Centrum voor Wiskunde en Informatica

REPORT *RAPPORT*

A new algorithm for the recognition of series parallel graphs

L.A.M. Schoenmakers

Computer Science/Department of Algorithmics and Architecture

CS-R9504 1995

Report CS-R9504
ISSN 0169-118X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

A New Algorithm for the Recognition of Series Parallel Graphs

Berry Schoenmakers

CWI

P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

In this paper we develop a new linear-time algorithm for the recognition of series parallel graphs. The algorithm is based on a succinct representation of series parallel graphs for which the presence of an arc can be tested in constant time; space utilization is linear in the number of vertices. We show how to compute such a representation in linear time from a breadth-first spanning tree. Furthermore, we present a precise condition for the existence of such succinct representations in general, which is, for instance, satisfied by planar graphs.

AMS Subject Classification (1991): 05C85
CR Subject Classification (1991): G.2.2
Keywords & Phrases: Graph algorithms, series parallel graphs, decomposition trees, succinct representation.

1. Introduction

The class of series parallel graphs forms a subset of the class of directed multigraphs with a single source and sink. A common definition of this class is as follows (see, e.g., [3, 4, 10]). The graph consisting of two vertices s and t, say, connected by an arc (s, t) is the *basic* series parallel graph; vertex s is the source and vertex t is the sink. *Compound* series parallel graphs can be obtained from two smaller ones, say G and H, according to two composition rules.

- Series composition: identify the sink of G with the source of H.
- Parallel composition: identify the source of G with the source of H, and identify the sink of G with the sink of H.

We remark that this class of graphs is also called the class of *two-terminal* series parallel graphs (because the graphs contain a single source and sink).

Given a directed multigraph G, the recognition problem is to determine whether G is series parallel or not. This problem has received quite some attention and several

E-mail address: `berry@cwi.nl`

Report CS-R9504
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

linear-time solutions are known (see, e.g., [9, 10, 11]). Also some parallel recognition algorithms have been presented [5, 4].

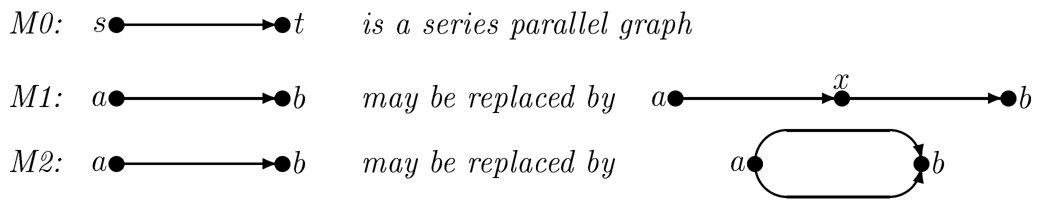
Our algorithm compares favourably in simplicity with these known algorithms. For instance, compared to [9, 10], the use of pointers is less pronounced. The algorithm is based on a succinct representation of series parallel graphs for which the presence of an arc can be tested in constant time. We show how to compute such a representation in linear time from a breadth-first spanning tree. Furthermore, we present a precise condition for the existence of such succinct representations in general, which is, for instance, satisfied by planar graphs. Also, we discuss a possible generalization to other recognition problems. We show that there exists a succinct representation when a graph is sufficiently "sparse" and leave as an open problem how to compute it in linear time, say.

2. Preliminaries

A first attempt at solving the recognition problem is to follow the above definition of series parallel graphs: if the input graph is not the basic series parallel graph, we have to check whether it can be obtained from two smaller graphs, either by series composition or by parallel composition. If neither applies then we may conclude that the input graph is not series parallel. Otherwise, we have to check the two smaller graphs recursively. The problem with this approach is that such large structures have to be manipulated, which complicates the design of an efficient algorithm.

It seems that the problem becomes more tractable if we take the following equivalent definition.

Definition 1 *The class of series parallel graphs are those graphs that can be obtained from the following set of rules, in which x denotes a "fresh" vertex.*

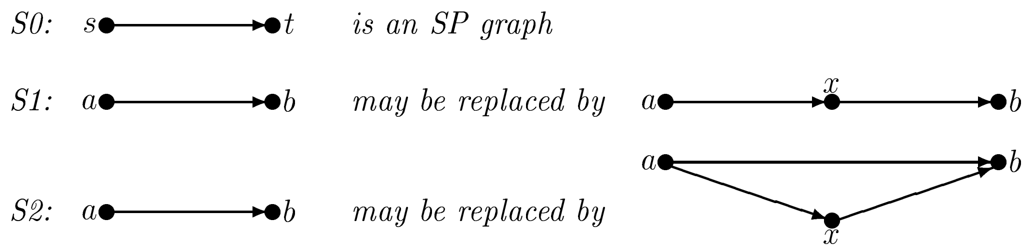


Roughly speaking, this definition says how series parallel graphs can be constructed by splitting an arc into two arcs either in series ("chopping") or in parallel ("doubling").

Comparing these definitions we see that Definition 1 describes how series parallel graphs are transformed locally to obtain larger ones, while the definition in Section 1 describes how to combine them to get larger ones. It is not difficult to prove that these definitions are indeed equivalent. Note that series composition corresponds to chopping and that parallel composition corresponds to doubling.

A direct consequence of rule M2 is that the multiplicity of arcs can be ignored when determining whether a graph is series parallel or not. For convenience we will therefore restrict the recognition problem to graphs without multiple arcs. We introduce the class of *SP graphs* as the series parallel graphs without multiple arcs.

Definition 2 *The class of series parallel graphs without multiple arcs (SP graphs, for short) are those graphs that can be obtained from the following set of rules, in which x denotes a "fresh" vertex.*



As rule S2 consists of an application of rule M2 followed by an application of rule M1, an SP graph is clearly series parallel. An interesting property of SP graphs is that these graphs are necessarily sparse. A simple proof by induction shows that $m \leq 2n - 3$ for an n-vertex SP graphs with m arcs. In the following we show how to exploit this fact to get a succinct representation of SP graphs.

3. Towards an Efficient Algorithm

We first derive a simple algorithm that forms the basis of our approach. Using Definition 2 we determine whether a graph G is SP or not. If G is not the basic SP graph, we check whether it contains the right-hand side of rule S1 or S2 (or both). That is, we check whether G contains vertices with exactly one predecessor and one successor. If this is the case, any such vertex may be removed by reducing G according to the appropriate rule. We then repeat the process on the reduced graph. If such vertex does not exist, then we conclude that G is not SP.

Since series parallel graphs are acyclic, and since it is well-known how to test for acyclicity in linear time, we assume that the input graph G is acyclic. We introduce graph $H = (V, E)$ as a variable and take as main invariant:

$$H \text{ is acyclic } \wedge \ (G \text{ is SP } \equiv \ H \text{ is SP}).$$

For obvious reasons, a vertex with a single predecessor and successor will be called a redex. Formally, vertex x is a *redex* when $\#P(x) = 1$ and $\#S(x) = 1$, where $P(x) = \{a \in V \mid (a, x) \in E\}$ and $S(x) = \{b \in V \mid (x, b) \in E\}$.

The algorithm is now described as follows. Initially, H is set equal to G, and the main repetition is given by ($z :\in Z$ denotes the assignment to z of an arbitrary element of Z):

$$
\begin{array}{ll}
\textbf{do } X \neq \emptyset \; \rightarrow & \\
\quad x :\in X; & \\
\quad a :\in P(x); & \\
\quad b :\in S(x); & \\
\quad \textbf{if } (a,b) \notin E \; \rightarrow E := E \cup \{(a,b)\} & \{\text{rule S1}\} \\
\quad [\,]\; (a,b) \in E \; \rightarrow \text{skip} & \{\text{rule S2}\} \\
\quad \textbf{fi}; & \\
\quad V, E := V \setminus \{x\}, E \setminus \{(a,x),(x,b)\} & \\
\textbf{od} &
\end{array}
$$

Note that vertices a, b, and x are mutually distinct because H is acyclic. In other words, $X \neq \emptyset$ implies that $\#V \geq 3$, so the repetition is executed at most $n-2$ times. Upon termination, graph G is SP if and only if graph H is equal to the basic SP graph.

We refine the algorithm by choosing a representation for H and keeping track of set X. Below, we describe how this can be done using $O(n^2)$ space. The subject of the following sections is how to refine this algorithm using $O(n)$ space only.

To implement assignment $a :\in P(x)$ it suffices to record *any* predecessor for each vertex x in H; once $x \in X$, a will be the unique predecessor of x in H. A similar account holds for assignment $b :\in S(x)$. For this purpose, we use arrays *pred* and *suc*, and maintain as an additional invariant:

$$
\begin{array}{lcl}
\#P(x) > 0 & \Rightarrow & (pred[x], x) \in E \\
\#S(x) > 0 & \Rightarrow & (x, suc[x]) \in E,
\end{array}
$$

for all $x \in V$. These arrays can easily be updated by inserting the assignments $pred[b] := a$ and $suc[a] := b$ in the repetition. Also, a special value $\perp \notin V$, say, is used such that $pred[x] = \perp$ when $\#P(x) = 0$, and similarly for *suc*.

To keep track of set X, we introduce array *deg* to record the number of predecessors and successors of each vertex in H. This array can easily be updated for the above operations on V and E. Set X can then be maintained using another array. Note that vertex x is a redex precisely when $deg[x] = 2 \wedge pred[x] \neq \perp \; \wedge suc[x] \neq \perp$.

The only part left is to represent E, and in particular, to implement $(a,b) \in E$ in constant time. An $n \times n$ adjacency matrix is appropriate for this purpose, but we cannot assume that G is given by its adjacency matrix because in that case a linear-time recognition algorithm is impossible. Given any reasonable representation for G, however, we can build its adjacency matrix in time proportional to the number of arcs in G. For this we use a well-known trick from [1, Exercise 2.12] to avoid the initialization of a bit vector (see also [7, pp.289–290],[8, Chapter 7]).

The above method thus requires linear time and quadratic space. The program is easily extended to detect that G is cyclic by checking whether $pred[x] \neq suc[x]$ when vertices are added to X. Also, by recording any arc in E, say (s, t), it is easy to check whether H is equal to the basic SP graph. (That is, upon termination, $H = (V, E)$ is then equal to the basic SP graph when $\#V = 2$, $\#E = 1$, and $s \neq t$.)

4. Succinct Representation of Series Parallel Graphs

As noted before, $m \leq 2n - 3$ for an n-vertex SP graph with m arcs, $n \geq 2$. In the program in the previous section we have used the arrays *pred* and *suc* to represent a *subset* of the arcs of H. A moment's reflection will show that there exists such a related representation for the *complete* set of arcs of any SP graph, using these two arrays only:

Lemma 1 *Any SP graph (V, E) with source s and sink t can be represented using two arrays pred and suc according to the following relation:*

$$(a, b) \in E \;\equiv\; a = pred[b] \;\vee\; suc[a] = b,$$

for any $a \in V \backslash \{t\}$ and $b \in V \backslash \{s\}$.

Proof To represent the basic SP graph with source s and sink t we set $pred[t] := s$ and $suc[s] := t$.

In case the SP graph is extended according to rule S1, the arrays are modified as follows. We set $pred[x] := a$ and $suc[x] := b$. If $suc[a] = b$, then we set $suc[a] := x$. And, if $pred[b] = a$, then we set $pred[b] := x$.

In case the SP graph is extended according to rule S2, it suffices to set $pred[x] := a$ and $suc[x] := b$. □

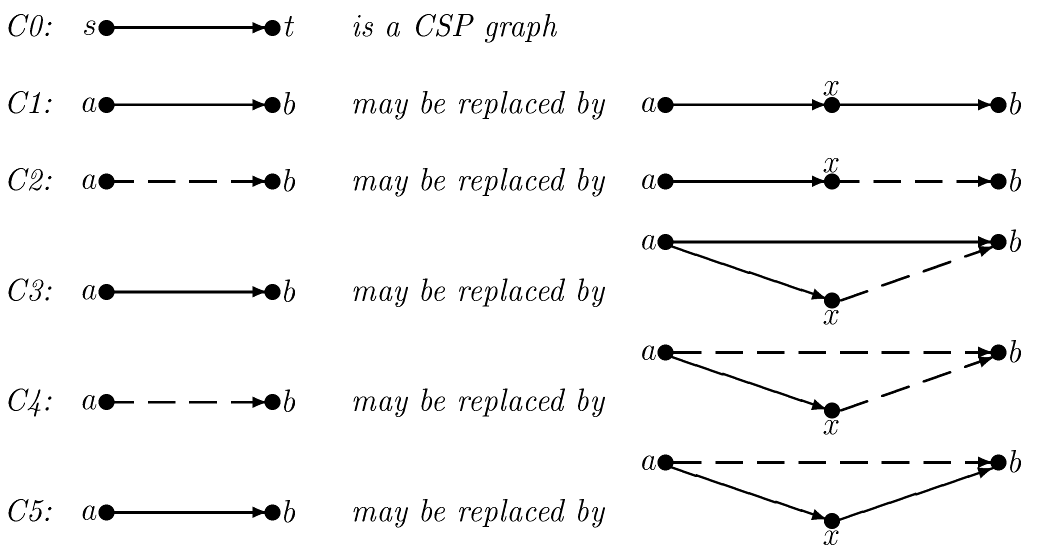
Note that $(pred[x], x) \in E$ for $x \in V \backslash \{s\}$, and that $(x, suc[x]) \in E$ for $x \in V \backslash \{t\}$, for the representation given by Lemma 1. The representation may be adapted slightly to make arrays *pred* and *suc* total. One solution is to define $pred[s] = \perp$ and $suc[t] = \perp$ (with $\perp \notin V$), another solution is to define $pred[s] = s$ and $suc[t] = t$. The relation in Lemma 1 should then be adapted accordingly.

So, we know there is a succinct representation for SP graphs. However, when reducing an arbitrary graph according to rules S1 and S2, it is not trivial how to maintain this representation. Also, it is not clear how we should initialize the arrays *pred* and *suc*. In the next sections these problems will be resolved by studying the spanning trees of series parallel graphs.

5. Series Parallel Graphs and their Spanning Trees

We now investigate the spanning trees of SP graphs. To this end we will color each arc of an SP graph red or blue such that the red arcs form a spanning tree rooted at the source. As will be shown the class of these red-blue SP graphs is exactly equal to the class of *CSP graphs* defined below. Red arcs will be depicted as solid arcs and blue arcs as dashed arcs.

Definition 3 *The class of CSP graphs are those red-blue SP graphs that can be obtained from the following set of rules, in which x denotes a "fresh" vertex.*



The class of CSP graphs are those CSP graphs that can be obtained without using rule C5.*

Lemma 2 *The red arcs of a CSP graph form a spanning tree rooted at the source.*

Proof Note that rules C1–C5 maintain as an invariant that every vertex, except for the source, has exactly one incoming red arc. By rule C0 this holds initially as well. This means that the red arcs form a spanning tree rooted at the source. □

Hence, CSP graphs induce spanning trees for the corresponding SP graphs. The next lemma shows that the converse is also true.

Lemma 3 *If the red arcs of a red-blue SP graph form a spanning tree, then it is a CSP graph.*

Proof Let G be a red-blue SP graph. We prove by induction on the number of vertices that G is a CSP graph.

If G is the basic SP graph, then its only arc must be red, which means that G is equal to the basic CSP graph.

If G is a compound SP graph, then it must contain a vertex x, say, for which either rule S1 or S2 is applicable (in the reverse direction). But, since the red arcs of G form a spanning tree this also means that exactly one of the rules C1–C5 is applicable (in the reverse direction): x has only one, necessarily red incoming arc, and b can have at most one incoming red arc.

By applying the appropriate rule on x, a smaller red-blue graph G' is obtained. Since application of any of the rules C1–C5 in the reverse direction also maintains as an invariant that every vertex, except for the source, has exactly one incoming red arc, the red arcs of G' also from a spanning tree. By the induction hypothesis we thus have that G' is a CSP graph. Then G is also a CSP graph because it can be obtained from G' by applying one of the rules C1–C5. □

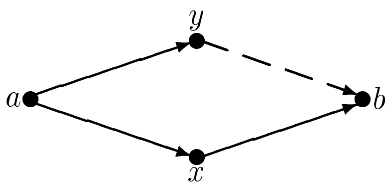
In a CSP graph, each vertex has at most one incoming red arc. The number of outgoing blue arcs, however, is not bounded by a constant. The class of CSP* graphs is important for the following two lemmas.

Lemma 4 *In CSP* graphs each vertex has at most one outgoing blue arc.*

Proof Rules C1–C4 maintain as an invariant that every vertex has at most one outgoing blue arc. By rule C0 this holds initially as well. □

Lemma 5 *If the red arcs of a red-blue SP graph form a breadth-first spanning tree, then it is a CSP* graph.*

Proof On account of Lemma 2 the graph is a CSP graph. Now suppose that the graph is not CSP*, hence that rule C5 has been applied to obtain the graph, and consider the last step in which C5 has been applied. Just after this step the red arcs do not form a breadth-first spanning tree because of the right-hand side of C5: in a breadth-first tree the root path for b should be the shortest path to the root, but, evidently, the root path through x is not the shortest path. In order that the spanning tree becomes breadth-first in the end, rule C2 has to be applied to the blue arc (a, b). (Note that rule C4 does not increase the length of the shortest path between a and b). The result then is, where x is produced by C5 and y is produced by C2:



However, the same result can be obtained by first applying C3 to arc (a, b), and then applying C1 to (a, b). In this way all applications of C5 can be eliminated, which shows that the graph is a CSP* graph. □

Summarizing, we have that if the red arcs of a red-blue SP graph form a breadth-first spanning tree, then any vertex has at most one outgoing blue arc and any vertex, except for the source, has exactly one incoming red arc. This ensures that the succinct representation of Lemma 1 can be computed in linear time.

6. The Reduction Strategy

Next, we investigate how to maintain the succinct representation while reducing a graph. As it turns out to be difficult to maintain the property that the red arcs induce a breadth-first spanning tree, we will maintain the somewhat weaker property that the graph is a CSP* graph (Lemma 5). Due to Lemma 4, this is still sufficient.

In order to achieve this we have to follow a particular *reduction strategy* when reducing a graph. Since a CSP* graph cannot contain a C5 redex, the reduction strategy will ensure that such redexes will not arise if the input graph is SP. To define the strategy we also color the redexes red or blue according to the color of the unique outgoing arc. (Recall that the incoming arc of a redex is always red.) So, C1 and C5 redexes are colored red, and C2–C4 redexes are colored blue. Furthermore, a redex is called *innermost* when its unique predecessor and its unique successor are only connected by a single arc or by other redexes (which are then innermost too).

Definition 4 *Let G be a red-blue graph with at least one redex. Then $red(G)$ denotes any graph obtained from G by applying one of the rules C1–C5 (in the reverse direction), where the redex is selected according to the following* reduction strategy*: red redexes go before blue redexes, and blue redexes should be innermost.*

The intuition behind the reduction strategy is that rule C2 is avoided as much as possible, since application of this rule (in the reverse direction) increases the number of outgoing blue arcs of some vertex (consider vertex a in rule C2). Note that ties between red redexes and ties between innermost blue redexes can be broken arbitrarily.

Lemma 6 *If G is a CSP* graph, then $red(G)$ is also a CSP* graph.*

Proof Clearly, $red(G)$ is a CSP graph, since G is a CSP graph (cf. proof of Lemma 3). To show that $red(G)$ is a CSP* graph we first observe that G cannot contain a C5 redex because it is a CSP* graph. Therefore, each application of rule C5 in a derivation of G must be followed by an application of C2 on the created blue arc. So, if $red(G)$ is obtained from G either by rule C1, C3, or C4, then $red(G)$ is also a CSP* graph, because all applications of C5 then have been cancelled.

The difficult case is when $red(G)$ is obtained from G by rule C2. Let x be the blue redex removed from G, and let a be its predecessor and b its successor. Due to the reduction strategy we then have that G does not contain any red redexes and that x is an innermost redex. So, between vertices a and b there can only be other blue redexes parallel to redex x. This means that the removal of x cannot cancel any application of C5. Hence, $red(G)$ is a CSP* graph also in this case. □

The converse of Lemma 5 is not true in general, that is, the red arcs of a CSP* graphs do not necessarily induce a breadth-first spanning tree. Similarly, it is not true that (in contrast with Lemma 6) the red arcs of $red(G)$ form a breadth-first spanning tree if the red arcs of G do so.

7. An Efficient Algorithm

We begin with an outline of the algorithm. For convenience we assume that the graph is acyclic, as we did in Section 3.

Given an acyclic graph G with n vertices, we first search for a source. From this source we then perform a breadth-first search, in the process building a representation for G using the arrays *pred* and *suc*, where *pred* is used to represent the arcs in the spanning tree and *suc* is used to represent the remaining arcs, if any. (The multiplicity of the arcs in G is ignored.) In case G cannot be successfully represented in this way, the graph cannot be SP due to Lemmas 4 and 5.

Given the succinct representation for G, the arcs of G may be colored red or blue. By adhering to the reduction strategy of Definition 4, we are able to maintain the representation, unless the input graph is not SP.

Initialization

To initialize arrays *pred* and *suc*, let s be any source of G, and set all entries of *pred* and *suc* to $\perp$. Then traverse G in breadth-first order starting from s, in the process updating arrays *pred* and *suc*. To implement the breadth-first traversal, list Q is used as a queue. Note that a vertex has not been visited when its *pred*-value is equal to $\perp$. Also recall that $S(x)$ denotes the set of successors of x in the graph.

The following program fragment thus initializes arrays *pred* and *suc*.

```
Q := [s];
do Q ≠ [] →
    x, Q := head.Q, tail.Q;
    do for each y ∈ S(x) →
        if pred[y] =⊥ → pred[y] := x; Q := Q ++ [y]
            if suc[x] =⊥ → suc[x] := y
            [] suc[x] ≠⊥ → skip
            fi
        [] pred[y] ≠⊥ →
            if suc[x] =⊥ → suc[x] := y
            [] suc[x] ≠⊥ →
                if x = pred[suc[x]] → suc[x] := y
                [] x ≠ pred[suc[x]] → "not SP"
                fi
            fi
        fi
    od
od
```

The difficult case is when $pred[y] \neq \perp \wedge suc[x] \neq \perp$: in this case the value of $suc[x]$ may only be changed when $x = pred[suc[x]]$, i.e., when the arc $(x, suc[x])$ is also represented by array *pred*.

Assuming that every vertex of G is reachable from source s (otherwise the graph cannot be SP), a successful initialization will establish that arrays *pred* and *suc* satisfy:

$$(a, b) \in E \;\equiv\; a = pred[b] \;\vee\; suc[a] = b,$$

for any a that is not a sink and for any $b \neq s$. Given this representation, arc $(a, b) \in E$ is now colored red if $a = pred[b]$ and blue otherwise. That is,

$$\begin{array}{rcl} (a,b) \text{ is red} & \equiv & a = pred[b] \\ (a,b) \text{ is blue} & \equiv & a \neq pred[b] \;\wedge\; suc[a] = b. \end{array}$$

Reduction

As a result of a successful initialization, we have that for each vertex x that is not a source nor a sink:

$$\begin{array}{rcl} (pred[x], x) \in E & \text{and} & (pred[x], x) \text{ is red} \\ (x, suc[x]) \in E & \text{and} & (x, suc[x]) \text{ is red} \;\equiv\; x = pred[suc[x]]. \end{array}$$

If x is a source then $pred[x] = \perp$, and if x is a sink then $suc[x] = \perp$.

The following program fragment, in which x is a redex selected according to the reduction strategy, maintains this representation:

$$\begin{array}{ll}
a, b := pred[x], suc[x]; & \\
e := (a = pred[b] \;\vee\; suc[a] = b); & \\
\textbf{if } \neg e \;\wedge\; x = pred[b] \;\rightarrow pred[b] := a & \{\text{rule C1}\} \\
\;[\!]\; \neg e \;\wedge\; x \neq pred[b] \;\rightarrow & \{\text{rule C2}\} \\
\qquad \textbf{if } a = pred[suc[a]] \;\rightarrow suc[a] := b & \\
\qquad \;[\!]\; a \neq pred[suc[a]] \;\rightarrow \text{“not SP”} & \\
\qquad \textbf{fi} & \\
\;[\!]\; e \;\wedge\; x \neq pred[b] \;\rightarrow \text{skip} & \{\text{rule C3 or C4}\} \\
\;[\!]\; e \;\wedge\; x = pred[b] \;\rightarrow \text{“not SP”} & \{\text{rule C5}\} \\
\textbf{fi}; & \\
\textbf{if } suc[a] = x \;\rightarrow suc[a] := b & \\
\;[\!]\; suc[a] \neq x \;\rightarrow \text{skip} & \\
\textbf{fi} &
\end{array}$$

In case the representation cannot be maintained or a C5 redex is found, we conclude that the graph is not SP.

Selection of redex

The final part is the implementation of the reduction strategy. Since it is easy to partition the redexes into red and blue redexes, and since redexes do not change color due to the reduction strategy, we will focus on the problem of finding innermost blue redexes. To this end we will assign a number $f[x] \in [0..n)$ to each vertex such that redex x is innermost when $f[suc[x]] - f[pred[x]]$ is minimal among all redexes. Here, n is the number of vertices of the input graph. There are several alternatives for this numbering.

A simple solution is to use a *topological* numbering (recall that the input graph is assumed to be acyclic). Function f is a topological numbering of the vertices of a directed graph, if f is an injection for which $f[x] < f[y]$ for any arc (x, y) in the graph. This is advantageous when the numbering of the vertices of the input graph is already topological, in which case f is the identity function, and hence superfluous. Otherwise, when the numbering has to be computed anyway, it is better to compute a numbering f satisfying $f[s] = 0$ and $f[x] < f[y]$, for any arc (x, y), such that the maximum value of f is minimized (f is not required to be an injection). This can be done easily in linear time for acyclic graphs with a single source s.

During the reduction process function f is considered as a constant, but function d defined by $d(x) = f[suc[x]] - f[pred[x]]$ is not constant. The way function d changes in a reduction step is however quite restricted. Firstly, $d(x)$ may change for an existing redex x, but in that case $d(x)$ increases. And, secondly, for any new redex y, say, created in a reduction step applied to a redex x, we have that $d(y) \geq d(x)$.

Therefore, using that $d(x)$ is in $[2..n)$ for the above proposals for f, we may use an array b of about n "buckets", where each bucket is a singly-linked list (used as a stack), to keep track of the innermost blue redexes. The domain of b is $[2..n)$, and redex x is added to bucket $d(x)$. For each reduction step, new blue redexes are added (if any), and blue redexes for which the d-value changes are added to the new bucket, but not deleted from the old one. Note that in each reduction step at most one d-value changes value, since this only occurs when a blue redex has a red redex as a predecessor that is being removed.

To search for the next innermost blue redex, the array of buckets is searched from left to right, starting from the position last visited. In this way, the amortized time for each search is constant. For each redex found it is checked whether the number of the bucket is equal to the current d-value of the redex, and if this is not the case the redex is discarded and the search proceeds. This technique is often called "lazy deletion".

This completes the description of the recognition algorithm. We note that the space utilization is limited. Apart from the arrays *pred* and *suc*, we need an array *deg* to record the number of predecessors and successors of each vertex (as in Section 3), an array for the red redexes (cf. array X in Section 3), and an array of buckets b. Possibly, an array for the numbering f is also required.

A simple observation is that the total number of red and blue redexes is at most n, and that the total number of not yet deleted blue redexes is also at most n. This further limits the combined space used for red and blue redexes.

8. Succinct representations in general

In Section 4, we have shown how SP graphs with n vertices can be represented using two arrays of length n only. A natural question is now to ask when c arrays of length n are sufficient to represent a given graph (possibly with multiple arcs and self-loops), for some constant c. The appropriate generalization of the relation in Lemma 1 is as follows:

$$(x, y) \in E \;\equiv\; (\exists\, i : 1 \leq i \leq c : suc_i[x] = y),$$

where suc_i, $1 \leq i \leq c$, denotes an array of length n.

Obviously, a necessary condition for the existence of such a representation is that the total number of arcs is at most c times the total number of vertices. And, clearly, this should hold for any vertex-induced subgraph as well. The next theorem shows that this condition is also sufficient.

Theorem 1 *In order that the arcs of a graph can be redirected such that each vertex has at most c outgoing arcs, it is necessary and sufficient that for each vertex-induced subgraph the number of arcs is at most c times the number of vertices.*

Proof It remains to show that the condition on the vertex-induced subgraphs is sufficient. We do so by showing how to redirect the arcs in case there is a vertex with a *surplus* (i.e., more than c) of outgoing arcs, such that the total surplus over all vertices decreases. After a finite number of such transformations, the total surplus will become zero, which means that each vertex has at most c outgoing arcs.

Let x be a vertex with a surplus. The idea is now to consider the subgraph H induced by the set of all vertices reachable from x. Since H satisfies the condition and x has more than c outgoing arcs, there exists a vertex y in H with less than c outgoing arcs. And, by the definition of H, there exists a directed path from x to y. The arcs on this path from x to y are now redirected by reversing them. This transformation decreases the surplus of x by one, does not create a surplus for y, and does not change any of the surpluses of the other vertices on the path. Thus, the total surplus is decreased by one. □

It is left to the reader to check that SP graphs indeed satisfy the condition of the theorem. As another example we consider the class of planar graphs (without multiple arcs and self-loops, as usual). It is a well-known fact that $m \leq 3n$ for an n-vertex planar graph with m arcs ($m \leq 3n - 6$ actually, for $n \geq 3$). Since any subgraph of a planar graph is planar as well, the condition of the theorem is obviously satisfied. This guarantees the existence of a succinct representation of planar graphs (using 3 arrays of length n) for which the presence of an arc can be tested in constant time.

An interesting problem is how the time complexity of constructing such a representation behaves. Considering c as a constant, we first note that the proof of Theorem 1 implies an $O(n^2)$ algorithm for this problem: the initial surplus is $O(n)$ and each transformation takes $O(n)$ time. Inspired by [12, p.12], we are able to improve this to $O(n\sqrt{n})$ by reducing the problem to the computation of a maximum matching for a certain bipartite graph with $O(n)$ edges. (A maximum matching can be computed in $O((m+n)\sqrt{n})$ time for bipartite graphs of n vertices and m edges, see [6].)

Given a graph G, the bipartite (undirected) graph $G' = (V', E')$ is constructed as follows. The vertex set V' consists of c copies of each vertex of G on the one hand, and one copy of each arc of G on the other hand. Further, for each vertex x of G, the edge set E' contains an edge between each copy of x and each copy of an arc of G of which x is an endpoint. This defines G' as a graph with $cn + m$ vertices and at most $2cm$ edges, if G has n vertices and m arcs. Since $m \leq cn$ if G satisfies the condition of Theorem 1, it follows that $(m+n)\sqrt{n}$ is $O(n\sqrt{n})$.

Given a maximum matching for G', the arcs of G are redirected as follows. If the copy of arc (x, y) is connected to a copy of y in the matching, then the arc (x, y) is replaced by (y, x) in G, that is, the arc is redirected from y to x. Otherwise, the copy of the arc is connected to a copy of x, and the graph remains unchanged. As a result, each vertex will end up with at most c outgoing arcs in G.

However, it is not clear that the special structure of graph G' cannot be exploited to compute a maximum matching in less time. In other words, an $O(n)$ solution is not excluded, certainly given the fact that a linear time algorithm for finding—not necessarily optimal—tree-decompositions of a graph exists [2]. We leave this as an open problem.

9. Conclusion

We have shown how an efficient and relatively simple algorithm for the recognition of series parallel graphs can be obtained using the technique of breadth-first traversal. Only at a very late stage in the implementation of the algorithm, pointers are used. We have shown that the space utilization is limited, and also that it is advantageous that the input graph is topologically sorted. If so desired, the algorithm can be extended to compute the so-called decomposition tree (in case the graph is indeed series parallel), see e.g. [10].

Theorem 1 of the previous section indicates a way to generalize our approach. We are quite pleased with this theorem, since its statement as well as its proof do not refer to any boundary cases. A possible application would be a recognition algorithm for planar graphs. The suggested approach is to introduce a characterization like Definition 2, and to find a way to compute a succinct representation using three arrays (which exists on account of Theorem 1), and then to find a way to reduce the graph while maintaining the succinct representation. We leave this as an open problem.

REFERENCES

1. Aho A.V., Hopcroft J.E., Ullman J.D.: *The Design and Analysis of Computer Algorithms*, Addison-Wesley (1974).
2. Bodlaender H.L.: *A linear time algorithm for finding tree-decompositions of small treewidth*, Proceedings of the 25th Annual ACM Symposium on Theory of Computing (1993) 226–234.
3. Duffin R.J.: *Topology of series-parallel networks*, J. Math. Anal. Appl. 10 (1965) 303–318.
4. Eppstein D.: *Parallel recognition of series-parallel graphs*, Inform. and Comput. 98 (1992) 41–55.
5. He X., Yesha Y.: *Parallel recognition and decomposition of two-terminal series parallel graphs*, Inform. and Comput. 75 (1987) 15–38.
6. Hopcroft J.E., Karp R.M.: *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput. 2 (1973) 225–231.
7. Mehlhorn K.: *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag (1984).
8. Schoenmakers B.: *Data Structures and Amortized Complexity in a Functional Setting*, Ph.D. thesis, Eindhoven University of Technology, The Netherlands (1992).
9. Valdes J.: *Parsing Flowcharts and Series-Parallel Graphs*, Technical report STAN-CS-78-682, Stanford University, California (1978).
10. Valdes J., Tarjan R.E., Lawler E.L.: *The recognition of series parallel digraphs.*, SIAM J. Comput. 11 (1982) 298–313.
11. Wagner D.K.: *Forbidden subgraphs and graph decomposition*, Networks 17 (1987) 105–110.
12. White N., Whiteley W.: *The algebraic geometry of motions of bar-and-body frameworks*, SIAM J. Alg. Disc. Meth. 8 (1987) 1–32.