Pseudoflow Solvers for the Maximum Flow Problem

Charles Anderson

September 2003

Preface

This is a collection of implementations of various forms of the pseudoflow algorithm and heuristics for it.

Contents

1 Introduction

Currently, we support the following variant to control the execution of the algorithm:

- Initial normalized tree
- Strong branch management
- Merging - delayed normalization

The latter two are controlled by pointer to functions. Although this involves some extra overhead, it is the most flexible method for experimenting with new variants. In the future, if we want maximum performance to compare to other maximum flow programs, we can use conditional compilation.

2 Pseudoflow Solver

The solver manages the entire problem instance. Initially this includes the nodes and edges. These are stored as contiguous arrays (as opposed to allocating each one individually with `new`) to make memory management fast and simple. These arrays are allocated when the problem instance is read.

3a ⟨*Solver protected members* 3a⟩≡ (136c) 3c▷

```
Node*       nodes;
Edge*       edges;
```

Defines:
`edges`, used in chunks 3b, 71b, 121b, 123a, 125–28, and 131a.
`nodes`, used in chunks 3b, 6a, 55a, 60a, 62, 73a, 75, 79b, 82, 88a, 90, 92, 95a, 110, 114a, 115a, 123–26, 132–34, 195, 202, and 212.
Uses `Edge` 179a and `Node` 164b.

3b ⟨*default Solver constructor* 3b⟩≡ (134c) 3d▷

```
nodes = nil;
edges = nil;
```

Uses `edges` 3a and `nodes` 3a.

Just for safety checking, we should also keep track of the number of elements in each of these arrays.

3c ⟨*Solver protected members* 3a⟩+≡ (136c) ◁3a 5b▷

```
int         numNodes;
int         numEdges;
```

Uses `numEdges` 3d and `numNodes` 3d.

3d ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁3b 3f▷

```
numNodes = numEdges = 0;
```

Defines:
`numEdges`, used in chunks 3c, 71b, 121b, 123a, 125a, 128b, 129a, and 131a.
`numNodes`, used in chunks 3c, 13a, 24c, 35a, 55a, 60a, 62, 73a, 75, 76b, 78, 79b, 82, 83a, 86–88, 90a, 92, 94a, 95a, 107–110, 115a, 123–27, 129a, and 131–34.

We also need to know the source and sink nodes in the graph.

3e ⟨*Solver data* 3e⟩≡ (136c) 31b▷

```
NodePtr     sourceNode;
NodePtr     sinkNode;
```

Uses `sinkNode` 3f and `sourceNode` 3f.

3f ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁3d 5c▷

```
sourceNode = sinkNode = nil;
```

Defines:
`sinkNode`, used in chunks 3e, 56–58, 62, 71–73, 80a, 82, 88b, 89a, 92, 95–97, 103a, 105, 108–111, 115a, 124, 126a, and 131–34.
`sourceNode`, used in chunks 3e, 56–58, 62, 64a, 67a, 71b, 73b, 80a, 82, 88b, 89a, 92, 95, 96, 98a, 103a, 105, 107, 109–111, 115a, 124, 126a, and 130–34.

At the highest level, solving is pretty simple: we select the lowest labeled strong branch, process it looking for mergers and relabling nodes that do not merge, and continue until we cannot find a strong branch. We can select different policies for managing the list of strong branches (the buckets).

The solver function below performs the usual, full normalization after each merger: push the flow all the way from the strong root to the weak root. Later we will introduce another solver that only performs a partial normalization after each merger.

4a ⟨*Solver methods* 4a⟩≡ (136c) 6b▷

```
virtual void solve(AddBranchPtr addFunc);
```

Uses `AddBranchPtr` 136a and `solve` 4b.

4b ⟨*Solver method implementations* 4b⟩≡ (135a) 7a▷

```
void PhaseSolver::solve(AddBranchPtr addFunc)
{
    IFTRACE(int currentPhase = 0;)
    addBranchFunc = addFunc;
    renormalizeFunc = &PhaseSolver::fullRenormalize;
    initGlobalRelabel();

    NodePtr strongBranch = getLowestBranch();
    while (strongBranch != nil) {
        NodeLabel strongLabel =  strongBranch->getLabel();
        IFTRACE(
            if ( strongLabel != currentPhase) {
                currentPhase = strongLabel;
                TRACE(trout << "Phase " << currentPhase << endl;);
            }
        )
        ⟨terminate early based on counting labels 5a⟩
        bool performedMerger = processBranch(*strongBranch);
        STATS(performedMerger ? 0 : numEmptyBranchScans++ );
        checkForRelabel();
        strongBranch = getLowestBranch();
    }
    IFDEBUG(checkMergers());
}
```

Defines:
`solve`, used in chunks 4a, 7a, 195, 198a, 200b, 202, 210, and 211.

Uses `addBranchFunc` 26a, `AddBranchPtr` 136a, `checkForRelabel` 78a, `checkMergers` 134a, `fullRenormalize` 20b, `getLabel` 138g, `getLowestBranch` 28a 46a, `initGlobalRelabel` 78c, `numEmptyBranchScans` 129b, `PhaseSolver` 136c, `processBranch` 7a 47b, and `renormalizeFunc` 15c.

For each possible value for a node's label, if we track the number of nodes with that label, then it is possible to terminate the algorithm early. If the lowest labeled strong branch has label ℓ, and there are no nodes with label $\ell - 1$, then the algorithm is done because the strong branch cannot merge with any weak branches, and the labels of the strong branches will only increase further during execution.

5a ⟨*terminate early based on counting labels* 5a⟩≡ (4b 29a 119a)

```
if ((labelCount[strongLabel - 1] == 0) &&
    (strongLabel > maxInitialLabel))
{
    bool earlyTerm = false;
    if (relabelFrequency > 0.0) {
        ⟨perform last global relabel 6a⟩
    } else {
        earlyTerm = true;
    }

    if (earlyTerm) {
        cout << "Early termination at label " << strongLabel << endl;
        break;
    }
}
```

Uses `labelCount` 5c, `maxInitialLabel` 63b, and `relabelFrequency` 77d.

5b ⟨*Solver protected members* 3a⟩+≡ (136c) ◁3c 25d▷

```
int* labelCount;
```

Uses `labelCount` 5c.

5c ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁3f 8c▷

```
labelCount = nil;
```

Defines:
`labelCount`, used in chunks 5, 6a, 13a, 35a, 47a, 62, 75, 92, and 123a.

Some of the labeling methods are fairly complex, and we have been kinda lax With global relabeling, we can develop a gap in the node labels before we are done processes - i.e. while there are still merger arcs. If we perform another global relabel, that should close the gap by raising the labels of various weak and strong nodes. If the gap persists, we really are done. Note that if we are done before this final relabel, the relabel operation should prune out all of the strong nodes because they will no longer be able to reach the sink. In this case, `getLowestBranch` should return null.

6a ⟨*perform last global relabel* 6a⟩≡ (5a)

```
addBranchFifo(*strongBranch);
cout << "Performing final global relabel at phase " << strongLabel
     << " with " << numRemovedNodes << " nodes pruned from the graph" << endl;
globalRelabel();
strongBranch = getLowestBranch();
if (strongBranch == nil) {
    break;
}
strongLabel =  strongBranch->getLabel();
if ((labelCount[strongLabel - 1] == 0) &&
    (strongLabel > INIT_WEAK_LABEL))
{
    earlyTerm = true;
}
```

Uses `addBranchFifo` 26c, `getLabel` 138g, `getLowestBranch` 28a 46a, `globalRelabel` 79a, `INIT_WEAK_LABEL` 60b, `labelCount` 5c, `nodes` 3a, and `numRemovedNodes` 129b.

2.1 Processing Branches

We process the tree one branch at a time looking for mergers or relabeling nodes. As soon as there is a merger, we stop processing this branch to allow the main loop to fetch the next, lowest-labeled branch from the buckets. It could be that we resume processing this branch. If there was no merger, we move the branch to the next bucket.

As an optimization, we check to see if there are any nodes that we could possibly merge with before invesigating the branch. If there are no nodes with a label one less than ours, we just relabel the whole branch. This shouldn't happen very often any more, especially since we count the nodes with each label value and terminate early. It might still be invoked as a side-effect of some initialization or normalization schemes - especially if we queue a branch with $\ell = 1$, the initial value for weak nodes.

6b ⟨*Solver methods* 4a⟩+≡ (136c) ◁4a 7b▷

```
virtual bool processBranch(Node& root);
```

Uses `Node` 164b and `processBranch` 7a 47b.

7a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁4b 8a▷

```
bool PhaseSolver::processBranch(Node& root)
{
    bool performedMerger = processSubtree(root);
    if (!performedMerger) {
        addStrongBranch(root);  // XXX maybe this should be in solve
    }

    return performedMerger;
}
```

Defines:
`processBranch`, used in chunks 4b, 6b, 29a, 34, 46b, 47a, and 119a.
Uses `addStrongBranch` 24c 51a, `Node` 164b, `PhaseSolver` 136c, `processSubtree` 8a, and `solve` 4b.

Below is recursive code to perform the DFS scan of a node's children and to look for merger arcs among its neighbors. The code processes the nodes in either post-order (i.e. the children are all processed before the neighbors of this node are scanned) or pre-order (scan the neighbors first). If this function performs a merger, it returns immediately. If any part of the sub-tree is split off and strong, it will be put in the appropriate bucket where `getLowestBranch` will find it. If there is no merger, the label of this node is incremented. The function returns true or false if we perform a merger.

7b ⟨*Solver methods* 4a⟩+≡ (136c) ◁6b 8e▷

```
    bool processSubtree(Node& node);
```

Uses `Node` 164b and `processSubtree` 8a.

8a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁7a 14a▷

```
bool PhaseSolver::processSubtree(Node& node)
{
    NodeLabel label = node.getLabel();
    STATS(numNodeVisits++);
    if (postOrderSearch) {
        ⟨process all children 8d⟩
        ⟨look for merger among neighbors 9b⟩
    } else {
        ⟨look for merger among neighbors 9b⟩
        ⟨process all children 8d⟩
    }

    // no merger
    ⟨verify no merger available 12a⟩
    ⟨increase node label 11b⟩
    TRACE(trout << "node " << node.getId() << " relabled to "
                << node.getLabel() << endl;);
    return false;
}
```

Defines:
`processSubtree`, used in chunks 7, 9a, and 47a.
Uses `getLabel` 138g, `Node` 164b, and `PhaseSolver` 136c.

8b ⟨*public Solver data* 8b⟩≡ (136c) 19b▷

```
bool postOrderSearch;
```

8c ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁5c 15c▷

```
postOrderSearch = true;
```

We process all children that have the same label as ours, `label`. If a child has a higher label, we just ignore it. By monotonicity, it can't have a label less than ours. If we process the sub-tree and that generates a merger we return immediately.

8d ⟨*process all children* 8d⟩≡ (8a)

```
if (scanChildren(node, label)) {
    return true;
}
```

Uses `scanChildren` 9a.

8e ⟨*Solver methods* 4a⟩+≡ (136c) ◁7b 10▷

```
bool scanChildren(Node& node, NodeLabel label);
```

Uses `Node` 164b and `scanChildren` 9a.

9a ⟨*Solver inline implementations* 9a⟩≡ (135b 136d) 11a▷

```
INLINE bool PhaseSolver::scanChildren(Node& node, NodeLabel label)
{
    while (node.hasMoreChildren()) {
        Node& child = node.getCurrentChild();
        assert(child.getLabel() >= label);
        if (node.minChildLabel > child.getLabel()) {
            node.minChildLabel = child.getLabel();
        }
        if (child.getLabel() == label) {
            bool performedMerger = processSubtree(child);
            if (performedMerger) {
                return true;
            }
        }
        node.advanceChildren();
    }
    return false;
}
```

Defines:
`scanChildren`, used in chunk 8.
Uses `advanceChildren` 143b, `getCurrentChild` 143b, `getLabel` 138g, `hasMoreChildren` 143b, `minChildLabel` 156c, `Node` 164b, `PhaseSolver` 136c, and `processSubtree` 8a.

Processing neighbors is a bit different. Here is where we look for a merger to a node with label exactly one less that ours (given that the arc has residual capacity). If we find that, we will merge and return true immediately.

9b ⟨*look for merger among neighbors* 9b⟩≡ (8a)

```
    if (scanNeighbors(node, label)) {
        return true;
    }
```

Uses `scanNeighbors` 11a.

Note: there is a slight problem below. We check for a positive capacity on the arc (instead of non-negative), but if we ever implemented a scheme where we didn't split immediately up zero residual capacity, we would have zero-capacity arcs that are in the tree (i.e. the parent arc), but we wouldn't want to use them for mergers. This would be a mess, but we could overcome it. Actually, it seems pretty easy: during phase ℓ, both the node and it's parent have label ℓ, so when we check the label of the neighbor, we won't try to merge.

Note deux: According to the algorithm, the only merger arcs during phase l should be to nodes with label $l-1$. However, when we introduced global relabeling and shortest path initialization, we found that for some unknown reason there were weak nodes with label $l-2$ that we were not merging to. Therefore, for the moment at least, the test has been changed from label $= l-1$ to label $< l$. The existance of these super-weak mergers actually contradicts the lowest label rule because given two strong nodes with label l, one might have a neighbor with label $l-1$ and the other with $l-2$. We according to 'lowest label' we should process the latter first, but we might not. Maybe this is a bug that I should find to better understand global labeling?

Note trios: We added code to check the neighbor's distance label and merged to it only if it was less than ours, but this provided only minimal/negligible difference. The code still must check the node labels to tell the difference between strong and weak nodes (weak nodes still have label less than ours). With a genrmf graph, I only saw a differnce of two in the number of mergers.

10 ⟨*Solver methods* 4a⟩+≡ (136c) ◁8e 12b▷

```
bool scanNeighbors(Node& node, NodeLabel label);
```

Uses `Node` 164b and `scanNeighbors` 11a.

11a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁9a 13a▷

```
INLINE bool PhaseSolver::scanNeighbors(Node& node, NodeLabel label)
{
    if (node.distance <= label) {
//      int dist = node.distance;
        while (node.hasMoreNeighbors()) {
            STATS(numArcScans++);
            Edge& neighborEdge = node.getCurrentNeighbor();
            Node& neighbor = *neighborEdge.getOtherNode(&node);
            if ((neighbor.getLabel() < label) &&
//              (neighbor.distance < dist) &&
                (neighborEdge.residCapacity(node) > 0))
            {
                TRACE(if (neighbor.getLabel() < (label - 1))
                       trout << "Super weak merger from " << node.getId()
                             << " (l=" << node.getLabel() << ") to "
                             << neighbor.getId()
                             << " (l=" << neighbor.getLabel() << ")" << endl; );
                merge(node, neighbor, neighborEdge);
                return true;
            }
            node.advanceNeighbors();
        }
    }

    return false;
}
```

Defines:
`scanNeighbors`, used in chunks 9b and 10.
Uses `advanceNeighbors` 144a, `distance` 156c, `Edge` 179a, `getCurrentNeighbor` 144a, `getLabel` 138g, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `merge` 15a 37c, `Node` 164b, `PhaseSolver` 136c, and `residCapacity` 169b.

If we scan a node and do not find a merger, we want to increase the label of the node. At the very least, we need to increment the label of the node by one. However, if we are using distance labeling, we may be able to do more.

11b ⟨*increase node label* 11b⟩≡ (8a)

```
    increaseNodeLabel(node);
```

Uses `increaseNodeLabel` 87a.

This is some debug/assertion code to verify that we haven't missed any available merger opportunites. The algorithm claims that the only weak nodes we can merge with should have label $l-1$. When we use distance labels, we've found that sometimes we miss a merger with a node labeled $l-2$. This checks for that.

12a ⟨*verify no merger available* 12a⟩≡ (8a)

```
#ifdef DEBUG
    ElIterator it = node.getNeighbors();
    for (EdgePtr edge = it.getNext(); edge != nil; edge=it.getNext()) {
        Node& neighbor = *edge->getOtherNode(&node);
        FlowAmount residCap = edge->residCapacity(node);
        if (residCap > 0 && (neighbor.isStrong(false)) == false) {
            if (neighbor.getLabel() < node.getLabel()) {
                TRACE(trout << "Missed merger from " << node.getId() << " (l="
                      << node.getLabel() << ") to " << neighbor.getId()
                      << " (l=" << neighbor.getLabel() << ")" << endl);
            }
        }
    }
#endif /*DEBUG*/
```

Uses `getLabel` 138g, `getNeighbors` 145a, `getNext` 150g, `getOtherNode` 167a, `isStrong` 162a, `Node` 164b, and `residCapacity` 169b.

Back in `processSubtree` we had a simple method call to increment the label on a node. Normally, we would just call `incrementLabel` on the node, but we want to track how many nodes we have with each label so that we can terminate the algorithm early.

12b ⟨*Solver methods* 4a⟩+≡ (136c) ◁10 12c▷

```
void incrementLabel(Node& node);
```

Uses `incrementLabel` 13b 138g and `Node` 164b.

However, we can't define it yet due to compiler dependencies with inline function because we treat incrementing the label as a special case of setting it. Setting an aribitrary value gets used by global relabeling. We need to update the counts when we do this. We also reset the iterations.

12c ⟨*Solver methods* 4a⟩+≡ (136c) ◁12b 13c▷

```
void setLabel(Node& node, NodeLabel newLabel);
```

Uses `Node` 164b and `setLabel` 13a 138g.

13a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁11a 13b▷

```
INLINE void PhaseSolver::setLabel(Node& node, NodeLabel newLabel)
{
    assert(newLabel > node.getLabel());
    labelCount[node.getLabel()]--;
    labelCount[newLabel]++;
    STATS(numRelabels++);
    STATS(if ((newLabel - node.getLabel()) > 1) numLabelSkips++; );
    node.setLabel(newLabel);
    node.resetIterations();
    node.minChildLabel = max(node.distance, newLabel); // move to resetIterations?
    STATS(if (newLabel >= numNodes)  numRemovedNodes++);
}
```

Defines:
  `setLabel`, used in chunks 12c, 13b, 61a, 65, 68, 73–76, 84a, 85, 87–90, 92, 112a, and 138f.
Uses `distance` 156c, `getLabel` 138g, `labelCount` 5c, `minChildLabel` 156c, `Node` 164b, `numLabelSkips` 129b, `numNodes` 3d, `numRelabels` 129b, `numRemovedNodes` 129b, `PhaseSolver` 136c, and `resetIterations` 144c.

Now, increment is trivial.

13b ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁13a 17a▷

```
INLINE void PhaseSolver::incrementLabel(Node& node)
{
    setLabel(node, node.getLabel() + 1);
}
```

Defines:
  `incrementLabel`, used in chunks 12b, 14a, 87a, and 138f.
Uses `getLabel` 138g, `Node` 164b, `PhaseSolver` 136c, and `setLabel` 13a 138g.

When we know that a branch cannot merge with any weak nodes, we need to increment the label of the branch, and all of its children that have the same label. This is really a stripped down version of `processSubtree`. Given the early termination code based on counting nodes and labels, this code isn't usually needed. It might be useful for some tree initialization schemes that result in strong branches with the initial weak label.

13c ⟨*Solver methods* 4a⟩+≡ (136c) ◁12c 14b▷

```
void relabelSubtree(Node& node);
```

Uses `Node` 164b and `relabelSubtree` 14a.

14a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁8a 15a▷

```
void PhaseSolver::relabelSubtree(Node& node)
{
    NodeLabel label = node.getLabel();
    node.resetIterations();
    while (node.hasMoreChildren()) {
        Node& child = node.getCurrentChild();
        assert(child.getLabel() >= label);
        if (child.getLabel() == label) {
            relabelSubtree(child);
        }
        node.advanceChildren();
    }

    incrementLabel(node);
    TRACE(trout << "node " << node.getId() << " Relabled to "
                << node.getLabel() << endl;);
}
```

Defines:
`relabelSubtree`, used in chunks 13c and 47a.
Uses `advanceChildren` 143b, `getCurrentChild` 143b, `getLabel` 138g, `hasMoreChildren` 143b, `incrementLabel` 13b 138g, `Node` 164b, `PhaseSolver` 136c, and `resetIterations` 144c.

2.2 Basic Merger Functions

The normal merger function rehangs the strong branch from the strong node, adds the merger arc from s to w, and renormalizes the tree by pushing the excess from the old strong root towards the weak root. There are different possible policies for renormalizing the tree after a merger, so we use a pointer to function as an indirection.

14b ⟨*Solver methods* 4a⟩+≡ (136c) ◁13c 15d▷

```
virtual void merge(Node& strong, Node& weak, Edge& edge);
```

Uses `Edge` 179a, `merge` 15a 37c, and `Node` 164b.

15a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁14a 16a▷

```
void PhaseSolver::merge(Node& strong, Node& weak, Edge& edge)
{
    TRACE( trout << "merge " << strong.getId() << " ("
                 << strong.getRootExcess() << ") to "
                 << weak.getId() << " ("
                 << weak.getExcess() << ", "
                 << weak.getRootExcess() << "), cap "
                 << edge.residCapacity(strong) << ", s-label "
                 << strong.getLabel() << endl;)
    STATS(numMergers++);

    Node& strongRoot = *strong.rehang(STATS(numRehangs));
    weak.addChild(strong, edge);
    CHECK_TREE(*weak.getRoot(), nil);

    // renormalize the tree
    (this->*renormalizeFunc)(strongRoot, weak);
}
```

Defines:
`merge`, used in chunks 11a, 14b, 37b, and 200a.
Uses `addChild` 142a, `Edge` 179a, `getExcess` 151d, `getLabel` 138g, `getRoot` 161c, `getRootExcess` 162c, `Node` 164b, `numMergers` 129b, `PhaseSolver` 136c, `rehang` 147, `renormalizeFunc` 15c, and `residCapacity` 169b.

15b ⟨*Solver private members* 15b⟩≡ (136c) 23a▷

```
RenormalizePtr renormalizeFunc;
```

Uses `renormalizeFunc` 15c and `RenormalizePtr` 136a.

15c ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁8c 19d▷

```
renormalizeFunc = &PhaseSolver::fullRenormalize;
```

Defines:
`renormalizeFunc`, used in chunks 4b, 15, 29a, 34, and 119a.
Uses `fullRenormalize` 20b and `PhaseSolver` 136c.

Push excess from the former strong root, `src` to the strong node and one step beyond to the weak node, `dest`. When this finishes, the remaining excess will be stored on the destination node. We save the parent pointer before the push in case there is a split, in which case the parent pointer will be `nil`.

15d ⟨*Solver methods* 4a⟩+≡ (136c) ◁14b 16b▷

```
void strongPush(Node& src, Node& dest);
```

Uses `dest` 165a, `Node` 164b, and `strongPush` 16a.

16a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁15a 18▷

```
void PhaseSolver::strongPush(Node& src, Node& dest)
{
    NodePtr currNode = &src;
    while (currNode != &dest) {
        NodePtr parent = currNode->getParentNode();
        STATS(numPushToParent++);
        bool performedSplit = currNode->pushToParent();
        performedSplit = splitZeroCapArc(*currNode, performedSplit);

        if (performedSplit) {
            STATS(numSplits++);
            if (isStrongNode(*currNode)) {
                addStrongBranch(*currNode);
            }
        }
        currNode = parent;
    }
}
```

Defines:
`strongPush`, used in chunks 15d, 20b, 31a, and 41a.
Uses `addStrongBranch` 24c 51a, `dest` 165a, `getParentNode` 141a, `isStrongNode` 17c, `Node` 164b, `numPushToParent` 129b, `numSplits` 129b, `PhaseSolver` 136c, `pushToParent` 152b, and `splitZeroCapArc` 17a.

We allow special handling if an edge has zero residual capacity and we didn't already split it. It could be split or left alone depending on `splitOnZeroCapacity`. Returns true if we perform a new split. Otherwise, we just return the the `alreadySplit` parameter.

16b ⟨*Solver methods* 4a⟩+≡ (136c) ◁15d 17b▷

```
bool splitZeroCapArc(Node& node, bool alreadySplit);
```

Uses `Node` 164b and `splitZeroCapArc` 17a.

17a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁13b 17c▷

```
INLINE bool PhaseSolver::splitZeroCapArc(Node& node, bool alreadySplit)
{
    if (alreadySplit == false &&
        node.getParentCapacity() == 0 &&
        splitOnZeroCapacity)
    {
        node.split();
        return true;
    }

    return alreadySplit;
}
```

Defines:
  `splitZeroCapArc`, used in chunks 16, 18, and 118a.
Uses `getParentCapacity` 141a, `Node` 164b, `PhaseSolver` 136c, `split` 145c, and `splitOnZeroCapacity` 19d.

We need to determine if a node is strong. Usually, this means that it has positive excess, but in some cases, we can choose to treat a node with zero excess as strong. We want to select that at runtime for experimentation.

17b ⟨*Solver methods* 4a⟩+≡ (136c) ◁16b 17d▷

```
bool isStrongNode(const Node& node);
```

Uses `isStrongNode` 17c and `Node` 164b.

17c ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁17a 25a▷

```
INLINE bool PhaseSolver::isStrongNode(const Node& node)
{
    FlowAmount excess =  node.getExcess();
    return ((excess > 0) || ((excess == 0) && zeroDeficitIsStrong)) ?
            true : false;
}
```

Defines:
  `isStrongNode`, used in chunks 16–19, 45a, 82, and 118a.
Uses `excess` 151b, `getExcess` 151d, `Node` 164b, `PhaseSolver` 136c, and `zeroDeficitIsStrong` 19d.

Pushing on the weak side is similar in that it uses `pushToParent` to push all of the flow, but it differs in that we push all the way to the root of the branch. The function returns true if we perform any splits - i.e. create any new strong branches.

17d ⟨*Solver methods* 4a⟩+≡ (136c) ◁17b 20a▷

```
bool weakPush(Node& src);
```

Uses `Node` 164b and `weakPush` 18.

18 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁16a 20b▷

```
bool PhaseSolver::weakPush(Node& src)
{
    bool result = false;
    NodePtr currNode = &src;
    NodePtr parent = currNode->getParentNode();

    while (parent != nil) {
        STATS(numPushToParent++);
        bool performedSplit = currNode->pushToParent();
        currNode->clearFlag(DEFERRED);
        performedSplit = splitZeroCapArc(*currNode, performedSplit);

        if (performedSplit) {
            STATS(numSplits++);
            CHECK_TREE(*currNode, nil);
            if (isStrongNode(*currNode)) {
                // should worry about l=1
                addStrongBranch(*currNode);
                result = true;
            }
        }
        currNode = parent;
        parent = currNode->getParentNode();
    }

    CHECK_TREE(*currNode, nil);
    currNode->clearFlag(DEFERRED);

    ⟨check for weak root becoming strong 19a⟩
    return result;
}
```

Defines:
`weakPush`, used in chunks 17d, 20b, 32b, 40d, 41c, 107, and 108.
Uses `addStrongBranch` 24c 51a, `clearFlag` 155d, `getParentNode` 141a, `isStrongNode` 17c, `Node` 164b, `numPushToParent` 129b, `numSplits` 129b, `PhaseSolver` 136c, `pushToParent` 152b, and `splitZeroCapArc` 17a.

After we have pushed flow all the way to the weak root, we need to check to see if the weak root is now strong. The weak root is pointed to by `currNode`. One special case we deal with here is if the weak root has the initial weak label, we know that when it goes strong, there will be no one for it to merge with (because the weak label is the lowest possible). Therefore, we can just relabel it here and now before adding it.

19a ⟨*check for weak root becoming strong* 19a⟩≡ (18)

```
if (isStrongNode(*currNode)) {
    addStrongBranch(*currNode);
    result = true;
}
```

Uses `addStrongBranch` 24c 51a and `isStrongNode` 17c.

In the code above, we used some flags to control some of the behavior of the code while it's running. We'll declare these public so that they can be easily inspected and tweaked. The first flag controls what happens if we push excess along an arc that matches the residual capacity. We could choose to either split it or leave it alone until the next push sends flow along the arc, which will cause an immediate split.

19b ⟨*public Solver data* 8b⟩+≡ (136c) ◁8b 19c▷

```
bool splitOnZeroCapacity;
```

Uses `splitOnZeroCapacity` 19d.

The second flag controls how we regard zero-deficit nodes/branches during the execution of the algorithm (they are always weak when start the algorithm). If we treat a zero-deficit node as weak, when the algorithm finishes, the strong nodes will be a minimal source set. However, this can lead to creating weak branch with root nodes whose label is greater than one. This in turn makes it very difficult (impossible?) to bound the maximum label that weak node can take (if weak roots have label one, no node can have a label greater than n). If we treat zero-deficit nodes as strong, weak roots will always have label one, we will never create any additional zero-deficit nodes/branches, but we have no easy way to find a minimal source set.

19c ⟨*public Solver data* 8b⟩+≡ (136c) ◁19b 67b▷

```
bool zeroDeficitIsStrong;
```

Uses `zeroDeficitIsStrong` 19d.

19d ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁15c 23b▷

```
splitOnZeroCapacity = true;
zeroDeficitIsStrong = false;
```

Defines:
`splitOnZeroCapacity`, used in chunks 17a and 19b.
`zeroDeficitIsStrong`, used in chunks 17c and 19c.

After we have attached the strong branch to the weak node, we need to renormalize the tree to push excess from the old strong root up to the weak node and onto the weak root. This is the default, full normalization procedure.

20a ⟨*Solver methods* 4a⟩+≡ (136c) ◁17d 24b▷
```
void fullRenormalize(Node& strongRoot, Node& weakNode);
```
Uses `fullRenormalize` 20b and `Node` 164b.

20b ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁18 22d▷
```
void PhaseSolver::fullRenormalize(Node& strongRoot, Node& weakNode)
{
    strongPush(strongRoot, weakNode);
    weakPush(weakNode);
}
```
Defines:
`fullRenormalize`, used in chunks 4b, 15c, 20a, 34, and 42b.
Uses `Node` 164b, `PhaseSolver` 136c, `strongPush` 16a, and `weakPush` 18.

2.3 Strong Branch Management

To clean up the management of the buckets, let's introduce a simple class.

Note: using a bucket class instead of separate head and tail pointers seemed to case a performance hit of about 1.7% - cher.20.20 w/o buckets = 35.7, with buckets =36.3 seconds.

20c ⟨*Node Bucket data* 20c⟩≡ (137a)
```
private:
    NodePtr     head;
    NodePtr     tail;
```

20d ⟨*Node Bucket methods* 20d⟩≡ (137a) 22a▷
```
public:
    void        insertHead(Node& root);
    void        insertTail(Node& root);
    NodePtr     removeHead();
```
Uses `insertHead` 21, `insertTail` 21, `Node` 164b, and `removeHead` 21.

We will always implement these as inline functions:

21 ⟨*Node Bucket inline implementations* 21⟩≡ (137a)

```
inline void
NodeBucket::insertHead(Node& root)
{
    if (head == nil) {
        root.setNextNil();
        head = &root;
        tail = &root;
    } else {
        root.setNext(head);
        head = &root;
    }
}
inline void
NodeBucket::insertTail(Node& root)
{
    if (head == nil) {
        root.setNextNil();
        head = &root;
        tail = &root;
    } else {
        tail->setNext(&root);
        tail = &root;
        root.setNextNil();
    }
}
inline NodePtr
NodeBucket::removeHead()
{
    NodePtr result = nil;
    if (head != nil) {
        result = head;
        head = result->getNext();
        if (head == nil) {
            tail = nil;
        }
    }

    return result;
}
```

Defines:
`insertHead`, used in chunks 20d and 26.
`insertTail`, used in chunks 20d and 26.

`removeHead`, used in chunks 20d, 22d, 27d, and 36b.
Uses `getNext` 150g, `Node` 164b, `setNext` 150e, and `setNextNil` 150e.

Here are a couple of obvious accessor functions.

22a ⟨*Node Bucket methods* 20d⟩+≡ (137a) ◁20d 22b▷

```
NodePtr getHead() const { return head; }
NodePtr getTail() const { return tail; }
```

Defines:
`getHead`, used in chunks 25a, 71b, 109c, 126b, 159, and 166c.
`getTail`, used in chunks 25a, 71b, 126b, 159, and 166c.

Empty a bucket.

22b ⟨*Node Bucket methods* 20d⟩+≡ (137a) ◁22a 22c▷

```
void emptyBucket() { head = tail = nil; }
```

Defines:
`emptyBucket`, used in chunks 109d and 115.

Typically, we remove nodes only from the head of the bucket. However, during global relabeling, we may wish to remove an arbitrary node. Fortunately, this doesn't happen often, so we can affort the cost of an $O(n)$ remove operation.

22c ⟨*Node Bucket methods* 20d⟩+≡ (137a) ◁22b 23c▷

```
void        removeNode(Node& node);
```

Uses `Node` 164b.

22d ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁20b 23d▷

```
void NodeBucket::removeNode(Node& node)
{
    NodePtr prev = nil;
    for (NodePtr curr = head; curr != nil; curr = curr->getNext()) {
        if (curr == &node) {
            if (prev == nil) {
                removeHead();
            } else {
                prev->setNext(curr->getNext());
                if (tail == curr) {
                    tail = prev;
                }
            }
            curr->setNextNil();
            return;
        }
        prev = curr;
    }
    assert("couldn't find node in bucket" == nil);
}
```

Uses `getNext` 150g, `Node` 164b, `removeHead` 21, `setNext` 150e, and `setNextNil` 150e.

23a ⟨*Solver private members* 15b⟩+≡ (136c) ◁15b 23e▷
```
NodeBucket* buckets;
```
Uses buckets 23b.

23b ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁19d 24a▷
```
buckets = nil;
```
Defines:
buckets, used in chunks 23a, 25–27, 36b, 83, 109, 115, and 123a.

23c ⟨*Node Bucket methods* 20d⟩+≡ (137a) ◁22c
```
void        dumpBucket();
```
Note: this is in the 'wrong' chunck Solver rather than bucket.

23d ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁22d 24c▷
```
void NodeBucket::dumpBucket()
{
    cout << "head = " << head;
    if (head != nil) {
        cout << " (" << head->getId() << "), tail = "
             << tail << " (" << tail->getId() << ")" << endl;
    } else {
        cout << ", tail = " << tail << endl;
        return;
    }
    for (NodePtr n = head; n != nil; n = n->getNext()) {
        cout << n->getId() << "  ";
    }
    cout << endl;
}
```
Uses getNext 150g.

Because we process the lowest labeled branch, we need a way to find the strong branch quickly. At the moment, we just search the list sequentially looking for the lowest labeled, non-empty bucket. To speed this a bit, we keep track of the lowest labeled bucket we've seen. In the future, this simple array of buckets could be replaced by some sort of heap.

Also, there are a couple of minor instances where we want to scan all of the buckets. In those cases, having the maximum label that we have seen is useful to avoid scanning every bucket.

23e ⟨*Solver private members* 15b⟩+≡ (136c) ◁23a 60b▷
```
NodeLabel   lowestLabel;
NodeLabel   highestLabel;
```
Uses highestLabel 24a and lowestLabel 24a.

24a ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁23b 26a▷
```
lowestLabel = highestLabel = 0;
```
Defines:
highestLabel, used in chunks 23e, 25c, 27d, 36, and 109c.
lowestLabel, used in chunks 23e, 25c, 27d, 28a, 36b, 79a, 119a, and 129a.

Adding a branch just requires finding the bucket based on the branch's label, and inserting it into the list.

24b ⟨*Solver methods* 4a⟩+≡ (136c) ◁20a 24d▷
```
virtual void addStrongBranch(Node& root);
```
Uses addStrongBranch 24c 51a and Node 164b.

24c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁23d 26c▷
```
void PhaseSolver::addStrongBranch(Node& root)
{
    assert((root.getParentNode() == nil) && (root.getExcess() >= 0));
    assert(root.getLabel() < numNodes);
    CHECK_TREE(root, nil);
    NodeLabel label = root.getLabel();

    if (branchInBucket(root) == false) {
        // actually add the branch
        (this->*addBranchFunc)(root);
        checkLowestLabel(label);
    }
}
```
Defines:
addStrongBranch, used in chunks 7a, 16a, 18, 19a, 24b, 30a, 39–41, 47a, 50b, 60a, 75, 83b, 92, and 118a.
Uses addBranchFunc 26a, branchInBucket 25a, checkLowestLabel 25c, getExcess 151d, getLabel 138g, getParentNode 141a, Node 164b, numNodes 3d, and PhaseSolver 136c.

Determining if a branch is already in a bucket is a little tricky because we don't have a flag that to tell us directly if the branch is already in the bucket. Instead, we will just look at the `next` pointer. If the pointer is non-null, then it must already be in a bucket. However, the pointer might be null if it's the last node in the bucket, so we have to check for that too. If it is in the bucket, we'll check its label against the label of the nodes already in the bucket.

24d ⟨*Solver methods* 4a⟩+≡ (136c) ◁24b 25b▷
```
bool branchInBucket(Node& root);
```
Uses branchInBucket 25a and Node 164b.

25a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁17c 25c▷

```
INLINE bool PhaseSolver::branchInBucket(Node& root)
{
    bool result = false;
    if ((root.getNext() != nil) ||
        (buckets[root.getLabel()].getTail() == &root))
    {
        result = true;
        assert(
            (buckets[root.getLabel()].getHead()->getLabel() == root.getLabel())
            || (buckets[root.getLabel()].getHead()->getParentNode() != nil)
        );
    }

    return result;
}
```

Defines:
`branchInBucket`, used in chunks 24, 50a, and 55a.
Uses `buckets` 23b, `getHead` 22a 166d, `getLabel` 138g, `getNext` 150g, `getParentNode` 141a, `getTail` 22a 166d, `Node` 164b, and `PhaseSolver` 136c.

We keep track of the lowest label every time a new branch is put in a bucket.

25b ⟨*Solver methods* 4a⟩+≡ (136c) ◁24d 26b▷

```
    void checkLowestLabel(NodeLabel label);
```

Uses `checkLowestLabel` 25c.

25c ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁25a 27d▷

```
INLINE void PhaseSolver::checkLowestLabel(NodeLabel label)
{
    if (label < lowestLabel) {
        lowestLabel = label;
    } else if (label > highestLabel) {
        highestLabel = label;
    }
}
```

Defines:
`checkLowestLabel`, used in chunks 24c, 25b, 45a, and 51b.
Uses `highestLabel` 24a, `lowestLabel` 24a, and `PhaseSolver` 136c.

25d ⟨*Solver protected members* 3a⟩+≡ (136c) ◁5b 27a▷

```
    AddBranchPtr addBranchFunc;
```

Uses `addBranchFunc` 26a and `AddBranchPtr` 136a.

26a ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁24a 27b▷
```
addBranchFunc = &PhaseSolver::addBranchFifo;
```
Defines:
addBranchFunc, used in chunks 4b, 24c, 25d, 29a, 34, 119a, 195, 198a, 200c, 202, 210, and 211.
Uses addBranchFifo 26c and PhaseSolver 136c.

26b ⟨*Solver methods* 4a⟩+≡ (136c) ◁25b 27e▷
```
void addBranchFifo(Node& root);
void addBranchLifo(Node& root);
void addBranchWave(Node& root);
```
Uses addBranchFifo 26c, addBranchLifo 26c, and Node 164b.

26c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁24c 26d▷
```
void PhaseSolver::addBranchFifo(Node& root)
{
    buckets[root.getLabel()].insertTail(root);
}
void PhaseSolver::addBranchLifo(Node& root)
{
    buckets[root.getLabel()].insertHead(root);
}
```
Defines:
addBranchFifo, used in chunks 6a, 26, 45a, and 200c.
addBranchLifo, used in chunks 26b, 51b, 198a, and 200c.
Uses buckets 23b, getLabel 138g, insertHead 21, insertTail 21, Node 164b, and PhaseSolver 136c.

The "wave" branch management is rather weak: it looks to see if the branch root we are adding is the same as the branch root we started processing. If so it puts the branch at the head of the list. This is lame because what we really want to know is if this branch is the same branch as we are processing. If we just did a merger with it, it was probably rehung so that the root is no longer the same.

26d ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁26c 28a▷
```
void PhaseSolver::addBranchWave(Node& root)
{
    if (lastRoot == &root) {
        buckets[root.getLabel()].insertHead(root);
    } else {
        buckets[root.getLabel()].insertTail(root);
    }
}
```
Uses buckets 23b, getLabel 138g, insertHead 21, insertTail 21, lastRoot 27b, Node 164b, and PhaseSolver 136c.

The last branch root returned by `getLowestBranch`.

27a ⟨*Solver protected members* 3a⟩+≡ (136c) ◁25d 27c▷
```
NodePtr lastRoot;
```
Uses `lastRoot` 27b.

27b ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁26a 31c▷
```
lastRoot = nil;
```
Defines:
`lastRoot`, used in chunks 26, 27, 36b, 40d, 41d, 49, and 50a.

As the algorithm runs, we need to remove the lowest labeled branch. First we need to locate the lowest labeled bucket, then we need to remove the head of the list. If there are no more strong branches, we just return `nil`.

The search always starts at the `lowestLabel` bucket. As we scan buckets, we increment `lowestLabel` until we find a non-empty bucket.

27c ⟨*Solver protected members* 3a⟩+≡ (136c) ◁27a 58b▷
```
NodePtr getLowestBucket();
```
Uses `getLowestBucket` 27d.

27d ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁25c 78a▷
```
INLINE NodePtr PhaseSolver::getLowestBucket()
{
    NodePtr result = nil;
    while (lowestLabel <= highestLabel) {
        result = buckets[lowestLabel].removeHead();
        if (result != nil) {
            result->setNextNil();
            break;
        } else {
            lowestLabel++;
        }
    }

    lastRoot = result;
    return result;
}
```
Defines:
`getLowestBucket`, used in chunks 27c, 28a, and 46a.
Uses `buckets` 23b, `highestLabel` 24a, `lastRoot` 27b, `lowestLabel` 24a, `PhaseSolver` 136c, `removeHead` 21, and `setNextNil` 150e.

The above method does the real work, but we have a virtual wrapper function to allow us to modify the behavior for the Simplex solver. Here in the base class, we just check the label.

27e ⟨*Solver methods* 4a⟩+≡ (136c) ◁26b 28b▷
```
virtual NodePtr getLowestBranch();
```
Uses `getLowestBranch` 28a 46a.

28a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁26d 29a▷

```
NodePtr PhaseSolver::getLowestBranch()
{
    NodePtr result = getLowestBucket();
    assert((result != nil) ? (result->getLabel() == lowestLabel) : 1);
    return result;
}
```

Defines:
`getLowestBranch`, used in chunks 4b, 6a, 27e, 29, and 45c.
Uses `getLabel` 138g, `getLowestBucket` 27d, `lowestLabel` 24a, and `PhaseSolver` 136c.

2.4 Delayed Normalization

An alternative solving method involves 'delayed normalization'. In this case, we process the branches as before, but during a merger, the tree is not totally renormalized. In particular, we do not push excess up the weak branch. This leaves some 'pseudo-weak' branches, branches that we treat as weak, but they have excess in them that may, in fact, make them strong. At the end of a phase, we need to scan for deferred nodes and complete normalization on them. This may lead to the creation of more strong branches, in which case we need to continue executing the algorithm.

28b ⟨*Solver methods* 4a⟩+≡ (136c) ◁27e 30c▷

```
void delayedNormalizeSolve(AddBranchPtr addFunc);
```

Uses `AddBranchPtr` 136a and `delayedNormalizeSolve` 29a.

29a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁28a 31a▷

```
void PhaseSolver::delayedNormalizeSolve(AddBranchPtr addFunc)
{
    addBranchFunc = addFunc;
    renormalizeFunc = &PhaseSolver::strongOnlyRenormalize;
    initGlobalRelabel();

    int currentPhase = 0;
    NodePtr strongBranch = nil;
    NodeLabel strongLabel =  0;

    while (true) {
        ⟨get strong branch - normalize if needed 29b⟩
        ⟨terminate early based on counting labels 5a⟩
        processBranch(*strongBranch);
    }
}
```

Defines:
`delayedNormalizeSolve`, used in chunks 28b and 200b.
Uses `addBranchFunc` 26a, `AddBranchPtr` 136a, `initGlobalRelabel` 78c, `PhaseSolver` 136c, `processBranch` 7a 47b, `renormalizeFunc` 15c, and `strongOnlyRenormalize` 31a.

To get the next strong branch, we call `getLowestBranch`, but that might return `nil` if the only remaining 'strong' branches are still weak due to deferred normalization. In that case, we perform the deferred normalizations and get the lowest branch again.

29b ⟨*get strong branch - normalize if needed* 29b⟩≡ (29a 119a) 30a▷

```
strongBranch = getLowestBranch();
if (strongBranch == nil) {
    performDeferredNormalizations();
    ⟨get non-null branch 29c⟩
}
```

Uses `getLowestBranch` 28a 46a and `performDeferredNormalizations` 32a.

If we get `nil` again, we really are done and should break out of the loop.

29c ⟨*get non-null branch* 29c⟩≡ (29b 30b)

```
strongBranch = getLowestBranch();
if (strongBranch == nil) {
    break;
}
```

Uses `getLowestBranch` 28a 46a.

Once we have the branch, we look to see if its label is different than the current phase - i.e. are we trying to go to a new (higher) phase. If so, we put the branch back and do the renomalizations, which might lead to a lower phase, which is why we put it back.

30a ⟨*get strong branch - normalize if needed* 29b⟩+≡ (29a 119a) ◁29b 30b▷

```
strongLabel =  strongBranch->getLabel();
if (strongLabel != currentPhase) {
    addStrongBranch(*strongBranch); // should addHead
    performDeferredNormalizations();
    checkForRelabel();
```

Uses `addStrongBranch` 24c 51a, `checkForRelabel` 78a, `getLabel` 138g, and `performDeferredNormalizations` 32a.

Now, we try again to get a new branch. If the label is different than the current phase, then we really are starting a new phase.

30b ⟨*get strong branch - normalize if needed* 29b⟩+≡ (29a 119a) ◁30a

```
    ⟨get non-null branch 29c⟩
    strongLabel =  strongBranch->getLabel();
    if (strongLabel != currentPhase) {
        currentPhase = strongLabel;
        TRACE( trout << "Phase " << currentPhase << endl;)
    }
}
```

Uses `getLabel` 138g.

During the normal merger function, we enable deferred normalization by using the follow function that will only renormalize the strong branch. This will leave excess at the weak node. The weak branch will still be considered weak so that strong nodes can still merge to it.

30c ⟨*Solver methods* 4a⟩+≡ (136c) ◁28b 31d▷

```
void strongOnlyRenormalize(Node& strongRoot, Node& weakNode);
```

Uses `Node` 164b and `strongOnlyRenormalize` 31a.

31a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁29a 32a▷

```
void PhaseSolver::strongOnlyRenormalize(Node& strongRoot, Node& weakNode)
{
    strongPush(strongRoot, weakNode);
    if (weakNode.flagIsSet(DEFERRED) == false) {
        weakNode.setFlag(DEFERRED);
        weakNode.next2 = deferredNodes;
        deferredNodes = &weakNode;
    }
}
```

Defines:
`strongOnlyRenormalize`, used in chunks 29a, 30c, and 119a.
Uses `deferredNodes` 31c, `flagIsSet` 155d, `next2` 150c, `Node` 164b, `PhaseSolver` 136c, `setFlag` 155d, and `strongPush` 16a.

31b ⟨*Solver data* 3e⟩+≡ (136c) ◁3e 63a▷

```
NodePtr deferredNodes;
static const int DEFERRED = 0x10;
```

Uses `deferredNodes` 31c.

31c ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁27b 63b▷

```
deferredNodes = nil;
```

Defines:
`deferredNodes`, used in chunks 31 and 32.

To perform the delayed normalizations, we run down the list of deferred nodes. For each node, we perform a complete weak push up to the weak root. Note that it is possible for multiple nodes in a branch to be weak (actually, we're hoping for that). This means that a deferred push could make a branch strong, and later we'll be doing another push to the same branch - i.e. a 'weak' push within a strong branch.

This is fine, but it does mean that we cannot use the same 'next' pointer to build the list of deferred weak nodes as we use for building the list of strong nodes. A deferred weak node (still in the list of weak nodes) could be made into a strong branch root due to a push from a descendent node, which means it needs to be in the strong list.

31d ⟨*Solver methods* 4a⟩+≡ (136c) ◁30c 33▷

```
bool performDeferredNormalizations();
```

Uses `performDeferredNormalizations` 32a.

32a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁31a 34▷

```
bool PhaseSolver::performDeferredNormalizations()
{
    int numWeakPushes = 0;
    int numSplittingPushes = 0;

    NodePtr currNode = deferredNodes;
    while (currNode != nil) {
        deferredNodes = currNode->next2;
        currNode->next2 = nil;
        ⟨normalize deferred node 32b⟩
    }

    deferredNodes = nil;

    TRACE( trout << "Performed " << numWeakPushes << " pushes with "
                 << numSplittingPushes << " splitting pushes"  << endl; )

    return (numSplittingPushes > 0) ? true : false;
}
```

Defines:
`performDeferredNormalizations`, used in chunks 29–31.
Uses `deferredNodes` 31c, `next2` 150c, and `PhaseSolver` 136c.

32b ⟨*normalize deferred node* 32b⟩≡ (32a)

```
    if (currNode->flag != 0) {
        if (currNode->getExcess() > 0) {
            numWeakPushes++;
            TRACE( trout << "deferred push from node " << currNode->getId()
                         << " label " << currNode->getLabel() << endl;)
            if (weakPush(*currNode)) {
                numSplittingPushes++;
            }
        } else {
            TRACE( trout << "no deferred push from node " << currNode->getId() << endl; )
        }
        currNode->flag = 0;
    }
    currNode = deferredNodes;
```

Uses `deferredNodes` 31c, `flag` 155d, `getExcess` 151d, `getLabel` 138g, and `weakPush` 18.

There is another strategy for performing deferred normalizations that we have not yet implemented. The basic idea is to try to push in parallel from the leaves up to the root. What we'd like to acheive is that the excess from two parallel paths would merge at one node and we'd then only have to do one push from that node up to the root.

We cannot really implement that directly. We can approximate it by maintaining another set of buckets indexed by node label. We process it from maximum label down to zero. We push excess from a node up along the path to the root so long as the parent has the same label. As soon as we push flow up to a new labeled node, we queue that node to be processed later when we get to that label. This scheme might not detect the merger of two paths immediately, but it would be detected as soon as we crossed into a new label.

This scheme might be an improvement due to coalescing the excess from multiple paths into a single push up to the root. On the other hand, it involves a fair amount of overhead to be queueing and dequeueing the nodes all the time. Not to mention, it's more compilicated, of course.

2.5 Highest Root Label

Here is a totally different solving function. It is inspired by the highest label variant of push-relabel. Due to our tree/branch and label structure, we cannot exactly perform a highest label on a per-node basis the way push-relabel does. For example, if we choose the highest labeled node in a branch and increase its label arbitrarily after an empty scan, then things will fail. If we loose the bound on the difference of node labels in the graph, then we can't bound the maximum label of a node (invariant 1). It would seem strange to have strong nodes with residual capacity between them, but widely different labels. If we try to prune a node, but other nodes in the branch have access to weak nodes, then the node should still be active.

So, instead what we do is process the branch with the highest labeled root. We process it in the usual way - from the root down only looking at nodes with label ℓ. If we do not merge, we will relabel the top part of the branch and put it back in the bucket list. This will surely be the highest labeled branch, so we will process it again.

At some point, we will achieve a condition similar to the main algorithm's early termination - i.e. there will be no nodes in the tree with label one less than ours. We cannot terminate the algorithm at this time, but we can prune the branch (subject to some conditions discussed below).

For this method, we do not support delayed normalization, and we did not even consider implementing it for Simplex. Therefore, it is mutually exclusive with delayed normalization and simplex.

33 ⟨*Solver methods* 4a⟩+≡ (136c) ◁31d 35b▷

```
void highestLabelSolve(AddBranchPtr addFunc);
```

Uses `AddBranchPtr` 136a and `highestLabelSolve` 34.

34 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁32a 36a▷

```
void PhaseSolver::highestLabelSolve(AddBranchPtr addFunc)
{
    IFTRACE(int currentPhase = 0;)
    addBranchFunc = addFunc;
    renormalizeFunc = &PhaseSolver::fullRenormalize;
    initGlobalRelabel();

    NodePtr strongBranch = getHighestBranch();
    while (strongBranch != nil) {
        NodeLabel strongLabel =  strongBranch->getLabel();
        IFTRACE(
            if ( strongLabel != currentPhase) {
                currentPhase = strongLabel;
                TRACE(trout << "Phase " << currentPhase << endl;);
            }
        )
        ⟨prune gap branch if needed 35a⟩
        bool performedMerger = processBranch(*strongBranch);
        STATS(performedMerger ? 0 : numEmptyBranchScans++ );
        checkForRelabel();
        strongBranch = getHighestBranch();
    }
    IFDEBUG(checkMergers());
}
```

Defines:
`highestLabelSolve`, used in chunks 33, 119a, and 200b.
Uses `addBranchFunc` 26a, `AddBranchPtr` 136a, `checkForRelabel` 78a, `checkMergers` 134a, `fullRenormalize` 20b, `getHighestBranch` 36a, `getLabel` 138g, `initGlobalRelabel` 78c, `numEmptyBranchScans` 129b, `PhaseSolver` 136c, `processBranch` 7a 47b, and `renormalizeFunc` 15c.

If there is a gap, we can prune this branch and all others with the same label or higher. (Actually, we're only pruning this branch. We could/should prune all nodes with the same label, as push-relabel does.)

Note that this is a little tricky when we deal with something like saturate-all with distance to sink labeling. This creates weak nodes with fairly high labels. We can't simply look for no nodes with label $\ell - 1$ because after we've processed some strong nodes with high labels, we could get down to some that have labels less than some weak nodes out there. Therefore, during initializaiton, we need to figure out the highest labeled weak node and avoid pruning until we reach that level.

Another alternative would be to scan the label counts from `strongLabel` up to `maxWeakLabel`. If there are no nodes in that range, we could safely prune this branch without waiting for its label to exceed `maxWeakLabel`. On the one hand, that would involve an expensive iterative test here. On the other hand, midlessly raising the label of this strong branch above `maxWeakLabel` is expensive. Furthermore, if we make that iterative scan and it comes up empty, we could reduce `maxWeakLabel` to `strongLabel` thus reducing its overhead in the future.

35a ⟨*prune gap branch if needed* 35a⟩≡ (34)

```
if ((labelCount[strongLabel - 1] == 0) && (strongLabel > maxWeakLabel))
{
    TRACE(trout << "Found a gap at label " << strongLabel << endl);
    setBranchLabel(*strongBranch, numNodes);
    strongBranch = getHighestBranch();
    if (strongBranch == nil) {
        break;
    }
    strongLabel =  strongBranch->getLabel();
}
```

Uses `getHighestBranch` 36a, `getLabel` 138g, `labelCount` 5c, `maxWeakLabel` 63b, `numNodes` 3d, and `setBranchLabel` 84a.

The first, obvious thing we need for the highest labeled root solver, is a function to fetch the highest labeled branch. Confusingly, we will override the virtual function `getLowestBranch` to do this.

35b ⟨*Solver methods* 4a⟩+≡ (136c) ◁33 55b▷

```
NodePtr getHighestBranch();
```

Uses `getHighestBranch` 36a.

36a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁34 56a▷

```
NodePtr PhaseSolver::getHighestBranch()
{
    NodePtr result = nil;
    ⟨find highest non-empty bucket 36b⟩
    assert((result != nil) ? (result->getLabel() == highestLabel) : 1);
    return result;
}
```

Defines:
`getHighestBranch`, used in chunks 34 and 35.
Uses `getLabel` 138g, `highestLabel` 24a, and `PhaseSolver` 136c.

Finding the highest labeled non-empty bucket, is basically the reverse of `getLowestBucket`: search from `highestLabel` down to `lowestLabel`.

36b ⟨*find highest non-empty bucket* 36b⟩≡ (36a)

```
    while(highestLabel > lowestLabel) {
        result = buckets[highestLabel].removeHead();
        if (result != nil) {
            result->setNextNil();
            break;
        } else {
            highestLabel--;
        }
    }

    lastRoot = result;
```

Uses `buckets` 23b, `highestLabel` 24a, `lastRoot` 27b, `lowestLabel` 24a, `removeHead` 21, and `setNextNil` 150e.

2.6 Simplex Merger

Another, more involved method of solving is using a 'simplex-style' merger function. The simplex merger avoids multiple splits during the merger computing the bottleneck capacity along the cycle from the strong root, through the merger arc, and up to the weak root. We only push excess equal to this bottleneck capacity, so there will be at most one split during the merger, in most cases the strong root will remain strong. Note that due to the complexity of all of this, we do not support delayed normalization.

As we shall see, this is a pretty significant deviation from the code we have develeoped above, we will implement this as a sub-class of `PhaseSolver` rather than relying on tricks with pointers to functions.

37a ⟨*Simplex declaration* 37a⟩≡ (137b)

```
class SimplexSolver : public PhaseSolver
{
    public: ⟨Simplex public declarations 37b⟩
    private: ⟨Simplex private declarations 44a⟩
};
```

Defines:
`SimplexSolver`, used in chunks 37c, 42b, 44–48, 51a, 53–55, and 200a.
Uses `PhaseSolver` 136c.

2.6.1 Simplex merge function

37b ⟨*Simplex public declarations* 37b⟩≡ (37a) 42a▷

```
virtual void merge(Node& strong, Node& weak, Edge& edge);
```

Uses `Edge` 179a, `merge` 15a 37c, and `Node` 164b.

37c ⟨*Simplex implementations* 37c⟩≡ (135c) 42b▷

```
void SimplexSolver::merge(Node& strong, Node& weak, Edge& edge)
{
    TRACE( trout << "simplexMerge " << strong.getId() << " ("
                 << strong.getRootExcess() << ") to "
                 << weak.getId() << " ("
                 << weak.getExcess() << ", "
                 << weak.getRootExcess() << "), cap "
                 << edge.residCapacity(strong) << ", s-label "
                 << strong.getLabel() << endl;)
    STATS(numMergers++);

    ⟨simplex merge function 38a⟩
}
```

Defines:
`merge`, used in chunks 11a, 14b, 37b, and 200a.
Uses `Edge` 179a, `getExcess` 151d, `getLabel` 138g, `getRootExcess` 162c, `Node` 164b, `numMergers` 129b, `residCapacity` 169b, and `SimplexSolver` 37a.

Compute the bottleneck capacity in the strong and weak branches.

38a ⟨*simplex merge function* 38a⟩≡ (37c) 38b▷

```
FlowAmount strongCapacity, weakCapacity;
NodePtr strongChild, weakChild;
Node& strongRoot = findBottleneckArc(strong, true, strongCapacity, strongChild);
Node& weakRoot = findBottleneckArc(weak, false, weakCapacity, weakChild);
FlowAmount mergerCapacity = edge.residCapacity(strong);
FlowAmount excess = strongRoot.getExcess();
TRACE( trout << "strongCapacity=" << strongCapacity <<
                ", mergerCapacity=" << mergerCapacity <<
                ", weakCapacity=" << weakCapacity << endl;);
```

Uses `excess` 151b, `findBottleneckArc` 53a, `getExcess` 151d, `Node` 164b, and `residCapacity` 169b.

First, let's address a special case that seems to come up a lot with simplex: zero-excess root. Initially, we used to filter them out via `getLowestBranch` and `addStrongBranch`, but that leads to weak branches with roots that have label greater than one. Therefore, we'll just merge it to a weak branch without worrying about pushing any flow around.

38b ⟨*simplex merge function* 38a⟩+≡ (37c) ◁38a 39a▷

```
if (excess == 0) {
    ⟨fix degenerate flags for whole strong branch 48b⟩
    strong.rehangWeak();
    weak.addChild(strong, edge);
    CHECK_TREE(weakRoot, nil);
}
```

Uses `addChild` 142a, `excess` 151b, and `rehangWeak` 149.

Figure out where the bottleneck is. If there is less excess than the bottlenck, we will just perform a normal merger.

39a ⟨*simplex merge function* 38a⟩+≡ (37c) ◁38b

```
else if ((excess < strongCapacity) && (excess < mergerCapacity) &&
    (excess < weakCapacity))
{
    // normal merger without splits
    ⟨fix degenerate flags for whole strong branch 48b⟩
    mergeStrongBranch(strong, weak, edge);
} else if ((strongCapacity <= mergerCapacity) &&
           (strongCapacity <= weakCapacity))
{
    ⟨strong side bottleneck 39b⟩
} else if ((strongCapacity > mergerCapacity) &&
           (mergerCapacity <= weakCapacity))
{
    ⟨merger arc bottleneck 40b⟩
} else {
    ⟨weak side bottleneck 40e⟩
}
```

Uses `excess` 151b and `mergeStrongBranch` 42b.

• If the bottleneck is on the strong side, we need to pull as much flow down from the strong root as possible (i.e. the bottleneck capacity).

39b ⟨*strong side bottleneck* 39b⟩≡ (39a) 39c▷

```
assert(strongChild != nil);
assert(strongChild->getParentDownCapacity() == strongCapacity);
pullFromRoot(*strongChild, strongCapacity);
assert(strongChild->getExcess() == strongCapacity);
NodePtr newRoot = findNewRoot(strongRoot, *strongChild);
strongChild->split();
CHECK_TREE(strongRoot, nil);
```

Uses `findNewRoot` 48d, `getExcess` 151d, `getParentDownCapacity` 141a, `pullFromRoot` 54a, and `split` 145c.

We need to fix up the degenerate branch flags, which will be explained later.

39c ⟨*strong side bottleneck* 39b⟩+≡ (39a) ◁39b 39d▷

```
⟨fix degenerate flags in strong branch 48a⟩
```

The call above to `findNewRoot` computes which node, if any, from the strong branch needs to be re-inserted in the buckets.

39d ⟨*strong side bottleneck* 39b⟩+≡ (39a) ◁39c 40a▷

```
if (newRoot != nil) {
    addStrongBranch(*newRoot);
}
```

Uses `addStrongBranch` 24c 51a.

The remaining fragment of the strong branch can be merged with the weak node, and we can will let our variant of `merge` in the base class push from the child node (which is now the root of the fragment) all the way up to the weak root.

40a ⟨*strong side bottleneck* 39b⟩+≡ (39a) ◁39d
```
mergeStrongBranch(strong, weak, edge);
```
Uses `mergeStrongBranch` 42b.

• When the merger arc is the bottleneck, the processing is similar except that we don't have to merge a fragment of the strong branch to the weak branch. We pull down as much flow as we can to the strong node.

40b ⟨*merger arc bottleneck* 40b⟩≡ (39a) 40c▷
```
pullFromRoot(strong, mergerCapacity);
assert(strong.getExcess() >= mergerCapacity);
```
Uses `getExcess` 151d and `pullFromRoot` 54a.

Push the flow to the weak node (saturating the merger arc).

40c ⟨*merger arc bottleneck* 40b⟩+≡ (39a) ◁40b 40d▷
```
edge.setDirectionTo(&weak);
STATS(numPushToParent++);
edge.increaseFlow(mergerCapacity);
strong.decrementExcess(mergerCapacity);
weak.incrementExcess(mergerCapacity);
CHECK_TREE(strongRoot, nil);
```
Uses `decrementExcess` 151d, `increaseFlow` 168a, `incrementExcess` 151d, and `numPushToParent` 129b.

Then we insert the last strong root and perform a weak push (which won't split). We know that the last root is the correct branch to insert (instead of calling `findNewRoot`) because the merger is happening at or below the last root, and it's still strong because the merger arc is the bottleneck.

40d ⟨*merger arc bottleneck* 40b⟩+≡ (39a) ◁40c
```
addStrongBranch(*lastRoot);
weakPush(weak);
CHECK_TREE(weakRoot, nil);
```
Uses `addStrongBranch` 24c 51a, `lastRoot` 27b, and `weakPush` 18.

• Finally, when the bottleneck arc is on the weak side, the code is rather unique. We want to end up with the weak node dangling from the strong node, which is the reverse of how we usually think of mergers. Again we will start by pulling flow down to the strong node.

40e ⟨*weak side bottleneck* 40e⟩≡ (39a) 41a▷
```
pullFromRoot(strong, weakCapacity);
assert(strong.getExcess() >= weakCapacity);
```
Uses `getExcess` 151d and `pullFromRoot` 54a.

Next, we will push it over the merger arc to the weak node. Then, we push it up to the bottleneck child on the weak side using `strongPush`, even though it's the weak side of the tree.

41a ⟨*weak side bottleneck* 40e⟩+≡ (39a) ◁40e 41b▷

```
edge.setDirectionTo(&weak);
STATS(numPushToParent++);
edge.increaseFlow(weakCapacity);
strong.decrementExcess(weakCapacity);
weak.incrementExcess(weakCapacity);
strongPush(weak, *weakChild);
```

Uses `decrementExcess` 151d, `increaseFlow` 168a, `incrementExcess` 151d, `numPushToParent` 129b, and `strongPush` 16a.

Now, we push the flow across the bottleneck arc, which we can just use `pushToParent` since the pointers parent-child relationship is correct. However, we still need to split ourselves.

41b ⟨*weak side bottleneck* 40e⟩+≡ (39a) ◁41a 41c▷

```
NodePtr weakParent = weakChild->getParentNode();
assert(weakParent != nil);
STATS(numPushToParent++);
weakChild->pushToParent();
weakChild->split();
assert(weakChild->getExcess() == 0);
```

Uses `getExcess` 151d, `getParentNode` 141a, `numPushToParent` 129b, `pushToParent` 152b, and `split` 145c.

At this point, we have excess in the amount of the bottleneck capacity sitting on the weak parent node, so we can easily push it up to the weak root without any splitting. The `weakPush` method will also take care of the case where we push enough excess to the root that the branch becomes strong.

41c ⟨*weak side bottleneck* 40e⟩+≡ (39a) ◁41b 41d▷

```
weakPush(*weakParent);              // no split
CHECK_TREE(weakRoot, nil);
```

Uses `split` 145c and `weakPush` 18.

Finally, we need to clean things up. The weak fragment from `weak` to `weakChild` needs to be rehung and attached under the strong node. And then the strong node needs to be re-inserted into the strong buckets.

41d ⟨*weak side bottleneck* 40e⟩+≡ (39a) ◁41c

```
weak.rehangWeak();
strong.addWeakChild(weak, edge);
⟨set degenerate flags in weak branch 43⟩
CHECK_TREE(strongRoot, nil);
addStrongBranch(*lastRoot);
```

Uses `addStrongBranch` 24c 51a, `addWeakChild` 142c, `lastRoot` 27b, and `rehangWeak` 149.

In the code above, we used to call the base class `merge` function to merge part or all of the strong branch to the weak branch. However this causes problems when we've been dealing with degenerate branches. In the base class, we only rehang trees in the strong branch, so we can exploit the 'current element' in the children arrays to speed rehanging and subsequent searches of the strong branches. With degenerate branches, this breaks down.

Therefore, we have a copy of the base class merge function with different calls to tree manipulation functions. Also, we hard-code the renomalization function to renormalize the whole tree.

42a ⟨*Simplex public declarations* 37b⟩+≡ (37a) ◁37b 45c▷

```
void mergeStrongBranch(Node& strong, Node& weak, Edge& edge);
```

Uses `Edge` 179a, `mergeStrongBranch` 42b, and `Node` 164b.

42b ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁37c 44b▷

```
void SimplexSolver::mergeStrongBranch(Node& strong, Node& weak, Edge& edge)
{
    Node& strongRoot = *strong.rehangWeak();
    weak.addChild(strong, edge);
    CHECK_TREE(*weak.getRoot(), nil);
    // renormalize the tree
    fullRenormalize(strongRoot, weak);
}
```

Defines:
`mergeStrongBranch`, used in chunks 39a, 40a, and 42a.
Uses `addChild` 142a, `Edge` 179a, `fullRenormalize` 20b, `getRoot` 161c, `Node` 164b, `rehangWeak` 149, and `SimplexSolver` 37a.

2.6.2 Degenerate branches

The code above for the three classes of mergers looks plausible, but there is a serious problem during a merger when the bottleneck is on the weak side. Suppose during phase ℓ, the strong node is s, the weak node is w, and the bottleneck arc is (u, v), where u is closest to w. We want to suspend the fragment of the weak branch below the bottleneck from w, and attach it to the strong node, s, via the merger arc. This violates the monotoicity of paths from the strong root because we know that w has label $\ell - 1$, while s has label ℓ. The other weak nodes have labels less than or equal to $\ell - 1$.

In terms of algorithmic complexity, this violation of monotonicity is not a problem on the strong side, but it is a problem in our implementation because we count on being able to begin scanning a strong branch (during phase ℓ) from the root and knowing that labels only increase with depth so that when we encounter a node with a higher label (say $\ell + 1$), we can stop scanning that sub-tree during phase ℓ.

We will accomodate this by introducing the concept of *degenerate sub-trees*. After a simplex merger with the bottleneck arc in the weak branch, we rehang the weak fragment from w and attach it to s. We call this branch 'degenerate' because we know that the label monotoicity has been violated. A merger that creates a degenerate branch is also called degenerate. We know that all the nodes along the path from w to u have labels less than ℓ that are monotically decreasng. There may also be descendent nodes of the nodes along the path with labels less than ℓ, but they will maintain monotonicity within their local sub-trees.

We need to process these nodes and their descendents before we process any of the nodes with label ℓ again. (Note that all nodes along the path from s up to the strong root have label ℓ because we were in phase ℓ at the time of the merger.)

We process these nodes from the leaf node, u, up long the path of parent pointers until we get back to s. As we process each of these nodes and their children, we will be either merging or relabeling the nodes. If we reach s without merging, we will know that all nodes below s have label ℓ or more because when we cannot find a merger for a node, we increment its label. At this point, monotonicity has been restored (it is no longer degenerate) and we can place the branch into a strong bucket to continue processing during phase ℓ.

To accomplish this, we will treat each node along the path from the weak child up to the weak node as the root of a strong branch. We can process these sub-trees to yield mergers or increase the label.

This is the final step in the merger code with the bottleneck in the weak branch. Although we could do this inline (especially with the aid of `noweb`, we will use a separate function so that we can profile the performance of this function.

43 ⟨*set degenerate flags in weak branch* 43⟩≡ (41d)

```
queueDegeneratePath(*weakChild, weak);
```

44a ⟨*Simplex private declarations* 44a⟩≡ (37a) 44c▷

```
void queueDegeneratePath(Node& first, Node&last);
```

Uses Node 164b.

This function will involve tracing up the path from the first node to the last node adding each node as a new strong root.

44b ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁42b 45a▷

```
void SimplexSolver::queueDegeneratePath(Node& first, Node& last)
{
    NodePtr currNode = &first;
    while (currNode != &last) {
        addDegenerateBranch(*currNode);
        currNode = currNode->getParentNode();
    }
    addDegenerateBranch(last);
}
```

Defines:
queueDegneratePath, never used.
Uses addDegenerateBranch 45a, getParentNode 141a, Node 164b, and SimplexSolver 37a.

Adding degenerate branches is similar to addStrongBranch except that the assumptions are very different - e.g. we're not dealing with root nodes, and a degenerate branch better not already be in a bucket. Furthermore, we want to mark degenerate nodes to keep track of them if they get merged to a weak branch. One subtle, yet significant difference is that we always want to add degenerate roots to the tail of the bucket. This way if a node and its parent both have the same label, we will process the child before the parent - i.e. move up along the path towards the original strong node.

44c ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁44a 45b▷

```
void addDegenerateBranch(Node& root);
```

Uses addDegenerateBranch 45a and Node 164b.

45a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁44b 46a▷

```
void SimplexSolver::addDegenerateBranch(Node& root)
{
    assert(isStrongNode(root) == false);
    NodeLabel label = root.getLabel();
    assert(root.getNext() == nil);

    // mark the branch and add it
    assert((root.flag & !ADJACENT) == 0);
    root.setFlag(DEGENERATE_SUBTREE);
    CHECK_TREE(root, root.getParentEdge());
    addBranchFifo(root);

    checkLowestLabel(label);
}
```

Defines:
addDegenerateBranch, used in chunk 44.
Uses addBranchFifo 26c, checkLowestLabel 25c, DEGENERATE_SUBTREE 45b, flag 155d, getLabel 138g, getNext 150g, getParentEdge 141a, isStrongNode 17c, Node 164b, setFlag 155d, and SimplexSolver 37a.

45b ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁44c 47b▷

```
static const int DEGENERATE_SUBTREE = 4;
```

Defines:
DEGENERATE_SUBTREE, used in chunks 45–48, 50a, 51a, and 55a.

So, now that we have degenerate branches in the buckets, we need to deal with them. The first issue is getting them out. The problem is that between the time we added a degenerate node and when we find it as a strong root, there might have been a merger that 'demoted' it to weak. In this case, we need to ignore it and continue looking.

One subtle case to be aware of is when we queue the degenerate branch, we queue the root of the entire branch as well as the degenerate nodes along the path below `strong`. If there is a subsequent merger where there is no bottleneck, the entire strong branch will be rehung and made weak. The root for the entire strong branch will still be in its bucket. However, now it will no longer be a root, so we will know to ignore it.

45c ⟨*Simplex public declarations* 37b⟩+≡ (37a) ◁42a 46b▷

```
virtual NodePtr getLowestBranch();
```

Uses getLowestBranch 28a 46a.

46a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁45a 47a▷

```
NodePtr SimplexSolver::getLowestBranch()
{
    NodePtr result = getLowestBucket();
    while (result != nil) {
        if (result->isRootNode()) {
            // a normal strong root - may have zero exess
            break;
        } else if (result->flagIsSet(DEGENERATE_SUBTREE)) {
            // found a degenerate root that is still strong
            break;
        }
        result = getLowestBucket();
    }

    return result;
}
```

Defines:
`getLowestBranch`, used in chunks 4b, 6a, 27e, 29, and 45c.
Uses `DEGENERATE_SUBTREE` 45b, `flagIsSet` 155d, `getLowestBucket` 27d, `isRootNode` 141a, and `SimplexSolver` 37a.

Once we get the lowest labeled branch, we will need to process it. This is basically the same as we do in the base class, except that we want to keep an eye out for degenerate sub-trees. After we process on of these, if there were no mergers (i.e. it was relabeled), we do not want to put it back in the bucket because we know its monotonicity has been restored.

46b ⟨*Simplex public declarations* 37b⟩+≡ (37a) ◁45c 48c▷

```
virtual bool processBranch(Node& root);
```

Uses `Node` 164b and `processBranch` 7a 47b.

47a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁46a 48d▷

```
bool SimplexSolver::processBranch(Node& root)
{
    bool performedMerger = false;
    degenerateBranch = root.flagIsSet(DEGENERATE_SUBTREE);

    if (labelCount[root.getLabel() - 1] == 0) {
        TRACE( trout << "Relabel branch " << root.getId()
                     << " with label " << root.getLabel() << endl; )
        relabelSubtree(root);
        if (degenerateBranch) {
            root.clearFlag(DEGENERATE_SUBTREE);
        } else {
            addStrongBranch(root);
        }
    } else {
        performedMerger = processSubtree(root);
        if (!performedMerger) {
            if (degenerateBranch) {
                root.clearFlag(DEGENERATE_SUBTREE);
            } else {
                addStrongBranch(root);
            }
        }
    }
    IFCHECK { validateDegenerateBranches(); }
    return performedMerger;
}
```

Uses `addStrongBranch` 24c 51a, `clearFlag` 155d, `DEGENERATE_SUBTREE` 45b, `flagIsSet` 155d, `getLabel` 138g, `labelCount` 5c, `Node` 164b, `processBranch` 7a 47b, `processSubtree` 8a, `relabelSubtree` 14a, `SimplexSolver` 37a, and `validateDegenerateBranches` 55a.

47b ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁45b 52▷

```
bool degenerateBranch;
```

Defines:
`processBranch`, used in chunks 4b, 6b, 29a, 34, 46b, 47a, and 119a.

The code above creates degenerate branches during mergers with wthe bottleneck in the weak branch. One tricky complication is if we have a merger from a degenerate branch where the bottleneck arc is within the strong, degenerate branch, some of the nodes we flagged and put into buckets could become weak. In which case, we no longer want to process them as strong branches.

Rather than digging through the buckets trying to pull these nodes out, we will simply clear their degnerate flag. We will do this after the bottleneck arc has been saturated but before the strong branch has been merged and rehung. Therefore, we want to trace the path from `strong` up to `strongChild` clearing the degenerate flag. (*We could be a tad more efficient about this because we know that the degenerate nodes will be contiguous along the path, but there might be some non-degenerate nodes at the begining of the path. We could quit looking upon reaching the end of the degenerate portion of the path.*)

48a ⟨*fix degenerate flags in strong branch* 48a⟩≡ (39c)

```
if (degenerateBranch) {
    strong.clearFlagPath(DEGENERATE_SUBTREE, strongChild);
}
```

Uses `clearFlagPath` 156a and `DEGENERATE_SUBTREE` 45b.

Similarly, when there is no bottleneck arc, and the entire strong branch becomes weak, we need to clear the degenerate flags for the whole path from the strong merger node up to the strong root. We do this before performing the merger, so `strong` is at the bottom of the path and `strongRoot` is at the top.

48b ⟨*fix degenerate flags for whole strong branch* 48b⟩≡ (38b 39a)

```
if (degenerateBranch) {
    strong.clearFlagPath(DEGENERATE_SUBTREE, nil);
}
```

Uses `clearFlagPath` 156a and `DEGENERATE_SUBTREE` 45b.

In addition to tweaking the degenerate flags in the strong branch, we need to figure out what node to queue in a bucket as a strong root. This is non-trivial to figure out because need to to figure out where the breakpoint arc is in relation to the node that we began processing via `processBranch`. We assume that this is being invoked before the strong branch has been split at the bottleneck arc.

48c ⟨*Simplex public declarations* 37b⟩+≡ (37a) ◁46b 50b▷

```
NodePtr findNewRoot(Node& strongRoot, Node& strongChild);
```

Uses `findNewRoot` 48d and `Node` 164b.

48d ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁47a 51a▷

```
NodePtr SimplexSolver::findNewRoot(Node& strongRoot, Node& strongChild)
{
    ⟨find new root 49a⟩
}
```

Defines:
`findNewRoot`, used in chunks 39b and 48c.
Uses `Node` 164b and `SimplexSolver` 37a.

If our search started at the strong root, we know we were processing a non-degenerate branch and that the strong root is where to continue the search.

49a ⟨*find new root* 49a⟩≡ (48d) 49b▷

```
if (lastRoot == &strongRoot) {
    return &strongRoot;
}
```

Uses `lastRoot` 27b.

If we started our search at the strong child, the we want to contine the search at the parent (if its degenerate) or the root of the branch (if its not). However, both of these nodes should already be in a bucket from `queueDegeneratePath` or plain old `addStrongBranch` for the root of a degenerate branch. In this case, we return `nil` to signal that no branch needs to be queued.

49b ⟨*find new root* 49a⟩+≡ (48d) ◁49a 50a▷

```
if (lastRoot == &strongChild) {
    return nil;
}
```

Uses `lastRoot` 27b.

With those easy cases out of the way, we need to scan up from the the last root searching for the bottleneck arc (actually the parent node above the arc). If we don't find the arc, then we'll continue from the last root. Otherwise, it depends if the parent node is degenerate or not.

50a ⟨*find new root* 49a⟩+≡ (48d) ◁49b

```
NodePtr result = lastRoot;
NodePtr parent = strongChild.getParentNode();
NodePtr currNode = lastRoot->getParentNode();
while (currNode != nil) {
    if (currNode == parent) {
        if (currNode->flagIsSet(DEGENERATE_SUBTREE)) {
            if (branchInBucket(*currNode)) {
                result = nil;
            } else {
                result = currNode;
                TRACE( trout << "Found degenerate branch root that "
                                "is not already in bucket: "
                             << result->getId() << endl; );
            }
            break;
        } else {
            result = &strongRoot;
            break;
        }
    }
    currNode = currNode->getParentNode();
}

return result;
```

Uses `branchInBucket` 25a, `DEGENERATE_SUBTREE` 45b, `flagIsSet` 155d, `getParentNode` 141a, and `lastRoot` 27b.

Since it seems that we are implementing special versions of just about every method that touches branches, let's add a new version of `addStrongBranch`. This is mostly to verify the conditions that we expect to hold for the Simlex version - basically keeping an eye on degenerate branches.

50b ⟨*Simplex public declarations* 37b⟩+≡ (37a) ◁48c

```
virtual void addStrongBranch(Node& root);
```

Uses `addStrongBranch` 24c 51a and `Node` 164b.

51a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁48d 53a▷

```
void SimplexSolver::addStrongBranch(Node& root)
{
    NodeExcess excess = root.getExcess();
    if (excess > 0) {
        assert(root.flagIsSet(DEGENERATE_SUBTREE) == false);
        PhaseSolver::addStrongBranch(root);
    } else {
        assert(excess == 0);
        if (root.getParentNode() != nil) {
            assert(root.flagIsSet(DEGENERATE_SUBTREE));
            ⟨re-insert a degenerate branch 51b⟩
        } else {
            TRACE( trout << "Adding strong branch with zero excess: "
                         << root.getId() << endl; );
            PhaseSolver::addStrongBranch(root);
        }
    }
}
```

Defines:
`addStrongBranch`, used in chunks 7a, 16a, 18, 19a, 24b, 30a, 39–41, 47a, 50b, 60a, 75, 83b, 92, and 118a.
Uses `DEGENERATE_SUBTREE` 45b, `excess` 151b, `flagIsSet` 155d, `getExcess` 151d, `getParentNode` 141a, `Node` 164b, `PhaseSolver` 136c, and `SimplexSolver` 37a.

We need to be careful how we queue degenerate branches here. They are inserted in FIFO order for the first time in `addDegenerateBranch`. However, if this function is called to queue one, it must be from the Simplex `merge` function, so we must be requeing a branch. Therefore, we want to add it to the head of the list (i.e. LIFO) so that it can be discharged before its parent, if the parent has the same label.

51b ⟨*re-insert a degenerate branch* 51b⟩≡ (51a)

```
assert(root.getNext() == nil);
addBranchLifo(root);
checkLowestLabel(root.getLabel());
```

Uses `addBranchLifo` 26c, `checkLowestLabel` 25c, `getLabel` 138g, and `getNext` 150g.

2.6.3 Utility functions

The code above uses a method to find the bottleneck arc within the a branch. The bottleneck capacity is the downwards capacity in the strong branch and the upwards capacity in the weak branch. This function returns the root of the branch(like `Node.rehang`), the bottleneck capacity, and the node on the 'bottom' of the bottleneck arc. The bottleneck arc is the parent arc of the child node that is returned. If the node is a root (i.e. has no parent), the bottleneck capacity will be `MAXINT` and the bottleneck child will be `nil`. If there are multiple arcs along the path with the same bottleneck capacity, we will return the one closest to the root of a strong branch or the further one in a weak branch.

52 ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁47b 53b▷

```
Node& findBottleneckArc(Node& node,
                        bool isStrong,
                        FlowAmount& bottleneckCap,
                        NodePtr& bottleneckChild);
```

Uses `findBottleneckArc` 53a, `isStrong` 162a, and `Node` 164b.

53a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁51a 54a▷

```
Node& SimplexSolver::findBottleneckArc(Node& node,
                                       bool isStrong,
                                       FlowAmount& bottleneckCap,
                                       NodePtr& bottleneckChild)
{
    NodePtr currNode = &node;
    EdgePtr parentEdge = node.getParentEdge();
    bottleneckChild = nil;
    bottleneckCap = MAXINT;

    while (parentEdge != nil) {
        NodePtr parentNode = parentEdge->getOtherNode(currNode);
        if (isStrong) {
            FlowAmount res = parentEdge->residCapacity(*parentNode);
            if (res <= bottleneckCap) {
                bottleneckCap = res;
                bottleneckChild = currNode;
            }
        } else {
            FlowAmount res = parentEdge->residCapacity(*currNode);
            if (res < bottleneckCap) {
                bottleneckCap = res;
                bottleneckChild = currNode;
            }
        }

        currNode = parentNode;
        parentEdge = currNode->getParentEdge();
    }

    return (currNode == &node) ? node : *currNode;
}
```

Defines:
`findBottleneckArc`, used in chunks 38a and 52.
Uses `getOtherNode` 167a, `getParentEdge` 141a, `isStrong` 162a, `Node` 164b, `parentEdge` 140c, `parentNode` 140c, `residCapacity` 169b, and `SimplexSolver` 37a.

In the strong branch, we need a method to pull some of the excess from the root down to a node. This requires that there be sufficient capacity in the reverse direction in each arc along the path. The flow on each arc will be adjusted to reflect the flow that we are pulling down.

53b ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁52 54c▷

```
void pullFromRoot(Node& node, FlowAmount amount);
```

Uses `Node` 164b and `pullFromRoot` 54a.

54a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁53a 55a▷

```
void SimplexSolver::pullFromRoot(Node& node, FlowAmount amount)
{
    NodePtr currNode = &node;
    EdgePtr parentEdge = node.getParentEdge();

    if (amount == 0) {
        TRACE( trout << "Pull zero flow from root to " << node.getId() << endl; );
        return;
    }
    if (parentEdge == nil) {
        // node is a root - the excess is already here: nothing to do
        return;
    }

    while (parentEdge != nil) {
        Node& parentNode = *parentEdge->getOtherNode(currNode);
        assert(parentEdge->residCapacity(parentNode) >= amount);
        STATS(numPushToParent++);
        parentEdge->decreaseFlow(amount);
        currNode = &parentNode;
        parentEdge = parentNode.getParentEdge();
    }
    ⟨update excesses at endpoints 54b⟩
}
```

Defines:
`pullFromRoot`, used in chunks 39, 40, and 53b.
Uses `decreaseFlow` 168c, `excess` 151b, `flow` 167d, `getOtherNode` 167a, `getParentEdge` 141a, `Node` 164b, `numPushToParent` 129b, `parentEdge` 140c, `parentNode` 140c, `residCapacity` 169b, and `SimplexSolver` 37a.

The excess of the root node (pointed to by `currNode` will decrease, and the excess of the node that we are starting from will increase.

54b ⟨*update excesses at endpoints* 54b⟩≡ (54a)

```
assert(currNode->getExcess() >= amount);
currNode->decrementExcess(amount);
node.incrementExcess(amount);
```

Uses `decrementExcess` 151d, `getExcess` 151d, and `incrementExcess` 151d.

Here is a debug function that validates the queues of degenerate branches.

54c ⟨*Simplex private declarations* 44a⟩+≡ (37a) ◁53b

```
void validateDegenerateBranches();
```

Uses `validateDegenerateBranches` 55a.

55a ⟨*Simplex implementations* 37c⟩+≡ (135c) ◁54a

```
void SimplexSolver::validateDegenerateBranches()
{
    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        if (node.flagIsSet(DEGENERATE_SUBTREE)) {
            assert(branchInBucket(node));
            // look for same-label parent in same bucket
            NodePtr parent = node.getParentNode();
            assert(parent != nil);
            if ((parent->getLabel() == node.getLabel()) &&
                (parent->flagIsSet(DEGENERATE_SUBTREE)))
            {
                NodePtr currNode = node.getNext();
                while (currNode != parent) {
                    assert(currNode != nil);
                    currNode = currNode->getNext();
                }
            }
        }
    }
}
```

Defines:
`validateDegenerateBranches`, used in chunks 47a and 54c.
Uses `branchInBucket` 25a, `DEGENERATE_SUBTREE` 45b, `flagIsSet` 155d, `getLabel` 138g, `getNext` 150g, `getParentNode` 141a, `Node` 164b, `nodes` 3a, `numNodes` 3d, and `SimplexSolver` 37a.

2.7 Establishing an Initial Normalized Tree

There are a number of different ways that we can build an initial, normalized tree. Building the initial tree is totally independent of the execution of the algorithm.

2.7.1 Simple Initialization

At the very least, this simplest initialization needs to saturate all source- and sink.adjacent edges and source-adjacent nodes 2, and label the others 1. classify the nodes as strong or weak.

55b ⟨*Solver methods* 4a⟩+≡ (136c) ◁35b 56b▷

```
void buildSimpleTree(LabelMethod labelMethod);
```

Uses `buildSimpleTree` 56a.

56a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁36a 56c▷
```
void PhaseSolver::buildSimpleTree(LabelMethod labelMethod)
{
    saturateSourceSinkArcs();
    setLabelsAndStatus(labelMethod);
}
```
Defines:
`buildSimpleTree`, used in chunks 55b, 198a, and 199b.
Uses `PhaseSolver` 136c, `saturateSourceSinkArcs` 56c, and `setLabelsAndStatus` 59a.

The code to saturate the source- and sink-adjacent arcs is broken out into a separate function because many schemes for building the initial tree will want start by saturating these arcs. When we saturate an edge into the sink or out of the source, we remove it from the list of arcs for the node. This will prevent flow from going back to either the source or the sink without having to special-case these arcs at run-time. Also, we want to deal with arcs going into the source or out of the sink because these will never be used in an optimal solution.

56b ⟨*Solver methods* 4a⟩+≡ (136c) ◁55b 58c▷
```
    void saturateSourceSinkArcs();
```
Uses `saturateSourceSinkArcs` 56c.

56c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁56a 59a▷
```
void PhaseSolver::saturateSourceSinkArcs()
{
    for(sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Edge& edge = sourceNode->getCurrentNeighbor();
        ⟨saturate or remove current edge 57⟩
    }

    for(sinkNode->resetIterations();
        sinkNode->hasMoreNeighbors();
        sinkNode->advanceNeighbors())
    {
        Edge& edge = sinkNode->getCurrentNeighbor();
        ⟨saturate or remove current edge 57⟩
    }
}
```
Defines:
`saturateSourceSinkArcs`, used in chunks 56, 64a, 67a, 71b, and 72b.
Uses `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `hasMoreNeighbors` 144a, `PhaseSolver` 136c, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

If the arc goes straight from the source to the sink, we just saturated it (if we haven't already) and don't bother doing anything else.

57 ⟨*saturate or remove current edge* 57⟩≡ (56c) 58a▷

```
if ((edge.getSource() == sourceNode) &&
    (edge.getDest() == sinkNode))
{
    if (edge.residCapacity() > 0) {
        // we see these arcs twice: source and sink - only sat once
        edge.saturate();
    }
    continue;
}
```

Uses `getDest` 165d, `getSource` 165d, `residCapacity` 169b, `saturate` 171d, `sinkNode` 3f, `source` 165a, and `sourceNode` 3f.

When we saturate an edge, we remove it from the list of arcs for the source- or sink-adjacent node. This will prevent flow from going back to either the source or the sink without having to special-case these arcs at run-time. However, before performing flow recovery (in `convertToFlow`) we will need to add them back to return excess/deficit to the source/sink. Also, we want to remove the arcs going into the source or out of the sink because these will never be used in an optimal solution, but these do not have to be added back.

58a ⟨*saturate or remove current edge* 57⟩+≡ (56c) ◁57

```
if (edge.getSource() == sourceNode) {
    FlowAmount excess = edge.saturate();
    Node& srcAdjNode = *edge.getDest();
    srcAdjNode.removeNeighbor(edge);
    srcAdjNode.incrementExcess(excess);
    srcAdjNode.setFlag(SOURCE_ADJ);
}  else if (edge.getDest() == sourceNode) {
    // edge into source never used
    edge.getSource()->removeNeighbor(edge);
}
if (edge.getDest() == sinkNode) {
    FlowAmount excess = edge.saturate();
    Node& sinkAdjNode = *edge.getSource();
    sinkAdjNode.decrementExcess(excess);
    sinkAdjNode.removeNeighbor(edge);
    sinkAdjNode.setFlag(SINK_ADJ);
} else if (edge.getSource() == sinkNode) {
    // edge out of sink never used
    edge.getDest()->removeNeighbor(edge);
}
```

Uses `decrementExcess` 151d, `excess` 151b, `getDest` 165d, `getSource` 165d, `incrementExcess` 151d, `Node` 164b, `removeNeighbor` 139f, `saturate` 171d, `setFlag` 155d, `sinkNode` 3f, `source` 165a, and `sourceNode` 3f.

58b ⟨*Solver protected members* 3a⟩+≡ (136c) ◁27c 112b▷

```
static const int SOURCE_ADJ = 0x100;
static const int SINK_ADJ = 0x200;
static const int ADJACENT = SOURCE_ADJ | SINK_ADJ;
```

After we have built an initial tree structure, we need to assign labels to the nodes, and we need to add strong nodes to buckets based on their tables. We currently support three different labeling schemes. At this point in the execution, we assume all label counts are zero.

58c ⟨*Solver methods* 4a⟩+≡ (136c) ◁56b 59b▷

```
void setLabelsAndStatus(LabelMethod labelMethod);
```

Uses `setLabelsAndStatus` 59a.

59a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁56c 60a▷

```
void PhaseSolver::setLabelsAndStatus(LabelMethod labelMethod)
{
    switch(labelMethod)
    {
        case LABELS_CONSTANT:
            setConstantLabels();
            break;
        case LABELS_SINK_DIST:
            setSinkDistLabels();
            break;
        case LABELS_DEFICIT_DIST:
            setDeficitDistLabels();
            break;
        default:
            assert("invalid labeling method" == nil);
    }
    setInitialLabelCounts();
}
```

Defines:
`setLabelsAndStatus`, used in chunks 56a, 58c, 64a, 67a, 71b, and 116a.
Uses `PhaseSolver` 136c, `setConstantLabels` 60a, `setDeficitDistLabels` 89c, `setInitialLabelCounts` 62, and `setSinkDistLabels` 87c.

This simplest scheme is to give weak nodes label one and strong nodes label two.[1] We iterate over the root nodes. We label all nodes below a root with the label based on the excess of the root. Anything with positive excess is assumed to be strong, and everything else is assumed to be weak. Strong nodes are added as strong branches.

59b ⟨*Solver methods* 4a⟩+≡ (136c) ◁58c 60c▷

```
    void setConstantLabels();
```

Uses `setConstantLabels` 60a.

[1] We could give strong nodes label one, but they'd have no one to merge with until they are relabled to two.

60a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁59a 61a▷

```
void PhaseSolver::setConstantLabels()
{
    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        if (node.isRootNode()) {
            NodeLabel label = INIT_ZERO_LABEL;
            if (node.getExcess() > 0) {
                label = INIT_STRONG_LABEL;
            } else if (node.getExcess() < 0) {
                label = INIT_WEAK_LABEL;
            }
            labelBranch(node, label);

            if (node.getExcess() > 0) {
                addStrongBranch(node);
            }
        }
        node.resetIterations();
    }
}
```

Defines:
  `setConstantLabels`, used in chunks 59 and 75.
Uses `addStrongBranch` 24c 51a, `getExcess` 151d, `INIT_STRONG_LABEL` 60b, `INIT_WEAK_LABEL` 60b, `INIT_ZERO_LABEL` 60b, `isRootNode` 141a, `labelBranch` 61a, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, and `resetIterations` 144c.

60b ⟨*Solver private members* 15b⟩+≡ (136c) ◁23e

```
static const int INIT_STRONG_LABEL=2;
static const int INIT_WEAK_LABEL=1;
static const int INIT_ZERO_LABEL=1;
```

Defines:
  `INIT_STRONG_LABEL`, used in chunks 60a, 65, 68, and 76b.
  `INIT_WEAK_LABEL`, used in chunks 6a, 60a, 62, 63b, 75, 79b, 88a, 90a, and 92.
  `INIT_ZERO_LABEL`, used in chunk 60a.

This function sets the label of a node and all of its children. It also updates the count of how many nodes have each label value.

60c ⟨*Solver methods* 4a⟩+≡ (136c) ◁59b 61b▷

```
void labelBranch(Node& node, NodeLabel label);
```

Uses `labelBranch` 61a and `Node` 164b.

61a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁60a 62▷

```
void PhaseSolver::labelBranch(Node& node, NodeLabel label)
{
    node.setLabel(label);
    node.distance = node.minChildLabel = label;

    node.resetIterations();
    while (node.hasMoreChildren()) {
        Node& child = node.getCurrentChild();
        labelBranch(child, label);
        node.advanceChildren();
    }
    node.resetIterations();
}
```

Defines:
`labelBranch`, used in chunk 60.
Uses `advanceChildren` 143b, `distance` 156c, `getCurrentChild` 143b, `hasMoreChildren` 143b, `minChildLabel` 156c, `Node` 164b, `PhaseSolver` 136c, `resetIterations` 144c, and `setLabel` 13a 138g.

about keeping track of the label counts. This is especially hard if we re-use code that was intended for use while solving when the labels are correct vs. initialization when the labels are not yet set. Therefore, just to make sure everything is perfectly correct, we'll reset the label counts and then scan the nodes to count them correctly. Note that shortest path initialization (`buildSpTree`) does *not* use this method.

61b ⟨*Solver methods* 4a⟩+≡ (136c) ◁60c 63c▷

```
void setInitialLabelCounts();
```

Uses `setInitialLabelCounts` 62.

62 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁61a 64a▷

```
void PhaseSolver::setInitialLabelCounts()
{
    for (int i = 0; i <= numNodes; i++) {
        labelCount[i] = 0;
    }

    maxWeakLabel = maxInitialLabel = INIT_WEAK_LABEL;
    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        if ((&node == sourceNode) || (&node == sinkNode)) {
            continue;
        }

        NodeLabel l = node.getLabel();

        if (l < numNodes) {
            labelCount[l]++;
            if (l > maxInitialLabel) {
                maxInitialLabel = l;
            }
            if ((node.getExcess() < 0) && (l > maxWeakLabel)) {
                maxWeakLabel = l;
            }
        } else {
            labelCount[numNodes]++;
        }
    }
#ifdef DEBUG
    int total = 0;
    for (int i = 0; i <= numNodes; i++) {
        total += labelCount[i];
    }
    assert(total == (numNodes - 2));
#endif DEBUG
}
```

Defines:
`setInitialLabelCounts`, used in chunks 59a, 61b, and 75.
Uses `getExcess` 151d, `getLabel` 138g, `INIT_WEAK_LABEL` 60b, `labelCount` 5c, `maxInitialLabel` 63b, `maxWeakLabel` 63b, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, `sinkNode` 3f, and `sourceNode` 3f.

Above we computed a couple of maximum label values that are used by the solver functions to control early termination. The `maxInitialLabel` is pretty obvious. The `maxWeakLabel` is a bit of a misnomer. We only looked at nodes with negative excess. If we have an initialization scheme that generates weak branches, and gives them labels based on their depth, this value will be wrong. However, calling `isStrongNode` on every node would be prohibitive, $O(n^2)$.

63a ⟨*Solver data* 3e⟩+≡ (136c) ◁31b 77b▷

```
int maxWeakLabel;
int maxInitialLabel;
```

Uses `maxInitialLabel` 63b and `maxWeakLabel` 63b.

63b ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁31c 67c▷

```
maxWeakLabel = INIT_WEAK_LABEL;
maxInitialLabel = INIT_WEAK_LABEL;
```

Defines:
`maxInitialLabel`, used in chunks 5a, 62, 63a, 72b, 75, and 92.
`maxWeakLabel`, used in chunks 35a, 62, 63a, and 72b.
Uses `INIT_WEAK_LABEL` 60b.

2.7.2 Blocking Path Initialization

A simple extension to the idea of just simply saturating the source- and sink-adjacent arcs is to try to push flow from the source-adjacent nodes into the interior of the graph. Our objective is to try to build some simple but large components with a minimum of work.

This is a bit more involved than it initially appeared. The first thing to note is that the components we will build will be paths. This means that we can only push flow out of a node along a single outgoing arc. Also, we can only push flow into a node from a single neighbor. Otherwise, we might end up building components that are not trees.

Because we wish to build components that are as large as possible (rather than creating lots of small components), we want to avoid splitting edges. However, this can be problematic because many of the random graphs we test with have more in-capacity than out-capacity for the source-adjacent nodes. If we blindly avoid splitting, we won't move any flow from the source-adjacent nodes because they have more excess than out-capacity.

With all of this in mind, we proceed by finding nodes with positive excess that still have their initial label. We then push the flow out of the node and expect that every node we visit while pushing flow will be relabeled. Basically, we are using the node's label as a marker indicating that we have visited the node.

63c ⟨*Solver methods* 4a⟩+≡ (136c) ◁61b 64b▷

```
void buildBlockingPathTree(LabelMethod labelMethod);
```

64a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁62 65▷

```
void PhaseSolver::buildBlockingPathTree(LabelMethod labelMethod)
{
    saturateSourceSinkArcs();

    for (sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Node& node = *sourceNode->getCurrentNeighbor().getOtherNode(sourceNode);
        if ((node.getExcess() > 0) &&
            (node.getLabel() == Node::INITIAL_LABEL))
        {
            blockingPathPush(node);
        }
    }

    setLabelsAndStatus(labelMethod);
}
```

Defines:
`bulidBlockingPathTree`, never used.
Uses `advanceNeighbors` 144a, `blockingPathPush` 65 68, `getCurrentNeighbor` 144a, `getExcess` 151d, `getLabel` 138g, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `Node` 164b, `PhaseSolver` 136c, `resetIterations` 144c, `saturateSourceSinkArcs` 56c, `setLabelsAndStatus` 59a, and `sourceNode` 3f.

The real work in this greedy scheme is pushing the flow out of a node. This recursive function finds the 'best' edge to push the flow out on, and then we push the flow. When we push flow out, we make this node a child of the node at the other end of the arc, and then we try to push flow out of that node (hence the recursion). As we visit each node, we relabel it to prevent visiting the same node more than once.

If we reach a point where the flow cannot be pushed out of a node, that node then becomes the root of a strong branch. If we make it all the way to a sink-adjacent node, and its deficit exceeds the excess we are pushing, then it becomes a weak branch.

64b ⟨*Solver methods* 4a⟩+≡ (136c) ◁63c 66c▷

```
void blockingPathPush(Node& node);
```

Uses `blockingPathPush` 65 68 and `Node` 164b.

65 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁64a 67a▷

```
void PhaseSolver::blockingPathPush(Node& node)
{
    node.setLabel(INIT_STRONG_LABEL);

    NodeExcess excess = node.getExcess();
    Edge* bestOutEdge = nil;
    ⟨find best out-edge without split 66a⟩

    FlowAmount flow = excess;
    if (bestOutEdge != nil) {
        Node& otherNode = *bestOutEdge->getOtherNode(&node);
        TRACE( trout << "push positive excess (" << flow << " from "
                     << node.getId() << " to " << otherNode.getId() << endl);

        bestOutEdge->increaseFlow(flow);
        node.decrementExcess(flow);
        otherNode.incrementExcess(flow);
        otherNode.addChild(node, *bestOutEdge);

        ⟨recursively push flow out 66b⟩
    } else {
        TRACE( trout << "can't push excess from " << node.getId() << endl; )
    }
}
```

Defines:
`blockingPathPush`, used in chunks 64 and 66b.
Uses `addChild` 142a, `decrementExcess` 151d, `Edge` 179a, `excess` 151b, `flow` 167d, `getExcess` 151d, `getOtherNode` 167a, `increaseFlow` 168a, `incrementExcess` 151d, `INIT_STRONG_LABEL` 60b, `Node` 164b, `PhaseSolver` 136c, and `setLabel` 13a 138g.

To find the best out edge, we iterate over all of the edges looking for one that has at least enough capacity to handle the excess. Also, the edge must refer to a node that we haven't visited yet, as indicated by its label. If we find multiple edges with sufficient capacity, we take the one with the least capacity. Alternatively, we could take the first fit, but we do not expect that scanning every edge will be too burdensome.

66a ⟨*find best out-edge without split* 66a⟩≡ (65)

```
node.resetIterations();
while (node.hasMoreNeighbors()) {
    Edge& edge = node.getCurrentNeighbor();
    if (edge.getOtherNode(&node)->getLabel() == Node::INITIAL_LABEL)
    {
        if (edge.residCapacity(node) >= excess) {
            if (bestOutEdge == nil) {
                bestOutEdge = &edge;
            } else if (bestOutEdge->residCapacity(node) > edge.residCapacity(node)) {
                bestOutEdge = &edge;
            }
        }
    }
    node.advanceNeighbors();
}
node.resetIterations();
```

Uses `advanceNeighbors` 144a, `Edge` 179a, `excess` 151b, `getCurrentNeighbor` 144a, `getLabel` 138g, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `Node` 164b, `resetIterations` 144c, and `residCapacity` 169b.

Pushing flow out is very straightforward: just call `blockingPathPush(otherNode)`. In the past it was more complicated because we were setting the label while building the tree, and if we reached the sink, then we had created a weak branch instead of a strong one. However, now we have separated the two steps and we simply build the tree, pushing the excess as far as we can. Then we look at the result and apply labels as needed.

66b ⟨*recursively push flow out* 66b⟩≡ (65)

```
if (otherNode.getExcess() > 0) {    // ZERO_DEFICIT
    blockingPathPush(otherNode);
}
```

Uses `blockingPathPush` 65 68 and `getExcess` 151d.

2.7.3 Splitting Path Initialization

Greedy push with an arbitrary number of splits allowed.

66c ⟨*Solver methods* 4a⟩+≡ (136c) ◁64b 67d▷

```
void buildGreedyPathTree(LabelMethod labelMethod);
```

Uses `buildGreedyPathTree` 67a.

67a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁65 68▷

```
void PhaseSolver::buildGreedyPathTree(LabelMethod labelMethod)
{
    saturateSourceSinkArcs();

    for (sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Node& node = *sourceNode->getCurrentNeighbor().getOtherNode(sourceNode);
        if ((node.getExcess() > 0) &&
            (node.getLabel() == Node::INITIAL_LABEL))
        {
            int numSplits = maxSplits;
            splittingPathPush(node, numSplits);
        }
    }

    setLabelsAndStatus(labelMethod);
}
```

Defines:
`buildGreedyPathTree`, used in chunks 66c and 199b.
Uses `advanceNeighbors` 144a, `getCurrentNeighbor` 144a, `getExcess` 151d, `getLabel` 138g, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `maxSplits` 67b, `Node` 164b, `numSplits` 129b, `PhaseSolver` 136c, `resetIterations` 144c, `saturateSourceSinkArcs` 56c, `setLabelsAndStatus` 59a, and `sourceNode` 3f.

67b ⟨*public Solver data* 8b⟩+≡ (136c) ◁19c 77c▷

```
int maxSplits;
```

Defines:
`maxSplits`, used in chunks 67, 195, 202, and 208a.

The default maximum number of splits is arbitrarily set to ten. It's public so it can be easily changed by the driver program.

67c ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁63b 77d▷

```
maxSplits = 10;
```

Uses `maxSplits` 67b.

67d ⟨*Solver methods* 4a⟩+≡ (136c) ◁66c 71a▷

```
void splittingPathPush(Node& node, int& remainingSplits);
```

Uses `Node` 164b.

68 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁67a 71b▷

```
void PhaseSolver::splittingPathPush(Node& node, int& remainingSplits)
{
    node.setLabel(INIT_STRONG_LABEL);
    NodeExcess excess = node.getExcess();
    EdgePtr bestOutEdge = nil;
    FlowAmount bestEdgeCap = MAXINT;
    ⟨find best out-edge with split 69a⟩

    if (bestOutEdge != nil) {
        Node& otherNode = *bestOutEdge->getOtherNode(&node);
        if ((bestEdgeCap < excess) && (remainingSplits <= 0)) {
            return;
        }

        TRACE( trout << "push positive excess ("
                     << ((excess < bestEdgeCap) ? excess : bestEdgeCap)
                     << ") from " << node.getId() << " to "
                     << otherNode.getId() << endl);

        otherNode.addChild(node, *bestOutEdge);
        bool split = node.pushToParent();
        if (split) {
            remainingSplits--;
        }

        ⟨recursively push flow out 2 70c⟩

    } else {
        TRACE( trout << "can't push excess from " << node.getId() << endl; )
    }
}
```

Defines:
`blockingPathPush`, used in chunks 64 and 66b.
Uses `addChild` 142a, `excess` 151b, `getExcess` 151d, `getOtherNode` 167a, `INIT_STRONG_LABEL` 60b, `Node` 164b, `PhaseSolver` 136c, `pushToParent` 152b, `setLabel` 13a 138g, and `split` 145c.

To find the best edge, we iterate over all of the neighbors. We only consider nodes that haven't already been visited and had flow pushed through them (as indicated by the label). To be feasible and edge must have positive residual capacity. Also, if we cannot split any more, we need it to have capacity greater than the excess.

69a ⟨*find best out-edge with split* 69a⟩≡ (68)

```
for (node.resetIterations();
     node.hasMoreNeighbors();
     node.advanceNeighbors())
{
    Edge& edge = node.getCurrentNeighbor();
    Node& otherNode =  *edge.getOtherNode(&node);
    if (otherNode.getLabel() == Node::INITIAL_LABEL) {
        FlowAmount edgeCap =  edge.residCapacity(node);
        if ((edgeCap <= 0) ||
            ((edgeCap <= excess) && (remainingSplits <= 0)))
        {
            continue;
        }

        ⟨compare edge to best edge 69b⟩
    }
}
```

Uses `advanceNeighbors` 144a, `Edge` 179a, `excess` 151b, `getCurrentNeighbor` 144a, `getLabel` 138g, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `Node` 164b, `resetIterations` 144c, and `residCapacity` 169b.

If this is the first feasible edge we've seen, it must be the best.

69b ⟨*compare edge to best edge* 69b⟩≡ (69a) 70a▷

```
if (bestOutEdge == nil) {
    bestOutEdge = &edge;
    bestEdgeCap = edgeCap;
}
```

We're looking an edge with capacity greater than the excess to avoid splits. If there are multiple such edges, we'll choose the minimal one - i.e. the edge with capacity closest to the excess.

70a ⟨*compare edge to best edge* 69b⟩+≡ (69a) ◁69b 70b▷

```
else if (edgeCap > excess) {
    if (bestEdgeCap > edgeCap) {
        bestOutEdge = &edge;
        bestEdgeCap = edgeCap;
    } else if (bestEdgeCap < excess) {
        bestOutEdge = &edge;
        bestEdgeCap = edgeCap;
    }
}
```

Uses excess 151b.

If we can't find an edge with capacity that exceeds the excess, we want to find one that the most capacity. Again, looking for capacity closest to the excess.

70b ⟨*compare edge to best edge* 69b⟩+≡ (69a) ◁70a

```
else if (edgeCap > bestEdgeCap) {
    bestOutEdge = &edge;
    bestEdgeCap = edgeCap;
}
```

When we go to push flow out of `otherNode`, we want to keep our eyes open for reaching all the way to a sink-adjacent node. If we pushed flow to `otherNode` and its excess isn't positive, it must be sink adjacent, so we can stop pushing.

70c ⟨*recursively push flow out 2* 70c⟩≡ (68) 70d▷

```
if (otherNode.getExcess() > 0) {
    splittingPathPush(otherNode, remainingSplits);
} else {
    TRACE( trout << "found sink-adjacent node " << otherNode.getId() << endl;)
    CHECK_TREE(otherNode, nil);
}
```

Uses getExcess 151d.

If we pushed flow out of this node, and it still has positive excess, we must have split. We just recursively call ourselves to try to push again. This will repeat the search of neighbors, but we will be looking to push less excess this time, so the set of feasible arcs will different.

70d ⟨*recursively push flow out 2* 70c⟩+≡ (68) ◁70c

```
if (node.getExcess() > 0) {
    splittingPathPush(node, remainingSplits);
}
```

Uses getExcess 151d.

2.7.4 Staturate All Arcs

Another simple initialization scheme is to saturate all arcs in the graph. Then we classify each node as strong or weak depending on whether it has positive or negative excess.

71a ⟨*Solver methods* 4a⟩+≡ (136c) ◁67d 72a▷

```
void saturateAllArcs(LabelMethod labelMethod);
```

Uses `saturateAllArcs` 71b.

71b ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁68 72b▷

```
void PhaseSolver::saturateAllArcs(LabelMethod labelMethod)
{
    saturateSourceSinkArcs();

    for (int i = 0 ; i < numEdges; i++) {
        Edge& edge = edges[i];
        if ((edge.getDest() != sourceNode) &&
            (edge.getSource() != sinkNode) &&
            (edge.getFlow() == 0))
        {
            FlowAmount excess = edge.saturate();

            edge.getTail()->decrementExcess(excess);
            edge.getHead()->incrementExcess(excess);
        }
    }

    setLabelsAndStatus(labelMethod);
}
```

Defines:
`saturateAllArcs`, used in chunks 71a and 199b.

Uses `decrementExcess` 151d, `Edge` 179a, `edges` 3a, `excess` 151b, `getDest` 165d, `getFlow` 167d, `getHead` 22a 166d, `getSource` 165d, `getTail` 22a 166d, `incrementExcess` 151d, `numEdges` 3d, `PhaseSolver` 136c, `saturate` 171d, `saturateSourceSinkArcs` 56c, `setLabelsAndStatus` 59a, `sinkNode` 3f, and `sourceNode` 3f.

2.8 Distance Labeling

Taking a page from implementations of the push-relabel algorithm, we have code to establish and maintain our labels based on distances to the sink. We can do this at initialization time and/or periodically during the execution.

2.8.1 Shortest Path Initialization

To use distance labels during initialization, we start with the usual staturated source and sink arcs. Then, we perform a shortest-path search back from the sink linking all the zero-deficit nodes into weak branches along the shortest paths, via a breadth-first search. We use the node label as a distance label during this procedure. When we are done, `setSpInitialNodeStatus` will set the labels of the strong nodes based on the labels of their weak neighbors.

72a ⟨*Solver methods* 4a⟩+≡ (136c) ◁71a 74b▷

```
void buildSpTree(LabelMethod labelMethod);
```

Uses `buildSpTree` 72b.

72b ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁71b 75▷

```
void PhaseSolver::buildSpTree(LabelMethod labelMethod)
{
    saturateSourceSinkArcs();
    ⟨initialize all node labels 73a⟩
    resetQ();

    for(sinkNode->resetIterations();
        sinkNode->hasMoreNeighbors();
        sinkNode->advanceNeighbors())
    {
        ⟨load queue with weak roots 73b⟩
    }

    ⟨perform bfs labeling 73d⟩
    setSpInitialNodeStatus(labelMethod);
    maxWeakLabel = maxInitialLabel;             // needed for highest label

    IFDEBUG( checkBranches(); );
}
```

Defines:
`buildSpTree`, used in chunks 72a and 199b.
Uses `advanceNeighbors` 144a, `checkBranches` 133a, `hasMoreNeighbors` 144a, `maxInitialLabel` 63b, `maxWeakLabel` 63b, `PhaseSolver` 136c, `resetIterations` 144c, `resetQ` 93c, `saturateSourceSinkArcs` 56c, `setSpInitialNodeStatus` 75, and `sinkNode` 3f.

First, we need to set all the labels to infinity.

73a ⟨*initialize all node labels* 73a⟩≡ (72b)

```
for (int i = 1; i <= numNodes; i++) {
    nodes[i].setLabel(MAXINT);
}
```

Uses `nodes` 3a, `numNodes` 3d, and `setLabel` 13a 138g.

Then, we iterate over each sink-adjacent node, ignoring the source node if it's adjacent to the sink.

73b ⟨*load queue with weak roots* 73b⟩≡ (72b) 73c▷

```
Edge& edge = sinkNode->getCurrentNeighbor();
Node& edgeSource = *edge.getSource();
Node& edgeDest = *edge.getDest();
if ((&edgeSource == sourceNode) || (&edgeDest != sinkNode) ||
     (edgeSource.getExcess() >= 0))
{
    continue;
}
```

Uses `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getExcess` 151d, `getSource` 165d, `Node` 164b, `sinkNode` 3f, and `sourceNode` 3f.

Once we're sure we a sink-adjacent node and it's not the source, we label it, and start investigating its neighbors.

73c ⟨*load queue with weak roots* 73b⟩+≡ (72b) ◁73b

```
edgeSource.setLabel(1);
putNodeQ(edgeSource);
```

Uses `putNodeQ` 94a and `setLabel` 13a 138g.

Once the queue is loaded with the sink-adjacent nodes, we just pull nodes out of the queue until there are no more.

73d ⟨*perform bfs labeling* 73d⟩≡ (72b)

```
for (NodePtr nodep = getNextNodeQ(); nodep != nil; nodep = getNextNodeQ()) {
    int dist = nodep->getLabel() + 1;
    ElIterator it = nodep->getNeighbors();
    for (EdgePtr edgep = it.getNext(); edgep != nil; edgep = it.getNext()) {
        ⟨label weak neighbor as needed 74a⟩
    }
}
```

Uses `getLabel` 138g, `getNeighbors` 145a, `getNext` 150g, and `getNextNodeQ` 93d.

If our neighbor has zero excess and we have residual capacity, then we might be his shortest path to the sink. If his label is greater than our distance, it should be MAXINT, and we are a shortest path for him. Then, we become his parent, and we want to investigate all of his neighbors.

If his label is our distance, then we are not any closer, but may we have more capacity. In this case, we don't need to queue him to investigate his neighbors because he should already be in the queue from the time he got labeled.

74a ⟨*label weak neighbor as needed* 74a⟩≡ (73d)

```
Node& neighbor = *edgep->getOtherNode(nodep);
FlowAmount residCap = edgep->residCapacity(neighbor);
if ((neighbor.getExcess() == 0) && (residCap > 0)) {
    if (neighbor.getLabel() > dist) {
        neighbor.setLabel(dist);
        neighbor.distance = dist;
        putNodeQ(neighbor);
        nodep->addChild(neighbor, *edgep);
    } else if ((neighbor.getLabel() == dist) &&
                (neighbor.getParentCapacity() < residCap))
    {
        neighbor.split();
        nodep->addChild(neighbor, *edgep);
    }
}
```

Uses addChild 142a, distance 156c, getExcess 151d, getLabel 138g, getOtherNode 167a, getParentCapacity 141a, Node 164b, putNodeQ 94a, residCapacity 169b, setLabel 13a 138g, and split 145c.

Once we have built the weak trees, we need to establish the initial status of all nodes. If we want to set constant labels (one and two), we need to reset the distances while we're at it.[2] Therefore, in addition to the usual function, setConstantLabels, we also reset the distances.

The other initial labeling schemes (distance to sink or deficit), are equivalent, and we've already done the real work - computing the distances. The weak nodes are already correctly labeled, we just need to set the label count appropriately. For the strong nodes, we need to find the minimum label of its weak neighbors, and we also need to add it as a strong branch.

74b ⟨*Solver methods* 4a⟩+≡ (136c) ◁72a 77a▷

```
void setSpInitialNodeStatus(LabelMethod labelMethod);
```

Uses setSpInitialNodeStatus 75.

[2] This is due to the check we currently have in processSubtree to check that the label is greater than or equal to the distance. After we experiment with integrated distance labels, we may be able to remove that check and this code.

75 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁72b 79a▷

```
void PhaseSolver::setSpInitialNodeStatus(LabelMethod labelMethod)
{
    switch(labelMethod) {
    case LABELS_CONSTANT:
        setConstantLabels();
        setInitialLabelCounts();
        for (int i = 1; i <= numNodes; i++) {
            Node& node = nodes[i];
            node.distance = node.getLabel();
        }
        break;
    default:
        maxInitialLabel = INIT_WEAK_LABEL;
        for (int i = 1; i <= numNodes; i++) {
            Node& node = nodes[i];
            if (node.getExcess() > 0) {
                ⟨set label to minimum of weak neighbors 76a⟩
                addStrongBranch(node);
            }  else if (node.getLabel() == MAXINT) {
                // notice with cher graphs that some zero-deficit nodes are
                // unreachable during initialization???
                node.setLabel(INIT_WEAK_LABEL);
            }
            labelCount[node.getLabel()]++;
            node.resetIterations();
            if (node.getLabel() > maxInitialLabel) {
                maxInitialLabel = node.getLabel();
            }
        }
    }
}
```

Defines:
`setSpInitialNodeStatus`, used in chunks 72b, 74b, and 87c.

Uses `addStrongBranch` 24c 51a, `distance` 156c, `getExcess` 151d, `getLabel` 138g, `INIT_WEAK_LABEL` 60b, `labelCount` 5c, `maxInitialLabel` 63b, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, `resetIterations` 144c, `setConstantLabels` 60a, `setInitialLabelCounts` 62, and `setLabel` 13a 138g.

For a strong node, we iterate over its neighbors looking for nodes with non-positive excess and find the minimum label. *We don't really care about in/out arcs, but source/sink node could still be a problem?*

76a ⟨*set label to minimum of weak neighbors* 76a⟩≡ (75) 76b▷

```
NodeLabel minLabel = MAXINT;
for (node.resetIterations();
     node.hasMoreNeighbors();
     node.advanceNeighbors())
{
    Edge& neighborEdge = node.getCurrentNeighbor();
    Node& neighbor = *neighborEdge.getOtherNode(&node);
    if (neighbor.getExcess() <= 0) {
        if (minLabel > neighbor.getLabel()) {
            minLabel = neighbor.getLabel();
        }
    }
}
```

Uses advanceNeighbors 144a, Edge 179a, getCurrentNeighbor 144a, getExcess 151d, getLabel 138g, getOtherNode 167a, hasMoreNeighbors 144a, Node 164b, and resetIterations 144c.

We then set the label to the minimum plus one. There is one catch that we noticed with Cheriyan graphs: all the neighbors could be unreachable from the sink, so they'd have MAXINT labels. In that case, we just use the standard initial label for strong nodes.

76b ⟨*set label to minimum of weak neighbors* 76a⟩+≡ (75) ◁76a

```
if (minLabel < MAXINT) {
    assert(minLabel < numNodes);
    node.setLabel(minLabel + 1);
    node.distance = minLabel + 1;
} else {
    node.setLabel(INIT_STRONG_LABEL);
}
```

Uses distance 156c, INIT_STRONG_LABEL 60b, numNodes 3d, and setLabel 13a 138g.

2.8.2 Global Relabeling Heuristic

During the execution of the algorithm, regardless of the initialization method used we can periodically relabel nodes based on their distances to the sink in the residual graph. Setting the labels this way 'guides' the excess from the strong nodes toward the sink node.[3] Because global relabeling is a rather expensive operation, we don't want to perform it too often.

We cannot necessarily blindly set the labels to their distances because we wish to preserve monotonicity within a branch. While labeling nodes, if we discover that a strong branch cannot reach the sink, we can eliminate it from further processing.

77a ⟨*Solver methods* 4a⟩+≡ (136c) ◁74b 77e▷

```
void globalRelabel();
```

Uses `globalRelabel` 79a.

Before getting into the implementation of the the relabeling function, let's define some small functions that we call from the solver loop to perform the global relabeling. Global relabeling is invoked based on the number of relabels that the algorithm has (normally) performed. So, we will maintain a counter that we compare to the number relabels to tell us when to trigger the global relabel.

77b ⟨*Solver data* 3e⟩+≡ (136c) ◁63a 93a▷

```
int relabelCounter;
```

Defines:
`relabelCounter`, used in chunk 78.

This is then set and updated as a factor of the number of nodes in the graph. A non-positive value means no relabeling.

77c ⟨*public Solver data* 8b⟩+≡ (136c) ◁67b

```
float relabelFrequency;
```

Uses `relabelFrequency` 77d.

77d ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁67c 120d▷

```
relabelFrequency = 0.0;
```

Defines:
`relabelFrequency`, used in chunks 5a, 77, 78, 195, 202, and 208a.

Now, we can check the counter, and if needed perform the relabeling and 'schedule' the next relabel.

77e ⟨*Solver methods* 4a⟩+≡ (136c) ◁77a 78b▷

```
void checkForRelabel();
```

Uses `checkForRelabel` 78a.

[3] Note that because the saturate-all method may create deficit nodes that are not adjacent to the sink, we actually set the distances to the nearest deficit node. Therefore, we are directing flow towards deficit nodes rather than the sink.

78a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁27d 78c▷

```
INLINE void PhaseSolver::checkForRelabel()
{
    if (numRelabels > relabelCounter) {
        globalRelabel();
        relabelCounter = numRelabels + (int)(numNodes * relabelFrequency);
    }
}
```

Defines:
`checkForRelabel`, used in chunks 4b, 30a, 34, and 77e.
Uses `globalRelabel` 79a, `numNodes` 3d, `numRelabels` 129b, `PhaseSolver` 136c, `relabelCounter` 77b, and `relabelFrequency` 77d.

Then, we just need to schedule the first relabel operation. If the frequency is zero, then we set the counter to MAXINT to prevent relabeling from ever happening.

78b ⟨*Solver methods* 4a⟩+≡ (136c) ◁77e 83c▷

```
    void initGlobalRelabel();
```

Uses `initGlobalRelabel` 78c.

78c ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁78a 87a▷

```
INLINE void PhaseSolver::initGlobalRelabel()
{
    if (relabelFrequency > 0.0) {
        relabelCounter = numRelabels + (int)(numNodes * relabelFrequency);
    } else {
        relabelCounter = MAXINT;
    }
}
```

Defines:
`initGlobalRelabel`, used in chunks 4b, 29a, 34, 78b, and 119a.
Uses `numNodes` 3d, `numRelabels` 129b, `PhaseSolver` 136c, `relabelCounter` 77b, and `relabelFrequency` 77d.

Now we can present the relabeling function, which is conceptually rather simple. We initialize all the distances, then perform a BFS to set the distances to their new value. Then we try to set the labels to match their distances, respecting monotonicity.

79a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁75 84a▷

```
void PhaseSolver::globalRelabel()
{
    STATS(numGlobalRelabels++);
    TRACE( trout << "Begin global relabeling.  Lowest label: " << lowestLabel << endl; );
    bool pruningEnabled = true;
    ⟨set initial distance labels 79b⟩
#ifdef SINK_DIST_RELABEL
    ⟨load the bfs queue with sink-adj nodes without labeling 80a⟩
#endif /* SINK_DIST_RELABEL */
    ⟨search backwards from the sink labeling nodes 80b⟩
    ⟨relabel/remove each branch 82⟩
    IFDEBUG( checkBranches(); )
}
```

Defines:
globalRelabel, used in chunks 6a, 77a, and 78a.
Uses checkBranches 133a, lowestLabel 24a, numGlobalRelabels 129b, and PhaseSolver 136c.

We start by labeling most nodes with an initial distance of MAXINT. The exception is nodes with negative excess. Normally, these would be adjacent to the sink, but with saturate-all, they can be anywere in the graph. Because they are weak roots, we will label them with a distance of one, even though they are not adjacent to the sink.

79b ⟨*set initial distance labels* 79b⟩≡ (79a)

```
    resetQ();
    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
#ifdef SINK_DIST_RELABEL
        node.distance = MAXINT;
#else
        if (node.getExcess() < 0) {
            node.distance = 1;
            putNodeQ(node);
            // if (node.getLabel() > INIT_WEAK_LABEL) - disable pruning
        } else {
            node.distance = MAXINT;
        }
#endif /* SINK_DIST_RELABEL */
    }
```

Uses distance 156c, getExcess 151d, getLabel 138g, INIT_WEAK_LABEL 60b, Node 164b, nodes 3a, numNodes 3d, putNodeQ 94a, and resetQ 93c.

80a ⟨*load the bfs queue with sink-adj nodes without labeling* 80a⟩≡ (79a)

```
for(sinkNode->resetIterations();
    sinkNode->hasMoreNeighbors();
    sinkNode->advanceNeighbors())
{
    Edge& edge = sinkNode->getCurrentNeighbor();
    Node& edgeSource = *edge.getSource();
    Node& edgeDest = *edge.getDest();
    if ((&edgeSource != sourceNode) && (&edgeDest == sinkNode))

    {
        edgeSource.distance = 1;
        putNodeQ(edgeSource);
    }
}
```

Uses `advanceNeighbors` 144a, `distance` 156c, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getSource` 165d, `hasMoreNeighbors` 144a, `Node` 164b, `putNodeQ` 94a, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

Once we've labeled the nodes and put the negative excess ones in the queue, we processing nodes in the queue until done. This performs a breadth-first search.

80b ⟨*search backwards from the sink labeling nodes* 80b⟩≡ (79a)

```
for (NodePtr nodep = getNextNodeQ(); nodep != nil; nodep = getNextNodeQ()) {
    int dist = nodep->distance + 1;
    ElIterator it = nodep->getNeighbors();
    for (EdgePtr edge = it.getNext(); edge != nil; edge=it.getNext()) {
        ⟨label neighbor as needed 81⟩
    }
}
```

Uses `distance` 156c, `getNeighbors` 145a, `getNext` 150g, and `getNextNodeQ` 93d.

If this is the first time we've visited this node, it's distance label will still be `MAXINT`. In that case, we check if we have residual capacity from the neighbor to ourselves. If so, we label the node and put it in the queue to eventually process its neighbors. If the neighbor's distances is less than `MAXINT` we must have already visited it, and its distance is less than or equal to our current one. So, we are done with it.

81 ⟨*label neighbor as needed* 81⟩≡ (80b)

```
Node& neighbor = *edge->getOtherNode(nodep);
if (neighbor.distance == MAXINT) {
//if (neighbor.distance > dist) {   // allow visting node multiple times - fix comment ab
    FlowAmount residCap = edge->residCapacity(neighbor);

    if (residCap > 0) {
        neighbor.distance = dist;
        putNodeQ(neighbor);
    }
}
```

Uses `distance` 156c, `getOtherNode` 167a, `Node` 164b, `putNodeQ` 94a, and `residCapacity` 169b.

After we've labeled the nodes, we want to scan all of the branches. For strong branches, we would like to remove them or update their labels to reflect their distances. For weak branches, we will just update the labels. However, with saturate-all initialization, we have to beware of nodes that (temporarily) have infinite distance labels due to the 'wacky' state that saturating all arcs creates (e.g. we cannot even perform flow recovery immediately after saturating all arcs).

To do this relabeling, we must iterate over all nodes in the graph looking for roots because we don't track weak branches. *Tracking weak branches wouldn't be too much work because we never create any new ones. If we had a doubly-linked list it would be pretty simple.*

82 ⟨*relabel/remove each branch* 82⟩≡ (79a)

```
for (int i = 1; i <= numNodes; i++) {
    Node& node = nodes[i];
    if ((&node == sourceNode) || (&node == sinkNode)) {
        continue;
    }
    if (node.isRootNode()) {
        int distance = node.distance;
        if (isStrongNode(node)) {
            ⟨relabel/remove strong branch 83a⟩
        } else {
            if (distance < numNodes) {
                relabelBranch(node);
            }
        }
    }
}
```

Uses `distance` 156c, `isRootNode` 141a, `isStrongNode` 17c, `Node` 164b, `nodes` 3a, `numNodes` 3d, `relabelBranch` 85, `sinkNode` 3f, and `sourceNode` 3f.

If the root of a strong branch has node label of `numNodes`, it has already been removed from the graph (by this code). Otherwise if it has a distance label of MAXINT, we assume the whole branch does too, and we can remove it from the bucket. We set the labels of all nodes in the branches to `numNodes` to prevent any strong branches from trying to merge with this branch. The exception is if we have disabled pruning. In which case we ignore the branch, leaving the labels as they are and the branch in the bucket.

83a ⟨*relabel/remove strong branch* 83a⟩≡ (82)

```
NodeLabel oldLabel = node.getLabel();
if (oldLabel < numNodes) {
    // we haven't already pruned this branch
    if ((distance == MAXINT) && pruningEnabled) {
        buckets[oldLabel].removeNode(node);
        setBranchLabel(node, numNodes);
    } else {
        ⟨relabel strong branch 83b⟩
    }
}
```

Uses `buckets` 23b, `distance` 156c, `getLabel` 138g, `numNodes` 3d, and `setBranchLabel` 84a.

If the label is less than MAXINT, we can relabel it. If we change the label of the root, we need to move it to another bucket.

83b ⟨*relabel strong branch* 83b⟩≡ (83a)

```
NodeLabel newLabel = relabelBranch(node);
if (newLabel != oldLabel) {
    assert(newLabel > oldLabel);
    buckets[oldLabel].removeNode(node);
    addStrongBranch(node);
}
```

Uses `addStrongBranch` 24c 51a, `buckets` 23b, and `relabelBranch` 85.

This is a straightfoward, recursive DFS of the children to set the labels of all nodes of a branch to a specified value.

83c ⟨*Solver methods* 4a⟩+≡ (136c) ◁78b 84b▷

```
void setBranchLabel(Node& node, NodeLabel l);
```

Uses `Node` 164b and `setBranchLabel` 84a.

84a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁79a 85▷

```
void PhaseSolver::setBranchLabel(Node& node, NodeLabel l)
{
    TRACE( trout << "relabeling " << node.getId() << " from "
                         << node.getLabel() << " to " << l << endl; );
    setLabel(node, l);
    ElIterator it = node.getChildren();
    for (EdgePtr edge = it.getNext(); edge != nil; edge=it.getNext()) {
        setBranchLabel(*edge->getOtherNode(&node), l);
    }
}
```

Defines:
`setBranchLabel`, used in chunks 35a, 83, and 92.
Uses `getChildren` 145a, `getLabel` 138g, `getNext` 150g, `getOtherNode` 167a, `Node` 164b, `PhaseSolver` 136c, and `setLabel` 13a 138g.

To relabel a branch, we need to respect the monotonicity in the tree. In a weak branch, monotonicity doesn't matter so long as the branch stays weak, but if it later becomes strong, we must have monotonicity for the rest of the implementation to work. Ideally, we'd like to increase our label to match our distance. However, our new label cannot be higher than any of our children. So, first we recursively set the labels of our children and in the process compute the minimum label among them. Then we update our own label. *we should profile and see if this should be an iterative DFS by converting the queue to a stack.*

84b ⟨*Solver methods* 4a⟩+≡ (136c) ◁83c 86b▷

```
NodeLabel relabelBranch(Node& node);
```

Uses `Node` 164b and `relabelBranch` 85.

85 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁84a 87c▷

```
NodeLabel PhaseSolver::relabelBranch(Node& node)
{
    ⟨skip infinite labels 86a⟩
    NodeLabel minLabel = MAXINT;
    ElIterator it = node.getChildren();
    for (EdgePtr edge = it.getNext(); edge != nil; edge=it.getNext()) {
        NodeLabel l = relabelBranch(*edge->getOtherNode(&node));
        if (l < minLabel) {
            minLabel = l;
        }
    }

    if (minLabel == MAXINT) {
        // node has no children
        minLabel = max(node.distance, node.getLabel());
    }
    assert(minLabel >= node.getLabel());
    node.minChildLabel = minLabel;

    if (node.distance > node.getLabel()) {
        NodeLabel newLabel = min(minLabel, node.distance);
        assert(newLabel >= node.getLabel());

        if (newLabel > node.getLabel()) {
            TRACE( trout << "relabeling " << node.getId() << " from "
                         << node.getLabel() << " to " << minLabel << endl; );
            setLabel(node, newLabel);
        }
    }

    return node.getLabel();
}
```

Defines:
`relabelBranch`, used in chunks 82–84, 89a, and 92.
Uses `children` 139b, `distance` 156c, `getChildren` 145a, `getLabel` 138g, `getNext` 150g, `getOtherNode` 167a, `minChildLabel` 156c, `Node` 164b, `PhaseSolver` 136c, and `setLabel` 13a 138g.

If we do not split when an arc goes to zero residual capacity during renormalization (`splitOnZeroCapacity=false`), then we can have nodes in a branch whose distance label is `MAXINT` because their parent edge has no residual capacity (and their are no other paths to the node). In this case, we do not relabel the node.

86a ⟨*skip infinite labels* 86a⟩≡ (85)

```
if (node.distance > numNodes) {
    cout << "Not relabeling " << node.getId() << " due to infinite distance" << endl;
    return node.getLabel();
}
```

Uses `distance` 156c, `getLabel` 138g, and `numNodes` 3d.

After scanning a node's neighbors and children, if we found no merger we need to increase the label by at least one. If we are using distance labels, it is possible to increase the node label by more, perhaps up to the distance label. However, we cannot increase it higher than any of our children because that would violate monotonicity.

One other issue to be aware of is that sometimes we are asked to increase the label of a node with infinite distance. This can happen during global relabeling in two known cases. First, if we don't split an arc with zero residual capacity, the top of a branch may have a path to the sink (and hence finite distance) while a sub-tree below a zero-capacity arc has no path. Secondly, with saturate-all initialization, we can have nodes with no path to the sink, and hence infinite label. Therefore, if we encounter a node with infinite distance, we just increment the label rather than increasing it up to its distance.

86b ⟨*Solver methods* 4a⟩+≡ (136c) ◁84b 87b▷

```
void increaseNodeLabel(Node& node);
```

Uses `increaseNodeLabel` 87a and `Node` 164b.

87a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁78c 93c▷

```
INLINE void PhaseSolver::increaseNodeLabel(Node& node)
{
    if ((node.distance > 0) && (node.distance < numNodes)) {
        NodeLabel oldLabel = node.getLabel();
        NodeLabel newLabel = min(node.distance, node.minChildLabel);
        newLabel = max(newLabel, oldLabel + 1);
        setLabel(node, newLabel);
        if (node.distance < newLabel) {
            node.distance = newLabel;
        }
    } else {
        incrementLabel(node);
        STATS(numRelabels++);           // XXX redunant
    }
    CHECK_TREE(node, node.getParentEdge());     // check monotonicity
}
```

Defines:
`increaseNodeLabel`, used in chunks 11b and 86b.
Uses `distance` 156c, `getLabel` 138g, `getParentEdge` 141a, `incrementLabel` 13b 138g, `minChildLabel` 156c, `Node` 164b, `numNodes` 3d, `numRelabels` 129b, `PhaseSolver` 136c, and `setLabel` 13a 138g.

This labeling method sets the label to the distance from the node to the sink in the original (not residual) graph. Unfortunately, this is a lot of cut-and-paste from the global relabeling code.

87b ⟨*Solver methods* 4a⟩+≡ (136c) ◁86b 89b▷

```
    void setSinkDistLabels();
```

Uses `setSinkDistLabels` 87c.

87c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁85 89c▷

```
void PhaseSolver::setSinkDistLabels()
{
    resetQ();
    ⟨initialize distances 88a⟩
    ⟨load the bfs queue with sink-adj nodes 88b⟩
    ⟨perform bfs distance labeling in orig graph 88c⟩
    //setSpInitialNodeStatus();
    ⟨scan nodes and put in buckets 92⟩

    IFDEBUG( checkBranches(); );
}
```

Defines:
`setSinkDistLabels`, used in chunks 59a and 87b.
Uses `checkBranches` 133a, `PhaseSolver` 136c, `resetQ` 93c, and `setSpInitialNodeStatus` 75.

First, we need to set all the labels to infinity.

88a ⟨*initialize distances* 88a⟩≡ (87c)

```
for (int i = 1; i <= numNodes; i++) {
    nodes[i].distance = MAXINT;
    nodes[i].setLabel(INIT_WEAK_LABEL);
}
```

Uses distance 156c, INIT_WEAK_LABEL 60b, nodes 3a, numNodes 3d, and setLabel 13a 138g.

88b ⟨*load the bfs queue with sink-adj nodes* 88b⟩≡ (87c)

```
for(sinkNode->resetIterations();
    sinkNode->hasMoreNeighbors();
    sinkNode->advanceNeighbors())
{
    Edge& edge = sinkNode->getCurrentNeighbor();
    Node& edgeSource = *edge.getSource();
    Node& edgeDest = *edge.getDest();
    if ((&edgeSource != sourceNode) && (&edgeDest == sinkNode))

    {
        edgeSource.distance = 1;
        edgeSource.setLabel(1);
        putNodeQ(edgeSource);
    }
}
```

Uses advanceNeighbors 144a, distance 156c, Edge 179a, getCurrentNeighbor 144a, getDest 165d, getSource 165d, hasMoreNeighbors 144a, Node 164b, putNodeQ 94a, resetIterations 144c, setLabel 13a 138g, sinkNode 3f, and sourceNode 3f.

88c ⟨*perform bfs distance labeling in orig graph* 88c⟩≡ (87c)

```
for (NodePtr nodep = getNextNodeQ(); nodep != nil; nodep = getNextNodeQ()) {
    int dist = nodep->distance + 1;
    ElIterator it = nodep->getNeighbors();
    for (EdgePtr edgep = it.getNext(); edgep != nil; edgep = it.getNext()) {
        ⟨set neighbor distance as needed 89a⟩
    }
}
```

Uses distance 156c, getNeighbors 145a, getNext 150g, and getNextNodeQ 93d.

89a ⟨*set neighbor distance as needed* 89a⟩≡ (88c)

```
Node& neighbor = *edgep->getOtherNode(nodep);
if ((&neighbor != sourceNode) && (&neighbor != sinkNode) &&
    (edgep->getDest() == nodep) &&          // want an 'in' arc
    (neighbor.distance > dist))
{
        neighbor.distance = dist;
        // neighbor.setLabel(dist);  - using relabelBranch
        putNodeQ(neighbor);
}
```

Uses `distance` 156c, `getDest` 165d, `getOtherNode` 167a, `Node` 164b, `putNodeQ` 94a, `relabelBranch` 85, `setLabel` 13a 138g, `sinkNode` 3f, and `sourceNode` 3f.

This function sets distance labels to the nearest negative deficit node, and it sets the node labels to the highest value possible while still observing monotonicity. Then, it puts the strong nodes in buckets.

89b ⟨*Solver methods* 4a⟩+≡ (136c) ◁87b 93b▷

```
void setDeficitDistLabels();
```

Uses `setDeficitDistLabels` 89c.

89c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁87c 95a▷

```
void PhaseSolver::setDeficitDistLabels()
{
    resetQ();
    ⟨initialize distances and load queue 90a⟩
    ⟨perform bfs distance labeling 90c⟩
    ⟨scan nodes and put in buckets 92⟩

    IFDEBUG( checkBranches(); );
}
```

Defines:
`setDeficitDistLabels`, used in chunks 59a and 89b.
Uses `checkBranches` 133a, `PhaseSolver` 136c, and `resetQ` 93c.

We establish initial distances and labels. All nodes get an initial label of one (INIT_WEAK_LABEL). Deficit nodes get a distance of one, and they get inserted into the BFS queue. All other nodes get an infinite distance.

90a ⟨*initialize distances and load queue* 90a⟩≡ (89c) 90b▷

```
for (int i = 1; i <= numNodes; i++) {
    Node& node = nodes[i];
    node.setLabel(INIT_WEAK_LABEL);
    if (node.getExcess() < 0) {
        node.distance = 1;
        putNodeQ(node);
    } else {
        node.distance = MAXINT;
    }
}
```

Uses distance 156c, getExcess 151d, INIT_WEAK_LABEL 60b, Node 164b, nodes 3a, numNodes 3d, putNodeQ 94a, and setLabel 13a 138g.

90b ⟨*initialize distances and load queue* 90a⟩+≡ (89c) ◁90a

```
// maybe I should scan the sink-adj nodes looking for
// zero-defiict nodes
```

Uses nodes 3a.

The BFS is simple: pull a node out of the queue and scan its neighbors. Continue until there are no nodes in the queue.

90c ⟨*perform bfs distance labeling* 90c⟩≡ (89c)

```
for (NodePtr nodep = getNextNodeQ(); nodep != nil; nodep = getNextNodeQ()) {
    int dist = nodep->distance + 1;
    ElIterator it = nodep->getNeighbors();
    for (EdgePtr edgep = it.getNext(); edgep != nil; edgep = it.getNext()) {
        ⟨set zero-deficit neighbor distance as needed 91⟩
    }
}
```

Uses distance 156c, getNeighbors 145a, getNext 150g, and getNextNodeQ 93d.

When we scan a node, we are looking for neighbors that can reach us in the residual graph - i.e. positive residual capacity from the neighbor to us. We are only interested in neighbors with a distance greater than one more than our distance (due to BFS, the distance should always be infinity). If we find such a node, we set its distance and label, and put it at the end of the queue for subsequence processing.

91 ⟨*set zero-deficit neighbor distance as needed* 91⟩≡ (90c)

```
Node& neighbor = *edgep->getOtherNode(nodep);
if ((neighbor.distance > dist) &&
//      (edgep->getDest() == nodep))            // want an 'in' arc
(edgep->residCapacity(neighbor) > 0))       // hack - no resid cap check
{
        neighbor.distance = dist;
        putNodeQ(neighbor);
}
```

Uses `distance` 156c, `getDest` 165d, `getOtherNode` 167a, `Node` 164b, `putNodeQ` 94a, and `residCapacity` 169b.

After we have performed the BFS, the labels are all initialized to either their distance or the default value of one, and the distances reflect the distance to a deficit node, which may still be infinity. For simplicity, let's assume that this code is only called by `saturateAll` so we have no branch structure to worry about - we only worry about nodes.

If an excess node has infinite distance, it cannont reach a weak node, so we can ignore it (by setting its label to `numNodes` and not putting it in a bucket). Other strong nodes go into buckets, and any other nodes are initialized and ready to go. For all nodes, we update the label count appropriately.

92 ⟨*scan nodes and put in buckets* 92⟩≡ (87c 89c)

```
maxInitialLabel = INIT_WEAK_LABEL;
for (int i = 1; i <= numNodes; i++) {
    Node& node = nodes[i];
    if ((&node == sourceNode) || (&node == sinkNode)) {
        continue;
    }
    if (node.isRootNode() && (node.distance < MAXINT)) {
        relabelBranch(node);
    }
    if (node.getExcess() > 0) {
        if (node.distance == MAXINT) {
            setBranchLabel(node, numNodes);
            continue;
        } else {
            addStrongBranch(node);
        }
    }  else if (node.distance == MAXINT) {
        if (node.isRootNode()) {
            // noticed with cher graphs that some zero-deficit nodes are
            // unreachable during initialization???
            // XXX this might be an uneeded hack now
            // 12/8/00 - found case with gpps where strong branch was
            // pruned, but child still had infinite distance, so it
            // was getting reset to one.  isRootNode check will block that.
            node.setLabel(INIT_WEAK_LABEL);
            node.distance = 1;
        }
    }
    labelCount[node.getLabel()]++;          // XXX - unneeded?
    node.resetIterations();
    if (node.getLabel() > maxInitialLabel) {
        maxInitialLabel = node.getLabel();
    }
```

```
    }
```

Uses addStrongBranch 24c 51a, distance 156c, getExcess 151d, getLabel 138g, INIT_WEAK_LABEL 60b, isRootNode 141a, labelCount 5c, maxInitialLabel 63b, Node 164b, nodes 3a, numNodes 3d, relabelBranch 85, resetIterations 144c, setBranchLabel 84a, setLabel 13a 138g, sinkNode 3f, and sourceNode 3f.

2.8.3 Queue of Nodes

Here is a queue of nodes. This ought to be moved to Node.nw.

93a ⟨*Solver data* 3e⟩+≡ (136c) ◁77b 95b▷

```
    NodePtr* nodeQ;
    NodePtr* qRead;
    NodePtr* qWrite;
```

93b ⟨*Solver methods* 4a⟩+≡ (136c) ◁89b 111b▷

```
    void resetQ();
    NodePtr getNextNodeQ();
    void putNodeQ(Node& node);
```

Uses getNextNodeQ 93d, Node 164b, putNodeQ 94a, and resetQ 93c.

93c ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁87a 93d▷

```
INLINE void PhaseSolver::resetQ()
{
    qRead = qWrite = nodeQ;
    *qRead = nil;
}
```

Defines:
resetQ, used in chunks 72b, 79b, 87c, 89c, and 93b.
Uses PhaseSolver 136c.

93d ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁93c 94a▷

```
INLINE NodePtr PhaseSolver::getNextNodeQ()
{
    if (qRead < qWrite) {
        return *qRead++;
    } else {
        return nil;
    }
}
```

Defines:
getNextNodeQ, used in chunks 73d, 80b, 88c, 90c, and 93b.
Uses PhaseSolver 136c.

94a ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁93d 113▷

```
INLINE void PhaseSolver::putNodeQ(Node& node)
{
    assert(qWrite <= (nodeQ + numNodes + 1));
    *qWrite = &node;
    qWrite++;
}
```

Defines:
`putNodeQ`, used in chunks 73c, 74a, 79–81, 88–91, and 93b.
Uses `Node` 164b, `numNodes` 3d, and `PhaseSolver` 136c.

2.9 Recovering a Feasible Flow

When the pseudoflow algorithm completes, there are no merger arcs between the strong nodes and weak nodes, and there is excess at the roots of strong branches and deficits at the roots of weak branches. We need to return the excess/deficit to the source/sink. We do this by reducing the flows on arcs along paths to the source/sink.

To find paths to the source/sink we start from nodes with excess/deficit and perform a depth first search along neighbor arcs that have what we call 'reduction capacity', which is like residual capacity except that it only applies to reducing the flow on arcs. When we find the source/sink, we reduce the flow along the path we identified by an amount equal to the capacity of the bottleneck arc. While we search, we label each node with a mark value to identify cycles.

This code is not really a bottleneck, so we don't need to optimize it much. However, if we were to scan the neighbors of the source and sink node, we could easily return flow in one simple push. This works really well for the sink because (unless were are using saturate-all) all of the deficit nodes are adjacent to the sink.

94b ⟨*Solver public declarations* 94b⟩≡ (136c) 96c▷

```
void convertToFlow();
```

Uses `convertToFlow` 95a.

95a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁89c 97a▷

```
void PhaseSolver::convertToFlow()
{
    currentMark = 1024;
    ⟨replace source and sink arcs 96a⟩
    returnSinkAdjDeficit();
    // returnSourceAdjExcess();          XXX - broken!?!

    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        if (&node == sourceNode || &node == sinkNode) {
            continue;
        }
        if (node.getExcess() != 0) {
            pushRootExcess(node);
        }
    }
}
```

Defines:
`convertToFlow`, used in chunks 94b, 195, 202, and 212.
Uses `getExcess` 151d, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, `pushRootExcess` 100, `returnSinkAdjDeficit` 97a, `returnSourceAdjExcess` 98a, `sinkNode` 3f, and `sourceNode` 3f.

95b ⟨*Solver data* 3e⟩+≡ (136c) ◁93a 120c▷

```
    int currentMark;
```

When we initialized the psuedoflow (in `saturateSourceSinkArcs`), we removed the arcs from/to the source/sink to prevent flow from going back to the source/sink. Now, we actaully want to be able to get to the source and sink, so we add them back. We begin by replacing arcs that go from the sink to a source-adjacent node in the source-adjacent node.

96a ⟨*replace source and sink arcs* 96a⟩≡ (95a) 96b▷

```
for(sourceNode->resetIterations();
    sourceNode->hasMoreNeighbors();
    sourceNode->advanceNeighbors())
{
    Edge& edge = sourceNode->getCurrentNeighbor();
    Node& edgeSource = *edge.getSource();
    Node& edgeDest = *edge.getDest();
    if ((&edgeSource == sourceNode) && (&edgeDest != sinkNode)) {
        edgeDest.addNeighbor(edge);
    }
}
```

Uses `addNeighbor` 139d, `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getSource` 165d, `hasMoreNeighbors` 144a, `Node` 164b, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

Replace arcs that go from a sink-adjacent node in to the sink in the sink-adjacent node.

96b ⟨*replace source and sink arcs* 96a⟩+≡ (95a) ◁96a

```
for(sinkNode->resetIterations();
    sinkNode->hasMoreNeighbors();
    sinkNode->advanceNeighbors())
{
    Edge& edge = sinkNode->getCurrentNeighbor();
    Node& edgeSource = *edge.getSource();
    Node& edgeDest = *edge.getDest();
    if ((&edgeSource != sourceNode) && (&edgeDest == sinkNode)) {
        edgeSource.addNeighbor(edge);
    }
}
```

Uses `addNeighbor` 139d, `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getSource` 165d, `hasMoreNeighbors` 144a, `Node` 164b, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

For initializations schemes other than saturate-all, we know that all of the weak roots will be adjacent to the sink. This makes returning their deficit trivial - just one arc, and it should always have sufficient capacity.

96c ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁94b 97b▷

```
void returnSinkAdjDeficit();
```

Uses `returnSinkAdjDeficit` 97a.

97a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁95a 98a▷

```
void PhaseSolver::returnSinkAdjDeficit()
{
    for(sinkNode->resetIterations();
        sinkNode->hasMoreNeighbors();
        sinkNode->advanceNeighbors())
    {
        Edge& edge = sinkNode->getCurrentNeighbor();
        Node& sinkAdjNode = *edge.getSource();
        if (&sinkAdjNode != sinkNode) {
            FlowAmount excess = sinkAdjNode.getExcess();
            if (excess < 0) {
                FlowAmount amt = -excess;
                if (amt > edge.getFlow()) amt = edge.getFlow();
                edge.reduceFlow(amt);
                sinkAdjNode.incrementExcess(amt);
                TRACE( trout << "returned " << -excess << " units from "
                             << sinkAdjNode.getId() << " to sink" << endl; );
            }
        }
    }
}
```

Defines:
`returnSinkAdjDeficit`, used in chunks 95a and 96c.
Uses `advanceNeighbors` 144a, `Edge` 179a, `excess` 151b, `getCurrentNeighbor` 144a, `getExcess` 151d, `getFlow` 167d, `getSource` 165d, `hasMoreNeighbors` 144a, `incrementExcess` 151d, `Node` 164b, `PhaseSolver` 136c, `reduceFlow` 104a 168e, `resetIterations` 144c, and `sinkNode` 3f.

For strong nodes, we are not so lucky. However, we can still pick up some 'low hanging fruit'. Chances are there is no excess at a source-adjacent node because it has merged with weak nodes. However, we can trace the path up to the root of a source-adjacent node. We can return the excess at the root to the source via the path we navigated to find the root. This will remove excess from some strong nodes in the graph but not all of them because our root may have merged and split. This will be more efficient that discovering the path from the root node.

97b ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁96c 99c▷

```
void returnSourceAdjExcess();
```

Uses `returnSourceAdjExcess` 98a.

98a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁97a 100▷

```
void PhaseSolver::returnSourceAdjExcess()
{
    for(sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Edge& edge = sourceNode->getCurrentNeighbor();
        Node& srcAdjNode = *edge.getDest();
        FlowAmount bottleCap = edge.getFlow();
        ⟨find bottleneck capacity to strong root 98b⟩
        if (bottleCap > 0) {
            ⟨reduce flow along path to strong root 99b⟩
        }
    }
}
```

Defines:
`returnSourceAdjExcess`, used in chunks 95a and 97b.
Uses `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getFlow` 167d, `hasMoreNeighbors` 144a, `Node` 164b, `PhaseSolver` 136c, `resetIterations` 144c, and `sourceNode` 3f.

First we need to find the bottleneck capacity of all arcs along the path to the root of the branch. We do this iteratively rather than recursively. Theoretically, this should yeild better performance.

98b ⟨*find bottleneck capacity to strong root* 98b⟩≡ (98a) 99a▷

```
Node* currNode = &srcAdjNode;
while (currNode->isRootNode() == false) {
    FlowAmount parentCap = currNode->getParentEdge()->getFlow();
    if (parentCap < bottleCap) {
        bottleCap = parentCap;
    }
    currNode = currNode->getParentNode();
}
```

Uses `getFlow` 167d, `getParentEdge` 141a, `getParentNode` 141a, `isRootNode` 141a, and `Node` 164b.

Then we need to take the minimum of that and excess of the root. If the root has a deficit, we indicate that by setting the bottleneck capacity to zero.

99a ⟨*find bottleneck capacity to strong root* 98b⟩+≡ (98a) ◁98b

```
FlowAmount branchExcess = currNode->getExcess();
if (branchExcess > 0) {
    if (branchExcess < bottleCap) {
        bottleCap = branchExcess;
    }
} else {
    bottleCap = 0;
}
```

Uses `getExcess` 151d.

Once we know the bottleneck capacity, reducing the flow is pretty trivial starting from the arc out of the source. Again we do this iteratively.

99b ⟨*reduce flow along path to strong root* 99b⟩≡ (98a)

```
edge.reduceFlow(bottleCap);
currNode = &srcAdjNode;
while (currNode->isRootNode() == false) {
    currNode->getParentEdge()->reduceFlow(bottleCap);
    currNode = currNode->getParentNode();
}
currNode->decrementExcess(bottleCap);
TRACE( trout << "returned " << bottleCap << " units from "
             << srcAdjNode.getId() << " ("
             << srcAdjNode.getExcess() << ") to source" << endl; );
```

Uses `decrementExcess` 151d, `getExcess` 151d, `getParentEdge` 141a, `getParentNode` 141a, `isRootNode` 141a, `reduceFlow` 104a 168e, and `source` 165a.

If a node has excess/deficit, we begin search from the node among its neighbors trying to find a path back to the source/sink. Each time we find a path, we reduce the excess/deficit on this node. We continue until the node has zero excess.

99c ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁97b 101▷

```
void pushRootExcess(Node& node);
```

Uses `Node` 164b and `pushRootExcess` 100.

100 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁98a 102▷

```
void PhaseSolver::pushRootExcess(Node& node)
{
    bool seekingSource = (node.getExcess() > 0) ? true : false;
    while (node.getExcess() != 0) {
        currentMark++;
        FlowAmount bottleCap = (seekingSource) ?
                               node.getExcess() : -node.getExcess();

        bool found = visitNode(node, node, bottleCap,
                                    seekingSource);
        if (found) {
            TRACE( trout << "Pushing " << bottleCap
                         << " units of excess from " << node.getId()
                         << " (" << node.getExcess() << ")" << endl;)
            if (seekingSource) {
                node.decrementExcess(bottleCap);
                assert(node.getExcess() >= 0);
            } else {
                node.incrementExcess(bottleCap);
                assert(node.getExcess() <= 0);
            }
        } else {
            //assert("There must be a path to the source/sink" == nil);
            cout << "There must be a path to the source/sink " << node.getId() << endl;
            return;
        }

        if (node.getExcess() != 0) {
            assert(node.hasMoreNeighbors());
        }
    }
}
```

Defines:
`pushRootExcess`, used in chunks 95a and 99c.
Uses `decrementExcess` 151d, `excess` 151b, `getExcess` 151d, `hasMoreNeighbors` 144a, `incrementExcess` 151d, `Node` 164b, `PhaseSolver` 136c, `source` 165a, and `visitNode` 102.

To dispose of the excess/deficit at a node, we recursively search its neighbors looking for the source/sink. Once we find the source/sink, we reduce the flow along the path from the node to the source/sink by the amount of the bottleneck capacity along the path. We ignore arcs with zero residual capacity.

The other possible outcome of our search is that we find a cycle. Hoch97 says we want to reduce the flow of all arcs along the cycle. However, we have found that this is unnecessary. If we find a node that we've already visited, we just ignore it and search for other paths to the source/sink. Theoretically, ours is an $O(m^2)$ algorithm compared to $O(mn)$. However, we found no appreciable difference in practice. The basic difference is that cycle-cancelling expends effort reducing the flow along cycles which does not reduce the excess/deficit, which is a bit of a waste of effort. Whereas our algorithm does not waste the effort, but may expend more effort because it could continue 'stumbling' into the same cycles over and over.

101 ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁99c 103c▷

```
bool visitNode(Node& node, Node& prevNode, FlowAmount& bottleCap,
               bool seekingSource);
```

Uses `Node` 164b and `visitNode` 102.

102 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁100 104a▷

```
bool
PhaseSolver::visitNode(Node& node, Node& prevNode, FlowAmount& bottleCap,
                       bool seekingSource)
{
    ⟨check for terminating recursion 103a⟩
    bool result = false;
    node.flag = currentMark;

    node.resetIterations();
    while (node.hasMoreNeighbors()) {
        Edge& neighborEdge = node.getCurrentNeighbor();
        Node& neighbor = *neighborEdge.getOtherNode(&node);
        FlowAmount residCap = neighborEdge.reductionCapacity(
                                    seekingSource ? neighbor : node);
        if (residCap > 0) {
            if (&neighbor != &prevNode) {
                ⟨visit neighbor node 103b⟩
            }
        } else {
            // useless arc - remove it
            node.removeNeighbor(neighborEdge);
        }
        node.advanceNeighbors();
    }

    return result;
}
```

Defines:
`visitNode`, used in chunks 100, 101, and 103b.
Uses `advanceNeighbors` 144a, `Edge` 179a, `flag` 155d, `getCurrentNeighbor` 144a, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `Node` 164b, `PhaseSolver` 136c, `reductionCapacity` 170a, `removeNeighbor` 139f, and `resetIterations` 144c.

We terminate the recursion when we find the source/sink. We could have eliminated the tail recursion by checking for it before visiting the node (in the `while` loop), but this seems simpler. Note, it should be impossible to find a path to the sink/source if we are searching for the source/sink.

103a ⟨*check for terminating recursion* 103a⟩≡ (102)

```
if (node.flag == currentMark) {
    return false;
} else if (&node == sourceNode) {
    assert(seekingSource);
    return true;
} else if (&node == sinkNode) {
    assert(!seekingSource);
    return true;
}
```

Uses `flag` 155d, `sinkNode` 3f, and `sourceNode` 3f.

To visit a node, we compute a new, tentative bottleneck capacity, but we do not assign it to the bottleneck parameter (which is passed by reference) until we find the source/sink. After visiting the node (and its neighbors), if we found something, we will reduce the flow on the arc, and break the search to return. to the calling function.

103b ⟨*visit neighbor node* 103b⟩≡ (102)

```
FlowAmount newBottleCap = (bottleCap < residCap) ? bottleCap : residCap;
bool status = visitNode(neighbor, node, newBottleCap,
                                    seekingSource);
// see if we found source/sink or a cycle
if (status) {
    bottleCap = newBottleCap;
    reduceFlow(node, neighborEdge, bottleCap);
    result = status;
    break;          // stop checking neighbors
}
```

Uses `neighbors` 139b, `reduceFlow` 104a 168e, `source` 165a, `stop` 214b, and `visitNode` 102.

When we reduce the flow on an edge, if the edge becomes saturated, we need to remove it from the appropriate node. Currently, this code is 'unidirectional' (i.e. we ignore the orientation of the arc), so we express everything in terms of the from/to direction we are moving flow along.

103c ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁101 104b▷

```
void reduceFlow(Node& from, Edge& edge,
                FlowAmount amt);
```

Uses `Edge` 179a, `Node` 164b, and `reduceFlow` 104a 168e.

104a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁102 105▷

```
void PhaseSolver::reduceFlow(Node& from, Edge& edge, FlowAmount amt)
{
    edge.reduceFlow(amt);

    if (edge.getFlow() == 0) {
        from.removeNeighbor(edge);
    }
}
```

Defines:
`reduceFlow`, used in chunks 97a, 99b, 103, and 168d.
Uses `Edge` 179a, `getFlow` 167d, `Node` 164b, `PhaseSolver` 136c, and `removeNeighbor` 139f.

2.10 Source-Sink Parametric Sensitivity Analysis

We support simple parametric sensitivity analysis based on the form used by GGT. Given a sequence of increasing values of a parameter, λ, the source-adjacent arcs have non-decreasing capacity, and the sink-adjacent arcs have non-increasing capacity.

We establish the initial capacity of the network by setting an initial value for λ and solve the resulting network. Then, we can set new values of λ, update the capacities, and re-solve the network in less time than starting from scratch.

Setting the initial capacity is simple: just set the parameter for the source- and sink-adjacent arcs.

NB: this code chokes if there is a parametric arc from the source to the sink. It gets saturated once out of the source, and then we get an assertion violation when we try to saturate it into the sink.

104b ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁103c 106a▷

```
void setInitialCapacity(double lambda);
```

Uses `setInitialCapacity` 105 172c.

105 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁104a 106b▷

```
void PhaseSolver::setInitialCapacity(double lambda)
{
    for(sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Edge& edge = sourceNode->getCurrentNeighbor();
        Node& otherNode = *edge.getDest();
        if (&otherNode != sourceNode) {
            bool negative = edge.setInitialCapacity(lambda);
            if (negative) {
                sourceNode->removeNeighbor(edge);
                edge.reInit(otherNode, *sinkNode);
                sinkNode->addNeighbor(edge);
            }
        }
    }

    for(sinkNode->resetIterations();
        sinkNode->hasMoreNeighbors();
        sinkNode->advanceNeighbors())
    {
        Edge& edge = sinkNode->getCurrentNeighbor();
        Node& otherNode = *edge.getSource();
        if (&otherNode != sinkNode) {
            bool negative = edge.setInitialCapacity(lambda);
            if (negative == false) {
                sinkNode->removeNeighbor(edge);
                edge.reInit(*sourceNode, otherNode);
                sourceNode->addNeighbor(edge);
            }
        }
    }
}
```

Defines:
`setInitialCapacity`, used in chunks 104b, 172b, and 202.
Uses `addNeighbor` 139d, `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getSource` 165d, `hasMoreNeighbors` 144a, `Node` 164b, `PhaseSolver` 136c, `reInit` 174a, `removeNeighbor` 139f, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

Updating the capacity to the next parameter value is a bit more involved. Adjusting the capacities once a flow has been established may create excess at the the source- or sink-adjacent node. This excess needs to be pushed towards the root of the branch containing the node. This may create a new strong branch if the amount of excess exceeds the previous deficit at a weak root. For strong branches, it will only increase the excess at the root. Because of this, we do not technically need to perform this since we are just making a strong branch stronger, and we already know that it will be in the source-set of any cut involving higher values of λ. Note also that there may be splits as we push the new excess up the tree. Fortunately, `weakPush` handles all aspects of this operation: pushing excess, splitting branches, and queuing newly strong branch roots.

106a ⟨*Solver public declarations* 94b⟩+≡ (136c) ◁104b

```
void setNextCapacityParameter(double oldLambda, double newLambda);
```

Uses `setNextCapacityParameter` 106b.

106b ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁105 112a▷

```
void PhaseSolver::setNextCapacityParameter(double oldLambda, double newLambda)
{
    assert(newLambda >= oldLambda);
    ⟨remove strong nodes from graph 109c⟩
    ⟨increase capacity on arcs out of source 107⟩
    ⟨decrease capacity on arcs into sink 108⟩
}
```

Defines:
`setNextCapacityParameter`, used in chunks 106a and 202.
Uses `PhaseSolver` 136c.

We increase the capacity of the arcs out of the source node. This will create excess at the source-adjacent node. If this node is already marked as strong, we can just leave the excess because we have already removed the node and its branch from the graph. Otherwise, we push the flow to the (weak) root which may cause the branch to become strong.

107 ⟨*increase capacity on arcs out of source* 107⟩≡ (106b)

```
STATS(int numSourceInc = 0; int numSourceSplit = 0; );
for(sourceNode->resetIterations();
    sourceNode->hasMoreNeighbors();
    sourceNode->advanceNeighbors())
{
    Edge& edge = sourceNode->getCurrentNeighbor();
    Node& sourceAdjNode = *edge.getDest();
    if (&sourceAdjNode != sourceNode) {
        FlowAmount delta = 0;
        bool crossed = edge.increaseLambda(oldLambda, newLambda, delta);
        assert(crossed == false);
        if (delta > 0) {
            STATS(numSourceInc++; );
            edge.increaseFlow(delta);
            sourceAdjNode.incrementExcess(delta);
            if (sourceAdjNode.getLabel() < numNodes) {
                bool split = weakPush(sourceAdjNode);
                STATS( if(split) numSourceSplit++; );
            }
        }
    }
}
STATS(cout << "new lambda: " << newLambda << "  numSourceInc = " <<
        numSourceInc << "  numSourceSplit = " << numSourceSplit << endl; );
```

Uses `advanceNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, `getDest` 165d, `getLabel` 138g, `hasMoreNeighbors` 144a, `increaseFlow` 168a, `increaseLambda` 173b, `incrementExcess` 151d, `Node` 164b, `numNodes` 3d, `resetIterations` 144c, `sourceNode` 3f, `split` 145c, and `weakPush` 18.

Processing the arcs into the sink is more complicated because the capacity is actually negative and increasing towards zero (but we take the absolute value so it looks like it's decreasing). If it crosses zero, we must move the arc to be adjacent to the source. change in capacity.

108 ⟨*decrease capacity on arcs into sink* 108⟩≡ (106b) 111a▷

```
STATS(int numSinkInc = 0; int numSinkSplit = 0; int numSinkCross = 0; );
for(sinkNode->resetIterations();
    sinkNode->hasMoreNeighbors();
    sinkNode->advanceNeighbors())
{
    Edge& edge = sinkNode->getCurrentNeighbor();
    Node& sinkAdjNode = *edge.getSource();
    if (&sinkAdjNode != sinkNode) {
        FlowAmount delta = 0;
        bool crossed = edge.increaseLambda(oldLambda, newLambda, delta);
        if (delta > 0) {
            STATS(numSinkInc++; );
            if (crossed) {
                STATS(numSinkCross++; );
                ⟨move arc to source 109a⟩
            } else {
                edge.decreaseFlow(delta);
            }
            sinkAdjNode.incrementExcess(delta);
            if (sinkAdjNode.getLabel() < numNodes) {
                bool split = weakPush(sinkAdjNode);
                STATS(if (split) numSinkSplit++; );
            }
        }
    }
}
STATS(cout << "new lambda: " << newLambda << "  numSinkInc = "
           << numSinkInc << "  numSinkSplit = " << numSinkSplit
           << "  numSinkCross = " << numSinkCross << endl; );
```

Uses `advanceNeighbors` 144a, `decreaseFlow` 168c, `Edge` 179a, `getCurrentNeighbor` 144a, `getLabel` 138g, `getSource` 165d, `hasMoreNeighbors` 144a, `increaseLambda` 173b, `incrementExcess` 151d, `Node` 164b, `numNodes` 3d, `resetIterations` 144c, `sinkNode` 3f, `split` 145c, and `weakPush` 18.

When the capacity of a sink-adjacent arc crosses zero, we need to move it from the sink to the source. Note that we do not need to update the neighbor list of the node because we already removed sink as a neighbor during initialization ([saturateSourceSinkArcs]).

109a ⟨*move arc to source* 109a⟩≡ (108) 109b▷

```
sinkNode->removeNeighbor(edge);
edge.reInit(*sourceNode, sinkAdjNode);
sourceNode->appendNeighbor(edge);
```

Uses `appendNeighbor` 139d, `reInit` 174a, `removeNeighbor` 139f, `sinkNode` 3f, and `sourceNode` 3f.

Next we need to set the flow amount on the arc. This is the new, positive capacity, which is less than delta - i.e. saturate the arc. However, we increase the excess of the node by the full delta amount (see above).

109b ⟨*move arc to source* 109a⟩+≡ (108) ◁109a

```
edge.saturate();
```

Uses `saturate` 171d.

Should this be a separate function so that we can account for its time and be able to easily remove it for comparisons?

First, we mark all of the active, strong nodes as being strong. We do this by iterating over the buckets and then proceeding down each branch that we find.

109c ⟨*remove strong nodes from graph* 109c⟩≡ (106b) 109d▷

```
int maxFullBucket = 0;
for (int i = 1; i <= highestLabel; i++) {
    NodePtr node = buckets[i].getHead();
    while (node != nil) {
        maxFullBucket = i;
        labelSubtree(*node, numNodes);
        node = node->getNext();
    }
}
```

Uses `buckets` 23b, `getHead` 22a 166d, `getNext` 150g, `highestLabel` 24a, `labelSubtree` 112a, and `numNodes` 3d.

Next, we need to 'remove' the strong nodes from the graph. This can effectively be done by just emptying the buckets that contain all of the strong branches.

109d ⟨*remove strong nodes from graph* 109c⟩+≡ (106b) ◁109c 110▷

```
// remove all strong roots from bucket lists
for (int i = 1; i <= maxFullBucket; i++) {
    buckets[i].emptyBucket();
}
```

Uses `buckets` 23b and `emptyBucket` 22b.

For maximum efficiency, we should remove all arcs from weak nodes to strong nodes. It's not clear if this is mandatory. On the one hand, one could argue that if we leave them, we're just scaning useless arcs when a weak node later becomes strong, which is inefficient, but benign. On the other hand, if we don't remove them, then the labels of the strong nodes that we have removed from the graph will not increase and might be mistaken for weak nodes based on their low labels. This might be OK, but it would be 'resurrecting' old strong nodes, which is a waste at the very least.

110 ⟨*remove strong nodes from graph* 109c⟩+≡ (106b) ◁109d

```
/*
for (int i = 1; i <= numNodes; i++) {
    Node& node = nodes[i];
    if (node.flagIsSet(STRONG_NODE) || &node == sourceNode || &node == sinkNode) {
        continue;
    }
    for (node.resetIterations();
         node.hasMoreNeighbors();
         node.advanceNeighbors())
    {
        Edge& edge = node.getCurrentNeighbor();
        Node& otherNode = *edge.getOtherNode(&node);
        if (otherNode.flagIsSet(STRONG_NODE)) {
            assert(edge.residCapacity(otherNode) == 0);
            node.removeNeighbor(edge);
        }
    }
}
*/
```

Uses `advanceNeighbors` 144a, `Edge` 179a, `flagIsSet` 155d, `getCurrentNeighbor` 144a, `getOtherNode` 167a, `hasMoreNeighbors` 144a, `Node` 164b, `nodes` 3a, `numNodes` 3d, `removeNeighbor` 139f, `resetIterations` 144c, `residCapacity` 169b, `sinkNode` 3f, and `sourceNode` 3f.

Here's a little debug code to print out the capacity of the adjusted source and sink arcs.

111a ⟨*decrease capacity on arcs into sink* 108⟩+≡ (106b) ◁108

```
#ifdef TRACING
    FlowAmount sourceCapacity = 0;
    for(sourceNode->resetIterations();
        sourceNode->hasMoreNeighbors();
        sourceNode->advanceNeighbors())
    {
        Edge& edge = sourceNode->getCurrentNeighbor();
        Node& sourceAdjNode = *edge.getDest();
        if (&sourceAdjNode != sourceNode) {
            sourceCapacity += edge.getCapacity();
        }
    }
    TRACE(trout << "Old lambda " << oldLambda << " new lambda " << newLambda << endl; );
    TRACE(trout << "Source capacity: " << sourceCapacity << endl; );

    FlowAmount sinkCapacity = 0;
    for(sinkNode->resetIterations();
        sinkNode->hasMoreNeighbors();
        sinkNode->advanceNeighbors())
    {
        Edge& edge = sinkNode->getCurrentNeighbor();
        Node& sinkAdjNode = *edge.getSource();
        if (&sinkAdjNode != sinkNode) {
            sinkCapacity += edge.getCapacity();
        }
    }
    TRACE( trout << "Sink capacity: " << sinkCapacity << endl; );
#endif /*TRACING*/
```

Uses `advanceNeighbors` 144a, `capacity` 167d, `Edge` 179a, `getCapacity` 167d, `getCurrentNeighbor` 144a, `getDest` 165d, `getSource` 165d, `hasMoreNeighbors` 144a, `Node` 164b, `resetIterations` 144c, `sinkNode` 3f, and `sourceNode` 3f.

111b ⟨*Solver methods* 4a⟩+≡ (136c) ◁93b 114b▷

```
    void labelSubtree(Node& node, NodeLabel l);
```

Uses `labelSubtree` 112a and `Node` 164b.

112a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁106b 115a▷

```
void PhaseSolver::labelSubtree(Node& node, NodeLabel l)
{
    node.resetIterations();
    while (node.hasMoreChildren()) {
        Node& child = node.getCurrentChild();
        labelSubtree(child, l);
        node.advanceChildren();
    }

    setLabel(node, l);
}
```

Defines:
`labelSubtree`, used in chunks 109c and 111b.
Uses `advanceChildren` 143b, `getCurrentChild` 143b, `hasMoreChildren` 143b, `Node` 164b, `PhaseSolver` 136c, `resetIterations` 144c, and `setLabel` 13a 138g.

2.11 General Parametric Sensitivity Analysis

We also support more general parametric sensitivity analysis. We allow the capacity of any edge to be adjusted in any way. During this process, we keep saturated arcs saturated by increasing their flow. If we decrease the capacity of an arc (staturate or not) below it's existing flow, we reduce the flow on the arc to match the capacity. This process creates excess and deficit throughout the graph. So, we renormalize the graph by pushing excess up towards the root and pulling excess down to nodes with deficit. This process can lead to strong nodes becoming weak and weak nodes becoming strong. Also, as arcs are saturated, we split the tree.

After the tree is re-normalized, we can re-solve it. Unfortunately, we have not been able to establish any complexity results that would suggest that this process is any faster than just repeatedly re-applying the solver. However, we expect it will run faster in practice.

First, we need a function to adjust the capacity of an arc.

112b ⟨*Solver protected members* 3a⟩+≡ (136c) ◁58b 129b▷

```
int adjustCapacity(EdgePtr hint, NodeId src, NodeId dest, FlowAmount newCap);
```

Uses `adjustCapacity` 113 175a and `dest` 165a.

113 ⟨*Solver inline implementations* 9a⟩+≡ (135b 136d) ◁94a

```
INLINE FlowAmount PhaseSolver::adjustCapacity(EdgePtr hint, NodeId srcId,
                                  NodeId destId, FlowAmount newCap)
{
    Edge* edge = nil;
    ⟨search for edge 114a⟩
    assert((edge->getSource()->getId() == srcId) && (edge->getDest()->getId() == destId));
    FlowAmount flowDelta = edge->adjustCapacity(newCap);
    if (flowDelta != 0) {
        // does the orientation matter?
        // positive flow delta means send more flow to the dest
        edge->getDest()->incrementExcess(flowDelta);
        edge->getSource()->decrementExcess(flowDelta);
    }

    return flowDelta;
}
```

Defines:
`adjustCapacity`, used in chunks 112b, 114a, 127b, and 174b.
Uses `decrementExcess` 151d, `dest` 165a, `Edge` 179a, `flow` 167d, `getDest` 165d, `getSource` 165d, `incrementExcess` 151d, and `PhaseSolver` 136c.

To search for an edge, we first see if the caller knows where to look based on the hint. If the hint is null or wrong, we have to search for exhaustively, but iterating over the neighbors of the source node. If we wanted to be extra efficient, we could search on either the source or the destination, depending on which has fewer neighbors.

114a ⟨*search for edge* 114a⟩≡ (113)

```
NodePtr srcNodep = &nodes[srcId];
NodePtr destNodep = &nodes[destId];
if ((hint != nil) && (hint->getSource() == srcNodep) &&
    (hint->getDest() == destNodep))
{
    edge = hint;
} else {
    // just search from the source
    cout << "adjustCapacity: searching for (" << srcId << "," << destId << ")" << endl;
    ElIterator it = srcNodep->getNeighbors();
    for (edge = it.getNext(); edge != nil; edge=it.getNext()) {
        if (edge->getDest() == destNodep) {
            break;
        }
    }
}
```

Uses `adjustCapacity` 113 175a, `getDest` 165d, `getNeighbors` 145a, `getNext` 150g, `getSource` 165d, `nodes` 3a, and `source` 165a.

Once we have adjusted the capacities, we need to renormalize the tree. We want to proceed from the leaves of each branch pushing or pulling excess from the parent until we reach the root. Since we don't have a list of leaf nodes, we will scan all nodes to find roots and perform a post-order traversal of each branch.

After renormalizing the flow in the branches, the labels will be invalid. Therefore, we empty the strong buckets as we iterate from 1 to n, and label all the nodes with distance to deficit when we're done.

114b ⟨*Solver methods* 4a⟩+≡ (136c) ◁111b 116b▷

```
void renormalizeTree(LabelMethod initialLabeling);
```

Uses `renormalizeTree` 115a.

115a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁112a 117▷

```
void PhaseSolver::renormalizeTree(LabelMethod initialLabeling)
{
    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        ⟨reset bucket i 115b⟩
        buckets[i].emptyBucket();
        node.setNextNil();
        if (&node == sourceNode || &node == sinkNode) {
            continue;
        }
        if (node.isRootNode()) {
            // could/should be more efficient with this by recognizing newly
            // created roots that are already normalized
            renormalizeBranch(node);
        }
    }

    ⟨relabel nodes and branches 116a⟩
}
```

Defines:
renormalizeTree, used in chunks 114b and 211.
Uses buckets 23b, emptyBucket 22b, isRootNode 141a, Node 164b, nodes 3a, numNodes 3d, PhaseSolver 136c, renormalizeBranch 117, setNextNil 150e, sinkNode 3f, and sourceNode 3f.

During renormalization, we want to empty all the buckets for strong nodes because strong nodes may become weak, and weak nodes may become strong. It's simple to just empty them all. While we are performing the loop from one to numNodes, can reset/empty the buckets too. Technically, we don't need to loop all the way up to numNodes, but it's simpler this way. Also, we need to remove every node from the bucket it may belong to by reseting its 'next' pointer.

115b ⟨*reset bucket i* 115b⟩≡ (115a)

```
buckets[i].emptyBucket();
node.setNextNil();
```

Uses buckets 23b, emptyBucket 22b, and setNextNil 150e.

After we renormalize all the branches, the node labels are no longer valid. We could try to be tricky by only relabeling the nodes that need it, but instead we just use a brtue-force relabeling like we use during initialization. Distance-to-deficit and constant labels are both appropriate because they make sure the weak roots have label one, and the strong nodes have appropriate labels to merge with their weak neighbors.

We match the relabeling method with the method used to initially label the nodes, to be consistent between the initial solve and the subsequent solving. For example, if constant labeling works well with the instances and deficit distance performs poorly, it would be bad to relabel with deficit distance.

One bug we found was that the distances on the nodes also have to be reset; otherwise, the label will want to 'jump' back up to its previous, higher value. Also note that distance to sink could be bad because after renormalization, we could have the strong nodes on the sink side of the cut and the weak nodes on the source side. As the strong nodes increase label, there could be a gap which would trigger early termination.

116a ⟨*relabel nodes and branches* 116a⟩≡ (115a)

```
if (initialLabeling == LABELS_CONSTANT) {
    setLabelsAndStatus(LABELS_CONSTANT);
} else {
    // distance to sink and deficit are treated the same
    // XXX -sucks: setLabelsAndStatus(LABELS_DEFICIT_DIST);
    setLabelsAndStatus(LABELS_CONSTANT);
}
```

Uses `distance` 156c and `setLabelsAndStatus` 59a.

Renormalizing a branch is where the real work is. As is usually the case, we will iterate over the branch recursively to renormalize all of our children. They will push/pull excess to/from us as needed. Then, we can look at our excess/deficit to decide what we need from our parent.

116b ⟨*Solver methods* 4a⟩+≡ (136c) ◁114b 118b▷

```
void renormalizeBranch(Node& node);
```

Uses `Node` 164b and `renormalizeBranch` 117.

117 ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁115a 119a▷

```
void PhaseSolver::renormalizeBranch(Node& node)
{
    node.resetIterations();
    while (node.hasMoreChildren()) {
        Node& child = node.getCurrentChild();
        renormalizeBranch(child);
        node.advanceChildren();
    }

    ⟨renormalize this node 118a⟩
    node.resetIterations();
}
```

Defines:
`renormalizeBranch`, used in chunks 115a and 116b.
Uses `advanceChildren` 143b, `getCurrentChild` 143b, `hasMoreChildren` 143b, `Node` 164b, `PhaseSolver` 136c, and `resetIterations` 144c.

To renormalize this node, we push excess to our parent or pull some down if have a deficit. This could saturate the link to our parent, in which case we have to split. If we split a strong branch, we need to add it as a strong branch. If this node is already the root of a branch, there is nothing to do.

118a ⟨*renormalize this node* 118a⟩≡ (117)

```
if (node.isRootNode() == false) {
    FlowAmount excess = node.getExcess();
    if (excess > 0) {
        // XXX - should include: STATS(numPushToParent++);
        bool performedSplit = node.pushToParent();
        performedSplit = splitZeroCapArc(node, performedSplit);
        if (performedSplit) {
            STATS(numSplits++);
// if (isStrongNode(node)) {
    // addStrongBranch(node);
// }
        }
    } else if (excess < 0) {
        // XXX - should include: STATS(numPushToParent++);          // actually it's a pull,
        bool performedSplit = node.pullFromParent();
        performedSplit = splitZeroCapArc(node, performedSplit);
        if (performedSplit) {
            STATS(numSplits++);
        }
    }
}
```

Uses `addStrongBranch` 24c 51a, `excess` 151b, `getExcess` 151d, `isRootNode` 141a, `isStrongNode` 17c, `numPushToParent` 129b, `numSplits` 129b, `pullFromParent` 153c, `pushToParent` 152b, and `splitZeroCapArc` 17a.

Here is a solver function that is specially engineered for warm-start. It begins by performing lowest-label, delayed normalization iterations. Once the lowest label reaches 3, it switches to highest label. We re-use the code chunks from `delayNormalizeSolve`.

118b ⟨*Solver methods* 4a⟩+≡ (136c) ◁116b 119b▷

```
void highestLabelResolve(AddBranchPtr addFunc);
```

Uses `AddBranchPtr` 136a and `highestLabelResolve` 119a.

119a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁117 120a▷

```
void PhaseSolver::highestLabelResolve(AddBranchPtr addFunc)
{
    addBranchFunc = addFunc;
    renormalizeFunc = &PhaseSolver::strongOnlyRenormalize;
    initGlobalRelabel();

    int currentPhase = 0;
    NodePtr strongBranch = nil;
    NodeLabel strongLabel =  0;

    while (true) {
        ⟨get strong branch - normalize if needed 29b⟩
        ⟨terminate early based on counting labels 5a⟩
        processBranch(*strongBranch);
        if (strongLabel == 3) {
            break;
        }
    }

    // now, finish it off with highest label
    lowestLabel = 0;            // hack
    highestLabelSolve(addFunc);
}
```

Defines:
`highestLabelResolve`, used in chunks 118b and 211.
Uses `addBranchFunc` 26a, `AddBranchPtr` 136a, `highestLabelSolve` 34, `initGlobalRelabel` 78c, `lowestLabel` 24a, `PhaseSolver` 136c, `processBranch` 7a 47b, `renormalizeFunc` 15c, and `strongOnlyRenormalize` 31a.

2.12 Input Output Functions

We need routines to read Dimacs problem files and write Dimacs flow files. At the moment, these are member functions on the solver. An alternative design would be to put them in a separate class or file to allow us to support multiple input file formats for the same solver class.

Our function to read a problem instance is given a file name and return true if it sucessfully read the problem instance from the specified file.

119b ⟨*Solver methods* 4a⟩+≡ (136c) ◁118b 126d▷

```
bool readDimacsInstance(const char* filename);
```

Uses `readDimacsInstance` 120d.

Our code will be divided into two steps. In the first steps we will determine the basic dimensions of the problem, initialize the edges, and compute the degree of each node. Once we know the degree of the nodes, we can initialize them and add all of their neighbors.

The code below uses C standard I/O (`stdio`) because I'm too lazy to figure out how to use C++ I/O streams for input.

120a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁119a 127a▷

```
bool PhaseSolver::readDimacsInstance(const char* filename)
{
    FILE* fp = fopen(filename, "r");
    if (fp != NULL) {
        instanceFilename = filename;
        ⟨local variables for reading 123b⟩
        ⟨read instance and initialize edges 121a⟩
        ⟨initialize nodes 126a⟩
    } else {
        perror("Unable to read problem instance");
        return false;
    }
    return true;
}
```

Uses `PhaseSolver` 136c and `readDimacsInstance` 120d.

120b ⟨*implementation header files* 120b⟩≡ (135a)

```
#include <stdio.h>
```

120c ⟨*Solver data* 3e⟩+≡ (136c) ◁95b

```
    const char* instanceFilename;
```

120d ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁77d 130a▷

```
    instanceFilename = "<unknown instance>";
```

Defines:
`readDimacsInstance`, used in chunks 119b, 120a, 195, 202, and 209c.

Reading the problem instance is farily straightforward. We read lines until the end of file. Each line in the file is identified by a 'type character' that specifies what type of line we are reading. The type then determines how much more information is on the line. The additional information will begin at the third character in the line.

121a ⟨*read instance and initialize edges* 121a⟩≡ (120a) 121b▷

```
while (!feof(fp)) {
    char buffer[500];
    if (fgets(buffer, 500, fp) == NULL) {
        break;
    }
    char* moreInfo = &buffer[2];
    char type;
    sscanf(buffer, "%c", &type);
    ⟨parse line 122⟩
}
```

We should check that the number of edges that are supposed to be in the file is how many we actually read. This can be a problem if we run out of disk space when generating an instance.

121b ⟨*read instance and initialize edges* 121a⟩+≡ (120a) ◁121a

```
if (nextEdge < numEdges) {
    cerr << "Not enough edges - expected " << numEdges << " found "
         << nextEdge << endl;
    return false;
}
```

Uses `edges` 3a and `numEdges` 3d.

Parsing the type code is a pretty simple switch statment.

122 ⟨*parse line* 122⟩≡ (121a)

```
switch (type) {
    case 'c':       // comment
        break;
    case 'p':       // problem dimensions
        ⟨allocate problem instance 123a⟩
        break;
    case 'n':   {   // specify source or sink
        ⟨specify source/sink 124⟩
        break;
    }
    case 'a':   {   // read an edge
        ⟨read edge 125a⟩
        break;
    }
    default:
        cerr << "Unrecognized input line: " << buffer << endl;
        break;
}
```

Uses source 165a.

The problem instance has a problem type string, number of nodes, and number of instances. In addition to allocating the array of nodes, we also allocate and zero an array of integers that we will use to count the degree of each node as we read each edge. Note that node id's begin at one rather than zero, so we need to allocate an extra node (node zero), that never really gets used.

123a ⟨*allocate problem instance* 123a⟩≡ (122)

```
char typeBuffer[20];
numTokens = sscanf(moreInfo, "%s %d %d", typeBuffer, &numNodes, &numEdges);
if ((numTokens != 3) || (numNodes <= 0) || (numEdges <= 0)) {
    cerr << "Invalid problem instance line: " << buffer << endl;
    return false;
}
nodes = new Node[numNodes + 1];
edges = new Edge[numEdges];
buckets = new NodeBucket[numNodes + 1];
nodeDegrees = new int[numNodes + 1];
labelCount = new int[numNodes + 1];
nodeQ = new NodePtr[numNodes + 1];

for (int i = 0; i <= numNodes; i++) {
    nodeDegrees[i] = labelCount[i] = 0;
    nodeQ[i] = nil;
}
```

Uses `buckets` 23b, `Edge` 179a, `edges` 3a, `labelCount` 5c, `Node` 164b, `nodes` 3a, `numEdges` 3d, and `numNodes` 3d.

123b ⟨*local variables for reading* 123b⟩≡ (120a) 125b▷

```
int numTokens = 0;
int* nodeDegrees = nil;
```

The source and sink in the graph are simply specified by a node number and 's' for the source and 't' for the sink.

124 ⟨*specify source/sink* 124⟩≡ (122)

```
char sourceSinkFlag;
int nodeNumber;
numTokens =  sscanf(moreInfo, "%d %c", &nodeNumber, &sourceSinkFlag);
if ((numTokens != 2) || (nodeNumber <= 0) || (nodeNumber > numNodes)) {
    cerr << "Invalid source/sink line: " << buffer << endl;
    return false;
}

if (sourceSinkFlag == 's') {
    sourceNode = &nodes[nodeNumber];
} else if (sourceSinkFlag == 't') {
    sinkNode = &nodes[nodeNumber];
} else {
    cerr << "Invalid source/sink line: " << buffer << endl;
    return false;
}
```

Uses `nodes` 3a, `numNodes` 3d, `sinkNode` 3f, `source` 165a, and `sourceNode` 3f.

An edge is specified by a source, destination and a capacity. However, when we are performing parametric sensitivity analysis, we have an additional parameter, b, and the capacity is actually interpreted as the a parameter.

125a ⟨*read edge* 125a⟩≡ (122)

```
int source, dest, capacity;
double bParam = 0;
numTokens =  sscanf(moreInfo, "%d %d %d %lf",
                    &source, &dest, &capacity, &bParam);
if ((numTokens < 3) || (source <= 0) || (source > numNodes) ||
    (dest <= 0) || (dest > numNodes)) {
    cerr << "Invalid edge line: " << buffer << endl;
    return false;
}

if (nextEdge < numEdges) {
    if (numTokens == 3) {
        // non-parametric must have non-negative capacity
        if (capacity < 0) {
            cerr << "Invalid edge line: " << buffer << endl;
            return false;
        }
        edges[nextEdge].init(nodes[source], nodes[dest], capacity);
    } else {
        edges[nextEdge].init(nodes[source], nodes[dest], capacity, bParam);
    }
    nextEdge++;
    nodeDegrees[source]++;
    nodeDegrees[dest]++;
} else {
    cerr << "Too many edges - graph should only contain " << numEdges << endl;
    return false;
}
```

Uses `capacity` 167d, `dest` 165a, `edges` 3a, `init` 155a 170e, `nodes` 3a, `numEdges` 3d, `numNodes` 3d, and `source` 165a.

125b ⟨*local variables for reading* 123b⟩+≡ (120a) ◁123b

```
int nextEdge = 0;
```

To initialize the nodes, we call their `init` method to allocate space for neighbors based on the degree of each node that we observered while reading edges. We can trivially assign each node it's id number. We special-case the source and sink nodes to give them larger adjacency lists to support parameteric analysis where nodes can move between the source and sink depending on the value of lambda.

126a ⟨*initialize nodes* 126a⟩≡ (120a) 126b▷

```
// until the nodes are initialized, the nodes pointed at by sourceNode
// and sinkNode don't have ids, so we have to compute the ids by
// pointer arithmetic
int sourceId = sourceNode - nodes;
int sinkId = sinkNode - nodes;
for (int i = 1; i <= numNodes; i++) {
    if ((i == sourceId) || (i == sinkId)) {
    nodes[i].init(i, nodeDegrees[sourceId] + nodeDegrees[sinkId]);
} else {
    nodes[i].init(i, nodeDegrees[i]);
}
}
```

Uses `init` 155a 170e, `nodes` 3a, `numNodes` 3d, `sinkNode` 3f, and `sourceNode` 3f.

Once the nodes are initialized, we can easily iterate over the list of edges and add each endpoint to the other's list of neighbors. Note, we only scan the edges that we actually read (based on `nextEdge`) rather than the number we could have seen (specified by `numEdges`).

126b ⟨*initialize nodes* 126a⟩+≡ (120a) ◁126a 126c▷

```
for (int i = 0; i < nextEdge; i++) {
    edges[i].getHead()->appendNeighbor(edges[i]);
    edges[i].getTail()->appendNeighbor(edges[i]);
}
```

Uses `appendNeighbor` 139d, `edges` 3a, `getHead` 22a 166d, and `getTail` 22a 166d.

Finally, let's remember to free up the memory we allocated to count node degrees.

126c ⟨*initialize nodes* 126a⟩+≡ (120a) ◁126b

```
delete[] nodeDegrees;
```

During general parametric analysis, to adjust the whole graph, we support reading a Dimacs file with the new capacities. If the arcs occur in the same order as we read them initially, then our hint system in `adjustCapacity` will work by just stepping through edge array. This function is just a bastard version of `readDimacsInstance`.

126d ⟨*Solver methods* 4a⟩+≡ (136c) ◁119b 128a▷

```
bool readNewCapacities(const char* filename);
```

Uses `readNewCapacities` 127a.

127a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁120a 128b▷

```
bool PhaseSolver::readNewCapacities(const char* filename)
{
    FILE* fp = fopen(filename, "r");
    if (fp != NULL) {
        int nextEdgeNum = 0;
        while (!feof(fp)) {
            char buffer[500];
            if (fgets(buffer, 500, fp) == NULL) {
                break;
            }
            char type;
            sscanf(buffer, "%c", &type);
            if (type == 'a') {
                ⟨process new capacity 127b⟩
            }
        }
        fclose(fp);
    } else {
        perror("Unable to read new capacities");
        return false;
    }

    return true;
}
```

Defines:
`readNewCapacities`, used in chunks 126d and 211.
Uses `PhaseSolver` 136c.

Processing a new arc capacity is pretty straightforward - just read the line and call `adjustCapacity`.

127b ⟨*process new capacity* 127b⟩≡ (127a)

```
int source, dest, capacity;
int numTokens =  sscanf(&buffer[2], "%d %d %d",
                        &source, &dest, &capacity);
if ((numTokens != 3) || (source <= 0) || (source > numNodes) ||
    (dest <= 0) || (dest > numNodes)) {
    cerr << "Invalid edge line: " << buffer << endl;
    return false;
}

adjustCapacity(&edges[nextEdgeNum], source, dest, capacity);
nextEdgeNum++;
```

Uses `adjustCapacity` 113 175a, `capacity` 167d, `dest` 165a, `edges` 3a, `numNodes` 3d, and `source` 165a.

After we read an instance and solve it, we need to write the result out. The Dimacs flow file is very similar to the instance in that every line is identified by an initial type character followed by arguments.

128a ⟨*Solver methods* 4a⟩+≡ (136c) ◁126d 128d▷
```
void writeDimacsFlow(ostream& dout);
```
Uses writeDimacsFlow 128b 131a.

128b ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁127a 128e▷
```
void PhaseSolver::writeDimacsFlow(ostream& dout)
{
    for (int i = 0; i < numEdges; i++) {
        edges[i].writeFlow(dout) << endl;
    }
}
```
Defines:
writeDimacsFlow, used in chunks 128a, 195, and 202.
Uses edges 3a, numEdges 3d, PhaseSolver 136c, and writeFlow 176d.

128c ⟨*header include files* 128c⟩≡ (135d 164b) 138b▷
```
//class ostream;
```

We collect various statistics while the program is running. This method prints them into the Dimacs output file as comments.

128d ⟨*Solver methods* 4a⟩+≡ (136c) ◁128a 130c▷
```
virtual void writeStats(ostream& dout, bool writeFlow);
```
Uses writeFlow 176d and writeStats 128e.

128e ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁128b 131a▷
```
void PhaseSolver::writeStats(ostream& dout, bool writeFlow)
{
    ⟨write statistics 129a⟩
    if (writeFlow) {
        ⟨write flow amount 130b⟩
    }
}
```
Defines:
writeStats, used in chunks 128d, 195, 202, and 210–12.
Uses PhaseSolver 136c and writeFlow 176d.

129a ⟨*write statistics* 129a⟩≡ (128e)

```
dout << "c  numNodes:      " << numNodes        << endl;
dout << "c  numArcs:       " << numEdges        << endl;
dout << "c  numMergers:    " << numMergers      << endl;
dout << "c  numArcScans:    " << numArcScans      << endl;
dout << "c  numNodeVisits:    " << numNodeVisits      << endl;
dout << "c  numRehangs:    " << numRehangs      << endl;
dout << "c  numPushToParent: " << numPushToParent      << endl;
dout << "c  numSplits:     " << numSplits       << endl;
dout << "c  numRelabels:   " << numRelabels     << endl;
dout << "c  numLabelSkips: " << numLabelSkips   << endl;
dout << "c  numGlobalRelabels:   " << numGlobalRelabels     << endl;
dout << "c  numEmptyBranchScans: " << numEmptyBranchScans << endl;
dout << "c  numRemovedNodes: " << numRemovedNodes << endl;
dout << "c  lowestLabel:   " << lowestLabel     << endl;
```

Uses `lowestLabel` 24a, `numEdges` 3d, `numEmptyBranchScans` 129b, `numGlobalRelabels` 129b, `numLabelSkips` 129b, `numMergers` 129b, `numNodes` 3d, `numPushToParent` 129b, `numRelabels` 129b, `numRemovedNodes` 129b, and `numSplits` 129b.

129b ⟨*Solver protected members* 3a⟩+≡ (136c) ◁112b

```
int numMergers;
int numArcScans;
int numNodeVisits;
int numRehangs;
int numPushToParent;
int numSplits;
int numRelabels;
int numLabelSkips;
int numGlobalRelabels;
int numEmptyBranchScans;
int numRemovedNodes;
```

Defines:
`numArcsScans`, never used.
`numEmptyBranchScans`, used in chunks 4b, 34, 129a, and 130a.
`numGlobalRelabels`, used in chunks 79a, 129a, and 130a.
`numLabelSkips`, used in chunks 13a, 129a, and 130a.
`numMergers`, used in chunks 15a, 37c, 129a, and 130a.
`numPushToParent`, used in chunks 16a, 18, 40, 41, 54a, 118a, 129a, and 130a.
`numRelabels`, used in chunks 13a, 78, 87a, 129a, and 130a.
`numRemovedNodes`, used in chunks 6a, 13a, 129a, and 130a.
`numSplits`, used in chunks 16a, 18, 67a, 118a, 129a, 130a, 195, 198, 199, 202, and 208a.

numNodeVisits numRehangs

130a ⟨*default Solver constructor* 3b⟩+≡ (134c) ◁120d

```
numMergers = numPushToParent = numSplits = numRelabels = numLabelSkips = 0;
numArcScans = numNodeVisits = numGlobalRelabels = numEmptyBranchScans =
    numRehangs = numRemovedNodes = 0;
```

Uses numEmptyBranchScans 129b, numGlobalRelabels 129b, numLabelSkips 129b, numMergers 129b, numPushToParent 129b, numRelabels 129b, numRemovedNodes 129b, and numSplits 129b.

To compute the flow amount, we just scan the neighbors of the source node and sum the flows on those arcs. If we were paranoid, we'd also scan the sink arcs and compare the amounts. We write it as a comment as well as an 's' line.

130b ⟨*write flow amount* 130b⟩≡ (128e)

```
int flowAmt = 0;
for (sourceNode->resetIterations();
     sourceNode->hasMoreNeighbors();
     sourceNode->advanceNeighbors())
{
    Edge& edge = sourceNode->getCurrentNeighbor();
    flowAmt += edge.getFlow();
}
dout << "c  flowValue:   " << flowAmt     << endl;
dout << "c"                                    << endl;
dout << "s " << flowAmt << endl;
```

Uses advanceNeighbors 144a, Edge 179a, getCurrentNeighbor 144a, getFlow 167d, hasMoreNeighbors 144a, resetIterations 144c, and sourceNode 3f.

We also use the methods of this class for data file conversion when converting from parametric files to non-parameteric files. We used to do this with a separate program written in Python, but the floating point semantics are slightly different. Therefore, we can read a parameteric instance and set a value for lambda, then dump it out. This should have identical values for the arc capacities as when we use the parametric solver.

130c ⟨*Solver methods* 4a⟩+≡ (136c) ◁128d 131b▷

```
void writeInstance(ostream& dout);
```

131a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁128e 132a▷

```
void PhaseSolver::writeInstance(ostream& dout)
{
    dout << "p  max  " << numNodes << "  " << numEdges << endl;
    dout << "n  " << sourceNode->getId() << " s" << endl;
    dout << "n  " << sinkNode->getId() << " t" << endl;
    for (int i = 0; i < numEdges; i++) {
        edges[i].writeArcInstance(dout) << endl;
    }
}
```

Defines:
`writeDimacsFlow`, used in chunks 128a, 195, and 202.
Uses `edges` 3a, `numEdges` 3d, `numNodes` 3d, `PhaseSolver` 136c, `sinkNode` 3f, `sourceNode` 3f, and `writeArcInstance` 177a.

Here is a method to print the final disposition of all of the nodes in the graph. This should be called before converting the pseudoflow to a flow.

131b ⟨*Solver methods* 4a⟩+≡ (136c) ◁130c 132b▷

```
    void dumpNodes(ostream& out);
```

Uses `dumpNodes` 132a.

132a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁131a 133a▷

```
void PhaseSolver::dumpNodes(ostream& out)
{
    out << "c" << endl
        << "c  Node Dump" << endl
        << "c Node  id   excess  parent   label  strong/weak" << endl;

    for (int i = 1; i <= numNodes; i++) {
        Node& node = nodes[i];
        out << "c " << setw(8) << node.getId() << " " << setw(8);
        if (&node == sourceNode) {
            out << "source";
        } else if (&node == sinkNode) {
            out << "sink";
        } else {
             out << node.getExcess() << setw(8);
             if (node.isRootNode()) {
                out << "NULL";
             } else {
                out << node.getParentNode()->getId();
             }

             out << setw(8) << node.getLabel();
             out << setw(8) <<
                 (node.isStrong(false) ? "strong" : "weak");
         }
         out << endl;
    }
    out << endl << "c" << endl;
}
```

Defines:
dumpNodes, used in chunks 131b, 195, 198a, 199a, 202, and 212.
Uses excess 151b, getExcess 151d, getLabel 138g, getParentNode 141a, isRootNode 141a, isStrong 162a, Node 164b, nodes 3a, numNodes 3d, PhaseSolver 136c, sinkNode 3f, source 165a, and sourceNode 3f.

2.13 Debug Code

Here is a debug method to find all branches (weak and strong) and check them.

132b ⟨*Solver methods* 4a⟩+≡ (136c) ◁131b 133b▷

```
void checkBranches();
```

Uses checkBranches 133a.

133a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁132a 134a▷

```
void PhaseSolver::checkBranches()
{
    for (int i = 0; i < numNodes; i++) {
        Node& node = nodes[i];
        if ((&node == sourceNode) || (&node == sinkNode)) {
            continue;
        }
        if (node.isRootNode()) {
            if (node.checkTree(nil) == false) {
                cout << endl << "Quitting after one bad branch" << endl;
                break;
            }
        }
    }
}
```

Defines:
`checkBranches`, used in chunks 72b, 79a, 87c, 89c, and 132b.
Uses `checkTree` 158c, `isRootNode` 141a, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, `sinkNode` 3f, and `sourceNode` 3f.

Here is a method to check for mergers between strong nodes and weak nodes. After we complete solving, there should be no mergers available.

133b ⟨*Solver methods* 4a⟩+≡ (136c) ◁132b

```
    void checkMergers();
```

Uses `checkMergers` 134a.

134a ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁133a 134c▷

```
void PhaseSolver::checkMergers()
{
    cout << "Checking for missed mergers." << endl;
    for (int i = 0; i < numNodes; i++) {
        Node& node = nodes[i];
        if ((&node == sourceNode) || (&node == sinkNode)) {
            continue;
        }
        if (node.isStrong(false)) {
            ElIterator it = node.getNeighbors();
            for (EdgePtr edge = it.getNext(); edge != nil; edge=it.getNext()) {
                Node& neighbor = *edge->getOtherNode(&node);
                if ((neighbor.isStrong(false) == false) &&
                    (edge->residCapacity(node) > 0))
                {
                    TRACE(trout << "Found merger from "
                          << node.getId() << " (" << node.getLabel() << ")  to "
                          << neighbor.getId() << " (" << neighbor.getLabel() << ")"
                          << endl);
                }
            }
        }
    }
}
```

Defines:
`checkMergers`, used in chunks 4b, 34, and 133b.
Uses `getLabel` 138g, `getNeighbors` 145a, `getNext` 150g, `getOtherNode` 167a, `isStrong` 162a, `Node` 164b, `nodes` 3a, `numNodes` 3d, `PhaseSolver` 136c, `residCapacity` 169b, `sinkNode` 3f, and `sourceNode` 3f.

2.14 File Boiler Plate

134b ⟨*C++ overhead* 134b⟩≡ (136c 164b 179a 216b) 154c▷

```
PhaseSolver();
```

Uses `PhaseSolver` 136c.

134c ⟨*Solver method implementations* 4b⟩+≡ (135a) ◁134a

```
PhaseSolver::PhaseSolver()
{
    ⟨default Solver constructor 3b⟩
}
```

Uses `PhaseSolver` 136c.

We start with the boiler-plate implementation file.

135a ⟨* 135a⟩≡ 135b▷

```
// $Revision: 1.88 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "PhaseSolver.h"
#include "debug.h"
#include <fstream.h>
#include <iostream>
#include <iomanip>
#include <assert.h>
using namespace std;
⟨implementation header files 120b⟩

⟨Solver method implementations 4b⟩
```

Uses PhaseSolver 136c.

Through magic of the C preprocessor and the macro-like facilities of *noweb*, we can easily define the inline functions out-of-line to allow us to collect better profile information - i.e. collect data on the inline functions that would otherwise not show up in the function call traces of the profiler.

135b ⟨* 135a⟩+≡ ◁135a 135c▷

```
#ifndef INLINE_SOLVER
#define INLINE /*inline*/
⟨Solver inline implementations 9a⟩
#undef INLINE
#endif /*INLINE_SOLVER*/
```

Now, we'll add the simplex solver methods.

135c ⟨* 135a⟩+≡ ◁135b 163c▷

```
⟨Simplex implementations 37c⟩
```

The header defines the PhaseSolver class, its member functions, member data, and inline functions. Again, we have more boiler-plate.

135d ⟨header 135d⟩≡ 136a▷

```
// $Revision: 1.88 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#ifndef SOLVER_H
#define SOLVER_H
#include "types.h"
#include "Node.h"
#include <iostream>
using namespace std;
⟨header include files 128c⟩
```

Uses Node 164b.

We have used some pointer to member function types to select run-time behavior.

136a ⟨*header* 135d⟩+≡ ◁135d 136b▷

```
class PhaseSolver;
typedef void (PhaseSolver::* AddBranchPtr)(Node& root);
typedef void (PhaseSolver::* RenormalizePtr)(Node& strongRoot, Node& weakNode);
```

Defines:
`AddBranchPtr`, used in chunks 4, 25d, 28b, 29a, 33, 34, 118b, 119a, and 198a.
`RenormalizePtr`, used in chunk 15b.
Uses `Node` 164b and `PhaseSolver` 136c.

Then, we need to define the node buckets before we use them.

136b ⟨*header* 135d⟩+≡ ◁136a 136c▷

⟨*Node Bucket definition* 137a⟩

Now, we can finally get to defining the solver.

136c ⟨*header* 135d⟩+≡ ◁136b 136d▷

```
class PhaseSolver
{
public:
    ⟨Solver public declarations 94b⟩
    ⟨Solver methods 4a⟩
    ⟨C++ overhead 134b⟩
    ⟨public Solver data 8b⟩
protected:
    ⟨Solver protected members 3a⟩
private:
    ⟨Solver data 3e⟩
    ⟨Solver private members 15b⟩
};
```

Defines:
`PhaseSolver`, used in chunks 4b, 7–9, 11a, 13–18, 20b, 24–29, 31a, 32a, 34, 36a, 37a, 51a, 56, 59–62, 64a, 65, 67a, 68, 71b, 72b, 75, 78, 79a, 84a, 85, 87, 89c, 93–95, 97a, 98a, 100, 102, 104–106, 112a, 113, 115a, 117, 119a, 120a, 127, 128, 131–36, 194a, 198–200, 202, and 206.

In case we want the inline functions to really be inlined, we also define them here in the header file.

136d ⟨*header* 135d⟩+≡ ◁136c 137b▷

```
#ifdef INLINE_SOLVER
#define INLINE inline
⟨Solver inline implementations 9a⟩
#undef INLINE
#endif /*INLINE_SOLVER*/
#endif /*SOLVER_H*/
```

And finally, let's fill out the definition for Node Buckets.

137a ⟨*Node Bucket definition* 137a⟩≡ (136b)

```
class NodeBucket
{
    public: NodeBucket() { head = tail = nil; }
    ⟨Node Bucket methods 20d⟩
    ⟨Node Bucket data 20c⟩
};
⟨Node Bucket inline implementations 21⟩
```

We can also add the simplex solver class.

137b ⟨*header* 135d⟩+≡ ◁136d 164b▷

```
⟨Simplex declaration 37a⟩
```

3 Nodes in the Graph

Nodes belong to both the original, complete graph as well as the tree maintained by the solver. Therefore, they have neighbors (in the original graph), and a parent and children (in the tree). In this lowest label implementation, they also have labels associated with them. We identify nodes by an id or node number, which remains constant during the execution of the algorithm.

138a ⟨*Node data* 138a⟩≡ (164b) 139a▷
```
NodeId      id;
NodeLabel   label;
```

138b ⟨*header include files* 128c⟩+≡ (135d 164b) ◁128c 139b▷
```
#include "types.h"
```

138c ⟨*default Node constructor* 138c⟩≡ (154d) 140b▷
```
id = -1;  label = -1;
```

138d ⟨*Node methods* 138d⟩≡ (164b) 138f▷
```
public:
    NodeId getId() const;
```

138e ⟨*Node inline implementations* 138e⟩≡ (164) 138g▷
```
INLINE NodeId Node::getId() const
{ return id; }
```
Uses `Node` 164b.

As the algorithm progesses, the we check the label and increment it by one.

138f ⟨*Node methods* 138d⟩+≡ (164b) ◁138d 139c▷
```
public:
    NodeLabel getLabel() const;
    void incrementLabel();
    void setLabel(NodeLabel l);
```
Uses `getLabel` 138g, `incrementLabel` 13b 138g, and `setLabel` 13a 138g.

138g ⟨*Node inline implementations* 138e⟩+≡ (164) ◁138e 141a▷
```
INLINE NodeLabel Node::getLabel() const
{ return label; }
INLINE void Node::incrementLabel()
{ label++; }
INLINE void Node::setLabel(NodeLabel l)
{ label = l; }
```
Defines:
`getLabel`, used in chunks 4b, 6a, 8a, 9a, 11–15, 24–26, 28a, 30, 32b, 34–37, 45a, 47a, 51b, 55a, 62, 64a, 66a, 67a, 69a, 73–76, 79b, 83–87, 92, 107, 108, 132a, 134a, and 138f.
`incrementLabel`, used in chunks 12b, 14a, 87a, and 138f.
`setLabel`, used in chunks 12c, 13b, 61a, 65, 68, 73–76, 84a, 85, 87–90, 92, 112a, and 138f.
Uses `Node` 164b.

To represent the graph, each node needs to store an list of neighbors. To represent the tree, each node has list of children, which allows us to search from the root down the tree.

139a ⟨*Node data* 138a⟩+≡ (164b) ◁138a 140a▷
```
EdgeList    neighbors;
EdgeList    children;
```
Uses `children` 139b, `EdgeList` 193a, and `neighbors` 139b.

139b ⟨*header include files* 128c⟩+≡ (135d 164b) ◁138b 140c▷
```
#include "EdgeList.h"
```
Defines:
`children`, used in chunks 85, 139a, 142–45, 147, 149, 155a, 157, and 160.
`neighbors`, used in chunks 103b, 139, 144, 145a, 155a, and 163a.
Uses `EdgeList` 193a.

We can nodes add to these lists. Adding a neighbor simply adds it to the neighbor list.

139c ⟨*Node methods* 138d⟩+≡ (164b) ◁138f 139e▷
```
void appendNeighbor(Edge& edge);
void addNeighbor(Edge& edge);
```
Uses `addNeighbor` 139d, `appendNeighbor` 139d, and `Edge` 179a.

139d ⟨*Node method implementations* 139d⟩≡ (163c) 139f▷
```
void Node::appendNeighbor(Edge& edge)
{ neighbors.appendEdge(&edge); }
void Node::addNeighbor(Edge& edge)
{ neighbors.addHoleElement(&edge); }
```
Defines:
`addNeighbor`, used in chunks 96, 105, and 139c.
`appendNeighbor`, used in chunks 109a, 126b, and 139c.
Uses `addHoleElement` 187b, `appendEdge` 181c, `Edge` 179a, `neighbors` 139b, and `Node` 164b.

During initialization, we remove arc from/to source- and sink- adjacent nodes so that we don't scan them and try to push flow back along these arcs. *We should/could compact the list after removing the arc.*

139e ⟨*Node methods* 138d⟩+≡ (164b) ◁139c 140d▷
```
void removeNeighbor(Edge& edge);
```
Uses `Edge` 179a and `removeNeighbor` 139f.

139f ⟨*Node method implementations* 139d⟩+≡ (163c) ◁139d 142a▷
```
void Node::removeNeighbor(Edge& edge)
{ neighbors.removeElement(&edge); }
```
Defines:
`removeNeighbor`, used in chunks 58a, 102, 104a, 105, 109a, 110, and 139e.
Uses `Edge` 179a, `neighbors` 139b, `Node` 164b, and `removeElement` 189b.

In order to scan from a node back up to the root, each node needs to have a pointer to its parent. During push operations, we also need to know the residual capacity of the edge between the parent and child, so we need to keep a pointer to the edge between the parent and the child. We can get the parent node from this edge, but profiling revealed this as a hot spot. Therefore, we also store a pointer to the parent node in addition to storing the edge.

140a ⟨*Node data* 138a⟩+≡ (164b) ◁139a 150a▷

```
EdgePtr     parentEdge;
NodePtr     parentNode;
```

Uses `parentEdge` 140c and `parentNode` 140c.

140b ⟨*default Node constructor* 138c⟩+≡ (154d) ◁138c 150c▷

```
parentEdge = nil;
parentNode = nil;
```

Uses `parentEdge` 140c and `parentNode` 140c.

140c ⟨*header include files* 128c⟩+≡ (135d 164b) ◁139b 215b▷

```
#include "Edge.h"
```

Defines:
`parentEdge`, used in chunks 53a, 54a, 140–42, 145c, 147, 149, 152b, 153c, 159, and 161c.
`parentNode`, used in chunks 53a, 54a, 140–42, 145c, 147, 149, 152–54, and 159–61.
Uses `Edge` 179a.

We can define some accessor methods to use the parent edge. The parent edge is `nil` if the node is a root.

140d ⟨*Node methods* 138d⟩+≡ (164b) ◁139e 141b▷

```
EdgePtr     getParentEdge() const;
NodePtr     getParentNode();
bool        isRootNode() const;
FlowAmount  getParentCapacity() const;
FlowAmount  getParentDownCapacity() const;
```

Uses `getParentCapacity` 141a, `getParentDownCapacity` 141a, `getParentEdge` 141a, `getParentNode` 141a, and `isRootNode` 141a.

141a ⟨*Node inline implementations* 138e⟩+≡ (164) ◁138g 143b▷

```
INLINE EdgePtr Node::getParentEdge() const
{ return parentEdge; }
INLINE NodePtr Node::getParentNode()
{ return parentNode; }
INLINE bool Node::isRootNode() const
{ return (parentEdge == nil) ? true : false; }
INLINE FlowAmount Node::getParentCapacity() const
{ return parentEdge->residCapacity(); }
INLINE FlowAmount Node::getParentDownCapacity() const
{ return parentEdge->residCapacity(*parentNode); }
```

Defines:
`getParentCapacity`, used in chunks 17a, 74a, and 140d.
`getParentDownCapacity`, used in chunks 39b and 140d.
`getParentEdge`, used in chunks 45a, 53a, 54a, 87a, 98b, 99b, 140d, 147, 149, and 157.
`getParentNode`, used in chunks 16a, 18, 24c, 25a, 41b, 44b, 50a, 51a, 55a, 98b, 99b, 132a, 140d, 145c, 147, 149, 156a, 157, and 161c.
`isRootNode`, used in chunks 46a, 60a, 82, 92, 98b, 99b, 115a, 118a, 132a, 133a, 140d, and 157.
Uses `Node` 164b, `parentEdge` 140c, `parentNode` 140c, and `residCapacity` 169b.

Adding a child is a more involved process than adding a neighbor because we have to update the parent edge. We begin by adding the child to the parent's list of children. Then, we update the orientation of the orientation of the edge to point from child to parent. And finally, we update set the child's parent edge.

When we add the child, we are free to insert it into the edge list in a hole rather than appending it to the list. This is due to the fact that this method is only called when we are joining a strong node with label ℓ to a weak node with label $\ell - 1$, which implies that the weak node is still in phase $\ell - 1$. This in turn means that the weak node does not need to look at the new child during its current phase, so we are safe to insert it behind the current cursor - i.e. in a hole if one exists.

141b ⟨*Node methods* 138d⟩+≡ (164b) ◁140d 142b▷

```
void addChild(Node& child, Edge& edge);
```

Uses `addChild` 142a, `Edge` 179a, and `Node` 164b.

142a ⟨*Node method implementations* 139d⟩+≡ (163c) ◁139f 142c▷

```
void Node::addChild(Node& child, Edge& edge)
{
    assert(edge.validateEdge(*this, child));
    assert(child.label > label);
    children.addHoleElement(&edge);
    edge.setDirectionTo(this);
    child.parentEdge = &edge;
    child.parentNode = this;
}
```

Defines:
addChild, used in chunks 15a, 38b, 42b, 65, 68, 74a, and 141b.
Uses addHoleElement 187b, children 139b, Edge 179a, Node 164b, parentEdge 140c, parentNode 140c, and validateEdge 177c.

To support the simplex iterations, we also need to be able to add a degenerate sub-tree as a child. In this case, its label will be less than or equal to ours, and we need to add it after the current child index to have it scan when we process this node next.

142b ⟨*Node methods* 138d⟩+≡ (164b) ◁141b 143a▷

```
    void addWeakChild(Node& child, Edge& edge);
```

Uses addWeakChild 142c, Edge 179a, and Node 164b.

142c ⟨*Node method implementations* 139d⟩+≡ (163c) ◁142a 145c▷

```
void Node::addWeakChild(Node& child, Edge& edge)
{
    assert(edge.validateEdge(*this, child));
    assert(child.label <= label);
    children.addAfterCurrent(&edge);
    edge.setDirectionTo(this);
    child.parentEdge = &edge;
    child.parentNode = this;
}
```

Defines:
addWeakChild, used in chunks 41d and 142b.
Uses addAfterCurrent 188a, children 139b, Edge 179a, Node 164b, parentEdge 140c, parentNode 140c, and validateEdge 177c.

We need methods to iterate over the neighbors and children. These are basically simple pass-through functions for the underlying lists of edges. First, iterating over children:

143a ⟨*Node methods* 138d⟩+≡ (164b) ◁142b 143c▷

```
void        beginChildren();
bool        hasMoreChildren();
Node&       getCurrentChild();
void        advanceChildren();
int         getNumChildren();
```

Uses `advanceChildren` 143b, `beginChildren` 143b, `getCurrentChild` 143b, `hasMoreChildren` 143b, and `Node` 164b.

143b ⟨*Node inline implementations* 138e⟩+≡ (164) ◁141a 144a▷

```
INLINE void Node::beginChildren()
{ children.beginIteration(); }
INLINE bool Node::hasMoreChildren()
{ return children.hasMoreElements(); }
INLINE Node& Node::getCurrentChild()
{ return *(children.getCurrentElement()->getOtherNode(this)); }
INLINE void Node:: advanceChildren()
{ children.advanceToNextElement(); }
INLINE int Node:: getNumChildren()
{ return children.getSize(); }
```

Defines:
`advanceChildren`, used in chunks 9a, 14a, 61a, 112a, 117, and 143a.
`beginChildren`, used in chunk 143a.
`getCurrentChild`, used in chunks 9a, 14a, 61a, 112a, 117, and 143a.
`hasMoreChildren`, used in chunks 9a, 14a, 61a, 112a, 117, and 143a.
Uses `advanceToNextElement` 185c, `beginIteration` 184c, `children` 139b, `getCurrentElement` 185b, `getOtherNode` 167a, `hasMoreElements` 185a, and `Node` 164b.

And we have basically the same code for iterating over neighbors:

143c ⟨*Node methods* 138d⟩+≡ (164b) ◁143a 144b▷

```
void        beginNeighbors();
bool        hasMoreNeighbors();
Edge&       getCurrentNeighbor();
void        advanceNeighbors();
```

Uses `advanceNeighbors` 144a, `beginNeighbors` 144a, `Edge` 179a, `getCurrentNeighbor` 144a, and `hasMoreNeighbors` 144a.

144a ⟨*Node inline implementations* 138e⟩+≡ (164) ◁143b 144c▷

```
INLINE void Node::beginNeighbors()
{ neighbors.beginIteration(); }
INLINE bool Node::hasMoreNeighbors()
{ return neighbors.hasMoreElements(); }
INLINE Edge& Node::getCurrentNeighbor()
{ return *neighbors.getCurrentElement(); }
INLINE void Node:: advanceNeighbors()
{ neighbors.advanceToNextElement(); }
```

Defines:
`advanceNeighbors`, used in chunks 11a, 56c, 64a, 66a, 67a, 69a, 72b, 76a, 80a, 88b, 96–98, 102, 105, 107, 108, 110, 111a, 130b, and 143c.
`beginNeighbors`, used in chunk 143c.
`getCurrentNeighbor`, used in chunks 11a, 56c, 64a, 66a, 67a, 69a, 73b, 76a, 80a, 88b, 96–98, 102, 105, 107, 108, 110, 111a, 130b, and 143c.
`hasMoreNeighbors`, used in chunks 11a, 56c, 64a, 66a, 67a, 69a, 72b, 76a, 80a, 88b, 96–98, 100, 102, 105, 107, 108, 110, 111a, 130b, and 143c.
Uses `advanceToNextElement` 185c, `beginIteration` 184c, `Edge` 179a, `getCurrentElement` 185b, `hasMoreElements` 185a, `neighbors` 139b, and `Node` 164b.

Finally, before we begin processing a node, we need to reset the iterations on both the children and the neighbors.

144b ⟨*Node methods* 138d⟩+≡ (164b) ◁143c 144d▷

```
void resetIterations();
```

Uses `resetIterations` 144c.

144c ⟨*Node inline implementations* 138e⟩+≡ (164) ◁144a 145a▷

```
INLINE void Node::resetIterations()
{ children.beginIteration(); neighbors.beginIteration(); }
```

Defines:
`resetIterations`, used in chunks 13a, 14a, 56c, 60a, 61a, 64a, 66a, 67a, 69a, 72b, 75, 76a, 80a, 88b, 92, 96–98, 102, 105, 107, 108, 110–12, 117, 130b, and 144b.
Uses `beginIteration` 184c, `children` 139b, `neighbors` 139b, and `Node` 164b.

We have a newer way of iterating over the neighbors and children of a node that uses a read-only iterator. This allows us to scan the neighbors/children without updating the current neighbor pointer.

144d ⟨*Node methods* 138d⟩+≡ (164b) ◁144b 145b▷

```
ElIterator  getNeighbors();
ElIterator  getChildren();
```

Uses `getChildren` 145a and `getNeighbors` 145a.

145a ⟨*Node inline implementations* 138e⟩+≡ (164) ◁144c 150e▷

```
INLINE ElIterator Node::getNeighbors()
{
    ElIterator it(neighbors);
    return it;          // lame object copy during return
}
INLINE ElIterator Node::getChildren()
{
    ElIterator it(children);
    return it;          // lame object copy during return
}
```

Defines:
`getChildren`, used in chunks 84a, 85, and 144d.
`getNeighbors`, used in chunks 12a, 73d, 80b, 88c, 90c, 114a, 134a, and 144d.
Uses `children` 139b, `neighbors` 139b, and `Node` 164b.

The opposite of adding a child is spliting off a sub-tree by removing a node from its parent. This is quite straightforward: remove the parent edge of the node from its parent's list of children, and null out the parent edge.

145b ⟨*Node methods* 138d⟩+≡ (164b) ◁144d 146▷

```
void split();
```

Uses `split` 145c.

145c ⟨*Node method implementations* 139d⟩+≡ (163c) ◁142c 147▷

```
void Node::split()
{
    assert(parentEdge != nil);
    TRACE( trout << "split " << id << " from "
                 << getParentNode()->id << " with excess "
                 << excess << endl;)
    getParentNode()->children.removeElement(parentEdge);
    parentEdge->toggleOrientation();
    parentEdge = nil;
    parentNode = nil;
}
```

Defines:
`split`, used in chunks 17a, 39b, 41, 68, 74a, 107, 108, 145b, 153a, and 154b.
Uses `children` 139b, `excess` 151b, `getParentNode` 141a, `Node` 164b, `parentEdge` 140c, `parentNode` 140c, and `removeElement` 189b.

Rehanging a tree from a node involves reversing the parent-child relationship between nodes along the path from the new root node up to the existing root.

At the center of the parent-child relationship is the edge between the parent and child. This is the `parentEdge` in the current node (`this`), and it is stored the `children` list in the parent node. We remove the edge from our parent's list of children and recursively tell the parent to rehang itself. Once the tree is rehung by the parent, we install as a sub-tree under our node by by setting the parent's `parentEdge` pointer to the edge and adding the edge to our children list. Since we reversed the orientation of the two nodes, we need to toggle the direction of the edge. Finally, the current node is the root of a sub-tree, so its parent edge is set to `nil`.

Because this procedure must ultimately scan up to the root of the branch, and we don't have any easy way to compute the root, we return the root to the caller so that it knows which is the root with excess to push.

As indicated, the code is written as a recursive function because it is easiest to code that way. In the future, if the function call overhead becomes a bottleneck, we can recode it iteratively.

In this code, we will take advantage of the fact that we know we must have been iterating through the children arrays of the nodes along the path to get to a node that we are going to rehang by. For a given node, v, we know that v must be the current node in the children array of the parent of v. This makes it simple and fast to find v in its parent's children array.

146 ⟨*Node methods* 138d⟩+≡ (164b) ◁145b 148▷

```
NodePtr rehang(STATS(int& numRehangs));
```

Uses `rehang` 147.

147 ⟨*Node method implementations* 139d⟩+≡ (163c) ◁145c 149▷

```
NodePtr Node::rehang(STATS(int& numRehangs))
{
    NodePtr oldRoot = nil;

    if (parentEdge != nil) {
        STATS(numRehangs++);
        NodePtr currNode = this;
        EdgePtr edge = parentEdge;
        NodePtr parNode = getParentNode();
        parentEdge = nil;
        parentNode = nil;

        while (parNode != nil) {
            STATS(numRehangs++);
            // assert(currNode->label == parNode->label);
            // XXX - broke this assertion for simplex
            assert(currNode->label <= parNode->label);
            assert(parNode != currNode);
            EdgePtr nextEdge = parNode->getParentEdge();

            parNode->children.removeCurrentElement(edge);
            parNode->parentEdge = edge;
            parNode->parentNode = currNode;
            currNode->children.addCurrentElement(edge);
            edge->toggleOrientation();

            if (nextEdge != nil) {
                currNode = parNode;
                edge = nextEdge;
                parNode = edge->getOtherNode(currNode);
            } else {
                oldRoot = parNode;
                break;
            }
        }
    } else {
        oldRoot = this;
    }

    return oldRoot;
}
```

Defines:
rehang, used in chunks 15a and 146.

Uses addCurrentElement 186c, children 139b, getOtherNode 167a, getParentEdge 141a, getParentNode 141a, Node 164b, parentEdge 140c, parentNode 140c, and removeCurrentElement 186a.

The code to rehang a branch above assumes that we are rehanging a strong branch. This is usually the case for the pseduoflow algorithm, but the simplex variant also need to rehang a weak branch. Therefore, we have a nearly identical version of the method except that it manipulates the `children` arrays differently because we cannot assume that we are inverting along the path defined by the current element cursors. Also, with simplex and rehanging weak branches, we can't make many assumptions about the labels of the nodes involved, so we don't assert the labels.

148 ⟨*Node methods* 138d⟩+≡ (164b) ◁146 150b▷

```
NodePtr rehangWeak();
```

Uses rehangWeak 149.

149 ⟨*Node method implementations* 139d⟩+≡ (163c) ◁147 154d▷

```
NodePtr Node::rehangWeak()
{
    NodePtr oldRoot = nil;

    if (parentEdge != nil) {
        NodePtr currNode = this;
        EdgePtr edge = parentEdge;
        NodePtr parNode = getParentNode();
        parentEdge = nil;
        parentNode = nil;

        while (parNode != nil) {
            //assert(currNode->label >= parNode->label);
            assert(parNode != currNode);
            EdgePtr nextEdge = parNode->getParentEdge();

            parNode->children.removeElement(edge);
            parNode->parentEdge = edge;
            parNode->parentNode = currNode;
            // maybe just addHole & reset??
            currNode->children.addAfterCurrent(edge);
            edge->toggleOrientation();

            if (nextEdge != nil) {
                currNode = parNode;
                edge = nextEdge;
                parNode = edge->getOtherNode(currNode);
            } else {
                oldRoot = parNode;
                break;
            }
        }
    } else {
        oldRoot = this;
    }

    return oldRoot;
}
```

Defines:
`rehangWeak`, used in chunks 38b, 41d, 42b, and 148.
Uses `addAfterCurrent` 188a, `children` 139b, `getOtherNode` 167a, `getParentEdge` 141a, `getParentNode` 141a, `Node` 164b, `parentEdge` 140c, `parentNode` 140c, and `removeElement` 189b.

In addition to the tree structure and a Node's neighbors, we need to build a linked list of nodes for use by the solver. This is a simple, singly-linked list with one pointer to the next element in the list.

150a ⟨*Node data* 138a⟩+≡ (164b) ◁140a 151a▷
```
NodePtr     nextNode;
```
Uses nextNode 150c.

150b ⟨*Node methods* 138d⟩+≡ (164b) ◁148 150d▷
```
NodePtr     next2;
```
Uses next2 150c.

150c ⟨*default Node constructor* 138c⟩+≡ (154d) ◁140b 151b▷
```
nextNode = nil;
next2 = nil;
```
Defines:
  next2, used in chunks 31a, 32a, and 150b.
  nextNode, used in chunk 150.

We need to be able to set it to nil or another value.

150d ⟨*Node methods* 138d⟩+≡ (164b) ◁150b 150f▷
```
void setNextNil();
void setNext(NodePtr next);
```
Uses setNext 150e and setNextNil 150e.

150e ⟨*Node inline implementations* 138e⟩+≡ (164) ◁145a 150g▷
```
INLINE void Node::setNextNil()
{ nextNode = nil; }
INLINE void Node::setNext(NodePtr next)
{ nextNode = next; }
```
Defines:
  setNext, used in chunks 21, 22d, and 150d.
  setNextNil, used in chunks 21, 22d, 27d, 36b, 115, and 150d.
Uses nextNode 150c and Node 164b.

Finally, we need a way to get the next element in the list.

150f ⟨*Node methods* 138d⟩+≡ (164b) ◁150d 151c▷
```
NodePtr getNext();
```
Uses getNext 150g.

150g ⟨*Node inline implementations* 138e⟩+≡ (164) ◁150e 151d▷
```
INLINE NodePtr Node::getNext()
{ return nextNode; }
```
Defines:
  getNext, used in chunks 12a, 21–23, 25a, 45a, 51b, 55a, 73d, 80b, 84a, 85, 88c, 90c, 109c, 114a, 134a, 150f, 163a, and 191.
Uses nextNode 150c and Node 164b.

A node can also have excess flow associated with it. Typically, this only occurs at a node that is adjacent to the pseduo-root of the tree. In our implementation, we store excess on nodes in the middle of the tree temporarily while we are pushing it towards the pseudo-root.

151a ⟨*Node data* 138a⟩+≡ (164b) ◁150a 155b▷

```
NodeExcess  excess;
```

Uses excess 151b.

151b ⟨*default Node constructor* 138c⟩+≡ (154d) ◁150c 155c▷

```
excess = 0;
```

Defines:
excess, used in chunks 17c, 38, 39a, 51a, 54a, 58a, 65, 66a, 68–71, 97a, 100, 118a, 132a, 145c, 151–54, 157, 161, and 162.

Of course, then we need methods to manipulate the excess.

151c ⟨*Node methods* 138d⟩+≡ (164b) ◁150f 152a▷

```
NodeExcess getExcess() const;
void incrementExcess(NodeExcess delta);
void decrementExcess(NodeExcess delta);
void setExcess(NodeExcess newExcess);
```

Uses decrementExcess 151d, getExcess 151d, incrementExcess 151d, and setExcess 151d.

151d ⟨*Node inline implementations* 138e⟩+≡ (164) ◁150g 152b▷

```
INLINE NodeExcess Node::getExcess() const
{ return excess; }
INLINE void Node::incrementExcess(NodeExcess delta)
{ excess += delta; }
INLINE void Node::decrementExcess(NodeExcess delta)
{ excess -= delta; }
INLINE void Node::setExcess(NodeExcess newExcess)
{ excess = newExcess; }
```

Defines:
decrementExcess, used in chunks 40c, 41a, 54b, 58a, 65, 71b, 99b, 100, 113, and 151c.
getExcess, used in chunks 15a, 17c, 24c, 32b, 37–41, 51a, 54b, 60a, 62, 64–68, 70, 73–76, 79b, 90a, 92, 95a, 97a, 99, 100, 118a, 132a, and 151c.
incrementExcess, used in chunks 40c, 41a, 54b, 58a, 65, 71b, 97a, 100, 107, 108, 113, and 151c.
setExcess, used in chunk 151c.
Uses excess 151b and Node 164b.

When a non-root node has excess, we need to push excess as much as possible from the node towards the root. We do this on a node-by-node basis - i.e. we just push the flow to our parent and let him worry about it. If we cannot push all of our excess to our parent, we return true, otherwise we return false. Note that if the amount of excess exactly matches the residual capacity, we will not split, but the calling function can choose to split us if he sees no remaining residual capacity.

152a ⟨*Node methods* 138d⟩+≡ (164b) ◁151c 153b▷

```
bool pushToParent();
```

Uses pushToParent 152b.

152b ⟨*Node inline implementations* 138e⟩+≡ (164) ◁151d 153c▷

```
INLINE bool Node::pushToParent()
{
    assert(parentEdge != nil);
    Edge& edge = *parentEdge;
    //assert(parentNode->excess <= 0);
    assert(edge.residCapacity() >= 0);

    if (edge.residCapacity() >= excess) {
        ⟨push without splitting 152c⟩
        return false;
    } else {
        ⟨push and split 153a⟩
        return true;
    }
}
```

Defines:
pushToParent, used in chunks 16a, 18, 41b, 68, 118a, and 152a.
Uses Edge 179a, excess 151b, Node 164b, parentEdge 140c, parentNode 140c, and residCapacity 169b.

Pushing when we have enough capacity, is pretty simple. Our parent gets all of our excess, our excess becomes zero, and the flow amount on the edge increases.

152c ⟨*push without splitting* 152c⟩≡ (152b)

```
FlowAmount flow = excess;
parentNode->excess += flow;
excess = 0;
edge.increaseFlow(flow);
```

Uses excess 151b, flow 167d, increaseFlow 168a, and parentNode 140c.

Pushing with insufficient capacity is similar, but we can only push as much flow as there is residual capacity on the arc, and we split the current node when done.

153a ⟨*push and split* 153a⟩≡ (152b)

```
FlowAmount flow = edge.residCapacity();
parentNode->excess += flow;
excess -= flow;
edge.increaseFlow(flow);
split();
```

Uses `excess` 151b, `flow` 167d, `increaseFlow` 168a, `parentNode` 140c, `residCapacity` 169b, and `split` 145c.

During normal operations, we only have to push to our parent. However, when we renormalize during general parametric analysis, we need to pull excess from our parent.

153b ⟨*Node methods* 138d⟩+≡ (164b) ◁152a 154e▷

```
bool pullFromParent();
```

Uses `pullFromParent` 153c.

153c ⟨*Node inline implementations* 138e⟩+≡ (164) ◁152b 156a▷

```
INLINE bool Node::pullFromParent()
{
    assert((parentEdge != nil) && (parentNode != nil));
    Edge& edge = *parentEdge;
    FlowAmount residCap = edge.residCapacity(*parentNode);
    assert(residCap >= 0);
    FlowAmount deficit = -excess;
    assert(deficit > 0);

    if (residCap >= deficit) {
        ⟨pull without splitting 154a⟩
        return false;
    } else {
        ⟨pull and split 154b⟩
        return true;
    }
}
```

Defines:
`pullFromParent`, used in chunks 118a and 153b.
Uses `Edge` 179a, `excess` 151b, `Node` 164b, `parentEdge` 140c, `parentNode` 140c, and `residCapacity` 169b.

Pulling when we have enough capacity, is pretty simple. Our parent gets all of our defict, our excess becomes zero, and the flow amount on the edge decreases.

154a ⟨*pull without splitting* 154a⟩≡ (153c)
```
FlowAmount flow = deficit;
parentNode->excess -= flow;
excess = 0;
edge.decreaseFlow(flow);
```
Uses `decreaseFlow` 168c, `excess` 151b, `flow` 167d, and `parentNode` 140c.

Pulling with insufficient capacity is similar, but we can only pull as much flow as there is residual capacity on the arc, and we split the current node when done.

154b ⟨*pull and split* 154b⟩≡ (153c)
```
FlowAmount flow = residCap;
parentNode->excess -= flow;
excess += flow;
edge.decreaseFlow(flow);
split();
```
Uses `decreaseFlow` 168c, `excess` 151b, `flow` 167d, `parentNode` 140c, and `split` 145c.

We need a default constructor because we will be creating an array of nodes in bulk when we know how many of them are in the graph.

154c ⟨*C++ overhead* 134b⟩+≡ (136c 164b 179a 216b) ◁134b 163b▷
```
Node();
```
Uses `Node` 164b.

154d ⟨*Node method implementations* 139d⟩+≡ (163c) ◁149 155a▷
```
Node::Node()
{
    ⟨default Node constructor 138c⟩
}
```
Uses `Node` 164b.

After constructing a node, we will need to initialize it. By the time we initialize it, we need to know the number of neighbors so we can allocate the neighbor and children arrays.

154e ⟨*Node methods* 138d⟩+≡ (164b) ◁153b 155d▷
```
void init(NodeId id, int numNeighbors);
```
Uses `init` 155a 170e.

155a ⟨*Node method implementations* 139d⟩+≡ (163c) ◁154d 157▷

```
void Node::init(NodeId id, int numNeighbors)
{
    this->id = id;
    label = INITIAL_LABEL;
    neighbors.init(numNeighbors);
    children.init(numNeighbors + 3);
}
```

Defines:
`init`, used in chunks 125a, 126a, 154e, 170, 171, 180, 181a, 195, 198b, 202, and 207.
Uses `children` 139b, `neighbors` 139b, and `Node` 164b.

155b ⟨*Node data* 138a⟩+≡ (164b) ◁151a 156b▷

```
public:
    static const int INITIAL_LABEL = 0;
    int flag;
```

Uses `flag` 155d.

155c ⟨*default Node constructor* 138c⟩+≡ (154d) ◁151b 156c▷

```
    flag = 0;
```

Uses `flag` 155d.

155d ⟨*Node methods* 138d⟩+≡ (164b) ◁154e

```
    void setFlag(int f) { flag |= f; }
    void clearFlag(int f) { flag &= ~f; }
    bool flagIsSet(int f) { return (flag & f) ? true : false; }
    void clearFlagPath(int f, NodePtr end);
```

Defines:
`clearFlag`, used in chunks 18, 47a, and 156a.
`flag`, used in chunks 32b, 45a, 102, 103a, 155, 157, and 161a.
`flagIsSet`, used in chunks 31a, 46a, 47a, 50a, 51a, 55a, and 110.
`setFlag`, used in chunks 31a, 45a, and 58a.
Uses `clearFlagPath` 156a.

This function clear a flag value along a path of parent pointers.

156a ⟨*Node inline implementations* 138e⟩+≡ (164) ◁153c

```
INLINE void Node::clearFlagPath(int f, NodePtr end)
{
    NodePtr currNode = this;
    while (currNode != end) {
        assert(currNode != nil);
        currNode->clearFlag(f);
        currNode = currNode->getParentNode();
    }
    if (end != nil) {
        end->clearFlag(f);
    }
}
```

Defines:
`clearFlagPath`, used in chunks 48 and 155d.
Uses `clearFlag` 155d, `getParentNode` 141a, and `Node` 164b.

156b ⟨*Node data* 138a⟩+≡ (164b) ◁155b

```
    int distance;
    int minChildLabel;
```

Uses `distance` 156c and `minChildLabel` 156c.

156c ⟨*default Node constructor* 138c⟩+≡ (154d) ◁155c

```
    distance = minChildLabel = 0;
```

Defines:
`distance`, used in chunks 11a, 13a, 61a, 74–76, 79–83, 85–92, 116a, 156b, and 157.
`minChildLabel`, used in chunks 9a, 13a, 61a, 85, 87a, and 156b.

3.1 Debugging

To aid in debugging the tree and graph manipulation functions, we have some functions that we can use to print and check the data structures.

156d ⟨*debugging functions* 156d⟩≡ (164b) 158b▷

```
    public: void printTree(int level);
```

Uses `printTree` 157.

157 ⟨*Node method implementations* 139d⟩+≡ (163c) ◁155a 158a▷

```
void Node::printTree(int level)
{
    if (level == 0) {
        cout << " Node id   address   level   label    dist  parent res-cap "
             << " excess   flags" << endl;
    }

    cout << setw(8) << id << "   "
         << setw(8) << this
         << setw(8) << level
         << setw(8) << label
         << setw(8) << distance;

    cout << setw(8);
    if (isRootNode()) {
        cout << "NULL";
        cout << setw(8) << 0;
    } else {
        cout << setw(8) << getParentNode()->id;
        cout << setw(8) << getParentEdge()->residCapacity(*this);
    }

    cout << setw(8) << excess << setw(8) << flag << endl;

    // examine children
    int numChildren = children.getSize();
    int childLevel = level + 1;
    for (int i = 0; i < numChildren; i++) {
        EdgePtr childEdge = children.getElement(i);
        if (childEdge != nil) {
            NodePtr child = childEdge->getOtherNode(this);
            child->printTree(childLevel);
        }
    }
}
```

Defines:
`printTree`, used in chunks 156d and 158a.
Uses `children` 139b, `distance` 156c, `excess` 151b, `flag` 155d, `getOtherNode` 167a, `getParentEdge` 141a, `getParentNode` 141a, `isRootNode` 141a, `Node` 164b, and `residCapacity` 169b.

Here's a 'C' version of the function that we can call from *gdb*.

158a ⟨*Node method implementations* 139d⟩+≡ (163c) ◁157 158c▷
```
void printTree(Node* n)
{
    n->printTree(0);
}
```
Uses Node 164b and printTree 157.

Sometimes, we also need to check the consistency of the tree, especially after operations like rehanging the tree. The code recursively explores the tree in pre-order to validate the pointers from a child node back to its parent and vice versa. For each node, we check its parent edge pointer then investigate the children. As soon as we detect any error in the tree, we stop immediately, since there may be zillions of errors, which would swamp the output that we look at.

158b ⟨*debugging functions* 156d⟩+≡ (164b) ◁156d 161b▷
```
    bool checkTree(EdgePtr expectedParentEdge);
```
Uses checkTree 158c.

158c ⟨*Node method implementations* 139d⟩+≡ (163c) ◁158a 161c▷
```
bool Node::checkTree(EdgePtr expectedParentEdge)
{
    bool treeOK = true;
    ⟨check parent pointers 159⟩

    if (treeOK) {
        ⟨check child pointers 160b⟩
    }
    ⟨check label and excess 161a⟩

    return treeOK;
}
```
Defines:
checkTree, used in chunks 133a, 158b, 160b, 194b, 198b, and 199a.
Uses Node 164b.

First we check our pointer to our parent. This could be `nil` if we are a root, otherwise it better bet the same edge that our parent tells us it should be. Furthermore, the edge should point from us (the tail) up to our parent.

159 ⟨*check parent pointers* 159⟩≡ (158c)

```
if (expectedParentEdge != nil) {
    if (parentEdge != expectedParentEdge) {
        cout << "\nError: node " << id << " parent edge wrong. "
             << " expected " << expectedParentEdge << endl
             << " \tfound " << parentEdge << endl;
        treeOK = false;
    } else {
        if (parentEdge->getTail() != this) {
            cout << "\nError: node " << id
                 << " parent edge orientation wrong: "
                 << *parentEdge << endl;
            treeOK = false;
        } else {
            if (parentEdge->getHead() != parentNode) {
                cout << "\nError: node " << id
                     << " parent edge does not match parentNode" << endl;
                treeOK = false;
            }
        }
    }
    ⟨check parent's children 160a⟩
} else {
    if (parentEdge != nil) {
            cout << "\nError: node " << id
                 << " expected nil parent edge, found: "
                 << *parentEdge << endl;
            treeOK = false;
    }
}
```

Uses `getHead` 22a 166d, `getTail` 22a 166d, `parentEdge` 140c, and `parentNode` 140c.

160a ⟨*check parent's children* 160a⟩≡ (159)

```
if (treeOK) {
    bool foundThis = false;
    int numChildren = parentNode->children.getSize();
    for (int i = 0; i < numChildren; i++) {
        EdgePtr childEdge = parentNode->children.getElement(i);
        if (childEdge != nil) {
            NodePtr childNode = childEdge->getOtherNode(parentNode);
            if (childNode == this) {
                foundThis = true;
                break;
            }
        }
    }
    if (foundThis == false) {
        cout << "\nError: node " << id << " can't find outself in "
             << "parent node " << parentNode->id << endl;
    }
}
```

Uses `children` 139b, `getOtherNode` 167a, and `parentNode` 140c.

To investigate the children, we just iterate over our list of children and check each of them in turn. We must skip holes in the list (denoted by `nil` entries).

160b ⟨*check child pointers* 160b⟩≡ (158c)

```
int numChildren = children.getSize();
for (int i = 0; i < numChildren; i++) {
    EdgePtr childEdge = children.getElement(i);
    if (childEdge != nil) {
        NodePtr childNode = childEdge->getOtherNode(this);
        treeOK = childNode->checkTree(childEdge);
        if (treeOK == false) {
            break;
        }
    }
}
```

Uses `checkTree` 158c, `children` 139b, and `getOtherNode` 167a.

161a ⟨*check label and excess* 161a⟩≡ (158c)

```
if (treeOK && parentNode != nil) {
    if (excess < 0) {
        cout << "\nWarning: node " << id
             << " non-zero excess in non-root node "
             << excess << endl;
        treeOK = false;
    }
    else if ((label < parentNode->label) && (flag == 0)) {
        cout << "\nError: node " << id
             << " label inversion for non-degenerate node" << endl;
        treeOK = false;
    }
}
```

Uses `excess` 151b, `flag` 155d, and `parentNode` 140c.

Here is an inefficient function to find the root of a node. It is only intended for debugging use where the performance hit of chasing up to the root is not a problem.

161b ⟨*debugging functions* 156d⟩+≡ (164b) ◁158b 161d▷

```
NodePtr getRoot();
```

Uses `getRoot` 161c.

161c ⟨*Node method implementations* 139d⟩+≡ (163c) ◁158c 162a▷

```
NodePtr Node::getRoot()
{
    Node* currNode = this;
    while (currNode->parentEdge != nil) {
        Node* parent = currNode->getParentNode();
        assert(parent != currNode);
        currNode = parent;
    }
    return currNode;
}
```

Defines:
`getRoot`, used in chunks 15a, 42b, 161, and 162.
Uses `getParentNode` 141a, `Node` 164b, and `parentEdge` 140c.

With that function, we can easily (if not efficiently) determine if a node is strong or weak.

161d ⟨*debugging functions* 156d⟩+≡ (164b) ◁161b 162b▷

```
bool isStrong(bool zeroExcessIsStrong);
```

Uses `isStrong` 162a.

162a ⟨*Node method implementations* 139d⟩+≡ (163c) ◁161c 162c▷

```
bool Node::isStrong(bool zeroExcessIsStrong)
{
    FlowAmount excess = getRoot()->excess;
    if (excess > 0) {
        return true;
    } else if (excess == 0 && zeroExcessIsStrong) {
        return true;
    } else {
        return false;
    }
}
```

Defines:
`isStrong`, used in chunks 12a, 52, 53a, 132a, 134a, and 161d.
Uses `excess` 151b, `getRoot` 161c, and `Node` 164b.

For tracing, printing the excess of the root is useful.

162b ⟨*debugging functions* 156d⟩+≡ (164b) ◁161d 162d▷

```
NodeExcess getRootExcess();
```

Uses `getRootExcess` 162c.

162c ⟨*Node method implementations* 139d⟩+≡ (163c) ◁162a 163a▷

```
NodeExcess Node::getRootExcess()
{ return getRoot()->excess; }
```

Defines:
`getRootExcess`, used in chunks 15a, 37c, and 162b.
Uses `excess` 151b, `getRoot` 161c, and `Node` 164b.

162d ⟨*debugging functions* 156d⟩+≡ (164b) ◁162b

```
FlowAmount computeExcess();
```

Uses `computeExcess` 163a.

163a ⟨*Node method implementations* 139d⟩+≡ (163c) ◁162c

```
FlowAmount Node::computeExcess()
{
    FlowAmount ex = 0;

    ElIterator it(neighbors);
    for (EdgePtr edge = it.getNext(); edge != nil; edge = it.getNext()) {
        if (edge->getSource() == this) {
            ex -= edge->getFlow();
        } else {
            ex += edge->getFlow();
        }
    }

    return ex;
}
```

Defines:
`computeExcess`, used in chunk 162d.
Uses `getFlow` 167d, `getNext` 150g, `getSource` 165d, `neighbors` 139b, and `Node` 164b.

We need a few function declarations for functions that the compiler will automatically generate for us. We don't want to use these functions because doing so would signal an error in our code that could result is dismal performance or bugs. We declare them private and provide no implementation so that compiler or linker will flag them as errors.

163b ⟨*C++ overhead* 134b⟩+≡ (136c 164b 179a 216b) ◁154c 175b▷

```
private:
    Node(const Node&);
    Node& operator=(const Node&);
```

Uses `Node` 164b.

3.2 File Boiler Plate

We start with the boiler-plate implementation file.

163c ⟨* 135a⟩+≡ ◁135c 164a▷

```
// $Revision: 1.33 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "Node.h"
#include <iostream.h>
#include <iomanip.h>

⟨Node method implementations 139d⟩
```

Uses `Node` 164b.

Through magic of the C preprocessor and the macro-like facilities of *noweb*, we can easily define the inline functions out-of-line to allow us to collect better profile information - i.e. collect data on the inline functions that would otherwise not show up in the function call traces of the profiler.

164a ⟨* 135a⟩+≡ ◁163c 178a▷

```
#ifndef INLINE_NODE
#define INLINE /*inline*/
⟨Node inline implementations 138e⟩
#undef INLINE
#endif /*INLINE_NODE*/
```

The header defines the `Node` class, its member functions, member data, and inline functions. Again, we have more boiler-plate.

164b ⟨header 135d⟩+≡ ◁137b 164c▷

```
// $Revision: 1.33 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#ifndef NODE_H
#define NODE_H
⟨header include files 128c⟩
#include "debug.h"

class Node
{
    ⟨Node methods 138d⟩
    ⟨debugging functions 156d⟩
    ⟨C++ overhead 134b⟩
private:
    ⟨Node data 138a⟩
};
```

Defines:
`Node`, used in chunks 3a, 6–18, 20–22, 24–26, 30c, 31a, 37, 38a, 42, 44–48, 50–55, 58a, 60–62, 64–69, 73–76, 79–105, 107, 108, 110–12, 115–17, 123a, 132–36, 138, 139, 141–45, 147, 149–58, 161–63, 169–71, 173c, 174a, 177, and 178a.

In case we want the inline functions to really be inlined, we also define them here in the header file.

164c ⟨header 135d⟩+≡ ◁164b 179a▷

```
#ifdef INLINE_NODE
#define INLINE inline
⟨Node inline implementations 138e⟩
#undef INLINE
#endif /*INLINE_NODE*/

#endif /*NODE_H*/
```

4 Edges

4.1 Structural Data

An edge represents a relationship between two nodes, the source and the destination.

165a ⟨*Edge data* 165a⟩≡ (179a) 165e▷
```
NodePtr     source;
NodePtr     dest;
```
Defines:
`dest`, used in chunks 15d, 16a, 112b, 113, 125a, 127b, 165–67, 170e, 171b, 174a, 176, and 177.
`source`, used in chunks 57, 58a, 99b, 100, 103b, 114a, 122, 124, 125a, 127b, 132a, 165–67, 169–71, 174a, 176, and 177.

165b ⟨*default Edge constructor* 165b⟩≡ (170c) 165f▷
```
source = dest = nil;
```
Uses `dest` 165a and `source` 165a.

165c ⟨*Edge methods* 165c⟩≡ (179a) 166a▷
```
NodePtr getSource() const;
NodePtr getDest() const;
```
Uses `getDest` 165d and `getSource` 165d.

165d ⟨*Edge inline implementations* 165d⟩≡ (178b 179b) 166b▷
```
INLINE NodePtr Edge::getSource() const
{ return source; }
INLINE NodePtr Edge::getDest() const
{ return dest; }
```
Defines:
`getDest`, used in chunks 57, 58a, 71b, 73b, 80a, 88b, 89a, 91, 96, 98a, 105, 107, 111a, 113, 114a, and 165c.
`getSource`, used in chunks 57, 58a, 71b, 73b, 80a, 88b, 96, 97a, 105, 108, 111a, 113, 114a, 163a, and 165c.
Uses `dest` 165a, `Edge` 179a, and `source` 165a.

For the purposes of the algorithm, an edge has an orientation that idicates which direction we are pushing additional flow - i.e. from child towards parent.

165e ⟨*Edge data* 165a⟩+≡ (179a) ◁165a 167b▷
```
bool        isForward;
```
Uses `isForward` 165f.

165f ⟨*default Edge constructor* 165b⟩+≡ (170c) ◁165b 167c▷
```
isForward = true;
```
Defines:
`isForward`, used in chunks 165, 166, 168–71, 174a, and 176a.

We can set the orientation in two ways: we can toggle or reverse it, or we can set the orientation such that the forward direction points towards either of the endpoints.

166a ⟨*Edge methods* 165c⟩+≡ (179a) ◁165c 166c▷
```
void toggleOrientation();
void setDirectionTo(NodePtr endpoint);
```

166b ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁165d 166d▷
```
INLINE void Edge::toggleOrientation()
{ isForward = isForward ? false : true; }
INLINE void Edge::setDirectionTo(NodePtr endpoint)
{
    assert(endpoint == source || endpoint == dest);
    isForward = (endpoint == dest) ? true : false;
}
```
Uses `dest` 165a, `Edge` 179a, `isForward` 165f, and `source` 165a.

Since the source and destination of the edge are fixed with regard to the original, forward orientation, we also need a way to get the current head and tail of the edge - i.e. with respect to the orientation, the edge goes from the tail to the head.

166c ⟨*Edge methods* 165c⟩+≡ (179a) ◁166a 166e▷
```
NodePtr getTail() const;
NodePtr getHead() const;
```
Uses `getHead` 22a 166d and `getTail` 22a 166d.

166d ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁166b 167a▷
```
INLINE NodePtr Edge::getTail() const
{ return (isForward == true) ? source : dest; }
INLINE NodePtr Edge::getHead() const
{ return (isForward == true) ? dest : source; }
```
Defines:
`getHead`, used in chunks 25a, 71b, 109c, 126b, 159, and 166c.
`getTail`, used in chunks 25a, 71b, 126b, 159, and 166c.
Uses `dest` 165a, `Edge` 179a, `isForward` 165f, and `source` 165a.

Regardless of the orientation of the node, given one endpoint we want to be able to get the other node and ask if an edge is incident to a given node.

166e ⟨*Edge methods* 165c⟩+≡ (179a) ◁166c 167d▷
```
NodePtr getOtherNode(const NodePtr node) const;
bool isIncident(const NodePtr node) const;
```
Uses `getOtherNode` 167a and `isIncident` 167a.

167a ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁166d 168a▷
```
INLINE NodePtr Edge::getOtherNode(const NodePtr node) const
{ return (node == source) ? dest : source; }
INLINE bool Edge::isIncident(const NodePtr node) const
{ return ((node == source) || (node == dest)) ? true : false; }
```
Defines:
`getOtherNode`, used in chunks 11a, 12a, 53a, 54a, 64–69, 74a, 76a, 81, 84a, 85, 89a, 91, 102, 110, 134a, 143b, 147, 149, 157, 160, and 166e.
`isIncident`, used in chunk 166e.
Uses `dest` 165a, `Edge` 179a, and `source` 165a.

4.2 Flow Data

167b ⟨*Edge data* 165a⟩+≡ (179a) ◁165e 172a▷
```
FlowAmount  capacity;
FlowAmount  flow;
```
Uses `capacity` 167d and `flow` 167d.

167c ⟨*default Edge constructor* 165b⟩+≡ (170c) ◁165f
```
capacity = flow = 0;
```
Uses `capacity` 167d and `flow` 167d.

167d ⟨*Edge methods* 165c⟩+≡ (179a) ◁166e 167e▷
```
FlowAmount getFlow() const { return flow; }
FlowAmount getCapacity() const { return capacity; }
```
Defines:
`capacity`, used in chunks 111a, 125a, 127b, 167–73, 175–77, 180, 181, 187b, and 188b.
`flow`, used in chunks 54a, 65, 113, 152–54, 167–71, 174–76, 195, 198b, 202, 207, and 212.
`getCapacity`, used in chunk 111a.
`getFlow`, used in chunks 71b, 97, 98, 104a, 130b, and 163a.

Although the capacity of an edge is fixed during the duration of the algorithm, the flow varies. We usually speak of *increasing* the flow, but this is done with respect to the current orientation of the edge, so it may in fact be reducing the amount of flow if the current orientation is backwards.

167e ⟨*Edge methods* 165c⟩+≡ (179a) ◁167d 168b▷
```
public:
    void increaseFlow(FlowAmount amt);
```
Uses `increaseFlow` 168a.

168a ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁167a 168c▷
```
INLINE void Edge::increaseFlow(FlowAmount amt)
{
    if (isForward) {
        assert((capacity - flow) >= amt);
        flow += amt;
    } else {
        assert(flow >= amt);
        flow -= amt;
    }
}
```
Defines:
`increaseFlow`, used in chunks 40c, 41a, 65, 107, 152c, 153a, 167e, and 168c.
Uses `capacity` 167d, `Edge` 179a, `flow` 167d, and `isForward` 165f.

For simplex mergers, we need to be able to decrease the flow on an arc with respect to its current orientation. This is pretty trivial using `increaseFlow`.

168b ⟨*Edge methods* 165c⟩+≡ (179a) ◁167e 168d▷
```
public:
    void decreaseFlow(FlowAmount amt);
```
Uses `decreaseFlow` 168c.

168c ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁168a 168e▷
```
INLINE void Edge::decreaseFlow(FlowAmount amt)
{
    increaseFlow(-amt);
}
```
Defines:
`decreaseFlow`, used in chunks 54a, 108, 154, and 168b.
Uses `Edge` 179a and `increaseFlow` 168a.

Another similar method is used during flow recovery is to reduce the flow amount regardless of its orientation. We assume that the caller has already used `reductionCapacity` to check this edge.

168d ⟨*Edge methods* 165c⟩+≡ (179a) ◁168b 169a▷
```
    void reduceFlow(FlowAmount amt);
```
Uses `reduceFlow` 104a 168e.

168e ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁168c 169b▷
```
INLINE void Edge::reduceFlow(FlowAmount amt)
{
    assert(amt <= flow);
    flow -= amt;
}
```
Defines:
`reduceFlow`, used in chunks 97a, 99b, 103, and 168d.
Uses `Edge` 179a and `flow` 167d.

We need to to define the residual capacity with respect to the current orientation and from one node to the other.

169a ⟨*Edge methods* 165c⟩+≡ (179a) ◁168d 169c▷

```
public:
    FlowAmount residCapacity() const;
    FlowAmount residCapacity(const Node& node) const;
```

Uses `Node` 164b and `residCapacity` 169b.

169b ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁168e 170a▷

```
INLINE FlowAmount Edge::residCapacity() const
{
    if (isForward) {
        return capacity - flow;
    } else {
        return flow;
    }
}
INLINE FlowAmount Edge::residCapacity(const Node& node) const
{
    if (&node == source) {
        return capacity - flow;
    } else {
        return flow;
    }
}
```

Defines:
`residCapacity`, used in chunks 11a, 12a, 15a, 37c, 38a, 53a, 54a, 57, 66a, 69a, 74a, 81, 91, 110, 134a, 141a, 152, 153, 157, and 169a.
Uses `capacity` 167d, `Edge` 179a, `flow` 167d, `isForward` 165f, `Node` 164b, and `source` 165a.

During flow recovery, we look for path with residual capacity that only involve reducing the flow on arcs. We encounter the edges as either in or out arcs, so we are looking for capacity with respect to a certain node. So, the capacity is either the existing flow or zero.

169c ⟨*Edge methods* 165c⟩+≡ (179a) ◁169a 170b▷

```
    FlowAmount reductionCapacity(const Node& fromNode) const;
```

Uses `Node` 164b and `reductionCapacity` 170a.

170a ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁169b 173b▷

```
INLINE FlowAmount Edge::reductionCapacity(const Node& fromNode) const
{
    if (&fromNode == source) {
        return flow;
    } else {
        return 0;
    }
}
```

Defines:
`reductionCapacity`, used in chunks 102 and 169c.
Uses `Edge` 179a, `flow` 167d, `Node` 164b, and `source` 165a.

We need a default constructor because we will be creating an array of edges in bulk when we know how many of them are in the graph.

170b ⟨*Edge methods* 165c⟩+≡ (179a) ◁169c 170d▷

```
Edge();
```

Uses `Edge` 179a.

170c ⟨*Edge method implementations* 170c⟩≡ (178a) 170e▷

```
Edge::Edge()
{
    ⟨default Edge constructor 165b⟩
}
```

Uses `Edge` 179a.

After constructing an edge, we will need to initialize it. The default orientation will be from the source to the destination.

170d ⟨*Edge methods* 165c⟩+≡ (179a) ◁170b 171a▷

```
void init(Node& src, Node& dst, FlowAmount cap);
```

Uses `init` 155a 170e and `Node` 164b.

170e ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁170c 171b▷

```
void Edge::init(Node& src, Node& dst, FlowAmount cap)
{
    source = &src;
    dest = &dst;
    flow = 0;
    isForward = true;
    capacity = cap;
}
```

Defines:
`init`, used in chunks 125a, 126a, 154e, 170, 171, 180, 181a, 195, 198b, 202, and 207.
Uses `capacity` 167d, `dest` 165a, `Edge` 179a, `flow` 167d, `isForward` 165f, `Node` 164b, and `source` 165a.

We also need to initialize parameteric arcs where the capacity is defined by two parameters, a and b.

171a ⟨*Edge methods* 165c⟩+≡ (179a) ◁170d 171c▷

```
#ifdef PARAM_SENSE
    void init(Node& src, Node& dst, FlowAmount aParam, double bParam);
#endif PARAM_SENSE
```

Uses `init` 155a 170e and `Node` 164b.

171b ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁170e 171d▷

```
#ifdef PARAM_SENSE
void Edge::init(Node& src, Node& dst, FlowAmount _aParam, double _bParam)
{
    source = &src;
    dest = &dst;
    flow = 0;
    isForward = true;
    capacity = 0;
    aParam = _aParam;
    bParam = _bParam;
}
#endif PARAM_SENSE
```

Uses `capacity` 167d, `dest` 165a, `Edge` 179a, `flow` 167d, `init` 155a 170e, `isForward` 165f, `Node` 164b, and `source` 165a.

At the begining of the algorithm, we need to saturate the source- and sink-adjacent edges.

171c ⟨*Edge methods* 165c⟩+≡ (179a) ◁171a 172b▷

```
    FlowAmount saturate();
```

Uses `saturate` 171d.

171d ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁171b 172c▷

```
FlowAmount Edge::saturate()
{ assert(flow == 0); flow = capacity; return flow; }
```

Defines:
`saturate`, used in chunks 57, 58a, 71b, 109b, 171c, 175a, 198b, and 199b.
Uses `capacity` 167d, `Edge` 179a, and `flow` 167d.

4.3 Parametric Flow

To perform parametric sesnitivity analysis, we will specify the capacity in terms of a simple, linear function: $c = a + b\lambda$. Because our implementation assumes integer flows and capacities, we will round the capacity function to the nearest integer. This function will apply to the arcs our of the source and into the sink. The source-adjacent arcs will increase in capacity as a function of λ and the sink-adjacent arcs will decrease.

Therefore, we will store the two parameters, a and b, on the edge, and λ will be provided externally. To simply things, we will conditional compilation (ifdef) rather than a sub-class.

172a ⟨*Edge data* 165a⟩+≡ (179a) ◁167b

```
#ifdef PARAM_SENSE
    FlowAmount aParam;
    double      bParam;
#endif PARAM_SENSE
```

We set the initial capacity based on the initial value of lambda. Because we take the absolute value of the the capacity, we need to signal the caller as to whether the capacity is actually negative.

172b ⟨*Edge methods* 165c⟩+≡ (179a) ◁171c 173a▷

```
    bool setInitialCapacity(double lambda);
```

Uses `setInitialCapacity` 105 172c.

172c ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁171d 174a▷

```
bool Edge::setInitialCapacity(double lambda)
{
    bool negative = false;
#ifdef PARAM_SENSE
    FlowAmount newCapacity = (FlowAmount)(aParam + bParam * lambda);
    negative = (newCapacity < 0) ? true : false;
    capacity = iabs(newCapacity);
#endif PARAM_SENSE

    return negative;
}
```

Defines:
`setInitialCapacity`, used in chunks 104b, 172b, and 202.
Uses `capacity` 167d and `Edge` 179a.

172d ⟨*Edge header include files* 172d⟩≡ (179a) 176b▷

```
#include <math.h>
```

Given a new value for λ, we need to update the arc capacity. This method is actually specific to the formulation used in the mining problem. In particular, sink-adjacent nodes have negative weights, but the arc between them and the sink uses the absolute value of the capacity. Furthermore, as λ increases, the block value can change from negative to positive. In the formulation, we need to know when this happens so we can move the node from the sink to the source.

Therefore, to provide all this support, we have two input parameters (the old and new lambda values) and two output paramters (the amount of change and whether we changed sign). We use the old lambda to compute the current capacity with its real sign (not absolute valued). Then we can compute the new capacity with the new lambda, taking the absolute value if necessary.

173a ⟨*Edge methods* 165c⟩+≡ (179a) ◁172b 173c▷

```
bool increaseLambda(double oldLambda, double newLambda, FlowAmount& delta);
```

Uses `increaseLambda` 173b.

173b ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁170a 175a▷

```
INLINE bool Edge::increaseLambda(double oldLambda,
                                 double newLambda, FlowAmount& delta)
{
    assert(newLambda >= oldLambda);
    bool crossedZero = false;
#ifdef PARAM_SENSE
    FlowAmount oldCapacity  = (FlowAmount)(aParam + bParam * oldLambda);
    FlowAmount newCapacity  = (FlowAmount)(aParam + bParam * newLambda);
    delta = newCapacity - oldCapacity;
    capacity = iabs(newCapacity);
    crossedZero = ((oldCapacity < 0) && (newCapacity >= 0)) ? true : false;
#endif PARAM_SENSE

    return crossedZero;
}
```

Defines:
`increaseLambda`, used in chunks 107, 108, and 173a.
Uses `capacity` 167d and `Edge` 179a.

When we increase the capacity of an arc into the sink above zero, we move the arc to the source. This involves changing the source and destination. We assume that `increaseLambda` has already established the new, positive capacity.

173c ⟨*Edge methods* 165c⟩+≡ (179a) ◁173a 174b▷

```
void reInit(Node& src, Node& dst);
```

Uses `Node` 164b and `reInit` 174a.

174a ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁172c 176a▷

```
void Edge::reInit(Node& src, Node& dst)
{
    assert((source == &dst) || (dest == &src));   // should be: (v,t) <--> (s,v)
    source = &src;
    dest = &dst;
    flow = 0;
    isForward = true;
}
```

Defines:
`reInit`, used in chunks 105, 109a, and 173c.
Uses `dest` 165a, `Edge` 179a, `flow` 167d, `isForward` 165f, `Node` 164b, and `source` 165a.

For general parametric analysis, we need to set the capacity to a new arbitrary value. If the arc was saturated to begin with, we keep it saturated by increasing the flow amount. If the new capacity is less than the flow amount, we reduce the flow amount. We always return the amount of adjustment to the flow.

174b ⟨*Edge methods* 165c⟩+≡ (179a) ◁173c 176c▷

```
    FlowAmount adjustCapacity(FlowAmount newCap);
```

Uses `adjustCapacity` 113 175a.

175a ⟨*Edge inline implementations* 165d⟩+≡ (178b 179b) ◁173b

```
INLINE FlowAmount Edge::adjustCapacity(FlowAmount newCap)
{
    assert(newCap >= 0);
    if (capacity == newCap) {
        return 0;
    }

    FlowAmount delta = 0;
    if ((flow == capacity) && (flow > 0)) {
        // already saturated
        delta = newCap - capacity;
        flow = newCap;
    } else {
        if (flow > newCap) {
            // need to reduce flow - arc is saturate at new capacity
            delta = newCap - flow;
            flow = newCap;
        }
    }
    capacity = newCap;
    assert((flow >= 0) && (flow <= capacity));
    return delta;
}
```

Defines:
`adjustCapacity`, used in chunks 112b, 114a, 127b, and 174b.
Uses `capacity` 167d, `Edge` 179a, `flow` 167d, and `saturate` 171d.

4.4 Miscellaneous Functions

We need a way to print the contents of an edge to an output stream, mostly for debugging.

175b ⟨*C++ overhead* 134b⟩+≡ (136c 164b 179a 216b) ◁163b 215a▷

```
friend ostream& operator<<(ostream& out, const Edge& edge);
```

Uses `Edge` 179a.

176a ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁174a 176d▷

```
ostream& operator<<(ostream& out, const Edge& edge)
{
    out << "(" << &edge << "): ";
    if (edge.isForward == true) {
        out << edge.source->getId() << "(src) --> "
            << edge.dest->getId() << "(dest)";
    } else {
        out << edge.dest->getId() << "(dest) --> "
            << edge.source->getId() << "(src)";
    }
    return out << " [" << edge.flow << "," << edge.capacity << "]";
}
```

Uses `capacity` 167d, `dest` 165a, `Edge` 179a, `flow` 167d, `isForward` 165f, and `source` 165a.

176b ⟨*Edge header include files* 172d⟩+≡ (179a) ◁172d

```
#include <iosfwd>
```

We also need to print the flow data in Dimacs format. Since we already used the insertion operator, we'll just define a 'normal' method.

176c ⟨*Edge methods* 165c⟩+≡ (179a) ◁174b 176e▷

```
ostream& writeFlow(ostream& out);
```

Uses `writeFlow` 176d.

176d ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁176a 177a▷

```
ostream& Edge::writeFlow(ostream& out)
{
    return out << "f "
               << setw(6) << source->getId() << " "
               << setw(6) << dest->getId() << " "
               << setw(6) << flow;
}
```

Defines:
`writeFlow`, used in chunks 128, 176c, 195, 198a, 199a, and 202.
Uses `dest` 165a, `Edge` 179a, `flow` 167d, and `source` 165a.

Here is a simple method that writes the arc out in Dimacs problem instance format (i.e. .max format) with the source, destination and capacity.

176e ⟨*Edge methods* 165c⟩+≡ (179a) ◁176c 177b▷

```
ostream& writeArcInstance(ostream& out);
```

Uses `writeArcInstance` 177a.

177a ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁176d 177c▷

```
ostream& Edge::writeArcInstance(ostream& out)
{
    return out << "a  "
               << setw(6) << source->getId() << " "
               << setw(6) << dest->getId() << " "
               << setw(6) << capacity;
}
```

Defines:
`writeArcInstance`, used in chunks 131a and 176e.
Uses `capacity` 167d, `dest` 165a, `Edge` 179a, and `source` 165a.

Here's another debugging function that just validates that a given edge is in fact and edge between two specified nodes. We don't care which is the source or destination. If the edge or nodes are not valid, we return false so that the caller can throw an assertion failure.

177b ⟨*Edge methods* 165c⟩+≡ (179a) ◁176e

```
bool validateEdge(Node& node1, Node& node2);
```

Uses `Node` 164b and `validateEdge` 177c.

177c ⟨*Edge method implementations* 170c⟩+≡ (178a) ◁177a

```
bool Edge::validateEdge(Node& node1, Node& node2)
{
    bool result = false;
    if (source == &node1) {
        result = (dest == &node2) ? true : false;
    } else {
        result = (dest == &node1) ? true : false;
    }

    return result;
}
```

Defines:
`validateEdge`, used in chunks 142 and 177b.
Uses `dest` 165a, `Edge` 179a, `Node` 164b, and `source` 165a.

4.5 File Boiler Plate

We start with the boiler-plate implementation file.

178a ⟨* 135a⟩+≡ ◁164a 178b▷

```
// $Revision: 1.24 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "Edge.h"
#include "Node.h"
#include <iostream>
#include <iomanip>
using namespace std;

⟨Edge method implementations 170c⟩
```

Uses Edge 179a and Node 164b.

Through magic of the C preprocessor and the macro-like facilities of *noweb*, we can easily define the inline functions out-of-line to allow us to collect better profile information - i.e. collect data on the inline functions that would otherwise not show up in the function call traces of the profiler.

178b ⟨* 135a⟩+≡ ◁178a 191d▷

```
#ifndef INLINE_EDGE
#define INLINE /*inline*/
⟨Edge inline implementations 165d⟩
#undef INLINE
#endif /*INLINE_EDGE*/
```

The header defines the `Edge` class, its member functions, member data, and inline functions. Again, we have more boiler-plate.

179a ⟨*header* 135d⟩+≡ ◁164c 179b▷

```
// $Revision: 1.24 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#ifndef EDGE_H
#define EDGE_H
#include "types.h"
#include "debug.h"
⟨Edge header include files 172d⟩
using namespace std;

class Edge
{
public:
    ⟨Edge methods 165c⟩
    ⟨C++ overhead 134b⟩
private:
    ⟨Edge data 165a⟩
};
```

Defines:
`Edge`, used in chunks 3a, 11a, 14b, 15a, 37, 42, 56c, 65, 66a, 69a, 71b, 73b, 76a, 80a, 88b, 96–98, 102–105, 107, 108, 110, 111a, 113, 123a, 130b, 139–44, 152b, 153c, and 165–78.

In case we want the inline functions to really be inlined, we also define them here in the header file.

179b ⟨*header* 135d⟩+≡ ◁179a 192b▷

```
#ifdef INLINE_EDGE
#define INLINE inline
⟨Edge inline implementations 165d⟩
#undef INLINE
#endif /*INLINE_EDGE*/

#endif /*EDGE_H*/
```

4.6 Lists of Edges

To represent the original graph, we need a list of the neighbors of a node. To represent a tree, we need a list of the children Nodes. However, as we scan for merger arcs or rehang the tree, we need more than just a pointer to a Node, we also need the edge that defines the relationship between the two nodes. Therefore, we need a list of Edges rather than a list of Nodes.

In principal, the number of edges in the list is bounded by the degree of the node. *However, for a child list, during rehang operations we will be removing nodes and adding them. Because we want to avoid compacting the list (slow) after deleting a node, we should allow the list to be dynamic. For the moment, we will ignore this and will revisit it later.*

A list is simply an array of Edge pointers, a capacity, and a current size.

180a ⟨*EdgeList data* 180a⟩≡ (192b) 184b▷

```
EdgePtr*    array;
int         capacity;
int         size;
```

Uses capacity 167d.

Here are constructors and an initialization method for use after a constructor has already been called (e.g. after allocating an array of NodeLists).

180b ⟨*EdgeList methods* 180b⟩≡ (192b) 181b▷

```
public:
    EdgeList();
    EdgeList(int cap);
    void init(int cap);
```

Uses EdgeList 193a and init 155a 170e.

180c ⟨*EdgeList inline implementations* 180c⟩≡ (192a 193b) 184c▷

```
INLINE EdgeList::EdgeList()
{ array = nil; capacity = size = 0; }
INLINE EdgeList::EdgeList(int cap) : array(nil)
{ init(cap); }
```

Uses capacity 167d, EdgeList 193a, and init 155a 170e.

181a ⟨*EdgeList method implementations* 181a⟩≡ (191d) 181c▷

```
void EdgeList::init(int cap)
{
    delete array;
    array = new EdgePtr[cap];
    capacity = cap; size = 0;
    for (int i = 0; i < cap; i++) {
        array[i] = nil;
    }
    beginIteration();
}
```

Uses `beginIteration` 184c, `capacity` 167d, `EdgeList` 193a, and `init` 155a 170e.

The point of the list is to store items, so we need a way to add elements to the list. The simplist is to add an element to the end of the list.

181b ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁180b 181d▷

```
    void appendEdge(EdgePtr edgep);
```

Uses `appendEdge` 181c.

181c ⟨*EdgeList method implementations* 181a⟩+≡ (191d) ◁181a 182a▷

```
void EdgeList::appendEdge(EdgePtr edgep)
{
    if (size == capacity) {
        compactList();
    }
    assert((array != nil) && (size < capacity));
    array[size] = edgep;
    size++;
}
```

Defines:
`appendEdge`, used in chunks 139d, 181b, and 186c.
Uses `capacity` 167d, `compactList` 182a, and `EdgeList` 193a.

Sometimes we can end up with a number of holes in the list (possibly the whole list) and we need to compact the list to move the holes to the end. In general we try to avoid doing this often (e.g. everytime we remove an element), but we have found cases where we have no choice.

181d ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁181b 184a▷

```
    void compactList();
```

Uses `compactList` 182a.

182a ⟨*EdgeList method implementations* 181a⟩+≡ (191d) ◁181c 187b▷

```
void EdgeList::compactList()
{
    int hole;
    ⟨search for first hole 182b⟩
    int nonHole;
    ⟨search for first non-hole 182c⟩
    ⟨check for big hole at end 183a⟩

    while (nonHole < size) {
        ⟨copy from non-hole to hole 183b⟩
        ⟨advance hole and nonHole 183d⟩
    }
    if (currentIndex >= size) {
        currentIndex = hole;
    }
    size = hole;
}
```

Defines:
`compactList`, used in chunk 181.
Uses `currentIndex` 184c and `EdgeList` 193a.

It's pretty easy to find a hole. It had better be before the end of the list. Otherwise, the list is full, and there's nothing we can do about it.

182b ⟨*search for first hole* 182b⟩≡ (182a)

```
for (hole = 0; hole < size; hole++) {
    if (array[hole] == nil) {
        break;
    }
}
assert(hole < size);
```

Similarly, it's easy to find a non-hole after the hole. It's OK if it runs past the end of the list.

182c ⟨*search for first non-hole* 182c⟩≡ (182a)

```
for (nonHole = hole + 1; nonHole < size; nonHole++) {
    if (array[nonHole] != nil) {
        break;
    }
}
```

After searching for the first hole and non-hole, if we didn't find a non-hole, there are two possibilites: the list is empty or it contains some elements followed by all empty slots to the end of the list. If the list contains any elements, the index of the first hole equals size of the list. Either way, we don't have to copy anything around and we are done.

183a ⟨*check for big hole at end* 183a⟩≡ (182a)

```
if (nonHole >= size) {
    if (hole == 0) {
        currentIndex = size = 0;
    } else {
        size = hole;
        if (currentIndex > hole) {
            currentIndex = hole;
        }
    }
    return;
}
```

Uses currentIndex 184c.

When we have a pointer to the hole and non-hole, we can copy one element from into the hole.

183b ⟨*copy from non-hole to hole* 183b⟩≡ (182a) 183c▷

```
array[hole] = array[nonHole];
array[nonHole] = nil;
```

If the current index is between the hole and the non-hole, it should point at the hole (which just got filled in).

183c ⟨*copy from non-hole to hole* 183b⟩+≡ (182a) ◁183b

```
if ((currentIndex > hole) && (currentIndex <= nonHole)) {
    currentIndex = hole;
}
```

Uses currentIndex 184c.

Now, we know the next element after the hole is the next hole. Then we need to search for the next non-hole.

183d ⟨*advance hole and nonHole* 183d⟩≡ (182a)

```
hole++;
assert((hole < size) && (array[hole] == nil));
nonHole++;
while ((nonHole < size) && (array[nonHole] == nil)) {
    nonHole++;
}
```

We need methods to iterate over these lists of edges. We have no requirement to support multiple simultaneous iterations, so we can implement the functions and the iteration state directly on the edge list class ({em someday using STL would make sense). The methods are pretty straightforward.

184a ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁181d 185d▷

```
void beginIteration();
bool hasMoreElements();
EdgePtr getCurrentElement();
void advanceToNextElement();
```

Uses `advanceToNextElement` 185c, `beginIteration` 184c, `getCurrentElement` 185b, and `hasMoreElements` 185a.

With this simple interface, we only need to keep track of the current index into the array to implement a cursor, and begining the iteration is trivial.

184b ⟨*EdgeList data* 180a⟩+≡ (192b) ◁180a

```
int currentIndex;
```

Uses `currentIndex` 184c.

184c ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁180c 184e▷

```
INLINE void EdgeList::beginIteration()
{ currentIndex = 0; }
```

Defines:
`beginIteration`, used in chunks 143, 144, 181a, and 184a.
`currentIndex`, used in chunks 182–86 and 188–90.
Uses `EdgeList` 193a.

Because we allow 'holes' to exist within a childlist, the code for the other methods needs to be prepared to skip over holes. So, we add a method to skip over any holes begining at the current position. It will either leave the cursor pointing to the first non-hole or the end of the array.

184d ⟨*EdgeList internal methods* 184d⟩≡ (192b)

```
void skipHoles();
```

Uses `skipHoles` 184e.

184e ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁184c 185a▷

```
INLINE void EdgeList::skipHoles()
{
    while ((array[currentIndex] == nil) && (currentIndex < size)) {
        currentIndex++;
    }
}
```

Defines:
`skipHoles`, used in chunks 184 and 185.
Uses `currentIndex` 184c and `EdgeList` 193a.

With `skipHoles` in place, the other methods are pretty simple to implement. Basically, they use `skipHoles` to update the cursor, and perform their real work. `hasMoreElements` just tells us if there are any more non-hole elements in the list.

185a ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁184e 185b▷

```
INLINE bool EdgeList::hasMoreElements()
{
    skipHoles();
    return (currentIndex < size) ? true : false;
}
```

Defines:
`hasMoreElements`, used in chunks 143b, 144a, and 184a.
Uses `currentIndex` 184c, `EdgeList` 193a, and `skipHoles` 184e.

`getCurrentElement` returns the element at the current position. This should almost never be a hole because callers should call `hasMoreElements` first. Currently, we raise an assertion if this happens. We can fix it later if need be.

185b ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁185a 185c▷

```
INLINE EdgePtr EdgeList::getCurrentElement()
{
    assert((currentIndex < size) && (array[currentIndex] != nil));
    return array[currentIndex];
}
```

Defines:
`getCurrentElement`, used in chunks 143b, 144a, and 184a.
Uses `currentIndex` 184c and `EdgeList` 193a.

The `advanceToNextElement` moves us to the next non-hole element unless we're at the end of the array already, in which case it is a no-op.

185c ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁185b 186a▷

```
INLINE void EdgeList::advanceToNextElement()
{
    if (currentIndex < size) {
        currentIndex++;
        skipHoles();
    }
}
```

Defines:
`advanceToNextElement`, used in chunks 143b, 144a, and 184a.
Uses `currentIndex` 184c, `EdgeList` 193a, and `skipHoles` 184e.

While performing rehang operations, a Node needs to remove the child pointed to by the current cursor. This is the cause of holes in the list. We pass in the value that we expect to be at the cursor, mostly out of paranoia.

185d ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁184a 186b▷

```
    void removeCurrentElement(EdgePtr edgep);
```

Uses `removeCurrentElement` 186a.

186a ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁185c 186c▷

```
INLINE void EdgeList::removeCurrentElement(EdgePtr edgep)
{
    assert((currentIndex < size) && (array[currentIndex] == edgep));
    array[currentIndex] = nil;
}
```

Defines:
`removeCurrentElement`, used in chunks 147 and 185d.
Uses `currentIndex` 184c and `EdgeList` 193a.

Similarly, during rehang operations we need to put a Edge into the list at the current cursor location, which should be a hole. In this case, we can just stuff the element in the hole. The exceptions are if the list is empty, or if the current element is at the begining of the list. Another case comes up with pre-order searches where the size is greater than zero, but the current index is still zero. In these cases we aren't replacing the current cursor location but rather appending to the list.

186b ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁185d 187a▷

```
    void addCurrentElement(EdgePtr edgep);
```

Uses `addCurrentElement` 186c.

186c ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁186a 189b▷

```
INLINE void EdgeList::addCurrentElement(EdgePtr edgep)
{
    if ((size == 0) || (currentIndex == size)) {
        appendEdge(edgep);
    } else if (array[currentIndex] == nil) {
        array[currentIndex] = edgep;
    } else {
        appendEdge(edgep);
    }
}
```

Defines:
`addCurrentElement`, used in chunks 147 and 186b.
Uses `appendEdge` 181c, `currentIndex` 184c, and `EdgeList` 193a.

Another way to add an element to the list is to add it at the first hole or if there are no holes, to append it. Currently, we use a brute-force search for a hole, but in the future, we can speed this by keeping track of the lowest indexed hole we create. If this method adds the edge to a hole 'behind' the current index, we do not bother to back up the current index - i.e. we assume that we don't need to scan what we are adding.

When this is made more efficient, it should be inlined. To track the lowest hole, we should init the value to 'size' and update it upon removing elements. The toggle for rehang might unnecessarily disturb it. Might want a 'second lowest hole' member, too?

187a ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁186b 187c▷

```
void addHoleElement(EdgePtr edgep);
```

Uses `addHoleElement` 187b.

187b ⟨*EdgeList method implementations* 181a⟩+≡ (191d) ◁182a 188a▷

```
void EdgeList::addHoleElement(EdgePtr edgep)
{
    int index = 0;
    while ((array[index] != nil) && (index < capacity)) {
        index++;
    }

    assert(index < capacity);
    array[index] = edgep;
    if (index == size) {
        size++;
    }
}
```

Defines:
`addHoleElement`, used in chunks 139d, 142a, and 187a.
Uses `capacity` 167d and `EdgeList` 193a.

Yet another way to add an element (required by simplex) is to add it at or after the current element, or if that doesn't work add it to a hole. No matter where we add it, we need to make sure it gets scanned by the current index. So we might need to update the current index if we add it to a hole behind the current index. It is safe (but a bit inefficient) to back up the current index in the list.

187c ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁187a 189a▷

```
void addAfterCurrent(EdgePtr edgep);
```

Uses `addAfterCurrent` 188a.

188a ⟨*EdgeList method implementations* 181a⟩+≡ (191d) ◁187b 190c▷
```
void EdgeList::addAfterCurrent(EdgePtr edgep)
{
    ⟨add after the current element 188b⟩
    ⟨add before current element and back up 188c⟩
}
```
Defines:
`addAfterCurrent`, used in chunks 142c, 149, and 187c.
Uses `EdgeList` 193a.

First we scan from the current index to the end of the list looking for a hole. If we find one, just add the element and return because we don't need to udpate the index.

188b ⟨*add after the current element* 188b⟩≡ (188a)
```
for (int index = currentIndex; index < capacity; index++) {
    if (array[index] == nil) {
        array[index] = edgep;
        if (index == size) {
            size++;
        }
        return;
    }
}
```
Uses `capacity` 167d and `currentIndex` 184c.

If that failed, we start at the element before the current index and scan backwards. If we find a hole, we'll insert the element and update the current index.

188c ⟨*add before current element and back up* 188c⟩≡ (188a) 188d▷
```
for (int index = (currentIndex - 1); index >= 0; index--) {
    if (array[index] == nil) {
        array[index] = edgep;
        currentIndex = index;
        return;
    }
}
```
Uses `currentIndex` 184c.

If both of those failed, something is seriously wrong; just throw an assertion violation.

188d ⟨*add before current element and back up* 188c⟩+≡ (188a) ◁188c
```
assert(false);
```

When a node is begin split from its parent, we need to remove a child edge which is probably not the current edge. At the moment, we implement this via a linear search. In the future, perhaps we could use a hashtable, but it might be difficult to iterate over that given that a phase can be started and stopped multiple times.

189a ⟨*EdgeList methods* 180b⟩+≡ (192b) ◁187c

```
void removeElement(EdgePtr edgep);
```

Uses `removeElement` 189b.

189b ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁186c 191a▷

```
INLINE void EdgeList::removeElement(EdgePtr edgep)
{
    if (array[currentIndex] == edgep) {
        array[currentIndex] = nil;
        return;
    }

    for (int i = 0; i < size; i++) {
        if (array[i] == edgep) {
            array[i] = nil;
            ⟨adjust the list size 190a⟩
            return;
        }
    }
    assert("failed to find edge" == nil);       // force assertion failure
}
```

Defines:
`removeElement`, used in chunks 139f, 145c, 149, and 189a.
Uses `currentIndex` 184c and `EdgeList` 193a.

One complication is if we remove the last element in the list, we should shrink the size of the list. If there are holes 'behind' the last element, we should shrink the list even further. We should also back up the current element as we go.

190a ⟨*adjust the list size* 190a⟩≡ (189b)

```
if (i == (size - 1)) {
    size--;
    int endProbe = i - 1;
    while (endProbe >= 0) {
        if (array[endProbe] == nil) {
            size--;
            endProbe--;
        } else {
            break;
        }
    }
    if (currentIndex > size) {
         currentIndex = size;
    }
}
```

Uses `currentIndex` 184c.

Here are some direct accessor methods for use during debugging. These really should be 'hidden' and shared only to `friends`.

190b ⟨*EdgeList debug methods* 190b⟩≡ (192b)

```
int getSize() const;
EdgePtr getElement(int i) const;
```

190c ⟨*EdgeList method implementations* 181a⟩+≡ (191d) ◁188a

```
int EdgeList::getSize() const
{ return size; }
EdgePtr EdgeList::getElement(int i) const
{
    assert(i < size);
    return array[i];
}
```

Uses `EdgeList` 193a.

4.6.1 Iterator class

190d ⟨*ElIterator public declarations* 190d⟩≡ (193a) 191b ▷

```
ElIterator(EdgeList& l);
```

Uses `EdgeList` 193a.

191a ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁189b 191c▷
```
INLINE ElIterator::ElIterator(EdgeList& l) : list(l), position(-1)
{ }
```
Uses EdgeList 193a.

191b ⟨*ElIterator public declarations* 190d⟩+≡ (193a) ◁190d
```
    EdgePtr getNext();
```
Uses getNext 150g.

191c ⟨*EdgeList inline implementations* 180c⟩+≡ (192a 193b) ◁191a
```
INLINE EdgePtr ElIterator::getNext()
{
    EdgePtr result = nil;
    while (++position < list.size) {
        if (list.array[position] != nil) {
            result = list.array[position];
            break;
        }
    }

    return result;
}
```
Uses getNext 150g.

4.6.2 File Boiler Plate

We start with the boiler-plate implementation file.

191d ⟨* 135a⟩+≡ ◁178b 192a▷
```
// $Revision: 1.20 $, $Date: 2003/09/15 18:42:04 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "EdgeList.h"
#include <iostream.h>
#include <iomanip.h>
#include <assert.h>

⟨EdgeList method implementations 181a⟩
```
Uses EdgeList 193a.

Through magic of the C preprocessor and the macro-like facilities of *noweb*, we can easily define the inline functions out-of-line to allow us to collect better profile information - i.e. collect data on the inline functions that would otherwise not show up in the function call traces of the profiler.

192a ⟨* 135a⟩+≡ ◁191d

```
#ifndef INLINE_EDGELIST
#define INLINE /*inline*/
⟨EdgeList inline implementations 180c⟩
#undef INLINE
#endif /*INLINE_EDGELIST*/
```

The header defines the `EdgeList` class, its member functions, member data, and inline functions. Again, we have more boiler-plate.

192b ⟨*header* 135d⟩+≡ ◁179b 193a▷

```
// $Revision: 1.20 $, $Date: 2003/09/15 18:42:04 $
// This C++ code was generated by noweb from the corresponding .nw file
#ifndef EDGELIST_H
#define EDGELIST_H
#include "types.h"
#include "debug.h"

class ElIterator;
class EdgeList
{
    ⟨EdgeList methods 180b⟩
    ⟨EdgeList debug methods 190b⟩
protected:
    ⟨EdgeList internal methods 184d⟩
private:
    ⟨EdgeList data 180a⟩
    friend class ElIterator;
};
```

Uses `EdgeList` 193a.

Here's the declaration for our iterator class.

193a ⟨*header* 135d⟩+≡ ◁192b 193b▷

```
class ElIterator
{
    EdgeList& list;
    int position;
public:
    ⟨ElIterator public declarations 190d⟩
private:
    ElIterator();
};
```

Defines:
`EdgeList`, used in chunks 139, 180–82, and 184–92.

In case we want the inline functions to really be inlined, we also define them here in the header file.

193b ⟨*header* 135d⟩+≡ ◁193a

```
#ifdef INLINE_EDGELIST
#define INLINE inline
⟨EdgeList inline implementations 180c⟩
#undef INLINE
#endif /*INLINE_EDGELIST*/

#endif /*EDGELIST_H*/
```

5 Driver Programs

The driver programs for the parametric and non-parametric solvers are very similar. So, we have combined them here via `noweb` to reduce the duplication of code. However, since these programs originated as separate programs, and I'm short of time/lazy, their 'LP-ness' leaves something to be desired.

5.1 Non-parametric Solver

194a ⟨*llps* 194a⟩≡ 195▷

```
// $Revision: 1.12 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "PhaseSolver.h"
⟨Timer declaration and implementation 216b⟩

⟨common declarations 194b⟩
```

Uses `PhaseSolver` 136c.

194b ⟨*common declarations* 194b⟩≡ (194a 202 206)

```
#include <iostream.h>
#include <fstream.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>
#ifdef NEED_GETOPT
#include <getopt.h>
#endif /*NEED_GETOPT*/

extern char* buildFlags;
extern char* buildDate;

// set to true to cause (time consuming) tree checks to take place
bool checkTree = false;
// set to true to print verbose tracing messages
bool tracingEnabled = false;
```

Uses `checkTree` 158c.

195 ⟨*llps* 194a⟩+≡ ◁194a

```
void
usage()
{
    cerr << "Usage: llps [-d] [-f] [-t] [-x] [-g freq][-I init] [-N norm] [-B branch] input-g
    ⟨common option usage 198b⟩
    cerr << "buildFlags: " << buildFlags << endl;
    cerr << "buildDate: " << buildDate << endl;
}

int
main(int argc, char** argv)
{
    ⟨common main variables 198a⟩

    // parse arguments
    int ch;
    while ((ch = getopt(argc, argv, "dfg:txI:s:M:N:B:O:L:")) != EOF) {
        switch (ch) {
        ⟨common option parsing 199a⟩
        default:
            usage();
            return 1;
        }
    }

    argc -= optind;
    //argv += optind;

    if (argc != 2) {
        usage();
        return 1;
    }

    const char* instanceName = argv[optind];
    const char* outputName = argv[optind + 1];

    ofstream dout(outputName, ios::out);
    if (dout == nil) {
        cerr << "Unable to open output file: " << outputName << endl;
        return 1;
    }

    dout << "c" << endl;
```

```
dout << "c  instance: " << instanceName << endl;
dout << "c  buildFlags: " << buildFlags << endl;
dout << "c  buildDate: " << buildDate << endl;
dout << "c  argv: ";
for (char** ap = argv; *ap != nil; ap++) {
    dout << *ap << " ";
}
dout << endl;

if (numSplits >= 0) {
    solver->maxSplits = numSplits;
}
if (relabelFreq >= 0.0) {
    solver->relabelFrequency = relabelFreq;
}
solver->postOrderSearch = postOrder;

time_t beginTime = time(0);
Timer readTimer;
readTimer.start();
bool readOK = solver->readDimacsInstance(instanceName);
readTimer.stop();

if (readOK == false) {
    dout << "c  CANNOT READ INSTANCE" << endl;
    return 1;
}

cout << "read problem instance OK" << endl;

Timer totalTimer;
totalTimer.start();                // total time is everything except I/O

// build initial tree
Timer treeTimer;
Timer solveTimer;
treeTimer.start();
solveTimer.start();                // tree time is part of solve
(solver->*initFunc)(labelMethod);
treeTimer.stop();

// solve
(solver->*solverFunc)(addBranchFunc);
solveTimer.stop();
```

```
    // convert to flow
    Timer convertTimer;
    convertTimer.start();
    if (!dumpNodes) {
        // don't convert to flow if we're dumping the nodes,
        // which only makes sense after phase I before conversion
        solver->convertToFlow();
    }
    convertTimer.stop();

    totalTimer.stop();

    cout << "time to readInstance: "    << readTimer << endl;
    cout << "time to establish tree: "  << treeTimer << endl;
    cout << "done solving: "            << solveTimer << endl;
    cout << "time to convert flow: "    << convertTimer << endl;
    cout << "total time: "              << totalTimer << endl;

    time_t endTime = time(0);
    dout << "c  beginRun: "             << ctime(&beginTime);
    dout << "c  endRun: "               << ctime(&endTime);

    dout << "c  timeToRead: "           << readTimer << endl;
    dout << "c  timeToInitialize: "     << treeTimer << endl;
    dout << "c  timeToSolve: "          << solveTimer << endl;
    dout << "c  timeToFlow: "           << convertTimer << endl;
    dout << "c  totalTime: "            << totalTimer << endl;

    solver->writeStats(dout, true);
    solver->writeStats(cout, true);

    if (dumpNodes) {
        solver->dumpNodes(dout);
        solver->writeDimacsFlow(dout);
    } else if (writeFlow) {
        solver->writeDimacsFlow(dout);
    }

    dout.close();

    return 0;
}
```

Uses addBranchFunc 26a, convertToFlow 95a, dumpNodes 132a, flow 167d,

init 155a 170e, maxSplits 67b, nodes 3a, numSplits 129b, readDimacsInstance 120d, relabelFrequency 77d, solve 4b, start 214b, stop 214b, Timer 216b, writeDimacsFlow 128b 131a, writeFlow 176d, and writeStats 128e.

198a ⟨*common main variables* 198a⟩≡ (195 202 207)

```
bool dumpNodes = false;
bool writeFlow = false;
void (PhaseSolver::* initFunc)(LabelMethod) = &PhaseSolver::buildSimpleTree;
void (PhaseSolver::* solverFunc)(AddBranchPtr) = &PhaseSolver::solve;
AddBranchPtr addBranchFunc = &PhaseSolver::addBranchLifo;
PhaseSolver* solver = new PhaseSolver();
int numSplits = -1;
bool postOrder = false;
float relabelFreq = 0.0;
LabelMethod labelMethod = LABELS_CONSTANT;
```

Uses addBranchFunc 26a, addBranchLifo 26c, AddBranchPtr 136a, buildSimpleTree 56a, dumpNodes 132a, numSplits 129b, PhaseSolver 136c, solve 4b, and writeFlow 176d.

198b ⟨*common option usage* 198b⟩≡ (195 202 207)

```
cerr << "\t -d   dump the final disposition of each node" << endl;
cerr << "\t -f   write the flow values for each arc" << endl;
cerr << "\t -g   specify the global relabel frequency" << endl;
cerr << "\t -t   perform checkTree operations frequently" << endl;
cerr << "\t -x   perform verbose tracing of operations" << endl;
cerr << "\t -I   initialization function: simple, path, saturate, greedy " << endl;
cerr << "\t -s   specify the number of splits for path init" << endl;
cerr << "\t -B   strong bucket management: fifo, lifo, wave" << endl;
cerr << "\t -M   merger function: pseudo, simplex" << endl;
cerr << "\t -N   normalization method: immed, delayed, highest" << endl;
cerr << "\t -O   search order: pre, post" << endl;
cerr << "\t -L   inital node labels: const sink deficit" << endl;
```

Uses checkTree 158c, flow 167d, init 155a 170e, and saturate 171d.

199a ⟨*common option parsing* 199a⟩≡ (195 202 208a) 199b▷

```
case 'd':
    dumpNodes = true;
    break;
case 'f':
    writeFlow = true;
    break;
case 'g':
    relabelFreq = atof(optarg);
    break;
case 't':
    checkTree = true;
    break;
case 'x':
    tracingEnabled = true;
    break;
```

Uses `checkTree` 158c, `dumpNodes` 132a, and `writeFlow` 176d.

199b ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁199a 199c▷

```
case 'I':
    if (strcmp(optarg, "simple") == 0) {
        initFunc = &PhaseSolver::buildSimpleTree;
    } else if (strcmp(optarg, "path") == 0) {
        initFunc = &PhaseSolver::buildBlockingPathTree;
    } else if (strcmp(optarg, "saturate") == 0) {
        initFunc = &PhaseSolver::saturateAllArcs;
    } else if (strcmp(optarg, "greedy") == 0) {
        initFunc = &PhaseSolver::buildGreedyPathTree;
        // should also support greedy:N where N is numSplits
    } else if (strcmp(optarg, "shortest") == 0) {
        initFunc = &PhaseSolver::buildSpTree;
    } else {
        cerr << "Invalid initialization option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

Uses `buildGreedyPathTree` 67a, `buildSimpleTree` 56a, `buildSpTree` 72b, `numSplits` 129b, `PhaseSolver` 136c, `saturate` 171d, and `saturateAllArcs` 71b.

199c ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁199b 200a▷

```
case 's':
    numSplits = atoi(optarg);
    break;
```

Uses `numSplits` 129b.

200a ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁199c 200b▷

```
case 'M':
    if (strcmp(optarg, "simplex") == 0) {
        solver = new SimplexSolver();
    } else if (strcmp(optarg, "pseudo") == 0) {
        solver = new PhaseSolver();
    } else {
        cerr << "Invalid merge function option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

Uses `merge` 15a 37c, `PhaseSolver` 136c, and `SimplexSolver` 37a.

200b ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁200a 200c▷

```
case 'N':
    if (strcmp(optarg, "delayed") == 0) {
        solverFunc = &PhaseSolver::delayedNormalizeSolve;
    } else if (strcmp(optarg, "highest") == 0) {
        solverFunc = &PhaseSolver::highestLabelSolve;
    } else if (strcmp(optarg, "immed") == 0) {
        solverFunc = &PhaseSolver::solve;
    } else {
        cerr << "Invalid normalization option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

Uses `delayedNormalizeSolve` 29a, `highestLabelSolve` 34, `PhaseSolver` 136c, and `solve` 4b.

200c ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁200b 201a▷

```
case 'B':
    if (strcmp(optarg, "fifo") == 0) {
        addBranchFunc = &PhaseSolver::addBranchFifo;
    } else if (strcmp(optarg, "lifo") == 0) {
        addBranchFunc = &PhaseSolver::addBranchLifo;
    } else if (strcmp(optarg, "wave") == 0) {
        addBranchFunc = &PhaseSolver::addBranchWave;
    } else {
        cerr << "Invalid branch-management option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

Uses `addBranchFifo` 26c, `addBranchFunc` 26a, `addBranchLifo` 26c, and `PhaseSolver` 136c.

201a ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁200c 201b▷

```
case 'O':
    if (strcmp(optarg, "post") == 0) {
        postOrder = true;
    } else if (strcmp(optarg, "pre") == 0) {
        postOrder = false;
    } else {
        cerr << "Invalid pre/post order option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

201b ⟨*common option parsing* 199a⟩+≡ (195 202 208a) ◁201a

```
case 'L':
    if (strcmp(optarg, "const") == 0) {
        labelMethod = LABELS_CONSTANT;
    } else if (strcmp(optarg, "sink") == 0) {
        labelMethod = LABELS_SINK_DIST;
    } else if (strcmp(optarg, "deficit") == 0) {
        labelMethod = LABELS_DEFICIT_DIST;
    } else {
        cerr << "Invalid initial labeling option " << optarg << endl;
        usage();
        return 1;
    }
    break;
```

5.2 Parametric Solver

202 ⟨*pllps* 202⟩≡

```
// $Revision: 1.12 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "PhaseSolver.h"
⟨Timer declaration and implementation 216b⟩
⟨common declarations 194b⟩

void
usage()
{
    cerr << "Usage: pllps [-x] [-g freq] [-t] [-d] [-f] -V lam1,lam2,... [-I init] [-N norm] [-B
    ⟨common option usage 198b⟩
    cerr << "\t -V   specify values for lambda" << endl;
    cerr << "buildFlags: " << buildFlags << endl;
    cerr << "buildDate: " << buildDate << endl;
}

double*
getLambdas(int& numLambdas, char* str)
{
    // figure out how many elements - find each comma and replace
    // it with a null.
    int numCommas = 0;
    char* curr = index(str, ',');
    while (curr != nil) {
        numCommas++;
        *curr = '\0';
        curr++;
        curr = index(curr, ',');
    }
    numLambdas = numCommas + 1;
    double* result = new double[numLambdas];

    // parse each element with atof
    curr = str;
    for (int i = 0; i < numLambdas; i++) {
        result[i] = atof(curr);
        curr += strlen(curr) + 1;
    }

    return result;
}
```

```
int
main(int argc, char** argv)
{
    ⟨common main variables 198a⟩
    double* lambdaValues = nil;
    int numLambdas = 0;

    // parse arguments
    int ch;
    while ((ch = getopt(argc, argv, "dfg:txV:L:I:s:M:N:B:O:")) != EOF) {
        switch (ch) {
        ⟨common option parsing 199a⟩
        case 'V':
            lambdaValues = getLambdas(numLambdas, optarg);
            break;
        default:
            usage();
            return 1;
        }
    }

    argc -= optind;
    //argv += optind;

    if ((argc != 2) || (numLambdas < 1)) {
        usage();
        assert(0);
        return 1;
    }

    const char* instanceName = argv[optind];
    const char* outputName = argv[optind + 1];

    ofstream dout(outputName, ios::out);
    if (dout == nil) {
        cerr << "Unable to open output file: " << outputName << endl;
        return 1;
    }

    dout << "c" << endl;
    dout << "c  instance: " << instanceName << endl;
    dout << "c  buildFlags: " << buildFlags << endl;
    dout << "c  buildDate: " << buildDate << endl;
```

```
dout << "c  argv: ";
for (char** ap = argv; *ap != nil; ap++) {
    dout << *ap << " ";
}
dout << endl;

if (numSplits >= 0) {
    solver->maxSplits = numSplits;
}
if (relabelFreq >= 0.0) {
    solver->relabelFrequency = relabelFreq;
}
solver->postOrderSearch = postOrder;


time_t beginTime = time(0);
Timer initTimer;
initTimer.start();
bool readOK = solver->readDimacsInstance(instanceName);
initTimer.stop();

if (readOK == false) {
    dout << "c  CANNOT READ INSTANCE" << endl;
    return 1;
}

cout << "read problem instance OK" << endl;
Timer totalTimer;
totalTimer.start();

solver->setInitialCapacity(lambdaValues[0]);
// build initial tree
Timer treeTimer;
treeTimer.start();
(solver->*initFunc)(labelMethod);
treeTimer.stop();

// solve
Timer solveTimer;
solveTimer.start();
Timer* lambdaTimers = new Timer[numLambdas];

for (int i = 0; i < numLambdas; i++) {
    lambdaTimers[i].start();
```

```
        if (i > 0) {
            solver->setNextCapacityParameter(lambdaValues[i-1], lambdaValues[i]);
        }

        (solver->*solverFunc)(addBranchFunc);
        lambdaTimers[i].stop();
    }

    solveTimer.stop();

    // convert to flow
    Timer convertTimer;
    convertTimer.start();
    if (!dumpNodes) {
        // don't convert to flow if we're dumping the nodes,
        // which only makes sense after phase I before conversion
        solver->convertToFlow();
    }
    convertTimer.stop();

    totalTimer.stop();

    cout << "time to readInstance: " << initTimer << endl;
    cout << "time to establish tree: " << treeTimer << endl;
    cout << "done solving: " << solveTimer << endl;
    cout << "time to convert flow: " << convertTimer << endl;
    cout << "total time: " << totalTimer << endl;
    for (int i = 0; i < numLambdas; i++) {
        cout << "lambdaTime[" << i << "]: " << lambdaTimers[i] << endl;
    }

    time_t endTime = time(0);
    dout << "c  beginRun: " << ctime(&beginTime);
    dout << "c  endRun: " << ctime(&endTime);

    dout << "c  timeToRead: " << initTimer << endl;
    dout << "c  timeToInitialize: " << treeTimer << endl;
    dout << "c  timeToSolve: " << solveTimer << endl;
    dout << "c  timeToFlow: " << convertTimer << endl;
    dout << "c  totalTime: " << totalTimer << endl;
    for (int i = 0; i < numLambdas; i++) {
        dout << "c  lambdaTime[" << i << "]: " << lambdaTimers[i] << endl;
    }
```

```
    solver->writeStats(dout, true);

    if (dumpNodes) {
        solver->dumpNodes(dout);
        solver->writeDimacsFlow(dout);
    } else if (writeFlow) {
        solver->writeDimacsFlow(dout);
    }

    dout.close();

    return 0;
}
```

Uses addBranchFunc 26a, convertToFlow 95a, dumpNodes 132a, flow 167d, init 155a 170e, maxSplits 67b, nodes 3a, numSplits 129b, PhaseSolver 136c, readDimacsInstance 120d, relabelFrequency 77d, setInitialCapacity 105 172c, setNextCapacityParameter 106b, solve 4b, start 214b, stop 214b, Timer 216b, writeDimacsFlow 128b 131a, writeFlow 176d, and writeStats 128e.

5.3 General Parametric Solver

206 ⟨*gpps* 206⟩≡ 207▷

```
// $Revision: 1.12 $, $Date: 2003/09/15 18:39:40 $
// This C++ code was generated by noweb from the corresponding .nw file
#include "PhaseSolver.h"
⟨Timer declaration and implementation 216b⟩

⟨common declarations 194b⟩
```

Uses PhaseSolver 136c.

207 ⟨*gpps* 206⟩+≡ ◁206 213a▷

```
void
usage()
{
    cerr << "Usage: llps [-d] [-f] [-t] [-x] [-g freq][-I init] [-N norm] [-B branch] input-g
    ⟨common option usage 198b⟩
    cerr << "buildFlags: " << buildFlags << endl;
    cerr << "buildDate: " << buildDate << endl;
}

int
main(int argc, char** argv)
{
    ⟨common main variables 198a⟩
    ⟨parse gpps arguments 208a⟩
    ⟨gpps solving 209a⟩
    return 0;
}
```

Uses `flow` 167d and `init` 155a 170e.

208a ⟨*parse gpps arguments* 208a⟩≡ (207) 208b▷

```
int ch;
while ((ch = getopt(argc, argv, "dfg:txI:s:M:N:B:O:L:")) != EOF) {
    switch (ch) {
    ⟨common option parsing 199a⟩
    /*
    case 'F':
        parameterFiles = getParamFileList(numParaeterFiles, optarg);
        break;
     */
    default:
        usage();
        return 1;
    }
}

if (numSplits >= 0) {
    solver->maxSplits = numSplits;
}
if (relabelFreq >= 0.0) {
    solver->relabelFrequency = relabelFreq;
}
solver->postOrderSearch = postOrder;
```

Uses `maxSplits` 67b, `numSplits` 129b, and `relabelFrequency` 77d.

Get the input files and the output file.

208b ⟨*parse gpps arguments* 208a⟩+≡ (207) ◁208a

```
argc -= optind;

if (argc < 2) {
    usage();
    return 1;
}

int numInputFiles = argc - 1;
char** inputFiles = (char**)calloc(numInputFiles, sizeof(char*));
for (int i = 0; i < numInputFiles; i++) {
    inputFiles[i] = argv[optind + i];
}

const char* outputName = argv[optind + numInputFiles];
```

209a ⟨*gpps solving* 209a⟩≡ (207) 209c▷
⟨*open output and write header* 209b⟩

209b ⟨*open output and write header* 209b⟩≡ (209a)
```
ofstream dout(outputName, ios::out);
if (dout == nil) {
    cerr << "Unable to open output file: " << outputName << endl;
    return 1;
}

dout << "c" << endl;
//dout << "c  instance: " << instanceName << endl;
dout << "c  buildFlags: " << buildFlags << endl;
dout << "c  buildDate: " << buildDate << endl;
dout << "c  argv: ";
for (char** ap = argv; *ap != nil; ap++) {
    dout << *ap << " ";
}
dout << endl;
```

Read the initial instance

209c ⟨*gpps solving* 209a⟩+≡ (207) ◁209a 210▷
```
time_t beginTime = time(0);
Timer readTimer;
readTimer.start();
bool readOK = solver->readDimacsInstance((const char*)inputFiles[0]);
readTimer.stop();

if (readOK == false) {
    dout << "c  CANNOT READ INSTANCE" << endl;
    return 1;
}

cout << "read problem instance OK" << endl;
```

Uses `readDimacsInstance` 120d, `start` 214b, `stop` 214b, and `Timer` 216b.

Build the initial tree.

210 ⟨*gpps solving* 209a⟩+≡ (207) ◁209c 211▷

```
Timer totalTimer;
totalTimer.start();                // total time is everything except I/O

Timer treeTimer;
Timer* cutTimers = new Timer[numInputFiles];
Timer* solveTimers = new Timer[numInputFiles];
treeTimer.start();
cutTimers[0].start();
(solver->*initFunc)(labelMethod);
treeTimer.stop();

// solve
solveTimers[0].start();
(solver->*solverFunc)(addBranchFunc);
solveTimers[0].stop();
cutTimers[0].stop();
cout << "stats after solve 0" << endl;
solver->writeStats(cout, false);
```

Uses `addBranchFunc` 26a, `solve` 4b, `start` 214b, `stop` 214b, `Timer` 216b, and `writeStats` 128e.

Solve the remaining instances.

211 ⟨*gpps solving* 209a⟩+≡ (207) ◁210 212▷

```
for (int i = 1; i < numInputFiles; i++) {
    bool ok = solver->readNewCapacities((const char*)inputFiles[i]);
    if (ok) {
        cutTimers[i].start();
        solveTimers[i].start();
        solver->renormalizeTree(labelMethod);
        (solver->*solverFunc)(addBranchFunc);
        //solver->highestLabelResolve(addBranchFunc);
        solveTimers[i].stop();
        cutTimers[i].stop();
        cout << "stats after solve " << i << endl;
        solver->writeStats(cout, false);
    } else {
        cerr << "Cannot read input file: " << inputFiles[i] << endl;
        break;
    }
}
```

Uses `addBranchFunc` 26a, `highestLabelResolve` 119a, `readNewCapacities` 127a, `renormalizeTree` 115a, `solve` 4b, `start` 214b, `stop` 214b, and `writeStats` 128e.

Convert the final solution to a flow.

212 ⟨*gpps solving* 209a⟩+≡ (207) ◁211

```
Timer convertTimer;
convertTimer.start();
if (!dumpNodes) {
    // don't convert to flow if we're dumping the nodes,
    // which only makes sense after phase I before conversion
    solver->convertToFlow();
}
convertTimer.stop();

totalTimer.stop();

cout << "time to readInstance: "    << readTimer << endl;
cout << "time to establish tree: "  << treeTimer << endl;
cout << "time to convert flow: "    << convertTimer << endl;
cout << "total time: "              << totalTimer << endl;
Timer cutTimer, solveTimer;
for (int i = 0; i < numInputFiles; i++) {
    cout << "cutTimers[" << i << "]: " << cutTimers[i] << endl;
    cutTimer += cutTimers[i];
    cout << "solveTimers[" << i << "]: " << solveTimers[i] << endl;
    solveTimer += solveTimers[i];
}
cout << "cutTime: "         << cutTimer << endl;

time_t endTime = time(0);
dout << "c  beginRun: "             << ctime(&beginTime);
dout << "c  endRun: "               << ctime(&endTime);

dout << "c  timeToRead: "           << readTimer << endl;
dout << "c  timeToInitialize: "     << treeTimer << endl;
dout << "c  timeToFlow: "           << convertTimer << endl;
dout << "c  totalTime: "            << totalTimer << endl;
dout << "c  cutTime: "              << cutTimer << endl;
for (int i = 0; i < numInputFiles; i++) {
    dout << "c  cutTimers[" << i << "]: " << cutTimers[i] << endl;
    dout << "c  solveTimers[" << i << "]: " << solveTimers[i] << endl;
}

solver->writeStats(dout, true);

dout.close();
```

Uses convertToFlow 95a, dumpNodes 132a, flow 167d, nodes 3a, start 214b, stop 214b, Timer 216b, and writeStats 128e.

213a ⟨*gpps* 206⟩+≡ ◁207

```
/*
char**
getParameterNames(int& numParameters, char* str)
{
    // figure out how many elements - find each comma and replace
    // it with a null.
    int numCommas = 0;
    char* curr = index(str, ',');
    while (curr != nil) {
        numCommas++;
        *curr = '\0';
        curr++;
        curr = index(curr, ',');
    }
    numParameters = numCommas + 1;
    char** result = new char*[numParameters];

    // parse each element with atof
    curr = str;
    for (int i = 0; i < numParameters; i++) {
        result[i] = curr;
        curr += strlen(curr) + 1;
    }

    return result;
}
*/
```

5.4 Timer

The Timer is used to store timing information during the running of an algorithm. The operations that one performs are: start the timer, stop it, and get the times.

213b ⟨*Timer functions* 213b⟩≡ (216b) 215d▷

```
public:
    void start();
    void stop();
    void getTimes(float& wall, float& user, float& sys) const;
```

Uses getTimes 214c, start 214b, and stop 214b.

We will store CPU times (user and system) and ellapsed wall clock time. We need both the start and stop times for these.

214a ⟨*Timer data* 214a⟩≡ (216b)

```
private:
    struct tms          startCpu;
    clock_t             startWall;
    struct tms          stopCpu;
    clock_t             stopWall;
```

Defines:
`startCpu`, used in chunks 214–16.
`startWall`, used in chunks 214–16.
`stopCpu`, used in chunks 214–16.
`stopWall`, used in chunks 214–16.

To start or stop a timer, we use the `times(2)`, at least on Unix.

214b ⟨*Timer implementation* 214b⟩≡ (216c) 214c▷

```
void
Timer::start()
{
    startWall = times(&startCpu);
}
void
Timer::stop()
{
    stopWall = times(&stopCpu);
}
```

Defines:
`start`, used in chunks 195, 202, and 209–13.
`stop`, used in chunks 103b, 195, 202, and 209–13.
Uses `startCpu` 214a, `startWall` 214a, `stopCpu` 214a, `stopWall` 214a, and `Timer` 216b.

To get the time difference between the start and stop times, we just perform subtraction.

214c ⟨*Timer implementation* 214b⟩+≡ (216c) ◁214b 215c▷

```
void
Timer::getTimes(float& wall, float& user, float& sys) const
{
    wall = (stopWall - startWall) / (float)CLK_TCK;
    user = (stopCpu.tms_utime - startCpu.tms_utime) / (float)CLK_TCK;
    sys  = (stopCpu.tms_stime - startCpu.tms_stime) / (float)CLK_TCK;
}
```

Defines:
`getTimes`, used in chunks 213b and 215c.
Uses `startCpu` 214a, `startWall` 214a, `stopCpu` 214a, `stopWall` 214a, and `Timer` 216b.

We also want to implement an `operator<<`, to print the times onto an output stream.

215a ⟨*C++ overhead* 134b⟩+≡ (136c 164b 179a 216b) ◁175b 215f▷
```
friend ostream& operator<<(ostream& out, const Timer& timer);
```
Uses `Timer` 216b.

215b ⟨*header include files* 128c⟩+≡ (135d 164b) ◁140c
```
class ostream;
```

215c ⟨*Timer implementation* 214b⟩+≡ (216c) ◁214c 215e▷
```
ostream&
operator<<(ostream& out, const Timer& timer)
{
    float wall, user, sys;
    timer.getTimes(wall, user, sys);
    out << "[" << wall << " wall, "
        << user << " user, "
        << sys << " sys]";
    return out;
}
```
Uses `getTimes` 214c and `Timer` 216b.

For general sensitivty analysis, we want to sum a series of timers. We do this by leaving the start time at zero and accumulating the time in the stop time.

215d ⟨*Timer functions* 213b⟩+≡ (216b) ◁213b
```
Timer& operator+=(const Timer& other);
```
Uses `Timer` 216b.

215e ⟨*Timer implementation* 214b⟩+≡ (216c) ◁215c 216a▷
```
Timer& Timer::operator+=(const Timer& other)
{
    stopWall += other.stopWall - other.startWall;
    stopCpu.tms_utime += (other.stopCpu.tms_utime - other.startCpu.tms_utime);
    stopCpu.tms_stime += (other.stopCpu.tms_stime - other.startCpu.tms_stime);
    return *this;
}
```
Uses `startCpu` 214a, `startWall` 214a, `stopCpu` 214a, `stopWall` 214a, and `Timer` 216b.

Of course we need to construct and initialize timers.

215f ⟨*C++ overhead* 134b⟩+≡ (136c 164b 179a 216b) ◁215a
```
public: Timer();
```
Uses `Timer` 216b.

216a ⟨*Timer implementation* 214b⟩+≡ (216c) ◁215e

```
Timer::Timer()
{
    stopWall = startWall = 0;
    stopCpu.tms_utime = startCpu.tms_utime = 0;
    stopCpu.tms_stime = startCpu.tms_stime = 0;
}
```

Uses `startCpu` 214a, `startWall` 214a, `stopCpu` 214a, `stopWall` 214a, and `Timer` 216b.

216b ⟨*Timer declaration and implementation* 216b⟩≡ (194a 202 206) 216c▷

```
#include <time.h>
#include <sys/times.h>
#include <iostream.h>
class Timer
{
    ⟨Timer functions 213b⟩
    ⟨Timer data 214a⟩
    ⟨C++ overhead 134b⟩
};
```

Defines:
`Timer`, used in chunks 195, 202, 209c, 210, 212, and 214–16.

216c ⟨*Timer declaration and implementation* 216b⟩+≡ (194a 202 206) ◁216b

```
⟨Timer implementation 214b⟩
```

6 List of all chunks from all files

⟨*Solver public declarations* 94b⟩
⟨*specify source/sink* 124⟩
⟨*strong side bottleneck* 39b⟩
⟨*terminate early based on counting labels* 5a⟩
⟨*Timer data* 214a⟩
⟨*Timer declaration and implementation* 216b⟩
⟨*Timer functions* 213b⟩
⟨*Timer implementation* 214b⟩
⟨*update excesses at endpoints* 54b⟩
⟨*verify no merger available* 12a⟩
⟨*visit neighbor node* 103b⟩
⟨*weak side bottleneck* 40e⟩
⟨*write flow amount* 130b⟩
⟨*write statistics* 129a⟩

7 Index