FIT5216: Modelling Discrete Optimization Problems
Assignment 3: Maintaining Drones

1 Overview

For this assignment, your task is to write a MiniZinc model for a given problem specification.

- Submit your work to the MiniZinc auto grading system (using the submit button in the MiniZinc IDE). You must submit via the IDE to be graded and receive marks.
- Submit your model (copy and paste the contents of the .mzn file) and report using the Moodle assignment.

You have to submit by the due date (Sunday 29th May 2022, 11:55pm), using MiniZinc and using the Moodle assignment, to receive full marks. You can submit as often as you want before the due date. Late submissions without special consideration receive a penalty of 10% per day. Submissions are not accepted more than 7 days after the original deadline.

This is an **individual assignment**. Your submission has to be **entirely your own work**. We will use similarity detection software to detect any attempt at collusion, and the **penalties are quite harsh**. If in doubt, contact your teaching team with any questions!

2 Problem Statement

You are setting up a modern drones logistic system. Orders come in from customers, and their package is delivered by a drone which then returns to the warehouse to pick up the next order. Before it flies out it needs to be serviced and recharged.

The visit of a drone back to the warehouse involves a series of possible actions

```
enum ACTION = { LAND, INSPECT, FULLSERVICE, RECHARGE, PACK, TAKEOFF };
```

The warehouse has a number of resources that perform these services: e.g. landing pads and charging stations

```
array[ACTION] of int: resources; % number of resources for each service
```

The aim is to plan the actions at the warehouse for each drone, over some time horizon defined by

```
int: horizon;  % end time of plan
set of int: TIME = 0..horizon;
```

There are orders that need to be served by the drones, with given specifications

```
int: norders;                    % number of orders;
set of int: ORDER = 1..norders;
array[ORDER] of int:  dist;      % distance warehouse to delivery
enum WC = { ULTRA, LIGHT, MEDIUM, HEAVY };
array[ORDER] of WC:  weight;     % weight catgeory of order
array[ORDER] of TIME: available; % when the order can be packed
array[ORDER] of int:  value;     % value of order
```

There a number of drones arriving back at the warehouse, with characteristics defined by

```
int: ndrones;
set of int: DRONE = 1..ndrones;
array[DRONE] of TIME: arrival;  % when they arrive back
array[DRONE] of int: charge;    % how much charge they have left
```

The decisions to be made for each drone, are for each action:

- at what time does it start; and
- which of the identical resources for that task is it actually run on.

```
array[DRONE, ACTION] of var TIME: start;                % start time for action
array[DRONE, ACTION] of var 0..max(resources): resource; % resrouce used for action
```

Not every action will be required for each drone. For actions that dont occur there resource number should be 0, but their start time should still be in order. The critical decision is for each drone which order it is assigned

```
array[DRONE] of var 0..norders: order; % which order given to drone (0 for none)
```

The aim of the schedule is to maximize the value of the orders that have taken off, by the end of the time horizon. Interestingly in the warehouse drones are moved around by conveyor belt, which means the sequence of actions is fixed in the order given, and even if a drone does not require some action, it still moves via conveyor through one of the resources for that action (except FULLSERVICE actions which are managed by humans off the conveyer system).

You should approach the model in stages. Note you will be unable to gain full marks without dealing with each stage.

3 Stage A

For this stage we assume each drone goes through the actions in order: LAND, INSPECT, FULLSERVICE, RECHARGE, PACK, TAKEOFF, and each of these has unit duration, except FULLSRVICE which has 0 duration.

We simply need to ensure that

- The actions for each drone occur in the specified order after its arrival
- Each drone and each action is assigned to a resource that actually exists in the warehouse (if the action is used) or 0 otherwise.
- Each order is assigned to at most one drone.
- If the drone is given an order, then it must be assigned a resource for all actions, even if some of them may have 0 duration. If it has no order it only should be assigned a resource for LAND and INSPECT actions.
- Each drone doesnt land before its arrival time

- An order is not packed before it is available.

Given the small test data

```
resources = [2,1,1,2,2,2];
horizon = 10;
norders = 6;
dist = [100,30,20,15,20,35];
weight = [HEAVY,LIGHT,LIGHT,ULTRA,MEDIUM,LIGHT];
packtime = [1,1,1,3];
available = [0,0,3,5,7,5];
value = [2,5,3,5,10,8];
ndrones = 4;
arrival = [0,0,6,4];
charge  = [2,0,0,10];
```

Then a possible solution is given by

```
start =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    0,       1,           2,        2,    3,       4
 |    0,       1,           2,        2,    3,       4
 |    6,       7,           8,        8,    9,      10
 |    4,       5,           6,        6,    7,       8
 |];
resource =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    2,       1,           1,        2,    2,       2
 |    2,       1,           1,        2,    2,       2
 |    2,       1,           1,        2,    2,       2
 |    2,       1,           1,        2,    2,       2
 |];
order = [1, 2, 3, 4];
```

which is visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: LIRPT......    1
(2) Drone  2: LIRPT......    2
(3) Drone  3: ......LIRPT    3
(4) Drone  4: ....LIRPT..    4
```

which shows the action for each drone at each time, where the letter code gives the first letter of the ACTION. Zero duration actions do not show up in the visualization. Notice how the FULLSERVICE actions which dont actually occur are still scheduled directly after INSPECT and before RECHARGE actions.

4 Stage B

While the schedules for each job are valid, they ignore the fact that there are a limited number of resources. Drones 1 and 2 both use the single inspection resource at the same time. In this stage you need to add constraints to ensure that:

- There are never more actions of any type running than the number of resources for that task; and
- There is no overlap in actions assigned to the same resource.

With these constraints enforced a possible solution is

```
start =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    0,       2,           3,        3,    4,       5
 |    0,       1,           2,        2,    3,       4
 |    6,       7,           8,        8,    9,      10
 |    4,       5,           6,        6,    7,       8
 |];
resource =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    1,       1,           1,        1,    1,       2
 |    2,       1,           1,        1,    2,       2
 |    2,       1,           1,        2,    2,       2
 |    2,       1,           1,        1,    2,       2
 |];
order = [1, 2, 3, 4];
```

visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: L.IRPT.....    1
(2) Drone  2: LIRPT......    2
(3) Drone  3: ......LIRPT    3
(4) Drone  4: ....LIRPT..    4
Land Pad 1  : 1..........
Land Pad 2  : 2...4.3....
Inspect  1  : .21..4.3...
Full     1  : ..........
Charger  1  : ..21..4....
Charger  2  : ........3..
Packer   1  : ....1......
Packer   2  : ...2...4.3.
TakeoffP 1  : ..........
TakeoffP 2  : ....21..4.3
```

The second part shows the drone being serviced by each resource at each time. Note that drone 1 and 2 land simultaneously but use different landing pads. The extra constraint ensured that drone 1 is not inspected at the same time as drone 2.

Note that even if a drone has a 0 duration action, it cannot overlap with another action on the same resource. Consider the following fragment of a visualization

```
TIME           00000000001
               01234567890
(1) Drone  1: L.IPT.....    1
(2) Drone  2: LIRRRRPT..    2
```

where Drone 1 and Drone 2 are both assigned to Charger 1 for their RECHARGE action. The schedule is illegal since Drone 1 starts its 0 duration RECHARGE at time 3, in the middle of the RECHARGE action for drone 2 starting at time 2 ending at time 6 on the same Charger. This is because drone 1 still passes through Charger 1 by conveyor belt at time 3 in this schedule, which conflicts with drone 2.

5 Stage C

In reality the duration of certain actions depends on the current state of the drone and the next order. In this stage you need to add constraints to ensure that

- The packing time for the drone depends on the weight of the order it will carry, given by the data:

  ```
  array[WC] of int: packtime;
  ```

- The charging time depends on the current charge level and the distance for the order it will carry. If the charge level is 10 then it doesnt need recharging, if the charge level is 5 or above and the distance at most 50km then it doesnt need recharging, otherwise it needs recharging. There are choices on the charging method: fastcharging takes 1 time unit, while slow charging takes 4 time units. We need to record the fastcharge decisions for each drone.

  ```
  array[DRONE] of var bool: fastcharge;
  ```

- A full service is required if the charge level is zero, and the next order is HEAVY or MEDIUM (so there is a next order); or if the inspection shows some problems. A full service requires 10 time units. For now we assume the inspection always passes.

With `packtime = [1,1,1,3];` and these constraints added a possible solution is

```
start =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    3,       4,           6,        6,    7,      10
 |    0,       1,           2,        2,    8,       9
 |    6,       7,           8,        8,    9,      10
 |    4,       5,           6,        6,    6,       8
```

```
 |];
resource =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    2,       1,           1,        2,    2,       1
 |    2,       1,           1,        2,    1,       2
 |    2,       1,           1,        2,    1,       2
 |    2,       1,           1,        2,    2,       2
 |];
order = [1, 4, 2, 3];
fastcharge = [true, false, true, false];
```

This is visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: ...LI.RPPPT F  1
(2) Drone  2: LIRRRR..PT.    4
(3) Drone  3: ......LIRPT F  2
(4) Drone  4: ....LIP.T..    3
Land Pad 1  : ..........
Land Pad 2  : 2..14.3....
Inspect  1  : .2..14.3...
Full     1  : ..........
Charger  1  : ..........
Charger  2  : ..22221.3..
Packer   1  : ........23.
Packer   2  : ......4111.
TakeoffP 1  : ..........1
TakeoffP 2  : ........423
```

Notice how drone 1 requires 3 packing steps, since it assigned a heavy order. Drone 3 needs a fast charge to get away in time. Drone 4 doesnt need a recharge since its on full charge to begin with.

6 Stage D

The objective is to maximize throughput of valuable customer orders. You should now add an objective to your model. The aim is to maximize profit and minimize costs, but also to minimize the time take to serve them. Profit is given by the sum of the value of the orders that have taken off by the horizon time. The only cost we calculate is given by the number of fastcharges which each cost 5. The secondary objective is to minimize total turn around time, that is the total time spent before the drones take off with another order. The turnaround time for a drone is the take off time minus the arrival time. If a drone doesnt take an order its turn around time is 0. Once we have maximized profit we need to minimize the turnaround time spent to make the profit.

An optimal profit solution for the sample data is:

```
start =
```

```
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    1,       3,           4,        4,    8,       9
 |    0,       2,           3,        3,    7,       8
 |    6,       7,           8,        8,    9,      10
 |    4,       8,           9,        9,    9,      10
 |];
resource =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    2,       1,           1,        2,    1,       2
 |    2,       1,           1,        1,    1,       2
 |    1,       1,           1,        2,    1,       1
 |    2,       1,           1,        1,    2,       2
 |];
order = [2, 4, 6, 5];
fastcharge = [false, false, true, false];
```

which is visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: .L.IRRRRPT.    2
(2) Drone  2: L.IRRRRPT..    4
(3) Drone  3: ......LIRPT F  6
(4) Drone  4: ....L...IPT    5
Land Pad 1  : ......3....
Land Pad 2  : 21..4......
Inspect  1  : ..21...34..
Full     1  : ..........
Charger  1  : ...2222....
Charger  2  : ....11113..
Packer   1  : .......213.
Packer   2  : .........4.
TakeoffP 1  : ..........3
TakeoffP 2  : ........214
```

showing the order for each drone, and which have used a fastcharge. The total profit is 23: we earn 28 from the orders and pay 5 for one fastcharge. The turnaround time here is $9 + 8 + 4 + 6 = 27$.

But once we try to minimize turn around time as well we get optimal solution

```
start =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    0,       2,           3,        3,    7,       8
 |    0,       1,           2,        2,    6,       7
 |    6,       7,           8,        8,    8,       8
 |    4,       5,           6,        6,    6,       7
 |];
resource =
```

```
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    2,       1,           1,         2,    1,       1
 |    1,       1,           1,         1,    1,       2
 |    1,       1,           0,         0,    0,       0
 |    1,       1,           1,         1,    2,       1
 |];
order = [5, 2, 0, 6];
fastcharge = [false, false, false, false];
```

which is visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: L.IRRRRPT..    5
(2) Drone  2: LIRRRRPT...    2
(3) Drone  3: ......LI...    0
(4) Drone  4: ....LIPT...    6
Land Pad 1  : 2...4.3....
Land Pad 2  : 1..........
Inspect  1  : .21..4.3...
Full     1  : ..........
Charger  1  : ..2222.....
Charger  2  : ...1111....
Packer   1  : ......21...
Packer   2  : ......4....
TakeoffP 1  : .......41..
TakeoffP 2  : .......2...
```

Notice how we abandon order 4, and not use drone 3 which arrives late, to get the same profit of 23, but reduce the turnaround time to $8 + 7 + 3 = 18$.

7 Stage E

WARNING: this stage is substantially harder than the previous ones. You can still complete the later stages without completing this one.

The resources for each action are connected by conveyor belts which move the drones from one resource to another. While the landing and takeoff pads are connected to each resource by a separate conveyor, the inspection resources move themselves from one conveyor to another, and full service is by humans which take the drones off the conveyor system altogether, the conveyors from RECHARGE to PACK resources are restricted. There is only one conveyor to each PACK resource. It takes 5 time units to reconfigure it to connect to a different RECHARGE resource.

If a packing resource is assigned to a drone that is coming from a different recharge resource than the previous drone it served then the conveyor must be reconfigured. Note that the reconfiguration can start as soon as the previous drone arrives (i.e. the time step that we start packing the previous drone). Notice that drones that dont need a recharge still travel by a conveyor through the recharge station they are assigned, so the constraint applies even if they have a 0 duration recharge.

With this constraint an optimal solution is

```
start =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    1,       2,           3,        3,    7,       8
 |    0,       1,           2,        2,    6,       7
 |    6,       7,           8,        8,   10,      10
 |    4,       5,           6,        7,    7,       8
 |];
resource =
[| LAND: INSPECT: FULLSERVICE: RECHARGE: PACK: TAKEOFF:
 |    1,       1,           1,        2,    1,       2
 |    1,       1,           1,        1,    2,       2
 |    2,       1,           0,        0,    0,       0
 |    1,       1,           1,        1,    2,       1
 |];
order = [4, 6, 0, 5];
fastcharge = [false, false, false, false];
```

which is visualized as

```
TIME          00000000001
              01234567890
(1) Drone  1: .LIRRRRPT..    4
(2) Drone  2: LIRRRRPT...    6
(3) Drone  3: ......LI...    0
(4) Drone  4: ....LI.PT..    5
Land Pad 1  : 21..4......
Land Pad 2  : ......3....
Inspect  1  : .21..4.3...
Full     1  : ..........
Charger  1  : ..2222.....
Charger  2  : ...1111....
Packer   1  : .......1...
Packer   2  : ......24...
TakeoffP 1  : ........4..
TakeoffP 2  : .......21..
```

Note how previously Packer 1 packed drones 1 and 2 with drone 2 coming from Charger 1 and drone 1 coming from Charger 2. Now to avoid the changeover time, drone 1 and 2 are packed on separate machines

As another example consider a solution visualized as:

```
TIME          00000000001
              01234567890
(1) Drone  1: LIRPPPT....
(2) Drone  2: LIRRRRPT...
```

```
Charger 1   : ..1........
Charger 2   : ..2222.....
Packer 1    : ...1112....
```

which violates the constraint. We start packing drone 2 from Charger 2 at time 6. There is no time to reconfigure the conveyor since the previous drone 1 comes from Charger 1. If we reconfigure we need to delay the packing of drone 2 to time 8 since the previous drone starts packing at time 3. The corrected solution is visualized as:

```
TIME           00000000001
               01234567890
(1) Drone  1: LIRPPPT....
(2) Drone  2: LIRRRR..PT.
Charger 1   : ..1........
Charger 2   : ..2222.....
Packer 1    : ...111..2..
```

which violates the

8 Stage F

The problem as described may have a number of symmetries and dominances:

- symmetries that arise from the problem statement
- symmetries that arise from the modelling of the problem decisions
- symmetries that arise from particular data
- dominances that arise from the particular data

Write a report about the symmetries and dominances that exist in the problem, of each of the different kinds. Add symmetry and dominance breaking constraints to your model to help overcome these symmetries. Carefully explain how the symmetry and dominance breaking constraints you define are correct, and how they removes symmetries or dominated solutions, but always leaves an equivalent or better solution. You may want to try different approaches to breaking the same symmetry or dominance.

Explore the effectiveness of the symmetry and dominance breaking constraints you define on the problem instances given in the `data` directory. Give a table that illustrates the difference in solving time and/or the size of the search space (e.g. failures) with and without each symmetry/dominance breaking constraint, for each instance. Write a conclusion about the effectiveness of each of the symmetry/dominance breaking constraints you tried.

9 Stage G

The different valid data files are quite different in nature. For the final stage you should examine each data file, and consider which constraints have the most effect in restricting profit. This is

called *sensitivity analysis*. For each data file (except `drones0.dzn`) in the `data` subdirectory point out the *single* number appearing in the data file, where some change would most help increase the overall profit of the solutions for this data. One caveat you are not allowed to change the `value` data (that would be a bit too easy). For example for the sample data making order 5 available earlier might allow it to be filled. *Justify* why your selection is a good one for each data file. Explain your reasoning to arrive at your decision, with a paragraph for each file.

10 Instructions

Edit the provided `mzn` model files to solve the problems described above. You are provided with some sample data files to try your model on. Your implementations can be tested locally by using the *Run* icon in the MiniZinc IDE or by using,

```
minizinc ./modelname.mzn ./datafile.dzn
```

at the command line.

The report has to be submitted in PDF format. It does not have to follow any particular structure (e.g. you do not need an abstract, table of contents etc.). The explanations should be concise. The overall limit for the report is 4 pages, including all graphs, tables etc.

11 Marking

The auto-grading server is using the `chuffed` solver to grade this assignment. Grades are calculated based first on the correctness of the found solutions, and then on the quality of solution that is achieved. Your model needs to be able to find at least one solution for each instance within the 1 minute time-out in order to get any marks.

You can get a maximum of 40 marks for this assignment, which will be worth 25% of your overall unit marks.

The marks for the correctness of your model are automatically calculated. You will receive up to 9 marks for locally tested data and up to 16 marks for your model as tested on the auto-grading server. You will only get full marks if you implement all stages.

The code comments and your report will be marked by your tutors, and they are worth the remaining 15 marks, with the following allocation:

Stage A-E (code comments): 4 marks; Stage F (symmetry): 8 marks; Stage G (sensitivity): 3 marks;