

Do 2 questions from 3, 25 marks each question

1a

A constraint $C_{j,k}$ is arc-consistent if for every value in the domain D_j there is a supporting value in the domain D_k , and a problem is arc-consistent if every constraint is arc-consistent. Consequently, we can instantiate any variable in isolation and be guaranteed that we can select a second variable and assign it a consistent value

1b.

```
revise(j,k) : boolean
begin revised := false
  for each value x in Dj
  do begin supported := false
    for each value y in Dk while not supported
    do supported := check(x,j,y,k)
      if ¬supported
      then begin
        Dj := Dj \ x
        revised := true
      end
    end
  end
revised
end
```

AC3(C)

```
begin Q := C
  while notEmpty(Q)
  do begin
    (j,k) := dequeue(Q)
    if revise(j,k)
    then Q := Q U {(a,b) | (a,b) in C and b = j}
    end
  end
end
```

1c

AC3 queues up constraints that need revision because one of the variables involved in that constraint has lost some value, but these values are not specified. In AC5 we enqueue tuples of the form (j,k,x) , where variable k has lost value x from its domain and we now need to revise the domain of variable j . This gives AC5 an advantage, because if we know something about the semantics of the constraint, we can then efficiently revise that domain with respect to the lost value

1d

x is in $\{1..6\}$, y in $\{6..9\}$, z in $\{2,3\}$

This is because constraint propagation uses bounds consistency, exploiting the properties of the relations to achieve this level of consistency with greater efficiency. From the result it would then appear that AC5 was used

1e

x in $\{1..6\}$, y in $\{6,9\}$, z in $\{2,3\}$

Since nothing is known, by default, about the explicit relation it is processed using AC3, achieving a stronger level of consistency. However, it pays the cost: it is more computationally expensive.

2a

Have a set of variables W , where $W[i]$ is in $\{0, w_i\}$, and w_i is the weight of the i th item. We also have a set of variables V , where $V[i]$ is in $\{0, v_i\}$, and v_i is the value of the i th item. We have the optimising variable $\text{sum}V$. $\text{sum}V$ is the sum of the values of the items in the knapsack, and has a domain $\{0..y\}$ where y is the sum of the values of the items, i.e. the value of the knapsack can be in the range 0 to the maximum where all items are in the knapsack.

We then have the following constraints:

- (i) $W[i] = 0$ if and only if $V[i] = 0$. That is, the i th item has zero weight contribution when it adds no value to the knapsack. Since we have the biconditional, this also means that when the i th item is added to the knapsack it contributes its weight and its value
- (ii) $\text{SumVars}(W) \leq X$. That is, the sum of the weights must not exceed the knapsack's capacity.

2b

As a variable ordering heuristic we might prefer items with largest v/w i.e. the most value per unit weight. This might be good as we want valuable items that do not over fill the knapsack. When we have a tie between two items we might choose the largest, as this will have the largest value. As a value ordering heuristic we might prefer to select a value that is not zero, i.e. we will choose to take valuable items rather than exclude them.

There might be a symmetric heuristic, where we select variables with low v/w ratios (not valuable) and prefer to assign the value zero, i.e. exclude them from the knapsack. However, the first solution would then be an empty knapsack, and it might then be tedious to reach an optimum.

2c

We optimise by a sequence of decisions. Assume we have maximising variable V . We start off with a low value of V and find a satisfying assignment to the variables. We then increase V and try and find another solution that respects V . Assume we are incrementing V . When we discover that the problem is insoluble for a value of $V = x$, we then know that the optimum value was at the previous decision problem, when $V = x-1$. We might also post this constraint inside search, rather than start again, and we then have branch and bound.

3a

In MBO we sequence variables such that we minimise the largest *distance* between constrained variables. That is, if there is a constraint between variables V and W and V is instantiated in position i and W is instantiated in position j , and $i < j$, then the distance between them is $j - i$. We then order the variables to minimise the largest distance. The justification for this being a good heuristic is that the largest distance we need to backtrack in order to resolve a conflict is reduced. This will then go some way to reducing thrashing.

MAXDEG chooses variable in order of non-decreasing degree sequence, looking first at the variable that is involved in most constraints. The intuition behind this, is that we prefer to make decisions that have greatest effect, again leaving easy decisions to later on in the search.

3b

SDF chooses to instantiate the variable with the smallest domain. This is a realisation of the fail-first principle, attempting to solve the hardest parts of the problem first, because if they fail, there is no need to consider the remaining *easy* decisions.

3c

The heuristics are NOT algorithm independent. For example, SDF really only works for an algorithm that looks forwards, propagating into the future variables. So, it is really only suitable for algorithms like FC (forward checking) and MAC (maintaining arc consistency), otherwise it would behave as a static ordering heuristic. MBO is most likely a waste of effort when we have an intelligent search algorithm, such as DB (dynamic backtracking) or CBJ (conflict-directed backjumping). These algorithms can jump back to the source of a conflict, and naturally reduce thrashing. Note also, finding the best minimum bandwidth ordering is NP-complete.

3d

The heuristics are NOT problem independent. If we have a problem with some very large domains, some very small domains, and all constraints very weak, a SDF ordering will perform much worse than the anti-heuristic, instantiating largest domains first.

MAXDEG might just be dreadful, because it ignores the tightness of constraints. Therefore, we could have 2 variables that constrain each other tightly, and do not constrain any other variable. All other variables could form a clique with loose constraints. We end up making our hardest decisions late on, and for unintelligent search this will be a recipe for a good thrashing.

3e

On unsolvable problems SDF encourages us to make the hardest decisions first, encouraging search to make early failures. Therefore, we expect to find short proofs of insolubility. When a problem is solvable, search will still make mistakes, branching off into wrong decisions. When it does this, SDF should cut those branches off quickly. But more interestingly, if each value in the domain of a variable is equally likely to occur in a solution, then by choosing a variable with smallest domain we increase the probability that an assignment is in a solution! That is, the SDF heuristic then displays an element of promise!

3f

For good old fashioned chronological backtracking, to prove insolubility we must consider all values in the domain of a variable, and the order will then be immaterial. Therefore value ordering does not help us when proving insolubility.