function [depth, total\_nodes, max\_size] = problem(current, search\_type)

%Including the diameter of the 8 tile puzzle in case the given problem

%Is an impossible problem

depth = 0; total\_nodes = 0; max\_size = 0; diameter = 31;

goal\_state = [1 2 3; 4 5 6; 7 8 0];

%nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))

queue = Queue(current, 0, 0);

% do while notEMPTY(nodes)

% if empty, return failure

while get\_Queue\_size(queue) > 0

%node = REMOVE-FRONT(nodes)

[state, depth] = pop\_from\_Queue(queue);

%Check for max size of the queue only to be able to return it

if max\_size < get\_Queue\_size(queue)

max\_size = get\_Queue\_size(queue);

end

%if problem.GOAL-TEST(node.STATE) succeeds then return node

if isequal(state, goal\_state)

goal = 'Success!'

return

end

%In the case that the puzzle is unsolvable

if depth <= diameter

%Returns the index of the 0

num\_index = find(state == 0);

%Convert the index into a row and column index

row\_index = mod((num\_index - 1), 3) + 1;

col\_index = floor((num\_index - 1)/ 3) + 1;

%Omitted the process for moving blank down, left and right

if row\_index > 1

%move blank up

node = move\_blank(state, row\_index, col\_index, 1);

%Check for repeated states. If so, don't update anything

if ~(repeated\_states(queue, node))

%Uniform Cost Search, h(n) = 0, f(n) = depth + 1

if isequal(search\_type, 1)

cost = depth + 1;

add\_to\_Queue(queue, node, depth + 1, cost);

%Misplaced Tiles, h(n) = f(n) + g(n), f(n) = depth + 1

elseif isequal(search\_type, 2)

cost = misplaced\_tile(node, goal\_state) + depth + 1;

add\_to\_Queue(queue, node, depth + 1, cost);

%Manhattan Distance, f(n) = depth + 1

else

cost = manhattan\_distance(node, goal\_state) + depth + 1;

add\_to\_Queue(queue, node, depth + 1, cost);

end

total\_nodes = total\_nodes + 1;

end

end

end

'Failure. Puzzle is not solvable.'

end

classdef Queue < handle

%Set to handle because we only want 1 queue

methods

%Constructor function

function obj = Queue(matrix, val1, val2)

%State and all\_nodes are 3D matrices

obj.state = cat(3, matrix);

obj.depth = val1;

obj.cost = val2;

obj.all\_nodes = cat(3, matrix);

end

function index = find\_smaller(obj, total)

%Set the index to in case the obj.cost is empty

index = 0;

if isempty(obj.cost)

else

%If not, we set index equal to the index previous

%The node which is larger than it

for i = 1:size(obj.cost, 2)

if obj.cost(i) >= total

index = i - 1;

break;

else

%If it doesn't enter the if statement, then it

%Goes to the very end of queue

index = size(obj.cost, 2);

end

end

end

end

function add\_to\_Queue(obj, matrix, deep, total)

%Add the node to the all\_nodes queue

obj.all\_nodes = cat(3, matrix, obj.all\_nodes);

q\_size = size(obj.cost, 2);

%Implement the indexing function

index = find\_smaller(obj, total);

%If statement for if inserting at the front, back or in the

%middle of the queue

%Do this for all 3 of the queues

if isequal(index, 0)

%Front of queue

obj.state = cat(3, matrix, obj.state);

obj.depth = [deep, obj.depth];

obj.cost = [total, obj.cost];

elseif isequal(index, q\_size)

%End of queue

obj.state = cat(3, obj.state, matrix);

obj.depth = [obj.depth, deep];

obj.cost = [obj.cost, total];

else

%Inserting in the middle, in between index and index + 1

obj.state = cat(3, obj.state(:, :, 1:index), matrix, obj.state(:, :, (index + 1):q\_size));

obj.depth = [obj.depth(1:index), deep, obj.depth((index + 1):q\_size)];

obj.cost = [obj.cost(1:index), total, obj.cost((index + 1):q\_size)];

end

end

end

end

puzzle = [1 2 3; 4 0 6; 7 5 8];

Enter the chosen puzzle and either 1 for uniform cost search, 2 for A\* with the Misplaced Tile heuristic, or 3 for A\* with the Manhattan Distance heuristic

[depth, total\_nodes, max\_size] = problem(puzzle, 3);

Expanding state

1 2 3

4 0 6

7 5 8

Expanding best state

1 2 3

4 5 6

7 0 8

With g(n) = 1 and h(n) = 1

Expanding best state

1 2 3

4 5 6

7 8 0

With g(n) = 2 and h(n) = 0

Success!

To solve this problem, the search algorithm expanded 6 nodes

The maximum number of nodes in the queue at any one point in time was 4 nodes

The depth of the goal nodes was 2

I tested all three algorithms on four different initial states with varying depths in order to see the difference in performance, time complexity and space complexity, in the three algorithms.

The first puzzle, labeled easy is shown below:

1 2 0

4 5 3

7 8 6

All three algorithms found the goal node at a depth of 2, demonstrating that each algorithm is optimal.

Uniform Cost Search expanded a total of 9 nodes, and had max 5 nodes in the queue.

A\* with Misplaced Tile expanded a total of 4 nodes, and had max 2 nodes in the queue.

A\* with Manhattan Distance expanded a total of 4 nodes, and had max 2 nodes in the queue.

The second puzzle, labeled doable is shown below:

0 1 2

4 5 3

7 8 6

All three algorithms found the goal node at a depth of 4, demonstrating that each algorithm is optimal.

Uniform Cost Search expanded a total of 43 nodes, and had max 19 nodes in the queue.

A\* with Misplaced Tile expanded a total of 7 nodes, and had max 3 nodes in the queue.

A\* with Manhattan Distance expanded a total of 7 nodes, and had max 3 nodes in the queue.

The third puzzle, labeled medium was found in Keogh’s slides and told to have a depth of 12. This initial state is shown below:

1 2 3

4 5 8

6 7 0

All three algorithms found the goal node at a depth of 12, demonstrating that each algorithm is optimal.

Uniform Cost Search expanded a total of 2409 nodes, and had max 906 nodes in the queue.

A\* with Misplaced Tile expanded a total of 135 nodes, and had max 58 nodes in the queue.

A\* with Manhattan Distance expanded a total of 84 nodes, and had max 37 nodes in the queue.

The final puzzle that I tested, labeled ‘Oh Boy’, is shown below:

8 7 1

6 0 2

5 4 3

All three algorithms found the goal node at a depth of 22, demonstrating that each algorithm is optimal. However, Uniform Cost Search took approximately 3 hours to reach the solution, the Misplaced Tile heuristic took 2 minutes to reach the solution, and the Manhattan Distance heuristic took less than a second to reach the solution.

Uniform Cost Search expanded a total of 116684 nodes, and had max 25437 nodes in the queue.

A\* with Misplaced Tile expanded a total of 9087 nodes, and had max 3345 nodes in the queue.

A\* with Manhattan Distance expanded a total of 433 nodes, and had max 166 nodes in the queue.

I decided to plot the time complexity and space complexity of each of the algorithms based on with the x-axis as the depth and the logarithmic y-axis as either the total nodes expanded or the max queue size. I also decided to plot the time complexity and space complexity divided by the total nodes expanded or the max queue size for A\* with Manhattan Distance at that depth (setting Manhattan distance heuristic as a constant) in order to see how many magnitudes worse the Uniform Cost Search and Misplaced Tile Heuristic were at varying depths.







The last two graphs with the Manhattan Heuristic as a constant are quite interesting and telling. They look almost exactly the same. We can conclude that the time and space complexity for the inferior algorithms are both equally worse than the best heuristic. That is to say that for the uniform cost search algorithm, both the time and space complexity are two magnitudes worse than that of the best heuristic, instead of the time complexity being two magnitudes worse, and the space complexity being only one magnitude worse. We can also conclude that for problems that are easy to solve such as the ‘easy’ and ‘doable’ initial states, the uniform cost search algorithm performed less than a magnitude worse than the best and weak heuristic. As the problems got more complex, such as the ‘medium’ initial state having a depth of 12, the weaker heuristic, the Misplaced Tile heuristic, still had a very similar space and time complexity as the tighter heuristic, and both performed an entire magnitude better than having no heuristic. However, for the extremely difficult puzzles, such as the ‘Oh Boy’ initial state, A\* with a stronger heuristic was a magnitude better in both space and time complexity as the weaker heuristic, and over two magnitudes faster and more space efficient than the uniform cost search algorithm.