```
########################################
#
# Model for minimum Steiner tree problem
# Using directed extended flow formulation
#
# Include parser file before this file as
#       ziml SNDParser.zpl thismodel.zpl
#
########################################


########################################
#
# Constructing helpful sets
#
########################################

##########
# set of all destination nodes
##########
set  R := T without S1;

########################################
#
# Constructing model
#
########################################

##########
# edge selection variables
##########
var x[E]   binary;

##########
# flow variables
##########
var f[R*A] >= 0;

##########
# objective
##########
minimize cost: sum<v,w> in E: l(v,w) * x[v,w];

##########
# flow conservation for each flow
##########
subto flow:
  forall <t> in R:
     forall <v> in V without S1:
          sum <w,v> in A: f[t,w,v]
        - sum <v,w> in A: f[t,v,w] ==
        if v==t then 1 else 0 end;

##########
# linking flow and edge selection variables
##########
subto vub:
  forall <t1> in R:
     forall <t2> in R:
        forall <v,w> in E:
           x[v,w] >= f[t1,v,w] + f[t2,w,v];
```

```
########################################
#
# Model for minimum survivable network problem
# Using extended flow formulation with two roots
#
# Include parser file before this file as
#       ziml SNDParser.zpl thismodel.zpl
#
########################################


########################################
#
# Constructing sets and functions for
# directed flow formulation ...
#
########################################

##########
# set of commodities
##########
set L := V without S1;
set H := T without S1;

########################################
#
# Constructing model
#
########################################

##########
# edge selection variables
##########
var x[E]   binary;

var y1[A];
var y2[A];

##########
# flow variables
##########
var f1[L*A] >= 0 <= 1;
var f2[H*A] >= 0 <= 1;

##########
# objective
##########
minimize cost: sum<v,w> in E: l(v,w) * x[v,w];

####################
# first tree
####################

##########
# flow conservation for each flow
##########
subto flow1:
  forall <t> in L:
     forall <v> in V without S1:
          sum <w,v> in A: f1[t,w,v]
        - sum <v,w> in A: f1[t,v,w] ==
         if (v == t) then 1 else 0 end;


##########
# linking flow and tree variables
##########
subto vub1:
  forall <t1> in L:
```



```
     forall <v,w> in A:
            y1[v,w] >= f1[t1,v,w];

####################
# second tree
####################

##########
# flow conservation for each flow
##########
subto flow2:
  forall <t> in H:
     forall <v> in V without S1:
          sum <w,v> in A: f2[t,w,v]
        - sum <v,w> in A: f2[t,v,w] ==
         if (v == t) then 1 else 0 end;

##########
# linking flow and tree variables
##########
subto vub2:
  forall <t1> in H:
        forall <v,w> in A:
           y2[v,w] >= f2[t1,v,w];

####################
# disjointness of paths
####################
subto vuba:
  forall <i,j> in E:
        x[i,j] >= y1[i,j]+y1[j,i];
subto vubb:
  forall <i,j> in E:
        x[i,j] >= y2[i,j]+y2[j,i];

subto disjoint:
  forall <i,j> in E:
      forall <t> in H:
        f1[t,i,j] + f2[t,i,j] + f1[t,j,i] + f2[t,j,i] <= x[i,j];
```

```
########################################
#
# Model for capacitated network design problem
# Using disaggregated fractional multicommodity
# edge-flow formulation.
#
# Include parser file before this file as
#       ziml SNDParser.zpl thismodel.zpl
#
########################################

########################################
#
# Constructing model
#
########################################

##########
# total edge capacity variables
##########
var x[E] >= 0;

##########
# capacity module installation variables
##########
var y[E*K]  integer >= 0;

##########
# setup variables
##########
var z[E]  binary;

##########
# flow variables
##########
var f[C*A] >= 0 <= 1;

##########
# objective
##########
minimize cost: sum<v,w,k> in E*K: cost(v,w,k) * y[v,w,k] +
               sum<v,w> in E: setupcost(v,w) * z[v,w];

##########
# flow conservation for each flow
##########
subto flow:
  forall <s,t> in C:
     forall <v> in V without { <t> }:
          sum <w,v> in A: f[s,t,w,v]
        - sum <v,w> in A: f[s,t,v,w] ==
         if (v == s) then 1 else 0 end;

##########
# linking flow and capacity installation variables
##########
subto cap:
  forall <v,w> in E:
    sum <s,t> in C: demand(s,t) * (f[s,t,v,w] + f[s,t,w,v])
    <= sum <k> in K: cap(v,w,k) * y[v,w,k];

##########
# setup edge, whenever there is some flow across ...
##########
subto setup:
  forall <v,w> in E:
     forall <s,t> in C:
        f[s,t,v,w] + f[s,t,w,v] <= z[v,w];
```



```
##########
# expressing the total edge capacity
##########
subto dummy:
  forall <v,w> in E:
     x[v,w] == sum <k> in K: cap(v,w,k) * y[v,w,k];
```

```
########################################
#
# Model for capacitated network design problem
# Using aggregated fractional multicommodity edge-flow formulation.
#
# Include parser file before this file as
#       ziml SNDParser.zpl thismodel.zpl
#
########################################

param BIG_M := 10000;

########################################
#
# Constructing model
#
########################################

##########
# total edge capacity variables
##########
var x[E] >= 0;

##########
# capacity module installation variables
##########
var y[E*K]  integer >= 0;

##########
# setup variables
##########
var z[E]  binary;

##########
# flow variables
##########
var f[V*A] >= 0;

##########
# objective
##########
minimize cost: sum<v,w,k> in E*K: cost(v,w,k) * y[v,w,k] +
               sum<v,w> in E: setupcost(v,w) * z[v,w];

##########
# flow conservation for each flow
##########
subto flow:
  forall <s> in V:
     forall <v> in V without { <s> }:
          sum <w,v> in A: f[s,w,v]
        - sum <v,w> in A: f[s,v,w] ==
          if (v > s) then demand(s,v)
          else 0
          end;

##########
# linking flow and capacity installation variables
##########
subto cap:
  forall <v,w> in E:
    sum <s> in V: (f[s,v,w] + f[s,w,v])
    <= sum <k> in K: cap(v,w,k) * y[v,w,k];

##########
# expressing the edge setup
##########
subto setup:
  forall <v,w> in E:
```



```
     sum <k> in K: y[v,w,k] <= BIG_M * z[v,w];

##########
# expressing the total edge capacity
##########
subto dummy:
  forall <v,w> in E:
     x[v,w] == sum <k> in K: cap(v,w,k) * y[v,w,k];
```

```
########################################
#
# Model for capacitated network design problem with 1+1 protection
# against node failures.
# Using disaggregated multicommodity edge-flow formulation.
#
# Include parser file before this file as
#       ziml SNDParser.zpl thismodel.zpl
#
########################################

########################################
#
# Constructing model
#
########################################

##########
# total edge capacity variables
##########
var x[E] >= 0;

##########
# capacity module installation variables
##########
var y[E*K]  integer >= 0;

##########
# setup variables
##########
var z[E]  binary;

##########
# flow variables
##########
var f[C*A] binary;

##########
# objective
##########
minimize cost: sum<v,w,k> in E*K: cost(v,w,k) * y[v,w,k] +
               sum<v,w> in E: setupcost(v,w) * z[v,w];

##########
# flow conservation for each flow
##########
subto flow:
  forall <s,t> in C:
     forall <v> in V without { <t> }:
          sum <w,v> in A: f[s,t,w,v]
        - sum <v,w> in A: f[s,t,v,w] ==
         if (v == s) then 2 else 0 end;

##########
# diversification for each flow
##########
subto flowcap:
  forall <s,t> in C:
     forall <v> in V without { <s>,<t> }:
          sum <w,v> in A: f[s,t,w,v] <= 1;

##########
# linking flow and capacity installation variables
##########
subto cap:
  forall <v,w> in E:
    sum <s,t> in C: demand(s,t) * (f[s,t,v,w] + f[s,t,w,v])
    <= sum <k> in K: cap(v,w,k) * y[v,w,k];
```



```
##########
# setup edge, whenever there is some flow across ...
##########
subto setup:
  forall <v,w> in E:
     forall <s,t> in C:
        f[s,t,v,w] + f[s,t,w,v] <= z[v,w];

##########
# expressing the total edge capacity
##########
subto dummy:
  forall <v,w> in E:
     x[v,w] == sum <k> in K: cap(v,w,k) * y[v,w,k];
```

```
####################
#
# Parser for SND problems
#
# This parser reads the problems constructed from SNDlib
# instances (not in the origianl SNDlib file format!)
# into data structures for various network design problems.
#
# The parser constructs undirected and simple networks and commodities.
#
# Sets, parameters, and functions used internally in this parser
# have the prefix _parser_ and should not be used outside this parser.
#
# The following sets, parameters, and functions are constucted
# for use in the mathematical model:
#
# FOR ALL PROBLEMS:
# ----------------
# set V : set of all netwok nodes (strings)
#
# set E : set of undirected edges <u,v> with u and v in V
#
# set A : set of directed arcs <u,v> corresponding to E
#         (If <u,v> in E, then <u,v> and <v,u> are both in A.)
#
# FOR STEINER TREE AND SURVIVABLE NETWORK DESIGN:
# ----------------------------------------------
#
# set T : set of all terminal nodes, subset of V
#         In Steiner Tree problems, T is the set of terminals.
#         In Survivable Network Design problems, T is the set of nodes
#         requiring edge- or node-connectivity  2, all other nodes (V-T)
#         require connectivity 1.
#
# set S1: set containing one arbitrary node <s1> from T
# set S2: set containing another arbitrary node <s2> != <s1> from T
#
# function l(u,v) :       length (setup cost) of edge <u,v> in E
#
# FOR CAPACITATED NETWORK DESIGN:
# ------------------------------
#
# set C : set of undirected commodities <s,t> with s and t in V
#
# set K : set of edge capacity units
#
# function demand(s,t) :    bandwidth demand of commodity <s,t> in C
#
# function l(u,v) :         setup cost for installing edge <u,v> in E at all
#
# function cost(u,v,k) :    cost for installing one unit <k> in K
#                           on edge <u,v> in E
#
# function cap(u,v,k) :     bandwidth provided by installing one unit <k> in K
#                           on edge <u,v> in E
#
####################

# param file := "atlanta.txt";

##########
# read and construct different node sets
##########
set _parser_N := { read file as "<2s>" match "^N "};
set _parser_T := { read file as "<2s>" match "^T "};

set V := _parser_N union _parser_T;
set T := _parser_T;
```



```
##########
# set of capacity indices
##########
set K := {1,2};


##########
# read and construct simple, undirected edge set
##########
set _parser_E := { read file as "<4s,5s>" match "^L "};

set E := {<u,v> in V*V with (u<v) and (<u,v> in _parser_E or <v,u> in _parser_E)
};

##########
# set of directed arcs
##########
set  A := { <v,w> in V*V with <v,w> in E or <w,v> in E };

##########
# read setup cost and first two (capacity,cost) pairs of each edge
##########
param _parser_setup[V*V] := read file as "<4s,5s> 10n" match "^L " default 0;
param _parser_cap1[V*V]  := read file as "<4s,5s> 12n" match "^L " default 0;
param _parser_cost1[V*V] := read file as "<4s,5s> 13n" match "^L " default 0;
param _parser_cap2[V*V]  := read file as "<4s,5s> 14n" match "^L " default 0;
param _parser_cost2[V*V] := read file as "<4s,5s> 15n" match "^L " default 0;

defnumb _parser_func_cap(i,j,k)  := if ( k==1 ) then _parser_cap1[i,j]
                                    else _parser_cap2[i,j]
                                    end;

defnumb _parser_func_cost(i,j,k) := if ( k==1 ) then _parser_cost1[i,j]
                                    else _parser_cost2[i,j]
                                    end;

defnumb cap(i,j,k) := if ( <i,j> in _parser_E ) then _parser_func_cap(i,j,k)
                      else _parser_func_cap(j,i,k)
                      end;
defnumb cost(i,j,k) := if ( <i,j> in _parser_E ) then _parser_func_cost(i,j,k)
                       else _parser_func_cost(j,i,k)
                       end;
defnumb setupcost(i,j) := if ( <i,j> in _parser_E ) then
                                _parser_setup[i,j] + _parser_func_cost(i,j,1)
                          else
                                _parser_setup[j,i] + _parser_func_cost(j,i,1)
                          end;

##########
# read and construct undirected, simple commodity set
##########
set _parser_C := { read file as "<4s,5s>" match "^D "};

set C := { <s,t> in V*V with (s<t) and (<s,t> in _parser_C or <t,s> in _parser_C
 ) };


##########
# read and construct demand values
##########

param _parser_dem[V*V] := read file as "<4s,5s> 8n" match "^D " default 0;
param _parser_sumdem[<v> in V] := sum <w> in V without {<v>}: (_parser_dem[v,w]+
_parser_dem[w,v]);
param sumdem := sum <v> in V: _parser_sumdem[v];

defnumb demand(i,j) := _parser_dem[i,j] + _parser_dem[j,i];
```

```
##########
# define edge length
##########
defnumb l(i,j) := setupcost(i,j);



##########
# helpful sets for Steiner Tree and SND:
##########

set S1 := argmin <v> in T: length(v);
set S2 := argmin <v> in (T without S1): length(v) ;
```