Major (Nodal and Arc) Network Arrays

Aleksandar Donev

February 2000

Contents

1 Module Network_Data_Structures  

2 Formatting rules for HPF/F90 files

1

4
1 Module Network Data Structures

This module contains in it all the major counters and arrays connected to the network representation. The module does not do any work itself—it is just a connection between the problem-dependent user procedures and modules and the optimization software itself. I like using modules in this way to avoid excessive argument lists. The optimization libraries themselves will work with any arrays so long as they are prepared in the same way as the arrays declared in this module. Note that all of these variables must be given correct values before passing them to the optimization library, except that initial guesses may sometimes be omitted.

The module variables are all public, so they can be used and manipulated as the user desires. Also, please note that here and everywhere in this library integers that can reach a number proportional to the number of network elements (arcs or nodes) are declared as integer (kind = i_wp) to allow for very large networks, although it is very hard to imagine such a simulation would ever be possible.

n_nodes the number of simple nodes in the network, n_c. Here I only count the nodes where no special boundary conditions, such as fixed-potential conditions, are applied. For example, source and sink nodes will be excluded from this count.

n_special_nodes the number of special nodes in the network, n_p. Special nodes are assumed to have some form of special boundary conditions applied to them and will require separate handling later on. Source and sink nodes will be special for example. The total number of nodes in the network is therefore n = n_c + n_p, although I will often say that n_c = n for brevity.

n_arcs is the number of simple arcs in the network, m_c. These will be handled using the dual Newton method in the algorithm (with possible proximal-point regularization).

n_special_arcs the number of special arcs, m_p. These arcs are zero-cost arcs and will have to be handled separately in the algorithm, for example, with a proximal-point variation. All arcs connected to a special node must also be treated as special, for reasons to become clear soon. Therefore, m = m_c + m_p.

n_dim is a fixed parameter giving the spatial dimensionality of the network. It is not really used for the optimization, but it enters in the nodal coordinate arrays nodes_coordinates, which has several uses in the preparation of the network problem. Therefore I include it here. I made this a fixed parameter in order to utilize better compiler optimization and to avoid allocating myriads of small arrays.

heads_tails heads-and-tails array, to be allocated with dimension (2, −n_special_arcs : n_arcs) and assigned proper values elsewhere. The fact that I start from negative indices is very important! In fact, all arc arrays must have extent (−n_special_arcs : n_arcs) in their dimension corresponding to different arcs. Thus, when dealing with the simple, interior arcs, one can just use heads_tails[1::n_arcs], and then use heads_tails[−n_special_arcsː−1] when dealing with the special arcs—which may index special nodes (see below). The zeroth element is a dummy arc and should be set to have heads_tails[1:2, 0] = 0.
nodes_coords coordinates of the nodes, to be allocated with dimensions \((n_{\text{dim}}, -n_{\text{special\,nodes}} : n_{\text{nodes}})\). Just as with arcs, special nodes will have negative indices, and in fact all nodal arrays must be indexed with extent \((-n_{\text{special\,nodes}} : n_{\text{nodes}})\) in the nodal dimension. Here this is even a more stringent restriction, simply because the heads_fails array contains indices into nodal arrays, so that if the lower bound is messed up these indices will be wrong! Fortran assumes a lower bound of 1 for all assumed-size array dummy arguments. Therefore the lower bound for nodal arrays must be passed to most routines, and it will usually be called node_offset, with a declaration of \((-\text{node\_offset} :)\) for the extent in the nodal dimension for all nodal arrays. Therefore, one should use node_offset = n_special_nodes when working with the special nodes, or node_offset = -1 (just my own choice of sign...) when only working with simple nodes. The zeroth element is a dummy node and can be used for various things, such as rooting trees.

\(\text{arc\_flows}, \text{arc\_voltages}, \text{nodes\_potentials} \text{ and } \text{supplies\_demands}\) the flow vector \(\vec{x}\) arcs_flows \(-n_{\text{special\,arcs}}:n_{\text{arcs}}\), the tension (potential drop) vector \(\vec{t}\), the potentials of the nodes \(\text{nodes\_potentials} = -n_{\text{special\,nodes}}:n_{\text{nodes}}\), and the supply-demand vector \(\vec{b}\) supplies_demands \(-n_{\text{special\,nodes}}:n_{\text{nodes}}\), are some of the major arrays in the optimization algorithm and will either be returned as results of the optimization or are inputs to the optimization.

\(\text{arc\_mask} \text{ and } \text{nodes\_mask}\) are two logical masks which will be used for different purposes at different times. For example, they will be needed when plotting the network with the routines from Network\_Graphics, or they may be used to build spanning trees, etc.

\(n_{\text{cost\,parameters}}\) is related to the separable cost function. The cost function on each arc \(c_{(i,j)}\) depends on the flow through the arc \(x_{(i,j)}\) and some other parameters \(\alpha_{(i,j)}^{(1)}, \alpha_{(i,j)}^{(2)}, \ldots, \alpha_{(i,j)}^{(p)}\) (such as capacity, resistance, etc.)

\[
f(\vec{x}) = \sum_{c(i,j)} f(x_{(i,j)}, \alpha_{(i,j)}^{(1)}, \ldots, \alpha_{(i,j)}^{(p)})
\]

So \(n_{\text{cost\,parameters}} = p\) is the number of parameters entering the cost function which vary from arc to arc. This number of course depends on the particular model, but the general idea is applicable to almost all of our examples, so I include it in this module.

\(n_{\text{cost\,parameters}}\) is an array that holds the vectors \(\vec{c}^{(1)}, \ldots, \vec{c}^{(p)}\) and is thus of extent \((n_{\text{cost\,parameters}}, -n_{\text{special\,arcs}} : n_{\text{arcs}})\). Now, of course, some of the special arcs may not really have these parameters defined, such as zero-cost arcs or infinite-cost arcs. But space should still be allocated for these entries, and proper values to the special status arrays be assigned (read next few items).

\(\text{special\,arcs\_status} \text{ and } \text{special\,nodes\_status}\) are byte integer arrays of extents \((-n_{\text{special\,nodes}} : 0)\) and \((-n_{\text{special\,arcs}} : 0)\) respectively which tell the status (kind) of each special arc and node. The possible kinds of arcs supported so far are in the public parameters low_cost_arc for arcs with zero (i.e. non-strictly convex) costs, high_cost_arcs for arcs with infinite cost (which have little use) and regular_cost_arcs for arcs that follow the above form of the separable cost function. Also possibly dummy_arcs can be used to denote fictional arcs used as sentinels in the argument. For nodes, nodes with fixed supply-demand are denoted via regular_nodes, while the more difficult case of nodes with fixed potentials is denoted via fixed_nodes_potentials. Again dummy_nodes can be used to denote sentinel ghost nodes such as root nodes of spanning trees for example.
\[n_{\text{regular\_cost\_arcs}}, n_{\text{high\_cost\_arcs}}, n_{\text{low\_cost\_arcs}}, n_{\text{regular\_nodes}} \text{ and } n_{\text{fixed\_potential\_nodes}}\] are counters for the different kinds of special nodes and arcs described above. The special nodes should be ordered (i.e., numbered so that) in the nodal arrays in order of increasing status, i.e., first should come the fixed-potential nodes, then the regular special nodes and finally any dummy nodes (which always include the zeroth ghost node). Since we know how many nodes of each kind there are, we don't really need the status array \[\text{special\_nodes\_status}\], however, the requirement to order the special nodes or arcs according to status is a tough design choice which has not been fixed yet, so for now I keep both! Similarly, the special arcs should be ordered according to increasing status, i.e., first should come the low cost, then high-cost, then regular, and finally dummy special arcs. Please remember that all non-special nodes or arcs (i.e., elements with positive indices) must be regular and are this not included in this discussion!

"WEAVE.f90" 1.1 ≡

\[
\text{PROGRAM Network\_Data\_Structures}
\]
\[
\text{USE } \text{Precision}
\]
\[
\text{USE } \text{Error\_Handling}
\]
\[
\text{USE } \text{System\_Monitors}
\]
\[
\text{IMPLICIT } \text{NONE}
\]
\[
\text{PRIVATE}
\]
\[
\text{INTEGER (KIND = i\_wp), PUBLIC, SAVE :: } n_{\text{nodes}}, n_{\text{arcs}}, n_{\text{special\_nodes}}, n_{\text{special\_arcs}}
\]
\[
\text{// n and m}
\]
\[
\text{INTEGER, PARAMETER, PUBLIC :: } n_{\text{dim}} = \text{NDIM} \quad \text{// Spatial dimensionality}
\]
\[
\text{INTEGER (KIND = i\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{heads\_tails}
\]
\[
\text{// Heads-and-tails array}
\]
\[
\text{REAL, DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{nodes\_coords} \quad \text{// Coordinates of nodes}
\]
\[
\text{REAL (KIND = r\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{nodes\_potentials}, \text{supplies\_demands}
\]
\[
\text{REAL (KIND = r\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{arcs\_flows}, \text{arcs\_voltages}
\]
\[
\text{LOGICAL (KIND = L\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{arcs\_mask}, \text{nodes\_mask}
\]
\[
\text{INTEGER, PUBLIC, SAVE :: } \text{n\_cost\_parameters}
\]
\[
\text{REAL (KIND = r\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{arcs\_cost\_parameters}
\]
\[
\text{// Cost parameters for the arcs}
\]
\[
\text{INTEGER (KIND = i\_wp), PUBLIC, SAVE :: } n_{\text{regular\_cost\_arcs}}, n_{\text{high\_cost\_arcs}}, n_{\text{low\_cost\_arcs}},
\]
\[
\text{n_{\text{regular\_nodes}}}, n_{\text{fixed\_potential\_nodes}} = 0
\]
\[
\text{// Special arcs come in 4 flavors, which we count and sort, while there are 3 types of nodes}
\]
\[
\text{INTEGER (KIND = i\_wp), DIMENSION (\ldots), ALLOCATABLE, PUBLIC :: } \text{special\_arcs\_status},
\]
\[
\text{special\_nodes\_status} \quad \text{// T for zero-cost special arcs and for fixed-potential special nodes}
\]
\[
\text{INTEGER (KIND = i\_wp), PARAMETER, PUBLIC :: } \text{low\_cost\_arc} = 1, \text{high\_cost\_arc} = 2,
\]
\[
\text{regular\_cost\_arc} = 3, \text{dummy\_arc} = 4
\]
\[
\text{// Types of arcs supported at present—the ordering is important here}
\]
\[
\text{INTEGER (KIND = i\_wp), PARAMETER, PUBLIC :: } \text{fixed\_potential\_node} = 1, \text{regular\_node} = 2,
\]
\[
\text{dummy\_node} = 3 \quad \text{// Types of nodes supported at present—the ordering is important!}
\]
\[
\text{END MODULE Network\_Data\_Structures}
\]
2 Formatting rules for HPF/F90 files

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

```fortran
@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__WORD(...)
  INTEGER :: #.
@m _DECLARE__WP(...)
  INTEGER (KIND = _WP) :: #.
@m _DECLARE__RP(...)
  REAL (KIND = _RP) :: #.
@m _DECLARE__DP(...)
  REAL (KIND = _DP) :: #.
@m _FULLEXTENT(rank) :: DO (DIM, 2, rank) { , : }
@m _VARSEQUENCE(variable, _start, _end)
  _IFELSE (#, _start DO (DIM, #VAL (_start + 1), _end) { , variable@DIM }
@m _NESTEDLOOPSTART(variable, array, rank, _kind)
  _DO (DIM, _rank, 1, -1) { DO variable@DIM = _LBOUND(array, _kind, DIM = DIM),
                          _UBOUND(array, _kind, DIM = DIM) }
@m _NESTEDLOOPEND(rank) _DO (DIM, 1, _rank) { END DO }
@m _DUMMY(...)
@m _DISPLAYARRAY(message, array)
  IF (_SIZE(array) \leq 20) THEN
    WRITE(message@print_unit, "(A)"") message
    WRITE(message@print_unit, "(2005.2)"") array
  END IF
```