Games, Puzzles, and Computation

CSCI 4341 Notes from 4/10/17 to 4/14/17

Group J: Cesar Lozano, Mario Salinas, Fernando Torres, Eric

What Happened this Week:

- Looked at AND, OR, CHOICE, and CHAIN gadgets for different combinatorial games.
- Explored Artificial Intelligence, a brief history of it, and how it relates to games.

Gadgets

Amazons:

Amazons was invented by Walter Zamkauskas in 1988. Both human and computer opponents are available for Internet play, and there have been several tournaments, both for humans and for computers.

Amazons has several properties which make it interesting for theoretical study. Like Go, it's endgames naturally separate into independent subgames; these have been studied using combinatorial game theory. Amazons has a very large number of moves available from a typical position, even more than in Go. This makes straightforward search algorithms impractical for computer play. As a result, computer programs need to incorporate more high-level knowledge of Amazons strategy.

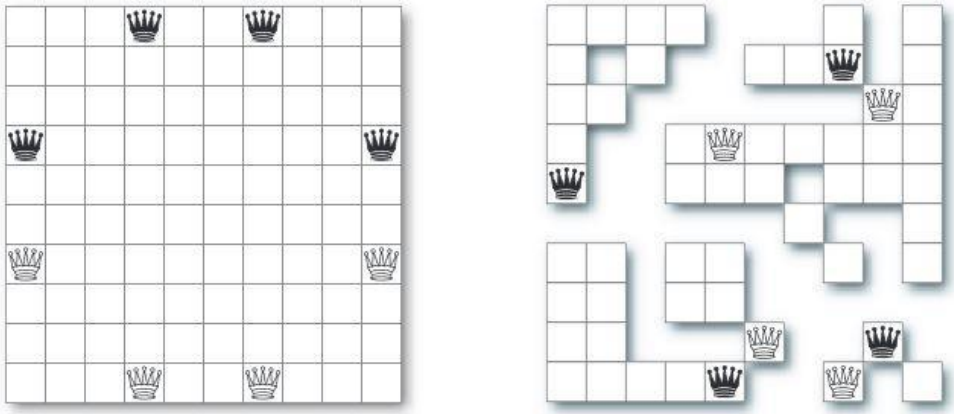


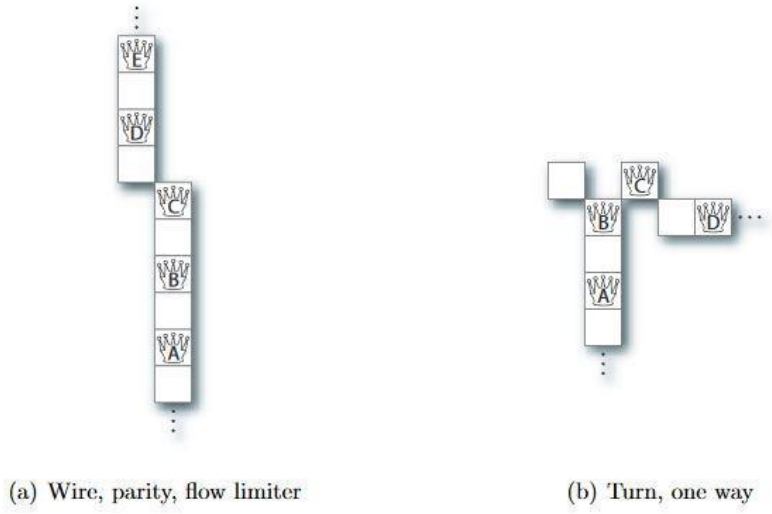
Figure 10-1: Amazons start position and typical endgame position.

Amazons Rules:

Amazons is normally played on a 10x10 board. The standard starting position, and a

typical endgame positions, are shown above. Each player has four amazons, which are immortal chess queens. White plays first, and play alternates. On each turn a player must first move an amazon, like a chess queen, and then fire an arrow from that amazon. The arrow also moves like a chess queen. The square that the arrow lands on is burned off the board; no amazon or arrow may move onto or across a burned square. There is no capturing. The first player who cannot move loses.

Amazons is a game of mobility and control, like Chess, and of territory, like Go. The strategy involves constraining the mobility of the opponent's amazon, and attempting to secure large isolated areas for one's own amazons. In the endgame shown in Figure 10-1, Black has access to 23 spaces, and with proper play can make 23 moves; White can also make 23 moves. Thus from this position, the player to move will lose.



(a) Wire, parity, flow limiter (b) Turn, one way

Figure 10-2: Amazons wiring gadgets.

Basic Wiring:

Signals propagate along wires, which will be necessary to connect the vertex gadgets. The figure 10-2(a) above shows the construction of a wire. Suppose that amazon A is able to move down one square and shoot down. This enables amazon B to likewise move down one and shoot down; C may now do the same. This is basic method of signal propagation. When an amazon moves backward (in the direction of input, away from the direction of output) and shoots backward, I will say that it has retreated.

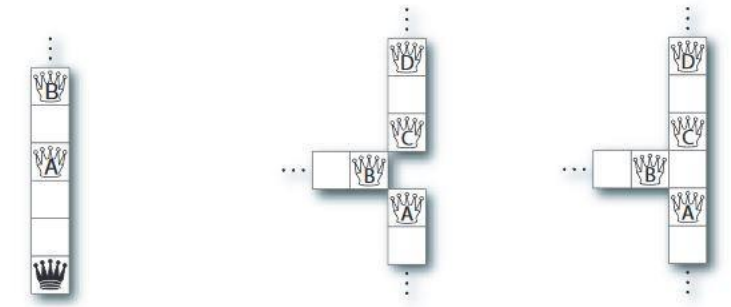
10_2 (a) illustrates two additional useful features. After C retreats, D may retreat, freeing up E. The result is that the position of the wire has been shifted by one in the horizontal direction. Also, no matter how much space is freed up feeding into the wire, D and E may still only retreat one square, because D is forced to shoot into the space vacated by C.

10_2(b) shows how to turn corners. Suppose A, then B may retreat. Then C may retreat, shooting up and left. D may then retreat. This gadget also has another useful property; signals

may only flow through it in one direction. Suppose D has moved and shot right. C may then move down and right, and shoot right. B may then move up and right, but it can only shoot into the square if just vacated. Thus, A is not able to move up and shoot up.

By combining the horizontal parity-shifting in Figure 10_2(a) with turns, we may direct a signal anywhere we wish. Using the unidirectional and flow-limiting properties of these gadgets, we can ensure that signals may never back up into outputs, and that inputs may never retreat more than a single space.

Variable, AND, OR, CHOICE:



(a) Variable (b) AND (c) OR, CHOICE

Figure 10-3: Amazons logic gadgets.

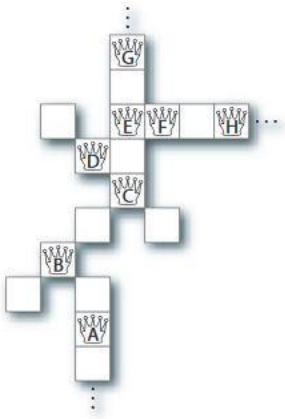


Figure 10-4: Amazons FANOUT gadget.

The variable gadget is shown in Figure 10-3(a). If White moves first in a variable, he can move A down, and shoot down, allowing B to later retreat. If Black moves first, he can move up and shoot up, preventing B from ever retreating. The AND and OR gadgets are shown in Figures 10-3(b) and 10-3(c). In each, A and B are the inputs, and D is the output. Note that, because the inputs are protected with flow limiters (Figure 10-2(a)), no input may retreat more than one

square; otherwise the AND might incorrectly activate. In an AND gadget, no amazon may usefully move until at least one input retreats. If B retreats, then a space is opened up, but C is unable to retreat there; similarly if just A retreats. But if both inputs retreat, then C may move down and left, and shoot down and right, allowing D to retreat. Similarly, in an OR gadget, amazon D may retreat if and only if either A or B first retreats.

The existing OR gadget also suffices as a CHOICE gadget, if we reinterpret the bottom input as an output: if B retreats, then either C or A, but not both, may retreat.

Fanout:

Implementing a FANOUT in Amazon's is a bit trickier. The gadget shown in Figure 10_4 above accomplishes this. A is the input; G and H are the outputs. First, observe that until A retreats, there are also no useful moves to be made. C, D, and F may not move without shooting back into the square they left. But if A retreats, then the unit and shoot two, but nothing is accomplished by this. But if A retreats, then the following sequence is enables: B down and right, shoot down; C down and left two, shoot down and left; D up and left, shoot down and right three; E down two, shoot down and left; F down and left, shoot left. This free up space for G and H to retreat, as required.

Cross Purposes:

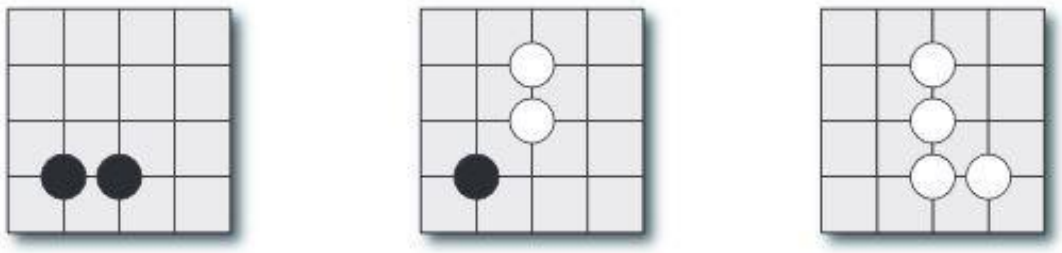


Figure 10-8: An initial Cross Purposes configuration, and two moves.

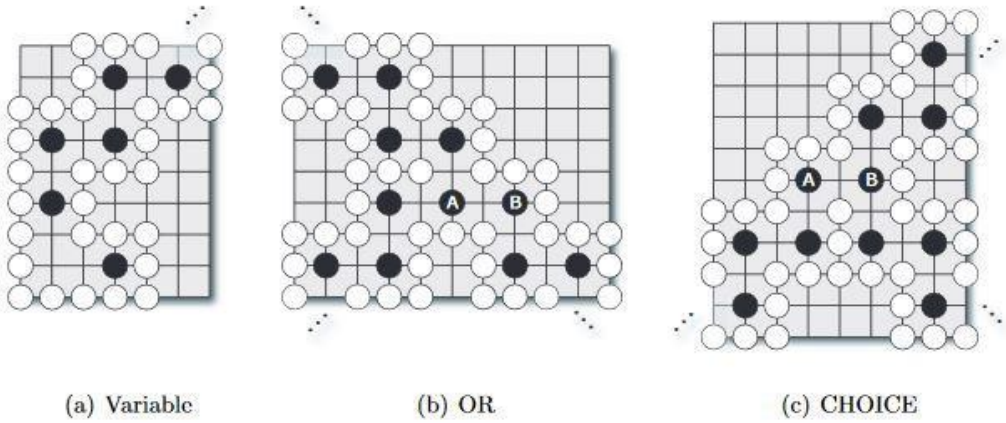
Cross Purposes Rules. Cross Purposes is played on the intersections of a Go board, with black and white stones. In the initial configuration, there are some black stones already on the board. A move consists of replacing a black stone with a pair of white stones, placed in a row either directly above, below, to the left, or to the right of the black stone; the spaces so occupied must be vacant for the move to be made. See Figure 10-8. The idea is that a stack of crates, represented by a black stone, has been tipped over to lie flat. Using this idea, we describe a move as *tipping* a black stone in a given direction.

The players are called *Vertical* and *Horizontal*. Vertical moves first, and play alternates. Vertical may only move vertically, up or down; Horizontal may only move horizontally, left or right. All the black stones are available to each player to be tipped, subject to the availability of empty space. The first player unable to move loses.

I give a reduction from planar Bounded Two-Player Constraint Logic showing that Cross-Purposese is PSPACE-complete.

Image from Games, Puzzles, and Computation by Robert Aubrey Hearn

Variable, OR, CHOICE:



(a) Variable (b) OR (c) CHOICE

Figure 10-11: Cross Purposes variable, OR, and CHOICE gadgets.

The variable gadget is shown in Figure 10-11(a). If Vertical moves first in a variable, he can begin to propagate a signal along the output wire. If Horizontal moves first, he will tip the bottom stone to block Vertical from activating the signal.

The OR gadget is shown in figure 10-11(b). The inputs are on the bottom; the output is on the top. Whether Vertical activates the left or right, allowing Vertical to activate the output. Here we must again be careful with available moves. Suppose Vertical has activated the right input? After he tips stone B down, Horizontal will have no move; he will already have tipped stone A left. This would give Vertical the last move even if he were unable to activate the final AND gadget; therefore, we must prevent this from happening. We will show how to do so after

describing the CHOICE gadget. As with Amazons the Konane, the existing OR gadget suffices to implement CHOICE, if we reinterpret it. This time the gadget must be rotated. The rotated version is showing in version 10-11(c). The input is on the left, and the outputs are on the right. When Vertical activates the inputs, and tips stone A down, Horizontal must tip stone B left. Vertical may then choose to propagate the signal to either the top or the bottom output; either choice blocks the other.

Artificial Intelligence:

Heuristics: Pattern matching that illicites a reasonable move.

- What we would consider intuition in humans.
- Artificial Intelligence utilizes some sort of pattern matching to decide the next move to play.

Hand Scripted AI: Just checks conditions set by programmers. It doesn't really learn.

Learning AI:

- Queue Learning
- Artificial Neural Network
- Monte Carlo Tree Search

Queue Learning:

- Simple, flexible, but limited because it required more user oversight.
- Stems from the Markov Decision Process:
    - Given a choice of paths, choosing a correct one "rewards" the AI in some way.
    - If you have every possible state of a game, every move will put you in different state.
    - Q learning attempts to learn the value of being in a given state and making a specific action at this state.
    - In Figure 11 states are the green nodes and actions are the orange nodes. Each edge has a cost and each yellow arrow is a "reward" for using that edge which will serve to update q.

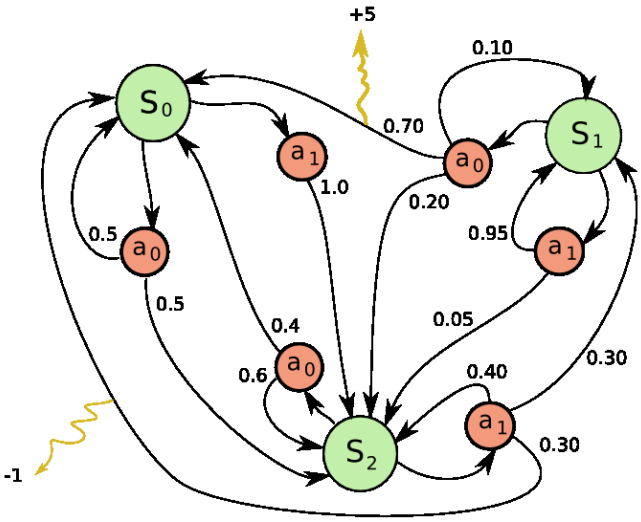


Figure 11

Process:

- Set a table with States, Moves, and a variable q that will serve to show how desirable a move is.
- Randomly set q; the output begins to stabilize as more reinforcement is introduced.
- Having an update formula for q is necessary:

Basic Formula: q = q + x where x = importance of q

Problems with Queue Learning:

- Table is often very large (large space complexity).
- Not as fast as other methods.

Good Tutorial on How to Use and Implement Queue Learning with a Neural Network:

https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0

Neural Network:

-Artificial neural networks (ANN) are systems for computing,
-Loosely based on the mammalian brain, and on how neurons work.
-Rely on interconnected elements which process inputs.
-By approximating a desired output, the network can approach and help understand how some systems work.

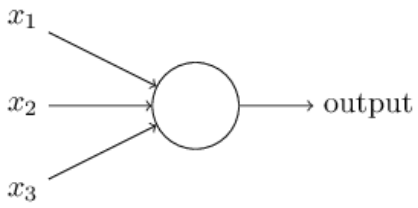
ANN's are good when:

-Discovering regularities in a set of patterns
-Data volume and diversity are very large
-The relationship between variables is vaguely understood
-Relationships are hard to show using conventional approaches.

An simple way to think of ANN's is seeing them as obscured functions, of which you can only observe outputs, from given inputs. If you give the obscure function many inputs, you could begin to notice patterns in the outputs. This way, using ANN's may help shed some light on obscured problems.

Structure:

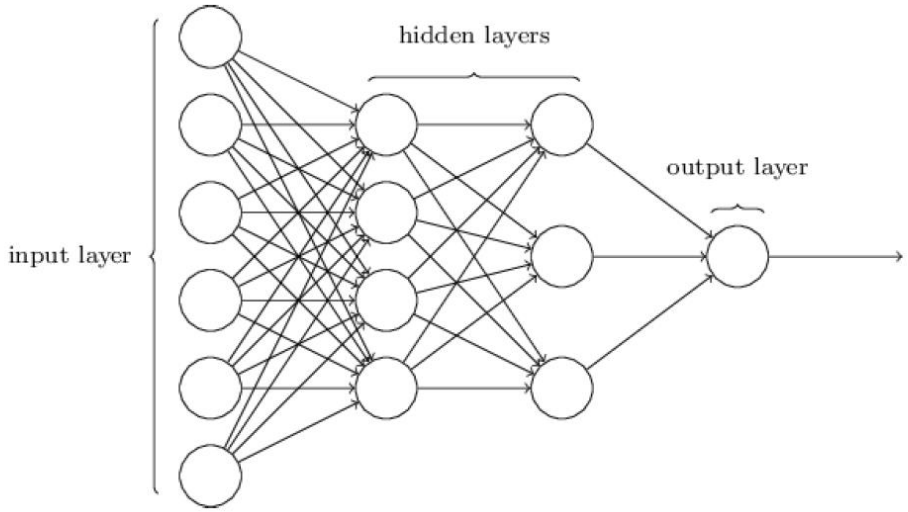
-Artificial Neurons: These nodes are the building blocks of the networks. They take inputs, and generate an output.
-The inputs have respective weights indicating their importance in the node's calculations. -They are summed into a weighted sum
-If the sum meets the neuron's threshold value, or bias, the neuron may fire.



-Layers: Neural networks are organized in layers. These layers process input which may come from past layers, or may be the original input handed to the system. This way, deeper layers can make more and more complex decisions based on their given inputs.

Networks with many layers may become a means to engage in sophisticated decision making.

-Input layer: Information supervisors give to the system.
-Output layer: The result the system returns.
-Hidden layers: Neither input nor output layers, just processinglayers.



Monte Carlo Tree Search (MCTS):

- MCTS is a method for making optimal decisions in AI problems such as in combinatorial games. It combines random simulations with a precision tree search.
- Tree is built node by node according to the outcomes of simulated playouts.

Steps:

1. Selection: starting at the root node, recursively select optimal child nodes until a leaf node is reached.
2. Expansion: if the leaf isn't a terminal node (not an end to the game) then create one or more child nodes and select one.
3. Simulation: run a simulated playout form the selected not until a result is reached.
4. Backpropagation: update the current move sequence with the simulation result.

- Things to Keep in Mind: Each node must contain the estimated value based on simulation results and the number of times it has been visited. In its simplest and most memory efficient implementation MCTS will add one child node per iteration, but it is often beneficial to add more than one child node per iteration.

Node Selection:

- A node is chosen if it maximizes some quantity such as a return or reward.
- Upper Confidence Bound formula generally used:

$$v_i + C \times \sqrt{\frac{\ln N}{n_i}}$$

- v_i = *estimated value of the node*
- C = *tunable bias parameter*
- n_i= *number of times the node has been visited*
- N = *number of times the node's parent has been visited*

Other Implementation to Consider is UCB1:

https://jeremykun.com/2013/10/28/optimism-in-the-face-of-uncertainty-the-ucb1-algorithm/

Tutorial on MCTS:

https://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/

Additional References:

http://outlace.com/Reinforcement-Learning-Part-3/

https://medium.com/@harvitronix/using-reinforcement-learning-in-python-to-teach-a-virtual-car-to-avoid-obstacles-6e782cc7d4c6

http://neuralnetworksanddeeplearning.com/chap1.html

http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html