E3.16
ISE3.23

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING
EXAMINATIONS 2005

EEE/ISE PART III/IV: MEng, BEng and ACGI

Corrected Copy

ARTIFICIAL INTELLIGENCE

Thursday, 5 May 10:00 am

Time allowed: 3:00 hours

There are SIX questions on this paper.

Answer FOUR questions.

All questions carry equal marks

Any special instructions for invigilators and information for candidates are on page 1.

Examiners responsible	First Marker(s) :	J.V. Pitt
	Second Marker(s) :	M.P. Shanahan

1 Consider the simple Blocks World illustrated in Figure 1.

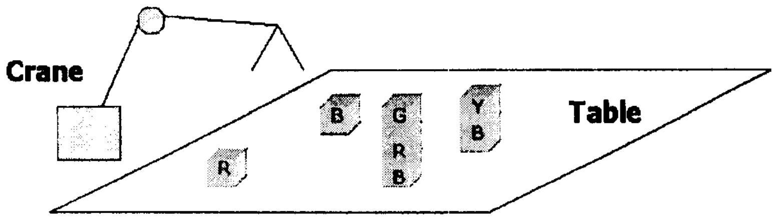


Figure 1: Blocks World

The domain consists of a 'large' number of blocks of four differently coloured blocks (red (R), blue (B), green (G) and yellow (Y)), scattered randomly over a surface called the table. Some blocks may already be on other blocks. There is a crane which can be used to move blocks around. The problem is to build towers (stacks of 2 or more blocks, one on top of each other). Where the towers are with respect to each other on the table is not important.

a) Give a representation of a state of the problem in Prolog or other declarative notation. Justify why this representation is a 'reasonable' formulation of the problem domain. [4]

b) Define the initial state, in Prolog or other declarative notation, as shown in Figure 1.
Define state change rules for the operations stack (put a block on top of a tower), unstack (take a block from the top of a tower and put it on the table), and move (move one block from the top of one tower to the top of another). [4]

c) Using Prolog or other declarative notation, define a predicate for extracting just the towers from a state (i.e., discarding the single blocks on the table). Then, using Prolog or other declarative notation, define the final state, if the towers in the final state must satisfy:

i. Every tower must contain at least 3 blocks.

ii. At least one tower must have all different colours.

iii. More than half the towers must be composed of blocks of just one colour.

The predicates length and member, with the obvious functionality, are given. [8]

d) Briefly describe the Uniform Cost search algorithm.
Suppose there was a cost associated with stacking blocks on one colour next to blocks of another colour. How would the state representation be changed? How would the stack state change rule be changed?
What would be the effect on the algorithm if the unstack operation deducted the cost from the total cost of the solution? [4]

2 a) In the context of A* search, define the terms admissible heuristic, monotonic function, and the pathmax equation. Explain the importance of these terms in A* search.

[5]

b) Evaluate A* search in terms of the criteria optimality, completeness and complexity.
Justify the evaluation of optimality and completeness.

[8]

c) Give two admissible heuristics for solving the 8-puzzle, and explain why they are admissible.
In general, given two admissible heuristics for an A* search, explain why one may be more 'efficient' ('more informed') than the other.
Briefly comment on ways to make heuristics 'more informed' for A* search.

[7]

3 a) Define four properties of a search space that can make systematic search difficult.

[4]

b) Explain, using an example, how a search space can be represented by a graph.

[4]

c) Give an explicit definition of the paths in a graph based on the graph representation given in part (b).

[2]

d) Give an inductive definition of the nodes, incidence relation, and paths in a graph, given one node and a set of operators. State the condition(s) that need to be satisfied for the inductive definition of the paths to be equivalent to the explicit definition given in part (c).

[6]

e) Explain how the General Graph Search Program computes the paths as defined in part (d), using depth first and breadth first algorithms.

[4]

4 a) Describe the Minimax Procedure for exhaustively searchable graphs representing two-player games.

[4]

b) The game of Noughts & Crosses is played on a 3 by 3 board, with each player taking it in turns to place either a 'o' or 'x' symbol. The winner is the first player to create a line of 3 'o', or 3 'x' symbols, vertically, horizontally, or diagonally.

Using Prolog or other declarative notation, give a state representation for the problem, define the goal state, the final state, and Max's state change rule for making a move.

[4]

c) Consider the state of the game shown in Figure 2.

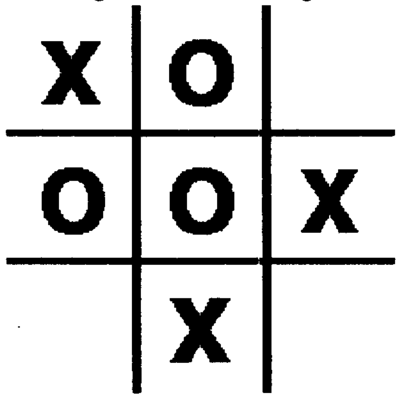


Figure 2: Noughts & Crosses Game

Assuming it is Max to move and Max is playing 'x', show the search space and the state space values assigned by the Minimax Procedure.
State any assumptions that have been made, and say why this might be important in game playing.

[4]

d) If exhaustive search is not feasible, describe two alternative strategies for determining the next best move.

[4]

e) Give an explicit rule-based strategy for playing the game of Noughts & Crosses, and describe it (at the top level) using Prolog or other declarative notation. Briefly comment on the advantages and/or disadvantages of exhaustive search compared to rule-based strategies for playing games.

[4]

5. a) Define *conceptualisation*, explain how it is used in knowledge representation, and indicate some of the issues involved.

[3]

b) Define *unification* and explain why it is important in theorem proving with resolution.

[3]

c) Consider the following statements:

Every canary can fly.
Every penguin can swim.
All birds are either canaries or penguins.
Opus is a bird.

i. Formalise these statements in first-order logic.

ii. Convert these statements into conjunctive normal form, and eliminate the quantifiers. Explain why this is not a set of horn clauses.

iii. Prove, using resolution, that opus can fly or swim.

[11]

d) Define *skolemisation*, explain how it is used in converting first-order logic into horn clauses, and indicate why horn clauses are computationally useful.

[3]

6. a) Explain how the KE proof procedure can be used to determine the entailment relation between a formula and a set of formulas. Comment on the issues of soundness, completeness and decidability with respect to the KE proof procedure for propositional logic.

[7]

b) Show, using the KE proof procedure, that $((p \rightarrow q) \rightarrow p) \rightarrow p$ is an axiom of propositional logic.

[1]

c) There are three boxes. One (and only one) box contains gold, the other two boxes are empty. Formalise this statement in propositional logic.

[3]

d) Each box from part (c) has a message printed on it, as follows. On Box 1: "The gold is not in Box 1". On Box 2: "The gold is not in Box 2". On Box 3: "The gold is in Box 2".
One (and only one) of these statements is true. Formalise these statements in propositional logic, and use the fact that only one is true to reduce the formula to its simplest form.

[4]

e) Show, using the KE proof procedure, that the set of formulas given in part (c) and part (d) entails that the gold is in Box 1.
What would be the result of trying to show that it was in Box 2 or in Box 3?

[5]

OK as master

SOLUTIONS - ARTIFICIAL INTELLIGENCE 2005 E3.16 ISE 3.23

1

(a)

As a list of lists, each sub-list is a tower.

Requirement is a good abstraction, deciding what goes into the representation and what needs to be left out, deciding what effects of actions are significant and what are not. The only thing we are interested in is towers. Therefore it is not necessary to represent the table, the crane, the relative positions of the towers, and so on. Furthermore, we stack and unstuck from the top, and this can be represented by the head of the list, so the representation is computationally convenient.

(b)

```
[ [red, green], [red], [red,blue,yellow], ...] whatever

state_change( stack, OldBW, NewBW ) :-
        append( Front, [ [B] | Back ], OldBW ),        %pick a block on the table
        append( Front, Back, TempBW ),
        append( Front1, [Tower | Back1], TempBW ),        %pick a tower
        append( Front1, [ [B|Tower] | Back1], NewBW ).    %stack the block

state_change( unstack, OldBW, NewBW ) :-
        append( Front, [ [B | Tower] | Back ], OldBW ),    %pick a tower
        append( Front, [ [B], Tower | Back ], TempBW ).    %unstack the block

state_change( move, OldBW, NewBW ) :-
        append( Front, [ [B | Tower1] | Back ], OldBW ),   %pick a tower
        append( Front, Back, TempBW ),
        append( Front1, [Tower2 | Back1], TempBW ),        %pick another tower
        append( Front1, [ Tower1, [B|Tower2] | Back1], NewBW ).
```

(c)

```
just_towers( [H|T1], [H|T2] ) :-
        length( H, L ),
        L > 2, !,
        just_towers( T1, T2 ).
just_towers( [_|T1], [_|T2] ) :-
        just_towers( T1, T2 ).

at_least_3( [] ).
at_least_3( [H | T] ) :-
        length( H, L ),
        L > 2,
        at_least_3( T ).

all_different( [H | T ] ) :-
        all_diff( H ).
all_different( [ _ | T ] ) :-
        all_different( T ).

all_diff( T ) :-
        length( T, L ),
                L < 5,
                ad( T ).

ad( [] ).
ad( [ H | T ] ) :-
                \+ member( H, T ),
                ad( T ).

just_one_colour( L ) :-
                j1c( L, 0, OneCol ),
                length( L, Tot ),
                Half is Tot DIV 2,
                OneCol > Half.

j1c( [], N, N ).
j1c( [ H|T], N, N2 ) :-
                onecol( H ), !,
                N1 is N + 1,
                j1c( T, N1, N2 ).
j1c( [ _|T], N, N ) :-
                j1c( T, N, R ).

onecol( [H|T] ) :-
                samecol( H, T ).


samecol( H, [] ).
samecol( H, [H|T] ) :-
                samecol( H, T ).
```

(d)
Uniform cost associates a cost with each state change rules. Expands node with least cost on search frontier next.

Strictly speaking, state representation is not changed, but node representation now takes a cost as well as the state.

State change rule would have to include cost of applying rule using state change function g.

We would lose optimality (cost of successor not greater than cost of node because unstack makes it 'cheaper').

2
(a)
heuristic: function which estimates cost of getting from current state to goal state 1
admissible: never over-estimates

Monotonic function: f-cost never decreases along any path in search space 1

Pathmax equation: makes heuristic admissible 1

Uniform cost is optimal and complete but inefficient; best first is neither. Uniform cost gives cost of path from start node to n, best first gives estimated cost of cheapest path from n to goal node. But combining the two gives estimated cost of cheapest solution path through h, given slight restriction on heuristic function h. We can then prove that this strategy is optimally efficient, given a slight restriction in h. The slight restriction is admissibility. 2

(b)
optimal yes 1
complete yes 1
complexity: number of nodes expanded is still exponential in the length of the path, although it is possible to get sub-exponential growth for certain types of heuristic 1
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

$$\begin{aligned} f(G') &= g(G') + h(G') \\ &= g(G') + 0 \qquad \text{G' is a goal state, we require h to be 0} \\ &= g(G') \end{aligned}$$ 3

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*	≥	f(n)	monotonicity
f(n)	≥	f(G')	otherwise A* expands n first
f*	≥	f(G')	transitivity of _
f*	≥	g(G')	a contradiction

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

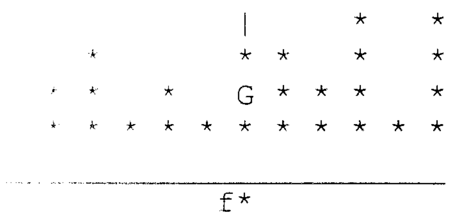
Completeness:
A* expands nodes in order of increasing f-cost
Each expansion has lower bound > 0 2
So A* must eventually expand all nodes n with f(n) less than or equal to f*, one of which must be a goal state
(unless there are an infinite number of nodes with f(n) < f*, or infinitie number of noides with finite total cost

3/12

(c)
H1 count number of tiles out of place (have to move this many tiles, maybe more) |
H2 count manhattan distance (have to move this distance, maybe more) |

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).

2



Only those nodes to the left of the f* bar will be expanded. In practice of course, we don't have h-actual we just have h. Thus we could have a redistribution of the histogram which pushes more of the nodes to the left of the f* bar.
In other words, we want to ensure f* < f(n) + h(n) < f-actual(n)
With the two heuristics given, the second is more informed.

making heuristics more accurate or efficient, i.e. more informed

- try to reduce the *effective branching factor b**
  - *b**: the branching factor that a uniform tree of depth *d* needs for *N* nodes
    $N = 1 + b* + (b*)^2 + (b*)^3 + \ldots + (b*)^d$
  - the best heuristics will minimize *b** (search is a straight line if *b** is *1*...)
- make *h(n)* as large as possible *without over-estimating*
  - e.g. by combining heuristics 2
  - but don't forget to consider the cost of computing the heuristic(s)...
  - e.g. by inventing heuristics
    - relax the problem
      - express logically and drop the conditions

4 | 12

3
(a)
size: get very big very quickly
have local maxima
may have loops
may have infinite paths
etc

(b)
Problem can be considered as set of states. Each state is node of a graph. States can be transformed one into another. States related by such transformation are encoded in incidence relation R. We're not interested in edges. Therefore search space can be represented as G = <N, R> where n is the set of nodes and R is the incidence relation.

Example can be anything reasonable…

(c)
$P_G = \bigcup_{i=0}^{\infty} P_i$
where
$P_0 = \{ \langle start \rangle \}$
$P_{i+1} = \{ p_i ++ \langle n_{i+1} \rangle \mid \exists\, p_i \in P_i .\ (frontier(p_i), n_{i+1}) \in R \}$

where ++ is append
frontier function gives last node in a sequence

(d)
G' = < start_node, Op > where
start_node is the root node
op is set of state transformers

$N_G = \bigcup_{i=0}^{\infty} N_i$
where
$N_0 = \{ \langle start \rangle \}$
$N_{i+1} = \{ n_{i+1} \mid \exists\, op \in Op .\ \exists\, n_i \in N_i .\ n_{i+1} = op(n_i) \}$

$R_G = \bigcup_{i=1}^{\infty} R_i$
where
$R_1 = \{ (n_0, n_1) \mid \exists\, op \in Op .\ n_1 = op(n_0) \}$
$R_{i+1} = \{ (n_i, n_{i+1}) \mid \exists\, op \in Op .\ \exists\, n_i \in N_i .\ n_{i+1} = op(n_i) \}$

$P'_G = \bigcup_{i=0}^{\infty} P'_i$
where
$P'_0 = \{ \langle start \rangle \}$
$P'_{i+1} = \{ p_i ++ \langle n_{i+1} \rangle \mid \exists\, op \in Op .\ \exists\, p_i \in P'_i .\ n_{i+1} = op(frontier(p_i)) \}$

For example:
$P'_0 = \{ \langle start \rangle \}$
$P'_1 = \{ \langle start,l1 \rangle, \langle start,l4 \rangle \}$
$P'_2 = \{ \langle start,l1,l2 \rangle, \langle start,l1,l3 \rangle, \langle start,l4,l5 \rangle, \langle start,l4,l6 \rangle, \}$

5/12

P'[illegible] = { <start,14,15,17>, <start,14,15,18>, }

i.e. the inductive definitions gives us the same set of paths as before provided the operators compute all elements of the incidence relation

(e)
Now using breadth first, the GGS creates all the paths in each P'_i before searching any of the paths in .

Using depth first search, it picks one path in the deepest P'_i, and computes all those members of P'_{i+1} which are one step extensions of that path, and carries on from there.

In this way, we can search a graph by computing it rather than exploring it if we were given the full explicit definition.

6|12

4
(a)
max is player trying to win, or MAXimize advantage

min is opponent who attempts to MINimize max's score. Assume that min uses the same information as max and attempts always to move to a state that is worst for max

each leaf node is given a score of 1 or 0, depending on whether the state is a win for max or min respectively

generate the graph

propagate leaf values up the graph according to the rules:

if the parent is a MAX, give it the minimum value of its children

if the parent is a MIN, give it the maximum value of its children

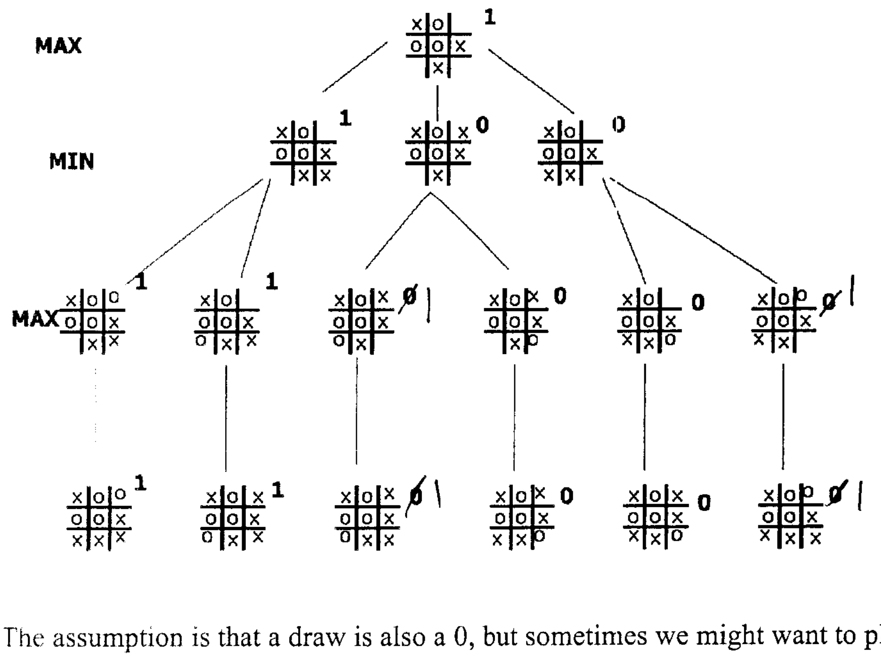
(b)
List of nine blanks, whose turn

([b,b,b,b,b,b,b,b,b], max)

([x,x,x|_], min)
([_,_,_,x,x,x|_], min etc)

```
state_change( move, (B, max), (NewB,min) ) :-
        append( Front, [b|Back], B ),
        append( Front, [x|Back], NewB ).
```

(c)



The assumption is that a draw is also a 0, but sometimes we might want to play for a draw if we cannot win,

7/12

(d)
use minimax to fixed ply and heuristics
use alphabeta search and prune tree

(e)

strategy is to take a win, stop opponent, take middle, take corner, go anywhere

in Prolog

```
suggest_move( Board, Move ) :-
        can_win( Board, Move ).
suggest_move( Board, Move ) :-
        stop_opponent( Board, Move ).
suggest_move( Board, Move ) :-
        middle_free( Board, Move ).
suggest_move( Board, Move ) :-
        corner_free( Board, Move ).
suggest_move( Board, Move ) :-
        whatevers_left( Board, Move ).
```

Can win involves looking for xxb, xbx or bxx combination in any of the 8 winning lines. Not expected to define these.

Difference comes in when more complicated games versus look ahead, etc. Anything sensible ill do.

8 | 12

5

(a)

conceptualisation is that part of knowledge representation concerned with ensuring that specific aspect of the domain are captured, with respect to key concepts, objects, and relations between them.

Usually defined as conceptual basis, relational basis, functional basis, determining what objects/concepts are involved, what relations hold between them, and what functional properties exist.

Issues include abstraction and granularity

(b)

Unification is algorithm for determining if two terms are equatable, a unifier beign a set of substitutions of values for variables which two terms identical.

Essential for resolution e.g. from p(X) V q(X) and ¬p(c), infer q(c), not q(X), which wooudl be unsound.

(c)

(i)

$\forall x.c(x) \rightarrow f(x)$
$\forall c.p(x) \rightarrow s(x)$
$\forall x.b(x) \rightarrow (c(x) \vee p(x))$
$b(o)$

(ii)

$\neg c(x) \vee f(x)$
$\neg p(x) \vee s(x)$
$\neg b(x) \vee c(x) \vee p(x)$
$b(o)$

Not horn clauses because third one contains more than one positive literal

(iii)

1	$\neg c(x) \vee f(x)$	
2	$\neg p(x) \vee s(x)$	
3	$\neg b(x) \vee c(x) \vee p(x)$	
4	$b(o)$	
5	$\neg(f(o) \vee s(o))$	
6	$\neg f(o)$	
7	$\neg s(o)$	
8	$\neg c(o)$	1, 6
9	$\neg b(o) \vee p(o)$	3, 8
10	$p(o)$	4, 9
11	$s(o)$	2, 10
contradiction		7, 11

(d)

skolemisation is the process of eliminating existential quantifiers from logical formulas.

9/12

$\forall x.\exists y.p(x,y)$ is converted into p(X,f(X)), where f is the skolem function which gives, for any X, the value y that made p(x, y) true.

Horn clause are essentially Prolog.

6

(a)

Want to show Pr |= f, where Pr is a set of formulas (premises), c a formula (conclusion), |= the entailment relation
Don't do this with truth tables (semantics), so do it with syntactic methods
So show Pr' |-KE f, where Pr' = ΛPr, i.e. the distributed-and over formulas in Pr
By deduction theorem this is same as showing |-KE Pr' -> f
Ke method is proof by refutation so start from –(Pr' -> f), which is the same as writing down all the premises and negating the conclusion, because the onlt way to make –(Pr' -> f) true is to make Pr' true and f false, and then we can eliminate all the & in Pr'
To do the proof by refutation in Ke we build a tree using the KE rules ...
Tree represents conjunction of formulas on each branch, and a disjunction of its branches
Effectively this is the DNF of the negated formula we are trying to prove.
If all the branches contain a contradiction, then the DNF is logically equivalent to false.
If the DNF is logically equivalent to false, then the fomula is always false, ie a contradiction, in which case its negation must be a tautology, ie always true, which is what we trying to prove.

Soundness; if we prove something, it is a theorem, i.e |- -> |=, KE is sound

Completeness: if there is something to prove, we can prove it, i.e. |= -> |-, KE is complete

Decidable, proof procedure stops saying yes if there is a proof, stops saying no otherwise.
Propositional logic is decidable.

(b)

1	$\neg(((p \to q) \to p) \to p)$	
2	$((p \to q) \to p$	a, 1
3	$\neg p$	a, 1
4	$(p \to q)$	a, 2
5	$\neg p$	a, 2
6	p	a, 4
7	$\neg q$	a, 4
	close	3, 6

(c)

p = "gold in box 1"
q = "gold in box 2"
r = "gold in box 3"

Then get: $p \leftrightarrow (\neg q \wedge \neg r)$

$q \leftrightarrow (\neg p \wedge \neg r)$
$r \leftrightarrow (\neg p \wedge \neg q)$

(d)

Using formulation above, this amounts to -p & -q & q
Only one of which is true, i.e

$(\neg p \wedge \neg\neg q \wedge \neg q) \vee (\neg\neg p \wedge \neg q \wedge \neg q) \vee (\neg\neg p \wedge \neg\neg q \wedge q)$

which reduces to

$(p \wedge q) \vee (p \wedge \neg q)$

Using KE, we get

1	$p \leftrightarrow (\neg q \wedge \neg r)$	*P*1
2	$q \leftrightarrow (\neg p \wedge \neg r)$	*P*2
3	$r \leftrightarrow (\neg p \wedge \neg q)$	*P*3
4	$(p \wedge q) \vee (p \wedge \neg q)$	*P*4
5	$\neg p$	negated conclusion
6	$(p \wedge q)$	*PB*
7	p	*a*, 6
8	q	*a*, 6
9	$(\neg q \wedge \neg r)$	*b*, 1, 7
10	$\neg q$	*a*, 9
11	$\neg r$	*a*9
	close	8, 10
12	$\neg(p \wedge q)$	*PB*
13	$(p \wedge \neg q)$	*b*, 4, 12
14	p	*a*, 13
14	$\neg q$	*q*, 13
	close	5, 14

With –q or –r instead of –p, we would not have been able to close the branch, we'd only have been able to derive p, -q and –r.