e' = (a', b') \in P$ belonged to X_{i+1}, it could be found a path $a' \rightarrow b' \rightsquigarrow a \rightarrow b \rightsquigarrow a'' \rightarrow b''$ where $(a'', b'') \in X_{i+1}$; this would make X_{i+1} a critical mes. So, $P \cap X_{i+1} = \emptyset$.
Since $P \subseteq X_i$ and by hypothesis $X_i \sqsubseteq X_{i+1}$, also $P \sqsubseteq X_{i+1}$. Thus $\forall e' \in P$, $e' \sqsubseteq X_{i+1}$ and $e' \notin X_{i+1}$. Let $X_i^{e'}$ be the separator obtained from X_i via some $e' \in P$. If $X_i^{e'} \neq X_{i+1}$ then $X_0, X_1, \dots, X_n$ would not be a complete chain because there would exist a mes $X_i^{e'}$ such that $X_i \sqsubset X_i^{e'} \sqsubset X_{i+1}$, that contradicts Definition 3.3.6 and thus also the hypothesis. So $X_i^{e'} = X_{i+1}$, $\forall e' \in P$.

(iii) Let $U = X_{i+1} \setminus X_i$ be the set of new edges added from X_i to X_{i+1}; it is possible to prove that $e \sqsubseteq U$. Indeed if it was not, there would be a path from a to t, starting with e, passing either through an edge $e' \in X_i \setminus X_{i+1}$ or an edge $e' \in X_i \cap X_{i+1}$. Both these cases are not possible otherwise by point (i) X_i would be critical. Moreover because of point (ii) every edge in U is an immediate successor of every edge in P.

It is now possible to use these facts to contradict the assumption that e is a critical edge. Consider all the paths of the form $s \rightsquigarrow u \rightarrow z \rightarrow x \rightsquigarrow t$, with $(u, z) \in P$ and $(z, x) \in U$. Every mes, to cut such paths, must contain either a set of edges $P' \preceq P$ or a set of edges $U' \sqsupseteq U$. In both cases, since $P' \preceq e$ and $e \sqsubseteq U'$ (because of (i) and (iii)), e cannot belong to any mes (see Lemma 3.3.2) and hence cannot be a critical edge. □

Note that Definition 3.3.1 does not define a mes critical if it contains a critical edge, but it is critical if there exists a path between two of its edges. Moreover Theorem 3.3.1 is not enough to conclude that any complete chain of mes in an edge-weak graph contains at least a critical mes.

By continuing to refer to graph in Figure 3.2 (a), let C'' be another complete chain:



It is evident that it is required a way to check if any edge e is critical (and hence check the existence of an e-critical mes).

For this purpose, Theorem 3.3.2 is given which makes use of e-minimal mes.

Definition 3.3.7 (e-minimal mes). A mes X is *e-minimal* if $e \in X$ and $e \notin X'$, for every mes $X' \sqsubset X$.

Theorem 3.3.2. *If e is a critical edge, then all e-minimal mes are e-critical.*

Proof. Let $e = (a, b)$ be an edge and X be an e-critical mes, where $e' = (a', b') \in X$ is such that $a' \to b' \rightsquigarrow a \to b$. By contradiction, assume the existence of a mes X^* that is both e-minimal and not e-critical. Clearly, $e' \notin X^*$, because X^* is not e-critical.

Since X^* is a mes, all paths from s to t cross X^*. If $X^* \preceq e'$ then there exists a path that starts in X^* and reaches e'; because of the existence of $a' \to b' \rightsquigarrow a \to b$, X^* would be an e-critical mes. Therefore, it must be $e' \sqsubseteq X^*$.

Let $A = \{f = (u, z) \in X^* \mid a' \to b' \rightsquigarrow u \to z \neq (a, b)\} = \{f_1, \ldots, f_n\}$.

Since X^* is an e-minimal mes, it must contain at least the edge e. If $A = \emptyset$ then $e' \sqsubseteq e$, against minimality of X. Then it must be $A \neq \emptyset$.

For each element $f_i \in A$, consider the mes $X^*_{f_i} \sqsubset X^*$. By e-minimality of X^*, e does not belong to $X^*_{f_i}$. This implies that there exists a set of edges $P_i \subseteq prev(f_i)$ such that $P_i \preceq e$. Indeed, by construction $X^*_{f_i}$ is made up from (some of) X^*'s edges (except for f_i) and f_i's predecessors. Considering that e cannot be preceded by edges in X^*, e must be preceded by edges that are only in $X^*_{f_i}$, i.e. some of those in $prev(f_i)$.

Note that $P_i \neq \{e'\}$ otherwise $e' \preceq e$ and so e,e' could not belong to the same mes because of Lemma 3.3.2. Therefore, for each $i \in \{1, \ldots, n\}$, consider the set of edges $B_i \subseteq P_i$ that are on a path of the form $s \rightsquigarrow a \to b$ that does not pass through e'. Moreover, $B_i \neq \emptyset$ otherwise all paths from s to b, ending in e, would pass through e'.

Fix an $i \in \{1, \ldots, n\}$ where $f_i = (u_{f_i}, z_{f_i}) \in A$ and consider all paths of the form $s \rightsquigarrow u_{b_i} \to z_{b_i} \rightsquigarrow u_{f_i} \to z_{f_i} \rightsquigarrow t$, for all $(u_{b_i}, z_{b_i}) \in B_i$. Notice that $z_{b_i} = u_{f_i}$ because f_i must be an immediate successor of b_i. Given that $b_i \notin X$ and $f_i \notin X$, because in the former case e could not belong to X and in the latter case e' could not belong to X (since in either way X woud not be a mes), all these paths must cross X in some way.

(i) If X is crossed in the path $s \rightsquigarrow u_{b_i} \to z_{b_i} = u_{f_i} \to z_{f_i}$ then there exists a set of edges $L_i \subseteq X$ containing the predecessors of (u_{b_i}, z_{b_i}), so $L_i \preceq (u_{b_i}, z_{b_i})$. Since this argument works for every i, consider $L = \bigcup_i L_i \subseteq X$. Then, $L = \bigcup_i L_i \preceq \bigcup_i B_i = B$. But again, since $L \cup \{e'\} \preceq B \cup \{e'\} \preceq e$ and $L \cup \{e'\} \subseteq X$ therefore $e \notin X$ because of Lemma 3.3.2.

(ii) If X is crossed in the path $u_{b_i} \to z_{b_i} = u_{f_i} \to z_{f_i} \rightsquigarrow t$ then there exists a set of edges $R_i \subseteq X$ containing the successors of (u_{f_i}, z_{f_i}), so $(u_{f_i}, z_{f_i}) \sqsubseteq R_i$. Since this argument works for every i, consider $R = \bigcup_i R_i \subseteq X$. Then, $A = \{f_1, \ldots, f_n\} \sqsubseteq \bigcup_i R_i = R$. But again, since $e' \sqsubseteq A \cup \{e\} \sqsubseteq R \cup \{e\}$ and $R \cup \{e\} \subseteq X$, therefore $e' \notin X$ because of Lemma 3.3.2.

Hence e and e' cannot both belong to the same mes, contradicting the existence of X. □

Hence, with respect to the mes $\{(v_1, v_4), (v_5, v_4), (v_5, t), (v_6, t)\}$ in C'', it suffices to produce $\{(v_1, v_4), (v_3, v_6), (v_5, v_4), (v_5, t)\}$ which is a critical mes.

3.3.4 Polynomial-time Algorithm to checking Edge-Weakness

Based on the approach described above, the algorithm for checking edge-weakness is presented below. Algorithm 2 requires a di-graph G and it gives in output *true* if G is edge-weak, *false* otherwise.

To check edge-weakness a complete chain of mes is examined. For this purpose, X is initially equal to $out(s)$ and the *while* is executed until X is different from $in(t)$.

For each iteration of the loop, in Step 3 Algorithm 4 generates a new mes X'. Such a mes is the immediate successor of the mes X in the building chain. In detail Algorithm 4 chooses an edge from those belonging to the mes in input, produces a new mes following equation (3.1). Anyway the new mes is not necessarily an immeddiate successor (w.r.t. $\sqsubset$) of X. To abtain an immediate successor of X it suffices to consider the mes $X' = min_{e \in X} X_e$, where the minimum is calculated w.r.t. $\sqsubset$. Indeed, always considering the graph in Figure 3.2 (a), if $X = \{(s, v_1), (v_5, v_4), (v_5, t), (v_6, t)\}$ then $X_{(s,v_1)} = \{(v_1, v_4), (v_5, v_4), (v_5, t), (v_6, t)\} \sqsubset X_{(v_5,v_4)} = \{(v_4, t), (v_5, t), (v_6, t)\}$. At last Algorithm 4 gives such a minimum mes in output.

At this point, G is an edge-weak graph if either X' is a critical mes (hence there exists a path between two edges in X') or X' is an e-minimal mes. In the former case Algorithm 2 terminates giving in output *true*; in the latter case all edges in X' must be checked to determine if any of them is critical. To this aim, Algorithm 3 is called on every edge in $X' \setminus X$ (edges in X have already been controlled).

Algorithm 3 in Step 1 produces an e-minimal mes according to the edge and the mes in input. The condition $f \notin out(s)$ is added in Step 1 and in Step 5, to avoid an infinite loop. Hence, the algorithm chooses an edge f in the e-minimal mes and it generates the predecessor of the mes in input with respect to f, following equation (3.2). At the end, Algorithm 3 gives in output a new mes X^* which is a predecessor of X.

At this stage the criticality of X^* is checked. If the mes is critical then Algorithm 2 ends giving in output *true*, otherwise the next loop iteration is executed with X' replacing X.

If the main algorithm terminates at Step 11 then G has not chains with critical mes and hence it is not edge-weak.

The correctness of Algorithm 2 is given by Theorem 3.3.1 and by Theorem 3.3.2; the correctness of Algorithm 4 is proved by Theorem 3.2.1, Lemma 3.2.3 and Lemma 3.2.4. Indeed it adapts the pseudocode at lines 7 to 10 of Algorithm

1 in order to build a complete chain of mes generated from the bottom mvs $out(s)$ to $in(t)$. The correctness of Algorithm 3 is given by the following result.

Theorem 3.3.3. *X is e-minimal if and only if $(X \cup prev(f)) \setminus \{e\} \preceq e$, for every $f \in X$.*

Proof. Given a mes X and two edges $e, e' \in X$, consider the mes $X_{e'} \sqsubset X$. Note that $e \notin X_{e'}$ if and only if $e \in \mathcal{I}_s(X \cup prev(e'))$, i.e. $(X \cup prev(e')) \setminus \{e\} \prec e$. Since every predecessor (w.r.t. $\sqsubset$) of X can be obtained as $X_{e'}$, for some $e' \in X$, none of $X_{e'}$ can contain e in line with Definition 3.3.7. □

Algorithm 2 Checking Edge-Weakness

Input: A directed st-graph $G = (V, E)$, s is the source and t is the sink
function *isEdgeWeak*(G)

```
 1: X ← out(s);
 2: while X ≠ in(t) do
 3:     X' ← immediateMesRight(X)
 4:     if X' is edge-critical then
 5:         return TRUE
 6:     for all e ∈ X' \ X do
 7:         X* = minimalMes(X', e)
 8:         if X* is edge-critical then
 9:             return TRUE
10:     X ← X'
11: return FALSE
```

Algorithm 3 Generating an e-minimal mes smaller (w.r.t. $\sqsubset$) than X

Input: A mes X, an edge $e \in X$
function *minimalMes*(X, e)

```
1: A ← {f ∈ X | f ∉ out(s) ∧ (X ∪ prev(f)) \ {e} ⋠ e}
2: while A ≠ ∅ do
3:     choose f ∈ A
4:     X ← X^f
5:     A ← {f ∈ X | f ∉ out(s) ∧ (X ∪ prev(f)) \ {e} ⋠ e}
6: return X
```

Algorithm 4 Generating next minimal edge-separator

Input: An st-graph $G = (V, E)$, a mes X

function *immediateMesRight*(G, X)

1: $X' \leftarrow \emptyset$;
2: $L \leftarrow \emptyset$;
3: **for all** $e \in X \setminus in(t)$ **do**
4: Compute the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$;
5: **if** $G_t^{(X\setminus\{e\})\cup next(e)} \neq \emptyset$ **then**
6: Compute $\mathcal{I}_t((X \setminus \{e\}) \cup next(e))$;
7: $X' = ((X \setminus \{e\}) \cup next(e)) \setminus \mathcal{I}_t((X \setminus \{e\}) \cup next(e))$;
8: **if** $X' \notin L$ **then**
9: $L \leftarrow L \cup \{X'\}$;
10: **for all** $Y \in L$ **do**
11: **if** $Y \sqsubseteq X'$ **then**
12: $X' \leftarrow Y$;
13: **return** X';

Analysis

It is possible to calculate the reachability relation for every pair of edges in $O(n^3)$, one can fill a $n \times n$ matrix to check wheter X is critical in $O(n^2)$ in Alg. 2. Indeed, the time spent to see if an edge (a, b) reaches another edge (a', b') corresponds to that to see if b reaches a'.

Function *immediateMesRight* in Alg. 4 costs $O(m^2)$: in Step 6 if the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ is rapresented using the incidence matrix, checking if an edge has an immediate successor in $G_t^{(X\setminus\{e\})\cup next(e)}$ takes $O(m)$. Moreover, a mes can have at most m edges and the *for* loop at Step 10 costs $O(n^3)$.

Function *minimalMes* costs $O(m^2)$: there are at most $O(m)$ iterations of the *while* at Step 2 in Alg. 3 and each iteration costs $O(m)$ because of Step 4. Such a function is called at most $O(m)$ times. Indeed, if an edge e appears as a new node in $X' \setminus X$, it cannot have already appeared in a $X'' \sqsubset X'$ of the chain, otherwise there wolud exists a path from e to e and hence X'' (as well as X) would be critical and consequently function *isEdgeWeak* would have terminated in line 5 returning *true* as result. Thus, *minimalMes* costs $O(m^2)$ but at Alg. 2 it costs $O(m^3)$.

Finally, the *while* at Step 2 of Alg. 2 is executed at most $O(m)$ times, hence the total cost of *isEdgeWeak* is $O(m^4)$. Since in a di-graph the maximum number of edges is $n(n-1)/2$, the function in Alg. 2 costs $O(n^8)$.

An example of Algorithm 2 execution

Let G be the graph in Figure 3.2 (a).

- Initialization and 1st iteration of *while*:

1. At the beginning $X \leftarrow out(s) = \{(s, v_1), (s, v_2), (s, v_3)\}$.
2. Since X is different to $in(t)$, the *while* is executed.

    a) Algorithm 4 is called:

    i. For $e = (s, v_1)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (b).

    ii. $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\}$ is produced

    b) Since X' is not critical, Algorithm 3 is called for every edge in $X' \setminus X$:

    i. For $e = (v_1, v_4)$, it computed the e-minimal mes $A = \emptyset$. The same happens for (v_1, v_5) and (v_1, v_2).

    ii. Since Algorithm 3 has always given X' in output, $X^* = X'$ is not critical.

    c) The value of X is updated to X'.

- 2nd iteration of *while*:

1. Since $X = \{(v_1, v_4), (v_1, v_5), (v_1, v_2), (s, v_2), (s, v_3)\}$ is different to $in(t)$, the *while* is executed.

    a) Algorithm 4 is called:

    i. For $e = (s, v_2)$ it is computed the subgraph $G_t^{(X \setminus \{e\}) \cup next(e)}$ depicted in Figure 3.2 (g).

    ii. $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (v_2, v_5), (v_2, v_3), (s, v_3)\}$ is produced

    b) Since X' is not critical, Algorithm 3 is called for every edge in $X' \setminus X$:

    i. For $e = (v_2, v_5)$, it computed the e-minimal mes $A = \emptyset$. The same happens for $e = (v_2, v_3)$.

    ii. Since Algorithm 3 has always given X' in output, $X^* = X'$ is not critical.

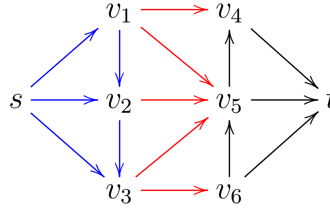
c) The value of X is updated to X'.

- 3rd iteration of *while*:

1. Let $X = \{(s, v_3), (v_1, v_4), (v_1, v_5), (v_2, v_3), (v_2, v_5)t\}$. Since X is different to $in(t)$, the *while* is executed.

   a) Algorithm 4 is called:

      i. For $e = (s, v_3)$ it is computed the subgraph $G_t^{(X\setminus\{e\})\cup next(e)}$ depicted below.



      ii. $X' \leftarrow \{(v_1, v_4), (v_1, v_5), (v_2, v_5), (v_3, v_5), (v_3, v_6)\}$ is produced

   b) Since X' is not critical, Algorithm 3 is called for every edge in $X' \setminus X$:

      i. For $e = (v_3, v_5)$ it computed the e-minimal mes $A = \{(v_1, v_4), (v_1, v_5), (v_2, v_5)\}$. Choosing $f = (v_2, v_5)$, $X^f = \{(v_1, v_4), (v_1, v_5), (v_3, v_5), (v_3, v_6), (s, v_2), (v_1, v_2)\}$ is produced. Since $A = \{(v_1, v_4), (v_1, v_5), (v_1, v_2)\} \neq \emptyset$, a second iteration of the *while* is made. Choosing $f = (v_1, v_5)$, $X^f = \{(v_3, v_5), (v_3, v_6), (s, v_2), (s, v_1)\}$ is produced.

      ii. Since Algorithm 3 has given $X^* = X^f$ in output and X^f is critical because the existence of the path from (s, v_1) to (v_3, v_6), Algorithm 2 ends giving in output *true*.

Hence G is edge-weak.

Chapter 4

Implementation

This chapter provides the implementation for the algorithms previously discussed.

The programming language used for such an implementation is Python 3.6. As described in Python official home [1]: "Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants, on the Mac, and on Windows 2000 and later."

Hence, the reasons for this choice of programming language lie in the strengths of Python: simplicity, libraries and a huge user community.

Apart from standard libraries, also the NetworkX 2.0 library is used in the implementation process. This package [2] provides classes for graph objects, generators to create standard graphs, IO routines for reading in existing datasets, algorithms to analyze the resulting networks and some basic drawing tools.

4.1 Code Organization

The source code is distributed between seven files:

- **alg1.py**, **alg2.py**, **alg3.py**, **alg4.py** contain, respectively, the implementation of Algorithm 1, Algorithm 2, Algorithm 3 and Algorithm 4.

- **utils.py** contains some functions of utility: two functions are for the creation of the graphs (the first one can create graphs from file .txt and the second one can create graphs from one of NetworkX's generators); three functions are for monitoring (in details these functions check if an edge is in the graph,

if a set of edges are in the graph and if a minimal edge separator is really minimal).

- **vgraph.py**, **gui.py** contain some functions for the graphic representation of the graphs. More specifically vgraph.py draws graphs and gui.py launches the application *Graph Viewer*. Such an application lets users choose how to build a graph (from database or from generator) and after it shows such a graph with some information. This information is the number of nodes, the number of edges and if the graph suffers from edge-weakness. Moreover, if the graph is edge weak on its picture is highlighted the path touching twice the mes (also the mes is highlighted).

More detailed information on functions is reported on 'pythondocs' of each function.

4.2 Source Code

4.2.1 Algorithm 1

```
1  from networkx.algorithms import approximation as approx
2  from alg4 import *
3  from utils import *
4  import networkx as nx
5  import random
6
7  '''
8  The function computes the immediate successor of 'mes' with respect
       to 'edge' in 'graph'.
9  @param graph: the starting digraph
10 @type graph: <class 'networkx.classes.digraph.DiGraph'>
11 @param mes: the mes of which the immediate successor is computed
12 @type mes: <class 'list'>
13 @param edge: the edge under which the successor of mes is computed
14 @type edge: <class 'tuple'>
15 @return: the immediate successor of mes_edge
16 @rtype: <class 'list'>
17 @raise: networkx.NetworkXException if graph not contains mes or the
       mes is not minimal
18 '''
19 def successor_mes_edge(graph, mes, edge):
```

```
20      if not is_minimal(graph, mes): raise nx.NetworkXException(str(mes
     ) + ' is not a minimal edge-separator')
21
22      s=0
23      t=len(graph.nodes())-1
24
25      in_t=list(graph.in_edges(t)) #edges incoming in t
26      if edge in in_t: return mes #an edge incidents to t has no
     successors
27
28      x_e_next=mes+(list(graph.edges(edge[1])))
29      x_e_next.remove(edge) #(X\{e}) U next(e)
30
31      g=graph.copy() #deep copy
32      g_t=compute_gt(g, x_e_next) #(G_t)^((X\{e}) U next(e))
33
34      i_t=[] #I_t((X\{e}) U next(e))
35      ed_g=list(g.edges())
36      ed_gt=list(g_t.edges())
37
38      for e in x_e_next:
39          next_e=list(graph.edges(e[1]))
40          l=[e2 for e2 in next_e if e2 not in ed_gt]
41          if len(l)==len(next_e) and e not in in_t:
42              i_t.append(e)
43
44      mes_r=[e for e in x_e_next if e not in i_t] #((X\{e}) U next(e))
     \#I_t((X\{e}) U next(e))
45      return mes_r
46
47  '''
48  The function computes the set of all minimal edge separators in '
     graph'. The number of keys of the dictionary, containing all these
      mes, is equal to the maximum number of levels in 'graph', that is
      equal to the shortest path from s to t.
49  @param graph: the starting digraph of which the set of all minimal
     edge separators is computed
50  @type graph: <class 'networkx.classes.digraph.DiGraph'>
51  @return: the dictionary having as key the integers indicating the
     level of the minimal edge separators, and as values the list of
```

```
    minimal edge separators
52 @rtype: <class 'dict'>
53 '''
54 def st_minimal_edge_separators(graph):
55     s=0
56     t=len(graph.nodes())-1
57 
58     max_lev=0 #number of levels
59     temp_lev=0
60     in_t=list(graph.in_edges(t)) #edges entering in t
61     for edge in in_t:
62         if edge[0]!=0 and approx.node_connectivity(graph, 0, edge[0])
    >0:
63             temp_lev=nx.shortest_path_length(graph, 0, edge[0])
64             if temp_lev>max_lev:
65                 max_lev=temp_lev
66 
67     L={} #dict containing all mes
68     i=0
69     while i<=max_lev+1:
70         L[i]=[]
71         i+=1
72 
73     out_s=graph.edges(s)
74 
75     j=0
76     L[j]+=[set(out_s)]
77 
78     while j<=max_lev:
79         l_j=L[j].copy() #L_j
80         l_j1=L[(j+1)].copy() #L_j+1
81         len_lj=len(l_j)
82 
83         l_j_j1=__st_minimal_edge_separators_aux__(graph, l_j.copy(),
    l_j, l_j1, j)
84 
85         for mes in l_j_j1[0]: #mes in L_j
86             if mes not in L[j]:
87                 L[j].append(mes)
88         for mes in l_j_j1[1]: #mes in L_j+1
```

```
89              if mes not in L[(j+1)]:
90                  L[(j+1)].append(mes)
91
92          if len_lj==len(L[j]): #no new mes are added to L_j hence all
    mes in L_j were examined
93              j+=1
94
95      for mes in L[(max_lev+1)]:
96          if mes not in L[max_lev]:
97              L[(max_lev)]+=[mes]
98      del L[(max_lev+1)]
99      return L
100
101 ''' The function  computes all immediate successors of all minimal
    edge separators in 'l' and puts each computed mes either in 'l_j'
    or in '_j1' with respect to 'j'. The function returns a tuple with
     'l_j' and 'l_j1' (containing the computed mes).'''
102 def __st_minimal_edge_separators_aux__(graph, l, l_j, l_j1, j):
103     s=0
104     t=len(graph.nodes())-1
105
106     for mes in l:
107         for edge in mes:
108             h_e=nx.shortest_path_length(graph, s, edge[0]) #distance
    from s to edge
109             mes_r=set(successor_mes_edge(graph, list(mes), edge)) #
    immediate successor of mes with respect to edge
110
111             if mes_r not in l_j+l_j1: #mes_r must not be just
    computed
112                 if h_e==j:
113                     l_j1.append(mes_r)
114                 else:
115                     l_j.append(mes_r)
116
117     return (l_j, l_j1)
```

4.2.2 Algorithm 2, Algorithm 3, Algorithm 4

```
1  from utils import *
2  from alg3 import *
3  from alg4 import *
4  
5  '''
6  The function tests  if 'graph' is edge−weak. If 'graph' is edge−weak
       the function returns a tuple. Such a tuple contains a bool value (
       True), a list (the path) and another list (the mes that is touched
        twice by the path). This choice of return type is done for
       facilitating the edge−weakness visualization of the graph.
7  @param graph: the starting digraph
8  @type graph: <class 'networkx.classes.digraph.DiGraph'>
9  @return: False if the graph is not edge−weak, (True, path, mes)
       otherwise
10 @rtype: <'bool'> / <'tuple'>
11 '''
12 def is_edge_weak(graph):
13     s=0
14     t=len(list(graph.nodes()))−1
15 
16     mes=list(graph.edges(s)) #first mes of every chain
17     in_t=list(graph.in_edges(t)) #last mes of every chain
18 
19     while mes!=in_t:
20         mes_r=immediate_mes_right(graph, mes) #an immediate successor
        of 'mes'
21         bool_path=__is_edge_critical__(graph, mes_r)
22         if bool_path[0]==True:
23             return (True, bool_path[1], mes_r)
24 
25         diff=[e for e in mes_r if e not in mes] #e in X\X'
26         for e in diff:
27             mes_m=minimal_mes(graph, mes_r, e) #X*
28             bool_path=__is_edge_critical__(graph, mes_m)
29             if bool_path[0]==True:
30                 return (True, bool_path[1], mes_m)
31 
32         if mes!=mes_r:
33             mes=mes_r
```

```
34         else:
35             break
36     return False
37 
38 ''' The function tests if there exists a path between two edges of '
    mes' in 'graph' '''
39 def __is_edge_critical__(graph, mes):
40     for edge in mes:
41         for edge1 in mes:
42             if edge!=edge1 and (approx.node_connectivity(graph, edge
    [1], edge1[0])>0):
43                 paths = nx.all_simple_paths(graph, edge[1], edge1[0])
44                 m=map(nx.utils.pairwise, paths) #paths expressed in
    edges
45                 ll=[]
46                 for path in m:
47                     l=[edge]
48                     l+=list(path)
49                     l.append(edge1)
50                     ll.append(l)
51                 return (True, ll[random.randint(0, len(ll)-1)])
52     return (False, [])
```

```
1 from networkx.algorithms import approximation as approx
2 from utils import *
3 import networkx as nx
4 import random
5 
6 '''
7 The function computes (G_s)^edges.
8 @param graph: the digraph on which the subgraph (G_s)^edges is
    computed
9 @type graph: <class 'networkx.classes.digraph.DiGraph'>
10 @param edges: the set of edges to remove from graph
11 @type edges: <class 'list'>
12 @return: the digraph (G_s)^edges
13 @rtype: <class 'networkx.classes.digraph.DiGraph'>
14 @raise: networkx.NetworkXException if graph not contains edges
15 '''
16 def compute_gs(graph, edges):
17     are_edges(graph, edges)
```

```
18
19     s=0
20     graph.remove_edges_from(edges)
21
22     g=nx.DiGraph()
23     g.add_edges_from(graph.edges(s))
24     for e in graph.edges():
25         if e[0]!=s and approx.node_connectivity(graph, s, e[0])>0: #
       edges reachable from s
26             g.add_edge(e[0], e[1])
27     return g
28
29 '''
30 The function tests if all paths from s to 'edge' pass through 'set_e
       '.
31 @param graph: the starting digraph
32 @type graph: <class 'networkx.classes.digraph.DiGraph'>
33 @param set_e: the set of edges on which the edge-precedence is tested
34 @type set_e: <class 'list'>
35 @param edge: the edge on which the edge-precedence is tested
36 @type edge: <class 'tuple'>
37 @return: True if edge is preceded by set_e, False otherwise
38 @rtype: <class 'bool'>
39 @raise: networkx.NetworkXException if graph not contains edge or
       set_e
40 '''
41 def edge_prec(graph, set_e, edge):
42     are_edges(graph, set_e)
43     is_edge(graph, edge)
44
45     s=0
46
47     if edge in set_e: return True
48     if edge[0]==s and edge not in set_e: return False
49
50     paths = nx.all_simple_paths(graph, s, target=edge[0])
51     m=map(nx.utils.pairwise, paths) #paths expressed in edges
52     for path in m:
53         l=list(path)
54         inter=[e for e in l if e not in set_e]
```

```
55          if inter==1:
56              return False
57      return True
58
59  '''
60  The function tests if all paths from s to each edge in 'edge_s1' pass
        through 'edge_s2'.
61  @param graph: the starting digraph
62  @type graph: <class 'networkx.classes.digraph.DiGraph'>
63  @param edge_s1: the set of edges on which the edges set-precedence is
        tested
64  @type edge_s1: <class 'list'>
65  @param edge_s2: the set of edges on which the edges set-precedence is
        tested
66  @type edge_s2: <class 'list'>
67  @return: True if edge_s1 is preceded by edge_s2, False otherwise
68  @rtype: <class 'bool'>
69  @raise: networkx.NetworkXException if graph not contains edge_s1 or
       edge_s2
70  '''
71  def edge_set_prec(graph, edge_s1, edge_s2):
72      are_edges(graph, edge_s1)
73      are_edges(graph, edge_s2)
74
75      is_in=[]
76      for edge in edge_s1:
77          is_in.append(edge_prec(graph, edge_s2, edge))
78
79      if False not in is_in:
80          return True
81      return False
82
83  '''
84  The function computes the immediate predecessor of 'mes' with respect
        to 'edge' in 'graph'.
85  @param graph: the starting digraph
86  @type graph: <class 'networkx.classes.digraph.DiGraph'>
87  @param mes: the mes of which the immediate predecessor is computed
88  @type mes: <class 'list'>
89  @param edge: the edge under which the predecessor of mes is computed
```

```
90 @type edge: <class 'tuple'>
91 @return: the immediate predecessor of mes^edge
92 @rtype: <class 'list'>
93 @raise: networkx.NetworkXException if graph not contains mes or the
       mes is not minimal
94 '''
95 def predecessor_mes_edge(graph, mes, edge):
96     if not is_minimal(graph, mes): raise nx.NetworkXException(str(mes
       ) + ' is not a minimal edge-separator')
97 
98     s=0
99     t=len(graph.nodes())-1
100 
101     out_s=list(graph.edges(s)) #edges outgoing from s
102     if edge in out_s: return mes #an edge incidents to s has not
       predecessors
103 
104     x_e_prev=mes+list(graph.in_edges(edge[0]))
105     x_e_prev.remove(edge) #(X\{e}) U prev(e)
106 
107     g=graph.copy() #deep copy
108     g_s=compute_gs(g, x_e_prev) #(G_s)^((X\{e}) U prev(e))
109 
110     i_s=[] #I_s((X\{e}) U prev(e))
111     ed_g=list(graph.edges())
112     ed_gs=list(g_s.edges())
113 
114     for e in x_e_prev:
115         prev_e=list(graph.in_edges(e[0]))
116         l=[e2 for e2 in prev_e if e2 not in ed_gs]
117         if len(l)==len(prev_e) and e not in out_s:
118             i_s.append(e)
119 
120     mes_l=[e for e in x_e_prev if e not in i_s] #((X\{e}) U prev(e))\
       I_s((X\{e}) U prev(e))
121     return mes_l
122 
123 '''
124 The function computes an 'edge'-minimal mes smaller (w.r.t. set-
       coverage) than 'mes'.
```

```
125 @param graph: the starting digraph
126 @type graph: <class 'networkx.classes.digraph.DiGraph'>
127 @param mes: the mes of which the 'edge'-minimal mes is computed
128 @type mes: <class 'list'>
129 @param edge: the edge under which the 'edge'-minimal mes is computed
130 @type edge: <class 'tuple'>
131 @return: 'edge'-minimal mes
132 @rtype: <class 'list'>
133 '''
134 def minimal_mes(graph, mes, edge):
135     x=mes.copy()
136     a=__compute_a__(graph, x, edge)
137     while len(a)>0:
138         f=a[random.randint(0, len(a)-1)]
139         x1=predecessor_mes_edge(graph, x, f) #X^f
140         x=x1.copy()
141         a=__compute_a__(graph, x, edge)
142     return x
143 
144 ''' The function computes the set A={f in 'mes' | f not in out(s) and
     ('mes' U prev(f))\{'edge'} not precedes 'edge'}. '''
145 def __compute_a__(graph, mes, edge):
146     s=0
147     out_s=list(graph.edges(s))
148     a=[]
149     for f in mes:
150         if f not in out_s:
151             prev_edge=list(graph.in_edges(f[0]))
152             x_f_prev=mes+prev_edge
153             x_f_prev.remove(edge)
154 
155             if not edge_prec(graph, x_f_prev, edge):
156                 a.append(f)
157     return a
```

```
1 from networkx.algorithms import approximation as approx
2 from utils import *
3 import networkx as nx
4 import random
5 
6 '''
```

```
7  The function computes (G_t)^edges.
8  @param graph: the digraph on which the subgraph (G_t)^edges is
       computed
9  @type graph: <class 'networkx.classes.digraph.DiGraph'>
10 @param edges: the set of edges to remove from graph
11 @type edges: <class 'list'>
12 @return: the digraph (G_t)^edges
13 @rtype: <class 'networkx.classes.digraph.DiGraph'>
14 @raise: networkx.NetworkXException if graph not contains edges
15 '''
16 def compute_gt(graph, edges):
17     are_edges(graph, edges)
18 
19     t=len(graph.nodes())-1
20     graph.remove_edges_from(edges)
21 
22     g=nx.DiGraph()
23     g.add_edges_from(graph.in_edges(t))
24     for e in graph.edges():
25         if e[1]!=t and approx.node_connectivity(graph, e[1], t)>0: #
       edges reaching t
26             g.add_edge(e[0], e[1])
27     return g
28 
29 '''
30 The function tests if all paths from 'edge' to t pass through 'set_e
       '.
31 @param graph: the starting digraph
32 @type graph: <class 'networkx.classes.digraph.DiGraph'>
33 @param set_e: the set of edges on which the edge-coverage is tested
34 @type set_e: <class 'list'>
35 @param edge: the edge on which the edge-coverage is tested
36 @type edge: <class 'tuple'>
37 @return: True if edge is covered by set_e, False otherwise
38 @rtype: <class 'bool'>
39 @raise: networkx.NetworkXException if graph not contains edge or
       set_e
40 '''
41 def edge_cover(graph, set_e, edge):
42     are_edges(graph, set_e)
```

```
43      is_edge(graph, edge)
44
45      t=len(graph.nodes())-1
46
47      if edge in set_e: return True
48      if edge[1]==t and edge not in set_e: return False
49
50      paths = nx.all_simple_paths(graph, edge[1], target=t)
51      m=map(nx.utils.pairwise, paths) #paths expressed in edges
52      for path in m:
53          l=list(path)
54          inter=[e for e in l if e not in set_e]
55          if inter==l:
56              return False
57      return True
58
59  '''
60  The function tests if all paths from each edge in 'edge_s1' to t,
        pass through 'edge_s2'.
61  @param graph: the starting digraph
62  @type graph: <class 'networkx.classes.digraph.DiGraph'>
63  @param edge_s1: the set of edges on which the edges set-coverage is
        tested
64  @type edge_s1: <class 'list'>
65  @param edge_s2: the set of edges on which the edges set-coverage is
        tested
66  @type edge_s2: <class 'list'>
67  @return: True if edge_s1 is covered by edge_s2, False otherwise
68  @rtype: <class 'bool'>
69  @raise: networkx.NetworkXException if graph not contains edge_s1 or
        edge_s2
70  '''
71  def edge_set_cover(graph, edge_s1, edge_s2):
72      are_edges(graph, edge_s1)
73      are_edges(graph, edge_s2)
74
75      is_in=[]
76      for edge in edge_s1:
77          is_in.append(edge_cover(graph, edge_s2, edge))
78
```

```
79      if False not in is_in:
80          return True
81      return False
82
83  '''
84  The function computes all immediate successors of 'mes' and returns
    one of them.
85  @param graph: the starting digraph
86  @type graph: <class 'networkx.classes.digraph.DiGraph'>
87  @param mes: the mes of which the immediate successor is computed
88  @type mes: <class 'list'>
89  @return: the immediate successor of mes
90  @rtype: <class 'list'>
91  @raise: networkx.NetworkXException if graph not contains mes or the
    mes is not minimal
92  '''
93  def immediate_mes_right(graph, mes):
94      if not is_minimal(graph, mes): raise nx.NetworkXException(str(mes
    ) + ' is not a minimal edge-separator')
95
96      s=0
97      t=len(graph.nodes())-1
98
99      L=[]
100     mes_r=[]
101
102     g=graph.copy() #deep copy
103     x=mes.copy()
104
105     in_t=list(g.in_edges(t)) #edges incoming in t
106
107     if mes==in_t: return mes #in_t is the last mes in every chain
108     diff=[e for e in mes if e not in in_t]
109
110     for e in diff:
111         x_e_next=mes+(list(graph.edges(e[1])))
112         x_e_next.remove(e) #(X\{e}) U next(e)
113
114         g_t=compute_gt(g, x_e_next) #(G_t)^((X\{e}) U next(e))
115
```

```
116         i_t=[] #I_t((X\{e}) U next(e))
117         ed_g=list(g.edges())
118         ed_gt=list(g_t.edges())
119 
120         for e in x_e_next:
121             next_e=list(graph.edges(e[1]))
122             l=[e2 for e2 in next_e if e2 not in ed_gt]
123             if len(l)==len(next_e) and e not in in_t:
124                 i_t.append(e)
125 
126         mes_r=[e for e in x_e_next if e not in i_t] #((X\{e}) U next(
    e))\I_t((X\{e}) U next(e))
127 
128         L.append(mes_r)
129         mes_r=[]
130         x_e_next=[]
131         g=graph.copy() #deep copy
132         x=mes.copy()
133 
134     if len(L)>0:
135         X1=L[random.randint(0, len(L)-1)]
136         for X in L:
137            if edge_set_cover(graph, X, X1):
138                X1=X
139         return X1
140     return []
```

4.2.3 Graph Viewer

```
1  from alg2 import *
2  from alg1 import *
3  from vgraph import *
4  from tkinter import filedialog as fd
5  from tkinter import messagebox
6  from tkinter import *
7  from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
8  import tkinter as tk
9
10 '''
11 This is a gui application for viewing graphs. Such graphs may be
       obtained taking a database in input, or the may be generated by a
       generator.
12 The application displays the graph (which is drawn by function in
       vgraph.py) and shows some informations about the graph.
13 The gui is realized using Tkinter package. The gui consists of a
       PanedWindow in which two Buttons are placed, a Canvas in which the
        drawn of the graph is palced and of another PanedWindow in which
       the Label with the informations on the graph is placed.
14 When the user clicks on 'Draw Graph from Database' the graph is
       directly shown; when the user clicks on 'Draw Graph from Generator
       ' a new window is shown. This window (DIALOG) contains two Labels
       with two Entries for entering the minimum and the maximum number
       of nodes for the graph under contruction. Once 'Compute the graph'
        is clicked the graph is displayed.
15 '''
16
17 WIN=tk.Tk()
18 WIN.title('Graph Viewer')
19 w=1000
20 h=1000
21 ws=WIN.winfo_screenwidth()
22 hs=WIN.winfo_screenheight()
23 x = (ws/2) - (w/2)
24 y = (hs/2) - (h/2)
25 WIN.geometry('%dx%d+%d+%d' % (w, h, x, y))
26 WIN.wm_iconbitmap("icona.ico")
27 WIN.focus_set()
28
```

```
29 f=plt.figure(figsize=(5,4))
30 a=f.add_subplot(111)
31
32 draw_digraph(nx.Graph(), 0, 0)
33
34 canvas=FigureCanvasTkAgg(f, master=WIN)
35 canvas.show()
36
37 label_info=Label(WIN, justify=LEFT, font=("Helvetica", 12),
       wraplength=500)
38
39 def show_graph(g):
40     s=0
41     t=len(g.nodes())-1
42
43     is_edge=is_edge_weak(g)
44     is_edge_str='No'
45     if (is_edge==False):
46         draw_digraph(g, s, t)
47     else:
48         draw_digraph(g, s, t, is_edge[1], is_edge[2])
49         is_edge_str='Yes.\n'+'- In the graph there exists the path: \
       n'+str(is_edge[1])+'\n'+'- The path passes twice through the mes:
       \n'+str(is_edge[2])
50
51     canvas.draw()
52
53     label_info['text']='Number of node: '+str(len(g.nodes()))+'\n'+'
       Number of edges: '+str(len(g.edges()))+'\n'+'The graph is edge-
       weak? '+is_edge_str
54
55 def new_graph_from_bd():
56     a.clear()
57     g=create_graph_from_file('prova.txt')
58     show_graph(g)
59
60 def choose_min_max_node():
61     DIALOG=tk.Tk()
62     DIALOG.title('Choosing number of nodes')
63     DIALOG.resizable(False, False)
```

```
64      w=400
65      h=200
66      ws=WIN.winfo_screenwidth()
67      hs=WIN.winfo_screenheight()
68      x = (ws/2) - (w/2)
69      y = (hs/2) - (h/2)
70      DIALOG.geometry('%dx%d+%d+%d' % (w, h, x, y))
71      DIALOG.wm_iconbitmap("icona.ico")
72      DIALOG.focus_set()
73
74      labelframe=LabelFrame(DIALOG, bg='white', relief = FLAT)
75      labelframe.pack(fill = "both", expand = "yes")
76      label_message=Label(labelframe, bg='white', justify=LEFT, text='\
     n Please enter the minimum and the maximum number of nodes  \n to
     compute the graph \n')
77      label_message.pack(side=tk.TOP)
78
79      label_entry_min=PanedWindow(DIALOG)
80      label_min=Label(label_entry_min, text='Minimum number of nodes
     (>1): ')
81      label_min.pack(side=LEFT)
82      content = StringVar()
83      entry_min=Entry(label_entry_min, bd=3, textvariable=content)
84      entry_min.pack(side=RIGHT)
85      label_entry_min.add(label_min)
86      label_entry_min.add(entry_min)
87      label_entry_min.pack(fill=tk.BOTH, expand=1)
88
89      label_entry_max=PanedWindow(DIALOG)
90      label_max=Label(label_entry_max, text='Maximum number of nodes
     (>1): ')
91      label_max.pack(side=LEFT)
92      entry_max=Entry(label_entry_max, bd=3)
93      entry_max.pack(side=RIGHT)
94      label_entry_max.add(label_max)
95      label_entry_max.add(entry_max)
96      label_entry_max.pack(fill=tk.BOTH, expand=1)
97
98      def callback():
99          s_min=entry_min.get()
```

```
100         s_max=entry_max.get()
101         if s_min.isdigit() and s_max.isdigit():
102             min_n=int(s_min)
103             max_n=int(s_max)
104             if min_n>1 and min_n<=max_n:
105                 print(min_n, max_n)
106                 DIALOG.destroy()
107                 new_graph_from_generator(min_n, max_n)
108 
109     button_ok=tk.Button(DIALOG, text="Compute the graph", command=
        callback)
110     button_ok.pack(side=tk.BOTTOM, fill=tk.BOTH, expand=1)
111 
112 def new_graph_from_generator(min_n, max_n):
113     a.clear()
114     g=create_graph_with_nx_generator(min_n, max_n)
115     show_graph(g)
116 
117 buttons_pained=PanedWindow()
118 
119 graph_db=tk.Button(buttons_pained, text="Draw Graph from Database",
        command=new_graph_from_bd, font=("Helvetica", 12))
120 graph_gen=tk.Button(buttons_pained, text="Draw Graph from Generator",
         command=choose_min_max_node, font=("Helvetica", 12))
121 
122 buttons_pained.add(graph_db)
123 buttons_pained.add(graph_gen)
124 
125 buttons_pained.pack(side=tk.TOP)
126 canvas.get_tk_widget().pack(fill=tk.BOTH, expand=1)
127 label_info.pack()
128 
129 tk.mainloop()
```

```
1 import networkx as nx
2 import numpy as np
3 import matplotlib.pyplot as plt
4 
5 '''
6 This function draw 'graph'. If 'path' and 'mes' are in input, then
      the function highlights red edges in 'path' and in yellow edges in
```

```
    'mes'.
@param graph: the graph to draw
@type graph: <class 'networkx.classes.digraph.DiGraph'>
@param s: the starting node in 'graph'
@type s: <class 'int'>
@param t: the ending node in 'graph'
@type t: <class 'int'>
@param path: the edges in path
@type path: <class 'list'>
@param mes: the edges in mes
@type mes: <class 'int'>
'''
def draw_digraph(graph, s, t, path=None, mes=None):
    pos=nx.circular_layout(graph)

    val_map = {s: 'r', t: 'r'}
    values = [val_map.get(node, 'b') for node in graph.nodes()]

    nx.draw_networkx_nodes(graph, pos, node_color =values, alpha=0.9)
    nx.draw_networkx_edges(graph, pos, edge_color='black', style='
dashed', arrows=True)

    if path!=None:
        nx.draw_networkx_edges(graph, pos, edgelist=path, width=2.0,
edge_color='red', arrows=True)

    if mes!=None:
        nx.draw_networkx_edges(graph, pos, edgelist=mes, width=3.0,
alpha=0.7, edge_color='yellow')

    labels={}
    labels[s]=r'$s$'
    labels[t]=r'$t$'
    for node in graph.nodes():
        if node!=s and node !=t:
            labels[node]=node
            nx.draw_networkx_labels(graph,pos, labels,font_size=16)

    plt.axis('off')
```

```
from networkx.algorithms import approximation as approx
```

```
import networkx as nx
import random

'''
This function creates a digraph starting from a file.txt.
@param path_graph: the path (relative or absolute) of the file
    containing the graph
@type path_graph: <class 'str'>
@return: the digraph G
@rtype: <class 'networkx.classes.digraph.DiGraph'>
'''
def create_graph_from_file(path_graph):
    g=nx.DiGraph()
    with open(path_graph) as f:
        line=random.choice(f.readlines())
        line=line[:len(line)-4]
        line=line.split(' {{')[1].split('}, {')
        for l in line:
            ll=l.split(',')
            n1=ll[0].strip()
            n2=ll[1].strip()
            g.add_edge(int(n1),int(n2))
        return g

'''
This function creates a digraph with networkx random generator. The
    graph in output is a simple graph, and all its vertices belong to
    an st-path.
@param min_n: the minimum number of nodes
@type min_n: <class 'int'>
@param max_n: the maximum number of nodes
@type max_n: <class 'int'>
@return: the digraph G
@rtype: <class 'networkx.classes.digraph.DiGraph'>
'''
def create_graph_with_nx_generator(min_n, max_n):
    n=random.randint(min_n, max_n)
    m=random.randint(n, (n-1)+(n-2)*(n-2))
    g=nx.gnm_random_graph(n, m, directed=True)

```

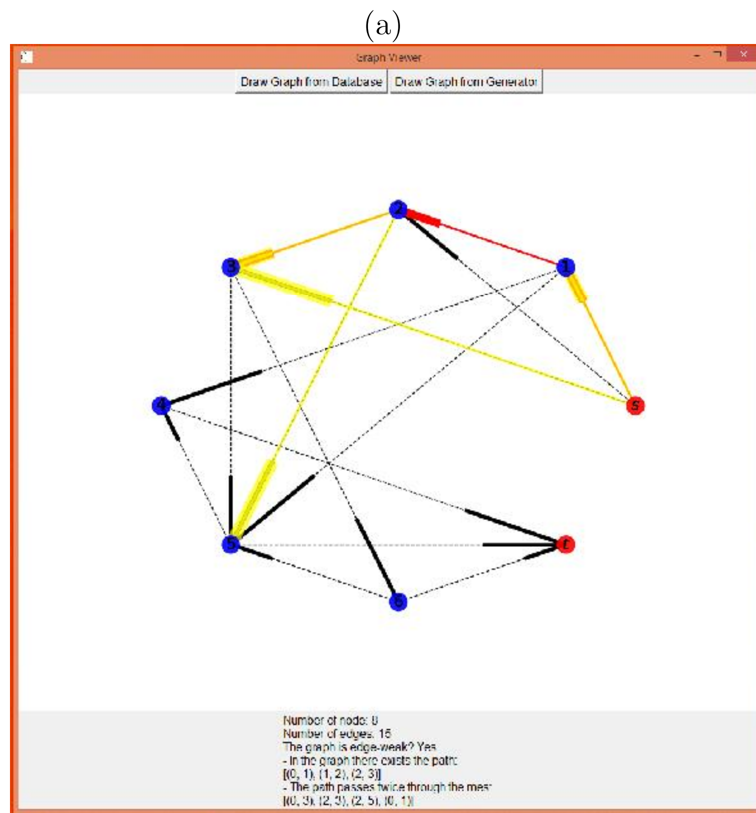
```
39     g.remove_edges_from(list(g.in_edges(0))) #remove nodes incoming
      in s
40     g.remove_edges_from(list(g.edges(len(g.nodes())-1))) #remove
      nodes outcoming from t
41     g.remove_nodes_from(list(nx.isolates(g))) #remove isolated nodes
42 
43     t=len(list(g.nodes()))-1
44 
45     if 0 in g.nodes() and t in g.nodes() and approx.node_connectivity
      (g, 0, t)>0: #remove nodes not in an st-path
46         paths=list(nx.all_simple_paths(g, 0, t))
47         nodes_ok=[]
48         nodes_no=[]
49         for path in paths:
50             nodes_ok+=path
51         for node in g.nodes():
52             if node not in nodes_ok:
53                 nodes_no.append(node)
54         g.remove_nodes_from(nodes_no)
55         relabel_nodes={}
56         for i in list(range(len(list(g.nodes())))):
57             relabel_nodes[list(g.nodes())[i]]=i
58         g=nx.relabel_nodes(g, relabel_nodes)
59 
60     else:
61         g=create_graph_with_nx_generator(min_n, max_n)
62     return g
63 
64 ''' The function tests if 'edge' is in 'graph'. If 'edge' not in '
      graph', the function raises an exception. '''
65 def is_edge(graph, edge):
66     if not graph.has_edge(edge[0], edge[1]):
67         raise nx.NetworkXException(str(edge) + ' not in graph' )
68     return
69 
70 ''' The function tests if 'edges' are in 'graph'. If 'edges' not in '
      graph', the function raises an exception. '''
71 def are_edges(graph, edges):
72     for edge in edges:
73         is_edge(graph, edge)
```

```
74
75 ''' The function tests if 'mes' is minimal, hence if 'mes' is a
    minimal edge separator and not only an edge separator. '''
76 def is_minimal(graph, mes):
77     are_edges(graph, mes)
78
79     s=0
80     t=len(graph.nodes())-1
81
82     if (mes==list(graph.edges(s))): return True
83
84     g=graph.copy()
85     x=mes.copy()
86
87     rem=[]
88     for edge in mes:
89         x.remove(edge)
90         g.remove_edges_from(x)
91         if approx.node_connectivity(g, s, t)==0:
92             rem.append(edge)
93         g=graph.copy()
94         x=mes.copy()
95
96     return len(rem)==0
```

4.2.4 Example of Graph Viewer

Figure 4.1 shows an example of Graph Viewer application. Figure 4.1 (a) displays how the window looks with edge-weak graph (the graph drawn in figure is that in Figure 3.2). Figure 4.1 (b) displays how the window looks with not edge-weak graph.

(a)



(b)

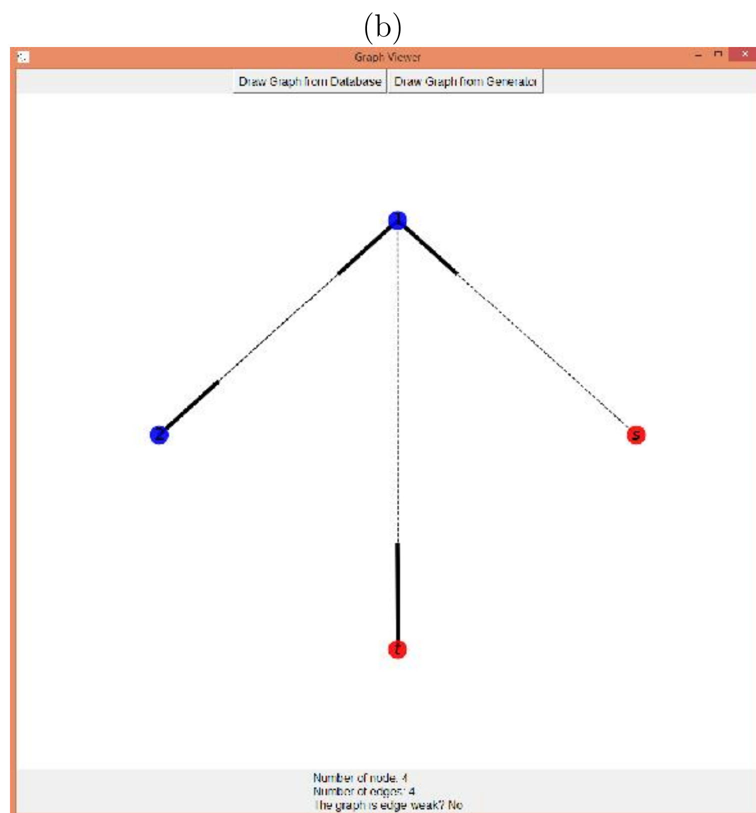


Figure 4.1: Examples of Graph Viewer: (a) the graph is edge-weak, (b) the graph is not edge-weak

4.3 Remarks

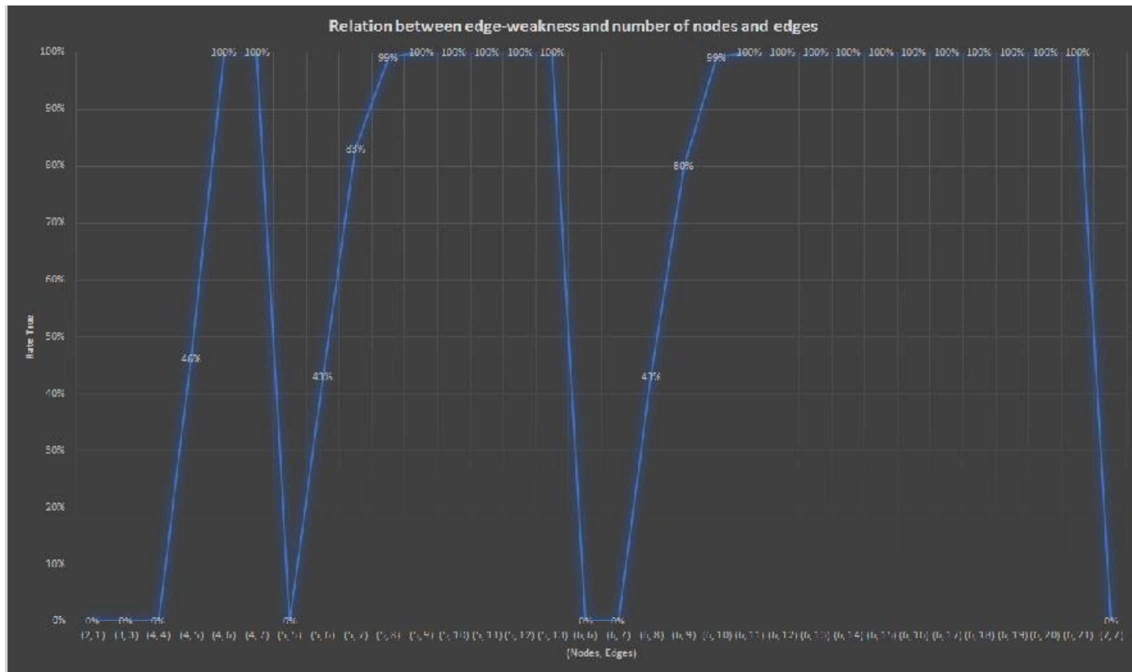


Figure 4.2: Diagram representing the relation between edge-weakness and amount of nodes and edges. The horizontal axis is labeled with pairs of nodes-edges and the vertical axis is labeled with the probabilities (in percentages) of the graph to be edge weak

The diagram in Figure 4.2 shows the results of the experiment conducted on 2719 graphs. Such an experiment concerns the execution of the function `is_edge_weak(graph)` on the above mentioned graphs for measuring the relation between edge-weakness and the amount of nodes and edges.

Looking at graphs with n nodes and varying on the number of edges, most of all graphs are generated and the following results are achieved:

- 4 **nodes and** 4 **edges:** 9 \ 9 graphs are not edge-weak;
- 4 **nodes and** 5 **edges:** 7 \ 13 graphs are not edge-weak and 6 \ 13 graphs are edge-weak;
- 4 **nodes and** 6 **edges:** 7 \ 7 graphs are edge-weak;
- 4 **nodes and** 7 **edges:** 1 \ 1 graph is edge-weak;
- 5 **nodes and** 5 **edges:** 66 \ 66 graphs are not edge-weak;

- 5 **nodes and** 6 **edges:** 157 \ 277 graphs are not edge-weak and 120 \ 277 graphs are edge-weak;

- 5 **nodes and** 7 **edges:** 103 \ 607 graphs are not edge-weak and 504 \ 607 graphs are edge-weak;

- 5 **nodes and** 8 **edges:** 6 \ 774 graphs are not edge-weak and 768 \ 774 graphs are edge-weak;

- 5 **nodes and** 9 **edges:** 596 \ 596 graphs are edge-weak;

- 5 **nodes and** 10 **edges:** 278 \ 278 graphs are edge-weak;

- 5 **nodes and** 11 **edges:** 78 \ 78 graphs are edge-weak;

- 5 **nodes and** 12 **edges:** 11 \ 11 graphs are edge-weak;

- 5 **nodes and** 13 **edges:** 1 \ 1 graph is edge-weak.

- 6 **nodes and** 6 **edges:** 504 \ 504 graphs are not edge-weak.

- 6 **nodes and** 7 **edges:** 2431 \ 4243 graphs are not edge-weak and 1812 \ 4243 graphs are edge weak.

- 6 **nodes and** 8 **edges:** 4169 \ 20038 are not edge-weak graphs are not edge-weak and 15869 \ 20038 graphs are edge weak.

- 6 **nodes and** 9 **edges:** 2697 \ 60759 are not edge-weak graphs are not edge-weak and 58062 \ 60759 graphs are edge weak.

- 6 **nodes and** 10 **edges:** 624 \ 126642 are not edge-weak graphs are not edge-weak and 126018 \ 126642 graphs are edge weak.

- 6 **nodes and** 11 **edges:** 379926 \ 379926 graphs are not edge-weak.

- 6 **nodes and** 12 **edges:** 1139778 \ 1139778 graphs are not edge-weak.

- 6 **nodes and** 13 **edges:** 200000 \ 200000 graphs are not edge-weak.

- 6 **nodes and** 14 **edges:** 76000 \ 76000 graphs are edge-weak.

- 6 **nodes and** 15 **edges:** 37470 \ 37470 graphs are edge-weak.

- 6 **nodes and** 16 **edges:** 14639 \ 14639 graphs are edge-weak.

- 6 **nodes and** 17 **edges:** 4337 \ 4337 graphs are edge-weak.

- 6 **nodes and** 18 **edges:** 1057 \ 1057 graphs are edge-weak.
- 6 **nodes and** 19 **edges:** 177 \ 177 graphs are edge-weak.
- 6 **nodes and** 20 **edges:** 14 \ 14 graphs are edge-weak.
- 6 **nodes and** 21 **edges:** 1 \ 1 graph is edge-weak.
- 7 **nodes and** 7 **edges:** 4198 \ 4198 graphs are not edge-weak.

Therefore, it is possible to note a linear behaviour in the number of nodes and edges: with the increase of the edges in respect of nodes, the probability that a graph is edge-weak increases. Such a probability is 0 if the number of nodes and edges is the same and it becomes certain with the maximum amount of edges.

Chapter 5

Conclusion

The dissertation shows from a graph theoretic prospective a type of inefficiency, called edge-weakness, in the standard flow network model. It also shows a polynomial time algorithm for checking this property. The algorithm iteratively builds a chain of mes and iteratively checks wheter the mes is critical or not. The problem is, if the mes is not critical then, for every edge e in the mes, the algorithm computes an e-minimal mes and checks if it is critical. Moreover, the algorithm elaborated for computing the minimal mes, is not very functioning. Since the computational complexity of the algorithm for testing edge-weakness is $O(n^8)$, a more efficient algorithm could be developed.

Acknowledgements

I would first like to thank Prof. Daniele Gorla for his valuable guidance which inspired me to do always better, and for his assistance: the door of his office was always open whenever I ran into a trouble spot or had some questions.

Finally, I must express my profound gratitude to my parents for providing me with unfailing support and continuous encouragement; to my sister for the patience she has me displayed in listining my thesis presentation over and over again and to all my friends. The road is already long and you have to support me at least until the acknowledgements of the master degree dissertation. So don't worry.

Thank you all.

Bibliography

[1] https://www.python.org.

[2] D. S. Aric Hagberg and P. Swart. *NetworkX Reference Release 2.0.dev20170818153659*. Aug. 18, 2017.

[3] H. Ariyoshi. Cut-set graph and systematic generation of separating sets. *IEEE Transactions on Circuit Theory*, 19:233 – 240, 1972.

[4] M. G. Bell and Y. Lida. *Transportation Network Analysis*. Wiley, 1987.

[5] A. Bondy and U. Murty. *Graph Theory*. Springer-Verlag, London, 2008.

[6] P. Cenciarelli, D. Gorla, and I. Salvo. Graph theoretic investigations on inefficiencies in network models. Available at https://arxiv.org/abs/1603.01983.

[7] P. Cenciarelli, D. Gorla, and I. Salvo. Graph theoretically comparing inefficiecies in network models.

[8] P. Cenciarelli, D. Gorla, and I. Salvo. *Depletable Channels: Dynamics and Behaviour*, pages 50 – 61. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[9] R. Cerulli, M. Gentili, and A. Iossa. Efficient preflow push algorithms. *Computers Operations Research*, 35(8):2694 – 2708, 2008. Queues in Practice.

[10] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[11] C. Demetrescu, I. Finocchi, and G. Italiano. *Algoritmi e Strutture Dati*. McGraw-Hill, 2004.

[12] D. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, USA, 2010.

[13] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *J. ACM*, 35(4):921–940, Oct. 1988.

[14] T. Kloks and D. Kratsch. Listing all minimal separators of a graph. *SIAM Journal on Computing*, 27(3):605–613, 1998.

[15] B.-F. W. Li-Pu Yeh and H.-H. Su. Efficient algorithms for the problems of enumerating cuts by non-decreasing weights. *Algorithmica*, 56:297 – 312, 2010.

[16] T. L. M. Ravindra K. Ahuja and J. B. Orlin. *Network Flows: Theory, Algorithms and Applications*. Prentice-Hall, New Jersey, 1993.

[17] N. Robertson, P. D. Seymour, and in collaboration with National Science Foundation (U.S.). *Graph Structure Theory: Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference on Graph Minors, Held June 22 to July 5, 1991, withSupport from the National Science Foundation and the Office of Naval Research*. American Mathematical.

[18] H. Shen and W. Liang. Efficient enumeration of all minimal separators in a graph. *Theor. Comput. Sci.*, 180(1-2):169–180, 1997.

[19] H. O. Shuji Tsukiyama, Isao Shirakawa and H. Ariyoshi. An algorithm to enumerate all cutsets of a graph in linear time per cutset. *Journal of the ACM (JACM)*, 27:619–632, 1980.

[20] R. Steinberg. Traffic flows on transportation networks. *Networks*, 13(1):156–157, 1983.