Simpler Editing of Graph-Based Segmentation Hierarchies using Zipping Algorithms
Supplementary Material

Stuart Golodetz[a,*], Irina Voiculescu[b], Stephen Cameron[b]

[a]*Department of Engineering Science, University of Oxford, Parks Road, Oxford OX1 3PJ, United Kingdom*
[b]*Department of Computer Science, University of Oxford, Wolfson Building, Parks Road, Oxford, OX1 3QD, United Kingdom*

Abstract

This document contains: (i) a more detailed explanation of hierarchical selections, (ii) a detailed qualitative example in which we compare non-sibling node merging to the equivalent sequence of manual split and merge operations, (iii) pseudo-code for our algorithms to aid those who wish to implement them, (iv) a number of important proofs about our algorithms, and (v) complexity analysis for our algorithms.

1. Hierarchical Selections

In order for users to interact with graph hierarchies, they must be able to specify parts of them with which to work. For example, to perform a multi-node unzip (see Section 4.1.2 of the main paper), they must be able to select a set of nodes to unzip.

Whilst it is possible to represent a selection of nodes in a hierarchy as a simple set, this models a slightly unnatural concept of selection that allows parent and child nodes in the hierarchy to be individually selected / deselected. A more natural, *hierarchical* concept of selection (Golodetz (2011); Golodetz et al. (2009)) can be derived from the notion that selecting / deselecting a node in the hierarchy should implicitly select / deselect all of its descendants. We refer to the data structure that models this concept of selection as a *hierarchical selection* over a graph hierarchy (see Figure 6 of the main paper): internally, it represents a selection of nodes in the base graph of the hierarchy using a set of stored nodes that is as small as possible (with a preference for nodes that are higher up in the hierarchy when there is a choice).

The three most important operations that hierarchical selections support are (i) adding a node, (ii) removing a node, and (iii) checking whether a node is selected. We briefly describe how node addition and removal work in the following subsections. To check whether a node is selected, we simply test whether it, or any of its ancestors in the graph hierarchy, are contained in the set of stored nodes.

1.1. Adding a node

There are three cases to consider when adding a node:

(i) The node is already selected (either it or one of its ancestors is contained in the set of stored nodes).

(ii) Neither the node itself nor any of its descendants is selected.

(iii) The node itself is not selected, but one or more of its descendants is selected.

In case (i), there is clearly nothing to do. Cases (ii) and (iii) require more work. In case (iii), all descendants of the node must first be removed from the set of stored nodes. Then, in both cases (ii) and (iii), the node to be selected must be added to the set of stored nodes. Finally, the selection must be *consolidated* by replacing any node whose children are all in the set of stored nodes with the node itself, until this is no longer possible. In practice, only ancestors of the added node need to be considered during consolidation: all other nodes in the representation will be unaffected by the addition of the node. An example of case (iii), which is the most general case, is shown in Figure 1. Additional implementation details, including pseudo-code, can be found in Golodetz (2011).

1.2. Removing a node

There are four cases to consider when removing a node:

(i) Neither the node itself nor any of its ancestors or descendants is contained in the set of stored nodes.

(ii) The node itself is contained in the set of stored nodes.

(iii) The node itself is not contained in the set of stored nodes, but one or more of its descendants is.

(iv) The node itself is not contained in the set of stored nodes, but one of its ancestors is.

In case (i), there is clearly nothing to do. Case (ii) simply involves removing the node from the set of stored nodes. In case (iii), it suffices to enumerate all the descendants of the node that are in the set of stored nodes and remove them.

Case (iv), in which one of the node's ancestors is in the set of stored nodes, is more interesting. In this case, we must find the *trail* of nodes in the hierarchy that leads down from the ancestor in question to the node we want to remove (not including the

*Corresponding author
Email addresses: stuart.golodetz@eng.ox.ac.uk (Stuart Golodetz), irina@cs.ox.ac.uk (Irina Voiculescu), cameron@cs.ox.ac.uk (Stephen Cameron)

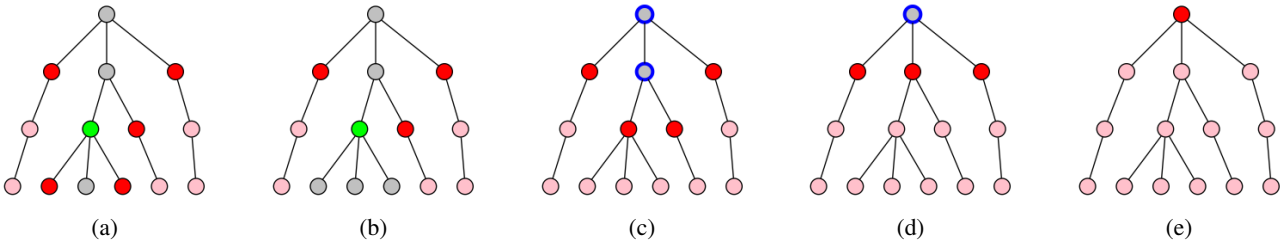


Figure 1: Adding a node to a hierarchical selection (red nodes are those that are explicitly stored in the representation, pink nodes are those that are implicitly selected because one of their ancestors is in the representation): (a) the node to be added is highlighted in green; (b) any selected descendants of the node being added are removed from the representation; (c) the new node is added to the representation, and the nodes to be checked during *consolidation* are circled in blue; (d) the parent of the added node is consolidated; (e) the grandparent of the added node is consolidated.

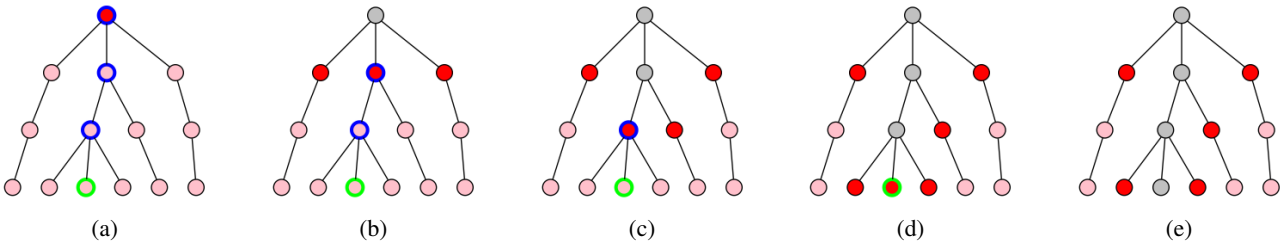


Figure 2: Removing a node from a hierarchical selection (red nodes are those that are explicitly stored in the representation, pink nodes are those that are implicitly selected because one of their ancestors is in the representation): (a) the node to be removed is circled in green, and the nodes on the trail leading down from its ancestor in the representation are circled in blue; (b)–(d) the nodes on the trail are replaced in the representation with their children, until the node to be removed is directly contained in the representation; (e) the node itself is then removed from the representation.

node itself). We then replace all the nodes on this trail (from the ancestor downwards) in the set of stored nodes with their children, until the node we want to remove is itself in the set of stored nodes, at which point we can simply remove it. An example of this process is shown in Figure 2. Further details and pseudo-code can be found in Golodetz (2011).

2. Non-Sibling Node Merging vs. Manual Split and Merge

The non-sibling node merging operation we describe in Section 5.1 of the main paper is designed to be both conceptually simple to understand and intuitive to use in practice. However, some non-sibling merges (particularly those that involve merging nodes whose lowest common ancestor is multiple layers up in the hierarchy) can involve significant structural changes behind the scenes. By hiding this complexity, non-sibling node merging significantly reduces the interaction burden on the user for such merges. In this section, we demonstrate this using a merge that is quite difficult to perform manually, but trivial to perform using our algorithm.

Figure 3 shows an example in which the user is trying to merge four nodes (shown in (a)) in layer 3 of a graph hierarchy for an abdominal CT scan (the four nodes together correspond to the right kidney of the patient in question). However, because the nodes do not share a common parent or grandparent in layers 4 and 5 of the hierarchy (as shown in (b) and (c)), they cannot be merged using straightforward sibling node merging; instead, we must perform a *non-sibling* node merge by unzipping them up to just below their common ancestor (in this case the top of the hierarchy) and zipping the resulting node chains back down the hierarchy to complete the merge. Performed manually, this is a laborious process: since there are two parents and two grandparents of the nodes, four separate split operations must be performed, and three sibling node merge operations must then be performed to complete the process. Moreover, whilst sibling node merges are easy for the user to specify, splits are not: for each split, the user must specify the components into which to split the node, which can consume significant time. Using our non-sibling node merging algorithm, exactly the same result can be achieved *in a single step*: the user simply needs to select the nodes (as in (a)) and request that they be merged; all of the tedious book-keeping can be performed automatically behind the scenes.

3. Pseudo-Code

Listings 1–5 provide commented pseudo-code for the algorithms we describe in the main paper, to assist those who wish to reimplement them. A reference for the pseudo-code language we use can be found in Appendix F of Golodetz (2011). The data structures we use (such as Vector, Set and Map) correspond to those that can be found in the standard libraries of many popular languages, although they largely mirror `std::vector`, `std::set` and `std::map` from C++ (Stroustrup (2013)).

In a practical implementation, the efficiency of our algorithms depends on the choice made about how to represent the higher-level edges in the hierarchy. For example, at various points, we need to compute the connected components of a set of nodes at a particular depth in the hierarchy, a task that can be achieved much more efficiently if the edges at that depth are readily available than if they have to be continually recomputed from the base graph (this difference in efficiency is particularly

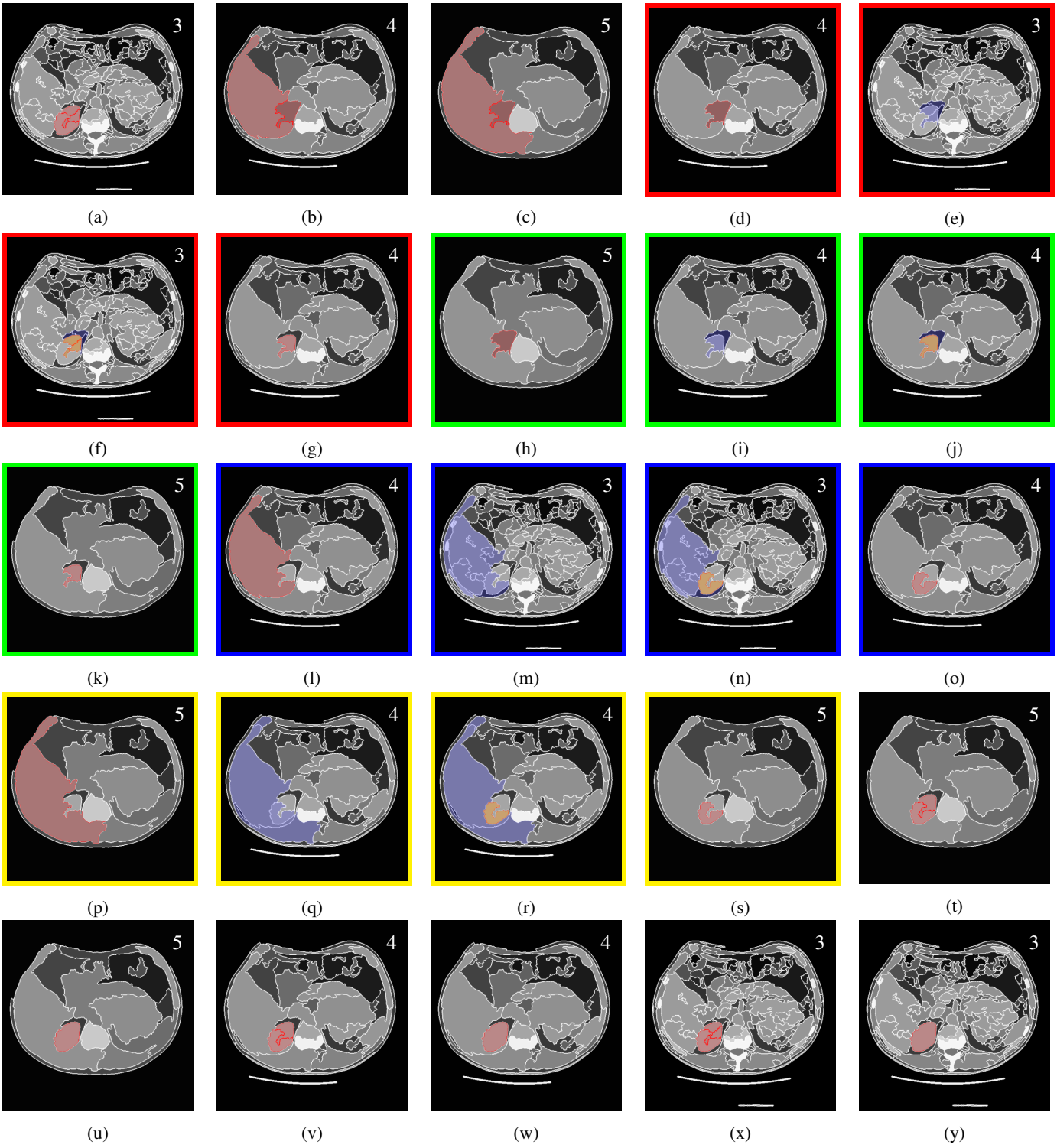


Figure 3: An example showing how an invocation of our *non-sibling node merging* algorithm can replace numerous manual steps, reducing the interaction burden on the user. The number on each image denotes its hierarchy layer. In (a), the user selects a set of nodes to be merged (in layer 3 of a hierarchy). The parents and grandparents of the nodes (in layers 4 and 5) are shown in (b) and (c) respectively: note that the nodes do not have a common parent or grandparent, so their parents and grandparents must be split before they themselves can be merged. Subfigures (d)-(g) show the first parent node being split: (d) the node to be split is selected; (e) the split is requested; (f) the components into which to split the node are specified; (g) the split is finalised. Subfigures (h)-(s) show the same process for the first grandparent, the second parent and the second grandparent (there are four subfigures for each, denoted with green, blue and yellow borders respectively). Finally, straightforward *sibling node merging* is used to zip the nodes together again back down the hierarchy: (t) and (u) show the grandparents being merged, (v) and (w) show the parents being merged, and (x) and (y) show the original nodes being merged. To achieve this result manually, we have had to perform the four split operations in (d)-(g), (h)-(k), (l)-(o) and (p)-(s) (specifying the split components in each case) and three sibling merge operations. The same result can be achieved *in a single step* using our non-sibling node merging algorithm.

relevant in a deep hierarchy). In *millipede*, we therefore explicitly store the edges at each depth, and maintain them when splitting or merging nodes. This needs to be borne in mind when we analyse the complexity of our algorithms in Section 5.

Furthermore, we note that for practical reasons, the implementations of our algorithms differ in a few places from the mathematical definitions presented in the main text. We mention these differences on a per-algorithm basis here for the avoidance of confusion.

3.1. Single-Node Unzipping

- The splitting of nodes is more general in the practical implementation. In the main text, we defined a `splitPar` operation that specifies the nodes that would result from splitting a parent node around some of its children. In practice, we define a more general hierarchy-mutating operation called `split_node` that splits the specified parent node into pieces corresponding to a specified set of connected components. For single-node unzipping, these connected components will be the ones specified in the definition of `splitPar`, but `split_node` can actually be used to split a node into pieces corresponding to an arbitrary set of connected components. This is practically useful in that it allows us to provide users with a general splitting operation in the interface.

- The chains are not stored as a set in practice. We observe that both of the higher-level algorithms (non-sibling node merging and parent switching) use single-node unzipping and then need to access the chain in the resulting chain set corresponding to the parent of the node that was being unzipped. To make this lookup efficient, we therefore implement the chain set as a vector (i.e. an array) of chains, and arrange for the first chain in the array to be the one that corresponds to the parent of the node that was being unzipped. To do this, we add a singleton chain containing the node to be unzipped to the chain set at the start of the algorithm. The final chain should not contain the actual node being unzipped (only the nodes above it), so we remove it from the end of the chain again before returning. The practical effect of this optimisation is to make the lookup of the relevant chain constant time.

3.2. Multi-Node Unzipping

- The grouping of current nodes by parent is handled straightforwardly in `splitPars` in the mathematical definition, whereas here we have to group them manually.

- Again, the chains are not stored as a set, since we need a convenient way to look up individual chains. Unlike for single-node unzipping, here there is no obvious chain to prioritise, so we instead store the chains as a map, keyed on the deepest node in each chain.

3.3. Non-Sibling Node Merging

- Unlike in the mathematical definition, here we avoid trying to merge the nodes in a set $K_i \in \mathcal{K}$ of size 1, to save computation.

4. Proofs

4.1. Our Editing Algorithms Preserve Region Contiguity

As explained in Section 3 of the main paper, an important property of each hierarchy that we consider in this paper is that the regions of all of the nodes in the hierarchy are connected in the hierarchy's base graph. For convenience, we will refer to this property as *region contiguity*. In this section, we prove that this property is preserved by all of our editing algorithms. First of all, recall the closed form definition of multi-node unzipping from the main paper:

$$\begin{aligned}\mathrm{splitAnc}_{\mathcal{H}}(d,N) = {} & \{(d,R') : R' \in \mathrm{ccs}_{\mathcal{H}}(\mathcal{R}(N))\} \\ & \cup \left\{(d,R'') : R'' \in \mathrm{ccs}_{\mathcal{H}}(\mathcal{R}(\Psi^d_{\mathcal{H}}(N)) \setminus \mathcal{R}(N))\right\}\end{aligned}$$

$$\mathrm{splitAncs}_{\mathcal{H}}(d,N) = \bigcup \left\{\mathrm{splitAnc}_{\mathcal{H}}(d, \Psi^-_{\mathcal{H}}(n) \cap N) : n \in \Psi^d_{\mathcal{H}}(N)\right\}$$

$$\mathrm{unzip}^{\mathcal{M}*}_{N;d_{\min}}(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus \Psi^d_{\mathcal{H}}(N_{>d})) \cup \mathrm{splitAncs}_{\mathcal{H}}(d, N_{>d}) & \text{if } d \in [d_{\min}, \mathcal{D}_{\max}(N)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

$$\mathrm{unzip}^{\mathcal{M}*}_{N;d_{\min}}(\mathcal{H}) = (\mathrm{unzip}^{\mathcal{M}^*}_{N;d_{\min}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H})$$

Lemma 1. *Multi-node unzipping preserves the region contiguity of the hierarchy.*

Proof. The effect of a multi-node unzip is to remove some of the ancestors of the nodes in N, and to replace them with the results of splitting them around their respective descendants in N. There are thus precisely two types of changes being made to the hierarchy: some existing nodes are being removed, and some new nodes resulting from calls to `splitAncs` are being added. Removing existing nodes from the hierarchy clearly preserves region contiguity, since the remaining nodes are a subset of those that were in the hierarchy before (whose regions were all connected). Moreover, each new node added comes from a call to `splitAncs`, and therefore indirectly from a call to `splitAnc`. But the regions of these nodes are connected by definition, since `splitAnc` yields nodes of the form (d, R'), where R' is always a connected component in the hierarchy's base graph. Hence multi-node unzipping must preserve the region contiguity of the hierarchy. □

Corollary 2. *Single-node unzipping preserves the region contiguity of the hierarchy.*

Proof. As can be seen from their definitions, multi-node unzipping strictly generalises single-node unzipping. In particular, unzipping a single node with single-node unzipping is equivalent to unzipping a singleton set containing that node with multi-node unzipping. As a result, since multi-node unzipping preserves region contiguity, so does single-node unzipping. □

Now recall the definition of chain zipping from the main paper:

$$\mathrm{merge}(N) = \begin{cases} \{(\mathcal{D}(N), \mathcal{R}(N))\} & \text{if defined}(\mathcal{D}(N)) \text{ and connected}(\mathcal{R}(N)) \\ N & \text{otherwise} \end{cases}$$

$$\begin{aligned}\mathrm{zipPre}_{\mathcal{H}}(C) = {} & |C| > 0 \text{ and } \forall c \in C \,.\, (|c| > 0 \text{ and } \forall n \in c \,.\, (n \notin V_{\mathcal{H}})) \\ & \text{and defined}(\pi_{\mathcal{H}}(N^C_{d_{\min}})) \text{ and } \forall d \,.\, \text{connected}(\mathcal{R}(N^C_d))\end{aligned}$$

$$\mathrm{zip}_C(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus N^C_d) \cup \mathrm{merge}(N^C_d) & \text{if } \mathrm{zipPre}_{\mathcal{H}}(C) \text{ and } d \in [d_{\min}, d_{\max}] \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

Lemma 3. *Chain zipping preserves the region contiguity of the hierarchy.*

Proof. The effect of a chain zipping operation is to remove the nodes in the chains being zipped from the hierarchy, and replace them with the results of merging the nodes at each depth in the chains together. As explained above, removing nodes from the hierarchy clearly preserves region contiguity. Moreover, all the nodes that are being added in this case result from merging together nodes that we ensure are connected in the definition of `zipPre` (note the last conjunct). As a result, we only ever add new nodes whose regions are connected, and hence chain zipping as a whole must preserve the region contiguity of the hierarchy. □

Theorem 4. *Since both our unzipping and zipping algorithms preserve the region contiguity of the hierarchy, any algorithm that performs a finite sequence of such unzipping and zipping operations also preserves it.*

Proof. Trivial (by induction on the length of the sequence). □

Corollary 5. *Both non-sibling node merging and parent switching preserve the region contiguity of the hierarchy.*

Proof. Both algorithms perform finite sequences of unzipping and zipping operations, and so preserve the region contiguity of the hierarchy by Theorem 4. □

4.2. Chain Zipping Reverts Single-Node Unzipping

In addition to proving that our zipping algorithms preserve important properties of the hierarchies on which they operate, it is also helpful to show how they relate to each other. In particular, the names of the operations imply that by applying an appropriate zipping operation, we ought be able to revert the effects of a previous unzip. In this section, we show that this is indeed the case, by demonstrating that zipping together the chains resulting from a single-node unzip has the effect of reverting that operation.

Lemma 6. *The regions of the 'current' nodes used during single-node unzipping of a node n to a depth* $d_{\min}$ *are all equal to the region of n, i.e.* $\forall d \in [d_{\min}, \mathcal{D}(n)] \,.\, (\mathcal{R}(\text{cur}_d) = \mathcal{R}(n))$.

Proof. As per the definition in the main paper, $\text{cur}_{\mathcal{D}(n)} = n$, so the base case is clearly true. Moreover, for $d_{\min} \leq d < \mathcal{D}(n)$, $\text{cur}_d = \pi_{\mathcal{H}_d}(\text{cur}_{d+1})$, i.e. cur_d is the parent of cur_{d+1} in $\mathcal{H}_d$, the hierarchy generated by removing the parent of cur_{d+1} in $\mathcal{H}_{d+1}$ and replacing it with the nodes in $\texttt{splitPar}_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\})$. By definition, this latter set contains a single node with a region of $\mathcal{R}(n)$, which is the node that will become cur_d, together with a number of other nodes corresponding to the connected components of what is left of the original parent region. The stated result thus follows directly from the definition of `splitPar`. □

Lemma 7. *The nodes added to* chains_d *during single-node unzipping of a node n are precisely those in* $\text{splitAnc}_{\mathcal{H}}(d, \{n\})$.

Proof. From the definition of chains_d, the nodes added are:

$$
\begin{aligned}
& \{\pi_{\mathcal{H}_d}(x) : [x] +\!\!+ xs \in \text{chains}_{d+1}\} \\
& \cup (\text{splitPar}_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\}) \setminus \{\pi_{\mathcal{H}_d}(x) : [x] +\!\!+ xs \in \text{chains}_{d+1}\}) \\
= \;& \text{splitPar}_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\}) \\
= \;& \{(d, R') : R' \in \text{ccs}_{\mathcal{H}_{d+1}}(\mathcal{R}(\{\text{cur}_{d+1}\}))\} \\
& \cup \{(d, R'') : R'' \in \text{ccs}_{\mathcal{H}_{d+1}}(\mathcal{R}(\Pi_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\})) \setminus \mathcal{R}(\{\text{cur}_{d+1}\}))\} \\
= \;& \{(d, R') : R' \in \text{ccs}_{\mathcal{H}_{d+1}}(\mathcal{R}(\{n\}))\} \\
& \cup \{(d, R'') : R'' \in \text{ccs}_{\mathcal{H}_{d+1}}(\mathcal{R}(\Psi^d_{\mathcal{H}_{d+1}}(\{n\})) \setminus \mathcal{R}(\{n\}))\} \\
= \;& \{(d, R') : R' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(\{n\}))\} \\
& \cup \{(d, R'') : R'' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(\Psi^d_{\mathcal{H}}(\{n\})) \setminus \mathcal{R}(\{n\}))\} \\
= \;& \text{splitAnc}_{\mathcal{H}}(d, \{n\})
\end{aligned}
$$

□

In this, the penultimate step follows from the observation that for every $d' < d + 1$, $V^{d'}_{\mathcal{H}_{d+1}} = V^{d'}_{\mathcal{H}}$, i.e. that only nodes at depth $d + 1$ or greater have changed by the time we get to hierarchy $\mathcal{H}_{d+1}$ in the unzip process.

Now, recall from the main paper that we use N^C_d to refer to the nodes in a set of chains C that are at depth d. As a result of Lemma 7, we can now assert the following:

Corollary 8. *If C is the set of chains resulting from a single-node unzipping of a node n to a depth* $d_{\min}$ *in hierarchy* $\mathcal{H}$*, then:*

$$\forall d \in [d_{\min}, \mathcal{D}(n)) \,.\, (N^C_d = \text{splitAnc}_{\mathcal{H}}(d, \{n\})).$$

Proof. If C is the set of chains resulting from the unzip, then:

$$C = \text{chains}_{d_{\min}}$$

By recursive application of Lemma 7, the nodes contained in $\text{chains}_{d_{\min}}$ (and thus C) are precisely:

$$\bigcup \{\text{splitAnc}_{\mathcal{H}}(d, \{n\}) : d_{\min} \leq d < \mathcal{D}(n)\}$$

Since each $\texttt{splitAnc}_{\mathcal{H}}(d, \{n\})$ in this only yields nodes at depth d, the result follows straightforwardly. □

Having shown that single-node unzipping yields chains of this form, it only remains to show that zipping these chains back together gives us back the original hierarchy:

Lemma 9. *Merging the results of splitting an ancestor of a node n about n yields a set containing that ancestor. Formally:*

$$\text{merge}(\text{splitAnc}_{\mathcal{H}}(d, \{n\})) = \{\psi^d_{\mathcal{H}}(n)\}$$

Proof. Looking at the definition of `splitAnc`, we argue that:

$$
\begin{aligned}
& \mathcal{R}(\text{splitAnc}_{\mathcal{H}}(d, \{n\})) \\
= \;& \bigcup(\text{ccs}_{\mathcal{H}}(\mathcal{R}(\{n\}))) \cup \bigcup(\text{ccs}_{\mathcal{H}}(\mathcal{R}(\Psi^d_{\mathcal{H}}(\{n\})) \setminus \mathcal{R}(\{n\}))) \\
= \;& \mathcal{R}(n) \cup (\mathcal{R}(\psi^d_{\mathcal{H}}(n)) \setminus \mathcal{R}(n)) \\
= \;& \mathcal{R}(\psi^d_{\mathcal{H}}(n))
\end{aligned}
$$

Then:

$$\begin{aligned} & \text{merge}(\text{splitAnc}_{\mathcal{H}}(d, \{n\})) \\ = \; & \{(d, \mathcal{R}(\text{splitAnc}_{\mathcal{H}}(d, \{n\})))\} \\ = \; & \{(d, \mathcal{R}(\psi^d_{\mathcal{H}}(n)))\} \\ = \; & \{\psi^d_{\mathcal{H}}(n)\} \end{aligned}$$

□

Theorem 10. *Zipping together the chains resulting from a single-node unzip reverts that unzip.*

Proof. By definition, chain zipping replaces N^C_d with $\text{merge}(N^C_d)$, for each d such that $\mathcal{D}_{\min}(N^C) \leq d \leq \mathcal{D}_{\max}(N^C)$. In the case of the chains returned by unzipping a node n to a depth $d_{\min}$ in a hierarchy $\mathcal{H}$, $\mathcal{D}_{\min}(N^C) = d_{\min}$ and $\mathcal{D}_{\max}(N^C) = \mathcal{D}(n) - 1$. Given Lemma 9, zipping those chains back together replaces $\texttt{splitAnc}_{\mathcal{H}}(d, \{n\})$ with $\psi^d_{\mathcal{H}}(n)$ at each d such that $d_{\min} \leq d < \mathcal{D}(n)$. This reverts the earlier unzip. □

4.3. Our Higher-Level Algorithms are Minimally Destructive

In order to make hierarchy editing more predictable for users, it is helpful to crystallise our understanding of which parts of a hierarchy can potentially be changed by the algorithms we propose, and which parts will necessarily stay the same. In this section, we examine in particular our non-sibling node merging and parent switching algorithms, and show that despite being non-local hierarchy operations, their effects are nonetheless confined to well-defined parts of the hierarchy. As a by-product of this analysis, we also show that our algorithms are *minimally destructive*: every hierarchy node that is modified by either a non-sibling node merge or a parent switch would need to be modified by *any* operations performing the same tasks.

4.3.1. Parent Switching

Consider a parent switch operation on a hierarchy $\mathcal{H}$ that switches the parent of a node n to a new parent p. As per the main paper, this will first unzip n to a depth of $d_{\min} = \mathcal{D}(\psi^+_{\mathcal{H}}(\{n, p\})) + 1$, just below the lowest common ancestor of n and p in $\mathcal{H}$, and then zip the resulting chain leading down to the parent of n to the chain leading down to p to complete the switch. The set of nodes N_m in $\mathcal{H}$ that will be modified by this operation can be specified as:

$$\begin{aligned} N_u &= \{\psi^d_{\mathcal{H}}(n) : d_{\min} \leq d \leq \mathcal{D}(p), \mathcal{R}(\psi^d_{\mathcal{H}}(n)) \supset \mathcal{R}(n)\} \\ N_z &= \{\psi^d_{\mathcal{H}}(p) : d_{\min} \leq d \leq \mathcal{D}(p)\} \\ N_m &= N_u \cup N_z \end{aligned}$$

For example, the nodes that would be modified by an example parent switching operation are shown in Figure 4(a), in which the parent of the green leaf is being switched to the node circled in yellow, nodes in N_u are shown in red and nodes in N_z are shown in orange. To show that parent switching is minimally destructive, first observe that making n a child of p in a modified hierarchy $\mathcal{H}'$ necessarily involves ensuring that:

1. The region of p and every other ancestor of n in $\mathcal{H}'$ contains the region of n.
2. No node in $\mathcal{H}'$ other than n or one of its ancestors has a region that contains the region of n.

To achieve this, we need to at the very least:

1. Remove $\mathcal{R}(n)$ from the region of any current ancestor of n in $\mathcal{H}$ that will still exist after the switch and no longer be an ancestor of n. The set of such nodes is precisely N_u, the set consisting of every ancestor of n below the lowest common ancestor of n and p in $\mathcal{H}$ whose region strictly contains that of n. Every such node will still exist after the switch (since it currently represents more than just $\mathcal{R}(n)$), but will no longer be an ancestor of n, and must therefore be modified by any operation effecting the switch. We note that existing ancestors of n at or above the lowest common ancestor of n and p are excluded from N_u, since they will still be ancestors of n after the switch.
2. Add $\mathcal{R}(n)$ to the region of every node that will be an ancestor of n after the parent switch and was not an ancestor before. The set of such nodes is precisely N_z, the set consisting of p and all of its ancestors below the lowest common ancestor of n and p in $\mathcal{H}$, since the lowest common ancestor and all ancestors of p above it are already ancestors of n in $\mathcal{H}$. Thus all nodes in N_z must inevitably be modified by any operation effecting the switch.

Since our parent switching algorithm therefore only modifies hierarchy nodes that must be modified by any algorithm performing the same task, it is minimally destructive as claimed.

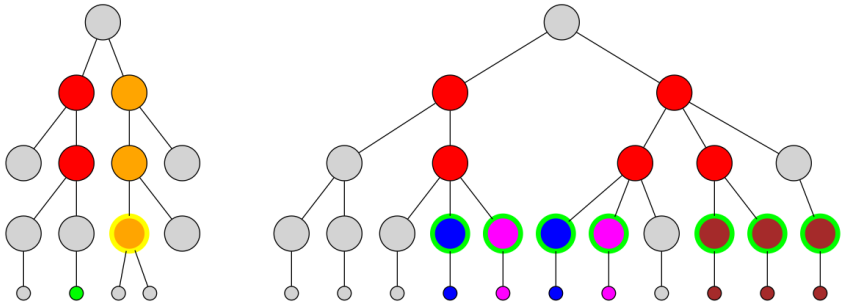
4.3.2. Non-Sibling Node Merging

A similar, though somewhat more complicated, argument can be made for non-sibling node merging. Consider a non-sibling merge of a set of nodes N in a hierarchy $\mathcal{H}$. As per the main paper, the nodes in N will first be divided into their connected components $\mathcal{K} = \{K_1, \ldots, K_k\}$. Then, the nodes in each component K_i will be unzipped to a depth of $d^i_{\min} = \mathcal{D}(a_i) + 1$, just below their lowest common ancestor $a_i = \psi^+_{\mathcal{H}}(K_i)$. Finally, the relevant chains for each component will be zipped together to complete the merge. The set of nodes N_m in $\mathcal{H}$ that will be modified by this operation can be specified as follows:

$$\begin{aligned} N^i_m &= \{\psi^d_{\mathcal{H}}(n) : n \in K_i, d^i_{\min} \leq d < \mathcal{D}(n), \mathcal{R}(\psi^d_{\mathcal{H}}(n)) \supset \mathcal{R}(n)\} \\ N_m &= \bigcup \{N^i_m : 1 \leq i \leq k\} \end{aligned}$$

For example, the nodes that would be modified by an example non-sibling node merging operation are shown in Figure 4(b), in which the set of nodes N being merged are circled in green, the nodes in N_m are shown in red, and the other colours indicate connected components of N.

To show that non-sibling node merging is minimally destructive, first observe that to make a new hierarchy $\mathcal{H}'$ in which the nodes in each K_i are merged, any operation will necessarily need to create in $\mathcal{H}'$, for each K_i, a new node m_i at depth $\mathcal{D}(N)$ whose region is $\mathcal{R}(K_i)$. Moreover, due to the nature of parent-child relationships in a segmentation hierarchy, it will also need to ensure that the region of each ancestor $\psi^d_{\mathcal{H}'}(m_i)$ of m_i is such that $\mathcal{R}(\psi^d_{\mathcal{H}'}(m_i)) \supseteq \mathcal{R}(K_i)$.



(a) (b)

Figure 4: Examples of (a) parent switching and (b) non-sibling node merging, showing the nodes that will be modified in each case. In (a), the parent of the green leaf is being switched to the node circled in yellow, nodes in N_u are shown in red and nodes in N_z are shown in orange (see text for details). In (b), the set of nodes N being merged are circled in green, the nodes that will be modified are shown in red, and the other colours indicate connected components of N.

For depths $d < d^i_{\min}$, i.e. depths at and above the depth of the lowest common ancestor a_i of the nodes in K_i, nothing needs to change: by definition, $\mathcal{R}(a_i) \supseteq \mathcal{R}(n)$ for every $n \in K_i$, and the region of every ancestor of a_i contains $\mathcal{R}(a_i)$. However, for each depth $d \geq d^i_{\min}$, i.e. each depth below that of a_i, any operation will necessarily have to create a new node whose region contains $\mathcal{R}(K_i)$, since by the definition of the lowest common ancestor of K_i, such nodes cannot exist in $\mathcal{H}$. To do this, it will necessarily have to modify the nodes specified in N^i_m, the (non-trivial) ancestors of the nodes in K_i at depths below that of a_i: it cannot avoid this, since otherwise there would be an overlap between the regions of the new nodes it is creating and the regions of the existing nodes at each depth.

Extending this argument over all of the components in $\mathcal{K}$, any operation that wants to perform this non-sibling node merge would therefore have to modify at least the nodes in N_m. Since our algorithm modifies exactly the nodes in N_m, it is minimally destructive as claimed.

5. Complexity Analysis

We first analyse the computational complexity of our algorithms in terms of the numbers of atomic split and merge operations involved in each case. By denoting the cost of an arbitrary parent-splitting operation as C_s and the cost of an arbitrary sibling node merge as C_m, this gives us a simple measure of complexity that is agnostic to the particular choice we make regarding the storage of higher-level edges in the hierarchy. However, as mentioned in Section 3, the storage or otherwise of higher-level edges can make a significant difference to the efficiency of our algorithms in practice. We thus also analyse the complexity of individual splits and merges, on the assumption that we maintain the set of higher-level edges at each depth, as is done in *millipede*.

5.1. Main Algorithms

Single-Node Unzipping. Consider a single-node unzip that unzips a node n to depth $d_{\min}$. Referring back to the closed-form definition in the main paper, this effects a split of each ancestor of n at depths $d \in [d_{\min}, \mathcal{D}(n))$. Its worst-case complexity is thus:

$$O((\mathcal{D}(n) - d_{\min})C_s)$$

Multi-Node Unzipping. Consider a multi-node unzip that unzips nodes in N to depth $d_{\min}$. In the worst case, all of these nodes will be at the same depth, namely $\mathcal{D}_{\max}(N)$, and each node in N will need to be unzipped individually up to depth $d_{\min}$. The worst-case complexity is thus:

$$O(|N|(\mathcal{D}_{\max}(N) - d_{\min})C_s)$$

Chain Zipping. Consider a chain zip that zips together the nodes in a set of chains C. In the worst case, this will involve merging nodes at every depth from $\mathcal{D}_{\min}(N^C)$, the smallest depth of any node in one of the chains, to $\mathcal{D}_{\max}(N^C)$, the greatest depth of any node in one of the chains. The worst-case complexity is thus:

$$O((\mathcal{D}_{\max}(N^C) - \mathcal{D}_{\min}(N^C))C_m)$$

Non-Sibling Node Merging. Consider a non-sibling node merge of a set of nodes N (all of which are at a common depth $\mathcal{D}(N)$). As per the definition, this first divides the nodes in N into their connected components, and then unzips the nodes in each component up to just below their lowest common ancestor. In the worst case, the total cost of these unzips is as great as possible, which happens when the lowest common ancestors for all the components are at some common depth, which we will call $d_{\min}$. The total cost of the unzips involved in the non-sibling node merge is thus the cost of unzipping every node in N to depth $d_{\min}$, i.e.

$$O(|N|(\mathcal{D}(N) - d_{\min})C_s).$$

Having unzipped all of the nodes, non-sibling node merging then zips the nodes in each component together again. Since the zip for each component involves $O((\mathcal{D}(N) - d_{\min})C_m)$ work, the total cost of the zips involved in the non-sibling node merge is maximised when there are as many zips as possible. In the worst case, there can be $O(|N|)$ such components, so the total cost of the zips is thus

$$O(|N|(\mathcal{D}(N) - d_{\min})C_m).$$

Combining these two gives us the overall cost of the non-sibling node merge, namely

$$O(|N|(\mathcal{D}(N) - d_{\min})(C_s + C_m)).$$

Parent Switching. Consider a parent switch operation that switches the parent of a node n to a new parent p. As per the definition of parent switching, this involves unzipping n to depth $d_{\min} = \mathcal{D}(\psi^+(\{n, p\})) + 1$, and then zipping the resulting chain leading down to the parent of n to the chain leading down to p. There is thus potentially one split and one merge operation at each depth $d \in [d_{\min}, \mathcal{D}(n))$, leading to a complexity of:

$$O((\mathcal{D}(n) - d_{\min})(C_s + C_m))$$

5.2. Auxiliary Operations

Parent Splitting. Consider using the `splitPar` operation defined in the main paper to split a parent node p in a hierarchy $\mathcal{H}$ around a node set N containing some of its children. This involves first computing both the connected components of N, and the connected components of $N' = \Pi^{-1}_{\mathcal{H}}(p) \setminus N$, i.e. the remaining children of p. Formally, we define the overall set of connected components into which we want to split p as

$$\mathcal{K} = \text{nccs}_{\mathcal{H}}(N) \cup \text{nccs}_{\mathcal{H}}(N'),$$

where `nccs` computes the connected components of a set of nodes, as in the main paper. Assuming that we maintain a graph $G^d_{\mathcal{H}} = (V^d_{\mathcal{H}}, E^d_{\mathcal{H}})$ for each depth d in $\mathcal{H}$, the components in $\mathcal{K}$ can be computed in linear time using depth-first search. More specifically, let $E[X, \cap]$ be the set of edges in E that have both endpoints in X. Then the cost of computing $\mathcal{K}$ is

$$O\left(|N| + \left|E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cap]\right| + |N'| + \left|E^{\mathcal{D}(N)}_{\mathcal{H}}[N', \cap]\right|\right).$$

Once the components have been computed, p must be removed from $V^{\mathcal{D}(p)}_{\mathcal{H}}$ and replaced with a new node corresponding to each component in $\mathcal{K}$. Assuming $V^{\mathcal{D}(p)}_{\mathcal{H}}$ is represented as a hash set, the expected complexity of this is $O(|\mathcal{K}|)$. In the worst case, where every node in N and N' yields a component,

$$O(|\mathcal{K}|) = O(|\mathcal{N}| + |\mathcal{N}'|) = O(|\Pi^{-1}_{\mathcal{H}}(p)|).$$

We must also remove all of the edges connected to p from $E^{\mathcal{D}(p)}_{\mathcal{H}}$, and replace them with the results of propagating all edges connected to any child of p up to depth $\mathcal{D}(p)$. Let $E[X, \cup]$ be the set of edges in E that have at least one endpoint in X. Then the set of edges we must remove from $E^{\mathcal{D}(p)}_{\mathcal{H}}$ is $E^{\mathcal{D}(p)}_{\mathcal{H}}[\{p\}, \cup]$, and the set of edges that must be added is

$$\left\{\{\pi_{\mathcal{H}'}(n_1), \pi_{\mathcal{H}'}(n_2)\} : \{n_1, n_2\} \in E^{\mathcal{D}(N)}_{\mathcal{H}}[\Pi^{-1}_{\mathcal{H}}(p), \cup]\right\},$$

in which $\pi_{\mathcal{H}'}(n)$ denotes the parent of n in the modified hierarchy. Assuming $E^{\mathcal{D}(p)}_{\mathcal{H}}$ is represented as a hash set, the expected overall complexity of the edge updates is thus

$$O\left(\left|E^{\mathcal{D}(N)}_{\mathcal{H}}[\Pi^{-1}_{\mathcal{H}}(p), \cup]\right|\right).$$

Propagating the edges upwards is the dominant cost of the whole operation, so this is also the expected complexity of parent splitting as a whole.

Sibling Node Merging. Consider merging a set of sibling nodes N in a hierarchy $\mathcal{H}$. To do this, we must remove all the nodes in N from $V^{\mathcal{D}(N)}_{\mathcal{H}}$ and replace them with a single new node, m, representing the result of the merge. Assuming $V^{\mathcal{D}(N)}_{\mathcal{H}}$ is represented as a hash set, the expected complexity of this is $O(|N|)$.

We must also remove all of the edges in $E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cap]$, i.e. the edges between nodes in N, and replace every edge in $E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cup] \setminus E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cap]$, i.e. the edges joining nodes in N to nodes not in N, with a new edge connecting m to the endpoint of the edge that is not in N. Assuming $E^{\mathcal{D}(N)}_{\mathcal{H}}$ is represented as a hash set, the expected complexity of this is $O(|E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cup]|)$.

The edge updates are the dominant cost of the merge operation, so $O(|E^{\mathcal{D}(N)}_{\mathcal{H}}[N, \cup]|)$ is also the expected complexity of sibling node merging as a whole.

References

Golodetz, S., 2011. Zipping and Unzipping: The Use of Image Partition Forests in the Analysis of Abdominal CT Scans. Ph.D. thesis. University of Oxford.

Golodetz, S., Voiculescu, I., Cameron, S., 2009. Automatic Spine Identification in Abdominal CT Slices using Image Partition Forests, in: ISPA.

Stroustrup, B., 2013. The C++ Programming Language. Addison Wesley.

Listing 1 Single-Node Unzipping

```
function unzip_node(node: NodeID, toDepth: Int) ↦ Vector<Chain>
  // Initialise the array of chains with a singleton chain containing only the node being unzipped.
  var chains: Vector<Chain>;
  chains.push_back([node]);

  var cur: NodeID := node;
  while cur.depth() ≠ toDepth
    // Calculate the connected components of the current node's siblings.
    var ccs: Vector<Set<NodeID>> := find_connected_components(siblings_of(cur));

    // Add in the component {cur} and split the current node's parent.
    ccs.push_back({cur});
    var result: Set<NodeID> := split_node(parent_of(cur), 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.push_back([n]);

    // Update the current node.
    cur := parent_of(cur);

  chains.front().pop_back();
  return chains;
```

Listing 2 Multi-Node Unzipping

```
function unzip_selection(sel: Selection, toDepth: Int) ↦ 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>> := sel.group_nodes_by_depth();
  var depth: Int := nodesByDepth.max_key();
  var curs: Set<NodeID> := nodesByDepth[depth];
  while depth ≠ toDepth
    // Group the current nodes by parent.
    var parentToSelectedChildMap: Map<NodeID,Set<NodeID>>;
    for cur: NodeID ∈ curs
      parentToSelectedChildMap[parent_of(cur)].insert(cur);

    // Split each parent node in turn.
    var result: Set<NodeID>;
    for (parent, selectedChildren) ∈ parentToSelectedChildMap
      var siblings: Set<NodeID> := siblings_of(selectedChildren);
      var ccs: Vector<Set<NodeID>> := find_connected_components(siblings);
      ccs.append(find_connected_components(selectedChildren));
      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 3 Chain Zipping

```
function zip_chains(chains: Vector<Chain>)
  // Check that there are chains to zip.
  if(chains.empty()) then throw;

  // Check that each chain is non-empty and doesn't contain a leaf.
  var minChainSize: Int := ∞;
  for each chain: Chain ∈ chains
    if chain.empty() or is_leaf(chain.back()) then throw;
    minChainSize := min(minChainSize, chain.size());

  // Check that the highest nodes in the chains are siblings.
  var commonParent := parent_of(chains[0].front());
  for each chain: Chain ∈ chains[1..]
    if parent_of(chain.front()) ≠ commonParent then throw;

  // Compute the sets of nodes to be merged at each level and check that the nodes are connected.
  var nodes := Vector<Set<NodeID>>(minChainSize);
  for i := 0 up to minChainSize - 1
    for each chain ∈ chains
      if i < chain.size() then nodes[i].insert(chain[i]);
    if not are_connected(nodes[i]) then throw;

  // Merge the nodes at each level, starting from the highest.
  for i := 0 up to minChainSize - 1
    merge_sibling_nodes(nodes[i]);
```

Listing 4 Non-Sibling Node Merging

```
function merge_nonsibling_nodes(nodes: Set<NodeID>) ↦ Set<NodeID>
  // Check that there are nodes to be merged.
  if nodes.empty() then throw;

  // Check that none of the nodes to be merged is a leaf, and that they are all at the same depth.
  var commonDepth: Int := lowest_indexed_node(nodes).depth();
  for each n: NodeID ∈ nodes
    if is_leaf(n) or n.depth() ≠ commonDepth then throw;

  var mergedNodes: Set<NodeID>;

  // Calculate the connected components of the nodes to be merged.
  var ccs := find_connected_components(nodes);

  // Merge the nodes in each connected component of size > 1.
  for each cc: Set<NodeID> ∈ ccs
    if cc.size() = 1 then continue; // nothing to do

    // Find the depth to which the nodes need to be unzipped.
    var toDepth: Int := find_common_ancestor_depth(cc) + 1;

    // Unzip each node in the component to the specified depth, in each case keeping the chain corresponding
    // to the unzipped node itself, which (by construction) will be the first one in the returned vector.
    // Since we want the actual nodes (and not just the nodes above them in their chains) to be merged,
    // we add them to the ends of their respective chains here as well.
    var chains: Vector<Chain>;
    for each n: NodeID ∈ cc
      var unzipResult: Vector<Chain> := unzip_node(n, toDepth);
      var chain: Chain := unzipResult.front();
      chain.push_back(n);
      chains.push_back(chain);

    // Zip the chains together to effect the merge.
    mergedNodes.insert(zip_chains(chains));

  return mergedNodes;
```

Listing 5 Parent Switching

```
function parent_switch(node: NodeID, newParent: NodeID)
  // Check that the node is at a greater depth than its proposed new parent.
  if node.depth() ≤ newParent.depth() then throw;

  // Check that the node is adjacent to at least one child of its proposed new parent.
  var adjacent: Bool := false;
  for each c: NodeID ∈ children_of(newParent)
    if is_adjacent(node, c) then
      adjacent := true;
      break;
  if not adjacent then throw;

  // Find the common ancestor of the old and new parents, and the chain leading down to the new parent.
  var oldParent: NodeID := parent_of(node);
  var commonAncestor: NodeID;
  var newChain: Chain;
  (commonAncestor, newChain) := find_common_ancestor_and_new_chain(oldParent, newParent);

  // Unzip the node to the common ancestor.
  var unzipResult: Vector<Chain> := unzip_node(node, commonAncestor.depth() + 1);

  // Zip the chain leading down to the node being moved to the new chain to complete the parent switch.
  // Note that the old chain required is the first chain in the unzip result (by construction).
  zipChains([unzipResult.front(), newChain]);
```