

Hash-Consing Digraphs Using Hop Trees

William H. Newman *

February 28, 2007

Abstract

We define the hop tree, a data structure to classify vertices in incremental partition refinement. Using hop trees and a trick of replaying partition refinement of frontier vertices, we show how to maintain a directed graph in minimized form (with respect to the relational coarsest partition) efficiently as new subgraphs are added. The expected cost of the fundamental operation, hash-consing a new subgraph G with respect to a previously minimized graph M , is

$$g_1 + g_2(\log g_2)(\log m_1)^2(\log m_2)(\log(g_2(\log m_1)))$$

operations, where g_1 and m_1 are the number of vertices of G and M , and g_2 and m_2 are the number of edges of G and M .

A simple procedure can be used to preprocess graphs with labelled edges so that our algorithm can be used to minimize them. For such graphs, the runtime cost of our algorithm is only $O(g_1 + g_2(\log m_1)^3)$.

For either the general case of relational coarsest partition or the special case of graphs with labelled edges, representing a minimized graph M requires $O(m_1 + m_2)$ cells of memory.

*5301 W. Spring Creek Pkwy. #737, Plano, TX 75024, william.newman@airmail.net

Contents

1	Introduction	4
2	The Problem	5
3	Our Solution	8
3.1	Overview	8
3.2	General Architecture	9
3.2.1	A Naive Algorithm and Naive Taxonomy	9
3.2.2	Compressing the Taxonomy	11
3.2.3	The Randomized Hopcroft Trick	11
3.2.4	The Hop Tree; Unperturbed Trajectories	15
3.2.5	Replaying Partition Refinement of Frontier Vertices	17
3.3	Top-Level Algorithm and Ephemeral Data Structures	19
3.3.1	The Interface: <code>minimize(G)</code>	19
3.3.2	Vertex Annotations: <code>ev</code> , <code>fv</code> , and <code>gv</code>	21
3.3.3	The <code>refine()</code> Operation	24
3.3.4	The <code>refine_once()</code> Operation	25
3.3.5	The <code>block</code> Class	25
3.3.6	The <code>stress()</code> Operation	29
3.3.7	The <code>bend()</code> Operation	32
3.3.8	The <code>diffdiff($(A_{\oplus}, A_{\ominus}), (B_{\oplus}, B_{\ominus})$)</code> Operation	33
3.4	Persistent Data Structures; the Taxonomy	35
3.4.1	Design Constraints	35
3.4.2	Satisfying Uninteresting Design Constraints	36
3.4.3	Avoiding Arity Skew; the <code>aw</code> Weight	38
3.4.4	The <code>subdown(r, σ, d)</code> Operation	42
3.4.5	The <code>superuptify1(d)</code> Operation	45
3.5	Asymptotic Efficiency	45
3.5.1	Possible Efficiency Improvements	49
4	Conclusion	51
A	Class Layout	52
B	Minimizing Graphs With Labelled Edges	53

C	Cost of the Randomized Hopcroft Rule	54
C.1	Results For Partitions	54
C.2	Results For Trees	55

List of Figures

1	A naive offline relational coarsest partition algorithm	10
2	An example graph.	12
3	The naive taxonomy	13
4	Compressing the taxonomy	13
5	Adding unperturbed trajectories to the taxonomy	15
6	An incremental minimization problem	17
7	The <code>minimize(G)</code> operation	20
8	The <code>ev(v)</code> operation	22
9	The <code>sdownt(ε)</code> operation	23
10	The <code>refine(v)</code> operation	24
11	The <code>refine_once()</code> operation	26
12	An example graph to motivate differential keys	27
13	The <code>block()</code> operation	28
14	The <code>postinit_broken()</code> operation	29
15	The <code>stress()</code> operation	30
16	The <code>stress_gv_as_antiedge(ε)</code> operation	31
17	The <code>stress_block_as_antiedge(β, ε)</code> operation	32
18	The <code>bend()</code> operation	33
19	The <code>bend_gv(γ)</code> operation	34
20	The <code>diffdiff($(A_{\oplus}, A_{\ominus}), (B_{\oplus}, B_{\ominus})$)</code> operation	34
21	The <code>became(v)</code> and <code>set_redundant_became()</code> operations	37
22	The <code>initial_kind(v)</code> operation	37
23	The <code>set_sdownt_and_enqueue_reserves(ϕ)</code> operation	38
24	The <code>rebalance()</code> operation	41
25	The physical leafward links of the taxonomy	43
26	The <code>subdownt(r, σ, d)</code> operation	44
27	The <code>stable_key($(\sigma_{\oplus}, \sigma_{\ominus}), s$)</code> operation	45
28	The <code>superuptify1(d)</code> operation	46
29	The <code>acquire_heirs_above_r(d, d_{previous})</code> operation	47
30	Layout of classes used in the pseudocode	53

1 Introduction

Mauborgne seems to have been the first to pose the problem of incrementally efficient partition refinement,[16] limiting the problem to graphs with labelled edges. His Mauborgne[16] and others[3, 8, 5] have proposed algorithms for that problem which are incrementally efficient for many inputs but which for difficult inputs fall back to an offline algorithm, such as that of Cardon and Crochemore.[2] Other published approaches to the problem include an algorithm which is always fast but which doesn't detect all equivalences[5] and a method for delaying minimization until global pickling time.[21]

Our algorithm solves the Mauborgne problem incrementally efficiently for all inputs. Our algorithm also solves a more general problem than the problem posed by Mauborgne: incrementally efficient relational coarsest partition. Mauborgne sought a generalization of the algorithm of Cardon and Crochemore,[2] while our algorithm generalizes the algorithm of Paige and Tarjan.[18] Our Appendix B shows how to transform a problem of partitioning graphs with labelled edges into a relational coarsest partition problem, while the opposite transformation seems to be impossible.

Our algorithm draws ideas from the offline graph equivalence algorithms of Hopcroft,[7] Cardon and Crochemore,[2] and Paige and Tarjan.[18] We also borrow a general design approach from the recent innovations of Eppstein *et al.*[20] in the problem of maintaining balance in multidimensional search trees.

Hopcroft, in his algorithm for minimizing finite automata,[7] introduced the key idea of not propagating those partition refinement updates related to the largest subblock of a refined block. This trick reduces the cost of partition refinement to $O(n \log n)$ and has been used by all subsequent algorithms. We introduce a randomized variant of the trick which has some convenient properties for our incremental algorithm. Roughly our approach is to Hopcroft's approach as randomized QUICKSORT is to deterministic MERGESORT.

Cardon and Crochemore generalized the approach of Hopcroft from finite automata to graphs with labelled edges (and also to graphs in which edge collections behave like multisets, so that order of edges is irrelevant but the number of copies of an edge is important; but such graphs seem less related to our problem). They showed how to handle splittings which create many offspring simultaneously, and though there are differences in detail (*e.g.*, their list accumulator sorted after accumulation corresponds roughly to our maps) we use the same idea.

Paige and Tarjan showed how to solve the problem of relational coarsest partition efficiently. In this problem, the way that edges may disappear when equivalences are discovered makes it impossible to apply the ideas of Cardon and Crochemore directly. We borrow from Paige and Tarjan the idea of maintaining a map from blocks to integer counts of appearances of blocks among the edges.

The “skip quadtree” structure of Eppstein *et al.* solves a long-standing problem of maintaining balance in a multidimensional search tree. Our approach is inspired by theirs, although quite different in detail, because the problem of classifying vertices has dimensionality and sparseness problems that can generally be ignored in the problem of classifying points in a real physical space, and because where Eppstein *et al.* know that boundaries at fractions like $i/2^n$ relative to some origin will suffice to classify their input, we can’t easily express our classification criteria for a generation of partition refinement until the previous generation is complete. One consequence is that our rebalancing cost has much higher variance than theirs: like them we expect to rewrite $O(\log n)$ branches upon insertion in a tree of $O(\log n)$ leaves, but where they have an exponentially small chance of having to rewrite all n branches, we have a $O(1/n)$ chance of having to rewrite all n branches.

2 The Problem

Our problem is a variant of the “relational coarsest partition” problem as defined below, parallel to the definition by Paige and Tarjan.[18] Paige and Tarjan’s offline algorithm computed the relational coarsest partition of a single fixed graph M ; our online algorithm allows M to grow with the progressive addition of new subgraphs G , and maintains a map from input vertices to vertices of a minimized graph (and also maintains some auxiliary data structures).

Although the problem of calculating $\text{rcp}(M)$ is essentially equivalent to the problem of calculating relational coarsest partition of M as defined by Paige and Tarjan, we will not use their notation. Instead, we use a notation closer to the implementation in a conventional programming language when the vertices of the graph happen to be typed records or objects. Because this programmer-style notation is fairly common in the graph-minimization literature[5] and because it has a straightforward correspondence to the mathematician-style notation of Paige and Tarjan (*e.g.*, we say that ver-

tices have labels, they say that there is an initial partition P , and the two statements are related by our operator `partition_by_labels`) there should be little confusion.

Our problem is formulated in terms of (directed) graphs of vertices.

Each vertex v has a label `label(v)`. Vertex labels can be compared for equality, sorted, and hashed. In some problems[14, 21] they may correspond to the class of an object in a programming language.

Each vertex v has a set of outgoing edges `edges(v)`. In some problems[14, 21] this set may correspond roughly to fields of an object in a programming language, using the transformation in Appendix B as necessary to make distinguishable fields correspond to labelled edges.

(Note that other symmetries of vertex edge collections (neither set-like symmetry as in our basic algorithm, nor tuple-like symmetry as in Appendix B) are sometimes used, though they seem considerably less common in practice than sets and tuples. The most common such symmetry seems to be what Paige and Tarjan[18] call s -stability, and which Cardon and Crochemore call “unlabelled graphs”. The general ideas in this paper might be useful for an incrementally efficient algorithm for that case, but we won’t attempt it here.)

A graph is a set of vertices which is closed under the `edges` relationship. We define the operator $\#_2$ to be the number of edges of a vertex, or the number of edges in a graph. By this we mean the number of physical references in the current representation of the graph, not the ideal minimal number of referred-to minimized vertex objects in some future minimized version of the graph.

We also sometimes write $\#_1$ for the number of vertices in a graph. ($\#_1$ is the same as the number of elements in the set, which in other contexts we would write as $\#$, but often we prefer the parallelism with $\#_2$.)

When we refer to a “block” we mean a set of vertices, with the connotation that the set is one element of some partition.

In describing offline problems, it is natural to define a one-argument function referring to the parents of a vertex or block. *E.g.*, Paige and Tarjan call this function $E^{-1}(S)$. In our online problem, however, the offline algorithms’ implicit assumption of a fixed universe of vertices does not hold: the contents of such a set could change as we encounter new graphs G . Therefore it is tidier to define relational coarsest partition in terms of the opposite

relationship,

$$\text{immreaches}(v, S) \equiv \exists s \in S : s \in \text{edges}(v). \quad (1)$$

We generalize this from blocks to partitions,

$$\text{immreaches}(V, S) \equiv \exists v \in V : \text{immreaches}(v, S) \quad (2)$$

$$\text{immreaches}(x, \Pi) \equiv \exists B \in \Pi : \text{immreaches}(x, B). \quad (3)$$

We define an operator $\text{split}(S, Q)$ to split a block or blocks Q into sub-blocks which are indistinguishable with respect to whether they immediately reach S . Our $\text{split}(S, Q)$ is just like the operator of the same name in Paige and Tarjan, except that we generalize it to accept either a block or a partition as either argument. In the block case,

$$\text{split}(S, Q) \equiv \{\forall B \in \text{raw_split}(S, Q) : B \neq \emptyset\} \quad (4)$$

where

$$\text{raw_split}(S, Q) \equiv \quad (5)$$

$$\bigcup_{B \in Q} \{\forall b \in B : \text{immreaches}(b, S)\} \vee \quad (6)$$

$$\bigcup_{B \in Q} \{\forall b \in B : \neg \text{immreaches}(b, S)\}. \quad (7)$$

Generalizing to partitions, we have

$$\text{split}(\emptyset, Q) \equiv Q \quad (8)$$

$$\text{split}(\{B\} \cup S, Q) \equiv \text{split}(S, \text{split}(B, S)). \quad (9)$$

(The ambiguity in Equation 9 of which element B to choose is harmless, because as pointed out by Paige and Tarjan, split is commutative with respect to different splitters.)

(All other ambiguities in the generalizations from blocks to partitions above are also harmless: *e.g.*, $\text{split}(\emptyset, Q)$ might refer either to splitting by an empty block or to splitting by an empty partition, but in both cases the result is the same.)

We say x is stable with respect to a splitter s if $\text{split}(s, x)$ is an identity on x .

We say that a partition Π' is a refinement of a partition Π iff $\forall B' \in \Pi' : \exists B \in \Pi : B' \subseteq B$. We say that a refinement Π' is coarser than a partition Π iff $|\Pi'| < |\Pi|$.

We define

$$\text{labels}(V) \equiv \bigcup_{v \in V} \{\text{label}(v)\} \quad (10)$$

and

$$\text{partition_by_labels}(G) \equiv \bigcup_{\ell \in \text{labels}(G)} \forall v \in V : \text{label}(v) = \ell. \quad (11)$$

Now we can define the relational coarsest partition: $\text{rcp}(G)$ is the coarsest refinement of $\text{partition_by_labels}(G)$ which is stable with respect to splitting by itself. It is unique, as shown in Theorem 2 of Paige and Tarjan.[18]

Less formally, the relational coarsest partition is the partition which groups together vertices which are equivalent with respect to bisimulation. Even less formally than that, it is the partition which groups together multiple vertices which can be usefully rerepresented as a single vertex in various graphs of practical interest, such as finite automata,[7] binary decision diagrams[1] and their cyclic generalizations,[16] combinatorial games[6] and various constructs (including both data flow graphs and graphs representing compound types) which appear in program analysis.[19, 11]

As noted earlier, most of the published work on the problem of incremental graph minimization has been on the subproblem of graphs with labelled edges, and many of the practical examples of graph minimization above also involve labelled edges. The relational coarsest partition problem doesn't associate labels with edges, but it is easy (by adding one new vertex and one new edge per old edge, as described in Appendix B) to transform a problem of minimizing a graph with labelled edges into a problem of minimizing a graph with unlabelled edges, *i.e.*, a relational coarsest partition problem.

3 Our Solution

3.1 Overview

Our main persistent data structure, the “taxonomy,” is built around a “hop tree” which has some design themes in common with skip quadtrees.[20]

To motivate the definition of a skip quadtree for n data points, one could start by imagining a naive quadtree which unfortunately could require more than $O(n)$ space. Then, to reduce that space cost, one could physically implement a corresponding compressed quadtree which takes only $O(n)$ space but unfortunately is unbalanced, possibly requiring $O(n)$ links to be traversed in search. Then, to reduce that time cost, one could layer a skip-list-like hierarchy of subsets over the compressed quadtree so that by descending through the hierarchy a path of expected length $O(\log n)$ exists to any point, and in terms of this descent-through-hierarchy search the tree is balanced.

Similarly, we will begin the definition of a hop tree by imagining a naive taxonomy tree for which unfortunately might require more than $O(n)$ nodes to represent the classification of n vertices. Then we will show how to implement a corresponding compressed taxonomy tree which only requires $O(n)$ nodes but which unfortunately is unbalanced, possibly requiring $O(n)$ links to be traversed in search for any classification. Then we will show how to avoid that time cost by maintaining and searching a randomized tree layered over the compressed tree. This randomized tree corresponds to a randomized version of the Hopcroft partition refinement trick. Where the usual deterministic Hopcroft trick guarantees that the number of times that any vertex is reclassified is deterministically bounded above by $\log_2 n$, our randomized trick will guarantee that the expected number of hop tree branches *en route* to the classification of any vertex is bounded above by $\log_2 n$.

3.2 General Architecture

3.2.1 A Naive Algorithm and Naive Taxonomy

In this subsection we will describe a classification scheme which we will not implement, but which has a close correspondence to the scheme which we will implement, the compressed taxonomy of Subsubsection 3.2.2.

Imagine computing the relational coarsest partition of a graph M nonincrementally. A naive algorithm to do so is in Figure 1.

Our naive algorithm is closely related to the naive algorithm given at the bottom of p. 978 by Paige and Tarjan[18]. In particular, their step “find a set S ” allows enough freedom of choice in splitters S that we can arrange the sequence of partitions found by their algorithm to be a superset of our sequence of partitions Π_r . Therefore, by their Theorem 2, our naive algorithm converges to the relational coarsest partition. Our naive algorithm is also

```

naive_rcp( $M$ ) :
   $r \leftarrow 0$ 
   $\Pi_0 \leftarrow \text{partition\_by\_labels}(M)$ 
  repeat
     $\Pi_{r+1} \leftarrow \text{split}(\Pi_r, \Pi_r)$ 
    if  $\Pi_{r+1} = \Pi_r$ 
      return  $\Pi_r$ 
     $r \leftarrow r + 1$ 

```

Figure 1: A naive offline relational coarsest partition algorithm

very nearly the same as the sequence of equivalences Π_i defined on p. 87 by Cardon and Crochemore,[2] differing in the way that Cardon and Crochemore are solving a related partition problem instead of relational coarsest partition.

In our online algorithm the refinement generation r not just an accidental feature of an illustrative derivation, it is actually important to the implementation, for two reasons. Most importantly, the partitions Π_r turn out to have relatively simple behavior when the discovery of new vertices causes us to rebalance the taxonomy tree. Less importantly, keeping track of the generation r helps us describe how features discovered in partition refinement of a new graph $G \cup M$ relate to earlier partition refinement of an old graph M . Thus, our r values appear in our persistent data structures as keys of maps, and also appear as keys in the priority queue of replayed events which we use to relate the ongoing partition refinement of G to the remembered partition refinement of M .

By creating records which correspond to the states of the naive algorithm, we can generate a naive taxonomy tree. It is a search tree in layers indexed by r . Each tree leaf corresponds to a block of the final `rcp`, and describes a complete classification of the vertices in that block. Each tree branch (which we will call a `kind`) corresponds to a block of some earlier Π_r , and describes a perhaps-incomplete classification of the vertices in that block. Each `kind` also contains references (as a map) to `kinds` of the next generation which correspond to subblocks of its block.

The keys of the map in a `kind` at generation r correspond to the pattern of blocks of Π_r which are seen in edges of vertices of that `kind`. In the interests of helping the taxonomy remain stable when used in an incremental algorithm, we avoid using blocks of Π_r directly in the keys. Instead, we avail

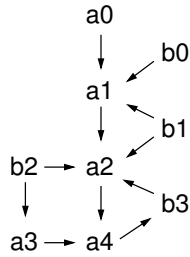


Figure 2: An example graph. Five vertices are labelled 'a' and four vertices are labelled 'b'. In the relational coarsest partition, vertex a_2 is equivalent to vertex a_3 , and vertex b_2 is equivalent to vertex b_3 .

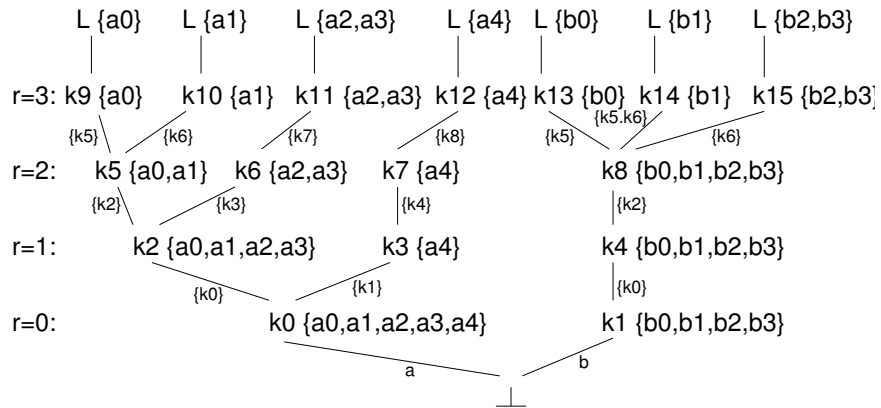


Figure 3: The naive taxonomy of the graph in Figure 2

ourselves of the 1-to-1 correspondence between blocks of Π_r and kinds at generation r , and use the kinds in the keys. Thus, each map key is a set kinds corresponding to the blocks reached by elements of the subblock which correspond to the mapped-to kind.

Figure 3 illustrates the naive taxonomy and refined partitions generated for the graph in Figure 2. It shows, for example, that the kind k_8 corresponds to the block $\{b_0, b_1, b_2, b_3\}$ at generation $r = 2$, and is split three ways; vertices of k_8 which immediately reach only vertices of k_5 end up in one leaf, vertices of k_8 which immediately reach both vertices of k_5 and vertices of k_6 end up in another leaf, and vertices of k_8 which immediately reach only vertices of k_6 end up in yet another leaf. At $r = 3$ the partition

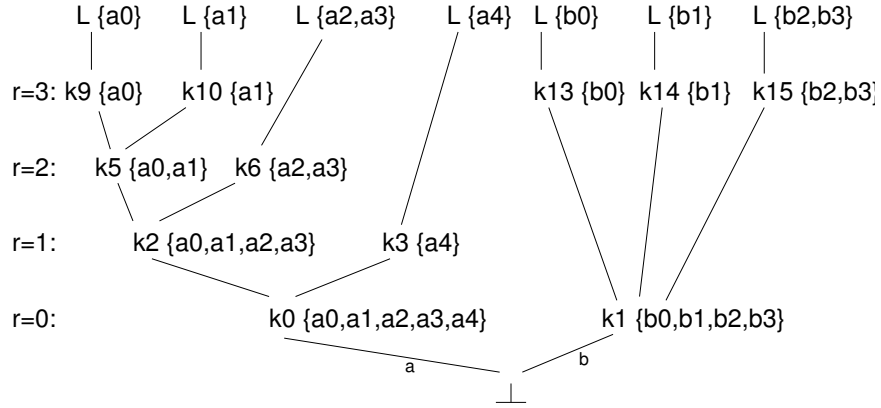


Figure 4: Compressing the taxonomy of the graph in Figure 3

is stable, so that vertices which are still in the same block are equivalent: a_2 is equivalent to a_3 , and b_2 is equivalent to b_3 .

3.2.2 Compressing the Taxonomy

A naive taxonomy tree can have $O(\#_1 M^2)$ branches: it is easy to construct a graph of n vertices where approximately $n/2$ kinds are discovered at an early generation, while $n/2$ refinement generations are required to discover the other kinds, so that approximately $n^2/4$ objects would be required to represent the taxonomy.

In order to make the number of nodes in our taxonomy grow only linearly with the number of unique vertices in M , we will compress the taxonomy using a rule similar to that used to generate compressed quadtrees:[20] a **kind** which has only a single **kind** as a descendant is elided. Equivalently, when of the current population of known-to-be-inequivalent vertices (*i.e.*, the population of vertices for which **leafs** have been allocated) more than one **kind** would hold equal subpopulations, we elide all except that **kind** whose generation r is smallest. The result of applying this rule is shown in Figure 4. The splitting keys are not shown, because the splitting keys we use in the taxonomy depend on an “unperturbed trajectory” concept that we haven’t defined yet; see Figure 5.

3.2.3 The Randomized Hopcroft Trick

Hopcroft showed that partition refinement could be converted from $O(n^2)$ to $O(n \log n)$ by, in effect, skipping some of the steps in the expansion of our Equation 9. All efficient partition refinement schemes are based on this trick, and ours is no exception. However, we choose to modify the trick slightly in order to make the performance of our incremental algorithm smoother.

A well-known design theme in randomized algorithms is that an algorithm which is efficient except for malicious input may be converted to an algorithm with efficient expected behavior by randomizing the problem. For example, a QUICKSORT implementation with a deterministic rule for choosing the splitter element may be inefficient for pathological input tailored to defeat its deterministic rule, but using a randomized rule defends against this.[13] It is in a similar spirit that we will randomize the Hopcroft trick.

We are defining a search tree data structure which corresponds to the progress of a partition refinement calculation. Using the Hopcroft trick in the partition refinement calculation causes every refined block to be no more than half the size of the parent block, and in our data structure this will translate into logarithmic depth, the balancing that we want. However, in an incremental calculation, a deterministic Hopcroft rule has a problem.

In order to guarantee the smallness of refined blocks, the deterministic Hopcroft rule to skip that block which has the largest number of vertices. Unfortunately, a graph M can be constructed whose taxonomy contains two huge blocks of nearly-equal size $O(M)$, and then a malicious sequence of small `minimize(G)` (with $|G| \ll |M|$) can cause the relative size of the blocks to be exchanged on every `minimize` operation. In our translation from algorithm to taxonomy, causing a previously-skipped block to become an explicit splitter is linearly expensive in the size of the block, so using a deterministic Hopcroft rule would allow the construction of inputs which defeated incremental efficiency.

To avoid this problem, we define a randomized rule: we assign to each vertex a random number, and when choosing a block to skip, we choose that block which contains the vertex with the smallest random number. The effect is similar to the effect of the deterministic Hopcroft rule: we show in Appendix C that the expected size of an unskipped subblock is no more than half the size of the parent block. Under the randomized rule it does still happen sometimes that a small G will cause a previously-skipped huge block to become a splitter. However, the inputs which will cause this to happen are

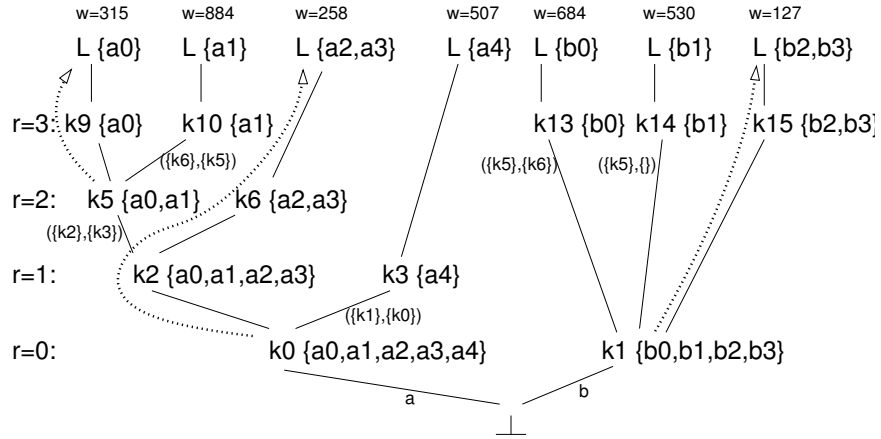


Figure 5: Adding unperturbed trajectories to the taxonomy of Figure 4. We show random weights w , unperturbed trajectories (the dotted arrows running to smallest- w leaf), and differential keys (tags like “(k6,k5)”).

no longer predictable, and so expected incremental efficiency is protected.

Should we use one random number for each input vertex, or only one random number for each input vertex which turns out to be unique? In fact, we do both. During incremental partition refinement, working with ephemeral data, we use one random sample for each input vertex whether or not it is unique, because in general we don’t know yet whether it is unique. (And, in fact, with a little more effort we could even use the deterministic Hopcroft rule; randomization is not important here.) But when rebalancing the persistent hop tree, we do know whether a vertex is unique, and at that time we use one random sample for each unique vertex. Balancing the persistent tree using one sample per non-unique vertex would still work somewhat, but the depth of the tree would tend to be bounded by the logarithm of the number of all input vertices, rather than by the logarithm of the number of unique vertices.

3.2.4 The Hop Tree; Unperturbed Trajectories

In Figure 5 we have assigned a random weight w to each leaf and derived unperturbed trajectories and “differential keys” from the weights.

The differential keys are the representation used in the persistent taxon-

omy to characterize a “swerve” from an unperturbed trajectory. A differential key is a pair of a set of positive terms and a set of negative terms. Let K_{\oplus} and K_{\ominus} be the sets of `kind` values which appear as splitting keys in the naive taxonomy of Figure 3: sets of `kinds` representing the naive partition blocks which immediately reached by the edges of all vertices in a subtree. Then a (stable) differential key is the pair of set differences $(K_{\oplus} \setminus K_{\ominus}, K_{\ominus} \setminus K_{\oplus})$. (Later we will also compute a less stable set difference expression which represents the same concept. The less-stable representation is used in ephemeral data structures where its instability with respect to rebalancing of the taxonomy tree doesn’t matter, then translated to the stable representation whenever it is to be used in the persistent taxonomy tree.)

We call this taxonomic data structure a “hop tree” because the English verbs “hop” and “skip” are related, and our tree maintains balance by translating a variant of the Hopcroft trick into something reminiscent of a skip quadtree.

It follows from the results in Appendix C that if traveling along an unperturbed trajectory has unit cost, then the expected cost of traveling from the root of the tree to any leaf is no more than the base two logarithm of the number of leaves. Thus, our “hop tree” of unperturbed trajectory shortcuts is balanced. Much of the rest of the paper will be arranging for all the operations we need to do with the hop tree shortcuts to be reasonably cheap.

Our use of the randomized Hopcroft trick is reminiscent not only of skip quadtrees, but of some kinds of perturbation theory for physical problems. We can reexpress the solutions to our original problem as some combination of solutions to some related problem. When the problems are relatively similar, then the relationship between the solutions can be relatively simple. When the related problem is much more tractable than the original problem, this can be a very useful trick.

In the three-body problem of classical mechanics, for example, if one of the bodies interacts relatively weakly with the other two, it can be helpful to solve the related two-body problem (which can be done exactly) and then reexpress the solutions to the three-body problem in terms of the solutions to the two-body problem. The solutions to the two-body problem in this case can be called the unperturbed trajectories, where the perturbation is the correction introduced to account for the interaction with the third body. We borrow that terminology (sometimes abbreviating “unperturbed trajectory” to `upt`) for our approach.

In our problem of classification of vertices, using the unbalanced com-

pressed taxonomy directly would require us to perform lookups at many nodes, and would cause the classification of a vertex to change many times, making our partition refinement expensive. However, we can guess that the progress of a vertex through partition refinement will be similar to the progress of that vertex in its block which has the minimal random number. We call that guess, that the vertex’s refinement history will be the same as a random representative, our unperturbed trajectory. Then we reexpress our problem in terms of deviations from that guess; that is our randomized version of the Hopcroft trick. (For the deterministic Hopcroft trick, the corresponding guess is effectively that at every split of a block of the partition, the vertex will end up in the largest subblock.)

The follow-the-minimal-vertex guess is obviously tractable. It may not be immediately obvious that the guess is a useful approximation, but it does seem to be. Formally, it can be seen to be useful from the bounds calculated in Appendix C. Intuitively, it is at least plausible that it since the deterministic Hopcroft guess is known to be useful, a similar randomized guess should also be useful.

3.2.5 Replaying Partition Refinement of Frontier Vertices

Besides the general taxonomic architecture we have developed so far, and the detailed engineering we have promised to do to make it efficient to work with it, little else is needed for an incrementally efficient partition refinement algorithm. Primarily we need the trick described in this subsection, of replaying partition refinement. (Secondarily we will some way to address the problem of “arity skew,” as in Subsubsection 3.4.3, but we have not introduced enough definitions to make it convenient to describe that problem yet.)

Consider an incremental graph minimization problem, such as the one shown in Figure 6. We could find the relational coarsest partition of $G \cup M$ by running `naive_rcp($G \cup M$)` from Figure 1. Unfortunately that approach would be inefficient in general, because anything like `naive_rcp($G \cup M$)` is at least linearly expensive in $|M|$, while G can be much smaller than M ; we seek an incremental algorithm whose cost tends to be proportional to $|G|$.

To generate an incremental algorithm from `naive_rcp(M)`, we observe that (1) adding vertices of G to the problem tends not to change the “refinement trajectory” of vertices of M through the taxonomy in partition refinement, and (2) in order to refine vertices of G , all we need to know about the refinement of M is contained in the refinement trajectory of “fron-

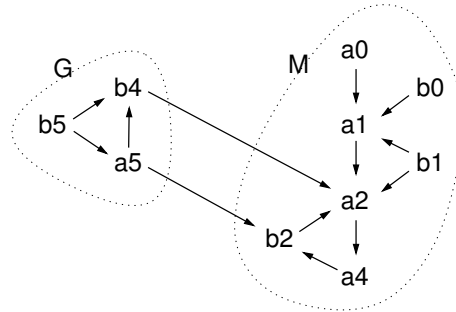


Figure 6: An incremental minimization problem. M is a minimized version of the graph of Figure 2, while G is not minimized yet.

tier vertices,” those vertices of M which are immediately reached by vertices of G .

To expand on point (1), consider Figure 4 describing the minimization of Figure 2. If we add more vertices to the problem as in Figure 6, much of the structure of Figure 4 will be preserved. New kinds may be added, and the blocks associated with old kinds may contain more vertices, but it will always be the case that at generation $r = 2$ the vertices a_0 and a_1 are in the same block, while the vertex b_0 is in a block different from any of the other vertices of Figure 2. Thus, any properties that we can express in terms of sets which correspond exactly to the Π_r partitions of $\text{naive_rcp}(G \cup M)$ will tend to be stable under addition of new subgraphs to the minimization problem.

To expand on point (2), consider Figure 3. The keys needed to find progressively sharper refinements of a vertex’s classification are the classifications of its edges: e.g., the refinement of a_2 is driven (directly) only by the refinement of a_4 , not by any other vertex. Therefore, if an oracle had told us the refinement states of a_4 (k_0 at $r = 0$, k_3 at $r = 1$, etc.), we could have calculate the fully refined classification of a_2 without having to process any other vertices. Furthermore, this is true not just for individual vertices but for subgraphs: e.g., with oracular knowledge of the refinement states of a_4 , we could refine not only a_2 but some of its ancestors (a_0 and b_0 , e.g.), without ever processing other vertices of the the graph (b_2 , e.g.). Furthermore, a look at Figure 5 should suggest that for the purposes of refining a new subgraph G which touches the already minimized subgraph described

by the taxonomy of M , retaining the taxonomy will make it practical for us to provide such oracular knowledge. If in the refinement of G we want to know the refinement history of a vertex v of M , we can calculate it by going to the leaf associated with v , walking rootward through the tree, and taking note of the swerves.

To expand further on points (1) and (2), adding new unique vertices will create leaves with new random values w , and therefore can change the unperturbed trajectories in Figure 5. (*I.e.*, when we discover new unique vertices we rebalance the search tree.) In the presence of such changes, we will need to take care to preserve the kinds of stability we care about for point (1), and to make sure that inexpensive updates suffice to keep it inexpensive to look up histories for point (2). We will address these and other concerns in Subsection 3.4.

3.3 Top-Level Algorithm and Ephemeral Data Structures

Our design ideas in the previous section were too vague about many low-level details for us to give pseudocode for a complete algorithm yet. However, they were sufficiently precise that before we go on to low-level details in the next section, we can write the top level of pseudocode.

3.3.1 The Interface: `minimize(G)`

Our algorithm will maintain a map `became` and an already-minimized graph M , both initially empty, and it will provide an operator `minimize(G)`. The effect of `minimize(G)` will be to replace M with $M \cup G$ and to update `became` as necessary to maintain the invariant that `became` corresponds to the relational coarsest partition of M :

$$\forall B \in \text{rcp}(M) : \forall b \in B : \text{became}(b) \in B. \quad (12)$$

Our algorithm will also maintain some private data structures (notably the taxonomy) to help do this efficiently.

By reporting `became` instead of reporting `rcp` directly, the algorithm supports a “hash consing” usage pattern where the heap space used to represent the result of minimization is only proportional to the number of unique vertices, not proportional to the raw number of vertices in the pre-minimization

```

minimize( $G$ ) :
  assert( $G \cup M$  is closed under edges) /* (See text.) */
   $s \leftarrow (-1)$  /* “the generation of splitter blocks” before there actually are any */
   $r \leftarrow 0$  /* the generation of blocks being created */
  for  $v \in G$ 
    ev( $v$ ) /* creating initial partition of  $G$  as a side effect (Figure 8) */
  refine() /* to stability through increasing  $s$  and  $r$  (Figure 10) */
  set_redundant_became() /* (Figure 21) */
  rebalance() /* (Figure 24) */
  Clear vertex_ev. /* (a map used in Figure 8) */
  Clear kind_block. /* (a map used in Figure 13) */

```

Figure 7: The `minimize(G)` operation: Minimize the vertices of G with respect to old M , replace old M with minimized $G \cup M$, and update the `became` map to describe minimized $G \cup M$.

input graphs. `rcp` represented directly as a partition would necessarily grow linearly with the raw number of vertices of the input, because every input vertex is present in some block. If `became` is implemented as a pointer field in each vertex, and the algorithm is implemented as a library in a garbage-collected environment, and the application calling the library doesn't hold onto references to redundant vertices, the redundant vertices can be freed automatically. More work is needed with with manual storage allocation, but it is still straightforward to discard the redundant vertices. (In order to maximize the effect, the input graph can be rewritten so that every edge reference e is replaced with `became(e)`, and so that duplicate elements are deleted from edge collections.)

`minimize(G)` is defined in Figure 7. It begins with an unnecessarily strict assertion, that $G \cup M$ is closed under `edges`. Our algorithm doesn't depend on this in any essential way, but it is easier to describe the algorithm and its invariants when there are only two sets of vertices (vertices of G and vertices of M) rather than three (vertices of G , vertices of M , and vertices reached from G which are not in either of the other two sets). `minimize(G)` creates an initial partition which corresponds to `partition_by_labels(G)` (represented as `blocks` created as a side effect of looking up `ev` annotation objects), refines the partition one generation r at a time until the partition is consistent with itself and with the partition of M (in the process updating

the taxonomy so that it contains any vertices of G not equivalent to vertices of M), puts any discovered equivalences into the `became` map, and rebalances the taxonomy.

We rebalance our search tree only at the end of partition refinement, where it is more usual to rebalance a search tree continuously, after every insertion. If we rebalanced the persistent hop tree every time a `leaf` was created, we would need to keep the ephemeral `ev`-related data structures up to date in order to continue using them in partition refinement. We avoid that difficulty by putting off rebalancing our persistent data structures until partition refinement is complete, so that we no longer need our ephemeral data structures. Thus, we don't avoid creating the confusion: our postponed rebalancing does still confuse the relationship between the unperturbed trajectories running through the ephemeral data structures and the unperturbed trajectories in the new state of the persistent taxonomy. However, we do avoid the need to care about the confusion, because the confused ephemeral data structures are never used, merely discarded.

3.3.2 Vertex Annotations: `ev`, `fv`, and `gv`

During an incremental partition refinement calculation we associate ephemeral “annotation” data with the vertices we process explicitly: all vertices of G , and those vertices of M which are “frontier vertices.” Here we define that association.

`ev` names both an abstract base class of ephemeral vertex annotations and the function (defined in Figure 8) used to look up the `ev` associated with a vertex.

Every `ev` has fields `vertex` and `antiedges`. The `vertex` field is a reference to a vertex of $G \cup M$, fixed at construction time. The `antiedges` field is a set which accumulates all relevant parents, *i.e.*, the “refined objects” (defined just below) whose `vertex` values immediately reach the `ev.vertex` of the `ev`.

A “refined object” is an object which needs to know the refinement trajectories of edges of some associated vertex; it is either a `gv` or a `block`. `gv` is defined later in this Subsubsection, and its refinement depends on the refinement trajectories of edges of its `gv.vertex`. `block` is defined in Subsubsection 3.3.5, and its refinement depends on the refinement trajectories of edges of its `block.upt.vertex`, *i.e.*, of edges of the unperturbed trajectory of vertices associated with the `block`.

A `fv` annotates a frontier vertex, *i.e.*, a vertex of M which are relevant to

```

ev( $v$ ) :
   $\varepsilon \leftarrow \text{vertex\_ev}[v]$ 
  if  $\varepsilon = \perp$ 
    if  $v \in M$ 
       $\varepsilon \leftarrow \text{new fv}$ 
       $\varepsilon.\text{vertex} \leftarrow v$ 
      set\_sdownt\_and\_enqueue\_reserves( $\varepsilon$ ) /* (Figure 23) */
       $\text{vertex\_ev}[v] \leftarrow \varepsilon$ 
    else
       $\varepsilon \leftarrow \text{new gv}$ 
       $\text{vertex\_ev}[v] \leftarrow \varepsilon$  /* so recursion through edges( $v$ ) terminates */
       $\varepsilon.\text{vertex} \leftarrow v$ 
       $d \leftarrow \text{initial\_kind}(v)$  /* (Figure 22) */
       $\beta \leftarrow \text{block}(d)$  /* (Figure 13) */
       $\varepsilon.\text{block} \leftarrow \beta$ 
       $\beta.\text{gvs} \leftarrow \beta.\text{gvs} \cup \{\varepsilon\}$ 
      for  $t \in \text{edges}(v)$ 
         $\tau \leftarrow \text{ev}(t)$ 
         $\tau.\text{antiedges} \leftarrow \tau.\text{antiedges} \cup \{\varepsilon\}$ 
  return  $\varepsilon$ 

```

Figure 8: The $\text{ev}(v)$ operation: Return an ev for v , constructing it if necessary.

the partition refinement of G because they are immediately reached by refined objects. `fvs` inherit the fields of the `ev` class, and also contain machinery to support replay of the history of their splittings in an earlier partition refinement calculation.

A `gv` annotates a vertex of G . `gv` inherits the fields of `ev`, and also contains machinery to follow the progress of the vertex through blocks as the partition is refined, and to follow the classification of its edges. As in offline partition refinement algorithms,[2, 18] we use doubly linked lists to associate refined vertices with blocks of the partition. As in the offline relational coarsest partition algorithm of Paige and Tarjan,[18] we also maintain a map from possible refinement states to a count of the number of edges of `gv.vertex` which are currently in that refinement state. Paige and Tarjan call their map *count*, and use pointers to blocks as the keys of the map; we call our map `gv.ecounts`, and use references to `kinds` as keys of the map.

Besides the map `gv.ecounts`, which has an analogue in the offline algorithm, in our incremental algorithm we use two related collections which have no analogue, the fields `gv.erises` and `gv.efalls`. `gv.erises` is the set of `gv.ecounts` keys whose mapped-to values have risen from zero to one (or more); `gv.efalls` is the analogous set of keys whose values have fallen to zero. (Accumulating `erises` and `efalls` collections lets us collect small subsets of edges as splitting keys efficiently in graphs where vertex outdegree is large and the number of edge transitions per refinement generation is proportionally small. It is not needed in an offline algorithm because we need such splitting keys only for our persistent taxonomy, and an offline algorithm doesn't need a persistent taxonomy.) Like several other sets in the problem, the `erises` and `efalls` sets might in practice be represented as a LIFO or FIFO queue.

We can use the operation `sdownnt(ε)` (Figure 9) to look up “the block” in which `ε .vertex` in the current generation of partition refinement. “The block” is imaginary in that when `ε` is an `fv`, there might be no actual `block` in the current incremental partition refinement calculation. Thus, `sdownnt(ε)` returns not a `block` but the corresponding `kind`, which will necessarily exist. The mnemonic for the curious name is that *s* is a variable which records the generation of the partition currently being used as a source of splitters, `upt` refers to the end of an unperturbed trajectory, and `downnt` refers to the beginning of an unperturbed trajectory.

```

sdownt( $\varepsilon$ ) :
  assert( $s \geq 0$ ) /* There is no suitable kind to return for  $s = -1$ . */
  if  $\varepsilon$  is an fv
    return  $\varepsilon$ .sdownt
  else /*  $\varepsilon$  is a gv */
     $d \leftarrow \varepsilon$ .block.sdownt
    if  $d.r > s$ 
       $d \leftarrow d$ .superdownt
      assert( $d.r \leq s$ ) /* One adjustment is always enough. */
    return  $d$ 

```

Figure 9: The `sdownt(ε)` operation: Return the `kind` corresponding to the block which holds `ε .vertex` in the naive partition Π_s .

```

refine() :
  while broken_blocks  $\neq \emptyset \vee \neg$ pq_is_empty(reserves)
    refine_once()

```

Figure 10: The `refine(v)` operation: Refine the partition of G to self-consistency.

3.3.3 The `refine()` Operation

The `refine()` operation, defined in Figure 10, refines a partition of the vertices of G so that for each interesting value of r , it corresponds (for all elements of G) to the partition Π_r which would be computed by `naive_rcp($M \cup G$)`.

The set of “interesting values of r ” includes enough values of r to generate all partitions Π_r which differ with respect to vertices of G . It can’t in general include all values of r because in general r can have $O(|M|)$ values, and iterating over $O(|M|)$ values when refining a possibly-small graph G would destroy incremental efficiency. Therefore, the `refine()` algorithm uses a priority queue, `reserves`, to allow r to jump through the taxonomy to potentially-interesting values, ignoring ranges of values which can be shown to be uninteresting for G . The values of r in `reserves` are the generations where the unperturbed trajectory of at least one `fv.vertex` changes. (Thus, “replayed swerves:” an `fv.vertex` is in M not G , so we are replaying classification information from a previous partition refinement, and a “swerve” is

a transition from one unperturbed trajectory to another.)

The set `broken_blocks` contains just-created `blocks`; it is a sort of complement of the set `stressed_blocks` (defined later) of `blocks` whose edges have just swerved. Elements of `stressed_blocks` might be in violation of the invariant that all `gvs` associated with a `block` have the same edge pattern as `block.upt.vertex`. Elements of `broken_blocks` are subdivisions created to restore that invariant. The refinement algorithm alternates between propagating swerved-ness (from elements of `broken_blocks` and/or `reserves`, upward through `antiedges`, into `stressed_blocks`) and creating new swerves (from `stressed_blocks` back into `broken_blocks`).

Though it is simplest to think of the `stressed_blocks` and `broken_blocks` collections as sets, in practice they needn't be implemented as full-blown sets. LIFO or FIFO queues are a good alternative; with a bit of care one can even avoid pushing duplicate entries.

Elements of the `reserves` priority queue are inserted by the `ev(v)` operator (Figure 8) when it creates an `fv`.

Elements of the `broken_blocks` collection are inserted by the `block()` operator (Figure 13) when it creates a `block`.

3.3.4 The `refine_once()` Operation

The `refine_once()` operation, defined in Figure 11, performs one round of partition refinement. As discussed earlier, it increases r to some potentially-interesting value and updates the partition of G to correspond to the state Π_r which would be generated by `naive_rcp($M \cup G$)`.

3.3.5 The `block` Class

In order to define the `stress()` and `bend()` operations, we will need various operations on the `block` class.

Abstractly a `block` corresponds to a block in one of the partitions Π_r of `naive_rcp($G \cup M$)` (where which value r it matches varies as the refinement of G proceeds). In the concrete implementation, a `block` also has 1-1 correspondences with several other values: a `block` β corresponds

- to $\beta.upt$, the `leaf` at the end of the unperturbed trajectory in the taxonomy tree,

```

refine_once() :
  /* Finish initialization of new blocks. */
  postinit_broken() /* (Figure 14) */

  /* Blocks of what generation are being used as splitters? */
  if broken_blocks  $\neq \emptyset$ 
     $s \leftarrow r$  /* generation of new blocks of  $G$ , just as in an offline algorithm */
  else
     $s \leftarrow \text{pq\_top}(\text{reserves}).\text{kind.r}$  /* next relevant generation of  $M$  */
  assert(The partition of  $G$  matches naive  $\Pi_s$  of  $G \cup M$ .)

  /* Consume all elements of broken_blocks and top elements of reserves. */
  assert(stressed_blocks =  $\emptyset$ )
  stress() /* (Figure 15) */
  assert(broken_blocks =  $\emptyset$ )
  assert(ecounts values now match naive  $\Pi_s$  of  $G \cup M$ .)

  /* Blocks of what generation are created in broken_blocks? */
   $r \leftarrow 1 + s$ 

  /* Consume all elements of stressed_blocks. */
  bend() /* (Figure 18) */
  assert(stressed_blocks =  $\emptyset$ )
  assert(The partition of  $G$  matches naive  $\Pi_r$  of  $G \cup M$ .)

```

Figure 11: The `refine_once()` operation: One round of partition refinement

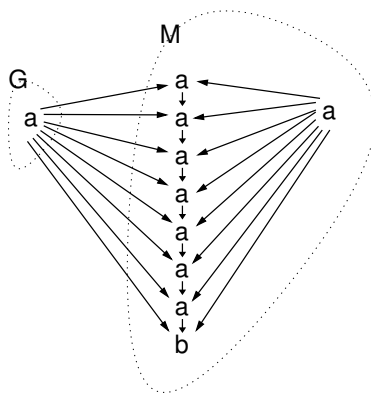


Figure 12: An example graph to motivate differential keys: One member of a family of graph minimization problems where it would be prohibitively expensive to iterate over all edges of a vertex of G at the generations where the edges swerve

- to `β .upt.vertex`, the `leaf.vertex` whose behavior characterizes the unperturbed trajectory, and
- to `β .downt`, that unelided `kind` in the compressed-taxonomy representation of the unperturbed trajectory whose generation r is smallest.

The value `β .upt` is implemented as a field of `block`. The number of edges of each kind of `β .upt.vertex` is implemented as another slot holding a map, `β .ecounts`, and transitions between zero and nonzero values in `β .ecounts` are recorded in `β .erises` and `β .efalls`. The relationship between `β .upt.vertex`, `β .ecounts`, and `β .dzeros` is parallel to that between `gv.vertex`, `gv.ecounts`, and `gv.erises` and `gv.efalls`. Furthermore, the parallel between `block.upt.vertex`, and `gv.vertex` extends to the interpretation of an in an `ev.antiedges` collection. For a `block` in `antiedges`, the raw upstream vertex is `block.upt.vertex`; for a `gv` in `antiedges`, the raw upstream vertex is `gv.vertex`.

Keeping track of `erises` and `efalls` fields lets us construct a splitting key in $O(1)$ time in a generation when $O(1)$ edges of the `block.upt.vertex` change state. That is important, because for a vertex v of outdegree $\#_2 v$ there can be $O(\#_2 v \log \#_1 M)$ such generations where such state change occurs, as in the graph in Figure 12. Any algorithm which required us to iterate over the

```

block(downt) :
   $\beta \leftarrow \text{kind\_block}[\text{downt}]$ 
  if  $\beta = \perp$ 
     $\beta \leftarrow \text{new block}$ 
     $\text{kind\_block}[\text{downt}] \leftarrow \beta$ 
     $\beta.\text{downt} \leftarrow \text{downt}$ 
    /* (leaving block.ecounts for postinit_broken later) */
     $\text{broken\_blocks} \leftarrow \text{broken\_blocks} \cup \{\beta\}$ 
  return  $\beta$ 

```

Figure 13: The `block()` operation

edges of v at every such state change would require more than $O((\#_2G)^2)$ operations to partition G , which would be unacceptably slow.

A `block` has an associated doubly linked list of `gvs`, and each `gv` keeps a reference to its current `block`. We won't spell out the fields and assignments involved in maintaining the doubly linked list, but instead pretend that there is a set of `gvs` stored in a field `block.gvs`. Converting this to a doubly linked list implementation is straightforward: as in offline partition refinement algorithms,[2, 18] it will just happen that we never do any operations on the set abstraction which can't be implemented in $O(1)$ operations in the doubly linked list representation.

We have already defined the set `stressed_blocks` of blocks which have been stressed. Within each block, we also collect the set of `gvs` which have been stressed, in the field `stressed_gvs`. (As in the `stressed_blocks` case, sets are conceptually simple but a real implementation may use LIFO or FIFO queues.)

In general a vertex is considered stressed if any of its edges is on a different unperturbed trajectory in the generation r than it was in the generation $r - 1$, and a block is considered stressed if $\beta.\text{upt.vertex}$ is stressed or its $\beta.\text{stressed_gvs}$ is nonempty. The generation $r = 0$ is a special case, because the generation $r - 1$ is not defined; we say that for $r = 0$, every vertex is stressed.

A `block` is obtained by calling `block(downt)`, defined in Figure 13, which creates a new object if necessary.

It is easier to initialize the `block.ecounts` collection and unperturbed trajectory values if we wait until all `gvs` have been inserted and all blocks

```

postinit_broken() :
  for  $\beta \in \text{broken\_blocks}$ 
     $\gamma \leftarrow$  random choice from  $\beta.\text{gvs}$ 
    if  $\beta.\text{downt.upt.vertex} = \perp$ 
       $\beta.\text{downt.upt.vertex} \leftarrow \gamma.\text{vertex}$ 
       $\text{vertex\_leaf}[\gamma.\text{vertex}] \leftarrow \beta.\text{upt}$  /* (map used in Figure 21) */
    for  $e \in \text{edges}(\beta.\text{downt.upt.vertex})$ 
       $\varepsilon \leftarrow \text{ev}(e)$ 
       $\varepsilon.\text{antiedges} \leftarrow \varepsilon.\text{antiedges} \cup \{\beta\}$ 
      if  $s \geq 0$ 
        Increment  $\beta.\text{ecounts}[\text{sdown}(\varepsilon)]$ .

```

Figure 14: The `postinit_broken()` operation

have advanced to the same generation of partition refinement. Therefore, after we have finished creating all the blocks for this generation, we call `postinit_broken()`, defined in Figure 14.

Note that the cost of `postinit_broken()` grows with the number of edges of $\beta.\text{upt.vertex}$. In order to preserve incremental asymptotic efficiency we will need to guarantee that the number of edges of $\beta.\text{upt.vertex}$ is very seldom much larger than the number of edges of the gv.vertex which forced the creation of the block. If we didn't go out of our way to avoid this situation, it would be easy to create input which made the situation likely. *E.g.*, if M contained mostly vertices with label “a” and outdegree 10^5 , while G contained a single vertex v with label “a” and outdegree 3, the initial partition and perhaps later partitions as well would tend to put v in a block whose upt.vertex had outdegree 10^5 , so we would require more than 10^5 operations to minimize G even though $\#_2 G$ was only 3. Our need to avoid this situation will drive us to complicate our hop trees slightly: instead of using simple random values to balance the persistent taxonomy, we will use random values biased by vertex outdegree (as described in Subsubsection 3.4.3).

3.3.6 The `stress()` Operation

The `stress()` operation (defined in Figure 15) updates `gv.ecounts`, `block.ecounts`, and related quantities like `erises` and `efalls`. Upon return from `stress()`, edge counts are correct and objects whose edge counts

```

stress() :
  for  $\beta \in$  broken_blocks
    for  $\gamma \in \beta$ .gvs
      stress_antiedges( $\gamma$ )
  while pq_top(reserves).kind.r = s
     $\rho \leftarrow$  pq_pop(reserves)
     $\phi \leftarrow \rho$ .fv
     $\phi$ .sdownnt  $\leftarrow$   $\rho$ .kind
    stress_antiedges( $\phi$ )
  broken_blocks  $\leftarrow$   $\emptyset$ 
stress_antiedges( $\varepsilon$ ) :
  for  $\alpha \in \varepsilon$ .antiedges
    if  $\alpha$  is of type gv
      stress_gv_as_antiedge( $\alpha, \varepsilon$ )
    else
      stress_block_as_antiedge( $\alpha, \varepsilon$ )

```

Figure 15: The `stress()` operation: Update ecounts and related data.

changed between zero and nonzero values have been collected for processing by `bend()`.

The `stress()` operation calls `stress_gv_as_antiedge()` and `stress_block_as_antiedge()`, defined in Figures 16 and 17. `stress_gv_as_antiedge()` and `stress_block_as_antiedge()` have almost identical structure, and in a real implementation this commonality should be factored into a shared block of code. We don't factor it out in our generic pseudocode only because the natural way to factor it out depends on implementation details. *E.g.*, if the implementation language supports multiple inheritance, `gv` might inherit both from `ev` and of `antiedge` (and perhaps from a third base class `element_of_doubly_linked_list` as well, to implement its relationship with `block`), `block` could also inherit from `antiedge`, and the two operations might be rewritten as a single operation `stress_as_antiedge`. Or given precise syntax for creating and using writable references (as in many real programming languages, but not in pseudocode) it is straightforward to implement the two operations as calls to a shared function, passing as function arguments writable references to the relevant data.

```

stress_gv_as_antiedge( $\gamma, \varepsilon$ ) :
  assert( $\varepsilon.\text{vertex} \in \text{edges}(\gamma.\text{vertex})$ )
   $d_s \leftarrow \text{sdownnt}(\varepsilon)$  /* (Figure 9) */
   $d_{\text{previous}} \leftarrow d_s.\text{superdownnt}$ 
  if  $d_{\text{previous}} \neq \text{nil}$ 
    assert( $\gamma.\text{ecounts}[d_{\text{previous}}] > 0$ )
    Decrement  $\gamma.\text{ecounts}[d_{\text{previous}}]$ .
    if  $\gamma.\text{ecounts}[d_{\text{previous}}] = 0$ 
       $\gamma.\text{efalls} \leftarrow \gamma.\text{efalls} \cup \{d_{\text{previous}}\}$ 
  Increment  $\gamma.\text{ecounts}[d_s]$ .
  if  $\gamma.\text{ecounts}[d_s] = 1$ 
     $\gamma.\text{erises} \leftarrow \gamma.\text{erises} \cup \{d_s\}$ 
  assert(( $\gamma.\text{erises} \cup \gamma.\text{efalls}$ )  $\neq \emptyset$ )
   $\beta \leftarrow \gamma.\text{block}$ 
   $\beta.\text{stressed_gvs} \leftarrow \beta.\text{stressed_gvs} \cup \{\gamma\}$ 
   $\text{stressed\_blocks} \leftarrow \text{stressed\_blocks} \cup \{\beta\}$ 

```

Figure 16: The `stress_gv_as_antiedge(ε)` operation: Update `$\gamma.\text{ecounts}$` and related data to reflect the new kind of `$\varepsilon.\text{vertex}$` (which is one of the edges of `$\gamma.\text{vertex}$`).

```

stress_block_as_antiedge( $\beta, \varepsilon$ ):
  assert( $\varepsilon$ .vertex  $\in$  edges( $\beta$ .upt.vertex))
   $d_s \leftarrow$  sdown( $\varepsilon$ ) /* (Figure 9) */
   $d_{\text{previous}} \leftarrow d_s$ .superdown
  if  $d_{\text{previous}} \neq \text{nil}$ 
    assert( $\beta$ .ecounts[ $d_{\text{previous}}$ ] > 0)
    Decrement  $\beta$ .ecounts[ $d_{\text{previous}}$ ].
    if  $\beta$ .ecounts[ $d_{\text{previous}}$ ] = 0
       $\beta$ .efalls  $\leftarrow$   $\beta$ .efalls  $\cup$  { $d_{\text{previous}}$ }
  Increment  $\beta$ .ecounts[ $d_s$ ].
  if  $\beta$ .ecounts[ $d_s$ ] = 1
     $\beta$ .erises  $\leftarrow$   $\beta$ .erises  $\cup$  { $d_s$ }
  assert(( $\beta$ .erises  $\cup$   $\beta$ .efalls)  $\neq$   $\emptyset$ )
  stressed_blocks  $\leftarrow$  stressed_blocks  $\cup$  { $\beta$ }

```

Figure 17: The `stress_block_as_antiedge(β, ε)` operation: Update `β .ecounts` and related data to reflect the new kind of `ε .vertex` (which is one of the edges of `β .upt.vertex`).

`stress()` updates `gv.ecounts`, `block.ecounts`, and related properties such as `erises` and `efalls`. Upon return from `stress()`, edge counts are correct and objects whose edge counts changed between zero and nonzero values have been collected for processing by `bend()`.

3.3.7 The `bend()` Operation

The `bend()` operation (defined in Figure 18) enforces the invariant that `block.gvs` elements have `gv.ecounts` patterns which match `block.ecounts`, moving `gvs` to new `blocks` as necessary to achieve this.

As in the offline algorithm of Paige and Tarjan,[18] this match is only with respect to equal nonzeroness of `ecounts`, not with respect to exact equality of `ecounts`. (The relational coarsest partition doesn't distinguish one copy of an edge from many copies, it only distinguishes zero and nonzero: thus, *e.g.*, in Figure 2 vertex `b2` is equivalent to vertex `b3`.)

The test

$$\text{if } \beta.\text{erises} = \emptyset \wedge \beta.\text{erises} = \emptyset$$

in `bend_block` arranges that the algorithm doesn't call `bend_gv` too many

```

bend() :
  for  $\beta \in \text{stressed\_blocks}$ 
    bend_block( $\beta$ )
  stressed_blocks  $\leftarrow \emptyset$ 
bend_block( $\beta$ ) :
  if  $\beta.\text{erises} = \emptyset \wedge \beta.\text{erises} = \emptyset$ 
    bend_gvs( $\beta.\text{stressed\_gvs}$ )
  else
    bend_gvs( $\beta.\text{gvs}$ )
     $\beta.\text{stressed\_gvs} \leftarrow \emptyset$ 
     $\beta.\text{erises} \leftarrow \emptyset$ 
     $\beta.\text{efalls} \leftarrow \emptyset$ 
bend_gvs(gvs) :
  for  $\gamma \in \text{gvs}$ 
    bend_gv( $\gamma$ )

```

Figure 18: The `bend()` operation: Force `gv.ecounts` values to correspond to `block.ecounts` values.

times, but does call `bend_gv` when an edge of the unperturbed trajectory swerves (even when no edges of the `gv.vertex` swerve). The expected number of times that a `gv` for a vertex v will be in `$\beta.\text{stressed_gvs}$` during partition refinement is $O(\#_2 v (\log \#_2 v) (\log \#_1 M))$; this is a bound on the number of times that a `gv` will pass through the condition-is-true branch of the `if`. (The extra factor of $(\log \#_2 v)$ arises from a transformation which will be introduced in Subsubsection 3.4.3.) Every other time that a `gv` passes through `bend_gv`, it will be when `$\beta.\text{erises} \cup \beta.\text{efalls}$` is nonempty while `$\text{gv.erises} \cup \text{gv.efalls}$` is empty, and thus the `gv` will swerve. The expected number of swerves for a `gv` over partition refinement is $O((\log \#_2 v) (\log \#_1 M))$, which is dominated by the expected number of times that the `gv` will enter `$\beta.\text{stressed_gvs}$` , above. Thus the sum over all G of the two terms is $O(\#_2 G (\log \#_2 G) (\log \#_1 M))$.

3.3.8 The `diffdiff((A_\oplus, A_\ominus), (B_\oplus, B_\ominus))` Operation

Each of our splitting keys is a pair of a positive-sense set of `kind` objects with a negative-sense set of `kind` objects, representing a difference between ordinary sets. In two places (in Figure 19 when creating ephemeral keys, and

```

bend_gv( $\gamma$ ) :
   $\beta \leftarrow \gamma.\text{block}$ 
   $\sigma \leftarrow \text{diffdiff}((\gamma.\text{erises}, \gamma.\text{efalls}), (\beta.\text{erises}, \beta.\text{efalls}))$ 
  if  $\sigma \neq (\emptyset, \emptyset)$  /* if  $\gamma$  is distinguishable from unperturbed trajectory */
     $\beta.\text{gvs} \leftarrow \beta.\text{gvs} \setminus \{\gamma\}$ 
     $d \leftarrow \beta.\text{downt}$ 
     $d' \leftarrow \text{subdownt}(r, \sigma, d)$  /* (Figure 26) */
     $b' \leftarrow \text{block}(d')$ 
     $\gamma.\text{block} \leftarrow b'$ 
     $b'.\text{gvs} \leftarrow b'.\text{gvs} \cup \{\gamma\}$ 
   $\gamma.\text{erises} \leftarrow \emptyset$ 
   $\gamma.\text{efalls} \leftarrow \emptyset$ 

```

Figure 19: The `bend_gv(γ)` operation: One gv worth of the `bend()` operation

```

diffdiff(( $A_{\oplus}, A_{\ominus}$ ), ( $B_{\oplus}, B_{\ominus}$ )) :
  assert( $A_{\oplus} \cap A_{\ominus} = A_{\oplus} \cap B_{\ominus} = B_{\oplus} \cap A_{\ominus} = B_{\oplus} \cap B_{\ominus} = \emptyset$ )
   $D_{\oplus} \leftarrow (A_{\oplus} \setminus B_{\oplus}) \cup (B_{\ominus} \setminus A_{\ominus})$ 
   $D_{\ominus} \leftarrow (A_{\ominus} \setminus B_{\ominus}) \cup (B_{\oplus} \setminus A_{\oplus})$ 
  return ( $D_{\oplus}, D_{\ominus}$ )

```

Figure 20: The `diffdiff((A_{\oplus}, A_{\ominus}), (B_{\oplus}, B_{\ominus}))` operation: The difference of two set differences (given that both arguments are differences relative to a third set so that the difference can still be expressed in pair-of-sets form).

in Figure 28 when adjusting stable keys when rebalancing the taxonomy) we compute differences of such differences, using the operation `diffdiff(...)`, defined in Figure 20.

To motivate the set operations in `diffdiff(...)`, it may be helpful to think of a stable key $(S_{\oplus}, S_{\ominus})$ as representing a map from objects of type `kind` to one of the three values $+1, 0$, and -1 . The elements of the set S_{\oplus} are those objects which map to $+1$, the elements of the set S_{\ominus} are those objects which map to -1 , and those objects which map to 0 are in neither set. Then the result of `diffdiff(A, B)` is a map which represents the arithmetic difference of the mapped-to values of A and B . And the method of construction of the arguments A and B , where A represents the difference between two edge patterns π_0 and π_1 , and B represents the related difference between π_1 and a third edge pattern π_2 , guarantees that the algorithm will never ask `diffdiff(...)` to do the impossible by representing a map to values outside $-1, 0, 1$.

3.4 Persistent Data Structures; the Taxonomy

3.4.1 Design Constraints

Before we define our taxonomy data structure, we'll review constraints which have been imposed on it by our design so far, noting the three requirements which most strongly tend to make the design interesting.

The taxonomy must be compressed into space $O(\#_1M + \#_2M)$.

The taxonomy must support looking up a `kind` corresponding to the initial partition of a vertex v , creating it if necessary. This is the operation `initial_kind(v)`, called from Figure 8 and defined in Figure 22.

The taxonomy must support lookup rootward from a `leaf` to its `downt`, the start of the unperturbed trajectory.

The taxonomy must allow a lookup leafward from a `kind` to `upt`, the `leaf` at the end of the unperturbed trajectory.

Given a leaf, the taxonomy must allow the refinement history to be read in a form suitable for setting up the `reserves` priority queue used to replay partition refinement. Maintaining a `kind.superdownt` field makes this easy.

(Because we maintain our `kind.superdownt` field value in order to support reading refinement history, it is convenient to use the field also when calculating d_{previous} in `stress_gv_as_antiedge` (Figure 16) and `stress_block_as_antiedge` (Figure 17), but the uses of this field in those

calculations do not seem to be essential. Probably if the field didn't exist, it would be possible to remember d_{previous} values from a little earlier, and pass it in as an argument.)

Given a **downt** (*i.e.*, a **kind** which happens to be at the beginning of its unperturbed trajectory) d and a splitting key σ , the taxonomy must be able to return a corresponding **subdownt**, inserting a new branch if necessary. This is the operation **subdownt**(r, σ, d), used in Figure 19 and defined in Figure 26. This is the first requirement that helps to make the design interesting, because it interacts strongly with balancing and rebalancing requirements: any rebalance-related design decisions, especially, tend to have repercussions in the implementation of **subdownt**.

The taxonomy must be rebalanceable efficiently after a partition refinement calculation. Note that besides the obvious local effects, this requirement constrains our choice of splitting keys. The number of keys which refer to an edge vertex of **kind** k can be very large, and we don't want our rebalancing to require us to rewrite a very large set of keys very often. This is the second requirement that tends to make the design interesting.

For the efficiency reasons discussed in Subsubsection 3.3.5, the taxonomy must avoid "arity skew": the taxonomy must make it very uncommon for a small-outdegree vertex of G to have as its unperturbed trajectory a huge-outdegree vertex of M . This is the third requirement that tends to make the design interesting. As the name connotes, we will conflate this requirement with the ordinary balancing requirement, choosing a balancing weight which causes our search tree to satisfy this requirement as long as it is balanced.

3.4.2 Satisfying Uninteresting Design Constraints

The implementation of the uninteresting requirements from the previous subsection follows, without much thought, from the general architectural ideas in Subsection 3.2.

We implement the **leaf.downt** and **kind.upt**, and **kind.superdownt** relationships as class fields, creating up-to-date-ness invariants which must be preserved as we create objects or rebalance the taxonomy.

We also need a map relating minimized vertices to their associated **leaf**, so that **set_sdownt_and_enqueue_reserves**(ϕ) will be able to find taxonomic information about $\phi.\text{vertex}$. We share this map from minimal vertex to **leaf** with the map which implements the **became**(v) operator, as shown in Figure 21.

```

became( $v$ ) :
    return vertex_leaf[ $v$ ].vertex
set_redundant_became() :
    /* Set vertex_leaf[ $v$ ] for redundant vertices. */
    for  $v \in G$ 
         $\gamma \leftarrow \text{ev}(v)$ 
        /* (When  $v = \gamma.\text{block.upt}$ , this is a no-op, setting vertex_leaf[ $v$ ] */
        /* to the same value already set in Figure 14.) */
        vertex_leaf[ $v$ ]  $\leftarrow \gamma.\text{block.upt}$ 

```

Figure 21: The `became(v)` and `set_redundant_became()` operations

```

initial_kind( $v$ ) :
     $d \leftarrow \text{label\_kind}[\text{label}(v)]$ 
    if  $d = \perp$ 
         $d \leftarrow \text{new kind}$ 
        label_kind[label( $v$ )]  $\leftarrow d$ 
         $d.\text{superdownt} \leftarrow \text{nil}$ 
         $d.\text{r} \leftarrow 0$ 
        /* ( $d.\text{aw}$  and  $d.\text{perfect\_hash}$  are set later, in Figure 24.) */
         $\lambda \leftarrow \text{new leaf}$ 
         $d.\text{upt} \leftarrow \lambda$ 
         $\lambda.\text{downt} \leftarrow d$ 
        /* ( $\lambda.\text{vertex}$  is set later, in Figure 14.) */
        /* ( $\lambda.\text{aw}$  is set later, in Figure 24.) */

```

Figure 22: The `initial_kind(v)` operation: Return the `kind` corresponding to that block of Π_0 which contains v .

```

set_sdownnt_and_enqueue_reserves( $\phi$ ) :
   $d \leftarrow \text{vertex\_leaf}[\phi.\text{vertex}].\text{sdownnt}$ 
  while  $d \neq \text{nil} \wedge d.r > s$ 
     $\rho \leftarrow \text{new reserve}$ 
     $\rho.\text{fv} \leftarrow \phi$ 
     $\rho.\text{kind} \leftarrow d$ 
    insert( $\rho, \text{reserves}$ )
     $d \leftarrow d.\text{sdownnt}$ 
   $\phi.\text{sdownnt} \leftarrow d$ 

```

Figure 23: The `set_sdownnt_and_enqueue_reserves(ϕ)` operation: Read the refinement history for (the frontier vertex of M described by) ϕ , and use the history to set $\phi.\text{sdownnt}$ and to create ϕ -related entries in the `reserves` queue.

We implement `initial_kind(v)` in Figure 22.

We implement `set_sdownnt_and_enqueue_reserves(ϕ)` in Figure 23.

3.4.3 Avoiding Arity Skew; the `aw` Weight

In this subsection we will postpone the first two of our interesting design requirements while we address the third requirement. We will define a new arity-corrected weight `aw` with balancing properties similar to the w defined in Subsection 3.2, and also with the additional property that search in an `aw`-balanced tree avoids arity skew. Having done that, we will have only two interesting design requirements left, and they will be formally just the same as the first two interesting requirements: we just define our unperturbed trajectories in terms of the new total order on `aw` (replacing the old total order on bare w), leaving the problem formally unchanged.

Our solution here has the advantage of being simple to implement (a few lines of code corresponding to Equations 13, 14, 15, and 16), but it suffers from the disadvantage of contributing several logarithmic terms to the asymptotic runtime cost of the algorithm. For a possible way to make the opposite tradeoff, eliminating the extra logarithmic terms by complicating the algorithm and its data structures, see Subsubsection 3.5.1.

We define a weight `aw` for a minimized vertex v (such as a vertex which

appears in a `leaf.vertex` field in our taxonomy) to be a pair,

$$\mathbf{aw}(v) \equiv (a(v), w). \quad (13)$$

Here, just as in the original weight system, w is a random number generated once some time after the allocation of `leaf`. $a(v)$ is the integer part of the logarithm of the minimized arity,

$$a(v) \equiv \left\lfloor \frac{\log_B A(v)}{g} \right\rfloor, \quad (14)$$

where the base $B > 1$ is some arbitrary constant and the minimized arity is the number of inequivalent edges of v , which we can calculate by counting the number of distinct values of `became` after minimization,

$$A(v) \equiv \#\{\forall e \in \mathbf{edges}(v) : \mathbf{became}(e)\}. \quad (15)$$

The pairs are ordered lexicographically:

$$((a, w) < (a', w')) \equiv a < a' \vee (a = a' \wedge w < w'). \quad (16)$$

A direct application of the expression in Equation 14 will fail for vertices which have no edges. This can be handled by approximating the logarithm as a huge negative value, by adding an offset of $+1$ to the logarithm argument, or by performing the initial partition both by label and by nonzeroness of outdegree and handling the zero-outdegree vertices specially because they don't need any partition refinement.

When we use `aw` instead of w to balance our taxonomy, the general effect is that in partition refinement of a vertex $v \in G$, we pass through as many as $1 + a(v)$ layers of balanced trees. All the vertices $v' \in G$ which have $a(v') = 0$ are in the first layer, all the vertices of $v' \in G$ which have $a(v') = 1$ are in the second layer, and so forth. Any layer might of course be empty, but in general each layer can have $O(\#_1 M)$ vertices. Thus, since each layer is a balanced tree, the overall expected number of swerves in each layer is $O(\log \#_1 M)$, and the expected number of swerves in classifying v is $O((1 + a(v)) \log \#_1 M)$. Because $v \in G$, we know that $A(v) \leq \#_2 G$, so we can bound the expected number of swerves in classifying v as $O((\log \#_2 G)(\log \#_1 M))$.

Because of the way that in partition refinement the unperturbed trajectory tends to be first vertices with $a(v) = 0$, then $a(v) = 1$, and so forth, a small-arity vertex v of G will tend never to see a huge-arity vertex of M as

its unperturbed trajectory. In the simple implementation shown in the pseudocode, the effect is a only an amortized bound on the cost of iterating over the edges of unperturbed trajectory vertices is $O(\#_2 v)$: for any given huge-outdegree vertex v_{huge} of M with $a(v_{\text{huge}}) = a_{\text{huge}}$, there can be one expensive `minimize(G)` operation for each value $\alpha \in \{a_{\text{huge}} - 1, a_{\text{huge}} - 2, \dots\}$ for which the graph G contains smaller-outdegree vertices (with $a(v) = \alpha$) which see v_{huge} as their unperturbed trajectory. Each such expensive `minimize(G)` operation will insert its smaller-outdegree vertex in the taxonomy tree in such a way that afterwards the huge-outdegree vertex will no longer be able to cause any subsequent `minimize(G)` to be expensive. Thus the worst case is for v_{huge} to cause one expensive `minimize(G)` operation for each element of some strictly decreasing sequence of α values. But because the outdegree decays exponentially through that sequence, the size of the summed problem is only of the order of the outdegree of v_{huge} .

If for some reason this amortized guarantee is insufficient, it would be straightforward to modify the code to provide a non-amortized guarantee. `rebalance` logic could be modified so that it immediately created enough dummy kinds (or even full-blown dummy vertices, perhaps using dummy labels so that dummy labels would never completely match input vertices) to solve the problem immediately, before returning from `minimize(G)`. That is, whenever `rebalance` detected that a single swerve would increase a of the unperturbed trajectory by more than 1, it would eagerly pay the price of creating enough taxonomy entries that no single swerve increased a by more than 1. Because of the exponential decay in size with decreasing a , the sum of the sizes of dummy vertices required would only be of the same order as the size of the huge-outdegree vertex which caused the placeholders to be allocated.

Having settled on this balancing weight scheme, it is straightforward to give (in Figure 24) a concrete implementation of the top levels of the `rebalance()` operation which was called from Figure 7. These top-level definitions invoke the lower-level operation `superuptify1(d)` to cause $d.\text{superupt}$ to assume from d the role of being the beginning of the unperturbed trajectory running to $d.\text{upt}$. Now our design problem has been reduced to arrange for the taxonomy tree to support both the `subdownnt(r, σ, d)` operation introduced earlier and the `superuptify1(d)` operation just introduced here.

Note that in the special case of graphs with labelled edges, it is easy to eliminate all analytical complexity and runtime cost associated with `aw`

```

rebalance() :
  for  $v \in G$ 
     $\lambda \leftarrow \text{vertex\_leaf}[v]$ 
    if  $\lambda.\text{vertex} = v$ 
      assert( $\lambda.\text{aw} = \perp$ )
       $w \leftarrow \text{random number}$ 
       $\lambda.\text{aw} = (a(v), w)$ 
      maybe_superuptify( $\lambda.\text{downt}$ )
maybe_superuptify( $d$ ) :
   $d_{\text{super}} \leftarrow d.\text{superdownt}$ 
  if  $d_{\text{super}} \neq \text{nil}$ 
     $\text{aw}_{\text{super}} \leftarrow d_{\text{super}}.\text{upt}.\text{aw}$ 
    if  $\text{aw}_{\text{super}} = \perp \vee d.\text{upt}.\text{aw} < \text{aw}_{\text{super}}$ 
      superuptify1( $d$ ) /* (Figure 28) */
      maybe_superuptify( $d_{\text{super}}$ )

```

Figure 24: The `rebalance()` operation: Rebalance the taxonomy with respect to new leafs (after first completing initialization of the `aw` data required).

weights. In this special case it remains important to avoid arity skew, but arity skew be eliminated completely at $O(1)$ runtime cost by modifying the initial partition. For graphs with labelled edges, the outdegree of a vertex is constant in partition refinement, and so we can replace our initial `partition_by_labels()` with a partition by equality of the pair (`label`, outdegree). Having done that, the outdegree of any unperturbed trajectory vertex u is always equal to the outdegree of the corresponding $v \in G$ being refined, so $a = a'$ in every comparison using Equation 16, so the total order of `aw` behaves exactly like the total order of bare w .

3.4.4 The `subdownt`(r, σ, d) Operation

In order to support our insertion and rebalancing operations, we make two choices of representation.

First, we choose to use sets of `kinds` of the splitting partition generation s to represent splitting keys stored in persistent data structures. This differs slightly from the sets of `kinds` which were generated in the ephemeral calculations and passed to `subdownt`(r, σ, d): in the ephemeral calculations it was more natural always to work with those `kinds` which in the current balance of the taxonomy are at the beginnings of their unperturbed trajectories. The translation is straightforward, and the benefit is that the translated keys are more stable under rebalancing operations.

Second, we choose to represent each branch in the taxonomy, corresponding to a `subdownt`(r, σ, d) lookup, as a two-level map rooted at the `downt` of the unperturbed trajectory. The outer map, called `heirs`, is keyed by partition refinement generation r ; the inner map, called `siblings`, is keyed by σ .

As a consequence of these choices, our lookup operation is approximately

$$\text{subdownt}(r, \sigma, d) \approx d.\text{heirs}[r].\text{siblings}[\text{stable_key}(r, \sigma)]. \quad (17)$$

The exact operation differs slightly because of the same-map optimization described in the next paragraph, and is substantially more complicated because it must handle on-demand creation of new `kinds`.

The `d.heirs` map is only used when d is the beginning (or “`downt`”) of its unperturbed trajectory, and the `d.siblings` map is only used when d is not the beginning of its unperturbed trajectory. Therefore, any given d only uses one of the two maps, so the implementation of the two maps can

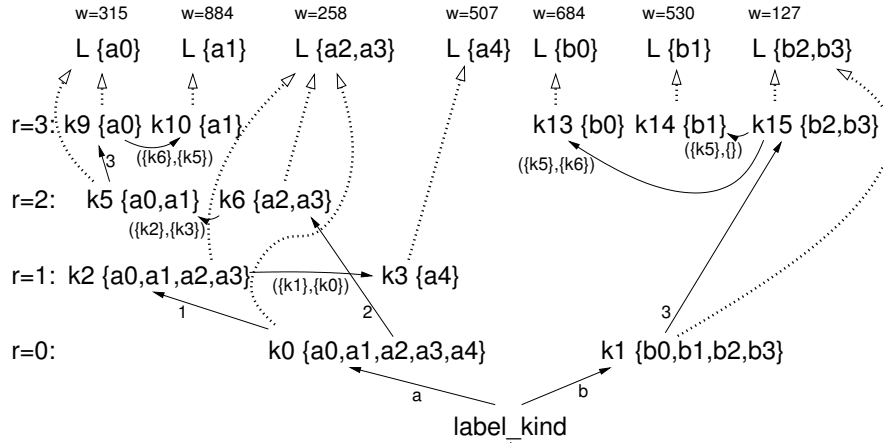


Figure 25: The physical leafward links of the taxonomy of Figure 5. Map values (in `label_kind` and `kind relatives`) are shown as solid arrows, `kind.upt` values are shown as dotted arrows.)

share a single field of `kind`. We call that shared field `kind relatives`, and Equation 17 becomes

$$\text{subdownt}(r, \sigma, d) \approx d.\text{relatives}[r].\text{relatives}[\text{stable_key}(r, \sigma)]. \quad (18)$$

Figure 25 shows the `relatives` and `upt` links used to implement `subdownt(r, σ, d)` in the example taxonomy of Figure 5 and earlier.

It is straightforward to write code which preserves the read-existing-`kind` behavior of Equation 18 while also creating new subkinds on demand; we do so in Figure 26.

The `stable_key(σ, s)` operation (Figure 26) translates `σ` to a key which is relatively easy to adjust when we rebalance the hop tree. The basic idea of the translation is that within our `stress` and `bend` calculations it is most natural always to represent a block of `naive_rcp` by its associated most-rootward `kind`, “its `downt`,” but that that is not a good representation in a key of the the persistent taxonomy because rebalancing the tree can cause the `downt` to change and we want to avoid having to rewrite the contents of distant keys when we rebalance the tree. Instead, we translate all `kind` objects in `σ` into the corresponding `kinds` at generation `s`, giving a key which doesn’t need to be changed when one of its elements changes between `downtness` and `nondowntness`.

```

subdownt( $r, \sigma, d$ ) :
  assert( $\sigma \neq (\emptyset, \emptyset)$ )
   $\Sigma \leftarrow \text{stable\_key}(\sigma, 1 - r)$  /* (Figure 27) */
  return sibling( $\Sigma, \text{heir}(r, d), d$ )
heir( $r, d$ ) :
   $h \leftarrow d.\text{relatives}[r]$ 
  if  $h = \perp$ 
     $h \leftarrow \text{new kind}$ 
     $d.\text{relatives}[r] \leftarrow h$ 
     $h.r \leftarrow r$ 
     $h.\text{superdownt} \leftarrow d.\text{superdownt}$ 
     $h.\text{upt} \leftarrow d.\text{upt}$ 
  return  $h$ 
sibling( $\Sigma, h, d_{\text{super}}$ ) :
   $d \leftarrow h.\text{relatives}[\Sigma]$ 
  if  $d = \perp$ 
     $d \leftarrow \text{new kind}$ 
     $h.\text{relatives}[\Sigma] \leftarrow d$ 
     $d.r \leftarrow r$ 
     $d.\text{superdownt} \leftarrow d_{\text{super}}$ 
     $\lambda \leftarrow \text{new leaf}$ 
     $d.\text{upt} \leftarrow \lambda$ 
     $\lambda.\text{downt} \leftarrow d$ 
    /* ( $\lambda.\text{vertex}$  is set later, in Figure 14.) */
    /* ( $\lambda.\text{aw}$  is set later, in Figure 24.) */
  return  $d$ 

```

Figure 26: The **subdownt**(r, σ, d) operation: Return the beginning kind for that unperturbed trajectory whose edge pattern differs from $d.\text{upt}$ by σ .

```

stable_key(( $\sigma_{\oplus}$ ,  $\sigma_{\ominus}$ ),  $s$ ) :
    return (stable_kinds( $\sigma_{\oplus}$ ,  $s$ ), stable_kinds( $\sigma_{\ominus}$ ,  $s$ ))
stable_kinds( $D$ ,  $s$ ) :
    return { $\forall d \in D : \text{stable\_kind}(d, s)$ }
stable_kind( $d$ ,  $s$ ) :
    if  $d.r = s$ 
        return  $d$ 
    else
        return  $d.\text{relatives}[s]$ 

```

Figure 27: The $\text{stable_key}((\sigma_{\oplus}, \sigma_{\ominus}), s)$ operation: Translate the kinds of $(\sigma_{\oplus}, \sigma_{\ominus})$ into kinds of refinement generation s .

Even these relatively stable keys still need to be rewritten when their own unperturbed trajectory changes: the key is only stable with respect to distant rebalancing. A key in $d.\text{siblings}$ represents the difference between some swerved subkind d' and the unperturbed trajectory which passes through d . Thus, a change of its own local trajectory (*i.e.*, of the value of $d.\text{upt}$) will require the use of a new key. Fortunately the results of Appendix C show that the expected number of stable keys which must be rewritten upon inserting a new leaf is tolerably low.

3.4.5 The $\text{superuptify1}(d)$ Operation

It now remains only to implement the operation of a **kind** taking from some other **kind** in the role of immediate next step on the unperturbed trajectory of its superkind: the operation $\text{superuptify1}(d)$, used in Figure 24.

The implementation of this operation is an exercise in coding and proof-reading, not design: its required behavior follows directly from the various invariants required by earlier design decisions. The result is given in Figure 28.

3.5 Asymptotic Efficiency

In the limit when M and G are of similar size, the performance of our algorithm is messy but also uninteresting: one could simply fall back to running the algorithm of Paige and Tarjan on $G \cup M$ (and translating the result into our taxonomy). Therefore we will only analyze our algorithm in the

```

superuptify1(d) :
  r ← d.r
  u ← d.upt
  dsuper ← d.superdownt
  u0 ← dsuper.upt
  d0 ← dsuper.relatives[r]
  d.superdownt ← dsuper.superdownt
  d0.superdownt ← dsuper
  dsuper.upt ← u
  dsuper.relatives[r] ← d
  u.downt ← dsuper
  u0.downt ← d0
   $\rho$  ← new empty map
  (D+, D-) ← the key such that d0.relatives[(D+, D-)] = d
  for  $\Sigma \in$  keys of d0.relatives
    if  $\Sigma = (D_+, D_-)$ 
       $\rho[(D_-, D_+)] \leftarrow d_0$ 
    else
       $\Sigma' \leftarrow \text{diffdiff}(\Sigma, (D_+, D_-))$  /* (Figure 20) */
       $\rho[\Sigma'] \leftarrow d_0.\text{relatives}[\Sigma]$ 
      d0.relatives[ $\Sigma$ ] ←  $\perp$ 
  acquire_heirs_above_r(d0, dsuper) /* (Figure 29) */
  for h ∈ mapped-to values of dsuper.relatives
    h.upt ← u
  acquire_heirs_above_r(dsuper, d) /* (Figure 29) */
  d.relatives ←  $\rho$ 

```

Figure 28: The `superuptify1(d)` operation: Adjust the taxonomy so that `d.superdownt.upt ← d.upt` and all invariants are preserved.

```

acquire_heirs_above_r( $d, d_{\text{previous}}$ ) :
  for  $r \in \text{keys of } d_{\text{previous}}.\text{relatives}$ 
    if  $r > d.r$ 
       $h \leftarrow d_{\text{previous}}.\text{relatives}[r]$ 
       $d_{\text{previous}}.\text{relatives}[r] \leftarrow \perp$ 
       $d.\text{relatives}[r] \leftarrow h$ 
       $h.\text{upt} \leftarrow d.\text{upt}$ 
       $h.\text{superdownt} \leftarrow d.\text{superdownt}$ 
      for  $d' \in \text{mapped-to objects in } h.\text{relatives}$ 
         $d'.\text{superdownt} \leftarrow d$ 
        for  $h' \in \text{mapped-to objects in } d'.\text{relatives}$ 
           $h'.\text{superdownt} \leftarrow d$ 

```

Figure 29: The `acquire_heirs_above_r(d, d_{previous})` operation: Transfer from d_{previous} to d any heirs whose `kind.r` is sufficiently large.

limit where G is so small compared to M that an incremental calculation is worthwhile.

We can implicitly preprocess G into $G' = \forall_{v \in G} : \#_2 v > 0$. No modification of the algorithm is necessary, simply notice that in our algorithm a vertex with no edges is processed in $O(1)$ operations by being classified in the initial partition and left (basically) untouched thereafter. (It has a slight effect on processing of other vertices with the same label, but that effect can be amortized as part of the cost of processing the vertices with edges.) Thus, the asymptotic cost of `minimize(G)` is the $O(\#_1 G)$ “preprocessing” operations plus the cost of `minimize(G')`. This transformation makes it simpler to express the asymptotic cost because $\#_1 G' \leq \#_2 G' = \#_2 G$, so generally we can avoid carrying separate $\#_1 G'$ terms and only work with $\#_2 G$.

The cost of `minimize(G')` is dominated by the processing of edges. Processed edges appear in two ways: by being edges of G , or by being edges of the unperturbed trajectory vertices which appear in the classification of vertices of G . Each vertex $v \in G'$ passes through $O(\log(\#_2 v) \log(\#_1 M))$ classifications in partition refinement: as many as $a(v) = \log(\#_2 v)$ layers of balanced trees in the persistent taxonomy, each of which has depth $O(\log(\#_1 M))$ (and some relatively negligible number of splits in the ephemeral part of the taxonomy, because $\#_1 G \ll \#_1 M$). The individual values $\log(\#_2 v)$ are difficult to sum, so we bound them conservatively with $\log(\#_2 v) \leq \log(\#_2 G)$. This

gives a simple bound on the number of classifications of a vertex of G' in partition refinement,

$$n_{cg} = O(\log(\#_2 G) \log(\#_1 M)). \quad (19)$$

In the partition refinement of a vertex $v \in G'$, our **aw** balancing ensures that the corresponding unperturbed trajectory vertex u has outdegree of the same order as the outdegree of v . Then we can say that the number of processed edges is

$$n_{pe} = O(\#_2 G n_{cg}) = O(\#_2 G \log(\#_2 G) \log(\#_1 M)). \quad (20)$$

The most expensive case for processing an edge is when the edge runs to a frontier vertex. We count two related subcases, the number of “frontier edges” which run to frontier vertices,

$$n_{fe} \leq n_{pe}, \quad (21)$$

and the number of distinct frontier vertices (which is also the number of **fv** annotation objects),

$$n_{fv} \leq n_{fe}. \quad (22)$$

For each **fv** object, we must create one **reswerve** object to replay each of its

$$n_{cm} = O(\log(\#_2 M) \log(\#_1 M)) \quad (23)$$

future (relative to s of the ongoing calculation) partially refined classifications in the partition refinement of M . Each of those **reswerve** objects must be inserted and removed from the priority queue **reswerves**, at a cost of

$$n_{pq} = O(\log n_{fv}) = O(\log(\#_2 G \log \#_1 M)) \quad (24)$$

for each removal. That is the dominant term of our asymptotic cost. Thus, the cost of **minimize**(G') is $O(n_{fv} n_{cm} n_{pq})$, and the cost of **minimize**(G) is

$$\begin{aligned} n_G &= O(\#_1 G + n_{fv} n_{cm} n_{pq}) \\ &= O(g_1 + g_2(\log g_2)(\log m_1)^2(\log m_2)(\log(g_2(\log m_1)))) \end{aligned} \quad (25)$$

operations.

Besides this bottleneck at the priority queue, many other calculations in the algorithm have an asymptotic cost which is significant in absolute terms, at least as expensive as the algorithm of Paige and Tarjan. However, all these cost terms can be bounded in terms of the values defined above and seen to be relatively negligible compared to Equation 25.

3.5.1 Possible Efficiency Improvements

We will discuss several ways to modify the algorithm which might allow some of the logarithmic cost factors to be removed. Note, however, that it is not necessarily worth proceeding with such modifications, especially if they make the algorithm more complicated. For realistic data which fits in a 32-bit address space, an $O(\log \#_1 M)$ improvement seems unlikely to work out to more than an order of magnitude. A factor of 20 is theoretically possible with a mere $1M$ vertices, but only if their taxonomy tree is perfectly binary. Other considerations not visible in big-O analysis could easily be worth an order of magnitude: *e.g.*, the difference between $O(1)$ hash table lookups and $O(1)$ structure field lookups, or the difference between an $O(1)$ set implemented with heap-allocated indirection or implemented as a bitmap stored in a single machine word, or the difference in cache performance between $O(1)$ data structures which differ in constant factors in size and locality. *E.g.*, in graphs containing a mixture of vertex types, some vertices edges are labelled and other vertices whose edges are unlabelled (*e.g.*, in graphs representing a combinatorial game[6]), it might not affect our asymptotic performance to handle labelled-edge vertices on an optimized code path using bitmaps to represent sets and avoiding the allocation and manipulation of redundant hash tables, but still it might be as important as removing a factor of $O(\log \#_1 M)$ from the asymptotic cost.

In the important special case of graphs with labelled edges,[16, 3, 14] various special tricks are possible. As described in Subsubsection 3.4.3, by using an initial partition in which labelled-edge vertices are grouped by equality both of label and of arity, the runtime cost associated with the `aw` weight is eliminated. Beyond that, as can be seen by comparing the simpler offline algorithm for labelled edges[2] to the more complicated offline algorithm for relational coarsest partition,[18] labelled edges allow us process only transitions from `ecounts[d] = 0` to `ecounts[k] = 1` (ignoring transitions from 1 to 0), and allow us to avoid maintaining an explicit `ecounts` map at all. We can also arrange for vertex outdegree always to be small by using a trivial pre-processing step to rewrite any large-outdegree vertex as a tree of vertices of small outdegree. Given small outdegree, sets of edge indices can be encoded as bitmaps in a single machine word, and we can afford to iterate over all edges of a `gv` in any generation in which any any of its edges swerve. We will not discuss this in more detail here, but note that the downloadable library `1yybnyz-0.1`,[15] contained an ancestor of the algorithm in this paper which

used many of these tricks.

In our main problem, relational coarsest partition on graphs whose edges are unlabelled, it doesn't seem to be possible in general to preprocess a graph into a graph of vertices of small outdegree. However, two other schemes might allow the elimination of some logarithmic cost factors.

One possible way to eliminate the logarithmic cost factors corresponding to our **aw** scheme for avoiding arity skew would be to use more sophisticated data structures so that arity skew could be tolerated instead of being transformed away. For example, at **rebalance** time the splitting history of each edge of each just-interned vertex is known, and the algorithm could create a more sophisticated representation of an interned vertex, rewriting the flat set representation of the interned vertex edges as a tree recapitulating the splitting history. With this representation available, and if **reserves** were made a little bit more expressive, the logic for setting up a **block.upt** at partition refinement generation r could use a single **reserves** entry to represent arbitrarily many edges which are indistinguishable in Π_r . Given such a representation, the cost of manipulating any unperturbed trajectory vertex u corresponding to a $v \in G$ could be proportional to $\#_2 v$ regardless of the hugeness of final minimized $\#_2 u$, so arity skew would no longer need to be avoided. This would be several logarithmic factors more efficient than our **aw** scheme to avoid arity skew, but could also be significantly more complicated, and it would only be asymptotically more efficient for graphs M in which $\#_1 M$ takes on many values; in some reasonable problem domains such graphs may be uncommon.

A possible way to eliminate another factor of $O(\log \#_1 M)$ would be by passing fewer values through the **reserves** queue. A vertex v of G only sees $O(\#_2 v (\log \#_1 M))$ reserves on its edges during partition refinement, but in our algorithm the refinement of v can cause $O(\#_2 v N_1 N_2) = O(\#_2 v (\log \#_1 M)^2)$ elements to be put into **reserves**, and processing each of those elements requires $O(\log \#_1 M)$ operations. Perhaps by some rearrangement of terms that could be reduced. For example, consider that a heap (*i.e.*, a data structure which implements a priority queue) can be merged with another heap in $O(1)$ time [12, 17] (postponing logarithmic expense to the pop operation). Perhaps our shared global heap **reserves**, holding reserve entries for every **block** that we ever created, could be separated into one subheap for each **block**, and we could pop only from those subheaps that we still care about (those whose associated **block.gvs** collection is still nonempty). If fast merge operations let us create those subheaps quickly, then the current

cost of $O(\log \#_1 M)$ operations to push each of $O(\#_2 G n_0 n_1)$ swerves could be replaced with a negligible cost of creating the subheaps plus a cost of $O(\log \#_1 M)$ operations to pop each of $O(\#_2 G n_0)$ swerves. It might be possible either to generate the to-be-merged subheaps very quickly on the fly, or to store them in the persistent taxonomy. A straightforward storage scheme could require logarithmic space, but since the heaps for nearby `leaf` objects share much of their structure, it might be possible to use purely functional heaps[17] and arrange enough sharing to keep space requirements linear.

It is also easy to reduce the effective cost of insertion and deletion of events in `reserves` for many problems: collect together all events of equal priority. Then our simple `reserves` queue of prioritized events becomes a queue of prioritized collections of events paired with a map from r to collections, and handling of n events at priority r requires $O(n + \log \#_1 M)$ operations rather than $O(n \log \#_1 M)$ operations. However, this doesn't change the worst case asymptotic costs reported above.

The algorithm's approach of batching partition refinement by entire generations of splitter `kinds` is probably wasteful. This approach requires the taxonomy `siblings` maps to be general enough to support arbitrarily complicated keys, and rewriting the complicated keys at rebalancing time is a lot of work. Barring deep changes to the algorithm and data structures, some batching is probably required in order to keep the taxonomy stable under rebalancing, but switching to a finer granularity of batching might be relatively easy.

4 Conclusion

Even in its current form our algorithm should be useful. Even without an asymptotically efficient algorithm, a number of papers have been published on incrementally minimizing graphs with labelled edges, and our algorithm is an improvement on the methods in those papers. The asymptotic cost of our algorithm improves on the old $O(\#_2(G \cup M) \log \#_1(G \cup M))$ worst case cost of previous partially-incremental solutions,[16, 3, 8] and its guarantee to find all equivalences can be more valuable than the ability of previous completely-incremental solutions[14] to find only some equivalences. Furthermore, our extension from labelled edges to the relational coarsest partition problem is a useful increase in generality.

Beyond that, it seems likely that our algorithm can be made both simpler

and asymptotically faster. For the purposes of the author’s own prototype software, a demonstration that the problem can be solved in $\#_2 G (\log \#_2 (G \cup M))^\nu$ is of considerable significance, but for production calculations it will become more important to reduce the exponent ν . It seems likely it is possible to minimize a graph with respect to relational coarsest partition in $O((\#_2 G)(\log \#_1 M)^2)$ operations, and the author knows no compelling reason to believe that $O((\#_2 G)(\log \#_1 M)^1)$ is impossible.

Our algorithm only supports the addition of new vertices with edges running to the minimized graph. It would be very useful to have another incrementally efficient algorithm to maintain a graph in minimized form as edges of existing vertices are modified. A single edge modification operation might either rewrite an individual edge or redirect all edges referring to some vertex v so that they refer to another vertex v' instead. The second kind of operation, sometimes called creating a “transient” [21], appears naturally when graphs are rewritten according to local rules, *e.g.*, when an optimizing compiler modifies the flow graph of the compiled program. Transient operations would also facilitate the general definition and efficient implementation of higher-order functions on cyclic data. [22] It’s not clear whether such an algorithm is possible. Failing to find such an algorithm, it would be also be interesting to reduce the problem to an outstanding difficult problem like incremental reachability [10] or incremental garbage collection. [9]

Less-ambitious extensions could also be useful. Turbak and Wells [22] suggest integrating incremental graph minimization with cyclic `unfold` operations, and that should be straightforward even without support for transients. It might also be useful, for such `unfold` operations or in other applications, to convert the `leaf.vertex` field to a weak pointer; with some care, it should then be possible to cause the taxonomy to free storage used to represent otherwise-unreferenced subgraphs, and to maintain balance as it does so.

A Class Layout

Figure 30 summarizes, in trivial UML, the associations between classes and fields described elsewhere in the paper.

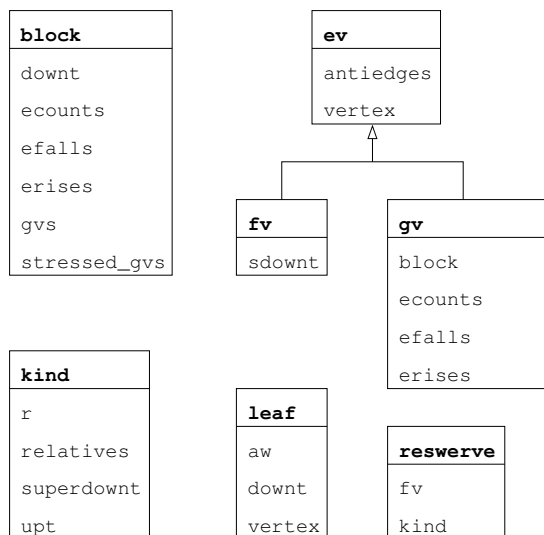


Figure 30: Layout of classes used in the pseudocode

B Minimizing Graphs With Labelled Edges

At the cost of adding one new vertex and one new edge for each old vertex, we can transform the problem of partitioning a labelled-edge graph into a relational coarsest partition problem so that it can be solved by our algorithm.

Let G_L be a graph with labelled edges. The edges of each vertex v of G_L are a tuple of vertices of G_L . Unlike in the unlabelled-edge graphs we have worked with elsewhere in the paper (where a vertex's edge collection is a set), in the labelled-edge graph the order of appearance of edges is significant: the "label" of an edge is its position in the tuple.

Let Λ be a sequence of unique labels none of which appear as vertex labels in G_L . For example, if the vertex labels of G_L are all letters, then the elements of Λ could be the infinite sequence $(0, 1, 2, \dots)$. The length of Λ is arbitrary but must be least as large as the length of the longest tuple of edges in G_L .

Now create a graph G_U with unlabelled edges from G_L by replacing each tuple T of old edges with a set of new edges. Each new edge runs to a new vertex of outdegree 1. Each new vertex is labelled with that label whose position in Λ corresponds to the position of the old edge in T . The edge leaving each new vertex runs to the vertex of G_L which was the corresponding

element of T . (Thus, the properties of the edge labels have been translated into a new layer of vertex labels.)

The equivalence properties of G_U have a simple relationship to the equivalence properties of G_L : two vertices of G_L are equivalent (under the usual labelled-edge equivalence rule, as in Cardon and Crochemore,[2] *e.g.*) iff the corresponding vertices of G_U are equivalent (under relational coarsest partition).

C Cost of the Randomized Hopcroft Rule

C.1 Results For Partitions

Theorem 1 (Expected Shrinkage) *Let Π be a partition of a set S into more than one block, let $s \in S$ be a uniformly distributed random sample, and let $C(s)$ be the largest block of Π such that $s \notin C(s)$. Then*

$$\frac{\langle \#C(s) \rangle}{\#S} \leq \frac{1}{2} \quad (26)$$

Proof: There are two cases, one in which all blocks of Π hold fewer than half the elements of S , and one in which some block A of Π holds more than half the elements of S . In the first case the result follows immediately; we will now consider the second case.

Define $C(B)$ to be the largest block of Π such that $C(B) \cap B = \emptyset$. Then the left side of Equation 26 can be expanded as

$$\frac{\langle \#C(s) \rangle}{\#S} = P(s \in A) \frac{\#C(A)}{\#S} + \sum_{B \in \Pi \wedge B \neq A} P(s \in B) \frac{\#C(B)}{\#S}. \quad (27)$$

Then because $\#C(A) \leq \#S - \#A$ and $B \neq A \rightarrow C(B) = \#A$, we have

$$\begin{aligned} \frac{\langle \#C(s) \rangle}{\#S} &\leq P(s \in A) \left(1 - \frac{\#A}{\#S}\right) + P(s \notin A) \frac{\#A}{\#S} \\ &= \frac{\#A}{\#S} \left(1 - \frac{\#A}{\#S}\right) + \left(1 - \frac{\#A}{\#S}\right) \frac{\#A}{\#S} \\ &= 2\phi(1 - \phi) \end{aligned} \quad (28)$$

for $\phi = \#A/\#S$. By elementary calculus, this expression has a maximum value of $1/2$ (found at $\phi = 1/2$). ■

C.2 Results For Trees

Let a tree be recursively defined as either a leaf or a nonempty set of subtrees (which have no leaves in common). Let $\#_\beta\tau$ and $\#_\lambda\tau$ be the number of branches and leaves, respectively, in the tree τ .

Given a map $w(\lambda)$ from leaves λ to ordered values w , define the unperturbed trajectory $u(\beta)$ of any branch β to be the path from β to that λ it reaches whose $w(\lambda)$ value is smallest.

A new leaf μ can be inserted in a tree in either of two ways. First, “insertion in a branch:” μ may be inserted directly in an existing branch β , so that β becomes $\beta \cup \{\mu\}$. Second, “insertion in an edge:” an element ϵ of an existing branch β may be replaced by a new branch $\beta' = \{\mu, \epsilon\}$, so that β becomes $(\beta \setminus \epsilon) \cup \{\beta'\}$. Either way, we choose $w(\mu)$ to be a new random sample from $[0, 1]$.

Theorem 2 (Expected Cost of Rebalancing) *For any insertion of μ into a tree τ , define $R(\tau)$ be the number of branches whose u values are changed by the insertion. Then given random choices of w (as uncorrelated random samples from a totally ordered set sufficiently large that collision probability is negligible),*

$$\langle R(\tau) \rangle = O(\log \#_\lambda\tau). \quad (29)$$

Proof: Observe that insertion of μ can only change the u value of a branch if that branch reaches μ , and that all branches which reach μ lie on the path from μ back to the root of the tree. Let $B_\mu = \{\beta_1, \beta_2, \dots, \beta_n\}$ be the set of branches of τ which lie on that path, with β_n the root of the tree and β_1 the branch which was modified by the insertion.

Observe that

$$D_\mu \equiv \{\forall b \in B_\mu : \#_\lambda b\} \subseteq \{1, 2, \dots, \#_\lambda\tau\}. \quad (30)$$

For any i , $u(\beta_i)$ is changed by the insertion iff $w(\mu)$ is smaller than the w values of all the leaves reached by β_i . The probability of change, $P_R(\beta_i)$, is the probability that $w(\mu)$ is the smallest element of the set which contains $w(\mu)$ and the w values of all the leaves reached by β_i . This is a set of $1 + \#_\lambda\beta_i$ uncorrelated unique values, so the probability that any given element is the smallest in the set is $(1 + \#_\lambda\beta_i)^{-1}$.

The expected number of changes is the sum of P_R ,

$$\langle R(\tau) \rangle = \sum_{b \in B_\mu} P_R(b) = \sum_{b \in B_\mu} (1 + \#\lambda\beta_i)^{-1} = \sum_{d \in D} (1 + d)^{-1}. \quad (31)$$

Then using Equation 30,

$$\langle R(\tau) \rangle \leq \sum_{i \in \{1, \dots, \#\lambda\tau\}} (1 + i)^{-1} < \int_1^{1+\#\lambda\tau} \frac{dx}{x} = O(\log \#\lambda\tau), \quad (32)$$

QED. ■

References

- [1] BDD
- [2] CardonCrochemore
- [3] Considine
- [4] Coq
- [5] That which I can't remember makes me stronger!
- [6] CGTThesis2006
- [7] Hopcroft
- [8] HorbachWoop
- [9] IncrementalGC
- [10] IncrementalReachability
- [11] IncrMinOnTypes
- [12] IntroductionToAlgorithms
- [13] IntroductionToAlgorithms p. 161 (?)
- [14] Javafolk

- [15] lyybnyz-download
- [16] Mauborgne
- [17] PurelyFunctionalDataStructures
- [18] PaigeTarjan
- [19] PrinciplesOfProgramAnalysis
- [20] SkipQuadtree
- [21] Tack
- [22] TurbakWells

TODO: Replace mnemonic placeholders in the bibliography with actual citations.