Brief Announcement: Partial Reversal Acyclicity

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1145/1993806.1993880">http://dx.doi.org/10.1145/1993806.1993880</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Association for Computing Machinery (ACM)</td>
</tr>
<tr>
<td>Version</td>
<td>Author's final manuscript</td>
</tr>
<tr>
<td>Accessed</td>
<td>Fri Dec 14 04:19:27 EST 2018</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/73185">http://hdl.handle.net/1721.1/73185</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Creative Commons Attribution-Noncommercial-Share Alike 3.0</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td><a href="http://creativecommons.org/licenses/by-nc-sa/3.0/">http://creativecommons.org/licenses/by-nc-sa/3.0/</a></td>
</tr>
</tbody>
</table>
Brief Announcement: Partial Reversal Acyclicity

Tsvetomira Radeva
MIT, Cambridge, MA
radeva@csail.mit.edu

Nancy Lynch
MIT, Cambridge, MA
lynch@csail.mit.edu

ABSTRACT

Partial Reversal (PR) is a link reversal algorithm which ensures that an initially directed acyclic graph (DAG) is eventually a destination-oriented DAG. While proofs exist to establish the acyclicity property of PR, they rely on assigning labels to either the nodes or the edges in the graph. In this work we show that such labeling is not necessary and outline a simpler direct proof of the acyclicity property.

Categories and Subject Descriptors

G.2 [Discrete Mathematics]: Graph Theory—Graph algorithms

General Terms

Algorithms, Theory

Keywords

Partial Reversal, Link Reversal, Graph Algorithms

1. INTRODUCTION

Link reversal algorithms were first introduced in [1] to provide an efficient graph structure for routing. The main goal of link reversal algorithms is to ensure that all the nodes in a directed acyclic graph (DAG) have paths to a destination node or nodes. These algorithms can also be used to solve problems such as leader election and mutual exclusion [4].

This work focuses on a specific link reversal algorithm: partial reversal (PR). In PR the initial graph is a DAG with a single destination node; however, some nodes may not have paths to the destination. The goal is to ensure that all nodes have paths to the destination. In PR only the nodes, except for a sink, that become sinks (all their incident edges are incoming) take steps. Each node u maintains a list of the edges reversed by its neighbors since the last time u took a step. When u becomes a sink, it reverses only the edges which are not in the list, and then clears the list.

A key property of PR is that it does not create cycles in the graph. This has been proved in [1] and [4]. The proof in [1] assigns to each node an integer 3-tuple such that each edge is directed from a node with a lexicographically larger value of the 3-tuple to a node with a smaller value. The proof establishes the existence of such an assignment forming a total order on the nodes. Consequently, no cycles exist in the graph. In [4], PR is described as a special case of the Binary Link Labels (BLL) algorithm, which uses an assignment of binary labels to edges. The proof of acyclicity of PR in [4] follows from a specific such assignment.

Here, we outline a novel proof of the acyclicity property of PR that is agnostic to node and edge labels. First, we introduce a simpler version of PR and prove its acyclicity property without recourse to external or dynamic labeling. Next, we provide a simulation relation from the original algorithm to the new one, and consequently, our new acyclicity proof also applies to the original PR algorithm.

2. ORIGINAL ALGORITHM

We model the system as an undirected graph \( G = (V, E) \) with a set of nodes \( V \) and a set of edges \( E \). For each node \( u \), \( nbr_u \) is the set of neighbors of \( u \) in \( G \). Consider a directed version of \( G \), denoted \( G' = (V, E') \), such that for a given edge \( \{u, v\} \in E \) either \( \{u, v\} \in E' \) or \( \{v, u\} \in E' \), but not both. Let \( G_{init} \) be such \( G' \) corresponding to the initial state. Let \( in-nbr_u \) and \( out-nbr_u \) be the sets of nodes corresponding to incoming and outgoing edges of a node \( u \) in \( G_{init} \). Note that the sets \( in-nbr_u \) and \( out-nbr_u \) are static and remain unchanged.

Next, we present the original PR algorithm [1], and express it as an I/O automaton (PR) (as described in [2]) with a single set of actions – \( reverse(S) \). The set \( S \) represents all the nodes that are taking a step together. PR has a variable \( list[u] \), for each node \( u \), which consists of all neighbors of \( u \) that took a step since the last time \( u \) took a step. Additionally, PR has a \( dir[u, v] \) variable for each ordered pair \( (u, v) \), which represents the direction of the edge between \( u \) and \( v \).

**Algorithm 1 PR automaton**

**Signature:** \( reverse(S), S \subseteq V \setminus \{D\}, S \neq \emptyset \)

**States:** for each \( u, v \) where \( \{u, v\} \in E \):
- \( dir[u, v] \in \{in, out\} \), initially in if \( v \in in-nbr_u \), else out
- \( dir[v, u] \in \{in, out\} \), initially in if \( u \in in-nbr_v \), else out

**Initial states:** for each \( u, list[u] \), a set of nodes \( W \subseteq \negbr_u \), initially empty

**Transitions:** \( reverse(S) \)

**Precond:** for each \( u \in S \), for each \( v \in \negbr_u \), \( dir[u, v] = in \)

**Effect:** for each \( u \in S \)
- if \( list[u] \neq \negbr_u \) then for each \( v \in \negbr_u \setminus list[u] \):
  - \( dir[u, v] := out; dir[v, u] := in; \)
  - \( list[u] := list[u] \cup \{v\} \)
- else for each \( v \in \negbr_u \):
  - \( dir[u, v] := out; dir[v, u] := in; list[u] := list[u] \cup \{u\} \)

**Tasks:** \( \{reverse(S), S \subseteq V \setminus \{D\}, S \neq \emptyset \} \)
The precondition for the reverse(S) action is that all nodes in S are sinks. The effect of the reversal is that the edge between u and each neighbor of u not in list[u] is reversed. However, if list[u] contains all neighbors of u, then the edges to all neighbors are reversed. Also, each neighbor v of u that has its edge to u reversed, adds u to list[v]. Finally, after reversing the particular edges, u empties list[u]. The following corollaries establish the possible contents of list[u] for any node u.

**Corollary 1.** In all reachable states, for each node u, list[u] ⊆ in-nbrs_u or list[u] ⊆ out-nbrs_u.

**Corollary 2.** In all reachable states, if u is a sink, then list[u] = in-nbrs_u or list[u] = out-nbrs_u.

### 3. NEW ALGORITHM

In NewPR, nodes use only the initial in-nbrs and out-nbrs sets to determine which edges to reverse in each step. Whenever a node is a sink, it reverses either its in-nbrs or out-nbrs set, alternating between the two. A history variable count[u] keeps track of the number of steps u has taken so far, and a derived variable parity[u] represents the parity of count[u]. The precondition for a node u to perform a reverse(u) action is that it is a sink. The effect of the reversal is that, depending on the value of parity[u], either the edges corresponding to in-nbrs_u or out-nbrs_u are reversed. Also, count[u] is incremented.

**Algorithm 2** NewPR automaton

**Signature:** reverse(u), u ∈ V, u ≠ D

**States:** for each u, v where {u, v} ∈ E:
- dir[u, v] ∈ {in, out}, initially in if v ∈ in-nbrs_u, else out
- dir[v, u] ∈ {in, out}, initially in if u ∈ in-nbrs_u, else out
for each node u, count[u], integer, initially 0

**Derived:** for each node u, parity[u] ∈ {even, odd}, even if count[u] is even; else odd

**Precond:** for each v ∈ nbrs_u, dir[u, v] = in

**Effect:** if parity[u] = even then
- for each v ∈ in-nbrs_u: dir[u, v] := out; dir[v, u] := in
else
- for each v ∈ out-nbrs_u: dir[u, v] := out; dir[v, u] := in
- count[u] := count[u] + 1

**Tasks:** \{reverse(u), u ∈ V, u ≠ D\}

Note that it is possible that a node u does not reverse any edges because both in-nbrs_u = ∅ or out-nbrs_u = ∅. This case occurs only if u is initially a sink or a source. When such an action is performed, u increments its step counter without reversing any edges. Therefore, u remains a sink but now the parity has the correct value, so u can reverse its incident edges in the next step.

A main difference between the two algorithms is that while PR keeps a dynamic list of nodes, NewPR maintains two static lists of nodes to determine the edges to be reversed. The description of the algorithm in NewPR is simpler and makes the algorithm easier to understand.

Since the input to the PR algorithm is a DAG, we can embed it in a plane, ensuring all edges are initially directed from left to right. Therefore, for each node u all nodes in in-nbrs_u are to the left of u, and all nodes in out-nbrs_u are to the right of u. The following invariants establish some properties of NewPR, and are combined into the main theorem concluding that PR maintains acyclicity.

**Invariant 1.** In any reachable state, if u and v are neighbors, then:
(a) If parity[u] = parity[v] = even, then the edge {u, v} is directed from left to right.
(b) If parity[u] = parity[v] = odd, then the edge {u, v} is directed from right to left.

**Invariant 2.** In any reachable state, if u and v are neighbors, then:
(a) If count[u] = n, then count[v] ∈ {n - 1, n, n + 1}.
(b) If count[u] = n, where n is odd, and v is to the right of u, then count[v] = n.
(c) If count[u] = n, where n is even, and v is to the left of u, then count[v] = n.
(d) If count[u] > count[v], then the edge {u, v} is directed from u to v.

**Theorem 1.** PR maintains acyclicity.

**Proof.** Suppose in contradiction that there exists a cycle in some reachable state s of the system. Therefore, there is a sequence of nodes: u, s_1, ..., s_n, u such that the edges between these nodes are directed from u to v_1, from v_i to v_{i+1} for all 1 ≤ i < n, and from v_n to u. By Invariant 2 (d) the number of steps of the nodes in the sequence is non-increasing, and because the nodes form a cycle, s.count[u] = s.count[v_1] = ... = s.count[v_n]. Let v_{i-1}, v_i, v_{i+1} be a sequence on nodes where v_i is the rightmost node in the cycle. Assume the edge {v_{i-1}, v_i} is directed from left to right, and the edge {v_i, v_{i+1}} is directed from right to left. Since, s.count[v_{i-1}] = s.count[v_i] = s.count[v_{i+1}], s.parity[v_{i-1}] = s.parity[v_i] = s.parity[v_{i+1}] = p. By Invariant 1 (b) applied to v_{i-1} and v_i, p = even. By Invariant 1 (a) applied to v_i and v_{i+1}, p = odd, a contradiction.

### 4. SIMULATION RELATION

We define a simulation relation R from states of PR to states of NewPR which guarantees that the two algorithms preserve the same edge directions. Let s be a state of PR and t be a state of NewPR. We define (s, t) ∈ R if:

1. t.G' = s.G'
2. For each u, if t.parity[u] = even, then s.list[u] ⊆ out-nbrs_u; else s.list[u] ⊆ in-nbrs_u.

**Theorem 2.** For each reachable state s of PR there exists a reachable state t of NewPR such that (s, t) ∈ R.

**Corollary 3.** PR maintains acyclicity.

The proof follows from Theorem 1, Theorem 2, and the fact that by part 1 of the simulation relation both algorithms produce the same directed versions of the graph.

### 5. REFERENCES


