Dear Students

Below is an anonymized sample of an eight-puzzle project report.

This was a very nice report, earning the student an A.

I am *not* claiming this report is perfect, or that it is the *only* way to do a high-quality project. It is simply an example of high-quality work.

Note that the student created the report in LaTex, in converting to MS word, we slightly messed up the very neat original formatting.

Minor points:

- The students output trace is neat, but not in the *exact* format I requested. So I took away some points!
- For every Figure or Table in a report, there needs to be some text in the body of the report that explicitly points to it, and interprets it. The next sentence is a sample. As we can see in Figure 1, the Fowl Heuristic is much faster than the Fish heuristic, especially as we consider harder problems.
- Look at your figures carefully. Did you label the X-axis and the Y-axis? Does your figure work in B/W or do you need a color printout?
- Do you have an *explicit* conclusion to your report? As we have discovered empirically, the cost of owning a dog is approximately 240% the cost of owning a cat, over the life of the animal. However this cost gap close for smaller dog breeds.

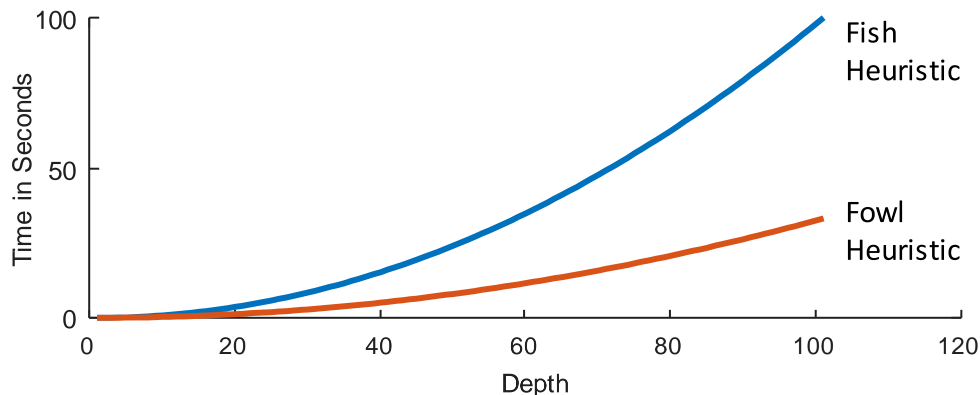


Figure 1: A comparison of two heuristic on the Rubix Sphere Problem, for increasingly hard problems.

Assignment 1 CS170: Introduction to Artificial Intelligence
Lorem Ipsum Dr. Eamonn Keogh
SID
Email
3-11-2017

In completing this assignment I consulted:

- The Blind Search and Heuristic Search lecture slides and notes annotated from lecture.

- Python 2.7.14, 3.5, and 3.6 Documentation. This is the URL to the Table of Contents of 2.7.14: https://docs.python.org/2/contents.html

- For the randomly generated puzzles: http://www.puzzlopia.com/puzzles/puzzle-8/play

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

- All subroutines used from **heapq**, to handle the node structure of states.

- All subroutines used from **copy**, to deepcopy and correctly modify states.

CS170: Assignment 1 Write Up

Lorem Ipsum, SID 12345678

Introduction

This assignment is the first project in Dr. Eamonn Keogh's Introduction to AI course at the University of California, Riverside during the quarter of Fall 2017. The following write up is to detail my findings through the course of project completion.
It explores Uniform Cost Search, and the Misplaced Tile and Manhattan Distance heuristics applied to A*. My language of choice was Python (version 3), and the full code for the project is included.

Comparison of Algorithms

The three algorithms implemented are as follows: Uniform Cost Search, A* using the Misplaced Tile heuristic, and A* using the Manhattan Distance heuristic.

Uniform Cost Search

As noted in the initial assignment prompt, **Uniform Cost Search** is simply A* with h(n) hardcoded to 0, and it will only expand the cheapest node, whose cost is in g(n). In the case of this assignment, there are no weights to the expansions, and each expanded node will have a cost of 1.

The Misplaced Tile Heuristic

The second algorithm implemented is A* with the **Misplaced Tile Heuristic**. The heuristic looks to the number of "misplaced" tiles in a puzzle. For example:

2

```
A puzzle:
[[1, 2, 4],
 [3, 0, 6],
 [7, 8, 5]]

goal state:
[[1, 2, 3],
 [4, 5, 6],
 [7, 8, 0]]
```

Not counting 0 (the placeholder for the blank/missing tile), g(n) is set to the number of tiles not in their current goal state position are counted; in this example, g(n) = 3. This assigns a number, where lower is better, to node expansion based on how many misplaced tiles there are after any given position change of the space. When applied to the n-puzzle, queue will expand the node with the cheapest cost, rather than expanding each of the child nodes as Uniform Cost Search would.

The Manhattan Distance Heuristic

The **Manhattan Distance Heuristic** is similar to the Misplaced Tile Heuristic such that it considers the cost of future expansions and looks at misplaced tiles, but has a different rationale to it. The heuristic considers all of the misplaced tiles *and* the number of tiles away from its goal state position would be. The resulting g(n) is the sum of all the cost of all misplaced tile distances.

Using the same example above, not counting the position of 0, it can be seen that tiles 4, 3, and 5 are out of place. Based on their positions in the puzzle and their goal state positions, g(n) = 8.

Comparison of Algorithms on Sample Puzzles

There were six puzzles of varying difficulty given to implement. The easiest of the six is a trivial puzzle (the puzzle being the goal state) and the hardest puzzle is impossible to solve (the goal state, but the position of tiles 7 and 8 swapped). The puzzle configurations themselves can be seen in nPuzzle.py. See Figure 1 (page 3) and Figure 2 (page 4) for a visual representation of the number of nodes expanded and the maximum queue size, respectively.

It was found that the difference between the three algorithms was relatively negligible when given easier puzzles, but the heuristics (and how good the heuristic was) made a significant difference in the space complexity when solving more difficult but still solvable puzzles.

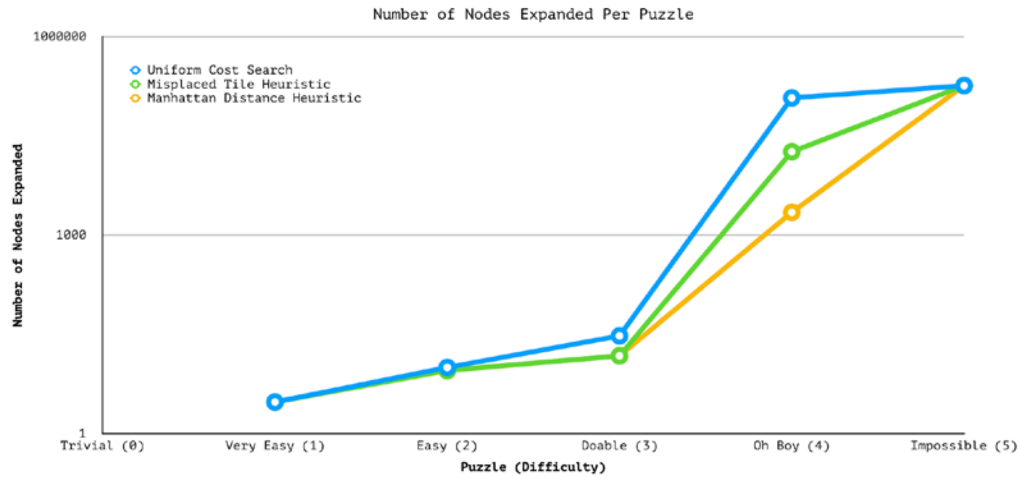
Additional Examples

For the sake of comparison, I have a few made up puzzles, and run each of the algorithms on them. See Figures 3 (page 5) and 4 (page 6) for a comparison of the number of nodes expanded, and the maximum queue size reached.

```
Puzzle 1:
[[5, 1, 3],
 [8, 6, 0],
 [2, 7, 4]]

Puzzle 2:
[[4, 8, 0],
 [6, 5, 7],
 [3, 2, 1]]

Puzzle 3:
[[3, 5, 8],
 [4, 2, 6],
 [0, 1, 7]]

Puzzle 4:
[[5, 1, 8],
 [2, 4, 6],
 [7, 3, 0]]

Puzzle 5:
[[5, 1, 3],
 [8, 6, 0],
 [2, 7, 4]]
```

Number of Nodes Expanded

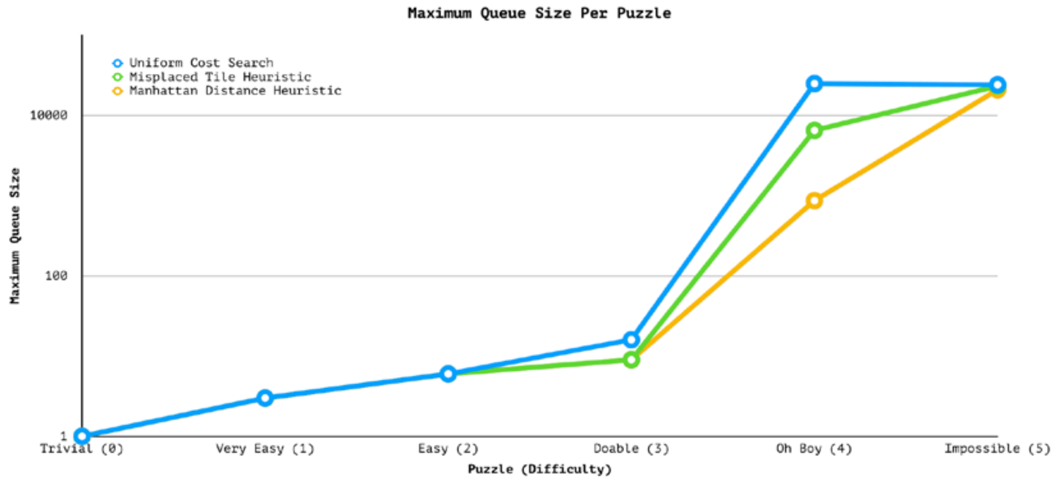
	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial (0)	0	0	0
Very Easy (1)	3	3	3
Easy (2)	10	9	9
Doable (3)	30	15	15
Oh Boy (4)	118332	18168	2205
Impossible (5)	181439	181439	181439



The Number of Nodes Expanded, Preset Puzzles

Maximum Queue Size

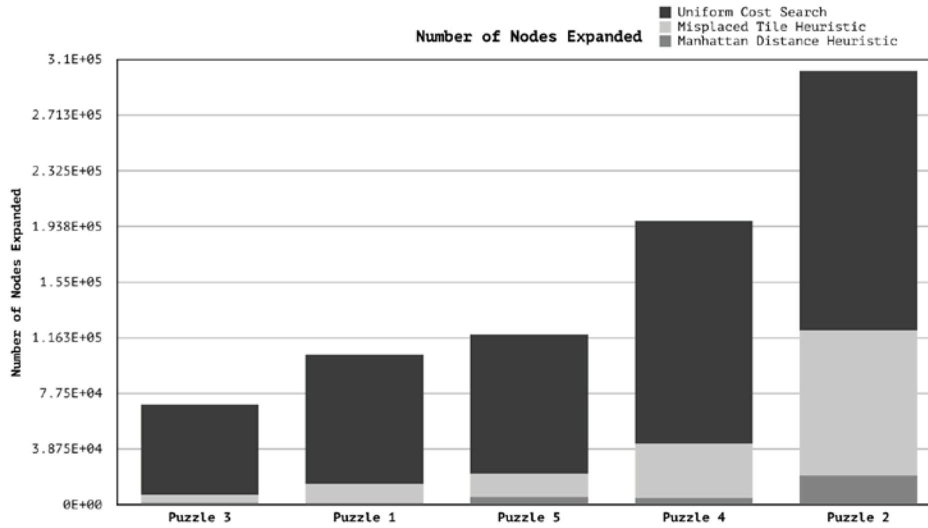
	Uniform Cost Search	Misplaced Tile Heuristic	Manhattan Distance Heuristic
Trivial (0)	1	1	1
Very Easy (1)	3	3	3
Easy (2)	6	6	6
Doable (3)	16	9	9
Oh Boy (4)	24969	6519	868
Impossible (5)	24188	23314	20922



The Maximum Queue Size, Preset Puzzles

Number of Nodes Expanded

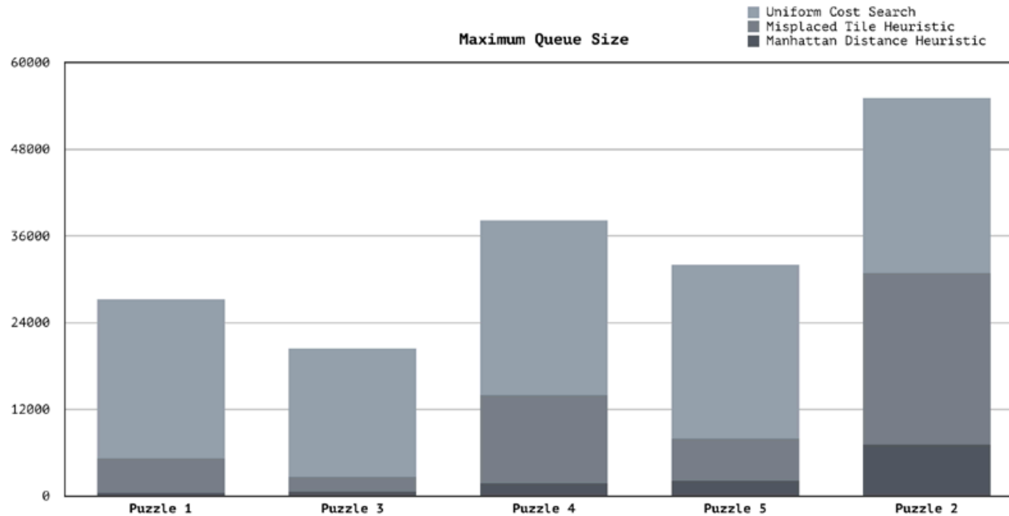
	Manhattan Distance Heuristic	Misplaced Tile Heuristic	Uniform Cost Search
Puzzle 3	1507	5340	62698
Puzzle 1	1144	13307	90374
Puzzle 5	5553	16166	96901
Puzzle 4	4608	38054	154909
Puzzle 2	20317	100978	180949



The Number of Nodes Expanded, Randomly Generated Puzzles

Maximum Queue Size

	Manhattan Distance Heuristic	Misplaced Tile Heuristic	Uniform Cost Search
Puzzle 1	454	4789	21983
Puzzle 3	614	2024	17861
Puzzle 4	1788	12202	24188
Puzzle 5	2106	5908	23963
Puzzle 2	7153	23713	24230



The Maximum Queue Size, Randomly Generated Puzzles

Conclusion

Considering the list of the three algorithms and the comparisons between them: Uniform Cost Search, Misplaced Tiles, and Manhattan Distance, it can be said that:

- It can be seen that out of the three algorithms, the Manhattan Distance Heuristic performed the best, followed by the Misplaced Tiles Heuristic, followed by Uniform Cost Search (or in this case, effectively also called Breadth-First Search).
- The Misplaced Tile and Manhattan Distance heuristics improve the efficiency of algorithms. Uniform Cost Search, h(n) having been hardcoded to 0, became Breadth First Search, which has a time complexity of $O(b^d)$ and also a space complexity of $O(b^d)$, where b is the branching factor and d is the depth of the solution in the search tree.

- While both the Misplaced Tile Heuristic and Manhattan Distance Heuristic improved the run time and space cost of Uniform Cost Search, it is clear that the Manhattan Distance Heuristic performed better between the two. It can be concluded that while a relevant heuristic will perform better than a blind search, not all heuristics are made equal.

The following is a traceback of a given puzzle and requested algorithm, detailed in the assignment specifications.

```
Welcome to my 170 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create
your own.
2
Enter your puzzle, using a zero to represent the blank. Please only enter valid
8-puzzles. Enter the puzzle demilimiting the numbers with a space. RET only when
finished.

Enter the first row: 1 2 3
Enter the second row: 4 0 6
Enter the third row: 7 5 8
Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile
Heuristic, or (3) the Manhattan Distance Heuristic.
3


[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

[1, 2, 3]
[4, 6, 0]
[7, 5, 8]

[1, 2, 3]
[4, 0, 6]
[7, 5, 8]


Number of nodes expanded: 13
Max queue size: 8
```

nPuzzle.py

```
import TreeNode
import heapq as min_heap_esque_queue  # because it sort of acts like a min heap

trivial = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 0]]
veryEasy = [[1, 2, 3],
            [4, 5, 6],
            [7, 0, 8]]
easy = [[1, 2, 0],
        [4, 5, 3],
        [7, 8, 6]]
doable = [[0, 1, 2],
          [4, 5, 3],
          [7, 8, 6]]
oh_boy = [[8, 7, 1],
         [6, 0, 2],
         [5, 4, 3]]
impossible = [[1, 2, 3],
              [4, 5, 6],
              [8, 7, 0]]

eight_goal_state = [[1, 2, 3],
                    [4, 5, 6],
                    [7, 8, 0]]


def main():
    puzzle_mode = input("Welcome to an 8-Puzzle Solver. Type '1' to use a default puzzle, or '2' to create your own."
                        + '\n')
    if puzzle_mode == "1":
        select_and_init_algorithm(init_default_puzzle_mode())

    if puzzle_mode == "2":
        print("Enter your puzzle, using a zero to represent the blank. " +
              "Please only enter valid 8-puzzles. Enter the puzzle demilimiting " +
              "the numbers with a space. RET only when finished." + '\n')
        puzzle_row_one = input("Enter the first row: ")
        puzzle_row_two = input("Enter the second row: ")
        puzzle_row_three = input("Enter the third row: ")

        puzzle_row_one = puzzle_row_one.split()
        puzzle_row_two = puzzle_row_two.split()
        puzzle_row_three = puzzle_row_three.split()

        for i in range(0, 3):
            puzzle_row_one[i] = int(puzzle_row_one[i])
            puzzle_row_two[i] = int(puzzle_row_two[i])
            puzzle_row_three[i] = int(puzzle_row_three[i])

        user_puzzle = [puzzle_row_one, puzzle_row_two, puzzle_row_three]
        select_and_init_algorithm(user_puzzle)

    return


def init_default_puzzle_mode():
    selected_difficulty = input(
        "You wish to use a default puzzle. Please enter a desired difficulty on a scale from 0 to 5." + '\n')
    if selected_difficulty == "0":
        print("Difficulty of 'Trivial' selected.")
        return trivial
    if selected_difficulty == "1":
        print("Difficulty of 'Very Easy' selected.")
        return veryEasy
```

```
    if selected_difficulty == "2":
        print("Difficulty of 'Easy' selected.")
        return easy
    if selected_difficulty == "3":
        print("Difficulty of 'Doable' selected.")
        return doable
    if selected_difficulty == "4":
        print("Difficulty of 'Oh Boy' selected.")
        return oh_boy
    if selected_difficulty == "5":
        print("Difficulty of 'Impossible' selected.")
        return impossible


def print_puzzle(puzzle):
    for i in range(0, 3):
        print(puzzle[i])
    print('\n')


def select_and_init_algorithm(puzzle):
    algorithm = input("Select algorithm. (1) for Uniform Cost Search, (2) for the Misplaced Tile Heuristic, "
                      "or (3) the Manhattan Distance Heuristic." + '\n')
    if algorithm == "1":
        uniform_cost_search(puzzle, 0)
    if algorithm == "2":
        uniform_cost_search(puzzle, 1)
    if algorithm == "3":
        uniform_cost_search(puzzle, 2)


def uniform_cost_search(puzzle, heuristic):

    starting_node = TreeNode.TreeNode(None, puzzle, 0, 0)
    working_queue = []
    repeated_states = dict()
    min_heap_esque_queue.heappush(working_queue, starting_node)
    num_nodes_expanded = 0
    max_queue_size = 0
    repeated_states[starting_node.board_to_tuple()] = "This is the parent board"

    stack_to_print = []  # the board states are stored in a stack

    while len(working_queue) > 0:
        max_queue_size = max(len(working_queue), max_queue_size)
        # the node from the queue being considered/checked
        node_from_queue = min_heap_esque_queue.heappop(working_queue)
        repeated_states[node_from_queue.board_to_tuple()] = "This can be anything"
        if node_from_queue.solved():  # check if the current state of the board is the solution
            while len(stack_to_print) > 0:  # the stack of nodes for the traceback
                print_puzzle(stack_to_print.pop())
            print("Number of nodes expanded:", num_nodes_expanded)
            print("Max queue size:", max_queue_size)
            return node_from_queue

        stack_to_print.append(node_from_queue.board)
```

Note: I deleted the rest of the code (so it cannot be trivially copied) The full code took 240 lines, including the above, and including blank lines for readability.