

Simplifying TugGraph using Zipping Algorithms

SUPPLEMENTARY MATERIAL

S. Golodetz^a, A. Arnab^b, I.D. Voiculescu^c, S.A. Cameron^c

^aOxford Research Group, FiveAI Ltd., Oxford, United Kingdom

^bDepartment of Engineering Science, University of Oxford, United Kingdom

^cDepartment of Computer Science, University of Oxford, United Kingdom

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:

$$\begin{aligned}
 & \text{splitAnc}_{\mathcal{H}}^{(2)}(d, N) \\
 &= \text{tugged}(d, N) \cup \text{remnants}(\psi_{\mathcal{H}}^d(N), N) \\
 &= \{(d, R') : R' \in \text{ccs}(\mathcal{R}(N))\} \\
 &\cup \{(\mathcal{D}(\psi_{\mathcal{H}}^d(N)), R') : R' \in \text{ccs}(\text{remnantRegion}(\psi_{\mathcal{H}}^d(N), N))\} \\
 &= \{(d, R') : R' \in \text{ccs}(\mathcal{R}(N))\} \\
 &\cup \{(d, R') : R' \in \text{ccs}(\mathcal{R}(\psi_{\mathcal{H}}^d(N)) \setminus R(N))\} \\
 &= \text{splitAnc}_{\mathcal{H}}^{(1)}(d, N)
 \end{aligned}$$

By extensionality, `splitAnc`⁽²⁾ is thus equivalent to `splitAnc`⁽¹⁾, as required. \square

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

Proof. Observe that any classic unzip of the form `unzip` _{$N; d_{\min}$} (\cdot) can be equivalently expressed as a generalised unzip of the form `unzip` _{$N; V_{\mathcal{H}}^{d_{\min}}$} (\cdot), i.e. a generalised unzip in which C has been set to $V_{\mathcal{H}}^{d_{\min}}$. To see why

this must be the case, simply compare the definitions of the two operations from the main paper. First note that $\mathcal{D}_{\min}(C) = \mathcal{D}_{\min}(V_{\mathcal{H}}^{d_{\min}}) = d_{\min}$, so the cases for the two definitions precisely match. Then reason that for every $d \in (\mathcal{D}_{\min}(C), \mathcal{D}_{\max}(N))$:

$$\begin{aligned}
 & \Omega_{\mathcal{H}}^d(N, C) \\
 &= \Omega_{\mathcal{H}}^d(N, V_{\mathcal{H}}^{d_{\min}}) \\
 &= \Psi_{\mathcal{H}}^d(N_{>d}) \cap \Psi_{\mathcal{H}}^-(V_{\mathcal{H}}^{d_{\min}}) \\
 &= \{n \in \Psi_{\mathcal{H}}^d(N_{>d}) : \mathcal{D}(n) > d_{\min}\} \\
 &= \Psi_{\mathcal{H}}^d(N_{>d})
 \end{aligned}$$

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. \square

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

```
1 function unzip_nodes(nodes: Set<NodeID>, cut: Cut, unzipMode: UnzipMode) ⇨ Map<NodeID,Chain>
2   var chains: Map<NodeID,Chain>;
3
4   // Unzip the nodes in the selection, starting with those deepest in the hierarchy.
5   var nodesByDepth: Map<Int,Set<NodeID> > := group_by_depth(nodes);
6   var depth: Int := nodesByDepth.max_key();
7   var curs: Set<NodeID> := nodesByDepth[depth];
8   while depth > cut.min_depth()
9     // Group the current nodes by parent.
10    var parentToSelectedChildMap: Map<NodeID,Set<NodeID>>;
11    for cur: NodeID ∈ curs
12      parentToSelectedChildMap[parent_of(cur)].insert(cur);
13
14    // For each parent node in turn:
15    var result: Set<NodeID>;
16    for (parent, selectedChildren) ∈ parentToSelectedChildMap
17      // If the parent node is on the cut, remove all of its selected children from the list of current nodes.
18      if cut.contains(parent) then
19        curs := curs \ selectedChildren;
20
21      // Determine the unselected children of the parent node.
22      var unselectedChildren: Set<NodeID> := children_of(parent) \ selectedChildren;
23
24      // Calculate the connected components of the selected children.
25      var ccs: Vector<Set<NodeID> > := find_connected_components(selectedChildren);
26
27      // Depending on the unzip mode, add to these either the connected components of the unselected children,
28      // or a single (potentially unconnected) component containing the unselected children (if any).
29      if unzipMode = UNZIPMODE_DEFAULT then
30        ccs.append(find_connected_components(unselectedChildren));
31      else if !unselectedChildren.empty() then
32        ccs.push_back(unselectedChildren);
33
34      // Split the parent node and store the results.
35      result.append(split_node(parent, ccs));
36
37      // Prepend each existing chain with its head node's parent, and remove that parent from the split results.
38      for each chain: Chain ∈ chains
39        var p: NodeID := parent_of(chain.front());
40        chain.push_front(p);
41        result.erase(p);
42
43      // Add a new singleton chain for each remaining node in the split results.
44      for each n: NodeID ∈ result
45        chains.insert(n, [n]);
46
47      // Update the current nodes and the depth.
48      curs := parents_of(curs);
49      depth := depth - 1;
50
51      // Add in any new nodes from the selection whose depth we have now reached.
52      curs.append(nodesByDepth[depth]);
53
54   return chains;
```

Listing 2 FastTug

```
1 function tug_fast(node: NodeID, cut: Cut)
2   // Find the leaves that are adjacent to those subsumed by the selected node (the 'adjacent leaves').
3   var adjLeaves: Set<NodeID> := find_adjacent_leaves(node);
4
5   // Unzip these nodes up to the cut, ripping their ancestors if necessary in the process.
6   var chains: Map<NodeID,Chain> := unzip_nodes(adjLeaves, cut, UNZIPMODE_RIP);
7
8   // Determine which nodes might have been ripped, ordered by depth (greatest first).
9   var maybeRippedNodes: Map<Int,Set<NodeID>,Greater<Int> >;
10  for (_,chain) ∈ chains
11    for n in chain
12      maybeRippedNodes[n.layer()].insert(n);
13
14  // Fix up the nodes in non-increasing order of depth.
15  for (_,parents) ∈ maybeRippedNodes
16    for parent ∈ parents
17      var children: Set<NodeID> := children_of(parent);
18      var ccs: Vector<Set<NodeID> > := find_connected_components(children);
19      split_node(parent, children);
```

Listing 3 SimpleTug

```
1 function tug_simple(node: NodeID, cut: Cut)
2   // Find the leaves that are adjacent to those subsumed by the selected node (the 'adjacent leaves').
3   var adjLeaves: Set<NodeID> := find_adjacent_leaves(node);
4
5   // Unzip these nodes up to the cut.
6   unzip_nodes(adjLeaves, cut, UNZIPMODE_DEFAULT);
```

References

- [1] S. Golodetz, I. Voiculescu, S. Cameron, Simpler Editing of Graph-Based Segmentation Hierarchies using Zipping Algorithms, *Pattern Recognition* 70 (2017) 44–59. [1](#)
- [2] D. Archambault, T. Munzner, D. Auber, Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths, *IEEE Transactions on Visualization and Computer Graphics* 17 (3) (2011) 276–289. [1](#)
- [3] S. Golodetz, Zipping and Unzipping: The Use of Image Partition Forests in the Analysis of Abdominal CT Scans, Ph.D. thesis, University of Oxford (2011). [1](#)
- [4] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference*, 2nd Edition, Addison Wesley, 2012. [1](#)