Discrete Optimization Specialization: Workshop 12
Warning the Village

1 Introduction

As Wukong is fanning the flames of the fire mountains, a village in danger of being burned by the flames being fanned towards it. Nezha wants to send a message to all the villagers as soon as possible, to stand by their houses prepared to stop the fire consuming them. He has m magic birds that can instantly arrive at one house to give a complex message, but there are $n > m$ houses to warn. He decides to send them to m houses ordering the house occupant to act as a *messenger*, i.e. run and deliver the message to other houses close by, and then return to their own house. He needs to pick the houses to message and the route of each of the messengers, so that they arrive back at their home with everyone having received the message in the least possible time.

Warning the Village — `village.mzn`

The data for the problem is encoded as:

- n—the number of houses, each of which has a coordinate x and y, and
- m—the number of messengers Nezha can use (which is also equal to the number of magic birds).

Finally there is a time limit *limit* by which everyone must be warned.

The coordinates, $(x[i], y[i])$, of each house i are given as data. The time it takes a messenger to run from house i to house j is $|x[i] - x[j]| + |y[i] - y[j]|$, which is the Manhattan distance between the houses.

An example data file is

```
n = 10;
x = [0, 3, 4, 6, 2, 8, 4, 3, 1, 6];
y = [7, 3, 6, 2, 0, 5, 2, 4, 1, 9];
m = 3;
limit = 40;
```

Create a model for the Village Warning problem. This is a routing problem. We will need to consider (a) from each house, which house to visit next, and (b) adding dummy houses (one for each messenger) representing the return back to their own home. Now we have a graph with nodes, which include both the houses and dummy houses. The key decision variables are (a) for each node, which node to visit next, and (b) who the messengers are. A starting point for the model is:

```
% Warning the village
int: n; % number of houses
set of int: HOUSE = 1..n;
array[HOUSE] of int: x;
```

```
array[HOUSE] of int: y;
int: m; % number of messengers
int: limit; % max time allowed
set of int: MESSENGER = 1..m;
array[MESSENGER] of var HOUSE: messenger;
% nodes
% 1..n are houses
% n+1..n+m are end trip node
set of int: NODE = 1..n+m;
array[NODE] of var NODE: next;   % next node visited after this one
var 0..limit: endtime;           % when all visits are over
```

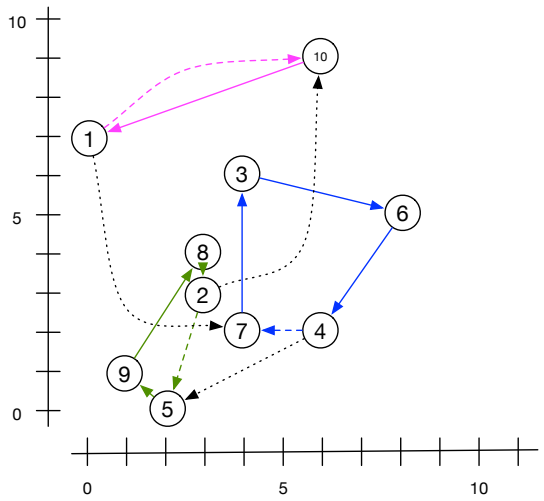
The usual trick to modeling this kind of routing problems is to have the next node of each of the dummy houses pointing at the next messenger's house, so that altogether the `next` relation models a single cycle through all nodes. Tricky parts of the model are tracking how far each messenger runs, in a loop from their own house and back again.

Your output function should return the `next` and `messenger` arrays, as well as the time when everyone is warned and home. For example, using the data above, your model might find the following solution.

```
messenger = [7, 5, 10];
next = [11, 13, 6, 12, 9, 4, 3, 2, 8, 1, 5, 10, 7];
endtime = 16;
```

Note that messenger 7 takes 16 minutes to visit houses 3, 6, 4, and back to 7, while messenger 5 takes 12 minutes to visit houses 9, 8, 2, and back to 5, and messenger 10 takes 16 minutes to visit house 1 and return to 10.

The solution is graphically illustrated by the diagram below:



The dashed arcs represent the last leg of each messenger returning to home. The dotted arcs represent the `next` relation.

Once you have created a model you need to find a suitable search strategy. Note that this is a challenging problem, and we are unlikely to prove optimality of solutions, even for the very small data file above.

You will need to think about

- what variables to search on, perhaps you should search on some intermediates you introduced,
- what the best variable and value selections are,
- what restarting strategy you should use, and
- how you can use large neighbourhood search to improve the results.

Use the `-s` statistics flag (or check box in the IDE) to compare how much search each search strategy uses. Use the command line `--time-limit` *X* (or the control in the solver configuration in the IDE) to limit the execution time to *X* milliseconds.

Note you can make use of large neighbourhood search in Gecode using the `relax_and_reconstruct` annotation, which takes two arguments: a list of integer variables to search on, and an integer saying what percentage of the variables are fixed to their previous values. All other variables become free, and the solver will determine their values. You will need to add the line

```
rnclude "gecode.mzn"
```

to your model to use it.

Given the problems here are difficult the most important thing to judge effectiveness is

- *robustness*: how good a solution it finds in some fixed time.

What do you think is the best search strategy overall? Justify your answer.

2 Technical Requirements

For completing the workshop you will need MiniZinc 2.2.x (`http://www.minizinc.org/software.html`).