Abstract

This document contains some auxiliary proofs, and pseudo-code for our algorithms to aid implementers.

1. Auxiliary Proofs

**Proposition 1.** The new definition of splitAnc (the ancestor-splitting operation on which unzipping relies) in Table 2 of the main paper is equivalent to the original definition from [1].

**Proof.** Denote the original definition of splitAnc, defined in Table 2 of [1], as splitAnc\(^{(1)}\), and the new definition, defined in Table 2 of the main paper here, as splitAnc\(^{(2)}\).

Then:

\[
\text{splitAnc}^{(2)}(d, N) = \text{tugged}(d, N) \cup \text{remnants}(\psi^d_H(N))
\]

\[
\cup \{(d, R') : R' \in \text{ccs}(R(N)) \}
\]

\[
\cup \{(d, R') : R' \in \text{ccs}(\text{remnantRegion}(\psi^d_H(N), N)) \}
\]

\[
\cup \{(d, R') : R' \in \text{ccs}(\varphi^d_H(N)) \setminus R(N) \}
\]

\[
= \text{splitAnc}^{(1)}(d, N)
\]

By extensionality, splitAnc\(^{(2)}\) is thus equivalent to splitAnc\(^{(1)}\), as required. □

**Proposition 2.** Generalised multi-node unzipping generalises classic multi-node unzipping from [1].

**Proof.** Observe that any classic unzip of the form unzips\(_{\text{min}}(\cdot)\) can be equivalently expressed as a generalised unzip of the form unzips\(_{\text{min}}(V_{\text{dmin}}^{dmin})\), i.e. a generalised unzip in which \(C\) has been set to \(V_{\text{dmin}}^{dmin}\). To see why this must be the case, simply compare the definitions of the two operations from the main paper. First note that \(D_{\text{min}}(C) = D_{\text{min}}(V_{\text{dmin}}^{dmin}) = d_{\text{min}}\), so the cases for the two definitions precisely match. Then reason that for every \(d \in (D_{\text{min}}(C), D_{\text{max}}(N))\):

\[
\Omega_{\psi^d_H(N, C)} = \Omega_{\psi^d_H(N, V_{\text{dmin}}^{dmin})}
\]

\[
= \psi_{\psi^d_H(N, d)}(V_{\text{dmin}}^{dmin})
\]

\[
= \{ n \in \psi_{\psi^d_H(N, d)}(V_{\text{dmin}}^{dmin}) : D(n) > d_{\text{min}} \}
\]

\[
= \psi_{\psi^d_H(N, d)}
\]

The definitions of the classic and generalised unzips described above are then precisely the same. Since any classic unzip can be expressed as a generalised unzip in this way, generalised multi-node unzipping then clearly generalises classic multi-node unzipping as required. □

2. Pseudo-Code

Listings 1–3 provide commented pseudo-code for generalised multi-node unzipping, FastTug [2] and SimpleTug, respectively, with the intention of making it easy for others to reimplement them if desired. The pseudo-code language we use is formally described in Appendix F of [3], and its data structures are largely based on those in the C++ Standard Library [4].
Listing 1 Generalised Multi-Node Unzipping

```typescript
function unzip_nodes(nodes: Set<NodeID>, cut: Cut, unzipMode: UnzipMode) → Map<NodeID,Chain>
var chains: Map<NodeID,Chain>;

// Unzip the nodes in the selection, starting with those deepest in the hierarchy.
var nodesByDepth: Map<Int,Set<NodeID>> := group_by_depth(nodes);
var depth: Int := nodesByDepth.max_key();
var curs: Set<NodeID> := nodesByDepth[depth];
while depth > cut.min_depth()
    // Group the current nodes by parent.
    var parentToSelectedChildMap: Map<NodeID,Set<NodeID>>;
    for cur: NodeID ∈ curs
        parentToSelectedChildMap[parent_of(cur)].insert(cur);
    
    // For each parent node in turn:
    var result: Set<NodeID>;
    for (parent, selectedChildren) ∈ parentToSelectedChildMap
        // If the parent node is on the cut, remove all of its selected children from the list of current nodes.
        if cut.contains(parent) then
            curs := curs \ selectedChildren;
        
        // Determine the unselected children of the parent node.
        var unselectedChildren: Set<NodeID> := children_of(parent) \ selectedChildren;
        
        // Calculate the connected components of the selected children.
        var ccs: Vector<Set<NodeID>> := find_connected_components(selectedChildren);
        
        // Depending on the unzip node, add to these either the connected components of the unselected children, 
        // or a single (potentially unconnected) component containing the unselected children (if any).
        if unzipMode = UNZIPMODE_DEFAULT then
            ccs.append(find_connected_components(unselectedChildren));
        else if !unselectedChildren.empty() then
            ccs.push_back(unselectedChildren);
        
        // Split the parent node and store the results.
        result.append(split_node(parent, ccs));
    
    // Prepend each existing chain with its head node’s parent, and remove that parent from the split results.
    for each chain: Chain ∈ chains
        var p: NodeID := parent_of(chain.front());
        chain.push_front(p);
        result.erase(p);
    
    // Add a new singleton chain for each remaining node in the split results.
    for each n: NodeID ∈ result
        chains.insert(n, [n]);
    
    // Update the current nodes and the depth.
    curs := parents_of(curs);
    depth := depth - 1;

    // Add in any new nodes from the selection whose depth we have now reached.
    curs.append(nodesByDepth[depth]);

return chains;
```

Listing 2 FastTug

```plaintext
function tug_fast (node : NodeID , cut : Cut)
  // Find the leaves that are adjacent to those subsumed by the selected node (the 'adjacent leaves').
  var adjLeaves : Set<NodeID> := find_adjacent_leaves (node);
  // Unzip these nodes up to the cut, ripping their ancestors if necessary in the process.
  var chains : Map<NodeID , Chain> := unzip_nodes (adjLeaves , cut , UNZIPMODE_RIP);
  // Determine which nodes might have been ripped, ordered by depth (greatest first).
  var maybeRippedNodes : Map<Int , Set<NodeID> , Greater<Int> >;
  for (_, chain ) ∈ chains
    for n in chain
      maybeRippedNodes [n. layer () ]. insert (n);
  // Fix up the nodes in non-increasing order of depth.
  for (_, parents ) ∈ maybeRippedNodes
    for parent ∈ parents
      var children : Set<NodeID> := children_of (parent);
      var ccs : Vector<Set<NodeID> > := find_connected_components (children);
      split_node (parent , children);
```

Listing 3 SimpleTug

```plaintext
function tug_simple (node : NodeID , cut : Cut)
  // Find the leaves that are adjacent to those subsumed by the selected node (the 'adjacent leaves').
  var adjLeaves : Set<NodeID> := find_adjacent_leaves (node);
  // Unzip these nodes up to the cut.
  unzip_nodes (adjLeaves , cut , UNZIPMODE_DEFAULT);
```

References


