Anthony Mendez

862307065

Amend303@ucr.edu

November 1, 2022

Project 1 – Eight Puzzle

Link To Code: https://github.com/Antheagao/Project1

Report Outline

References

- Starting out with C++: From Control Structures through Objects 9th Edition, Tony Gaddis

  – Used for information on the C++ STL, such as vectors, queues, sets, and stacks.

- https://cplusplus.com/

  – Used to find functions for vectors and queues

- Blind Search and Heuristic Search lecture slides

  – Used to implement Uniform Cost, Misplaced Tile, and Manhattan Distance searches, along with their heuristics and operators.

Introduction

This project focuses on the 8-Table puzzle and the relation that several search algorithms have in solving it. The puzzle is ordered as a matrix that is ordered randomly. The 8 puzzle is considered solved when the squares in the matrix are sorted with the blank at the bottom rightmost corner.

The project follows several principles for an AI search, such as initial state, operators, and goal state. The initial state can be random and the operators are used to create branches towards the goal state. The table is considered solved once the goal state is reached. There are four operators used for this project that move the blank a certain direction (up, down, left, right).

I chose to use C++ as the programming language for this project. A general search algorithm is used to solve the 8-Table using Uniform Cost, Misplaced Tile, and Manhattan Distance searches. The general search algorithm determines which search to perform based on the queueing function passed to it.

Uniform Cost Search

As mentioned in the lecture slides, Uniform Cost search is a blind search that does not use a heuristic, h(n) = 0. Uniform Cost determines which node to travel to by the cost of the edge to get there, g(n). However, a tile slide is considered to always have a cost of 1 for each state expansion. As a result, each expanded node is stored in a queue so the search can travel to each node in a depth-first search behavior. The search terminates with the goal state if successful or failure if there was no solution.

The Misplaced Tile Heuristic

The Misplaced Tille heuristic performs a similar expansion to Uniform Cost, in fact it is the heuristic that makes them different. A heuristic is often seen as a hint for how close one state is to another. Misplaced Tile uses the goal state to calculate how many squares are in the incorrect places of the current state while ignoring the blank. Thus, out of all the expanded nodes, the reachable node with the lowest h(n) will be stored in the queue.

The Manhattan Distance Heuristic

The Manhattan Distance heuristic varies from misplaced tile by its heuristic calculation. It uses the goal state and the current state to calculate the distance of each misplaced square from its goal square. The general calculation of this heuristic is |x1 - x2| + |y1 - y2| [1].

Data Chart

This data chart from figure 1 provides the amount of node expansions and depth for 5 levels of problems that were provided in the example report. Uniform Cost makes the most expansions, followed by Misplaced Tile and then Manhattan Distance. The lines are very similar at the start, but the node expansions begin to grow at an incredible rate for Uniform and Misplaced. This portrays that only Manhattan Distance should be used for searches with a large depth.

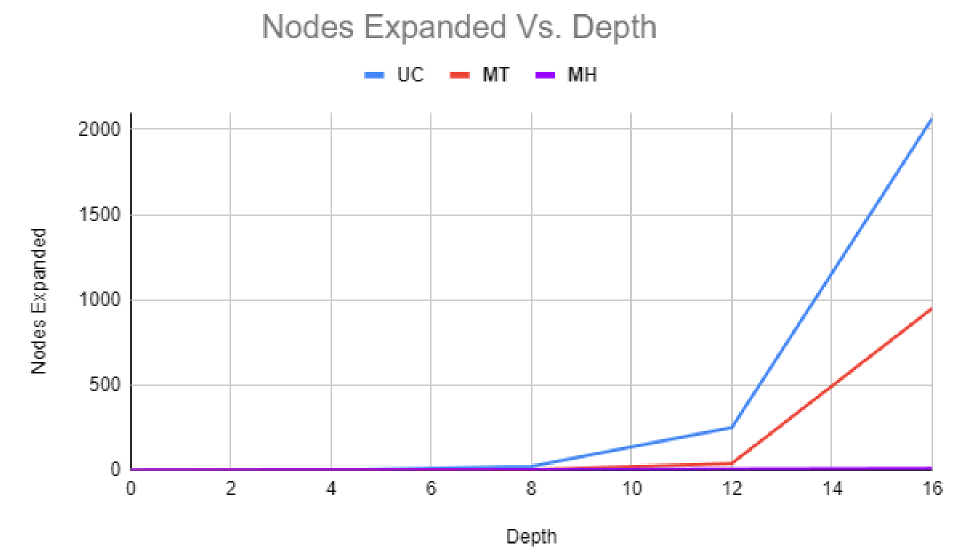


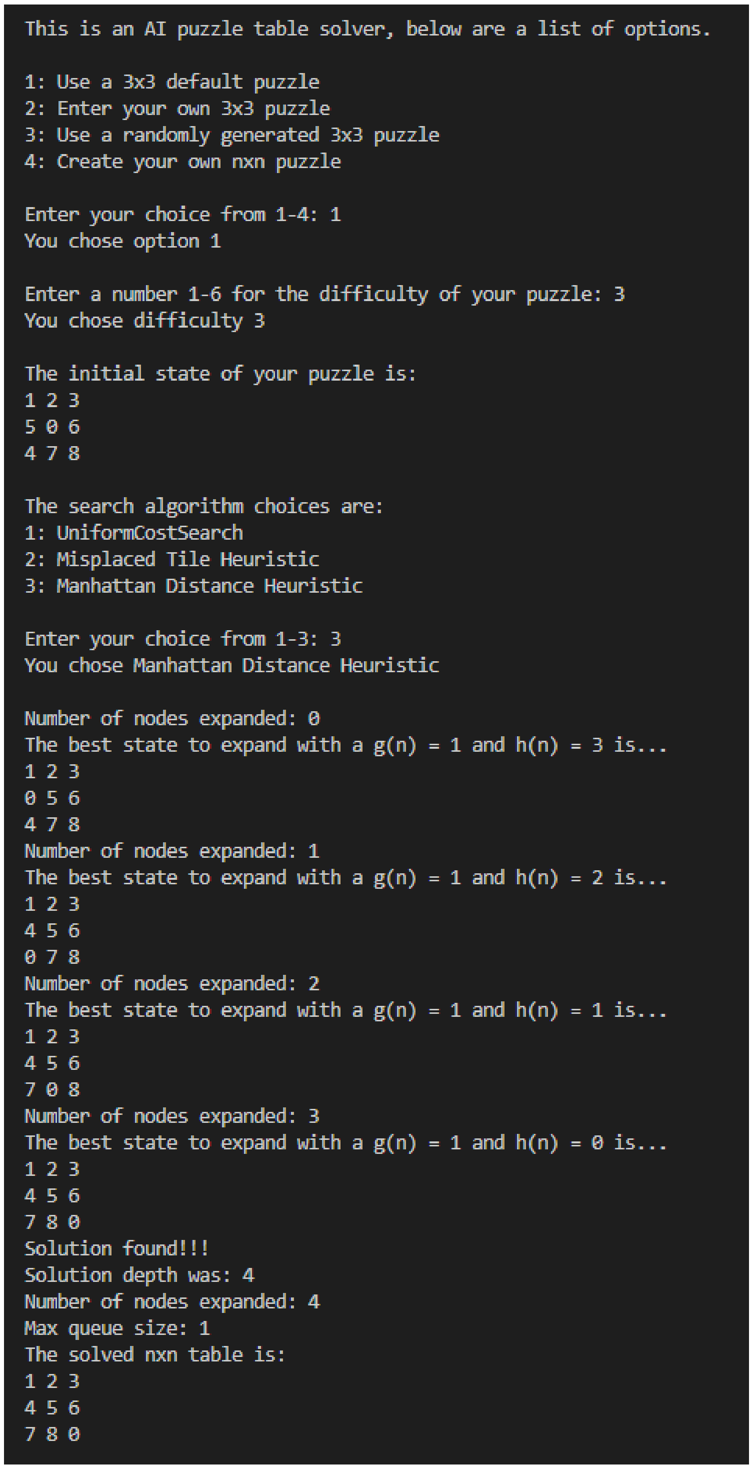
Figure 1: Nodes expanded compared to depth for the searches performed on the 8-Table.

[1] https://en.wikipedia.org/wiki/Taxicab_geometry

Conclusion

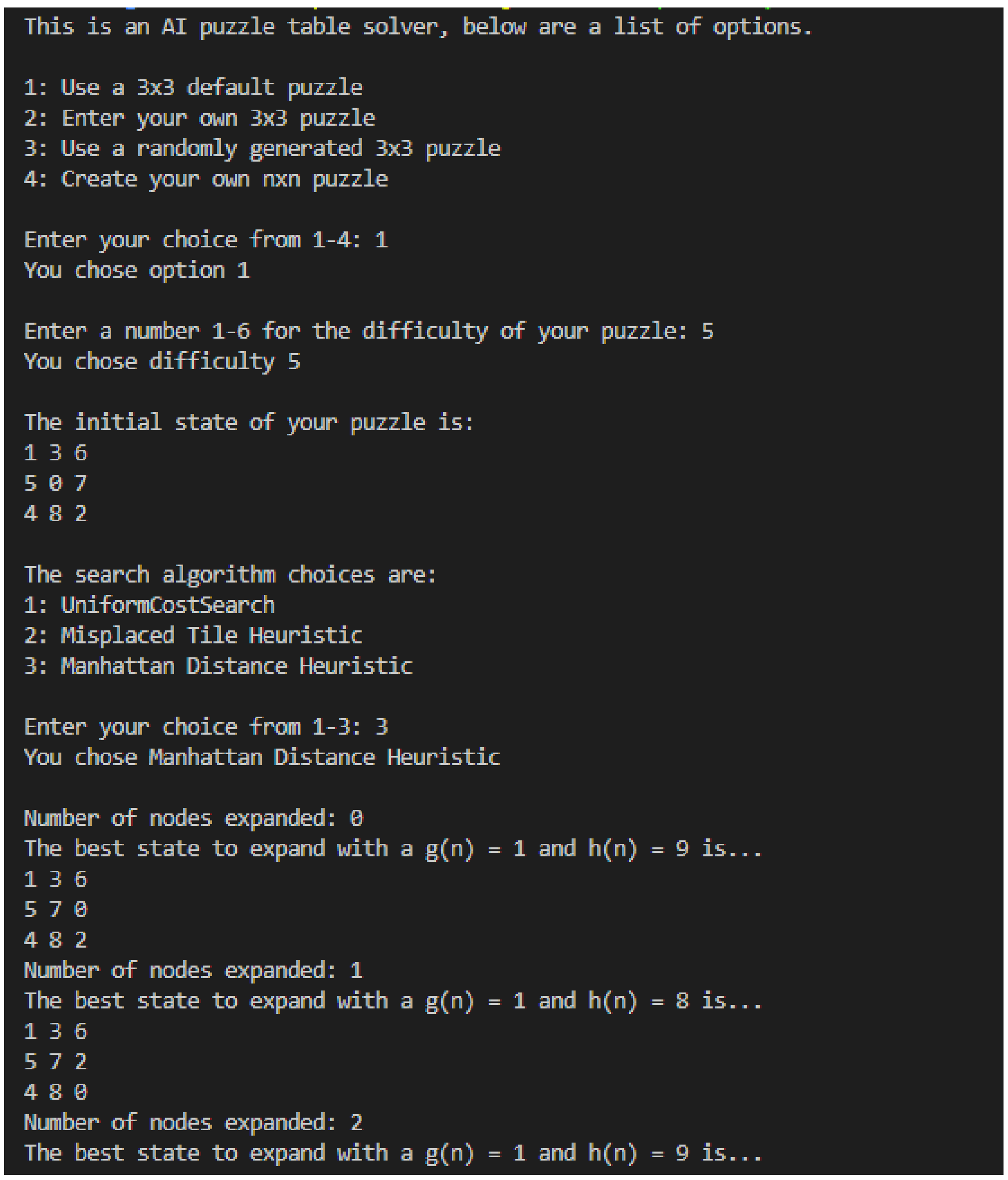
After testing the three searches, figure 1 shows us that the Manhattan Distance heuristic is the most efficient. Manhattan Distance arrives at the goal node without making as much node expansions as Uniform Cost or Misplaced Tile. This data is logical because a better heuristic can make search algorithms much more efficient at finding the goal state fast.

Easy problem output Manhattan Distance Depth 4

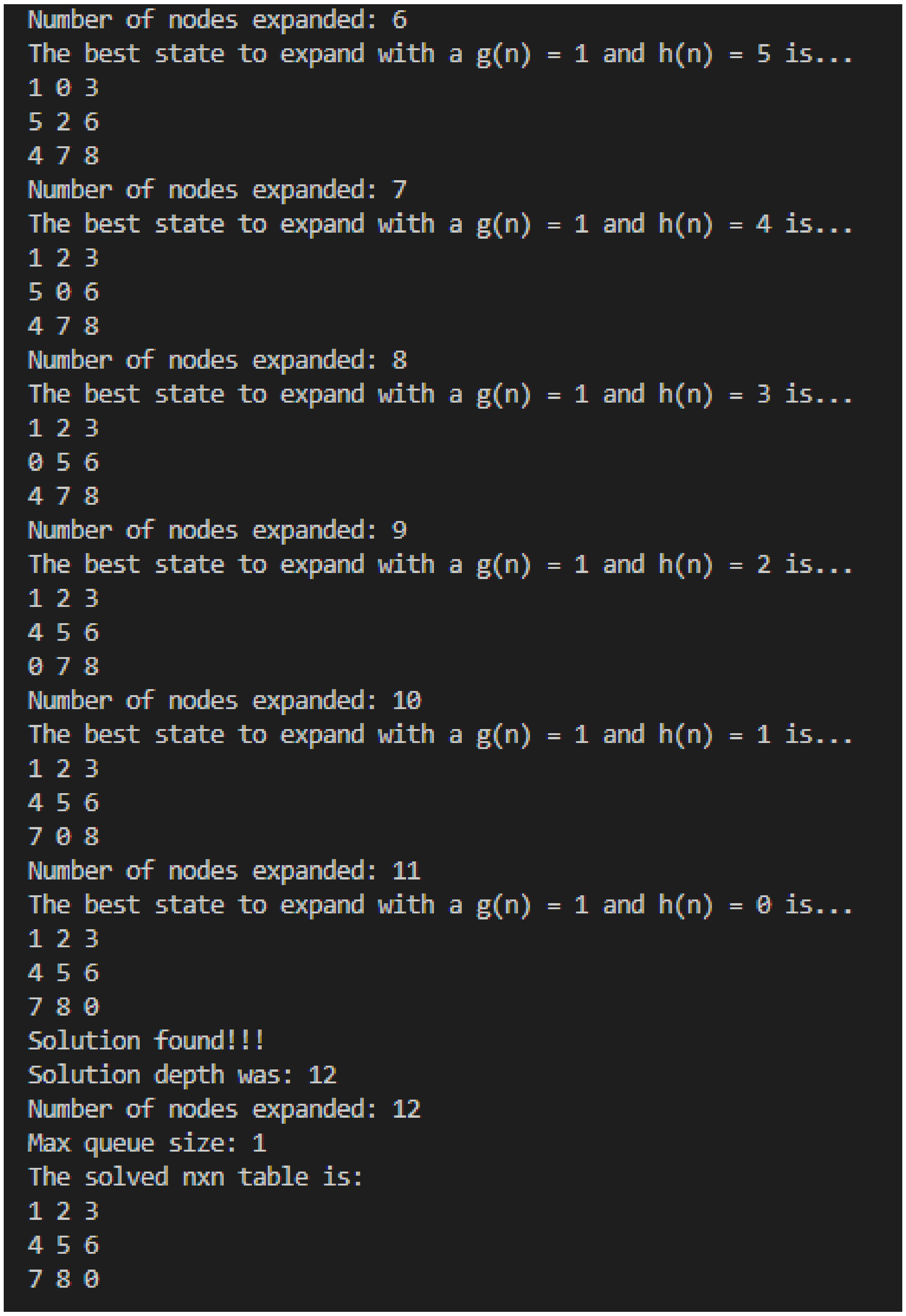


```
This is an AI puzzle table solver, below are a list of options.

1: Use a 3x3 default puzzle
2: Enter your own 3x3 puzzle
3: Use a randomly generated 3x3 puzzle
4: Create your own nxn puzzle

Enter your choice from 1-4: 1
You chose option 1

Enter a number 1-6 for the difficulty of your puzzle: 3
You chose difficulty 3

The initial state of your puzzle is:
1 2 3
5 0 6
4 7 8

The search algorithm choices are:
1: UniformCostSearch
2: Misplaced Tile Heuristic
3: Manhattan Distance Heuristic

Enter your choice from 1-3: 3
You chose Manhattan Distance Heuristic

Number of nodes expanded: 0
The best state to expand with a g(n) = 1 and h(n) = 3 is...
1 2 3
0 5 6
4 7 8
Number of nodes expanded: 1
The best state to expand with a g(n) = 1 and h(n) = 2 is...
1 2 3
4 5 6
0 7 8
Number of nodes expanded: 2
The best state to expand with a g(n) = 1 and h(n) = 1 is...
1 2 3
4 5 6
7 0 8
Number of nodes expanded: 3
The best state to expand with a g(n) = 1 and h(n) = 0 is...
1 2 3
4 5 6
7 8 0
Solution found!!!
Solution depth was: 4
Number of nodes expanded: 4
Max queue size: 1
The solved nxn table is:
1 2 3
4 5 6
7 8 0
```

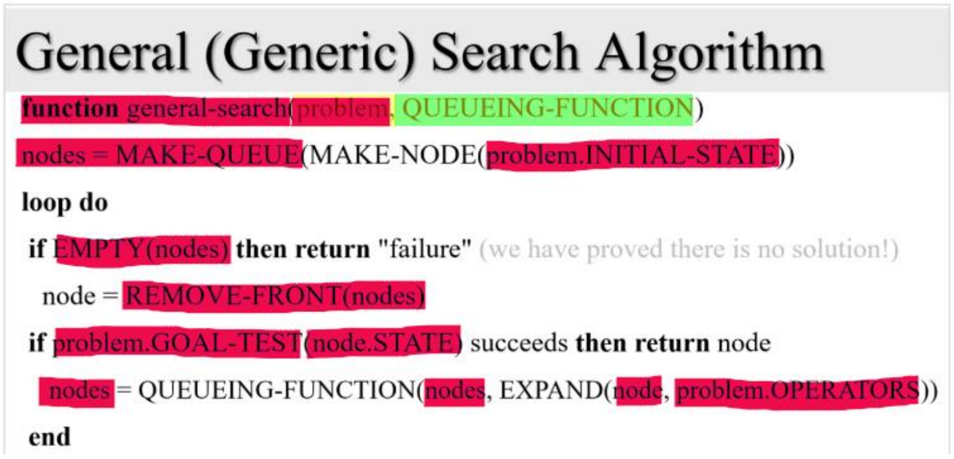
Hard problem output Manhattan Distance Depth 12



```
This is an AI puzzle table solver, below are a list of options.

1: Use a 3x3 default puzzle
2: Enter your own 3x3 puzzle
3: Use a randomly generated 3x3 puzzle
4: Create your own nxn puzzle

Enter your choice from 1-4: 1
You chose option 1

Enter a number 1-6 for the difficulty of your puzzle: 5
You chose difficulty 5

The initial state of your puzzle is:
1 3 6
5 0 7
4 8 2

The search algorithm choices are:
1: UniformCostSearch
2: Misplaced Tile Heuristic
3: Manhattan Distance Heuristic

Enter your choice from 1-3: 3
You chose Manhattan Distance Heuristic

Number of nodes expanded: 0
The best state to expand with a g(n) = 1 and h(n) = 9 is...
1 3 6
5 7 0
4 8 2
Number of nodes expanded: 1
The best state to expand with a g(n) = 1 and h(n) = 8 is...
1 3 6
5 7 2
4 8 0
Number of nodes expanded: 2
The best state to expand with a g(n) = 1 and h(n) = 9 is...
```

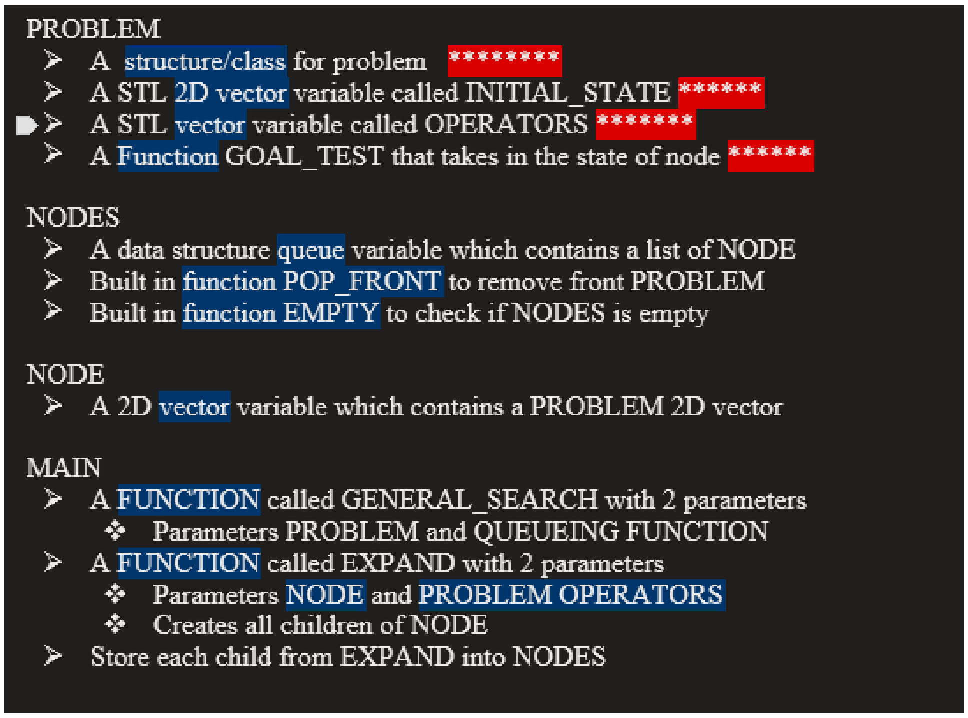


```
Number of nodes expanded: 6
The best state to expand with a g(n) = 1 and h(n) = 5 is...
1 0 3
5 2 6
4 7 8
Number of nodes expanded: 7
The best state to expand with a g(n) = 1 and h(n) = 4 is...
1 2 3
5 0 6
4 7 8
Number of nodes expanded: 8
The best state to expand with a g(n) = 1 and h(n) = 3 is...
1 2 3
0 5 6
4 7 8
Number of nodes expanded: 9
The best state to expand with a g(n) = 1 and h(n) = 2 is...
1 2 3
4 5 6
0 7 8
Number of nodes expanded: 10
The best state to expand with a g(n) = 1 and h(n) = 1 is...
1 2 3
4 5 6
7 0 8
Number of nodes expanded: 11
The best state to expand with a g(n) = 1 and h(n) = 0 is...
1 2 3
4 5 6
7 8 0
Solution found!!!
Solution depth was: 12
Number of nodes expanded: 12
Max queue size: 1
The solved nxn table is:
1 2 3
4 5 6
7 8 0
```



General (Generic) Search Algorithm

```
function general-search(problem, QUEUEING-FUNCTION)
nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
loop do
 if EMPTY(nodes) then return "failure" (we have proved there is no solution!)
   node = REMOVE-FRONT(nodes)
 if problem.GOAL-TEST(node.STATE) succeeds then return node
   nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
 end
```



PROBLEM
- A structure/class for problem *********
- A STL 2D vector variable called INITIAL_STATE ******
- A STL vector variable called OPERATORS *******
- A Function GOAL_TEST that takes in the state of node ******

NODES
- A data structure queue variable which contains a list of NODE
- Built in function POP_FRONT to remove front PROBLEM
- Built in function EMPTY to check if NODES is empty

NODE
- A 2D vector variable which contains a PROBLEM 2D vector

MAIN
- A FUNCTION called GENERAL_SEARCH with 2 parameters
  - Parameters PROBLEM and QUEUEING FUNCTION
- A FUNCTION called EXPAND with 2 parameters
  - Parameters NODE and PROBLEM OPERATORS
  - Creates all children of NODE
- Store each child from EXPAND into NODES