625 Programming Assignment 1

Topic: Search and Game Playing

1. Implement six search algorithms to solve 8-puzzle: `dfs`, `bfs`, `ids`, `greedy best-first (hence-forth ``greedy'')`, `a-star`, `ida-star`.
   - Test and compare time and space complexity for all cases.
   - Test and compare the effect of different heuristic functions (for the informed search algorithms).
2. Implement `min-max` and `alpha-beta` search for game playing.
   - Instead of a real game, your program will search a finite tree.

Part of this project is inspired by:

http://www.cs.utexas.edu/users/novak/asg-8p.html.

Part 1: Search

8-Puzzle with Search

- Input: a board configuration

  `'(1 3 4 8 6 2 7 0 5)`

- Output: sequence of moves

  `'(UP RIGHT UP LEFT DOWN)`

- Search methods to be implemented (use the exact function interface):

  `dfs, bfs, ids, greedy, a-star, ida-star.`

- Use h_1 (number of tiles out-of-place), and h_2 (sum of manhattan distance) for those requiring heuristics (make the functions to take the function as an argument).

- This is an **individual project**.

Submission Materials

Use the exact filename as shown below (in **bold**).

- Program code (**eight.lsp**): put all the code in a single text file.
  – Ample indentation and documentation is required.
- Documentation (**eight-report.pdf**): user manual, results, analysis.
- Inputs and outputs (include in a separate file **eight-result.pdf**; truncate output for search sessions that produce too much output). Report results for the three cases below:
  – Easy: `'(1 3 4 8 6 2 7 0 5)`
  – Medium: `'(2 8 1 0 4 3 7 6 5)`
  – Hard: `'(5 6 7 4 0 8 3 2 1)`

Submission Materials (Cont'd)

Continued from the previous page

- For each run, report the **number of nodes visited**. Except for IDA*, report the **maximum length of the node list** during the execution of the search. For IDA* report the **maximum depth of the recursion**.
- Compare the time and space complexity (from above) of various search methods using the Easy, Medium, and Hard case examples.
- For each method, comment on the strengths and weaknesses.
- Some search methods may fail to produce an answer. Analyze why it failed and report your findings.
- Do not run your algorithm for more than 10 minutes.

Function call interface

- See `http://courses.cs.tamu.edu/choe/20spring/625/src/eight-interface.lsp`
- Exactly follow the interfaces and function names.

Tips

Checking for duplicate states

```
(defun dupe (state node-list)
  (dolist (node node-list nil)
     (if (equal state (first node))
         (return-from dupe T))))
```

(You may use a state-list to save space, rather than a node-list, or better yet, use some kind of hash function.)

Note: This will lead to exponential growth in storage, but for this domain where it is very easy to loop back to a previously visited state, the overhead is minimal.

Node Representation

1	3	4
8	6	2
7		5

A node in the search tree has the following data structure:

```
'((1 3 4 8 6 2 7 0 5);blank is stored as 0
   h                  ;heuristic function value
   depth              ;depth from the root
   path))             ;list of moves from
                      ;  the start
```

Sorting

```
'((1 3 4 8 6 2 7 0 5);blank is stored as 0
   h                  ;heuristic function value
   depth              ;depth from the root
   path))             ;list of moves from
                      ;  the start
```

Sorting a node list, e.g. according to the heuristic:

```
(sort <node-list>
#'(lambda (x y) (< (second x) (second y)) )
)
```

lambda : read **define-anonymous function**

`#'something = (function something)`

cf. `'something = (quote something)`

Sorting: Alternatives

```
(defun sort-node-list (node-list)
  (sort node-list
    #'(lambda (x y) (< (second x) (second y)) )))

; the above is equivalent to :
(defun sort-node-list (node-list)
  (sort node-list
    (function (lambda (x y) (< (second x) (second y)) )))

; the above is equivalent to :
(defun compare-h ( x y )
  (< (second x) (second y)))

(defun sort-node-list (node-list)
  (sort node-list #'compare-h))
```

Sorting Pitfalls

- sort will alter the content of the first argument.

```
(setq vlist '( 7 2 9 3 1 10 5))
(sort vlist #'(lambda (x y) (< x y)))
--> (1 2 3 5 7 9 10)
vlist
--> (7 9 10)
```

- Always retrieve the returned result.

```
(setq vlist (sort vlist #'(lambda (x y) (< x y)))
```

Lambda Expression

`lambda` expression can basically replace any occurrences of function names, i.e. it works like an anonymous function:

```
(defun mysqr (x) (* x x))
(mysqr '11)

; the above is the same as
((lambda (x) (* x x)) '11)

; some more examples
(defun myop (x op)
    (eval (list op (first x) (second x))))

(myop '(2 3) '*)

(myop '(2 3) '(lambda (x y) (* x y)))
```

Sorting: Example

```
(setq test-node-list
   '((list1 10 0 0) (list2 87 0 0)
     (list 100 0 0) (list 5 1 0 0))
)

(defun sort-node-list (node-list)
   (sort node-list
       #'(lambda (x y) (< (second x) (second y)) )
   )
)

(sort-node-list test-node-list)
```

* You can use any combination of values to sort, and do ascending or descending sorts by changing the **lambda** function.

Utility Routines

Source is available on the course web page:

`http://courses.cs.tamu.edu/choe/20spring/625/src/eight-util.lsp`

- `(apply-op <operator> <node>)`: return new node after applying operator on current node
- `(print-tile <state>)`: prints out the board
- `(print-answer <state> <path>)`: prints boards after each move in the path, starting from the state.
- `(while <cond> <expr1> <expr2> ...)`: while loop macro.

See http://courses.cs.tamu.edu/choe/10fall/625/src/eight-util.txt for example runs.

DFS working code

See `http://courses.cs.tamu.edu/choe/20spring/625/src/dfs.lsp` for a functioning DFS code.

You can either use the recursive version (`dfs`) or iterative version (`dfs-iter`) as the base. The iterative version is more memory-efficient, so it is recommended that you use this.

Other tips

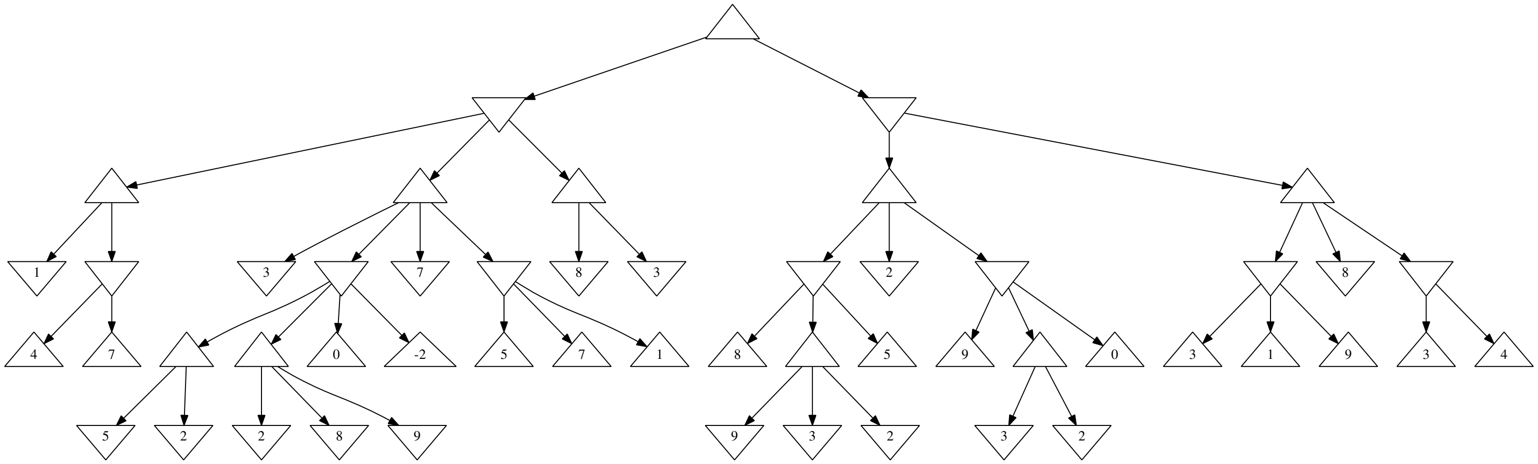
For this assigment, it is highly recommended that you compile and run your program. See ROB, “Lisp: compiling”.

Part 2: Game Playing

Game Tree Search

For this part, you will implement simple min-max and alpha-beta pruning algorithms for a game-tree search. The game will be given as a simple LISP list representing a game tree. Leaves will represent end-game state. Values at the leaves will represent the utility value.

For example, `'(((1 (4 7)) (3 ((5 2) (2 8 9) 0 -2) 7 (5 7 1)) (8 3)) (((8 (9 3 2) 5) 2 (9 (3 2) 0)) ((3 1 9) 8 (3 4 ))))` is the LISP representation for the game tree below.



The root node is assumed to be a MAX node.

Task

- Write two functions `min-max` and `alpha-beta` to conduct game tree search.
- The function call should be as follows:

```
; Example: (min-max '(1 (5 7) 4))
(min-max tree)

; Example: (alpha-beta '(1 (5 7) 4))
(alpha-beta tree)
```

- The output should be as follows:
  - `min-max`: solution path (a series of numbers indicating the path taken). For example, given a tree `'(1 (5 7) 4)`, the solution path will be `2, 1`. Max node root will select the second child (subtree `(5 7)`), which is a Min node, and it will choose the first child (leaf `5`).
  - `alpha-beta`: on top of the solution path, you should also indicate the MIN and MAX cuts. (1) Indicate whether it is a `MIN cut` or a `MAX cut`. (2) Output the local context where the cut is made: e.g., `MIN cut after (2 3) in subtree (1 (2 3) 4).`

Submission Materials

Use the exact filename as shown below (in **bold**).

- Program code (**game.lsp**): put all the code in a single text file.
  – Ample indentation and documentation is required.
- Documentation (**game-report.pdf**): user manual, results, analysis.
- Inputs and outputs (include in a separate file **game-result.pdf**; truncate output for search sessions that produce too much output). Report results for the five cases below:

```
'((4 (7 9 8) 8) (((3 6 4) 2 6) ((9 2 9) 4 7 (6 4 5))))

'(((1 4) (3 (5 2 8 0) 7 (5 7 1)) (8 3)) (((3 6 4) 2 (9 3 0)) ((8 1 9) 8
(3 4 ))))

'(5 (((4 7 -2) 7) 6))

'((8 (7 9 8) 4) (((3 6 4) 2 1) ((6 2 9) 4 7 (6 4 5))))

'(((1 (4 7)) (3 ((5 2) (2 8 9) 0 -2) 7 (5 7 1)) (8 3)) (((8 (9 3 2) 5) 2
(9 (3 2) 0)) ((3 1 9) 8 (3 4 ))))
```

Grading Criteria

- Analysis, program comments, readability: 15%
- Search: 60%

  dfs, bfs, ids: 5% each

  greedy, a-star, ida-star: 15% each
- Game Playing: 25%

  min-max: 10%

  alpha-beta: 15%

Submission (both parts)

You may use Python or C/C++ or Matlab/Octave instead of Lisp, but the input and output to your program should be the same as required for Lisp.

- Submit a single zip file (**prog1.zip**) including all pdf files to eCampus.
- See the course web page for details.
- 1% per hour late penalty.