Week 11

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Announcements 1
2. AI Search 1
    2.1. Adversarial Search 3
3. Beyond Board Games 9

1. Announcements

- Quiz 1 on Wednesday! Reviews at Yale & Harvard, Harvard review recorded online.
- Last lecture at Yale Friday, 11/20 (cake!), last lecture at Harvard Monday, 11/23 (also cake!).
- CS50 Expo at Yale Friday, 11/20, 3 PM - 4:30 PM

2. AI Search

- Say we have a task we want to perform - e.g., booking a set of flights from Boston to San Francisco - and we want a software agent to automate it.
- We can look and see, if I'm in Boston, what flights are available to me.
    # Let's say there's a flight to Miami, a flight to Chicago, and a flight to New York.
    # Then from each of those cities, we can see what options we have.
        # Let's say from Chicago, there's a flight straight to San Francisco and a flight to Denver.
        # Maybe the San Francisco option is perfect for me, but maybe it's too expensive and I want to look at other options.
        # Instead, we could fly to Denver, and then to Austin, and then to Phoenix, and then to San Francisco.

    # We're still not done - there could be flights through lots of other cities that could be better.

    # We have to look exhaustively at all the options!

    # If we fill in all the flights between these cities, we've built up this graph:

When we represent these kinds of problems, we're not going to explicitly represent them as this graph, because the graph doesn't show the history - i.e., if I'm flying from Austin to Phoenix, the graph doesn't show whether I came through Nashville or Miami or Denver.

Instead, we can represent this as a **search tree**, where the root is Boston, and its children are all the places I can go from Boston: Chicago, New York, and Miami.

    # Similarly, from Chicago we can go to San Francisco or Denver, so the children of the Chicago node are San Francisco and Denver. San Francisco is our goal, so it's a **leaf node** - we'll never go somewhere after San Francisco.

    # We can fill in the rest of the search tree the same way:

Now we have a complete tree with all of the paths I can take from Boston to San Francisco, which shows not only where I end up but what route I took to get there.

Nothing about this tree tells us what's the *best* path to take, though.

    # There could be many different ways of determining what's the best route:

        - Minimum number of miles in the air (which would give us the Boston $\rightarrow$ Chicago $\rightarrow$ San Francisco route)
        - Cost (might be the Boston $\rightarrow$ Miami $\rightarrow$ Nashville $\rightarrow$ Phoenix route)
        - Total time
        - Any other metric we want - which airports have the best food, etc!
        - Each metric might give us a different route that we'd see as being the "best".

This entire class of problems, where we'll build a search tree and compare all the paths to see how well they fulfill some criterion, we'll call **search problems**.

    # We have lots of algorithms for exploring these trees, some of which we've already seen:

        # **Depth first search**, where we go all the way down one path and then go back up to the top and start down another path

        # **Breadth first search**, where we look at all the top-level nodes, then all the nodes one level down, then all the nodes two levels down, and so on

    # These search trees are fundamental to AI, but they don't always get everything perfectly right.

2.1. Adversarial Search

- Sometimes we're building one of these search trees, but we don't actually get to make all the decisions about which way to go.
- This is how game-playing systems work!
- Imagine if I get to choose where I fly from Boston, but then someone else gets to pick where I go next from there.

    # We'll need a slightly different approach - we can't just search through the tree anymore, because we're not in control of all the decisions.

- Let's think about Tic-Tac-Toe. On the first move, if I'm playing X, I have nine options of where to play.

    # If I pick to put the X in the center, then the other player - O - gets to choose which of the eight options for where to go next.

    # Then once O picks a move, I get to choose a place to put my X next… and so on.

    # We can build a tree this way, continuing until either someone wins or the board is completely full (these would be our leaf nodes).

    # If this were a regular search problem, we could just say X goes here, O goes way over there, X goes here, O goes way over there, X goes here, look, I won, and the game would be over in five moves - but I don't get to choose where my opponent plays.

- We need a new strategy that takes this into account, and a common one to use is **minimax strategy**: choosing the move which leaves our opponent with the worst set of possibilities.
- Here's how that might play out in pseudocode:

```
Generate the entire game tree
Evaluate each terminal node (high values are good for your side)
Filter values from the terminal nodes up through the tree:
    At nodes controlled by your opponent, choose the minimum value of the
 children
    At nodes controlled by you, choose the maximum value of the children
When you reach the top of the tree, you have an optimal solution
```

  # First, we make the whole game tree like we did in the regular search problem, and determine the values of all the leaf nodes (for example, a win for me in Tic-Tac-Toe would score high, while a win for my opponent would score low, and maybe a draw would be in the middle).
  # We then go up the tree from the bottom, picking the highest scoring option for turns that we control and assuming that our opponent will make the best move available to them (i.e., the lowest scoring) on turns that they control.
  # By the time we get to the top, we have a path that we know is optimal.
- Let's consider a game somewhat more complex than Tic-Tac-Toe, where there are lots of different scores we can end up with. Let's say that we'll start, then our opponent will take a turn, then we get another turn.
  # Following our minimax algorithm, we start from the various terminal states of the game. We have the last turn, so we can pick the maximum among our options at each node.
  # On our opponent's turn, though, we have to assume that he'll give us the worst possible option, so we put the minimum option at that node.
  # Then at the level above that, we get to choose again, so we again take the maximum.
  # When we're done - back at the top node - our tree looks like this:

  # If we got to choose every step, we could just take the path that leads to the score of 11! But we don't get to make that choice, because our opponent can force us

down worse paths on his turn (in particular, if we start down that path, he can force us to get only three points). The best we can guarantee is the path that leads us down to five points.

- Let's look at how this works for Tic-Tac-Toe in `tic-tac-toe.c`[1], where you can play against a computer, defining Player 1 as the computer, playing O, and Player -1 as the human, playing X.

  # We have a whole bunch of helper functions to clear the screen, draw the board, check for input, check if there's a winner, and so on, but the one we care about here is called `minimax`:

[1] http://cdn.cs50.net/2015/fall/lectures/11/m/src11m/tic-tac-toe.c.src

```
int minimax(int hypothetical_board[DIM_MAX][DIM_MAX], int player, bool
 mymove, int depth)
{
    int i, j, score;

    // if we have gone too deep, return;
    if (depth > MAX_DEPTH)
    return 0;

    // see if someone has won
    int winner = check_for_winner(hypothetical_board);
    if (winner != 0)
    {
        return winner;
    }

    int move_row = -1;
    int move_col = -1;
    if (mymove)
        score = -2; //Losing moves are preferred to no move
    else
        score = 2;

    // For all possible locations (moves),
    for (i = 0; i < d; i++)
    {
        for (j = 0; j < d; j++)
        {
            if (hypothetical_board[i][j] == 0)
            {
                // If this is a legal move,
                hypothetical_board[i][j] = player; //Try the move
                int thisScore = minimax(hypothetical_board,
 -1*player, !mymove, depth+1);

                if (mymove)
                {
                    // my move, so maximize the score
                    if (thisScore > score) {
                        score = thisScore;
                        move_row = i;
                        move_col = j;
                    }
                }
                else
                {
                    // not my move, so minimize the score
```

    # This is exactly the algorithm that we saw before in pseudocode, but we're keeping track of an additional parameter called `depth` that stores how far down the tree we've gotten so we can stop traversing the tree when we get too deep.

    # For each move I can make, the function constructs what the board would look like with that move on the board, and then recursively calls `minimax` itself with the new configuration.

    # If it's my move, we find the option that gives the maximum score; if it's my opponent's move, we find the option that gives the minimum score.

    # Everything else is just bookkeeping.

    # Volunteers Gourav and Layla come up to demonstrate how this actually runs.

        # Gourav plays 3x3 Tic-Tac-Toe against the computer and the game ends in a tie.

        # Layla then plays 4x4 Tic-Tac-Toe against the computer and wins.

        # We can see that the computer player makes "smart" moves, but because there's a maximum depth setting on how deep down the tree it looks, it doesn't always make the optimal move (which is how Layla was able to win).

- These systems that just brute-force check all the possible paths deeper and deeper into a minimax tree perform quite well on standard board games like Tic-Tac-Toe.
- We can easily enumerate all the possibilities for Tic-Tac-Toe, but what about more elaborate games? What if that entire brute-force tree is just too big to compute and search through?

    # We have to do something smarter. We can try going less deep in the tree, but then we have to evaluate incomplete final states. We need a good **evaluation function** to specify which boards are better, even if nobody has yet won or lost.

        # For example, we could count in chess how many pieces are left, possibly with weighting for different types of pieces.

        # We can also take into account board position, etc.

    # It also becomes really important not to just search straight down to the depth limit - instead, we use the **alpha-beta principle**, that can basically be summed up as "If you have an idea that is bad, don't waste time seeing how truly awful it is."

    # We'll keep track of the best and worst possible outcomes in each part of the tree, and if any part of the tree has a really bad outcome that looks likely, we'll abandon that whole part of the tree.

  # We can go through the tree replacing the last level where we make the decision with a certainty, because we have complete control over that last decision and we'll pick the maximum - let's say it has a value of 8.

  # Then at the minimization level (our opponent's turn) directly above that, we know we can never get more than 8 points in this branch of the tree (if there were an option worth more than 8 points, our opponent could still pick this option and keep us down to just 8 points).

    # If we check the next path at the bottom level and see that we could get 9 points going that way, we can ignore that path and not look any more in that direction, because we know that our opponent will never let us go this way.

    # If we find on another path on this branch that the most we can get is 4, now we know our opponent will pick the 4 path for us over the 8 path, so the most we can get down this branch is 4.

  # The whole point of alpha-beta is to cut off parts of the tree and avoid having to look at them.

  # If we continue using these principles, we end up with an equivalent tree - where the best path is the one that leads to 5 points - but instead of looking at all 27 terminal states, we only have to look at 16 of the terminal states.

  # The tree grows exponentially downward, so even this simple optimization cutting off parts of the tree can lead to huge savings.

- We can compare the complexity of different games:

    # Improvements in chess-playing computational systems, for example, have been partly in increased processing power, but even more so in better evaluation functions and better pruning methods.

    # There are still games beyond the reach of computational systems - for example, Go, where the best computational opponents are at roughly the level of a skilled amateur player and certainly can't beat any professional Go player.

3. Beyond Board Games

- Traditional board games have certain constraints - we have perfect information about where the pieces are, the environment is static while we're taking our turn, the action space is discrete (only specific moves are allowed), and all the actions are deterministic (we know that attempting to make a certain move will actually cause it to happen as we intend, as long as it's a valid move).
- To move beyond board games, we have to break some of these assumptions.
- Even classic video games have non-static environments (the Pacman ghosts are moving, e.g.), although you usually still have perfect information.
    # Google recently trained a computational algorithm to play Atari 2600 games that watched only the screen inputs and would figure out how well it was doing based on its score (even with no idea of the rules of the game), a method called **reinforcement learning**.
        # For many of these games, this self-trained program with no input about the rules beat the top professional gamers.
- More often, though, when we think about computer players, we're thinking about the AI opponents we can actually play against in games like Starcraft or Civilization.
    # This category of computer games, which we'll call 4X for "eXplore, eXpand, eXploit, and eXterminate", give the computer player limited information and a dynamic environment, although still a discrete action space and deterministic actions.
- First-person shooters have computer opponents with limited information, a dynamic environment, a more continuous action space, and sometimes stochastic actions (you can try to jump over the wall, but you have a chance of failing).
- In robotics, we have to be able to go even further and deal with limited information, a dynamic environment, a completely continuous action space, and completely stochastic actions (i.e., all actions have a chance of failing).
    # First-person video games essentially run mini robotics architectures!
- Robotics and game-playing architectures are generally very similar.
- Let's go back to our Tic-Tac-Toe example. We have a robot here called Baxter, designed for small-scale manufacturing, but we're going to use it to play Tic-Tac-Toe.

Unlike typical manufacturing systems, Baxter is designed to be much safer - rather than responding to position commands without regards for whether other things are in the way, Baxter has a spring in each joint that we can control the amount of force on that spring rather than an exact position.

Baxter plays Tic-Tac-Toe based on the same kind of minimax algorithm that we encountered before. Playing against volunteer Luis, we can see that Baxter successfully blocks potential winning plays and forces a tie (which Baxter is then able to recognize).

Baxter uses a camera to recognize whether a blue X token or a red O token is at each point on the board, and uses a vacuum gripper to pick up its tokens, but otherwise it's using the same underlying logic.

At the very end, we see Baxter cheat, placing its token on top of its opponent's token.

    # One of the fantastic things about AI is that in addition to letting us build really interesting and intelligent devices, it tells us something about how humans are intelligent.

    # Some current research in Scaz's lab focuses on what happens when robots unexpectedly cheat.

        - Instead of using Baxter & Tic-Tac-Toe, they used a smaller robot called Now that played Rock-Paper-Scissors, and after many normal games, would change its gesture from a losing one to a winning one after seeing the player's gesture.
        - As a control, they'd also have the robot throw some of the games - change a winning gesture to a losing one.
        - People react to a robot cheating in order to win as if it's deliberately malicious and has an intent to hurt them, but to a robot cheating in order to lose as just a malfunctioning device.
        - This teaches us a lot about how humans think!