Source Code

July 13, 2018

Module Files

Graph Module (File : GraphMod.f90)

```fortran
module GraphMod

    implicit none

! ......... derived type to hold Graph Structure
    type Graph

         logical                             :: isdirected
         integer                             :: n                !no.
             of vertices
         integer                             :: m                !no.
             of edges
         integer, dimension( :, : ), allocatable :: edges

    end type Graph


! ......... overloading the assignment operator with subroutine
    incidenceMatrix
    interface assignment ( = )
        module procedure incidenceMatrix
    end interface


    contains


! .......... function to compute the adjacency matrix of the graph
    function adjacency( G ) result( A )

          implicit none

        type ( Graph ), intent( in )   :: G
        integer, dimension( G%n, G%n ) :: A
        integer                        :: i,j

        A = 0
        call confirmNoSelfLoop( G )

        if( G%isdirected .eqv. .false. ) then
              do i = 1, G%m
                A( G%edges( i, 2 ), G%edges( i, 1 ) ) = 1
                A( G%edges( i, 1 ), G%edges( i, 2 ) ) = 1
            end do
        else
             do i = 1, G%m
                   A( G%edges( i, 2 ), G%edges( i, 1 ) ) = 1
                 A( G%edges( i, 1 ), G%edges( i, 2 ) ) = -1
             end do
        end if

   end function


! ............ subroutine to compute the adjacency matrix of the graph
    subroutine adjacencyMatrix( A, G )

          implicit none

        type( Graph ), intent( in )      :: G
        integer, allocatable, intent( out ) :: A( :, : )
        integer                          :: i,j

        allocate( A ( G%n, G%n ) )

        A = 0

        call confirmNoSelfLoop( G )

        if( G%isdirected .eqv. .false. ) then
             do i = 1, G%m
                   A( G%edges( i, 2 ), G%edges( i, 1 ) ) = 1
                   A( G%edges( i, 1 ), G%edges( i, 2 ) ) = 1
             end do
        else
             do i = 1, G%m
                   A( G%edges( i, 2 ), G%edges( i, 1 ) ) = 1
                   A( G%edges( i, 1 ), G%edges( i, 2 ) ) = -1
              end do
          end if

    end subroutine


!   .......... function to compute the incidence matrix of the of the
    graph
    function incidence( G ) result( A )


          implicit none
          type ( Graph ) , intent( in ) :: G
          integer, dimension ( G%n, G%m ) :: A
          integer                      :: i,j

          A = 0

          call confirmNoSelfLoop( G )

          if( G%isdirected .eqv. .false. ) then
              do i = 1, G%m
                   A( G%edges( i, 2 ), i ) = 1
                   A( G%edges( i, 1 ), i ) = 1
              end do
        else
             do i = 1, G%m
                      A( G%edges( i, 2 ), i ) = 1
                      A( G%edges( i, 1 ), i ) = -1
                end do
        end if

    end function

!   .......... subroutine to compute the incidence matrix of the of the
    graph
   subroutine incidenceMatrix( A, G )

        implicit none

        type( Graph ), intent( in )      :: G
           integer, allocatable, intent( out ) :: A( :, : )
        integer                          :: i,j

        allocate( A( G%n, G%m ) )

        A = 0

        call confirmNoSelfLoop( G )

        if( G%isdirected .eqv. .false. ) then
             do i = 1, G%m
                   A( G%edges( i, 2 ), i ) = 1
                   A( G%edges( i, 1 ), i ) = 1
             end do
        else
             do i = 1, G%m
                   A( G%edges( i, 2 ), i ) = 1
                   A( G%edges( i, 1 ), i ) = -1
             end do
        end if

   end subroutine
```

```
!   ......... subroutine to confirm that there existed no self loops in
    the graph.
   subroutine confirmNoSelfLoop(G)

        type( Graph ), intent( in ) :: G
        integer                  :: i

        do i = 1, G%m
              if( G%edges( i, 1 ) == G%edges( i, 2 ) ) then
                  print *, "Self Loop Not Allowed. Exiting ..."
                  stop
             end if
        end do

    end subroutine


end module GraphMod

!Note: Self Loop is an edge from a node to itself
```

GraphUserInterface Module (File : GraphUserInterface.f90)

```
module GraphUserinterfaceMod

    use GraphMod

    implicit none


    contains

!   .......... subroutine to read data from a file and initialize the
    Graph
    subroutine readGraphData(G,filename)

        implicit none

          type( Graph ), intent( out )    :: G
        character( len = * ), intent( in ) :: filename
        integer                         :: ios,i,j,v1,v2,directed

         !open the input file
        open(unit = 10, file = filename, status = 'old', iostat = ios)

          !check for error in opening the file
        if(ios.ne.0) then
            print *,"Error in Opening Graph Input File in
                GraphUserInterface::readGraphData()"
             stop
        endif
        read( 10, * ) G%n, G%m, directed

        if ( directed == 1 ) then
             G%isdirected = .true.
        else
             G%isdirected = .false.
          end if


        allocate( G%edges( G%m, 2 ) )

        do i = 1, G%m
           read( 10, * ) v1, v2
            G%edges(i,1) = v1
            G%edges(i,2) = v2
        end do

    end subroutine


!   ......... subroutine to pretty-print the given 2-D matrix
    subroutine printMatrix( A )

       implicit none

        integer, dimension( :, : ), intent( in ) :: A
        integer                                  :: i,j
        integer                                  :: shapeArray(2)

        shapeArray = shape(A)

        do i = 1, shapeArray( 1 )
            write( *, * ) ( A( i, j ) , j = 1,shapeArray( 2 ) )
        end do

        write( *, * )
        write( *, * )

    end subroutine

end module
```

Graph Input (File : GraphInput.dat)

Format:
Number of nodes   Number of edges   Directed_or_not
Node1   Node2
.
.
GraphInput.dat
3 2 1

1 2
2 3

NetworkFlow Module (File: NetworkFlowMod.f90)

```
module NetworkFlowMod

    use GraphMod

    implicit none

!   ......... Derived Type to hold a Network Structure
    type Network

       type( Graph )                     :: G
       real, dimension( :, : ), allocatable :: FlowVector
       real, dimension( : ), allocatable :: CapacityVector
       real, dimension( : ), allocatable :: CostVector
       real, dimension( :, : ), allocatable :: VertexFlow

    end type


     contains


!   ......... subroutine to set the source, destination and flow of a
    given Graph
     subroutine setSourceDestinationFlow(N,A)

           implicit none

           real, intent( in ), dimension( :, : ) :: A
           type( Network ), intent( inout )  :: N
           integer                           :: shapeArray(2)

           shapeArray = shape( A )

           allocate( N%VertexFlow( shapeArray(1), shapeArray(2) ) )
           allocate( N%FlowVector( N%G%m, shapeArray(2) ) )

           N%FlowVector = 0
           N%VertexFlow = A

     end subroutine

!   ............. subroutine to convert a directed Network to a
    bidirectional edge
     subroutine bidirectional( BN, N )

           implicit none
           type( Network ), intent( in ) :: N
           type( Network ), intent( out ) :: BN
           integer                     :: i,shapeArray(2)

           BN%G%n = N%G%n
           BN%G%m = 2*N%G%m
           BN%G%isdirected = .true.

           shapeArray = shape( N%VertexFlow )

           allocate( BN%G%edges( BN%G%m, 2 ) )
           allocate( BN%CostVector( BN%G%m ) )
           allocate( BN%CapacityVector( BN%G%m ) )
           allocate( BN%VertexFlow( shapeArray(1), shapeArray(2) ) )


           do i = 1, N%G%m
                BN%G%edges( 2*i - 1, 1 ) = N%G%edges( i, 1 )
                BN%G%edges( 2*i - 1, 2 ) = N%G%edges( i, 2 )
                BN%G%edges( 2*i , 1 ) = N%G%edges( i, 2 )
                BN%G%edges( 2*i , 2 ) = N%G%edges( i, 1 )

                BN%CostVector( 2*i - 1 ) = N%CostVector( i )
                BN%CostVector( 2*i ) = N%CostVector( i )

                BN%CapacityVector( 2*i - 1 ) = N%CapacityVector( i )
                BN%CapacityVector( 2*i ) = N%CapacityVector( i )
           end do

        BN%VertexFlow = N%VertexFlow

     end subroutine

end module
```

NetworkUserInterface Module(File : NetworkUserInterface.f90)

```
module NetworkUserInterfaceMod

     use GraphMod
    use NetworkFlowMod

    implicit none


    contains

!   .......... subroutine to load the network data from a file to Network
    subroutine readNetworkData( N, filename )

       implicit none
       type( Network ), intent( out )  :: N
       character( len = * ), intent( in ) :: filename
       integer                         :: ios,i,directed

       open(unit = 10, file = filename, status = 'old', iostat = ios)


       if(ios .ne. 0) then
           print *, "Error in Opening File in
               NetworkUserInterfaceMod::readNetworkData()"
           stop
       endif

       read(10,*) N%G%n, N%G%m

       N%G%isdirected = .true.

       allocate( N%CostVector( N%G%m ) )
       allocate( N%G%edges( N%G%m, 2 ) )
       allocate( N%CapacityVector( N%G%m ) )

       do i = 1, N%G%m
           read( 10, * ) N%G%edges( i, 1 ), N%G%edges( i, 2 ),
               N%CostVector( i ),N%CapacityVector( i )
       end do

       close( 10 )

   end subroutine

!  .......... subroutine to load data regarding flow into the Network
   subroutine setSourceDestinationFlow_(N,numcommodities,filename)

       implicit none

       type( Network ), intent( inout ) :: N
       integer, intent( out )        :: numcommodities
       character( len = * )          :: filename
       integer                       ::src,dest,ios,i
       real                          ::flow

       open( unit = 10, file = filename, status='old', iostat=ios )

       if(ios .ne. 0) then
           print *, "Error in opening File. Exit Code: ",ios
           stop
       end if

       read( 10, * ) numcommodities

       allocate( N%VertexFlow( N%G%n, numcommodities ) )
       allocate( N%FlowVector( N%G%m, numcommodities ) )

       N%FlowVector = 0
       N%VertexFlow = 0

       do i = 1, numcommodities
           read( 10, * ) src, dest, flow
           N%VertexFlow( src, i ) = -flow
           N%VertexFlow( dest, i ) = flow
       end do

    end subroutine

!   ......... subroutine to read data regarding flow into an 2-D array
    subroutine readSourceDestinationFlowData(A,filename)

        implicit none

        real, allocatable, intent( out ) :: A( :, : )
        character( len = * ), intent( in ) :: filename
        integer                         :: numcommodities, n, src, dest,
            ios, i
       real                             :: flow

        open( unit = 10, file = filename, status='old', iostat=ios )

       if( ios .ne. 0 ) then
           print *, "Error in opening File. Exit Code: ", ios
           stop
       end if

       read( 10, * ) n, numcommodities

       allocate( A( n, numcommodities ) )

       do i = 1, numcommodities
           read( 10, * ) src, dest, flow
           A( src, i ) = -flow
           A( dest, i ) = flow
       end do

    end subroutine

!   ......... subroutine to read data regarding flow from the user
     subroutine uiSourceDestinationFlow_(src,dest,flow)

       implicit none

       integer, intent( out ) :: src, dest
       real, intent( out )  :: flow

       print *, "Enter Source Node, Destination Node and Flow "
       read( *, * ) src, dest, flow

    end subroutine
end module
```

Network Input Format

Number of nodes   Number of edges
Node1   Node2   Cost   Capacity
.
.
NetworkInput.dat
6 7
1 2 1 5
1 3 5 30
3 4 1 10
4 2 5 30
5 3 1 30
5 6 5 30
4 6 1 30

NetworkAMPLInterface Module(File : NetworkAMPLInterface.f90)

```
module NetworkAMPLInterfaceMod

     use GraphMod
    use NetworkFlowMod

    implicit none


    contains

!   ......... subroutine to print a data file which is to be used as
    input to AMPL/NEOS
    subroutine printAMPLDataFile( filename, N )

        implicit none

        type( Network ), intent( in )   :: N
        character( len = * ), intent( in ) :: filename
        integer                         ::
            ios,i,j,numcommodities,shapeArray( 2 )
            integer, allocatable            ::
                FlowConservationConstraintMatrix( :, : )

            FlowConservationConstraintMatrix = N%G

            shapeArray = shape( N%VertexFlow )
            numcommodities = shapeArray( 2 )
        open( unit = 10, file = filename, status = 'new', iostat = ios )

        write ( 10, * ) "param n := ",N%G%n,";"
        write ( 10, * ) "param m := ",N%G%m,";"
        write ( 10, * ) "param numcommodities := ",numcommodities,";"

        write ( 10, * ) "param: ","capacity ","cost ",":="
        do i = 1, N%G%m
            write( 10, * ) i, N%CapacityVector(i), N%CostVector(i)
        end do
        write( 10, * ) ";"

        write ( 10, * ) "param: ","netvertexflow ",":="
        do i = 1, N%G%n
            do j = 1, numcommodities
            write(10 , * ) " ",i," ",j," ",N%VertexFlow(i,j)
            end do
        end do
        write( 10 , * ) ";"

        write( 10, * ) "param: ","incmat"," :="
        do i = 1, N%G%n
            do j = 1, N%G%m
                write( 10, * ) i," ",j,"
                     ",FlowConservationConstraintMatrix(i,j)
            end do
        end do
        write( 10, * ) ";"

        close( 10 )

        print *, "Created Data File for AMPL: AMPLInput.dat"

            deallocate( FlowConservationConstraintMatrix )

    end subroutine


end module
```

NetworkTrafficModelInterface Module (File: NetworkTrafficModelInterface.f90)

```
module NetworkTrafficModelInterfaceMod

    use map_module ! system-level commands were commented out in
        subroutine write_input_file belonging to map_module
    use affinity_module
    use NetworkFlowMod

    implicit none
```

```
!   ......... currently the cost and capacity for every node is a fixed
    arbitrary number
     integer, parameter :: time_interval = 30, coverage_dist = 5000,
         intermediate_nodes = 20
     real, parameter  :: cost = 50.0, capacity = 10000000.0
     real, parameter  :: lat_upper_limit = 38.0, lat_lower_limit = 8.0,
         lon_lower_limit = 68.0, lon_upper_limit = 97.0


     contains

!   ......... subroutine to initialize a network from Math-Model Input
    file
     subroutine createNetwork(N,filename)

          implicit none

          type( Network ), intent( out )  :: N
          character( len = * ), intent( in ) :: filename
          integer                         :: num_cities(1)

          N%G%isdirected = .true.

          call read_data( filename ) !function definintion in
              map_module.f90

          num_cities = shape( population )
          N%G%n = num_cities( 1 )

          call createEdges( N )
          call fillVertexFlow( N )
          call fillCostCapacity( N )

     end subroutine

!   ......... subroutine to create edges between any cities that are
    within the coverage_dist
     subroutine createEdges(N)

          implicit none

          type( Network ), intent( inout )     :: N
          integer                              :: i, j, ctr
          real, allocatable,dimension( :, : )  :: dist

!         ..........the actual cities are numbered from 1 to n while the
    intermediate nodes are appended to the list
          N%G%n = N%G%n + intermediate_nodes

          allocate(dist(N%G%n,N%G%n))

          ctr = 0
          do i = 1, N%G%n
               do j = i+1, N%G%n
```

```
                     dist(i,j) = distance_intermediate( i, j, N )
                     if ( dist(i,j) < coverage_dist ) then
                           ctr = ctr +1
                     end if
               end do
          end do

          N%G%m = ctr

          allocate( N%G%edges( N%G%m, 2 ) )

          ctr = 1
          do i = 1, N%G%n
                do j = i+1, N%G%n
                     if ( dist(i,j) < coverage_dist ) then
                           N%G%edges( ctr, 1 ) = i
                           N%G%edges( ctr, 2 ) = j
                           ctr = ctr + 1
                     end if
               end do
          end do

           deallocate( dist )

     end subroutine

!   ......... subroutine to populate the Network with the total amount
    of flow through each node
     subroutine fillVertexFlow(N)

           implicit none

           type( Network ), intent( inout ) :: N
           integer                        :: i, j, k, actual_cities

           actual_cities = N%G%n - intermediate_nodes

           allocate( N%VertexFlow( N%G%n, actual_cities*( actual_cities
               - 1 )/2 ) )

           N%VertexFlow = 0

           k = 1
           do i = 1, actual_cities
                do j = i+1, actual_cities
                     N%VertexFlow( i, k ) = -ncalls( i, j,
                          time_interval )
                     N%VertexFlow( j, k ) = ncalls( i, j, time_interval
                          )
                     k = k + 1
                end do
           end do

     end subroutine
```

```
!   ......... subroutine to fill the Network with the cost and capacity
    of each edge
     subroutine fillCostCapacity(N)

           implicit none

           type( Network ), intent( inout ) :: N
           integer                       :: i

           allocate( N%CostVector( N%G%m ) )
           allocate( N%CapacityVector( N%G%m ) )

           do i = 1, N%G%m
                N%CostVector( i ) = cost
                N%CapacityVector( i ) = capacity
           end do

     end subroutine

!   ........ function to return distance between two cities
     real function distance_intermediate( city1, city2, N ) result (
          city1_city2_distance )

           implicit none

           integer,intent ( in )      :: city1, city2
           type( Network ), intent( in ) :: N
        real                       :: deglat1, deglon1, deglat2, deglon2
        real                       :: a, c, dlat, dlon, lat1, lat2
        integer                               :: actual_cities

           actual_cities = N%G%n - intermediate_nodes

           if( city1 > actual_cities ) then
                deglat1 = rand()*( lat_upper_limit - lat_lower_limit )
                    + lat_lower_limit
                deglon1 = rand()*( lon_upper_limit - lon_lower_limit )
                    + lon_lower_limit
           else
                deglat1 = getLatitude ( city1 )
             deglon1 = getLongitude ( city1 )
         end if

         if( city2 > actual_cities ) then
                deglat2 = rand()*( lat_upper_limit - lat_lower_limit )
                    + lat_lower_limit
                deglon2 = rand()*( lon_upper_limit - lon_lower_limit )
                    + lon_lower_limit
           else
                deglat2 = getLatitude ( city2 )
             deglon2 = getLongitude ( city2 )
         end if
```

```
           dlat = to_radian ( deglat2 - deglat1 )
        dlon = to_radian ( deglon2 - deglon1 )
           lat1 = to_radian ( deglat1 )
           lat2 = to_radian ( deglat2 )

           a =  ( sin ( dlat/2 ) ) ** 2 + cos ( lat1 ) * cos ( lat2 ) *
               ( sin ( dlon/2 ) ) ** 2
        c = 2 * asin ( sqrt ( a ) )
        city1_city2_distance = radius * c

     end function


end module
```

WeightedVertexNetworkFlow Module (File : WeightedVertexNetworkFlowMod.f90)

```
module WeightedVertexNetworkFlowMod

     use NetworkFlowMod
     use NetworkUserInterfaceMod

     implicit none

!   ........ overloading the assignment operator
     interface assignment ( = )
          module procedure splitNetwork
     end interface

!   ......... derived type to hold the Weighted Vertex Network
     type WeightedVertexNetwork
          type( Network )                    :: N
          integer, allocatable, dimension( : ) :: VertexWeights
     end type

     contains

!   ......... subroutine to convert a Weighted Vertex Network to a
    regular Network
   subroutine splitNetwork(N,W)

        implicit none

        type( Network ), intent( out )          :: N
        type( WeightedVertexNetwork ), intent( in ) :: W
        integer                                 :: i, j

        ! node x mapped to nodes 2*x-1 and 2*x

        N%G%n = 2*W%N%G%n
        N%G%m = W%N%G%m + W%N%G%n
        N%G%isdirected = .true.

        allocate( N%G%edges( W%N%G%n+W%N%G%m, 2 ) )
        allocate( N%CostVector( W%N%G%m+W%N%G%n ) )
       allocate( N%CapacityVector( W%N%G%m + W%N%G%n ) )

       do i = 1, W%N%G%m
          N%G%edges( i, 1 ) = 2*W%N%G%edges( i, 1 )
          N%G%edges( i, 2 ) = 2*W%N%G%edges( i, 2 ) - 1
          N%CostVector( i ) = W%N%CostVector( i )
          N%CapacityVector( i ) = W%N%CapacityVector( i )
       end do

       j = 1

       do i = W%N%G%m+1, W%N%G%m+W%N%G%n
          N%G%edges( i, 1 ) = 2*j - 1
          N%G%edges( i, 2 ) = 2*j
          N%CostVector( i ) = W%VertexWeights( j )
          N%CapacityVector( i ) = huge( N%CapacityVector( i ) )
          j = j + 1
       end do
   end subroutine

!   ........subroutine to set flow data in Network N which has been
    obtained by splitting a weighted vertex network
   subroutine setWeightedNetworkSourceDestinationFlow(N,A)

       implicit none

       type( Network ), intent( inout )   :: N
       real, intent( in ), dimension( :, : ) :: A
       integer                            ::
            src,dest,ios,i,numcommodities,shapeArray(2)
       real                               :: flow

       shapeArray = shape( A )
       numcommodities = shapeArray( 2 )

       allocate( N%VertexFlow( N%G%n, numcommodities ) )
       allocate( N%FlowVector( N%G%m+N%G%n, numcommodities ) )

       N%VertexFlow = A

   end subroutine

end module
```

WeightedVertexNetworkUserInterface (File : WeightedVertexNetworkUserInterface.f90)

```
module WeightedVertexNetworkUserInterfaceMod

     use WeightedVertexNetworkFlowMod


     contains

!   ........ subroutine to load data in a file into a
    WeightedVertexNetwork
     subroutine readWeightedVertexNetworkData( W, filename )

          implicit none

          character( len = * ), intent( in )       :: filename
          type( WeightedVertexNetwork ), intent (out ) :: W
       integer                                  :: i,ios

       open(unit = 10, file = filename, status = 'old', iostat = ios)

       if(ios.ne.0) then
          print *, "Error in Opening File in
              WeightedVertexNetworkUserInterface::readWeightedVertexNetworkData"
          stop
       endif
       read( 10, * ) W%N%G%n, W%N%G%m
       W%N%G%isdirected = .true.

       allocate( W%N%G%edges( W%N%G%m, 2 ) )
       allocate( W%N%CostVector( W%N%G%m ) )
       allocate( W%N%CapacityVector( W%N%G%m ) )
          allocate( W%VertexWeights( W%N%G%n ) )

       do i = 1, W%N%G%m
          read( 10, * ) W%N%G%edges( i, 1 ), W%N%G%edges( i, 2 ),
              W%N%CostVector( i ), W%N%CapacityVector( i )
       end do

       do i = 1, W%N%G%n
          read( 10, *) W%VertexWeights( i )
       end do

        close( 10 )

    end subroutine

!    ..........subroutine to load data regarding flow into an array A
    subroutine readWeightedNetworkSourceDestinationFlow(A,filename)

       implicit none

       real, intent( out ), allocatable, dimension( :, : ) :: A
       character( len = * ), intent( in )             ::filename
```

```
        integer
            ::src,dest,ios,i,numcommodities,n
        real                                           ::flow

        open( unit = 10, file = filename, status='old', iostat=ios)

        if(ios .ne. 0) then
            print *,"Error in opening File in
                WeightedVertexNetworkUserInterface:readWVNSourceDestinationFlow()
                "
            stop
        end if

        read( 10, * ) n, numcommodities

        allocate( A( 2*n, numcommodities ) )

        A = 0

        do i = 1, numcommodities
            read( 10, * ) src, dest, flow
            A( 2*src-1 , i ) = -flow
            A( 2*dest , i ) = flow
        end do

    end subroutine

end module
```

WeightedVertexNetwork Input Format

Number of nodes   Number of edges

Node1   Node2   Cost   Capacity

.

.

Node1_weight

Node2_weight

.

.

WVNetworkInput.dat

2 1

2 1 10 100

5

5

Flow Input Format

Number of commodities
source destination flow
.
.
FlowInput.dat
2
2 1 10
6 5 20

Test Files

Graph Test Program

```
program testgraph

  use GraphMod
  use GraphUserInterfaceMod

  implicit none

  type(Graph)                           :: G
  integer                               :: i,j
  integer,allocatable,dimension( :, : ) :: AdjMat
  integer,allocatable,dimension( :, : ) :: IncMat, IncMat1

  print *, "Program to test GraphMod.f90 and GraphUserInterface.f90 "

! ....... read user data
  call readGraphData( G, "../input/GraphInput.dat" )

   print *, " Printing Adjacency Matrix "

  call adjacencyMatrix( AdjMat, G )
  call printMatrix( AdjMat )

   print *, " Printing Incidence Matrix in two different ways "

   print *, " Method 1 "
  IncMat = G
  call printMatrix( IncMat )

  print *, " Method 2 "
  call incidenceMatrix( IncMat1, G )
  call printMatrix( Incmat1 )

   print *, "Graph Modules tested sucessfully "
   deallocate( G%edges )
```

```
end program testgraph
```

Network Test Program

```
program test1

    use NetworkFlowMod
    use NetworkUserInterfaceMod
     use NetworkAMPLInterfaceMod

    implicit none

    type(Network)    :: N, B
    integer          :: src, dest, numcommodities, i
    real             :: flow
    real,allocatable :: A( :, : )

    print *, "Program to test NetworkFlowMod.f90,
        NetworkUserInterfaceMod.f90 and NetworkUserInterfaceMod.f90"
    print *, " "

!   ...... read input data
    call readNetworkData( N, "../input/NetworkInput.dat" )

    print *, "Number of Vertices := ", N%G%n
    print *, "Number of Edges := ", N%G%m
     print *, " "

     print *, "   Edge No.    Vertex 1   Vertex 2   Cost    Capacity"
     do i = 1, N%G%m
           print *, i, N%G%edges(i,1), N%G%edges(i,2), N%CostVector(i),
               N%CapacityVector(i)
     end do
     print *, " "

     print *, "Reading data regarding the sources, sinks and amount of
          tele-traffic data from FlowInput.dat"
!   ..... load user-given Flow data into array A
    call readSourceDestinationFlowData(A,"../input/FlowInput.dat")
!   ...... use array A to set N%VertexFlow
    call setSourceDestinationFlow(N,A)
    print *, " "

     print *, "Calling subroutine to create input file for AMPL
          software"
    call printAMPLDataFile("../output/AMPLInput.dat",N)
    print *, " "

    print *, "Calling subroutine to convert the created network into a
        bidirectional Network"
    call bidirectional( B, N)
    print *, " "
    print *, "Number of Vertices := ", B%G%n
    print *, "Number of Edges := ", B%G%m
      print *, " "

      print *, "    Edge No.    Vertex 1   Vertex 2   Cost   Capacity"
      do i = 1, B%G%m
           print *, i, B%G%edges(i,1), B%G%edges(i,2), B%CostVector(i),
               B%CapacityVector(i)
      end do
      print *, " "

    deallocate(N%G%edges)
    deallocate(N%FlowVector)
    deallocate(N%CapacityVector)
    deallocate(N%CostVector)
    deallocate(N%VertexFlow)

    deallocate(B%G%edges)
    deallocate(B%CapacityVector)
    deallocate(B%CostVector)
    deallocate(B%VertexFlow)

end program test1
```

TrafficModel Test Program

```
program testtraffic

      use NetworkFlowMod
      use NetworkTrafficModelInterfaceMod
      use NetworkAMPLInterfaceMod


      implicit none

      type( Network ) :: N, B
      integer        :: i

      print *,"Program to test NetworkTrafficModelInterfaceMod"
      print *," "

      print *,"creating a network from file inputs.txt"
      call createNetwork(N,'../input/inputs-small.txt')

      print *," Network Description "
      print *, "Number of Vertices := ", N%G%n
    print *, "Number of Edges := ", N%G%m
      print *, " "

      print *,"Number of intermediate nodes inserted
          :=",intermediate_nodes
```

```
     print *," "

     print *, "   Edge No.    Vertex 1   Vertex 2   Cost   Capacity"
     do i = 1, N%G%m
          print *, i, N%G%edges(i,1), N%G%edges(i,2), N%CostVector(i),
              N%CapacityVector(i)
     end do
     print *, " "

     print *, "Calling subroutine to convert the created network into a
         bidirectional Network"
     call bidirectional(B, N)
     print *, " "

   print *, "Number of Vertices := ", B%G%n
   print *, "Number of Edges := ", B%G%m
     print *, " "

     print *, "    Edge No.    Vertex 1   Vertex 2   Cost   Capacity"
     do i = 1, B%G%m
          print *, i, B%G%edges(i,1), B%G%edges(i,2), B%CostVector(i),
              B%CapacityVector(i)
     end do
     print *, " "

     print *, "Calling subroutine to create input file for AMPL
         software"
     call printAMPLDataFile('../output/AMPLInput.dat',B)

end program
```

Weighted Vertex Network Test Program

```
program testweightedvertexnetworkflow

     use WeightedVertexNetworkFlowMod
     use WeightedVertexNetworkUserInterfaceMod
     use NetworkAMPLInterfaceMod

     implicit none

     type( WeightedVertexNetwork )   :: W
     type( Network )                 :: N
     integer                         :: i, numcommodities
     real,allocatable,dimension( :, : ) :: A

     print *, "Program to test WeightedVertexNetworkFlow Modules"
     print *, " "

     print *, " calling subroutine to read Weighted Vertex Network
         Input data "
     call readWeightedVertexNetworkData( W, '../input/WVNInput.dat' )
     print *, " "

     print *, "using overloaded assignment operator to convert a
         weighted vertex network into a regular network"
     N =  W

     print *,"Weighted Network Vertices:=",W%N%G%n
     print *,"Network Vertices:= ",N%G%n
     print *,"Weighted Network Edges:=",W%N%G%m
     print *,"Network Edges:=",N%G%m
     print *, " "

     print *, "Description of weighted vertex network"
     print *, "     Edge No.    Vertex 1   Vertex 2   Cost    Capacity"
     do i = 1,W%N%G%m
          print
              *,i,W%N%G%edges(i,1),W%N%G%edges(i,2),W%N%CostVector(i),W%N%CapacityVector(i)
     end do
     print*," "

     print *, "Description of new network"
     print *, "     Edge No.    Vertex 1   Vertex 2   Cost    Capacity"
     do i = 1,N%G%m
          print
              *,i,N%G%edges(i,1),N%G%edges(i,2),N%CostVector(i),N%CapacityVector(i)
     end do
     print*," "

     print *, "Reading data regarding the sources, sinks and amount of
         tele-traffic data from FlowInput.dat"
!   ..... read user-given Flow input data into array A
     call
         readWeightedNetworkSourceDestinationFlow(A,'../input/FlowInput.dat')
!   ...... use array A to set N%VertexFlow() in a Network which has been
    converted from a weighted vertex network
     call setWeightedNetworkSourceDestinationFlow(N,A)

     print *, "Calling subroutine to create input file for AMPL
         software"
     call printAMPLDataFile('../output/AMPLInput.dat',N)

end program
```

AMPL Input File

AMPL Model File (File : ampl-networkflow.mod)

```
param m >0;
param n >0;
param numcommodities >0;
```

```
set edges := {1..m};
set flow_at_nodes := {1..n};
set commodities := {1..numcommodities};

param netvertexflow{flow_at_nodes,commodities} ;
param capacity {edges}>0;
param cost{edges}>0;
param incmat{flow_at_nodes,edges};

var flow {e in edges,c in commodities} >= 0 ;

var totalflow {e in edges} = sum{c in commodities} flow[e,c];

minimize total_cost: sum{e in edges} cost[e]*totalflow[e];

subject to flowconservationconstraint{f in flow_at_nodes,c in
    commodities}:
   sum{e in edges} incmat[f,e]*flow[e,c] == netvertexflow[f,c];

subject to capacityconstraint{e in edges}:
    -capacity[e] <= totalflow[e] <= capacity[e];
```