Module Graph Algorithms

This module provides several graph algorithms to be used with network optimization codes. For example, a connected-components labeling routine is provided for extracting a single connected component from a graph. Also provided are very efficient novel routines for manipulating spanning trees related to network optimization. Finally, some routines related to updating minimal spanning trees under small perturbations to the weights are provided, as well as routines for conversion from heads-and-tails format into the more commonly used adjacency-list-based data structures.

A spanning tree in the sense used here is a subset of the arcs that contains no cycles, and has a chosen root node. Once the root is known, parenthood relations are determined uniquely. In this library, parenthood relations are usually stored in the array called parents (parents (n_special_nodes : n_nodes)), where the parent of each node is in fact a pointer (the index) of the parent arc, i.e. the arc that connects the node to its parent. Since our graph is stored as usual in a heads-tails array, we need to know whether the head is the parent, or the tail of the parent arc is the parent. Therefore each parent arc has its orientation, usually stored in the array orientations (orientations (n_special_nodes : n_nodes)), which can be one of the predefined public parameters head_is_parent, tail_is_parent, or no_parent (for example, for root node arcs). Root nodes should not have a parent, and this is especially important if dealing with spanning forests, which this library can handle in most routines. One would usually use the sign of the entry in parents to denote the orientation, but since I allow for arcs with negative numbers (special arcs), I do not do this in this code.

The parenthood relations described above uniquely determine a tree and a way to go from a node up to the root of the tree. A final important data structure in manipulating the tree is a thread-like ordering of the nodes, which is a permutation of the nodes, usually stored in the array level_ordering (level_ordering (0 : n_nodes + n_special_nodes)) in these codes. The requirement here is that every node comes after its parent in level_ordering. In other words, by traversing the nodes in forward order, one would always visit the parent before visiting the child. Vice versa, by traversing the nodes in the reverse-thread order, one would always visit all children of a node before visiting the node itself. These are crucial operations in traversing a tree when calculating flows or potentials in network flow algorithms. Please note that this array is not the usual thread-order defined by computer scientists. My definition is a very loose one, and so the ordering is not unique. I have developed a way to create this ordering in a two-pass procedure over the nodes of the tree, as given in the routine BuildTreeStructures, suited for a serial computer.

Finally, in maintaining and working with the trees, two more auxiliary node labels are used. One is the cardinality label for the nodes, which gives the number of nodes in the subtree rooted at the given node, counting the node itself, and is usually stored in cardinalities (cardinalities (n_special_nodes : n_nodes)). When talking about minimal or maximal weight spanning trees, denoted by MST from now on (I support both min and max ones, not just the usual min-weight trees), we have weights on the arcs, and the path node labels array path_labels (path_labels (n_special_nodes : n_nodes)) stores either the minimal (for MaxST) or maximal (for MinST) weight on the path from the node to the root. These paths do not have to be exact, only exact upper or lower bounds.

To tell the codes what kind of tree one wants to build or manipulate, the predefined parameters any_tree (meaning an arbitrary spanning tree), random_tree (which is like any_tree, only a different tree will be generated each time because an expansive random permutation of the arcs is used), min_cost_tree (for MinST) or max_cost_tree (for MaxST). These are further used in the front-end module Network_Spanning_Trees. Finally, the logical flag debug_graph_algs tells whether some useful statistics should be gathered (not complete yet).

The list of routines in this module along with a short description (please look below for more details) is:

1
ConnectedComponents labels each node with a number that is a chosen unique node from the connected component it belongs to.

BuildSpanningTree makes a spanning tree or forest by using the union-find algorithm. It accepts a list of arcs to consider in entering the tree, and checks whether each arc forms a cycle, and includes it in the tree if it does. This allows great flexibility in using this routine. For example, on can start with half-made trees and add a collection of arcs. Or, one can order the arcs according to weight and then build the MST, which is Kruskal’s algorithm for building MST’s. But remember to always initialize the tree parenthood relations to whatever it is that you want to start from. This routine creates just the parenthood relations for the tree.

BuildTreeStructures completes a spanning tree with the cardinality labels and creates a thread like level ordering of the nodes. The cardinalities may be provided as input if they are known, and this speeds the routine (for example, the routine UpdateSpanningTree maintains cardinality labels exactly, so these do not need to be rebuilt).

CalculatePathLabels calculates the path labels for the nodes in the weighted spanning tree.

UpdateSpanningTree takes a tree and arcs weights and a list of arcs to consider for entering either the minimal or maximal spanning tree. It can be used to update MST’s after small perturbation of the weights, or to build an MST from scratch by starting with a random tree. It uses the cardinality and parenthood relations to traverse Basic Equivalent Path (BEP) cycles and performs arc pivots.

PropagateNodesPotentials propagates potentials from the root of a tree down to the leaf nodes, given potentials for the roots and potential drops across each of the arcs. In terms of network optimization, this is a multiplication of an arc vector with the matrix \( B^{-T} \), where \( B \) is the basis for the node-arc incidence matrix. This routine can deal with forests as well.

PropagateArcsFlows propagates flows from the leaf nodes to the root by using the given supplies-demands on each node. This is in fact multiplication of a nodal vector with \( B^{-1} \).

"WEAVE.f90" 1.1 ≡

MODULE Graph_Algorithms
   USE Precision
   USE Error_Handling
   USE System_Monitors
   USE Sorting_Ranking
   USE Network_Matrix_Operations
   IMPLICIT NONE
PUBLIC :: ConnectedComponents, BackboneCycles, ContractMatchedArcs, BuildSpanningTree,
   BuildTreeStructures, PropagateNodesPotentials, PropagateArcsFlows, CalculatePathLabels,
   ReBuildSpanningTree, BuildNetworkMST, BasicDualNewtonSystem, EffectiveCycleResistances,
   FactorizeIncompleteQR, FactorizeIncompleteLDLt, CreateAdjacencyArrays,
   CalculateCongestionsDilations
PRIVATE

LOGICAL, PUBLIC, SAVE :: debug_graph_algs = F  // Whether to print useful statistics
INTEGER (KIND = I BYTE), PARAMETER, PUBLIC :: no_parent = 0, head_is_parent = 1,
   tail_is_parent = 2  // Possible orientations for parenthood relations
INTEGER (KIND = i\textunderscore byte), PARAMETER, PUBLIC :: any\textunderscore tree = 1, random\textunderscore tree = 2, min\textunderscore cost\textunderscore tree = 3, max\textunderscore cost\textunderscore tree = 4  // Types of spanning trees
INTEGER (KIND = i\textunderscore byte), PARAMETER, PUBLIC :: matching\textunderscore node = 2, dead\textunderscore node = 0  // Statuses used in matching code

CONTAINS

(ConnectedComponents 2.1.2)
(BackboneCycles 2.2.1)
(Contract\textunderscore Matched\textunderscore Arcs 3.1)
(Build\textunderscore Spanning\textunderscore Tree 4.1.1)
(Build\textunderscore Tree\textunderscore Structures 4.2.1)
(Propagate\textunderscore Nodes\textunderscore Potentials 5.1.3)
(Propagate\textunderscore Arcs\textunderscore Flows 5.2.1)
(Calculate\textunderscore Path\textunderscore Labels 6.1.1)
(Re\textunderscore Build\textunderscore Spanning\textunderscore Tree 6.2.1)
(Build\textunderscore Network\textunderscore MST 7.1.1)
(Basic\textunderscore Dual\textunderscore Newton\textunderscore System 7.4.1)
(Effective\textunderscore Cycle\textunderscore Resistances 7.5.1)
(Factorize\textunderscore Incomplete\textunderscore QR 7.2.1)
(Factorize\textunderscore Incomplete\textunderscore LDLt 7.3.1)
(Create\textunderscore Adjacency\textunderscore Arrays 8.1.1)
(Calculate\textunderscore Congestions\textunderscore Dilations 8.2.1)

END MODULE Graph\textunderscore Algorithms

2 Graph Connectivity Labeling

The routine \textit{ConnectedComponents} takes a graph and just labels the nodes according to the connected cluster they belong to, where each cluster get's its own label. The somewhat more expensive routine \textit{Backbone\textunderscore Cycles} is a bit more detailed and labels each cluster of cycles (blob in some terminology) with a unique label, so that dangling ends and the percolating backbone can be identified in a given graph. The first routine uses plain union-find algorithm as its basis, while backbone indentification requires both a union-find and a matching algorithm.

2.1 Labeling Connected Clusters

This procedure will label the connected components of a graph with integers (not controllable at this point) and possibly return the label for the largest connected-component. Therefore, all nodes that are connected to each other will be given the same integer label in the array \textit{labels} on exit, and this integer will be returned for the largest component in the optional argument \textit{largest\textunderscore component\textunderscore label} along with (optionally) the size of this component in \textit{largest\textunderscore component\textunderscore size}.

The algorithm can start with already partially labeled graphs–forests (for example, when the edges are processed in batches, as in a parallel system where inter-processor edges must be handled separately), in
which case use_labels should be set to \( T \) (default is \( F \)). Otherwise, it starts from a forest of all single-node trees and builds from there. But notice that here labels uses the root node for each connected-component tree as the label, and this can not be changed. So not just any integers can be used as labels. This was done in order to avoid using two arrays, one for the labels, the other for the parenthood relations, but it can easily be changed.

The following two macros perform two basic operation in union-find algorithms and basically and tree algorithm–they find the root of a given node node using the parenthood relations in the array parents. The second macro performs path compression along the path to the root:

"WEAVE.f90" 2.1.1 ≡

```fortran
@m _FindRoot(parents, node, root, climber)
  climber = node  // Start at the node
  DO WHILE (parents_climber ≠ climber)  // Until you reach a node that’s its own parent
    climber = parents_climber  // Climb one level up the tree
  END DO
  root = climber

@m _CompressPath(parents, node, root, climber, climber_parent)
  climber = node  // Start at the node
  DO WHILE (parents_climber ≠ root)  // Until you reach a node that’s its own parent
    climber_parent = parents_climber  // Climb one level up the tree
    parents_climber = root  // Jump pointer to root
    climber = climber_parent
  END DO
```

Here is the routine ConnectedComponents. The array labels is of dimension \((-\text{node}._\text{offset} :)\) and should be addressed as such in the given heads-and-tails array heads.tails. This is not to be confused or the code will crash!

\(\langle\text{ConnectedComponents 2.1.2}\rangle ≡

```
SUBROUTINE ConnectedComponents(heads.tails, node.offset, labels, use_labels,
  largest.component.label, largest.component.size)
  IMPLICIT NONE
  INTEGER (kind = i.up), DIMENSION (:), INTENT (IN) :: heads.tails  // Heads-tails array for G
  INTEGER (kind = i.up), INTENT (IN) :: node.offset  // Lower bound for nodal arrays
  INTEGER (kind = i.up), DIMENSION (-node.offset :), INTENT (INOUT) :: labels
    // With displaced counting
  LOGICAL, INTENT (IN), OPTIONAL :: use_labels
  INTEGER (kind = i.up), INTENT (OUT), OPTIONAL :: largest.component.label, largest.component.size
    // Largest component, if needed
  INTEGER (kind = i.up), DIMENSION (:), ALLOCATABLE :: heights
  INTEGER (kind = i.up) :: n_nodes, n_special_nodes, n_arcs
  INTEGER (kind = i.up) :: node, arc, head, tail, head_root, tail_root, root, diff_heights, child, parent,
    largest_component, climber, climber_parent
  INTEGER :: alloc_status
  LOGICAL :: reuse_labels
  n_special_nodes = -LBOUND(labels, i.up, DIM = 1)
```

4
\texttt{n\_nodes} = \texttt{JBOUND(labels, i\_wp, DIM = 1)}  // The number of nodes, \( n \)
\texttt{n\_arcs} = \texttt{SIZE(heads\_tails, i\_wp, DIM = 2)}  // \( m \)

\texttt{reuse\_labels} = \texttt{\_F}
\textbf{IF} (\texttt{PRESENT(use\_labels)})
\texttt{\_reuse\_labels} = \texttt{use\_labels}
\textbf{IF} (\texttt{\_!reuse\_labels}) \textbf{THEN}
\texttt{\_node} = \texttt{\_!n\_special\_nodes, n\_nodes}
\texttt{labels(\_node) = \_node}  // Start with forest of all nodes
\textbf{END \_DO}
\textbf{END IF}

\texttt{\_ALLOCATE(heights = \_!n\_special\_nodes : n\_nodes, stat = alloc\_status)}
\texttt{\_CALL RecordAllocation(n\_elements = n\_nodes + n\_special\_nodes + 1, mold = 1\_i\_wp,}
\texttt{\_caller = "ConnectedComponents", alloc\_status = alloc\_status)}
\texttt{\_heights = 0}  // Initialize heights

\texttt{\_FOR All\_Ares: \_DO} \texttt{arc = 1, n\_arcs}  // Not masked for now
\texttt{\_head = heads\_tails1, arc}
\texttt{\_tail = heads\_tails2, arc}
\hfill \texttt{\_Find\_Root(labels, head, head\_root, climber)}
\hfill \texttt{\_Find\_Root(labels, tail, tail\_root, climber)}
\hfill \texttt{\_Compress\_Path(labels, head, head\_root, climber, climber\_parent)}
\hfill \texttt{\_Compress\_Path(labels, tail, tail\_root, climber, climber\_parent)}

\texttt{\_Tree\_Union: \_IF (head\_root \_! tail\_root) \_THEN}  // Perform union operation
\texttt{\_diff\_heights = (heights head\_root \_! heights tail\_root)}
\texttt{\_IF (diff\_heights > 0) \_THEN}  // Choose which tree merges with which
\texttt{\_parent = head\_root}
\texttt{\_child = tail\_root}
\texttt{\_ELSE}
\texttt{\_parent = tail\_root}
\texttt{\_child = head\_root}
\texttt{\_END \_IF}
\texttt{\_labels(child) = parent}  // Hook the two trees-union
\texttt{\_IF (diff\_heights \_! 0)}
\texttt{\_heights parent = heights parent + 1}  // Otherwise heights are not-changed by union operation
\texttt{\_END \_IF \_Tree\_Union}
\texttt{\_END \_DO \_FOR All\_Ares}

\texttt{\_FOR All\_Nodes: \_DO \_node = \_!n\_special\_nodes, n\_nodes}  // Final compression of all paths
\hfill \texttt{\_Find\_Root(labels, node, root, climber)}
\hfill \texttt{\_Compress\_Path(labels, node, root, climber, climber\_parent)}
\texttt{\_END \_DO \_FOR All\_Nodes}

\texttt{/* We sometimes need to extract the largest component, so here I find this component and its}
\texttt{label. This will make who ever uses this routine much happier in many cases, as sorting would}
\texttt{be needed to determine this in the general case. */}
\texttt{\_heights = 0}  // Tree sizes will be stored here
\texttt{\_DO \_node = \_!n\_special\_nodes, n\_nodes}  // Count tree sizes
\texttt{\_heights(labels node) = heights(labels node) + 1}
\texttt{\_END \_DO}
\texttt{\_IF (PRESENT(largest\_component\_label) | PRESENT(largest\_component\_size)) \_THEN}
\texttt{\_largest\_component = \_MAXLOC(heights, i\_wp)}  // Assumes numbering starting at 1!
largest_component = largest_component - 1 - node_offset  // Correct for numberin
IF (PRESENT(largest_component_label))
    largest_component_label = labels(largest_component_1)
    IF (PRESENT(largest_component_size))
        largest_component_size = heights(largest_component_1)
    END IF
    CALL RecordAllocation(n_elements = -SIZE(heights, i_wp), mold = 1_i_wp)
    DEALLOCATE(heights)
END SUBROUTINE ConnectedComponents

This code is used in section 1.1.

2.2 Condensing Cycles

The structure of this routine is almost the same as ConnectedComponents, only now each backbone gets it's own label, as opposed to each connected cluster. These labels are returned in the array supernodes, and one can establish preexisting supernodes by passing use supernodes = T. The algorithm that is used is a relatively novel approach and it is based on previous work in rigidity percolation. It is essentially based on matching, and is an algorithm in which unique paths are established by making a forest in the network by adding the arcs one-by-one.

Before adding an arc we must check whether there is a cycle formed in this way in the network. This is done by reversing the parenthesis relations (stored here in the array cycles) on the path from the head to the root of its forest tree (called a supernode here) and then try to do the same with the tail. If there is a cycle through the network, then this will identify it, if not, the edge can be added to the cycles. Once a cycle is identified, the nodes in the cycle are merged together into a supernode using the famed union-find algorithm (set relations here are stored in the array supernodes). This process is extremely efficient and so clever, but there is little documentation of it here or elsewhere. Contact me if it sounds interesting.

Note that this particular version does some redundant operations for beauty and simplicity's sake, and is not fully optimized.

{BackboneCycles 2.2.1} ≡

SUBROUTINE BackboneCycles(node_offset, heads_tails, supernodes, use supernodes,
    largest supernode, largest supernode size)
IMPLICIT NONE
INTEGER(KIND = i_wp), INTENT(IN) :: node_offset       // Lower bound for nodal arrays
INTEGER(KIND = i_wp), DIMENSION(:, :), INTENT(IN) :: heads_tails   // Heads-tails array
INTEGER(KIND = i_wp), DIMENSION(-node_offset :), INTENT(INOUT) :: supernodes  // Cycle supernodes
LOGICAL, INTENT(IN), OPTIONAL :: use supernodes       // Are there already some supernodes?
INTEGER(KIND = i_wp), INTENT(OUT), OPTIONAL :: largest supernode, largest supernode size
INTEGER(KIND = i_wp), DIMENSION(:), ALLOCATABLE :: cycles, heights  // Path labels for tracing cycles and the heights of the union-find trees
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_arcs // Counters
INTEGER (KIND = i_wp) :: node, arc, head, tail, counter, max_height, supernode, last_supernode,
                   previous_supernode, next_supernode, child, parent, head_length, tail_length, head_supernode,
                   tail_supernode, largest_location, climber, climber_parent, supernode
INTEGER :: alloc_status
LOGICAL :: reuse_supernodes, cycle_closed, increase_height

n_special_nodes = -_BOUND(supernodes, i_wp, DIM = 1)
n_nodes = _BOUND(supernodes, i_wp, DIM = 1) // The number of nodes, n
n_arcs = _SIZE(heads_tails, i_wp, DIM = 2) // m
reuse_supernodes = F
IF (PRESENT(use_supernodes)) reuse_supernodes = use_supernodes
IF (~reuse_supernodes) THEN
  DO node = -n_special_nodes, n_nodes
    supernodes_node = node // Start with forest of all nodes
  END DO
END IF

ALLOCATE (cycles(-n_special_nodes : n_nodes), stat = alloc_status)
CALL RecordAllocation (n_elements = n_nodes + n_special_nodes + 1, mold = l_i_wp,
                  caller = "ConnectedComponents", alloc_status = alloc_status)
DO node = -n_special_nodes, n_nodes
  cycles_node = node // Start with no cycles
END DO

ALLOCATE (heights(-n_special_nodes : n_nodes), stat = alloc_status)
CALL RecordAllocation (n_elements = n_nodes + n_special_nodes + 1, mold = l_i_wp,
                  caller = "ConnectedComponents", alloc_status = alloc_status)
heights = 0 // Initialize heights

ForAllArcs: DO arc = 1, n_arcs // Not masked for now!
  head = heads_tails_1, arc
  tail = heads_tails_2, arc

  UncoverNode(head, supernodes, cycles, head, head_supernode, last_supernode, head_length,
             supernode, increase_height) // Uncover the head first and see if a cycle was formed

  UncoverNode(tail, supernodes, cycles, tail, tail_supernode, last_supernode, tail_length,
             supernode, increase_height) // Now also uncover the tail

/* We know that the superroots on the cycle have already been compressed, so that we only
  need to climb at most one step up to find the supernode; */
AddArc: IF (last_supernode == head_supernode) THEN // Condense the identified cycle
  supernode = head_supernode
END CondenseCycle: DO // Trace back the cycle
  supernodes_supernode = supernode // Label all with the shortest supernode
  IF (supernode == tail_supernode) EXIT CondenseCycle // Cycle was traced
  supernode = cycles_supernode // Move along the cycle (already compressed)
END DO CondenseCycle
IF (increase_height) heights_supernode = heights_supernode + 1
ELSE
  IF (head_length >= tail_length) THEN // Try to make as short of cycles as possible
    cycles_head_supernode = tail_supernode // Add the arc to the cycles
  ELSE
    cycles_tail_supernode = head_supernode // Add this arc to the cycles
END IF
END IF

END IF AddArc

END DO ForAllArcs

CALL RecordAllocation(n_elements = -SIZE(cycles, i_wp), mold = 1_i_wp)
DEALLOCATE(cycles)

ForAllNodes: DO node = -n_special_nodes, n_nodes  // Final compression of all supertrees
  FindRoot(supernodes, node, supernode, climber)
  CompressPath(supernodes, node, supernode, climber, climber_parent)
END DO ForAllNodes

/* We sometimes need to extract the largest component, so here I find this component and its
supernode label. This will make who ever uses this routine much happier in many cases, as
sorting would be needed to determine this in the general case. */
heights = 0  // Tree sizes will be stored here
DO node = -n_special_nodes, n_nodes  // Count tree sizes
  heights(supernodes_node) = heights(supernodes_node) + 1
END DO

IF (PRESENT(largest_supernode) | PRESENT(largest_supernode_size)) THEN
  largest_location = MAXLOC(heights, i_wp)  // Assumes numbering starting at 1!
  largest_location = largest_location - 1 - node_offset  // Correct for numbering
  IF (PRESENT(largest_supernode))
    largest_supernode = supernodes(largest_location)
  IF (PRESENT(largest_supernode_size))
    largest_supernode_size = heights(largest_location)
END IF

CALL RecordAllocation(n_elements = -SIZE(heights, i_wp), mold = 1_i_wp)
DEALLOCATE(heights)

END SUBROUTINE BackboneCycles

This code is used in section 1.1.1.

The following macro is a crucial operation in traversing the potential cycles formed by additions of arcs. It starts at node and it traverses its cycle until it reaches an uncovered node, i.e. a parent supernode. Notice that each node in the cycle has its supernode, and all operations must be performed on the supernodes to avoid updating the nodes all the time. Also notice that full path-compression is performed here, and in my experience this is a good optimization. I also decided to use height labels for the supernode union-sets, even though these do not bring any improvement and demand some extra memory. The nice thing about using heights in union-find is that they guarantee a logarithmic time bound, and may be useful in pathological graphs.

"WEAVE.f90" 22.2 ≡
@m UncoverNode(ID, supernodes, cycles, node, first Supernode, last_supernode, path_length, supernode, equal_heights)

path_length = 0
max_height = -HUGE(1_i_wp)

FindRoot(supernodes, node, supernode, climber)  // Find the first cluster supernode
CompressPath(supernodes, node, supernode, climber, climber_parent)  // Shorten supertree
increase$\_height = \mathcal{F}$
previous$\_supernode = supernode$  // Start from uncovered node
first$\_supernode = supernode$

$Uncover$.@&ID: DO  // Reverse the cycle path of this node
 IF ($heights$ $\_supernode > max$ $\_height$) THEN  // Find the tallest supertree and merge all to it
 $max$ $\_height = heights$ $\_supernode$
 $supersupernode = supernode$
 ELSE IF ($heights$ $\_supernode \equiv max$ $\_height$) THEN  // There is a subtree of the same height
 $equal$ $\_heights = \mathcal{F}$
 END IF

 node = cycles$\_supernode$  // Find the next node in the “half” cycle
cycles$\_supernode = previous$ $\_supernode$;  // Reverse the parenthood relation
previous$\_supernode = supernode$  // Save for later
\_FindRoot ($supernodes$, node, supernode, climber)  // Find the cluster supernode
\_CompressPath ($supernodes$, node, supernode, climber, climber$\_parent$)  // Shorten supertrees

 IF ($supernode \equiv previous$ $\_supernode$) THEN  // We reached the end of a path or cycle
 last$\_supernode = supernode$
 EXIT $Uncover$.@&ID
 END IF

 path$\_length = path$ $\_length + 1$  // The length of this piece of a cycle
 END DO $Uncover$.@&ID

3 Graph Contraction Via Matching

The following routine is useful for ordering the nodes or arcs in a graph based on their connectivity proximity, as used in the module $Support\_Trees$ in the routine $ST\_Matching$NodesMapping. The routine $Contract\_Matched$Arcs is a rather general routine though, and it performs matching of the arcs contained in the linked list of arcs $arcs$.$sequence$. An arc in the sequence will be matched only if its two endpoints are not matched already. It is meant to be called inside a loop from another routine that updates the list of arcs to try to match, so all of the arrays needed are passed to it as arguments. Also, it uses some clever schemes to mask which nodes to match, and this is done based on the parity of the number of graph contractions performed so far.

I made these matching routines myself and it would take a while to fully document them. They are not really complete and optimized yet either, so I leave this for some future time. One thing to notice is that I use arc number 0 explicitly as a sentinel end-of-list value, so it can not be included in the matching even if you want to (for some silly reason).

\langle Contract\_Matched\_Arcs 31 \rangle \equiv

**Subroutine** $Contract\_Matched\_Arcs$ (node$\_offset$, arc$\_offset$, heads$\_tails$, arcs$\_sequence$, first$\_arc$, supernodes, heights, nodes$\_ordering$, end$\_nodes$, nodes$\_mask$, reuse$\_matching$, update$\_heads$,$\_tails$, odd$\_cycle$)
IMPLICIT NONE

INTEGER (KIND = i_wp), INTENT (IN) :: node_offset, arc_offset
INTEGER (KIND = i_wp), INTENT (INOUT) :: first_arc        // First arc in the list
INTEGER (KIND = i_wp), DIMENSION (:, -arc_offset :), INTENT (INOUT) :: heads, tails
INTEGER (KIND = i_wp), DIMENSION (:, -arc_offset :), INTENT (INOUT) :: arcs_sequence
INTEGER (KIND = i_wp), DIMENSION (:, -node_offset :), INTENT (INOUT) :: supernodes, heights
INTEGER (KIND = i_wp), DIMENSION (:, -node_offset :), INTENT (INOUT), OPTIONAL :: nodes_ordering, end_nodes
INTEGER (KIND = i_byte), DIMENSION (:, -node_offset :), INTENT (INOUT) :: nodes_mask
LOGICAL, INTENT (IN), OPTIONAL :: reuse_matching, update_heads, tails
LOGICAL, INTENT (IN), OPTIONAL :: odd_cycle
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_arcs, n_special_arcs
INTEGER (KIND = i_wp) :: node, arc, head, tail, head_root, tail_root, root, diff_heights, child, parent,
index, climber, climber_parent, end_node, prevs_arc
INTEGER :: alloc_status
INTEGER (KIND = i_byte) :: matched_node
LOGICAL :: reuse_supernodes, contract_heads, tails, delete_arc, sequence_nodes

n_special_nodes = -1 BOUND (supernodes, i_wp, DIM = 1)
n_nodes = BOUND (supernodes, i_wp, DIM = 1)        // The number of nodes, n
n_special_arcs = -1 BOUND (heads, i_wp, DIM = 2)
n_arcs = BOUND (heads, i_wp, DIM = 2)        // m

matched_node = 1 i_byte
IF (PRESENT (odd_cycle)) THEN
  IF (odd_cycle) matched_node = -matched_node        // On even cycles
END IF
sequence_nodes = (PRESENT (end_nodes) AND PRESENT (nodes_ordering))
contract_heads = (PRESENT (update_heads) AND contract_heads) = update_heads

reuse_supernodes = FALSE
IF (PRESENT (reuse_matching))
  reuse_supernodes = reuse_matching
IF (NOT reuse_supernodes) THEN        // Initialize the arrays
  DO node = -n_special_nodes, n_nodes
    supernodes_node = node        // Start with forest of all nodes
  END DO
  DO node = -n_special_nodes, n_nodes
    nodes_ordering_node = node        // Every node by itself
    end_nodes_node = node
  END DO
END IF
heights = 0
END IF

/* Now we start matching the supernodes using the arcs in the arcs sequence (linked list): */
previous_arc = 0
arc = first_arc
ContractArcs: DO        // Not masked
  IF (arc = 0) EXIT ContractArcs        // We exhausted all arcs

  head = heads, tails, arc
tail = heads\_tails\_2, arc
delete\_arc = \mathcal{F}

\_FindRoot(supernodes, head, head\_root, climber)
\_FindRoot(supernodes, tail, tail\_root, climber)
\_CompressPath(supernodes, head, head\_root, climber, climber\_parent)
\_CompressPath(supernodes, tail, tail\_root, climber, climber\_parent)

TreeUnion: \textbf{IF} \ ((head\_root \equiv tail\_root) \mid (nodes\_mask\_head\_root \equiv dead\_node) \mid (nodes\_mask\_tail\_root \equiv dead\_node)) \textbf{THEN} \\
// The arc is redundant
\textbf{ELSE}  \ // Perform union operation
\textbf{IF} \ (nodes\_mask\_head\_root \equiv \neg \text{matched\_node}) \textbf{THEN}
nodes\_mask\_head\_root = \text{matching\_node}
\textbf{ELSE IF} \ (nodes\_mask\_head\_root \geq \text{matching\_node}) \textbf{THEN}
nodes\_mask\_head\_root = -(\text{MIN}(nodes\_mask\_head\_root, \text{HUGE}(1_i\text{byte}) - 1_i\text{byte}) + 1_i\text{byte})
// Mark as unmatched in this round
\textbf{END IF}
\textbf{IF} \ (nodes\_mask\_tail\_root \equiv \neg \text{matched\_node}) \textbf{THEN}
nodes\_mask\_tail\_root = \text{matching\_node}
\textbf{ELSE IF} \ (nodes\_mask\_tail\_root \geq \text{matching\_node}) \textbf{THEN}
nodes\_mask\_tail\_root = -(\text{MIN}(nodes\_mask\_tail\_root, \text{HUGE}(1_i\text{byte}) - 1_i\text{byte}) + 1_i\text{byte})
// Mark as unmatched in this round
\textbf{END IF}

ContractArc: \textbf{IF} \ ((nodes\_mask\_head\_root \neq \text{matched\_node}) \land nodes\_mask\_tail\_root \neq \text{matched\_node}) \textbf{THEN} \\
// Both endpoints are unmatched and available
\textbf{DELETE\_ARC} = \mathcal{T}  \ // Skip this arc in the list
\text{diff\_heights} = (\text{heights\_head\_root} - \text{heights\_tail\_root})
\textbf{IF} \ (\text{diff\_heights} > 0) \textbf{THEN}  \ // Choose which tree merges with which
\text{parent} = \text{head\_root}
\text{child} = \text{tail\_root}
\textbf{ELSE}
\text{parent} = \text{tail\_root}
\text{child} = \text{head\_root}
\textbf{END IF}
nodes\_mask\_parent = \text{matched\_node}  \ // Mark the node as a matched supernode
nodes\_mask\_child = \text{dead\_node}  \ // We are done with this node now
supernodes\_chad = \text{parent}  \ // Hook the two trees-union
\textbf{IF} \ (\text{diff\_heights} \equiv 0)
\text{heights\_parent} = \text{heights\_parent} + 1
\textbf{IF} \ (sequence\_nodes) \textbf{THEN}  \ // Keep a time record of the matching
\text{end\_node} = \text{end\_nodes\_parent}  \ // The end of one of the ordering lists
nodes\_ordering\_end\_node = \text{child}  \ // Merge the two lists for the matching
\text{end\_nodes\_parent} = \text{end\_nodes\_chad}
\textbf{END IF}
\textbf{END IF} \ ContractArc
\textbf{END IF} \ TreeUnion
IF (deleteArc) THEN // Arc is to be skipped
  IF (arc ≡ firstArc) THEN // We must move the pointer to the first arc
    firstArc = arcs_sequenceArc // The next arc becomes first
  ELSE // Shortcut this arc in the list
    arcs_sequencePreviousArc = arcs_sequenceArc // Skip this arc in the list
  END IF
ELSE
  previousArc = arc
END IF
arc = arcs_sequenceArc // Move to the next arc in the list
END DO ContractArcs

arc = firstArc

UpdateNodes: DO
  IF (arc ≡ 0) EXIT UpdateNodes // We exhausted all arcs
  head = heads_tails1, arc
tail = heads_tails2, arc
  findRoot(supernodes, head, head_root, climber)
  findRoot(supernodes, tail, tail_root, climber)
  /* The following two may be redundant work (??): */
  compressPath(supernodes, head, head_root, climber, climber_parent)
  compressPath(supernodes, tail, tail_root, climber, climber_parent)
  IF (contract_heads_tails) THEN
    heads_tails1, arc = head_root // Update HT
tails_tails2, arc = tail_root
  END IF
  IF (nodes_mask_head_root < -matching_node)
    nodes_mask_head_root = ABS(nodes_mask_head_root)
  IF (nodes_mask_tail_root < -matching_node)
    nodes_mask_tail_root = ABS(nodes_mask_tail_root)
  arc = arcs_sequenceArc // Move to next arc
END DO UpdateNodes
END SUBROUTINE ContractMatchedArcs

This code is used in section 1.1.

4 Building Spanning Trees

This section deals with routines for building spanning trees of the kind needed in network flow algorithms. Then I will give algorithms for maintaining these spanning trees under changes of the weights of the arcs.

4.1 Building Parenthood Relations
This routine will build a spanning tree by examining the arcs in \texttt{arc.list} one by one and adding all arcs that do not cause a cycle. It also builds parenthesis relations for the tree. It also works for forests as well, as do all spanning-tree related routines in this module. The logical flag \texttt{use.parenthood} (defaulting to \texttt{T}) should be used to tell whether the routine should use the already provided parenthesis relations in \texttt{parents} and \texttt{orientations}, or initialize these to a forest of all single nodes and start building from there (the default).

As in all routines I give, the nodes are numbered starting from \texttt{-n\_special\_nodes}, and the arcs are numbered from \texttt{-n\_special\_arcs}, which are called \texttt{node.offset} and \texttt{arc.offset} in the argument lists from now on. These do not default to 1, but have to always be given (Fortran restrictions).

\texttt{(BuildSpanningTree 4.1.1) \equiv}

\texttt{SUBROUTINE BuildSpanningTree(arc.offset, node.offset, arc.list, heads.tails, orientations, parents, use.parenthood)}

\texttt{IMPLICIT NONE}

\texttt{INTEGER (KIND = i_wp), INTENT (IN) :: arc.offset, node.offset}
\texttt{// Lower bound for arc and nodal arrays}

\texttt{INTEGER (KIND = i_wp), DIMENSION (:), INTENT (IN) :: arc.list}
\texttt{// Order in which to examine arcs}

\texttt{INTEGER (KIND = i_sp), DIMENSION (: - node.offset :), INTENT (IN) :: heads.tails}
\texttt{// Heads-tails array for \textit{G}}

\texttt{INTEGER (KIND = i_wp), DIMENSION (: - node.offset :), INTENT (INOUT) :: parents}
\texttt{// With displaced counting}

\texttt{INTEGER (KIND = i_byte), DIMENSION (: - node.offset :), INTENT (INOUT) :: orientations}
\texttt{// Orientations for basic arcs in the tree, one of no_parent, tail_is_parent or head_is_parent}

\texttt{LOGICAL, INTENT (IN), OPTIONAL :: use.parenthood}

\texttt{INTEGER (KIND = i_wp), DIMENSION (:), ALLOCATABLE :: heights, forest, cardinalities} \\
\texttt{// The}
\texttt{heights of the union-find trees and the union sets in the spanning forest, and approximate}
\texttt{cardinalities of the tree nodes}

\texttt{INTEGER (KIND = i_wp) :: n\_nodes, n\_special\_nodes, n\_special\_arcs, n\_arcs} \\
\texttt{// Counters}

\texttt{INTEGER (KIND = i_wp) :: index, node, arc, head, tail, head\_root, tail\_root, root, diff\_heights, child,}
\texttt{parent, parent\_node, climber, climber\_parent, diff\_cardinalities} \\
\texttt{// Temporaries}

\texttt{INTEGER :: alloc\_status}

\texttt{INTEGER (KIND = i_byte) :: orientation, climber\_orientation} \\
\texttt{// Temporaries}

\texttt{LOGICAL :: initialize\_forest}

\texttt{n\_special\_nodes = -1 \_BOUND(parents, i_wp, DIM = 1)}
\texttt{n\_nodes = -1 \_BOUND(parents, i_wp, DIM = 1)} \\
\texttt{// The number of nodes, n}
\texttt{n\_special\_arcs = -1 \_BOUND(heads.tails, i_wp, DIM = 2)}
\texttt{n\_arcs = -1 \_BOUND(heads.tails, i_wp, DIM = 2)} \\
\texttt{// m}

\texttt{initialize\_forest = \texttt{T}} \\
\texttt{// Start from a single-node forest}

\texttt{IF (PRESENT(use.parenthood)) initialize\_forest = \texttt{-use.parenthood}}

\texttt{ALLOCATE (heights(-n\_special\_nodes : n\_nodes), stat = alloc\_status)}
\texttt{CALL RecordAllocation(n\_elements = n\_nodes + n\_special\_nodes + 1, mold = 1\_wp,}
\texttt{caller = "BuildSpanningTree", alloc\_status = alloc\_status)}
\texttt{ALLOCATE (forest(-n\_special\_nodes : n\_nodes), stat = alloc\_status)}
\texttt{CALL RecordAllocation(n\_elements = n\_nodes + n\_special\_nodes + 1, mold = 1\_wp,}
\texttt{caller = "BuildSpanningTree", alloc\_status = alloc\_status)}
\texttt{ALLOCATE (cardinalities(-n\_special\_nodes : n\_nodes), stat = alloc\_status)}

13
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = l1_wp,
caller = "BuildSpanningTree", alloc_status = alloc_status)

IF (initialize_forest) THEN
  DO node = -n_special_nodes, n_nodes
    forest_node = node   // Start with forest of all nodes separately
  END DO
  orientations = no_parent   // No parenthood relations at start
  parents = 0    // Dummy arc is parent of all at start
  cardinalities = 1   // Each node is by itself
ELSE    // Find the root of each node to initialize the forest
  InitializeTrees: DO node = -n_special_nodes, n_nodes    // Initialize the forest
    orientation = orientations_node
    IF (orientation ≠ no_parent) THEN
      forest_node = heads_tails(orientation, parents_node)    // Find the parent
    ELSE
      forest_node = node   // Node is by itself in the forest
    END IF
  END DO InitializeTrees
  cardinalities = 0
  // This may be an expensive operation if we start with an almost complete spanning tree:
  ShortenTrees: DO node = -n_special_nodes, n_nodes    // Compress all paths
    FindRoot(forest, node, root, climber)
    CompressPath(forest, node, root, climber, climber_parent)
    cardinalities_root = cardinalities_root + 1
  END DO ShortenTrees
END IF

heights = 0    // All trees are of height 0 (or 1, as you please...)

ForAllArcs: DO index = 1, _SIZE(arcs_list, i_up)
  // Try to add each arc in arcs_list to the tree
  arc = arcs_list_index
  head = heads_tails1, arc
  tail = heads_tails2, arc
  /* To decide whether to add this arc to the tree I use the standard union-find algorithm, as in
   * the connected-components labeling algorithm. This requires extra memory, but speeds the
   * decision process considerably: */
  FindRoot(forest, head, head_root, climber)
  FindRoot(forest, tail, tail_root, climber)
  CompressPath(forest, head, head_root, climber, climber_parent)
  CompressPath(forest, tail, tail_root, climber, climber_parent)

AddArc: IF (head_root ≠ tail_root) THEN    // Add this arc to the tree
  diff_heights = (heights_head_root - heights_tail_root)    // Which set tree is higher
  IF (diff_heights > 0) THEN    // Choose which tree merges with which
    parent = head_root
    child = tail_root
  ELSE
    parent = tail_root
    child = head_root
  END IF
  forest_child = parent    // Hook the two trees-union

14
IF (diff_heights \equiv 0)
    heights_{parent} = heights_{parent} + 1  // Otherwise heights are not-changed by union operation
    cardinalities_{parent} = cardinalities_{parent} + cardinalities_{child}  // Update cardinalities

/* Now the difficult and expensive part is maintaining the parent/hood relations after this arc was added to the tree—this requires reversing parenthood orientations on the path from either the head or tail to its true parent root. Here I assume that a smaller tree (smaller cardinality) implies a shorter path to the true parent root, which is usually true for trees with no special structure: */

diff_cardinalities = \left(\text{cardinalities}_{\text{head-root}} - \text{cardinalities}_{\text{tail-root}}\right)

IF (diff_cardinalities > 0) THEN  // Choose which tree needs parents updating
    parent = arc
    orientation = tail_is_parent
    climber = tail
ELSE
    parent = arc
    orientation = head_is_parent
    climber = head
END IF

ReverseParenthood: DO  // Reverse the parenthood relations on one of the root paths
    climber_orientation = orientations_{climber}  // Save this
    climber_parent = parents_{climber}  // Save

    parents_{climber} = parent  // Reverse the parenthood relations
    IF (orientation \equiv \text{head_is_parent}) THEN  // Reverse orientation of parent-relation
        orientations_{climber} = tail_is_parent
    ELSE IF (orientation \equiv \text{tail_is_parent}) THEN
        orientations_{climber} = head_is_parent
    END IF

    IF (climber_orientation \equiv \text{no_parent}) EXIT ReverseParenthood  // Reached the root
    climber = parents_{tails_{climber_orientation, climber_parent}}  // Climb up the tree
    parent = climber_parent  // Go one step up to the parent root
    orientation = climber_orientation
END DO ReverseParenthood

END IF AddArc

END DO ForAllArcs

CALL RecordAllocation (n_elements = -\text{size}(\text{forest, i_wp}), mold = l_{i_wp})
DEALLOCATE (forest)
CALL RecordAllocation (n_elements = -\text{size}(\text{heights, i_wp}), mold = l_{i_wp})
DEALLOCATE (heights)
CALL RecordAllocation (n_elements = -\text{size}(\text{cardinalities, i_wp}), mold = l_{i_wp})
DEALLOCATE (cardinalities)

END SUBROUTINE BuildSpanningTree

This code is used in section 1.1.

4.2 Building Additional Tree Data Structures
Although the parenthesis relations built by BuildSpanningTree uniquely determine the spanning tree, a few other data-structures, such as the cardinalities of the nodes or a thread-like root-to-leaf ordering of the nodes is needed in most tree algorithms. These are built in the routine BuildTreeStructures. If the cardinalities are already known, rebuild_cardinalities should be set to \( F \), and the routine will be speeded up. Please note that the dummy node at position 0 is not treated as a regular node but is left over as the first node in the ordering always.

The way that the node ordering level_ordering is built is a novel simple procedure which starts from the leafs of the trees (identified through their cardinality being 1-first pass through the nodes) and adds successive levels of parent nodes to the ordering until the root is reached (second pass through nodes). To determine when a node is ready to be added, I make sure that all the children of the node have been already added (simply via counting them!), and then add the node. Other schemes are possible too, such as a breath-first search from the root node, but we do not have the data structures for that. This procedure is actually very fast and simple and works well.

\[
\langle \text{BuildTreeStructures } 4.2.1 \rangle \\
\]

\textbf{SUBROUTINE BuildTreeStructures} (arc_offset, node_offset, heads_tails, tree_mask, orientations, parents, cardinalities, level_ordering, rebuild_cardinalities)

\textbf{IMPLICIT} NONE

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{INTENT} (\textbf{IN}) :: arc_offset, node_offset

// Lower bound for arc and nodsal arrays

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{DIMENSION} (;, \textbf{-arc_offset :}), \textbf{INTENT} (\textbf{IN}) :: heads_tails

// Heads-tails array

\textbf{LOGICAL} (\textbf{kind} = \textbf{l}up), \textbf{DIMENSION} (-arc_offset :), \textbf{INTENT} (\textbf{OUT}), \textbf{OPTIONAL} :: tree_mask

// A tree mask

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{DIMENSION} (-node_offset :), \textbf{INTENT} (\textbf{IN}) :: parents

// With displaced counting

\textbf{INTEGER} (\textbf{kind} = \textbf{i}byte), \textbf{DIMENSION} (-node_offset :), \textbf{INTENT} (\textbf{IN}) :: orientations

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{DIMENSION} (-node_offset :), \textbf{INTENT} (\textbf{INOUT}) :: cardinalities

// Cardinalities of tree nodes

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{DIMENSION} (0 :), \textbf{INTENT} (\textbf{OUT}) :: level_ordering // A thread-like ordering of the tree nodes by levels-starting from the root and going to the leaf nodes

\textbf{LOGICAL}, \textbf{INTENT} (\textbf{IN}), \textbf{OPTIONAL} :: rebuild_cardinalities

// Build cardinalities from scratch or use old ones?-default is build from scratch

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up), \textbf{DIMENSION} (:), \textbf{ALLOCATABLE} :: counters // A temporary array

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs

\textbf{INTEGER} (\textbf{kind} = \textbf{i}up) :: index, node, arc, head, tail, parent, parent_arc, parent_node, level_start, level_end, cimbers, cycle_root

\textbf{INTEGER} :: side, alloc_status

\textbf{INTEGER} (\textbf{kind} = \textbf{i}byte) :: orientation

\textbf{LOGICAL} :: new_cardinalities // An indicator

\textbf{n} \_ \textbf{special} \_ \textbf{nodes} = \textbf{-} \textbf{J} \textbf{BOUND} (\textbf{parents}, \textbf{i}up, \textbf{DIM} = 1)

\textbf{n} \_ \textbf{nodes} = \textbf{J} \textbf{BOUND} (\textbf{parents}, \textbf{i}up, \textbf{DIM} = 1) // The number of nodes, \textbf{n}

\textbf{n} \_ \textbf{special} \_ \textbf{arcs} = \textbf{-} \textbf{J} \textbf{BOUND} (\textbf{heads_tails}, \textbf{i}up, \textbf{DIM} = 2)

\textbf{n} \_ \textbf{arcs} = \textbf{J} \textbf{BOUND} (\textbf{heads_tails}, \textbf{i}up, \textbf{DIM} = 2) // \textbf{m}

\textbf{new} \_ \textbf{cardinalities} = \textbf{F}

\textbf{IF} (\textbf{PRESENT} (\textbf{rebuild} \_ \textbf{cardinalities})) \textbf{new} \_ \textbf{cardinalities} = \textbf{rebuild} \_ \textbf{cardinalities}

16
/* We need either the cardinalities or the nodal child degrees in order to initialize the level ordering of the nodes. If there are no cardinalities provided, it is easier to build child degrees (the number of children of each node counting the node itself): */

IF (new_cardinalities) THEN  // Initialize the cardinalities temporarily with nodal child degrees
    cardinalities = 1
END IF

ChildDegrees: DO node = -n_special_nodes, n_nodes  // Count the number of children of each node
    orientation = orientations_node
    IF (orientation ≠ no_parent) THEN  // Add 1 to the number of children of the parent
        parent_arc = parents_node
        parent_node = heads_fails orientation, parent_arc
        cardinalities parent_node = cardinalities parent_node + 1
    END IF
END DO ChildDegrees
END IF

ALLOCATE (counters (n special_nodes : n_nodes), stat = alloc_status)  // Temporary counters
CALL RecordAllocation (n_elements = n_nodes + n_special_nodes + 1, mold = 1_iwp, caller = "BuildSpanningTree", alloc_status = alloc_status)

/* This way of building an ordering of the nodes into levels may not be the best if we are dealing with a forest of many smaller trees, since it puts all the leaves first, then all the one-level-higher nodes, and so on. A better scheme in that case would be to put all the nodes of one tree in the forest next to one another—this will speed cache performance most likely. This can be done easily since we have cardinalities and we can calculate offsets. But in general using the ordering computed here the nodes can be reordered trivially later on, so I leave this task up to an external user routine if the user thinks this will be beneficial. */

level_start = UBOUND (level ordering, iwp) + 1  // Initialize to end of array
level_end = level_start  // Sweeping pointer as nodes are added to the ordering
DO node = -n_special_nodes, n_nodes  // Find the leaf nodes to start the level structure
    IF (cardinalities node = 1) THEN  // This is a leaf non-dummy node
        level_end = level_end - 1
        level ordering level_end = node  // Add to first level
    END IF
END DO

BuildLevels: DO WHILE (level_end ≥ 1)

    counters = 1

    level_start = level_start - 1  // Add the parent of this node to the ordering
    node = level ordering level_start
    orientation = orientations_node
    IF (orientation = no_parent) CYCLE BuildLevels  // Root node

    parent_node = heads_fails (orientation, parents_node)  // Try to add this node to the list
    IF (new_cardinalities) THEN  // counters is child-degree
        counters parent node = counters parent node + 1
    ELSE  // counters is in fact the cardinality
        counters parent node = counters parent node + counters node
    END IF
    IF (counters parent node = cardinalities parent node) THEN  // Add the node to the level list
        level_end = level_end - 1
        level ordering level_end = parent node
    END IF
END DO BuildLevels

CALL RecordAllocation (n_elements = SIZE (counters, iwp), mold = 1_iwp)
DEALLOCATE(counters)

IF(new_cardinalities) THEN // Now calculate the true cardinalities
  cardinalities = 1

BuildCardinalities: DO index = n_nodes + n_special_nodes, 0, -1
  node = level_ordering_index
  orientation = orientations_node
  IF(orientation ≡ no_parent) CYCLE BuildCardinalities
  parent_node = heads_tails(orientation, parents_node)
  cardinalities parent_node = cardinalities parent_node + cardinalities node
END DO BuildCardinalities
END IF

IF(PRESENT(tree_mask)) THEN // Build a mask that selects the tree arcs from the non-tree arcs
  tree_mask = F_lxp
  DO node = n_special_nodes, n_nodes
    orientation = orientations_node
    IF(orientation ≠ no_parent)
      tree_mask(parents node) = T_lxp
    END IF
  END DO
END IF

END SUBROUTINE BuildTreeStructures

This code is used in section 1.1.

5 Linear Algebra with $B^{-1}$ and $B^{-T}$

After a tree is built, it determines a basis $B$ for the network. The routine PropagateNodesPotentials multiplies a tension vector $t$ with $B^{-T}$ to give a potential vector $\lambda = B^{-T} t$. The routine PropagateArcsFlows multiplies a supply-demand vector $b$ with $B^{-1}$ to give a flow vector $x = B^{-1} b$. These are essentially routines that traverse the tree from the root down to leaf nodes or vice versa and propagate either potentials or flows along the way. These are essential steps in algorithms based on trees. Some of the routines support a non-pure basis $F$, which has the same structure as a basis $B$, but the off-diagonal entries are not 1 or $-1$ but some real multiplier (associated here with the node whose parent-arc we are looking at via the array called nodes_multipliers). These are meant to be used with some of the preconditioning/factorization routines in the next section.

Note that the routines in this module are not meant to perform atomistic operations, such as changing the potential of one node, or pivoting one arc, as used in simplex codes. Rather, these are streamlined operations acting on the whole network meant to be used in conjugate-gradient solvers to perform matrix-vector multiplications.

5.1 Propagating Potentials

This routine will essentially multiply the arc vector arcs_voltages with $B^{-T}$ to give the nodal vector nodes_potentials. It traverses the tree from the root to the leafs using level_ordering. The most important
thing to watch for is that this routine expects the potentials of the root nodes (i.e. nodes which do not have real parents) to be pre-set before calling this routine. The usual way of setting these to 0 is not a good design choice in this case, especially since this routine can deal with forests, which have multiple root. Also, it is important to note that although the routine accepts the potential drops on all arcs as input, only the values for the tree arcs will actually be used, the rest will simply be ignored.

An optional argument nodes multipliers can be used when the basis B is not a pure basis, but some tree matrix F with non-unit off-diagonal entries. Here each multiplier is associated to the corresponding tree arc via the node whose parent arc it is (to save on wasted storage for any multipliers of non-tree arcs).

**Important note on efficiency:** Although I made efforts to have the code be efficient, clarity and versatility were my focus. In particular, some of the IF statements inside the loops can be eliminated if we are certain that we are dealing with a spanning tree, and not a spanning forest. Such an assumption is made in the next section, but not here because these routines make sense on a forest as well. Also, statements such as:

\[
\langle \text{SampleCode 5.1.1} \rangle \equiv \\
\text{IF} (\text{orientation} \equiv \text{tail_is_parent}) \text{ THEN } \\
\quad \text{nodes} \_\text{potentials} \_\text{node} = \text{nodes} \_\text{potentials} \_\text{parent} \_\text{node} + \text{arcs} \_\text{voltages} \_\text{parent} \_\text{arc} \\
\text{ELSE} \\
\quad \text{nodes} \_\text{potentials} \_\text{node} = \text{nodes} \_\text{potentials} \_\text{parent} \_\text{node} - \text{arcs} \_\text{voltages} \_\text{parent} \_\text{arc} \\
\text{END IF}
\]

could be replaced with code that avoids the conditional and instead uses extra flops:

\[
\text{nodes} \_\text{potentials} \_\text{node} = \text{nodes} \_\text{potentials} \_\text{parent} \_\text{node} + (3 - 2 \times \text{orientation}) \times \text{arcs} \_\text{voltages} \_\text{parent} \_\text{arc}
\]

which would be more efficient on a lot of machines. However, on the Pentium I benchmarked this code on this was not the case and conditionals seemed to be handled well inside loops, so I chose to go with the elegant version. But this may need to be changed on some other architectures and compilers!

\[
\langle \text{PropagateNodesPotentials 5.1.3} \rangle \equiv \\
\text{SUBROUTINE PropagateNodesPotentials (arc} \_\text{offset, node} \_\text{offset, heads} \_\text{tails, orientations, parents, level} \_\text{ordering, arcs} \_\text{voltages, tree} \_\text{arcs} \_\text{voltages, nodes} \_\text{potentials, nodes} \_\text{multipliers})}
\]

\text{IMPLICIT NONE}

\text{INTEGER (\text{kind} = \text{i} \_\text{up}), INTENT (\text{in}) :: arc} \_\text{offset, node} \_\text{offset}

// Lower bound for arc and nodal arrays

\text{INTEGER (\text{kind} = \text{i} \_\text{up}), DIMENSION (, = arc} \_\text{offset :), INTENT (\text{in}) :: heads} \_\text{tails}

// Heads-tails array for G

\text{INTEGER (\text{kind} = \text{i} \_\text{up}), DIMENSION (= node} \_\text{offset :), INTENT (\text{in}) :: parents}

// With displaced counting

\text{INTEGER (\text{kind} = \text{i} \_\text{byte}), DIMENSION (= node} \_\text{offset :), INTENT (\text{in}) :: orientations}
INTEGER (KIND = i_wp), DIMENSION (0 :), INTENT (IN) :: level_ordering
  // A thread-like ordering of the tree nodes by levels starting from the root
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (INOUT) :: nodes_potentials
  // The potentials of the nodes—the roots should have their potentials pre-set!
REAL (KIND = r_wp), DIMENSION (-arc_offset :), INTENT (IN), OPTIONAL :: arcs_voltages
  // The potential drops across the arcs only tree arcs will be used
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (IN), OPTIONAL :: tree_arcs_voltages
  // Only tree arcs
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (IN), OPTIONAL :: nodes_multipliers
  // Arc multipliers, used when B is not a pure basis but has arc gains

INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs
INTEGER (KIND = i_wp) :: index, node, arc, parent_arc, parent_node
LOGICAL :: non_pure

n_special_nodes = -1.BOUND(parents, i_wp, DIM = 1)
n_nodes = +1.BOUND(parents, i_wp, DIM = 1)  // The number of nodes, n
n_special_arcs = -1.BOUND(heads_tails, i_wp, DIM = 2)
n_arcs = +1.BOUND(heads_tails, i_wp, DIM = 2)  // m
non_pure = PRESENT(nodes_multipliers)

ArcOrNodeBased: IF (PRESENT(tree_arcs_voltages)) THEN  // Faster and better

PropagateTreePotentials: DO i = 0, n_nodes + n_special_nodes
  // Start from root node and propagate the potentials down the tree to the leaf nodes
  /* Here I assume that nodes_potentials_node for root nodes is the potential of the rooting virtual node connected to (the root of) this tree in the forest. Please note that this is handled differently below, where I assume that nodes_potentials_node for root nodes is the potential of the root node itself, since in that case I do not explicitly have virtual rooting nodes (no storage array for that). These are both equivalent if we are dealing with one tree only and not a forest: WRONG: THE POTENTIALS OF THE ROOTS ARE OVERWRITTEN IN THE CASE WITH MULTIPLIERS */

  node = level_ordering(index)
  orientation = orientations_node

  RootNode: IF (orientation == no_parent) THEN
    IF (non_pure) THEN  // Use a virtual parent (the node itself)
      nodes_potentials_node = tree_arcs_voltages_node  // Overwrite the input
    ELSE
      nodes_potentials_node = nodes_potentials_node + tree_arcs_voltages_node
    END IF
  ELSE  // Use the parent potential
    parent_node = heads_tails_orientation, parent(node)  // This is the parent node
    IF (non_pure) THEN
      nodes_potentials_node = nodes_multipliers_node * nodes_potentials_parent_node +
                              tree_arcs_voltages_node
    ELSE
      nodes_potentials_node = nodes_potentials_parent_node + tree_arcs_voltages_node
    END IF
  END IF RootNode
END DO PropagateTreePotentials

20
ELSE IF (PRESENT(arcs_voltages)) THEN          // Slower!

PropagatePotentials: DO  index = 0,  n_nodes + n_special_nodes
    // Start from root node and propagate the potentials down the tree to the leaf nodes
    node = level_ordering[index]
    orientation = orientations[node]
    IF (orientation \equiv no_parent) THEN
        IF (non_pure) nodes_potentials_node = 0, \omega \rho  // We must ground it
            CYCLE PropagatePotentials
    END IF
    // Leave root potentials alone!
    parent_arc = parents_node
    parent_node = heads_tail_orientation, parent_arc  // This is the parent node

Multipliers: IF (non_pure) THEN
    IF (orientation \equiv tail_is_parent) THEN
        nodes_potentials_node = nodes_multipliers_node * nodes_potentials_parent_node +
                         arcs_voltages_parent_arc
    ELSE
        nodes_potentials_node = nodes_multipliers_node * nodes_potentials_parent_node -
                         arcs_voltages_parent_arc
    END IF
    ELSE
    IF (orientation \equiv tail_is_parent) THEN
        nodes_potentials_node = nodes_potentials_parent_node + arcs_voltages_parent_arc
    ELSE
        nodes_potentials_node = nodes_potentials_parent_node - arcs_voltages_parent_arc
    END IF
END IF Multipliers
END DO PropagatePotentials
END IF ArcOrNodeBased
END SUBROUTINE PropagateNodesPotentials

This code is used in section 1.1.

5.2 Propagating Flows

This routine will essentially multiply the nodal vector supplies_demands with $B^{-1}$ to give the arc vector tree_flows. It traverses the tree from the leaves to the root using level_ordering. This routine will leave the rooting arcs, i.e., the fictitious arc parents_node for which orientations_node \equiv no_parent alone. If the supplies-demands vector adds up to a total in/outflow of zero, the flow that should go through the rooting arc will also be zero. To allow for this not to be the case and give the user some useful info, the actual flow that is left over after all the pushing is done is returned in the optional flow_imbalance.

Again, non-pure bases are supported via the optional nodes_multipliers. No array temporary is needed here for storing the excess flows at each node, even though such a temporary would speed the loops somewhat. It is also important to note that I leave the roots and their rooting arcs alone in this collection of libraries. The user will probably have to deal with them separately before or after calling these routines.

21
SUBROUTINE PropagateArcsFlows (arc_offset, node_offset, heads_tails, orientations, parents,
   level_ordering, supplies_demands, arcs_flows, tree_arcs_flows, flow_imbalance, nodes_multipliers)
IMPLICIT NONE
INTEGER (KIND = i_wp), INTENT (IN) :: arc_offset, node_offset
INTEGER (KIND = i_wp), DIMENSION (:, -arc_offset :), INTENT (IN) :: heads_tails
INTEGER (KIND = i_wp), DIMENSION (-node_offset :], INTENT (IN) :: parents
INTEGER (KIND = i_byte), DIMENSION (-node_offset :], INTENT (IN) :: orientations
INTEGER (KIND = i_wp), DIMENSION (0 :], INTENT (IN) :: level_ordering
REAL (KIND = r_wp), DIMENSION (-node_offset :], INTENT (IN) :: supplies_demands
REAL (KIND = r_wp), DIMENSION (-arc_offset :], INTENT (OUT), OPTIONAL :: arcs_flows
   // The feasible flows in the tree arcs. The non-tree arcs are not assigned flows!
REAL (KIND = r_wp), DIMENSION (-node_offset :], INTENT (OUT), OPTIONAL :: tree_arcs_flows
   // Only tree arcs are of interest
REAL (KIND = r_wp), INTENT (OUT), OPTIONAL :: flow_imbalance
   // Total excess flow in the forest
REAL (KIND = r_wp), DIMENSION (-node_offset :], INTENT (IN), OPTIONAL :: nodes_multipliers
   // Arc multipliers, used when B is not a pure basis but has arc gains
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs
INTEGER (KIND = i_wp) :: index, node, arc, parent_arc, parent_node, grandparent_arc
INTEGER :: alloc_status
INTEGER (KIND = i_byte) :: orientation, parent.orientation
LOGICAL :: non_pure
n_special_nodes = -i.BOUND(parents, i_wp, DIM = 1)

n_nodes = i.BOUND(parents, i_wp, DIM = 1)  // The number of nodes, n
n_special_arcs = -i.BOUND(heads_tails, i_wp, DIM = 2)
n_arcs = i.BOUND(heads_tails, i_wp, DIM = 2)  // m
non_pure = PRESENT(nodes_multipliers)
IF (PRESENT(flow_imbalance)) flow_imbalance = 0.0_rwp

NodeOrArcBased: IF (PRESENT(tree_arcs_flows)) THEN  // Faster and simpler—try it first
   // We assume tail.is.parent for all tree arcs!!!
   tree_arcs_flows = supplies_demands  // Initialize
PropagateTreeFlows: DO index = n_nodes + n_special_nodes, 0, -1
   // Start from leaf nodes and propagate the flows up to the root, without intermediate storage
   node = level_ordering(index)
   orientation = orientations(node)
   IF (orientation = no.parent) THEN
      IF (PRESENT(flow_imbalance)) flow_imbalance = flow_imbalance + tree_arcs_flows(node)
   CYCLE PropagateTreeFlows  // This is a tree root
   END IF
   parent.node = heads_tails(orientation, parents(node))  // This is the parent node
   IF (non_pure) THEN
      tree_arcs_flows(parent.node) = tree_arcs_flows(parent.node) + nodes_multipliers(node) * tree_arcs_flows(node)
   ELSE
      tree_arcs_flows(parent.node) = tree_arcs_flows(parent.node) + tree_arcs_flows(node)
   END IF
END DO PropagateTreeFlows
ELSE IF (PRESENT(\texttt{arc\_flows})) THEN // More complicated and slower

\textbf{InitialFlows:} DO node = \texttt{\_n\_special\_nodes, n\_nodes}  
orientation = \texttt{orientations\_node}  
IF (orientation \equiv \texttt{no\_parent}) CYCLE InitialFlows  
parent\_arc = \texttt{parents\_node}  
IF (orientation \equiv \texttt{tail\_is\_parent}) THEN  
arc\_flows\_parent\_arc = \texttt{supplies\_demands\_node}  
ELSE  
arc\_flows\_parent\_arc = \texttt{-supplies\_demands\_node}  
END IF  
END DO InitialFlows

\textbf{PropagateFlows:} DO index = \texttt{n\_nodes + n\_special\_nodes, 0, -1}  
// Start from leaf nodes and propagate the flows up to the root, without intermediate storage  
node = level\_ordering\_index  
orientation = \texttt{orientations\_node}  
IF (orientation \equiv \texttt{no\_parent}) THEN  
IF (PRESENT(\texttt{flow\_imbalance})) THEN  
flow\_imbalance = flow\_imbalance + \texttt{supplies\_demands\_node}  
END IF  
CYCLE PropagateFlows  
END IF  
parent\_arc = \texttt{parents\_node}  
parent\_node = heads\_tail\_orientation, parent\_arc  // This is the parent node  
parent\_orientation = \texttt{orientations\_parent\_node}  
IF (parent\_orientation \equiv \texttt{no\_parent}) THEN  // The parent is a root  
IF (PRESENT(\texttt{flow\_imbalance})) THEN  
flow\_imbalance = flow\_imbalance + \texttt{arc\_flows\_parent\_arc}  
END IF  
CYCLE PropagateFlows  // Leave root potentials alone!  
END IF  
grandparent\_arc = \texttt{parents\_parent\_node}  

\textbf{Multipliers:} IF (\texttt{non\_pure}) THEN  
IF (parent\_orientation \equiv orientation) THEN  
arcs\_flows\_grandparent\_arc = arcs\_flows\_grandparent\_arc + \texttt{nodes\_multipliers\_node} *  
arcs\_flows\_parent\_arc  
ELSE  
arcs\_flows\_grandparent\_arc = arcs\_flows\_grandparent\_arc - \texttt{nodes\_multipliers\_node} *  
arcs\_flows\_parent\_arc  
END IF  
ELSE  
IF (parent\_orientation \equiv orientation) THEN  
arcs\_flows\_grandparent\_arc = arcs\_flows\_grandparent\_arc + arcs\_flows\_parent\_arc  
ELSE  
arcs\_flows\_grandparent\_arc = arcs\_flows\_grandparent\_arc - arcs\_flows\_parent\_arc  
END IF  
END IF Multipliers

END DO PropagateFlows

END IF NodeOrArcBased

IF (PRESENT(\texttt{tree\_arc\_flows}) \land \texttt{PRESENT(\texttt{arc\_flows})) THEN // Copy the result
CopyFlows: DO node = –nodes, nodes  
orientation = orientations
IF (orientation = no parent) CYCLE CopyFlows  
parent_arc = parent
IF (orientation = tail is parent) THEN  
flows parent = tree flows
ELSE  
flows parent = –tree flows
END IF
END DO CopyFlows
END IF
IF (PRESENT(flow imbalance)) flow imbalance = ABS(flow imbalance)  // Total flow excess
END SUBROUTINE PropagateArcsFlows

This code is used in section 1.1.

6 Maintaining Minimal/Maximal Spanning Trees

The following set of routines provides algorithms for efficient maintenance of minimal spanning trees under changes in the edge weights.

6.1 Initializing Node Path Labels

The routine CalculatePathLabels calculates path labels for the nodes in a spanning forest. It does this with an efficient single-pass loop through the nodes from the root to the leaves (using level ordering). Recall that for a minimal-weight spanning trees path labels are defined as the maximal weight (or an upper bound on it, to be precise) on the path from the node to the root, while for maximal-weight STs it is a lower bound on the minimal weight on the path to the root. These labels are not really used in this module, but they can be efficiently used to reject a lot of candidate arcs for entry into a spanning tree just by looking at the path labels, as is done in the module NetworkSpanningTrees:

(CalculatePathLabels 6.1.1) ≡

SUBROUTINE CalculatePathLabels (arc_offset, node_offset, heads_tails, level_ordering, orientations,  
parents, path_labels, arcs_weights, tree_type)
IMPLICIT NONE
INTEGER (KIND = i_up), INTENT (IN) :: arc_offset, node_offset
INTEGER (KIND = i_up), DIMENSION (:, arc_offset :), INTENT (IN) :: heads_tails  // Heads-tails
INTEGER (KIND = i_up), DIMENSION (0 :), INTENT (IN) :: levelOrdering  
// Ordering from root to nodes
INTEGER (KIND = i_up), DIMENSION (node_offset :), INTENT (IN) :: parents
INTEGER (KIND = i_byte), DIMENSION (-node_offset :), INTENT (IN) :: orientations
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (OUT) :: path_labels  // Max or Min
REAL (KIND = r_wp), DIMENSION (-arc_offset :), INTENT (IN) :: arcs_weights   // Weights on arcs
INTEGER (KIND = i_byte), INTENT (IN) :: tree_type      // min_cost_tree or max_cost_tree
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes
INTEGER (KIND = i_wp) :: index, node, arc, head, tail, parent_arc, parent_node
INTEGER (KIND = i_byte) :: orientation
INTEGER :: alloc_status
REAL (KIND = r_wp) :: initial_value

n_special_nodes = INTEGER(BOUND(parents, i_wp, DIM = 1))
n_nodes = INTEGER(BOUND(parents, i_wp, DIM = 1))  // The number of nodes, n
IF (tree_type == min_cost_tree) THEN
  initial_value = -HUGE(1.0_r_wp)
ELSE
  initial_value = HUGE(1.0_r_wp)
END IF

PathLabels:  DO index = 0, n_nodes + n_special_nodes
  // Start from root node and propagate the potentials down the tree to the leaf nodes
  node = level_ordering(index)
  orientation = orientations(node)
  IF (orientation == no_parent) THEN
    path_labels(node) = initial_value  // Initialize the root path labels
  CYCLE PathLabels  // Leave root potentials alone!
  END IF
  parent_arc = parents(node)
  parent_node = heads(parent_arc, orientation, parent_node)  // This is the parent node
  IF (tree_type == min_cost_tree) THEN
    path_labels(node) = MAX(path_labels(parent_node), arcs_weights(parent_arc))
  ELSE
    path_labels(node) = MIN(path_labels(parent_node), arcs_weights(parent_arc))
  END IF
END DO PathLabels

END SUBROUTINE CalculatePathLabels

This code is used in section 1.1.

6.2 Rebuilding a Spanning Tree

The following routine will update a spanning tree by checking whether the arcs in arcs_list offer advantageous exchanges with some arc in their basic equivalent path (BEP) cycle. If there is such an advantageous change, the examined arc enters the spanning tree, while the arc with the maximal or minimal weight in the BEP leaves the tree. This is a pivoting operation, and requires updating the parenthood and cardinality relations. Cardinalities are used to trace the BEP without having to go all the way to the root, as described in other people’s work on simplex-like codes and similar non-greedy MST algorithms.

The user can choose whether he wants a minimal or maximal weight ST via tree_type, and a helpful statistics on the number of actual pivots performed, divided by the number of arcs (cycles) examined, is
returned in the optional \texttt{percent\_pivoted}. If this is small the starting tree can be judged near-optimal with good confidence.

\begin{verbatim}
<ReBuildSpanningTree 6.2.1> ≡

\textbf{SUBROUTINE} \texttt{ReBuildSpanningTree}(arc\_offset, node\_offset, heads\_tails, arcs\_list, arcs\_weights, orientations, parents, cardinalities, percent\_pivoted, tree\_type)
\textbf{IMPLICIT} NONE
INTEGER (KIND = i\_wp), INTENT (IN) :: arc\_offset, node\_offset
INTEGER (KIND = i\_wp), DIMENSION (:), INTENT (IN) :: heads\_tails
INTEGER (KIND = i\_wp), DIMENSION (:), INTENT (IN) :: arc\_offset
// Order in which to examine non-tree arcs for entry into the tree. These do not have to be non-tree arcs though.
REAL (KIND = r\_wp), DIMENSION (arc\_offset), INTENT (IN) :: arcs\_weights
// Weights on arcs
INTEGER (KIND = i\_wp), DIMENSION (node\_offset), INTENT (INOUT) :: parents, cardinalities
// For tree nodes
INTEGER (KIND = i\_byte), DIMENSION (node\_offset), INTENT (INOUT) :: orientations
// For tree arcs
REAL, INTENT (OUT), OPTIONAL :: percent\_pivoted
// How many traces resulted in pivots
INTEGER (KIND = i\_byte), INTENT (IN) :: tree\_type
// min\_cost\_tree or max\_cost\_tree

/* Local variables: */
INTEGER (KIND = i\_wp) :: n\_traces, n\_pivots
// Operation counters
INTEGER (KIND = i\_wp) :: n\_nodes, n\_special\_nodes, n\_special\_arcs, n\_arcs
// Counters
INTEGER (KIND = i\_wp) :: node, arc, arc\_index, entering\_arc, leaving\_arc, climbers\_2, head, tail,
climb, parent, climber\_parent, climber\_cardinality, cardinality, cycle\_root, subtree\_root,
hooking\_node, unhooking\_node, subtree\_size
INTEGER (KIND = i\_byte) :: orientation, climber\_orientation, leaving\_orientation
INTEGER :: alloc\_status, side, leaving\_side
LOGICAL :: weights\_test, tree\_arc, climbers\_at\_root\_2

REAL (KIND = r\_wp) :: max\_tree\_weight
n\_special\_nodes = -\_j\_bound(parents, i\_wp, DIM = 1)
n\_nodes = \_j\_bound(parents, i\_wp, DIM = 1) // The number of nodes, n
n\_special\_arcs = -\_j\_bound(heads\_tails, i\_wp, DIM = 2)
n\_arcs = \_j\_bound(heads\_tails, i\_wp, DIM = 2) // m
n\_traces = 0 // Counter for the number of BEP cycle traces
n\_pivots = 0 // Counter for the number of pivots

\textbf{TestPivot}: \textbf{DO} arc\_index = \_j\_bound(arcs\_list, i\_wp), \_j\_bound(arcs\_list, i\_wp)

\begin{verbatim}
entering\_arc = arcs\_list\_arc\_index
head = heads\_tails\_1, entering\_arc
tail = heads\_tails\_2, entering\_arc

// We test to make sure this is not a tree arc in order to make the code more robust:
tree\_arc = (orientations\_head \equiv tail\_is\_parent) \land (parents\_head \equiv entering\_arc) \lor ((orientations\_tail \equiv head\_is\_parent) \land (parents\_head \equiv entering\_arc))
\textbf{IF} (tree\_arc \&\& entering\_arc \equiv 0) \textbf{CYCLE} \textbf{TestPivot} // This is a tree arc!
/* Test BEP cycle for an exchange of arcs: */
\textbf{DO} arc\_index = arcs\_list\_arc\_index

\begin{verbatim}
leaving\_arc = entering\_arc
climbers = (head, tail) // Start tracing the cycle formed by entering\_arc
climbers\_at\_root = \text{true} // In case we are dealing with a forest!
\end{verbatim}
\end{verbatim}

\textbf{TraceCycle}: \textbf{DO}

\end{verbatim}
\end{verbatim}

26
IF (climbers₁ ≡ climbers₂) THEN  // Found a joint in the tree
cycle_root = climbers₁
EXIT TraceCycle
END IF

ChooseSide: IF (climbers_at_root₁ ∧ climbers_at_root₂) THEN
  // This arc crossed between trees in the forest—we have different tree roots
  CYCLE TestPivot
ELSE IF (climbers_at_root₁) THEN  // One climber finished at a root
  side = 2
ELSE IF (climbers_at_root₂) THEN  // One climber finished at a root
  side = 1
ELSE  // Use cardinalities to make a smart decision
  IF (cardinalities_climbers₁ ≤ cardinalities_climbers₂) THEN
    side = 1  // Climb on side 1
  ELSE
    side = 2  // Climb on side 2
  END IF
END IF

END IF ChooseSide

parent = parents(climbers_side)  // Candidate to leave the tree—an arc in the cycle
orientation = orientations(climbers_side)

ClimbUp: IF (orientation ≡ no_parent) THEN  // Reached a root
  climbers_at_root_side = T
ELSE
  IF (tree_type ≡ min_cost_tree) THEN
    weights_test = (arcs_weights_leaving_arc < arcs_weights_parent)
  ELSE
    weights_test = (arcs_weights_leaving_arc > arcs_weights_parent)
  END IF
  IF (weights_test) THEN
    leaving_arc = parent
    leaving_orientation = orientation
    leaving_side = side
  END IF
  climbers_side = heads_tails_orientation, parent  // Climb up one level on this side
END IF ClimbUp

END DO TraceCycle

PivotArc: IF (leaving_arc ≡ entering_arc) THEN  // Arc did not qualify to enter the MST
  CYCLE TestPivot
ELSE  // We need to perform a pivot operation
  n_pivots = n_pivots + 1
  IF (leaving_orientation ≡ head_is_parent) THEN
    unhooking_node = heads_tails₁, leaving_arc  // This is where subtree unhooks from
    subtree_root = heads_tails₂, leaving_arc
  ELSE
    unhooking_node = heads_tails₂, leaving_arc  // This is where subtree unhooks from
subtree_root = heads_tails1, leaving_arc
END IF
subtree_size = cardinalities subtree_root

IF (leaving_side ≡ 1) THEN
hooking_node = heads_tails2, entering_arc
climber = heads_tails1, entering_arc
orientation = head_is_parent
ELSE
hooking_node = heads_tails1, entering_arc
climber = heads_tails2, entering_arc
orientation = tail_is_parent
END IF
parent = entering_arc  // Initialize for ReverseParenthood
cardinality = 0  // Initialize

ReverseParenthood: DO  // Reverse the parenthood relations on part of the cycle
  climber_parent = parents climber  // Save
  climber_orientation = orientations climber  // Save this too
  climber_cardinality = cardinalities climber  // Save
  parents climber = parent  // Reverse the parenthood relations
  IF (orientation ≡ head_is_parent) THEN  // Reverse orientation of parent-relation
    orientations climber = tail_is_parent
  ELSE IF (orientation ≡ tail_is_parent) THEN
    orientations climber = head_is_parent
  END IF
  cardinalities climber = subtree_size - cardinality  // Update the cardinality
  IF (climber ≡ subtree_root) EXIT ReverseParenthood  // Reached the root
  climber = heads_tails climber_orientation, climber_parent  // Climb up the tree
  parent = climber_parent  // Go one step up to the parent root
  orientation = climber_orientation
  cardinality = climber_cardinality
END DO  ReverseParenthood

climber = unhooking_node
DO WHILE (climber ≠ cycle_root)  // Update cardinalities
  cardinalities climber = cardinalities climber - subtree_size
  climber = heads_tails (orientations climber, parents climber)  // Move up
END DO

climber = hooking_node
DO WHILE (climber ≠ cycle_root)  // Update cardinalities
  cardinalities climber = cardinalities climber + subtree_size
  climber = heads_tails (orientations climber, parents climber)  // Move up
END DO

END IF PivotArc
END DO TestPivot

IF (debug_graph alg) THEN
  WRITE(*, *, 'In ReBuildSpanningTree, n_traces="', n_traces, ", n_traces", ", n_pivots="', n_pivots, ", n_pivots", ", percent="', REAL(n_pivots) / REAL(n_traces)
END IF

IF (PRESENT(percent pivoted)) percent pivoted = REAL(n pivots) / REAL(n traces)

28
7 Preconditioning Laplacian Matrices via Incomplete MST Factorizations

For network optimization problems whose solution has a tree structure, such as linear min-cost problems, several incomplete preconditioners have been developed in the literature and are implemented here. These basically form a preconditioner of the form \( M = F D F^T \) where \( F \) is either a pure basis (called \( B \) above), in which case we have the MST preconditioner of Vaidya/Vega, or a tree matrix with non-unit off-diagonal entries, such as one coming from the incomplete zero-fill QR factorization of \( A D_A A^T \) of Resende and Pardalos, or the complete root-free Cholesky factorization of \( B D_B B^T + \Lambda \) of Mehrotra and Wang, where \( \Lambda \) is diagonal, usually the non-tree contribution to the diagonal of \( A D_A A^T \). Here all \( D \) matrices are diagonal conductivity matrices, although they could also be supplied as resistivities in some routines to avoid more expensive divisions.

7.1 Building the Preconditioner Spanning Tree

This procedure can be used to initialize or rebuild the spanning tree used by the preconditioners. It does not contain much new material, but is mostly a wrapper for the other routines in this module. If the tree is built from scratch, then Quick Ranking is used to sort the weights of the arcs and build the MST, however, if this is not the case, path labels are created and used to select the list of arcs candidates for entry into the MST. By setting \texttt{tree\_type = any\_tree} an initial dummy MST can be created very quickly.

\begin{verbatim}
(BuildNetworkMST 7.1.1) \equiv

SUBROUTINE BuildNetworkMST (arc\_offset, node\_offset, heads\_tails, arcs\_mask, arcs\_weights,
   level\_ordering, orientations, parents, cardinalities, path\_labels, tree\_mask, from\_scratch,
   defragment\_ordering, tree\_type)
IMPLICIT NONE
INTEGER (KIND = i\_up), INTENT (IN) :: arc\_offset, node\_offset
INTEGER (KIND = i\_up), DIMENSION (\_arc\_offset,:), INTENT (IN) :: heads\_tails   // Heads-tails
LOGICAL (KIND = L\_up), DIMENSION (\_arc\_offset,:), INTENT (IN), OPTIONAL :: arcs\_mask
   // Arc mask
REAL (KIND = r\_up), DIMENSION (\_arc\_offset,:), INTENT (IN), OPTIONAL :: arcs\_weights
   // Arc weights
INTEGER (KIND = i\_up), DIMENSION (0,:), INTENT (OUT) :: level\_ordering   // From root to nodes
INTEGER (KIND = i\_int), DIMENSION (\_node\_offset,:), INTENT (INOUT) :: parents, cardinalities
INTEGER (KIND = i\_int), DIMENSION (\_node\_offset,:), INTENT (INOUT) :: orientations
REAL (KIND = r\_up), DIMENSION (\_node\_offset,:), INTENT (OUT), OPTIONAL :: path\_labels
   // Max or Min
\end{verbatim}
LOGICAL (kind = L_wp, dimension (-arc_offset :), intent (inout) :: tree_mask
LOGICAL, intent (in), optional :: from_scratch // Is this a brand new tree? (F)
LOGICAL, intent (in), optional :: defragment_ordering // Pack each forest in level_ordering or not? (F)
INTEGER (kind = i_byte), intent (in) :: tree_type // min_cost_tree, max_cost_tree or any_tree
INTEGER (kind = i_wp), dimension (:), allocatable :: arcs_ordering, candidate_arcs, nodes_roots,
      nodes_ordering // Temporary storage arrays
INTEGER (kind = i_wp) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs // Counters
INTEGER (kind = i_wp) :: node, arc, arc_index, index, parent_arc, parent_node, head, tail,
      n_candidate_arcs, offset, root_node
INTEGER :: alloc_status
INTEGER (kind = i_byte) :: orientation
LOGICAL :: initialize_tree, mask_arcs, rebuild_cardinalities, reorder_nodes, tree_arc_test, path_labels_test
n_special_nodes = -BOUND(parents, i_wp, DIM = 1) // The number of nodes, n
n_special_arcs = -BOUND(heads_tails, i_wp, DIM = 2)
intialize_tree = F
if (PRESENT(from_scratch)) initialize_tree = from_scratch
mask_arcs = PRESENT(arcs_mask)
reorder_nodes = F
if (PRESENT(defragment_ordering)) reorder_nodes = defragment_ordering
BuildMST: if ((tree_type #= max_cost_tree) or (tree_type #= min_cost_tree)) then
  // Building an MST is a bit complicated because arcs need to be sorted
OrderArcs: if (initialize_tree) then // We need to order the arcs weights in this case
  rebuild_cardinalities = T // Cardinalities are not maintained here
  allocate (arcs_ordering(1 : n_special_arcs + n_arcs + 1), stat = alloc_status)
call RecordAllocation(n_elements = n_special_arcs + n_arcs + 1, mold = 1, iwp,
         caller = "BuildNetworkMST", alloc_status = alloc_status)
call QuickRank(array = arcs_weights, permutation = arcs_ordering, pivot_selection = 'R',
               partially_ranked = F)
arcs_ordering = arcs_ordering - n_special_arcs - 1 // The correctly numbered permutation
if (mask_arcs) then
  // Compress the arc list to contain only masked arcs
  n_candidate_arcs = 0
  do arc_index = 1, n_special_arcs + n_arcs + 1
     arc = arcs_ordering(arc_index)
     if (arc_mask) then
        n_candidate_arcs = n_candidate_arcs + 1
        arcs_ordering(n_candidate_arcs) = arc
     end if
  end do
else
  n_candidate_arcs = n_special_arcs + n_arcs + 1 // Consider all arcs
end if
if (tree_type #= max_cost_tree) then // Consider arcs in reverse order
call BuildSpanningTree(arc_offset = n_special_arcs, node_offset = n_special_nodes,
          arcs_list = arcs_ordering(n_candidate_arcs+1:1), heads_tails = heads_tails,
          orientations = orientations, parents = parents, use_parenthood = F)
end if
30
ELSE // Arcs are in right order—from small to large weights
    CALL BuildSpanningTree(arc_offset = n_specialArcs, node_offset = n_specialNodes,
        arcs_list = arcs_ordering; n_candidateArcs, heads_tails = heads_tails, 
        orientations = orientations, parents = parents, use_parenthesis = F)
END IF

CALL RecordAllocation(n_elements = SIZE(arcs_ordering, i_wp) - mold, mold = 1, i_wp)
DEALLOCATE (arcs_ordering, stat = alloc_status)

ELSE OrderArcs
    /* In this case we want to reoptimize a legal spanning forest. Masking the arc is not supported,
    since arcs between trees can never enter the tree in this scheme (one must first rebuild a forest
    with the new mask and then reoptimize this forest in this case!): */

    rebuild_cardinalities = F // Cardinalities are not maintained here
    ALLOCATE (candidateArcs (1 : n_specialArcs + nArcs + 1), stat = alloc_status)
    CALL RecordAllocation(n_elements = n_specialArcs + nArcs + 1, mold = 1, i_wp,
        caller = "UpdateSpanningTree", alloc_status = alloc_status)
    CALL CalculatePathLabels(arc_offset = n_specialArcs, node_offset = n_specialNodes,
        heads_tails = heads_tails, orientations = orientations, parents = parents,
        levelOrdering = levelOrdering, path_labels = path_labels, arcs_weights = arcs_weights,
        tree_type = tree_type)

    n_candidateArcs = 0 // How many arcs are eligible to enter the MST
    DO arc = -n_specialArcs, nArcs // Find eligible arcs for entry into tree
        head = heads_tails1, arc
        tail = heads_tails2, arc
        /* An eligible arc is a non-tree arc whose weight compares well with the path labels: */
        treeArcTest = ~tree_maskArc
        IF (maskArcs) treeArcTest = treeArcTest AND arcs_maskArc
        IF (treeArcTest) THEN
            IF (tree_type = min_cost_tree) THEN
                path_labelsTest = (arcs_weights弧 <= MAX(path_labelshead, path_labelsTail))
            ELSE
                path_labelsTest = (arcs_weights弧 >= MIN(path_labelshead, path_labelsTail))
            END IF
        ENDIF
        IF (path_labelsTest) THEN // The arc is eligible
            n_candidateArcs = n_candidateArcs + 1
            candidateArcs(n_candidateArcs) = arc
        ENDIF
    END DO
END IF OrderArcs

ELSE BuildMST // We just need a random spanning forest
    /* I do not support random trees here, only any_tree, since sorting is expensive
    rebuild_cardinalities = T // Cardinalities are not maintained here

    REBUILD MST
    CALL UpdateSpanningTree
    CALL CalculatePathLabels
    CALL RecordAllocation(n_elements = SIZE(candidateArcs, i_wp) - mold, mold = 1, i_wp)
    DEALLOCATE (candidateArcs, stat = alloc_status)
END IF
ALLOCATE (candidate_arcs (1 : n_special_arcs + n_arcs + 1), stat = alloc_status)
CALL RecordAllocation (n_elements = n_special_arcs + n_arcs + 1, mold = l_i_wp,
caller = "UpdateSpanningTree", alloc_status = alloc_status)

IF (mask_arcs) THEN
  // Compress the arc list to contain only masked arcs in sequence
  n_candidate_arcs = 0
  DO arc = -n_special_arcs, n_arcs
    IF (arcs_mask_arc) THEN
      n_candidate_arcs = n_candidate_arcs + 1
      candidate_arcs (n_candidate_arcs) = arc
    END IF
  END DO
ELSE
  DO arc = -n_special_arcs, n_arcs
    candidate_arcs (arc + n_special_arcs + 1) = arc
  END DO
  n_candidate_arcs = n_special_arcs + n_arcs + 1  // Consider all arcs
END IF

CALL BuildSpanningTree (arc_offset = n_special_arcs, node_offset = n_special_nodes,
arcs_list = candidate_arcs 1 : n_special_arcs, heads_tails = heads_tails,
orientations = orientations, parents = parents, use_parenthood = -initialize_tree)

CALL RecordAllocation (n_elements = -SIZE(candidate_arcs, i_wp), mold = l_i_wp)
DEALLOCATE (candidate_arcs, stat = alloc_status)

END IF BuildMST

/ * Now we can build the rest of the tree data-structures. In the case of a spanning forest, there may
be a better ordering of the nodes onto levels, where each tree in the forest is stored in a contiguous
section of level_ordering. So it may be wise to recompute the ordering here, for cache performance
and such? */
ReorderNodes: IF (reorder_nodes) THEN  // We will reorder the nodes to improve cache performance
ALLOCATE (nodes_ordering (0 1 : n_special_nodes + n_nodes), stat = alloc_status)
CALL RecordAllocation (n_elements = n_special_nodes + n_nodes + 1, mold = l_i_wp,
caller = "BuildNetworkMST", alloc_status = alloc_status)
ALLOCATE (nodes_roots (-n_special_nodes : n_nodes), stat = alloc_status)
CALL RecordAllocation (n_elements = n_special_nodes + n_nodes + 1, mold = l_i_wp,
caller = "BuildNetworkMST", alloc_status = alloc_status)

CALL BuildTreeStructures (arc_offset = n_special_arcs, node_offset = n_special_nodes,
hheads_tails = heads_tails, orientations = orientations, parents = parents,
cardinalities = cardinalities, level_ordering = nodes_ordering, tree_mask = tree_mask,
rebuild_cardinalities = rebuild_cardinalities)

offset = 0  // Start here
DefragmentOrdering: DO index = 0, n_nodes + n_special_nodes
  // Offset each root into level_ordering accordingly
  node = nodes_ordering (index)
  orientation = orientations (node)
  IF (orientation = no_parent) THEN  // This is a root node
    level_ordering (offset) = node
    parents (node) = offset  // Save this offset—I am overwriting parents here!
    nodes_roots (node) = node  // This is a root
  END IF
END DO

32
offset = offset + cardinalities_{node}  // Next root will go here
\textbf{cycle} DefragmentOrdering  // Go to next node
\textbf{end if}
parent_{node} = heads_{tails}(orientation, parents_{node})  // This is the parent node
root_{node} = nodes_{roots}_{parent_{node}}
nodes_{roots}_{node} = root_{node}  // Save the root of this forest
parents_{root_{node}} = parents_{root_{node}} + 1  // One more child has been added
level_ordering(parents_{root_{node}}) = node  // Add this child to the ordering
\textbf{end do} DefragmentOrdering
\textbf{call} RecordAllocation(n\_elements = - size(nodes\_ordering, i\_wp), mold = l_{i\_wp})
\textbf{deallocate}(nodes\_ordering, \texttt{STAT} = alloc\_status)
\textbf{call} RecordAllocation(n\_elements = - size(nodes\_roots, i\_wp), mold = l_{i\_wp})
\textbf{deallocate}(nodes\_roots, \texttt{STAT} = alloc\_status)
\textbf{else} ReorderNodes  // Use the default ordering
\textbf{call} BuildTreeStructures(arc\_offset = n\_special\_arcs, node\_offset = n\_special\_nodes, heads\_tails = heads\_tails, orientations = orientations, parents = parents, cardinalities = cardinalities, level\_ordering = level\_ordering, tree\_mask = tree\_mask, rebuild\_cardinalities = rebuild\_cardinalities)
\textbf{end if} ReorderNodes
\textbf{end subroutine} BuildNetworkMST

This code is used in section 1.1.

### 7.2 Incomplete QR preconditioner

This routine computes the Resende/Pardalos incomplete QR factorization of the matrix $AD_A A^T$ in the form $FD_F F^T$, where $F$ is given via the non-unit off-diagonal $\texttt{nodes\_multipliers}$ and the diagonal $D_F$ is given via $\texttt{nodes\_conductances}$. The construction of this preconditioner is very simple, but in practice the incomplete Cholesky factorization of Mehrotra/Wang seems more efficient. This preconditioner approximates a diagonal preconditioner when the MST preconditioner $BD_B B^T$ is very bad, and approximates the MST preconditioner when it is good, so it is sort of an adaptive transition between the two. Similar arguments apply to the the Cholesky factorization, though less formally than the QR preconditioner.

$$\text{FactorizeIncompleteQR 7.2.1} \equiv$$

\textbf{subroutine} FactorizeIncompleteQR(arc\_offset, node\_offset, heads\_tails, orientations, parents, level\_ordering, tree\_mask, arcs\_conductances, nodes\_conductances, nodes\_resistances, nodes\_multipliers)

\texttt{implicit NONE}
\texttt{integer (kind = i\_wp), intent (in) :: arc\_offset, node\_offset}
\texttt{integer (kind = i\_wp), dimension (i, -arc\_offset :), intent (in) :: heads\_tails}
\texttt{integer (kind = i\_wp), dimension (-node\_offset :), intent (in) :: parents}
\texttt{integer (kind = i\_wp), dimension (-node\_offset :), intent (in) :: orientations}
\texttt{integer (kind = i\_wp), dimension (0 :), intent (in) :: level\_ordering}

33
LOGICAL (KIND = l_wp), DIMENSION (-arc_offset : ), INTENT (IN) :: tree_mask
REAL (KIND = r_wp), DIMENSION (-arc_offset : ), INTENT (IN) :: arcs_conductances
REAL (KIND = r_wp), DIMENSION (-node_offset : ), INTENT (OUT), OPTIONAL :: nodes_resistances, nodes_multiplier
REAL (KIND = r_wp), DIMENSION (-node_offset : ), INTENT (OUT) :: nodes_multipliers
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs
INTEGER (KIND = i_wp) :: index, node, arc, parent_arc, parent_node, head, tail
INTEGER :: alloc_status
INTEGER (KIND = i_wp) :: orientation

n_special_nodes = J_BOUND(parents, i_wp, DIM = 1)
n_nodes = J_BOUND(parents, i_wp, DIM = 1)  // The number of nodes, n
n_special_arcs = J_BOUND(heads tails, i_wp, DIM = 2)
n_arcs = J_BOUND(heads tails, i_wp, DIM = 2)  // m

IF (PRESENT(nodes_conductances)) THEN
  _FactorizeIncompleteQR(C, nodes_conductances)
ELSE IF (PRESENT(nodes_resistances)) THEN
  _FactorizeIncompleteQR(R, nodes_resistances)
  nodes_resistances = 1.0_wp / (nodes_resistances + EPSILON(1.0_wp))
END IF

END SUBROUTINE FactorizeIncompleteQR

This code is used in section 1.1.

"WEAVE.f90" 7.2.2 ≡
@m _FactorizeIncompleteQR(ID, nodes_conductances)
nodes_conductances = 0.0_wp

Conductances.@&ID: DO arc = -n_special_arcs, n_arcs
  IF (-tree_mask(arc)) THEN  // Tree arcs only contribute to the child node!
    head = heads tails 1, arc
tail = heads tails 2, arc
    nodes_conductances head = nodes_conductances head + arcs_conductances arc
    nodes_conductances tail = nodes_conductances tail + arcs_conductances arc
  END IF
END DO Conductances.@&ID

Multipliers.@&ID: DO node = -n_special_nodes, n_nodes
  orientation = orientations node
  IF (orientation ≡ no parent) THEN
    nodes_multipliers node = 1.0_wp  // Just a sentinel value
    CYCLE Multipliers.@&ID
  END IF
  parent arc = parents node
  nodes_conductances node = nodes_conductances node + arcs_conductances parent arc
  nodes_multipliers node = arcs_conductances parent arc / nodes_conductances node
END DO Multipliers.@&ID

7.3 Diagonal \(L \text{LDL}^T\) Cholesky preconditioner
This routine computes a root-free full $FDF^T$ Cholesky factorization of $BD_BB^T + \Lambda$ as described by Mehrotra and Wang (here $L = F$). The interesting thing is that this factorization does not require any fill so long as $\Lambda$ is diagonal. Here I take $\Lambda = \text{Diag}(ND_NN^T)$ to be the non-basic contribution to the diagonal of $AD_AA^T$, though Mehrotra/Wang propose some other choices as well without sufficient support.

\[
\text{〈FactorizeIncompleteLDLt 7.3.1〉} \equiv
\]
\[
\text{SUBROUTINE FactorizeIncompleteLDLt}(\text{arc_offset, node_offset, heads_tails, arcs_mask, orientations, parents, level_ordering, tree_mask, arcs_conductances, nodes_conductances, nodes_resistances, nodes_multipliers, diagonal_nodes_conductances)}
\]
\[
\text{IMPLICIT NONE}
\]
\[
\text{INTEGER (KIND = i_up), INTENT (IN)} :: \text{arc_offset, node_offset}
\]
\[
\text{INTEGER (KIND = i_up), DIMENSION (1: -arc_offset :), INTENT (IN)} :: \text{heads_tails}
\]
\[
\text{LOGICAL (KIND = i_up), DIMENSION (-arc_offset :), INTENT (IN), OPTIONAL :: arcs_mask}
\]
\[
\text{INTEGER (KIND = i_up), DIMENSION (-node_offset :), INTENT (IN) :: parents}
\]
\[
\text{INTEGER (KIND = i_up), DIMENSION (0 :), INTENT (IN) :: level_ordering}
\]
\[
\text{LOGICAL (KIND = i_up), DIMENSION (-arc_offset :), INTENT (IN), OPTIONAL :: tree_mask}
\]
\[
\text{// Not really needed, but added to make the same interface as above}
\]
\[
\text{REAL (KIND = r_up), DIMENSION (-arc_offset :), INTENT (IN) :: arcs_conductances}
\]
\[
\text{REAL (KIND = r_up), DIMENSION (-node_offset :), INTENT (OUT), OPTIONAL :: nodes_conductances, nodes_resistances} \quad // y
\]
\[
\text{REAL (KIND = r_up), DIMENSION (-node_offset :), INTENT (OUT) :: nodes_multipliers}
\]
\[
\text{REAL (KIND = r_up), DIMENSION (-node_offset :), INTENT (IN), OPTIONAL :: diagonal_nodes_conductances} \quad // \Lambda \text{ if given explicitly}
\]
\[
\text{INTEGER (KIND = i_up) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs}
\]
\[
\text{INTEGER (KIND = i_up) :: index, node, arc, parent_arc, parent_node, head, tail}
\]
\[
\text{INTEGER :: alloc_status}
\]
\[
\text{INTEGER (KIND = i_up) :: orientation}
\]
\[
\text{n_special_nodes} = -1 \text{BOUND}(parents, i_up, \text{DIM} = 1)
\]
\[
\text{n_nodes} = -1 \text{BOUND}(parents, i_up, \text{DIM} = 1) \quad // \text{The number of nodes, } n
\]
\[
\text{n_special_arcs} = -1 \text{BOUND}(heads_tails, i_up, \text{DIM} = 2)
\]
\[
\text{n_arcs} = -1 \text{BOUND}(heads_tails, i_up, \text{DIM} = 2) \quad // m
\]
\[
\text{IF (PRESENT(nodes_conductances)) THEN}
\]
\[
\text{FactorizeIncompleteLDLt(C, nodes_conductances)}
\]
\[
\text{ELSE IF (PRESENT(nodes_resistances)) THEN}
\]
\[
\text{FactorizeIncompleteLDLt(R, nodes_resistances)}
\]
\[
\text{nodes_resistances} = 1.0_{\text{r}_up} / (\text{nodes_resistances} + \text{EPSILON}(1.0_{\text{r}_up}))
\]
\[
\text{END IF}
\]
\[
\text{END SUBROUTINE FactorizeIncompleteLDLt}
\]

This code is used in section 1.1.

"WEAVE.f90" 7.3.2 ≡

```fortran
@m FactorizeIncompleteLDLt(ID, nodes_conductances)
sizeConductances_@&ID:IF (PRESENT(diagonal_nodes_conductances)) THEN \quad // The diagonal term \Lambda \text{ is given explicitly}
```

35
nodes_conductances = diagonal_nodes_conductances   // Start with non-basic contribution
AddDiagonal @& ID:: do node = -n_special_nodes, n_nodes   // Add the basic diagonal contribution
    orientation = orientations_node
    if (orientation · no_parent) cycle AddDiagonal @& ID
    nodes_conductances_node = nodes_conductances_node + arcs_conductances (parents_node)
end do AddDiagonal @& ID
else
    nodes_conductances = 0.0, _wp
    if (PRESENT (arcs_mask)) then
MaskedConductances @& ID:: do arc = -n_special_arcs, n_arcs
        if (arcs_mask [arc]) then
            head = heads_tails_1, arc
tail = heads_tails_2, arc
            nodes_conductances_head = nodes_conductances_head + arcs_conductances [arc]
            nodes_conductances_tail = nodes_conductances_tail + arcs_conductances [arc]
        end if
end do MaskedConductances @& ID
else
Conductances @& ID:: do arc = -n_special_arcs, n_arcs
    head = heads_tails_1, arc
tail = heads_tails_2, arc
    nodes_conductances_head = nodes_conductances_head + arcs_conductances [arc]
    nodes_conductances_tail = nodes_conductances_tail + arcs_conductances [arc]
end do Conductances @& ID
end if
end if InitializeConductances @& ID
Multipliers @& ID:: do index = n_nodes + n_special_nodes, 0, -1
    node = level_ordering_index
    orientation = orientations_node
    if (orientation · no_parent) then
        nodes_multipliers_node = 1.0, _wp   // Just a sentinel value
    cycle Multipliers @& ID
end if
parent_arc = parents_node
parent_node = heads_tails [orientation, parent_arc]   // This is the parent node
nodes_multipliers_node = arcs_conductances parent_arc / nodes_conductances_node
nodes_conductances parent node = nodes_conductances parent node - nodes_multipliers_node *
    arcs_conductances parent arc
end do Multipliers @& ID

7.4 Solving a Basic Dual Newton System

The following routine actually solves a system of the form \((FD_F F^T)x = b\), where \(F\) is either a pure or non-pure basis of the network. Both nodes multipliers for \(F\) and the conductances/resistances in \(D_F\) are optional. In fact, \(D_F\) can be given as either arc or nodal conductances/resistances, via optional arguments (usually one should supply only 1 argument for \(D_F\)). The potentials of the root nodes should be pre-specified
in \( \text{nodes\_excess\_potentials} \) because the solution can only be determined up to a constant displacement—the potential of the root node (under the assumption that the supply-demand vector \( b \) has zero sum, which it must be in order for the system to be feasible!).

Here I solve the system by step-wise solving the three different systems with \( F, D_F \) and \( F^T \). I am not aware of a way to solve this system in one pass through the arcs, and I believe such a pass does not exist.

\[
\langle \text{BasicDualNewtonSystem 7.4.1} \rangle \equiv
\]

\[
\text{SUBROUTINE BasicDualNewtonSystem (node\_offset, arc\_offset, heads\_tails, orientations, parents, level\_ordering, tree\_mask, nodes\_excess\_potentials, nodes\_excess\_flows, arcs\_conductances, arcs\_resistances, arcs\_excess\_voltages, tree\_arcs\_excess\_voltages, nodes\_conductances, nodes\_resistances, nodes\_multipliers) }
\]

\[
\text{IMPLICIT NONE}
\]

\[
\text{INTEGER, INTENT (IN) :: node\_offset, arc\_offset}
\]

\[
\text{INTEGER (KIND = i\_wp), DIMENSION (, -arc\_offset :), INTENT (IN) :: heads\_tails}
\]

// Heads-tails array

\[
\text{INTEGER (KIND = i\_wp), DIMENSION (-node\_offset :), INTENT (IN) :: parents}
\]

// With displaced counting

\[
\text{INTEGER (KIND = i\_byte), DIMENSION (-node\_offset :), INTENT (IN) :: orientations}
\]

\[
\text{INTEGER (KIND = i\_wp), DIMENSION (0 :), INTENT (IN) :: level\_ordering}
\]

\[
\text{LOGICAL (KIND = L\_wp), DIMENSION (-arc\_offset :), INTENT (IN) :: tree\_mask}
\]

\[
\text{REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (INOUT) :: nodes\_excess\_potentials}
\]

\[
\text{REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (IN) :: nodes\_excess\_flows}
\]

\[
\text{REAL (KIND = r\_wp), DIMENSION (-arc\_offset :), INTENT (IN), OPTIONAL :: arcs\_conductances, arcs\_resistances}
\]

\[
\text{REAL (KIND = r\_wp), DIMENSION (-arc\_offset :), INTENT (OUT), OPTIONAL :: arcs\_excess\_voltages}
\]

// Temporary

\[
\text{REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (OUT),}
\]

\[
\text{OPTIONAL :: tree\_arcs\_excess\_voltages}
\]

// Temporary

\[
\text{REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (IN),}
\]

\[
\text{OPTIONAL :: nodes\_conductances, nodes\_resistances, nodes\_multipliers}
\]

\[
\text{INTEGER (KIND = i\_wp) :: n\_nodes, n\_special\_nodes, n\_special\_arcs, n\_arcs}
\]

\[
\text{INTEGER (KIND = i\_wp) :: index, node, arc, parent\_arc, parent\_node, head, tail}
\]

\[
\text{INTEGER :: alloc\_status}
\]

\[
\text{INTEGER (KIND = i\_byte) :: orientation}
\]

\[
\text{INTEGER :: temp}
\]

\[
\text{n\_special\_nodes = -\_BOUND(parents, i\_wp, DIM = 1)}
\]

\[
\text{n\_nodes = \_BOUND(parents, i\_wp, DIM = 1) // The number of nodes, n}
\]

\[
\text{n\_special\_arcs = -\_BOUND(heads\_tails, i\_wp, DIM = 2)}
\]

\[
\text{n\_arcs = \_BOUND(heads\_tails, i\_wp, DIM = 2) // m}
\]

\textbf{ArcOrNodeBased: IF (PRESENT(arcs\_excess\_voltages)) THEN} // Arc-based temporary

\[
\text{IF (PRESENT(nodes\_multipliers)) THEN}
\]

// Solve \( Fy = b \) using arcs\_excess\_voltages as a temporary for the tree flows

\textbf{CALL PropagateArcsFlows (arc\_offset = arc\_offset, node\_offset = node\_offset, heads\_tails = heads\_tails, orientations = orientations, parents = parents, level\_ordering = level\_ordering, supplies\_demands = nodes\_excess\_flows, arcs\_flows = arcs\_excess\_voltages, nodes\_multipliers = nodes\_multipliers)}

\textbf{ELSE}

37
CALL PropagateArcsFlows (arc_offset = arc_offset, node_offset = node_offset, heads_tails = heads_tails, orientations = orientations, parents = parents, levelOrdering = levelOrdering, supplies_demands = nodes_excess_flows, arcs_flows = arcs_excess_voltages)    // Basic arc flows
END IF

/* Now solve \( D_F t = y \) for \( t \) using a different method depending on how \( D_F \) is specified: */
IF (PRESENT (arcs_conductances)) THEN
  WHERE (tree_mask)
    arcs_excess_voltages = arcs_excess_voltages / arcs_conductances
END IF
IF (PRESENT (arcs_resistances)) THEN
  WHERE (tree_mask)
    arcs_excess_voltages = arcs_excess_voltages * arcs_resistances
END IF
IF (PRESENT (nodes_conductances)) THEN
  Conductances: DO node = 1:n_special_nodes, n_nodes
    IF (orientations_node \( \equiv \) no_parent) CYCLE Conductances
      parent_arc = parent_node
      arcs_excess_voltages_{parent, arc} = arcs_excess_voltages_{parent, arc} / nodes_conductances_{node}
    END DO Conductances
  END IF
IF (PRESENT (nodes_resistances)) THEN
  Resistances: DO node = 1:n_special_nodes, n_nodes
    IF (orientations_node \( \equiv \) no_parent) CYCLE Resistances
      parent_arc = parent_node
      arcs_excess_voltages_{parent, arc} = arcs_excess_voltages_{parent, arc} * nodes_resistances_node
    END DO Resistances
  END IF
IF (PRESENT (nodes_multipliers)) THEN
  // Finally solve \( F^T x = t \) for \( x \)
  CALL PropagateNodesPotentials (arc_offset = arc_offset, node_offset = node_offset, heads_tails = heads_tails, orientations = orientations, parents = parents, levelOrdering = levelOrdering, arcs_voltages = arcs_excess_voltages, nodes_potentials = nodes_excess_potentials, nodes_multipliers = nodes_multipliers)
ELSE
  CALL PropagateNodesPotentials (arc_offset = arc_offset, node_offset = node_offset, heads_tails = heads_tails, orientations = orientations, parents = parents, levelOrdering = levelOrdering, arcs_voltages = arcs_excess_voltages, nodes_potentials = nodes_excess_potentials)
END IF
ELSE IF (PRESENT (tree_arcs_excess_voltages)) THEN
  // Node-based temporary
ELSE IF (PRESENT (nodes_multipliers)) THEN
  CALL PropagateArcsFlows (arc_offset = arc_offset, node_offset = node_offset, heads_tails = heads_tails, orientations = orientations, parents = parents, levelOrdering = levelOrdering, supplies_demands = nodes_excess_flows, tree_arcs_flows = tree_arcs, nodes_multipliers = nodes_multipliers)
ELSE
  CALL PropagateArcsFlows (arc_offset = arc_offset, node_offset = node_offset, heads_tails = heads_tails, orientations = orientations, parents = parents, levelOrdering = levelOrdering, supplies_demands = nodes_excess_flows, tree_arcs_flows = tree_arcs, nodes_multipliers = nodes_multipliers)
END IF

38
END IF

/
* Now solve $D_F t = y$ for $t$ using a different method depending on how $D_F$ is specified: */

IF (PRESENT(arcs_conductances)) THEN
  DO node = -n_special_nodes, n_nodes
    IF (orientations_node $\equiv$ no_parent) THEN // The virtual rooting arc has infinite conductance
      tree_arcs_excess_voltages_node = 0.0, wp
    ELSE
      tree_arcs_excess_voltages_node = tree_arcs_excess_voltages_node / arcs_conductances_parents(node)
    END IF
  END DO
END IF

IF (PRESENT(arcs_resistances)) THEN
  DO node = -n_special_nodes, n_nodes
    IF (orientations_node $\equiv$ no_parent) THEN // The virtual rooting arc has zero resistance
      tree_arcs_excess_voltages_node = 0.0, wp
    ELSE
      tree_arcs_excess_voltages_node = tree_arcs_excess_voltages_node * arcs_resistances_parents(node)
    END IF
  END DO
END IF

IF (PRESENT(nodes_conductances)) THEN
  tree_arcs_excess_voltages = tree_arcs_excess_voltages / nodes_conductances
END IF

IF (PRESENT(nodes_resistances)) THEN
  tree_arcs_excess_voltages = tree_arcs_excess_voltages * nodes_resistances
END IF

IF (PRESENT(nodes_multipliers)) THEN // Finally solve $F^T x = t$ for $x$
  CALL PropagateNodesPotentials (arc_offset = arc_offset, node_offset = node_offset,
    heads_tails = heads_tails, orientations = orientations, parents = parents,
    level_ordering = level_ordering, tree_arcs_voltages = tree_arcs_excess_voltages,
    nodes_potentials = nodes_excess_potentials, nodes_multipliers = nodes_multipliers)
ELSE
  CALL PropagateNodesPotentials (arc_offset = arc_offset, node_offset = node_offset,
    heads_tails = heads_tails, orientations = orientations, parents = parents,
    level_ordering = level_ordering, tree_arcs_voltages = tree_arcs_excess_voltages,
    nodes_potentials = nodes_excess_potentials)
END IF

END IF ArcOrNodeBased

END SUBROUTINE BasicDualNewtonSystem

This code is used in section 1.1.

### 7.5 Calculating Effective Cycle Resistances

The following routine will sum the resistances of the arcs in the BEP cycle of each non-basic arc. This is an operation that can for example be used in preconditioning certain systems of equations:
\( \text{EffectiveCycleResistances } 7.5.1 \) \equiv

**SUBROUTINE** \text{EffectiveCycleResistances} (arc\_offset, node\_offset, heads\_tails, parents, cardinalities, orientations, level\_ordering, tree\_mask, arcs\_resistances, cycles\_resistances)

**IMPLICIT** NONE

INTEGER (KIND = i\_wp), INTENT (IN) :: arc\_offset, node\_offset
INTEGER (KIND = i\_wp), DIMENSION (;, arc\_offset :), INTENT (IN) :: heads\_tails \quad \text{// Heads-tails}
LOGICAL (KIND = i\_wp), DIMENSION (arc\_offset :), INTENT (IN) :: tree\_mask
INTEGER (KIND = i\_wp), DIMENSION (0 :), INTENT (IN) :: level\_ordering \quad \text{// Ordering from root to nodes}
INTEGER (KIND = i\_wp), DIMENSION (node\_offset :), INTENT (IN) :: parents, cardinalities
INTEGER (KIND = i\_wp), DIMENSION (node\_offset :), INTENT (IN) :: orientations
REAL (KIND = r\_wp), DIMENSION (arc\_offset :), INTENT (IN) :: arcs\_resistances \quad \text{// Weights on arcs}
REAL (KIND = r\_wp), DIMENSION (arc\_offset :), INTENT (OUT) :: cycles\_resistances
REAL (KIND = r\_wp), DIMENSION (:), ALLOCATABLE :: nodes\_resistances \quad \text{// Temporary}
INTEGER (KIND = i\_wp) :: n\_nodes, n\_special\_nodes, n\_special\_arcs, n\_arcs
INTEGER (KIND = i\_wp) :: index, node, arc, parent\_arc, parent\_node, climbers\_2, head, tail, parent, cycle\_root
INTEGER (KIND = i\_byte) :: orientation
INTEGER :: side, alloc\_status

n\_special\_nodes = _LB\_BOUND(parents, i\_wp, DIM = 1)
n\_nodes = _UB\_BOUND(parents, i\_wp, DIM = 1) \quad \text{// The number of nodes, n}
n\_special\_arcs = _LB\_BOUND(heads\_tails, i\_wp, DIM = 2)
n\_arcs = _UB\_BOUND(heads\_tails, i\_wp, DIM = 2) \quad \text{// m}

ALLOCATE (nodes\_resistances (n\_special\_nodes : n\_nodes), STAT = alloc\_status)
CALL RecordAllocation(n\_elements = n\_nodes + n\_special\_nodes + 1, mold = 1.0\_wp, caller = "EffectiveCycleResistances", alloc\_status = alloc\_status)

**PathResistances:** DO index = 0, n\_nodes + n\_special\_nodes \quad \text{// Total resistance of path to root}
node = level\_ordering(index)
orientation = orientations\_node
IF (orientation \equiv \text{no_parent}) THEN
    nodes\_resistances\_node = 0.0\_wp \quad \text{// Initialize the root path labels}
    CYCLE PathResistances \quad \text{// Leave root potentials alone!}
END IF
parent\_arc = parents\_node
parent\_node = heads\_tails\_orientation, parent\_arc \quad \text{// This is the parent node}
nodes\_resistances\_node = nodes\_resistances\_parent\_node + arcs\_resistances\_parent\_arc
END DO PathResistances

**CycleResistances:** DO arc = -n\_special\_arcs, n\_arcs
IF (tree\_mask\_arc) THEN \quad \text{// For basic arcs there is no cycle}
    cycles\_resistances\_arc = 0.0\_wp
ELSE \quad \text{// For non-basic arcs we need to trace the BEP cycle}
    head = heads\_tails\_1, arc
tail = heads\_tails\_2, arc
    climbers = ([head, tail]) \quad \text{// Start tracing the cycle formed by arc}
END DO CycleResistances

TraceCycle: DO

40
IF (climbers₁ ≡ climbers₂) THEN
    cycle_root = climbers₁
    EXIT TraceCycle
END IF
IF (cardinalities_climbers₁ ≤ cardinalities_climbers₂) THEN  // Side to climb on
    side = 1  // Climb on side 1
ELSE
    side = 2  // Climb on side 2
END IF
parent = parents(climbers_side)  // Candidate to leave the tree-an arc in the cycle
orientation = orientations(climbers_side)
climbers_side = heads(tails orientation, parent)  // Climb up one level on this side
END DO TraceCycle

cycles_resistances_arc = nodes_resistances_head - nodes_resistances_cycle_root + nodes_resistances_tail -
                        nodes_resistances_cycle_root

END IF
END DO CycleResistances
END SUBROUTINE EffectiveCycleResistances

This code is used in section 1.1.

8 Graph Format Conversion Routines

The following routine will take a heads and tails array and convert it into an adjacency-list-like representation, suitable for use with the SCOTCH library:

8.1 Adjacency lists

〈CreateAdjacencyArrays 8.1.1〉 ≡

SUBROUTINE CreateAdjacencyArrays (arc_offset, node_offset, heads_tails, neighbours,
    my_neighbours, incident_arcs, n_edges, include_loops)
IMPLICIT NONE
INTEGER, INTENT (IN) :: node_offset, arc_offset
INTEGER (KIND = i_up), DIMENSION (:, -arc_offset :) INTENT (IN) :: heads_tails
INTEGER (KIND = i_up), DIMENSION (0 :, INTENT (OUT) :: neighbours, my_neighbours, incident_arcs
INTEGER (KIND = i_up), INTENT (OUT), OPTIONAL :: n_edges
    // The actual number of edges (without self-loops)
LOGICAL, INTENT (IN), OPTIONAL :: include_loops  // Should self loops be included or not (T)
INTEGER (KIND = i_up), DIMENSION (:), ALLOCATABLE :: nodes_degrees  // The nodal degrees
INTEGER (KIND = K_INT) :: n_nodes, n_special_nodes, n_arcs, n_special_arcs, n_vertices  // Counters
INTEGER (KIND = K_INT) :: node, arc, head, tail, special_node, special_arc, degree, head_neighbour,
                        tail_neighbour, node_shift, arc_shift, arc_counter, node_counter
INTEGER :: alloc_status, status
LOGICAL :: graph_OK, add_loops

n_special_nodes = node_offset
n_nodes = LBOUND(my_neighbours, 1, wp, DIM = 1) - 2 - n_special_nodes
    // The first and last entry in my_neighbours are sentinel values!
_n特殊_arcs = LBOUND(heads_tails, 1, wp, DIM = 2)
_n特殊_arcs = n_nodes + n_special_nodes + 1  // For now I add all the vertices always!
node_shift = n_special_nodes + 1  // Shift in node numbering
arc_shift = n_special_arcs + 1  // Shift in arc numbering
add_loops = T  // By default I include self loops
IF (PRESENT(include_loops)) add_loops = include_loops

AllocateNodeArray (nodes_degrees, 1, wp, -ALLOCATED, "CreateAdjacencyArrays")
    */ First we need to calculate nodal degrees: */
nodes_degrees = 0
DO arc = n_special_arcs, n_arcs
    // Note: Here we need to separately add to the head and tail due to loop arcs!
    head = heads_tails[1, arc]
    tail = heads_tails[2, arc]
    IF ((head /= tail) \& add_loops) THEN
        nodes_degrees[head] = nodes_degrees[head] + 1
        nodes_degrees[tail] = nodes_degrees[tail] + 1
    END IF
END DO

/* Now we do a SUM_PREFIX on the nodal degrees to figure out the offsets in my_neighbours: */
arc_counter = 1
DO node = n_special_nodes, n_nodes
    my_neighbours[node + node_shift] = arc_counter
    arc_counter = arc_counter + nodes_degrees[node]
END DO

my_neighbours[0] = 0  // Sentinel value
my_neighbours[n_vertices + 1] = arc_counter  // Sentinel end-of-array value
IF (PRESENT(n_edges)) n_edges = arc_counter - 1

DO arc = n_arcs, -n_special_arcs, -1
    // Now we traverse the arcs in reverse order to make the adjacency lists
    head = heads_tails[1, arc]
    tail = heads_tails[2, arc]
    IF ((head /= tail) \& add_loops) THEN
        nodes_degrees[head] = nodes_degrees[head] - 1
        head_neighbour = my_neighbours[head + node_shift] + nodes_degrees[head]
        nodes_degrees[tail] = nodes_degrees[tail] - 1
        tail_neighbour = my_neighbours[tail + node_shift] + nodes_degrees[tail]
        neighbours[head_neighbour] = tail + node_shift
        neighbours[tail_neighbour] = head + node_shift
        incident_arcs[head_neighbour] = arc
        incident_arcs[tail_neighbour] = arc
    END IF
END DO
neighbours = 0
incident_arcs = 0
_DeallocateArray(nodes_degrees, 1i_wp, ALLOCATED)
END SUBROUTINE CreateAdjacencyArrays

This code is used in section 1.1.

8.2 Tree Partitioning

⟨CalculateCongestionsDilations 8.2.1⟩ ≡

SUBROUTINE CalculateCongestionsDilations(node_offset, arc_offset, heads_tails, orientations,
parents, cardinalities, level_ordering, tree_mask, joints_and_roots, arcs_conductances,
ars_congestions_dilations, nodes_conductances, normalize)
IMPLICIT NONE
INTEGER, INTENT (IN) :: node_offset, arc_offset
INTEGER (KIND = i_wp), DIMENSION (:, -arc_offset :), INTENT (IN) :: heads_tails
// Heads-tails array
INTEGER (KIND = i_wp), DIMENSION (-node_offset :), INTENT (IN) :: parents, cardinalities
INTEGER (KIND = i_wp), DIMENSION (-node_offset :), INTENT (IN) :: orientations
INTEGER (KIND = i_wp), DIMENSION (0 :), INTENT (IN) :: level_ordering
LOGICAL (KIND = l_wp), DIMENSION (-arc_offset :), INTENT (IN) :: tree_mask
INTEGER (KIND = i_wp), DIMENSION (-arc_offset :), INTENT (OUT) :: joints_and_roots
REAL (KIND = r_wp), DIMENSION (-arc_offset :), INTENT (IN) :: arcs_conductances
REAL (KIND = r_wp), DIMENSION (-arc_offset :), INTENT (OUT) :: arcs_congestions_dilations
REAL (KIND = r_wp), DIMENSION (-node_offset :), INTENT (OUT) :: nodes_conductances
LOGICAL, INTENT (IN), OPTIONAL :: normalize // Divide by conductances or not
INTEGER (KIND = i_wp) :: n_nodes, n_special_nodes, n_special_arcs, n_arcs
INTEGER (KIND = i_wp) :: index, node, arc, parent, arc, parent_node, head, tail, climb, cycle_root, parent
INTEGER :: side, alloc_status
INTEGER (KIND = i_byte) :: orientation
INTEGER :: temp

n_special_nodes = -LBOUND(parents, i_wp, DIM = 1)

n_nodes = UBOUND(parents, i_wp, DIM = 1) // The number of nodes, n

n_special_arcs = -LBOUND(heads_tails, i_wp, DIM = 2)

n_arcs = UBOUND(heads_tails, i_wp, DIM = 2) // m

BEP Joints: DO arc = -n_special_arcs, n_arcs
  IF (tree_mask_arc) THEN // For basic arcs there is no cycle
    joints_and_roots_arc = 0
  ELSE // For non-basic arcs we need to trace the BEP cycle
    head = heads_tails_i, arc
  END IF

43
\[
tail = heads_{tails \_2, arc} \\
climbers = ([/ head, tail /]) \quad // \text{Start tracing the cycle formed by arc}
\]

**TraceCycle:**

\[
\text{IF (climbers}_1 \equiv \text{climbers}_2) \text{ THEN} \\
cycle\_root = \text{climbers}_1 \\
\text{EXIT TraceCycle} \\
\text{END IF} \\
\text{IF (cardinalities}_{\text{climbers}_1} \leq \text{cardinalities}_{\text{climbers}_2}) \text{ THEN} \\
\quad // \text{Side to climb on} \\
side = 1 \quad // \text{Climb on side 1} \\
\text{ELSE} \\
side = 2 \quad // \text{Climb on side 2} \\
\text{END IF} \\
parent = \text{parents(\text{climbers}_side)} \quad // \text{Candidate to leave the tree-an arc in the cycle} \\
orientation = \text{orientations(\text{climbers}_side)} \\
\text{IF (orientation} \equiv \text{no}\_\text{parent}) \text{ THEN} \quad // \text{The arc is between two different trees} \\
cycle\_root = 0 \\
\text{EXIT TraceCycle} \\
\text{END IF} \\
climbers\_side = \text{heads}\_\text{tails}_{\text{orientation, parent}} \quad // \text{Climb up one level on this side} \\
\text{END DO TraceCycle} \\
\text{joints\_and\_roots}_{arc} = \text{cycle}_\text{root} \\
\text{END IF} \\
\text{END DO BEPJoints} \\
\text{nodes\_conductances} = 0.0_{\text{\_wp}} \\
\text{DO arc} = -n\_\text{special\_arcs}, n\_\text{arcs} \\
\text{IF (tree\_mask}_{arc} \text{ THEN} \\
\quad \text{nodes\_conductances}_{\text{heads}\_\text{tails}(1:2, arc)} = \text{nodes\_conductances}_{\text{heads}\_\text{tails}(1:2, arc)} + \\
\quad \text{nodes\_conductances}_{\text{arcs\_conductances}} \\
\quad \text{nodes\_conductances}_{\text{joints\_and\_roots}(arc)} = \text{nodes\_conductances}_{\text{joints\_and\_roots}(arc)} - \\
\quad \text{nodes\_conductances}_{\text{arcs\_conductances}} \\
\text{END IF} \\
\text{END DO} \\
\text{CALL PropagateArcsFlows(\text{arc}_\text{offset} = \text{arc}_\text{offset}, \text{node}_\text{offset} = \text{node}_\text{offset},} \\
\text{heads\_tails = heads\_tails, orientations = orientations, parents = parents,} \\
\text{level\_ordering = level\_ordering, supplies\_demands = nodes\_conductances,} \\
\text{arcs\_flows = arcs\_congestions\_dilations)} \\
\text{IF (present(normalize)) THEN} \\
\text{IF (normalize) THEN} \\
\quad \text{WHERE (tree\_mask)} \quad // \text{Calculate the congestions for tree arcs} \\
\quad \text{arcs\_congestions\_dilations} = \text{arcs\_congestions\_dilations}/(\text{arcs\_conductances} + \text{EPSILON}(1.0_{\text{\_wp}})) \\
\quad \text{// We must take ABS later because orientation does not matter} \\
\text{END WHERE} \\
\text{END IF} \\
\text{END IF} \\
\text{/* HOW ABOUT THE ORIENTATIONS OF THE ARCS?! */} \\
\text{nodes\_conductances}_{\text{level\_ordering}(0:1)} = 0.0_{\text{\_wp}} \\
\text{CALL PropagateNodesPotentials(\text{arc}_\text{offset} = \text{arc}_\text{offset}, \text{node}_\text{offset} = \text{node}_\text{offset},} \\
\text{heads\_tails = heads\_tails, orientations = orientations, parents = parents,} \\
44
\begin{verbatim}
level_ordering = level_ordering, arcs_voltages = arcs_congestions_dilations,
nodes_potentials = nodes_conductances)

DO arc = n_special_arcs, n_arcs
  IF (~tree_mask_{arc}) THEN
    arcs_congestions_dilations_{arc} = nodes_conductances_{heads_tails(1, arc)} -
    nodes_conductances_{joints_and_roots(\text{arc})} + nodes_conductances_{heads_tails(2, \text{arc})} -
    nodes_conductances_{joints_and_roots(\text{arc})}
  END IF
END DO

WHERE (tree_mask) arcs_congestions_dilations = ABS(arcs_congestions_dilations)

END SUBROUTINE CalculateCongestionsDilations
\end{verbatim}

This code is used in section 1.1.
@m CASE_TYPE TYPE
@m _TYPE TYPE
@m _NULL > NULL()
@m _PRIVATE PRIVATE
@m _SIZE(array, _kind, ...)
    @ifelse (#0, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array, #), KIND=_kind))
@m _MAXLOC(array, _kind, ...)
    @ifelse (#0, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array, #), KIND=_kind))
@m _MINLOC(array, _kind, ...)
    @ifelse (#0, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array, #), KIND=_kind))
@m _LBOUND(array, _kind, ...)
    @ifelse (#0, 0, INT(LBOUND(array, DIM=1), KIND=_kind),
              INT(LBOUND(array, #), KIND=_kind))
@m _UBOUND(array, _kind, ...)
    @ifelse (#0, 0, INT(UBOUND(array, DIM=1), KIND=_kind),
              INT(UBOUND(array, #), KIND=_kind))
@m _GenericInterface(generic_name, ...)
    INTERFACE generic_name MODULE PROCEDURE #.
    END INTERFACE generic_name
@m _Declare i_word(...)
    INTEGER :: #.
@m _Declare j_wp(...)
    INTEGER (KIND = i_wp) :: #.
@m _Declare r_wp(...)
    REAL (KIND = r_wp) :: #.
@m _Declare r_sp(...)
    REAL (KIND = r_sp) :: #.
@m _Declare r_dp(...)
    REAL (KIND = r_dp) :: #.
@m _FullExtent(_rank) : DO (DIM, 2, _rank) { ; }