A2 - CS4300 Lab Report

1. Introduction

A* is an optimal search algorithm that is used by an agent to find its way to a certain destination. The formula for A* is f(n) = g(n) + h(n) where g(n) is the cost that it takes to get to a certain location X and h(n) is the Manhattan distance that it takes to get to the final destination from X.

Constraint propagation is a technique that is used to decrease the number of valid values for a particular variable. Decreasing the number of values can lead to the agent decreasing the number values in another variable which can once again cause the same effect for another variable. Arc-consistency is one way to do constraint propagation which works by tying binary constraints to the values in a domain. When using Arc-consistency there are a series of nodes and edges used where nodes are treated as different variables and edges represent a binary constraint between the two variables. When it comes to using Arc-consistency in this lab the question is

- How does using arc consistency improve the agent's performance (where performance is measured by the agent's success rate).
- Hypothesis : Using arc consistency will increase the performance, because the agent is no longer making random moves.

2. Method

The first agent initially just randomly selects FORWARD, ROTATE_LEFT, or ROTATE_RIGHT for its action until it reaches the gold, recording all of the cells it has been to in a matrix. It then uses this matrix with the A-Star algorithm to trace a safe path back to the initial agent position. The heuristic for this function is the Manhattan algorithm, which just sums the distances in the vertical and horizontal directions.

The second agent uses the arc consistency algorithm AC3 to determine whether there are any pitless nodes that the agent has not explored yet. The labels sets for each node consist of C for clear, P for pit, and B for breeze. The constraint graph is represented by an adjacency matrix for edges between nodes, a list of label sets for all the nodes, and a boolean matrix which constrains which arc states are valid. These constraints are that a pit cell can't be next to a unary clear cell and vice versa. Whenever it enters a new node that does not have a breeze, the surrounding cells can't have pits, so it updates the domain accordingly. The AC3 then iterates through every arc in the graph makes sure the constraints on all the arcs are still enforced. The algorithm then uses the results to find a safe cell that hasn't been explored and uses A-Star to find a path there. If this is not possible, it will revert to picking a random action very step.

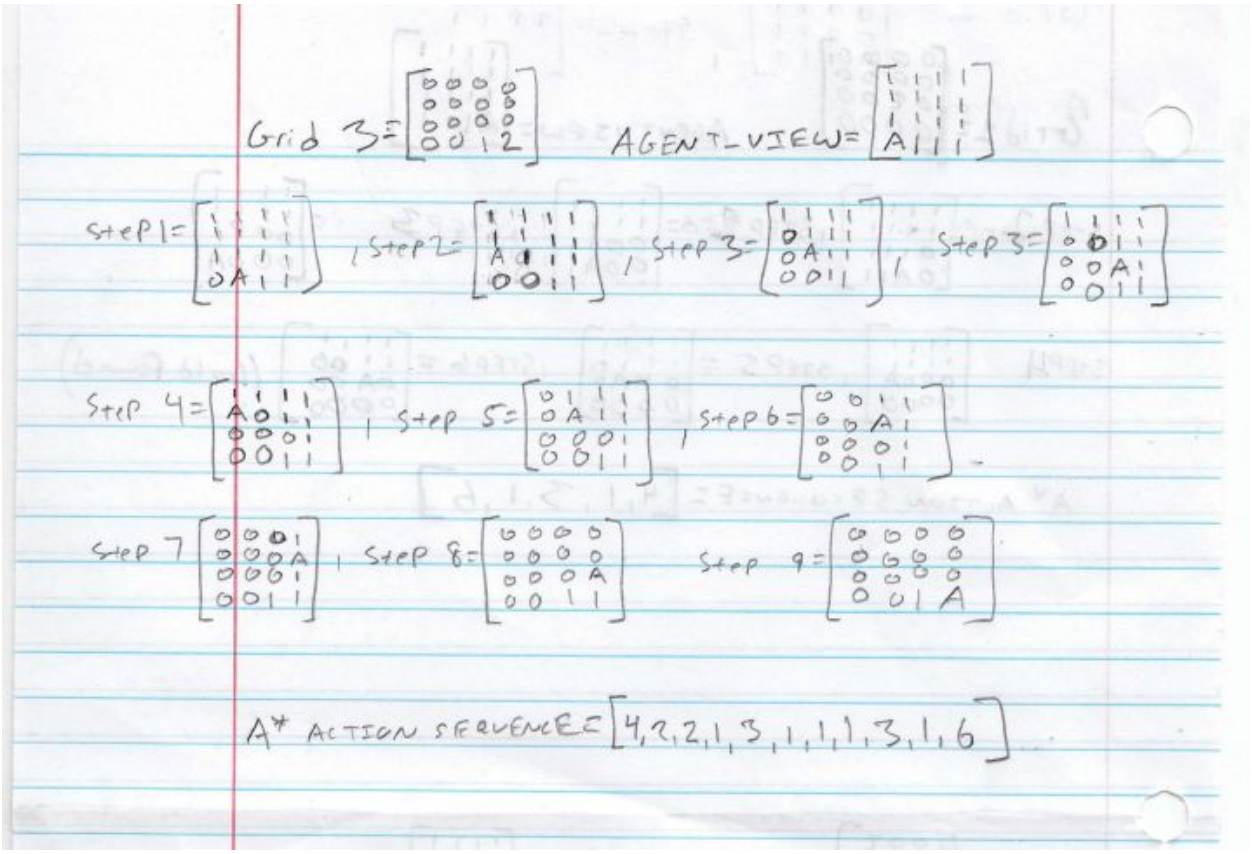
Performance for both is based on how the fraction of the time the agent climbs out of the grid after grabbing the gold.

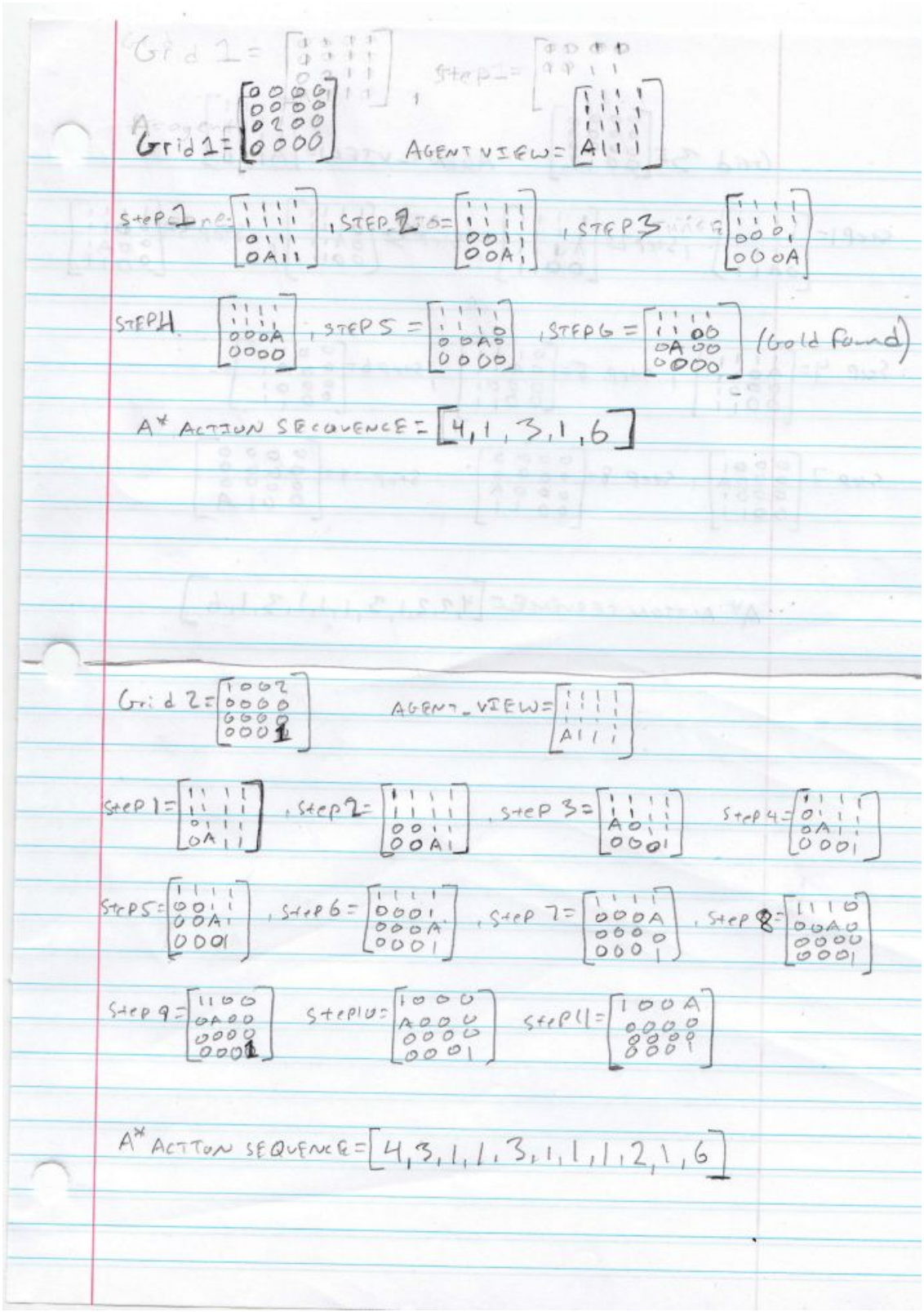
3. Verification of Program

In order to verify the program I created 3 different grids. Each grid has its own set of pits, and a gold location. I then hand wrote the steps that the agent would take to get to the gold. In each exercise I started out with what the grid looks like and what the agent is able to see. In each step I placed the agent into a new location based on the percepts and its knowledge of the world. I was also able to update the agent's knowledge of the world based on the percepts as well. Once the agent got to the gold I then was able give each of the actions needed to get back to the original spot and to climb out.

I was able to then compare my handwritten work against the code in order to verify my code. Because the actions were very similar and then only differences appeared due to the way an action would appear in the queue, I know that the code is working properly.

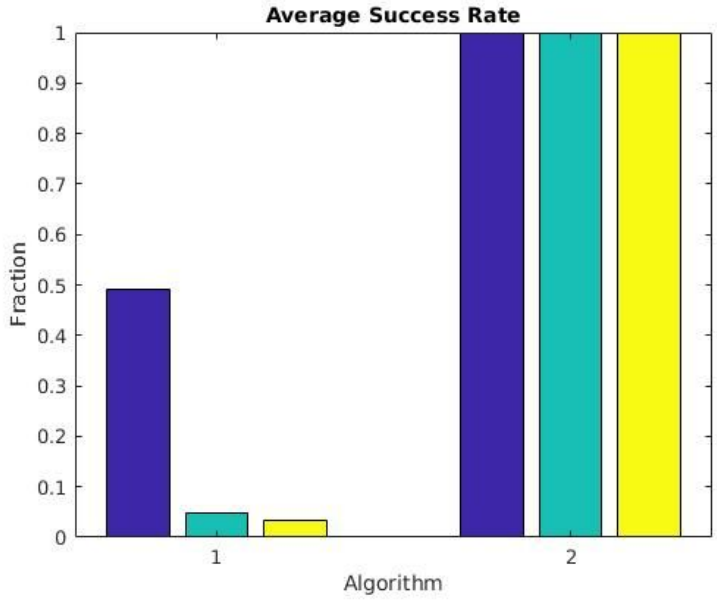
Also to verify the arc consistency function I also tested it against the n-queens code that was provided for the assignment. The function was able to return the proper charts and so I know that was working as well.




Grid 1=
Step 1=
Grid 1=
AGENT VIEW=
Step 1=
, STEP 2=
, STEP 3=
STEP 4
, STEP 5 =
, STEP 6 = (Gold found)
A* ACTION SEQUENCE = [4, 1, 3, 1, 6]
Grid 2=
AGENT_VIEW=
Step 1=
, Step 2=
, Step 3=
Step 4=
Step 5=
, Step 6=
, Step 7=
, Step 8=
Step 9=
Step 10=
Step 11=
A* ACTION SEQUENCE = [4, 3, 1, 1, 3, 1, 1, 1, 2, 1, 6]

4. Data and Analysis

Below is a graph of the average success rates of 500 trials of the Astar (left) and AC3(right) algorithms on each of the three boards shown above



In addition, the 95% confidence intervals for this data are as follows:

	AStar	AC3
Board 1	0.4460 - 0.5340	1 - 1
Board 2	0.0292 - 0.0668	1 - 1
Board 3	0.0181 - 0.0480	1 - 1

5. Interpretation

As can be seen the AC3 functions succeeded 100% of the time while the AStar function without proper pathfinding to the gold was around 50% for the first graph and less than 10% for the two other graphs. This is because without usings the AStar functions used random actions to find the gold. In the first grid there were no pits and so the chance of success were higher because the agent could not die, so as long as it found the gold, it could get back with the AStar algorithm. The other two grids did have pits and so the agent's chance was greatly reduced because it could fall in a pit which would result in an automatic failure.

The reason why the AC3 functions had 100% results was because the pits were spaced in a way that the agent never had to make any guesses. Because of that it could alway properly navigate the grid and eventually find gold.

6. Critique

The most obvious we could change to make this experiment more useful would be choosing different boards to test our agents on, specifically boards that have a greater number of pits. This would force the AC3 to be less deterministic, so it wouldn't just succeed 100% of the time. We could also increase the number of repetitions on each board, as due to time constraints, we were only able to run each board 500 times. This would tighten our confidence intervals and help make our mean averages more accurate. One interesting way to possibly improve the success rate of the AC3 agent could be to change its behavior when there are no known "safe" cells. Instead of picking actions randomly, it could choose a random cell that has not been visited and use Astar to trace a path there. That it would always be attempting to expand its model of the world, which could help reduce instances when the agent runs out of steps because is just acting randomly.

7. Log

William Garnes: Approximately 20 hours was spent coding

Cameron Jackson: 18 hours