Module 19.2: nag_short_path
Shortest Path Problems

`nag_short_path` contains a procedure for finding the shortest path through a directed or undirected acyclic network.

Contents

Introduction

This module is concerned with finding the shortest path through a *directed* or *undirected acyclic* network, which consists of a set of points called *vertices* and a set of *arcs* that connect certain pairs of distinct vertices. An acyclic network is one in which there are no paths connecting a vertex to itself. An arc whose origin vertex is i and whose destination vertex is j can be written as $i \rightarrow j$. In an undirected network the arcs $i \rightarrow j$ and $j \rightarrow i$ are equivalent (i.e., $i \leftrightarrow j$), whereas in a directed network they are different. Note that the shortest path may not be unique and in some cases may not even exist (e.g., if the network is disconnected).

Procedure: nag_short_path_find

1 Description

nag_short_path_find attempts to determine a path $n_s \rightarrow n_e$ between two specified vertices n_s and n_e of shortest length.

The network is assumed to consist of n vertices which are labelled by the integers $1, 2, \ldots, n$. The lengths of the arcs between the vertices are defined by the n by n distance matrix D, in which the element d_{ij} gives the length of the arc $i \rightarrow j$; $d_{ij} = 0$ if there is no arc connecting vertices i and j (as is the case for an acyclic network when $i = j$). For example, if $n = 4$ and the network is directed, then

$$D = \begin{pmatrix} 0 & d_{12} & d_{13} & d_{14} \\ d_{21} & 0 & d_{23} & d_{24} \\ d_{31} & d_{32} & 0 & d_{34} \\ d_{41} & d_{42} & d_{43} & 0 \end{pmatrix}.$$

If the network is undirected, D is *symmetric* since $d_{ij} = d_{ji}$ (i.e., the length of the arc $i \rightarrow j \equiv$ the length of the arc $j \rightarrow i$).

As the matrix D is usually *sparse*, only the non-zero elements are required and must be supplied via the arguments distance, row_index and col_index.

The method used by nag_short_path_find is described in detail in Section 6.1.

Let m denote the number of vertices between n_s and n_e for which the path $n_s \rightarrow n_e$ is of shortest length. The argument path is a pointer array because the exact value of m is not known in advance. The procedure allocates the required amount of memory to path; on exit from the procedure, $m =$ SIZE(path).

2 Usage

```
USE nag_short_path

CALL nag_short_path_find(num_vertex, distance, row_index, col_index, &
                         short_path_len, path  [, optional arguments])
```

3 Arguments

Note. All array arguments are assumed-shape arrays. The extent in each dimension must be exactly that required by the problem. Notation such as '**x**(n)' is used in the argument descriptions to specify that the array x must have exactly n elements.

This procedure derives the values of the following problem parameters from the shape of the supplied arrays.

n_z — the number of non-zeros

n_z must satisfy the constraint $1 \leq n_z \leq pn(n-1)$, where $p = 1$ if the network is directed and $p = 1/2$ if the network is undirected (see the optional argument undirected).

3.1 Mandatory Arguments

num_vertex — integer, intent(in)

Input: n, the number of vertices.

Constraints: num_vertex ≥ 2.

distance(n_z) — real(kind=*wp*), intent(in)

Input: the non-zero elements of the distance matrix D. More precisely, `distance`(k) must contain the value of the non-zero element with indices (`row_index`(k),`col_index`(k)); this is the length of the arc from the vertex with label `row_index`(k) to the vertex with label `col_index`(k). Elements with the same row and column indices are not allowed. If `undirected` = `.true.`(see Section 3.2), then only those non-zero elements in the strict upper triangle of D need be supplied since $d_{ij} = d_{ji}$.

Constraints: `distance`(k) > 0.0, for $k = 1, 2, \ldots, n_z$.

row_index(n_z) — integer, intent(in)

col_index(n_z) — integer, intent(in)

Input: `row_index`(k) and `col_index`(k) must contain the row and column indices, respectively, for the non-zero element stored in `distance`(k).

Constraints:

if `undirected` = `.false.`, $1 \leq$ `row_index`(k) $\leq$ `num_vertex`, $1 \leq$ `col_index`(k) $\leq$ `num_vertex` and `row_index`(k) $\neq$ `col_index`(k);

if `undirected` = `.true.`, $1 \leq$ `row_index`(k) $<$ `col_index`(k) $\leq$ `num_vertex`.

short_path_len — real(kind=*wp*), intent(out)

Output: the length of the shortest path between the specified vertices n_s and n_e.

path(:) — integer, pointer

Output: contains details of the shortest path between the specified vertices n_s and n_e. More precisely, `first_vertex` = `path`(1) $\rightarrow$ `path`(2) $\rightarrow \ldots \rightarrow$ `path`(m) = `last_vertex`.

Note: the procedure creates a target array of shape (m).

3.2 Optional Arguments

Note. Optional arguments must be supplied by keyword, not by position. The order in which they are described below may differ from the order in which they occur in the argument list.

first_vertex — integer, intent(in), optional

last_vertex — integer, intent(in), optional

Input: n_s and n_e, the labels of the first and last vertices respectively, between which the shortest path is sought.

Constraints:

$1 \leq$ `first_vertex` $\leq$ `num_vertex`,

$1 \leq$ `last_vertex` $\leq$ `num_vertex`,

`first_vertex` $\neq$ `last_vertex`.

Default: `first_vertex` = 1 and `last_vertex` = `num_vertex`.

undirected — logical, intent(in), optional

Input: specifies whether the network is directed or undirected as follows:

if `undirected` = `.true.`, then the network is undirected;

if `undirected` = `.false.`, then the network is directed.

Default: `undirected` = `.false.`.

error — type(nag_error), intent(inout), optional

The NAG *fl*90 error-handling argument. See the Essential Introduction, or the module document nag_error_handling (1.2). You are recommended to omit this argument if you are unsure how to use it. If this argument is supplied, it *must* be initialized by a call to nag_set_error before this procedure is called.

4 Error Codes

Fatal errors (error%level = 3):

error%code	Description
301	An input argument has an invalid value.
302	An array argument has an invalid shape.
303	Array arguments have inconsistent shapes.
320	The procedure was unable to allocate enough memory.

Failures (error%level = 2):

error%code	Description
201	No connected network exists between vertices first_vertex and last_vertex.

5 Examples of Usage

A complete example of the use of this procedure appears in Example 1 of this module document.

6 Further Comments

6.1 Algorithmic Detail

This procedure is based upon Dijkstra's algorithm (see [1]), which attempts to find a path $n_s \rightarrow n_e$ between two specified vertices n_s and n_e of shortest length $d(n_s, n_e)$.

The algorithm proceeds by assigning labels to each vertex, which may be *temporary* or *permanent*. A temporary label can be changed, whereas a permanent one cannot. For example, if vertex p has a permanent label (q, r) then r is the distance $d(n_s, r)$ and q is the previous vertex on a shortest length $n_s \rightarrow p$ path. If the label is temporary, then it has the same meaning but it refers only to the shortest $n_s \rightarrow p$ path found so far. A shorter one may be found later, in which case the label may become permanent.

The algorithm consists of the following steps.

1. Assign the permanent label $(-, 0)$ to vertex n_s and temporary labels $(-, \infty)$ to every other vertex. Set $k = n_s$ and go to 2.

2. Consider each vertex y adjacent to vertex k with a temporary label in turn. Let the label at k be (p, q) and at y (r, s). If $q + d_{ky} < s$, then a new temporary label $(k, q + d_{ky})$ is assigned to vertex y; otherwise no change is made in the label of y. When all vertices y with temporary labels adjacent to k have been considered, go to 3.

3. From the set of temporary labels, select the one with the smallest second component and declare that label to be permanent. The vertex it is attached to becomes the new vertex k. If $k = n_e$ go to 4. Otherwise go to 2 unless no new vertex can be found (e.g., when the set of temporary labels is 'empty' but $k \neq n_e$, in which case no connected network exists between vertices n_s and n_e).

4. To find the shortest path, let (y, z) denote the label of vertex n_e. The column label (z) gives $d(n_s, n_e)$ while the row label (y) then links back to the previous vertex on a shortest length $n_s \rightarrow n_e$ path. Go to vertex y. Suppose that the (permanent) label of vertex y is (w, x), then the next previous vertex is w on a shortest length $n_s \rightarrow y$ path. This process continues until vertex n_s is reached. Hence the shortest path is

$$n_s \rightarrow \ldots \rightarrow w \rightarrow y \rightarrow n_e,$$

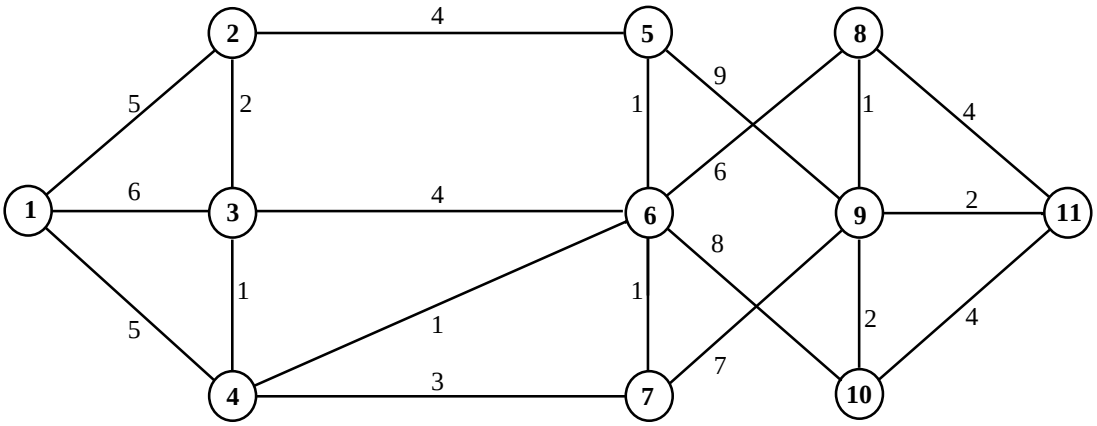
which has length $d(n_s, n_e)$.

6.2 Accuracy

The results are exact, except for the obvious errors in summing the distances in the length of the shortest path.

Example 1: Shortest Path Problem

To find the shortest path between vertices 1 and 11 for the undirected network



1 Program Text

Note. The listing of the example program presented below is double precision. Single precision users are referred to Section 5.2 of the Essential Introduction for further information.

```
PROGRAM nag_short_path_ex01

  ! Example Program Text for nag_short_path
  ! NAG f190, Release 4. NAG Copyright 2000.

  ! .. Use Statements ..
  USE nag_examples_io, ONLY : nag_std_in, nag_std_out
  USE nag_short_path, ONLY : nag_short_path_find
  ! .. Implicit None Statement ..
  IMPLICIT NONE
  ! .. Intrinsic Functions ..
  INTRINSIC KIND
  ! .. Parameters ..
  INTEGER, PARAMETER :: wp = KIND(1.0D0)
  ! .. Local Scalars ..
  INTEGER :: j, nnz, num_vertex
  REAL (wp) :: short_path_len
  LOGICAL :: undirected
  ! .. Local Arrays ..
  INTEGER, ALLOCATABLE :: col_index(:), row_index(:)
  INTEGER, POINTER :: path(:)
  REAL (wp), ALLOCATABLE :: distance(:)
  ! .. Executable Statements ..
  WRITE (nag_std_out,*) 'Example Program Results for nag_short_path_ex01'

  READ (nag_std_in,*)          ! Skip heading in data file

  ! Read number of non-zeros (nnz)
  READ (nag_std_in,*) nnz

  ALLOCATE (col_index(nnz),row_index(nnz),distance(nnz)) ! Allocate storage

  ! Read in problem data
  READ (nag_std_in,*) num_vertex, undirected
  READ (nag_std_in,*) (distance(j),row_index(j),col_index(j),j=1,nnz)

  ! Find the shortest path between vertices 1 and num_vertex
```

```
      CALL nag_short_path_find(num_vertex,distance,row_index,col_index, &
       short_path_len,path,undirected=undirected)

      ! Print details of shortest path

      WRITE (nag_std_out,fmt='(/ 1x, a, 10(i2, :, '' to ''))') &
       'Shortest path = ', path
      WRITE (nag_std_out,fmt='(/ 1x, a, g16.7 )') 'Length of shortest path = ' &
       , short_path_len

      DEALLOCATE (col_index,row_index,distance,path) ! Deallocate storage

    END PROGRAM nag_short_path_ex01
```

2 Program Data

```
Example Program Data for nag_short_path_ex01
 20            : nnz
 11   .TRUE.   : num_vertex, undirected
 6.0   6   8
 1.0   8   9
 2.0   9  11
 4.0   2   5
 1.0   3   4
 6.0   1   3
 4.0   3   6
 1.0   4   6
 2.0   2   3
 3.0   4   7
 5.0   1   2
 7.0   6  10
 1.0   5   6
 4.0   8  11
 9.0   5   9
 1.0   6   7
 8.0   7   9
 4.0  10  11
 2.0   9  10
 5.0   1   4   : distance, row_index, col_index
```

3 Program Results

```
 Example Program Results for nag_short_path_ex01

 Shortest path =  1 to  4 to  6 to  8 to  9 to 11

 Length of shortest path =     15.00000
```

References

[1] Dijkstra E W (1959) A note on two problems in connection with graphs *Numer. Math.* **1** 269–271