Assignment 1 CS170: Introduction to Artificial Intelligence

Andrew Pirelli Dr. Eamonn Keogh
SID# 861223915
apire001@ucr.edu
November 5th, 2018

In completing this assignment I consulted:

- The blind search and A* search review slides from lecture.
- Python 3.7.1 documentation. Here is the URL of the table of contents: https://docs.python.org/3/
- The source for the diameter of the 8-puzzle I mentioned in the conclusion is found here: http://www.aiai.ed.ac.uk/~gwickler/eightpuzzle-uninf.html
- Numpy documentation, found here: http://www.numpy.org/
- Project 1 handout and briefing slides. As well as the example report provided for ideas on what topics I should cover and how I should format report. Note: I did not copy any of their ideas or writing, I simply used it as a very loose template for my report.

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

- Numpy matrix data structures and functions, imported from the numpy library
- Plotting functions, imported from the matplotlib.pyplot
- Some built in python functions for arrays, such as remove and append.

First page of code:

```
import numpy as np #numpy for calculations and matrix operations
import matplotlib.pyplot as plt #For plotting data from searches

def takeinput():
    print('Enter size n of n x n puzzle to be solved: ')
    size = input()
    size = int(size)
    print('Select type of search: ')
    print('1 for uniform search, 2 for misplaced tile search, 3 for manhattan distance
search')
    searchtype = input()
    print('Enter numbers for ' + str(size) + 'x' + str(size) + ' matrix seperated by spaces.
Blank is represented by 0.')
    values = input()
    matrix = np.zeros((size,size))
    if (len(values) < ((size**2)*2 - 1)):
        return matrix, 0, 0
    k = 0
    for i in range(size): #load values into nxn matrix
        for j in range(size):
            matrix[i, j] = int(values[k])
            k = k + 2
    return matrix, searchtype, size

def findtarget(matrix, size, target):
    for i in range(size):
        for j in range(size):
            if matrix[i, j] == target:
                return i,j

def swapleft(matrix, x, y):
    matrix1 = np.copy(matrix)
    matrix1[x, y], matrix1[x-1, y] = matrix1[x-1, y], matrix1[x, y]
    return matrix1

def swapright(matrix, x, y):
    matrix1 = np.copy(matrix)
    matrix1[x, y], matrix1[x+1, y] = matrix1[x+1, y], matrix1[x, y]
```

Last Page of Code:

```
start, searchtype, size = takeinput() #start is starting matrix, searchtype is search type
if searchtype == '1':
    print('Solving matrix using uniform search: ')
    print(start)
    nodesexpanded, depth, maxqueuesize = uniformcostsearch(start, size)
    print('Number of nodes expanded: ')
    print(nodesexpanded)
    print('Depth of solution: ')
    print(int(depth))
    print('Maximum queue size: ')
    print(maxqueuesize)
elif searchtype == '2':
    print('Solving matrix using misplaced tile search: ')
    print(start)
    nodesexpanded, depth, maxqueuesize = misplacedtilesearch(start, size)
    print('Number of nodes expanded: ')
    print(nodesexpanded)
    print('Depth of solution: ')
    print(int(depth))
    print('Maximum queue size: ')
    print(maxqueuesize)
elif searchtype == '3':
    print('Solving matrix using manhattan distance search: ')
    print('Expanding state')
    print(start)
    nodesexpanded, depth, maxqueuesize = manhattandistancesearch(start, size)
    print('Number of nodes expanded: ')
    print(nodesexpanded)
    print('Depth of solution: ')
    print(int(depth))
    print('Maximum queue size: ')
    print(maxqueuesize)
else:
    print('invalid input')
```

Note: My final program is about 10 pages of code consisting of helper functions, the user interface, plots, search functions and other functions.

Manhattan Distance A* Trace (As specified by the Project 1 Handout):
Enter size n of n x n puzzle to be solved:
3
Select type of search:
1 for uniform search, 2 for misplaced tile search, 3 for manhattan distance search
3
Enter numbers for 3x3 matrix separated by spaces. Blank is represented by 0.
1 2 3 4 0 6 7 5 8
Solving matrix using manhattan distance search:
Expanding state
[[1. 2. 3.]
 [4. 0. 6.]
 [7. 5. 8.]]
The best state to expand with a g(n) = 1 and h(n) = 2 is...
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 0. 8.]]
The best state to expand with a g(n) = 2 and h(n) = 0 is...
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 0.]]
Number of nodes expanded:
2
Depth of solution:
2
Maximum queue size:
7

Note: I assume that the starting node is at depth 0, and the queue I use is a minimum queue, as in it returns the node in the queue with the lowest f(n). (f(n) = g(n) + h(n) in this case)

CS170 Assignment 1 Write Up

Andrew Pirelli, SID 861223915

Development Challenges:

I wrote this program in Python, which is not my “native” programming language, so to speak. Because of this, my code is not nearly as clean as it could be. That being said, it does complete the tasks correctly, and I assume, optimally as well. However optimizations could be made on the code itself to implement more instances of python’s fast pre-compiled code operations compared to my implementations. My development process was fairly straightforward, I started with the simplest search algorithm, uniform search, then implemented the others with their additional features on top of these previous ones.

Specific examples of this would be adding the misplaced tiles heuristic to the distance heuristic from the uniform search to adjust the queueing function. Another would be updating the misplaced tiles heuristic with some extra lines of code to turn it into the manhattan distance heuristic. Overall, writing this code in a language I was not as comfortable in taught me a lot about that language (Python, in this case) and made me think more closely about my implementation.

Algorithm Analysis:

- The algorithms which I implemented and am analyzing are all different implementations of A* search, which expands nodes based on the lowest f(n), where f(n) = g(n) + h(n). g(n) being the distance to the node n and h(n) being the heuristic for that node n.
- Uniform search is effectively A* search with the heuristic set to always equal 0. This makes it the same as breadth first search, which is $O(b^d)$ time complexity and $O(b^d)$ space complexity. For the eight puzzle, the branching factor *b* is about 2.67, while the depth *d* should be no larger than 31, as the diameter for any solvable 8-puzzle is at most 31 moves.
- The source for the diameter is on the cover page. As for the branching factor, I calculated it by counting 4 spots where only 2 moves were possible (the corners), 4 spots where only 3 moves were possible (spots in between the corner spots) and 1 spot where 4 moves were possible (the center). This becomes (4/9)*2 + (4/9)*3 + (1/9)*4 = 2.67.
- The misplaced tile heuristic search counts the amount of tiles which do not match the tiles in the goal state (solved puzzle) and uses that number as the heuristic. It is not always very close to the actual amount of moves required, but is admissible since it is always less than or equal to the amount of moves it requires to reach the goal state.

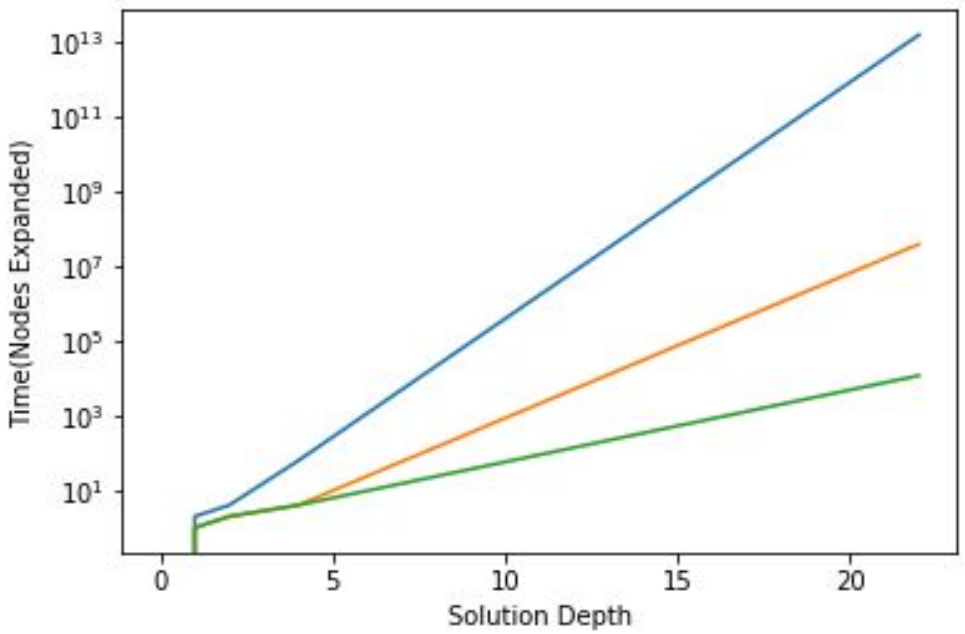
- The manhattan distance heuristic search counts the distance of the tiles that are misplaced to their goal positions. This heuristic is generally very close to the actual amount of moves required, and is admissible as well. The manhattan distance is calculated by $h(n) = h(n) + |x_{curr} - x_{goal}| + |y_{curr} - y_{goal}|$ for each tile.

Using Test Cases from 8-Puzzle Briefing Slides in Ascending Difficulty:
Runtime Analysis:
Uniform Search:
Solution Depth = [0, 1, 2, 4, 22]
Nodes Expanded = [0, 2, 4, 62, $1.5 * 10^{13}$]

Misplaced Tile Heuristic A* Search:
Solution Depth = [0, 1, 2, 4, 22]
Nodes Expanded = [0, 1, 2, 4, 38127442]

Manhattan Distance Heuristic A* Search:
Solution Depth = [0, 1, 2, 4, 22]
Nodes Expanded = [0, 1, 2, 4, 11687]

Blue line is Uniform Search, orange line is Misplaced Tile Heuristic A* Search and green line is Manhattan Distance Heuristic A* Search. Plotted in logspace(y) as it better shows all three of the lines. As you can see there is an exponential difference in run time between the three algorithms.

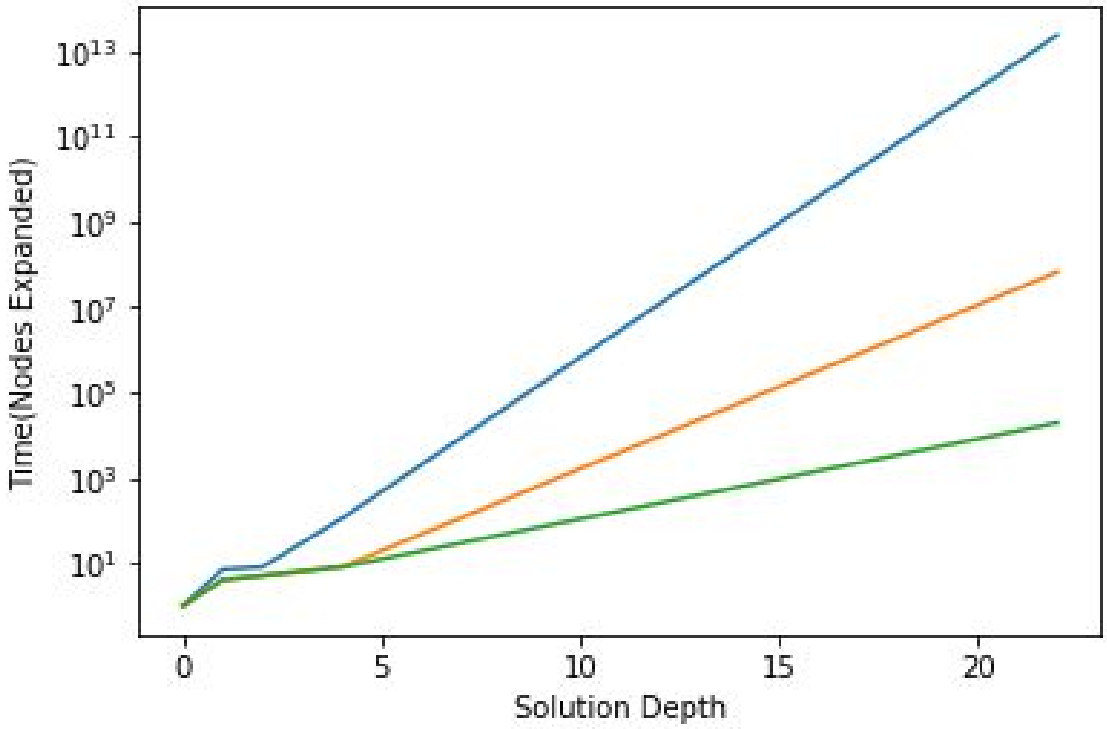


Memory Analysis:
Uniform Search:
Solution Depth = [0, 1, 2, 4, 22]
Maximum Queue Size = [0, 2, 4, 62, $2.3 * 10^{13}$]

Misplaced Tile Heuristic A* Search:
Solution Depth = [0, 1, 2, 4, 22]
Nodes Expanded = [1, 4, 5, 8, 64816651]

Manhattan Distance Heuristic A* Search:
Solution Depth = [0, 1, 2, 4, 22]
Nodes Expanded = [1,4,5,8,19099]

Blue line is Uniform Search, orange line is Misplaced Tile Heuristic A* Search and green line is Manhattan Distance Heuristic A* Search. Plotted in logspace(y) as it better shows all three of the lines. As you can see there is an exponential difference in memory used between the three algorithms.



Conclusion:

- The uniform search, misplaced tile heuristic search and manhattan distance heuristic search all have significant differences between their runtimes (nodes expanded) and memory (maximum amount of nodes in the queue) which increases exponentially as the problems become more complex.
- It is clear that having a not so great heuristic (misplaced tile heuristic) is far better than having no heuristic at all (uniform search) in terms of both runtime and memory for more complex problems.
- For simple problems having a heuristic does not make a very significant difference.
- The good heuristic (manhattan distance) distinguishes itself much more greatly from the not so great heuristic (misplaced tiles) as the problems become increasingly complex as well.