1 Module Support_Trees

This file provides the basic routines for manipulating support-tree preconditioners, as used in the module Dual_Network_Solvers.

This documentation is not finished.

"WEAVE.f90" 1.1 ≡

MODULE Support_Trees
    USE Precision
    USE Error_Handling
    USE System_Monitors
    USE Network_Matrix_Operations
    USE Graph_Algorithms
    IMPLICIT NONE
    PUBLIC :: ST_PropagateNodesPotentials, ST_PropagateArcsFlows, ST_PropagateArcsConductances,
             ST_BasicDualNewtonSystem, ST_DualNewtonPreconditioner, ST_ArcsMask,
             ST_MatchingNodesMapping
    PRIVATE
    CONTAINS
    ⟨ST_PropagateNodesPotentials 2.1.0.1⟩
    ⟨ST_PropagateArcsFlows 2.2.0.1⟩
    ⟨ST_PropagateArcsConductances 2.2.1.1⟩
    ⟨ST_BasicDualNewtonSystem 2.2.2.1⟩
    ⟨ST_DualNewtonPreconditioner 2.3.0.1⟩
    ⟨ST_ArcsMask 2.3.1.1⟩
    ⟨ST_MatchingNodesMapping 2.4.0.1⟩
END MODULE Support_Trees

2 Operations on k support trees

2.1 Propagating Potentials

Propagates potentials from the root of the support tree to the leaves:
\{ST\_PropagateNodesPotentials 2.1.0.1\} \equiv

\textbf{SUBROUTINE } ST\_PropagateNodesPotentials (height, degree, arcs\_voltages, nodes\_potentials, potentials\_and\_voltages) \\
\textbf{IMPLICIT} NONE \\
\textbf{INTEGER} (\texttt{KIND} = i\_up), \textbf{INTENT} (\texttt{IN}) :: height, degree \quad // Height of the $k$-way tree \\
\textbf{REAL} (\texttt{KIND} = r\_up), \textbf{DIMENSION} (0 :), \textbf{INTENT} (\texttt{INOUT}), \textbf{OPTIONAL} :: nodes\_potentials, potentials\_and\_voltages \\
\quad // The potentials of the nodes—the roots should have their potentials pre-set! \\
\quad // Here I allow the possibility to use one array potentials\_and\_voltages for both potentials and 
\quad potential drops! \\
\textbf{REAL} (\texttt{KIND} = r\_up), \textbf{DIMENSION} (0 :), \textbf{INTENT} (\texttt{IN}), \textbf{OPTIONAL} :: arcs\_voltages \\
\quad // The potential drops across the arcs-only tree arcs will be used \\
\textbf{INTEGER} (\texttt{KIND} = i\_up) :: n\_nodes, n\_tree\_leafs \\
\textbf{INTEGER} (\texttt{KIND} = i\_up) :: node, parent\_node \\
\begin{align*}
n\_tree\_leafs &= degree^\text{height} \\
n\_nodes &= \left( (n\_tree\_leafs - 1) * degree \right) / (degree - 1) \\
\end{align*} \\
\quad // The number of nodes $n$ excluding the root node 0 \\
\textbf{IF} (\texttt{PRESENT(arcs\_voltages)} \land \texttt{PRESENT(nodes\_potentials)}) \textbf{THEN} \quad // There are two separate arrays 
\quad \texttt{node} = 1 \\
\quad \textbf{DO} parent\_node = 0, n\_nodes - n\_tree\_leafs \\
\quad \quad \texttt{nodes\_potentials}_\texttt{node}:(\texttt{node}+\texttt{degree}-1) = \texttt{nodes\_potentials}_\texttt{parent\_node} + \texttt{arcs\_voltages}_\texttt{node}:(\texttt{node}+\texttt{degree}-1) \\
\quad \quad \texttt{node} = \texttt{node} + \texttt{degree} \\
\quad \textbf{END DO} \\
\textbf{ELSE IF} (\texttt{PRESENT(potentials\_and\_voltages)}) \textbf{THEN} \\
\quad // Here one array is used for both the arc voltages and the node potentials 
\quad \texttt{node} = 1 \\
\quad \textbf{DO} parent\_node = 0, n\_nodes - n\_tree\_leafs \\
\quad \quad \texttt{potentials\_and\_voltages}_\texttt{node}:(\texttt{node}+\texttt{degree}-1) = \texttt{potentials\_and\_voltages}_\texttt{parent\_node} + \\
\quad \quad \texttt{potentials\_and\_voltages}_\texttt{node}:(\texttt{node}+\texttt{degree}-1) \\
\quad \quad \texttt{node} = \texttt{node} + \texttt{degree} \\
\quad \textbf{END DO} \\
\textbf{END IF} \\
\textbf{END SUBROUTINE } ST\_PropagateNodesPotentials

This code is used in section 1.1.1.

\section{2.2 Propagating Flows}

Propagates flow from the nodes of the support-tree to the root:

\{ST\_PropagateArcsFlows 2.2.0.1\} \equiv
SUBROUTINE ST_PropagateArcsFlows (height, degree, supplies_demands, arcs_flows, supplies_and_flows)
IMPLICIT NONE
INTEGER (KIND = i_wp), INTENT (IN) :: height, degree    // Height of the k-way tree
REAL (KIND = r_wp), DIMENSION (0 :), INTENT (IN), OPTIONAL :: supplies_demands
REAL (KIND = r_wp), DIMENSION (0 :), INTENT (OUT), OPTIONAL :: arcs_flows
   // The feasible flows in the tree arcs
REAL (KIND = r_wp), DIMENSION (0 :), INTENT (INOUT), OPTIONAL :: supplies_and_flows
   // To save on memory we can share the two arrays because there are no loop-carried dependencies
INTEGER (KIND = i_wp) :: n_nodes, n_free_leafs
INTEGER (KIND = i_wp) :: node, parent_node, child_node

n_free_leafs = degree * height
n_nodes = ((n_free_leafs - 1) * degree) / (degree - 1)
   // The number of nodes n excluding the root node 0
IF (PRESENT (arcs_flows) AND PRESENT (supplies_demands)) THEN
   // There are two arrays--the code is clearer
   arcs_flows = supplies_demands
   // There may be some unnecessary copying here if only leafs have non-zero supplies demands
   node = n_nodes
   PropagateFlows: DO parent_node = n_nodes - n_free_leafs, 0, -1
      DO child_node = node, node - degree + 1, -1
         arcs_flows (parent_node, child_node) = arcs_flows (parent_node, child_node) + arcs_flows (child_node)
      END DO
      node = node - degree
   END DO
   PropagateFlows
ELSE IF (PRESENT (supplies_and_flows)) THEN    // Use one array for both the excess and in-flows
   node = n_nodes
   DO parent_node = n_nodes - n_free_leafs, 0, -1
      DO child_node = node, node - degree + 1, -1
         supplies_and_flows (parent_node, child_node) = supplies_and_flows (parent_node, child_node) + supplies_and_flows (child_node)
      END DO
      node = node - degree
   END DO
END IF
END SUBROUTINE ST_PropagateArcsFlows

This code is used in section 1.1.

2.2.1 Calculating cut conductances

This routine assigns the conductances to the support-tree arcs based on the cuts between partitions:

\[ \langle \text{ST_PropagateArcsConductances 2.2.1} \rangle \equiv \]
SUBROUTINE ST_PropagateArcsConductances (height, degree, arc_offset, node_offset, heads_tails, nodes_mapping, arcs_conductances, nodes_conductions, nodes_resistances, ST_arcs_conductions, ST_arcs_resistances, ST_nodes_temp)

IMPLICIT NONE

INTEGER (kind = i_wp), INTENT (IN) :: height, degree  // Height of the k-way tree
INTEGER (kind = i_wp), INTENT (IN) :: node_offset, arc_offset
INTEGER (kind = i_wp), DIMENSION (:, -arc_offset,:), INTENT (IN) :: heads_tails
INTEGER (kind = i_wp), DIMENSION (-node_offset,:), INTENT (IN) :: nodes_mapping
REAL (kind = r_wp), DIMENSION (-arc_offset,:), INTENT (OUT), OPTIONAL :: nodes_conductions, nodes_resistances
REAL (kind = r_wp), DIMENSION (-arc_offset,:), INTENT (IN) :: arcs_conductions
REAL (kind = r_wp), DIMENSION (0 :), INTENT (OUT), OPTIONAL :: ST_arcs_conductions, ST_arcs_resistances
REAL (kind = r_wp), DIMENSION (0 :), INTENT (OUT) :: ST_nodes_temp  // A temporary

INTEGER (kind = i_wp) :: n_tree_nodes, n_tree_supernodes, n_tree_leaves
INTEGER (kind = i_wp) :: tree_node, tree_supernode
INTEGER (kind = i_wp) :: n_nodes, n_special_nodes, n_arcs, n_special_arcs
INTEGER (kind = i_wp) :: node, arc, head, tail, joint, first_climber, second_climber
n_special_nodes = -1,BOUND(nodes_mapping, i_wp, DIM = 1)
n_nodes = 1,BOUND(nodes_mapping, i_wp, DIM = 1)  // The number of nodes, n
n_special_arcs = -1,BOUND(heads_tails, i_wp, DIM = 2)

n_tree_leaves = degree**height
n_tree_nodes = ((n_tree_leaves - 1) * degree) / (degree - 1)  // The number of nodes n excluding the root node 0

IF (PRESENT(nodes_conductions)) nodes_conductions = 0.0_r_wp
IF (PRESENT(nodes_resistances)) nodes_resistances = 0.0_r_wp
IF (PRESENT(ST_arcs_conductions) AND PRESENT(nodes_conductions)) THEN
  PropagateConductions(CC, _SumNodesConductions, ST_arcs_conductions, nodes_conductions)
ELSE IF (PRESENT(ST_arcs_resistances) AND PRESENT(nodes_resistances)) THEN
  PropagateConductions(RG, _SumNodesResistances, ST_arcs_resistances, nodes_resistances)
END IF
ST_arcs_resistances = 1.0_r_wp / (ST_arcs_resistances + EPSILON (1.0_r_wp))
ELSE IF (PRESENT(ST_arcs_resistances) AND PRESENT(nodes_resistances)) THEN
  PropagateConductions(CR, _SumNodesConductions, ST_arcs_conductions, nodes_resistances)
END IF
nodes_resistances = 1.0_r_wp / (nodes_resistances + EPSILON (1.0_r_wp))
ELSE IF (PRESENT(ST_arcs_resistances) AND PRESENT(nodes_resistances)) THEN
  PropagateConductions(RR, _SumNodesResistances, ST_arcs_resistances, nodes_resistances)
END IF
nodes_resistances = 1.0_r_wp / (nodes_resistances + EPSILON (1.0_r_wp))
ELSE  // There is no need to calculate nodes conducections/resistances
  IF (PRESENT(ST_arcs_conductions)) THEN
    PropagateConductions(C, _Dummy, ST_arcs_conductions)
  ELSE IF (PRESENT(ST_arcs_resistances)) THEN
    PropagateConductions(R, _Dummy, ST_arcs_resistances)
  END IF
END IF
END SUBROUTINE ST_PropagateArcsConductions

This code is used in section 1.1.
"WEAVE.f90" 2.2.1.2
@m _SumNodesConductances(nodes_conductances, ...)
  nodes_conductances_head = nodes_conductances_head + arcs_conductances_arc
  nodes_conductances_tail = nodes_conductances_tail + arcs_conductances_arc

"WEAVE.f90" 2.2.1.3
@m _PropagateConductances(ID, _ExtraUpdate, ST_arcs_conductances, ...)
  ST_nodes_temp = 0.0_wp
  BoundaryArcs_@ID:DO arc = -n_special_arcs, n_arcs
    head = heads_tails1, arc
tail = heads_tails2, arc
    !_ExtraUpdate(1.)
    head = nodes_mapping_head
tail = nodes_mapping_tail
    IF (head # tail) THEN
      _CalculateJoint(ID, joint, head, tail)  // May be slow
      ST_nodes_temp_head = ST_nodes_temp_head + arcs_conductances_arc
      ST_nodes_temp_tail = ST_nodes_temp_tail + arcs_conductances_arc
      ST_nodes_temp_joint = ST_nodes_temp_joint - arcs_conductances_arc - arcs_conductances_arc
    END IF
  END DO BoundaryArcs_@ID
  CALL ST_PropagateArcsFlows(height = height, degree = degree,
    supplies_demands = ST_nodes_temp, arcs_flows = ST_arcs_conductances)

"WEAVE.f90" 2.2.1.4
@m _CalculateJoint(ID, joint, head, tail)  // Is there a better way to do this
  first_climber = head
  second_climber = tail
  FindJoint_@ID:DO
    IF (first_climber > second_climber) THEN
      first_climber = (first_climber - 1) / degree  // Jump one level up
    ELSE IF (second_climber > first_climber) THEN
      second_climber = (second_climber - 1) / degree
    ELSE
      joint = first_climber
      EXIT FindJoint_@ID
    END IF
  END DO FindJoint_@ID

2.2.2  ST Tree Basis Solver
This is an intermediary routine.

\[
\langle \text{ST\_BasicDualNewtonSystem} \ 22.2.1 \rangle \equiv
\]

**SUBROUTINE** \text{ST\_BasicDualNewtonSystem} (height, degree, nodes\_excess\_potentials,
nodes\_excess\_flows, excess\_potentials\_and\_flows, arcs\_conductances, arcs\_resistances)

**IMPLICIT** NONE

INTEGER (KIND = i\_wp), INTENT (IN) :: height, degree  // Height of the k-way tree
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (INOUT), OPTIONAL :: nodes\_excess\_potentials,
excess\_potentials\_and\_flows
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (IN), OPTIONAL :: nodes\_excess\_flows
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (IN), OPTIONAL :: arcs\_conductances,
arc\_resistances

**IF** (PRESENT(nodes\_excess\_potentials) \& PRES\text{ENT}(nodes\_excess\_flows)) **THEN**

CALL \text{ST\_PropagateAr\text{csFlows}} (height = height, degree = degree,
supplies\_demands = nodes\_excess\_flows, arcs\_flows = nodes\_excess\_potentials)

**IF** (PRESENT(arcs\_conductances)) **THEN**

nodes\_excess\_potentials = nodes\_excess\_potentials / arcs\_conductances

**END IF**

**IF** (PRESENT(arcs\_resistances)) **THEN**

nodes\_excess\_potentials = nodes\_excess\_potentials * arcs\_resistances

**END IF**

nodes\_excess\_potentials\_0 = 0.0_{\text{wp}}

CALL \text{ST\_PropagateNodesPotentials} (height = height, degree = degree,
potentials\_and\_voltages = nodes\_excess\_potentials)

**ELSE IF** (PRESENT(excess\_potentials\_and\_flows)) **THEN**

CALL \text{ST\_PropagateAr\text{csFlows}} (height = height, degree = degree,
supplies\_and\_flows = excess\_potentials\_and\_flows)

**IF** (PRESENT(arcs\_conductances)) **THEN**

excess\_potentials\_and\_flows = excess\_potentials\_and\_flows / arcs\_conductances

**END IF**

**IF** (PRESENT(arcs\_resistances)) **THEN**

excess\_potentials\_and\_flows = excess\_potentials\_and\_flows * arcs\_resistances

**END IF**

excess\_potentials\_and\_flows\_0 = 0.0_{\text{wp}}

CALL \text{ST\_PropagateNodesPotentials} (height = height, degree = degree,
potentials\_and\_voltages = excess\_potentials\_and\_flows)

**END IF**

**END SUBROUTINE** \text{ST\_BasicDualNewtonSystem}

This code is used in section 1.1.

### 2.3 Support-Tree Preconditioner
This is the routine that applies the support-tree preconditioner:

\[
\langle \text{ST\_DualNewtonPreconditioner} \ 23.0.1 \rangle \equiv
\]

**SUBROUTINE** \text{ST\_DualNewtonPreconditioner} (height, degree, node\_offset, nodes\_mapping, 
nodes\_excess\_flows, nodes\_excess\_potentials, nodes\_resistances, ST\_excess\_flows, 
ST\_excess\_potentials, ST\_potentials\_and\_flows, ST\_arcs\_resistances)

**IMPLICIT** 

INTEGER (KIND = i\_wp), INTENT (IN) :: height, degree // Height of the k-way tree
INTEGER (KIND = i\_wp), INTENT (IN) :: node\_offset
INTEGER (KIND = i\_wp), DIMENSION (-node\_offset :), INTENT (IN) :: nodes\_mapping
REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (IN) :: nodes\_excess\_flows
REAL (KIND = r\_wp), DIMENSION (-node\_offset :), INTENT (IN), OPTIONAL :: nodes\_resistances
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (OUT), OPTIONAL :: ST\_potentials\_and\_flows
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (OUT), OPTIONAL :: ST\_excess\_flows,
ST\_excess\_potentials
REAL (KIND = r\_wp), DIMENSION (0 :), INTENT (IN) :: ST\_arcs\_resistances

INTEGER (KIND = i\_wp) :: n\_nodes, n\_special\_nodes
INTEGER (KIND = i\_wp) :: node, special\_node

n\_special\_nodes = J\_BOUND(nodes\_mapping, i\_wp, DIM = 1)
n\_nodes = J\_BOUND(nodes\_mapping, i\_wp, DIM = 1) // The number of nodes, n

\/** Now propagate the excess flows up the support tree and then propagate the excess potentials down the support tree to the leaves: */

TemporaryChoice: IF (PRESENT(ST\_excess\_potentials) \&\& PRESENT(ST\_excess\_flows)) THEN

DO node = -n\_special\_nodes, n\_nodes
 // Collect the excess flow onto the leaves of the support tree

ST\_excess\_flownodes\_mapping(node) = ST\_excess\_flownodes\_mapping(node) + nodes\_excess\_flownode

END DO

CALL ST\_BasicDualNewtonSystem (height = height, degree = degree, 
nodes\_excess\_potentials = ST\_excess\_potentials, nodes\_excess\_flows = ST\_excess\_flows, 
arcs\_resistances = ST\_arcs\_resistances)

DO node = -n\_special\_nodes, n\_nodes // Restore the original values

ST\_excess\_flownodes\_mapping(node) = ST\_excess\_flownodes\_mapping(node) - nodes\_excess\_flownode

END DO

NodesPotential: IF (PRESENT(nodes\_excess\_potentials)) THEN

IF (PRESENT(nodes\_resistances)) THEN

DO node = -n\_special\_nodes, n\_nodes
 // Propagate the excess potentials down to the nodes of the graph

nodes\_excess\_potentials\_node = ST\_excess\_potentials\_node\_mapping(node) + 
nodes\_excess\_flows\_node\_mapping(node) + nodes\_resistances\_node

END DO

ELSE

DO node = -n\_special\_nodes, n\_nodes // Just copy the potential from 
the support\_node // This is not the most efficient way, but it is 
nice

nodes\_excess\_potentials\_node = ST\_excess\_potentials\_node\_mapping(node)

END DO

END IF

END IF

ELSE IF (PRESENT(ST\_potentials\_and\_flows)) THEN // There is one temporary only

7
\[ S_{T, potentials and flows} = 0.0 \times_{wp} \quad / / \text{ Initialize assuming harmonic internal nodes} \]

\{
    \text{DO } \text{node} = -n_{special\_nodes}, n_{nodes} \\
    \quad \text{// Collect the excess flow onto the leafs of the support tree} \\
    \quad S_{T, potentials and flows, mapping}^{n_{nodes}}(\text{node}) = S_{T, potentials and flows, mapping}^{n_{nodes}}(\text{node}) + \\
    \quad \text{nodes, excess flows, mapping}^{n_{nodes}}(\text{node}) \\
    \text{END DO} \\
    \text{CALL } S_{T, BasicDualNewtonSystem}(\text{height} = \text{height}, \text{ degree} = \text{degree}, \text{ excess potentials and flows} = S_{T, potentials and flows}, \text{ arcs, resistances} = S_{T, arcs, resistances}) \\
\}

\text{NodesPotentials PF: IF } (\text{PRESENT}(\text{nodes, excess potentials})) \text{ THEN} \\
\quad \text{IF } (\text{PRESENT}(\text{nodes, resistances})) \text{ THEN} \\
\quad \quad \text{DO } \text{node} = -n_{special\_nodes}, n_{nodes} \\
\quad \quad \quad \text{// Propagate the excess potentials down to the nodes of the graph} \\
\quad \quad \quad \text{nodes, excess potentials, node} = S_{T, potentials and flows, mapping}^{n_{nodes}}(\text{node}) + \\
\quad \quad \quad \text{nodes, excess flows, mapping}^{n_{nodes}}(\text{node}) \times \text{nodes, resistances, node} \\
\quad \quad \text{END DO} \\
\quad \text{ELSE} \\
\quad \quad \text{DO } \text{node} = -n_{special\_nodes}, n_{nodes} / / \text{ Just copy the potential from} \\
\quad \quad \quad \text{the support, node} / / \text{ This is not the most efficient way, but it is} \\
\quad \quad \text{nice: nodes, excess potentials, node} = S_{T, potentials and flows, mapping}^{n_{nodes}}(\text{node}) \text{ ENDDO ENDDIF ENDDIF NodesPF} \\
\}

\text{TemporaryChoice} \\
\text{END SUBROUTINE } S_{T, DualNewtonPreconditioner}

This code is used in section 1.1.

### 2.3.1 Making Masks

This is just a simple short routine that builds a mask for selecting the (by default) internal (or external) arcs, i.e. the arcs that do not cross (or cross) across partitions for a given support tree nodes mapping:

\[
\langle S_{T, ArcsMask} \rangle_{2.3.1.1} \equiv \\
\text{SUBROUTINE } S_{T, ArcsMask}(\text{node, offset}, \text{ arc, offset}, \text{ heads, tails, arcs, mask, nodes, mapping,}} \\
\quad \text{mask, internal, arcs}) \\
\quad \text{IMPLICIT NONE} \\
\quad \text{INTEGER } (\text{KIND} = i_{wp}), \text{ INTENT IN} :: \text{node, offset, arc offset} \\
\quad \text{INTEGER } (\text{KIND} = i_{wp}), \text{ DIMENSION (:, -arc offset):}, \text{ INTENT IN} :: \text{heads, tails} \\
\quad \text{LOGICAL } (\text{KIND} = i_{wp}), \text{ DIMENSION (-arc offset:), INTENT OUT} :: \text{ arcs, mask} \\
\quad \text{INTEGER } (\text{KIND} = i_{wp}), \text{ DIMENSION (-node offset:), INTENT IN} :: \text{nodes, mapping} \\
\quad \text{LOGICAL, INTENT IN}, \text{ OPTIONAL :: mask, internal, arcs} \\
\quad \text{INTEGER } (\text{KIND} = i_{wp}) :: n_{nodes}, n_{special\_nodes}, n_{arc}, n_{special\_arc} \\
\quad \text{INTEGER } (\text{KIND} = i_{wp}) :: \text{ node, arc, head, tail} \\
\quad \text{LOGICAL :: internal, arcs} \\
\quad n_{special\_nodes} = -1 \text{BOUND(nodes, mapping, i_{wp}, DIM = 1)} \\
\quad n_{nodes} = 1 \text{BOUND(nodes, mapping, i_{wp}, DIM = 1)} / / \text{ The number of nodes, n} \\
\quad n_{special\_arc} = -1 \text{BOUND(heads, tails, i_{wp}, DIM = 2)}
\]
2.4 Partitioning Graphs for Support Trees using Matching

This is an experimental routine that implements heaviest graph matching. It works but its effects and utility are not clear:

\[
\text{SUBROUTINE } \text{ST\_MatchingNodesMapping \ (node\_offset, arc\_offset, heads\_tails, arcs\_conductances, } \\
\hspace{1em}\text{nodes\_mapping, n\_random\_matchings, n\_components, mapping\_offset, heavy\_matching)} \\
\text{USE } \text{Sorting\_Ranking} \\
\text{IMPLICIT } \text{NONE} \\
\text{INTEGER } (\text{KIND } = \text{i\_offset}), \text{ DIMENSION ( , -arc\_offset : )}, \text{ INTENT (IN) : heads\_tails} \\
\text{INTEGER } (\text{KIND } = \text{r\_offset}), \text{ DIMENSION ( -arc\_offset : )}, \text{ INTENT (IN), OPTIONAL : arcs\_conductances} \\
\text{INTEGER } (\text{KIND } = \text{r\_offset}), \text{ DIMENSION ( -node\_offset : )}, \text{ INTENT (OUT) : nodes\_mapping} \\
\text{INTEGER } (\text{KIND } = \text{r\_offset}), \text{ INTENT (IN), OPTIONAL : n\_components, mapping\_offset} \\
\text{INTEGER, INTENT (IN), OPTIONAL : n\_random\_matchings} \quad \text{// When to start ordering} \\
\text{LOGICAL, INTENT (IN), OPTIONAL : heavy\_matching} \\
\text{REAL } (\text{KIND } = \text{r\_offset}), \text{ DIMENSION ( )}, \text{ ALLOCATABLE :: arcs\_weights} \\
\text{INTEGER } (\text{KIND } = \text{r\_offset}), \text{ DIMENSION ( )}, \text{ ALLOCATABLE :: supernodes, heights, arcs\_sequence, } \\
\hspace{1em}\text{arc\_reverse\_sequence, end\_nodes, first\_ars, next\_ars} \\
\text{INTEGER } (\text{KIND } = \text{r\_offset}), \text{ DIMENSION ( )}, \text{ ALLOCATABLE :: heads\_tails} \quad \text{// Temporary buffer} \\
\text{INTEGER } (\text{KIND } = \text{i\_byte}), \text{ DIMENSION ( )}, \text{ ALLOCATABLE :: nodes\_mask} \\
\text{INTEGER } (\text{KIND } = \text{i\_offset}) : : n\_nodes, n\_special\_nodes, n\_ars, n\_special\_ars \\
\text{INTEGER } (\text{KIND } = \text{i\_offset}) : : n\_node, arc, head, tail, next\_node, search\_node, n\_alone, index, previous\_arc, \\
\hspace{1em}\text{next\_arc, first\_arc, previous\_search\_arc, n\_unmatched\_nodes, n\_unmatched\_ars, counter, } \\
\hspace{1em}\text{first\_reverse\_arc, search\_arc, adjacent\_node, next\_search\_arc, n\_supernodes, head\_supernode, } \\
\hspace{1em}\text{tail\_supernode, climber, climber\_parent} \\
\text{REAL :: compression}
\]
REAL (kind = r_wp) :: max_weight
INTEGER :: alloc_status, n_contractions, n_random_contractions, debug_unit
INTEGER, PARAMETER :: n_max_idle_contractions = 10 // Used in forced matching
LOGICAL :: order_arcs, preorder_arcs, delete_arc, insert_arc, odd_cycle

order_arcs = F
IF (PRESENT(heavy_matching)) order_arcs = heavy_matching

debug_unit = 10
IF (debug_unit /= 6) THEN
  OPEN (UNIT = debug_unit, FILE = "Matching.log", ACTION = "WRITE", STATUS = "UNKNOWN")
END IF

n_special_nodes = LBOUND(nodes_mapping, i_wp, DIM = 1) // The number of nodes, n
n_nodes = LBOUND(nodes_mapping, i_wp, DIM = 1)

AllocateNodalArray (supernodes, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
AllocateNodalArray (heights, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
AllocateNodalArray (end_nodes, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
AllocateNodalArray (nodes_mask, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
AllocateArcArray (arc_sequence, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
Allocate (heads_tails, i_wp, ALLOCATED, "ST_MatchingNodesMapping", STAT = alloc_status)
call RecordAllocation (n_elements = SIZE(heads_tails, i_wp), mold = i_wp, caller = "ST_MatchingNodesMapping", alloc_status = alloc_status)

heads_tails = heads_tails // Temporary copy of HT to be modified

n_random_contractions = 0
IF (PRESENT(n_random_matchings)) n_random_contractions = n_random_matchings
preorder_arcs = F
IF (order_arcs) THEN
  AllocateNodalArray (first_arcs, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
  // Start of adjacency lists for each node
  AllocateArcArray (next_arcs, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
  // Adjacency lists links
  AllocateArcArray (arc_reverse_sequence, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
  AllocateArcArray (arc_weights, i_wp, ALLOCATED, "ST_MatchingNodesMapping")
  IF (PRESENT(arc_conductances)) THEN
    preorder_arcs = T
    arcs_weights = arc_conductances
  ELSE // Start with all unit weights
    arcs_weights = 1.0_wp
  END IF
END IF

PreorderArcs: IF (preorder_arcs) THEN // Sort the arcs weights
  IF ((n_special_arcs > 0) AND (n_arcs > 0)) arcs_weights = -HUGE(1.0_wp)
  // Make sure arc 0 is first in the list
  call QuickRank (array = arc_weights, permutation = arc_reverse_sequence,
                  partially_ranked = F, pivot_selection = 'R')
  first_arc = arc_reverse_sequence(n_arcs)
  previous_arc = first_arc
DO
  index = n_arcs, -n_special_arcs, -1
  arc = arcs_reverse_sequence_index - n_special_arcs - 1
  arcs_sequence_previous_arc = arc
  previous_arc = arc
END DO
arc_sequence_end = 0 // End the sequence
ELSE // Just order the arcs in default (reverse) order
  first_arc = n_arcs
  // Put regular arcs first, then special arcs, in reverse order
  DO arc = n_arcs, -n_special_arcs + 1, -1
    arcs_sequence_end = arc - 1
  END DO
  arc_sequence_end = n_special_arcs - n_special_arcs = 0 // End the sequence
  IF ((n_special_arcs > 0) AND (n_arcs > 0)) arcs_sequence_end = -1 // Skip over arc 0
END IF
PreorderArcs
@#if 0
  arc = first_arc
  WRITE(*, *) "Arcs list at contraction ", n_contractions
CountArcs: DO // Not masked
  IF (arc .equiv. 0) EXIT CountArcs // We exhausted all arcs
  WRITE(unit = *, fmt = "(15)", advance = "NO") arc
  arc = arcs_sequence_end // Move to the next arc
END DO CountArcs
WRITE(*, *)
PAUSE
@#endif
DO
  node = -n_special_nodes, n_nodes
  supernodes_node = node
  nodes_mapping_node = node
  end_nodes_node = node
END DO
heights = 0 // Do we really need heights??
nodes_mask = matching_node // All nodes available
n_unmatched_nodes = n_special_nodes + n_nodes + 1
n_unmatched_arcs = n_special_arcs + n_arcs + 1
n_contractions = 0
odd_cycle = T
CoarsenGraph: DO WHILE (first_arc .ne. 0)
  n_contractions = n_contractions + 1
  WRITE(debug_unit, *) "At contraction ", n_contractions, " still unmatched arcs=", n_unmatched_arcs, "nodes=", n_unmatched_nodes
  CALL ContractMatchedArcs(node_offset = n_special_nodes, arc_offset = n_special_arcs,
    heads_tails = heads_tails, arcs_sequence = arcs_sequence, first_arc = first_arc,
    supernodes = supernodes, heights = heights, nodes_ordering = nodes_mapping,
    end_nodes = end_nodes, nodes_mask = nodes_mask, reuse_matching = T,
    update_heads_tails = T, odd_cycle = odd_cycle)
  odd_cycle = NOT odd_cycle // Swap the parity here
  IF (order_arcs AND (n_contractions > n_random_contractions)) THEN
    (ContractArcs 2.4.0.2)
END IF
END IF

END DO

IF (PRESENT(mapping_offset)) THEN
    counter = mapping_offset
ELSE
    counter = 0
END IF

n_supernodes = 0
DO node = -n_special_nodes, n_nodes // Can we maybe improve this?
    IF (supernodes_node_eq node) THEN // This is a root node
        n_supernodes = n_supernodes + 1
        search_node = node
    END DO

TraverseList: DO
    next_node = nodes_mapping(search_node)
    nodes_mapping(search_node) = counter
    counter = counter + 1
    IF (next_node_eq search_node) EXIT TraverseList
    search_node = next_node
END DO TraverseList

WRITE (debug_unit, *) "Supernode #", n_supernodes, ["", node,"] ", " children list stored up to index ", counter

END IF
END DO

IF (PRESENT(n_components)) THEN
    compression = REAL(n_components) / REAL(n_nodes + n_special_nodes)
    nodes_mapping = INT(compression * REAL(nodes_mapping), i_wp)
END IF

_DeallocateArray(supernodes, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
_DeallocateArray(heights, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
_DeallocateArray(end_nodes, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
_DeallocateArray(nodes_mask, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
_DeallocateArray(ars_sequence, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
_DeallocateArray(ars_weights, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
CALL RecordAllocation(n_elements = SIZE(heads_tails, i_wp), mold = l_i_wp,
caller = "ST_MatchingNodesMapping")
DEALLOCATE (heads_tails, STAT = alloc_status)

IF (order arcs) THEN
    _DeallocateArray(first_arcs, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
    _DeallocateArray(next_arcs, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
    _DeallocateArray(ars_reverse_sequence, l_i_wp, ALLOCATED, "ST_MatchingNodesMapping")
END IF

END SUBROUTINE ST_MatchingNodesMapping

This code is used in section 1.1.

\langle ContractArcs 2.4.0.2 \rangle \equiv
Mark all the arcs in the list as unchecked, free the nodes adjacency lists, and mark the still active supernodes as available for matching in the next round (I decided to do this here, not in
GraphAlgorithms. Here I also choose to reverse the order of the list, because this will simplify reordering the arcs in correct order while merging duplicate arcs:

```plaintext
previous_arc = 0
arc = first_arc
n_unmatched_arcs = 0
```

**InitializeIndicators:**
```plaintext
IF (arc ≡ 0) EXIT InitializeIndicators  // We exhausted all arcs
n_unmatched_arcs = n_unmatched_arcs + 1
head = heads_tails₁, arc
tail = heads_tails₂, arc
first_arcs_head = 0
first_arcs_tail = 0
arcs_reverse_sequence_arc = previous_arc
previous_arc = arc
arc = arcs_sequence_arc  // Move to next arc
END DO InitializeIndicators
first_reverse_arc = previous_arc  // The old tail is the new head now
```

```plaintext
IF (arc ≡ 0) EXIT CompressArcs  // We exhausted all arcs
```

```plaintext
delete_arc = False
head = heads_tails₁, arc
tail = heads_tails₂, arc
IF (head < tail) THEN
  node = head
  adjacent_node = tail
ELSE IF (head > tail) THEN
  node = tail
  adjacent_node = head
ELSE  // This is a self-loop still not removed
  delete_arc = True
END IF
```

**SearchList:**
```plaintext
IF (~delete_arc) THEN  // Add to the adjacency list if not duplicate
previous_search_arc = 0
search_arc = first_arcs_node  // Head of the list
IF (search_arc ≡ 0)
  n_unmatched_nodes = n_unmatched_nodes + 1
```

**CheckDuplicate:**
```plaintext
IF (search_arc ≡ 0) EXIT CheckDuplicate
search_node = MAX(heads_tails₁, search_arc, heads_tails₂, search_arc)
```

**DuplicateArc:**
```plaintext
IF (search_node ≡ adjacent_node) THEN  // This arc is a duplicate
```
\[ delete\_arc = \top \]
\[ arcs\_weights\_search\_arc = arcs\_weights\_search\_arc + arcs\_weights\_arc \]

// Merge arcs—this is a crucial step in the algorithm

ReorderArc: do  // Find the correct insertion point
  if (previous_search_arc == 0) exit ReorderArc
  if (arcs_weights_search_arc = arcs_weights_previous_search_arc)
    exit ReorderArc
  previous_search_arc = arcs_reversed_sequence_previous_search_arc
end do ReorderArc

// Now reinsert it in the sequence:
InsertArcAfter(search_arc, previous_search_arc, next_search_arc)

exit CheckDuplicate  // There can only be one duplicate here
else if (search_node > adjacent_node) then  // Not a duplicate—stop and insert
  exit CheckDuplicate  // Insert the arc after previous_search.arc
end if DuplicateArc

previous_search_arc = search_arc
search_arc = next arcs search_arc
end do CheckDuplicate

end if SearchList

DeleteArc: if (delete_arc) then  // Delete the arc from arcs_sequence
  DeleteArc(arc, previous_arc, next_arc)
else  // Insert the arc in the adjacency list
  if (previous_search_arc != 0)
    next arc = next arcs previous_search arc
    next arcs previous_search arc = arc
  else
    next arc = first arcs node
    first arcs node = arc
  end if
  next arcs arc = next arc
end if DeleteArc

arc = arcs_sequence arc  // Move to the next arc
end do CompressArcs

@# if 1
n alone = 0  // Number of nodes not matched so far at all
arc = first arc
max weight = arcs weights first arc

UnmatchedNodes: do
  if (arc == 0) exit UnmatchedNodes // We exhausted all arcs
  node = MIN(heads_tails_1, arc; heads_tails_2, arc)
  if (nodes mask node >= MIN(n contractions,
    1 + n max idle contractions / n contractions) + matching_node)
    n alone = n alone + 1
  end if
end do UnmatchedNodes

14
arcs_weights_{arc} = arcs_weights_{arc} + max_weight  // Forcefully push this arc to the front
_DeleteArc(arc, previous_{arc}, next_{arc})

search_{arc} = next_{arc}  // Save the next arcs

next_{arc} = first_{arc}  // Insert near the front of the list

DO  // Find the correct insertion point
  IF (next_{arc} = 0) EXIT
  IF (arcs_weights_{arc} ≥ arcs_weights_{next_{arc}})
    EXIT
    next_{arc} = arcs_sequence_{next_{arc}}
  END DO
_InsertArcBefore(arc, next_{arc}, previous_{arc})

arc = search_{arc}

CYCLE UnmatchedNodes  // Move to the next arc

END IF

arc = arcs_sequence_{arc}  // Move to the next arc

END DO UnmatchedNodes

IF (n_{alone} > 0) WRITE (debug_unit, *) n_{alone}, " nodes have still not been matched at contraction ", n_{contractions}

@endif

This code is used in section 2.4.0.1.

"WEAVE.f90" 2.4.0.3 ≡
@m _DeleteArc(arc, previous_{arc}, next_{arc})

  previous_{arc} = arcs_reverse_sequence_{arc}
  next_{arc} = arcs_sequence_{arc}

  IF (previous_{arc} = 0 ∧ next_{arc} = 0) THEN
    first_{arc} = 0
    first_reverse_{arc} = 0
  ELSE IF (previous_{arc} = 0) THEN  // We must move the pointer to the first arc
    first_{arc} = next_{arc}  // The next arc becomes first
    arcs_reverse_sequence_{first_{arc}} = 0
  ELSE IF (next_{arc} = 0) THEN
    first_reverse_{arc} = previous_{arc}
    arcs_sequence_{first_reverse_{arc}} = 0
  ELSE  // Shortcut this arc in the list
    arcs_sequence_{previous_{arc}} = next_{arc}  // Skip this arc in the list
    arcs_reverse_sequence_{next_{arc}} = previous_{arc}
  END IF

"WEAVE.f90" 2.4.0.4 ≡
@m _InsertArcAfter(arc, previous_{arc}, next_{arc})

  IF (previous_{arc} = 0) THEN
    next_{arc} = first_{arc}
    arcs_reverse_sequence_{next_{arc}} = arc
  first_{arc} = arc
ELSE IF (previous_arc \equiv \text{first\_reverse\_arc}) \text{ THEN}
  next\_arc = 0
  arcs\_sequence_{previous\_arc} = arc
  first\_reverse\_arc = arc
ELSE
  next\_arc = arcs\_sequence_{previous\_arc}
  arcs\_sequence_{previous\_arc} = arc
  arcs\_reverse\_sequence_{next\_arc} = arc
END IF
arcs\_sequence_{arc} = next\_arc
arcs\_reverse\_sequence_{arc} = previous\_arc

@m \_InsertArcBefore (arc, next\_arc, previous\_arc)
IF (next\_arc \equiv \text{first\_arc}) \text{ THEN}
  previous\_arc = 0
  arcs\_reverse\_sequence_{first\_arc} = arc
  first\_arc = arc
ELSE IF (next\_arc \equiv 0) \text{ THEN}  // This means insert at the end
  previous\_arc = first\_reverse\_arc
  arcs\_sequence_{first\_reverse\_arc} = arc
  first\_reverse\_arc = arc
ELSE
  previous\_arc = arcs\_reverse\_sequence_{next\_arc}
  arcs\_reverse\_sequence_{next\_arc} = arc
  arcs\_sequence_{previous\_arc} = arc
END IF
arcs\_sequence_{arc} = next\_arc
arcs\_reverse\_sequence_{arc} = previous\_arc
CASE "TYPE TYPE

m _TYPE TYPE

m _NULL > NULL()

m _PRIVATE PRIVATE

m _SIZE(array, _kind, ...)

m _MAXLOC(array, _kind, ...)

m _MINLOC(array, _kind, ...)

m _LBOUND(array, _kind, ...)

m _UBOUND(array, _kind, ...)

m GenericInterface(generic_name, ...)

INTERFACE generic_name MODULE PROCEDURE #.

END INTERFACE generic_name

m _Declare_i_word(...)

INTEGER :: #.

m _Declare_i_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 _UnExtent(_rank) : DO (DIM, 2, _rank) { : }

m _VarSequence(_variable, _start, _end)

m _NestedLoopStart(_variable, _array, _rank, _kind)

m _DisplayArray(message, array)

IF (SIZE(array) <= 20) THEN

WRITE(message, print_unit, "(A)" message)

WRITE(message, print_unit, "((05.2))" array)

END IF