Backtracking and Games

Eric Roberts
CS 106B
January 26, 2015

Searching in a Branching Structure

- The recursive structure for finding the solution path in a maze comes up in a wide variety of applications, characterized by the need to explore a range of possibilities at each of a series of choice points.
- The primary advantage of using recursion in these problems is that doing so dramatically simplifies the bookkeeping. Each level of the recursive algorithm considers one choice point. The historical knowledge of what choices have already been tested and which ones remain for further exploration is maintained automatically in the execution stack.
- Many such applications are like the maze-solving algorithm in which the process *searches* a branching structure to find a particular solution. Others, however, use the same basic strategy to explore *every* path in a branching structure in some systematic way.

Exercise: Generating Subsets

- Write a function

```
Vector<string> generateSubsets(string set);
```

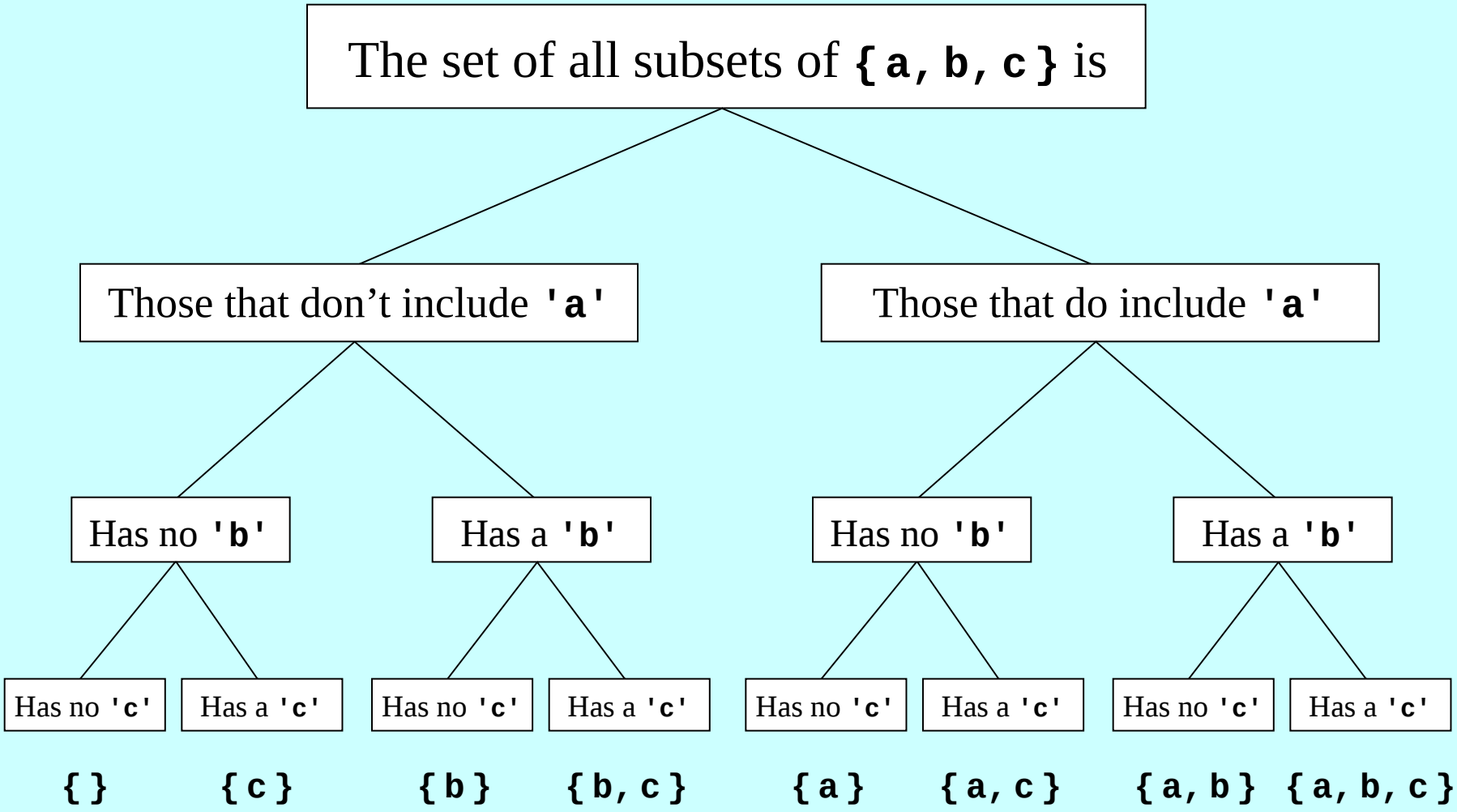
  that generates a vector showing all subsets of the set formed from the letters in **`set`**.

- For example, calling **`generateSubsets("abc")`** should return a vector containing the following eight strings, in some order:

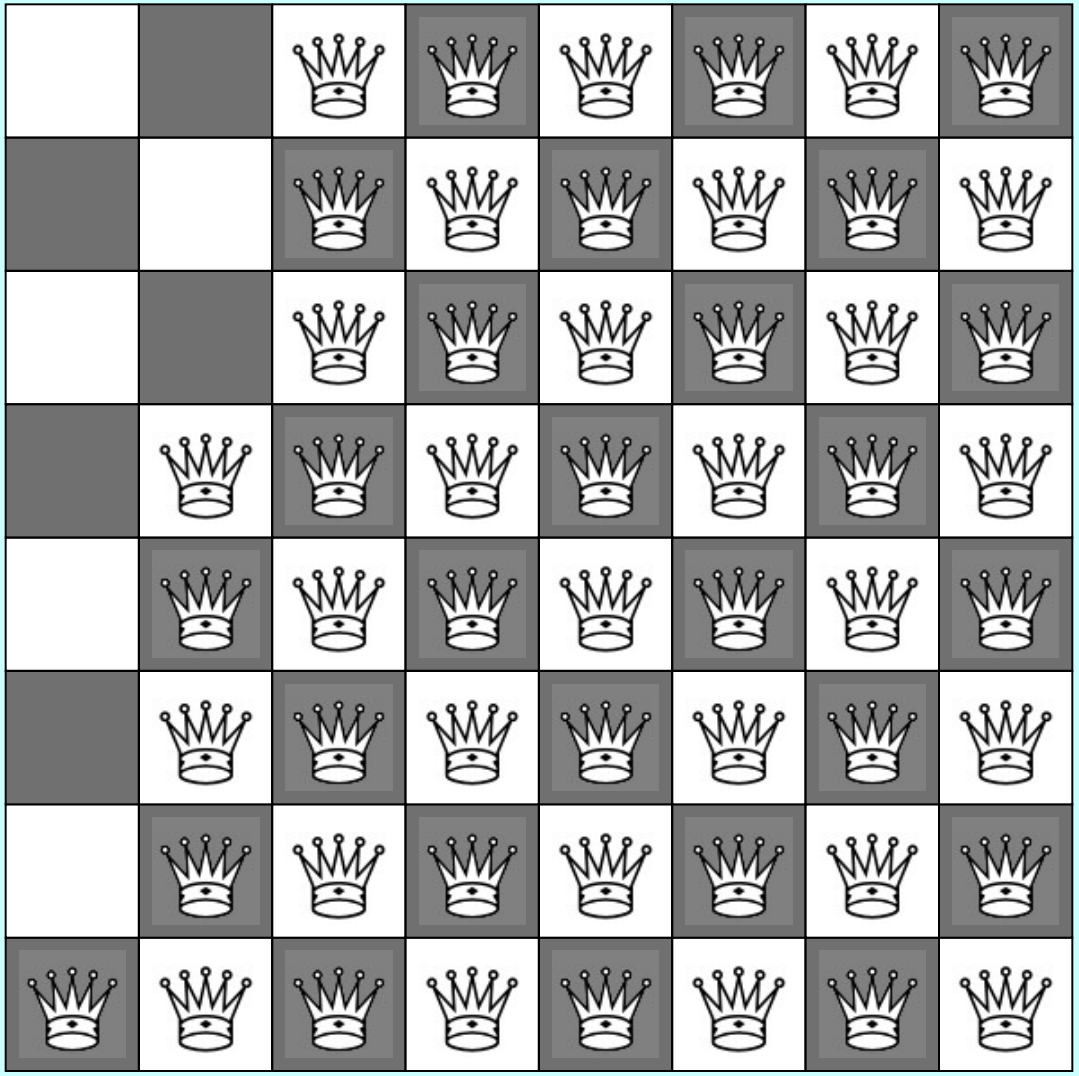
```
""      "a"     "ab"    "abc"
        "b"     "ac"
        "c"     "bc"
```

- The solution process requires a branching structure similar to that used to solve a maze. At each level of the recursion, you can either exclude or include the current letter from the list of subsets, as illustrated on the following slide.

The Subset Tree

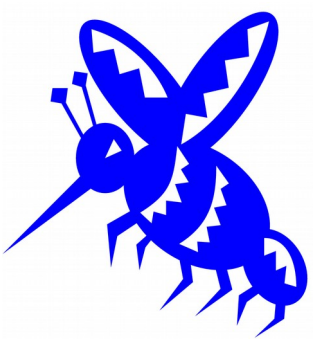


The Eight Queens Problem



What Most Students Want to Do

```
bool solveQueens(Grid<char> & board) {
   int n = board.numRows();
   for (int c1 = 0; c1 < n; c1++) {
      board[0][c1] = 'Q';
      for (int c2 = 0; c2 < n; c2++) {
         board[1][c2] = 'Q';
         for (int c3 = 0; c3 < n; c3++) {
            board[2][c2] = 'Q';

            . . .

                  if (boardIsLegal(board)) return true;

            . . .

            board[2][c2] = ' ';
         }
         board[1][c2] = ' ';
      }
      board[0][c2] = ' ';
   }
   return false;
}
```



Exercise: Find a Recursive Solution

- Assuming that you have the function **`boardIsLegal`** from the previous slide, how would you write a *recursive* solution that works with the following main program:

```
int main() {
   int n = getInteger("Enter size of board: ");
   Grid<char> board(n, n);
   for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
         board[i][j] = ' ';
      }
   }
   if (solveQueens(board)) {
      displayBoard(board);
   } else {
      cout << "There is no solution for this board" << endl;
   }
   return 0;
}
```

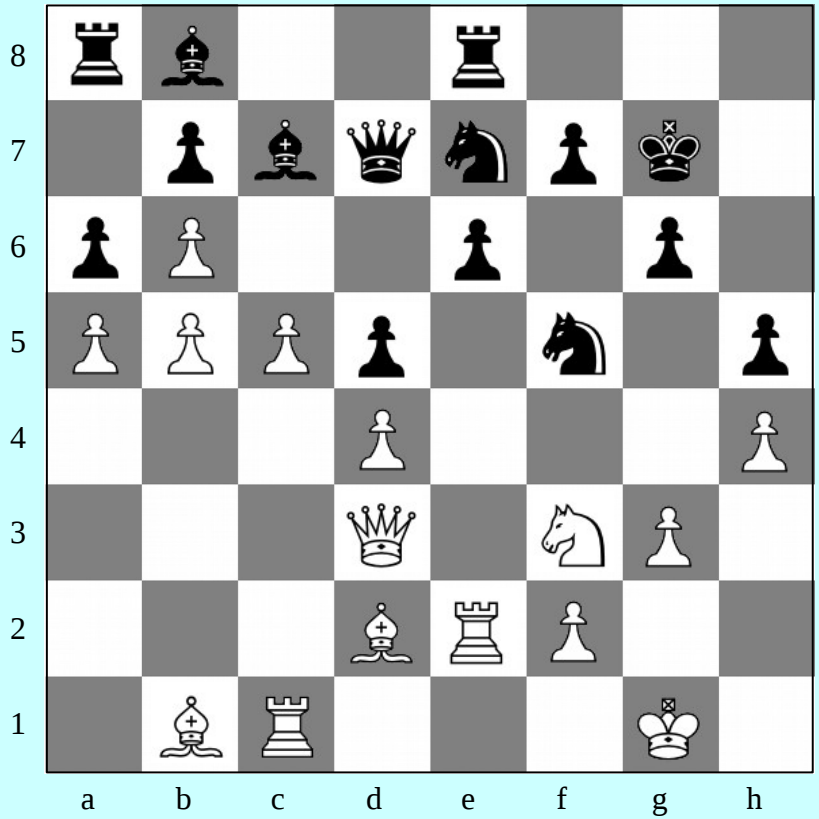
- What aspect of NQueens can you use to have the recursion move toward simpler instances of some common subproblem?

Deep Blue Beats Gary Kasparov

In 1997, IBM's Deep Blue program beat Gary Kasparov, who was then the world's human champion. In 1996, Kasparov had won in play that is in some ways more instructive.

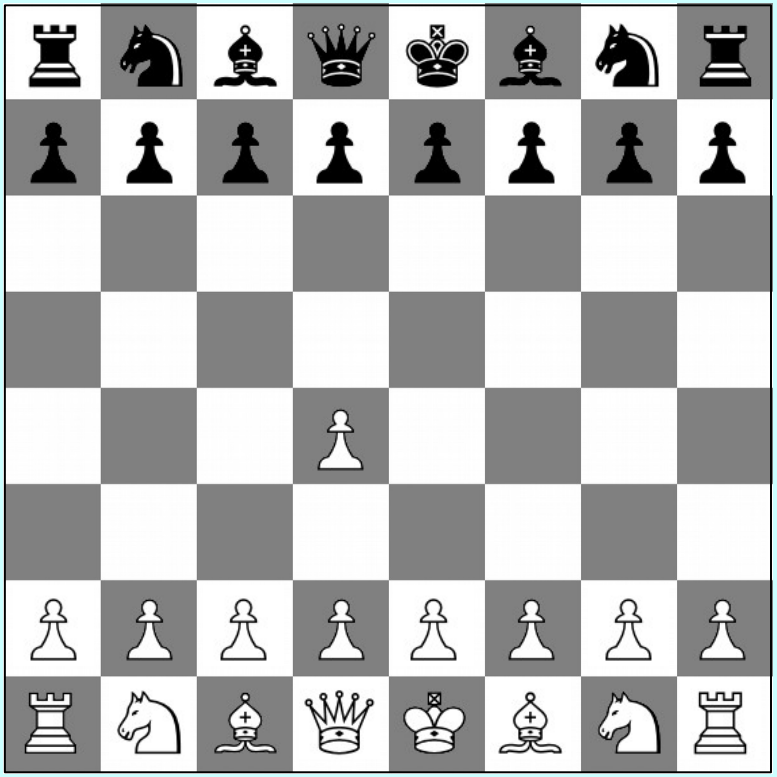
Game 6

Kasparov	Deep Blue
30. b6	30. Bb8 ??



Recursion and Games

- In 1950, Claude Shannon wrote an article for *Scientific American* in which he described how to write a chess-playing computer program.
- Shannon's strategy was to have the computer try every possible move for white, followed by all of black's responses, and then all of white's responses to those moves, and so on.
- Even with modern computers, it is impossible to use this strategy for an entire game, because there are too many possibilities.

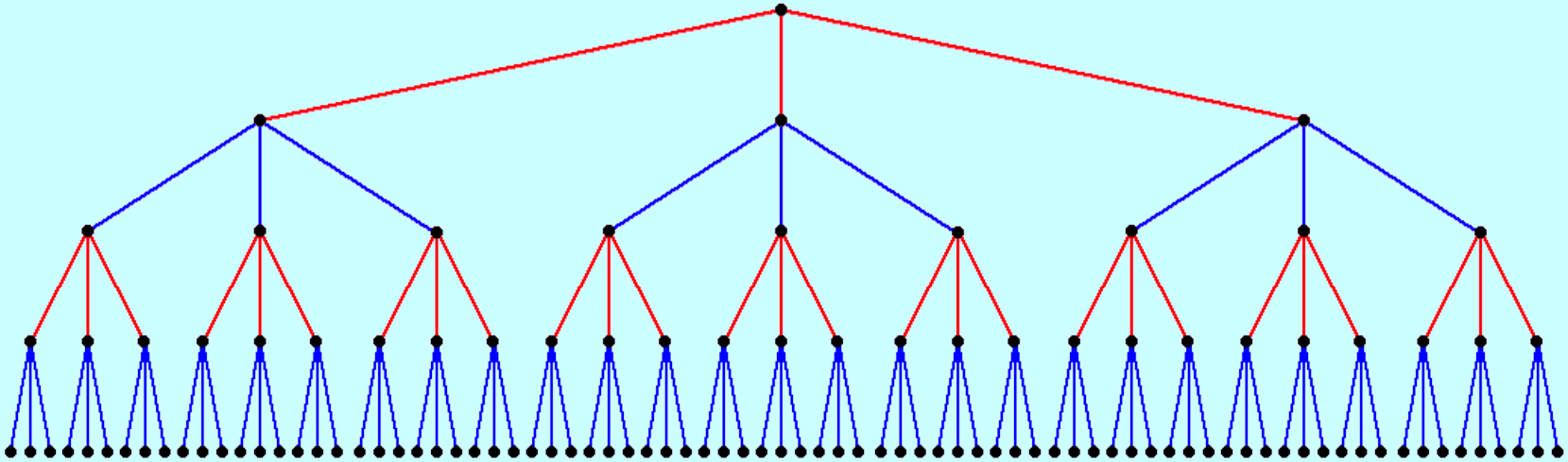


Positions evaluated: $\sim 10^{53}$

. . . millions of years later . . .

Game Trees

- As Shannon observed in 1950, most two-player games have the same basic form:
  - The first player (red) must choose between a set of moves
  - For each move, the second player (blue) has several responses.
  - For each of these responses, red has further choices.
  - For each of these new responses, blue makes another decision.
  - And so on . . .



A Simpler Game

- Chess is far too complex a game to serve as a useful example. The text uses a much simpler game called ***Nim***, which is representative of a large class of two-player games.
- In Nim, the game begins with a pile of coins between two players. The starting number of coins can vary and should therefore be easy to change in the program.
- In alternating turns, each player takes one, two, or three coins from the pile in the center.
- The player who takes the last coin loses.

A Sample Game of Nim

Nim

```
There are 11 coins left.
I'll take 2.
There are 9 coins left.
Your move: 1
There are 8 coins left.
I'll take 3.
There are 5 coins left.
Your move: 2
There are 3 coins left.
I'll take 2.
There is only one coin left.
You lose.
```

Good Moves and Bad Positions

- The essential insight behind the Nim program is bound up in the following mutually recursive definitions:
  - A ***good move*** is one that leaves your opponent in a bad position.
  - A ***bad position*** is one that offers no good moves.
- The implementation of the Nim game is really nothing more than a translation of these definitions into code.

Coding the Nim Strategy

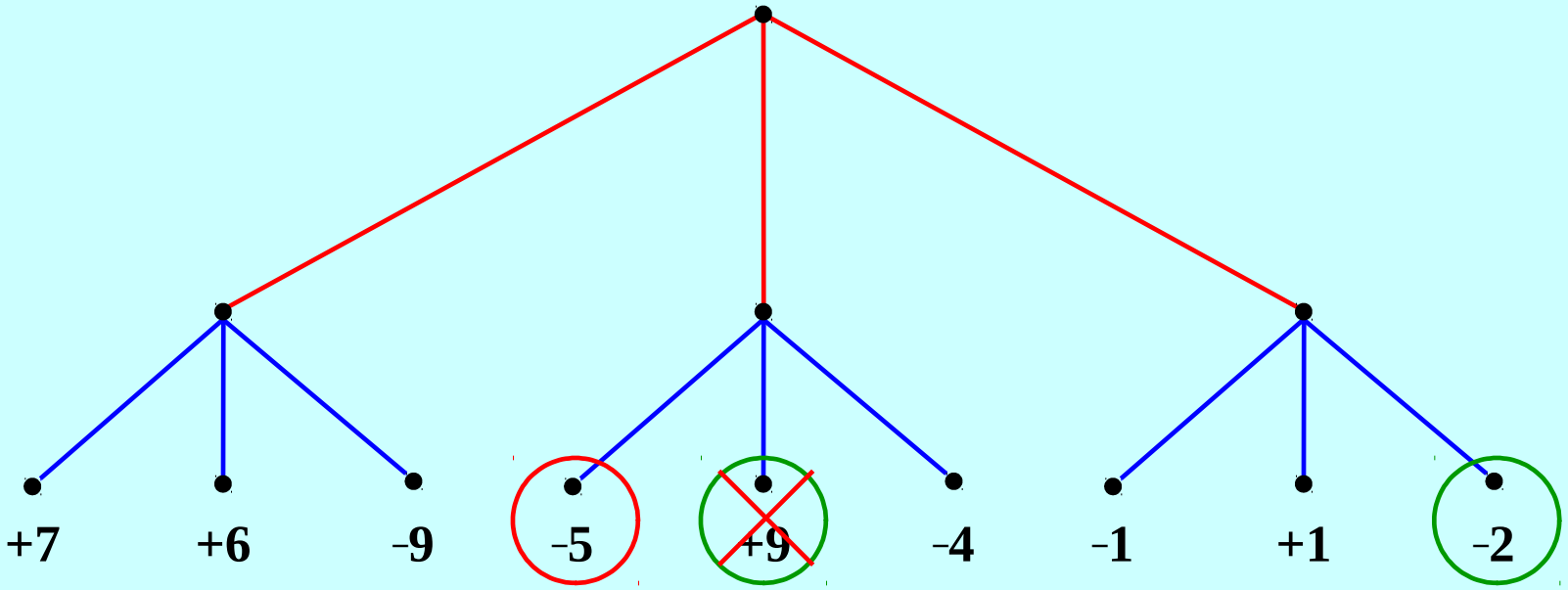
```
/*
 * Looks for a winning move, given the specified number of coins.
 * If there is a winning move in that position, findGoodMove returns
 * that value; if not, the method returns the constant NO_GOOD_MOVE.
 * This implementation depends on the recursive insight that a good move
 * is one that leaves your opponent in a bad position and a bad position
 * is one that offers no good moves.
 */

   int findGoodMove(int nCoins) {
      int limit = (nCoins < MAX_MOVE) ? nCoins : MAX_MOVE;
      for (int nTaken = 1; nTaken <= limit; nTaken++) {
         if (isBadPosition(nCoins - nTaken)) return nTaken;
      }
      return NO_GOOD_MOVE;
   }

/*
 * Returns true if nCoins is a bad position.  Being left with a single
 * coin is clearly a bad position and represents the simple case.
 */

   bool isBadPosition(int nCoins) {
      if (nCoins == 1) return true;
      return findGoodMove(nCoins) == NO_GOOD_MOVE;
   }
```

The Minimax Algorithm

- Games like Nim are simple enough that it is possible to solve them completely in a relatively small amount of time.
- For more complex games, it is necessary to cut off the analysis at some point and then evaluate the position, presumably using some function that looks at a position and returns a ***rating*** for that position. Positive ratings are good for the player to move; negative ones are bad.
- When your game player searches the tree for best move, it can't simply choose the one with the highest rating because you control only half the play.
- What you want instead is to choose the move that minimizes the maximum rating available to your opponent. This strategy is called the ***minimax*** algorithm.

A Minimax Illustration

- Suppose that the ratings two turns from now are as shown.
- From your perspective, the +9 initially looks attractive.
- Unfortunately, your can't get there, since the –5 is better for your opponent.
- The best you can do is choose the move that leads to the –2.



The End