The University of Melbourne
Department of Computing and Information Systems
Semester 2 Assessment 2015

COMP90046 Constraint Programming

Time Allowed: 3 Hours
Reading Time: 15 minutes

Authorized materials: Books and calculators are not permitted.

Instructions to Invigilators: One 21 page script. Exam paper may leave the room.

Instructions to students:

This exam counts for 70% of your final grade. There are 13 pages and 8 questions for a total of 70 marks. Attempt to answer all of the questions. Values are indicated for each question and subquestion — be careful to allocate your time according to the value of each question.

Question 1 [4 marks]

How many integers between 1 and 200 have digits that sum up to 9? The answer is 17 being {9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 108, 117, 126, 144, 153, 162, 171, 180}

We consider the generalization of this problem: how many integers between l and u sum up to s.

(a) Write a MiniZinc model that, given l, u and s, determines a number n which is in the range $l..u$ and whose digits sum to s.

You can assume that l, u and s are all positive integers. The data format should be

```
int: l;
int: u;
int: s;
```

For example the data for the problem above is

```
l = 1;
u = 200;
s = 9;
```

Ensure that any decision variables you define have tight upper and lower bounds. To help you, given a number k the number of digits d in k is given by

```
int: d = ceil( ln(k+1) / ln(10) );
```

e.g. for $k = 1000$, $d = 4$ and for $k = 999$, $d = 3$, and in MiniZinc exponentiation is written as $\texttt{pow}(x, y) = x^y$. The model should output n. [3 marks].

```
int:  l;
int:  u;
int:  s;
int:  d = ceil( ln(u+1) / ln(10) );
var l..u:  num;
array[1..d] of var 0..9:  digit;
constraint sum(digit) = s;
constraint sum(i in 1..d)(pow(10,d - i) * digit[i]) =
num;
solve satisfy;
```

Common error was to try to produce all answers, the question only asked for one possible n!

(b) Explain how you can use your model to determine the number of integers between l and u that add up to s. [1 mark].

Run the model using all solutions and count the number

Question 2 [8 marks]

In 1779 Euler proposed the following "36 Officer Problem": given 6 regiments each with 6 officers of 6 different ranks, is it possible to arrange the 36 officers in a square of 6 × 6, so that no row or column contains a duplicate rank or regiment. The mathematician Gastor proved it impossible in 1901.

We consider a generalization of the problem: given n regiments each with n officers of n different ranks, is it possible to arrange the n^2 officers in a $n \times n$ square so that no row or column contains a duplicate rank or regiment. It turns out for many n this is possible. Here is a solution for $n = 4$, shown as (regiment, rank) pairs. Note how no regiment or rank is duplicated in any row or column and each pair occurs exactly once.

```
(4,1) (3,2) (1,3) (2,4)
(2,2) (1,1) (3,4) (4,3)
(3,3) (4,4) (2,1) (1,2)
(1,4) (2,3) (4,2) (3,1)
```

(a) Write a MiniZinc model that given n determines a solution to the $n \times n$ officer problem, if possible. The data format is simply

```
int: n;
```

For example the data for the example above is

```
n = 4;
```

Ensure that any decision variables you define have tight upper and lower bounds. Note you do not have to provide an output item! [6 marks].

```
int:n;  % number of regiments and ranks
set of int: REG = 1..n;
set of int: RANK = 1..n;
set of int: OFF = 1..n*n;
set of int: ROW = 1..n;
set of int: COL = 1..n;
array[ROW,COL] of var OFF: x;
array[ROW,COL] of var REG: reg;
array[ROW,COL] of var RANK: rank;
constraint forall(r in ROW, c in COL) % relationship with vars
                (reg[r,c] = (x[r,c] - 1) div n + 1 /\
                 rank[r,c] = (x[r,c] - 1) mod n + 1 /\
                 x[r,c] = (reg[r,c] - 1) * n + rank[r,c]); % only need this line
include "alldifferent.mzn";
constraint alldifferent(r in ROW, c in COL)(x[r,c]);  % no pair reappears
constraint forall(r in ROW)
                (alldifferent(c in COL)(reg[r,c]) /\ % regiments and ranks
```

```
                    alldifferent(c in COL)(rank[r,c])); % different in columns
constraint forall(c in COL)
                   (alldifferent(r in ROW)(reg[r,c]) /\ % regiment and ranks
                    alldifferent(r in ROW)(rank[r,c])); % different in rows
solve :: int_search([x[r,c] | r in ROW, c in COL], dom_w_deg, indomain_min, complete)
      satisfy;
output [ "x    = \(x);\nreg  = \(reg);\nrank = \(rank);\n"];
```

Common mistakes: ignoring the row,rank pairs alldifferent constraint

(b) Since the regiments and ranks are interchangeable, their are significant symmetries in the problem. Illustrate a symmetry breaking constraint that could be added to your model to improve the solving behavior. Explain why it is correct! [2 marks].

Lots of possibilities: make the first row fixed to (1,1),(2,2), ... (n,n)

```
constraint forall(c in COL)(reg[1,c] = c /\ rank[1,c] = c);
```

Question 3 [7 marks]

In a steel mill there are a number of tasks to schedule on a single roller. Each task has a required duration in minutes. There are two kinds of tasks: "hot" tasks and "cold" tasks. Assume data is given in the form:

```
int: n;                              % number of tasks
set of int: TASK = 1..n;
array[TASK] of int: duration;        % duration of each task (mins)
set of TASK: cold;                   % which tasks are cold
set of TASK: hot = TASK diff cold; % which tasks are hot
int: makespan;                       % total rolling time available
```

The principal decision variables are

```
array[TASK] of var 0..makespan: start;  % start time of each task
var 0..makespan: endtime;               % when rolling finishes
```

(a) Give a MiniZinc model to ensure that all tasks are finished before `endtime` and no two tasks overlap in the schedule, and the endtime is minimized. There is no need to repeat the above declarations. [3 marks].

```
constraint forall(t in TASK)(start[t] + duration[t] <= endtime);
include "disjunctive.mzn";
constraint disjunctive(start, duration);
solve minimize endtime;
```

(b) A further constraint is that a "cold" task cannot be performed on the roller immediately after a "hot" task. If the previous task to a "cold" task is "hot" then we must wait at least 30 minutes after the "hot" task ends before we can start the "cold" task, to let the roller cool. Give additions to the MiniZinc model to ensure the "hot"/"cold" constraint is not violated. [4 marks].

Most efficient is to assign a previous task to each task, and constrain a previous hot task to get an extra 30 minutes break

```
set of int: TASK0 = 0..n;
array[TASK0] of var TASK0: prev; % previous task
include "alldifferent.mzn";
constraint alldifferent(prev);
constraint forall(t in TASK)
                 (prev[t] > 0 ->
                  start[prev[t]] + duration[prev[t]] + 30*(t in cold
                 /\ prev[t] in hot) <= start[t]);
```

Question 4 [10 marks]

(a) If the solver used does not support a constraint directly it needs to be decomposed into simpler constraints. Here are two predicate definitions for the constraint `mymin`(x, y, z) which encodes $x = min(y, z)$.

```
predicate mymin(var int: x, var int: y, var int: z) =  %%(A)
          x <= y /\ x <= z /\ (x = y \/ x = z);
predicate mymin(var int: x, var int: y, var int: z) =  %%(B)
          (y <= z /\ x = y) \/ (y > z /\ x = z);
```

Explain which decomposition is likely to be better and why? [2 marks].

(A) is substantially better, since it propagates much stronger than (B). For example if x in 0..12, y in 0..7, z in 4..9 it propagates that x in 0..7 whereas (B) propagates nothing. Remember disjunction is very weak for solvers.

(b) A strict valley in a sequence of numbers is a element of the sequence which is strictly less than both its two neighbors. End points of the sequence can never be strict valleys. For example the sequence [1,1,4,8,2,7,3,3,6,1] has one strict valley at the 5th position e.g. 8,2,7. The combinatorial function

```
function var int: strictvalley(array[int] of var int: x)
```

returns the number of strict valleys in the sequence x. Give a MiniZinc function definition for `strictvalley`. [3 marks].

```
function var int: strictvalley(array[int] of var int: x) =
                  sum(i in 1..length(x)-2)(x[i] > x[i+1] /\ x[i+1] < x[i+2]);
```

(c) Explain if the function definition you gave above can be used in a negative context, and why this is the case. [1 mark].

Yes it can, it introduces no local variables that are not defined.

(d) In order to run a MiniZinc model using a MIP solver, all variables and constraints must be eventually mapped to integer variables and linear integer constraints. In one sentence each briefly informally describe how MiniZinc maps

i. Boolean variables: `var bool: b`
ii. fixed array lookup: `x = a[y]`, where a is a fixed array of ints.
iii. global constraints: e.g. `alldifferent`

[4 marks].

Boolean variables are represented by 0-1 integers. Array lookup needs to break y in to Booleans/01s to represent the constraint

```
array[lb(y)..ub(y)] of var 0..1: yb;
constraint x = sum(i in lb(y)..ub(y))(a[i] * yb[i]);
constraint y = sum(i in lb(y)..ub(y))(i*yb[i]);
```

Global constraints have a separate linearization for the MIP solver which writes them in terms of linear constraints.

Question 5 [7 marks]

Consider the following partial model for a nurse rostering problem

```
int: k; % number of nurses
set of int: NURSE = 1..k;
int: m; % number of days
set of int: DAY = 1..m;
set of int: SHIFT = 1..3;
int: day = 1; int: night = 2; int: dayoff = 3;
int: o; % nurses required on day shift
int: l; % lower bound for nurses on nightshift
int: u; % upper bound for nurses on nightshift

array[NURSE,DAY] of var SHIFT: x;

constraint forall(n in NURSE, d in 1..m-2)
           ( x[n,d] = night /\ x[n,d+1] = night -> x[n,d+2] = dayoff /\
             x[n,d] = night -> x[n,d+1] != day );
constraint forall(n in NURSE)
           ( x[n,m-1] = night -> x[n,m] != day );

solve satisfy;
```

The model has constraints to ensure that after two night shifts in a row, a nurse must have a dayoff, and that directly after a night shift, a nurse cannot take a day shift

(a) Write constraints for this model that ensure that on every day there are exactly o nurses assigned to a day shift, and between l and u nurses assigned to a night shift. Write the constraint in the most efficient form you can. [3 marks].

```
constraint forall(d in DAY)
                (global_cardinality_low_up([x[n,d] | n in NURSE],
                                           [day,night], [o,l], [o,u]));
```

You could just count them but this was required for full marks!

(b) After adding correct constraints of the previous section that constrain the number of nurses on particular shifts, you find the model returns UNSATISFIABLE for all the data sets. Explain in a paragraph what process you would use to find the error in the model. [2 marks].

- Comment out constraints one by one until the problem is unsatisfiable, to find the smallest set of constraints which still cause unsatisfiability, then concentrate on those
- Build an expected solution by hand for a small example and add it into the model, hoping to get a better message from MiniZinc

(c) Explain where the error is in the above model, and how to correct it. [2 marks].

```
constraint forall(n in NURSE, d in 1..m-2)
           ( (x[n,d] = night /\ x[n,d+1] = night -> x[n,d+2] = dayoff) /\
             (x[n,d] = night -> x[n,d+1] != day) );
```

Question 6 [7 marks]

Given a problem with for the constraints: $y = abs(x)$, $y = 2z$, $x \neq 0$, $z \neq 1$ and initial domains $D(x) = D(y) = D(z) = -7..7$ and the search decision $x \leq 0$.

(a) Show the result of idempotent bounds propagation for the constraints: initially, and after the addition of the new constraint by search. Assume that at each stage in the search the propagation loop continually examines all propagators in the order shown until a fixpoint is reached. Show the domains of all variables after any change and the propagator which caused it, e.g.

constraint	$D(x)$	$D(y)$	$D(z)$
	$-7..7$	$-7..7$	$-7..7$
$y = abs(x)$	?	?	?

Recall that bounds propagators only examine and update bounds. [4 marks].

```
% -7..7          -7..7         -7..7
% y = abs(x)
% -7..7          0..7          -7..7
% y = 2z
% -7..7          0..6          0..3
% y = abs(x)
% -6..6          0..6          0..3
----- fixpoint now add search constraint ----
% x <= 0
% -6..0          0..6          0..3
% x != 0
% -6..-1         0..6          0..3
% y = abs(x)
% -6..-1         1..6          0..3
% y = 2z
% -6..1          2..6          1..3
% z != 1
% -6..1          2..6          2..3
% y = abs(x)
% -6..-2         2..6          2..3
% y = 2z
% -6..-2         4..6          2..3
```

(b) Show the result of domain propagation for the constraints: initially, and after the addition of the new constraint by search. Assume that at each stage in the search the propagation loop continually examines all propagators in the order shown until a fixpoint is reached. Show the domains of all variables after any change and the propagator which caused it. [3 marks].

```
% -7..7            -7..7         -7..7
% y = abs(x)
% -7..7            0..7          -7..7
% y = 2z
% -7..7            0,2,4,6       0..3
% x != 0
% -7..-1,1..7      0,2,4,6       0..3
% z != 1
% -7..-1,1..7      0,2,4,6       0,2..3
% y = abs(x)
% -6,-4,-2,2,4,6   2,4,6         0,2..3
% y = 2z
% -6,-4,-2,2,4,6   4,6           2..3
% y = abs(x)
% -6,-4,4,6        4,6           2,3
```

```
----- fixpoint now add search constraint -----
% x <= 0
% -6, -4          4,6          2,3
```

Question 7 [11 marks]

(a) Consider the following MiniZinc predicate:

```
predicate strange(array[int] of var int: x, int: y) =
 forall(i,j in index_set(x))(
        if i < j then x[i] + y <= x[j]
        else true endif );
```

Show the conjunction of primitive constraints of the form $u + v \leq w$ from unrolling the MiniZinc constraint

```
constraint strange([a,b,c,d],3);
```

[3 marks].

```
% a + 3 <= b
% a + 3 <= c
% a + 3 <= d
% b + 3 <= c
% b + 3 <= d
% c + 3 <= d
```

(b) The definition of `strange` is not as simple as it could be and generates redundant constraints. Give a new MiniZinc definition of `strange` that is as efficient as possible but still has the same logical meaning. [3 marks].

```
predicate strange(array[int] of var int: x, int: y) =
 forall(i in min(index_set(x)) .. max(index_set(x)) - 1)
       (x[i] + y <= x[i+1]);
```

(c) Briefly explain (in at most two sentences) the *relational semantics* used by MiniZinc. [2 marks].

The relational semantics of MiniZinc says that undefined values flow upwards to the nearest enclosing Boolean context where they become false. For example `bool2int(x > 3 * 4y + z div u)` evaluates to 0 when $u = 0$!

(d) Compiling the MiniZinc model

```
array[1..5] of var 1..5: x;
var 3..6: u;
include "alldifferent.mzn";
constraint alldifferent(i in 1..5 where i < u)(x[i]);
solve satisfy;
```

gives an error message

```
MiniZinc: type error: no function or predicate with this signature
found: 'alldifferent(array[int] of var opt int)'
```

Explain why this error arises, and how you could rewrite the constraint to avoid the error. [3 marks].

The problem arises since the use of the variable in the where clause means that the array construction is creating and array of var opt ints, rather var ints. The best solution is to make the values of interest to the alldifferent explicit and replace the others by zero using alldifferent_except_zero.

```
constraint alldifferent_except_zero( [ x[i] * (i < u) | i in 1..5 ]);
```

Question 8 [16 marks]

An aircargo planning operation has a single plane to shift cargoes around. There are N airports to visit, and E flight legs from airport to airport. There are C cargos which have to be delivered from one airport to another. The plane can carry only one cargo at a time. This is a planning problem with *maxsteps* actions that can be taken, and the possible actions are

Action	shorthand	integer representation
do nothing	`donothing`	0
pickup cargo c	`pickup` c	c
move to airport n	`move` n	$C+n$
drop current cargo	`drop`	$C+N+1$

Actions are represented by integers $0..C+N+1$.

Data for the problem is as follows:

```
int: N;                          % number of airports
set of int: NODE = 1..N;         % airports
int: E;                          % number of flight routes
set of int: EDGE = 1..E;         % routes
array[EDGE] of NODE: fst;        % first airport
array[EDGE] of NODE: snd;        % second airport

int: C;                          % number of cargoes
set of int: CARGO = 1..C;        % cargoes
array[CARGO] of NODE: from;      % cargo pickup place
array[CARGO] of NODE: to;        % cargo dropoff place
```

```
set of int: ACT = 0..C+N+1;     % possible actions a,(egs where C = 2, N = 6)
int: donothing = 0;             % donothing,         0 represents donothing
set of int: PICKUP = 1..C;      % pickup a,     e.g. 2 represents pickup 2
set of int: MOVE = C+1..C+N;    % move (a - C), e.g. 6 represents move 4
int: drop = C + N + 1;          % drop,              9 represents drop

int: maxsteps;                  % max actions
set of int: STEP = 0..maxsteps; % steps

NODE: start;                    % aircraft start
```

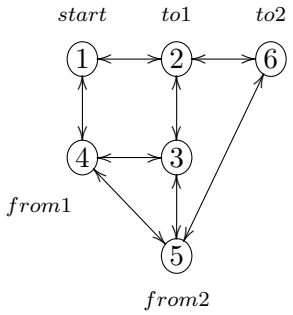
For example the data file:

```
N = 6;
E = 8+8;
fst = [1,1,2,2,3,3,4,5] ++ [2,4,3,6,4,5,5,6];
snd = [2,4,3,6,4,5,5,6] ++ [1,1,2,2,3,3,4,5];

C = 2;
from = [4,5];
to   = [2,6];

maxsteps = 20;

start = 1;
```



describes a situation of six airports with 8 possible routes, that are bidirectional, 2 cargos and the plane starting at airport 1. The airport network is shown to the right

The objective of the plan is to deliver all the cargoes from their pickup position to their dropoff position. A cargo can only be picked up at its pickup place, and only dropped off at its dropoff place. The plane can only move from airport a to airport b if their is an edge (a, b).

The expected output is a sequence of actions, and the number of steps taken, for example

```
act = [6, 1, 3, 4, 9, 5, 7, 2, 8, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];
steps = 10;
```

This indicates a plan of `move 4, pickup 1, move 1, move 2, drop, move 3, move 5, pickup 2, move 6, drop` followed by doing nothing. It takes 10 actions. Clearly each action is allowed: moves are along edges, pickups occur at the right place, and drop offs occur at the right place, and all cargoes are delivered.

(a) The important state of the problem is: the position of the plane, what it is carrying (if anything), and what cargoes have been delivered. The key decisions are the action

at each step, and the number of steps required. Give MiniZinc variable declarations for the key decisions and state representation of the problem. [3 marks].

```
set of int: CARGO0 = 0..C;                  % cargo or 0 = nothing
array[STEP] of var NODE:   posn;            % position of plane
array[STEP] of var CARGO0: carry;           % what its carrying if anything
array[STEP] of var ACT:    what;            % action taken
array[STEP,CARGO] of var bool: delivered;   % what cargo is delivered
var STEP: steps;                            % steps in plan
```

(b) Define a MiniZinc predicate that updates the position of the plane given the next action. It should ensure that if the action is a move action then it is legitimate, and if the action is not a move action, that the position does not change. [3 marks].

```
predicate act_posn(var NODE: posn_b, var ACT: act, var NODE: posn_a) =
          if act in MOVE then
             posn_a = act - C /\
             exists(e in EDGE)(fst[e] = posn_a /\ snd[e] = posn_b)
          else posn_a = posn_b endif;
```

(c) Define a MiniZinc predicate that updates state for what the plane is carrying, and which cargoes are delivered depending on the next action. It should make sure that pickup and drop actions are legitimate, e.g. you cant pickup a cargo that has been delivered, you can only pickup a cargo from its "from" position, you only drop a cargo at its "to" position. For actions that do not change this state it should ensure the state is unchanged. [4 marks].

```
predicate act_carry(var CARGO0: carry_b, var NODE: posn_b,
                    array[CARGO] of var bool: del_b,
                    var ACT: act, var CARGO0: carry_a,
                    array[CARGO] of var bool: del_a) =
          if act = drop then
             carry_b != 0 /\ posn_b = to[carry_b] /\ carry_a = 0 /\
             del_a[carry_b] = true /\
             forall(c in CARGO)(c != carry_b -> del_a[c] = del_b[c])
          elseif act in PICKUP then
             carry_b = 0 /\ posn_b = from[act] /\ carry_a = act /\
             del_b[act] = false /\ del_a = del_b
          else carry_a = carry_b /\ del_a = del_b endif;
```

(d) Define a MiniZinc constraint that ensures that the plan delivers all cargos. [1 marks].

```
constraint sum(s in STEP)(what[s] = drop) = C;
```

If all C cargoes are dropped they are delivered. Alternatively

```
constraint forall(c in CARGO)(delivered[steps,c]);
```

(e) Give a correct solve item for the model, together with a search annotation that will make a CP solver effective at tackling your model. In a sentence justify why the search annotation is likely to be useful. [2 marks].

```
solve :: int_search([steps], input_order, indomain_min, complete)
      minimize steps;
```

The aim is to minimize the number of steps, by searching from steps from 0 .. we find an optimal plan first, and dont get stuck trying to find very long plans.

(f) Complete your MiniZinc model to give a full solution to the problem. [3 marks].

```
constraint posn[0] = start /\ carry[0] = 0 /\ when[0] = 0 /\
           what[0] = donothing /\
           forall(c in CARGO)(delivered[0,c] = false);

constraint forall(s in 1..maxsteps)
                 (act_posn(posn[s-1], what[s], posn[s]) /\
                  act_carry(carry[s-1], posn[s-1], [ delivered[s-1,c] | c in CARGO],
                            what[s], carry[s], [delivered[s,c] | c in CARGO]) /\
                  (s > steps -> what[s] = donothing));
```

Make sure that the starting state is correct, and then ensure that the effect of the action on posn and what is carried is correct, finally ensure nothing happens after the number of steps.