

THE UNIVERSITY OF HONG KONG

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

CSIS0270 Artificial Intelligence

Date: May 12, 2004

Time: 9:30am–12:30 pm

Candidates may use any calculator which fulfils the following criteria: (a) it should be self-contained, silent, battery-operated and pocket-sized; (b) it should have numeral-display facilities only and should be used only for the purpose of calculation; (c) it should not have any printing device, alphanumeric keyboard, or graphic display; and (d) it should not contain any recorded data or program. It is the candidate's responsibility to ensure that the calculator operates satisfactorily and the candidate must record the name and type of the calculator on the front page of the examination scripts. Lists of permitted/prohibited calculators will not be made available to candidates for reference, and the onus will be on the candidate to ensure that the calculator used will not be in violation of the criteria listed above.

Answer all questions.

1. (*Searching, 15%*) To perform a state space search in a general graph where all action costs are 1, a student suggests using IDS, with “never go to any visited state” as the strategy for avoiding repeated states. The pseudo-code looks like this:

```
def IDS(init_state):
    i = 0
    while True:
        closed = set()           # create a new, global set
        result = DFS_l(init_state, i)
        if result != failed:
            return result
        i += 1
def DFS_l(state, limit):
    if GoalState(state):
        return state
    if limit == 0 or (state in closed):
        return failed
    closed.insert(state)
    for new_state in Successors(state):
        sln = DFS_l(new_state, limit - 1)
        if (sln != failed):
            return sln
    return failed
```

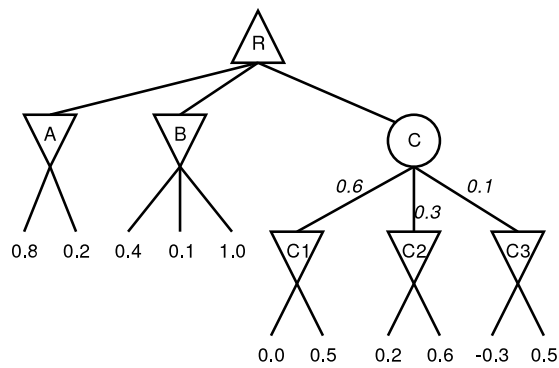
- a. (5%) The algorithm is not optimal. Give an example to show the problem.
- b. (3%) Assume the tree is finite. Is the algorithm complete? Explain your answer.
- c. (7%) Modify the algorithm above slightly so that it is both optimal and complete. Is the algorithm better or worse than BFS with the same strategy to deal with repeated states?

2. (CSP, 18%) We want to help a busy businessman to make schedules of meetings. Each meeting is to be scheduled to a single contiguous slot. Each meeting has a duration, and is restricted to be held only during certain time. Also, it is a policy to schedule meeting only at hour boundaries (e.g., we will not start a meeting at 2:30pm, only at 2pm or 3pm). As an example, he might want to schedule the following meetings (for simplicity we consider meetings of the same week):

Meeting	Duration	Restriction: Meeting can be held only during...
1	3 hours	Monday 11am–4pm
2	4 hours	Tue 8am–noon or 2pm–6pm
3	2 hours	Tue 9am–1pm or Wed noon–2pm
4	3 hours	Wed 10am–2pm
5	1 hour	Monday 10am–2pm
6	3 hours	Monday 11am–5pm or Tuesday noon–3pm
7	2 hours	Tue or Wed, 8am–noon
8	4 hours	Wed or Thu, noon–5pm

We can view the problem as a constraint satisfaction problem: the variables are the times at which each meeting starts; their domains are the times that do not cause the restrictions to be violated, and the constraints are that time of different meetings should not overlap.

- a. (4%) Draw the CSP corresponding to the above problem, taking care that constraints should be drawn only for meetings that can possibly overlap according to the restrictions. Also explain why this precaution is taken.
 - b. (4%) Solve the problem using the Most Restricted Variable heuristic and the Degree heuristic with Forward Checking. Ties are broken by always choosing a variable that corresponds to the smallest meeting number. Values are chosen in fixed order of time: earlier meeting time is always tried first.
 - c. (3%) Suggest how the Least Constraining Value heuristic can be used in this setting, with an example of its application in one of the nodes above.
 - d. (3%) Apart from getting a solution faster, the solution given when the Least Constraining Value heuristic is applied is usually preferable. Suggest a reason.
 - e. (4%) What addition to the CSP can capture a restriction that “during each day the businessman do not want to have more than 3 meetings”?
3. (Two-player games, 12%) In a game, all evaluations are known to be within the range $[-1, 1]$. The MAX player has to make a decision, based on the following game tree (numbers in italic are the probability taking each branch):



- a. (2%) Use expecti-minimax to compute the game value at the top of the tree, and hence give the best action. Show all your steps.
 - b. (4%) With the Alpha-Beta algorithm, will pruning happen in one of the branches of B, and if so, when? Explain.
 - c. (6%) Trace the Alpha-Beta algorithm when processing the branch leading to the node C. Show how alpha and beta change, and hence show when pruning happens.
4. (Logic, 10%) Consider the propositional logic sentence $P: (A \Rightarrow B) \Rightarrow C$.
- a. (2%) Convert the sentence into CNF. Show all steps.
 - b. (2%) Negate your result in part (a), and convert the resulting sentences into CNF. Show all steps.
 - c. (3%) Compare the result with the result you get if you turn $\neg P$ into CNF directly. The latter is much simpler. There is a rule that allows you to simplify the sentence you get in part (b) to the simpler form. Name the rule, and explain it.
 - d. (3%) In propositional logic it is easy to negate a sentence already in CNF. But in predicate calculus it is basically impossible. Explain why.
5. (Prolog, 18%) Prolog provides lists as the primary data structure, although sometimes we want something more than just a plain list, e.g., a First-In-First-Out queue.
- a. (4%) It is possible, although not particularly efficient, to implement a queue as a list. So an empty queue is represented as `[]`, a queue after pushing 1, 2 and 3 to the queue in that order would become `[1, 2, 3]`. With this implementation, write three predicates `newqueue/1`, `enqueue/3` and `dequeue/3`, so that `newqueue(Q)` will bind a new queue to `Q`, `enqueue(Q, L, NewQ)` will add all elements in `L` to the queue `Q` and bind the result to `NewQ`, and `dequeue(Q, X, NewQ)` will dequeue one element in `Q`, bind it to `X`, and bind the remainder of the queue to `NewQ`. Dequeue fail (i.e., give “no solution”) if there is no element in the queue. E.g.,


```

newqueue(Q) => Q = []
enqueue([2], [1,2,3], Q1) => Q1 = [2,1,2,3]
dequeue([2,1,2,3], X, Q1) => X = 2, Q1 = [1,2,3]
dequeue([], X, Q1) => Fail.

```

 Built-in predicates are allowed.
 - b. (6%) A more efficient way to represent the queue is called a “difference set”: the queue

is represented as a structure `queue(Q, V)`. `Q` is a partial list: the last element of the queue is a variable `V`, and the elements before it are the elements in the queue. So an empty list is represented as `queue(V, V)`, while a queue with three elements 1, 2 and 3 is represented as `queue([1, 2, 3|V], V)`. E.g.,

```

newqueue(Q) ⇒ Q = queue(A, A)
enqueue(queue([2|A], A), [1, 2, 3], Q1) ⇒ Q1 = queue([2, 1, 2, 3|B], B)
dequeue([2, 1, 2, 3|B], X, Q1) ⇒ X = 2, Q1 = queue([1, 2, 3|B], B)
dequeue(queue(A, A), X, Q1) ⇒ Fail.

```

This allows all the three predicates above to be implemented in $O(1)$ time. In particular, we can enqueue by simply binding something to the variable `V`. Write these predicates. Again, built-in predicates are allowed.

- c. (8%) Using the above three operations, write a breadth-first-search predicate `bfs/2`: `bfs(InitState, GoalState)` should start the BFS from `InitState`, find a goal state, and bind it to `GoalState`. If there are multiple answers, it should be possible to continue the search using continuations. You only need the goal state, not the full path to it. Your predicate should work with either representation of queues above. (Hint: you will need an auxiliary predicate.)

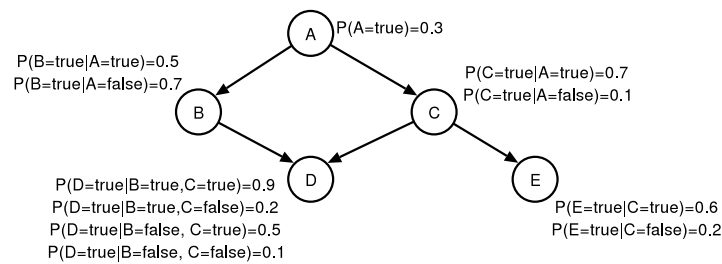
You may assume the existence of the following predicates: `successors/2` finds the successors of a state—`successors(State, L)` will bind `L` to all successors of `L`; `goal/1` checks for goal—`goal(X)` is true if and only if `X` is a goal state.

6. (*Planning*) Consider the following planning problem, expressed in STRIPS form, with the convention that `X` and `NotX` are opposing literal (where if we don't know about either, both will be false):

- Initial state: $P1 \wedge P2$.
- Action A1: Precondition $P1 \wedge P2$, Effects $\neg P1, \neg NotP3, NotP1, P3$.
- Action A2: Precondition $P2$, Effects $\neg NotP1, \neg P2, P1, NotP2$.
- Action A3: Precondition $P3 \wedge NotP2$, Effects $\neg NotP2, \neg P3, P2, NotP3$.
- Goal state: $NotP2 \wedge NotP3$.

- a. (3%) Give a fully-ordered plan for the problem. Show all the intermediate states.
- b. (6%) Suppose the POP algorithm is used to find a solution of the above problem in the plan space. Give the shortest solution found, making annotations about ordering constraints and causal links, and describes how each conflict is resolved.
- c. (6%) Draw the serial planning graph for the above problem from level 0 up to level 2. Show all the mutexes.

7. (*Probabilistic reasoning, 12%*) Consider the following belief network:



- a. (3%) By finding the full joint probability distribution from the belief network, find the conditional probability that B is true, given D and E are true.
- b. (3%) Suppose we know that E is true with perfect certainty, but for D we can only be 80% certain (the other 20% is that D is false). Describe how you can compute the conditional probability in step (a).
- c. (3%) The belief network is not a tree. Draw an equivalent network which is a tree, by clustering.
- d. (3%) Using the resulting tree, apply the variable elimination algorithm to derive the conditional probability you've found in part (a) again.

END OF PAPER