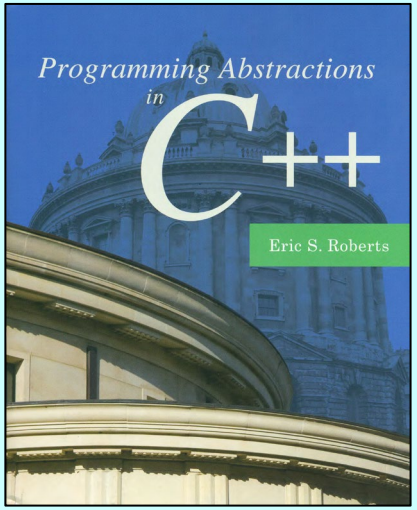
CHAPTER 9

Backtracking Algorithms

Truth is not discovered by proofs but by exploration. It is always experimental.

—Simone Weil, *The New York Notebook,* 1942



Recursive Backtracking

- For many real-world problems, the solution process consists of working your way through a sequence of decision points in which each choice leads you further along some path. If you reach a dead end or otherwise discover that you have made an incorrect choice somewhere along the way, you have to backtrack to a previous decision point and try a different path. Algorithms that use this approach are called ***backtracking algorithms***.
- If you think about a backtracking algorithm as the process of repeatedly exploring paths until you encounter the solution, the process appears to have an *iterative* character. As it happens, however, most problems of this form are easier to solve *recursively*. The fundamental recursive insight is simply this: a backtracking problem has a solution if and only if at least one of the smaller backtracking problems that result from making each possible initial choice has a solution.

Solving a Maze

A journey of a thousand miles begins with a single step.
—Lao Tzu, 6th century B.C.E.

- The example most often used to illustrate ***recursive backtracking*** is the problem of solving a ***maze***, which has a long history in its own right.
- The most famous maze in history is the labyrinth of Daedalus in Greek mythology where Theseus slays the Minotaur.
- There are passing references to this story in *Homer*, but the best known account comes from Ovid in *Metamorphoses*.

Metamorphoses
—Ovid, 1 A.C.E.

. . . When Minos, willing to conceal the shame
That sprung from the reports of tatling Fame,
Resolves a dark inclosure to provide,
And, far from sight, the two-form'd creature hide.

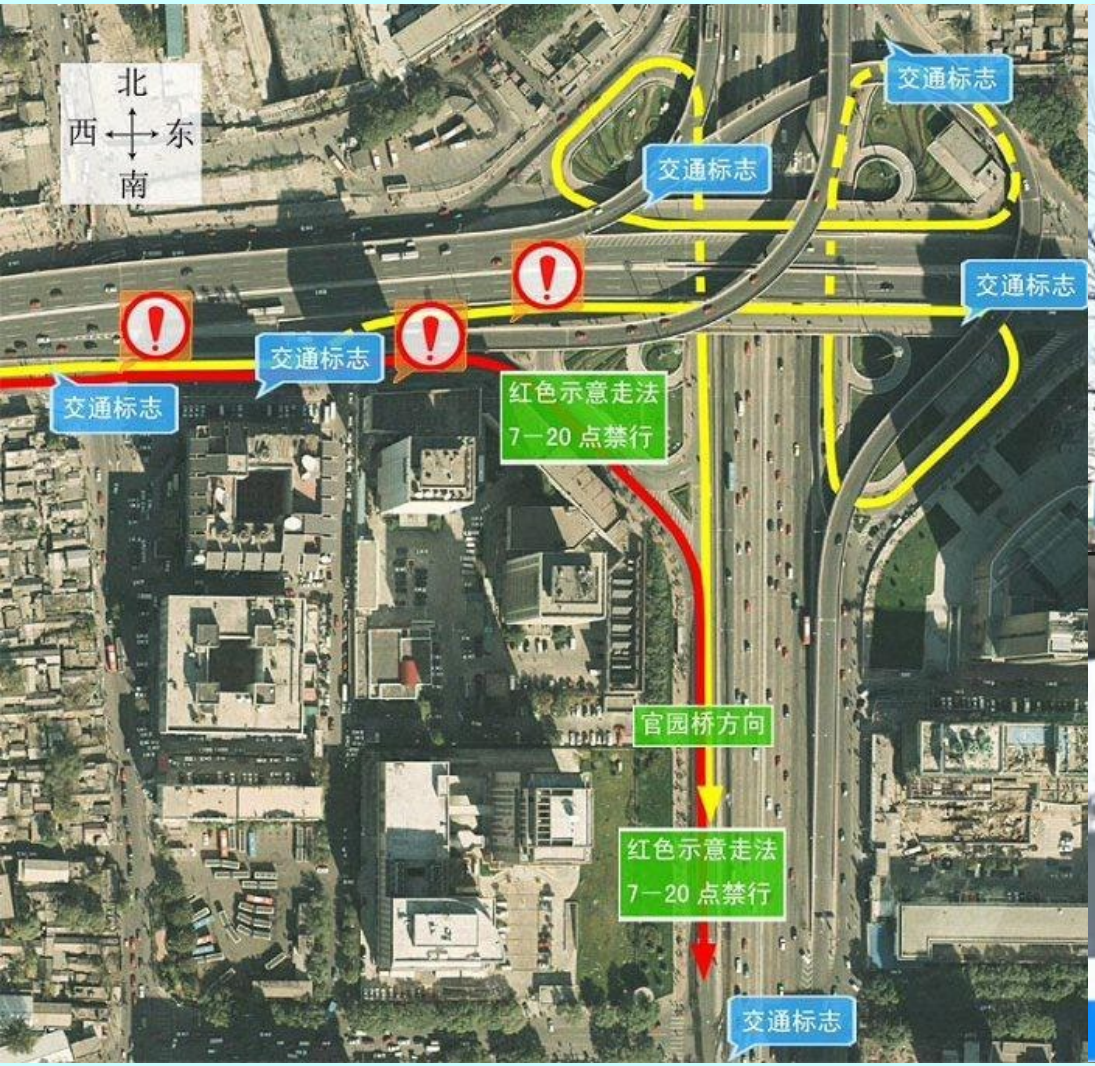
Great Daedalus of Athens was the man
That made the draught, and form'd the wondrous plan;
Where rooms within themselves encircled lye,
With various windings, to deceive the eye. . . .
Such was the work, so intricate the place,
That scarce the workman all its turns cou'd trace;
And Daedalus was puzzled how to find
The secret ways of what himself design'd.

These private walls the Minotaur include,
Who twice was glutted with Athenian blood:
But the third tribute more successful prov'd,
Slew the foul monster, and the plague remov'd.
When Theseus, aided by the virgin's art,
Had trac'd the guiding thread thro' ev'ry part,
He took the gentle maid, that set him free,
And, bound for Dias, cut the briny sea.
There, quickly cloy'd, ungrateful, and unkind,
Left his fair consort in the isle behind . . .

Solving a Maze

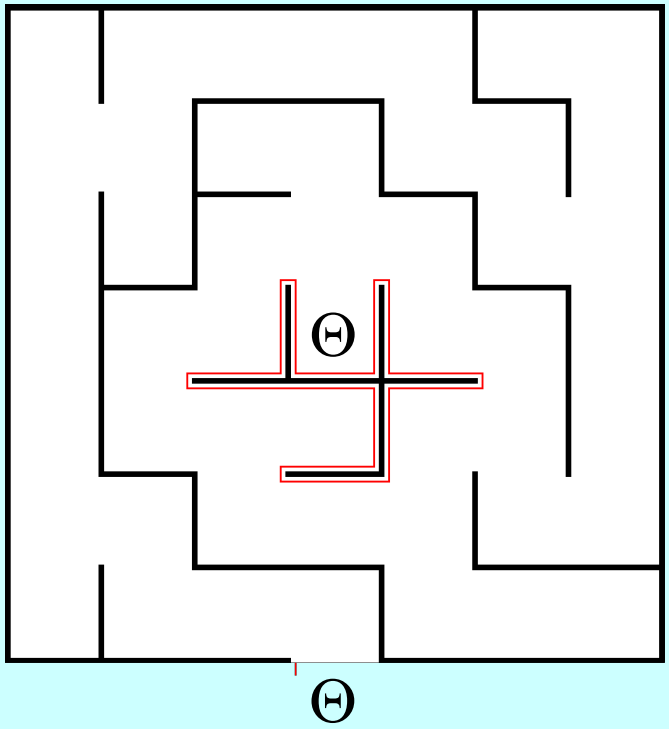
A journey of a thousand miles begins with a single step.

—Lao Tzu, 6th century B.C.E.



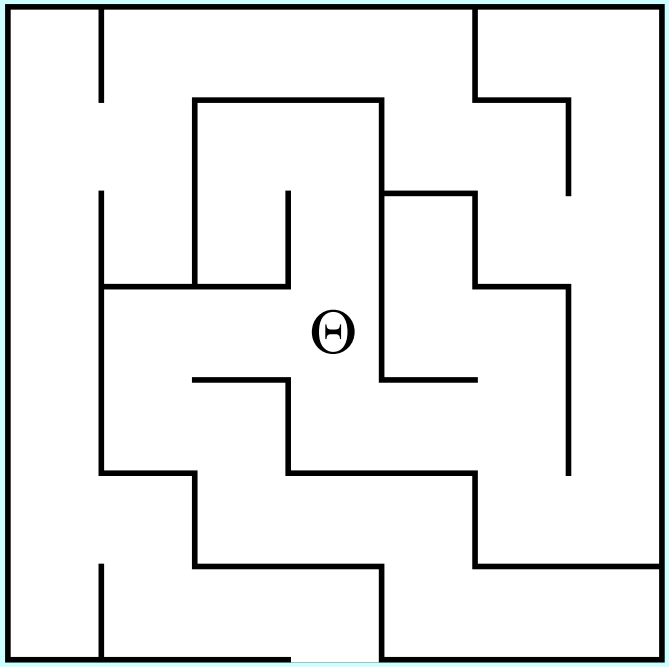
The Right-Hand Rule

- The most widely known strategy for solving a maze is called the ***right-hand rule***, in which you put your right hand on the wall and keep it there until you find an exit.
- If Theseus applies the right-hand rule in this maze, the solution path looks like this.
- Unfortunately, the right-hand rule doesn't work if there are loops in the maze that surround either the starting position or the goal.
- In this maze, the right-hand rule sends Theseus into an infinite loop.



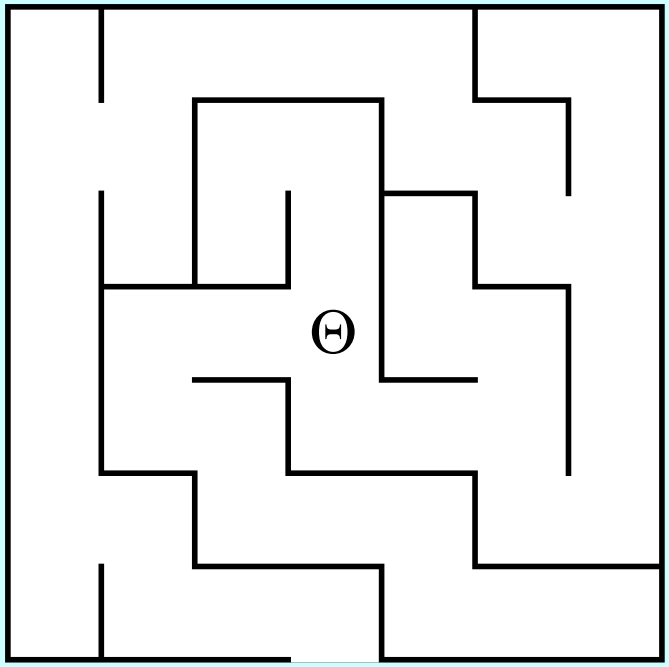
A Recursive View of Mazes

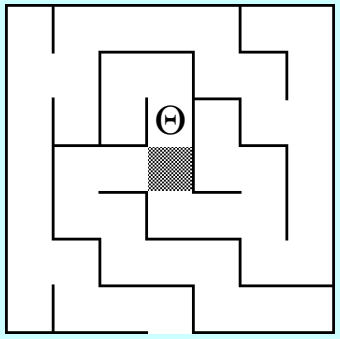
- It is also possible to solve a maze recursively. Before you can do so, however, you have to find the right recursive insight.
- Consider the maze shown at the right. How can Theseus transform the problem into one of solving a simpler maze?
- The insight you need is that a maze is solvable only if it is possible to solve one of the simpler mazes that results from shifting the starting location to an adjacent square and taking the current square out of the maze completely.

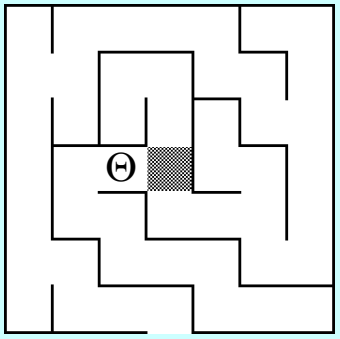


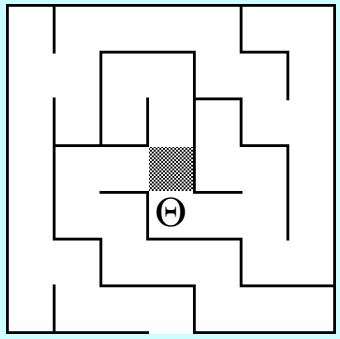
A Recursive View of Mazes

- Thus, the original maze is solvable only if one of the three mazes at the bottom of this slide is solvable.
- Each of these mazes is "simpler" because it contains fewer squares.
- The simple cases are:
  - Theseus is outside the maze
  - There are no directions left to try









Enumerated Types in C++

- It is often convenient to define new types in which the possible values are chosen from a small set of possibilities. Such types are called ***enumerated types***.
- In C++, you define an enumerated type like this:

```
enum name { list of element names };
```

- The code for the maze program uses **`enum`** to define a new type consisting of the four compass points, as follows:

```
enum Direction {
   NORTH, EAST, SOUTH, WEST
};
```

- You can then declare a variable of type **`Direction`** and use it along with the constants **`NORTH`**, **`EAST`**, **`SOUTH`**, and **`WEST`**.

The **Maze** Class

```
/*
 * Class: Maze
 * -----------
 * This class represents a two-dimensional maze contained in a rectangular
 * grid of squares.  The maze is read in from a data file in which the
 * characters '+', '-', and '|' represent corners, horizontal walls, and
 * vertical walls, respectively; spaces represent open passageway squares.
 * The starting position is indicated by the character 'S'.  For example,
 * the following data file defines a simple maze:
 *
 *       +-+-+-+-+-+
 *       |     |
 *       + +-+ + +-+
 *       |S  |     |
 *       +-+-+-+-+-+
 */

class Maze {

public:
```

The `Maze` Class

```
/*
 * Constructor: Maze
 * Usage: Maze maze(filename);
 *        Maze maze(filename, gw);
 * ------------------------------
 * Constructs a new maze by reading the specified data file.  If the
 * second argument is supplied, the maze is displayed in the center
 * of the graphics window.
 */

   Maze(std::string filename);
   Maze(std::string filename, GWindow & gw);

/*
 * Method: getStartPosition
 * Usage: Point start = maze.getStartPosition();
 * ---------------------------------------------
 * Returns a Point indicating the coordinates of the start square.
 */

   Point getStartPosition();
```

The `Maze` Class

```
/*
 * Method: isOutside
 * Usage: if (maze.isOutside(pt)) . . .
 * ------------------------------------
 * Returns true if the specified point is outside the boundary of the maze.
 */

   bool isOutside(Point pt);

/*
 * Method: wallExists
 * Usage: if (maze.wallExists(pt, dir)) . . .
 * ------------------------------------------
 * Returns true if there is a wall in direction dir from the square at pt.
 */

   bool wallExists(Point pt, Direction dir);

/*
 * Method: markSquare
 * Usage: maze.markSquare(pt);
 * ---------------------------
 * Marks the specified square in the maze.
 */

   void markSquare(Point pt);
```

The `Maze` Class

```
/*
 * Method: unmarkSquare
 * Usage: maze.unmarkSquare(pt);
 * -----------------------------
 * Unmarks the specified square in the maze.
 */

   void unmarkSquare(Point pt);

/*
 * Method: isMarked
 * Usage: if (maze.isMarked(pt)) . . .
 * -----------------------------------
 * Returns true if the specified square is marked.
 */

   bool isMarked(Point pt);

/* Private section goes here */

};
```

The `Maze` Class

```
private:

/* Structure representing a single square */

   struct Square {
      bool marked;
      bool walls[4];
   };

/* Instance variables */

   Grid<Square> maze;
   Point startSquare;
   bool mazeShown;
   double x0;
   double y0;
   int rows;
   int cols;

/* Private functions go here */
```

The `solveMaze` Function

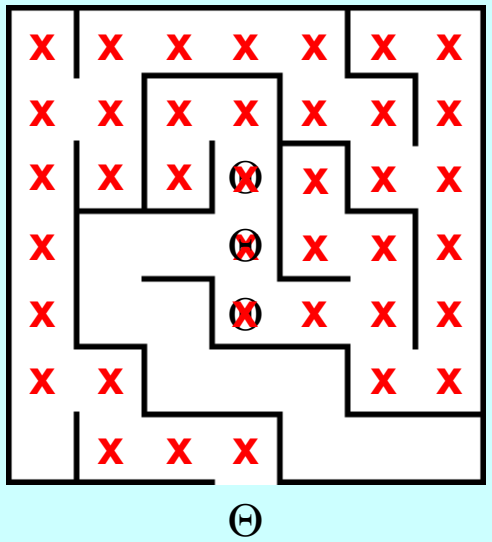
```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses recursion
 * to solve the submazes that result from marking the current square
 * and moving one step along each open passage.
 */

bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjacentPoint(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

Tracing the `solveMaze` Function

```
bool solveMaze(Maze & maze, Point start) {
```

```
bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjPt(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

start	dir
(3, 4)	



Don't follow the recursion more than one level.
Depend on the recursive leap of faith.

Exercise: Keeping Track of the Path

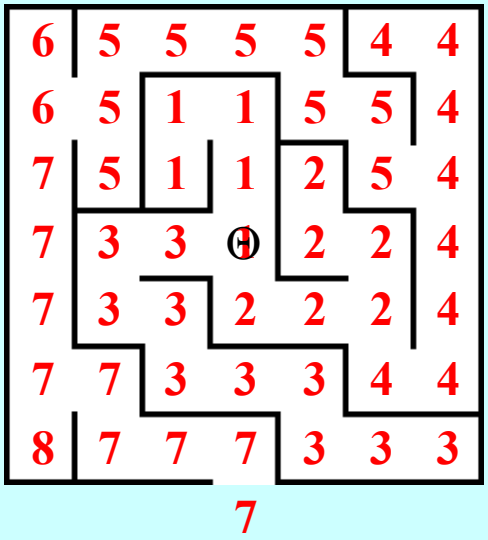
- As described in Exercise 3 in Chapter 9 in the textbook, it is possible to build a better version of **`solveMaze`** so that it keeps track of the solution path as the computation proceeds.
- Write a new function:

```
bool findSolutionPath(Maze & maze, Point start,
                      Vector<Point> & path);
```

that records the solution path in a vector of **`Point`** values passed as a reference parameter. The **`findSolutionPath`** function should return a Boolean value indicating whether the maze is solvable, just as **`solveMaze`** does.

Recursion and Concurrency

- The recursive decomposition of a maze generates a series of independent submazes; the goal is to solve any one of them.
- If you had a multiprocessor computer, you could try to solve each of these submazes in parallel. This strategy is analogous to cloning yourself at each intersection and sending one clone down each path.



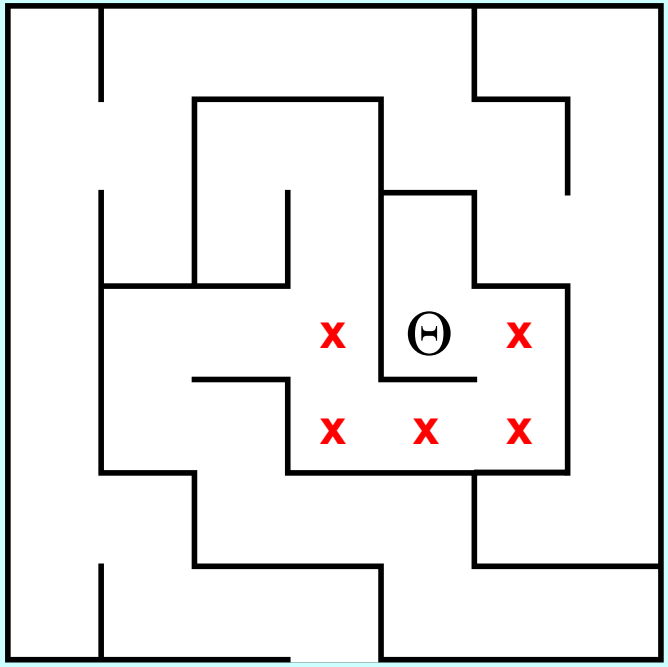
- Is this parallel strategy fundamentally more efficient? The million-dollar question: solving a problem (NP) vs. checking a solution (P)

Reflections on the Maze Problem

- The **`solveMaze`** program is a useful example of how to search all paths that stem from a branching series of choices. At each square, the **`solveMaze`** program calls itself recursively to find a solution from one step further along the path.
- To give yourself a better sense of why recursion is important in this problem, think for a minute or two about what it buys you and why it would be difficult to solve this problem iteratively.
- In particular, how would you answer the following questions:
  - What information does the algorithm need to remember as it proceeds with the solution, particularly about the options it has already tried?
  - In the recursive solution, where is this information kept?
  - How might you keep track of this information otherwise?

Consider a Specific Example

- Suppose that the program has reached the following position:

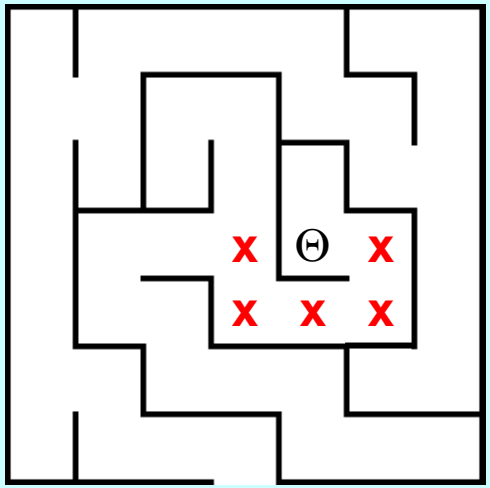


- How does the algorithm keep track of the "big picture" of what paths it still needs to explore?

Each Frame Remembers One Choice

```
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
bool solveMaze(Maze & maze, Point start) {
   if (maze.isOutside(start)) return true;
   if (maze.isMarked(start)) return false;
   maze.markSquare(start);
   for (Direction dir = NORTH; dir <= WEST; dir++) {
      if (!maze.wallExists(start, dir)) {
         if (solveMaze(maze, adjPt(start, dir))) {
            return true;
         }
      }
   }
   maze.unmarkSquare(start);
   return false;
}
```

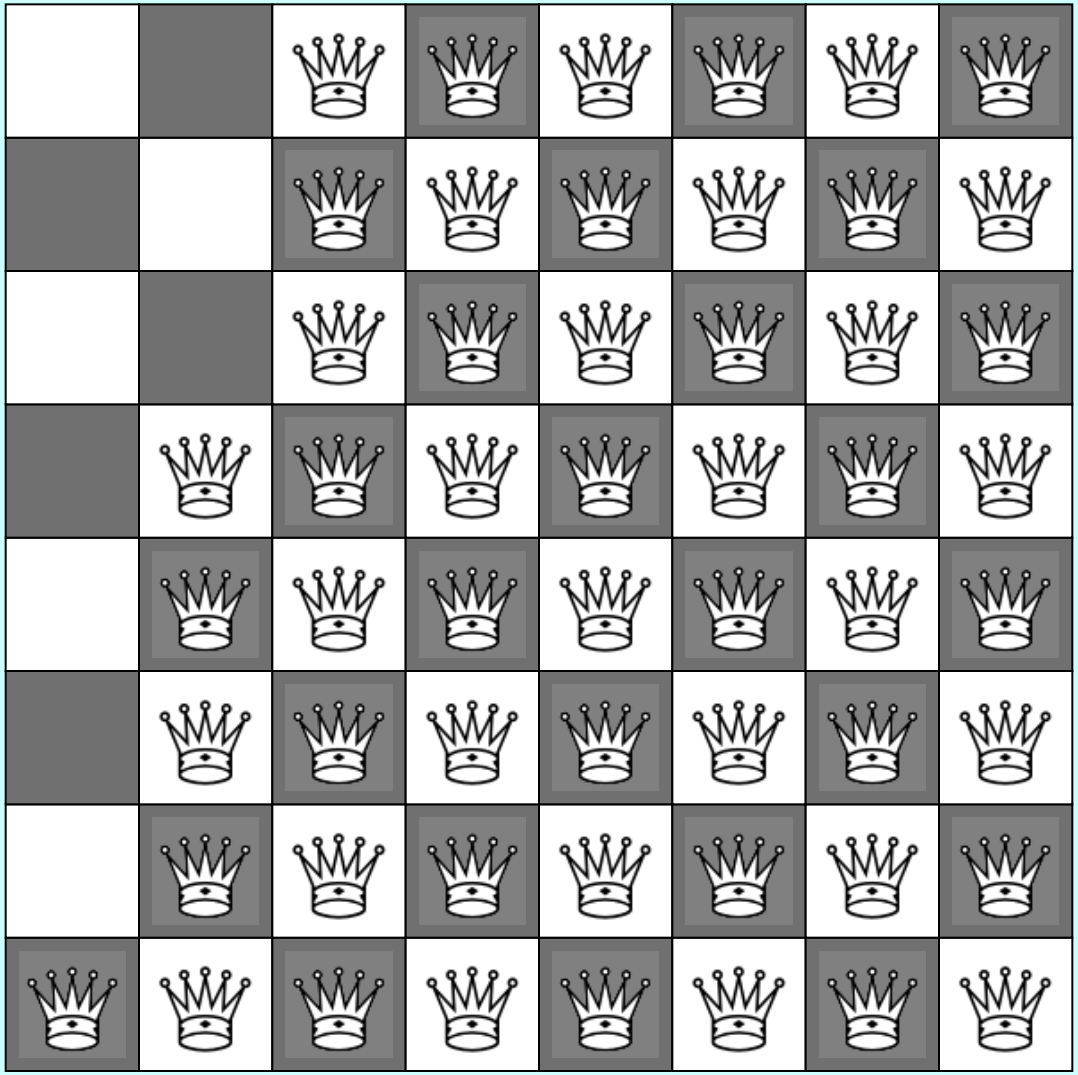
start	dir
(4, 3)	



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.

The Eight Queens Problem



What You Might Attempt to Do

```
bool solveQueens(Grid<char> & board) {
   int n = board.numRows();
   for (int c0 = 0; c0 < n; c0++) {
      board[0][c0] = 'Q';
      for (int c1 = 0; c1 < n; c1++) {
         board[1][c1] = 'Q';
         for (int c2 = 0; c2 < n; c2++) {
            board[2][c2] = 'Q';

            . . .

                  if (boardIsLegal(board)) return true;

            . . .

            board[2][c2] = ' ';
         }
         board[1][c1] = ' ';
      }
      board[0][c0] = ' ';
   }
   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?

NQueens

```
/*
 * Function: solveQueens
 * Usage: bool flag = solveQueens(board, nPlaced);
 * -----------------------------------------------
 * Tries to solve the N-Queens problem on the specified board,
 * which is already populated with nPlaced queens by earlier
 * recursive calls.  The function returns true if a solution
 * is found, otherwise it returns false.
 */

bool solveQueens(Grid<char> & board, int nPlaced) {
   int n = board.numRows();
   if (nPlaced == n) return true;
   for (int row = 0; row < n; row++) {
      if (queenIsLegal(board, row, nPlaced)) {
         board[row][nPlaced] = 'Q';
         if (solveQueens(board, nPlaced + 1)) return true;
         board[row][nPlaced] = ' ';
      }
   }
   return false;
}
```

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.
  - E.g., if you ever find yourself with just one coin on the table, you're in a bad position (you have to take that coin and lose). On the other hand, things look good if you find yourself with two, three, or four coins. In any of these cases, you can always take all but one of the remaining coins (good moves), leaving your opponent in the bad position of being stuck with just one coin.
- 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;
   }
```

Coding the Nim Strategy

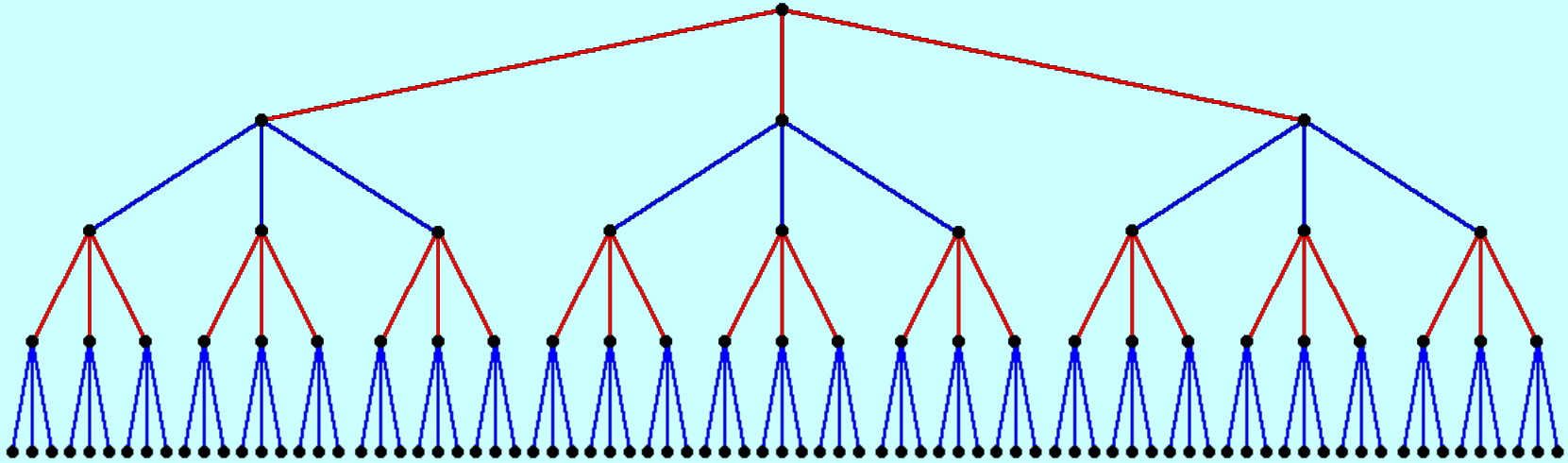
- `isBadPosition(1)` `true`
- `findGoodMove(2)` `1`
- `isBadPosition(2)` `false`
- `findGoodMove(3)` `2`
- `isBadPosition(3)` `false`
- `findGoodMove(4)` `3`
- `isBadPosition(4)` `false`
- `findGoodMove(5)` `NO_GOOD_MOVE`
- `isBadPosition(5)` `true`
- `...`

A generalized program for two-player games

```
struct Move {
   int nTaken;
};

void play() {
   initGame();
   while (!gameIsOver()) {
      displayGame();
      if (getCurrentPlayer() == HUMAN) {
         makeMove(getUserMove());
      } else {
         Move move = getComputerMove();
         displayMove(move);
         makeMove(move);
      }
      switchTurn();
   }
   announceResult();
}

// For Nim, move is simply nTaken, and getComputerMove() looks like this:
int getComputerMove() {
   int nTaken = findGoodMove(nCoins);
   return (nTaken == NO_GOOD_MOVE) ? 1 : nTaken;
}
```

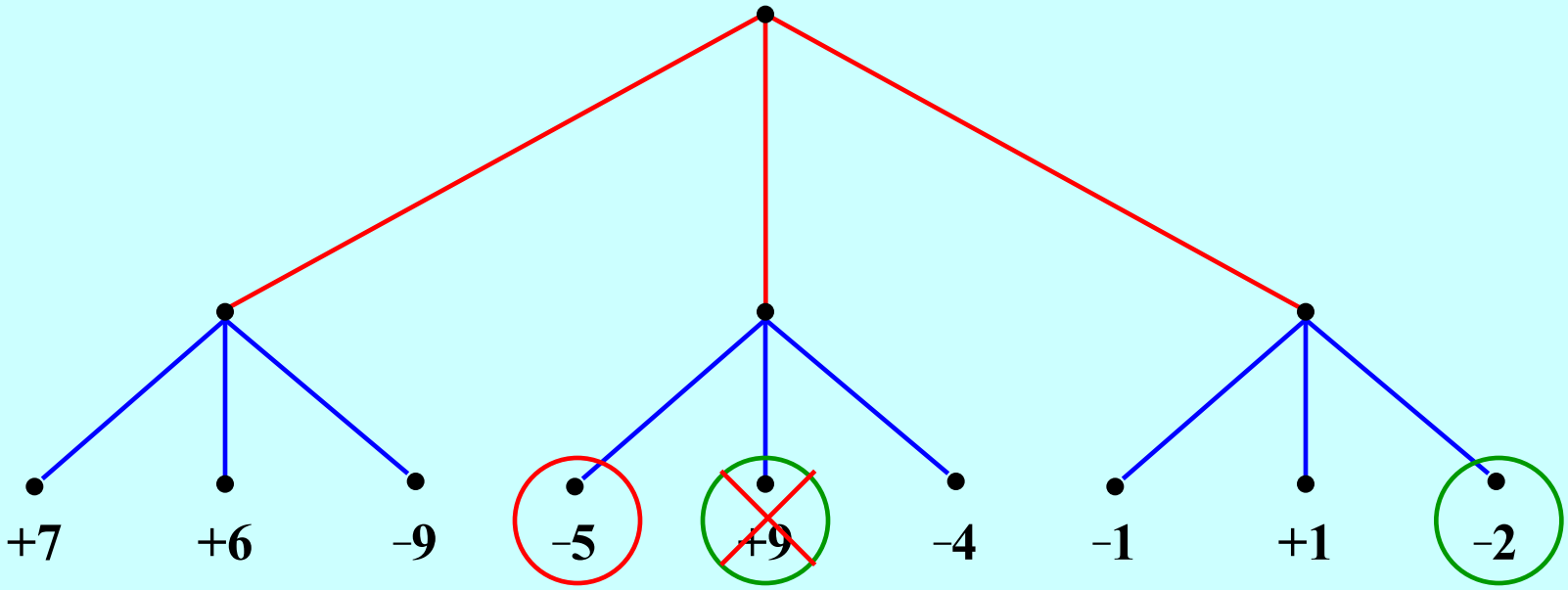
Game Trees

- 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 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 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.

The Minimax Algorithm

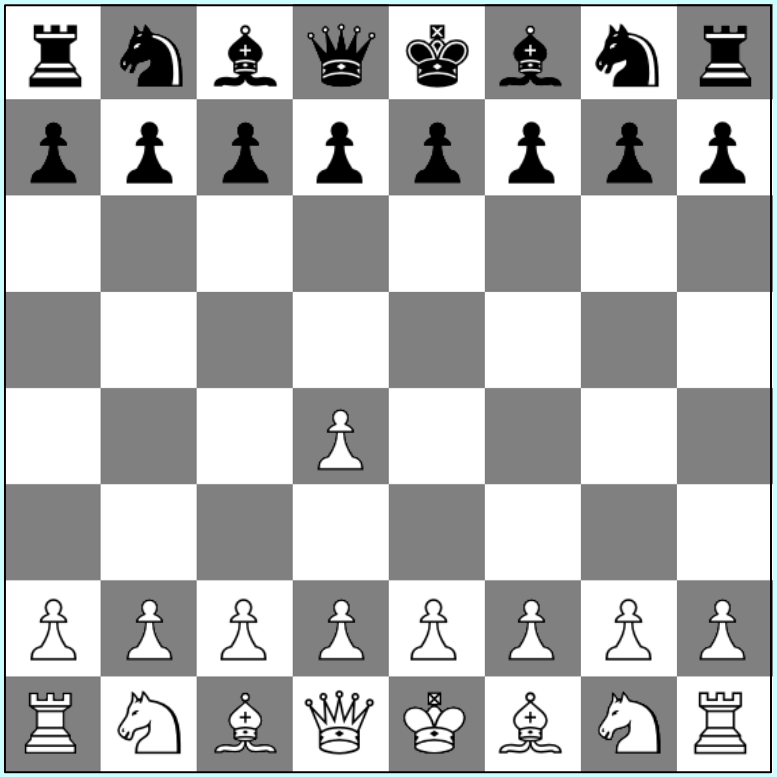
- Games like Nim are simple enough that it is possible to solve them completely in a relatively small amount of time.
- For more complex games, it is necessary to cut off the analysis at some point and then evaluate the position, presumably using some function that looks at a position and returns a ***rating*** 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.

FIGURE 9-6 Generalized implementation of the minimax algorithm

```
/*
 * Method: findBestMove
 * Usage: Move move = findBestMove();
 *        Move move = findBestMove(depth, rating);
 * ----------------------------------------------
 * Finds the best move for the current player and returns that move as the
 * value of the function.  The second form is used for later recursive calls
 * and includes two parameters.  The depth parameter is used to limit the
 * depth of the search for games that are too difficult to analyze.  The
 * reference parameter rating is used to store the rating of the best move.
 */

   Move findBestMove() {
      int rating;
      return findBestMove(0, rating);
   }

   Move findBestMove(int depth, int & rating) {
      Vector<Move> moveList;
      Move bestMove;
      int minRating = WINNING_POSITION + 1;
      generateMoveList(moveList);
      if (moveList.isEmpty()) error("No moves available");
      for (Move move : moveList) {
         makeMove(move);
         int moveRating = evaluatePosition(depth + 1);
         if (moveRating < minRating) {
            bestMove = move;
            minRating = moveRating;
         }
         retractMove(move);
      }
      rating = -minRating;
      return bestMove;
   }

/*
 * Method: evaluatePosition
 * Usage: int rating = evaluatePosition(depth);
 * --------------------------------------------
 * Evaluates a position by finding the rating of the best move starting at
 * that point.  The depth parameter is used to limit the search depth.
 */

   int evaluatePosition(int depth) {
      if (gameIsOver() || depth >= MAX_DEPTH) {
         return evaluateStaticPosition();
      }
      int rating;
      findBestMove(depth, rating);
      return rating;
   }
```

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: $\sim 10^{53}$

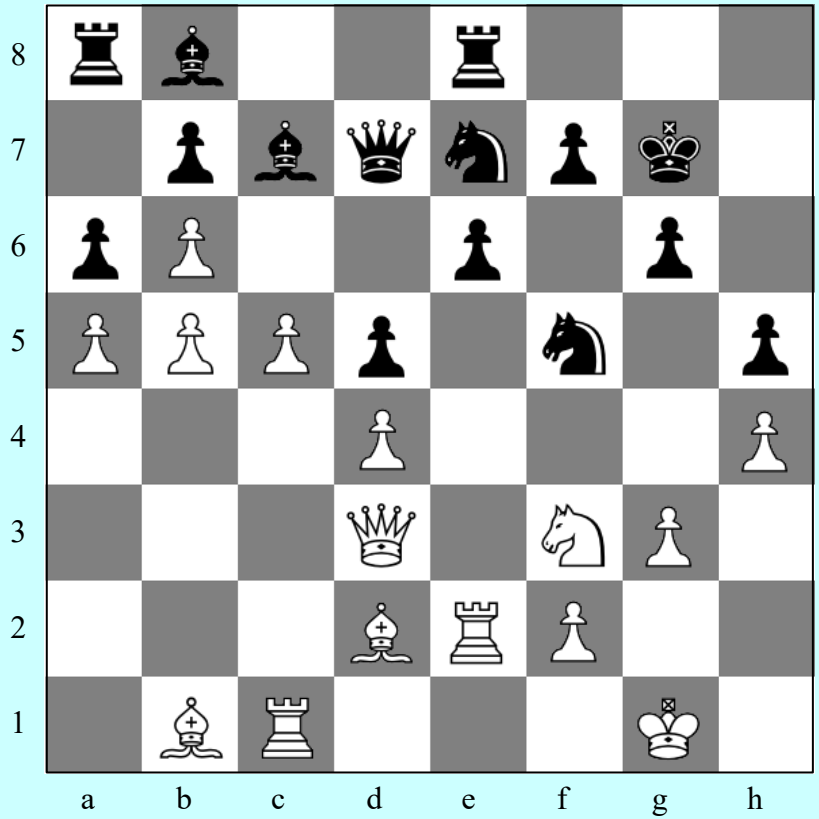
. . . millions of years later . . .

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 ??

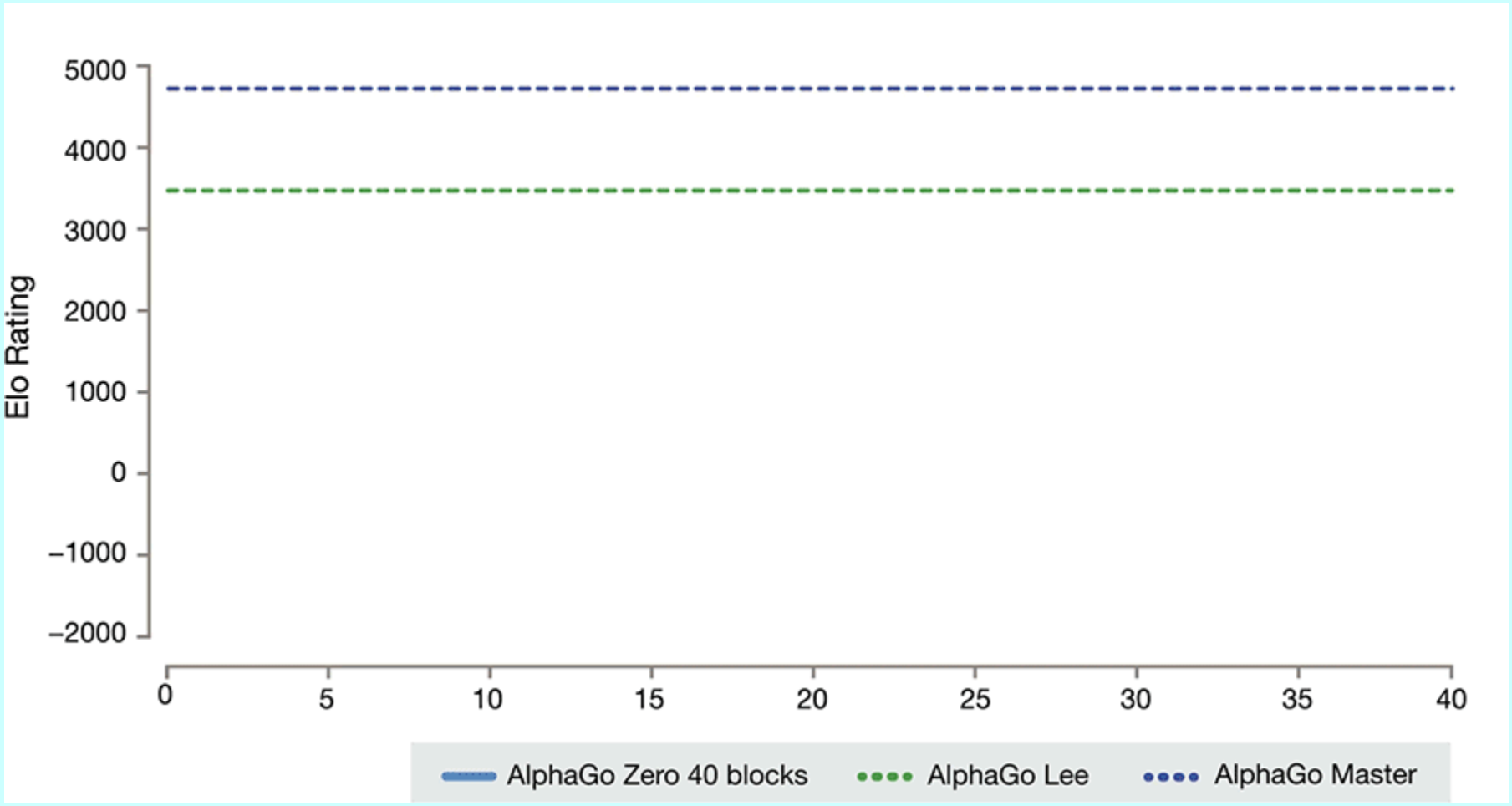


AlphaGo vs. Lee Sedol

AlphaGo's algorithm

- Go is even more difficult than Chess. ~361! moves (~10^{768}).
- AlphaGo's algorithm uses a combination of machine learning and tree search techniques, combined with extensive training, both from human and computer play.
- It uses Monte Carlo tree search, guided by a "value network" and a "policy network", both implemented using deep neural network technology.
- The system's neural networks were initially bootstrapped from human gameplay expertise from recorded historical games, using a database of around 30 million moves.
- Once it had reached a certain degree of proficiency, it was trained further by being set to play large numbers of games against other instances of itself, using reinforcement learning to improve its play.

AlphaGo Zero



Libratus beats poker pros

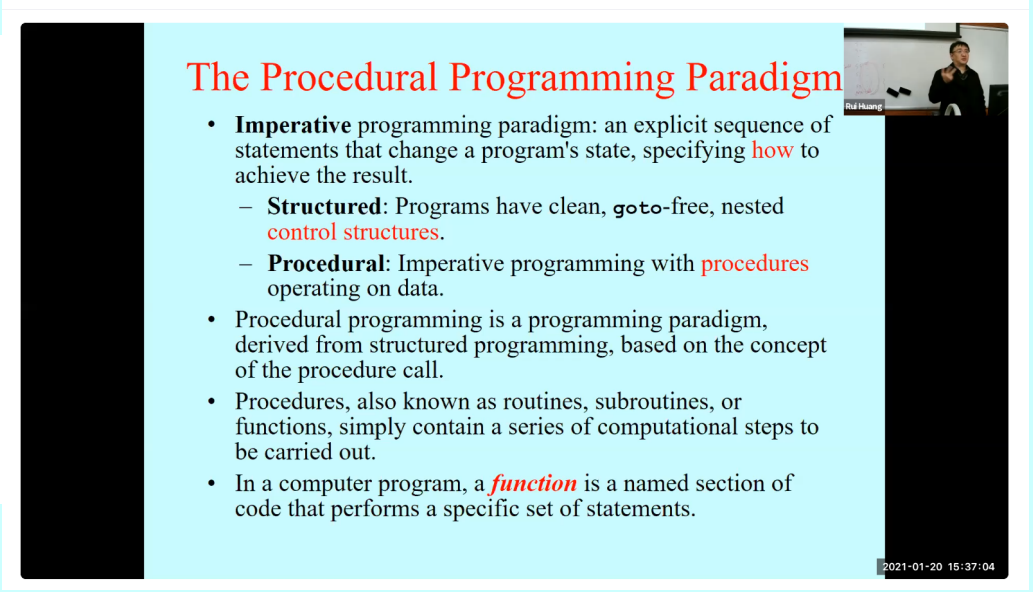
- What about Mahjong?
  Limited information (luck, bluff, …) games are harder for AI.

What is AI?

- What is intelligence, artificial intelligence, machine intelligence…?
- Is machine intelligence just pure computing power? A story about Chinese chess endgame.
- Art is about creativity. Can AI do art?
- What is exactly creativity?
- The infinite monkey theorem
- What if we have infinite computers?

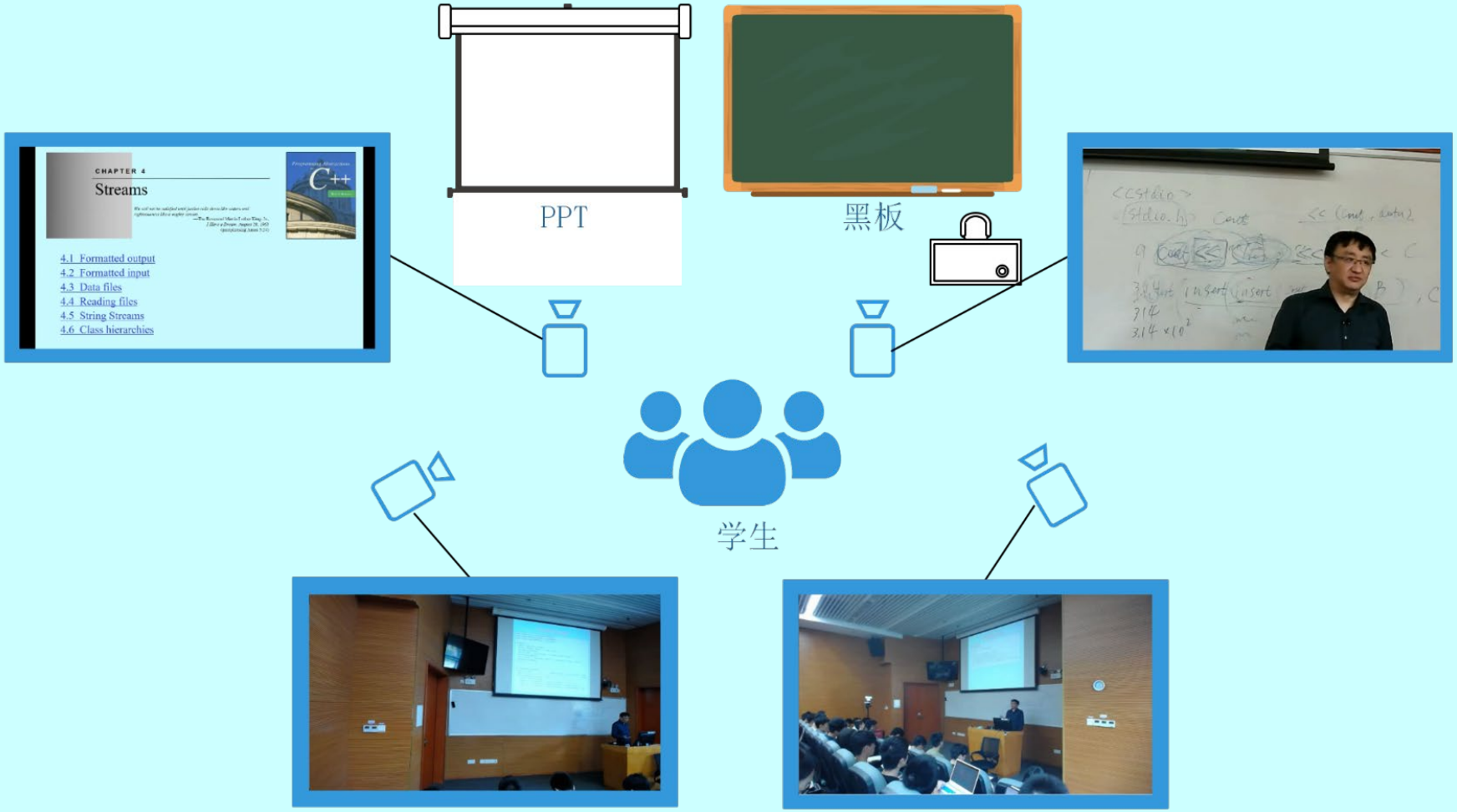
AI on Campus

- What you see on Zoom (online)



AI on Campus

- What you see in a real classroom (onsite)

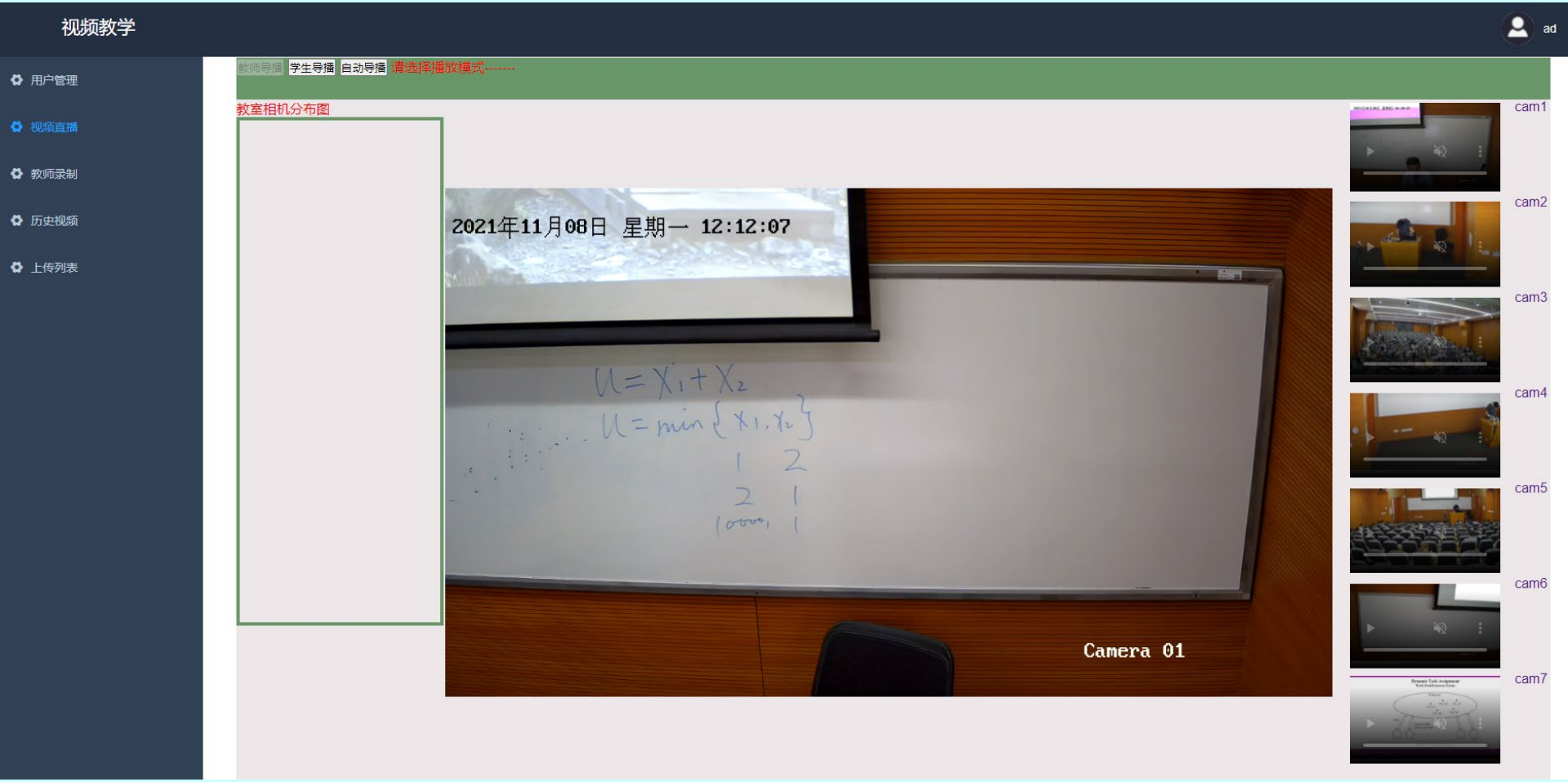


AI on Campus

- What we help you see online with our AI system

AI on Campus

- What our AI system looks like (in development, join us!)



The End