CS188 Fall 1992 | Intro to AI Stuart Russell | **Midterm solutions**

1. **(15 pts.) Definitions**
   Provide brief, *precise* definitions of the following:
   (a) Completeness: an inference procedure is complete iff all logical entailments of a given theory can be proved by the procedure.
   (b) Validity: a sentence is valid iff it is true in all possible worlds under all interpretations.
   (c) Agent: a physical object that can be analysed as having perceptions and producing actions.
   (d) Procedural attachment: a method for connecting a non-logical solution mechanism to specific types of goals and assertions in order to speed up theorem-proving or generate side-effects.
   (e) Heuristic search: any search mechanism employing domain-specific information to guide the search process (usually, information about the estimated quality of partial solutions).

2. **(22+3 pts.) Search**

   (a) (3) Termination condition: when the two open lists have a non-empty intersection of *states* — remember that the open lists usually contain *nodes*, which may differ on aspects other than the state; if the lists just contain states, then there is no way to get two different paths from the common state! Once there is some intersection, it only disappears again if the state in common is expanded by one of the two searches. Hence we can get away with checking if the node being expanded is on the other list.
   (b) (2) Solution extraction: applying `get-path` to each of the two nodes sharing state, we obtain two halves of the solution. We need to reverse the path from the goal node and append it to the end of the path from the start node. We also need to make sure that the common node does not appear twice!
   (c) (2) If we use `successors` to generate nodes from the goal state, we must be sure that the steps are reversible: i.e., if A is a successor of B then B must be a successor of A.
   (d) (9) The key issues are:
      i. Failure occurs if either of the lists becomes empty.
      ii. For intersection must check *states*, not nodes.
      iii. We have to check intersection after *each* expansion, not after one expansion of both lists (otherwise the two searches might cross over. Since the expansions are going to alternate, it's easiest to switch the arguments each time:

```
(defun bds (open1 open2 &optional (inorder? t))
  (cond ((or (null open1) (null open2)) 'fail)
        ((check-for-solution open1 open2 inorder?))
        (t (bds open2 (append (cdr open1) (successors (car open1))) (not inorder?)))))

(defun check-for-solution (open1 open2 inorder?)
  (let ((join2 (car (member (car open1) open2 :key #'node-state :test #'equal))))
    (when join2
      (let ((join1 (car open1)))
        (if inorder?
          (append (get-path join1) (reverse (cdr (get-path join2))))
          (append (get-path join2) (reverse (cdr (get-path join1)))))))))
```

   We weren't too worried about the nitty-gritty details of the solution extraction, which can be rather annoying, as long as the basic aspects mentioned above appear somewhere.

(e) (3) Search space: each half has branching factor b and depth $d/2$, so the total amount of search is roughly $2b^{d/2}$. Although there is no heuristic information, this is better than breadth-first or DFID. How? Because we assume the goal state is known, which gives us a big advantage.

(f) (3) Heuristic function: the idea is to guide each frontier towards the other. A simple, and acceptable, idea, is to use a heuristic function for each so that h_1 estimates the distance to the goal, h_2 the distance to the start. Unfortunately, the two searches might then pass eachother by and end up meeting pretty near the start or goal, in which case there is no gain. Since the solution is going to go through some node in the opposite frontier, a second idea is to estimate the cost to reach the goal/start *via one of the nodes in the opposite frontier*. Finding the least cost opposing node to head for is not at all easy to do efficiently, but appears to be worthwhile in practice.

(g) (3, extra credit) Using the latter idea, we can ensure admissibility by ensuring that the estimates of complete path cost are all optimistic, as in A*.

3. (10 pts.) Logic

Represent the following sentences in predicate calculus:

(a) (2) Calculators contain at least one battery:
$\forall c\ Calculator(c) \Rightarrow \exists b\ Battery(b) \wedge PartOf(b, c)$

(b) (3) All calculators have a 4 button below the 7 button:
$\forall c\ Calculator(c) \Rightarrow$
$\exists b_4 b_7 Button(b_4) \wedge Button(b_7) \wedge PartOf(b_4, c) \wedge PartOf(b_7, c) \wedge$
$Label(b_4, 4) \wedge Label(b_7, 7) \wedge Below(b_4, b_7)$

(c) (3) HP calculators are cheaper than Sharp calculators.
$\forall hs\ Calculator(h) \wedge Calculator(s) \wedge Maker(h, HP) \wedge Maker(s, Sharp)$
$\Rightarrow < (Price(h), Price(s))$

(d) (2) Only nerds have calculators.
$\forall xc\ Owns(x, c) \wedge Calculator(c) \Rightarrow Nerd(x)$

4. (11 pts.) Inference

(a) (2) Modus Ponens:

$$\frac{P,\ P \Rightarrow Q}{Q}$$

(b) (6) Soundness proof using resolution: obviously to do this we must assume that resolution is sound itself. Use resolution to prove that Q follows from the premises by negating Q, adding to the CNF version of the premises. This gives us:
(1) P (2) $\neg P \vee Q$ (3) $\neg Q$
Resolving (2) and (3) we get (4) $\neg P$.
Resolving (1) and (4) we get the empty clause.

(c) (3) Trivially, yes, because we can convert propositional calculus to predicate calculus by adding an empty argument list () to each proposition symbol; and resolution is complete for predicate calculus.

5. (10 pts.) Situation calculus, knowledge representation

(a) (6)
Some people wrote axioms for actions other than *buy*, which was interesting. The main things are 1) get the basic situation calculus stuff right — facts needing a situation argument, and outcomes described by facts about the result situation; 2) ownership switches; 3) money changes hands. Many people forgot that you need money to buy things, the shop no longer owns the object after the sale, or that prices vary from shop to shop! Some people appear not to have been shopping before, in which case I apologize for an unfair question. One technical representation issue: it's not possible for payment to take place by transfer of $price(y)$ from one person to the other; true, some dollars change hands, but the price of the object can't be *those particular dollars*; conversely, if the price is a number rather than some particular dollars, then it makes no sense to say that the shop now owns that number.

Here's a straightforwrd way to do it:
$Price(x, y, p)$ means x charges price p for object y

$Owns(x, y, s)$ means x owns y in situation s
$Funds(x, m, s)$ means x has funds m in situation s.
$\forall xyzpmn\ Owns(z, y, s) \wedge Price(z, y, p) \wedge Funds(x, m, s) \wedge \geq (m, p) \wedge Funds(z, n, s) \Rightarrow$
$Owns(x, y, result(buy(x, y, z), s)) \wedge$
$Funds(x, -(m, p), result(buy(x, y, z), s)) \wedge$
$Funds(z, +(n, p), result(buy(x, y, z), s)) \wedge$
$\neg Owns(z, y, result(buy(x, y, z), s))$
We need frame axioms to make sure that other people's funds, and ownership of all other objects, are unchanged, along with any other situation-dependent predicates there might be.

(b) (4) Generally speaking, not much to choose between them since 1) blocks world actions are reversible 2) the branching factor is therefore about the same in both directions. This might not be the case for an incompletely specified goal, though.

6. (12 pts.) Games against nature

This question was generally answered pretty well. The answers to part d) were especially good, considering that this is a current research issue. Most people in the class seem to have more sense than most AI researchers.

(a) (3) Solution plan: in the original solution, each step was a single action. Since the only form of failure has no side effects, we can keep trying again and will eventually succeed. So each step becomes a "loop until success" for the same action. (On average, it will take $1/(1-p)$ tries to make a move.)

(b) (2) If we just treat each operator as a "loop until success", then because the average total solution cost is the sum of the average step costs, the same algorithm will apply provided we just multiply the g and h costs by the appropriate amount. In fact, even if we left them untouched the solution returned would be the same.

(c) (3) If each action fails with probability jp ($jp < 1$), where j is the number on the tile being moved, then we need to recognize that some moves are more expensive. Each step cost going into the g cost is multiplied by some factor (in fact, $1/(1-jp)$) and the h estimates for each tile are multiplied by the same factor (since the total expected cost to move 3 squares, say, is 3 times the total expected cost to move one square).

(d) (4) If actions sometimes "mess up" by moving some other adjacent tile into the empty square, then we are in a real bind. Now we can't just keep trying until success, because we might mess up the pattern totally. Since we can observe action outcomes, we can tell when an undesired event has occurred, and try to undo it. This means our plan has to have a repair action inserted for every step, conditional on the various possible failures. Unfortunately, the repairs can fail too, leading us even further away from the "mainline" solution path, so we need repairs for repairs ad infinitum. If failures are sufficiently unlikely, we could terminate this regress at some point, leaving us with a conditional solution plan that succeeds with high probability.

This process is pretty ugly, on the whole, because we have to anticipate so many events that may never actually occur and each has to be dealt with in its own way. The clean way to think about it is the following. Since, in principle, a sequence of unexpected outcomes can lead us to any state in the state space, we could just calculate, for each state, what is the right thing to try (the most likely to get us to a solution quickly) and store it. *Dynamic programming* techniques are designed for this kind of problem; for example, we can build up such a table by starting from the goal state and working back.

A more sophisticated approach is to put *planning steps* into the plan itself. Thus, we could have a main line plan just as in part a), and conditional branches so that if unexpected results occur, the search mechanism is reinvoked to solve the new problem that arises. This gives us a plan that's guaranteed to work and doesn't require considering possibilities that don't actually arise. Obviously, we could combine this mechanism with the ideas in the previous paragraph in various degrees, depending on the failure probabilities.