New Mexico State University

CS 475, Artificial Intelligence I, Fall 1997

Midterm Exam

There are a total of 100 points. Please answer the questions on the separate sheets provided. Answer the specific question asked. Be specific, but do not go overboard.

1. (16 points) Suppose that you are designing a program to generate cross-word puzzles using state-space search. Outline a solution by describing the "goalp", "successors", and "h" functions you would define. Do not write any code or actual algorithms. However, your answer should be specific enough to show your understanding of the components of state-space search.

2. (6 points) Is the heuristic you defined in question 1 admissible? Please explain.

3. (16 points)

   Suppose there is no goal in the state space and the state space is finite. Which of the following searches, as they were defined in lecture, are guaranteed to eventually search the entire state space? For those that are **not**, please give a **brief** explanation.

   Depth-first search

   Breadth-first search

   Iterative deepening search

   Iterative widening search

   Uniform cost search

   Best-first search

   Beam search

   A* search

   IDA* search

   Hill climbing

4. (8 points) Imagine that you are supposed to solve a number of problems using heuristic search, and that for some strange reason, the only search algorithms you have to choose from are beam-search and best-first-search (as defined in lecture). When might beam-search be preferable? When might best-first-search be preferable? (Note: there is not one unique answer to this question. Please just give a reasonable answer.)

5. (8 points) What are essential difference(s) between best-first-search and A*-search (as defined in lecture)?

6. (5 points) When is A*-search guaranteed to find an optimal path to a goal?

7. (4 points) Is A* with **h** = 0 (i.e., h(x) = 0 for all problem states x) guaranteed to find an optimal path to a goal? Please briefly explain.

8. (4 points) Is A* with **g** = 0 (i.e., g(x) = 0 for all problem states x) guaranteed to find an optimal path to a goal? Please briefly explain.

9. Attached are the generate and search files from the course web page.

   (a) (7 points) What exactly does the following return (note that *grammar* is set to *bigger-grammar*): **(rewrites 'np)**

   (b) (10 points) In this question you will implement a random search algorithm. In particular, it will explore at random the nodes that have been generated so far during search. The search is not completely random in that (1) before a node is considered, its parent must have already been considered, and (2) left to run its course, the search is exhaustive (it is able to search the entire state space, if the state space is finite). Assume that the following function has already been defined for you:

   (random list) : takes a list as input, and returns a list with the same values in a random order.

   Implement the search algorithm by writing the appropriate call to **tree-search**.

10. Consider the following set of first-order predicate calculus sentences.

    forall X,Y p(X,Y) $\rightarrow$ q(X,Y).
    forall W,Z p(W,t(Z)) $\rightarrow$ r(W,t(Z)).
    motherof(john,grandmother(sally)).

    (a) (4 points) Identify which symbols are function symbols and which are predicate symbols.

    (b) (12 points)
    Give an interpretation that satisfies the above set of sentences. There is one restriction: the interpretations of the predicate and function symbols must not be empty sets. To make this question feasible, assume that there are only 10 elements in the domain, which we will call o1 through o10.

```
;; Generates a random sentence according to *grammar* (a global
;; variable currently set to *bigger-grammar*).  To generate a
;; random sentence, enter (generate 's).  (In fact, you can generate
;; any type of phase defined in the grammar with generate --
;; try (generate 'np), (generate 'vp), (generate 'det), etc.)
;;
;; Each grammar rule has the following format:
;; (<non-terminal> -> <list of alternatives>)
;; An alternative that is a list of elements, say (n1...nj),
;; is an and-node:  choosing that alternative amounts to
;; choosing all of n1...nj, in that order.

(setf *little-grammar* '(
    (s -> (np vp))
    (np -> (det noun) noun)
    (vp -> (verb np))
    (det -> a the)
    (noun -> cat cats dog dogs)
    (verb -> chased saw)))

(setf *bigger-grammar* '(
    (s -> (np vp))
    (np -> (det adj* noun pp*) name)
    (np -> pronoun)
    (vp -> (verb np))
    (adj* -> () (adj adj*))
    (pp* -> () (pp pp*))
    (pp -> (prep np))
    (adj -> colorless silly cold nice happy)
    (prep -> on with in)
    (det -> a the)
    (noun -> cat cats dog dogs)
    (name -> Sam Juan Anile)
    (pronoun --> he she it they)
    (verb -> chased saw visited)))

(setf *grammar* *bigger-grammar*)
; setf *grammar* to the grammar you want to use.

(defun rule-lhs (rule) (first rule))

(defun rule-rhs (rule) (rest (rest rule)))
```

```
;=====================================
; rewrites (cat):  cat is a non-terminal of the grammar.
; Returns all of the alternative ways to rewrite
; cat according to *grammar*.  The data structure is
; the same as a right-hand-side:  alternatives that
; are symbols are or-nodes; alternatives that are lists
; are and-nodes.

; This is the "first-rest" recursive version of rewrites.
; It works fine.
;(defun rewrites (cat) (rewriteHelper cat *grammar*))

;(defun rewriteHelper (cat grammar)
;   (cond ((null grammar) nil)
;         ((equal cat (rule-lhs (first grammar)))
;             (append (rule-rhs (first grammar))
;                     (rewriteHelper cat (rest grammar))))
;         (t (rewriteHelper cat (rest grammar)))))

;===================================
; rewrites (cat):  cat is a non-terminal of the grammar.
; Returns all of the alternative ways to rewrite
; cat according to *grammar*.  The data structure is
; the same as a right-hand-side:  alternatives that
; are symbols are or-nodes; alternatives that are lists
; are and-nodes.

(defun rewrites (cat)
   (apply #'append
          (mapcar #'(lambda (rule)
                       (if (equal cat (rule-lhs rule))
                           (rule-rhs rule)
                           nil))
                  *grammar*)))

(defun random-elt (choices)
   (elt choices (random (length choices))))

(defun generate (phrase)
; See notes below on calling generate with nil.
   (if (listp phrase)
```

```
      (apply #'append (mapcar #'generate phrase))
      (let ((choices (rewrites phrase)))
        (if (null choices)
            (list phrase)
            (generate (random-elt choices))))))

;  Suppose that generate is called with nil or the equivalent ().
;  This happens when the () option is chosen in adj* or pp* in
;  *bigger* grammar.  What happens?

;  listp applied to nil returns true, so the apply-append-mapcar
;  statement is executed.  MAPCAR, WHEN ITS SECOND ARGUMENT IS
;  NIL, SIMPLY RETURNS NIL.  Applying append to nil yields nil,
;  so the final result is nil.


;Here is a function that returns the syntactic structure of the
; sentence which is implied by the rule applications.

(defun generate-tree (phrase)
;Generates a random sentence or phrase, returning the parse tree.
   (if (listp phrase)
       ; change 1--don't flatten the result by applying append
       ; to the result of the mapcar.
       (mapcar #'generate-tree phrase)
       (let ((choices (rewrites phrase)))
         (if (null choices)
             (list phrase)
             ;change 2--cons the phrase onto the front of the
             ;  rewrite of the phrase.
             (cons phrase (generate-tree (random-elt choices))))))
```

```lisp
(defun h (state)
  (terpri)
  (format t "h(~S) = " state)
  (read))

(defun successors (state)
   (terpri)
   (format t "Enter successors of ~S (in a list) --> " state)
   (read))

(defun goalp (state)
   (terpri)
   (format t "Is ~S a goal? (nil for no, t for yes) " state)
   (read))

(defun tree-search (myfun open)
     (cond ((null open) 'fail)
           ((goalp (first open)) (first open))
           (t (tree-search myfun
               (funcall myfun
                        (successors (first open))
                        (rest open))))))

(defun depth-first-search (start)
    (tree-search #'append (list start)))

(defun breadth-first-search (start)
    (tree-search #'(lambda  (list1 list2) (append list2 list1))
                 (list start)))

(defun best-first-search (start)
    (tree-search #'(lambda (list1 list2)
                     (sort (append list1 list2)
                             #'(lambda (state1 state2)
                                 (< (h state1) (h state2)))))
                  (list start)))
```