

## Searching by "Generate and Test"

Considering the problem of searching a set of data stored in some data structure: "Is  $x \in S$ ?"

We don't have a set  $S$ , but know how to recognize what we find it: "Is there an  $x$  such that  $P(x)$ ?"

How to enumerate all possible candidates, can use *generate and test*: test all possibilities in turn.

Can be more clever: avoid trying things that won't work, e.g. by pruning.

What if the set of possible candidates is infinite?

## General Recursive Algorithm

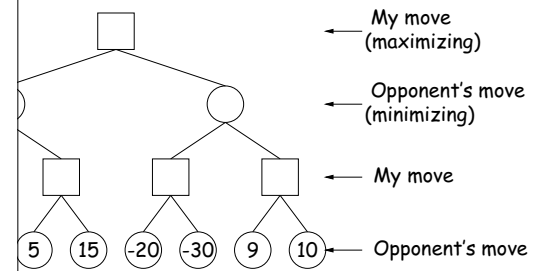
**isKnightMove** a sequence of knight moves starting at ROW, COL is all squares that have been hit already and up one square away from ENDROW, ENDCOL. B[i][j] is true if row i and column j have been hit on PATH so far. Return true if it succeeds, else false (with no change to PATH). Called initially with PATH containing the starting square, and ENDROW, ENDCOL the ending square (only) marked in B. \*/

```

boolean isKnightMove(boolean[][] b, int row, int col,
                    int endRow, int endCol, List path) {
    if (row == endRow && col == endCol) return true;
    for (int[] move : allPossibleMovesFrom(row, col)) {
        int r = row + move[0], c = col + move[1];
        if (r < 0 || r > 7 || c < 0 || c > 7 || b[r][c]) continue;
        path.add(new Move(r, c));
        if (isKnightMove(b, r, c, endRow, endCol, path)) return true;
        path.remove(path.size() - 1);
    }
    return false;
}
    
```

## Game Trees

View the space of possible continuations of the game as a tree. At each position, each edge is a move.



The numbers at the bottom are the values of those final positions. Smaller numbers are of more value to *my opponent*.

I move? What value can I get if my opponent plays as optimally?

## CS61B Lecture #22

Backtracking searches, game trees (DSIJ, Section 6.5)

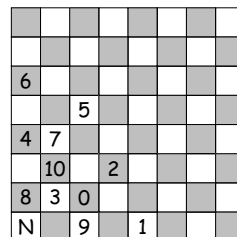
## Backtracking Search

Backtracking search is one way to enumerate all possibilities.

**Knight's Tour**. Find all paths a knight can travel on a chessboard that touches every square exactly once and ends up where it started.

Below, the numbers indicate position numbers (knight starts at N)

(N) is stuck; how to handle this?



## Other Kind of Search: Best Move

Problem of finding the *best* move in a two-person game.

Assign a *heuristic value* to each possible move and pick the best one (using *static evaluation*). Examples:

- Number of black pieces – number of white pieces in checkers.
- Sum of white piece values – weighted sum of black piece values in chess (Queen=9, Rook=5, etc.)
- Number of pieces to strategic areas (center of board).

Be careful of misleading. A move might give us more pieces, but set up a trap for a response from the opponent.

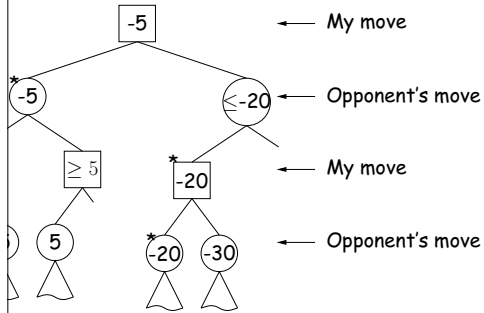
When you move, look at *opponent's* possible moves, assume he will pick the best one for him, and use that as the value.

How do you respond to his response?

Organize this sensibly?

## Alpha-Beta Pruning

See this tree as we search it.



In this position, I know that the opponent will not choose to move to the right (since he already has a  $-5$  move).

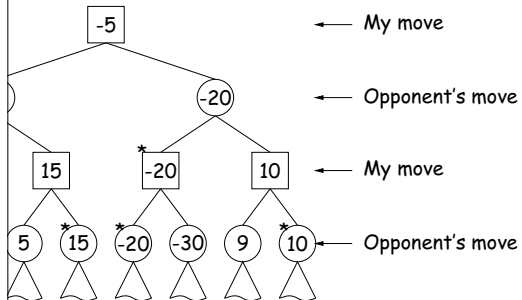
In this position, my opponent knows that I will never choose to move to the right (since I already have a  $-5$  move).

10:22 2018

CS61B: Lecture #22 8

## Game Trees, Minimax

The search space of possible continuations of the game as a tree. At each position, each edge a move.



At each position, we guess the values we guess for the positions (larger means better). Starred nodes would be chosen.

At each position, choose child (next position) with maximum value; opponent chooses minimum value ("Minimax algorithm")

10:22 2018

CS61B: Lecture #22 7

## Overall Search Algorithm

At each position, choose whose move it is (maximizing player or minimizing player), and for a move estimated to be optimal in one direction or the other.

At each position, be exhaustive down to a particular depth in the game tree. At each position, we guess values.

At each position, and  $\beta$  limits:

At each position, the maximizing player does not care about exploring a position further once its value is larger than what the minimizing player knows he can get ( $\beta$ ), because the minimizing player will never allow that to come about.

At each position, the minimizing player won't explore a position whose value is smaller than what the maximizing player knows he can get ( $\alpha$ ).

At each position, the maximizing player will find a move with value  $\alpha$ .

At each position,  $\text{current position, search depth } -\infty, +\infty$

At each position, player:

At each position,  $\text{current position, search depth } -\infty, +\infty$

10:22 2018

CS61B: Lecture #22 10

## Cutting off the Search

At each position, to traverse game tree to the bottom, you'd be able to do so if it's possible.

At each position, possible near the end of a game.

At each position, typically, game trees tend to be either infinite or impossibly large.

At each position, use a maximum *depth*, and use a heuristic value computed on its own (called a *static valuation*) as the value at that position.

At each position, use *iterative deepening*, repeating the search at increasing depths until time is up.

At each position, sophisticated searches are possible, however (take CS188).

10:22 2018

CS61B: Lecture #22 9

## One-Level Search for Minimizing Player

```

double MinMaxMin(Position posn, double alpha, double beta) {
    if (posn.isMaxPlayerWon())
        return posn.getMaxPlayerWon();
    if (posn.isMinPlayerWon())
        return posn.getMinPlayerWon();
    double bestSoFar = -infinity;
    for (Move M : posn.getLegalMoves()) {
        if (M.isArtificialMove())
            continue;
        Position next = posn.makeMove(M);
        double value = MinMaxMax(next, alpha, beta);
        bestSoFar = max(bestSoFar, value);
        if (beta <= alpha)
            break;
    }
    return bestSoFar;
}
    
```

10:22 2018

CS61B: Lecture #22 12

## Pseudocode for Searching (One Level)

A basic kind of game-tree search is to assign some heuristic value to each position, looking at just the next possible move:

```

double MinMaxMax(Position posn, double alpha, double beta) {
    if (posn.isMaxPlayerWon())
        return posn.getMaxPlayerWon();
    if (posn.isMinPlayerWon())
        return posn.getMinPlayerWon();
    double bestSoFar = -infinity;
    for (Move M : posn.getLegalMoves()) {
        if (M.isArtificialMove())
            continue;
        Position next = posn.makeMove(M);
        double value = MinMaxMin(next, alpha, beta);
        bestSoFar = max(bestSoFar, value);
        if (beta <= alpha)
            break;
    }
    return bestSoFar;
}
    
```

At each position,  $\text{current position, search depth } -\infty, +\infty$

10:22 2018

CS61B: Lecture #22 11

### Code for Searching (Minimizing Player)

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

### Code for Searching (Maximizing Player)

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