Question 1 (20 marks)

Briefly define the three main components of production systems.

1. The set of production rules.

This is a condition-action pair and designs a single chunk of problem-solving knowledge.

It contains two parts: the condition part of the rule is about when to apply this rule. The action part is about the action to take.

1. Working memory.

It contains a description of the current state of the world in a reasoning process. This description is a pattern that is matched against the condition part of a production to select appropriate problem-solving actions. Once the content in the working memory is matched with the conditional element of the rules, the action (or actions) associated with that condition may then be performed. The actions of production rules are specifically designed to alter the contents of working memory.

1. The recognize–act cycle.

Working memory will be initialized at the beginning of the problem description with a set of patterns. Then the system matches against the conditions of the production rules; this produces a subset of the production rules, (namely conflict set). The conditions of the rules must match the patterns in working memory. The productions in the conflict set are said to be enabled. One of the productions in the conflict set is then selected (conflict resolution) and the production is fired. To fire a rule, its action is performed, changing the contents of working memory. After the selected production is fired, the control cycle repeats with the modified working memory. Whole process terminates when the contents of working memory do not match any rule’s conditions

Question 2 (20 marks)

Contrast the use of a data-driven strategy versus a goal-driven strategy for the control of search in production systems.

When the current state of the world (describing by true statements) matches the CONDITIONs of the production rules. It will cause the action of the rule and further to create another true descriptor for the world is referred to as data-driven search.

In contrast, when the goal is matched against the ACTION part of the rules in the production rule set about the world and their CONDITIONs are then can be set up as subgoals to be shown to be “true”, this is referred to as goal-driven problem-solving.

According to the book, these two concepts can be also defined as the following:

Data-driven search begins with a problem description and infers new knowledge from the data. This is done by applying rules of inference, legal moves in a game, or other state-generating operations to the current description of the world and adding the results to that problem description. This process continues until a goal state is reached.

Goal-driven search begins with a goal and works backward to the facts of the problem to satisfy that goal. To implement this in a production system, the goal is placed in working memory and matched against the ACTIONs of the production rules. These ACTIONs are matched (by unification, for example) just as the CONDITIONs of the productions were matched in the data-driven reasoning. All production rules whose conclusions (ACTIONs) match the goal form the conflict set.

Question 3 (20 marks)

Briefly describe the blackboard architecture for problem solving.

Blackboard system is a control mechanism. Blackboard extends the production system. It can organize the production memory into separate modules. Each of them corresponse to a different set of production rules. Blackboard acts like an agent to integrate all of those modules (sunset of rules) and coordinate the corresponding actions.

According to the book, this also can be interpreted as below:

The blackboard architecture is a model of control that has been applied to these and other problems requiring the coordination of multiple processes or knowledge sources. A blackboard is a central global data base for the communication of independent asynchronous knowledge sources focusing on related aspects of a particular problem.

Question 4 (40 marks)

**Depth-First**

Depth-first. By default, Prolog use deep first, I also wrote another deep first solution in case this is not counted.

*Code：*

writelist([]).

writelist([H|T]):-

write(H), writelist(T).

empty\_stack([]).

stack(Top, Stack, [Top|Stack]).

member\_stack(Element, Stack):-

member(Element, Stack).

reverse\_print\_stack(S):-

empty\_stack(S).

reverse\_print\_stack(S):-

stack(E, Rest, S),

reverse\_print\_stack(Rest),

write(E), nl.

unsafe(state(X,Y,Y,C)):-

opp(X, Y).

unsafe(state(X,W,Y,Y)):-

opp(X, Y).

move(state(X,X,G,C), state(Y,Y,G,C)):-

opp(X,Y), not(unsafe(state(Y,Y,G,C))),

writelist(['try farmer takes wolf ',Y,Y,G,C]),nl.

move(state(X,W,X,C), state(Y,W,Y,C)):-

opp(X,Y), not(unsafe(state(Y,W,Y,C))),

writelist(['try farmer takes goat ',Y,W,Y,C]),nl.

move(state(X,W,G,X), state(Y,W,G,Y)):-

opp(X,Y), not(unsafe(state(Y,W,G,Y))),

writelist(['try farmer takes cabbage ',Y,W,G,Y]),nl.

move(state(X,W,G,C), state(Y,W,G,C)):-

opp(X,Y), not(unsafe(state(Y,W,G,C))),

writelist(['try farmer takes self ',Y,W,G,C]),nl.

move(state(F,W,G,C), state(F,W,G,C)):-

writelist([' BACKTRACK from: ',F,W,G,C]),nl,fail.

path(Goal, Goal, Been\_stack):-

nl, write('Solution Path is: '), nl,

reverse\_print\_stack(Been\_stack).

path(State, Goal, Been\_stack):-

move(State, Next\_state),

not(member\_stack(Next\_state, Been\_stack)),

stack(Next\_state, Been\_stack, New\_been\_stack),

path(Next\_state, Goal, New\_been\_stack).

opp(e,w).

opp(w,e).

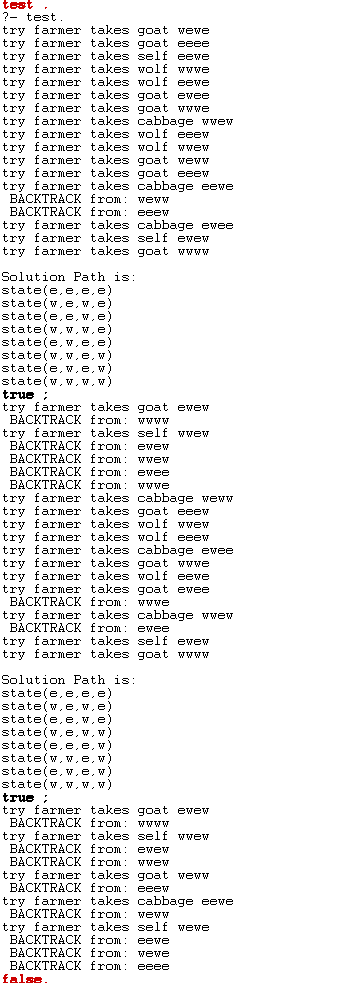
go(Start, Goal):-

empty\_stack(Empty\_been\_stack),

stack(Start, Empty\_been\_stack, Been\_stack),

path(Start, Goal, Been\_stack).

test:-go(state(e,e,e,e),state(w,w,w,w)).



Farmers, wolves, sheep, and cabbage cross the river. The state of the state (x, y, z, w) is the current location of the four objects (the east shore and the west shore of the river are represented by e and w respectively), move(state(x, y, z, w), state (u, v, w, g)) means moving from one side of the river to the other. Of course, the movement must meet the requirements. The state after the move must be in a safe state, with unsafe (state()) to detect whether the location is safe. Use unsafe (state()) to detect whether the location is safe.

And the farmer moves with the object, using go(start, goal) to indicate the state from the current state and the target state to be reached, path(Start, Goal, Been\_stack). Indicates one of the conditions that the moving path needs to satisfy.

**Breadth first**

In this part, I used lisp to perform both the breadth first and deep first search.

*Code：*

(defun turn-over-state (state &rest numbers)

(let ((new-state (copy-list state)))

(dolist (num numbers)

(setf (nth num new-state)

(if (eq (nth num new-state) 'w)

'e

'w)))

new-state))

(defun illegal-state (state)

(or

(and (eq (cadr state) (caddr state))

(not (eq (car state) (cadr state))))

(and (eq (cadr state) (cadddr state))

(not (eq (car state) (cadr state))))))

(defun find-legal-state (state)

(let (ans

(single (turn-over-state state 0)))

(if (not (illegal-state single))

(push single ans))

(loop for i from 1 to 3 do

(if (eq (nth i state) (car state))

(let ((new-state (turn-over-state state 0 i)))

(if (not (illegal-state new-state))

(push new-state ans)))))

ans))

(defun bfs (init-state)

(loop

with queue = (list (list init-state init-state))

while (not (null queue)) do

(loop for new-state in (find-legal-state (caar queue)) do

(if (equal new-state '(w w w w))

(format t "answer is ~a~%" (append (cdar queue) (list new-state))))

(if (not (find new-state (cdar queue) :test #'equal))

(setf queue

(append queue

(list (append (list new-state) (append (cdar queue) (list new-state))))))))

(pop queue)))

(defun dfs (state father-state-list)

(if (equal state '(w w w w))

(format t "answer is ~a~%" (append father-state-list (list state)))

(progn

(setf father-state-list

(append father-state-list (list state)))

(loop for new-state in (find-legal-state state) do

(if (not (find new-state father-state-list :test #'equal))

(dfs new-state father-state-list))))))

In lisp I also defined the legal state and illegal state just like safe state and un safe state as what I did within prolog. However, this time, I figured out that I could use queue to perform the bfs easily because of its first in first out mechanism. Note that, please define the initial state while executing the program (i.e. e e e e)

