The University of Melbourne
Department of Computing and Information Systems
Semester 2 Assessment 2014

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 [6 marks]

Here is an odd little puzzle which occurred the other day at an archery meeting. The young lady who carried off the first prize scored exactly one hundred points. The scores on the target are: 16, 17, 23, 24, 39 and 40. Can you figure out how many arrows she must have used to accomplish the feat?

(a) Write a MiniZinc model to determine, for a given set of scores and a target total score, the minimum number of arrows needed to achieve exactly that score. The data format should be

```
array[int] of int: scores;
int: target;
```

For example the data for the problem above is

```
scores = [16,17,23,24,39,40];
target = 100;
```

Ensure that any decision variables you define have tight upper and lower bounds. [4 marks].

```
array[int] of int:  scores;
int:  target;
int:  maxarrow = target div min(scores);
array[index_set(scores)] of var 0..maxarrow:  hits;
constraint sum(i in index_set(scores))(scores[i]*hits[i])
= target;
solve minimize sum(i in index_set(scores))(hits[i]);
scores = [16,17,23,24,39,40];
target = 100;
```

(b) Which kind of constraint solver is likely to be best at solving your model: a constraint programming (CP) solver or a mixed integer programming (MIP) solver? Briefly explain the reasons for your answer. [2 marks].

A MIP solver since the constraints are all linear.

Question 2 [8 marks]

(a) There are at least 3 ways to represent a decision about a set of integers in MiniZinc. Briefly explain 3 methods to do so. Explain which method you would use for representing decisions for each of the following cases, and why:

i. Choosing a set of at most 3 elements of numbers from $\{1, \ldots, 10000\}$
ii. Choosing a subset of $\{1, \ldots, 12\}$
iii. Choosing a subset of $\{1, \ldots, 20\}$ which adds up to 30.

[3 marks].

- `var set of U`: native set representation
- `array[1..maxcard] of var (U union { null })`: an array of the elements in the array, usually sorted, with an extra value to `null` represent no element
- `array[U] of var bool` or `array[U] of var 0..1`: an indicator array representing which values are in

(i) The second representation is most efficient for (i) since its an array of size 3.

```
array[1..3] of var 0..10000:  S;
constraint S[1] >= S[2] /\ S[2] >= S[3];
```

(ii) The first representation is probably best for (ii) since its a small set, although the third would be ok.

```
var set of 1..12:  S;
```

(iii) All representations are probably acceptable. The third might be preferable for easy summing.

```
array[1..20] of var 0..1:S;
constraint sum(i in 1..20)(i*S[i]) = 30;
```

The second has a tight cardinality bound of 7, so might be fine

```
array[1..7] of var 0..20:S;
constraint sum(S) = 30;
constraint forall(i in 1..6)(S[i] >= S[i+1]);
```

(b) Explain how a multiset decision is usually represented in MiniZinc. Discuss briefly why there is no built in `multiset` type in MiniZinc. [2 marks].

A multiset of the set U is represnted as `array[U] of var 0..c` where c is the cardinality bound. There is no built in mujltiset type since the array representation is essentially exactly a multiset representation.

(c) Sometimes when modelling it is necessary to take the input data format and create some new data. Briefly explain three different ways in which this can be managed in MiniZinc, and contrast their advantages and disadvantages. [3 marks].

- Add it to the data file. Advantages: simple, can be programmed externally; Disadvantages: needs to be checked with assertions to see its correct
- Add constraints to compute it. Advantages: uses power of solver. Disadvantages: data is not known while creating model, solving overhead
- Rewrite the data in MiniZinc. Advantages: data is available and correct; Disadvantages: may require complex coding

Question 3 [7 marks]

Given n workers numbered $1..n$ that each need to be assigned to a different task from a set of n tasks numbered $1..n$ where the value of worker i working on task j is given by $v[i, j]$:

(a) Write a MiniZinc model to find the assignment that maximizes the total value of the assignment. Make use of global constraints if you can. [4 marks].

```
      int:  n;
      set of int:  WORK = 1..n;
      set of int:  TASK = 1..n;
      array[WORK,TASK] of int:  v;
      array[WORK] of var TASK: t;
      constraint alldifferent(t);
      solve maximize sum(i in WORK)(v[w,t[w]]);
```

(b) Suppose worker 4 must work on a task numbered lower than that worked on by worker 3. Write a constraint expressing this for the model for part (a). [1 mark].

```
      constraint t[4] < t[3];
```

(c) Suppose task 2 must be worked on by a worker numbered at least 2 different from the worker for task 1. Modify your model of part (a) to allow this constraint to be expressed easily. [2 marks].

```
      array[TASK] of var WORK: w;
      constraint inverse(t,w);
      constraint abs(w[2] - w[1]) >= 2
```

Question 4 [6 marks]

There are many counting global constraints of interest in combinatorial problems. One of the simplest is `count_eq` which enforces that c is the number of times that a position in the x array takes the value y. It is defined as

```
predicate count_eq(array[int] of var int: x, var int: y, var int: c) =
    c = sum(i in index_set(x)) ( bool2int(x[i] == y) );
```

In this question we examine some other counting globals. The combinatorial constraint

```
common(var int: n1, var int: n2,
       array[int] of var int: c1, array[int] of var int: c2)
```

holds if $n1$ is the number of positions in the array $c1$ taking a value in the array $c2$, and $n2$ is the number of positions in the array $c2$ taking a value in the array $c1$. For example

- `common(3,4,[1,9,1,5],[2,1,9,9,6,9])` holds since three positions in [1,9,1,5] are in the set {1,2,6,9} of values of [2,1,9,9,6,9], and similarly 4 values in [2,1,9,9,6,9] are in the set $\{1, 5, 9\}$.
- `common(4,1,[1,1,1,1],[2,1,9,9,6,9])` holds

(a) Give a predicate definition for `common`. [4 marks].

```
predicate common(var int:  n1, var int:  n2,
    array[int] of var int:  c1, array[int] of var int:  c2) =
    n1 = sum(i in index_set(c1))
            (bool2int(exists(j in index_set(c2))(c1[i] = c2[j])) /\
    n2 = sum(i in index_set(c2))
            (bool2int(exists(j in index_set(c1))(c2[i] = c1[j]));
```

(b) There is a relationship between $n1$ and $n2$ which always holds in every solution of the `common` constraint. Write down the strongest redundant constraint you can that only mentions $n1$ and $n2$. [1 mark].

```
constraint n1 = 0 <-> n2 = 0;
```

(c) The combinatorial constraint

```
uses(array[int] of var int: c1, array[int] of var int: c2)
```

holds if the set of values in the array $c2$ is included in the set of values in the array $c1$. Write a definition of `uses` making use of the `common` constraint. [1 mark].

```
predicate uses(array[int] of var int:  c1,
               array[int] of var int:  c2) =
          let { var 0..length(c1):  n1 } in
          common(n1, length(c2), c1, c2);
```

Question 5 [11 marks]

Consider the following model:

```
predicate red(var int: x) =
          let { var int: y; } in x = 4*y + 1;
predicate blue(var int: x) =
          let { var int: y; } in red(x) \/ x = 2*y;
var 0..5: a;
var 0..5: b;
constraint b = a + 2;
solve satisfy;
output [show(a)," ",show(b)];
constraint red(a) -> blue(b);
```

(a) Give the set of solutions expected from the model. [3 marks].

(a,b) = {(0, 2), (2, 4), (3, 5)}

(b) Give the contexts: positive, negative or mixed for each of the expressions: `red(a)`, `blue(b)` and `red(x)` appearing in the model. [2 marks].

red(a): negative
blue(b): positive
red(x): positive

(c) In fact MiniZinc complains about something in the last line of the model. Explain what the problem is, and how the model could be rewritten to avoid the problem (without changing the set of solutions intended). [2 marks].

Use of local variable in the definition of `red(a)` in a negative context. This would lead to universal quantification and is not allowed. If we ensure y has a unique value given x the problem disappears.

```
     predicate red(var int:x) =
               let { var int:  y = (x - 1) div 4; } in x = 4*y + 1;
```

(d) Ignoring the problem in the previous part (or assuming its fixed) when asked for all solutions MiniZinc outputs an infinite number of solutions, all the same. Explain why this occurs, and correct the definition of the model to avoid generating an infinite set of solutions. [2 marks].

The local variable in the definition of `blue` can take any value, so we try each possible value for this variable. Push the y definition into the disjunct, and make it unique

```
     predicate blue(var int:x) =
               red(x) \/ let { var int:  y = x div 2; } in x = 2*y;
```

(e) Consider the two following logically equivalent constraint expressions. Explain which is preferable and why? [2 marks].

```
not exists(i in 1..n)                  forall(i in 1..n)
          (forall(j in 1..n-1)               (exists(j in 1..n-1)
                 (x[i,j] <= x[i,j+1]));             (x[i,j] > x[i,j+1]));
```

(a) (b)

The second version (b) is preferable since the forall at the top level, builds a top level set of exists constraints. These propagate much better than the negated constraint generated by (a)

Question 6 [14 marks]

Given a problem with for the constraints: $x = y \times z$, $z \neq 1$ and initial domains $D(x) = 0..9$, $D(y) = 0..5$, $D(z) = 0..5$ and search decisions in order $x \geq 4$, $z \leq 3$, $y = 3$.

(a) Show the result of bounds propagation for the constraints: initially, and after every addition of a new constraint by the 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. Recall that bounds propagators only examine and update bounds. [4 marks].

c	x	y	z
init	0..9	0..5	0..5
$x \geq 4$	4..9		
$x = y * z$		1..5	1..5
$z \neq 1$			2..5
$x = y * z$		1..4	
$z \leq 3$			2..3
$x = y * z$		2..4	
$y = 3$		3	
$x = y * z$	6..9		

(b) Show the result of domain propagation for the constraints: initially, and after every addition of a new constraint by the 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].

c	x	y	z
init	0..9	0..5	0..5
$x = y * z$	0..6, 8..9		
$z \neq 1$			0, 2..5
$x = y * z$	0, 2..6, 8..9		
$x \geq 4$	4..9		
$x = y * z$	4..6, 8..9	1..4	2..5
$z \leq 3$			2..3
$x = y * z$	4, 6, 8, 9	2..4	
$y = 3$		3	
$x = y * z$	6, 9		

(c) Give pseudo-code defining a bounds propagator for the constraint: $x = sign(y)$ where $sign(d)$ returns the sign of the integer argument: 0 if $d = 0$, 1 if $d > 0$ and -1 if $d < 0$. Make the propagator as strong as possible. Use functions $\mathsf{lb}(v)$ and $\mathsf{ub}(v)$ to access the current bounds of variable v, and functions $\mathsf{setlb}(v, d)$ and $\mathsf{setub}(v, d)$ to set the bounds of a variable. You can assume that the setting functions won't change to a weaker bound: e.g. $\mathsf{setlb}(x,\mathsf{lb}(x) - 1)$ will have no effect. [3 marks].

```
setub(x,1);
setlb(x,-1);
if (lb(x) = 0) setlb(y,0);
if (lb(x) = 1) setlb(y,1);
if (ub(x) = 0) setub(y,0);
if (ub(x) = 1) setub(y,-1);
if (lb(y) = 0) setlb(x,0);
if (lb(y) > 0) setlb(x,1);
if (ub(y) = 0) setub(x,0);
if (ub(y) < 0) setub(x,-1);
```

(d) Constraint programming solvers require that propagators are: correct and checking. Define each concept in a short paragraph, explain why each is necessary for the solver to operate correctly. [4 marks].

A correct propagator never removes values from the domain of a variable which can take part in a solution of the constraint given the current domain. Incorrect propagators can miss solutions by erroneously removing them.
A checking propagator is guaranteed to cause a false domain if all the variables in the constraint are fixed in the current domain and it does not represnent a solution. Checking propagators guarantee that once all variables are fixed the constraints are actually satisfied.

Question 7 [7 marks]

Consider the following MiniZinc predicate:

```
predicate strange(array[int] of var int: s,
                  array[int] of int: d) =
 forall(i,j in index_set(s))(
       let {  var bool: b1;
              var bool: b2 = not(b1); } in
       if i != j then
             (b1 -> s[i] + d[i] <= s[j])
          /\ (b2 -> s[j] + d[j] <= s[i])
       else true endif );
```

(a) Show the conjunction of primitive constraints of the form $b = not(b')$ and $b \to a + d \leq c$ that result from unrolling the MiniZinc constraint

```
constraint strange([x,y,z],[3,1,2]);
```

[4 marks].

```
b211 = not(b111)
b212 = not(b112)
b112 -> x + 3 <= y
b212 -> y + 1 <= x
b213 = not(b113)
b113 -> x + 3 <= z
b213 -> z + 2 <= x
b221 = not(b121)
b221 -> x + 3 <= y
b121 -> y + 1 <= x
b222 = not(b122)
b223 = not(b123)
b123 -> y + 1 <= z
b223 -> z + 2 <= y
b231 = not(b131)
b231 -> x + 3 <= z
b131 -> z + 2 <= x
b232 = not(b132)
b232 -> y + 1 <= z
b132 -> z + 2 <= y
b233 = not(b133)
```

(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:  s,
                  array[int] of int:  d) = disjunctive(s,d);
```

Question 8 [11 marks]

The company Splashdown delivers portable lavatories to events. Each event has a given demand for lavatories, a duration, a start date, a contract price, and a penalty price. Splashdown can choose to ignore an event earning nothing, or fulfil the event contract totally earning the contract price, or fulfil it partially earning the contract price minus the penalty price for each missing lavatory. Splashdown has a limited stock of lavatories, but it may choose to buy new lavatories at a cost, or subcontract out some of the lavatories required for an event, paying the daily cost of the subcontracting times the number of lavatories subcontracted times the event duration. The aim is to make the most profit for the planning period.

Data for the problem is as follows:

```
int: n;                           % days of planning period
set of int: DAY = 0..n;
int: stock;                       % initial stock of lavatories
int: m;                           % number of events
set of int: EVENT = 1..m;
array[EVENT] of DAY: start;       % start day
array[EVENT] of int: duration;    % duration in days
array[EVENT] of int: demand;      % demand of lavatories
array[EVENT] of int: contract;    % contract price
array[EVENT] of int: penalty;     % penalty price per lavatory
int: buy_cost;                    % cost to buy new lavatory
int: daily_rent_cost;             % cost to rent for one day
```

For example the data file:

```
n = 10;
stock = 5;
m = 8;
start =    [0,0,1,2,3,3,5,7];
duration = [3,4,2,5,2,5,2,3];
demand =   [4,3,4,2,5,2,3,5];
contract = [60,70,35,55,50,45,35,80];
penalty =  [15,25,10,30,10,20,10,15];
buy_cost = 35;
daily_rent_cost = 5;
```

describes a situation where there is a 10 day plan, an initial stock of 5, with 8 events with varying parameters. The cost for buying a lavatory is 35, and the cost for subcontracting each lavatory is 5 per day.

Expected output is a plan of the form:

```
sent = [4, 0, 0, 1, 4, 0, 3, 5]
rented = [0, 3, 0, 1, 0, 0, 0, 0]
bought = 0
obj = 255
```

showing the number of lavatories sent to each event, the number rented for each event, the number bought, and the total profit. Here the plan is to totally fulfil events 1,2,4,7,8, partially fulfil event 5 (4 out of 5), and ignore events 3 and 6.

(a) Give a MiniZinc model for the Splashdown problem. You do not need to repeat the data declarations. Ensure that each integer variable is given a tight range of possibilities. Ensure the output item returns output in the correct form. [9 marks].

```
% splashdown
int: n;                       % days of planning period
set of int: DAY = 0..n;
int: stock;                   % initial stock of lavatories
int: m;                       % number of events
set of int: EVENT = 1..m;
array[EVENT] of DAY: start;    % start day
array[EVENT] of int: duration; % duration in days
array[EVENT] of int: demand;   % demand of lavatories
array[EVENT] of int: contract; % contract price
array[EVENT] of int: penalty;  % penalty price per lavatory
int: buy_cost;                % cost to buy new lavatory
int: daily_rent_cost;         % cost to rent for one day

int: maxd = max(demand);

array[EVENT] of var 0..maxd: sent;
array[EVENT] of var 0..maxd: rented;
array[EVENT] of var 0..maxd: total = [ sent[e] + rented[e]
                                     | e in EVENT ];

int: totald = sum(demand);
var 0..totald-stock: bought;
var int: new_stock = stock+bought;

constraint cumulative(start, duration, sent, new_stock);
constraint forall(e in EVENT)(total[e] <= demand[e]);

var int: rent_cost = daily_rent_cost *
                     sum(e in EVENT)(rented[e] * duration[e]);

var int: income = sum(e in EVENT)(bool2int(total[e] > 0)*
                  (contract[e] - penalty[e] * (demand[e] - total[e])));

var int: obj = income - bought * buy_cost - rent_cost;

solve maximize obj;

output ["sent = ", show(sent), "\n"]
++["rented = ", show(rented), "\n"]
++["bought = ", show(bought), "\n"]
++["obj = ", show(obj), "\n"]
;
```

(b) Give a good search annotation for the model. Explain why you think this is likely to give good search behaviour. [2 marks].

```
array[1..2*m] of var int:  decisions =
                           [ if i mod 2 = 0 then
                                -sent[i div 2]
                             else
                                rented[i div 2] endif
                           | i in 2..2*m+1 ];
solve ::  int_search(decisions, input_order, indomain_min, complete)
        maximize obj
```

Make decisions for each contract in order, trying to send the most possible of the lavatories, and renting the minimum. This leads to the maximum utilization of the current resources, without renting, and propagates strongly the available stock.