Minimum Convex Piecewise Linear Cost Tension Problem on Quasi-k Series-Parallel Graphs

Bruno Bachelet, Philippe Mahey

Minimum Convex Piecewise Linear Cost Tension Problem on Quasi Series-Parallel Graphs

Bruno Bachelet[1] and Philippe Mahey[2]
LIMOS, UMR 6158-CNRS,
Université Blaise-Pascal, BP 10125, 63173 Aubière, France.

Research Report LIMOS/RR03-19

[1]bachelet@isima.fr - http://frog.isima.fr/bruno
[2]mahey@isima.fr

Abstract

This article proposes an extension, combined with the out-of-kilter technique, of the aggregation method (that solves the minimum convex piecewise linear cost tension problem, or CPLCT, on series-parallel graphs) to solve CPLCT on quasi series-parallel graphs. To make this algorithm efficient, the key point is to find a "good" way of decomposing the graph into series-parallel subgraphs. Decomposition techniques, based on the recognition of series-parallel graphs, are thoroughly discussed.

Keywords: minimum cost tension, series-parallel graph, graph decomposition, series-parallel recognition.

Résumé

Cet article propose une extension, combinée avec la technique de mise-à-conformité, de la méthode d'agrégation (qui résout le problème de tension minimum à coûts convexes linéaires par morceaux, ou CPLCT, sur des graphes série-parallèles) afin de résoudre CPLCT sur des graphes quasi série-parallèles. Pour rendre cet algorithme efficace, le point clé est de trouver une "bonne" décomposition du graphe en sous-graphes série-parallèles. Des techniques de décomposition, basées sur la reconnaissance de graphes série-parallèles, sont discutées en détails.

Mots clés : tension de coût minimum, graphe série-parallèle, décomposition de graphe, reconnaissance de graphe série-parallèle.

Acknowledgements / Remerciements

This project was partially funded by France-Brazil cooperation project CAPES-COFECUB 398/02.

Abstract

This article proposes an extension, combined with the out-of-kilter technique, of the aggregation method (that solves the minimum convex piecewise linear cost tension problem, or CPLCT, on series-parallel graphs) to solve CPLCT on quasi series-parallel graphs. To make this algorithm efficient, the key point is to find a "good" way of decomposing the graph into series-parallel subgraphs. Decomposition techniques, based on the recognition of series-parallel graphs, are thoroughly discussed.

Keywords: minimum cost tension, series-parallel graph, graph decomposition, series-parallel recognition.

Introduction

[4] proposed an *aggregation* method to solve the *minimum convex piecewise linear cost tension* problem (or *CPLCT*) on *series-parallel* graphs (or *SP-graphs*). It was shown to be competitive on this class of graphs with the best *dual cost-scaling* algorithms (see [1]). This article proposes to combine the aggregation method with the *out-of-kilter* approach (cf. [10]) to provide an efficient method to solve CPLCT on a slightly more general family of graphs called *quasi series-parallel*.

Let $\pi : X \longmapsto \mathbb{R}$ be a function that assigns a potential to each node of the graph $G = (X; U)$. Let $m = |U|$ and $n = |X|$. The tension θ_u of an arc $u = (x; y)$ is the difference of potentials $\theta_u = \pi_y - \pi_x$ and is constrained to $\theta_u \in [a_u; b_u] \subset \mathbb{R}$. CPLCT can be modeled as a linear program:

$$(P) \begin{cases} \text{minimize} & \displaystyle\sum_{u \in U} c_u(\theta_u) \\ \\ \text{with} & \pi_y - \pi_x = \theta_{(x;y)},\ \forall (x; y) \in U \\ \\ & a_u \leq \theta_u \leq b_u,\ \forall u \in U \end{cases}$$

where c_u are convex piecewise linear functions. In this article, they will be defined as follows:

$$c_u(\theta_u) = \begin{cases} c_u^1(o_u - \theta_u)\ \text{, if } \theta_u < o_u \\ c_u^2(\theta_u - o_u)\ \text{, if } \theta_u \geq o_u \end{cases}$$

This problem is formally related to a minimum cost flow problem by duality (cf. [1]). It arises for instance in the synchronization of hypermedia documents where each document u has an elastic duration θ_u that can be adjusted around a referential value o_u (see [6]).

Section 1 proposes an overview of the SP-graphs and the aggregation method. It also defines *quasi* SP-graphs, that are not perfectly series-parallel. Section 2 presents an extension of the aggregation technique, called *reconstruction*, for quasi SP-graphs. To be efficient, this new algorithm rely on a "good" decomposition of the graph into SP-subgraphs. This is the key point of the procedure discussed in Sections 3 and 4. Section 5 presents comparative numerical results.

1 Aggregation Method

1.1 Series-Parallel Graphs

A common definition of series-parallel graphs is based on a recursive construction of the graphs (e.g. [8], [9], [13]) that is very intuitive and close to the way synchronization constraints are built in a hypermedia document. A graph is *series-parallel*, also called *SP-graph*, if it is obtained from a graph with only two nodes linked by an arc, applying recursively the two following operations:

- The *series composition*, applied upon an arc $u = (x; y)$, creates a new node z and replaces u by two arcs $u_1 = (x; z)$ and $u_2 = (z; y)$ (cf. Figure 1a). We call *series* the relation that binds u_1 and u_2 and denote it $u_1 + u_2$.
- The *parallel composition*, applied upon an arc $u = (x; y)$, duplicates u by creating a new one $v = (x; y)$ (cf. Figure 1b). We call *parallel* the relation that binds u and v and denote it $u//v$.

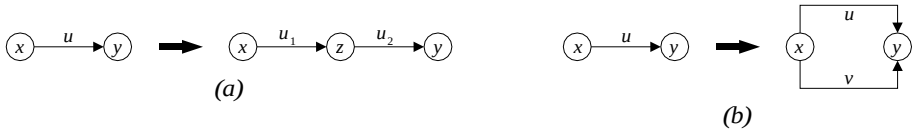


Figure 1: Series and parallel compositions.

The series and parallel relations are gathered under the term *SP-relations*. During the construction process, a SP-relation that binds two arcs can become a relation between two series-parallel subgraphs. Hence, the term *single SP-relation* is introduced to identify a SP-relation between two arcs. From the recursive definition of a SP-graph, it is easy to verify that a SP-graph has always at least a single SP-relation (the SP-relation created from the last composition).

The SP-relations are binary operations, so we can represent a SP-graph by a binary tree called *decomposition binary tree* or *SP-tree* (cf. [13], [7]). Figure 2 shows a SP-tree of a SP-graph. Section 3 explains how to find such a tree in linear time.

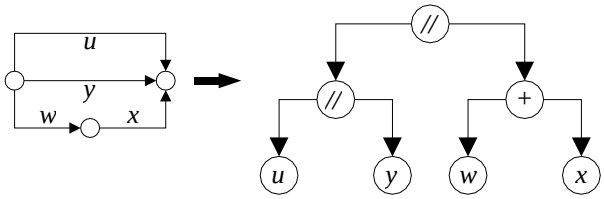


Figure 2: Example of SP-tree.

Numerical results (cf. [4]) showed that linear programming and the out-of-kilter method take advantage of the particular structure of the SP-graphs and behave really better on this class of graphs. However the dual cost-scaling approach does not work that well on these instances, whereas it proves to be the most efficient for non-specific graphs. Moreover, the aggregation method presents the best performance on SP-graphs.

1.2 Aggregation Method

The *aggregation* method, that allows to solve CPLCT only on SP-graphs, has been introduced in [4]. The algorithm works on a SP-tree T of the SP-graph G and is recursive: considering a SP-relation in T, it assumes that the optimal tensions of the two subgraphs implied in the relation are known, and from them, it is possible to quickly build the optimal tension of the whole SP-relation. Hence, starting from the leaves of T, the optimal tension of each SP-relation is built to finally reach the root of the tree T.

From the definition of a SP-graph, it is obvious that a SP-graph has only one source node and only one target node. Hence, the *main tension* $\overline{\theta}$ of a SP-graph is defined as the tension between its

source s and target t, i.e. $\overline{\theta} = \pi_t - \pi_s$. To get an efficient algorithm, the *minimum cost function* C_G of a SP-graph G must be defined. This function represents the cost of the optimal tension where the main tension is forced to a given value:

$$C_G(x) = \min\{\sum_{u \in U} c_u(\theta_u) \mid \theta \in T_G,\ \bar{\theta} = x\}$$

As each function c_u is convex, the minimum cost function is indeed convex. Let us consider two series-parallel subgraphs G_1 and G_2, and suppose that their minimum cost functions C_{G_1} and C_{G_2} are known. The minimum cost function $C_{G_1+G_2}$ of the SP-relation $G_1 + G_2$ is:

$$C_{G_1+G_2}(x) = \min_{x=x_1+x_2} C_{G_1}(x_1) + C_{G_2}(x_2)$$

Thus, $C_{G_1+G_2}$ is the inf-convolution $C_{G_1} \square C_{G_2}$. It is well known that this operation maintains convexity (e.g. [11]). The minimum cost function $C_{G_1//G_2}$ of the SP-relation $G_1//G_2$ is:

$$C_{G_1//G_2}(x) = C_{G_1}(x) + C_{G_2}(x)$$

Thus, $C_{G_1//G_2}$ is simply the sum $C_{G_1} + C_{G_2}$, which is convex if C_{G_1} and C_{G_2} are convex. From this assessment, a simple recursive algorithm can be proposed to build the minimum cost function C_G of a SP-graph G. As explained in [4], it is preferred to deal with the *t-centered minimum cost function* C_G^t of G that makes the method both more efficient and easier to understand:

$$C_G^t(x) = C_G(x + t) - C_G(t)$$

This way C_G^t represents the minimum additional cost to increase or decrease the main tension $\overline{\theta}_G$ from a reference tension t. If $t = \theta_G^*$, the minimum cost tension of a graph G, the cost function $C_G^* = C_G^{\overline{\theta_G^*}}$ equals 0 at the optimal tension.

Moreover we are interested in finding the minimum cost tension of G, thus the aggregation method provides the minimum cost function C_G^* with additional information on each of its pieces (cf. [4] for details). It allows for instance to reach optimally a new main tension t' from the reference tension θ_G^* in linear time (precisely $O(m)$ operations). This facility is widely used in the further reconstruction approach. The whole aggregation method performs in $O(m^3)$ operations.

1.3 Quasi SP-Graphs

A *quasi SP-graph* or *QSP-graph* $G = (X; U)$ is such that the removal of a minimal subset $U' \subset U$ of arcs from G makes the remaining graph $G' = (X; U \setminus U')$ series-parallel. The ratio $|U'|/|U|$ is called the *SP-perturbation* of the graph G. This value indicates how many arcs of G are disturbing the series-parallel property of G. From this definition, any connected graph is a QSP-graph, but we prefer to use this term for graphs with a small SP-perturbation (in applications issued from the hypermedia field, $10\ \%$ seems a satisfying threshold).

2 Reconstruction Approach

As the aggregation method can not be used "as it is" on QSP-graphs and the dual cost scaling approach proves to be less efficient than the out-of-kilter on SP-graphs, we propose in this section a method called *reconstruction* to solve CPLCT on QSP-graphs. This new approach combines the aggregation and the out-of-kilter techniques based on a *SP-decomposition* of the graph.

2.1 Decomposition Phase

We call *series-parallel component* or *SP-component* of a graph G a SP-subgraph of G. A partition P of the arcs of G defines a *series-parallel decomposition* or *SP-decomposition* of G, where each set of arcs in P induces a SP-subgraph of G. A SP-decomposition of G is thus a set of arc-disjoint SP-components, whose union is G. Section 3 discusses ways to obtain a "good" SP-decomposition that fits the reconstruction process.

The reconstruction method starts with the search of a SP-decomposition D of the graph G. Then CPLCT is solved on each SP-component of D with the aggregation method. Thus, for each component $D_u \in D$, its minimum cost function C_u^* and its optimal tension θ_u^* are known, so D_u can be seen as a single aggregated arc u with a convex piecewise linear cost function $c_u = C_u^*$ and a tension $\theta_u = \theta_u^*$.

2.2 Reconstruction Phase

The method attempts next to put the SP-components back together. An iterative process consists in adding one by one the aggregated arcs into a new graph, rebuilding the original graph G back. Starting with $H^0 = (X^0; U^0) = (\emptyset; \emptyset)$, at each step an aggregated arc $u = (x; y)$ is added, i.e. $H^k = (X^{k-1} \cup \{x; y\}; U^{k-1} \cup \{u\})$. The newly added arc u is the only one that may be out-of-kilter.

We remind that the kilter curve is defined from the tension θ_u and the flow φ_u of the arc u, and roughly that φ_u must equal ${c_u}'(\theta_u)$ for the arc to be in-kilter. If all the arcs of H^k are in-kilter, its tension θ is optimal. The idea of the out-of-kilter approach is to bring all the arcs on their kilter curve (e.g. [2]).

The newly added arc u may be out-of-kilter because even if its tension is optimal, its flow must be set to 0 in order to keep the flow conservation constraints on the whole graph H^k. Fortunately, as this arc is the only one out-of-kilter, repetitive search for a cycle or a cocycle to modify respectively either the flow or the tension of the arc u can be performed in $O((A + B)m)$ operations, where A is the maximum tension and B the maximum flow (i.e. the maximum derivative of its cost function) for the arc u. Once the arc is in-kilter, the whole tension on H^k is optimal and another aggregated arc v can be added.

2.3 Tension Adjustment

But adding an arc into the graph is not that obvious. Consider the aggregated arc $u = (x; y)$, the optimal tension θ_u of u may not be equal to the difference of potentials $\pi_y - \pi_x$. The tension of the arc u has to be optimally adjusted to be equal to $\pi_y - \pi_x$. That can be achieved using the minimum cost function C_u^* of the arc u to adapt optimally the main tension of the SP-component aggregated behind u. As said in Section 1, it can be performed in $O(m)$ operations (cf. [4]).

2.4 SP-Component Splitting

For the need of the reconstruction, we assume a partial order on the components of the SP-decomposition D, such that if the *source* node (i.e. without any predecessor) and/or the *target* node (i.e. without any successor) of a SP-component D_u belong to component D_v, then $D_v < D_u$. Decomposition methods presented in Section 3 prove that such an order exists for any graph. During the reconstruction phase, the SP-components are added following this partial order.

Another problem arises when adding an aggregated arc $u = (x; y)$ into the graph H^k: maybe the source node x and/or the target node y of u are not present in H^k, because simply hidden in one

of the aggregated arcs v of the graph H^k. Hence we need to split the SP-component behind such an arc v to make the node visible. The easiest way is to bring back all the arcs of the SP-component D_v into the graph H^k, (i.e. $H^k = H^{k-1} \cup D_v \setminus \{v\}$).

The drawback of this technique is that the biggest SP-component, added behind a single aggregated arc at the first step, is certainly split at the second step, because nodes will obviously be needed for the secondly added arc. This approach reveals to be extremely inefficient, as most of the arcs will be brought back in the graph at the second step only. Thus, we need a smarter way than splitting totally a SP-component. This is the purpose of Section 4 that proposes a technique to split a SP-component into several pieces, reducing their numbers to the minimum necessary.

2.5 Conservation of the In-Kilter Property

After the splitting of an aggregated arc $u = (x; y)$, the resulting arcs must be in-kilter to keep the whole graph H^k optimal. Thus, knowing their tension, it is straightforward to find an interval in which their flow must be: indeed, with our assumptions, the kilter curve is a step function (cf. Figure 3).

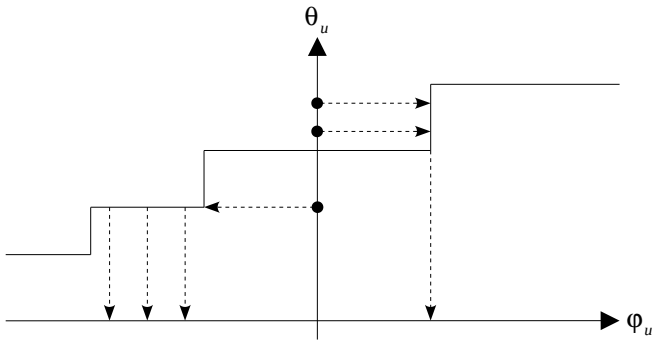


Figure 3: Kilter flow building.

For each arc v of the SP-component D_u, an interval $[e_v; f_v]$ can thus be defined. Let suppose an arc w from the source node x to the target node y of the SP-component with a capacity $[\varphi_u; \varphi_u]$, where φ_u is the flow of u in the whole graph H^k. Find a flow φ_v for each arc v of the SP-component D_u turns to find a feasible flow φ in the SP-component (with the added loop arc w). We propose to solve it in $O(m^3)$ operations with a technique using the SP-tree of the SP-component like the aggregation (cf. [3]). However, as this phase is not critical in terms of time consumption, any well-known method is suitable to solve this flow problem.

2.6 Complexity

Algorithm 1 summarizes the whole process of the reconstruction method. Let $k = |D|$ and p_u be the number of arcs of a SP-component D_u. The reconstruction algorithm needs, for each SP-component of D, eventually a splitting (with the tension adjustment and the conservation of the in-kilter property), an aggregation ($O(p_u^3)$ operations) and an out-of-kilter iteration ($O(m(A+B))$ operations). The whole splitting phase need $O(\min\{n; 2k\}m^3)$ operations (cf. Section 4). Thus, the reconstruction method requires $O(\min\{n; 2k\}m^3 + km(A + B))$ operations.

Algorithm 1 *Reconstruction method.*

```
find SP-decomposition D = {D_u}_{u=1...k};
let H be an empty graph;

for all SP-component D_u ∈ D do
 find minimum cost tension for D_u using aggregation;
 let u = (x; y) be the aggregated arc of D_u;

 while ∃D_v with D_v < D_u and (x ∈ D_v or y ∈ D_v) do
  split D_v (add each arc and node of D_v in H);
  find tension for each arc of D_v using C*_v;
  find flow for each arc of D_v;
 end while;

 add arc u in H;
 find minimum cost tension for H using out-of-kilter;
end for;

for all aggregated arc u ∈ H do
 split D_u (add each arc and node of D_u in H);
 find tension for each arc of D_u using C*_u;
end for;
```

3 Series-Parallel Decomposition

The recognition of a SP-graph is known for long as an easy problem that can be solved in linear time (cf. [13]). The various algorithms proposed by many authors can immediately be adapted and without altering their complexity to build the SP-tree during the recognition phase. These methods being very efficient, our discussion will not be on their complexity, but on the way they recognize a SP-graph, our goal being to find for any graph the "best" SP-decomposition suitable for the reconstruction process.

The characterization of such a decomposition is not that obvious. If it is too compact, i.e. with few SP-components, that will tend to produce a lot of splittings during the reconstruction (like the phenomenon explained for the first SP-component in Section 2). At the opposite if the decomposition is scattered, i.e. with many SP-components, the aggregation phase will become useless and the reconstruction will tend to a single out-of-kilter algorithm.

We choose to study here two recognition methods, the *reduction* and the *path* approaches, and propose heuristic extensions to find a "good" SP-decomposition. To present these algorithms, a recursive notation of the SP-trees is introduced. A tree with a root a, a left subtree T_l and a right subtree T_r is represented by $(a; T_l; T_r)$. The reverse operations of the series and parallel compositions, called *SP-reductions*, are defined as follows:

- The *series* reduction, noted S_x^{-1}, replaces the SP-relation $(y; x) + (x; z)$ by an arc $(y; z)$.
- The *parallel* reduction, noted P_u^{-1}, replaces the SP-relation $u//v$ by the arc u.

3.1 Reduction Approach, Recognition

This recognition approach is widespread because very intuitive. As explained in Section 1, a SP-graph has at least one single SP-relation. The idea is to find one, apply the associated SP-reduction and repeat until the graph is reduced to a single arc. If there is no more single SP-relation and the graph is not a single arc, that means the graph is not series-parallel.

This method has been proposed first in [13]. It appears then in [12] with some improvement: all the multiple arcs are first removed (no single parallel relation exist anymore), then series and

parallel-and-series (composition of a parallel relation with a series relation) relations are detected and reduced. The detection is easier (only the nodes are checked) and the reduction is more efficient (because with the parallel-and-series relation, two arcs and one node are removed in one step). However, this improvement does not change fundamentally the way the algorithm performs. A similar variant is proposed in [5] which proposes 18 reductions, most of them working only on non-directed graphs. These improvements are difficult to take into account with our goal of finding a SP-decomposition, because our graphs have non-specific structures, so the occurrences of these special reductions dedicated to SP-graphs will be rare.

3.2 Reduction Approach, Decomposition

A generic version of the method is presented in Algorithm 2. It builds at the same time a SP-decomposition of the graph where each SP-component is represented by its SP-tree. For this purpose, a function t is defined that associates a SP-tree t_u with each arc u of the graph G. At the beginning, each arc possesses a SP-tree with a single node that is the arc itself. We sketch below what happens during a SP-reduction.

- The series reduction S_x^{-1} removes the two arcs $u = (y; x)$ and $v = (x; z)$ of the relation $u + v$, and creates an arc $w = (y; z)$. The SP-tree of w is then $(+; t_u; t_v)$, the old t_u and t_v are removed.
- The parallel reduction P_u^{-1} removes the arc v of the relation $u//v$. The SP-tree of u becomes $(//; t_u; t_v)$, the old t_u and t_v are removed.

Algorithm 2 *SP-decomposition, with the reduction approach.*

```
for all arc u ∈ U do t_u ← (u; ∅; ∅);

while |U| ≠ 1 do
 if relation (y; x) + (x; z) found then apply SP-reduction S_x^{-1};
 else if relation u//v found then apply SP-reduction P_u^{-1};
 else
  find u with the biggest score and the smallest SP-tree;
  remove u from G;
 end if;
end while;

all the remaining trees are the components of a SP-decomposition;
```

To recognize a SP-graph, the method reduces successively each single SP-relation until there is either only one arc in the graph or no more single SP-relation to reduce. To find a whole SP-decomposition of any graph, the algorithm repeats until it blocks, so an arc must be removed to reveal SP-relations and allow to continue the reductions. When suppressing an arc, no parallel relation can appear, only series relations can. So the intuitive idea, whenever a blocking occur, is to remove the arc that reveals the most series relations, and if none exist the one that will contribute in revealing the most series relations. We propose a heuristic approach that assigns two scores to each node x: one used when x is source of an arc (s_x^+) and another used when x is the target of an arc (s_x^-). The score of an arc $u = (x; y)$ is $s_x^+ + s_y^-$, and is calculated as follows:

$$s_x^+ = \begin{cases} -M, \text{ if } d_x^+ = 1 \\ 0, \text{ if } d_x^+ > 1 \text{ and } d_x^- = 0 \\ 1/(d_x^+ - 1) + 1/d_x^-, \text{ if } d_x^+ > 1 \text{ and } d_x^- > 0 \end{cases}$$

$$s_x^- = \begin{cases} -M, \text{ if } d_x^- = 1 \\ 0, \text{ if } d_x^- > 1 \text{ and } d_x^+ = 0 \\ 1/(d_x^- - 1) + 1/d_x^+, \text{ if } d_x^- > 1 \text{ and } d_x^+ > 0 \end{cases}$$

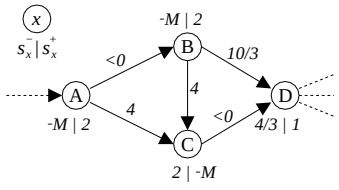


Figure 4: Example of scoring.

Here d_x^+ represents the number of arcs outgoing x and d_x^- the number of arcs incoming x. Intuitively, a node x with $d_x^+ = 2$ and $d_x^- = 1$ is the best candidate to be the source of the arc to remove because it reveals a series relation, its score is 2. The same way, an arc y with $d_y^- = 2$ and $d_y^+ = 1$ is the best candidate to be the target of the arc to remove. The best candidate arc has a score of 4. This score tends to decrease first for the extremities of the arcs having important numbers of incoming and outgoing arcs, because that means numerous arc suppressions are needed before they can become series relations. Removals that alter the connectivity of the graph are penalized with a negative score, hence if a node would have no incoming or outgoing arc after the suppression, its score is set to $-M$, with M such that even the best score for the other extremity will not make the score of the arc positive, any $M > 2$ is suitable. Figure 4 shows an example of scoring. In the case of scores equality, the arc that hides the smallest SP-component is removed, with the hope of revealing finally the biggest SP-component. If we consider no specific data structure for the graph, the complexity of Algorithm 2 is $O(m^2)$ operations, at each step the selection of the arc to be removed requires $O(m)$ operations (to assign a score to each remaining arc of the graph).

3.3 Path Approach, Recognition

The paths of a SP-graph are organized in a very specific way. In [9], this is formalized by the concept of *ear decomposition*. We propose here a quite different approach based on two kinds of nodes: the *branching nodes* (with more than one outgoing arc) and the *synchronization nodes* (with more than one incoming arc).

In a SP-graph, such nodes represent respectively the beginning and the end of parallel relations. We call *branching* two arcs outgoing a branching node (i.e. the beginning of a parallel relation). In the same way, a *synchronization* represents two arcs incoming a synchronization node (i.e. the end of a parallel relation). By mean of clarity, we sometimes identify a branching (respectively synchronization) by its branching (respectively synchronization) node.

The approach proposed here is based on a search through the graph in the topological order of the nodes (i.e. a node is visited only after all its predecessors). Let S_k be the set of nodes marked during the process until iteration k. A branching will be said *closed* at iteration k if there is a synchronization node $y \in S_k$ such that two arc-distinct directed paths P_1 and P_2 between x and y exist and contain each one of the two arcs of the branching; the two arcs incoming y that belong each to one of the paths form a synchronization y that *closes* x. A closed branching is associated with the first synchronization encountered during the search that closes it. The paths that allow the closure of the branching (in our definition P_1 and P_2) are called the *closure* paths of the branching.

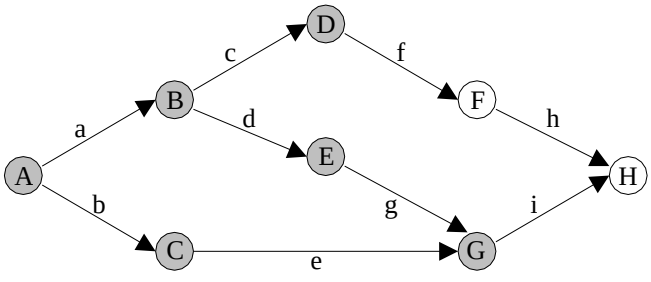


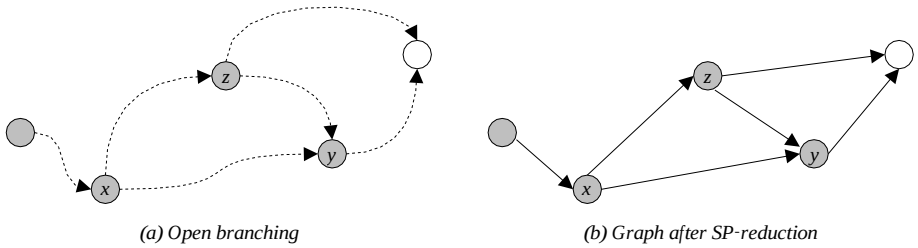
Figure 5: Example of branchings.

To illustrate these definitions, consider Figure 5. The gray nodes represent visited nodes at iteration k. A and B are branching nodes, G and H are synchronization nodes. $(a; b)$ is a closed

branching, the associated synchronization is $(e; g)$. On the contrary, branching $(c; d)$ is open. Closure paths for branching $(a; b)$ are $(a; d; g)$ and $(b; e)$. The graph of the example is not series-parallel. Intuitively, the problem comes from the open branching $(c; d)$. Let us prove the following proposition.

Proposition 1 *A graph is series-parallel if and only if at each iteration of the topological search through the graph, for each closed branching x, there are two closure paths between the branching x and its associated synchronization y that do not possess any open branching.*

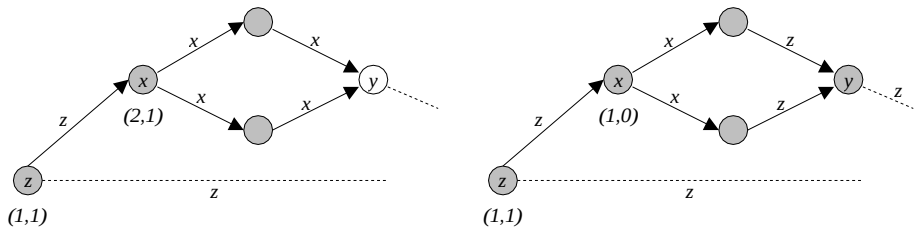
Proof: ($\Rightarrow$) At a given iteration, suppose there is an open branching z on one of the closure paths of a closed branching x associated with its synchronization y. That gives a graph with the general aspect presented by Figure 6a. Attempting a reduction with Algorithm 2 (with no arc suppression), the best that can be found is the graph illustrated by Figure 6b. The reduction can thus not be terminated without any arc suppression, so the graph is not series-parallel.
($\Leftarrow$) Suppose now that no such open branching exist, the recognition Algorithm 3 presented further proves that it is possible to find (with no arc suppression) the SP-tree associated with the graph, so it is series-parallel.



(a) Open branching *(b) Graph after SP-reduction*

Figure 6: Example of open branching on closure path.

To verify the proposition (in order to determine if a graph is series-parallel), we propose a marking process that allows to identify, arriving on a synchronization, the branchings that are closing and if a branching is open in one of the closure paths. Let Δ_u be the signature of an arc u, $\Delta_u = x$ means x is the last (according to the topological order of the nodes) open branching between the source node of the graph and the arc u. Let Δ_x be the signature of a node x, it is a couple $\Delta_x = (l_x; d_x)$. If the node is not a branching node then $d_x = 0$, else d_x indicates the number of branchings located at node x that are open. l_x indicates the level of branching, e.g. if $l_x = 2$ then there are two open branchings (including x) in the paths between the source node of the graph and the node x.



(a) Iteration k *(b) Iteration k+1*

Figure 7: Example of signature change.

The signature of an arc or a node can change while searching the graph in the topological order of the nodes. When two arcs u and v with the same signature x (at iteration k) meet at a

synchronization node y (at iteration $k+1$), they form a synchronization and closes thus a branching of node x (cf. Figure 7a). The signature of their branching is then modified: d_x is decreased of 1. If d_x equals 0, that means all the branchings of node x are closed. The signature of the arcs u and v must change, it becomes the open branching before x, i.e. the signature of one of the incoming arcs of branching x (cf. Figure 7b). Ideally, the signature of all the arcs of signature x must change, but for the needs of the algorithm (that searches the graph in the topological order of the nodes and thus never come back on already used arcs) this modification is not necessary.

Algorithm 3 *SP-decomposition, with the path approach.*

```
for all arc u ∈ U do t_u ← (u;∅;∅);

while |U| ≠ 1 do
 choose x in X such that all predecessors of x are visited yet;

 if d_x^- = 0 then Δ_x ← 0;
 else
  if d_x^- > 1 then closeBranchings(x,Δ,t);
  let u = (b;x) ∈ U;

  if d_x^+ = 1 then apply SP-reduction S_x^{-1};
  else Δ_x ← Δ_b + 1;
 end if;
end while;
```

Algorithm 3 proposes to search the graph following the topological order of the nodes, marking progressively the nodes and the arcs with the signature Δ. It is supposed that the graph has only one source and at least one arc, else it is sure that it is not series-parallel. From the source of the graph, the algorithm marks the nodes and the arcs of the graph. At each synchronization node y, it modifies certain signatures to represent the closure of associated branchings (cf. Algorithm 4). If at the end of this step, two incoming arcs u_1 and u_2 are not marked with the same signature, that means the graph is not series-parallel, because there are two paths P_1 and P_2 between the source and respectively u_1 and u_2. These paths have at least one branching node in common (at least the source). Let x be the last node (in the topological order) verifying the condition. If there were no open branching on the closure paths between x and y, u_1 and u_2 should have the same signature x. If they do not, that means at least one of the closure paths between x and y has an open branching, which signature is carried by u_1 or u_2.

Algorithm 4 *Branchings closure.*

```
let y be the synchronization node;
D ← {u = (x;y) ∈ X};
sort D in the decreasing order of the signatures;

while |D| > 1 do
 let u_1 = (x_1;y) and u_2 = (x_2;y) the two first arcs of D;
 if x_1 ≠ x_2 then G is not series-parallel; stop;
 apply SP-reduction P_{u_1}^{-1};
 D ← D \ {u_2};

 if d_{x_1}^+ = 1 then
  apply SP-reduction S_{x_1}^{-1} and let v be the resulting arc;
  D ← D \ {u_1} ∪ {v};
  sort D in the decreasing order of the signatures;
 end if;
end while;
```

Algorithm 4 performs the closure of the branchings at a given synchronization node y, and modifies consequently the signatures. It must check two-by-two the arcs incoming y and if they have identical signatures, close the corresponding branching if not already done. For an efficient search among the arcs, we propose to sort them in a set D, in the decreasing order of their signatures, i.e. u before $v \Rightarrow l_{\Delta_u} > l_{\Delta_v}$ or ($l_{\Delta_u} = l_{\Delta_v}$ and $\Delta_u > \Delta_v$) (we suppose any order on the

nodes, the idea is that arcs with the same signature are side-by-side). Hence, when looking the first two arcs of D, the priority is given to the branching with the highest level, which must absolutely be closed before any lower level branching. This process requires $O(m \log m)$ operations. So the whole Algorithm 3 requires $O(nm \log m)$ operations.

Algorithm 3 can easily be adapted, without altering its complexity, to build also the SP-tree associated with the graph, as in the reduction approach. The idea consists in reducing the graph with a series reduction each time a node with only one incoming and one outgoing arcs is visited. When a branching is closed, that means a parallel operation has been detected, it must then be reduced. The SP-tree is built once the whole graph is visited (there will remain then only a single arc). The management of the signatures of the arcs and the nodes is then facilitated, because the SP-reductions imply automatically the signatures. For instance, the signature Δ_x of a node x is limited to l_x (because d_x is always equal to $d_x^+ - 1$), and the signature of an arc $u = (x; y)$ is not necessary anymore, because it is always equal to x.

3.4 Path Approach, Decomposition

This approach offers a new way of recognition of a SP-graph, based on a single search through the graph in the topological order of the nodes. We explain now how to modify this method to build a SP-decomposition of the graph. With this algorithm, there are two ways to discover that a graph is not series-parallel. First, during the search through the nodes according to their topological order, if a circuit exists, the process loops (in Algorithm 3). But the circuit contains one of the nodes visited at last (i.e. nodes for which the successors have not been all visited yet), let S be the set of these nodes. In fact, no more nodes can be visited because two nodes at least of S are predecessors of each other, directly or indirectly. The process will be then to find such a circuit by looking one by one the incoming arcs of each node of S. Once the circuit is found, the implied incoming arc is removed and the recognition algorithm can go on.

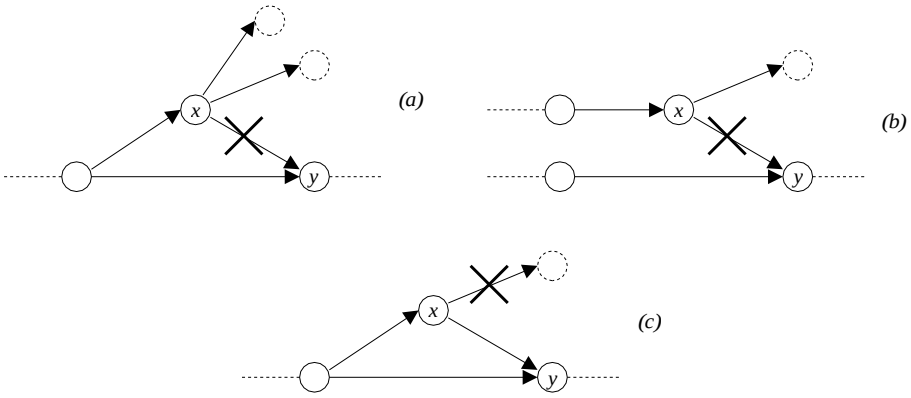


Figure 8: Examples of cases for arc deletion.

Another way to find that a graph is not series-parallel is when it is impossible to establish a synchronization y, i.e. an incoming arc $u = (x; y)$ has a signature different of the signatures of all the other arcs incoming y. Two intuitive possibilities remain then: either remove this arc (cf. Figure 8a) or if x has precisely two outgoing arcs (u and another v) as in Figures 8b and 8c, verify that the suppression of v can not offer to u the same signature than one of the other incoming arcs of y. Thus, we check that the signature of the incoming arc of x is identical with at least one of the signatures of the incoming arcs of y. To sum up, when an arc $u = (x; y)$ blocks the recognition, either it is removed, or its neighbor at branching x is removed, which allows the synchronization of u with one of its neighbor at synchronization y. These modifications to build

the SP-decomposition of the graph does not alter the complexity of the recognition Algorithm 3, because it is only punctual checks at the signatures of nodes.

4 SP-Component Splitting

4.1 Reconstruction Method Extension

As explained in Section 2.4, the reconstruction approach adds one by one the aggregated arcs in the graph H^k. However, when adding a new arc $u = (x; y)$, x and/or y may not exist, because hidden behind an aggregated arc of H^k. The node x is said *inside* the component $D_v = (X_v; U_v)$ if and only if $x \in X_v$ and is neither the source nor the target of D_v.

Suppose now that x is inside the SP-component D_v. The first idea in Section 2.4 was to remove the aggregated arc v from H^k and replace it by the whole component D_v (i.e. all the arcs of D_v are added into H^k, this way x is not inside a component anymore), but this approach is not making the reconstruction very efficient as it does not benefits really from the aggregation.

The second idea we present here is to minimally split the component D_v, i.e. find a SP-decomposition E_v of D_v so x is not inside a SP-component anymore with $|E_v|$ minimal, in order to preserve aggregated arcs as long as possible during the reconstruction process. In this case, D_v will be replaced in H^k by E_v where each component of E_v will be represented by an aggregated arc.

Algorithm 5 *Reconstruction method, with the minimal splitting approach.*

```
find SP-decomposition D = {D_u}_{u=1...k};
let H be an empty graph;

for all SP-component D_u ∈ D do
 find minimum cost tension for D_u using aggregation;
 let u = (x; y) be the aggregated arc of D_u;

 while ∃D_v with D_v < D_u and (x ∈ D_v or y ∈ D_v) do
  find minimal splitting of D_v into E_v;
  find tension for each arc of D_v using C*_v; [1]
  find flow for each arc of D_v; [2]

  for all component D_w ∈ E_v do
   find minimum cost tension for D_w using aggregation;
   find main tension θ_w of D_w in H (using values from [1]);
   find main flow φ_w of D_w in H (using values from [2]);
   add arc w aggregating D_w with tension θ_w and φ_w in H;
  end for;
 end while;

 add arc u in H;
 find minimum cost tension for H using out-of-kilter;
end for;

for all aggregated arc u ∈ H do
 split D_u (add each arc and node of D_u in H);
 find tension for each arc of D_u using C*_u;
end for;
```

Algorithm 5 proposes modifications for the reconstruction Algorithm 1 of Section 2, in order to consider a minimal splitting of the SP-components instead of their whole splitting. When a component is split, the flow and tension of all its arcs are still computed. And for each SP-component D_w of the splitting, its cost function C^*_w is determined using the aggregation method. To insert the arc w aggregating D_w into the graph H^k, w must have a valid flow and tension. They are computed directly from the values (worked out from *[1]* and *[2]* in Algorithm 5) of the single arcs in the component D_w.

4.2 Minimal Splitting

This section presents a technique to perform the minimal splitting of a SP-component. First, we can notice that x will always originate from a series composition, simply because parallel compositions never create a new node. (1) The first step will be to identify in the SP-tree T_v associated with the SP-component D_v which series composition generates x. (2) Then the two subgraphs of this series composition are extracted from D_v, they form thus the two first SP-components of E_v. However by the way they have been extracted, their source and target nodes (except x) stay inside D_v, so the problem has been moved from x to two other nodes z_1 and z_2. (3) The last step consists thus in going up the SP-tree until the root, and split some of the series compositions so that z_1 and z_2 (and later the nodes these new splittings will generate) so they are not inside any SP-component.

4.2.1 Phase 1: Splitting Pivot Search

The series composition that generates x must be identified. The SP-tree is explored from its root to its leaves, taking care of memorizing (using for instance a stack) the path that leads us to an arc a that has x as source or target. Once this arc is identified, we follow the path back to the root of the SP-tree, but during this trip we will try to find which series composition generates x. Supposing x inside D_v, the idea of Algorithm 6 is that if x is the source of a, then a is part of the right member of a series composition. On the opposite, if x is the target of a, then a is part of the left member of a series composition. Algorithm 6 results with p as the *pivot* of the splitting, i.e. the SP-tree of the series composition that generates x.

Algorithm 6 *Splitting pivot search.*

```
push T_v and 0 in S;
f ← 0;

while S ≠ ∅ and f = 0 do
 let t = (r; t_l; t_r) and i be the top of S;
 i ← i + 1;

 if i = 1 then [ First visit of t. ]
  if r is an arc (y; z) then
   if x = y then f ← 1;
   if x = z then f ← 2;
   pop t and i from S;
  else push t_l and 0 in S;
 else if i = 2 then [ Left subtree of t visited. ]
  push t_r and 0 in S;
 else [ Right subtree of t visited. ]
  pop t and i from S;
 end if;
end while;
```

```
p ← ∅;
if f = 0 then stop;

while S ≠ ∅ and p ≠ ∅ do
 let t = (r; t_l; t_r) and i be the top of S;

 if r = + then
  let t' = (r'; t'_l; t'_r) be the parent of t;
  if f = 1 and t = t'_r then p = t';
  if f = 2 and t = t'_l then p = t';
 end if;

 if p = ∅ then pop t and i from S;
end while;

if p ≠ ∅ then
 x is inside D_v;
 p is the pivot of the splitting;
end if;
```

4.2.2 Phase 2: First Splitting

Now the pivot is identified, the SP-tree T_v has to be split. We can extract p_l and p_r, the subtrees of p, from T_v. If the parent of p is a parallel composition or if p is the root of T_v, T_v remains a

SP-component after the extraction. But if the parent of p is a series composition, T_v will not be a SP-tree anymore.

Algorithm 7 *First splitting.*

```
while S ≠ ∅ do
 pop t = (r; t_l; t_r) and i from the top of S;
 let t' = (r', t'_l; t'_r) be the parent of t;

 if r = + then
  if t' = ∅ then apply PS1(t) (with a fictive parent for t);
  else if r' = // then
   apply PS1(t);
   stop;
  else if t = t'_l then apply RR1(t);
  else apply LR1(t);
 end if;
end while;
```

The idea of the second phase is to go on following the path up to the root of T_v, while neither a parallel composition, nor the root of T is found. Once it happens, the last series composition t encountered is split by the $PS1$ (*"Parallel Splitting I"*) operation illustrated by Figure 9, its two subtrees b and c being the first components of E_v.

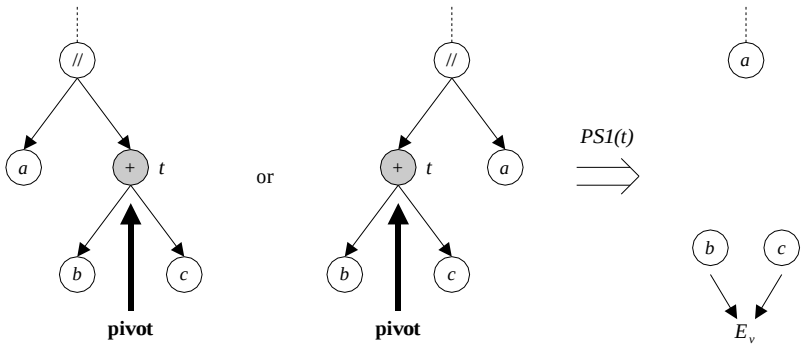


Figure 9: Parallel Splitting I.

But t is not necessary the pivot of the splitting for x, hence on the path up to the root of T_v, for each series composition, rotations must be performed to maintain the pivot as the current series composition. Algorithm 7 uses, depending on the situation, the operation $LR1$ (*"Left Rotation 1"*, cf. Figure 10) or the operation $RR1$ (*"Right Rotation I"*, cf. Figure 11) to perform the rotations.

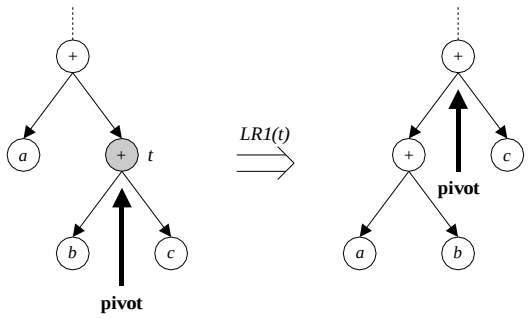


Figure 10: Left Rotation I.

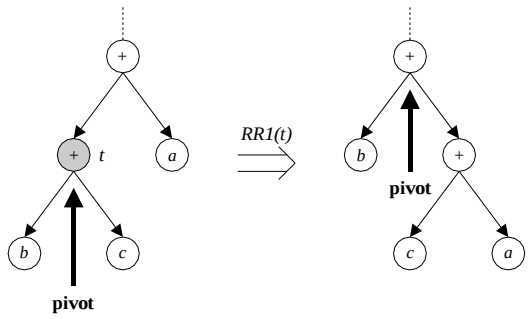


Figure 11: Right Rotation I.

4.2.3 Phase 3: Second Splitting

During $PS1(t)$, two subtrees b and c are extracted from T_v. x is no more inside T_v, because it is target of b and source of c. If t is the root of the SP-tree T_v, then the splitting is over, but on the other cases, the source node z_1 of b and the target node z_2 of c are inside the remains of T_v. To get them out, the parallel peer a of the subtree $b + c$ (cf. Figure 9) has to be extracted too, depending on the situation, with the operation LS (*"Left Splitting"*, cf. Figure 12) or the operation RS (*"Right Splitting"*, cf. Figure 13).

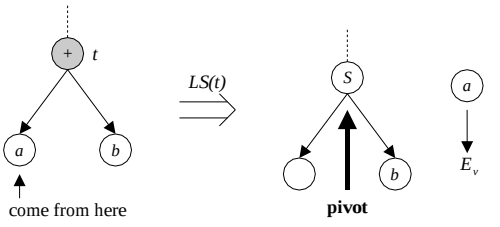


Figure 12: Left Splitting.

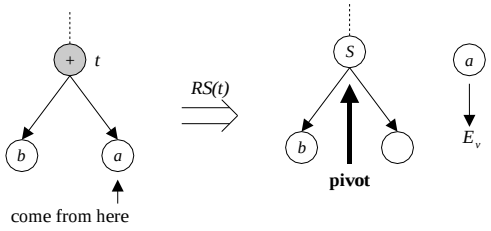


Figure 13: Right Splitting.

Now there is a "hole" in the SP-tree. But this hole is not different of the one made if x were removed in phase (2). So like we maintained the current series composition t as pivot of x along the path back to the root of T_v in phase (2), Algorithm 8 maintains the current series composition t as pivot of the "hole". Depending on the situation, operation $LR2$ (*"Left Rotation II"*, cf. Figure 14) or operation $RR2$ (*"Right Rotation II"*, cf. Figure 15) are used to perform rotations in the SP-tree T_v.

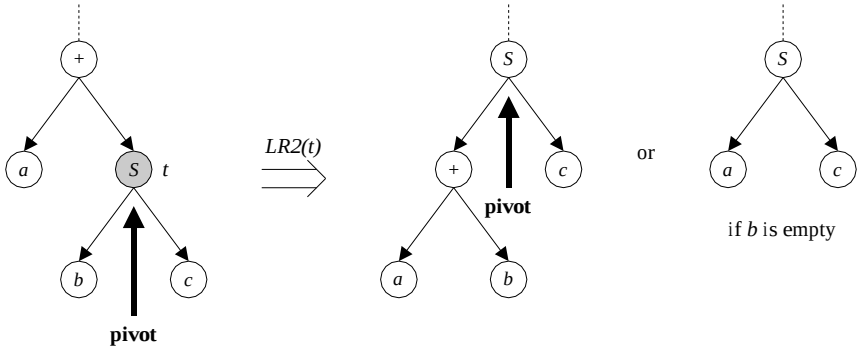


Figure 14: Left Rotation II.

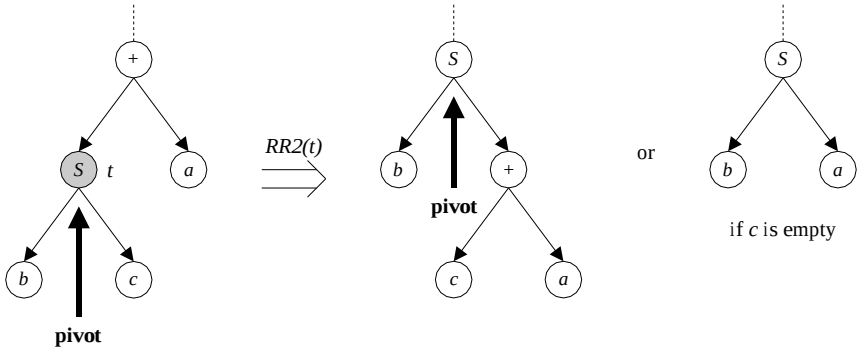


Figure 15: Right Rotation II.

As in phase (1), once a parallel composition is found, the subtrees b and c of the last series composition t encountered on the path back up to the root of T_v are extracted from T_v to become SP-components of E_v, using the operation $PS2$ (*"Parallel Splitting II"*, cf. Figure 16). Once this splitting achieved, the "hole" problem remains, hence the process of phase (3), cf. Algorithm 8, loops until the root of T_v is reached. The operations $LR2$, $RR2$ and $PS2$ of phase (3) are quite different from the operations $LR1$, $RR1$ and $PS1$ of phase (2), only because one of the subtrees (b or c) of the current series composition t can be empty (due to the splitting from LS or RS at the beginning of the phase).

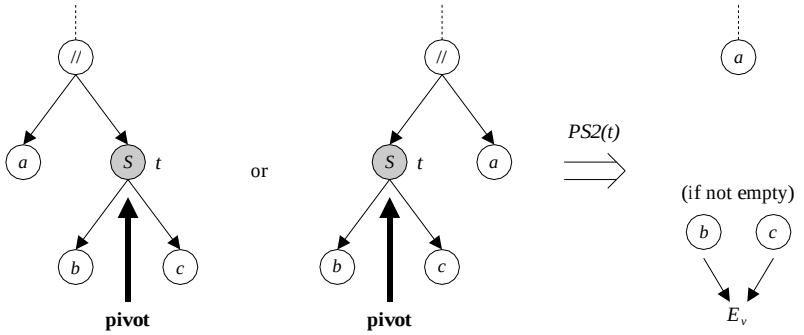


Figure 16: Parallel Splitting II.

The whole splitting process, regrouping the three phases, only needs $O(m)$ operations, the worst it can do is to visit all the nodes of the SP-tree T_v, which has $2m-1$ nodes ($m-1$ for the SP-relations and m for the arcs). All the operations $LR1$, $LR2$... need $O(1)$ operations.

Algorithm 8 *Second splitting.*

```
s ← false;

while S ≠ ∅ do
 pop t = (r; t_l; t_r) and i from the top of S;
 let t' = (r', t'_l; t'_r) be the parent of t;

 if r = + then
  if t' = ∅ then apply PS2(t) (with a fictive parent for t);
  else if r' = // then
   apply PS2(t);
   s ← false;
  else if not s then
   if i = 1 then apply LS(t); else apply LR(t);
   s ← true;
  else if t = t'_l then apply RR2(t);
  else apply LR2(t);
 end if;
end while;
```

4.3 Complexity

Consider now the complexity of the whole splitting phase. Let $k = |D|$ and p_u be the number of arcs of a SP-component D_u. The splitting of a SP-component D_v into a SP-decomposition E_v requires $O(p_v)$ operations for the splitting itself, $O(p_v^3)$ operations for the aggregations that follow (because, for each component D_w of E_v, $O(p_w^3)$ operations are needed, but $p_v = \sum_{D_w \in E_v} p_w$, so $p_v^3 > \sum_{D_w \in E_v} p_w^3$), and $O(p_v)$ operations to find the flow and the tension of each SP-component of E_v. To conclude, the complexity of a splitting is $O(p_v^3)$ operations, and it is clear that no more than $\min\{n-2; 2(k-1)\}$ splittings will be needed (no more than one per node and no more than 2 per SP-component of D). Thus, the whole splitting phase requires $O(\min\{n; 2k\}m^3)$ operations.

5 Numerical Results

Tables 1 and 2 present a practical comparison of methods, which is always difficult because of all kinds of biases. But the goal here is to get an idea of how the methods behave on QSP-graphs. Table 1 shows results when the size of the graphs varies and their SP-perturbation is set to $4\ \%$. Table 2 points the performances of the methods for various SP-perturbation, the graph size being set to $n = 500$ and $m = 3000$. Results are expressed in seconds on a Celeron 500 MHz processor under a Linux operating system. We used GNU C++ 2.95 compiler and its object-oriented features to implement the methods. The results are means of series of 10 tests on randomly generated graphs. Both A and B are fixed to 1000. The cardinality of the SP-decomposition, for both the reduction and the path approaches, is also presented.

Nodes	Arcs	Kilter	Dual	Reconstruction (Total Split)		Reconstruction (Min Split)		Components	
			Cost-Scaling	Reduction	Path	Reduction	Path	Reduction	Path
50	100	0.01	0.02	0.01	0.01	0.02	0.02	5	6
50	200	0.02	0.03	0.03	0.03	0.05	0.05	9	10
50	300	0.03	0.04	0.04	0.04	0.08	0.08	13	16
50	400	0.04	0.07	0.06	0.06	0.12	0.10	16	19
50	500	0.06	0.08	0.07	0.08	0.16	0.13	21	23
100	200	0.03	0.06	0.03	0.03	0.05	0.05	12	11
100	400	0.07	0.09	0.07	0.07	0.13	0.12	20	21
100	600	0.10	0.15	0.12	0.12	0.20	0.18	29	31
100	800	0.16	0.18	0.20	0.18	0.30	0.24	37	39
100	1000	0.23	0.26	0.25	0.24	0.41	0.34	43	48
500	1000	0.67	0.84	0.71	0.70	0.47	0.44	55	61
500	2000	1.48	1.57	2.11	1.97	1.21	0.95	107	116
500	3000	2.31	2.20	3.43	3.35	1.90	1.38	153	170
500	4000	3.21	3.24	4.58	4.63	2.65	1.97	198	220
500	5000	4.21	3.84	5.93	5.88	3.36	2.25	245	278
1000	2000	2.34	2.65	3.35	3.21	1.41	1.22	110	124
1000	4000	5.27	4.54	7.83	7.70	3.46	2.70	212	227
1000	6000	8.17	7.16	12.47	12.78	5.39	3.95	310	344
1000	8000	11.43	8.70	19.27	19.03	8.03	5.62	411	450

Table 1: Numerical results, graph size influence.

SP-Perturbation	Kilter	Dual	Reconstruction (Total Split)		Reconstruction (Min Split)		Components	
(%)		Cost-Scaling	Reduction	Path	Reduction	Path	Reduction	Path
2	1,85	2,28	1,80	1,87	1,37	1,13	79	85
3	2,02	2,30	2,62	2,55	1,68	1,27	117	129
4	2,21	2,45	3,20	3,07	1,85	1,45	154	166
5	2,43	2,34	3,82	3,70	2,04	1,55	196	211
6	2,63	2,27	4,50	4,49	2,35	1,78	225	242
7	2,80	2,11	5,38	5,10	2,66	2,03	264	279
8	2,83	2,12	5,45	5,48	2,80	2,26	299	321
9	3,07	2,11	6,02	5,68	3,04	2,42	333	360
10	3,27	2,12	6,82	6,42	3,42	2,78	369	393
11	3,31	2,18	6,86	6,85	3,49	2,97	401	423
12	3,49	2,14	7,65	7,39	4,05	3,40	435	466
13	3,52	2,16	7,84	7,60	4,07	3,59	467	499
14	3,88	2,11	8,67	8,34	4,61	3,98	495	525
15	3,86	2,06	8,94	8,63	4,81	4,41	535	568
20	4,47	2,02	11,70	11,38	6,69	6,43	690	719
30	5,95	1,91	17,14	16,06	11,27	10,72	1000	1036
40	6,41	1,89	20,39	19,16	14,92	14,41	1305	1333

Table 2: Numerical results, SP-perturbation influence.

The reconstruction method appears to be more efficient when the path approach is used to decompose. In fact, the reduction technique provides a more compact SP-decomposition that reveals less adapted for the reconstruction. Moreover, the minimal splitting appears to be a key to the good performances of the reconstruction algorithm. To conclude, the efficiency of the various

methods depends on both the SP-perturbation and the size of the graph. The reconstruction method is well suited for a SP-perturbation below $8\ \%$ and for large QSP-graphs, whereas the dual cost-scaling method is better suited for non-specific graphs or small QSP-graphs.

Conclusion

This article proposes a new algorithm to solve CPLCT on QSP-graphs that proves to be competitive with existing methods. It also describes a new way of recognizing a SP-graph and proposes two heuristic approaches to decompose a graph into SP-subgraphs. To find a "best" SP-decomposition of a graph needs to be formulated, e.g. to minimize the number of SP-components, and the complexity of the problem discussed.

References

[1] R.K. Ahuja, D.S. Hochbaum, and J.B. Orlin. Solving the Convex Cost Integer Dual Network Flow Problem. In *Management Science*, volume 49, pages 950–964, 2003.

[2] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin. *Network Flows - Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[3] B. Bachelet. *Modélisation et optimisation de problèmes de synchronisation dans les documents hypermédia*. PhD thesis, Université Blaise Pascal, Clermont-Ferrand, France, 2003.

[4] B. Bachelet and P. Mahey. Minimum Convex-Cost Tension Problems on Series-Parallel Graphs. Technical Report RR03-06, LIMOS, Université Blaise Pascal, Clermont-Ferrand, France, 2003.

[5] H.L. Bodlaender and B. de Fluiter. Parallel Algorithms for Series Parallel Graphs. In *4th Annual European Symposium on Algorithms*, pages 277–289. Springer-Verlag, 1996.

[6] M.C. Buchanan and P.T. Zellweger. Specifying Temporal Behavior in Hypermedia Documents. In *European Conference on Hypertext '92*, pages 262–271, 1992.

[7] A.K. Datta and R.K. Sen. An Efficient Scheme to Solve Two Problems for Two-Terminal Series Parallel Graphs. In *Information Processing Letters*, volume 71, pages 9–15. Elsevier Science, 1999.

[8] R.J. Duffin. Topology of Series-Parallel Networks. In *Journal of Mathematical Analysis and Applications*, volume 10, pages 303–318, 1965.

[9] D. Eppstein. Parallel Recognition of Series-Parallel Graphs. In *Information and Computation*, volume 98-1, pages 41–55, 1992.

[10] J.M. Pla. An Out-of-Kilter Algorithm for Solving Minimum Cost Potential Problems. In *Mathematical Programming*, volume 1, pages 275–290, 1971.

[11] R.T. Rockefellar. *Convex Analysis*. Princeton University Press, 1970.

[12] B. Schoenmakers. A New Algorithm for the Recognition of Series Parallel Graphs. Technical report, No CS-59504, Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands, 1995.

[13] J. Valdes, R.E. Tarjan, and E.L. Lawler. The Recognition of Series Parallel Digraphs. In *SIAM Journal on Computing*, volume 11-2, pages 298–313, 1982.