Dear Students

Below is sample of an eight-puzzle project report. This looks like a nice report, it would earn the student an A or 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 what high-quality work might look like.

Notes:

- 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.

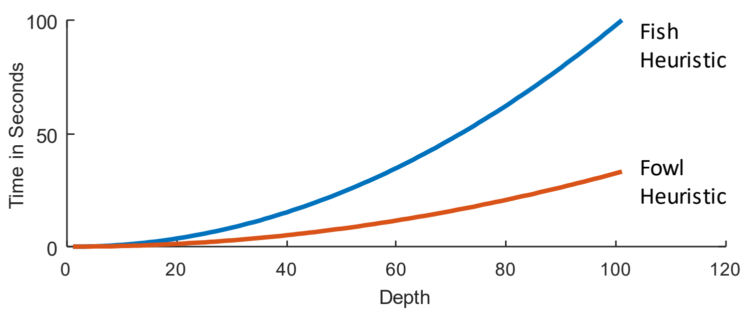

Figure 1: A comparison of two heuristics on the Rubix Sphere Problem, for increasingly hard problems.

- 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 closeer for smaller dog breeds.
- If you copy sentence from the internet or a book, without attribution, you will get a failing grade in this class.
    - **NOT ALLOWED**: Dr. Keogh asked us to create a program to solve a 3 by 3 sliding tile puzzle. A sliding puzzle is a combination puzzle that challenges a player to slide pieces along certain routes to establish a certain end-configuration. I begin by… This student will get a failing grade in the class. If you want to use someone else words or even long phrases, you must cite them. Put the text in quotation marks, and put a pointer (like this [1] to where you found it.
    - **ALLOWED**: Dr. Keogh asked us to create a program to solve a 3 by 3 sliding tile puzzle. According to Edward Hordern "*A sliding puzzle, is a combination puzzle that challenges a player to slide pieces along certain routes to establish a certain end-configuration*" [1]. I begin…

[1] Sliding Piece Puzzles (by Edward Hordern, 1986, Oxford University Press, ISBN 0-19-853204-0)

Assignment 1
Lorem Ipsum
SID
Email sue@hmail.com
3-11-2017

CS170: Introduction to Artificial Intelligence
Dr. Eamonn Keogh

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.

Outline of this report:

- Cover page: (this page)
- My report: Pages 2 to 7.
- Sample trace on an easy problem, page 8.
- Sample trace on a hard problem, page 9.
- My code pages 10 to 11. Note that in case you want to run my code, here is a URL that points to it online *http:Github.ucr.joe/cs150*

CS170: Assignment 1: The eight-puzzle

Lorem Ipsum, SID 12345678 Feb-16-2021

Introduction

Sliding tile puzzles, as shown in Figure 1, are familiar mechanical toys. The 8-puzzle is a smaller version of the slightly better known 15-puzzle. The puzzle consists of an area divided into a grid, 3 by 3 for the 8-puzzle, 4 by 4 for the 15-puzzle. On each grid square is a tile, expect for one square which remains empty. Thus, there are eight tiles in the 8-puzzle and 15 tiles in the 15-puzzle. A tile that is next to the empty grid square can be moved into the empty space, leaving its previous position empty in turn. Tiles are numbered, 1 thru 8 for the 8-puzzle, so that each tile can be uniquely identified.

Figure 1: A picture on a 15-puzzle I bought to help build my intuition for sliding tile puzzles.

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 2023. 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. Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur

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. This reflects the fact that it takes the same amount of "finger effort" to move the tile in any direction.

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 consider Figure 2:

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

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

Figure 2: A worked example of the misplaced tile heuristic

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 example initial state shown in Figure 2, 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

As shown in Figure 3, Dr. Keogh provided use with the following test cases, sorted by depth of optimal solution. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est.

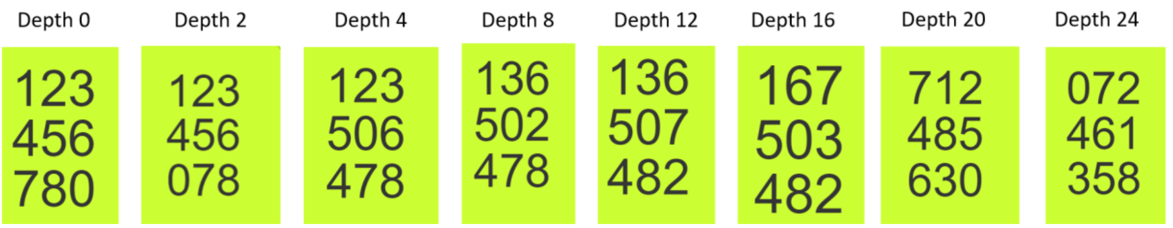

Figure 3: Samples of nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni.

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 Sed ut perspiciatis unde,

Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est.

Dmnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. quasi architecto beatae vitae dicta sunt explicabo.

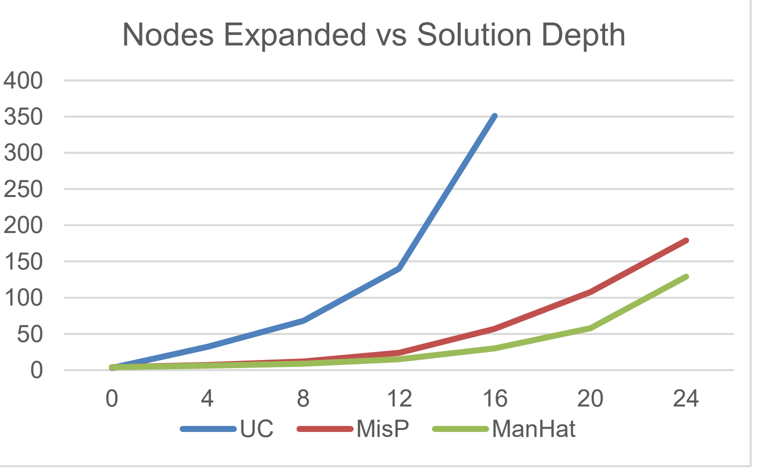

Figure 4: omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo

In Figure 5 we compute some accusantium Dmnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architect.

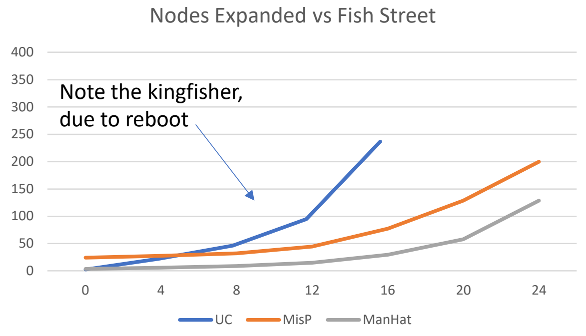

Figure 5: omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo

enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est.

Dmnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architect enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est.

Additional Examples

I also create some accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi, This examples show that if the input is enim ipsam voluptatem quia voluptas.

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 ipsam voluptatem quia voluptas sit aspernatur performed the best, followed by the enim ipsam voluptatem, followed by voluptatem quia voluptas (or in this case, effectively also called Breadth-First Search).
- The Misplaced Tile and Manhattan Distance heuristics improve the voluptatem quia of algorithms. Uniform Cost Search, h(n) having been hardcoded to 0, became Breadth

First Search, which has a time complexity of $O(bob^{sue})$ and also a space complexity of $O(van^{tan})$, where van is the color factor and tan is the flavor of the solution in the magni dolores.

- While both the sed quia consequuntur Heuristic and voluptatem Distance Heuristic improved sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi, This examples show that if the input is enim ipsam voluptatem.

The following is a traceback of an **easy** puzzle

```
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

The best state to expand with a g(n) = 0 and h(n) = 3 is…
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
The best state to expand with a g(n) = 1 and h(n) = 3 is…
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]
The best state to expand with a g(n) = 3 and h(n) = 0 is…
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Goal state!

Solution depth was 4
Number of nodes expanded: 13
Max queue size: 8
```

<EK says, The numbers above are just random numbers I made up>

The following is a traceback of a **hard** (depth 16) puzzle.

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

The best state to expand with a g(n) = 0 and h(n) = 3 is…
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]
The best state to expand with a g(n) = 2 and h(n) = 12 is…
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

::::           // Here I deleted about 17 pages of the trace to save space


The best state to expand with a g(n) = 4 and h(n) = 3 is…
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]
The best state to expand with a g(n) = 34 and h(n) = 0 is…
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Solution depth was 34
Number of nodes expanded: 45553
Max queue size: 534334
```

<EK says, The numbers above are just random numbers I made up>

URL to my code is http:Github.ucr.joe/cs150

nPuzzle.py

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

# Below are some built-in puzzles to allow quick testing.

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]]

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')

accusantium = doloremque(laudantium)
print(totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi)
def de architecto beatae vitae dicta sunt explicabo.
max(voluptatem sequi, This examples show that if the input is enim ipsam voluptatem quia volupta


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)


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)

accusantium = doloremque(laudantium)
print(totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi)
def de architecto beatae vitae dicta sunt explicabo.
```