Department of Computer Science

Simpler Editing of Spatially-Connected Graph Hierarchies using Zipping Algorithms

Stuart Golodetz, Irina Voiculescu and Stephen Cameron

CS-RR-15-06

Department of Computer Science, University of Oxford
Wolfson Building, Parks Road, Oxford, OX1 3QD

Simpler Editing of Spatially-Connected Graph Hierarchies using Zipping Algorithms

Stuart Golodetz *[1], Irina Voiculescu[2], and Stephen Cameron[2]

[1]Department of Engineering Science, University of Oxford, UK, OX1 3PJ
[2]Department of Computer Science, University of Oxford, UK, OX1 3QD

Abstract

Graph hierarchies, the data structures that result from hierarchically clustering the nodes of a graph, are widely used as a multi-scale way of representing data in many computing fields, including image segmentation, mesh simplification, hierarchical pathfinding and visualisation. As a result, significant efforts have been made over the years to find ways of constructing 'good' graph hierarchies, a task that is an instance of the hierarchical segmentation problem. However, it is still comparatively rare to see techniques for *editing* graph hierarchies, despite the fact that such editing has many potential uses, e.g. for tasks such as correcting initially poor segmentations of images, or visualising the effects of hypothetical changes to a network. We believe that this relative lack of attention may be partly explained by the fact that graph hierarchies in many domains have spatial connectivity constraints associated with them that must be preserved during editing, significantly complicating the development of techniques that require non-trivial changes to be made to the hierarchy. In this paper, we therefore propose a way of facilitating the development of editing algorithms for such 'spatially-connected graph hierarchies' by introducing multi-level split and merge algorithms ('zipping algorithms') for them that can be used as building blocks for more sophisticated editing. We demonstrate the advantages of our zipping algorithms by using them to solve the practical problem of how to merge non-sibling image regions in *millipede*, our tool for 3D hierarchical image segmentation. We also demonstrate how our zipping algorithms can be used to reformulate existing techniques in the literature to make them either more general or easier to understand.

1 Introduction

Graph hierarchies, the data structures that result from hierarchically clustering the nodes of a graph, are widely used as a multi-scale way of representing data in a variety of computing fields, including (to name but a few) image segmentation [1, 8, 13, 20, 26, 34, 35, 37, 39, 45], mesh simplification [4, 17, 30, 47], hierarchical pathfinding [9, 27, 28, 29, 32, 48] and visualisation [2, 3, 5, 10, 11, 31]. This ubiquity is not surprising – graphs are a standard choice for representing the interrelationships between objects, and hierarchically clustering said graphs is a well-understood way of abstracting away from the details of the objects concerned to represent problem domain instances in a compact and usable way. As a result, significant efforts have been made over the years to find ways of constructing 'good' graph hierarchies in different contexts.

However, despite the amount of research attention that has been paid to the construction of graph hierarchies, it is still comparatively rare to see techniques for *editing* graph hierarchies after they have been constructed – this is in some ways surprising, since graph hierarchy editing is demonstrably useful, for example for tasks such as correcting initially poor segmentations of images, or visualising the effects of hypothetical changes to a network. We believe that this relative lack of attention may be at least partly explained by the fact that graph hierarchies in many domains have spatial connectivity constraints associated with them that must be preserved when making hierarchy changes: algorithms designed to allow sophisticated hierarchy editing often have to make extensive changes to the hierarchy, and correctly preserving spatial connectivity in the

*E-mail: sgolodetz@gxstudios.net; Corresponding author

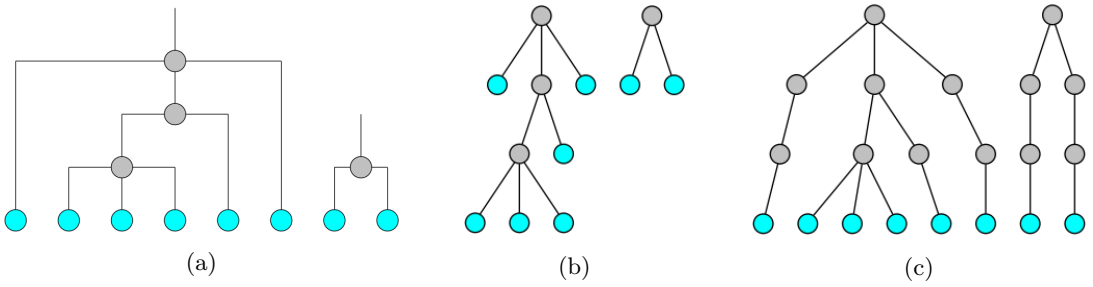


Figure 1: A graph hierarchy results from recursively grouping together the nodes of a base graph (drawn in cyan). Its hierarchical structure can be drawn as a dendogram (a) or an incomplete forest (b). A graph hierarchy whose hierarchical structure is a complete forest (c) is a special case that we refer to as a hierarchy of partitions, graph pyramid or partition forest.

face of these changes can be non-trivial. For what appears to be this reason, most existing editing algorithms, e.g. the grouping and ungrouping operations described in [5] or the cluster creation and deletion operations described in [10], are therefore aimed at making small, local changes to the hierarchy. Algorithms that do attempt more sophisticated editing, e.g. the TugGraph algorithm of Archambault et al. [3], have to spend most of their efforts on spatial connectivity preservation, generally in quite a problem-specific manner. This makes them significantly more complicated to develop and harder for others to understand.

In this paper, we therefore propose a way of alleviating these problems by introducing multi-level split and merge algorithms for graph hierarchies with such constraints (which we call 'spatially-connected graph hierarchies') that preserve spatial connectivity and can be used as building blocks for more sophisticated editing – we refer to these as 'zipping algorithms' because of the zip-like effects they have when applied to a hierarchy. By encapsulating the logic needed to preserve spatial connectivity in a single place, these algorithms make it easier for new, higher-level hierarchy editing techniques to be developed.

We demonstrate the effectiveness of our zipping algorithms via a sequence of three applications. In the first application, we show how to solve the problem of non-sibling node merging, namely how to merge nodes in a graph hierarchy when they do not share a common parent. This is a useful operation because it allows a user to perform 'intuitive' merges of nodes that are adjacent in space without needing to know about the hierarchy itself. In the second application, we show how to implement parent switching, the moving of a child node between parents in the hierarchy. This generalises the relinking technique described by Nacken in [39]. In our final application, we show how to reformulate the TugGraph algorithm [3] in terms of unzipping, our multi-level split algorithm, making it easier to understand and implement.

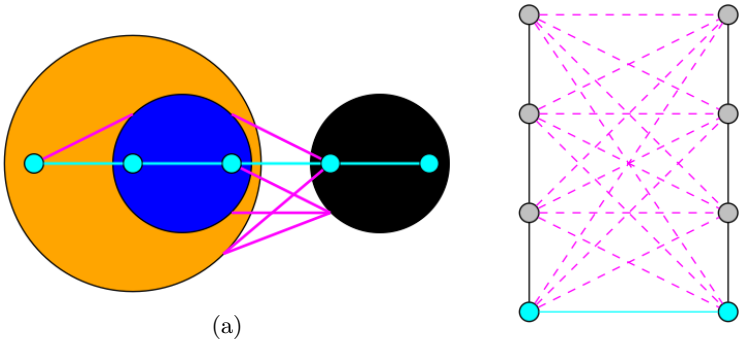
The organisation of this paper is as follows: in §2, we review graph hierarchies and the existing ways in which they are constructed and edited; in §3, we introduce our zipping algorithms; in §4, we demonstrate various applications of our algorithms; and in §5 we conclude.

2 Background

2.1 Graph Hierarchies

At an abstract level, a *graph hierarchy* is a data structure that results from recursively grouping together the nodes of a base graph. In a *spatially-connected graph hierarchy*, the nodes grouped must additionally be adjacent in space, so that every node in the resulting hierarchy ultimately represents a spatially-connected set of nodes in the base graph.

The hierarchical structure of a graph hierarchy can be visually depicted either as a dendogram (see Figure 1(a)) or as a forest whose lower layers may or may not be complete (see Figure 1(b)). Graph hierarchies whose forests only have complete layers (see Figure 1(c)) are an interesting special case, since any complete layer necessarily partitions the base graph. Complete graph hierarchies are often given a distinctive name in the literature to reflect this, e.g. hierarchy of partitions [26, 35]



(a) (b)

Figure 2: Clustering the nodes of a base graph induces higher-level edges in the hierarchy: (a) shows the higher-level edges that are induced in an example graph hierarchy; (b) shows that in the worst case a single base graph edge can cause a quadratic number of higher-level edges to be induced, since a higher-level edge implicitly exists between every pair of ancestors of the base edge's endpoints in the hierarchy. In both cases, the base graph edges are shown in cyan and the higher-level edges in magenta.

or graph pyramid [31] (indeed, in our own previous work we have referred to them as *partition forests* [20, 23, 24]).

As depicted in Figures 1(a)–(c), the process of hierarchically clustering the nodes of the base graph creates higher-level nodes in the hierarchy, each of which represents the aggregation of its descendants. However, it also induces higher-level edges between these nodes: conceptually, a higher-level edge exists between any two nodes that have descendants joined by an edge in the base graph (see Figure 2(a)). Since in the worst case, as shown in Figure 2(b), each edge in the base graph can be responsible for a number of higher-level edges that is quadratic in the height of the hierarchy, the choice of how to represent this connectivity information is crucial. The options available include storing all of the edges, storing just the base edges and recalculating the higher-level edges as necessary, or, especially in the case of a complete hierarchy, storing just the edges between nodes of the same height. Clearly, there is a trade-off to be made between the cost (both in terms of storage and maintenance) of explicitly tracking all of the high-level edges, and the cost of recalculating them on demand. The choice made will generally be application-specific and depend on how often higher-level edges need to be accessed versus how often they need to be updated. From a practical standpoint, our zipping algorithms can cope with any of these alternatives – the differences will be encapsulated in the individual split and merge operations on which we build.

In practice, we are interested not only in the existence of objects and the connections between them, but also in representing facts about the objects and the ways in which they interact. As a result, it is common to allow both nodes and edges in the hierarchy (whether or not all of the edges are explicitly part of the hierarchy representation) to have domain-specific properties associated with them. (For example, in the context of image segmentation, in which hierarchy nodes correspond to regions in an image, we might want to record the mean pixel value in each node's corresponding region.) Since higher-level nodes and edges owe their existence to the nodes and edges on which they are based, however, we must be careful about the precise way in which we allow property association to occur. For this reason, we allow the properties of nodes and edges in the base graph to be assigned arbitrarily (in domains such as image segmentation, they will generally be computed from the image), but we insist that all properties of higher-level nodes and edges be deterministically computable as a function only of the properties of their descendants. This reflects the fact that higher-level nodes and edges are in essence nothing more than aggregates of their descendants. As a result, we must be careful to give nodes suitable sets of properties to ensure that this computability requirement is satisfied – e.g. if we want to provide a mean pixel value property then we also need to provide an area property so that the mean pixel values of a set of nodes can be combined to compute the mean pixel value of their parent (see Figure 3).

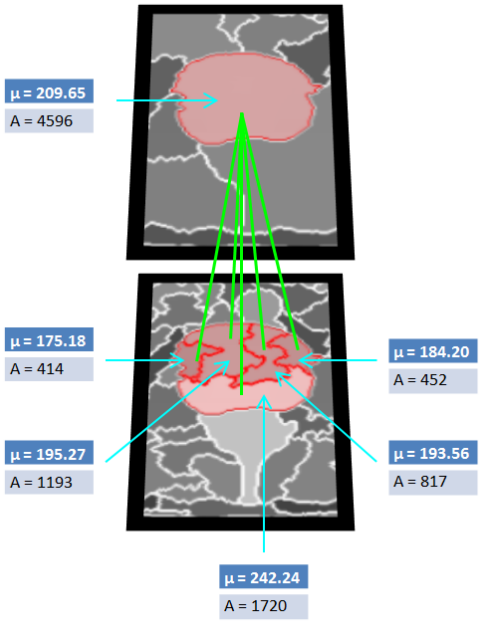


Figure 3: Since higher-level nodes and edges are nothing more than aggregates of their descendants, properties must be chosen in such a way as to ensure that they can be aggregated. In this node-focused example, the area $A(p)$ of the parent node p is calculated by simple summation of the areas of its children ($A(p) = \sum_c A(c)$) and its mean $\mu(p)$ is calculated using a normalised, weighted sum ($\mu(p) = \sum_c \mu(c)A(c) / \sum_c A(c)$). Note that the presence of the area property is essential to allow the parent's mean to be calculated.

2.2 Hierarchy Construction

The question of how to construct an appropriate hierarchy for a base graph is essentially a formulation of the hierarchical segmentation problem. Segmentation, the problem of how to decompose an entity into pieces, of which some or all have semantic meaning in a given domain, is a fundamental research topic that has spawned a variety of large sub-fields (e.g. image segmentation [40, 42] and mesh segmentation [44]). Hierarchical segmentation is the version of this problem whereby instead of simply decomposing an entity into a flat set of pieces, those pieces are then recursively subdivided into ever smaller pieces (alternatively, the constituent parts of the entity are initially grouped into small pieces, and then those pieces are grouped into ever larger pieces). Techniques for hierarchical segmentation can be divided into two categories: *agglomerative* (merge-based) and *divisive* (split-based).

2.2.1 Agglomerative Techniques

Agglomerative techniques [4, 6, 12, 15, 17, 43] work by recursively grouping the nodes of the base graph together, starting from the individual nodes. This can produce either a complete or an incomplete graph hierarchy, depending on the method used. For example, Beucher's waterfall technique [6, 21], which was originally designed for image segmentation, conceptually performs a watershed transform [7] on its input image to produce an initial partition of the image, and then generates higher-level partitions of the image by iteratively running a watershed transform on the most recent partition until convergence. This produces a complete graph hierarchy. By contrast, Felzenszwalb's algorithm [12] works by iteratively merging pairs of image regions whose mutual difference is small compared to the amounts by which they differ internally, thereby producing an incomplete graph hierarchy. The hierarchical face clustering techniques of Garland et al. [17] and Attene et al. [4] iteratively merge pairs of faces in a 3D mesh using dual edge contraction and also produce incomplete graph hierarchies as a result.

2.2.2 Divisive Techniques

Divisive techniques [30, 46, 47, 50, 51] work by recursively splitting the graph into pieces. Again, this can produce either a complete or an incomplete graph hierarchy, in this case depending on whether or not the splitting process is completely balanced; however, it usually produces incomplete graph hierarchies. For example, binary cut techniques such as the normalized cuts approach of Shi and Malik [46] define a measure of the cost of cutting a graph into two in different ways, and then choose a cut that minimises this cost. This can be applied recursively to construct an incomplete, binary graph hierarchy. Cut techniques can also produce k-way cuts at each stage: for example, the k-way fuzzy mesh decomposition technique described by Katz and Tal [30] decomposes a face-based submesh by picking a number of 'representative' faces in the submesh that are as far away from each other as possible and fuzzily assigning the other faces to the representatives based on distance. Leaving aside the fuzziness, it essentially produces an incomplete graph hierarchy. The DHSCAN network clustering algorithm of Yuruk et al. [50] works by clustering nodes based on the extent to which they share neighbours in the graph and also produces an incomplete graph hierarchy.

2.3 Hierarchy Editing

As mentioned in §1, graph hierarchy editing has generally received far less research attention than hierarchy construction over the years. However, a number of editing techniques can still be found in the literature [2, 3, 5, 10, 14, 18, 19, 23, 33, 39]. These can be roughly divided into local and non-local techniques, as we discuss in the following subsections.

There are at least three major reasons to want to edit graph hierarchies. The first reason is to refine the results of a hierarchical segmentation method, e.g. by introducing domain-specific knowledge. Despite the extensive research attention devoted to segmentation in a variety of fields, it remains in many ways a problem for which the desired results are often dependent on information that may be hard to provide to an automated algorithm. As such, it is often a good idea to allow for the possibility of manual refinement after the fact. The second reason is to visualise in advance the effects of costly or difficult-to-reverse changes to a real-world hierarchy – e.g. politicians might want to see the effects of changing local political boundaries on a map. The third reason is to gain insight into the structure of a network – this is the motivation behind techniques such as those of Auber and Jourdan [5] and Archambault et al. [3].

2.3.1 Local Techniques

Local editing techniques make only minor changes to the hierarchy. Common, simple operations include grouping a number of sibling nodes under a new node in the hierarchy or ungrouping them again, e.g. see the grouping operations in [5], the cluster creation and deletion operations in [10] or the (confusingly-named) node merging and node splitting operations in [14]. Other simple operations include merging a set of sibling nodes and replacing them with a new node, or splitting a single node into connected pieces [23]. A more interesting local technique can be found in [39], in which Nacken describes a method called 'relinking' to change the parent of an existing node (a similar technique also appeared in [23]). Although local, when applied to spatially-connected graph hierarchies this operation requires non-local checking to determine whether or not the planned change of parent is valid – in particular, we have to check not only that the node being moved is adjacent to its new parent, but that the move will not cause *any* of its old ancestors to become spatially disconnected. We will see in §4.2 that it is possible to implement a more general, non-local variant of this technique using our zipping algorithms.

2.3.2 Non-Local Techniques

Non-local editing techniques make more significant changes to the hierarchy, generally with a high-level goal in mind. For example, Glantz and Kropatsch [19] build on Nacken's work to perform multiple relinking operations to obtain a specific pattern in the lowest hierarchy layer.

Another non-local technique can be found in GrouseFlocks [2], a system for visualising large graphs by interactively creating spatially-connected hierarchies above them. The idea is to avoid visual clutter by hiding the contents of most of the nodes in the hierarchy in order to show a

relatively high-level partition (known as a 'cut') of the base graph. The nodes in the hierarchy have attributes, and the system supports attribute-based node selection (e.g. nodes in the authors' movie dataset have attributes such as 'genre', and the user can make selections such as 'action' vs. 'non-action' movies, or divide all of the movies into categories based on their genre). Since it is rarely the case that all of the nodes in a particular category will be in the same part of the hierarchy, the authors allow the user to recreate the hierarchy below the cut so as to group together nodes with the same attributes. The algorithm they describe, known as 'Reform-Below-Cut', works by deleting the intermediate hierarchy between the cut nodes and the leaves, and then grouping the leaves together again, based on the current selection and in a connectivity-respecting way, to create a new hierarchy below the cut. The goal is to group together spatially-connected nodes with similar attributes that are inconveniently in different parts of the hierarchy, and in this the algorithm shares something in common with the non-sibling node merging algorithm we describe in §4.1. However, there are a number of key differences between the two, most notably that Reform-Below-Cut destroys the intermediate hierarchy before starting again to create new structure, whereas non-sibling node merging preserves the existing intermediate hierarchy as much as possible.

The same authors also describe a further interesting technique called TugGraph [3], whose purpose is to modify a hierarchy in such a way as to 'separate out' the parts of the base graph that are adjacent to a selected cut node in the hierarchy (this can be a helpful tool when exploring the topological structure of a network). We defer a detailed discussion of this algorithm until §4.3, in which we show how to reformulate key steps of the algorithm in terms of our zipping algorithms to make it easier to understand.

2.3.3 Other Techniques

In addition to the techniques mentioned above, two other approaches that do not fit easily into either of the above categories deserve mentioning. Gerstmayer et al. [18] describe two editing operations, one of which marks pairs of regions that should be merged and the other of which marks individual regions that should be inhibited from merging. These operations differ from those mentioned above in that they 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 §3. The split and merge operations of Klava and Hirata [33] are also interesting, but are fundamentally different from the type of technique described above in that they operate on a cut through the hierarchy rather than modifying the hierarchy itself.

3 Zipping Algorithms

As highlighted in §2.3, the need to maintain the invariant that all nodes must represent a spatially-connected portion of the base graph can complicate algorithms that would otherwise be conceptually simple, e.g. Nacken's relinking technique [39], and make more involved algorithms like TugGraph [3] significantly harder to develop and understand. In this and the following sections, we first present our zipping algorithms and then show how they can help alleviate these difficulties by serving as building blocks for making potentially large changes to the hierarchy, whilst simultaneously preserving spatial connectivity constraints. Our *unzipping* algorithms effect multi-level splits in the hierarchy, whilst our *chain zipping* algorithm effects a multi-level merge.

3.1 Unzipping

In its simplest form, unzipping is a technique that allows an individual node to be 'separated out' from some or all of the nodes above it in the hierarchy. 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 defined to be the connected components of what is left of the ancestor after removing the node being unzipped from consideration. A more general form of unzipping is also possible, whereby multiple nodes are unzipped simultaneously – this form will prove useful for reformulating TugGraph in §4.3.

Listing 1 Single-Node Unzipping

```
function unzip_node : (node: NodeID; toDepth: Int) → Vector<Chain>

  var chains: Vector<Chain> := [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_ccs(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);

    // Update the chains.
    for each chain: Chain ∈ chains
      var p: NodeID := parent_of(chain.front());
      chain.push_front(p);
      result.erase(p);

    for each n: NodeID ∈ result
      chains.push_back([n]);

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

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

3.1.1 Single-Node Unzipping

The single-node unzipping algorithm takes as input the node to be unzipped and a depth to which to unzip it. It returns an array of chains that could in theory be zipped together in order to undo the operation.[1] A *chain* is a sequence of nodes $[n_1, ..., n_k]$, where $\forall i \in [1, k) \cdot n_i = parent(n_{i+1})$. It should be noted that the chains returned by unzipping are not necessarily all the same length, but their highest nodes are all at the same depth in the hierarchy.

The implementation of single-node unzipping is shown in Listing 1. As the algorithm progresses, we keep track of node chains that represent the strands being unzipped: these will be what is eventually returned as the result. In each iteration of the loop, we keep track of a 'current' node, initially the input node. We determine the siblings of this node (defined to be the other children of its parent) and calculate their connected components. To these we add a singleton component containing the current node itself. We then split the parent of the current node into the specified components, update the aforementioned chains and repeat the process with the parent of the current node. The loop terminates when we reach the user-specified target depth.

The construction of the chains is the most interesting part of the algorithm. Initially, the array of chains is initialised with a singleton chain containing only the node being unzipped.[2] During each iteration of the loop, each node resulting from that iteration's split node operation will either be used to augment an existing chain, or to add a new one: specifically, each existing chain is prepended with the parent of its head node, which is then removed from the set of nodes resulting from the split (it is guaranteed to be present because of the way the algorithm works); the remaining nodes resulting from the split are then used to create new singleton chains.

An example of single-node unzipping is shown in Figure 4, in which the node circled in green is progressively unzipped to the depth of the nodes circled in red.

[1] In practice, if primitive algorithms such as split and merge are implemented as undoable commands (e.g. see [16, 25]) then both unzipping and zipping can be implemented as composite commands, allowing them to be undone more easily by the simple expedient of undoing their constituent operations in reverse order.

[2] This is an implementation trick that ensures that the chain leading down to the node being unzipped is the first chain in the array: this allows it to be easily identified by higher-level algorithms that need ready access to it (they would otherwise have to search through all the chains to find it). The final chain should not contain the actual node being unzipped (only the nodes above it), so in practice we remove it from the end of the chain again before returning.

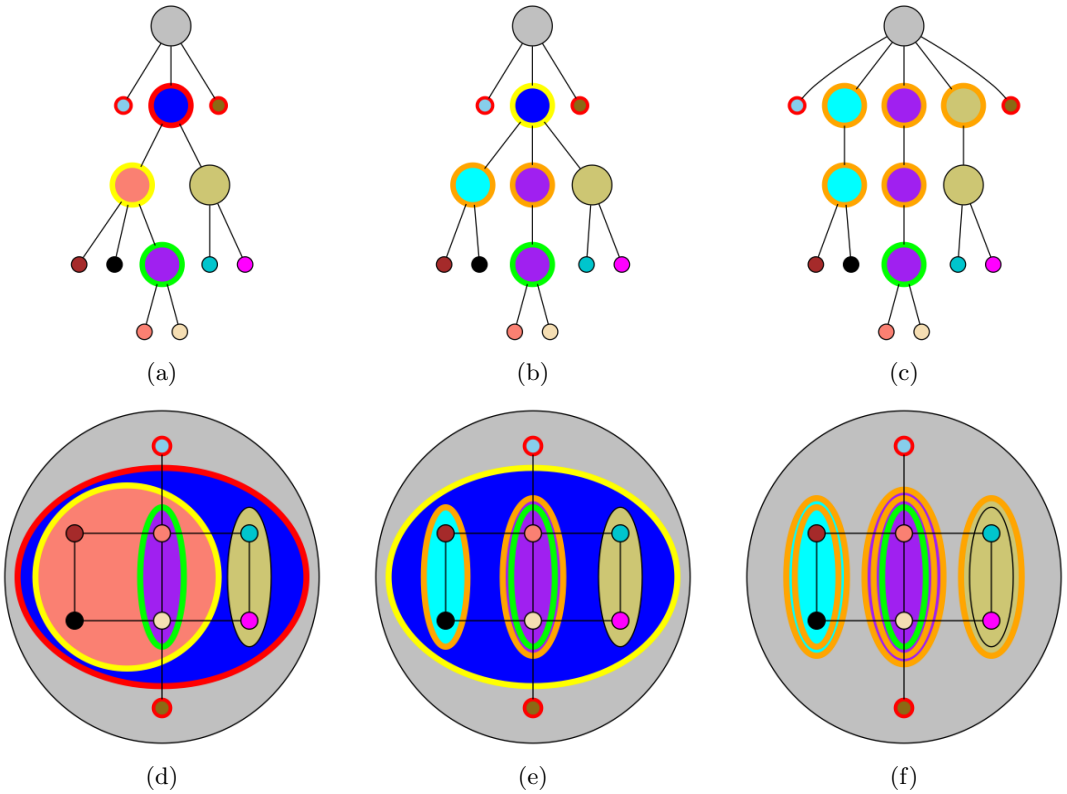


Figure 4: Using the single-node unzipping algorithm to unzip the node circled in green to the depth of the nodes circled in red: in (a) and (d), the parent node (circled in yellow) of the node being unzipped is considered for splitting; in (b) and (e), the parent is split into two nodes, one corresponding 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 (the first chain also contains the node circled in green during the algorithm itself).

3.1.2 Multi-Node Unzipping

Whilst it is possible to use the single-node unzipping algorithm described in the previous section repeatedly to unzip a set of nodes one at a time, this has a couple of potential drawbacks:

1. It can involve splitting the same ancestor nodes multiple times – this can cause us to perform a lot of unnecessary work in repeatedly finding similar connected components.
2. It forces the nodes in the set to be separated out individually from the rest of the hierarchy. If we want to separate out several connected nodes into a single chain, we have to separate each node out individually and then zip their chains together again.

For these reasons, it is helpful to extend the single-node approach to one that allows us to unzip multiple nodes at the same time. Our new algorithm will allow any set of nodes to be unzipped to a specified depth, provided that none of the selected nodes is a descendant of another. An easy way to ensure this latter constraint is to use a multi-level node selection data structure of the form we described in [20, 24] – whilst we do not revisit the implementation details here, the point of the data structure is to represent the selection of a set of nodes in the base graph using as few hierarchy nodes as possible (with a preference for those that are higher up the hierarchy when there is a choice). Since selecting a node implies the selection of all of its descendants, maintaining a minimal set of nodes in the representation automatically enforces the constraint that no node in the representation is a descendant of another.

Listing 2 Multi-Node Unzipping

```
function unzip_selection : (sel: Selection; toDepth: Int) → Map<NodeID,Chain>

  var chains: Map<NodeID,Chain>;

  // Unzip the nodes, starting with those lowest in the hierarchy.
  var nodesByDepth: Map<Int,Set<NodeID>> := sel.nodes_by_depth();
  var depth: Int := nodesByDepth.max_key();
  var curs: Set<NodeID> := nodesByDepth[depth];
  while depth ≠ toDepth
    // Determine which parent nodes will need to be split.
    var parentToSelectedChildMap: Map<NodeID,Set<NodeID>>;
    for cur ∈ 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_ccs(siblings);
      ccs.append(find_ccs(selectedChildren));
      result.append(split_node(parent, ccs));

    // Update the chains.
    for each chain: Chain ∈ chains
      var p: NodeID := parent_of(chain.front());
      chain.push_front(p);
      result.erase(p);

    for each n: NodeID ∈ result
      chains.insert(n, [n]);

    // Update the set of current nodes.
    curs := parents_of(curs);
    --depth;
    curs.append(nodesByDepth[depth]);

  return chains;
```

As shown in Listing 2, the multi-node unzipping algorithm takes as input a selection of nodes of the form just described and a depth to which to unzip them. In place of the array of chains returned by single-node unzipping, it returns a map of chains, keyed by the lowest nodes in the various chains – this is in order to make it easy to look up the chain corresponding to any one of the nodes being unzipped (the lookup can be performed by finding the chain corresponding to the new parent of the node in question). The algorithm itself works in quite a similar way to single-node unzipping, but with the following key differences:

1. Rather than maintaining a single current node, we maintain a set of them.
2. Since the individual nodes being unzipped can be at multiple levels of the hierarchy, we start with those deepest in the hierarchy first and only add in nodes higher up as the unzip reaches their depth (the introduction of higher-up nodes happens at the end of the main loop).
3. Since the current nodes in any given iteration of the unzip often have parents in common, we start each iteration by grouping the current nodes according to their parents. Each of the unique parents is then split. The splitting process first determines the connected components of those child nodes that are not in the current node set. It then augments these with the connected components of the children that are in the current node set before splitting the parent node into these components. Note that this produces a different result from that which would be obtained by multiple single-node unzips, since the current nodes at each stage are grouped together into components where possible.

An example of multi-node unzipping is shown in Figure 5, in which the green leaf nodes are unzipped to the depth of the nodes circled in red.

3.1.3 Chain Compression

The unzipping algorithms just described are *depth-preserving*: that is, they separate out branches of the hierarchy, but they do not change the depths of the nodes in the hierarchy. This is important in

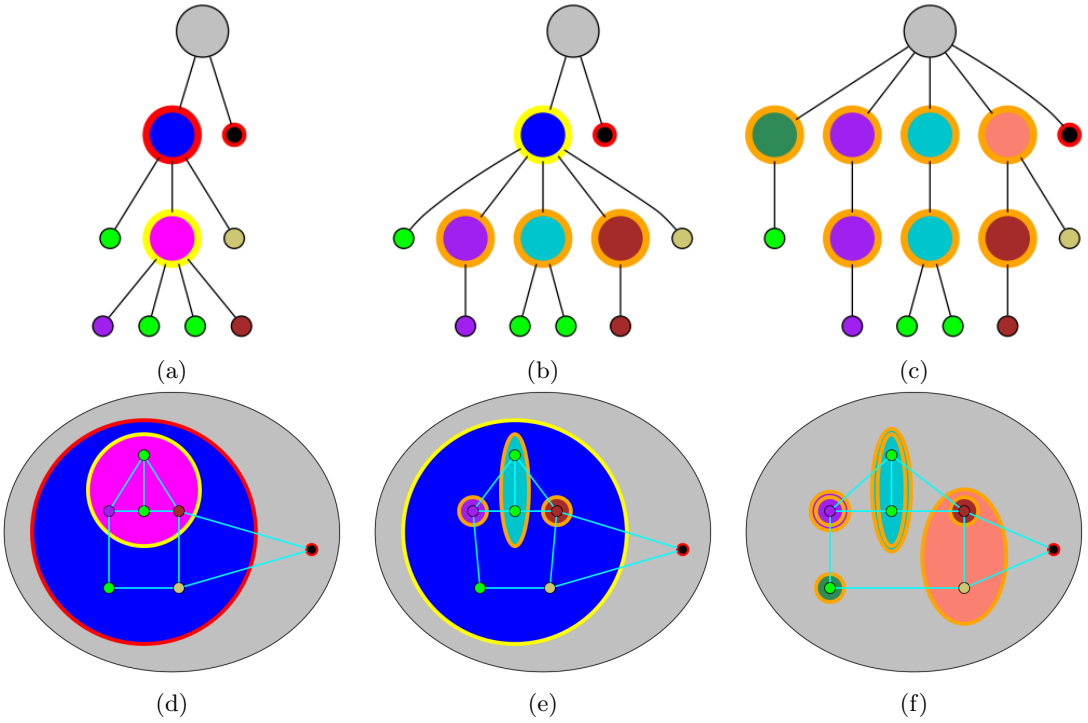


Figure 5: Using the multi-node unzipping algorithm to unzip the three green leaves to the depth of the nodes circled in red: (a) and (d) show the initial state of the graph hierarchy; (b) and (e) show it after the first unzipping iteration; (c) and (f) show its final state.

some contexts – in particular, it allows the algorithms to be applied to complete graph hierarchies, in which nodes cannot arbitrarily be created or deleted. It is also important if the resulting chains are to be zipped with other chains. However, when working with incomplete graph hierarchies, it can also result in chains of nodes in which each node aggregates the same portion of the base graph, and in some situations (e.g. when reformulating TugGraph in §4.3) it proves useful to remove some of the intermediate nodes to avoid this. See Figure 6 for an example.

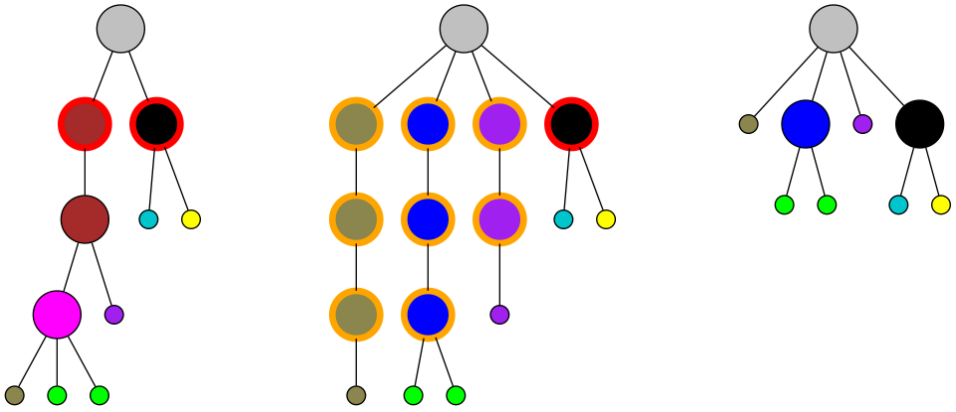
We call the process of removing intermediate nodes *chain compression*. This can be implemented very simply, as shown in Listing 3, by iterating over the input chains (e.g. those generated by an unzip) and removing any node that has only a single child.

3.2 Chain Zipping

Chain zipping is the inverse of the single-node unzipping algorithm described in the previous section: it takes a set of node chains whose highest nodes are all at the same depth in the hierarchy and zips them together using a sequence of sibling node merges. The algorithm takes as input an array of chains (not necessarily all of the same length) and merges together the corresponding nodes from the different chains from the top of the hierarchy downwards.

The implementation of chain zipping (see Listing 4) is straightforward, but there are a number of important preconditions that must be checked before the main operation in order to avoid invalid zips:

1. There must be chains to zip.
2. Each chain must be non-empty.
3. No chain may extend down as far as a leaf of the hierarchy – this prevents any leaves being merged. The merging of leaves should be avoided because they correspond to atomic objects in the problem domain (e.g. pixels in the case of image segmentation).
4. The highest nodes in the chains must be siblings of each other.
5. The sets of nodes to be merged at each depth must be connected.



(a) (b) (c)

Figure 6: An example of chain compression: unzipping the green leaves in (a) to the depth of the nodes circled in red results in chains of nodes (highlighted with an orange circle in (b)), in which each node aggregates the same portion of the base graph. Chain compression removes the intermediate nodes, resulting in a smaller hierarchy (c). However, it cannot be used in the context of complete graph hierarchies, and can interfere with later zipping.

Listing 3 Chain Compression

```
function compress_chains : (chains: Vector<Chain>) → ∅

  for chain: Chain ∈ chains
    for i := chain.size() - 1 down to 0
      var n: NodeID := chain[i];
      if |children_of(n)| = 1 then
        // Remove n, making its child directly descend from its parent.
        ungroup(n);
```

Listing 4 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: NodeID := 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]);
```

The sets of nodes to be merged at each depth are extracted from the chains as part of the process of checking that they are connected. For valid zips, the nodes in each set are then merged, starting with the nodes in the highest set and working downwards.

We do not provide an explicit example of chain zipping, but two implicit examples can be seen by considering Figures 4 and 5 in reverse.

4 Applications

The zipping algorithms described in §3 are broadly applicable (as previously stated, they are essentially multi-level equivalents of node splitting and node merging) and can act as useful building blocks in a wide variety of hierarchy-editing scenarios. In this section, we demonstrate their usefulness by showing how to apply them to three very different editing problems.

The first application we consider is *non-sibling node merging*, the problem of merging nodes in a graph hierarchy when they do not share a common parent. This is a useful operation in an interactive context because it enables users to merge entities that are adjacent in space in an intuitive way, without having to worry about the structure of the hierarchy. (For example, it proved particularly useful in our implementation of the user interface for *millipede* [22], the tool we developed for 3D hierarchical image segmentation, because it allowed users to merge adjacent image regions without reference to the hierarchy.) We show that non-sibling node merging can be implemented relatively straightforwardly in terms of our zipping algorithms, but would be much harder to implement without them. We further show that non-sibling node merging is needed for a significant proportion of the potential merges that a user might want to perform, and considerably reduces the interactivity burden on the user for individual merges. Finally, we compare non-sibling node merging to the Reform-Below-Cut algorithm described in [2].

The second application 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 in [39], but the algorithm described there 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 allows a node's parent to be changed in a wider range of scenarios. We compare this new algorithm to Nacken's approach to highlight the differences.

As a third application of our zipping algorithms, we consider reformulating the TugGraph approach of Archambault et al. [3] for exploring the portion of a graph hierarchy around a particular node. Given a node of interest, TugGraph works by 'tugging out' the portions of the hierarchy's base graph that are directly connected to the node from their containing nodes in the hierarchy, thus providing a user with an easy way to visualise the neighbourhood of a node. TugGraph is an effective algorithm, but its implementation is comparatively complex due to the need to preserve spatial connectivity constraints. We show that key parts of the algorithm can be reformulated using our zipping algorithms and that doing so makes the resulting implementation easier to understand.

4.1 Non-Sibling Node Merging

When editing a graph 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: an example can be seen in Figure 7, 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 sibling node merging, but is much easier to achieve using our zipping algorithms.

The algorithm we present, which we call *non-sibling node merging*, works by unzipping the nodes to be merged up to their common ancestor and then zipping together the resulting chains to effect the final merge. To make it slightly more general, it takes as input a set of arbitrary,

non-leaf nodes that share a common depth in the hierarchy, divides them into their connected components and then merges the nodes in each component individually. Since each component is connected, the node resulting from each merge represents a contiguous portion of the base graph. The algorithm returns the set of merged nodes as its result (i.e. there is one result node for each connected component of the input).

The implementation of non-sibling node merging is shown in Listing 5. The first step is to calculate the connected components of the input nodes: each connected component of size > 1 will be merged into one result node. To perform the merge for a connected component, we first determine the common ancestor of the component's nodes. We then unzip each node up to the common ancestor, keeping track of the chain of nodes leading back down to the node being unzipped and appending the actual node being unzipped to the chain in each case (it would not otherwise be present). Finally, all of these chains are zipped together to merge the nodes in the connected component.

A worked example of non-sibling node merging is shown in Figure 8, in which the goal is to perform a non-sibling merge on all of the nodes circled in green (a,d). These are initially divided into their connected components, before the nodes of each component are unzipped up to their common ancestors (b,e). The chains for each component are then zipped together to complete the merge (c,f).

4.1.1 Analysis

In order to demonstrate the benefits of non-sibling node merging to users, we examine a concrete implementation of it in *millipede* [22], a tool that we implemented to segment 3D abdominal CT images. Given such a 3D image, *millipede* first smoothes it using anisotropic diffusion filtering [41], and then uses Meijster and Roerdink's watershed method [38] and Beucher's waterfall technique [6, 21, 36] to construct a type of complete graph hierarchy that we call an *image partition forest*, or IPF, which contains partitions of the image at different scales [20]. These partitions can be visualised using 2D slices aligned with each of the principal axes (X-Y, X-Z and Y-Z), as shown in Figure 9 (the software also supports feature identification and the visualisation of the resulting features in 3D). We use *millipede* to demonstrate empirically that non-sibling node merging is needed for a significant proportion of the potential merges that a user might want to perform in such a graph hierarchy, and that it considerably reduces the interactivity burden on the user for individual merges.

To perform our analysis, we modified the *millipede* software to calculate the number of levels of unzipping that would be required to merge each adjacent pair of nodes in an IPF. (Nodes with the same parent require no unzipping, and can thus be merged by the sibling node merge algorithm; nodes whose parents differ must be unzipped up to their common ancestor as part of a non-sibling node merge.) We calculated how many levels of zipping were required for all adjacent node pairs in IPFs constructed from the slices of three 3D abdominal CT datasets (BT, MC and SD, all provided by the Churchill Hospital, Oxford).[3] To construct the IPFs, we first selected a representative set of slices from the middle of each dataset to obtain 3D volumes of size $512 \times 512 \times 11$ (we experimented with using more slices, but found that this made little difference to the results whilst increasing the computation time). Each volume was then processed using the pipeline described above to construct an IPF containing 5 branch layers in addition to its leaf layer. The slices chosen for each dataset, together with the values of the parameters we used to control our pipeline, are shown in Figure 10.

The results are shown in Figure 11. We observe that for each IPF, there was a high cumulative percentage ($> 70\%$) of node pairs that required at least one level of unzipping to merge (i.e that involved *non-sibling* node merging), and a relatively high cumulative percentage ($\approx 50\%$) of node pairs that required at least two levels of unzipping to merge. Thus, not only do the majority of the potential merges that the user might request involve non-sibling node merging, they often involve changes that require multiple levels of unzipping. This is costly in multiple ways: each individual node split that is required as part of the unzip operation has relatively high input requirements (since 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 to the IPF without an easy way to visualise the parent-child links carries a high cognitive overhead. Whilst we could

[3] We are unfortunately not able to make these datasets publicly available for reasons of patient confidentiality.

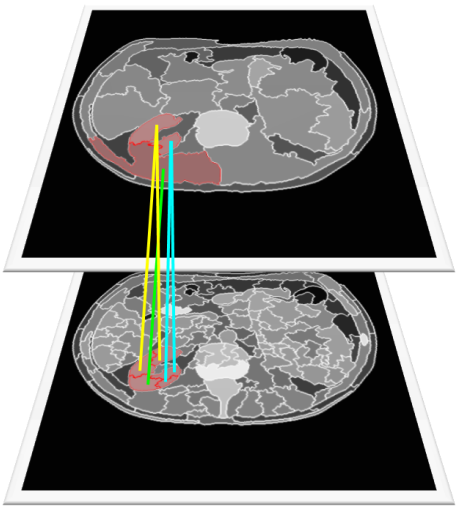


Figure 7: The need for non-sibling node merging: it is intuitive to want to merge nodes that are connected spatially, but this cannot be done straightforwardly if they are not also siblings in the hierarchy. For example, the user might like to merge the five highlighted regions in the lower layer in this figure 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.

Listing 5 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: Vector<Set<NodeID>> := find_ccs(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: NodeID := 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;
```

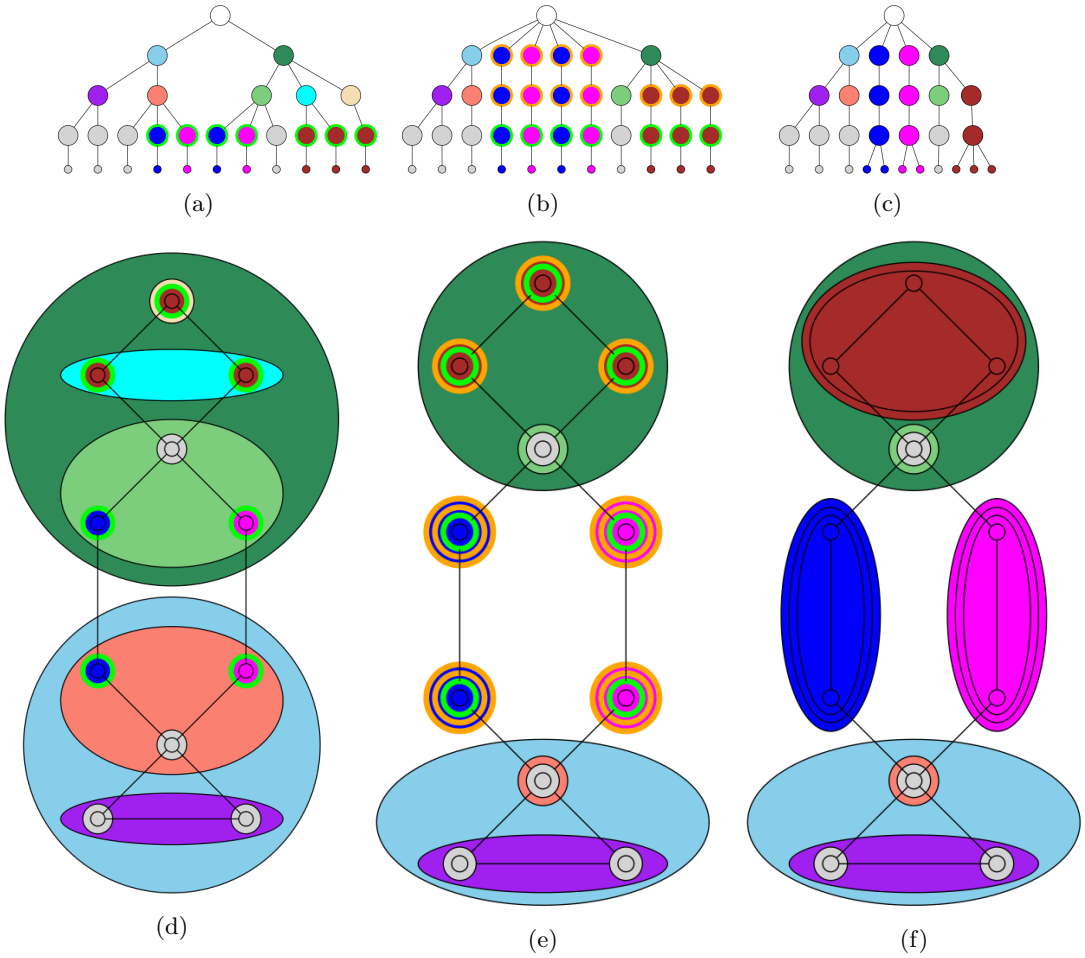


Figure 8: An example of non-sibling node merging: (a) and (d) show the initial state, in which we are about to perform a non-sibling merge on three connected components (blue, magenta and brown); (b) and (e) show the state after the nodes in each component have been unzipped up to their common ancestors; (c) and (f) show the final state.

circumvent this problem by simply restricting the user to sibling-only node merging, this would prohibit many useful merges. This motivates the development of a non-sibling node merging algorithm of the kind described in this paper: with such an algorithm available, each pairwise node merge becomes a simple operation in the UI (select the two nodes and click to merge).

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.

4.1.2 Comparison with Reform-Below-Cut

As observed in §2.3, our non-sibling node merging algorithm shares a common goal with the Reform-Below-Cut algorithm described by Archambault et al. [2] in trying to allow the user to group nodes that are connected spatially but that are not siblings in the hierarchy. However, the two algorithms transform the hierarchy in significantly different ways: in particular, Reform-Below-Cut groups leaves with similar attributes below a 'cut' through the hierarchy by deleting the branch nodes between the leaves and the cut and then creating new branch nodes containing the leaves to be grouped, whereas non-sibling node merging groups nodes in a fixed, non-leaf layer of the hierarchy together and avoids changing the hierarchy above those nodes to the greatest extent possible.

The reason for this difference in behaviour is that the two algorithms work in rather different

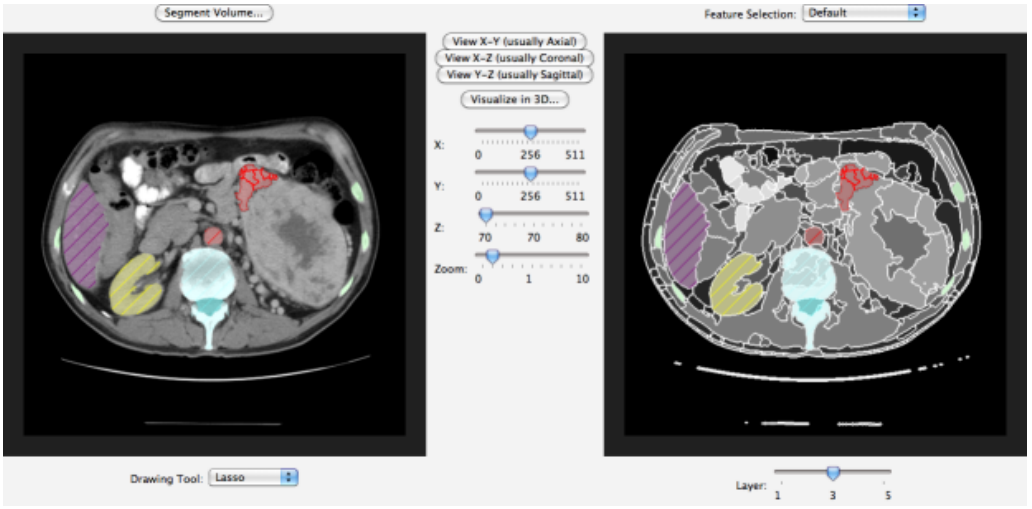


Figure 9: The segmentation window of *millipede*, showing an axial (X-Y) slice through a 3D abdominal CT image (left) and an axial slice through the 3rd layer of the corresponding (complete) graph hierarchy (right). The UI allows the user to view the image and the hierarchy from different directions (top, front and side), and to move around in space and switch between the various different layers of the hierarchy using sliders. Important features such as major organs can be identified using menus (not shown) or keyboard shortcuts. The organ identification results can also be visualised in 3D using the Multiple Material Marching Cubes (M3C) algorithm [49].

Dataset	Slices
BT	70–80
MC	110–120
SD	70–80

(a)

Parameter	Value
ADF Conductance	1.0
ADF Iterations	20
CT Window Level	40
CT Window Width	400
Waterfall Layers	5
Waterfall Method	Simplified Waterfall [21]

(b)

Figure 10: (a) The slices chosen from the 3 datasets we used for our experiments; (b) the parameter settings we used when constructing IPFs for those slices. The CT window level and width control how the images in the dataset are converted from Hounsfield units to greyscale (see [20] for further details). The ADF parameters control the anisotropic diffusion filtering process [41].

contexts. The *GrouseFlocks* system developed by Archambault et al. was designed with the primary aim of conveniently visualising a large graph, a context in which branch nodes in the hierarchy only really exist to temporarily group the base graph nodes for visualisation purposes and can be discarded with relative impunity in order to visualise different aspects of the base graph. By contrast, the goal of our *millipede* system is to produce a hierarchy in which some of the branch nodes will correspond to interesting features in the base image for identification purposes, a context in which the intermediate hierarchy may be of interest and should not be discarded.

4.2 Parent Switching

As discussed in §2.2, graph hierarchy construction techniques generally work by either recursively merging a graph's constituent nodes into larger ones, or by recursively splitting 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 graph hierarchies to contain nodes that have been inappropriately merged into the wrong parent. An example of this can be seen in Figure 7, in which the

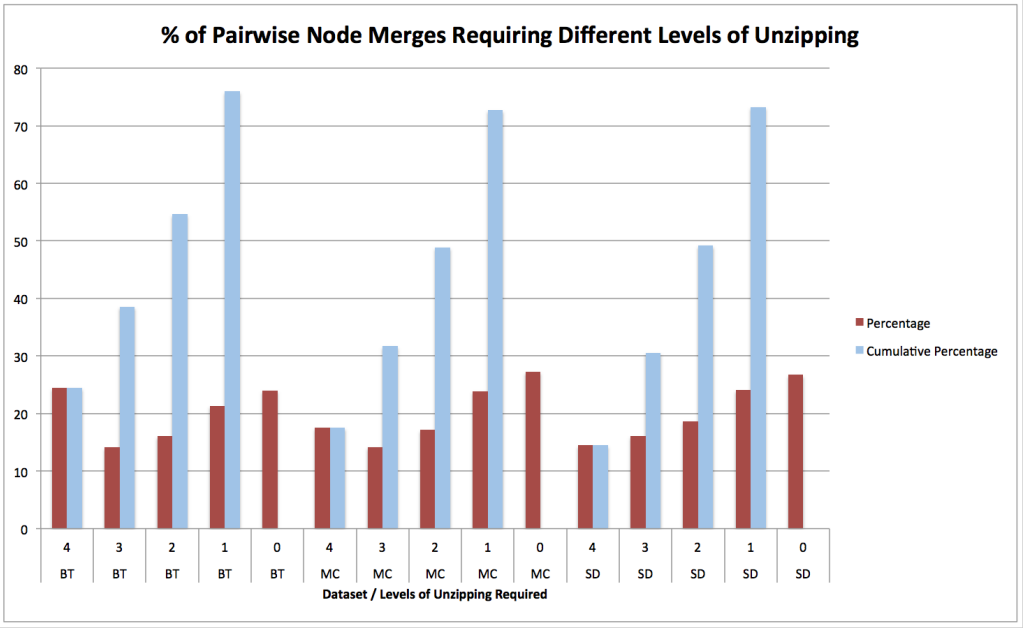


Figure 11: The percentages of pairwise merges of nodes in various IPFs (all with 5 branch layers) that required different levels of unzipping (see §4.1.1).

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 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 [39] observed, 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 works by non-locally checking both requirements and then performing a trivial update of the parent pointer of the node being moved if the move can be resolved locally. This is an entirely reasonable solution to the problem he was solving, but is actually quite restrictive as a solution to the more general problem we deal with here: in particular, 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. Once again, our zipping algorithms provide us with a convenient way of arranging for this to happen.

The algorithm we describe here, which we call *parent switching*, first checks that the node to be moved is adjacent to at least one child of its proposed new parent. Provided this is the case, it then walks up the hierarchy from both the old and new parents of the node being moved to find their common ancestor, keeping track of the chain of nodes leading down to the new parent in the process. It then unzips the node being moved up to this common ancestor, which ensures that all relevant ancestors of the node being moved are correctly split. Finally, it zips the chain leading from the common ancestor down to the node being moved to the chain leading down to the new parent, which has the desired effect of making the node being moved a child of its new parent. The implementation is as shown in Listing 6. A worked example is shown in Figure 12.

4.2.1 Analysis

As for non-sibling node merging, we analyse our parent switching approach using *millipede*, in this case by demonstrating 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 [39]. This allows the user considerably more flexibility in rearranging graph hierarchies, and isolates them from the need to understand the structure of the hierarchy above

Listing 6 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 := 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]);
```

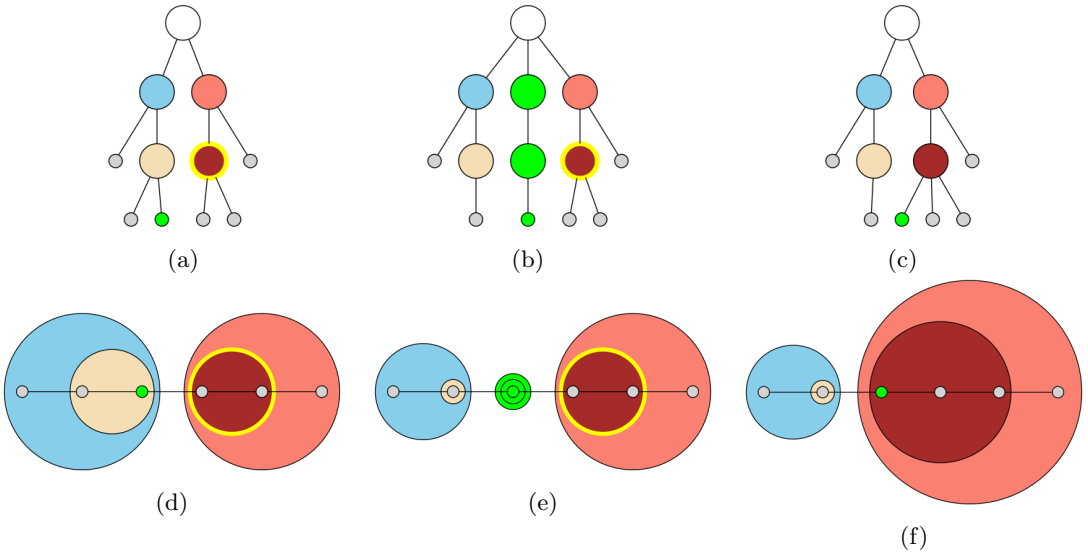


Figure 12: An example of parent switching: in (a), the node to be moved is shown in green and its new parent is circled in yellow; in (b), the node is unzipped up to the common ancestor of its old and new parents; in (c), the node's chain is zipped to the chain leading down to its new parent to complete the switch.

the node being moved.

To perform our analysis, we used the same IPFs that were constructed for the non-sibling node merging analysis (see §4.1.1). We first modified the *millipede* software to calculate all the possible parent switches in each IPF. These were determined 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 modified the software to 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

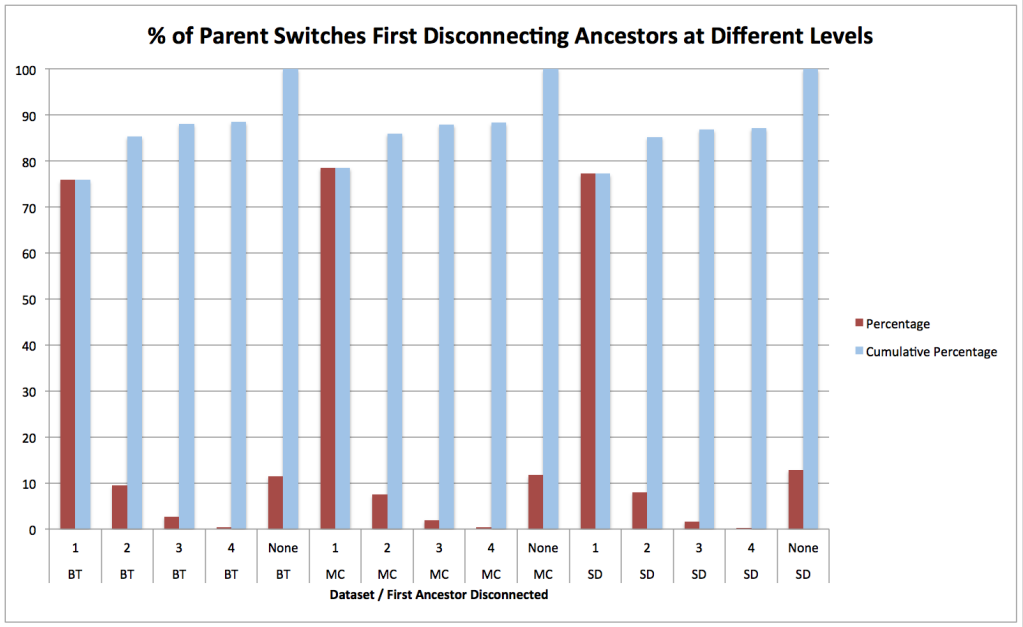


Figure 13: The percentages of possible parent switches in various IPFs (all with 5 branch layers) that first disconnect ancestors at different levels above the node being moved (see §4.2.1).

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 to be made.

The results are shown in Figure 13, and are consistent between the three datasets we tested. In each case, almost 90% 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 method. (This should not be interpreted as a criticism of Nacken's method, which worked well in its original context; however, it does illustrate the advantages of the method we describe here in our context.) Moreover, roughly 10% 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 why these switches are not possible, 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. Using our approach, we can simply show all the surrounding parents as possible targets of the operation and let our unzipping procedure handle the necessary changes to the hierarchy.

4.3 Reformulating TugGraph

As originally mentioned in §2.3, TugGraph [3] is a visualisation algorithm that can be used to modify a graph hierarchy in such a way as to 'separate out' the parts of the base graph that are adjacent to a selected node on the cut that is currently being viewed. The analogy is one of 'tugging' on the selected node in order to pull adjacent nodes that are currently hidden into view, thereby allowing a user to explore the topological structure around the node.

Two different versions of the algorithm were presented in [3]. The first, more easily understandable, version works as follows:

1. Traverse down the hierarchy from the selected node to find the leaves it subsumes.
2. Find the leaves adjacent to those leaves in the base graph (the 'adjacent leaves').

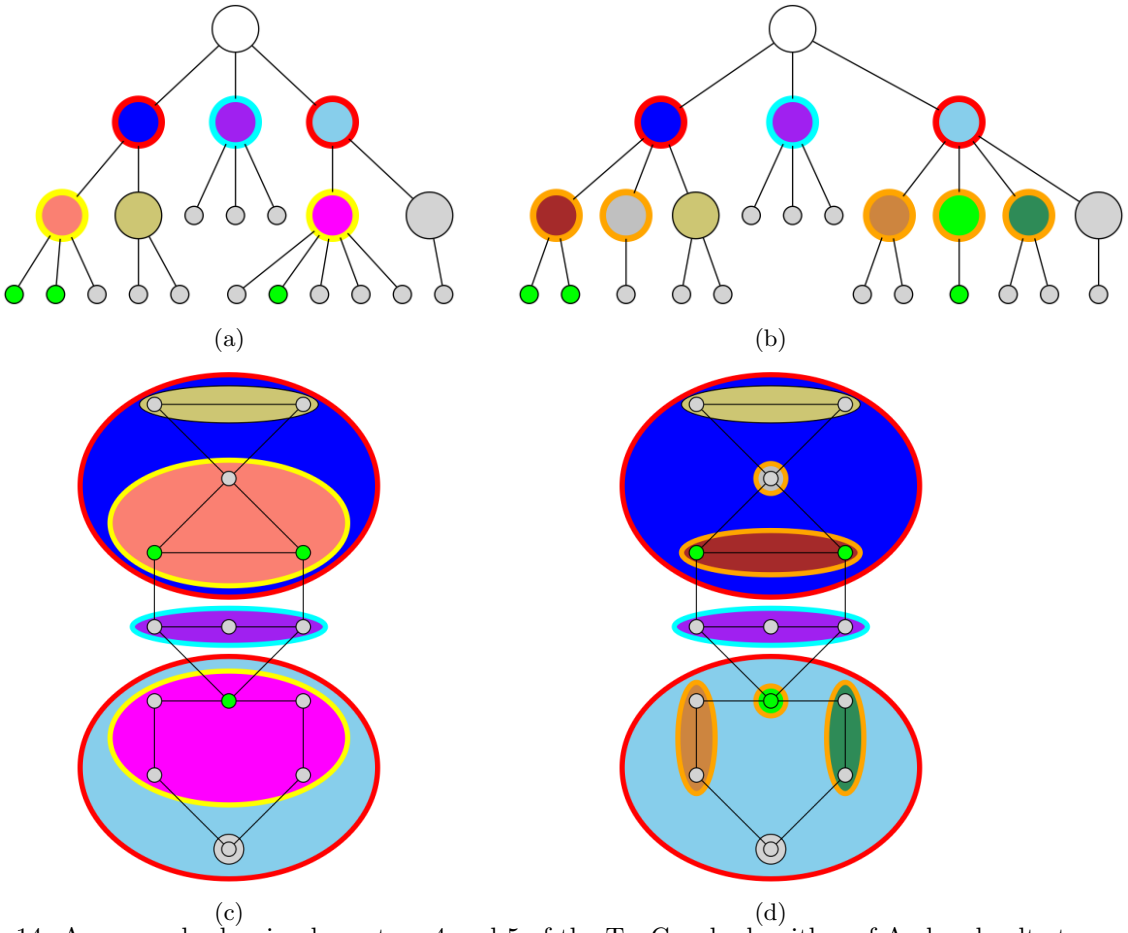


Figure 14: An example showing how steps 4 and 5 of the TugGraph algorithm of Archambault et al. [3] can be implemented using multi-node unzipping. In (a) and (c), we conceptually 'tug' on the node circled in cyan, with the intention of 'separating out' the adjacent leaves (green) within the adjacent cut nodes (red). The leaves below each adjacent cut node can be separated out using a multi-node unzip, with the result as shown in (b) and (d).

3. Traverse up the hierarchy from the adjacent leaves to find the nodes on the cut that are adjacent to the original selected node (the 'adjacent cut nodes').
4. For each adjacent cut node, make a copy of and then delete the intermediate hierarchy between it and the leaves it subsumes, and then group the adjacent leaves it now directly contains together based on their spatial connectivity in the base graph.
5. For each adjacent cut node, recreate the intermediate hierarchy between it and all other leaves it contains based on the previously-made copy.

This has the effect of creating new nodes, just below the cut, that together contain all the leaves adjacent to the originally-selected node, and the visualisation can subsequently be adjusted to show them. This is relatively straightforward, but quite inefficient – in particular, it has to recreate the intermediate hierarchy for all leaves within an adjacent cut node, even if they are not relevant to the node that has been selected. As a result, the authors present an improved second version of their algorithm that moves the adjacent leaves directly into nodes just below the cut, then works up the hierarchies below the adjacent nodes on the cut, splitting nodes and updating adjacency information as necessary using a priority queue-based implementation. This markedly improves the efficiency of the algorithm, but makes it harder to understand and more difficult to implement, due to the efforts that have to be made in correctly preserving the spatial connectivity constraints involved.

An alternative implementation can be formulated in terms of the multi-node unzipping algorithm we described in §3.1.2. Like the second approach described in [3], this implementation avoids

recreating parts of the intermediate hierarchy unnecessarily, but unlike that approach, it is also relatively easy to understand. The key insight is that we can rewrite steps 4 and 5 of the original approach as a number of multi-node unzips, one for each adjacent cut node. Specifically, if we jointly unzip the adjacent leaves below each adjacent cut node to a depth one greater than the depth of that node, we will produce an equivalent result to the original approaches, but in a straightforward and efficient way. An example of this process is shown in Figure 14.

5 Conclusions

In this paper, we have focused on the problem of editing graph hierarchies whilst preserving the connectivity constraint that every node must represent a connected portion of the graph on which the hierarchy is based. This is an important problem, both because graph hierarchies see widespread use as a way of representing the interactions between objects at multiple scales, and because there is no way of objectively defining the 'best' clustering of a set of objects for every problem – there is an inherent need to allow interactive modification of graph hierarchies after they have been constructed. However, due to the difficulties of preserving connectivity whilst making large hierarchy changes, most existing editing algorithms only allow local changes to a hierarchy. The few algorithms that do allow larger changes have to expend significant effort on connectivity preservation, complicating their implementation and making them harder to understand.

We believe that introducing the kind of multi-level split and merge algorithms we have presented in this paper offers a possible way of facilitating the development of future editing algorithms. Indeed, for the applications we have presented, we found that our zipping algorithms made it significantly easier than it otherwise would have been to design and reason about large-scale changes to graph hierarchies. For non-sibling node merging, our analysis using the *millipede* software system (see §4.1.1) indicated that in the real-world graph hierarchies we studied, a significant majority of the pairwise merge operations a user might wish to perform between spatially-adjacent nodes were between non-siblings, and would hence require split/sibling merge operations on multiple levels of the hierarchy to effect. Such non-sibling merges are much easier to effect using 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 analysis indicated that almost 90% of the switches that a user might wish to 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. Finally, our reformulation of TugGraph in §4.3 illustrated how our zipping algorithms can be used to simplify existing methods in the literature.

Acknowledgements

We are profoundly grateful to Dr. Zoe Traill of the Churchill Hospital, Oxford, both for providing us with abdominal CT scans for *millipede* and for giving up her time to help us interpret them. We also gratefully acknowledge the support of the UK Engineering and Physical Sciences Research Council (EPSRC) in funding Stuart Golodetz's doctoral work via a Doctoral Training Account (DTA).

References

[1] E. L. Andrade, E. Khan, J. C. Woods, and M. Ghanbari. Segmentation and Tracking using Region Adjacency Graphs, Picture Trees and Prior Information. In *IEEE Visual Information Engineering (VIE'03)*, pages 45–48, 2003.

[2] D. Archambault, T. Munzner, and D. Auber. GrouseFlocks: Steerable Exploration of Graph Hierarchy Space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 2008.

[3] D. Archambault, T. Munzner, and D. Auber. Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):276–289, 2011.

[4] M. Attene, B. Falcidieno, and M. Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22(3):181–193, 2006.

[5] D. Auber and F. Jourdan. Interactive Refinement of Multi-scale Network Clusterings. In *Proceedings of the Ninth International Conference on Information Visualisation*, pages 703–709, 2005.

[6] S. Beucher. Watershed, Hierarchical Segmentation and Waterfall Algorithm. In *Proceedings of ISMM'94*, pages 69–76, 1994.

[7] S. Beucher and C. Lantejoul. Use of Watersheds in Contour Detection. In *Proceedings of the International Workshop on Image Processing: Real-time Edge and Motion Detection/Estimation*, Rennes, France, September 1979.

[8] E. Borenstein and J. Malik. Shape Guided Object Segmentation. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR '06)*, pages 969–976, 2006.

[9] M. Dickheiser. Inexpensive Precomputed Pathfinding Using a Navigation Set Hierarchy. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 103–113. Charles River Media, 2004.

[10] P. Eades and M. L. Huang. Navigating Clustered Graphs using Force-Directed Methods. *Journal of Graph Algorithms and Applications*, 4(3):157–181, 2000.

[11] N. Elmqvist and J.-D. Fekete. Hierarchical Aggregation for Information Visualization: Overview, Techniques and Design Guidelines. *IEEE Transactions on Visualization and Computer Graphics*, 16(3):439–454, 2010.

[12] P. F. Felzenszwalb and D. P. Huttenlocher. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision*, 59(2):167–181, 2004.

[13] B. Fischer, C. Thies, M. O. Güld, and T. M. Lehmann. Content-Based Image Retrieval by Matching Hierarchical Attributed Region Adjacency Graphs. In *Proceedings of the SPIE*, volume 5370, pages 598–606, 2004.

[14] D. H. Fisher. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Learning*, 2(2):139–172, 1987.

[15] G. W. Flake, R. E. Tarjan, and K. Tsioutsiouliklis. Graph Clustering and Minimum Cut Trees. *Internet Mathematics*, 1(4):385–408, 2004.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[17] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical Face Clustering on Polygonal Surfaces. In *Proceedings of the 2001 Symposium on Interactive 3D Graphics (I3D)*, 2001.

[18] M. Gerstmayer, Y. Haxhimusa, and W. G. Kropatsch. Hierarchical Interactive Image Segmentation using Irregular Pyramids. In *Graph-Based Representations in Pattern Recognition*, pages 245–254. Springer, 2011.

[19] R. Glantz and W. G. Kropatsch. Relinking of Graph Pyramids by Means of a New Representation. In *Proceedings of the Czech Pattern Recognition Workshop*, 2000.

[20] S. Golodetz. *Zipping and Unzipping: The Use of Image Partition Forests in the Analysis of Abdominal CT Scans*. PhD thesis, University of Oxford, 2011.

[21] S. Golodetz, C. Nicholls, I. Voiculescu, and S. Cameron. Two Tree-Based Methods for the Waterfall. *Pattern Recognition*, 47(10):3276–3292, October 2014.

[22] S. Golodetz, C. Nicholls, V. Yeghiazaryan, J. Pumphrey, I. Ivan, I. Voiculescu, and S. Cameron. *millipede*. Available online (as of 12th April 2014) at https://github.com/sgolodetz/millipede.

[23] S. Golodetz, I. Voiculescu, and S. Cameron. Region Analysis of Abdominal CT Scans using Image Partition Forests. In *Proceedings of CSTST '08*, pages 432–7, Cergy-Pontoise, France, October 2008.

[24] S. Golodetz, I. Voiculescu, and S. Cameron. Automatic Spine Identification in Abdominal CT Slices using Image Partition Forests. In *Proceedings of ISPA '09*, pages 117–122, Salzburg, Austria, September 2009.

[25] S. M. Golodetz. A 3D Map Editor. Undergraduate thesis, Oxford University Computing Laboratory, May 2006.

[26] Y. Haxhimusa and W. Kropatsch. Hierarchical Image Partitioning with Dual Graph Contraction. Technical Report PRIP-TR-81, Vienna University of Technology, Vienna, Austria, July 2003.

[27] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical Optimization of Optimal Path Finding for Transportation Applications. In *Proceedings of the 5th International Conference on Information and Knowledge Management*, pages 261–268, Rockville, Maryland, United States, 1996.

[28] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):409–432, 1998.

[29] S. Jung and S. Pramanik. HiTi Graph Model of Topographical Road Maps in Navigation Systems. In *Proceedings of the 12th International Conference on Data Engineering*, pages 76–84. IEEE Computer Society, 1996.

[30] S. Katz and A. Tal. Hierarchical Mesh Decomposition using Fuzzy Clustering and Cuts. *ACM Transactions on Graphics (SIGGRAPH)*, 22(3):954–961, 2003.

[31] A. Kerren. Interactive Visualization of Graph Pyramids. In *Graph Drawing*, Dagstuhl, Germany, 2006.

[32] K. Kim, S. Yoo, and S. K. Cha. A Partitioning Scheme for Hierarchical Path Finding Robust to Link Cost Update. In *Proceedings of the 5th World Congress on Intelligent Transport Systems (CD-ROM)*, 1998.

[33] B. Klava and N. S. T. Hirata. Watershed segmentation: Switching back and forth between markers and hierarchies. In *Proceedings of the 8th International Symposium on Mathematical Morphology (ISMM)*, pages 29–30, Rio de Janeiro, Brazil, October 2007.

[34] W. G. Kropatsch and Y. Haxhimusa. Grouping and Segmentation in a Hierarchy of Graphs. In *Proceedings of the 16th IS&T SPIE Annual Symposium*, pages 193–204, 2004.

[35] O. Lezoray, C. Meurie, P. Belhomme, and A. Elmoataz. Multi-Scale Image Segmentation in a Hierarchy of Partitions. In *Proceedings of the 14th European Signal Processing Conference (EUSIPCO '06, CD-ROM)*, Florence, Italy, September 2006.

[36] B. Marcotegui and S. Beucher. Fast Implementation of Waterfall based on Graphs. In C. Ronse, L. Najman, and E. Decencière, editors, *Mathematical Morphology: 40 Years On*, pages 177–186. Springer Netherlands, 2005.

[37] R. Marfil, L. Molina-Tanco, A. Bandera, and F. Sandoval. The Construction of Bounded Irregular Pyramids with a Union-Find Decimation Process. *Lecture Notes in Computer Science*, pages 307–318, 2007.

[38] A. Meijster and J. Roerdink. A Disjoint Set Algorithm for the Watershed Transform. In *Proceedings of the 9th European Signal Processing Conference (EUSIPCO '98)*, pages 1665–1668, 1998.

[39] P. F. M. Nacken. Image Segmentation by Connectivity Preserving Relinking in Hierarchical Graph Structures. *Pattern Recognition*, 28(6):907–920, 1995.

[40] N. R. Pal and S. K. Pal. A Review on Image Segmentation Techniques. *Pattern Recognition*, 26(9):1277–1294, 1993.

[41] P. Perona and J. Malik. Scale-Space and Edge Detection Using Anisotropic Diffusion. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(7):629–639, July 1990.

[42] D. L. Pham, C. Xu, and J. L. Prince. Current Methods in Medical Image Segmentation. *Annual Review of Biomedical Engineering*, 2:315–37, 2000.

[43] L. Prasad and S. Swaminarayan. Hierarchical image segmentation by polygon grouping. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2008.

[44] A. Shamir. A Survey on Mesh Segmentation Techniques. *Computer Graphics Forum*, 27(6):1539–1556, 2008.

[45] X. Shen and M. Spann. Segmentation of 2D and 3D Images through a Hierarchical Clustering Based on Region Modelling. In *Proceedings of the 1997 International Conference on Image Processing (ICIP '97)*, pages 50–53, 1997.

[46] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, August 2000.

[47] J. Tierny, J.-P. Vandeborre, and M. Daoudi. Topology driven 3D mesh hierarchical segmentation. In *Proceedings of the IEEE International Conference on Shape Modeling and Applications (Shape Modeling International – SMI)*, pages 215–220, Lyon, France, 2007.

[48] W. van der Sterren. Path Look-Up Tables – Small Is Beautiful. In S. Rabin, editor, *AI Game Programming Wisdom 2*, pages 115–129. Charles River Media, 2004.

[49] Z. Wu and J. M. S. Jr. Multiple Material Marching Cubes Algorithm. *International Journal for Numerical Methods in Engineering*, 58(2):189–207, July 2003.

[50] N. Yuruk, M. Mete, and X. Xu. A Divisive Hierarchical Structural Clustering Algorithm for Networks. In *Proceedings of the Seventh IEEE International Conference on Data Mining (ICDM)*, pages 441–448, 2007.

[51] Z. Zivkovic, B. Bakker, and B. Kröse. Hierarchical Map Building and Planning based on Graph Partitioning. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 803–809, 2006.