A depth-first algorithm to reduce graphs in linear time

Miklós Bartha
Department of Computer Science
Memorial University of Newfoundland
St. John's, NL, Canada, A1B 3X5
Email: bartha@mun.ca

Miklós Krész
Faculty of Education
University of Szeged
6721 Szeged, Hungary
Email: kresz@jgypk.u-szeged.hu

***Abstract*—A redex in a graph G is a triple $r = (u, c, v)$ of distinct vertices that determine a 2-star. Shrinking r means deleting the center c and merging u with v into one vertex. Reduction of G entails shrinking all of its redexes in a recursive way, and, at the same time, deleting all loops that are created during this process. It is shown that reduction can be implemented in $\mathcal{O}(m)$ time, where m is the number of edges in G.**

***Keywords*-graphs and matchings; depth-first trees; shrinking edges in graphs;**

I. Introduction

The problem dealt with in this paper is of an elementary nature: given a graph G, shrink all 2-stars (redexes) in G in a recursive way to obtain a reduced graph $r(G)$ free from subdividing vertices. During this process shrinking may introduce loops in the graph. Such loops are eliminated on the fly, giving rise to potential new redexes on which shrinking continues.

As an example, consider the graph G_0 of Fig. 1 containing four redexes centered at vertices 4, 7, 11, and 14. Shrinking the ones at 11 and 14 successively gives rise to a new redex a in the resulting graph G_2. In graphs G_0 and G_1 of Fig. 1, small arrows indicate which two vertices are collapsed in one step of the shrinking procedure. Shrinking at a and 7 in G_2 yields G_3, showing another new redex b. Finally, shrinking at 4 and b in G_3 results in a single edge, which is $r(G_0)$.

It is easy to come up with an $\mathcal{O}(n^2)$ algorithm for the above problem, where n denotes the number of vertices in G. One would simply consider the adjacency matrix of G, and keep track of the current set of redexes in a list. The implementation of shrinking one redex in $\mathcal{O}(n)$ time is then straightforward. See [2], [9] for more details. Finding a linear-time algorithm for graph reduction is much harder. The key idea is the following. First isolate groups of redexes that are connected at one end. Such groups are called groves. In our example, the redex groves are $\{11, 14\}$, $\{7\}$, and $\{4\}$. Then find out, without performing any actual shrinking, if a particular grove reduces to a new redex, like the grove $\{11, 14\}$ in our example. Such a grove is called an implied redex. Locate and reduce implied redexes recursively, until no more can be found. Finally, shrink the remaining groves in an arbitrary order to obtain $r(G)$.

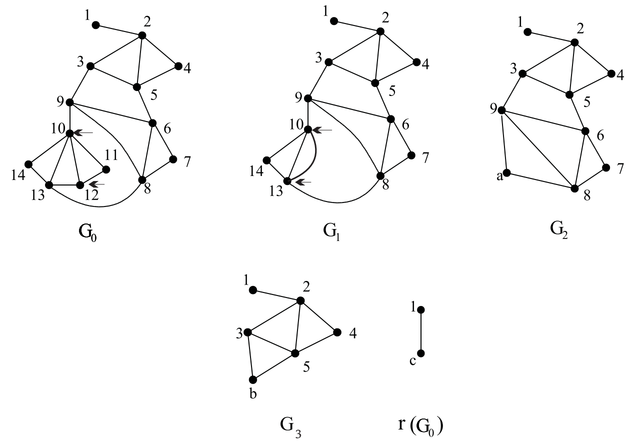

Figure 1. A reduction example

Reduction is matching invariant, so it can be used as a speed-up in matching algorithms. Due to its linear complexity, the gain can be quite substantial for sparse graphs. The situation looks even better in some experimental greedy matching algorithms, cf. [12], which have a linear time complexity to begin with. Moreover, reduction does not change the number of maximum/perfect matchings in graphs, so that it is helpful in the process of counting the number of such matchings. In particular, if a connected graph reduces to a single edge, like our example graph G_0, then it has a unique perfect matching. Even though the converse of this statement is not true, reduction is still a helpful technique in finding out if a graph has a unique perfect matching. The currently fastest unique perfect matching algorithm [7] runs in $\mathcal{O}(m \log^4 n)$ time on a graph with n vertices and m edges.

Reduction also plays an important role in the study of certain matching-based molecular switching devices called soliton automata [1], [2], [5]. As it was shown in [2], a soliton automaton defined by an elementary graph G is deterministic iff G reduces to a graph not containing even-length cycles. Yet another application of reduction along this line concerns finding a flexible maximum matching in graphs [8], [3]. With the help of the present algorithm it becomes possible to effectively find such a matching in linear time (whenever it exists), provided that an arbitrary maximum matching has previously been constructed for the graph.

Relating to other fundamental graph theory concepts, reduction simplifies the problem of finding a minimum size 2-(edge) connected spanning subgraph of a 2-connected graph [4], and that of finding a Hamiltonian cycle. Both problems are known to be NP-complete. The argument is as follows. The property of being 2-connected is preserved by reduction, and the desired 2-connected spanning subgraph of G can easily be retrieved from that of $r(G)$. If both G and $r(G)$ are also 2-vertex-connected, then this process is trivial, since each redex must be part of any spanning subgraph. For a graph that is not 2-vertex-connected, its minimum size 2-edge connected spanning subgraph is assembled from that of its 2-vertex-connected components (blocks). Dynamically, detaching such a component from G during reduction is analogous to eliminating a loop, so that it might create a new redex in the remainder graph and in the detached component as well. Thus, the algorithm becomes recursive. This type of recursion can be handled by a suitable modification of our Algorithm 2 described in Section 5.

As to Hamiltonian cycles it is obvious that, in order for G to have a Hamiltonian cycle, no redex grove can have a branch in it. That is, each maximum independent redex grove R of G must follow a path p_R, or G itself must be covered by an even-length cycle composed of individual redexes. In the latter case G has a trivial Hamiltonian cycle. Otherwise every Hamiltonian cycle of G must cover the whole path p_R for each grove R. Thus, G can be replaced by the graph G_p in which the internal points of p are deleted and the two endpoints are connected by a new redex, that is, a single edge with a subdividing vertex on it.

Reduction might turn out to be useful in other graph theory applications as well. We believe, however, that the algorithm presented in this paper is interesting by itself. The shrinking technique applied in our algorithm bears a close resemblance to shrinking e.g. in the Edmonds matching algorithm [6], [11], and, in general, it can be used to shrink graphs along a designated set of edges, which are specified simply by putting a subdividing vertex on them.

II. A GREEDY ALGORITHM TO SHRINK REDEX GROVES

Our notation and terminology on graphs will follow [11]. Let us fix a connected undirected graph $G = (V(G), E(G))$ for the rest of the paper. Loops and multiple edges are allowed to occur in G. A vertex $v \in V(G)$ is called *subdividing* if its degree is 2, *external* if the degree of v is 1, and *internal* if it is not external. For a subset $X \subseteq V(G)$ of vertices, $G[X]$ (or just $[X]$ if G is understood) will denote the subgraph of G induced by X. A *redex* in G is a triple $r = (u, c, v)$ of distinct vertices such that c is subdividing and adjacent to both u and v. The vertex c is called the *center* of the redex, while u and v are the two *focal vertices* (or just focuses) of r. The two edges incident with c are called focal, too. *Shrinking* a redex means deleting its center and merging its two focal vertices into one *sink* vertex. Graph G is called *reduced* if it does not contain redexes or loops. A reduced graph $r(G)$ is obtained from G by shrinking all redexes and removing all loops in it in a recursive fashion. As it was proved in [3], $r(G)$ is unique up to graph isomorphism.

Let R be a set of redexes in G. A vertex set $S \subseteq V(G)$ is *covered* by R if $S \subseteq \cup R$. The set R is a *redex cover* in G if it covers every redex in G. Two redexes are *connected* if they have at least one focal vertex in common. This relationship defines a graph $Rd(G)$ on the set of G's redexes as vertices, whereby an edge exists between r_1 and r_2 iff r_1 is connected to r_2. A *redex grove* is a non-empty set R of redexes such that $Rd(G)[R]$ is connected. Two redex groves are said to be *equivalent* if they cover the same set of vertices in G. A redex grove R is *independent* if it has no equivalent proper sub-grove, *maximal independent* if it is independent and cannot further be extended without losing this property, and *maximum* if there exists no other grove R' such that $\cup R \subset \cup R'$. Clearly, every redex grove has an equivalent independent sub-grove, which may not be unique.

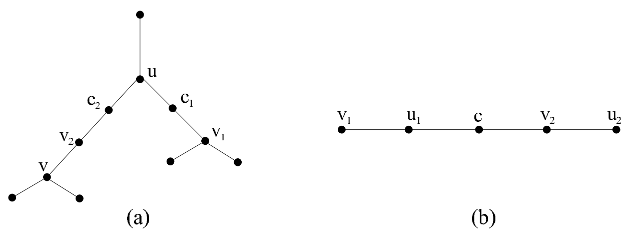

Figure 2. Redex groves

Examples. A triangle graph is covered by a redex grove consisting of three interconnected redexes, each of which is independent by itself. The graph in Fig. 2a has two maximal independent redex groves: $R_1 = \{(u, c_1, v_1), (u, c_2, v_2)\}$ and $R_2 = \{(v, v_2, c_2)\}$, both of which are maximum as well. The set $R_1 \cup R_2$ is not a redex grove, as it does not define a connected subgraph in $Rd(G)$. In the graph of Fig. 2b, $R = \{(u_1, c, v_2)\}$ is maximal but not maximum, for the grove $S = \{(v_1, u_1, c), (c, v_2, u_2)\}$ covers more vertices than R.

In order to describe our reduction algorithm we need a few technical statements on redex groves.

Proposition 2.1. *If R is an independent redex grove, then no vertex $v \in \cup R$ is the center of one redex and a focus of another at the same time.*

Proof. Let us assume, on the contrary, that there exist redexes $r_0 = (u, c, v)$ and $r_1 = (c, x, y)$ in R. Clearly, $x = u$ or $x = v$. Say $x = v$. Then u cannot be the center of some other redex in R, because this would make r_0 dispensable. Since R is a grove, there exists a chain of connected redexes $r_1, \dots, r_n$ ($n \geq 1$) such that u is a focal vertex of r_n. (Note that $n = 1$ identifies a triangle situation.) Thus, $R - r_0$ covers the same vertices as R, which contradicts R being independent. □

Corollary 2.2. *Let R be a maximal independent redex grove and $v \in \cup R$ be a focal vertex according to R. If $(v, c_1, x_1), (x_1, c_2, x_2), \dots, (x_{n-1}, c_n, x_n)$ is a sequence of connected redexes in G (not necessarily contained in R), then $x_n \in \cup R$.*

Proof. Let S denote the set of redexes $\{(x_{i-1}, c_i, x_i) \mid 1 \leq i < n\}$ with $x_0 = v$. Then $R \cup S$ is a redex grove, and, since R is maximal, $R \cup S$ covers the exact same vertices as R. Thus, $x_n \in R$. It may happen, though, that x_n is designated as center in R. □

The following lemma characterizes the relationship between two overlapping maximal independent redex groves. With a slight ambiguity, $\cup R$ will also denote the union of R's redexes as subgraphs of G. If ambiguity does arise, this meaning will be re-enforced by writing *graph* $\cup R$.

Lemma 2.3. *Let R and S be two non-equivalent maximal independent redex groves in G. Then the graph $\cup R \cap \cup S$ consists of n vertex-disjoint paths running exclusively on subdividing vertices such that one of the following three conditions is met.*

(i) $n = 0$, so that the graphs $\cup R$ and $\cup S$ are themselves vertex-disjoint;

(ii) $n = 1$, $\cup R \subset \cup S$ ($\cup S \subset \cup R$), and the only overlapping path between $\cup R$ and $\cup S$ is $\cup R$ (respectively, $\cup S$) itself, consisting of an odd number of subdividing vertices;

(iii) $n \geq 1$, $\cup R$ and $\cup S$ are not comparable, and each overlapping path is running on an even number of subdividing vertices.

Proof. By Proposition 2.1, each vertex $v \in \cup R$ and $u \in \cup S$ is either focus or center in R and S, respectively. It cannot happen that $x \in \cup R \cap \cup S$ is focus according to both R and S, because Corollary 2.2 would then imply that R and S are equivalent. Thus, every vertex in $\cup R \cap \cup S$ is subdividing. Consequently, $\cup R \cap \cup S$ indeed consists of n disjoint paths, as stated. (Even-length cycles are excluded, for $\cup R \neq \cup S$). Assuming that $n \geq 1$, let p be any of these paths. Observe that p has at least two vertices. Consider the vertices $u(p)$ and $v(p)$ outside p that are adjacent to p from its two ends. See Fig. 3. Since R and S are maximal, the degree of these vertices is different from 2, and each one is focus according to either R or S. If they are focus according to the same grove, say S, then p coincides with $\cup R$. See Fig. 3a, and recall that the grove R is connected. Hence we have (ii) above. Notice that in this case R is not maximum.

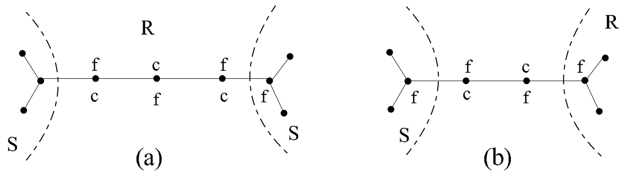

Figure 3. Overlapping maximal redex groves

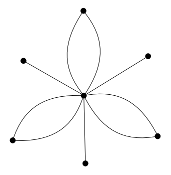

Figure 4. A flower graph

If $u(p)$ and $v(p)$ are focus according to different groves, then p consists of an even number of vertices. See Fig. 3b. Moreover, $u(p) \neq v(p)$, implying that $\cup R$ and $\cup S$ are not comparable. The proof is now complete. □

An immediate consequence of Lemma 2.3 is that a maximal independent redex grove R is not maximum only if $\cup R$ is a path running on an odd number of subdividing vertices.

Definition 2.4. Graph G is *internally redex-covered* if there exists a redex grove R such that, with the exception of some external vertices, R covers all of $V(G)$.

Definition 2.5. A *flower graph* is a loop-free graph G having a central vertex v such that each edge $e \in E(G)$ is incident with v, and the multiplicity of e is at most 2. Edges with multiplicity 1 and 2 are called *sepals* and *petals*, respectively, and a flower with n sepals ($n \geq 0$) and no petals is called an *n-star*. See Fig. 4.

Proposition 2.6. *If G is internally covered by an independent redex grove R, then $r(G)$ is a flower graph, which is a star iff the graph $\cup R$ is a tree.*

Proof. Clearly, R is maximal. Pick a redex $r \in R$ arbitrarily. Starting from r, shrink the redexes in R one-by-one in such a way that redexes corresponding to two consecutive shrinkings be always connected. As long as $\cup R$ follows a tree structure, all redexes collapse into one sink vertex. When, however, a vertex comes up repeatedly as focus in this sequence in a circular fashion, its underlying redex s will no longer exist, because its two focuses have previously been joined. In this case s contributes a petal to the reduced graph, and we continue as if s did not exist. The statement of the proposition is now obvious. □

Corollary 2.7. *An internally redex-covered graph can be reduced in linear time.*

Proof. In the procedure outlined above, fix u to be either focal vertex of the initial redex r, and perform shrinking in such a way that u keeps serving as the sink vertex. The shrinking of one redex then costs as much as the number of edges incident with the "other" focus (i.e., other than u). Thus, the overall cost of reduction is indeed linear. □

An important observation must be made regarding the reduction procedure presented in the proof of Proposition 2.6 above. We say that G *directly* reduces to the graph $d(G)$ obtained from G after having considered all redexes in R for shrinking. Clearly, $d(G)$ is still a flower graph, which has as many sepals as the number of external vertices left uncovered by R. Moreover, $d(G)$ coincides with $r(G)$, unless $d(G)$ is a 2-star, i.e., a redex. According to Lemma 2.3, this can happen in two different ways:

1. The grove R is maximum.
2. The grove R is not maximum, $\cup R$ is a path p running on an odd number of subdividing vertices, and G is obtained from p by attaching the vertices $u(p)$ and $v(p)$ described in the proof of Lemma 2.3 to the two endpoints of p.

In the latter case, the whole graph G can be covered by a different maximum redex grove R', in which both $u(p)$ and $v(p)$ appear as focal vertices. Thus, in order to define $d(G)$ in an unambiguous way, one must assume that R is maximum, not just maximal. Ensuring that R be maximum is straightforward: include a redex from the "side", that is, one having a non-subdividing vertex. If this is not possible, then G is a cycle, so that the problem does not even arise.

Let R be an arbitrary redex grove in (a general) graph G, and construct the graph $I(R)$ by augmenting $[\cup R]$ in the following way. For each vertex $v \in \cup R$ and edge e in $G - R$ incident with v, introduce a new external vertex v_e and connect it to v. It is obvious that $I(R)$ is internally covered by R. Moreover, if R is maximum in G, then it is such in $I(R)$.

Definition 2.8. An *implied redex* in G is a maximum independent redex grove R such that the graph $d(I(R))$ is a redex.

Theorem 2.9. *If G does not contain implied redexes, then $r(G)$ can be constructed in linear time.*

Proof. Without loss of generality we can assume that G is not a cycle. Let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be a system of pairwise non-equivalent maximum independent redex groves such that $\cup\mathcal{R}$ is a redex cover in G. Since R_1 is not an implied redex, its direct reduction inside G does not introduce a new redex into the resulting graph G_1. Moreover, by Lemma 2.3, the groves R_j, $j > 1$ will either remain intact, disappear, or be truncated to some maximum independent groves R'_j in G_1 such that $d(I(R'_j)) = d(I(R_j))$. Thus, G_1 does not contain implied redexes either, or at least there is none participating in the covering system $\mathcal{R}_1$ inherited from $\mathcal{R}$. (See also Corollary 2.10 below.) In this way, the reduction of G entails only the direct reduction of the groves $R_1, \ldots, R_m$, or whatever remains of them during this process.

As to the desired algorithm, choose an initial redex r in such a way that it includes a non-subdividing vertex. (Recall that G is not a cycle.) Locate a maximum independent redex grove R_1 containing r, and perform its direct reduction. By Corollary 2.7, this takes linear time regarding the size of $[\cup R_1]$. Repeat the process until no more redexes are found. It should be evident that the time complexity of this algorithm is indeed $\mathcal{O}(|E(G)|)$. □

Corollary 2.10. *Let $\mathcal{R} = \{R_1, \ldots, R_m\}$ be a system of pairwise non-equivalent maximum independent redex groves such that $\cup\mathcal{R}$ is a redex cover in G. If none of R_i, $1 \le i \le m$ is an implied redex, then G does not have implied redexes.*

Proof. By way of contradiction, assume that G does have an implied redex R. Join R to $\mathcal{R}$, and notice that it is not equivalent to any of the groves R_i. Repeat the direct reduction process eliminating the groves R_i. According to Lemma 2.3, the reduction of any R_i overlapping with R can only shorten R without being able to completely knock it out. Moreover, shortening R will still keep it as an impied redex. Thus, after the reduction, $r(G)$ will still contain an implied redex, which is impossible. □

The condition that G does not contain implied redexes is crucial in Theorem 2.9. If G does contain such redex groves, then, following the procedure outlined above, we may locate "good" groves (i.e., ones that are not implied redexes) first, and implied redexes later. In this way we have merely reproduced our original problem at the "macro" level, where the vertices are essentially the centers of the flower graphs $d(I(R_i))$, together with those old vertices of G not covered by $\cup\mathcal{R}$. Of course, the complexity of reduction accumulates in a recursive way, resulting in a non-linear-time algorithm. This argument also explains why one cannot simply go ahead and shrink redexes in G in an arbitrary order, still hoping to finish in linear time. Indeed, even though the shrinking of one redex r costs only as much as the number of edges incident with one focus of r, there is no guarantee that each edge will contribute only a constant number of times this way to the overall count.

Our task is therefore to find implied redexes (even recursively emerging ones) in linear time, replace each such grove by a single redex, and then come back to the greedy algorithm described in Theorem 2.9 to finish the job.

III. Flipping brackets in depth-first trees

A *depth-first tree* (DFT, for short) is a rooted spanning tree T_G such that every edge $e \in E(G)$ falls into one of the following two groups.

(i) *tree edges*, which belong to the tree T_G;

(ii) *back edges*, which connect vertices with their ancestors in T_G.

It is well-known that for every vertex $v \in V(G)$, G has a DFT rooted at v, which can be constructed in linear time. We shall assume that this construction deletes all loops that G might originally have. Notice that, among a number of parallel edges in G, at most one can be a tree edge. With a slight ambiguity we shall, in principle, identify G with the tree T_G and say that G is a DFT with *root* $(G) = v$. A DFT G is called *open* if its root is an external vertex. In this case the vertex adjacent to *root* (G) is called the *ground* of the tree, denoted *gr* (G). In the sequel we shall assume that our DFT G is open. From the reduction point of view generality is not jeopardized, for if G were not open, then attaching an extra redex to *root* (G) will make it such. The resulting graph also reduces to $r(G)$, essentially in the same amount of time.

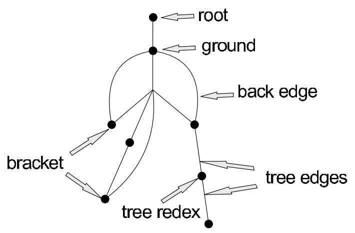

Figure 5. Depth-first tree concepts

Redexes come in two distinctive forms in a DFT G. A *tree redex* is one consisting of two tree edges, and a *transversal* redex (or *bracket*, for short) is a redex having one tree and one back edge. Notice that, since G is open, every tree redex r is "vertical" in G, meaning that the center of r has only one immediate descendant (son), namely a focus of r. Thus, if $r = (u, c, v)$ is a redex (tree or not), then $u < v$, where $<$ denotes the ancestor relationship in T_G. We say that r *spans* from u to v, or r is a redex $u \to v$, with u being the *bottom* and v being the *top* of r.

When identifying redexes in a DFT G, we shall follow a *bottom-up* alignment. This means that, during a bottom-up sweep, a subdividing vertex v always becomes the center of a redex, provided that v is not the peak of a petal and its son (if any) has not previously been chosen as center. In particular, a subdividing leaf of G (i.e., that of T_G) always identifies a bracket (or petal), even when its father is subdividing, too. See Fig. 5 for an illustration of the DFT concepts introduced so far. It is easy to see that the rule of bottom-up alignment identifies a redex cover $R(G)$ in G in a unique way. Moreover, each connected component of $Rd(G)[R(G)]$ is a maximum independent redex grove. (Recall that $Rd(G)$ is the redex graph of G.)

Let r and s be redexes in $R(G)$ spanning from a common bottom vertex u to v and x, respectively, such that $v < x$ (i.e., x is above v). *Flipping* s through r means replacing the (necessarily transversal) redex s by a new bracket $s' : v \to x$. Even though the resulting DFT G' is not isomorphic to G, shrinking both r and s in G is isomorphic to shrinking both r and s' in G'. Thus, flipping is reduction invariant. Furthermore, $s' \in R(G')$, and the maximum (independent) redex grove in $R(G')$ containing s' and r is obtained by substituting s' for s in the corresponding grove in $R(G)$ containing s and r. The union of this grove (as a subgraph of G') is a tree iff the union of the corresponding grove is such in G.

A set R of redexes in a DFT G is said to be *regular* if no vertex of G occurs more than once as the bottom of some redex in R, except when the redexes sharing their bottom share their top as well. Let *flip* (G) denote the graph obtained from G by successive flips, which eliminate all ties between bottom vertices of different redexes in $R(G)$, if at all possible. Clearly, *flip* (G) is uniquely determined and $R($*flip* $(G))$ is regular.

Proposition 3.1. *The DFT G is internally redex-covered by $R(G)$ with $d(G)$ being a star graph iff every internal vertex of flip (G), with the possible exception of the ground vertex, is the bottom or center of exactly one redex in $R($flip $(G))$.*

Proof. By the nature of flipping it is sufficient to prove that, whenever $R(G)$ is regular, G being internally redex-covered by $R(G)$ in such a way that $d(G)$ is a star is equivalent to having each non-center internal vertex of G different from *gr* (G) appear as the bottom of a unique redex in $R(G)$.

If $R(G)$ is regular, then the graph $\cup R(G)$ is cycle-free as a subgraph of G iff no two redexes in $R(G)$ have the same span. Given this fact, the condition that each non-center internal vertex, excluding *gr* (G), appears as the bottom of some redex in $R(G)$ is equivalent to saying that $\cup R(G)$ has one more vertices than edges, and covers all internal vertices of G (including *gr* (G)). In turn, this is equivalent to $\cup R(G)$ being a tree, so that our statement follows from Proposition 2.6. □

We use the following bottom-up algorithm to check if G is internally redex-covered with $d(G)$ being a star graph.

Algorithm 1. Execute Check_vertex(*root* (G)), and see if each internal vertex of G different from *gr* (G) is marked either center or bottom, but no vertex is marked doomed.

In Algorithm 1, the pseudo-code for the two recursive procedures Check_vertex(v) and Flip(u, v) is given below. In Flip(u, v), u.span is a pointer associated with the bottom of a redex r, which is intended to point – beyond a cascade of redexes to be flipped through – to the top v of r. As part of the flip, the span of each redex in the cascade is extended (in principle) up to v in order to speed up flips that are yet to come.

```
Check_vertex($v$):
  if leaf($v$) then
```

```
    { if deg($v$)=2 and not petal($v$)
      then mark_center($v$) }
  else { for each son $u$ of $v$:
        Check_vertex($u$);
        if deg($v$)=2 and not marked_center(son($v$))
        then { mark_center($v$);
              Flip(son($v$),father($v$)) }
        else for each back edge $(x, v)$ from a leaf $x$:
            if marked_center($x$) then
            Flip(father($x$),$v$) }

Flip($u, v$):
  if $u = v$ then mark_doomed($u$)
  else { if not marked_bottom($u$)
        then mark_bottom($u$)
        else Flip($u$.span,$v$);
        $u$.span:=$v$ }
```

Theorem 3.2. *Algorithm 1 correctly decides if G is an internally redex-covered graph for which $d(G)$ is a k-star. If so, the number of vertices left unmarked by the algorithm is $k + 1$. The algorithm runs in linear time.*

Proof. The first statement, with regard to correctness and the number k, is obvious by Proposition 3.1. As to linearity, observe that the only concerning issue is the number of times the pointers u.span must be assigned on the last line of Flip(u, v). Not counting the very first assignment of these pointers, let $T(k, n)$ denote the total number of pointer assignments needed to process a DFT G with $|R(G)| = n$, such that the first k redexes follow a regular arrangement without duplication. Clearly, $T(n, n) = 0$. Let r be the $(k + 1)$st redex. If r must be flipped through a cascade of l redexes, then this maneuver costs l pointer assignments. On the other hand, r will kill the whole cascade, so that the situation after the adjustment is the same as if we had only $n - l$ redexes to begin with, out of which the first $k - l + 1$ follow a regular arrangement with no duplication. Thus,

$$T(k, n) \leq \max(\{l{+}T(k{-}l{+}1, n{-}l) \mid 1{\leq}l \leq k\}, T(k{+}1, n))$$

if $k < n$. From this formula, $T(k, n) \leq n$ is provable by a trivial induction on $n - k$. □

IV. Identifying and reducing implied redexes in depth-first trees

Topologically, an implied redex R manifests itself in three different forms in a DFT G.

1. The graph $d(I(R))$ is a tree redex. In this case $[\cup R]$ is surrounded by a subtree S from the bottom and a supertree H from the top, so that the two edges e_b and e_t of the 2-star $d(I(R))$ provide a handle to S and $[\cup R]$, respectively, as tree edges in G. See Fig. 6a. Clearly, the endpoints of e_b and e_t incident with $[\cup R]$ must be focus in R, while the lower endpoint of e_b, as well as the higher endpoint of e_t, must not be subdividing.

2. The graph $d(I(R))$ is a bracket. In this case $[\cup R]$ itself is a subtree, connected to a supertree H from the top by the tree edge e_t. The edge e_b is a back edge, which originates from a focus in R. See Fig. 6b.

3. The graph $d(I(R))$ is an "illegal" horizontal redex. In this case $[\cup R]$, as a supertree, includes *root*(G), and the edges e_1, e_2 of the 2-star $d(I(R))$ are both tree edges. Notice that *gr*(G) is now center in R. See Fig. 6c.

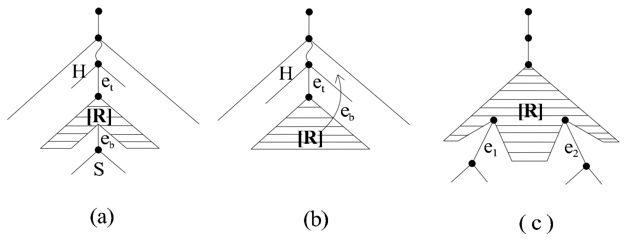

Figure 6. Implied redexes

Once an implied redex R is identified during a bottom-up sweep of G, it is reduced as follows.

Case 1: R (i.e., $d(I(R))$) is a tree redex. Each vertex of R, except the lower endpoint z of e_t, is marked "done", and practically forgotten. The lower endpoint u of e_b is marked bottom, and u.span is set to the upper endpoint of e_t. Vertex z is marked center, and the bottom-up sweep continues.

Case 2: R is a bracket. Again, each vertex of R, except the lower endpoint z of e_t, is forgotten, and the back edge e_b is moved in such a way that its lower endpoint coincide with z. If the upper endpoint v of e_t is the same as that of e_b, then v is marked doomed (petal), otherwise it is marked bottom. Vertex z is classified as leaf, marked center (if v is not doomed), and the bottom-up sweep continues.

Case 3: R is illegal. Now we do not attempt to replace $d(I(R))$ with a single redex, since this would trigger a top-down process of detecting new illegal redexes. Rather, we "exclude" R by deleting its root. Notice that this cannot be done a priori, for *gr*(G) may not originally have been a subdividing vertex, just popped up as an implied tree redex. Let G' denote the shortened graph. Since the bottom-up sweep has ended, G' will have no more implied redexes. Construct $r(G')$ by the greedy algorithm of Theorem 2.9, using the redex cover $R(G')$ broken down into a system $R_1, \ldots, R_m$ of maximum independent redex groves following the bottom-up alignment. (This system is actually constructed in a natural way during the bottom-up sweep detecting the implied redexes.) The vertex $w =$ *root*(G') is not covered by $R(G')$, therefore it is still present in $r(G')$ as an external vertex. Delete w, and reduce the resulting graph

– which may contain at most one redex – in linear time. Clearly, the end-result is the graph $r(G)$.

V. Detecting implied redexes in depth-first trees

We are now left with the most difficult question of how to detect implied redexes in a systematic way during a single bottom-up sweep of G. With this question answered, recursion – as described in Section 4 – will roll up all implied redexes and make G free of them.

Let $u \leq v$ be two vertices in $V(G)$. The *interval* determined by the pair (u, v) is the set $I(u, v) = \{x \in V(G) | u \leq x \leq v\}$. Again, if no confusion arises, $I(u, v)$ will also stand for the line $[I(u, v)]$ as a subgraph of G (rather, T_G, mind the back edges). The set of *principal* intervals in G, relative to $R(G)$, is the set $P(G)$ of intervals spanned by all tree redexes and back edges of G. Thus, any given principal interval might be classified in three different ways: a tree redex interval, a bracket interval, or an ordinary back-edge interval.

For a vertex $v \in V(G)$, let $P(v)$ denote the set of principal intervals of G running entirely within the subtree determined by v. If u is a vertex in this subtree not covered by any interval in $P(v)$, then add $\{u\}$ to $P(v)$ as a *degenerate* principal interval. If S is a set of principal intervals, then $\cup S$ is called a *region*, and a maximal connected subset of $\cup S$ (as a subgraph of T_G) is called a *domain* of that region. Clearly, every region is the union of pairwise disjoint domains. For every vertex $v \in V(G)$, let D_v denote the domain of $\cup P(v)$ containing v, and denote the restriction of $R(G)$ to D_v by $R(G)/v$. According to the description of implied redexes in Section 4, the following statement is an immediate consequence of Proposition 3.1.

Proposition 5.1. *Let v be an arbitrary vertex in G, and assume that $R(G)$ is regular. In order for D_v to form an implied redex in G it is necessary that each vertex of $D_v - v$ be the center or bottom of a unique redex in $R(G)/v$.*

Our strategy to detect implied redexes during a bottom-up sweep of G is based on Proposition 5.1. For every non-center vertex v, we locate the domain D_v and calculate its size *size* (D_v). At the same time, we also count the number of hits *hit* (D_v) by center or bottom vertices of redexes inside D_v, together with the numbers *down* (D_v) and *up* (D_v) of edges leaving D_v downwards and upwards, respectively. During this process, we impose regularity by flipping and mark any duplications of redexes doomed as in Algorithm 1. The condition that identifies an implied redex is then:

$hit\,(D_v) = size\,(D_v) - 1$, $up\,(D_v) = 1$ and $down\,(D_v) = 1$ for a tree redex;

$hit\,(D_v) = size\,(D_v) - 1$, $up\,(D_v) = 2$ and $down\,(D_v) = 0$ for a bracket; and

$hit\,(D_v) = size\,(D_v) - 1$, $up\,(D_v) = 0$ and $down\,(D_v) = 2$ for an illegal redex.

Furthermore, the tree edge leaving D_v upwards must not lead to a subdividing vertex (center), and, in case of a tree redex, the tree edge leaving D_v downwards must not be the focal tree edge of a bracket whose top is above v. (Remember that an implied redex is a maximum redex grove.) Also, no vertex in D_v can be marked doomed.

To describe our algorithm we use a straightforward attribute grammar terminology [10]. The attributes $v.size$, $v.hit$, $v.down$, and $v.up$ – representing *size* (D_v), *hit* (D_v), *down* (D_v), and *up* (D_v) – are assigned to each non-center vertex v, and the tree G is decorated by the values of these attributes calculated at each such vertex. The method of calculation is the same for each attribute, therefore we only deal with $v.hit$ here, leaving the elaboration of the other three attributes to the reader.

Calculating $v.hit$, assuming that $R(G)$ is regular:

```
Step 1. Initialize $v.hit := 0$ and set
          $S := \cup(P(u) | u$ is a son of $v)$.
Step 2. For each tree redex interval $I(u, v)$:
           increment $v.hit$ by 2;
           add $I(u, v)$ to $S$.
Step 3. For each remaining non-degenerate principal interval
        $I(u, v) \in P(v)$:
   Let $D_{v_1}, \ldots, D_{v_n}$, $n \geq 0$ be those domains (including
   degenerate ones) of region $\cup S$ that intersect $I(u, v)$ but do
   not contain $v$.
        increment $v.hit$ by $\sum_{i=1}^{n} v_i.hit$;
        increment $v.hit$ by 2 if $I(u, v)$ is a bracket interval;
        add $I(u, v)$ to $S$.
```

The major technical difficulty in implementing this algorithm in linear time is to keep track of the domains of the region $\cup P(v)$. The following code will accomplish this goal.

```
Process_vertex($v$):
  { if leaf($v$) then
       { if deg($v$)=2 and not petal($v$)
         then mark_center($v$) }
    else { $v$.hit:=0;
           for each son $u$ of $v$:
           Process_vertex($u$);
           if deg($v$)=2 and not marked_center(son($v$))
           then { mark_center($v$);
                  Flip(son($v$),father($v$));
                  father($v$).hit:= father($v$).hit+2;
                  Interval(son($v$),father($v$)) }
           else for each back edge ($x$,$v$):
                { if marked_center($x$) then
                  { Flip(father($x$),$v$);
                    $v$.hit:=$v$.hit+2 };
                  Interval($x$, $v$) }
         };
    if implied_redex($v$) then reduce($v$)
```

```
        /* as described in Section 4 */ }

Interval($u, v$):
  do while not ($u = v$ or $u$.jump=$v$)
     if marked_center($u$) then $u$:=father($u$)
     else { $x$:=$u$;
              /* $x$ is a swap variable */
            if not marked_seen($u$)
              /* domain top is reached */
            then { mark_seen($u$);
                   $v$.hit:=$v$.hit+$u$.hit;
                   $u$:=father($u$) }
            else $u$:=$u$.jump;
              /* go find the top */
            $x$.jump:=$v$
              /* adjust old $u$.jump */ }
```

The procedure Flip(u, v) is adopted from Section 3.

Algorithm 2. Process G by executing the procedure Process_vertex(*root*(G)), resulting in a graph G' free from implied redexes. Forget the DFT structure, and apply the greedy algorithm of Theorem 2.9 on G' to obtain $r(G)$, taking into account the adjustment described under Case 3 of Section 4.

Theorem 5.2. *Algorithm 2 constructs $r(G)$ in linear time.*

Proof. The correctness of Algorithm 2 should be evident. The pointers x.jump in the procedure Interval(u, v) are used to locate the top of the domain in the current region containing x. At every point in time, whenever x is already marked seen, x.jump points to the top v of a principal interval $I(u, v)$ in the current range for which:

1. vertex x lies on $I(u, v) - v$;

2. there is no other interval $I(u', v')$ also containing x such that $v < v'$ and x is the closest common ancestor of u and u'.

Thus, the top of the domain in question can always be reached by a number of jumps through the jump pointers. The top is recognized when an unseen non-center vertex x is reached for which x.jump is not yet defined. It is at this point where the attributes must be considered for addition. The Boolean function implied_redex(v) will check the attributes at vertex v for the conditions that identify an implied redex, as described in the paragraph following Proposition 5.1. As part of the check, it will also see if D_v is a maximum redex grove, that is, the two edges leaving D_v do not lead to subdividing vertices. If an edge is leaving downwards, then its other endpoint will have been marked center if it is subdividing. For a tree edge leaving upwards to the father of v, this condition is straightforward to check.

The question of linearity again boils down to how many times the pointers x.jump must be assigned on the last line of Interval(u, v). The situation is analogous to the one handled in the proof of Theorem 3.2 in connection with the span pointers. Let $I_1 = I(u_1, v_1)$ and $I_2 = I(u_2, v_2)$ be principal intervals in $P(G)$. We say that I_2 *overlaps* I_1 if $v_1 < v_2$ and $u_2 \leq x$ for some vertex $x \in I_1 - v_1$. The lowest possible x is then called the *split* of I_2 from I_1.

In our argument we shall consider *groups* of principal intervals according to the equivalence relation $I(u, v) \sim I(u', v')$ iff $v = v'$. The *top* of an interval group is the common top vertex of its intervals. We say that two groups are *non-overlapping* if their tops are not comparable in T_G, or they are but no interval from the higher group overlaps any interval from the lower one. Not counting the very first assignment of the pointers x.jump, let $T(k, n)$ denote the total number of assignments to these pointers that is necessary to process a DFT G with n groups of principal intervals, such that the first k groups are pairwise non-overlapping. Observe that $T(n, n) = 0$. (Remember that x.jump always points to the top of the interval I_1 containing x for which there is no overlap between I_1 and a higher interval I_2 such that the split of I_2 from I_1 is x, and we ignore the very first assignments.) Consider an interval $I(u, v)$ from the $(k + 1)$st group, and assume that the processing of $I(u, v)$ costs $l \geq 1$ pointer assignments that count (as second assignments). Then $I(u, v)$ overlaps l intervals from distinct groups among the first k ones, so that the pointer assignments incur at the split of $I(u, v)$ from these intervals. Let $I(u_1, v_1), \ldots, I(u_l, v_l)$ be these intervals such that $v_1 < \ldots < v_l < v$. See Fig. 7a, where $k = l = 2$, and the split vertices are x and v_1.

From the point of view of continuation, the situation after processing $I(u, v)$ is equivalent to having only $n - l + 1$ interval groups to start with, out of which the first $k - l + 1$ are non-overlapping. See Fig. 7b. In essence, $I(u, v)$ kills the whole cascade of interval groups containing the overlapped intervals, except for the lowest one containing $I(u_1, v_1)$, whose range is extended from v_1 to v_l. By killing $I(u_i, v_i)$, $i > 1$ and its group, we mean lifting the whole subtree T_{x_i} below the split x_i of $I(u, v)$ from $I(u_i, v_i)$ — deprived from its branches starting out on $I(u, v)$ — to v_l. All back edges originating from T_{x_i}, too, are lifted along with T_{x_i}. Furthermore, every back edge currently pointing to v_i is switched to v_l instead. The same lifting procedure applies to the subtrees below v_j, $1 \leq j < l$ as well, since v_j.jump will also point to the new top v. In Fig. 7b, $I(u_2, v_2)$ is killed and $I(u_1, v_1)$ is extended up to v_2.

Finally, the subtree T_x below the split x of $I(u, v)$ from $I(u_1, v_1)$ – deprived from the branches starting out on $I(u_1, v_1)$ – is also lifted to v_l. See again Fig. 7. The reader can easily verify that, from each seen vertex below v, the number of jumps necessary to reach v is the same in Fig. 7b as it is in Fig. 7a after processing the interval $I(u, v)$. The only exceptions are the abandoned split vertices and the vertices $v_j, j < l$, but the role of these vertices has been taken over by v_l. Therefore, since the cost of processing a number of non-overlapping groups is zero, we can simply replace G by the modified graph for the rest of the algorithm

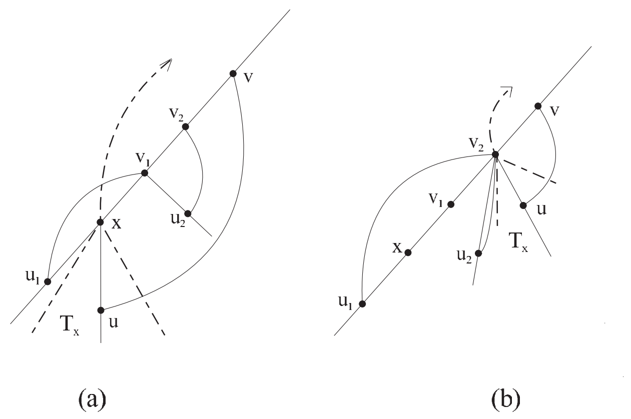

Figure 7. Resolving the overlap between intervals

dealing with the principal intervals that have not yet been processed. It is on this basis that the two situations are considered equivalent from the point of view of continuation. Due to the fact that the first k interval groups are pairwise non-overlapping, the DFT structure is not hurt by the lifting of the overlapped groups, and the depth of the system of groups has been reduced by $l-1$.

Repeat the above process for each interval in the $(k+1)$st group, and let G_k denote the DFT resulting from the modifications involved. Graph G_k will have $n-j$ interval groups for some $0 \leq j \leq k$ out of which at least the first $k-j+1$ are non-overlapping. (Observe that the old $(k+1)$st group will no longer overlap the previous ones.) Meanwhile, the total cost of processing the $(k+1)$st group in G by Algorithm 2 is not more than $j+n_{k+1}$, where n_{k+1} is the number of intervals in that group.

The conclusion of the argument above is that, for every $k<n$:

$$T(k,n) \leq \max(\{j+n_{k+1}+T(k-j+1,n-j) \mid 0 \leq j \leq k\}, \\ T(k+1,n)).$$

From this formula we have:

$$T(1,n) \leq j_1 + n_{k_1} + \ldots + j_t + n_{k_t},$$

where $0 \leq t < n$ and $\sum_{i=1}^{t} j_i < n$. Thus, the total number of pointer assignments is $\mathcal{O}(|E(G)|)$, as stated. The proof of Theorem 5.2 is now complete. $\square$

VI. Conclusion

We have presented a linear-time algorithm to shrink a graph G recursively along its 2-star subgraphs called redexes. The starting point was a greedy algorithm, which works in linear time only if G does not contain recursively incurring (implied) redexes. Implied redexes have been detected and reduced during a single bottom-up sweep of the depth-first tree of G, and the resulting graph was transferred to the greedy algorithm to construct the desired graph $r(G)$.

References

[1] M. Bartha, M. Krész, Structuring the elementary components of graphs having a perfect internal matching, *Theoretical Computer Science* **299** (2003), 179–210.

[2] M. Bartha, M. Krész, Deterministic soliton graphs, *Informatica* **30** (2006), 281–288.

[3] M. Bartha, M. Krész, Flexible matchings, *in* Proceedings of the 32nd International Workshop on Graph-Theoretic Concepts in Computer Science, Bergen, Norway, June 22-24, 2006. Lecture Notes in Computer Science 4271, pp. 313 - 324.

[4] J. Cheriyan, A. Sebő, Z. Szigeti, Improving on the 1.5 approximation of a smallest 2-edge connected spanning subgraph, *SIAM J. Discret. Math*, **14** (2001), 170–180.

[5] J. Dassow, H. Jürgensen, Soliton automata, *J. Comput. System Sci.* **40** (1990), 154–181.

[6] J. Edmonds, Paths, trees, and flowers, *Canad. J. Math.* **17** (1965), 449–467.

[7] Gabow, H. N., H. Kaplan, and R. E. Tarjan, Unique maximum matching algorithms, *Journal of Algorithms*, **40** (2001), 159–183.

[8] Golumbic, M. C., T. Hirst, and M. Lewenstein, Uniquely restricted matchings, *Algorithmica*, **31** (2001), 139–154.

[9] M. M. Hossain, Some Graph Algorithms of Molecular Switching Devices. Master's thesis, Memorial University of Newfoundland, Canada, 2007.

[10] D. E. Knuth, Semantics of context-free languages, *Math. Systems Theory* **2** (1968), 127–145.

[11] L. Lovász, M. D. Plummer, *Matching Theory*, North Holland, Amsterdam, 1986.

[12] J. Magun, Greedy Matching Algorithms, an Experimental Study, *J. of Experimental Algorithms* **3** (1998), 1–13.