420-500: Programming Assignment 2

Read every page very carefully before you begin.

1. Implement seven search algorithms to solve 8-puzzle: `dfs, bfs, dls, ids, greedy, a-star, ida-star`.
2. Test and compare time and space complexity for all cases.
3. Test and compare the effect of different heuristic functions (for the informed search algorithms).

This project is inspired by: http://www.cs.utexas.edu/users/novak/asg-8p.html.

Program 2: 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, dls, 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**.

Program 2: Required Material

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

- Program code (**eight.lsp**): put it in a single text file.
  – Ample indentation and documentation is required.
- Documentation (**README**): user manual plus results/analysis.
- Inputs and outputs (include in **README**; truncate output for search sessions that produce too much output):
  - 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)`

Program 2: Required Material (Cont'd)

Continued from the previous page

- For each run, report the **time** taken, the **number of nodes visited**. Except for IDA*, report the **maximum length of the node list (or recursion depth)** during the execution of the search.
- 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.

Program 2: Function interface

- See http://courses.cs.tamu.edu/choe/07fall/420/src/eight-interface.lsp
- Exactly follow the interfaces and function names.

Program 2 Tips (1)

Timing execution: use `(time (your-function-to-run))` to get the execution time.

```
* (time (car '(x x)))
real time : 0.000 secs
run time  : 0.000 secs
X


*
```

Program 2 Tips (2)

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 somekind of hash function.)

Program 2: State Representation

1	3	4
8	6	2
7		5

A node in the search tree has the following 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
```

Program 2: 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))
```

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.

Program 2: Utility Routines

Source is available on the course web page:
http://courses.cs.tamu.edu/choe/07fall/420/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/07fall/420/src/eight-util.txt for example runs.

Program 2: DFS working code

See http://courses.cs.tamu.edu/choe/07fall/420/src/dfs.lsp for a functioning DFS code.

Program 2: Other tips

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

Program 2: Grading Criteria

- analysis, program comments, readability: 15%
- dfs, bfs, dls, and ids: 10% each
- greety, a-star, and ida-star: 15% each

Program 2: Submission

- Turnin using CSNET turnin page.
- See the course web page for details.
- No late submission will be allowed.
- Only send **plain ASCII text** files. **Do not send MS-Word documents or other formatted text.**