A2 - Wumpus World A* Search

CS4300
Fall 2016
Leland Stenquist - 2,4,6
Ryan Keepers - 1,3,5

1. Introduction

This experiment utilized the A* algorithm with the Manhattan Distance heuristic on random Wumpus World boards to determine the lowest cost path for the Wumpus World Agent to travel from the initial state ([1,1,0], the lower left corner of the board) to the x,y location of the goal (the gold). Throughout the experiment, two alternative patterns of feeding states into a priority queue were utilized, each one focusing on a different behavior of positioning new states relative to other states of equal cost. The measure of efficiency for both patterns is the total number of nodes expanded by the algorithm.

The experiment sought to answer the following questions: What is the mean number of nodes expanded for each insertion option? Does one option for insertion result in a significant change in efficiency relative to the other option (10% or greater)? And if so, how often is the one option worse than the other?

2. Method

The purpose of the program is to find a solution to the Wumpus World problem with the minimum cost. To do this the A* algorithm was implemented. The A* algorithm is simple. Starting with a root node it expands the root's children to create a tree. It adds two functions, g(x) and h(x), to compute f(x) or a cost. This cost is used to place the child nodes into a priority queue. Once a node has been expanded, the next node on the frontier(the end of the tree) is selected, by lowest cost, from the queue and expanded. As the frontier is expanded the current node is checked against the goal or end state to see if a solution has been found.

In this case g(x) was the distance, of the child node, from the root in the tree. h(x) was the Manhattan Distance of the child node to the goal or gold. These two were added together to create f(x), or the cost, and to decide where the child should go in the tree. It should be noted that two different implementations were considered in the case that a child node's cost was equal to that of a node already in the tree. One implementation was to add the child in front of any node's with the same cost or greater. The other was to add the child behind all nodes of equal or lesser cost.

This version of the A* algorithm adds children immediately when a node is expanded from the frontier. Before it adds children to the queue though, it calls a function `CS4300_create_children`. This function has uses all three available actions to create the children one by one. Each child must meet three requirements to be created. It must not have the same state as the current state. It must not have the same state as a child already in the queue, and it must not be on a game ending cell (Wumpus, Pit, Wumpus and Gold). If these conditions are met the child is created and put into the queue.

A while loop is used to expand through the nodes. At the start of each iteration of this while loop the current node is checked to see if it matches the x and y of the goal (gold). If this is found to be true, a solution has been found. It is returned with the tree.

Failure is a possibility in Wumpus World. There are situation where the gold is blocked off and unreachable. These situations are not ignored in this version of the A* algorithm. If the A* runs until there is no frontier then it will simply return an empty solution.

Two harnesses were implemented to provide different statistical analysis of the A* algorithm. `CS4300_a2_harness` is the main harness that was used this harness takes in as input the number of trials. A* is then run that number of times with a random board each time for both options 1 and 2. It returns a 3x4 array with the mean, var, confidence low, and confidence high for both options as well as the comparison of both options. This harness has a modified version, `CS4300_a2_harness_mod`, which returns a struct with nx1 matrices for plotting.

The other harness is `CS4300_a2_q3_harness`. This harness takes in option 1 or 2. It then runs the A* algorithm on all the combinations boards with only gold. It returns a 16x3 array with the x and y location of the gold and the number of nodes in the tree generated by the algorithm.

3. Verification of Program

To verify the behavior of our implementation of A* we worked out the algorithm by hand for three boards. Two boards were chosen to represent trivial situations (immediate success and the earliest possible failure), and one board to represent a generic solvable state. The algorithm was solved for each board, providing a list of states for the nodes expanded by the algorithm in the same order that the algorithm would produce them. Afterward, the program was run and the output from the program was compared to the hand-created output.

1. Trivial Board: Gold at (1,1)

Option 1		**Option 2**	
By Hand	Program	By Hand	Program

Nodes			
[1,1,0]	1 1 0	[1,1,0]	1 1 0
Solution			
[1,1,0,0]	1 1 0 0	[1,1,0,0]	1 1 0 0

2. Unsolvable board: Gold at (2,2). Pits at (2,1) and (1,2)

Option 1		**Option 2**	
By Hand	Program	By Hand	Program
Nodes			
[[1,1,0], [1,1,3], [1,1,1], [1,1,2]]	1 1 0 1 1 3 1 1 1 1 1 2	[[1,1,0], [1,1,3], [1,1,1], [1,1,2]]	1 1 0 1 1 3 1 1 1 1 1 2
Solution			
No Sol'n		No Sol'n	

3. Basic board: Gold at (2,2). No Pits.

Option 1		**Option 2**	
By Hand	Program	By Hand	Program
Nodes			
[[1,1,0], [2,1,0], [1,1,3], [1,1,1], [3,1,0], [2,1,3], [2,1,1], [2,2,1], [2,1,2]]	1 1 0 2 1 0 1 1 3 1 1 1 3 1 0 2 1 3 2 1 1 2 2 1 2 1 2	[[1,1,0], [2,1,0], [1,1,3], [1,1,1], [3,1,0], [2,1,3], [2,1,1], [1,1,2], [1,2,1], [2,1,2], [2,2,1], [1,3,1], [1,2,0],	1 1 0 2 1 0 1 1 3 1 1 1 3 1 0 2 1 3 2 1 1 1 1 2 1 2 1 2 1 2 2 2 1 1 3 1 1 2 0

		[1,2,2]]	1 2 2
Solution			
[[1,1,0,0], [2,1,0,1], [2,1,1,3], [2,2,1,1]]	1 1 0 0 2 1 0 1 2 1 1 3 2 2 1 1	[[1,1,0,0], [2,1,0,1], [2,1,1,3], [2,2,1,1]]	1 1 0 0 2 1 0 1 2 1 1 3 2 2 1 1

In all three cases the program's output exactly matched the output from the hand-written algorithm. These three cases represent the three types of boards which are possible to generate (trivial, solvable, and unsolvable), and thus show that the program accurately runs the algorithm.

Minimum and Maximum node counts:
Since the gold cannot spawn on the agent's starting position when using CS4300_gen_board, then the minimum count for both options would be 4 nodes. This occurs on an unsolvable board where the agent is blocked within the starting location.

The maximum nodes which a board can produce is 64 (4 directions for each of the 16 locations on the board). However, it is unlikely that the board would encounter the actual maximum, since it would encounter the goal state before exploring all possible directions of the location containing the gold. The actual maximum is likely between 60 and 63.

In a run of 2000 random boards for each option the resultant minimum for both was 4 nodes maximum for both was 60 nodes.

4. Data and Analysis

Table 1: Shows the mean, variance, confidence low, and confidence high for the A algorithm's 2 insertion options and for a comparison of the two options.*

Data for 2000 Randomly Generated Boards on Each Option				
Option	**Mean**	**Variance**	**Conf Low**	**Conf High**

Option 1	16.9060	156.9176	16.3570	17.4550
Option 2	19.6945	179.9071	19.1067	20.2823
O1 < O2	0.5490	0.2477	0.5272	0.5708

Table 1

As seen in table 1, option 1 and 2 produce different result with option 1(insert before >= cost) being the better choice. This is reflected in the mean, variance, and confidence intervals. The mean is approximately 3 nodes smaller in option 1. The variance in option 1 is approximately 23 nodes smaller.

Table 2: Plots the unsorted and sorted array of nodes generated by the A algorithm on random boards, for options 1 and 2, after 2000 trials.*

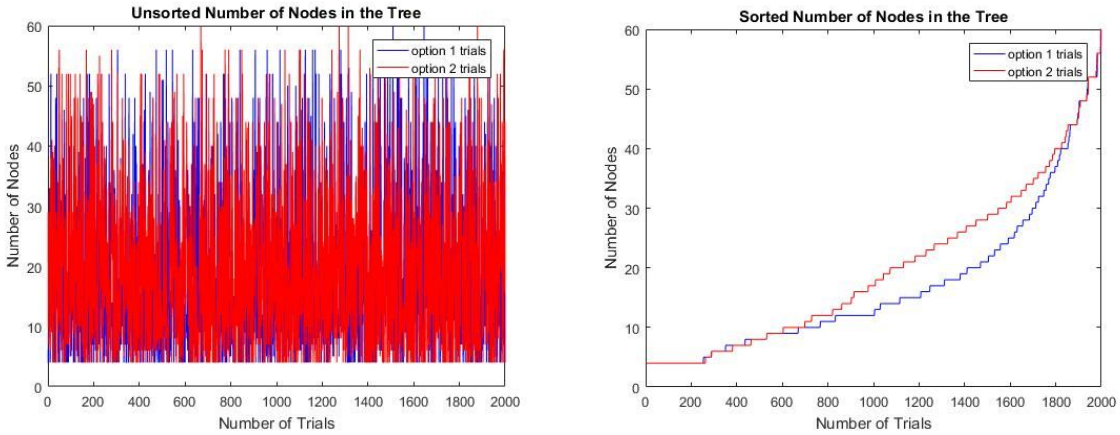

Table 2

Since the unsorted plot was hard to read a sorted version was created. Looking at the sorted version it is clear that from about 600 to 1800 trials, option 2(insert after <= cost) is generating more nodes. This also indicates that option 1 is more efficient than option 2.

Table 3: Histogram of number of nodes generated by the A algorithm on 2000 trials for random boards.*

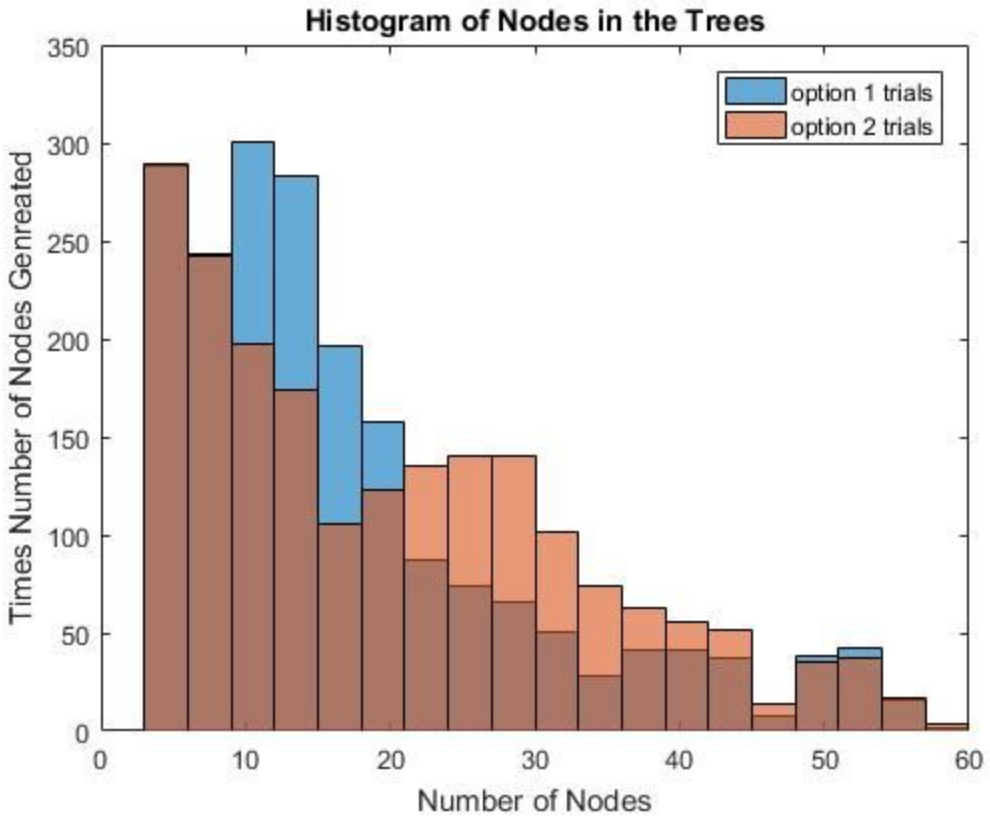

Table 3

Table 3 clearly shows that option is generating better results. It can be seen that option 1 represented as the blue bar is easily generating more results from about 4 to 20, but as the number of nodes increase from 20 to 48, option two, represented as salmon, clearly starts generating more cases. It is not until about 50 that both options 1 and 2 start to see similar results.

Table 4: Table for represents two 4x4 boards. Each x, y position on a board contains a number. That number is the size of tree that was created by the A algorithm to find the gold in that location assuming all the other cells are empty. One of the boards used option 1 to resolve ties on insertion. The other used option 2.*

Option 1

12	15	18	20
9	12	15	17
6	9	12	14
1	4	7	10

Option 2

12	25	38	50
9	20	31	41
6	14	22	29
1	4	7	10

Table 4

It is interesting to note cell (4,4) on these two tables. Option 2 generates 30 more nodes in its decision tree than option 1. This helps highlight why option 2 is less efficient. In the situation where the A* algorithm needs to find the gold at the farthest cell with no obstacles(wumpus, pit), it is generating a much larger decision tree. It can also be noted that only the cells with a 1 in the x or y position are equal for options 1 and 2. In all other cases, the tree generated by option 2 is greater.

5. Interpretation

The mean number of nodes expanded for Option 1 is generally 165, while the number of nodes expanded for Option 2 is generally 19.5. Therefore, on average, Option 2 generated 18% more nodes than Option 1.

The node counts for each option for 2000 trials (all 4000 boards were independently randomly generated) were compared in tandem. Option 1, trial 1 was compared against Option 2, trial 1, and so on. The mean expectation for Option 1 having less nodes than Option 2 is roughly 55%. Though this frequency might seem lower than expected given the average 18% increase in nodes, it does follow the hypothesis that 10% of randomly generated boards will produce less nodes when using Option 1.

The behavior causing the difference is a matter of how the Wumpus algorithm is designed. State children are always created in the sequence Forward, Turn Right, Turn Left. For any situation where the goal resides on a row above the starting row, this produces a node sequence where the first turn moves the agent's direction away from the goal, rather than towards it. When those nodes are placed in the priority queue, each of the Turn states will carry the same cost. As a result, the Option 1 (place before equal cost) will always place the Turn Left state before the Turn Right state.

As a result of this pattern, Option 1 will first expand nodes which bring the agent toward the center of the board, while Option 2 will first expand nodes which point the agent away from the center of the board. This makes the ordering for Option 1 more efficient for situations where the agent is facing Eastward (the starting direction) and the goal is above the agent (the majority of cases given that a little more than 75% of goal locations are above the agent's starting position). In that same situation, Option 2 will expand the left turn last of all the children, causing other nodes to expand first.

6. Critique

Attention to detail and instructions is a very important simulation concept. Constantly a simulation was run and on closer inspection of the instructions it had to be re-run. This is important because the purpose of the simulation is lost if it is not done correctly.

Changing what may seem like small algorithm implementations and comparing results helped to find a better solution. It was very surprising to see the difference between options 1 and 2 for queue insertion. These option created a simple change to the algorithm that had a significant effect. It seems testing small variation on an algorithm can be used to find more efficient implementations for that algorithm.

Large sample sizes and randomness are important. Generating a new random board for each of 2000 runs helped to give us more accurate statistics. Without these accurate statistics it would not have been as easy to compare options 1 and 2.

The problem was finding an acceptable A* algorithm to solve a board in Wumpus World using a search tree. While the search tree is guaranteed to find a possible solution, if one exists, the A* algorithm is not necessarily efficient. It took 4000 runs(2000 on each option) about 20 seconds to complete. This is on a game that created trees with a max of about a size of 60 and min of 4. If this algorithm were to be run on much larger problems, such as a chess game, it would quickly become slower.

Here are a few ideas of how to possibly improve this algorithm. The algorithm could possibly use percepts to possibly prune the tree making it faster. The percepts could also be used in the cost calculation to help order the queue. This is, of course, if percepts are provided and the children have access to them on creation.

7. Log

Ryan : 10 hrs.
Leland : 12 hrs