EE3-16 Artificial Intelligence Solutions

The Answers

1

a) Application, worked example

Transformer is a 2-tuple, atom and list of nodes it supplies. Subscriber is a term.

```
grid( [ (station, [
                (t1, [
                        (t4, [c1,c2,c3]),
                        (t5, [c4,c5])
                    ]),
                (t2, [c6,c7]),
                (t3, [
                        (t6, [c8,c9]),
                        c10
                    ])
        ])
      ]
    ).
```

[6]

b) Application

```
supplies( X, Y, Grid ) :-
        %get the subgrid rooted on X
        sub_grid( X, Grid, Subgrid ),
        %work out if Y is in that subgrid
        is_supplied_in( Y, Subgrid ).

sub_grid( X, [(X,SG)|_], SG ) :- !.
sub_grid( X, [(_,Xsg)|_], SG ) :-
        sub_grid( X, Xsg, SG ).
sub_grid( X, [_|T], SG ) :-
        sub_grid( X, T, SG ).

is_supplied_in( Y, Subgrid ) :-
        subscriber( Y ),
        member( Y, Subgrid ), !.
is_supplied_in( Y, Subgrid ) :-
        transformer( Y ),
        member( (Y,_), Subgrid ).
is_supplied_in( Y, [(_,SG)|_] ) :-
        is_supplied_in( Y, SG ), !.
is_supplied_in( Y, [_|T] ) :-
        is_supplied_in( Y, T ).

transformer( t1 ). %etc
subscriber( s1 ). %etc
```

[10]

c) Application

```
reconnect( X, [(T,SG)|Grid], [(T,Flat)|Grid] ) :-
        append( Fr, [(X,G)|Ba], SG ), !,
        append( Fr, G, Temp ),
        append( Temp, Ba, Flat ).

reconnect( X, [(T,G1)|Grid], [(T,G2)|Grid] ) :-
        reconnect( X, G1, G2 ), !.

reconnect( X, [G|Grid1], [G|Grid2] ) :-
        reconnect( X, Grid1, Grid2 ).
```

[4]

2

a) Bookwork

depth first: expand node at deepest level, backtrack to next deepest if no expansion
breadth first: expand all nodes at level d before any at level d+1
iddf: depth-first search at successive depth limits

b branching factor
d depth of solution
m max depth of tree

depth not sound, not complete, complexity space b^d time b*m
breadth sound, complete, complexity space b^d time b^d
iddf sound, complete, complexity space b^d time b*d

[6]

b) Application

(i) connected(node, [list of nodes])

(ii) List of length k specifying the nodes with a pebble will do. This assumes the pebbles are indistinguishable.

Example, and graph with node labels and a pebble at some nodes

(iii) Specify, in Prolog, the state transformer for the move operation.

```
statechange( move, Current, New ) :-
        append( Fr, [A|Ba], Current ),
        is_connected( A, B ),
        \+ member( B, Current ),
        append( Fr, [B|Ba], New ).

is_connected( A, B ) :-
        connected( A, L ),
        member( B, L ).

connected( 0, [1, 2, 3] ). %etc
```

(iv) append looks at each node with a pebble in turn
member generates each edge in turn.

[10]

c) List of 2-tuples, each tuple a node label and a number of pebbles at that node.

```
statechange( move, Current, New ) :-
        append( Fr, [(N,P)|Ba], Current ),
        P > 0,
        Pnew is P - 1,
        is_connected( N, M ),
        append( Fr, [(N,Pnew)|Ba], Temp ),
        append( Fx, [(M,Q)|Bx], Temp ),
        Qnew is Q + 1,
        limit( M, L ),
        Qnew < L,
        append( Fx, [(M,Qnew)|Bx], New ).
```

[4]

3

a) Bookwork

Choose to expand the node n with lowest f-cost given by actual cost of path from start state S to n as calculated by some cost function g, plus estimated cost of path from n to nearest goal state G.

Let f* be the actual cost of getting from S to G. A* expands all nodes for which f(n)<f*, some nodes for which f(n)=f*, including G, before it expands any nodes for which f(n)>f*.

This means that f* is optimal, complete, but still exponential in the number of nodes, although it is possible to get better results for certain types of heuristic.

[3]

b) Bookwork

heuristic, function which estimates cost, condition h(G)=0 for any goal state
admissible heuristic, never overestimates the actual cost

Importance of admissibility:

If we always under-estimate, at some point we *must* expand the nodes on the solution path ending in G, including G, before *at least one* of the nodes on a solution path ending in a sub-optimal goal G' (the 'at least one' may be G' itself, but it is enough).

[3]

c) Bookwork, application

Optimality:

Optimal solution has cost f* to get to optimal goal G
Suppose A* search returns path to sub-optimal goal G'
We show that this is impossible

$$
\begin{aligned}
f(G') &= g(G') + h(G') \\
&= g(G') + 0 \qquad \text{G' is a goal state, we require h to be 0} \\
&= g(G')
\end{aligned}
$$

If G' is sub-optimal then $g(G') > f^*$
Now consider a node n on path to optimal solution G

```
   |-           n0 (start)
   |            /  \
   |           /    \
 f*|         n       G'
   |        /
   |       /
   |-   G
```

Then:	f*	$\geq$	f(n)	monotonicity
	f(n)	$\geq$	f(G')	otherwise A* expands n first
	f*	$\geq$	f(G')	transitivity of $\geq$
	f*	$\geq$	g(G')	a contradiction

So either G' was optimal or A* does not return a sub-optimal solution.

Completeness:
A* expands nodes in order of increasing f-cost
Each expansion has lower bound > 0
So A* must eventually expand all nodes n with f(n) less than or equal to f*, one of which must be a goal state
(unless there are an infinite number of nodes with f(n) < f*, or infinite number of nodes with finite total cost

[5]

d) Bookwork, application
Let f* be cost of optimal node.
A* expands all nodes with f-cost less than f*.
A* expands some nodes with f-cost = f*.
A* expands no nodes with f-cost > f*.
Since f(n) = g(n) + h(n), this means that A* expands all those nodes such that h(n) < f* - g(n).
In other words, the more nodes for which this relation holds, the more nodes will be expanded by A* using this heuristic, and the less efficiently will the search space be explored.
Alternatively, consider histogram of nodes according to actual f-cost, whereby f-actual(n) = g(n) + h-actual(n).

```
|                 |     *    *
|   *             *  *     *    *
| *  *     *      G  *  *  *    *
| *  *  *  *  *   *  *  *  *  *  *
___________________________________
                 f*
```

[5]

e) Understanding and Application
Minimax: exhaustive search; alpha-beta depth-first search to fixed ply
Minimax: assign leaves win (1) or lose (0): alpha-beta: heuristic evaluation of quality
Minimax: full tree; alpha-beta, pruning (can effectively double 'lookahead').

Alpha beta, unless there is a forced-win and the search space is exhaustive, in which case the winner is whoever moves first and if it is minimax, there is nothing alphabeta can do about it.

[4]

4

a) Bookwork

resolution: is a single valid inference rule that produces a new clause implied by two clauses containing complementary literals

unification: Unification is a process of attempting to identify two symbolic expressions by the matching of terms and the replacement of certain sub-expressions (variables) by other expressions

[3]

b) Bookwork

skolemisation: eliminating existential quantifiers leaving an equisatisfiable formula by replacing existentially quantified variables by skolem constants and functions

[3]

c) Application

$\forall x.\forall y.\ coyote(x) \wedge roadrunner(y) \rightarrow chases(x,y)$
$\forall x.\forall y.\ roadrunner(x) \wedge saysbeepbeep(x) \rightarrow smart(x,y)$
$\forall x.\forall y.\ coyote(x) \wedge smart(y) \rightarrow avoids(y,x)$
$\forall x.\forall y.\ chases(x,y) \wedge avoids(y,x) \rightarrow frustrated(x)$

$\neg coyote(x1) \vee \neg roadrunner(y1) \vee chases(x1,y1)$
$\neg roadrunner(x2) \vee \neg saysbeepbeep(x2) \vee smart(x2,y2)$
$\neg coyote(x3) \vee \neg smart(y3) \vee avoids(y3,x3)$
$\neg chases(x4,y4) \vee \neg avoids(y4,x4) \vee frustrated(x4)$

$coyote(wilee)$
$roadrunner(rex)$

[4]

d) Application:

$\neg saysbeepbeep(rex) \rightarrow frustrated(wilee)$
$\neg saysbeepbeep(rex) \vee frustrated(wilee)$
$\neg(\neg saysbeepbeep(rex) \vee frustrated(wilee))$ *negated conclusion*
$\neg\neg saysbeepbeep(rex) \wedge \neg frustrated(wilee)$
$saysbeepbeep(rex) \wedge \neg frustrated(wilee)$
$saysbeepbeep(rex) \wedge \neg chases(wilee,y4) \vee \neg avoids(y4,wilee)$ $\{x4=wilee\}$
$saysbeepbeep(rex) \wedge \neg coyote(wilee) \vee \neg roadrunner(y1) \vee \neg avoids(y1,wilee)$
$\{x1=wilee,\ y4=y1\}$
$saysbeepbeep(rex) \wedge \neg roadrunner(y1) \vee \neg avoids(y1,wilee)$
$saysbeepbeep(rex) \wedge \neg avoids(rex,wilee)$ $\{y1=rex\}$
$saysbeepbeep(rex) \wedge \neg coyote(wilee) \vee \neg smart(rex)$ $\{x3=wilee,\ y3=rex\}$
$saysbeepbeep(rex) \wedge \neg smart(rex)$
$saysbeepbeep(rex) \wedge \neg roadrunner(rex) \vee \neg saysbeepbeep(rex)$ $\{x2=rex\}$
$saysbeepbeep(rex) \wedge \neg saysbeepbeep(rex)$
upside¬down¬T

[4]

e) Application

Prolog program is set of horn clauses

One positive literal is a fact

Disjunction of negated literals is a goal (query)
Disjunction of negated literals and a single positive literal is a clause.

Prolog can get stuck in infinite loops. Prolog inference is sound but not complete.
Trade efficiency for effectiveness

[4]

5

a) Application

1	$\neg((p\vee q) \to (p\vee r)) \to (p \vee (q\to r))$	$\neg$*conc*
2	$(p\vee q) \to (p\vee r)$	*a*, 1
3	$\neg((p \vee (q\to r))$	*a*, 1
4	$\neg p$	*a*, 3
5	$\neg(q\to r)$	*a*, 3
6	q	*a*, 5
7	$\neg r$	*a*, 5

Branch 1

8	$p \vee q$	*PB*1
9	$p \vee r$	*b*, 2, 8
10	r	*b*, 9, 4

close, 7, 10

Branch 2

11	$\neg(p \vee q)$	*PB*2
12	$\neg p$	*a*, 11
13	$\neg q$	*a*, 11

close 6, 13

[4]

b) Application

1	$q\to r$	*premise*
2	$r\to(p\wedge q)$	*premise*
3	$p\to(q\vee r)$	*premise*
4	$\neg(p\leftrightarrow q)$	$\neg$ *conclusion*

Branch 1

5	p	*PB*1
6	$\neg q$	*e*, 4, 5
7	$q \vee r$	*b*, 3, 5
8	r	*b*, 6, 7
9	$p \wedge q$	*b*, 2, 8
10	p	*a*, 9
11	q	*a*, 10

close 6, 11

Branch 2

12	$\neg p$	*PB*2
13	q	*e*, 4, 12
14	r	*b*, 1, 4
15	$p \wedge q$	*b*, 2, 14
16	p	*a*, 15
17	q	*a*, 15

close 12, 16

[4]

c) Application

(i)

$m \vee a \to j$
$\neg m \to a$
$a \to \neg j$

(ii) There are eight possibilities

$\neg m, \neg a, \neg j$
$\neg m, \neg a, j$
$\neg m, a, \neg j$
$\neg m, a, j$
$m, \neg a, \neg j$

$m, \neg a, j$
$m, a, \neg j$
m, a, j

One application of beta rules out
$\neg m, \neg a, \neg j$
$\neg m, \neg a, j$
$\neg m, a, j$
m, a, j

$m, a, \neg j$
branch on $m \vee a$, *branch* 1 *gives* $\neg j$ *close*
branch 2 $\neg(m \vee a)$, $\neg m$, $\neg a$, *close*

$\neg m, a, \neg j$
$\neg(m \vee a)$ *by beta on pr*1 *and* j
gives $\neg m$, $\neg a$, *close*

$m, \neg a, \neg j$
$\neg(m \vee a)$ *by beta on pr*1 *and* j
gives $\neg m$, $\neg a$, *close*

$m, \neg a, j$
beta simplifies pr1 with j
beta simplifies pr2 with m
beta simplifies pr3 with ¬a
all clause analysed, so mary comes, john comes, anne stays at home

[6]

d) Application:

$$1: \neg((\Box p \rightarrow \Diamond p) \leftrightarrow (\neg\Box p \vee \neg\Box\neg p))$$

branch 1

$1: (\Box p \rightarrow \Diamond p)$
$1: \neg(\neg\Box p \vee \neg\Box\neg p)$
$1: \neg\neg\Box p$
$1: \neg\neg\Box\neg p$
$1: \Box p$
$1: \Box\neg p$
$2: p$
$2: \neg p$
close

[2]

branch 2

$1: \neg(\Box p \rightarrow \Diamond p)$
$1: (\neg\Box p \vee \neg\Box\neg p)$
$1: \Box p$
$1: \neg\Diamond p$
$2: p$
$2: \neg p$
close

$[1]: \neg((\Box p \rightarrow \Diamond p) \leftrightarrow (\neg\Box p \vee \neg\Box\neg p))$

branch 1

$[1]: (\Box p \rightarrow \Diamond p)$ *PB*1
$[1]: \neg(\neg\Box p \vee \neg\Box\neg p)$
$[1]: \neg\neg\Box p$
$[1]: \neg\neg\Box\neg p$
$[1]: \Box p$
$[1]: \Box\neg p$
$[1]: \Diamond p$ — *beta on PB*1 *and* $\Box p$
$[1,1]: p$ — $\Diamond$
$[1,1]: \neg p$
close

branch 2

$[1]: \neg(\Box p \rightarrow \Diamond p)$
$[1]: (\neg\Box p \vee \neg\Box\neg p)$
$[1]: \Box p$
$[1]: \neg\Diamond p$

branch 2.1
$[1]: \neg\Box p$
$[1,1]: \neg p$
$[1,1]: p$
close

branch 2.1
$[1]: \Box p$
$[1]: \neg\Box\neg p$
$[1,2]: p$
$[1,2]: \neg p$
close

[6]