Simpler Editing of Graph-Based Segmentation Hierarchies using Zipping Algorithms

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

Graph-based image segmentation is popular, because graphs can naturally represent image parts and the relationships between them. Whilst many single-scale approaches exist, significant interest has been shown in segmentation hierarchies, which represent image objects at different scales. However, segmenting arbitrary images automatically remains elusive: segmentation is under-specified, with different users expecting different outcomes. Hierarchical segmentation compounds this, since it is unclear where in the hierarchy objects should appear. Users can easily edit flat segmentations to influence the outcome, but editing hierarchical segmentations is harder: indeed, many existing interactive editing techniques make only small, local hierarchy changes. In this paper, we address this by introducing 'zipping' operations for segmentation hierarchies to facilitate user interaction. We use these operations to implement algorithms for non-sibling node merging and parent switching, and perform experiments on both 2D and 3D images to show that these latter algorithms can significantly reduce the interaction burden on the user.

Keywords: graph-based, segmentation hierarchy, user interaction

1. Introduction

Image segmentation is a core problem in computer vision, with applications that include scene understanding, object detection and 3D visualisation. Graph-

*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)

(a) (b) (c)

Figure 1: Segmentation is an under-specified problem with many possible solutions for each image: (a) shows a target scene to be segmented; (b) shows the result of segmenting it into 'workstation', 'gamepad', 'desk' and 'background' classes; (c) shows the result of segmenting it into 'monitor', 'tower', 'desk', 'speaker', 'blind', 'wall' and 'floor' classes. Without knowing what the user wants, there is no reason to prefer one segmentation over the other.

based approaches to image segmentation [40] have long been popular, because graphs can naturally represent image parts and the relationships between them. *Hierarchical* graph-based segmentation approaches (e.g. [22, 26, 46]) can additionally represent objects in an image at different scales and levels of abstraction. However, the goal of segmenting arbitrary images automatically remains elusive because segmentation is an under-specified problem: despite some consistency between human segmentations of the same image [33], different users expect different outcomes (see Figure 1). This problem is only compounded when performing segmentation hierarchically, since multi-scale segmentation often involves representing individual objects at different levels of abstraction, and it is unclear where in the hierarchy to represent different objects (or object parts).

To solve this problem, it is common to ask users to interact with the segmentation to influence the desired outcome and correct any mistakes. This can be effective when the segmentation is flat (i.e. has only one level), but it is significantly more difficult for hierarchical segmentation, because hierarchies can be hard to edit once constructed due to the need to maintain the property that every node in a hierarchy represents a contiguous region in the underlying image: indeed, many editing algorithms in the literature (see Section 2) are

restricted to making only small, local changes to a hierarchy. For this reason, hierarchical segmentation approaches can struggle to make effective use of user input, and in some contexts are viewed instead as an alternative to more interactive methods: for example, Beucher's waterfall algorithm [7] can be viewed as a more 'automatic' alternative to watershed-from-markers [36].

In this paper, we argue that this separation between hierarchical and interactive methods is undesirable: the user is a key source of domain-specific information, and hierarchical approaches can benefit from such information as much as, if not more than, flat ones. To facilitate this, we propose multi-scale split/merge algorithms for segmentation hierarchies that encapsulate the logic needed to preserve spatial connectivity and can be used as building blocks for higher-level editing – we call these *zipping algorithms*, because the sequences of splits/merges involved in unzipping/zipping respectively resemble the opening/closing of a zip. We use them to develop two higher-level techniques that make hierarchy editing easier for users. *Non-sibling node merging* allows users to merge nodes in the hierarchy whose image regions are spatially-adjacent but that do not share a common parent: this allows users to edit a hierarchy 'intuitively' by clicking on adjacent image regions and requesting a merge, rather than needing to think about the hierarchy structure. *Parent switching* allows users to move a child node between parents in the hierarchy: this generalises Nacken's relinking technique [37] and allows users to fix places in the hierarchy where nodes have been merged into the wrong parent during construction. We similarly expose this to users via a simple, mouse-based interface (see Section 6).

Our algorithms have been implemented as part of our *millipede* segmentation system [18] and its more recent extensions [34, 47]. We perform experiments on both 2D and 3D images to demonstrate their benefits, and show that they significantly reduce the interaction burden on the user.

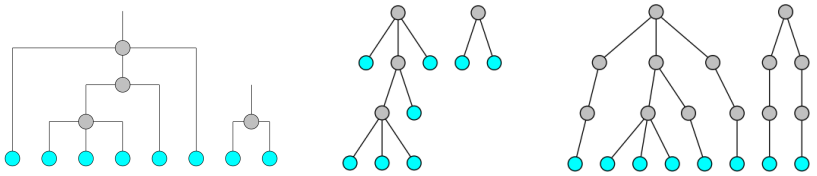
2. Related Work

Much existing research on editing graph-based segmentation hierarchies has focused on making small, local changes to a hierarchy. Auber's grouping operations [5], Eades' cluster creation and deletion operations [11] and Fisher's node merging and node splitting operations [13] all group a number of sibling nodes

under a new node in the hierarchy or ungroup them again. Other simple operations merge a set of sibling nodes and replace them with a new node, or split a single node into connected pieces [20]. Nacken's 'relinking' technique [37] allows users to change the parent of an existing node. Although local, when applied to graph-based segmentation hierarchies this operation requires non-local checking to determine whether the planned change of parent is valid.

Non-local techniques change hierarchies more significantly, often with a goal in mind. Glantz and Kropatsch [17] use multiple relinking operations to obtain a specific pattern in the lowest hierarchy layer. *GrouseFlocks* [2] visualises large graphs by interactively creating hierarchies above them and using non-horizontal cuts [23] to hide the contents of most nodes and avoid visual clutter. Klava's split and merge operations [30] work directly on cuts, without modifying the hierarchy. *TugGraph* [3] modifies a hierarchy to 'separate out' the parts of the base graph that are spatially-adjacent to a selected cut node, to help users explore the topological structure of a network. Several non-local editing operations have arisen from mathematical morphology [38]. Morphological flooding is a classical technique used to fill in catchment basins in an image, e.g. as a preprocessing step to reduce oversegmentation by the watershed transform. Cousty and Najman [9] show that a flooding on a hierarchy is equivalent to pruning the hierarchy based on a non-horizontal cut. Such cuts can be specified interactively [2, 30], or automatically, based on e.g. the Number of False Alarms [8] or Mumford-Shah energies [23, 29]. However, unlike our operations, they can only remove regions from an existing hierarchy, not create new ones.

Gerstmayer et al. [16] describe two editing operations: one marks pairs of regions that should be merged; the other marks individual regions that should be inhibited from merging. Unlike our algorithms, these operations require the recreation of the hierarchy above the marked nodes. However, inhibiting a region from being merged in higher layers of the hierarchy has a similar effect to the unzipping techniques we describe in Section 4. Finally, Maire et al. [32] present an interesting system for interactively editing hierarchies that can model both object-part relationships and local occlusions. These hierarchies differ from the ones we consider in that they allow sibling regions to overlap, but their work nevertheless reflects a growing interest in hierarchy editing in the field.



(a) (b) (c)

Figure 2: A graph-based segmentation hierarchy results from recursively grouping together the nodes of its base graph (drawn in cyan). Its hierarchical structure can be depicted in different ways, e.g. (a) and (b) visualise the same hierarchy as (respectively) a dendogram or an incomplete forest. A hierarchy whose hierarchical structure is a complete forest (c) is a special case that we refer to as a hierarchy of partitions, graph pyramid or partition forest.

3. Graph-Based Segmentation Hierarchies

A *graph-based segmentation hierarchy* $\mathcal{H}$ results from recursively clustering the nodes of an undirected graph $G_{\mathcal{H}} = (V_{\mathcal{H}}, E_{\mathcal{H}})$ that we call the *base graph* of $\mathcal{H}$. In this, $V_{\mathcal{H}}$ is the set of nodes in $G_{\mathcal{H}}$, and $E_{\mathcal{H}}$ is the set of edges. The base graph partitions the entity (e.g. an image) we are segmenting: each node in $V_{\mathcal{H}}$ represents part of the entity, no two represented parts overlap, and together the nodes in $V_{\mathcal{H}}$ represent the whole entity. For example, when segmenting an image, the base graph will typically contain a node for each pixel in the image.

Such a hierarchy can be depicted either as a dendogram (see Figure 2(a)) or as a forest whose lower layers may or may not be complete (see Figure 2(b)). Segmentation hierarchies whose forests only have complete layers (see Figure 2(c)) are an interesting special case, since any complete layer necessarily partitions the base graph, and are often given a distinctive name, e.g. hierarchy of partitions [25, 31], graph pyramid [28] or partition forest [18, 20, 21].

Hierarchically clustering the base graph nodes $V_{\mathcal{H}}$ of a hierarchy $\mathcal{H}$ creates a set $V^*_{\mathcal{H}} \supseteq V_{\mathcal{H}}$ of higher-level nodes in $\mathcal{H}$, each representing the aggregation of its descendants in $V_{\mathcal{H}}$. Each node $n \in V^*_{\mathcal{H}}$ can be represented as a pair $(\mathcal{D}(n), \mathcal{R}(n))$, in which $\mathcal{D}(n)$ is the *depth* of n. A node's depth is defined to be 0 for the root node, and one plus the depth of its parent otherwise. We call $\mathcal{R}(n) \subseteq V_{\mathcal{H}}$ the *region* of n, defined to be the set of n's descendants in $V_{\mathcal{H}}$. We enforce the restriction that the subgraph of $G_{\mathcal{H}}$ denoted by the region of each node must be *connected*, and it is this restriction that complicates editing, since this property must be maintained across all editing operations.

Ancestors/Descendants Set (Specified Relative Depth)	
Single Node	Node Set
$\Pi^i_{\mathcal{H}}(n) = \{n' \in V^{\mathcal{D}(n)-i}_{\mathcal{H}} : \mathcal{R}(n) \cap \mathcal{R}(n') \neq \emptyset\}$	$\Pi^i_{\mathcal{H}}(N) = \bigcup\{\Pi^i_{\mathcal{H}}(n) : n \in N\}$
Ancestors/Descendants Set (Specified Absolute Depth)	
Single Node	Node Set
$\Psi^d_{\mathcal{H}}(n) = \Pi^{\mathcal{D}(n)-d}_{\mathcal{H}}(n)$	$\Psi^d_{\mathcal{H}}(N) = \bigcup\{\Psi^d_{\mathcal{H}}(n) : n \in N\}$
Single / Common Ancestor (if one exists)	
Specified Relative Depth	Specified Absolute Depth
$\pi^i_{\mathcal{H}}(n) = n' \Leftrightarrow \Pi^i_{\mathcal{H}}(n) = \{n'\}$ and $i \geq 0$	$\psi^d_{\mathcal{H}}(n) = \pi^{\mathcal{D}(n)-d}_{\mathcal{H}}(n)$
$\pi^i_{\mathcal{H}}(N) = n' \Leftrightarrow \Pi^i_{\mathcal{H}}(N) = \{n'\}$ and $i \geq 0$	$\psi^d_{\mathcal{H}}(N) = \pi^{\mathcal{D}(N)-d}_{\mathcal{H}}(N)$
All Descendants (Any Depth)	$\Psi^-_{\mathcal{H}}(n)$
Lowest Common Ancestor	$\psi^+_{\mathcal{H}}(N)$

Table 1: Our notation to denote parent/child relationships between nodes in a hierarchy $\mathcal{H}$. In this, $V^d_{\mathcal{H}}$ denotes all nodes at depth d in $\mathcal{H}$, n denotes a single node, and N denotes a set of nodes. We overload the functions to accept single nodes or node sets for notational simplicity.

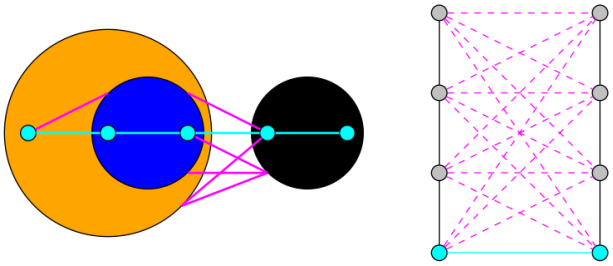
We extend our depth/region concepts to node sets as well as individual nodes to simplify later definitions. For a node set N, we let $\mathcal{D}_{\max}(N)$ and $\mathcal{D}_{\min}(N)$ be (respectively) the maximum and minimum depths of any node in N. Iff $\mathcal{D}_{\max}(N) = \mathcal{D}_{\min}(N)$, we define $\mathcal{D}(N)$ to be the common depth of the nodes in N. We further define $\mathcal{R}(N)$ to be the union of the regions of the nodes in N.

To express the parent/child relationships that exist between the nodes in a hierarchy $\mathcal{H}$, we first let $V^d_{\mathcal{H}} \subseteq V^*_{\mathcal{H}}$ denote the set of all nodes at depth d in $\mathcal{H}$. We then define functions that allow us to refer to the ancestor/descendants of a node or set of nodes at either an absolute depth, or a depth relative to the input node(s), as shown in Table 1. We also define functions to refer to all the descendants of a node and the lowest common ancestor of a set of nodes.

In addition to higher-level nodes, a hierarchy $\mathcal{H}$ can also be seen as containing higher-level edges $E^*_{\mathcal{H}}$ between any two nodes in $V^*_{\mathcal{H}}$ that have descendants joined by an edge in the base graph $G_{\mathcal{H}}$ (see Figure 3(a)). Formally:

$$E^*_{\mathcal{H}} = \{\{\psi^{d_1}_{\mathcal{H}}(n_1), \psi^{d_2}_{\mathcal{H}}(n_2)\} : \{n_1, n_2\} \in E_{\mathcal{H}} \text{ and } \forall i \,.\, 0 \leq d_i \leq \mathcal{D}(n_i)\} \quad (1)$$

Since, in the worst case, both $\mathcal{D}(n_1)$ and $\mathcal{D}(n_2)$ can equal $\texttt{height}(\mathcal{H})$, the maximum depth of any node in $\mathcal{H}$, each (base) edge in $E_{\mathcal{H}}$ can be responsible for



(a) (b)

Figure 3: A hierarchy $\mathcal{H}$ can be seen as containing higher-level edges $E^*_{\mathcal{H}}$ between any two nodes in $V^*_{\mathcal{H}}$ that have descendants joined by an edge in the base graph $G_{\mathcal{H}}$: (a) shows all of the edges present in an example hierarchy; (b) shows how each base graph edge can be responsible for a number of higher-level edges that is quadratic in the height of $\mathcal{H}$. In both cases, the base graph edges in $E_{\mathcal{H}}$ are shown in cyan, and the set $E^*_{\mathcal{H}}$ of higher-level edges contains both the edges in $E_{\mathcal{H}}$ and the higher-level magenta edges they induce.

a number of higher-level edges that is quadratic in the height of $\mathcal{H}$ (see also Figure 3(b)). How to represent this connectivity information is thus crucial, e.g. we can store all of the edges, store just the base edges and recalculate the higher-level edges as necessary, or store just the edges between nodes of the same depth. The trade-off between the storage and processing costs involved in each case is application-specific. The way in which our zipping algorithms are defined (see Section 4) makes them agnostic to this choice: they decide which split/merge operations to perform on a hierarchy, but do not concern themselves with edge maintenance. However, the choice does matter when analysing the computational complexity of the algorithms (see the supplementary material).

4. Zipping Algorithms

A key difficulty when editing segmentation hierarchies is the need to preserve hierarchy node connectivity (see Section 3): we cannot arbitrarily merge adjacent nodes or change their parents and expect the resulting hierarchy to be valid. However, we *can* encapsulate the connectivity preservation details in intermediate-level operations, freeing higher-level techniques like *non-sibling node merging* from dealing with them explicitly. In this section, we present two such operations: *unzipping* effects multi-level splits in the hierarchy; *chain zipping* effects a multi-level merge. We can expose the higher-level techniques we build upon them via an intuitive, mouse-based user interface (see Section 6),

Parent-Splitting Operations	
Children of 1 Parent	$\text{splitPar}_{\mathcal{H}}(N) = \{(\mathcal{D}(N)-1, R') : R' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(N))\}$ $\cup \{(\mathcal{D}(N)-1, R'') : R'' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(\pi_{\mathcal{H}}(N)) \setminus \mathcal{R}(N))\}$
Children of ≥ 1 Parent(s)	$\text{splitPars}_{\mathcal{H}}(N) = \bigcup\{\text{splitPar}_{\mathcal{H}}(\Pi_{\mathcal{H}}^{-1}(n) \cap N) : n \in \Pi_{\mathcal{H}}(N)\}$
Ancestor-Splitting Operations	
Descendants of 1 Ancestor	$\text{splitAnc}_{\mathcal{H}}(d, N) = \{(d, R') : R' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(N))\}$ $\cup \left\{(d, R'') : R'' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(\psi_{\mathcal{H}}^{d}(N)) \setminus \mathcal{R}(N))\right\}$
Descendants of ≥ 1 Ancestor(s)	$\text{splitAncs}_{\mathcal{H}}(d, N) = \bigcup \left\{\text{splitAnc}_{\mathcal{H}}(d, \Psi_{\mathcal{H}}^{-}(n) \cap N) : n \in \Psi_{\mathcal{H}}^{d}(N)\right\}$

Table 2: The operations we use to split the parent(s) or ancestor(s) of a set of nodes N in a hierarchy $\mathcal{H}$ around those nodes. In this, $\text{ccs}_{\mathcal{H}}(R)$ denotes the connected components of R in $G_{\mathcal{H}}$, the base graph of $\mathcal{H}$. Informally, the operations split the relevant parent(s) or ancestor(s) into nodes corresponding to the connected components of the nodes in N, and nodes corresponding to the connected components of what is left of the parent(s) or ancestor(s) after removing their children or descendants in N. See also Figure 4.

allowing the user to edit a hierarchy without explicit reference to its structure.

4.1. Unzipping

In its simplest form, unzipping is a technique that 'separates out' an individual node from some or all of the nodes above it in the hierarchy. Conceptually, this is achieved by performing a sequence of node splitting operations, starting at the node's parent and working upwards until a specified depth is reached. Each split divides one of the node's ancestors into pieces, one of which is a singleton piece containing the node being unzipped, and the others of which are the connected components of what is left of the ancestor after removing the node being unzipped from consideration. A more general algorithm that unzips multiple nodes simultaneously is also possible. In this form, we consider splitting the ancestors of a set of nodes (whose regions must not overlap) into pieces, some of which are the connected components of the nodes being unzipped, and the others of which are the connected components of what is left of the ancestors after removing the nodes being unzipped from consideration.

Formally, we define two operations for splitting parent nodes around some of their children, and corresponding operations for splitting ancestors around some of their descendants (see Table 2). The most basic operation, `splitPar`, determines the results of splitting a single parent node around some of its children: this yields a set of nodes (at the parent's depth) corresponding to the

Figure 4: An illustration of `splitPar`, the operation we use to determine how to split the parent of a set of nodes N around those nodes. From left to right: the parent node being split; the children of the node; the child nodes N around which to split the parent (highlighted in red) and their remaining siblings (in blue); the nodes resulting from the split.

connected components of the selected children and the connected components of what is left of the parent node after removing those children (see Figure 4). A more general operation, `splitPars`, determines the results of splitting multiple parents around their respective children simultaneously. As shown in Table 2, we can also define equivalent operations to split ancestor nodes around their respective descendants (which we call `splitAnc` and `splitAncs`).

4.1.1. Single-Node Unzipping

The single-node unzipping algorithm takes as input a node n to be unzipped and a depth $d_{\min}$ to which to unzip it, and returns an array of chains that can be zipped together in order to undo the operation (as proved in the supplementary material). A *chain* is a sequence of nodes $[n_1, ..., n_k]$, where each n_i is the parent of n_{i+1}. The chains returned by unzipping may differ in length, but their highest nodes are all at the same depth in the hierarchy.

Formally, we can define the operation either in closed form, which is helpful for proofs, or as a depth-based (tail) recursion that models the way in which the algorithm would actually be implemented (see Table 3). As shown by the closed form definition, the essence of the algorithm is to replace each ancestor of n at depths between $d_{\min}$ and $\mathcal{D}(n) - 1$ with nodes generated by splitting the ancestor around n, as described in the previous section. The tail-recursive definition bridges the gap between the closed form definition and the actual implementation of the algorithm (see the supplementary material). It models a loop over a depth d from $\mathcal{D}(n) - 1$ down to $d_{\min}$, maintaining at each iteration

Single-Node Unzipping (no chains, closed form)

$$\text{unzip}^{\mathcal{S}^*}_{n;d_{\min}}(\mathcal{H}) = \left(\text{unzip}^{\mathcal{S}^*}_{n;d_{\min}}(V^d_{\mathcal{H}}) : V^d_{\mathcal{H}} \in \mathcal{H}\right)$$

$$\text{unzip}^{\mathcal{S}^*}_{n;d_{\min}}(V^d_{\mathcal{H}}) = \begin{cases} (V^d_{\mathcal{H}} \setminus \{\psi^d_{\mathcal{H}}(n)\}) \cup \text{splitAnc}_{\mathcal{H}}(d, \{n\}) & \text{if } d \in [d_{\min}, \mathcal{D}(n)) \\ V^d_{\mathcal{H}} & \text{otherwise} \end{cases}$$

Single-Node Unzipping (with chains, recursively-expressed down loop)

Hierarchy Transformation Sequence: $\mathcal{H}_{\mathcal{D}(n)}(= \mathcal{H}) \rightsquigarrow \mathcal{H}_{(\mathcal{D}(n)-1)} \rightsquigarrow \ldots \rightsquigarrow \mathcal{H}_{d_{\min}}$

Initialisation: $\mathcal{H}_{\mathcal{D}(n)} = \mathcal{H} \quad \text{cur}_{\mathcal{D}(n)} = n \quad \text{chains}_{\mathcal{D}(n)} = \emptyset$

Iterative Updates $(\forall d \in [d_{\min}, \mathcal{D}(n))$:

$$V^i_{\mathcal{H}_d} = \begin{cases} (V^d_{\mathcal{H}_{d+1}} \setminus \{\pi_{\mathcal{H}_{d+1}}(\text{cur}_{d+1})\}) \cup \text{splitPar}_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\}) & \text{if } i = d \\ V^i_{\mathcal{H}_{d+1}} & \text{otherwise} \end{cases}$$

$$\text{cur}_d = \pi_{\mathcal{H}_d}(\text{cur}_{d+1})$$

$$\text{chains}_d = \{[\pi_{\mathcal{H}_d}(x)] +\!\!+ [x] +\!\!+ xs : [x] +\!\!+ xs \in \text{chains}_{d+1})\}$$
$$\cup \{[p] : p \in \text{splitPar}_{\mathcal{H}_{d+1}}(\{\text{cur}_{d+1}\}) \setminus \{\pi_{\mathcal{H}_d}(x) : [x] +\!\!+ xs \in \text{chains}_{d+1}\}\}$$

Final Result: $\text{unzip}^{\mathcal{S}}_{n;d_{\min}}(\mathcal{H}) = (\mathcal{H}_{d_{\min}}, \text{chains}_{d_{\min}}) \equiv (\text{unzip}^{\mathcal{S}^*}_{n;d_{\min}}(\mathcal{H}), \text{chains}_{d_{\min}})$

Table 3: Two analogous definitions of single-node unzipping. The closed form definition is helpful for proofs, whilst the depth-based (tail) recursion models how the algorithm would be implemented as a down loop from $\mathcal{D}(n) - 1$ to $d_{\min}$. Note that $+\!\!+$ concatenates two chains.

the current state of the hierarchy, $\mathcal{H}_d$, initially $\mathcal{H}$ itself, a 'current' node, cur_d, initially n, and the current set of chains we are constructing, chains_d, initially empty. In loop iteration d, we replace the parent of cur_{d+1} with the nodes generated by splitting it around cur_{d+1}, and then set cur_d to the parent of cur_{d+1} in the modified hierarchy $\mathcal{H}_d$. Subsequently, each existing chain is prepended with the parent of its head node, and a new chain is added for each node other than one of these parents that was generated by the split. The result of the algorithm is $(\mathcal{H}_{d_{\min}}, \text{chains}_{d_{\min}})$, the state of the hierarchy and chains after the last loop iteration. An example is shown in Figure 5.

Whilst it is possible to use single-node unzipping repeatedly to unzip a set of nodes one at a time, it can mean splitting the same ancestor nodes multiple times, forcing us to waste time by repeatedly finding similar connected components. Moreover, it separates out the nodes from the rest of the hierarchy

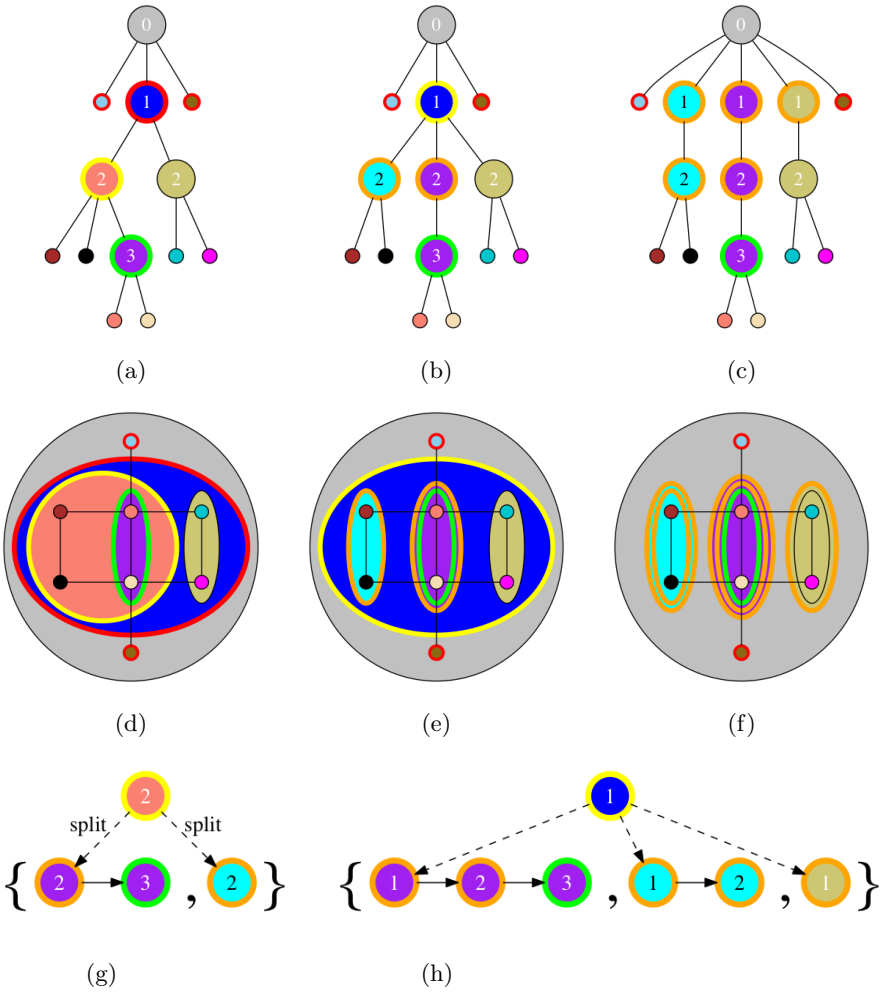
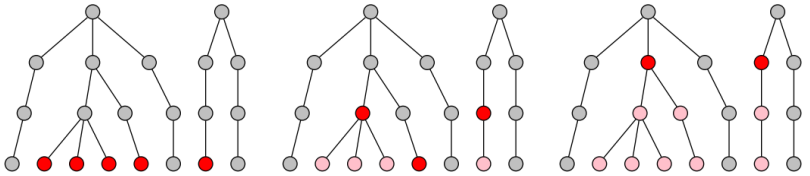


Figure 5: Using the single-node unzipping algorithm to unzip the node circled in green to the depth of the nodes circled in red (the numbers on the branch nodes denote their depth): in (a) and (d), the parent node (circled in yellow) of the node being unzipped is considered for splitting; in (b) and (e), the parent is split into two nodes, one corresponding to the node being unzipped and the other corresponding to the connected component formed by the node's siblings, and the grandparent of the node being unzipped is considered for splitting; in (c) and (f), the grandparent is split into three nodes, one corresponding to the node being unzipped and the other two corresponding to the connected components of the siblings. The chains at each stage of the unzip are circled in orange; (g) and (h) show the chain updates resulting from each split (the curly brackets denote a set of chains).

individually. To separate several connected nodes into a single chain, we have to unzip each node individually and then zip their chains together again. It is thus helpful to extend single-node unzipping to allow us to unzip multiple nodes at the same time (we ensure that none of the nodes is a descendant of another).



(a) (b) (c)

Figure 6: A hierarchical selection represents a set of (leaf) nodes in the base graph of a hierarchy using as few nodes as possible (preferring those that are higher up in the hierarchy when there is a choice). Here, (a)–(c) show three ways of representing the red nodes in (a): 5 nodes are needed for an explicit representation, but where leaves share common ancestors, fewer nodes can be stored (b, c). A hierarchical selection would represent the nodes as in (c), storing only the 2 nodes marked in red (the minimum number possible) and implicitly selecting their descendants. Throughout, red is used to denote nodes that are explicitly stored, whilst pink is used to denote nodes that are implicitly selected because of their ancestors.

4.1.2. Multi-Node Unzipping

Our multi-node unzipping algorithm takes a set of nodes N, represented as a *hierarchical selection* [18, 21], and a depth $d_{\min}$ to which to unzip them. A hierarchical selection (see Figure 6) is a data structure that represents a set of nodes in a hierarchy's base graph using as few hierarchy nodes as possible (preferring those higher up in the hierarchy when there is a choice). Selecting a node implicitly selects all of its descendants, so maintaining a minimal set of nodes in the representation ensures that no node in the representation is a descendant of another. See the supplementary material for further details.

As with single-node unzipping, we can define the operation either in closed form or as a depth-based (tail) recursion (see Table 4). As shown by the closed form definition, the algorithm replaces the ancestors of the nodes in N at each depth between $d_{\min}$ and $\mathcal{D}_{\max}(N) - 1$ with nodes generated by splitting those ancestors around their respective descendants in N (as described in Section 4.1). The tail-recursive definition models a loop over a depth d from $\mathcal{D}_{\max}(N) - 1$ down to $d_{\min}$, maintaining at each iteration the current state of the hierarchy, $\mathcal{H}_d$, initially $\mathcal{H}$ itself, a set of current nodes, curs_d, initially $N_{\mathcal{D}_{\max}(N)}$, the nodes of maximum depth in N, and the current set of chains we are constructing, chains_d, initially empty. In loop iteration d, we replace the parents of the nodes in curs_{d+1} with the nodes generated by splitting them around their respective children in curs_{d+1}. We then set curs_d to the union of the parents of the nodes

Multi-Node Unzipping (no chains, closed form)

Auxiliary Definitions: $N_{>d} = \{n \in N : \mathcal{D}(n) > d\}$

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

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

Multi-Node Unzipping (with chains, recursively-expressed down loop)

Hierarchy Transformation Sequence: $\mathcal{H}_{\mathcal{D}_{\max}(N)}(= \mathcal{H}) \rightsquigarrow \mathcal{H}_{(\mathcal{D}_{\max}(N)-1)} \rightsquigarrow \ldots \rightsquigarrow \mathcal{H}_{d_{\min}}$

Auxiliary Definitions: $N_d = \{n \in N : \mathcal{D}(n) = d\}$

Initialisation: $\mathcal{H}_{\mathcal{D}_{\max}(N)} = \mathcal{H} \quad \text{curs}_{\mathcal{D}_{\max}(N)} = N_{\mathcal{D}_{\max}(N)} \quad \text{chains}_{\mathcal{D}_{\max}(N)} = \emptyset$

Iterative Updates $(\forall d \in [d_{\min}, \mathcal{D}_{\max}(N))):$

$$V^i_{\mathcal{H}_d} = \begin{cases} (V^d_{\mathcal{H}_{d+1}} \setminus \Pi_{\mathcal{H}_{d+1}}(\text{curs}_{d+1})) \cup \text{splitPars}_{\mathcal{H}_{d+1}}(\text{curs}_{d+1}) & \text{if } i = d \\ V^i_{\mathcal{H}_{d+1}} & \text{otherwise} \end{cases}$$

$$\text{curs}_d = \Pi_{\mathcal{H}_d}(\text{curs}_{d+1}) \cup N_d$$

$$\text{chains}_d = \{[\pi_{\mathcal{H}_d}(x)] +\!\!+ [x] +\!\!+ xs : [x] +\!\!+ xs \in \text{chains}_{d+1})\}$$
$$\cup \{[p] : p \in \text{splitPars}_{\mathcal{H}_{d+1}}(\text{curs}_{d+1}) \setminus \{\pi_{\mathcal{H}_d}(x) : [x] +\!\!+ xs \in \text{chains}_{d+1}\}\}$$

Final Result: $\text{unzip}^{\mathcal{M}}_{N;d_{\min}}(\mathcal{H}) = (\mathcal{H}_{d_{\min}}, \text{chains}_{d_{\min}}) \equiv (\text{unzip}^{\mathcal{M}^*}_{N;d_{\min}}(\mathcal{H}), \text{chains}_{d_{\min}})$

Table 4: Two analogous definitions of multi-node unzipping. The closed form definition is helpful for proofs, whilst the depth-based (tail) recursion models how the algorithm would be implemented as a down loop from $\mathcal{D}_{\max}(N) - 1$ to $d_{\min}$.

in curs_{d+1} (in the modified hierarchy $\mathcal{H}_d$) and the nodes in N at depth d (we effectively add in nodes from N as we reach their depth). Subsequently, each existing chain is prepended with the parent of its head node, and a new chain is added for each node other than one of these parents that was generated by the split. The result of the algorithm is once again $(\mathcal{H}_{d_{\min}}, \text{chains}_{d_{\min}})$, the state of the hierarchy and chains after the last loop iteration.

An example is shown in Figure 7. Figure 8 shows the effects of performing repeated single-node unzips on the same example for comparison: unlike with multi-node unzipping, all three green leaves end up being fully unzipped up to the target depth, and the same ancestor nodes have to be split multiple times.

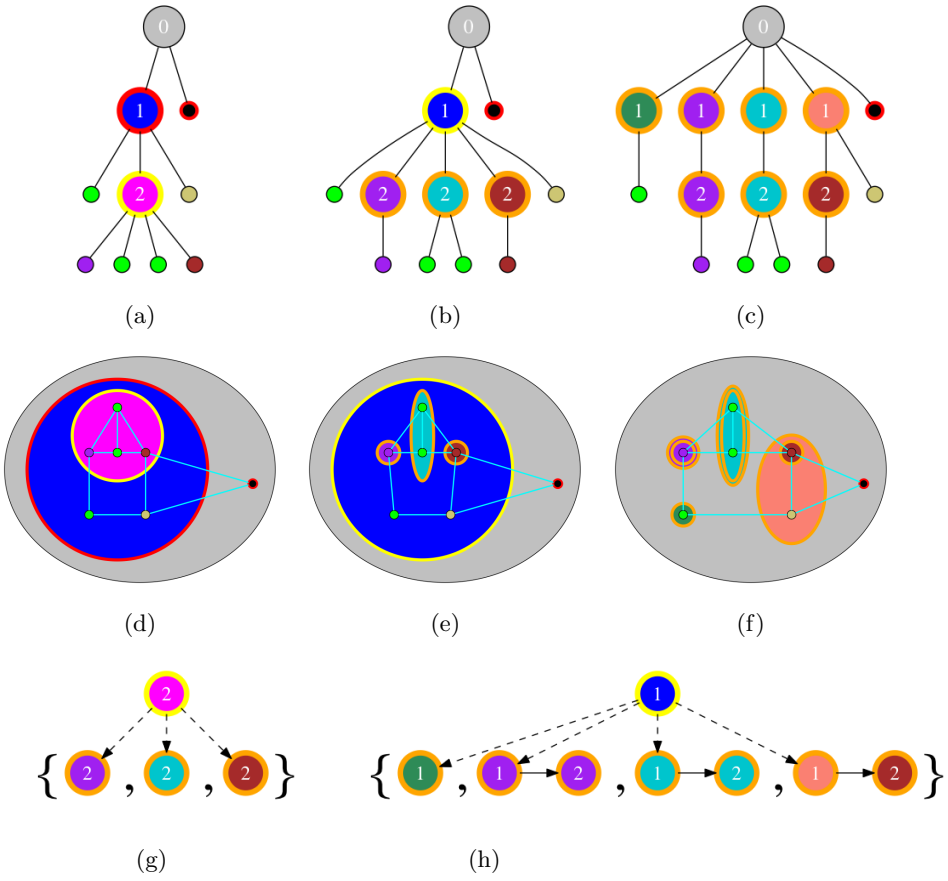


Figure 7: Using multi-node unzipping to unzip the three green leaves to the depth of the nodes circled in red (the numbers on the branch nodes denote their depth): (a) and (d) show the initial state of the hierarchy; (b) and (e) show it after the first unzipping iteration; (c) and (f) show its final state; (g) and (h) show the chain updates after each iteration.

4.2. Chain Zipping

Chain zipping is the inverse of single-node unzipping: it takes as input a set of node chains C (not necessarily all the same length), whose highest nodes must all be at the same depth in the hierarchy, and merges the corresponding nodes in the chains at each relevant depth from the top of the hierarchy downwards. Working downwards ensures that by the time we come to merge the nodes at any particular depth, they all have the same parent and can be merged as siblings.

Formally, we can define the effects of the operation in closed form, as shown in Table 5. This form allows us to formally prove that chain zipping can be used to invert single-node unzipping, as we do in the supplementary material. Letting $d_{\min}$ and $d_{\max}$ respectively be the minimum and maximum depths of

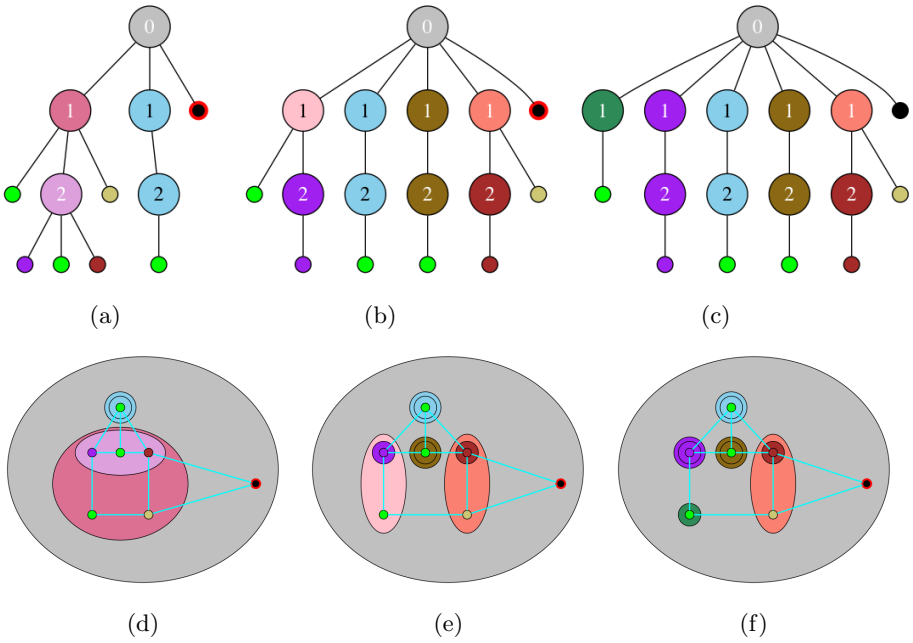


Figure 8: The results of unzipping each of the three green leaves in Figure 7(a) sequentially. In contrast to the results of multi-node unzipping in Figures 7(c) and 7(f), here the two green leaves that share a common parent at depth 2 end up in separate strands (c and f).

any node in one of the chains, the essence of the algorithm is to replace the set of chain nodes N_d^C at each depth $d \in [d_{\min}, d_{\max}]$ with a node generated by merging these nodes together.

An example is shown in Figure 9, and pseudo-code can be found in the supplementary material. A number of important preconditions must be checked in order to avoid invalid zips: (i) there must be chains to zip, (ii) each chain must be non-empty, (iii) no chain may extend down as far as a leaf, (iv) the highest nodes in the chains must be siblings, and (v) the nodes to be merged at each depth must be connected. These are checked by `zipPre` in Table 5. Precondition (iii) prevents leaves being merged, on the basis that they correspond to atomic objects in the problem domain (e.g. pixels in the case of image segmentation).

5. Higher-Level Editing Algorithms

As the multi-level counterparts of node splitting and node merging, the zipping algorithms described in Section 4 are broadly applicable as useful building blocks for hierarchy editing. In this section, we demonstrate this by using them

Chain Zipping

Auxiliary Definitions:

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

Inputs: C, the set of chains to zip

Helpers: $N^C = \{n \in c : c \in C\}$, the set of all nodes in C

$N^C_d = \{n \in N^C : \mathcal{D}(n) = d\}$, the set of all nodes of depth d in C

$d_{\min} = \mathcal{D}_{\min}(N^C)$, $d_{\max} = \mathcal{D}_{\max}(N^C)$

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

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

Table 5: A closed form definition of the chain zipping algorithm. In our *millipede* implementation, the algorithm is implemented as a loop that merges the nodes in the chains at each depth, starting from $d_{\min}$ and finishing at $d_{\max}$.

to develop algorithms to solve two quite different higher-level editing tasks.

The first task we consider is *non-sibling node merging*, the problem of merging nodes in a segmentation hierarchy when they do not share a common parent. This is a useful operation in an interactive context because it enables users to merge nodes that are adjacent in space in an intuitive way, without having to worry about the structure of the hierarchy. We show that non-sibling node merging can be implemented relatively straightforwardly in terms of our zipping algorithms, that it is needed for a significant proportion of the potential merges that a user can perform, and that it can considerably reduce the interaction burden on the user for individual merges. The second task we consider is *parent switching*, the problem of how to move a hierarchy node from being the child of one parent to the child of another. This problem was previously considered by Nacken [37], but his algorithm only allows a node's parent to be changed if the move would avoid disconnecting any of the node's ancestors. We present a more general algorithm that splits the node's ancestors as necessary and thereby

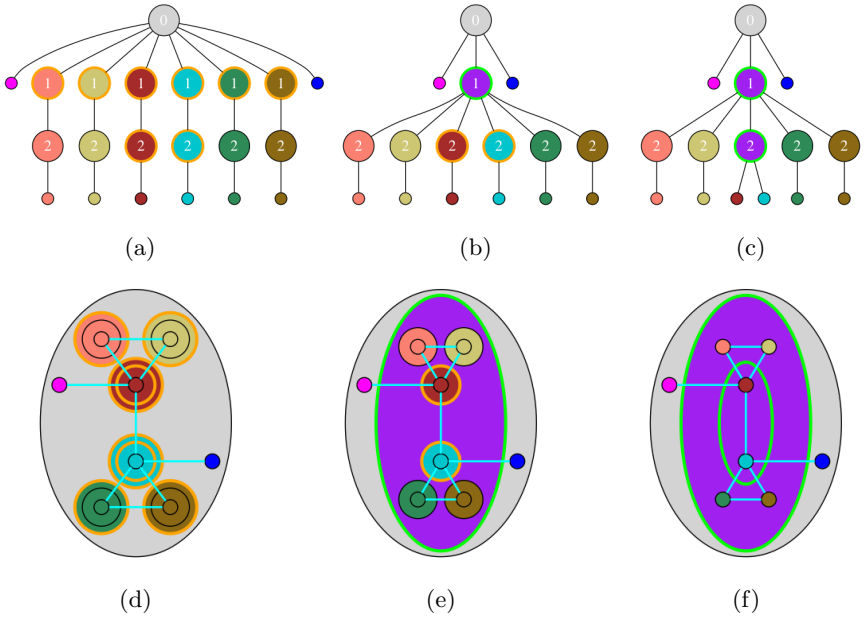


Figure 9: Using the chain zipping algorithm to zip together six chains (whose nodes are circled in orange): (a) and (d) show the initial state of the hierarchy; (b) and (e) show it after zipping together the depth 1 nodes in the chains; (c) and (f) show its final state, after zipping together the depth 2 nodes in the chains. The nodes resulting from the individual sibling node merges that occur during the process are circled in green.

allows its parent to be changed in a wider range of scenarios. We compare this new algorithm to Nacken's approach to highlight the differences.

5.1. Non-Sibling Node Merging

When editing a segmentation hierarchy, it is intuitive to want to merge nodes that are connected spatially, regardless of whether or not they share a common parent in the hierarchy (e.g. see Figure 10, in which a user might want to merge the nodes highlighted in the lower layer in order to create a combined region corresponding to a kidney). However, such a merge cannot be performed locally in the hierarchy: there is an inherent contradiction between the need for the node resulting from the merge to have a single parent (the hierarchy is a tree) and the fact that the nodes being merged may have different parents. In general, such a merge can only be performed by making significant non-local changes to the hierarchy above the nodes being merged so as to separate the nodes out from their ancestors and put them in a position to be merged as siblings. This would be hard to achieve using only local editing techniques such as node splitting and

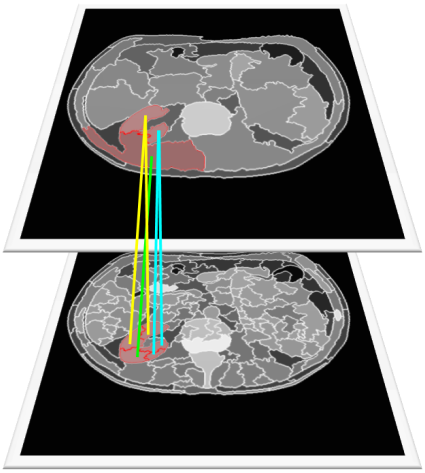


Figure 10: The need for non-sibling node merging: the user may want to merge nodes that are connected spatially, but this is non-trivial if they are not siblings in the hierarchy. Here, the user might like to merge the five highlighted regions in the lower layer to form a region corresponding to a kidney, but the regions do not have a common parent in the layer above (the coloured lines indicate the relevant parent-child relationships between the two layers). The lower-left kidney region is a particular problem, since its parent contains other regions that should not form part of the final kidney and would need to be split to allow the merge.

sibling node merging, but is much easier to achieve using our zipping algorithms. A qualitative example of this can be found in the supplementary material.

We define our non-sibling node merging algorithm as shown in Table 6. Informally, it takes a set, N, of arbitrary, non-leaf nodes that share a common depth, divides them into sets $\mathcal{K} = \{K_1, \ldots, K_k\}$ based on the connected components of their regions in the base graph, and then merges the nodes in each set individually. By construction, the region of the node resulting from merging the nodes in each set K_i will be connected. To merge the nodes in a set K_i, we first unzip them individually to a depth just below that of their lowest common ancestor, $a_i = \psi^+_{\mathcal{H}}(K_i)$. We then look up the chain corresponding to the new parent of each unzipped node (written as $\gamma(C_i^j, \pi_{\mathcal{H}_i^j}(n_i^j))$ in Table 6) and append the node itself to the chain in each case (since we want it to be merged as part of the zip). We then zip together the resulting chains to effect the merge. The whole process can be seen as a sequence of hierarchy transformations, with a number of unzips followed by a single zip for each K_i. An example is shown in Figure 11, and pseudo-code can be found in the supplementary material.

Non-Sibling Node Merging

Auxiliary Definitions:

$$\text{nccs}_{\mathcal{H}}(N) = \{\Psi_{\mathcal{H}}^{\mathcal{D}(N)}(R') : R' \in \text{ccs}_{\mathcal{H}}(\mathcal{R}(N))\} \text{ iff defined}(\mathcal{D}(N))$$

$$\gamma(C, n) = \{c \in C : \text{last}(c) = n\}$$

Inputs: N, the set of nodes to merge

Helpers:

$$\mathcal{K} = \text{nccs}_{\mathcal{H}}(N) = \{K_1, \ldots, K_k\}, \text{ where } K_i = \{n_i^1, \ldots, n_i^{|K_i|}\} \subseteq N$$

$$a_i = \psi_{\mathcal{H}}^{+}(K_i)$$

$$d_{\min}^i = \mathcal{D}(a_i) + 1$$

Hierarchy Transformation Sequence:

$$(\mathcal{H} =)\ \mathcal{H}_1^0 \overset{u}{\rightsquigarrow} \mathcal{H}_1^1 \overset{u}{\rightsquigarrow} \ldots \overset{u}{\rightsquigarrow} \mathcal{H}_1^{|K_1|} \overset{z}{\rightsquigarrow}$$

$$\ldots$$

$$\mathcal{H}_k^0 \overset{u}{\rightsquigarrow} \mathcal{H}_k^1 \overset{u}{\rightsquigarrow} \ldots \overset{u}{\rightsquigarrow} \mathcal{H}_k^{|K_k|} \overset{z}{\rightsquigarrow} H_{k+1}^0\ (= \mathcal{H}')$$

Intermediate Hierarchies:

$$\forall i \geq 1, j > 0 \quad (\mathcal{H}_i^j, C_i^j) = \text{unzip}^{\mathcal{S}}_{n_i^j; d_{\min}^i}(\mathcal{H}_i^{j-1})$$

$$\forall i \geq 1 \quad C_{i+1}^0 = \{\gamma(C_i^j, \pi_{\mathcal{H}_i^j}(n_i^j)) +\!\!+ (n_i^j) : 1 \leq j \leq |K_i|\}$$

$$\mathcal{H}_{i+1}^0 = \text{zip}_{C_{i+1}^0}(\mathcal{H}_i^{|K_i|})$$

$$\text{nonSiblingMerge}_N(\mathcal{H}) = \begin{cases} \mathcal{H}' & \text{if } |N| > 0 \text{ and defined}(\mathcal{D}(N)) \text{ and } \forall n \in N\ .\ (n \notin V_{\mathcal{H}}) \\ \mathcal{H} & \text{otherwise} \end{cases}$$

Table 6: A definition of the non-sibling node merging algorithm. Informally, this divides the input nodes in N into their connected components $\mathcal{K}$, unzips the nodes in each component K_i up to just below their common ancestor a_i, and then zips the resulting chains for each component back together again to complete the merge. We formalise this as a sequence of hierarchy transformations in which u denotes a single-node unzip and z denotes a chain zip.

5.2. Parent Switching

Segmentation hierarchy construction techniques generally either recursively merge a graph's constituent nodes into larger ones, or recursively split the graph itself into pieces. However, whilst the processes of merging or splitting the nodes during these algorithms are simple, choosing *when* to merge or split them can be extremely challenging in many problem domains. As a result, it is common for hierarchies to contain nodes that have been inappropriately merged into the wrong parent. An example of this can be seen in Figure 10, in which the lower-left part of a kidney has been merged into a non-kidney parent region.

The intuitive way in which a user might want to fix the problem would be

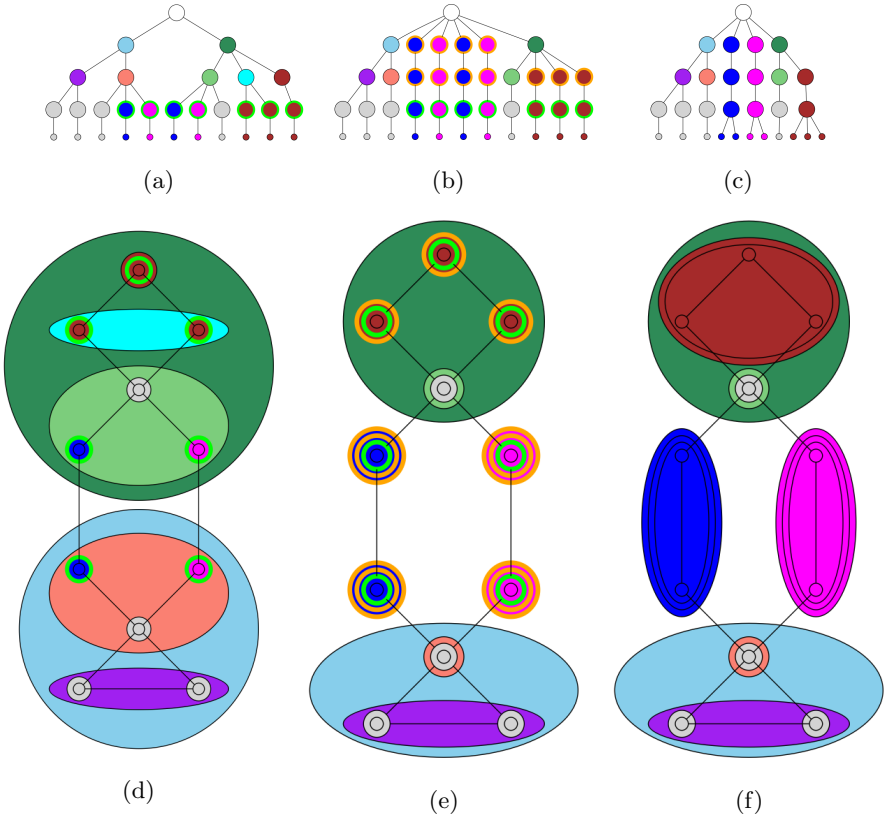


Figure 11: An example of non-sibling node merging: (a, d) show the initial state, in which we are about to perform a non-sibling merge on three connected components (blue, magenta and brown); (b, e) show the state after the nodes in each component have been unzipped up to just below their lowest common ancestors; (c, f) show the final state. For simplicity, this figure first shows the result of performing all the unzips involved in the operation (b, e), and then shows the final result after performing all of the zips involved (c, f). In reality, we perform a set of unzips followed by a zip for each component in turn, but the end result is the same. The chain nodes resulting from the various unzips are circled in orange in (b, e).

to move a node from being the child of one parent to the child of another, but not all such moves are valid, and of those that are, not all can be performed without making non-local changes to the hierarchy. Indeed, as Nacken observed [37], there are two requirements that must be met for a local move to be possible: the child node being moved must be spatially-adjacent to its new parent (this is required even for non-local moves), and moving it must not cause any of its ancestors (including its parent) to become spatially disconnected. Nacken's 'connectivity-preserving relinking' approach non-locally checks both requirements and then performs a trivial update of the parent pointer of the

Parent Switching

Auxiliary Definitions: $\text{ancChain}^{d'}_{\mathcal{H}}(n) = \left(\psi^{d}_{\mathcal{H}}(n) : d' \leq d \leq \mathcal{D}(n)\right)$

$\gamma(C, n) = \{c \in C : \text{last}(c) = n\}$

Inputs: n, the node whose parent is to be switched

p, the new parent of the node

Helpers: $d_{\min} = \mathcal{D}(\psi^{+}_{\mathcal{H}}(\{n, p\})) + 1$

Hierarchy Transformation Sequence: $\mathcal{H} \overset{u}{\rightsquigarrow} \mathcal{H}_u \overset{z}{\rightsquigarrow} \text{parentSwitch}_{n;p}(\mathcal{H})$

$$(\mathcal{H}_u, C_u) = \text{unzip}_{n;d_{\min}}(\mathcal{H})$$

$$\begin{aligned} C &= \left\{\text{ancChain}^{d_{\min}}_{\mathcal{H}_u}(\pi_{\mathcal{H}_u}(n)), \text{ancChain}^{d_{\min}}_{\mathcal{H}_u}(p)\right\} \\ &= \left\{\gamma(C_u, \pi_{\mathcal{H}_u}(n)), \text{ancChain}^{d_{\min}}_{\mathcal{H}_u}(p)\right\} \end{aligned}$$

$$\text{parentSwitch}_{n;p}(\mathcal{H}) = \begin{cases} \text{zip}_C(\mathcal{H}_u) & \text{if } \mathcal{D}(n) = \mathcal{D}(p) + 1 \text{ and } \exists c \in \Pi^{-1}(p) \,.\, \text{adjacent}(n, c) \\ \mathcal{H} & \text{otherwise} \end{cases}$$

Table 7: The parent switching algorithm: this first checks that n, the node whose parent is to be switched, is adjacent to a child of its new parent, p. It then unzips n to a depth $d_{\min}$, just below the lowest common ancestor of n and p. Finally, it zips the chain leading from just below the common ancestor down to the parent of n to the chain leading down to p.

node being moved if the move can be resolved locally. This is an entirely reasonable solution to the problem he was solving, but is quite restrictive as a solution to the more general problem we deal with here: it is not strictly necessary to prevent moves that can cause ancestors to become disconnected as long as we can arrange for those ancestors to be appropriately split. Our zipping algorithms provide us with a convenient way of arranging for this to happen.

We define our parent switching algorithm as shown in Table 7. It takes a node n, whose parent is to be switched, and its proposed new parent p, and first checks that n is adjacent to at least one child of p. if so, it unzips n to a depth $d_{\min} = \mathcal{D}(\psi^{+}_{\mathcal{H}}(\{n, p\})) + 1$, i.e. just below the lowest common ancestor of n and p. It then zips the resulting chain leading down to the parent of n, written as $\gamma(C_u, \pi_{\mathcal{H}_u}(n))$ in Table 7, to $\texttt{ancChain}^{d_{\min}}_{\mathcal{H}_u}(p)$, a chain leading down from depth $d_{\min}$ to p, which has the desired effect of making n a child of p. Note that this latter chain can be computed as a by-product of computing $\psi^{+}_{\mathcal{H}}(\{n, p\})$, so no additional work is required. An example of parent switching is shown in

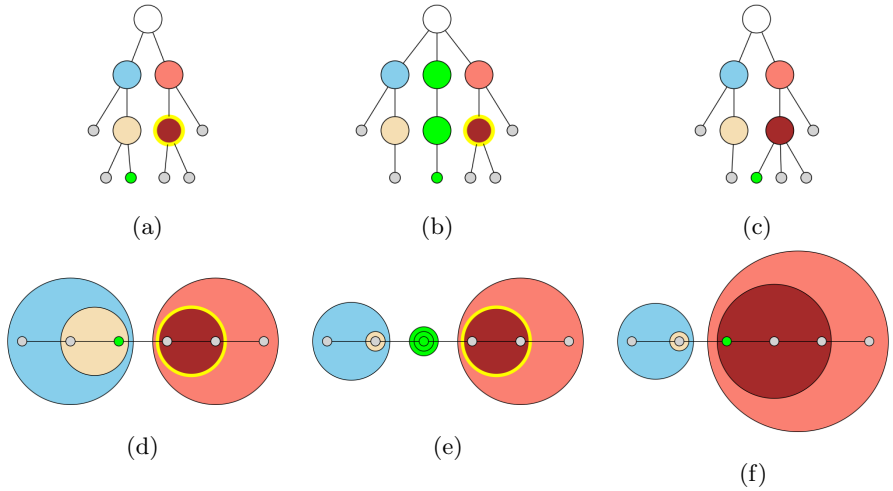


Figure 12: An example of parent switching: in (a), the node n to be moved is shown in green and its new parent p is circled in yellow; in (b), n is unzipped to just below the lowest common ancestor of n and p; in (c), the resulting chain leading down to the parent of n is zipped to a chain leading down to p to complete the switch.

Figure 12, and pseudo-code can be found in the supplementary material.

6. User Interface

We have implemented the algorithms described in Sections 4 and 5 in our *millipede* hierarchical image segmentation system [18]. In this section, we describe *millipede*'s user interface, and show how the user can invoke our algorithms to edit the graph hierarchy. Our system was designed to support the interactive segmentation of 3D medical images, which radiologists conventionally view slice-by-slice from one of three axis-aligned perspectives: axial (top-down), coronal (front) or sagittal (side). To make our system more intuitive for them, we adopted a similar approach to navigating around hierarchical segmentations of such images, with the notable addition of a 'layer' control to switch between hierarchy layers. The graph hierarchies we use in our system are complete (see Figure 2(c)), with layers 0-indexed from the leaf layer up (i.e. the deepest nodes in our hierarchies are in layer 0).

The key elements of our interface are shown in Figure 13. The left-hand view in the window shows an axial slice through a 3D abdominal CT scan that the user is trying to segment; the right-hand view shows the corresponding axial slice through one of the layers of a hierarchical segmentation of the scan.

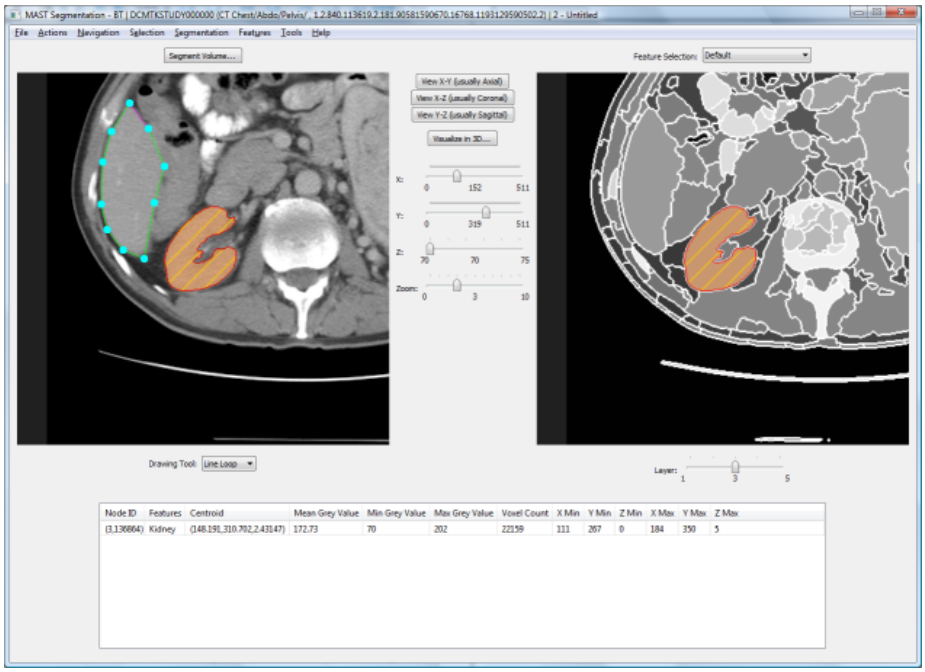


Figure 13: The *millipede* segmentation window: the left-hand view shows an axial slice through a 3D abdominal CT scan that the user is trying to segment; the right-hand view shows the corresponding axial slice through one of the layers of the hierarchical segmentation. Users can navigate using the controls between the views, and interact with the hierarchy by selecting regions in the right-hand view on which to perform editing operations.

The controls between the two views allow the user to change their position in the scan; the layer control beneath the right-hand view allows them to change hierarchy layer. Users interact with the hierarchy using operations of the kind described in Sections 4 and 5. A *hierarchical selection* (see Section 4.1.2 and the supplementary material) of nodes in the hierarchy is maintained to allow them to specify the inputs to these operations. They manipulate this selection by clicking on image regions in the right-hand view. The selection is maintained when changing hierarchy layers, allowing them to select a region in one layer and then drop down to the layer below and deselect one of its children, whilst keeping the rest of the region selected. The properties of nodes currently stored in the selection are shown in a table at the bottom of the window.

To perform non-sibling node merging in *millipede*, users simply select the regions to be merged in the right-hand view of the segmentation window, and then tell the system to perform the merge. For example, in Figure 10, they

would simply need to select the five highlighted regions in the lower layer and ask the system to merge them. One slight complication is that our algorithm is designed to merge nodes that are all at the same depth in the hierarchy (see Section 5.1), whereas a hierarchical selection can contain nodes at different depths. Since the hierarchies we use in *millipede* are complete, we can resolve this by simply performing the merge at the depth of the lowest node(s) in the selection, having split nodes that are higher up in the hierarchy into their descendants at that depth. For incomplete hierarchies (see Figure 2(b)), this strategy must be adapted slightly, since some selected nodes may not have descendants at the depth of the lowest node in the selection: the easiest solution is simply to prevent merges for which this situation arises.

The parent switching interface (see Figure 14) is more involved than that for merging, since the user needs to specify both the child node whose parent is to be changed, and its new parent in the layer above. These are provided separately: the user first selects the region with the incorrect parent and requests a parent switch, causing the system to highlight potential new parents in green. The user then selects the correct parent and tells the system to finalise the switch.

7. Evaluation

We demonstrate the benefits of non-sibling node merging and parent switching to users by quantitatively evaluating their usefulness and scalability on the 500 natural 2D images of the Berkeley Segmentation Data Set (BSDS500) [1], the 8629 synthetic 2D images of Crowley and Zisserman [10], and three 3D greyscale abdominal CT images kindly provided by the Churchill Hospital, Oxford. The BSDS500 is a popular benchmark for comparing hierarchy construction algorithms. Although its performance measures are not designed to evaluate hierarchy editing, it contains a rich set of natural images that can be used to demonstrate our algorithms' effectiveness for a wide variety of inputs. The Crowley dataset was chosen as a complete contrast to the BSDS500, demonstrating that our algorithms are equally effective for synthetic images (in this case paintings). Our abdominal CT images demonstrate the additional applicability of our algorithms to both greyscale and 3D images.

In addition to our quantitative results, we provide a qualitative example

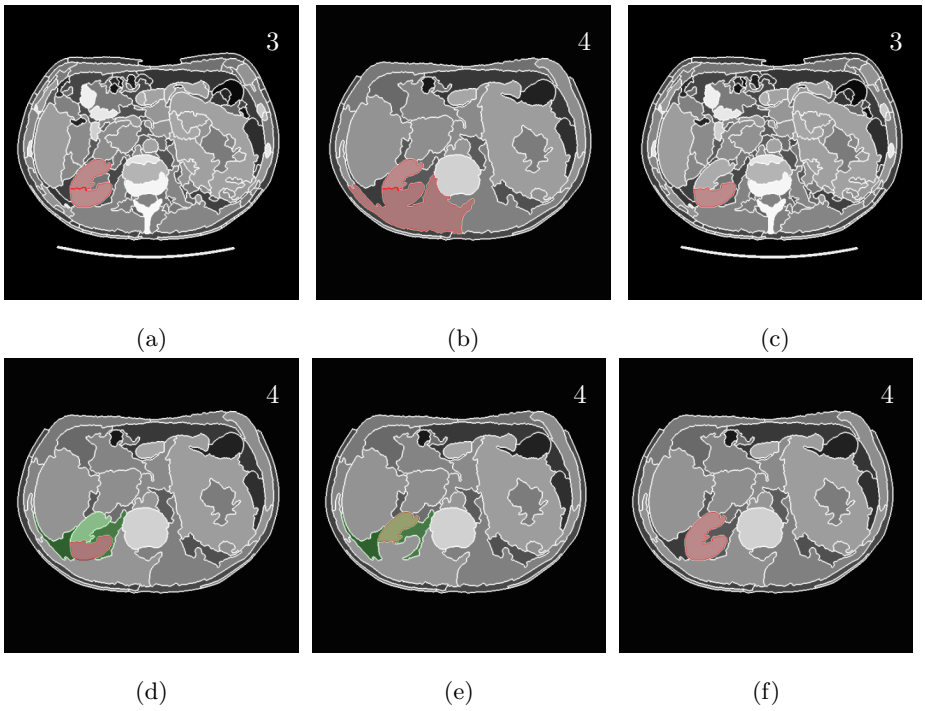


Figure 14: The user interface for parent switching (the number on each image denotes its hierarchy layer). The two regions selected in (a) are both part of the right kidney and should share a parent in the hierarchy, but one has been incorrectly merged into a different parent in the layer above (b). To correct this, the user selects the region with the incorrect parent (c) and requests a parent switch, causing the system to highlight potential new parents in green (d). The user then selects the correct parent region (e) and tells the system to finalise the switch. Afterwards, both of the original regions have the same parent in the layer above (f).

showing the effectiveness of non-sibling node merging in Figure 16. A further qualitative example can be found in the supplementary material, where we also analyse the computational complexity of all of our algorithms, and formally prove that our higher-level algorithms are optimal in the sense that they change as little of the hierarchy as possible to achieve their goals.

7.1. Hierarchy Construction

Segmentation hierarchies can be constructed using either agglomerative techniques [1, 4, 7, 44, 12, 14, 15, 39, 42], which recursively merge the nodes of a graph, or divisive techniques [27, 43, 45, 48, 49], which recursively split a graph into pieces. Whilst the hierarchies produced by these methods vary, our techniques are applicable regardless of the construction method used. We therefore experiment on the two types of hierarchy implemented in *millipede*, both of

which can be used for either 2D or 3D images: (i) shallow, 6-layer hierarchies based on Beucher's watershed/waterfall approach [7], and (ii) deep hierarchies with a specified number of layers that can be used to evaluate scalability. Various popular approaches [39, 1] produce similarly deep hierarchies, so an ability to edit them is important. For example, Najman and Schmitt [39] assign weights to the arcs separating catchment basins and then threshold the resulting weighted contour image at different levels to produce a hierarchy potentially containing thousands of layers, whilst Arbeláez et al. [1] present a general way of constructing hierarchies from non-closed contours based on the oriented watershed transform and ultrametric contour maps.

Watershed/Waterfall Hierarchies: Beucher's watershed transform [6] treats an image as a landscape and divides it into its catchment basins (one per local minimum in the image). Watersheds separate adjacent catchment basins. The idea is that by pre-transforming the image to make objects correspond to catchment basins, dividing the landscape into its catchment basins will segment the objects (e.g. we might perform the transform on the gradient magnitude of the image if the objects have similar values throughout their extent, since this will take high values at the objects' boundaries and low values within them). The watershed will oversegment non-smooth input images that contain many local minima. One way to mitigate this is to run the watershed iteratively to produce a waterfall hierarchy [7], which recursively clusters the small regions in the initial watershed partition to produce larger, more meaningful regions. To do this, we first smooth the input image using anisotropic diffusion filtering [41], an edge-preserving technique that reduces noise (and the number of local minima) in the image whilst trying to avoid blurring boundaries between objects. We segment the result using Meijster and Roerdink's watershed method [35] with a 6-connected neighbourhood to produce an initial partition of the image: this will become the lowest branch layer in the hierarchy. We construct higher branch layers by iteratively invoking a version of Beucher's waterfall transform [19] on a minimum spanning tree (MST) of the region adjacency graph of this lowest branch layer. This produces a complete segmentation hierarchy called an *image partition forest*, or IPF, whose layers partition the image at different scales [18].

Deep Hierarchies: This waterfall hierarchy construction approach builds

3D CT Image	Slices
BT	70–80
MC	110–120
SD	70–80

Parameter	Value
ADF Conductance	1.0
ADF Iterations	20
CT Window Level	40
CT Window Width	400

Table 8: Left: The slices chosen from the three 3D CT images we used for our experiments. Right: The parameter settings we used when constructing our IPFs. The ADF parameters control the anisotropic diffusion filtering [41]. Window level/width are used for CT images and control how their pixel values are converted from Hounsfield units to greyscale [18].

shallow hierarchies that quickly converge to a single region, but it can be easily adapted to build deep hierarchies with a specified number of layers. To do this, we modify the behaviour of each iterative transformation we perform on the MST to construct a new hierarchy layer: instead of performing a waterfall step, we now calculate the ideal number of merges that should take place in the current layer to achieve convergence to a single region at (but not before) the top of the hierarchy: we define this as the ceiling of the number of layers that still need to be constructed, divided by the number of remaining edges in the MST (i.e. the number of merges that still need to take place). We then merge edges in the MST, starting from those with the lowest weight, until we either reach this ideal number of merges or run out of edges to merge. This layer construction process is iterated until the specified number of layers have been constructed. It provides us with an effective means to control the rate of convergence to a single image region for evaluating scalability.

When constructing either type of hierarchy for a 3D CT image, we select a representative set of slices from the middle of the image to obtain a $512 \times 512 \times 11$ volume (we experimented with more slices, but found that this made little difference to the results whilst increasing the memory requirements and computation time). Table 8 shows the slices we chose for the 3D CT images and the parameters we use for hierarchy construction.

7.2. Usefulness of Non-Sibling Node Merging

To demonstrate the usefulness of non-sibling node merging, we show (i) that most potential merges that a user can perform between spatially-adjacent nodes in a segmentation hierarchy are between non-siblings, and (ii) that our algorithm considerably reduces the interaction burden on the user for such merges. We

Pairwise Node Merges Requiring Different Levels of Unzipping						
	BSDS500 [1]		Crowley [10]		3D CT	
Levels	%	Cumul.	%	Cumul.	%	Cumul.
4	5.33	5.33	4.99	4.99	18.83	18.83
3	9.68	15.01	9.40	14.39	14.80	33.63
2	16.73	31.74	16.34	30.73	17.28	50.91
1	27.27	59.01	27.72	58.45	23.10	74.01
0	40.99	100.00	41.55	100.00	25.99	100.00

Table 9: The average percentages of pairwise node merges that required different levels of unzipping in 6-layer IPFs for images from various datasets (see Section 7.2).

do this by constructing, for each image under consideration, a 6-layer waterfall-based IPF, as described in the previous section, and then calculating the number of levels of unzipping that would be required to merge each spatially-adjacent pair of nodes at equal depths in the image's IPF. Sibling nodes require no unzipping, and can be merged trivially; nodes whose parents differ must be unzipped up to their common ancestor as part of a non-sibling node merge.

Table 9 shows the percentages of pairwise node merges that required different levels of unzipping, averaged over the images in each dataset. We observe that across all three datasets, the average percentage of IPF node pairs that required at least one level of unzipping to merge (i.e. that involved *non-sibling* node merging) was never lower than 55%, and indeed was $> 70\%$ for our 3D CT images. Moreover, many node pairs ($> 50\%$ for our 3D CT images) required more than one level of unzipping to merge. Thus, not only do the majority of the pairwise merges that the user can request involve non-sibling node merging, they often require *multiple* levels of unzipping. As our example in the supplementary material shows, this is costly: each node split that is required as part of the unzip operation has relatively high input requirements (the user has to specify the components into which to split the node), there are corresponding sibling node merges to be performed, and keeping track of the changes required without an easy way to visualise the IPF's parent-child links carries a high cognitive overhead. Whilst we could circumvent this problem by restricting the user to sibling-only node merging, this would prevent many useful merges. In reality, it is also worth noting that many node merges are not pairwise, since the user often wants to combine many regions of the image at once. In this case, there is an even higher probability that multiple levels of unzipping will be required.

Parent Switches First Disconnecting Ancestors at Different Levels						
	BSDS500 [1]		Crowley [10]		3D CT	
First Ancestor	%	Cumul.	%	Cumul.	%	Cumul.
1	71.60	71.60	70.85	70.85	77.21	77.21
2	5.34	76.94	5.28	76.13	8.33	85.54
3	1.61	78.55	1.55	77.68	2.11	87.65
4	0.26	78.81	0.23	77.91	0.33	87.98
N/A	21.19	100.00	22.09	100.00	12.02	100.0

Table 10: Average percentages of possible parent switches in 6-layer IPFs from various datasets that first disconnect ancestors at different levels above the node being moved (Section 7.3).

7.3. Usefulness of Parent Switching

To demonstrate the usefulness of parent switching, we show that the proportion of nodes that can be moved from one parent to another is significantly higher with our approach than it would be using Nacken's connectivity-preserving relinking method [37]: this allows the user considerably more flexibility in rearranging segmentation hierarchies, and isolates them from the need to understand the structure of the hierarchy above the node being moved.

To show this, we first calculate all possible parent switches in each IPF by looking at the parents of the adjacent nodes of each non-leaf node in the IPF (specifically, each non-leaf node can potentially be moved to the parent of any adjacent node, provided that that parent is not also its own parent). We then calculate, for each possible switch, the level (relative to the node being moved) of the lowest ancestor (if any) that would be disconnected by the switch. This measure has value 1 for those switches that first disconnect the parent of the node being moved, 2 for those that first disconnect the grandparent, and so on. It can be calculated by examining the connectedness of each ancestor's descendants in the layer of the node being moved, after removing the node itself from consideration. Only those switches that do not disconnect any ancestor would be allowed using Nacken's relinking approach, whereas our method splits ancestors as necessary and thereby allows all possible switches.

Table 10 shows the percentages of possible parent switches that first disconnected ancestors at different levels above the node being moved, averaged over the images in each dataset. In each case, more than 75% of the possible switches would disconnect at least one ancestor, meaning that they would not be allowed using Nacken's relinking approach but would be allowed by our

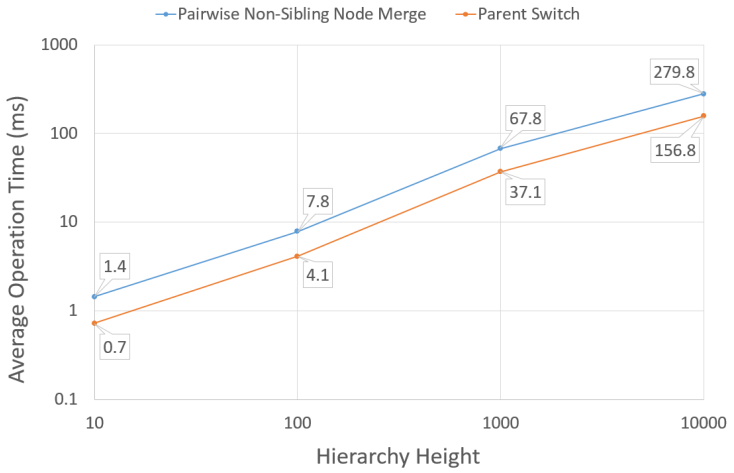


Figure 15: The time taken (in ms) by an average pairwise non-sibling node merge or parent switch in hierarchies of various heights on images from the BSDS500 [1]. Increasing the hierarchy height increases the number of layers involved in an average operation, but decreases the branching factor, so the time taken increases just under linearly with hierarchy height.

method. Moreover, 7-11% of the possible switches in each case would not disconnect the original parent of the node being moved, but would disconnect an ancestor higher up the hierarchy. To understand these switches, the user needs to be aware of the structure of a much larger portion of the hierarchy, since higher nodes generally have significantly more descendants than lower ones. Our method could thus be considered more intuitive than Nacken's approach, in that it obviates the need for the user to think about the connectivity of a node's ancestors when switching it to a new parent.

7.4. Scalability

To evaluate the scalability of our algorithms, we first use our deep hierarchy construction technique to construct hierarchies with 10, 100, 1000 and 10000 layers for each image in the BSDS500 dataset [1]. For each of these 2000 hierarchies, we then uniformly sample 100 pairwise non-sibling node merges and 100 parent switches, and time how long each operation takes. (In both cases, the uniform sampling is done by enumerating all the possible operations and uniformly sampling random indices.) Finally, we use these results to compute the average costs of both operations for each height of hierarchy we consider.

As shown in Figure 15, the average cost of both operations increases just

under linearly with the height of the hierarchy. This matches intuition: the number of layers involved in an average non-sibling node merge or parent switch increases roughly linearly with the hierarchy height, whilst the cost of each split and merge performed decreases slightly in deeper hierarchies due to a decrease in the branching factor. Our analysis in the supplementary material shows that the complexity of each algorithm is linear in both the number of layers involved and the split/merge costs, and our empirical results bear this out. Practically, the low cost of both pairwise non-sibling node merging and parent switching make them suitable for interactive use even in hierarchies with 10000 layers (although the cost of non-sibling node merges that are not pairwise can be higher, since the operation is also linear in the number of nodes being merged).

8. Conclusions

Despite the significant progress made in automatic segmentation in recent years, segmentation itself remains an under-specified problem, because the desired outcome is both user- and application-dependent. As a result, it is important to allow users to refine segmentations produced by automated algorithms. This can be straightforward for flat (one level) segmentations, but is significantly more difficult for hierarchical ones due to the need to ensure that the regions of all the hierarchy nodes remain connected: indeed, many existing editing algorithms allow only small, local hierarchy changes.

The multi-scale split and merge ('zipping') algorithms we have presented here offer a potential solution to this problem: by encapsulating the details of connectivity preservation, they free higher-level operations from the need to deal with connectivity explicitly, making them easier to develop and reason about. The *non-sibling node merging* and *parent switching* techniques we have presented attest to the efficacy of this approach: both are much simpler when expressed in terms of our zipping algorithms than they would be without them.

Both of these higher-level techniques bring practical benefits to the user, non-sibling node merging because it allows users to merge nodes that are adjacent in space intuitively, without worrying about the structure of the hierarchy, and parent switching because it allows them to correct the parent links of nodes that were wrongly merged during hierarchy construction. For non-sibling node merg-

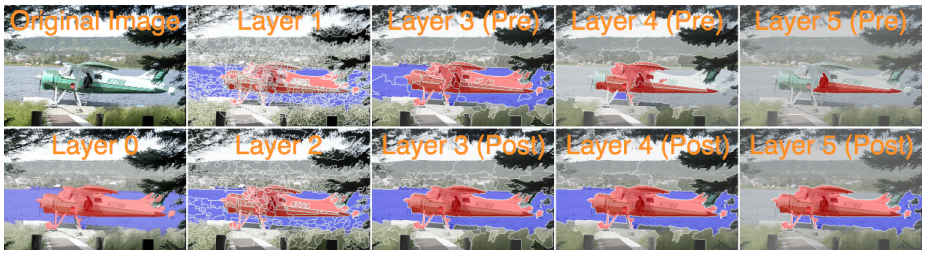


Figure 16: A qualitative example showing non-sibling node merging being used to merge together spatially-adjacent regions for an aircraft and a lake in layer 3 of a segmentation hierarchy for the `aero_2007_006232` image of the Gulshan dataset [24]. Two separate merges are performed, one on the regions marked in red and another on the regions marked in blue; both merges involve more than one connected component. The layers below the merge layer are unaffected by the merges. Some regions in higher layers are split by the merges, making the remaining parts of them easier to identify as other objects in subsequent editing. All hierarchy regions remain connected at the end of the process.

ing, our experiments show that over half the pairwise node merge operations a user can perform between spatially-adjacent nodes are between non-siblings, and would require split/sibling merge operations on multiple levels of the hierarchy to effect. Such non-sibling merges are much easier to effect using our zipping algorithms, which remove the need for the user to specify the parameters of the individual split/sibling merge operations involved. For parent switching, our experiments indicate that more than 75% of the switches that a user can perform involve the disconnection of at least one ancestor node in the hierarchy. Such switches would be prevented by Nacken's relinking approach, but can be supported relatively straightforwardly using our zipping-based approach.

Acknowledgements

We thank Dr. Zoe Traill for providing us with CT scans and helping us interpret them, and EPSRC for funding S. Golodetz's Doctoral Training Award (DTA). We also thank Anurag Arnab for his feedback on the final draft of this paper.

References

[1] Arbeláez, P., Maire, M., Fowlkes, C., Malik, J., 2011. Contour Detection and Hierarchical Image Segmentation. TPAMI 33, 898–916.

[2] Archambault, D., Munzner, T., Auber, D., 2008. GrouseFlocks: Steerable Exploration of Graph Hierarchy Space. TVCG 14.

[3] Archambault, D., Munzner, T., Auber, D., 2011. Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths. TVCG 17.

[4] Attene, M., Falcidieno, B., Spagnuolo, M., 2006. Hierarchical mesh segmentation based on fitting primitives. The Visual Computer 22.

[5] Auber, D., Jourdan, F., 2005. Interactive Refinement of Multi-scale Network Clusterings, in: ICIV.

[6] Beucher, S., 1990. Segmentation d'Images et Morphologie Mathématique. Ph.D. thesis. E.N.S. des Mines de Paris.

[7] Beucher, S., 1994. Watershed, Hierarchical Segmentation and Waterfall Algorithm, in: ISMM.

[8] Cardelino, J., Caselles, V., Bertalmío, M., Randall, G., 2013. A Contrario Selection of Optimal Partitions for Image Segmentation. SIIMS 6.

[9] Cousty, J., Najman, L., 2014. Morphological Floodings and Optimal Cuts in Hierarchies, in: ICIP, pp. 4462–4466.

[10] Crowley, E.J., Zisserman, A., 2014. The State of the Art: Object Retrieval in Paintings using Discriminative Regions, in: BMVC.

[11] Eades, P., Huang, M.L., 2000. Navigating Clustered Graphs using Force-Directed Methods. JGAA 4.

[12] Felzenszwalb, P.F., Huttenlocher, D.P., 2004. Efficient Graph-Based Image Segmentation. International Journal of Computer Vision 59.

[13] Fisher, D.H., 1987. Knowledge Acquisition Via Incremental Conceptual Clustering. Machine Learning 2.

[14] Flake, G.W., Tarjan, R.E., Tsioutsiouliklis, K., 2004. Graph Clustering and Minimum Cut Trees. Internet Mathematics 1.

[15] Garland, M., Willmott, A., Heckbert, P.S., 2001. Hierarchical Face Clustering on Polygonal Surfaces, in: I3D.

[16] Gerstmayer, M., Haxhimusa, Y., Kropatsch, W.G., 2011. Hierarchical Interactive Image Segmentation using Irregular Pyramids, in: Graph-Based Representations in Pattern Recognition.

[17] Glantz, R., Kropatsch, W.G., 2000. Relinking of Graph Pyramids by Means of a New Representation, in: CPRW.

[18] 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.

[19] Golodetz, S., Nicholls, C., Voiculescu, I., Cameron, S., 2014. Two Tree-Based Methods for the Waterfall. Pattern Recognition 47.

[20] Golodetz, S., Voiculescu, I., Cameron, S., 2008. Region Analysis of Abdominal CT Scans using Image Partition Forests, in: CSTST.

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

[22] Grundmann, M., Kwatra, V., Han, M., Essa, I., 2010. Efficient Hierarchical Graph-Based Video Segmentation, in: CVPR.

[23] Guigues, L., Cocquerez, J.P., Men, H.L., 2006. Scale-Sets Image Analysis. IJCV 68.

[24] Gulshan, V., Rother, C., Criminisi, A., Blake, A., Zisserman, A., 2010. Geodesic Star Convexity for Interactive Image Segmentation, in: CVPR.

[25] Haxhimusa, Y., Kropatsch, W., 2003. Hierarchical Image Partitioning with Dual Graph Contraction. Technical Report PRIP-TR-81. TU Wien.

[26] Hickson, S., Birchfield, S., Essa, I., Christensen, H., 2014. Efficient Hierarchical Graph-Based Segmentation of RGBD Videos, in: CVPR.

[27] Katz, S., Tal, A., 2003. Hierarchical Mesh Decomposition using Fuzzy Clustering and Cuts. TOG 22.

[28] Kerren, A., 2006. Interactive Visualization of Graph Pyramids, in: Graph Drawing.

[29] Kiran, B.R., Serra, J., 2014. Global-local optimizations by hierarchical cuts and climbing energies. Pattern Recognition 47.

[30] Klava, B., Hirata, N.S.T., 2007. Watershed segmentation: Switching back and forth between markers and hierarchies, in: ISMM.

[31] Lezoray, O., Meurie, C., Belhomme, P., Elmoataz, A., 2006. Multi-Scale Image Segmentation in a Hierarchy of Partitions, in: EUSIPCO.

[32] Maire, M., Yu, S.X., Perona, P., 2013. Hierarchical Scene Annotation, in: BMVC.

[33] Martin, D., Fowlkes, C., Tal, D., Malik, J., 2001. A Database of Human Segmented Natural Images and its Application to Evaluating Segmentation Algorithms and Measuring Ecological Statistics, in: ICCV.

[34] Marčan, M., Voiculescu, I., 2016. Unsupervised segmentation of MRI knees using image partition forests, in: SPIE Medical Imaging.

[35] Meijster, A., Roerdink, J., 1998. A Disjoint Set Algorithm for the Watershed Transform, in: EUSIPCO.

[36] Meyer, F., Beucher, S., 1990. Morphological Segmentation. VCIR 1.

[37] Nacken, P.F.M., 1995. Image Segmentation by Connectivity Preserving Relinking in Hierarchical Graph Structures. Pattern Recognition 28.

[38] Najman, L., Cousty, J., 2014. A graph-based mathematical morphology reader. PRL 47, 3–17.

[39] Najman, L., Schmitt, M., 1996. Geodesic Saliency of Watershed Contours and Hierarchical Segmentation. TPAMI 18.

[40] Peng, B., Zhang, L., Zhang, D., 2013. A survey of graph theoretical approaches to image segmentation. Pattern Recognition 46.

[41] Perona, P., Malik, J., 1990. Scale-Space and Edge Detection Using Anisotropic Diffusion. TPAMI 12.

[42] Prasad, L., Swaminarayan, S., 2008. Hierarchical image segmentation by polygon grouping, in: CVPR.

[43] Shi, J., Malik, J., 2000. Normalized Cuts and Image Segmentation. TPAMI 22.

[44] de Souza, K.J.F., de Albuquerque Araújo, A., do Patrocínio Jr, Z.K.G., Guimarães, S.J.F., 2014. Graph-based hierarchical video segmentation based on a simple dissimilarity measure. PRL 47.

[45] Tierny, J., Vandeborre, J.P., Daoudi, M., 2007. Topology driven 3D mesh hierarchical segmentation, in: SMI.

[46] Xu, C., Xiong, C., Corso, J.J., 2012. Streaming Hierarchical Video Segmentation, in: ECCV.

[47] Yeghiazaryan, V., Voiculescu, I., 2016. Unsupervised 3D Renal Segmentation Based on Image Partitioning, in: SPIE Medical Imaging.

[48] Yuruk, N., Mete, M., Xu, X., 2007. A Divisive Hierarchical Structural Clustering Algorithm for Networks, in: ICDM.

[49] Zivkovic, Z., Bakker, B., Kröse, B., 2006. Hierarchical Map Building and Planning based on Graph Partitioning, in: ICRA.