CISC 481 Programming Assignment 1

September 20, 2020

Assignment Objectives

After completing this assignment, you'll understand the inner workings of the following search algorithms:

- iterative-deepening search
- breadth-first search
- A* search

You'll also gain some experience using a software profiler, which is a tool that lets you gather and analyze performance characteristics of your code. You'll use the statistics you gather to see in a very real sense the difference in time and space complexity of the above algorithms.

The n-Puzzle

This is a classic problem studied in Artificial Intelligence. You have a $m \times m$ grid of numbered tiles $1, 2, 3, \ldots, m^2 - 1$ and one blank space. The blank space allows tiles to be slid around on the board, and the goal is to slide the tiles into some particular configuration (usually in numerical order from left to right, top to bottom). For the project, we'll be dealing with the 8-puzzle, which has 8 numerical tiles on a $3x3$ grid. See Figure 1 from Russell & Norvig for a graphical depiction of the puzzle.

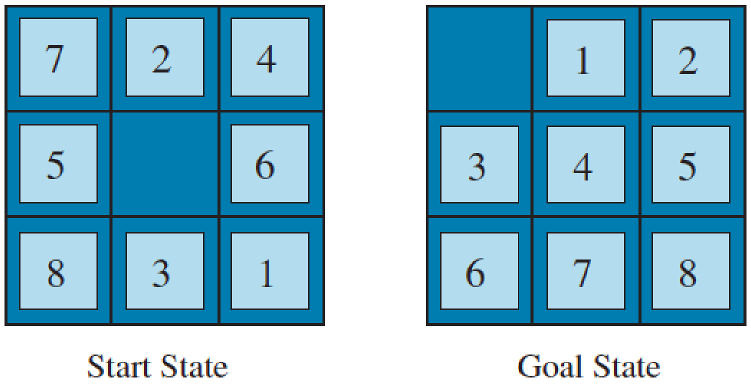

Figure 1: The 8 puzzle, with a typical start state and the goal that we're interested in.

Representing the Problem

[My examples here are in Lisp, but feel free to use something similar that's easy to parse in your target language.]

Probably the most straightforward way to represent an 8 puzzle board is as a 3×3 array of the numbers 1 to 8 and `nil` (for representing the blank). In Figures 2, 3, and 4 I give some names to a few initial states that I'll refer to throughout the writeup.

```
(defvar *puzzle-0* #2A((3 1 2)
                       (7 nil 5)
                       (4 6 8)))
```

Figure 2: This board is only a few moves away from the goal. It's useful for testing, and may be the only board your iterative deepening search will solve in any reasonable amount of time.

```
(defvar *puzzle-1* #2A((7 2 4)
                       (5 nil 6)
                       (8 3 1)))
```

Figure 3: The puzzle from the book, depicted in Figure 1. This will also be useful for testing, and I'll use it to provide example output here. An optimal solution to it should be 26 steps.

```
(defvar *puzzle-2* #2A((6 7 3)
                       (1 5 2)
                       (4 nil 8)))

(defvar *puzzle-3* #2A((nil 8 6)
                       (4 1 3)
                       (7 2 5)))

(defvar *puzzle-4* #2A((7 3 4)
                       (2 5 1)
                       (6 8 nil)))

(defvar *puzzle-5* #2A((1 3 8)
                       (4 7 5)
                       (6 nil 2)))

(defvar *puzzle-6* #2A((8 7 6)
                       (5 4 3)
                       (2 1 nil)))
```

Figure 4: Several other boards I randomly generated (along with the reverse-order board, which isn't so randomly generated. It's a good example, though, because it takes a lot of steps to solve). You'll be using these to profile and compare your breadth-first and A* searches.

Rules of Movement (Actions)

A tile in an n-puzzle can only move orthogonally (that is, it can move horizontally or vertically), and it can only move into the blank square. As such, the components of an *action* are are a *direction* which can be one of `up`, `down`, `left`, or `right`; and a *row* and *column* specifying the location of a tile to be moved in *direction*. If the neighboring element in *direction* of location $(row, column)$ is `nil`, then performing the action causes $(row, column)$ to swap values with that neighbor. Otherwise, the action is invalid. All actions have a path cost of 1.

Part 1 [10 pts]

Write a function `possible-actions` that takes a board as input and outputs a list of all actions possible on the given board.[1]

Part 2 [10 pts]

Write a function `result` that takes as input an action and a board and outputs the new board that will result after actually carrying out the input move in the input state. Be certain that you do not accidentally modify the input board variable.[2]

Part 3 [5 pts]

Write a function `expand` that takes a board as input, and outputs a list of all states that can be reached in one Action from the given state.[3]

Part 4 [15 pts]

Implement an *iterative deepening search* which takes an initial board and a goal board and produces a list of actions that form an optimal path from the initial board to the goal. Test your search on `*puzzle-0*`. You can try running it on some of the other puzzles, but don't feel discouraged if it takes a very long time before returning an answer.

Part 5 [15 pts]

Implement a *breadth-first search* which, like the iterative deepening search, takes an initial board and a goal and gives an optimal sequence of actions from the initial state to the goal. Test your search on `*puzzle-0*` and `*puzzle-1*`.

Part 6 [20 pts]

For this part, you'll be implementing *A* search*.

[1]This is the *ACTIONS* function we discussed in class and in the book.
[2]This is the *RESULT* function we discussed in class; the state transition model
[3]**Hint:** This is a trivial extension of Parts 1 and 2.

Part 6.1

Similar to breadth-first and iterative deepening, your A* search should take as input an initial board and a goal board. It should additionally take a *heuristic function*[4]

You can test your code by passing as the heuristic a function that always returns 0[5], which should reduce your search to *uniform-cost search*. Test it on `*puzzle-0*`. It may take a long time to solve any of the others.

Part 6.2

Now you'll implement the two classic n puzzle heuristic functions - *number of misplaced tiles* and *Manhattan distance*. Test your code with both of these on `*puzzle-0*` and `*puzzle-1*`. You'll probably notice that you have to wait some time (but hopefully not as long as with `(constantly 0)`!) to get back an answer when using the misplaced tiles heuristic. This illustrates very clearly just how much choosing a good heuristic matters for the practicality of A* search.

Part 7 [25 pts]

In this final part, you'll be benchmarking your searches to get an empirical sense of the differences in time/space complexity between them. For this you'll need a profiler. **NB** that some profilers also include memory usage statistics, which may give a good idea about the maximum amount of memory used at any point on a given run. If the profiler for your language does not include this, you'll have to instrument your code to keep track of the maximum number of nodes on the frontier at any one time over the entire run. In Figures 5 and 6 I list what I get when profiling runs of my reference implementations of breadth-first and A* search. This output was generated using SBCL's profiler. SBCL is the Common Lisp implementation that I use. Like most of Common Lisp tooling, the profiler is easily accessible using SLIME in Emacs[6]

What we're interested in here is "consed" (which is a good proxy for the total amount of memory used) and the number of calls we made to `expand`. The former gives us an idea of the space complexity, and the latter the time complexity. Note that A* performs orders of magnitude better than breadth-first on both counts! You can also see that A* is better in terms of actual time - taking less than half a second to complete where breadth-first takes over four seconds.

Part 7.1

Profile each of iterative deepening search, breadth-first search, and A* search using Manhattan distance solving `*puzzle-0*`. Even on this simple puzzle solvable in only 6 moves, you should be able to get a sense of the difference in performance characteristics between these three algorithms.

[4]At this point, if you're using a language which does not have first-class functions and higher-order functions, you'll have to figure out some way to parameterize the heuristic function to your search. This could be using function pointers in a language such as C/C++, or passing in a *e.g.* `Heuristic` object that has an `evaluate` method in Java. Common Lisp, as well as Python, JavaScript, and Haskell to name a few have first-class and higher-order functions.

[5]In Lisp, you can achieve this by calling `(constantly 0)`.

[6]If you're a VIM user, there's the analogous SLIMV. There's also a WIP LSP to SWANK bridge (SWANK performs the same role as a LSP implementation would between the Lisp implementation and Emacs/SLIME) that might get you the functionality you need in VSCode.

```
  seconds  |     gc     |   consed   |  calls |  sec/call   |  name
--------------------------------------------------------
     0.390 |      0.012 | 34,292,496 |      1 |   0.390106 | CISC481-20F-PROG1::A*-SEARCH
     0.005 |      0.000 |          0 | 28,555 |   0.000000 | CISC481-20F-PROG1::PARENT
     0.004 |      0.000 |  1,211,440 |  6,757 |   0.000001 | CISC481-20F-PROG1::RESULT
     0.004 |      0.000 |     32,768 |  2,531 |   0.000001 | CISC481-20F-PROG1::HEAP-POP
     0.003 |      0.000 |          0 |  3,943 |   0.000001 | CISC481-20F-PROG1::MANHATTAN-DISTANCE
     0.002 |      0.000 |    393,104 |  2,531 |   0.000001 | CISC481-20F-PROG1::POSSIBLE-ACTIONS
     0.000 |      0.000 |     65,536 |  2,531 |   0.000000 | CISC481-20F-PROG1::EXPAND
     0.000 |      0.000 |          0 |  3,990 |   0.000000 | CISC481-20F-PROG1::REHEAP-UP
     0.000 |      0.000 |    189,232 |  3,944 |   0.000000 | CISC481-20F-PROG1::HEAP-PUSH
--------------------------------------------------------
     0.409 |      0.012 | 36,184,576 | 54,783 |            | Total

estimated total profiling overhead: 0.08 seconds
overhead estimation parameters:
  4.0000003e-9s/call, 1.522e-6s total profiling, 6.66e-7s internal profiling

These functions were not called:
 CISC481-20F-PROG1::BREADTH-FIRST-SEARCH
 CISC481-20F-PROG1::BREADTH-FIRST-SEARCH-ARRAY
 CISC481-20F-PROG1::DEPTH-LIMITED-SEARCH
 CISC481-20F-PROG1::ITERATIVE-DEEPENING-SEARCH
 CISC481-20F-PROG1::MANHATTAN-LINEAR-CONLICT
 CISC481-20F-PROG1::NUM-MISPLACED CISC481-20F-PROG1::QUEUE-POP
 CISC481-20F-PROG1::QUEUE-PUSH CISC481-20F-PROG1::SOLVABLEP
```

Figure 5: Profiler output for A* on `*puzzle-1*` using the Manhattan Distance.

Part 7.2

Just to drive the point home about choosing a good heuristic, profile A* on `*puzzle-2*` once using the number of misplaced tiles, and then once using the Manhattan distance.

Part 7.3

Generate profiler reports for both of A* using Manhattan distance and breadth-first search on `*puzzle-2*`, `*puzzle-3*`, `*puzzle-4*`, `*puzzle-5*`, and `*puzzle-6*`.[7] Then calculate averages over each of the five runs for the amount of memory used and number of calls to `expand` for both A* and breadth-first search.

Solutions to `*puzzle-0*` and `*puzzle-1*`

Hopefully the listings in Figures 7, 8, and 9 will be useful for testing your code. Note that A* and breadth-first search find different solutions for `*puzzle-1*` - there's more than one optimal solution to this puzzle and they each find their own.

Submitting

You should submit all of your code - *document it appropriately, as you'll be graded on style.* You should also submit listing of all of the raw profiler reports that you're asked to generate in Part 7, as well as the averages you calculated.

[7]For a total of 10 separate profiler reports.

```
  seconds  |     gc     |     consed    |    calls   |  sec/call  |  name
----------------------------------------------------------------
     3.765 |      0.644 | 2,378,644,624 |          1 |   3.765357 | CISC481-20F-PROG1::BREADTH-FIRST-SEARCH
     0.246 |      0.039 |    91,115,616 |    460,676 |   0.000001 | CISC481-20F-PROG1::RESULT
     0.031 |      0.000 |    25,836,432 |    172,834 |   0.000000 | CISC481-20F-PROG1::POSSIBLE-ACTIONS
     0.010 |      0.000 |             0 |    172,834 |   0.000000 | CISC481-20F-PROG1::QUEUE-POP
     0.005 |      0.000 |       262,144 |    178,115 |   0.000000 | CISC481-20F-PROG1::QUEUE-PUSH
     0.000 |      0.000 |     2,523,136 |    172,834 |   0.000000 | CISC481-20F-PROG1::EXPAND
----------------------------------------------------------------
     4.058 |      0.683 | 2,498,381,952 |  1,157,294 |            | Total

estimated total profiling overhead: 1.76 seconds
overhead estimation parameters:
  4.0000003e-9s/call, 1.522e-6s total profiling, 6.66e-7s internal profiling

These functions were not called:
 CISC481-20F-PROG1::A*-SEARCH
 CISC481-20F-PROG1::BREADTH-FIRST-SEARCH-ARRAY
 CISC481-20F-PROG1::DEPTH-LIMITED-SEARCH CISC481-20F-PROG1::HEAP-POP
 CISC481-20F-PROG1::HEAP-PUSH
 CISC481-20F-PROG1::ITERATIVE-DEEPENING-SEARCH
 CISC481-20F-PROG1::MANHATTAN-DISTANCE
 CISC481-20F-PROG1::MANHATTAN-LINEAR-CONLICT
 CISC481-20F-PROG1::NUM-MISPLACED CISC481-20F-PROG1::PARENT
 CISC481-20F-PROG1::REHEAP-UP CISC481-20F-PROG1::SOLVABLEP
```

Figure 6: Profiler output for breadth-first search on `*puzzle-1*`.

```
(:ACTION-SEQUENCE
 ((RIGHT 1 0) (UP 2 0) (LEFT 2 1) (DOWN 1 1) (RIGHT 1 0) (DOWN 0 0))
 :STATE-SEQUENCE
 (#2A((3 1 2) (7 NIL 5) (4 6 8)) #2A((3 1 2) (NIL 7 5) (4 6 8))
  #2A((3 1 2) (4 7 5) (NIL 6 8)) #2A((3 1 2) (4 7 5) (6 NIL 8))
  #2A((3 1 2) (4 NIL 5) (6 7 8)) #2A((3 1 2) (NIL 4 5) (6 7 8))
  #2A((NIL 1 2) (3 4 5) (6 7 8)))
 :PATH-COST 6)
```

Figure 7: Optimal solution to `*puzzle-0*`

Extra Credit [6.25 pts]

Warning: I highly recommend you do **not** attempt this section before you're sure you've completed all the main parts of the assignment.

With that out of the way, consider the following two 15 puzzle instances:

```
(defvar *15-puzzle-1* #2A((13 10 11 6)
                          (5 7 4 8)
                          (1 12 14 9)
                          (3 15 2 nil)))

(defvar *15-puzzle-2* #2A((13 10 11 6)
                          (5 7 4 8)
                          (2 12 14 9)
                          (3 15 1 nil)))
```

One of these is solvable, the other is not! If you can figure out which one has a solution, try solving it first with your breadth-first search, and then with your A* search. You may have to look into using a better heuristic than even the Manhattan distance with A*!

What happens with breadth-first search on this puzzle? Include some profiler output, stack

```
(:ACTION-SEQUENCE
 ((RIGHT 1 0) (DOWN 0 0) (LEFT 0 1) (UP 1 1) (UP 2 1) (RIGHT 2 0) (DOWN 1 0)
  (LEFT 1 1) (LEFT 1 2) (DOWN 0 2) (RIGHT 0 1) (RIGHT 0 0) (UP 1 0) (LEFT 1 1)
  (LEFT 1 2) (UP 2 2) (RIGHT 2 1) (RIGHT 2 0) (DOWN 1 0) (LEFT 1 1) (LEFT 1 2)
  (DOWN 0 2) (RIGHT 0 1) (UP 1 1) (RIGHT 1 0) (DOWN 0 0))
 :STATE-SEQUENCE
 (#2A((7 2 4) (5 NIL 6) (8 3 1)) #2A((7 2 4) (NIL 5 6) (8 3 1))
  #2A((NIL 2 4) (7 5 6) (8 3 1)) #2A((2 NIL 4) (7 5 6) (8 3 1))
  #2A((2 5 4) (7 NIL 6) (8 3 1)) #2A((2 5 4) (7 3 6) (8 NIL 1))
  #2A((2 5 4) (7 3 6) (NIL 8 1)) #2A((2 5 4) (NIL 3 6) (7 8 1))
  #2A((2 5 4) (3 NIL 6) (7 8 1)) #2A((2 5 4) (3 6 NIL) (7 8 1))
  #2A((2 5 NIL) (3 6 4) (7 8 1)) #2A((2 NIL 5) (3 6 4) (7 8 1))
  #2A((NIL 2 5) (3 6 4) (7 8 1)) #2A((3 2 5) (NIL 6 4) (7 8 1))
  #2A((3 2 5) (6 NIL 4) (7 8 1)) #2A((3 2 5) (6 4 NIL) (7 8 1))
  #2A((3 2 5) (6 4 1) (7 8 NIL)) #2A((3 2 5) (6 4 1) (7 NIL 8))
  #2A((3 2 5) (6 4 1) (NIL 7 8)) #2A((3 2 5) (NIL 4 1) (6 7 8))
  #2A((3 2 5) (4 NIL 1) (6 7 8)) #2A((3 2 5) (4 1 NIL) (6 7 8))
  #2A((3 2 NIL) (4 1 5) (6 7 8)) #2A((3 NIL 2) (4 1 5) (6 7 8))
  #2A((3 1 2) (4 NIL 5) (6 7 8)) #2A((3 1 2) (NIL 4 5) (6 7 8))
  #2A((NIL 1 2) (3 4 5) (6 7 8)))
 :PATH-COST 26)
```

Figure 8: Optimal solution to `*puzzle-1*` as found by A*.

traces, or other error information to back up your conclusion here.

For full credit, you should also include a listing of the optimal solution to whichever of these is solvable. Explain how you got it (*e.g.* "A* search with the linear conflict + Manhattan distance heuristic"). If you used a new heuristic, you should include your implementation of it with the rest of your code.

```
(:ACTION-SEQUENCE
 ((RIGHT 1 0) (DOWN 0 0) (LEFT 0 1) (UP 1 1) (LEFT 1 2) (UP 2 2) (RIGHT 2 1)
  (RIGHT 2 0) (DOWN 1 0) (LEFT 1 1) (LEFT 1 2) (UP 2 2) (RIGHT 2 1) (RIGHT 2 0)
  (DOWN 1 0) (LEFT 1 1) (LEFT 1 2) (DOWN 0 2) (RIGHT 0 1) (RIGHT 0 0) (UP 1 0)
  (LEFT 1 1) (LEFT 1 2) (DOWN 0 2) (RIGHT 0 1) (RIGHT 0 0))
 :STATE-SEQUENCE
 (#2A((7 2 4) (5 NIL 6) (8 3 1)) #2A((7 2 4) (NIL 5 6) (8 3 1))
  #2A((NIL 2 4) (7 5 6) (8 3 1)) #2A((2 NIL 4) (7 5 6) (8 3 1))
  #2A((2 5 4) (7 NIL 6) (8 3 1)) #2A((2 5 4) (7 6 NIL) (8 3 1))
  #2A((2 5 4) (7 6 1) (8 3 NIL)) #2A((2 5 4) (7 6 1) (8 NIL 3))
  #2A((2 5 4) (7 6 1) (NIL 8 3)) #2A((2 5 4) (NIL 6 1) (7 8 3))
  #2A((2 5 4) (6 NIL 1) (7 8 3)) #2A((2 5 4) (6 1 NIL) (7 8 3))
  #2A((2 5 4) (6 1 3) (7 8 NIL)) #2A((2 5 4) (6 1 3) (7 NIL 8))
  #2A((2 5 4) (6 1 3) (NIL 7 8)) #2A((2 5 4) (NIL 1 3) (6 7 8))
  #2A((2 5 4) (1 NIL 3) (6 7 8)) #2A((2 5 4) (1 3 NIL) (6 7 8))
  #2A((2 5 NIL) (1 3 4) (6 7 8)) #2A((2 NIL 5) (1 3 4) (6 7 8))
  #2A((NIL 2 5) (1 3 4) (6 7 8)) #2A((1 2 5) (NIL 3 4) (6 7 8))
  #2A((1 2 5) (3 NIL 4) (6 7 8)) #2A((1 2 5) (3 4 NIL) (6 7 8))
  #2A((1 2 NIL) (3 4 5) (6 7 8)) #2A((1 NIL 2) (3 4 5) (6 7 8))
  #2A((NIL 1 2) (3 4 5) (6 7 8)))
 :PATH-COST 26)
```

Figure 9: Optimal solution to `*puzzle-1*` as found by breadth-first search.