Project 1 CS205: Artificial Intelligence

Tyson Loveless Dr. Eamonn Keogh
ID 861118716
tlove004@ucr.edu
16-February-2018

In completing this assignment I consulted:

- The lecture slides on Uninformed Search and Informed Search
- Analyzed puzzles come from the project handout as well as various online resources.[1,2]
- The Wikipedia page for the 15-puzzle: `https://en.wikipedia.org/wiki/15_puzzle`
- Python 2.7.10 documentation.[3]
- For checking the solvability of a puzzle: `https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html`

All important code is original. Unimportant subroutines that are not completely original are:

- The `heappop` and `heappush` functions from Python's **heapq** library for maintaining the priority queue in the general search function
- The `split` functions from Python's regular expression library (**re**) to parse user's input
- The `ceil` and `sqrt` functions from Python's **math** library

[1] Random puzzle generator for 8-puzzles: `https://www.jaapsch.net/puzzles/javascript/fifteenj.htm`
[2] Max depth: `http://w01fe.com/blog/2009/01/the-hardest-eight-puzzle-instances-take-31-moves-to-solve/`
[3] Python 2.7.10 docs: `https://docs.python.org/release/2.7.10/`

1 Introduction

In this project, we were tasked with solving the 8-puzzle using three related searches: uniform cost search, A* search with the misplaced tile heuristic, and A* search with the manhattan distance heuristic. The 8-puzzle is a simple game in which a 3x3 grid of numbers 1-8 (one blank) are mixed and can be slid to reorient their positions. The goal is to order the numbers, with the blank space appearing in the 9th position. In order to evaluate and compare the effectiveness of the algorithms, several test cases (initial puzzle states) were solved with each algorithm. The remainder of the report is organized as follows: section 2 discusses the different algorithms being compared, section 3 presents details on the implementation, section 4 summarizes the results, and section 5 gives concluding remarks on the project. Appendix A lists the tested puzzles, Appendix B has the requested trace, and Appendix C contains the a subset of the source code developed for the project.[4]

2 Evaluated Algorithms

To try to get a good understanding of how the same basic algorithm can be improved through heuristics, we evaluate three variations of A*. As A*'s evaluation function is simply $f(n) = g(n) + h(n)$ where $g(n)$ is the depth cost and $h(n)$ is the heuristic cost.

2.1 Uniform Cost Search

Uniform Cost Search (UCS) is a form of blind search, relying only on the expansion cost to choose which nodes to traverse. Since UCS does not have any heuristic guiding the search, we define the evaluation of UCS as $f(n) = g(n)$. As $g(n)$ for this problem is simply the depth, UCS is essentially breadth first search, so it works as a good complete and optimal baseline to compare the two heuristic versions of A* against.

2.2 A* with Misplaced Tiles

As discussed, A* evaluates which nodes to expand based on a combination of depth cost ($g(n)$) and heuristic cost ($h(n)$) at a given node n. In this version of A*, $h(n)$ is equal to the number of *misplaced tiles* (MT). The value is simply found by counting the number of tiles that are not in the correct (goal) position. We can verify that this heuristic is admissible by observing that it can never overestimate how good the current state of the puzzle is, as a MT means that it is 1 *or more* moves away from being in the correct position. As we have a bit of information about the search space while searching, the heuristic helps to guide the expansion of nodes, hopefully reaching the goal state faster than UCS.

2.3 A* with Manhattan Distance

This version of A* defines a different $h(n)$: the Manhattan Distance (MD) for each misplaced tile. As with the MT heuristic, the MD heuristic evaluates how good the current state of the puzzle is before choosing which node to expand. MD is defined as the sum of the difference in row and column distance from the goal. For example,

[4] To save trees – for the complete code, visit: `https://github.com/tlove004/8-puzzle.git`

```
1  2  3
*  5  6
7  8  4
```

has a total MD of 3, where the 4 tile is 1 row and 2 columns from the goal, while all other tiles are in their goal positions. Again, it is easy to see that MD is an admissible heuristic by observing how the estimated distance to goal relates to the actual distance. For the state of the given puzzle above, we can easily see that 3 moves would not actually solve this puzzle, as other tiles would be displaced from their positions in the meantime (in fact, the solution to this puzzle requires at least 11 moves). As we will see, this heuristic works much better than the MT heuristic in most cases in both time and space.

3 Details on Implementation

The implementation is done in Python 2.7.10 with the referenced libraries. As each of the three searches are simple variations of a general A* search, the three searches are simple modular adaptations of the same function, with UCS setting $h(n)$ to 0 and the MT and MD heuristics evaluated as described. Also, in order to keep the tests productive, an algorithm for determining solvability is utilized.[5,6] This helped when testing random puzzles to immediately return a message that the test is unsolvable, rather than waiting to expand all possible states.[7] As the diameter is known for the 8-puzzle and 15-puzzle, these are included to limit the search depth depending on the size of the given puzzle.[8]

The general search function (implemented in `general_search.py`) is modeled after the pseudocode given in the slides and uses a priority queue with the value of $f(n)$ dictating how child nodes are enqueued. The state of a puzzle is defined in `puzzle.py`, along with definitions of valid operators, the MD and MT evaluation functions, and helper functions for checking if a puzzle is solvable and generating a default goal puzzle for a given puzzle size. `eight_puzzle.py` implements the user interface to build and search a puzzle. A user is given the option to use a default puzzle, enter a custom puzzle, or change the size of the puzzle,[9] then given the option to use the default goal state (the blank being in the final lower right position), or to enter a custom goal state, and finally given the option between the three search functions implemented. If the puzzle is solvable, or if you are using a custom goal state, the user interface outputs how nodes are being expanded as the search progresses, providing the values of $g(n)$ and $h(n)$ during each expansion. If a solution is found, the total number of expanded nodes, and maximum number of nodes within the queue during the search as well as the depth to the goal is displayed. Finally, the user is able to print out the solution trace to see the optimal solution to the puzzle.

[5] From `https://www.cs.bham.ac.uk/~mdr/teaching/modules04/java2/TilesSolvability.html`

[6] For an odd-edged puzzle, there must be an even number of inversions for solvability; even-edged puzzles require checking even or odd inversions depending on if the blank is in an even or odd row. See `puzzle.py`

[7] There are $9! = 362,880$ unique possible configurations of the 8-puzzle, of which only half are solvable

[8] The 8-puzzle takes no more than 31 moves to solve in the worst case, while the 15-puzzle takes no more than 80 moves. The diameter for the 24-puzzle is not known, but an upper bound of 208 has been established, and is used as the diameter for 24-puzzle instances (see `https://en.wikipedia.org/wiki/15_puzzle#solvability`)

[9] 8-, 15-, and 24-puzzles are supported, with expansion to larger $n^2 - 1$ puzzles easily accomplished

4 Results

4.1 Interpreting the Results

Using several initial puzzle states of differing solution depths[10], we compare and contrast the time(number of nodes expanded) and space(maximum number of nodes in the queue at a given time) complexity for each algorithm and come to a conclusion on the best algorithm for the puzzle configuration.

SEARCH	UNIFORM COST SEARCH		MISPLACED TILE HEURISTIC		MANHATTAN DISTANCE HEURISTIC	
NAMED TEST	# EXPANDED	MAX QUEUE SIZE	# EXPANDED	MAX QUEUE SIZE	# EXPANDED	MAX QUEUE SIZE
Impossible (inf)	0	0	0	0	0	0
Trivial (0)	0	0	0	0	0	0
Very Easy (1)	2	1	2	1	2	1
Easy (2)	6	3	4	2	4	2
Doable (4)	16	8	7	3	7	3
Oh Boy (22)	141246	39537	16899	6301	977	377
test1 (2)	12	7	6	4	6	4
test2 (11)	1396	578	147	61	100	43
test3 (20)	91376	31014	7428	2860	1021	398
test4 (28)	458662	71271	143920	40739	1160	3239
31 (31)	519140	73071	280324	56374	44995	15013
31-2 (31)	519144	73067	276668	56059	42410	14442

Table 1: Results of Various Test Puzzles (Solution depths in parenthesis)

Table 1 summarizes the results. Note that the puzzle named `Impossible` and the puzzle named `Trivial` do not expand any nodes and never enqueue the initial state. This is because, prior to searching, we test if the puzzle is solvable and if the puzzle is already in the goal state. From these results, it is easy to see that the heuristic searches perform orders of magnitude better than the blind uniform cost search. Furthermore, the Manhattan Distance heuristic significantly outperforms the Misplaced Tiles heuristic as expected. The reason this is the case is intuitive: the MD heuristic provides more information to the search function than the MT heuristic. While the MT heuristic counts the number of tiles needed to be arranged, the MD heuristic gives a sense of *how far* a tile is from its goal.

Figure 1 gives some visual clarity to the data. Each of the 11 test puzzles have bar graphs showing the number of expanded nodes and scatter plot points showing the maximum number of nodes in the queue. The results clearly show the heuristic searches outperforming UCS across the board.

[10]See Appendix A to view the initial states of the tested puzzles

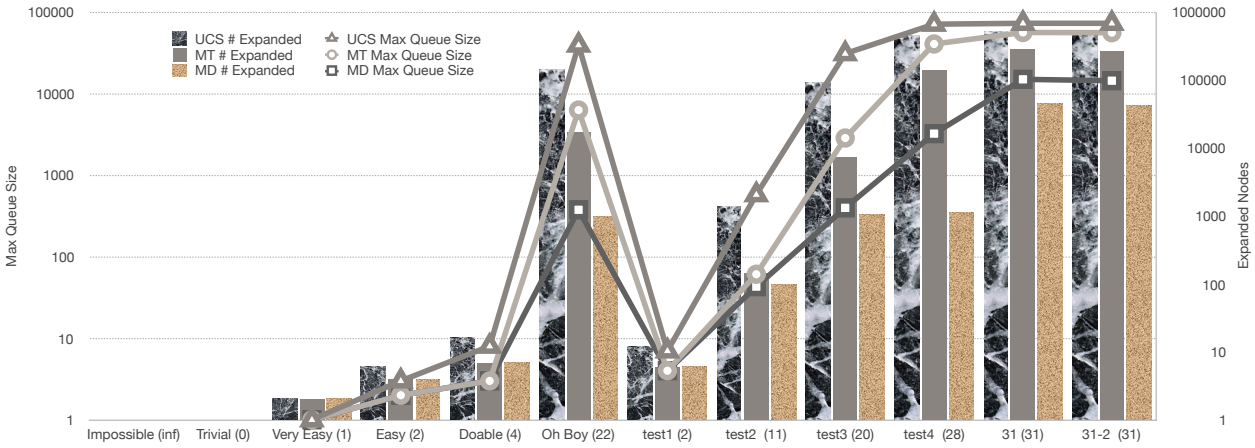

Figure 1: Expanded Nodes and Maximum # of Nodes in Queue for each Puzzle

5 Conclusion

This project compared three search algorithms' usefulness in solving the 8-puzzle. The results clearly showed that admissible heuristics drive faster convergence to the optimal solution, while also requiring less space than Uniform Cost Search. Between the two heuristics used for the A* algorithm, it is clear that the additional information computed with the Manhattan Distance goes a long way to significantly outperform the Misplaced Tile heuristic. This was magnified on search spaces that had a deeper optimal solution.

Appendix A - The Test Puzzles

Trivial	1	2	3	Very	1	2	3	Easy	1	2	0
	4	5	6	Easy	4	5	6		4	5	3
	7	8	9		7	0	8		7	8	6
Doable	0	1	2	Oh Boy	8	7	1	Impossible	1	2	3
	4	5	3		6	0	2		4	5	6
	7	8	6		5	4	3		8	7	0
31-1	6	4	7	31-2	8	6	7	test1	1	2	3
	8	5	0		2	5	4		4	0	6
	3	2	1		3	0	1		7	5	8
test2	1	2	3	test3	4	2	8	test4	0	8	7
	4	5	7		6	0	3		6	5	4
	8	0	6		7	5	1		3	2	1

Appendix B - Requested Trace

Below is the trace for the `test1` puzzle using the MD heuristic.

```
Welcome to Tyson Loveless' 8-puzzle solver.
Type "1" to use a default puzzle, "2" to enter your own puzzle, or "3" to change
   the puzzle size: 1

Please select a puzzle difficulty between 0 and 11: 4

Here is the chosen puzzle:
       1  2  3
       4  *  6
       7  5  8

Enter "1" for the standard goal state or "2" to search for a specific goal:  1

We are looking for this goal:
       1  2  3
       4  5  6
       7  8  *
     Enter your choice of algorithm
         1. Uniform Cost Search
         2. A* with the Misplaced Tile heuristic.
         3. A* with the Manhattan distance heuristic.

         3
The best state to expand with a g(n) = 0 and h(n) = 2 is...
       1  2  3
       4  *  6
       7  5  8
```

```
         Expanding this node...
The best state to expand with a g(n) = 1 and h(n) = 1 is...
       1  2  3
       4  5  6
       7  *  8
         Expanding this node...

Goal!!

To solve this problem the search algorithm expanded a total number of 6 nodes.
The maximum number of nodes in the queue at any one time was 4
The depth of the goal node was 2
 Print Solution Trace? y
Expanding node with g(n) = 0 and h(n) =  2
       1  2  3
       4  *  6
       7  5  8
...
Expanding node with g(n) = 1 and h(n) =  1
       1  2  3
       4  5  6
       7  *  8
...
       1  2  3
       4  5  6
       7  8  *

Goal!!

To solve this problem the search algorithm expanded a total number of 6 nodes.
The maximum number of nodes in the queue at any one time was 4
The depth of the goal node was 2
  Done.
```

Appendix C - Implementation Example

general_search.py

```
# format for search taken from project prompt, all functions developed by Tyson Loveless

from heapq import heappush, heappop # for priority queue structure
import puzzle

. . . some functions skipped to save trees . . .


# general form of search function based on project instructions
def search(problem, function):
    global diameter
    global TOTAL_EXPANDED
    global MAX_QUEUE_SIZE
    # in case you enter a custom puzzle that is already solved
    if problem.GOAL_TEST(problem.INITIAL_STATE.STATE):
        print "\n\nThe puzzle isn't mixed up yet!"
        return problem.INITIAL_STATE, TOTAL_EXPANDED, MAX_QUEUE_SIZE
```

```
nodes = []   # our priority queue
closed = {}  # states that have been traversed
heappush(nodes, [float('inf'), problem.INITIAL_STATE])    # root node pushed to pqueue
while True:
    if not nodes: # nodes is empty, no soln found, we lose
        return 0, 0, 0
    # keep track of max queue size
    MAX_QUEUE_SIZE = max(MAX_QUEUE_SIZE, nodes.__len__())
    # queue is popped with highest priority first (least cost)
    cost, node = heappop(nodes)
    closed[tuple(node.STATE)] = True
    if printing:
        if function is 1:
            print "The best state to expand with a g(n) =
                     %d and h(n) = %d is..." % (node.DEPTH, 0)
        elif function is 2:
            print "The best state to expand with a g(n) =
                     %d and h(n) = %d is..." % (node.DEPTH, node.mis())
        else:
            print "The best state to expand with a g(n) =
                     %d and h(n) = %d is..." % (node.DEPTH, node.man())
        printPuzzle(node.STATE)
        print '          Expanding this node...'
    else:
        if function is 2:
            node.mis()
        elif function is 3:
            node.man()
    # for every child to be expanded, we check if its state has already been visited
    #   if so, we do nothing, otherwise we push it to our priority queue based on our
    #   given queueing function heuristics
    for child in expand(node, problem.OPERATORS):
        if tuple(child.STATE) not in closed:
            if child.DEPTH <= diameter:
                if function is 1:
                    heappush(nodes, [child.DEPTH, child])
                elif function is 2:
                    heappush(nodes, [child.DEPTH + child.mis(), child])
                else:
                    heappush(nodes, [child.DEPTH + child.man(), child])
                TOTAL_EXPANDED += 1
        # success is checked once a node is expanded, to avoid expanding extra nodes
        if problem.GOAL_TEST(child.STATE):
            return child, TOTAL_EXPANDED, MAX_QUEUE_SIZE
```