Spanning-Trees-Related Data Structures

Aleksandar Donev

February 2000

Contents

1 Module *Network.Spanning.Trees* ........................................ 1
   1.1 Allocation/Deallocation of the Spanning-Tree Structures .......... 2
   1.2 Creating a Spanning Tree ........................................... 3
   1.3 Updating a Spanning Tree ......................................... 6

2 Formatting rules for HPF/F90 files .................................... 9
1 Module Network_Spanning_Trees

This module contains some front-end wrapper routines for the spanning-tree routines in module Graph_Algorithms. In particular, global public arrays for the parenthood relations in the tree, the nodal cardinalities and the level-ordering of the tree nodes are given in the arrays tree_nodes_parents, tree_arcs_orientations, tree_nodes_cardinalities and tree_nodes_ordering. Also public are the node path labels tree_path_labels, if needed. Read the documentation of Graph_Algorithms in order to understand these better.

An important thing to notice is that the initialization/creation routines in this module have a different meaning from the ones in the rest of the front-end modules. Usually the initialization routines just read some files or pre-set some variables to default values, while the creation routines then allocate and calculate the arrays, and the destruction routines merely deallocate these. Here the routine InitializeSpanningTree allocates the arrays, but does not assign any values. Values are actually assigned when a real instance of a tree is created with CreateSpanningTree, which builds a random or min/max-weight (cost) tree. The destruction routine DestroySpanningTree then deallocates the tree arrays. This is done so because the tree naturally changes throughout the optimization, even though the network does not (since it is a subset of the whole network). Therefore although the tree-related arrays are allocated throughout, the values are continually changed without changing the size–reallocating. In particular, an MST can be rebuilt if the weights on the arcs change with UpdateSpanningTree at the lowest possible cost.

"WEAVE.f90" 1.0.1

\begin{verbatim}
MODULE Network_Spanning_Trees
  USE Precision
  USE Error_Handling
  USE System_Monitors
  USE Sorting_Ranking
  USE Graph_Algorithms
  USE Network_Data_Structures
  IMPLICIT NONE
  PUBLIC :: InitializeSpanningTree, CreateSpanningTree, UpdateSpanningTree, DestroySpanningTree
  PRIVATE
    PUBLIC :: any_tree, random_tree, min_cost_tree, max_cost_tree
    INTEGER (KIND = i_wp), DIMENSION (:), ALLOCATABLE, PUBLIC :: tree_nodes_parents,
                     tree_nodes_cardinalities, tree_nodes_ordering // Nodal arrays
    REAL (KIND = r_wp), DIMENSION (:), ALLOCATABLE, PUBLIC :: tree_path_labels // Also nodal
    INTEGER (KIND = i_byte), DIMENSION (:), ALLOCATABLE, PUBLIC :: tree_arcs_orientations
                     // Arc array
  CONTAINS
    (InitializeSpanningTree 1.1.1)
    (CreateSpanningTree 1.2.1)
    (UpdateSpanningTree 1.3.1)
    (DestroySpanningTree 1.1.2)
END MODULE Network_Spanning_Trees
\end{verbatim}
1.1 Allocation/Deallocation of the Spanning-Tree Structures

All data structures that I use are arrays—there are absolutely no dynamic data-structures in this module! The initialization routine here allocates all the arrays:

**(InitializeSpanningTree 1.1.1)**

```fortran
SUBROUTINE InitializeSpanningTree()

INTEGER :: alloc_status

ALLOCATE (tree_nodes_parents(-n_special_nodes : n_nodes), STAT = alloc_status)
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = l_iwp,
  caller = "InitializeSpanningTree", alloc_status = alloc_status)
ALLOCATE (tree_arcs_orientations(-n_special_nodes : n_nodes), STAT = alloc_status)
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = l_iwp,
  caller = "InitializeSpanningTree", alloc_status = alloc_status)
ALLOCATE (tree_nodes_cardinalities(-n_special_nodes : n_nodes), STAT = alloc_status)
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = l_iwp,
  caller = "InitializeSpanningTree", alloc_status = alloc_status)
ALLOCATE (tree_nodes_ordering(0 : n_nodes + n_special_nodes), STAT = alloc_status)
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = l_iwp,
  caller = "InitializeSpanningTree", alloc_status = alloc_status)
ALLOCATE (tree_path_labels(-n_special_nodes : n_nodes), STAT = alloc_status)
CALL RecordAllocation(n_elements = n_nodes + n_special_nodes + 1, mold = 1.0_t_wp,
  caller = "InitializeSpanningTree", alloc_status = alloc_status)

tree_arcs_orientations = no_parent  // Initialize to empty forest just in case

END SUBROUTINE InitializeSpanningTree
```

This code is used in section 1.0.1.
\[ \text{DestroySpanningTree } 1.1.2 \equiv \]

\textbf{SUBROUTINE DestroySpanningTree( )}

\textbf{INTEGER :: alloc_status}

\textbf{CALL RecordAllocation(n_elements = -size(tree_nodes_parents, i_wp), mold = 1_i_wp)}
\textbf{DEALLOCATE(tree_nodes_parents)}

\textbf{CALL RecordAllocation(n_elements = -size(tree_arcs Orientations, i_wp), mold = 1_i_byte)}
\textbf{DEALLOCATE(tree_arcs Orientations)}

\textbf{CALL RecordAllocation(n_elements = -size(tree_nodes_cardinalities, i_wp), mold = 1_i_wp)}
\textbf{DEALLOCATE(tree_nodes_cardinalities)}

\textbf{CALL RecordAllocation(n_elements = -size(tree_nodes_ordering, i_wp), mold = 1_i_wp)}
\textbf{DEALLOCATE(tree_nodes_ordering)}

\textbf{CALL RecordAllocation(n_elements = -size(tree_path_labels, i_wp), mold = 1_i_wp)}
\textbf{DEALLOCATE(tree_path_labels)}

\textbf{END SUBROUTINE DestroySpanningTree}

This code is used in section 1.0.1.

### 1.2 Creating a Spanning Tree

This routine creates a spanning tree along with all its data structures (the public tree arrays above). The user can request a given type of tree, either a min/max cost tree (parameters \textit{min_cost_tree} and \textit{max_cost_tree}), a \textit{random_tree} (which requires sorting to calculate a random subset of the arcs, but does not need weights), or \textit{any_tree}, which does not sort the weights on the arcs (and therefore does not need weights) and is thus cheapest (but may produce regular-looking trees).

If requested, the total weight of the spanning tree is returned in \textit{total_weight}. To aid in choosing the sorting algorithm (Hash or Radix for single-precision data or Quick for double-double arithmetic is expensive), the user can give the distribution of the arcs weights in \textit{weights_distribution}. Also, if timing information is needed, the timers \textit{tree_timer} and \textit{thread_timer} can be used to time the creating of the tree (with parenthood relations) and building the cardinalities and thread ordering separately.
§1.2.1 [7] Spanning-Trees-Related Data Structures  Creating a Spanning Tree 4

\(<\text{CreateSpanningTree} 1.2.1>\) ≡

**SUBROUTINE** CreateSpanningTree\( (\text{tree\_type}, \text{arcs\_weights}, \text{total\_weight}, \text{weights\_distribution}, \text{tree\_timer}, \text{thread\_timer}) \)

\(\text{INTEGER (kind = i\_byte), INTENT (IN) :: tree\_type}\)

\(// \text{ Either any\_tree, random\_tree (expensive to calculate) or min\_cost\_tree (max\_cost\_tree)}\)

\(\text{REAL (kind = r\_wp), DIMENSION (:), INTENT (IN), OPTIONAL :: arcs\_weights} // \text{ Indexed from 1!}\)

\(\text{REAL (kind = r\_wp), INTENT (OUT), OPTIONAL :: total\_weight} // \text{ The weight of the MST}\)

\(\text{CHARACTER (len = *), INTENT (OUT), OPTIONAL :: weights\_distribution}\

\(// \text{"Uniform", "Normal" or "Random" (unknown distribution-the default)}\)

\(\text{INTEGER (kind = i\_wp), DIMENSION (:), ALLOCATABLE :: arcs\_ordering}\)

\(\text{INTEGER (kind = i\_wp) :: arc, node, tree\_arc}\)

\(\text{INTEGER :: alloc\_status}\)

\(\text{LOGICAL :: order\_arcs}\)

\(\text{CHARACTER :: distribution}\)

\(\text{distribution = 'R'}\)

\(\text{IF (PRESENT(weights\_distribution))}\)

\(\text{distribution = weights\_distribution}\)

\(\text{order\_arcs = (PRESENT(arcs\_weights) \& \& (tree\_type \neq \text{any\_tree})) \| (tree\_type \equiv \text{random\_tree})}\)

\(// \text{ Order the arcs in a list}\)

\(\text{Order\_Arens: IF (order\_arcs) THEN}\)

\(\text{ALLOCATE (arcs\_ordering(-n\_special\_arcs : n\_arcs), STAT = alloc\_status)}\)

\(\text{CALL Record\_Allocation(n\_elements = n\_special\_arcs + n\_arcs + 1, mold = 1\_wp,}\)

\(\text{caller = "CreateSpanningTree", alloc\_status = alloc\_status})\)

\(\text{SELECT CASE (tree\_type)}\)

\(\text{CASE (min\_cost\_tree, max\_cost\_tree)}\)

\(\text{Single\_Precision: IF (r\_wp \equiv r\_sp) \} Use either hash or radix rank}\)

\(\text{SELECT CASE (distribution)}\)

\(\text{CASE ('U', 'u', 'N', 'n', 'G', 'g')} // Use Hash\_Rank for known distributions}\)

\(\text{CALL Hash\_Rank(array = arcs\_weights, permutation = arcs\_ordering,}\)

\(\text{distribution = distribution, partially\_ranked = F})\)

\(\text{CASE DEFAULT}\)

\(\text{CALL Radix\_Rank(array = arcs\_weights, permutation = arcs\_ordering)}\)

\(\text{END SELECT}\)

\(\text{ELSE} // \text{ Double real arithmetic is too expensive}\)

\(\text{CALL Quick\_Rank(array = arcs\_weights, permutation = arcs\_ordering, pivot\_selection = 'R',}\)

\(\text{partially\_ranked = F})\)

\(\text{END IF Single\_Precision}\)

\(\text{CASE DEFAULT} // \text{ Generate a random ordering of the arcs}\)

\(\text{CALL Disorder\_Permutation (disorder = 1.0, permutation = arcs\_ordering)}\)

\(\text{END SELECT}\)

\(\text{arcs\_ordering = arcs\_ordering - n\_special\_arcs - 1} // \text{ The correctly numbered permutation}\)

\(\text{IF (PRESENT(tree\_timer)) THEN}\)

\(\text{CALL Reset\_Timer (tree\_timer)}\)

\(\text{CALL Start\_Timer(tree\_timer)}\)

\(\text{END IF}\)

\(\text{IF (tree\_type \equiv max\_cost\_tree) THEN} // \text{ Consider arcs in reverse order}\)
CALL BuildSpanningTree \( \text{arc\_offset = } n_{\text{special\_arcs}}, \text{node\_offset = } n_{\text{special\_nodes}}, \text{arc\_list = } \text{arc\_ordering}, n_{\text{arcs}}; \{n_{\text{special\_arcs}}; \{-1\}; \text{heads\_tails = } \text{heads\_tails}, \text{orientations = } \text{tree\_arcs\_orientations}, \text{parents = } \text{tree\_nodes\_parents} \)

ELSE   // Arcs are in right order from small to large weights

CALL BuildSpanningTree \( \text{arc\_offset = } n_{\text{special\_arcs}}, \text{node\_offset = } n_{\text{special\_nodes}}, \text{arc\_list = } \text{arc\_ordering}, \text{heads\_tails = } \text{heads\_tails}, \text{orientations = } \text{tree\_arcs\_orientations}, \text{parents = } \text{tree\_nodes\_parents} \)

END IF

IF \( \text{PRESENT} \left( \text{tree\_timer} \right) \) THEN

CALL StopTimer \( \text{tree\_timer} \)

END IF

CALL RecordAllocation \( \text{n\_elements = } -\text{SIZE} \left( \text{arc\_ordering}, i_{\text{wp}} \right), \text{mold = } 1_{\text{wp}} \)

DEALLOCATE \( \text{arc\_ordering} \)

ELSE   // Consider arcs in natural numbering order

CALL BuildSpanningTree \( \text{arc\_offset = } n_{\text{special\_arcs}}, \text{node\_offset = } n_{\text{special\_nodes}}, \text{arc\_list = } \left\{ \left( \text{arc, arc = } -n_{\text{special\_arcs}}, n_{\text{arcs}} \right) \right\}, \text{heads\_tails = } \text{heads\_tails}, \text{orientations = } \text{tree\_arcs\_orientations}, \text{parents = } \text{tree\_nodes\_parents} \)

END IF

ORDER_ARCs

IF \( \text{PRESENT} \left( \text{thread\_timer} \right) \) THEN

CALL ResetTimer \( \text{thread\_timer} \)

CALL StartTimer \( \text{thread\_timer} \)

END IF

CALL BuildTreeStructures \( \text{arc\_offset = } n_{\text{special\_arcs}}, \text{node\_offset = } n_{\text{special\_nodes}}, \text{heads\_tails = } \text{heads\_tails}, \text{orientations = } \text{tree\_arcs\_orientations}, \text{parents = } \text{tree\_nodes\_parents}, \text{cardinalities = } \text{tree\_nodes\_cardinalities}, \text{level\_ordering = } \text{tree\_nodes\_ordering} \)

IF \( \text{PRESENT} \left( \text{thread\_timer} \right) \) THEN

CALL StopTimer \( \text{thread\_timer} \)

END IF

IF \( \text{PRESENT} \left( \text{total\_weight} \right) \land \text{PRESENT} \left( \text{arc\_weights} \right) \) THEN

\( \text{CalculateTreeWeight 1.2.3} \)

END IF

END SUBROUTINE CreateSpanningTree

This code is used in section 1.0.1.

This piece of code will calculate the total weight of the MST if needed, as is useful in collecting statistics and debugging:
\( \text{CalculateTreeWeight 1.2.3} \equiv \)

\[
\text{total}\_weight = 0.0, \text{wp} \quad // \text{Calculate the total weight of the spanning tree or forest}
\]
\[
\text{DO}\quad \text{node} = -\text{n\_special\_nodes, n\_nodes}
\]
\[
\text{IF (tree\_arcs\_orientations} \text{node} \neq \text{no\_parent}) \text{THEN} \quad // \text{Add this cost to the sum}
\]
\[
\text{tree}\_arc = \text{tree\_nodes}\_parents, \text{node}
\]
\[
\text{total}\_weight = \text{total}\_weight + \text{arcs}\_weights \text{tree}\_arc + \text{n\_special\_arcs} + 1
\]
\[
\text{END IF}
\]
\[
\text{END DO}
\]

This code is used in sections 1.2.1 and 1.3.1.

### 1.3 Updating a Spanning Tree

The following is a wrapper routine for re-building a minimal spanning tree starting from a near optimal tree. It uses a non-greedy approach to achieve this, as found in the routine `ReBuildSpanningTree`. To make the list of eligible arcs for pivoting, this code uses path labels as built in `CalculatePathLabels` and does not use any sorting. Tests indicate that this is a rather efficient procedure. The same optional statistics as in `CreateSpanningTrees` can be gathered if needed, and also the total number of pivots performed can be obtained in `n\_pivots`, as well as the total number of arcs examined for pivoting (cycle traces) in `n\_examined\_arcs`
\(\text{UpdateSpanningTree} \, 1.3.1\) \equiv

\textbf{Subroutine} \, \text{UpdateSpanningTree}(\text{tree\_type}, \text{arcs\_weights}, \text{total\_weight}, \text{n\_examined\_arcs}, \text{n\_pivots},
\text{tree\_timer}, \text{thread\_timer})

\textbf{INTEGER} \, \text{(kind = i\_byte)}, \text{INTENT (IN)} :: \text{tree\_type} \quad \text{// Either min\_cost\_tree or max\_cost\_tree}

\textbf{REAL} \, \text{(kind = r\_wp)}, \text{DIMENSION (:)}, \text{INTENT (IN)}, \text{OPTIONAL :: arcs\_weights}
\quad \text{// These start from 1 here}

\textbf{REAL} \, \text{(kind = r\_wp)}, \text{INTENT (OUT)}, \text{OPTIONAL :: total\_weight} \quad \text{// The weight of the MST}

\textbf{INTEGER} \, \text{(kind = i\_wp)}, \text{INTENT (OUT)}, \text{OPTIONAL :: n\_examined\_arcs}, \text{n\_pivots}
\quad \text{// Statistics on number of pivots and cycle traces}

\textbf{INTEGER, INTENT (IN), OPTIONAL :: tree\_timer, thread\_timer} \quad \text{// Timers to use in profiling}

\textbf{INTEGER} \, \text{(kind = i\_wp)}, \text{DIMENSION (:)}, \text{ALLOCATABLE :: candidate\_arcs, candidate\_order}

\textbf{INTEGER} \, \text{(kind = i\_wp)} :: arc, arc\_index, tree\_arc, node, head, tail, n\_candidate\_arcs

\textbf{INTEGER :: alloc\_status}

\textbf{LOGICAL :: tree\_arc\_test, path\_labels\_test}

\textbf{REAL :: percent\_pivoted}

\textbf{ALLOCATE (candidate\_arcs (0 : n\_special\_arcs + n\_arcs)), STAT = alloc\_status)}

\textbf{CALL RecordAllocation (n\_elements = n\_special\_arcs + n\_arcs + 1, mold = l\_wp,}
\text{caller = "UpdateSpanningTree", alloc\_status = alloc\_status)}

\textbf{CALL StartTimer (tree\_timer)}

\textbf{CALL StartTimer (tree\_timer)}

\textbf{CALL CalculatePathLabels (arc\_offset = n\_special\_arcs, node\_offset = n\_special\_nodes,}
\text{heads\_tails = heads\_tails, level\_ordering = tree\_nodes\_ordering,}
\text{orientations = tree\_arc\_orientations, parents = tree\_nodes\_parents,}
\text{path\_labels = tree\_path\_labels, arcs\_weights = arcs\_weights, tree\_type = tree\_type)}

n\_candidate\_arcs = 0 \quad \text{// How many arcs are eligible to enter the MST}

\textbf{DO arc = 1, n\_arcs + n\_special\_arcs + 1} \quad \text{// Find eligible arcs for entry into tree}
\text{arc\_index = arc - n\_special\_arcs - 1}
\text{head = heads\_tails\_1, arc\_index}
\text{tail = heads\_tails\_2, arc\_index}
\text{/* An eligible arc is a non-tree arc whose weight compares well with the path labels: */}
\text{tree\_arc\_test = \neg (((tree\_arc\_orientations\_head \equiv \text{tail is parent}) \land (tree\_nodes\_parents\_head \equiv arc\_index)) \lor (((tree\_arc\_orientations\_tail \equiv \text{head is parent}) \land (tree\_nodes\_parents\_tail \equiv arc\_index))))}

\textbf{IF (tree\_type = min\_cost\_tree) THEN}
\text{path\_labels\_test = (arcs\_weights\_arc \leq \text{MAX(tree\_path\_labels\_head, tree\_path\_labels\_tail)})}
\text{ELSE}
\text{path\_labels\_test = (arcs\_weights\_arc \geq \text{MIN(tree\_path\_labels\_head, tree\_path\_labels\_tail)})}
\text{END IF}

\textbf{IF (tree\_arc\_test \land path\_labels\_test \land (arc\_index \neq 0)) THEN} \quad \text{// The arc is eligible}
\text{n\_candidate\_arcs = n\_candidate\_arcs + 1}
\text{candidate\_arcs = n\_candidate\_arcs}
\text{END IF}

\textbf{END DO}

\textbf{CALL ReBuildSpanningTree (arc\_offset = n\_special\_arcs, node\_offset = n\_special\_nodes,}
\text{arcs\_list = candidate\_arcs, n\_candidate\_arcs, heads\_tails = heads\_tails,}
\text{arcs\_weights = arcs\_weights, orientations = tree\_arc\_orientations, parents = tree\_nodes\_parents,}
\text{cardinalities = tree\_nodes\_cardinalities, tree\_type = tree\_type, percent\_pivoted = percent\_pivoted)}

\textbf{IF (PRESENT (n\_examined\_arcs)) n\_examined\_arcs = n\_candidate\_arcs}
IF (PRESENT(n_pivots)) n_pivots = INT(percent_pivoted * REAL(n_candidate_arcs), i_wp)

CALL StopTimer (tree_timer)

CALL ResetTimer (thread_timer)

CALL StartTimer (thread_timer)

CALL BuildTreeStructures(arc_offset = n_special_arcs, node_offset = n_special_nodes,
    heads_tails = heads_tails, orientations = tree_arcs_orientations, parents = tree_nodes_parents,
    cardinalities = tree_nodes_cardinalities, level_ordering = tree_nodes_ordering,
    rebuild_cardinalities = false)

CALL StopTimer (thread_timer)

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

DEALLOCATE (candidate_arcs)

IF (PRESENT(total_weight)) THEN
    (CalculateTreeWeight 1.2.3)
ENDIF

END SUBROUTINE UpdateSpanningTree

This code is used in section 1.0.1.
2 Formatting rules for HPF/F90 files

These are just same auxiliary formatting rules and useful macros I use from time to time.

@m _SIZE(array, _kind,...)
   sIFELSE (#0, 0, INT(SIZE(array), KIND=_kind), INT(SIZE(array, #), KIND=_kind))
@m _MAXLOC(array, _kind,...)
   sIFELSE (#0, 0, INT(MAXLOC(array), KIND=_kind), INT(MAXLOC(array, #), KIND=_kind))
@m _MINLOC(array, _kind,...)
   sIFELSE (#0, 0, INT(MINLOC(array), KIND=_kind), INT(MINLOC(array, #), KIND=_kind))
@m _LBOUND(array, _kind,...) sIFELSE (#0, 0, INT(LBOUND(array, DIM=1), KIND=_kind),
   INT(LBOUND(array, #), KIND=_kind))
@m _UBOUND(array, _kind,...) sIFELSE (#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_1WORD( )
   integer :: i,
@m _DECLARE_1WP( )
   integer (kind = i_wp) :: i,
@m _DECLARE_2WP( )
   real (kind = r_wp) :: r,
@m _DECLARE_2SP( )
   real (kind = r_sp) :: r,
@m _DECLARE_2DP( )
   real (kind = r_dp) :: r
@m _FULLEXTENT(_rank) : sDO(DIM, 2, _rank) { , , }
@m _VARIABLE(_variable, _start, _end)
   _variable##_start$DO(DIM, $EVAL(_start + 1), _end) { , _variable$_DIM }
@m _NESTEDLOOP(start##_variable, _array, _rank, _kind)
   $DO (_DIM, _rank, 1, -1) { DO _variable$_DIM = _LBOUND(_array, _kind, DIM = _DIM),
   _LBOUND(_array, _kind, DIM = _DIM) }
@m _NESTEDLOOPEND(_rank) $DO (_DIM, 1, _rank) { END DO }
@m _DUMMY( )
@m _DISPLAYARRAY(message, array)
   IF (SIZE(array) <= 20) THEN
      WRITE(message$print_unit, "(A)" ) message
      WRITE(message$print_unit, "(20G5.2)" ) array
   END IF