CS61B Lecture 22

Friday, March 13, 2020

Announcements

- Lecture right now is still being recorded; we might move to Zoom in the future to allow people to ask questions.
- Homework deadlines are being moved to allow people to resettle.
- Project 2 is out and this lecture contains useful information before you get started.

Searching by "Generate and Test"

This is a course on data structures, and sometimes data structures is just one of a number of different ways to representing a particular situation. Suppose we are searching through a set that cannot or we do not want to store in memory. We want to know if some value x is in that set?

If we know how to enumerate all the possible candidates, we can use the approach of **generate and test**: test all possibilities by generating them on the fly and then checking them. It can be a very powerful technique in some senses, and it can also be a very weak technique. We can be a little more clever, such as avoid trying things that we know we won't work.

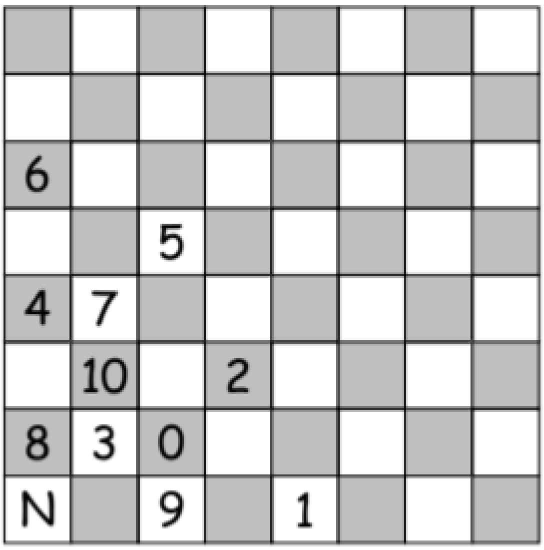
What happens if the set of possible candidates is infinite?

Backtracking Search

The idea is to try and be a little more clever. Let's systematically search through some set of things. An example is the Knight's Tour: find all the paths a knight can travel on a chess-board such that it touches every square exactly once and ends up one knight move from where it started.

In the example below, the numbers indicate position numbers, starting at 0. Knight N is stuck; how can we handle this?

The idea is that we can back up and try again, hence the name backtracking search.



Let's write some code! Here is the general recursive algorithm:

```
/** Append to PATH a sequence of knight moves starting at ROW, COL
 *  that avoids all squares that have been hit already and
 *  that ends up one square away from ENDROW, ENDCOL. B[i][j] is
 *  true iff row i and column j have been hit on PATH so far.
 *  Returns true if it succeeds, else false (with no change to PATH).
 *  Call initially with PATH containing the starting square, and
 *  the starting square (only) marked in B. */
boolean findPath(boolean[][] b, int row, int col,
int endRow, int endCol, List path) {
    if (path.size() == 64) { return isKnightMove(row, col, endRow, endCol); }
    for (r, c = all possible moves from (row, col)) {
        if (!b[r][c]) {
            b[r][c] = true; // Mark the square
            path.add(new Move(r, c));
            if (findPath(b, r, c, endRow, endCol, path)) {
                return true; // If our move is valid, then we move on
            }
            b[r][c] = false; // Backtrack out of the move.
            path.remove(path.size()-1);
        }
    }
    return false;
}
```

Here, `b` represents the board and it tells us whether we have visited a square yet. `row` and `col` represent the starting point, and `endRow` and `endCol` represent our end goal. `path` is a list of all the paths we have taken. `path` is considered an output parameter, while the others are all input parameters.

This is a similar problem to finding the way out of a maze, as we did before.

Best Move Search

Suppose we are not necessarily interested in just finding a solution, but the best move. How do we do that?

Consider what it means for a move to be good. Well, it probably means the one that leads us to a win. Now the new problem is how do we know if it leads to a win? We must make a guess.

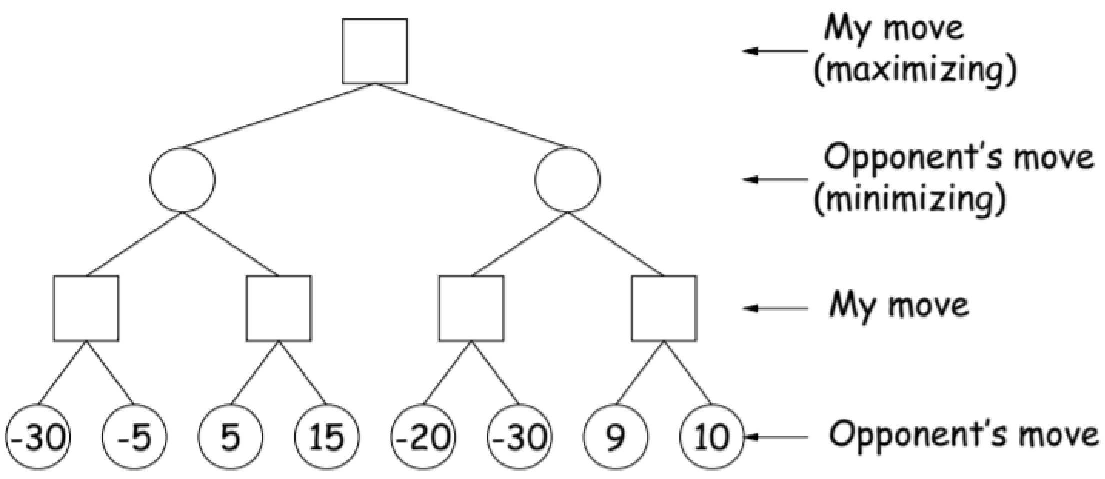
Making the guess can be based on some heuristic value. For example, in chess, that would be to assign points to each piece you have, or in checkers, to compare the number of black and white pieces. You could also assign some value to the nearnes of pieces to strategic areas (center of board).

All of these are just heuristics -- a move might give us more pieces, but set up a devastating response from the opponent. Thus, for each move, we should also look at the opponent's possible movies, assume he picks the best move for him, and use that as the value.

But what if **you** have a great response to his response? We are essentially looking ahead into the future -- what will the board look like several plays from now? More importantly, this leads to the question of how we can organize this sensibly.

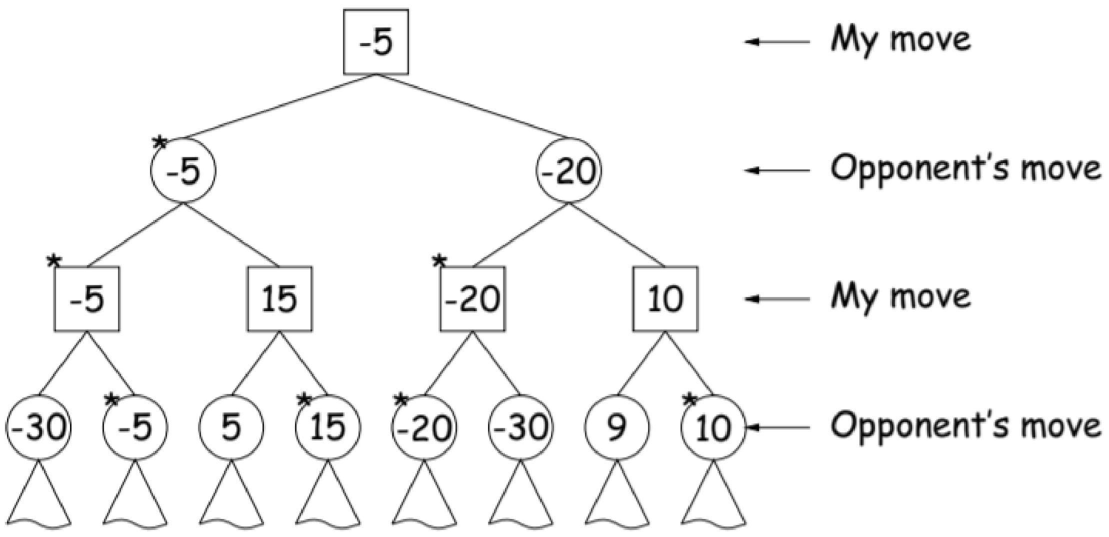
Game Trees

Think of the space of possible continuations of the game as a tree, where each node is a position, and each edge a move. This tree doesn't necessarily have to be generated in memory, but just as a concept in your head.



Suppose the numbers at the bottom are the values of those final positions to me. Smaller numbers are of more value to my opponent. What should I do? What value can I get if my opponent plays as well as possible?

If we keep repeating this procss, we end up with something like this:



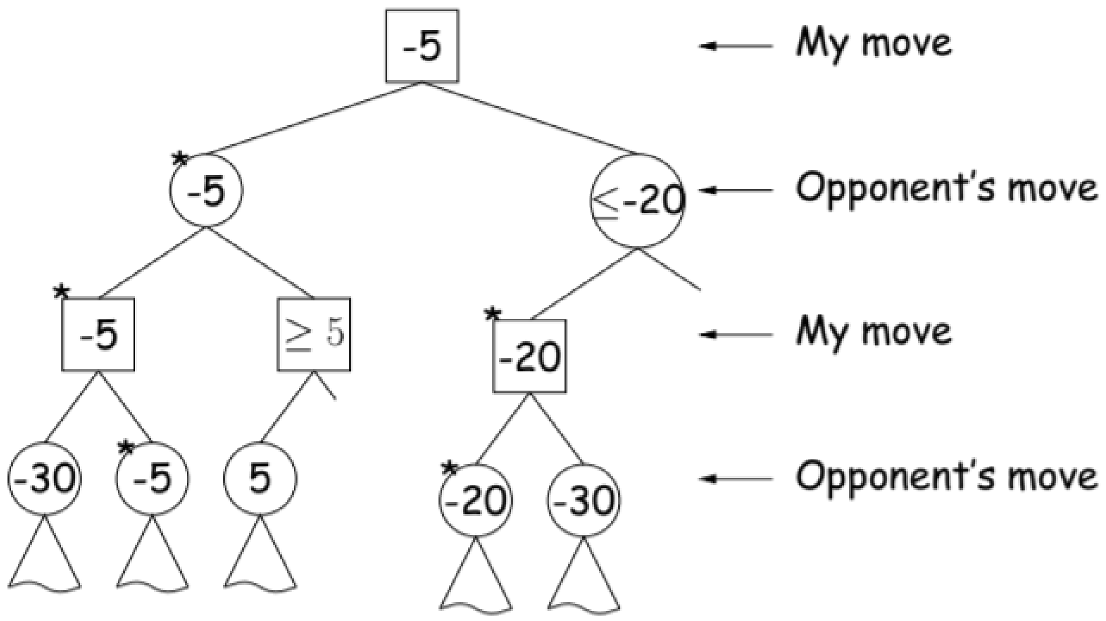
The numbers are the values we guess for positions, and the starred nodes would be chosen.

I will always choose the child with the maximum value as my next move, and the opponent always chooses the minimum. This is called the **minimax algorithm**.

How far do we go down? Well, if we took all the possibilities in chess, there are too many possibilities to even consider! It is interesting to see how DeepBlue and other programs handle this problem.

Alpha-Beta Pruning

We can **prune** this tree as we search it, deleting nodes we know we will not need to save space.



At the '≥ 5' position, I know that the opponent will not choose to move here (since he already has a −5 move). And at the '≤ −20' position, my opponent knows that I will never choose to move here (since I already have a −5 move).

In this small example, we don't save a lot, but you can imagine the savings growing exponentially as we move to larger and larger trees. We are reducing the so-called branching factor.

Cutting Off the Search

How do we know when to stop? If we could traverse the game tree to the bottom, then we would be able to force a win (if possible), and sometimes this is possible near the end of a game. However, usually, game trees tend to either be infinite or impossibly large.

Thus, we choose a maximum depth and use a heuristic value computed on the position alone, called a static valuation, as the value at that depth. Or, we might use iterative deepening, repeating the search at increasing depths until time is up. It's not necessarily recommended for Project 2, but you could try it.

Much more sophisticated searches are possible, however: to find out how, take CS188, which teaches artificial intelligence.

Overall Search Algorithm

The idea is to exhaustively search down to a particular limit, and as we go, we will pass down limits to the values we need.

Also pass α and β limits:

- High player does not care about exploring a position further once he knows its value is larger than what the minimizing player knows he can get (β), because the minimizing player will never allow that position to come about.
- Likewise, minimizing player won't explore a positions whose value is less than what the maximizing player knows he can get (α)

To start, a maximizing player will find a move with `findMax(current position, search depth −∞, + ∞ )` and the minimizing player will perform `findMin(current position, search depth −∞, + ∞ )`

Pseudocode for Search

Below is some pseudocode that describes our algorithm.

One-Level Search

```
Move simpleFindMax(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon()) {
        return artificial “Move” with value +∞;
    } else if (posn.minPlayerWon()) {
        return artificial “Move” with value −∞;
    }
    Move bestSoFar = artificial “Move” with value −∞;
    for (each M = a legal move for maximizing player from posn) {
        Position next = posn.makeMove(M);
        next.setValue(heuristicEstimate(next));
        if (next.value() >= bestSoFar.value()) {
            bestSoFar = next;
            alpha = max(alpha, next.value());
            if (beta <= alpha) { break; }
            }
    }
    return bestSoFar;
}
```

```
Move simpleFindMin(Position posn, double alpha, double beta) {
    if (posn.maxPlayerWon()) {
        return artificial “Move” with value +∞;
    } else if (posn.minPlayerWon()) {
        return artificial “Move” with value −∞;
    }
    Move bestSoFar = artificial “Move” with value +∞;
    for (each M = a legal move for minimizing player from posn) {
        Position next = posn.makeMove(M);
        next.setValue(heuristicEstimate(next));
        if (next.value() <= bestSoFar.value()) {
            bestSoFar = next;
            beta = min(beta, next.value());
            if (beta <= alpha) { break; }
        }
    }
    return bestSoFar;
}
```

Depth Search

```
/** Return a best move for maximizing player from POSN, searching
 *  to depth DEPTH. Any move with value >= BETA is also
 *  "good enough". */
Move findMax(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn)) { return simpleFindMax(posn, alpha, beta);
}
    Move bestSoFar = artificial “Move” with value −∞;
    for (each M = a legal move for maximizing player from posn) {
        Position next = posn.makeMove(M);
        Move response = findMin(next, depth-1, alpha, beta);
        if (response.value() >= bestSoFar.value()) {
            bestSoFar = next;
            next.setValue(response.value());
            alpha = max(alpha, response.value());
            if (beta <= alpha) { break; }
        }
    }
    return bestSoFar;
}
```

```
/** Return a best move for minimizing player from POSN, searching
 *  to depth DEPTH. Any move with value <= ALPHA is also
 *  "good enough". */
Move findMin(Position posn, int depth, double alpha, double beta) {
    if (depth == 0 || gameOver(posn)) { return simpleFindMin(posn, alpha, beta);
}
    Move bestSoFar = artificial “Move” with value +∞;
    for (each M = a legal move for minimizing player from posn) {
        Position next = posn.makeMove(M);
        Move response = findMax(next, depth-1, alpha, beta);
        if (response.value() <= bestSoFar.value()) {
            bestSoFar = next;
            next.setValue(response.value());
            beta = min(beta, response.value());
            if (beta <= alpha) { break; }
            }
        }
    return bestSoFar;
}
```

Notice we don't create any actual trees in the code, but we are doing what is called a tree-motivated search: we follow all the same procedures as we would if we created an actual tree.