CSCI 4202 – Introduction to Artificial Intelligence
Fall 2021 – Dr. Doug Williams
Programming Assignment 1 – Problem Solving Using Search
Due: Oct. 15, 2021

Sliding-Tile Puzzle

A sliding-tile puzzle is a rectangular grid of tile with one empty space. You can slide a tile into an adjacent empty space. The object of the puzzle is to rearrange the tiles into a given goal state. Figure 1 shows a typical instance of the 8-puzzle, which uses a 3 x 3 grid.

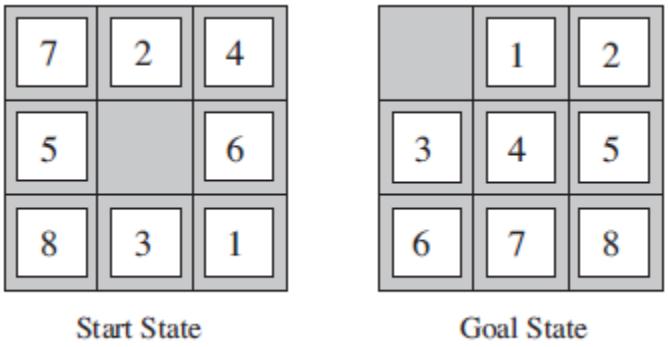

Figure 1. Typical instance of the 8-puzzle.

For this assignment, we will limit ourselves to n x n sliding-tile puzzles, where n > 1. Such a puzzle has tiles numbered 1 to n^2-1 plus the empty tile. For specific values of n, such puzzles are known as <n^2-1>-puzzles. The most common are 8-puzzles and 15-puzzles.

In this assignment you will write a series of programs to solve sliding-tile puzzles using various uninformed and informed (heuristic) methods.

You may use whatever operating system and programming language you like.

Part 1 – Reading and Validating Sliding-Puzzle Problems

We will use JSON, which is described below, to define specific sliding-tile puzzle instances for your programs to solve. An example JSON corresponding to the instance of the 8-puzzle in Figure 1 is:

```
{"n" : 3,
 "start" : [[7,2,4],
            [5,0,6],
            [8,3,1]],
 "goal" : [[0,1,2],
           [3,4,5],
           [6,7,8]]}
```

This is an 8-puzzle problem with a grid size of n = 3 and the specified 3 x 3 matrices for the start state and the goal state. Each state is a 3 x 3 matrix of non-negative integers and the empty space is denoted by the integer 0.

JSON

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

A description of the JSON format as well as links to parsers for various languages is available on the JSON web site at http://www.json.org.

Specifics

Find a JSON parser for the programming language you have chosen to use – there are generally several JSON parsers available for all major programming languages. There are links to such parsers at the site given above.

Read a sliding-tile puzzle problem using a JSON parser. The form of the internal data varies from language to language and even from parser to parser. Consult the documentation for your selected parser.

Check that the sliding-tile puzzle problem is valid. Specifically, there must be fields named n, start, and goal. The n field must be a positive integer greater than 1. The start and goal fields must be n x n matrices containing the integers 0 (for the empty space) to n^2-1.

You do not have to check that the problem is solvable – that is, that there is a series of moves that will transform the start state into the goal state.

Part 2 – Sliding-Tile Puzzle Rules

Given a sliding-tile puzzle state, you must be able to determine the rules that are applicable to that state that can be used to generate its successor states.

A rule has three parts:

- name – a simple name for the rule (e.g., up, left, down, right)
- precondition function – a Boolean function that accepts a state and returns true if the rule is applicable to state
- action function – a function that accepts a state and returns the successor state obtained by applying the rule

You can use these rules to implement functions such as applicable-rule, which returns a list of the rules applicable to a given state, and successor-state, which returns the successor state for a given state and rule.

Specifics

Encode the rules for the sliding-tile puzzle. Remember that it is easiest to consider moving the empty space up, left, down, or right. Using these rules, write routines to determine the rules applicable to a state and the successor state given a state and rule to apply.

Note that you could implement these as iterators or have them return lists (or vectors) or rules and states.

Part 3 – Backtracking Control Strategy

The simplest problem-solving strategy is a depth-first search implemented using a backtracking control strategy. See the paper Backtracking.pdf on the Canvas web site for the class. This paper gives two backtracking algorithms – BACKTRACK(DATA) and BACKTRACK1(DATALIST). We will be using BACKTRACK1(DATALIST), which avoids cycles, for this programming assignment.

The BACKTRACK1(DATALIST) algorithm implements a recursive depth-first search through a state space. The DATALIST argument is a list of the states in the current search path. The algorithm returns either a list of the rules to reach the goal state or failure if no goal was found. [You will have to decide how to represent failure in your implementation. Note that returning an empty list for failure is not a good idea because an empty list actually represents success when you are at the goal.]

Specifics

Implement the BACKTRACK1(DATALIST) algorithm to solve instances of the sliding-puzzle problem. The depth bound may be a global variable or passed as an argument.

Implement a main program to accept a sliding-tile puzzle problem and solve it using the BACKTRACK1(DATALIST) algorithm. Print the start state and the goal state; the solution and solution length; and the number of states that were examined.

Implement a main program to accept a sliding-tile puzzle problem and solve it using an iterative depth-first search using the BACKTRACK1(DATALIST) algorithm. Print the cumulative number of states examined and the final (optimal) solution.

Part 4 – Graph Search

Graph search is a general purpose algorithm for searching graphs and state space problems are a prime example of problems that are represented as graphs. Depending on the specific data structure used to store the list of unexplored nodes, you can get a breadth-first search (using a queue), a depth-first search (using a stack), or any of a number of best-first searches (using priority queues – or sorted lists – with various cost or heuristic functions).

You may either use the function Uniform-Cost-Search (problem) (from p. 84 in the text) or the GRAPHSEARCH algorithm from the Graph Search.pdf document on the Canvas web site for the course. The function Uniform-Cost-Search (problem) is probably the easier of the two to implement. But, it is not a true graph search algorithm (as noted in footnote 9 on page 95 in the text "… A* requires some extra bookkeeping to ensure optimality."). The GRAPHSEARCH algorithm has that "extra bookkeeping" by providing a true graph search.

Specifics

Implement either the Uniform-Cost-Search or GRAPHSEARCH algorithm and use it to perform a breadth-first search of sliding-tile problems.

Implement a main program to accept a sliding-tile puzzle and solve it using your implementation. Print the start and goal state; the solution and solution length; and the number of states generated and explored.

Part 5 – Algorithm A*

From the text p. 93:

> The most widely known form of best-first is called A* search (pronounced "A-star search"). It evaluates nodes by combining $g(n)$, the cost to reach the node, and $h(n)$, the cost to get from the node to the goal:
>
> $$f(n) = g(n) + h(n).$$
>
> Since $g(n)$ gives the path cost from the start node to node *n*, and $h(n)$ is the estimated cost of the cheapest path from *n* to the goal, we have
>
> $f(n) =$ the estimated cost of the cheapest solution through *n*.
>
> … The algorithm is identical to Uniform-Cost-Search except that A* uses *g* + *h* instead of *g*.

We will use the two heuristic functions described in the text on page 103.

- h_1 = the number of misplaced tiles.
- h_2 = the sum of the distances of the tiles from their goal positions (i.e., the Manhattan distance).

Specifics

Using your implementation from Part 4, implement Algorithm A*. Use the two heuristics above to solve instances of the sliding-tile puzzle.

Report

The Canvas web site for the course will include inputs for problems with solutions of lengths 1, 2, 3, 4, 5, 10, 15, 20, and 25, as well as the problem in the book with a solution length of 26. For initially testing your program, I recommend starting with problems with short solutions and then try the longer ones.

The results from this programming assignment will be turned in as a report in a single pdf file. The report will contain an introduction, problem description (based on this document), solution (design and code), results, and conclusions. The results sections will address each of the five parts above. Use longest solution length you are able to (reasonable) run – preferable the one from the book – in the results. The conclusions should include a graph of the nodes generated and examined for the various search techniques.

A grade of C requires Parts 1-3 to be implemented correctly. A grade of B requires Parts 1-4 to be implemented correctly, and a grade of A requires Parts 1-5 to be implemented correctly. [Note that these are the minimum requirements.]

Sample depth-first search using BACKTRACK1(DATALIST)

```
------------
start:
7 2 4
5 0 6
8 3 1
goal:
0 1 2
3 4 5
6 7 8
Solution length = 26
Nodes examined = 3321434
(list
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>))
```

Sample iterative deepening depth-first search using BACKTRACK1(DATALIST)

```
------------
start:
7 2 4
5 0 6
8 3 1
goal:
0 1 2
3 4 5
6 7 8
0: Cummulative nodes-examined = 1
1: Cummulative nodes-examined = 6
2: Cummulative nodes-examined = 23
3: Cummulative nodes-examined = 56
4: Cummulative nodes-examined = 113
5: Cummulative nodes-examined = 218
6: Cummulative nodes-examined = 419
7: Cummulative nodes-examined = 764
8: Cummulative nodes-examined = 1349
9: Cummulative nodes-examined = 2366
10: Cummulative nodes-examined = 4199
11: Cummulative nodes-examined = 7328
12: Cummulative nodes-examined = 12713
13: Cummulative nodes-examined = 21954
14: Cummulative nodes-examined = 38275
15: Cummulative nodes-examined = 66164
16: Cummulative nodes-examined = 114549
17: Cummulative nodes-examined = 197366
18: Cummulative nodes-examined = 342583
19: Cummulative nodes-examined = 590888
20: Cummulative nodes-examined = 1023321
21: Cummulative nodes-examined = 1762826
22: Cummulative nodes-examined = 3055675
23: Cummulative nodes-examined = 5266460
24: Cummulative nodes-examined = 9122181
25: Cummulative nodes-examined = 15715070
26: Cummulative nodes-examined = 19036504
Solution length = 26
(list
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'right #<procedure:...-1/program-1.rkt:138:14> #<procedure:...-1/program-1.rkt:141:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'down #<procedure:...-1/program-1.rkt:123:14> #<procedure:...-1/program-1.rkt:126:14>)
 (rule 'left #<procedure:...-1/program-1.rkt:110:14> #<procedure:...-1/program-1.rkt:113:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>)
 (rule 'up #<procedure:...-1/program-1.rkt:95:14> #<procedure:...-1/program-1.rkt:98:14>))
```

Sample Bread-First Search using Uniform-Cost-Search

```
------------
start:
7 2 4
5 0 6
8 3 1
goal:
0 1 2
3 4 5
6 7 8
Solution length = 27
Nodes generated = 451673
Nodes examined = 168884
'(#<node ((7 2 4) (5 0 6) (8 3 1)) [0]>
  #<node ((7 2 4) (0 5 6) (8 3 1)) [1]>
  #<node ((0 2 4) (7 5 6) (8 3 1)) [2]>
  #<node ((2 0 4) (7 5 6) (8 3 1)) [3]>
  #<node ((2 5 4) (7 0 6) (8 3 1)) [4]>
  #<node ((2 5 4) (7 3 6) (8 0 1)) [5]>
  #<node ((2 5 4) (7 3 6) (0 8 1)) [6]>
  #<node ((2 5 4) (0 3 6) (7 8 1)) [7]>
  #<node ((2 5 4) (3 0 6) (7 8 1)) [8]>
  #<node ((2 5 4) (3 6 0) (7 8 1)) [9]>
  #<node ((2 5 0) (3 6 4) (7 8 1)) [10]>
  #<node ((2 0 5) (3 6 4) (7 8 1)) [11]>
  #<node ((0 2 5) (3 6 4) (7 8 1)) [12]>
  #<node ((3 2 5) (0 6 4) (7 8 1)) [13]>
  #<node ((3 2 5) (6 0 4) (7 8 1)) [14]>
  #<node ((3 2 5) (6 4 0) (7 8 1)) [15]>
  #<node ((3 2 5) (6 4 1) (7 8 0)) [16]>
  #<node ((3 2 5) (6 4 1) (7 0 8)) [17]>
  #<node ((3 2 5) (6 0 1) (7 4 8)) [18]>
  #<node ((3 2 5) (6 1 0) (7 4 8)) [19]>
  #<node ((3 2 0) (6 1 5) (7 4 8)) [20]>
  #<node ((3 0 2) (6 1 5) (7 4 8)) [21]>
  #<node ((3 1 2) (6 0 5) (7 4 8)) [22]>
  #<node ((3 1 2) (6 4 5) (7 0 8)) [23]>
  #<node ((3 1 2) (6 4 5) (0 7 8)) [24]>
  #<node ((3 1 2) (0 4 5) (6 7 8)) [25]>
  #<node ((0 1 2) (3 4 5) (6 7 8)) [26]>)
```