Coding Assignment #2
03/14/2020

Prepared
for
Prof. Hammoud

Prepared
by
Abedin Sherifi

Overview

The Anytime Nonparametric A* (ANA*) algorithm is an A* variant resembling closest to the ARA* algorithm. The ANA* algorithm expands the open state s based on the following:

$$e(s) = (G - g(s)) / h(s)$$

This is exactly what makes ANA* different from ARA*. ARA* uses the following equation for the expansion of the open state s:

$$f(s) = g(s) + \text{epsilon} * h(s)$$

The epsilon parameter above gets set by the user and it usually increments in fixed amounts. The fine tuning of this parameter usually requires trial and error. This is the motivation behind the ANA* algorithm which removes the dependency of the algorithm based on user set parameters.

In this experiment, two searching algorithms are compared. ANA* is compared against the Breadth-First-Search (BFS). Results are posted in the next section below.

Breadth – First – Search (BFS) Results

	BFS			
Maze Type	**Trivial**	**Medium**	**Hard**	**Very Hard**
Max Row	23	203	403	1003
Max Col	23	203	403	1003
Cost (# Moves)	202	19999	64807	> 64807
Run Time (sec)	0.00529	50.683	970.501	> 970.501
Cost/Time (# moves / sec)	38185.255198	394.58990194	66.7768503072125	N/A

Table 1. BFS Results

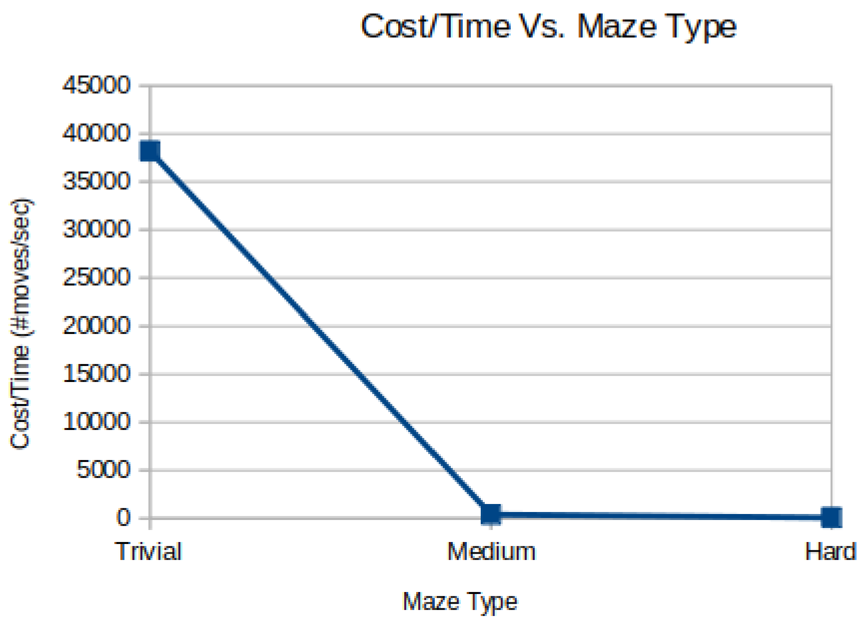

Figure 1. Cost/Time Vs. Maze Type for BFS

It can be noticed from Figure 1 and Table 1 that as the maze complexity increases, the cost increases drastically. We have to point out here that the cost function for BFS is O(V + E) where V represents vertices and E represents edges. It took approximately 970 seconds or 16 mins / 64807 moves to find a solution to the Hard Maze. It was taking over 1 hr and still was not able to find a solution for the Very Hard Maze after which the run of the code was interrupted.

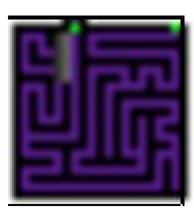

Figure 2. BFS Search Results for the Trivial Maze

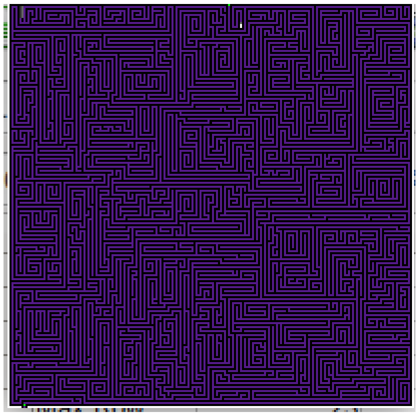

Figure 3. BFS Search Results for the Medium Maze

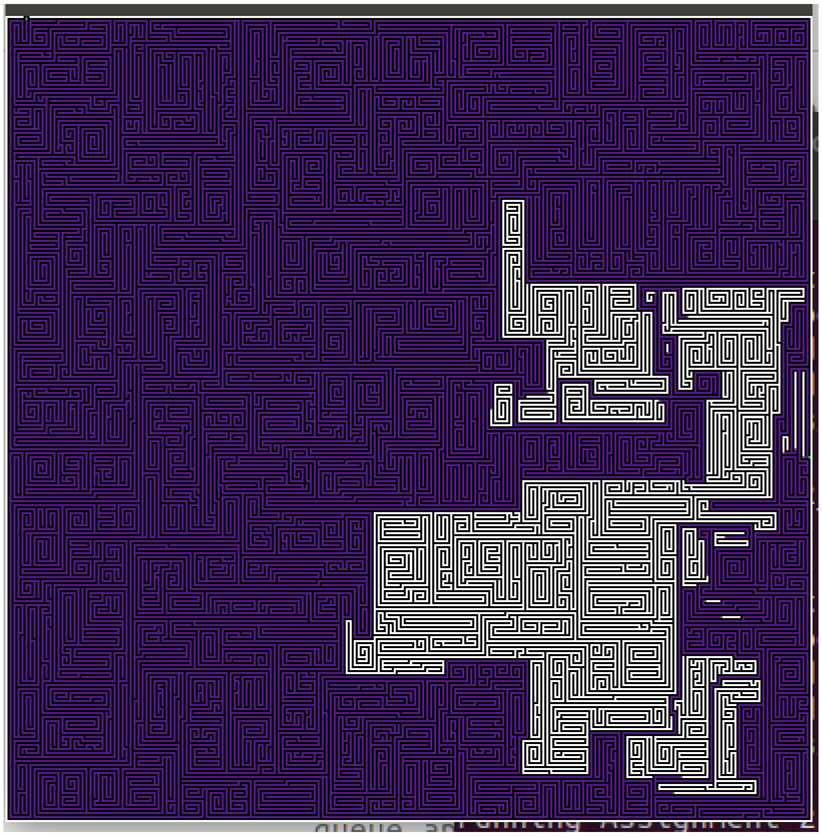

Figure 4. BFS Search Results for the Hard Maze

Tried to run the Very Hard Maze through the BFS algorithm, but it was taking way over an hour.

Anytime Nonparametric A* (ANA*) Results

	ANA*			
Maze Type	Trivial	Medium	Hard	Very Hard
Max Row	23	203	403	1003
Max Col	23	203	403	1003
Cost (# Moves)	147	3227	8287	18063
Time to Optimal Soln (sec)	0.013	0.237	0.651	6.274
Cost/Time (# moves / sec)	11307.692308	13616.033755	12729.6466973886	2879.0245457
Improved Solution Wake Count	1	1	2	2
Time to First Improved Solution	0.002	0.211	0.118	0.319

Table 2. ANA* Results

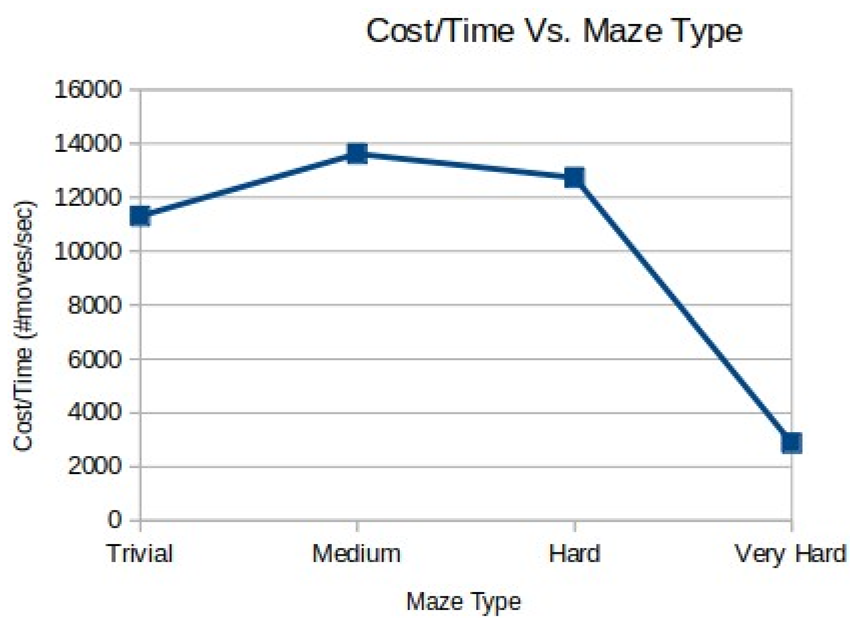

Figure 5. Cost/Time Vs. Maze Type for ANA*

It can be noticed from Figure 5 and Table 2 that as the maze complexity increases, the cost function increase as well. ANA* implements heuristics which over time improves the cost to the optimal solution. We have to point out here that for the Hard and Very Hard Maze, the ANA* algorithm used the Improve Solution function twice. Based on these results, it can be observed for example for the Very Hard Maze that the initial sub-optimal solution was captured in 0.319 seconds and time to the optimal solution in 6.274 seconds with 18063 moves. Comparing these statistics to the Hard Maze of BFS (970 seconds or 16 mins / 64807 moves), it can be concluded that the ANA* is truly superior to the BFS. ANA* solved the Very Hard Maze (optimal solution) in 6.274 seconds with 18063 moves compared to BFS which took over 16 mins and over 64807 moves!!!

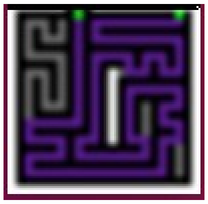

Figure 6. ANA* Search Results for the Trivial Maze

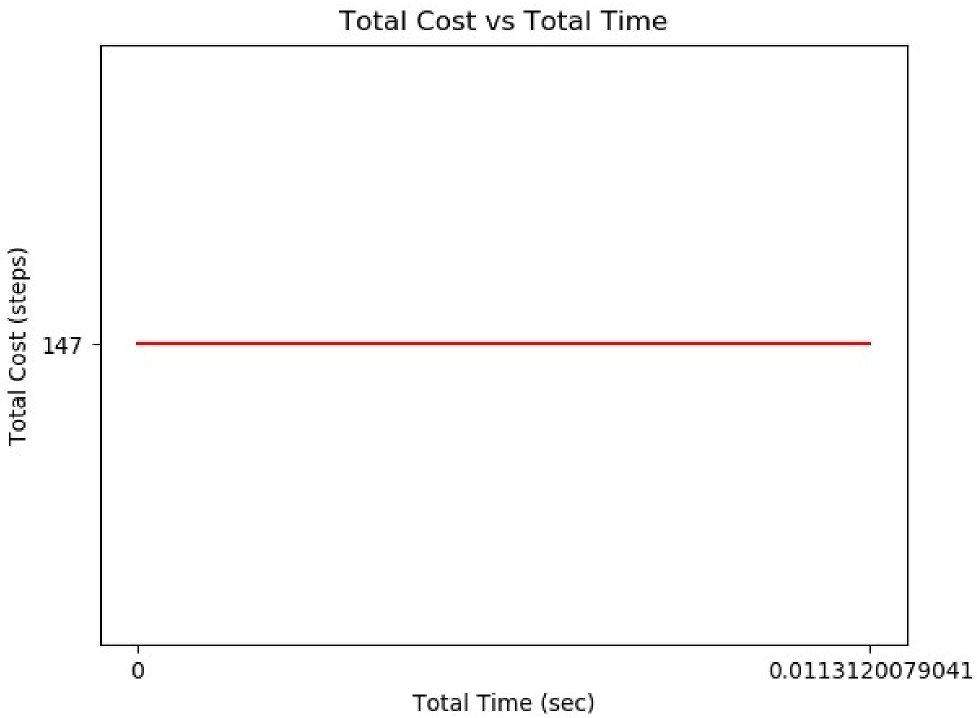

Figure 7. ANA* Total Cost Vs. Total Time for the Trivial Maze

Figure 8. ANA* Search Results for the Medium Maze

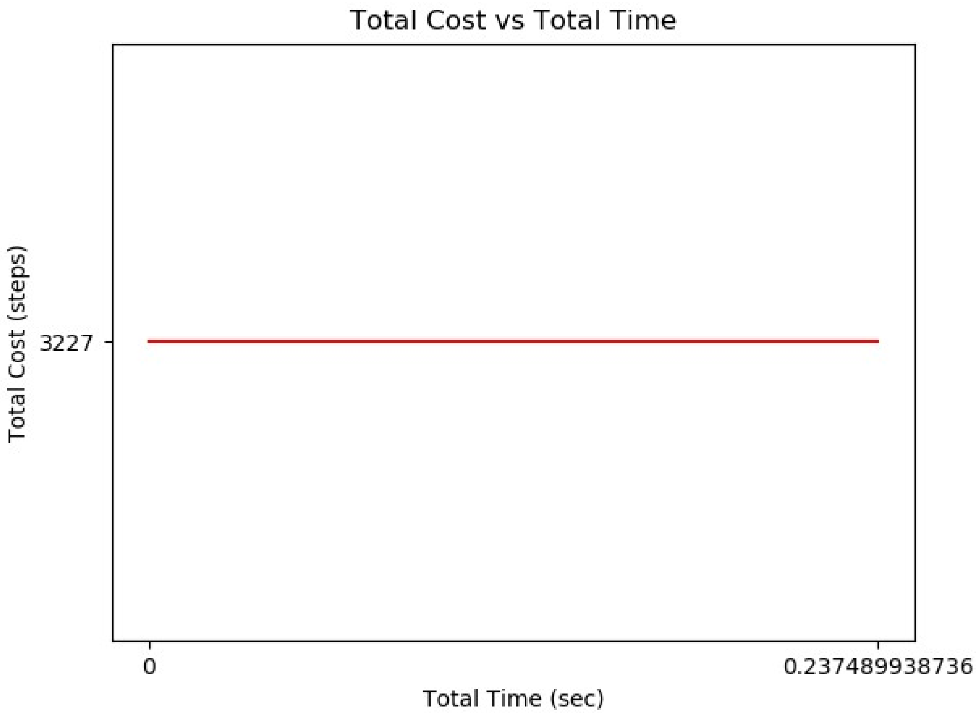

Figure 9. ANA* Total Cost Vs. Total Time for the Medium Maze

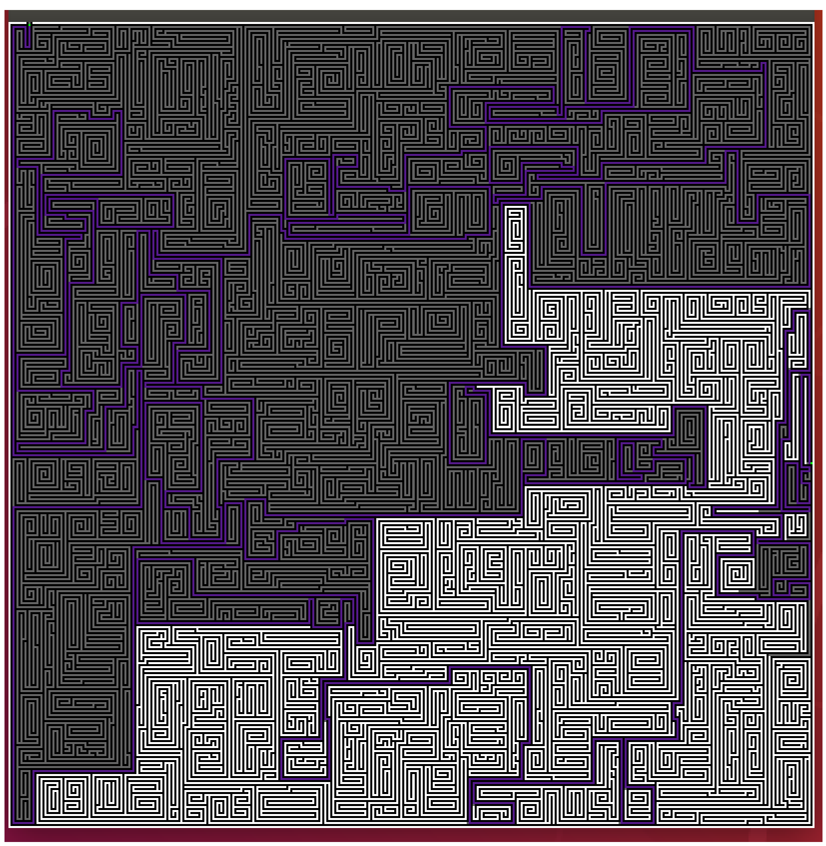

Figure 10. ANA* Search Results for the Hard Maze

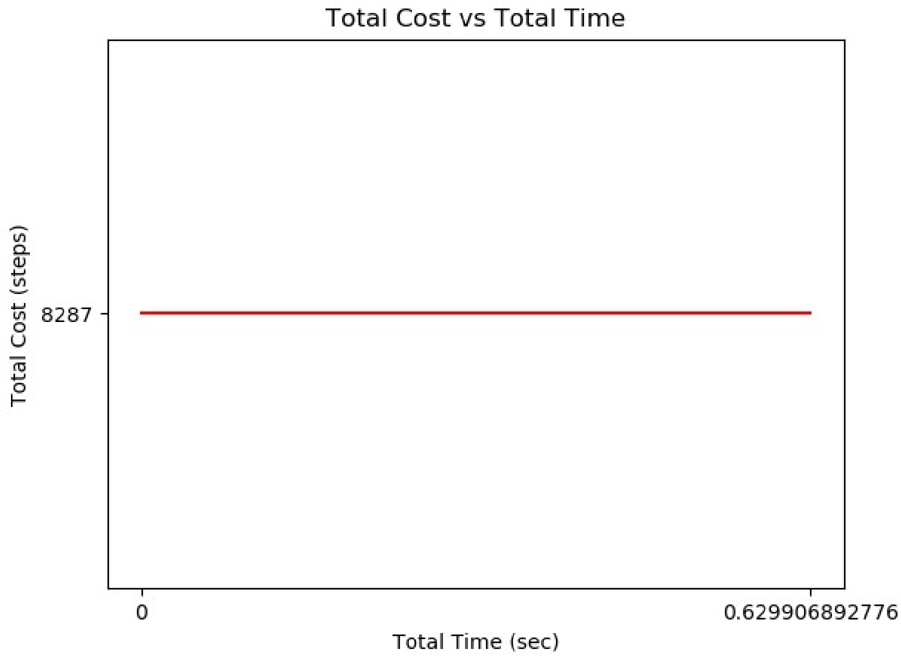

Figure 11. ANA* Total Cost Vs. Total Time for the Hard Maze

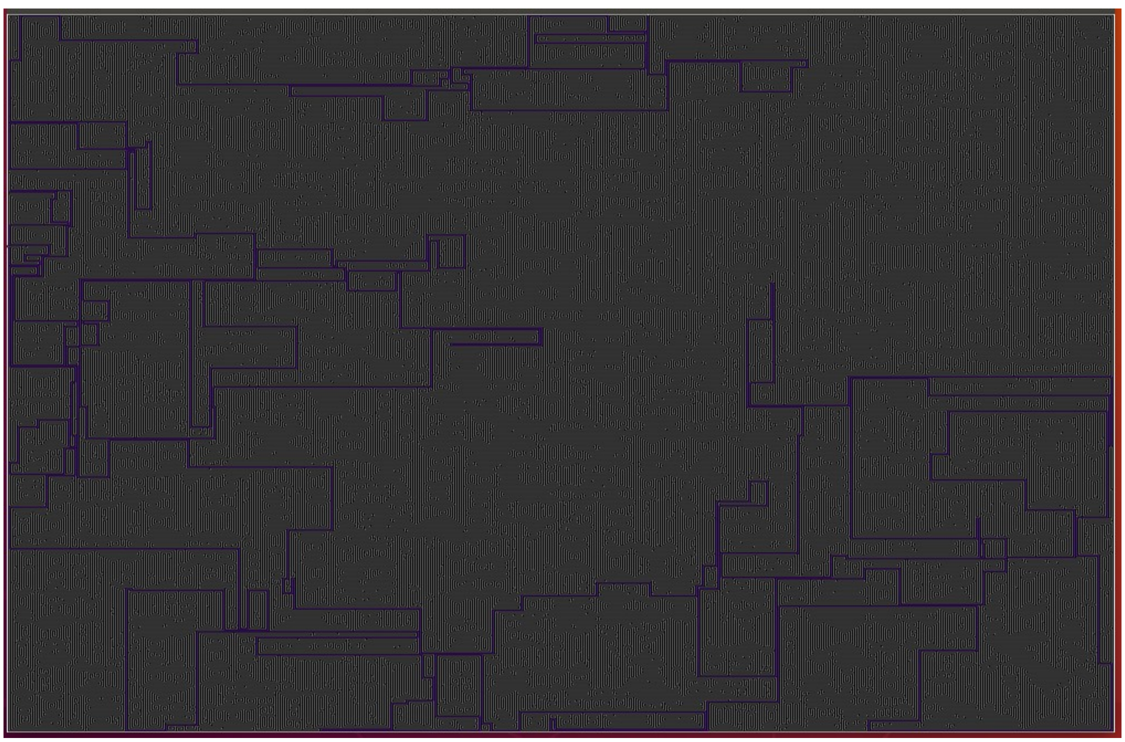

Figure 12. ANA* Search Results for the Very Hard Maze

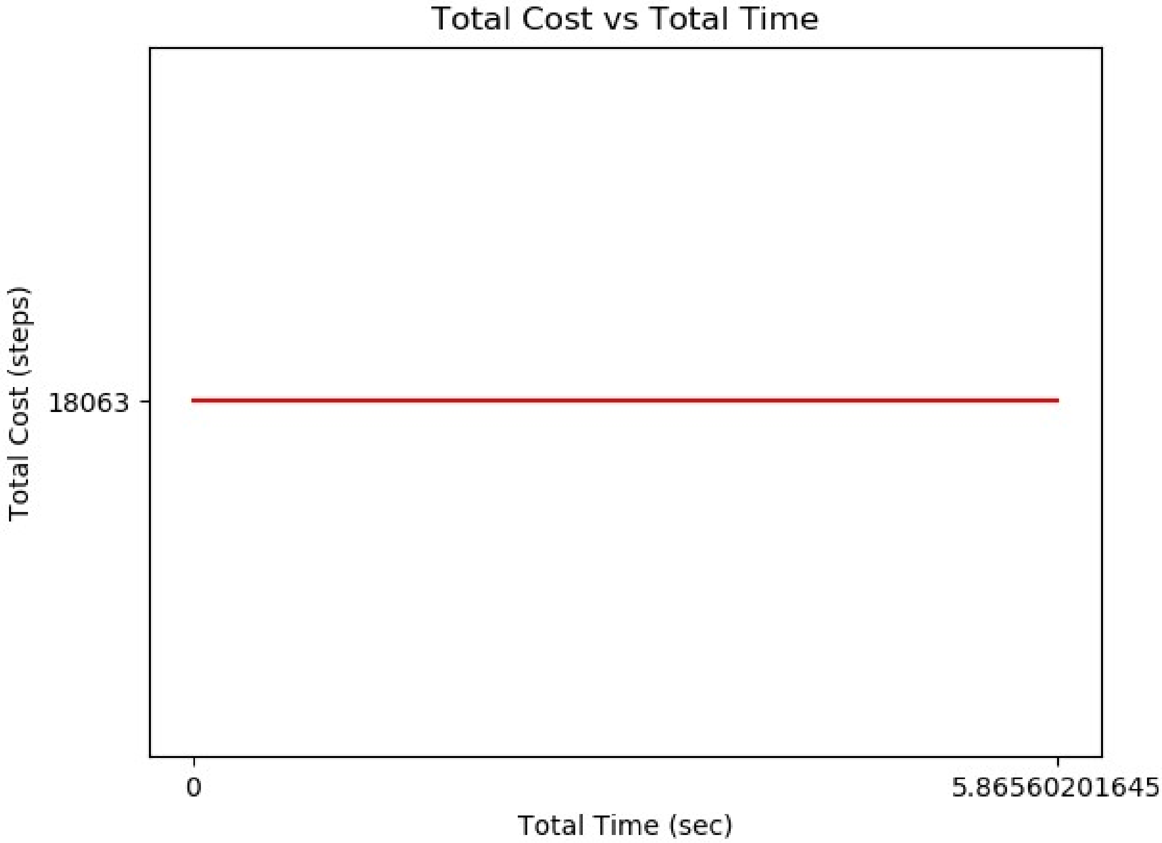

Figure 13. ANA* Total Cost Vs. Total Time for the Very Hard Maze

Summary

In summary, the ANA* is superior to the BFS algorithm. The ANA* algorithm finds an initial solution way faster, converges to the optimal solution very fast, and it finds the goal state from the start state in the fewest number of moves.

BFS Code

```
# Abedin Sherifi
# Coding Assignment #2
# RBE550
# 03/14/2020

import sys
from PIL import Image
import copy
import time

'''
These variables are determined at runtime and should not be changed or mutated by you
'''
start = (0, 0)  # a single (x,y) tuple, representing the start position of the search algorithm
end = (0, 0)  # a single (x,y) tuple, representing the end position of the search algorithm
difficulty = ""  # a string reference to the original import file

'''
These variables determine display coler, and can be changed by you, I guess
'''
NEON_GREEN = (0, 255, 0)
PURPLE = (85, 26, 139)
LIGHT_GRAY = (50, 50, 50)
DARK_GRAY = (100, 100, 100)

'''
These variables are determined and filled algorithmically, and are expected (and required) be mutated
by you
'''
path = []  # an ordered list of (x,y) tuples, representing the path to traverse from start-->goal
expanded = {}  # a dictionary of (x,y) tuples, representing nodes that have been expanded
frontier = {}  # a dictionary of (x,y) tuples, representing nodes to expand to in the future

#BFS Algorithm
def BFS(maze, max_x, max_y):
        queue = []
        queue.append(start)
        solved = False
        moves = 0
        expanded[start] = 1
        start_time = time.time()

        while len(queue) > 0:
            current_node = queue.pop(0)
            path.append(current_node)
            expanded[current_node] = (-1,-1)
            moves += 1
```

```
        if current_node == end:
            solved = True
            print "Maze has been solved!"
            end_time = time.time()
            completion_time = end_time - start_time
            print("Maze has been solved in {} moves in {} seconds".format(moves, completion_time))

                row = current_node[0]
                    col = current_node[1]
            while row != -1:
                path.append(current_node)
                row, col = expanded[(row, col)]
            path.reverse()
            return path


        row = current_node[0]
            col = current_node[1]
        for dr, dc in ((-1, 0), (0, -1), (1, 0), (0, 1)):
            new_r = row + dc
            new_c = col + dr
            new_node = (new_r, new_c)
            if (0 <= new_r < max_x and 0 <= new_c < max_y and new_node not in expanded.keys() and
maze[new_node[0], new_node[1]] != 1):
                expanded[new_node] = current_node
                queue.append((new_r, new_c))
    return


def search(map):
    """
    This function is meant to use the global variables [start, end, path, expanded, frontier] to search
through the
    provided map.
    :param map: A '1-concept' PIL PixelAccess object to be searched. (basically a 2d boolean array)
    """

    # O is unoccupied (white); 1 is occupied (black)
    print "pixel value at start point ", map[start[0], start[1]]
    print "pixel value at end point ", map[end[0], end[1]]

    # put your final path into this array (so visualize_search can draw it in purple)
    #path.extend([(8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7)])

    # put your expanded nodes into this dictionary (so visualize_search can draw them in dark gray)
    #expanded.update({(7, 2): True, (7, 3): True, (7, 4): True, (7, 5): True, (7, 6): True, (7, 7): True})

    # put your frontier nodes into this dictionary (so visualize_search can draw them in light gray)
    #frontier.update({(6, 2): True, (6, 3): True, (6, 4): True, (6, 5): True, (6, 6): True, (6, 7): True})
```

```
    BFS(map, max_x, max_y)

    visualize_search("out.png")  # see what your search has wrought (and maybe save your results)


def visualize_search(save_file="do_not_save.png"):
    """
    :param save_file: (optional) filename to save image to (no filename given means no save file)
    """
    im = Image.open(difficulty).convert("RGB")
    pixel_access = im.load()

    # draw expanded pixels
    for pixel in expanded.keys():
        pixel_access[pixel[0], pixel[1]] = DARK_GRAY

    # draw path pixels
    for pixel in path:
        pixel_access[pixel[0], pixel[1]] = PURPLE

    # draw frontier pixels
    for pixel in frontier.keys():
        pixel_access[pixel[0], pixel[1]] = LIGHT_GRAY

    # draw start and end pixels
    pixel_access[start[0], start[1]] = NEON_GREEN
    pixel_access[end[0], end[1]] = NEON_GREEN

    # display and (maybe) save results
    im.show()
    if (save_file != "do_not_save.png"):
        im.save(save_file)

    im.close()


if __name__ == "__main__":
    # Throw Errors && Such
    # global difficulty, start, end
    assert sys.version_info[0] == 2  # require python 2 (instead of python 3)
    assert len(sys.argv) == 2, "Incorrect Number of arguments"  # require difficulty input

    # Parse input arguments
    function_name = str(sys.argv[0])
    difficulty = str(sys.argv[1])
    print "running " + function_name + " with " + difficulty + " difficulty."

    # Hard code start and end positions of search for each difficulty level
```

```
if difficulty == "trivial.gif":
    start = (8, 1)
    end = (20, 1)
elif difficulty == "medium.gif":
    start = (8, 201)
    end = (110, 1)
elif difficulty == "hard.gif":
    start = (10, 1)
    end = (401, 220)
elif difficulty == "very_hard.gif":
    start = (1, 324)
    end = (580, 1)
else:
    assert False, "Incorrect difficulty level provided"

# Perform search on given image
im = Image.open(difficulty)
max_x, max_y = im.size
print('Max Row: ', max_x)
print('Max Col: ', max_y)
search(im.load())
```

ANA* Code

```
# Abedin Sherifi
# Coding Assignment #2
# RBE550
# 03/14/2020

import sys
from PIL import Image
import copy
import time
from Queue import PriorityQueue as PQ
import matplotlib.pyplot as plt

'''
These variables are determined at runtime and should not be changed or mutated by you
'''
start = (0, 0)  # a single (x,y) tuple, representing the start position of the search algorithm
end = (0, 0)    # a single (x,y) tuple, representing the end position of the search algorithm
difficulty = "" # a string reference to the original import file

'''
These variables determine display coler, and can be changed by you, I guess
'''
NEON_GREEN = (0, 255, 0)
PURPLE = (85, 26, 139)
LIGHT_GRAY = (50, 50, 50)
DARK_GRAY = (100, 100, 100)

'''
These variables are determined and filled algorithmically, and are expected (and required) be mutated
by you
'''
path = []       # an ordered list of (x,y) tuples, representing the path to traverse from start-->goal
expanded = {}   # a dictionary of (x,y) tuples, representing nodes that have been expanded
frontier = {}   # a dictionary of (x,y) tuples, representing nodes to expand to in the future

G = 10000000000000
E = 10000000000000

#Reverse Priority Queue
class RPQ(PQ):

    def put(self,xy):
        n_xy = xy[0] * (-1), xy[1], xy[2]
        PQ.put(self, n_xy)

    def get(self):
```

```
        xy = PQ.get(self)
        n_xy = xy[0] * (-1), xy[1], xy[2]
        return n_xy

#Node expansion
def ex_nodes(map, size, node):
    row = node[0]
    col = node[1]
    ans = []

    width = size[0]
    height = size[1]

    if (row+1 < width) and (map[row+1,col] != 1):
        ans.append((row+1,col))

    if (col+1 < height) and (map[row,col+1] != 1):
        ans.append((row,col+1))

  if (row>=1) and (map[row-1,col] != 1):
        ans.append((row-1,col))

    if (col>=1) and (map[row,col-1] != 1):
        ans.append((row,col-1))
    return ans

#Pruned states
def prune(OPEN, G, end):
    refresh_OPEN = RPQ(0)

    while not OPEN.empty():
        node = OPEN.get()
        e_s = node[0]
        g_s = node[1]
        state = node[2]
        h_s = 5 * (abs(end[0] - start[0]) + abs(end[1] - start[1]))

        if g_s + h_s < G:
            n_s = (G - g_s)/(h_s + 0.00001)
            refresh_OPEN.put((n_s, g_s, state))
    return refresh_OPEN

#Improve Solution
def Improve_Solution(pred, expanded, OPEN, G, E, map, size, end):

    while not OPEN.empty():
        current_node = OPEN.get()
        e_s = current_node[0]
        g_s = current_node[1]
```

```
        state = current_node[2]

        if e_s < E:
            E = e_s

        if state == end:
            G = g_s
            break

        for next_node in ex_nodes(map, size, state):
            new_cost = expanded[state] + 1
            if next_node not in expanded or new_cost < expanded[next_node] :
                expanded[next_node] = new_cost
                h_next = 5*(abs(end[0] - next_node[0]) + abs(end[1] - next_node[1]))
                if (new_cost + h_next) < G:
                    e_next_node = (G - new_cost)/(h_next + 0.00001)
                    OPEN.put((e_next_node, new_cost, next_node))
                pred[next_node] = state

    return pred, expanded, OPEN, G, E

#ANA* Algorithm
def ANAStar(map, size, start, end):

    global E, G
    pred = {}
    expanded = {}
    pred[start] = None
    expanded[start] = 0
  improve_soln_cnt = 0
    h_start = 5 * (abs(end[0] - start[0]) + abs(end[1] - start[1]))
    e_s = (G - 0)/(h_start + 0.00001)
  OPEN = RPQ(0)
    OPEN.put((e_s, 0, start))

    while not OPEN.empty():
        improve_soln_cnt += 1
        before = time.time()
        pred, expanded, OPEN, G, E = Improve_Solution(pred, expanded, OPEN, G, E, map,
size, end)
        after = time.time()
        total_time = after - before
        OPEN = prune(OPEN, G, end)

    path = []
    current_node = end
    while current_node != start:
        path.append(current_node)
        current_node = pred[current_node]
```

```
        path.append(start)
        path.reverse()

      print "Improved Solution Wake Count          : " + str(improve_soln_cnt)
        print "Time to find first improved solution  : " + str (total_time) + ' sec'

        frontier = {}
        for i in OPEN.queue:
                frontier[i[2]] = i[1]

        return path, expanded, frontier

#Search function
def search(map):

    global start, end, expanded, path, frontier

    """
    This function is meant to use the global variables [start, end, path, expanded, frontier] to search
through the
    provided map.
    :param map: A '1-concept' PIL PixelAccess object to be searched. (basically a 2d boolean array)
    """

    # O is unoccupied (white); 1 is occupied (black)
    print "pixel value at start point ", map[start[0], start[1]]
    print "pixel value at end point ", map[end[0], end[1]]

    # put your final path into this array (so visualize_search can draw it in purple)
    path.extend([(8,2), (8,3), (8,4), (8,5), (8,6), (8,7)])

    # put your expanded nodes into this dictionary (so visualize_search can draw them in dark gray)
    expanded.update({(7,2):True, (7,3):True, (7,4):True, (7,5):True, (7,6):True, (7,7):True})

    # put your frontier nodes into this dictionary (so visualize_search can draw them in light gray)
    frontier.update({(6,2):True, (6,3):True, (6,4):True, (6,5):True, (6,6):True, (6,7):True})

    path, expanded, frontier = ANAStar(map, size, start, end)

    Total_Moves = str(len(path))
    print "Path: ", Total_Moves

    visualize_search("out.png") # see what your search has wrought (and maybe save your results)

#Visualize function
def visualize_search(save_file="do_not_save.png"):
    """
    :param save_file: (optional) filename to save image to (no filename given means no save file)
    """
```

```
    im = Image.open(difficulty).convert("RGB")
    pixel_access = im.load()

    # draw expanded pixels
    for pixel in expanded.keys():
        pixel_access[pixel[0], pixel[1]] = DARK_GRAY

    # draw path pixels
    for pixel in path:
        pixel_access[pixel[0], pixel[1]] = PURPLE

    # draw frontier pixels
    for pixel in frontier.keys():
        pixel_access[pixel[0], pixel[1]] = LIGHT_GRAY

    # draw start and end pixels
    pixel_access[start[0], start[1]] = NEON_GREEN
    pixel_access[end[0], end[1]] = NEON_GREEN

    im.show()
    if(save_file != "do_not_save.png"):
        im.save(save_file)

    im.close()


if __name__ == "__main__":
    # Throw Errors && Such
    # global difficulty, start, end
    assert sys.version_info[0] == 2                          # require python 2 (instead of python 3)
    assert len(sys.argv) == 2, "Incorrect Number of arguments"      # require difficulty input

    # Parse input arguments
    function_name = str(sys.argv[0])
    difficulty = str(sys.argv[1])
    print "running " + function_name + " with " + difficulty + " difficulty."

    # Hard code start and end positions of search for each difficulty level
    if difficulty == "trivial.gif":
        start = (8, 1)
        end = (20, 1)
    elif difficulty == "medium.gif":
        start = (8, 201)
        end = (110, 1)
    elif difficulty == "hard.gif":
        start = (10, 1)
        end = (401, 220)
    elif difficulty == "very_hard.gif":
        start = (1, 324)
```

```
    end = (580, 1)
else:
    assert False, "Incorrect difficulty level provided"
# Perform search on given image
im = Image.open(difficulty)
size = im.size
print('Max Row: ', size[0])
print('Max Col: ', size[1])
before=time.time()
search(im.load())
after=time.time()
total_time=after - before
print "======================================================"
print "======================================================"
print "Total time to find the optimal solution: ", total_time, "sec"

y = [str(len(path)),str(len(path))]
x = [str(0),str(total_time)]
plt.figure(1)
plt.plot(x, y,'r-')
plt.title("Total Cost vs Total Time")
plt.xlabel('Total Time (sec)')
plt.ylabel('Total Cost (steps)')
plt.savefig('Maze.png')
plt.show()
```