Week 2 Project: Search Algorithms

Bookmark this page

ACADEMIC HONESTY

As usual, the standard honor code and academic honesty policy applies. We will be using automated **plagiarism detection** software to ensure that only original work is given credit. Submissions isomorphic to (1) those that exist anywhere online, (2) those submitted by your classmates, or (3) those submitted by students in prior semesters, will be detected and considered plagiarism.

INSTRUCTIONS

In this assignment you will create an agent to solve the **n-puzzle** game (i.e. the 8-puzzle game generalized to an n × n array). You may visit**mypuzzle.org/sliding** for a refresher of the rules of the game. You will implement and compare several search algorithms and collect some statistics related to their performances. Please read all sections of the instructions carefully:

I. Introduction
II. Algorithm Review
III. What You Need To Submit
IV. What Your Program Outputs
V. Important Information
VI. Before You Finish

NOTE: This project incorporates material learned from both **Week 2**(uninformed search) and **Week 3** (informed search). Since this project involves a fair amount of programming and design, we are releasing it now to let you get started earlier. In particular, do not worry if certain concepts (e.g. heuristics, A-Star, IDA-Star, etc.) are not familiar at this point; you will understand everything you need to know by Week 3.

I. Introduction

An instance of the n-puzzle game consists of a **board** holding n^2 − 1 distinct movable tiles, plus an empty space. The tiles are numbers from the set {1, …, n^2 − 1}. For any such board, the empty space may be legally swapped with any tile horizontally or vertically adjacent to it. In this assignment, we will represent the blank space with the number 0.

Given an initial **state** of the board, the combinatorial search problem is to find a sequence of moves that transitions this state to the goal state; that is, the configuration with all tiles arranged in ascending order ⟨0, 1, …, n^2 − 1⟩. The search space is the set of all possible states reachable from the initial state.

The blank space may be swapped with a component in one of the four directions {'Up', 'Down', 'Left', 'Right'}, one move at a time. The cost of moving from one configuration of the board to another is the same and equal to one. Thus, the total cost of path is equal to the number of moves made from the initial state to the goal state.

II. Algorithm Review

Recall from the lectures that searches begin by visiting the root node of the search tree, given by the initial state. Among other book-keeping details, three major things happen in sequence in order to visit a node:

- First, we **remove** a node from the frontier set.
- Second, we **check** the state against the goal state to determine if a solution has been found.
- Finally, if the result of the check is negative, we then **expand** the node. To expand a given node, we generate successor nodes adjacent to the current node, and add them to the frontier set. Note that if these successor nodes are already in the frontier, or have already been visited, then they should not be added to the frontier again.

This describes the life-cycle of a visit, and is the basic order of operations for search agents in this assignment—(1) remove, (2) check, and (3) expand. In this assignment, we will implement algorithms as described here. Please refer to lecture notes for further details, and review the lecture pseudocode before you begin the assignment.

IMPORTANT: Note that you may encounter implementations elsewhere that attempt to short-circuit this order by performing the goal-check on successor nodes immediately upon expansion of a parent node. For example, Russell & Norvig's implementation of breadth-first search does precisely this. Doing so may lead to edge-case gains in efficiency, but do not alter the general characteristics of complexity and optimality for each method. For simplicity and grading purposes in this assignment, **do not** make such modifications to algorithms learned in lecture.

III. What You Need To Submit

Your job in this assignment is to write `driver.py`, which solves any board when given an arbitrary starting configuration. The program will be executed as follows:

```
$ python driver.py <method> <board>
```

The method argument will be one of the following. You need to implement all four of them:

`bfs` (Breadth-First Search)
`dfs` (Depth-First Search)
`ast` (A-Star Search)
`ida` (IDA-Star Search)

The board argument will be a comma-separated list of integers containing no spaces. For example, to use the bread-first search strategy to solve the input board given by the starting configuration {0,8,7,6,5,4,3,2,1}, the program will be executed like so (with no spaces between commas):

```
$ python driver.py bfs 0,8,7,6,5,4,3,2,1
```

IMPORTANT: If you are using Python 3, please name your file `driver_3.py`, which will alert the grader to use the correct version of Python during submission and grading. If you name your file `driver.py`, the default version for our box is Python 2.

IV. What Your Program Outputs

When executed, your program will create / write to a file called `output.txt`, containing the following statistics:

`path_to_goal`: the sequence of moves taken to reach the goal
`cost_of_path`: the number of moves taken to reach the goal
`nodes_expanded`: the number of nodes that have been expanded
`fringe_size`: the size of the frontier set when the goal node is found
`max_fringe_size`: the maximum size of the frontier set in the lifetime of the algorithm
`search_depth`: the depth within the search tree when the goal node is found
`max_search_depth`: the maximum depth of the search tree in the lifetime of the algorithm
`running_time`: the total running time of the search instance, reported in seconds
`max_ram_usage`: the maximum RAM usage in the lifetime of the process as measured by the ru_maxrss attribute in the resource module, reported in megabytes

Example #1: Breadth-First Search

Suppose the program is executed for breadth-first search as follows:

```
$ python driver.py bfs 1,2,5,3,4,0,6,7,8
```

Which should lead to the following solution to the input board:

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

The output file (**example**) will contain **exactly** the following lines:

```
path_to_goal: ['Up', 'Left', 'Left']
cost_of_path: 3
nodes_expanded: 10
fringe_size: 11
max_fringe_size: 12
search_depth: 3
max_search_depth: 4
running_time: 0.00188088
max_ram_usage: 0.07812500
```

Example #2: Depth-First Search

Suppose the program is executed for depth-first search as follows:

```
$ python driver.py dfs 1,2,5,3,4,0,6,7,8
```

Which should lead to the following solution to the input board:

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & 5 \\ \hline 3 & 4 & \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & 2 & \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

$$parent = \begin{array}{|c|c|c|}\hline 1 & & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array} \implies child = \begin{array}{|c|c|c|}\hline & 1 & 2 \\ \hline 3 & 4 & 5 \\ \hline 6 & 7 & 8 \\ \hline \end{array}$$

The output file (**example**) will contain **exactly** the following lines:

```
path_to_goal: ['Up', 'Left', 'Left']
cost_of_path: 3
nodes_expanded: 181437
fringe_size: 2
max_fringe_size: 42913
search_depth: 3
max_search_depth: 66125
running_time: 5.01608433
max_ram_usage: 4.23940217
```

Note on Correctness

Of course, the specific values for `running_time` and `max_ram_usage` variables will vary greatly depending on the machine used and the specific implementation details; there is no "correct" value to look for. They are intended to enable you to check the time and space complexity characteristics of your code, and you should spend time to do so. All the other variables, however, will have **one and only one** correct answer for each algorithm and initial board specified in the sample test cases.* A good way to check the correctness of your program is to walk through small examples by hand, like the ones above.

* In general, **for any initial board whatsoever**, for BFS and DFS there is one and only one correct answer. For A* and IDA*, however, your output may vary a little, depending on specific implementation details. You will be fine as long as your algorithm conforms to all **specifications** listed in these instructions. As an example of an **unspecified** parameter, consider IDA-Star Search. The number of nodes expanded can vary depending on (1) the initial cost limit set for the first iteration, and (2) how much that limit is

incremented for each subsequent iteration. Depending on your choice of these parameters, a range of values for the`nodes_expanded` variable will be accepted as correct.

V. Important Information

Please read the following information carefully. Since this is the first programming project, we are providing many hints and explicit instructions. Before you post a clarifying question on the discussion board, make sure that your question is not already answered in the following sections.

1. Implementation

You will implement the following four algorithms as demonstrated in lecture. In particular:

- **Breadth-First Search**. Use an explicit queue, as shown in lecture.
- **Depth-First Search**. Use an explicit stack, as shown in lecture.
- **A-Star Search**. Use a priority queue, as shown in lecture. For the choice of heuristic, use the Manhattan priority function; that is, the sum of the distances of the tiles from their goal positions. Note that the blanks space is not considered an actual tile here.
- **IDA-Star Search**. As before, for the choice of heuristic, use the Manhattan priority function. Recall from lecture that implementing the Iterative Deepening Search (IDS) algorithm involves first implementing the Depth-Limited Search (DLS) algorithm as a subroutine. Similarly, implementing the IDA-Star Search algorithm involves first implementing a modified version of the DLS algorithm that uses the heuristic function in addition to node depth.

2. Order of Visits

In this assignment, where an arbitrary choice must be made, we always **visit** child nodes in the "**UDLR**" order; that is, ['Up', 'Down', 'Left', 'Right'] in that exact order. Specifically:

- **Breadth-First Search**. Enqueue in UDLR order; dequeuing results in UDLR order.
- **Depth-First Search**. Push onto the stack in reverse-UDLR order; popping off results in UDLR order.
- **A-Star Search**. Since you are using a priority queue, what happens when there are duplicate keys? What do you need to do to ensure that nodes are retrieved from the priority queue in the desired order?
- **IDA-Star Search**. For each iteration, you can handle node ordering as you would in depth-first search.

3. Hint for IDA-Star Search

Recall from lecture that DFS is not optimal. Is DLS optimal? How does IDS achieve optimality if DLS is not? In particular, recall that the general search algorithm keeps track of nodes that have been explored. Do we ever remove nodes from the explored set? For each iteration of DLS, how does this affect the optimality of the overall IDS algorithm? How should we handle the explored set differently? By the same token, consider what must be done to ensure that your IDA-Star algorithm is optimal.

4. Submission Test Cases

You can **submit** your project any number of times before the deadline. Only the final submission will be graded. Following each submission, all four of your algorithms will be automatically run on two sample test cases each, for a total of **8** distinct tests:

Test Case #1

```
python driver.py bfs 3,1,2,0,4,5,6,7,8
python driver.py dfs 3,1,2,0,4,5,6,7,8
python driver.py ast 3,1,2,0,4,5,6,7,8
python driver.py ida 3,1,2,0,4,5,6,7,8
```

Test Case #2

```
python driver.py bfs 1,2,5,3,4,0,6,7,8
python driver.py dfs 1,2,5,3,4,0,6,7,8
python driver.py ast 1,2,5,3,4,0,6,7,8
python driver.py ida 1,2,5,3,4,0,6,7,8
```

This is provided as a sanity check for your code and the required output format. In particular, this is intended to ensure that you do not lose credit for incorrect output formatting. **Make sure your code passes at least these two test cases**. You will see that the results of each test are assessed out of ten points: one point for the presence / correctness of each output variable, plus one point for code that executes and produces any output at all.

5. Grading and Stress Tests

After the project deadline has passed, we will grade your project by running **additional test cases** on your code. In particular, there will be five test cases in total, each tested on all four of your algorithms, for a total of **20** distinct tests. Similar to the submission test cases, each test will be graded out of ten points, for a total of 200 points. We will be using a wide variety of inputs to stress-test your algorithms to check for correctness of implementation. So, we recommend that you test your own code extensively.

Do not worry about checking for **malformed input** boards, including boards of non-square dimensions, or boards composed of improper tiles; that is, exactly one of each integer in the set {1, …, n^2 − 1}.

You will not be graded on the absolute values of your running-time or RAM usage statistics. The values of these statistics can vary widely depending on the machine. **However, we recommend that you take advantage of them in testing your code.** Try batch-running your algorithms on various inputs, and plotting your results on a graph to learn more about the space and time complexity characteristics of your code. Just because an algorithm provides the correct path to goal does not mean it has been implemented correctly...!

6. Tips on Getting Started

Begin by writing a class to represent the **state** of the game at a given turn, including parent and child nodes. We suggest writing a separate **solver** class to work with the state class. Feel free to experiment with your design, for example including a **board** class to represent the low-level physical configuration of the tiles, delegating the high-level functionality to the state class.

You will not be graded on your design, so you are at a liberty to choose among your favorite programming paradigms. Students have successfully completed this project using an entirely object-oriented approach, and others have done so with a purely functional approach. Your submission will receive full credit as long as your driver program outputs the correct information.

VI. Before You Finish

- **Make sure** your code passes at least the two submission test cases.
- **Make sure** your algorithms generate the correct solution for an arbitrary problem instance and an arbitrary board size. In particular, it should work for any size n.
- **Make sure** your program always terminates without error, and in a reasonable amount of time. You will receive zero points from the grader if your program fails to terminate. For a 3 x 3 input board, running times of more than a minute or two may indicate a problem with your implementation. If your implementation exceeds the time limit allocated (1 hour for all test cases), your grade may be incomplete.
- **Make sure** your program output follows the specified format exactly. In particular, for the path to goal, use square brackets to surround the list of items, use single quotes around each item, and capitalize the first letter of each item. Round floating point numbers to 8 places after the decimal.

You will not receive proper credit from the grader if your format differs from the provided examples above.
