Fred Morcos / Ahmed Hisham / Mohammed AlMehdar

# AI PROJECT 2 REPORT: THE WUMPUS WORLD EAGLE

The Wumpus world is a simple AI problem where the 'Agent' i.e. the intelligence program, should obtain a certain target 'Gold' from a map grid while avoiding certain obstacles 'Wumpus, Pit' .

The Wumpus world eagle problem is a specific version of the wumpus world, first of all the agent in this world uses an 'Eagle' , which we can consider as another agent or more suitably as the agent's tool or ability to explore the map locations in order to obtain information about the goal and obstacles especially without running into them and also without any action costs in the search phase - action costs are a criteria for the agent's performance, a score that increases with favorable actions and decreases with unfavorable ones. The 'eagle' is used as a search phase to explore all areas of the grid and , thus, formulate a plan for the agent to fellow or decide on the futility of any action instead (if the goal is unobtainable).

There are two sides to the eagle search problem, one is how to maximize the efficiency of the search itself i.e. the traversing of the nodes, the second is to formulate a plan that is 'optimal' i.e. guarantees the efficiency of the agent's actions as well i.e. choose a plan which will maximize the score . Other then implementing the search strategy another part to the project also incorporates creating a 4x4 grid random map ' world' generator which has one Wumpus, one gold, and a number of pits, creating an abstract data type. This will allow us to test our planner in various conditions (maps).

We choose to use A* search for our Wumpus world algorithm, the reason for this is that A* search is proven to be both optimally efficient and complete, that is, 'no other algorithm is guaranteed to expand fewer nodes then A*' , and it is also guaranteed to return the solution 'node' with the least cost.

A* search uses both an heuristic and a cost function, it evaluates nodes on the order of a cheapest solution function $f(n) = g(n) + h(n)$ , always placing the node with the smallest $f(n)$ first in the queue, and where $g(n)$ equals a given cost function for each of the agent's actions., and $h(n)$ is set to the well known 'Manhattan city block distance' function. $g(n)$ is based on the following requirements on the score:

"

- −1 for each move or turn.
- +1 for shooting the arrow and killing the wumpus.

- −10 for shooting the arrow and missing the wumpus.

“

In our implementation of the A* search algorithm, we only expand nodes which are considered to be valid, thus we ignore any nodes that result from movement outside the map grid for example or , a node that results from shooting when the Wumpus isn't in front of us (since this is never a favorable action in our situation) , so we do not need to consider the effect of the last point - shooting and missing - on the f(n) since we won't consider that node in the first place. However since moving and turning decrease our score, we consider them as a cost so each move or turn increases g(n) by one, on the other hand since shooting and killing the Wumpus increases our score its considered as a reduction to the cost thus we decrement the g(n) of the 'shoot the wumpus' node by one, the last option is to grab the gold which doesn't result in any cost so its a zero and leaving the cave which ends the problem so we don't need to consider the search any further. thus no need to consider it as a cost.

```
;;; defines costs for actions.
(defun cost (action)
        (cond
                ((eql action 'f) 1)
                ((eql action 'r) 1)
                ((eql action 'l) 1)
                ((eql action 'g) 0)
                ((eql action 's) -1)
        )
)
```

As for the heuristic function, its defined as the city block , or manhattan, distance , where $h(n) = |x_r - x_n| + |y_r - y_n|$ where r is the goal node and n is the node we're calculating h(n) for (the one we're placing in the queue) and x and y are the positions of the nodes on the 4x4 map grid.

This concludes it for the search algorithm , since this algorithm is both optimally efficient and complete, we use it for both goal nodes : grabbing the gold and going back to the cave.

The program also accounts for repeating states by storing a list of repeated states and discarding any repeated state node as invalid so that the search will not go on an infinite loop or expand more nodes then necessary, using the state data structure:

```
(defstruct (state
        (:print-function
    (lambda (struct stream depth)
      (declare (ignore depth))
      (format stream "[~A ~A ~A ~A ~A]"
        (state-cellNumber struct)
        (state-orientation struct)
        (state-isWumpusAlive struct)
        (state-gotGold struct)
        (state-hasArrow struct))))
         ) cellNumber orientation isWumpusAlive gotGold hasArrow)
```

The map is represented as a one dimensional list of cells, cells can either P (Pit) , which the agent can't move through, W (Wumpus) which is favorable to shoot and can't move through otherwise, G (Gold) and NILS (empty items). The eagle exploration phase (function) returns a list where the first item is the location of the gold in the map, the second item is the wumpus position and the rest of the list contains the locations of all the pits. We generate the map using the built-in lisp 'random' function to set a random number of pits and to set them along with the gold and Wumpus in a random free cell obtained using the getFreeCell function:

```
(defun generateMap ()
        (setf map (make-array 16))
        (setf (aref map (+ (random 15) 1)) 'W)
        (setf (aref map (getFreeCell)) 'G)
        (dotimes (i (random 13))
            (setf (aref map (getFreeCell)) 'P)
        )
```

```
        map
)

(defun getFreeCell ()
        (let ((n (+ (random 15) 1)))
                (cond ((eql (aref map n) NIL) n)
                        (t (getFreeCell))
                )
        )
)
```

We use a function getPosition which returns the new position (cell number) of the agent (state) from any action taken. In our heuristic function since the map is stored in a one dimensional list we need to figure out the x and y for a corresponding cell number i.e. map is represented as fellows:

| | | | |
|---|---|---|---|
| 12 | 13 | 14 | 15 |
| 8 | 9 | 10 | 11 |
| 4 | 5 | 6 | 7 |
| 0 | 1 | 2 | 3 |

So the heuristic function takes into account that 7 corresponds to x,y 3,1 .. and if the agent's at 7 facing north then the getPosition function should return that this results in a new position of 7+4 = 11. This is implemented in the functions as fellows:

```
;;; gets the next cell number in the hypothetical 2D map based on the current
;;; direction, the current cell number and the action to be executed.
(defun getPosition (num orientation action)
```

```
        (cond
                ((eql action 's) num)
                ((eql action 'g) num)
                ((eql action 'l) num)
                ((eql action 'r) num)
                ((eql action 'f)
                        (cond
                                ((and (eql orientation 'n) (<= (+ num 4) 15)) (+ num 4))
                                ((and (eql orientation 's) (>= (- num 4) 0)) (- num 4))
                                ((and (eql orientation 'e)
                                        (not (or (eql num 3) (eql num 7) (eql num 11) (eql
num 15)))) (+ num 1))
                                ((and (eql orientation 'w)
                                        (not (or (eql num 0) (eql num 4) (eql num 8) (eql num
12)))) (- num 1))
                                (t num)
                        )
                )
                (t num)
        )
    )

    ;;; heuristic function for problem1 (go and grab the gold), uses the city-block
    ;;; distance to calculate how close is the current cell number to the goal cell
    ;;; number (the cell with the gold).
    (defun heuristic1 (current)
            (setf i1 (rem current 4))
            (setf j1 (floor current 4))
```

```
        (setf i2 (rem (first explorations) 4))
        (setf j2 (floor (first explorations) 4))
        (setf res (+ (abs (- i2 i1)) (abs (- j2 j1))))
)

;;; heuristic function for problem2 (go back and climb out of the cave), uses the
;;; city-block distance to calculate how close is the current cell number to the
;;; goal cell number (0, 0).
(defun heuristic2 (current)
        (setf i1 (rem current 4))
        (setf j1 (floor current 4))
        (setf i2 0)
        (setf j2 0)
        (setf res (+ (abs (- i2 i1)) (abs (- j2 j1))))
)
```

## SAMPLE SESSIONS WITH THE WUMPUS WORLD EAGLE PROGRAM

The output is formated as fellows:

-

(Map)

-

(Score / Number of expanded nodes)

(Sequance of actions or NO WAY)

-

P W G P

P P

P P

P P

-

985 46

(F L F F S F R F G R F F R F L F R F C)

-

P P P P

P P P

W P G

P P P

-

NO-WAY

-

P P P

P P W P

P G

P P P

-

NO-WAY

-

G

P P W

P

-

985 67

(F L F F R F F G R S R F F L F F R F C)

## INSTRUCTIONS ON HOW TO RUN THE WUMPUS WORLD EAGLE PROGRAM

- starts C-lisp

- load the file Wumpus.lisp (load "Wumpus.lisp")

- run main (main)

# WUMPUS WORLD EAGLE PROJECT: SOURCE CODE

```
;;; abstract data type for the problem to solve, actions is the set
;;; of possible actions (children of expanded states), initState is
;;; the initial state node of the problem, goalFunction is a pointer
;;; to the function that checks for a goal node and costFunction is
;;; a pointer to the function that returns the cost of an action.
(defstruct problem actions initState goalFunction costFunction)

;;; abstract data type for a state, representing the current state
;;; of a node in the search tree. print-function is a function to
;;; print information about a state, cellNumber is the current cell
;;; number of the agent in the map, orientation is the agent's
;;; orientation, isWumpusAlive is whether the wumpus is still alive
;;; or not, gotGold is whether the agent has the gold or not and
;;; hasArrow is whether the agent still has the arrow or not.
(defstruct (state
                (:print-function
       (lambda (struct stream depth)
         (declare (ignore depth))
         (format stream "[~A ~A ~A ~A ~A]"
             (state-cellNumber struct)
             (state-orientation struct)
             (state-isWumpusAlive struct)
             (state-gotGold struct)
             (state-hasArrow struct))))
                ) cellNumber orientation isWumpusAlive gotGold hasArrow)

;;; abstract data type for a node in the search tree. print-function
;;; prints information about the searchNode, state is the current
;;; state at the current node, parentNode is a pointer to the parent
;;; of this node, operator is the action that was followed by
;;; expansion of the parent node to arrive to the current node, depth
;;; is the depth of the node in the tree and cost is the total cost
```

```
;;; from the root node to the current node.
(defstruct (searchNode
             (:print-function
      (lambda (struct stream depth)
        (declare (ignore depth))
        (format stream "[~A ~A ~A ~A]"
                       (searchNode-state struct)
           (searchNode-parentNode struct)
           (searchNode-operator struct)
           (searchNode-depth struct)
           (searchNode-cost struct))))
              ) state parentNode operator depth cost)

;;; global variable for the map.
(defparameter map (make-array 16))

;;; global variable for the eagle's exploration, the first item in
;;; the list is the cell number of the gold, the second item is the
;;; cell number of the wumpus and the rest of the list is the cell
;;; numbers of pits.
(defparameter explorations '())

;;; global variable for priority queue used in expansion.
(defparameter pQueue '())

;;; global variable for a list of already visited states to avoid revisiting states.
(defparameter foundStates '())

;;; global variable for the number of steps executed to dequeue and
;;; expand a node.
(defparameter steps 0)

;;; global variable for a flag that tells whether we are solving
;;; problem1 (go grab the gold) or problem2 (return and climb out of
;;; the cave).
```

```
(defparameter returnBack 'f)

;;; this will print the map to the console. loops over columns in every
;;; row and prints W, G, P, or space.
(defun showMap ()
        (dotimes (i 4)
                (if (eql (aref map (+ i 12)) nil) (princ '\ ) (princ (aref map (+ i 12))))
                (princ '\ )
        )
        (princ #\newline)
        (dotimes (i 4)
                (if (eql (aref map (+ i 8)) nil) (princ '\ ) (princ (aref map (+ i 8))))
                (princ '\ )
        )
        (princ #\newline)
        (dotimes (i 4)
                (if (eql (aref map (+ i 4)) nil) (princ '\ ) (princ (aref map (+ i 4))))
                (princ '\ )
        )
        (princ #\newline)
        (dotimes (i 4)
                (if (eql (aref map i) nil) (princ '\ ) (princ (aref map i)))
                (princ '\ )
        )
        (princ #\newline)
)

;;; this will randomly generate a map of 16 cells with the first cell free (safe),
;;; one wumpus, one cell with gold and a random number of pits in random cells.
(defun generateMap ()
        (setf map (make-array 16))
        (setf (aref map (+ (random 15) 1)) 'W)
        (setf (aref map (getFreeCell)) 'G)
        (dotimes (i (random 13))
                (setf (aref map (getFreeCell)) 'P)
```

```
		)
		map
)

;;; returns a random empty cell, used by generateMap as to not fill in an item
;;; in a cell that already has an item.
(defun getFreeCell ()
		(let ((n (+ (random 15) 1)))
			(cond ((eql (aref map n) NIL) n)
				(t (getFreeCell))
			)
		)
)

;;; this is the eagle's explorations, fills in the explorations list with the
;;; cell number of the gold in the first item, the cell number of the wumpus in
;;; the second item and cell numbers of pits in the rest of the list.
(defun eagleExplore ()
		(dotimes (i 16)
			(cond ((eql (aref map i) 'P) (setf explorations (cons i explorations))))
		)
		(dotimes (i 16)
			(cond ((eql (aref map i) 'W) (setf explorations (cons i explorations))))
		)
		(dotimes (i 16)
			(cond ((eql (aref map i) 'G) (setf explorations (cons i explorations))))
		)
)

;;; heuristic function for problem1 (go and grab the gold), uses the city-block
;;; distance to calculate how close is the current cell number to the goal cell
;;; number (the cell with the gold).
(defun heuristic1 (current)
		(setf i1 (rem current 4))
		(setf j1 (floor current 4))
```

```
        (setf i2 (rem (first explorations) 4))
        (setf j2 (floor (first explorations) 4))
        (setf res (+ (abs (- i2 i1)) (abs (- j2 j1))))
)

;;; heuristic function for problem2 (go back and climb out of the cave), uses the
;;; city-block distance to calculate how close is the current cell number to the
;;; goal cell number (0, 0).
(defun heuristic2 (current)
        (setf i1 (rem current 4))
        (setf j1 (floor current 4))
        (setf i2 0)
        (setf j2 0)
        (setf res (+ (abs (- i2 i1)) (abs (- j2 j1))))
)

;;; calculates the priority of a search node based on its heuristic function.
(defun priorityCost (node)
        (setf s (searchNode-state node))
        (+ (searchNode-cost node)
                (if (eql returnBack 'f)
                        (heuristic1 (state-cellNumber s))
                        (heuristic2 (state-cellNumber s))       ;; else
                )
        )
)

;;; delegate function to the sort function to sort two search nodes based on
;;; their priorities.
(defun sortPair (a b)
        (cond ((>= (priorityCost a) (priorityCost b)) nil)
                (t t)
        )
)
```

```
;;; inserts a set of search nodes into the priority queue, appends the nodes
;;; to the pqueue global variable then sorts them using sortPair as the
;;; delegate function.
(defun insertPQueue (list1 list2)
        (sort (append list1 list2) #'sortPair)
)

;;; the implementation of main search algorithm. it takes an instance problem and returns a goal
;;; node (if one is found) or nil. starting with the root node, the main idea is to check if the
;;; current node is a goal (then return it) or not (then remove it and add a set of nodes that
;;; results from applying each action to the current node).
(defun A-Search (problem)
        (setf foundStates (cons (problem-initState problem) foundStates))
        (setf node (make-searchNode :state (problem-initState problem) :depth 0 :cost 0))
        (setf pQueue (cons node '()))
        (loop
                (when (null pQueue)
                        (return nil)
                )
                (setf node (first pQueue))
                (setf pQueue (rest pQueue))
                (setf steps (+ 1 steps))
                (when (eql (funcall (problem-goalFunction problem) (searchNode-state node)) 't)
                        (return node)
                )
                (setf pQueue (insertPQueue pQueue (expand problem node)))
        )
)

;;; tests for the goal for problem1 (got the gold).
(defun firstgoalTest (x)
        (if (eql (state-gotGold x) 't) 't nil)
)

;;; tests for the goal for problem2 (got the gold and is in cell (0, 0).
```

```
(defun secondgoalTest (x)
        (if (and (eql (state-cellNumber x) '0) (eql (state-gotGold x) 't)) 't nil)
)

;;; gets the next cell number in the hypothetical 2D map based on the current
;;; direction, the current cell number and the action to be executed.
(defun getPosition (num orientation action)
        (cond
                ((eql action 's) num)
                ((eql action 'g) num)
                ((eql action 'l) num)
                ((eql action 'r) num)
                ((eql action 'f)
                        (cond
                                ((and (eql orientation 'n) (<= (+ num 4) 15)) (+ num 4))
                                ((and (eql orientation 's) (>= (- num 4) 0)) (- num 4))
                                ((and (eql orientation 'e)
                                        (not (or (eql num 3) (eql num 7) (eql num 11) (eql num
15)))) (+ num 1))
                                ((and (eql orientation 'w)
                                        (not (or (eql num 0) (eql num 4) (eql num 8) (eql num
12)))) (- num 1))
                                (t num)
                        )
                )
                (t num)
        )
)

;;; helper function that sets isWumpusAlive and hasArrow to false.
(defun killWumpus (state)
        (setf (state-isWumpusAlive state) 'f)
        (setf (state-hasArrow state) 'f)
)

;;; helper function that sets gotGold to true.
```

```
(defun gotGold (result)
		(setf (state-gotGold result) 't)
		result
)

;;; checks if state is an already repeated state (in the foundStates global
;;; list variable) and returns true or false accordingly.
(defun repeatedState (state)
		(setf output 'f)
		(dotimes (i (length foundStates))
			(setf tmp (nth i foundStates))
			(setf tmpOutput (equalStates state tmp))
			(if (eql tmpOutput 't) (setf output 't))
		)
		output
)

;;; checks if s1 and s2 states are the same or not.
(defun equalStates (s1 s2)
		(if (and (eql (state-cellNumber s1) (state-cellNumber s2))
					(eql (state-orientation s1) (state-orientation s2))
					(eql (state-isWumpusAlive s1) (state-isWumpusAlive s2))
					(eql (state-gotGold s1) (state-gotGold s2))
					(eql (state-hasArrow s1) (state-hasArrow s2))
				) 't 'f
		)
)

;;; expects the next state that results from applying the passed action on the passed state.
(defun getNextState (state action)
		;; the result state is supposed to be identical to the current state except for the changes
		;; that the action will do.
		(setf result '())
		(setf result (copy-state state))
```

```
;; the grap action is only effective if the current cell has the gold cell and I didn't grasped
;; it before. Otherwise, the result state will not be changed.
(if (and (eql action 'g) (eql (first explorations) (state-cellNumber state))
        (eql (state-gotGold state) 'f)) (return-from getNextState (gotGold result)))

;; there is no result state in case of a forward action and the next cell has a PIT.
(dotimes (i (length explorations))
        (if (and (eql action 'f) (eql (nth (+ i 2) explorations) (getPosition
                        (state-cellNumber state) (state-orientation state) 'f)))
                        (return-from getNextState nil))
)

;; there is no result state in case of a shoot action and the next cell doesn't has the Wumpus
;; or I don't have the arrow.
(if (eql action 's) (if (and (eql (state-hasArrow state) 't)
        (eql (second explorations) (getPosition (state-cellNumber state) (state-orientation
state) 'f)))
                (killWumpus result) (return-from getNextState nil))
)

;; using the getPosition function, predict the next cell.
(setf (state-cellNumber result)
        (getPosition (state-cellNumber state) (state-orientation state) action))

;; handling the effects of rotations.
(cond
        ((and (eql action 'l) (eql (state-orientation state) 'n)) (setf (state-orientation result)
'w))
        ((and (eql action 'l) (eql (state-orientation state) 'w)) (setf (state-orientation result)
's))
        ((and (eql action 'l) (eql (state-orientation state) 's)) (setf (state-orientation result) 'e))
        ((and (eql action 'l) (eql (state-orientation state) 'e)) (setf (state-orientation result)
'n))
        ((and (eql action 'r) (eql (state-orientation state) 'n)) (setf (state-orientation result)
'e))
        ((and (eql action 'r) (eql (state-orientation state) 'w)) (setf (state-orientation result)
'n))
        ((and (eql action 'r) (eql (state-orientation state) 's)) (setf (state-orientation result)
'w))
```

```
                ((and (eql action 'r) (eql (state-orientation state) 'e)) (setf (state-orientation result) 's))
        )

        ;; finally, if the result is not visited before, then return it.
        (if (eql (repeatedState result) 't) nil result)
    )

    ;;; expands the given node by all the actions list in the given problem.
    (defun expand (problem node)
        (setf res '())
        (dotimes (i (length (problem-actions problem)))
            ;; get the current action from the problem.
            (setf action (nth i (problem-actions problem)))

            ;; using the getNextState function, finds the effect of the current action
            (setf nextState (getNextState (searchNode-state node) action))

            ;; if the result state is valid, create a corresponding node for it.
            (if (not (null nextState))
                (setf res (cons (make-searchNode :state nextState
                                                 :parentNode node
                                                 :operator action
                                                 :depth (+ (searchNode-depth node) 1)
                                                 :cost (+ (searchNode-cost node)
                                                          (funcall (problem-costFunction problem) action))
                ) res))
            )
            (if (not (null nextState)) (setf foundstates (cons nextState foundstates)))
        )

        res
    )

    ;;; defines costs for actions.
```

```
(defun cost (action)
        (cond
                ((eql action 'f) 1)
                ((eql action 'r) 1)
                ((eql action 'l) 1)
                ((eql action 'g) 0)
                ((eql action 's) -1)
        )
)

;;; once the goal is achieved, will backtrack in the search tree starting from the
;;; goal node and find the set of actions that were executed to arrive to this node,
;;; the result list of actions from this function should be reversed.
(defun backtrack (node)
        (cond
                ((null (searchNode-parentNode node)) '())
                (t (cons (searchnode-operator node) (backtrack (searchnode-parentnode node))))
        )
)

(defun main ()
        ;; some initializations for the global variables.
        (setf map (make-array 16))
        (setf explorations '())
        (setf pQueue '())
        (setf foundStates '())
        (setf totalPath '())
        (setf totalCost 0)
        (setf steps 0)
        (setf returnBack 'f)

        (generateMap)
        (showMap)
        (eagleExplore)
```

```
;; problem 1 is to search for the path to the gold.
(setf problem1 (make-problem :actions '(f r l g s)
                                                    :initState (make-state :cellNumber '0
  :orientation 'e
  :isWumpusAlive 't
  :gotGold 'f
  :hasArrow 't)
                                                    :goalFunction #'firstgoalTest
                                                    :costFunction #'cost
))
;; resultant goal node.
(setf goalNode (A-Search problem1))
;; calculate the total cost until now.
(if (not (null goalNode)) (setf totalCost (searchNode-cost goalNode)))
;; find the list of actions to reach the gold.
(if (not (null goalNode)) (setf totalPath (reverse (backtrack goalNode))) (return-from main
'NO-WAY))

(setf returnBack 't)

;; problem 2 is to search for the path to home.
(setf problem2 (make-problem :actions '(f r l g s)
                                                    :initState (searchNode-state
goalNode)
                                                    :goalFunction #'secondgoalTest
                                                    :costFunction #'cost
))
;; resultant goal node.
(setf goalNode (A-Search problem2))
;; calculate the final total cost.
(if (not (null goalNode)) (princ (- 1000 (+ totalCost (searchNode-cost goalNode)))))
(princ '\ )
(princ steps)
;; append to the actions list, the list of actions to reach home.
(if (not (null goalNode)) (setf totalPath (append totalPath (reverse (backtrack goalNode)))))
```

```
            (return-from main 'NO-WAY-BACK))
        ;; finally append the climb command.
        (if (not (null goalNode)) (append totalPath (cons 'c '())) (return-from main 'NO-WAY-BACK))
)
```