

**Assignment 1**

**CSC3309 Artificial Intelligence**

Brandon Crane

Supervised by:

Dr. Violetta Cavalli-Sforza

3 October, 2016

1. Agents
   1. Simple Reflex – Smoke Alarm
      1. A smoke alarm is a device that detects the presence of smoke (through various methods). The reaction to the presence of smoke is to sound off an alarm. This device is only worried about the current precept, and acts in a stimulus – response behavior.
   2. Model-Based Reflex – Smart thermostat
      1. If a thermostat is set to “auto” mode, it has to regulate the temperature up and down to maintain the desired temperature (selected by user), as opposed to just “cool” or “heat”. The thermostat is trying to conserve as much energy as possible, so it learns the times of day that the user is out of the house (via motion tracking, door sensors, or something similar) and gives a greater temperature tolerance for when it should act in such a way as to get closer to the set temperature. The user does not need the temperature to be steady if they are not present. This agent keeps track of the parts of the world it cannot see (*internal* *state*), and it understands what will happen to the environment in response to its actions (*model*).
   3. Goal Based – Tic-tac-toe AI
      1. This agent is a good example of a goal based agent. The goal is to win – get three in a row without the opponent doing so first. It will have the same components of a model- based reflex agent, but it will use search or planning to formulate its next move.
   4. Utility Based – Travelling Salesman Business Application
      1. This is a good example because, this agent has a goal but it also takes into account certain costs. For example, traffic jams and road work can increase the cost of a valid path to the destinations. The agent can produce multiple path results (most optimal paths) and lets the user pick which path they want, but doesn’t have to behave this way.
   5. Learning – Go AI
      1. The search space for a Go game is too big for a simple search. Thus, a learning based agent is better for the job. A critic to give feedback on the actions of the agent is crucial to this game, as many opponents will have different strategies. A problem generator is also important so that it can look forward before it makes a move.
2. Environments
   1. Unobservable – Unlit room with no windows (as dark as you can get) with an agent that relies on a camera to act. Normal cameras depend on light to gather information from the environment around them. If you take away the light, the environment becomes unobservable.
   2. Strategic – Checkers against another player. The environment will only change if the agent, or another agent changes it. If an agent doesn’t act, there is no change. Therefore, this environment is strategic.
   3. Episodic environment – Grocery store checkout. Each item is scanned once. The result of the next scan does not depend on the previous scan.
   4. Discrete – Smoke Alarm. There are only two percepts and one action. Smoke, no smoke, and the reaction to smoke is to sound the alarm.
   5. Static – Peg Solitaire. The pegs do not change positions while the agent is deliberating the next move.
   6. Single-agent – Automated Taxi. The taxi is only concerned with reaching its destination. There is no other agent competing to get to that destination first. This environment can become multi-agent competitive if maximizing profit among a certain region is the goal.
   7. Known – Catapult on Earth. Earth’s gravity constant is known for most locations, and thus the calculations will produce predicted results.
3. Problem Formulation
   1. The representation for the state will be a one dimensional array. This will be filled with Boolean values (1 or 0) to represent if the spot has a peg in it or not. The array will act like a 2 dimensional array – if you divide the index by the height of the puzzle you will get the row number, and the remainder will be the column number (in a 2D view). This will promote faster access times.
   2. The available actions are jumps. A jump is taking a peg, moving orthogonally over an adjacent peg, and landing in an empty spot on the opposite side of that adjacent peg. The peg jumped over is then removed. This is the only operator in this game.
   3. C++
      1. int board[50] = {0, 0, 0, 1, 1, 1, 0, 0,

0, 0, 1, 1, 1, 0, 0,

1, 1, 1, 1, 1, 1, 1,

1, 1, 1, 0, 1, 1, 1,

1, 1, 1, 1, 1, 1, 1,

0, 0, 1, 1, 1, 0, 0,

0, 0, 1, 1, 1, 0, 0};

* + 1. A single dimensional array will be able to accommodate all board shapes including an equilateral triangle with sides of length 5. All that is needed are shape specifications given by the user at the beginning of the program.
  1. 1. bool goalTest (int \*board)

{

int total = 0;

for (int i = 0; i < 50; i++)

{

total += board[i];

}

cout << "total : " << total << endl;

if (board [25] == 1 && total == 1)

{

return true;

}

else

{

return false;

}

}

* + 1. bool inBounds(int pegNumber)

{

int outOfBounds[17] = {0, 1, 2, 6, 7, 8, 9, 13, 14, 36, 37, 41, 42, 43, 44, 48, 49};

for (int i=0; i < 17; i++)

{

if (pegNumber == outOfBounds[i])

{

return false;

}

}

return true;

}

int\* jump(int board[], int pegNumber, int direction)

{

// NSEW, North

if (direction == 0 && board[pegNumber - 14] == 0 && inBounds(board[pegNumber - 14]) )

{

board[pegNumber] = 0;

board[pegNumber - 14] = 1;

board[pegNumber - 7] = 0;

return board;

}

// South

else if (direction == 1 && board[pegNumber + 14] == 0 && inBounds(board[pegNumber + 14]) )

{

board[pegNumber] = 0;

board[pegNumber + 14] = 1;

board[pegNumber + 7] = 0;

return board;

}

// East

else if (direction == 2 && board[pegNumber + 2] == 0 && inBounds(board[pegNumber + 2]))

{

board[pegNumber] = 0;

board[pegNumber + 2] = 1;

board[pegNumber + 1] = 0;

return board;

}

// West

else if (direction == 3 && board[pegNumber - 2] == 0 && inBounds(board[pegNumber - 2]))

{

board[pegNumber] = 0;

board[pegNumber - 2] = 1;

board[pegNumber - 1] = 0;

return board;

}

else

{

return board;

}

}

void successor(int board[], int iPrime, int jPrime)

{

for (int i=iPrime; i<47; i++)

{

for(int j=jPrime; j<3; j++)

{

if(!goalTest(board))

{

successor(jump(board, i, j), i, j);

}

}

}

}

* + 1. void increaseStepCost(int &cost)

{

cost++;

}

void reduceStepCost(int &cost)

{

cost--;

}

1. Thinking about search
   1. It is very large. We can reduce this because of symmetry. Many states are rotations or flips of one state, and therefore, for this problem, are considered the same state. Recognizing this will reduce the branching factor.
   2. Any form of Depth First Search will be good for this puzzle. The puzzle does not have an infinite search space, and can be solved in a reasonable amount of moves. So instead of slowly but surely reaching a definite goal state in BFS, you can take your chances going straight for the goal and then backtracking with DFS.