Recursive Backtracking

CS 2213
Advanced Programming
Ch 6 – Backtracking Algorithms

Ch 6 is optional.

We will cover this chapter later if time permits

Turgay Korkmaz

Thanks to Eric S. Roberts, the author of our textbook, for providing all the slides for this chapter.

Solving a Maze

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

—Confucius, 5th 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.

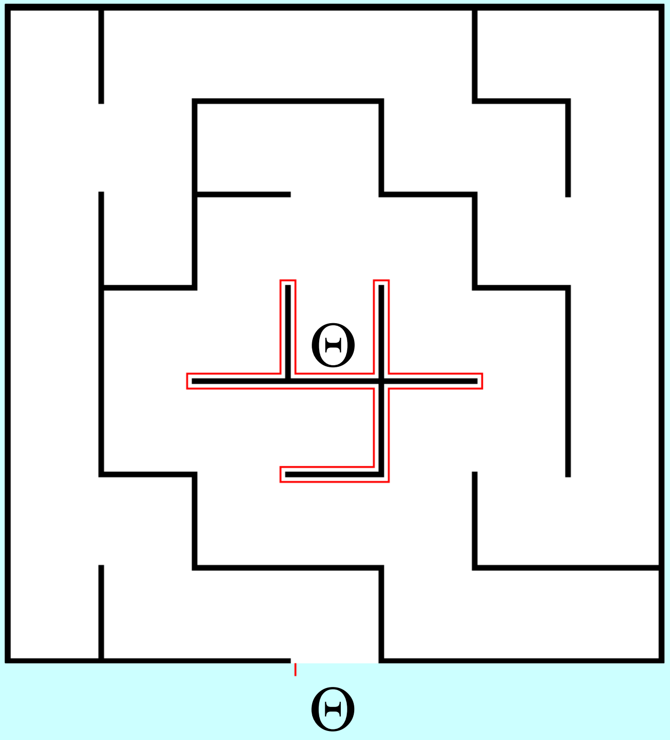
. . . When Minos, willing to conceal the shame
That sprung from the reports of tatling Fame,
Resolves a dark inclosure to provide,
And, far from sight, the two-form'd creature hide.

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 . . .

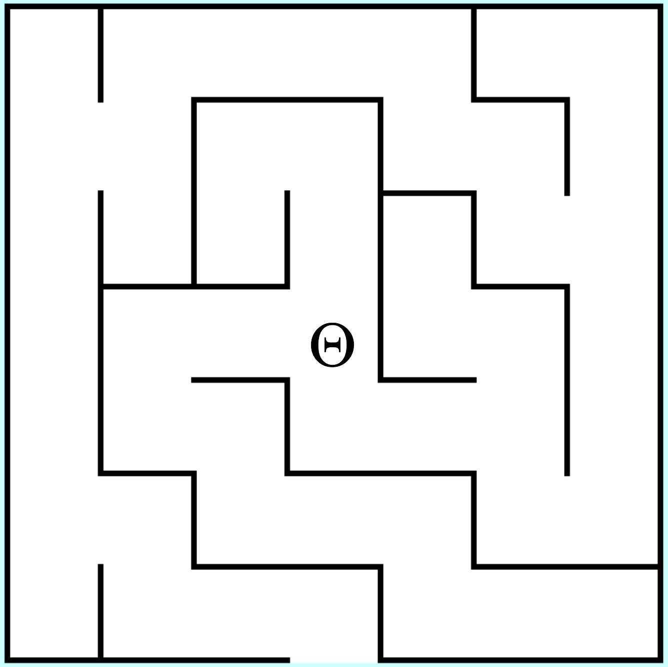
The Right-Hand Rule

- 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.



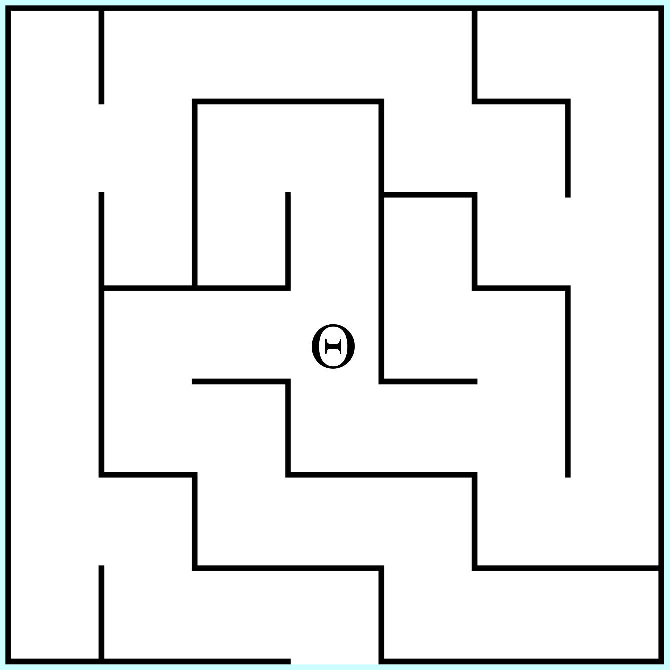
A Recursive View of Mazes

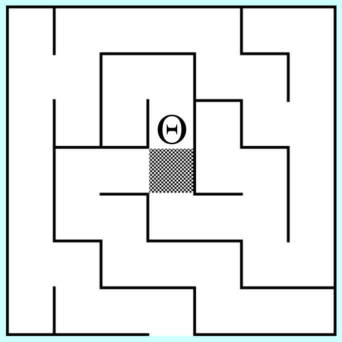
- It is also possible to solve a maze recursively. Before you can do so, however, you have to find the right recursive insight.
- Consider the maze shown at the right. How can Theseus transform the problem into one of solving a simpler maze?
- The insight you need is that a maze is solvable only if it 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.

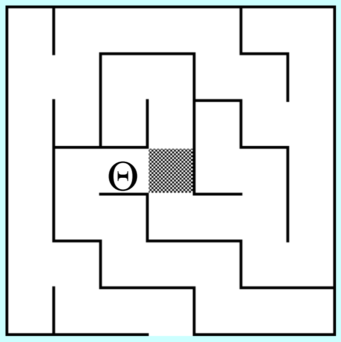


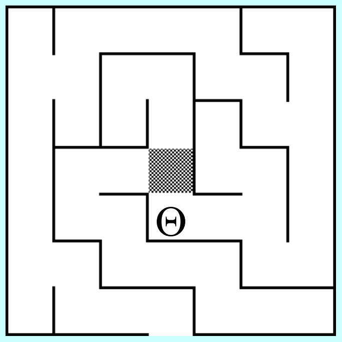
A Recursive View of Mazes

- 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









The `mazelib.h` Interface

```
/*
 * File: mazelib.h
 * ---------------
 * This interface provides a library of primitive operations
 * to simplify the solution to the maze problem.
 */

#ifndef _mazelib_h
#define _mazelib_h

#include "genlib.h"

/*
 * This type is used to represent the four compass directions.
 */

enum directionT { North, East, South, West };

/*
 * The type pointT is used to encapsulate a pair of integer
 * coordinates into a single value with x and y components.
 */

struct pointT {
   int x, y;
};
```

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 **`mazelib.h`** interfaces uses **`enum`** to define a new type consisting of the four compass points, as follows:

```
enum directionT {
   North, East, South, West
};
```

- You can then declare a variable of type **`directionT`** and use it along with the constants **`North`**, **`East`**, **`South`**, and **`West`**.

Structure Types in C

- The other new type mechanism included in the **`mazelib.h`** interface is the creation of a ***structure type*** to hold the *x* and *y* components of a point within a maze. That definition is

```
struct pointT {
   int x, y;
};
```

- This definition creates a new type called **`pointT`** that has two fields: an **`int`** named **`x`** and another **`int`** named **`y`**.
- You can declare variables of type **`pointT`** just as you would variables of any other type.
- Once you have a variable of type **`pointT`**, you can refer to the individual components by using a dot to select the appropriate field. For example, if **`currentLocation`** is a **`pointT`**, you can select its **`x`** component by writing **`currentLocation.x`**.

The **mazelib.h** Interface

```
/*
 * Function: ReadMazeMap
 * Usage: ReadMazeMap(filename);
 * -----------------------------
 * This function reads in a map of the maze from the specified
 * file and stores it in private data structures maintained by
 * this module.  In the data file, 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  |     |
 *       +-+-+-+-+-+
 *
 * Coordinates in the maze are numbered starting at (0,0) in
 * the lower left corner.  The goal is to find a path from
 * the (0,0) square to the exit east of the (4,1) square.
 */

void ReadMazeMap(string filename);
```

The `mazelib.h` Interface

```
/*
 * Function: GetStartPosition
 * Usage: pt = GetStartPosition();
 * -------------------------------
 * This function returns a pointT indicating the coordinates of
 * the start square.
 */

pointT GetStartPosition();

/*
 * Function: OutsideMaze
 * Usage: if (OutsideMaze(pt)) . . .
 * ---------------------------------
 * This function returns true if the specified point is outside
 * the boundary of the maze.
 */

bool OutsideMaze(pointT pt);
```

The **mazelib.h** Interface

```
/*
 * Function: WallExists
 * Usage: if (WallExists(pt, dir)) . . .
 * -------------------------------------
 * This function returns true if there is a wall in the indicated
 * direction from the square at position pt.
 */

bool WallExists(pointT pt, directionT dir);

/*
 * Functions: MarkSquare, UnmarkSquare, IsMarked
 * Usage: MarkSquare(pt);
 *        UnmarkSquare(pt);
 *        if (IsMarked(pt)) . . .
 * ------------------------------
 * These functions mark, unmark, and test the status of the
 * square specified by the coordinates pt.
 */

void MarkSquare(pointT pt);
void UnmarkSquare(pointT pt);
bool IsMarked(pointT pt);

#endif
```

The SolveMaze Function

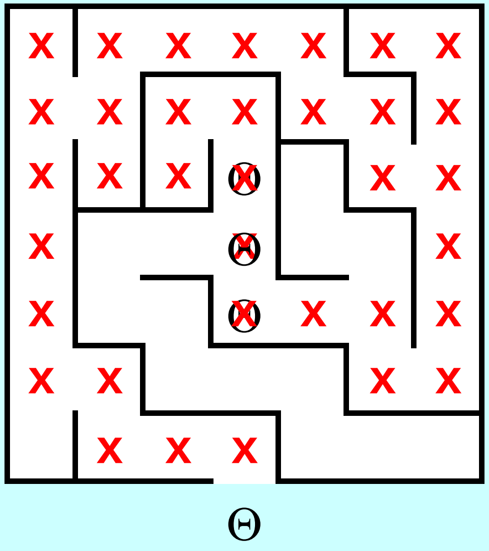
```
/*
 * Function: SolveMaze
 * Usage: if (SolveMaze(pt)) . . .
 * -------------------------------
 * This function attempts to generate a solution to the current maze from
 * point pt.  SolveMaze returns true if the maze has a solution.  The
 * implementation uses recursion to solve the submazes that result from
 * marking the current square and moving one step along each open passage.
 */

bool SolveMaze(pointT pt) {
   if (OutsideMaze(pt)) return true;
   if (IsMarked(pt)) return false;
   MarkSquare(pt);
   for (int i = 0; i < 4; i++) {
      directionT dir = directionT(i);
      if (!WallExists(pt, dir)) {
         if (SolveMaze(AdjacentPoint(pt, dir))) {
            return true;
         }
      }
   }
   UnmarkSquare(pt);
   return false;
}
```

Tracing the **`SolveMaze`** Function

```
bool SolveMaze(pointT pt) {
```

```
bool SolveMaze(pointT pt) {
   if (OutsideMaze(pt)) return true;
   if (IsMarked(pt)) return false;
   MarkSquare(pt);
   for (int i = 0; i < 4; i++) {
      directionT dir = directionT(i);
      if (!WallExists(pt, dir)) {
         if (SolveMaze(AdjacentPoint(pt, dir))) {
            return true;
         }
      }
   }
   UnmarkSquare(pt);
   return false;
}
```

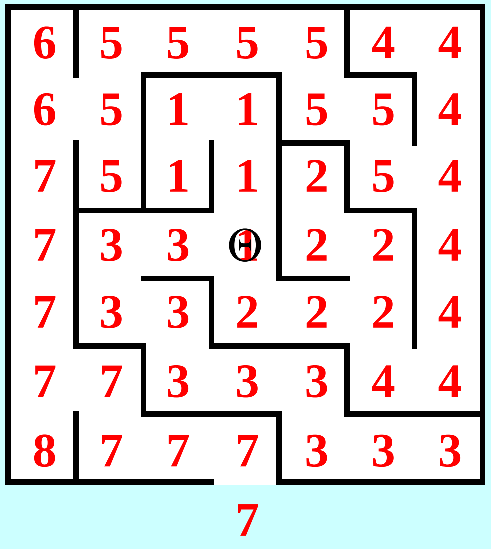
pt	i	dir
(3, 2)		



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

Recursion and Concurrency

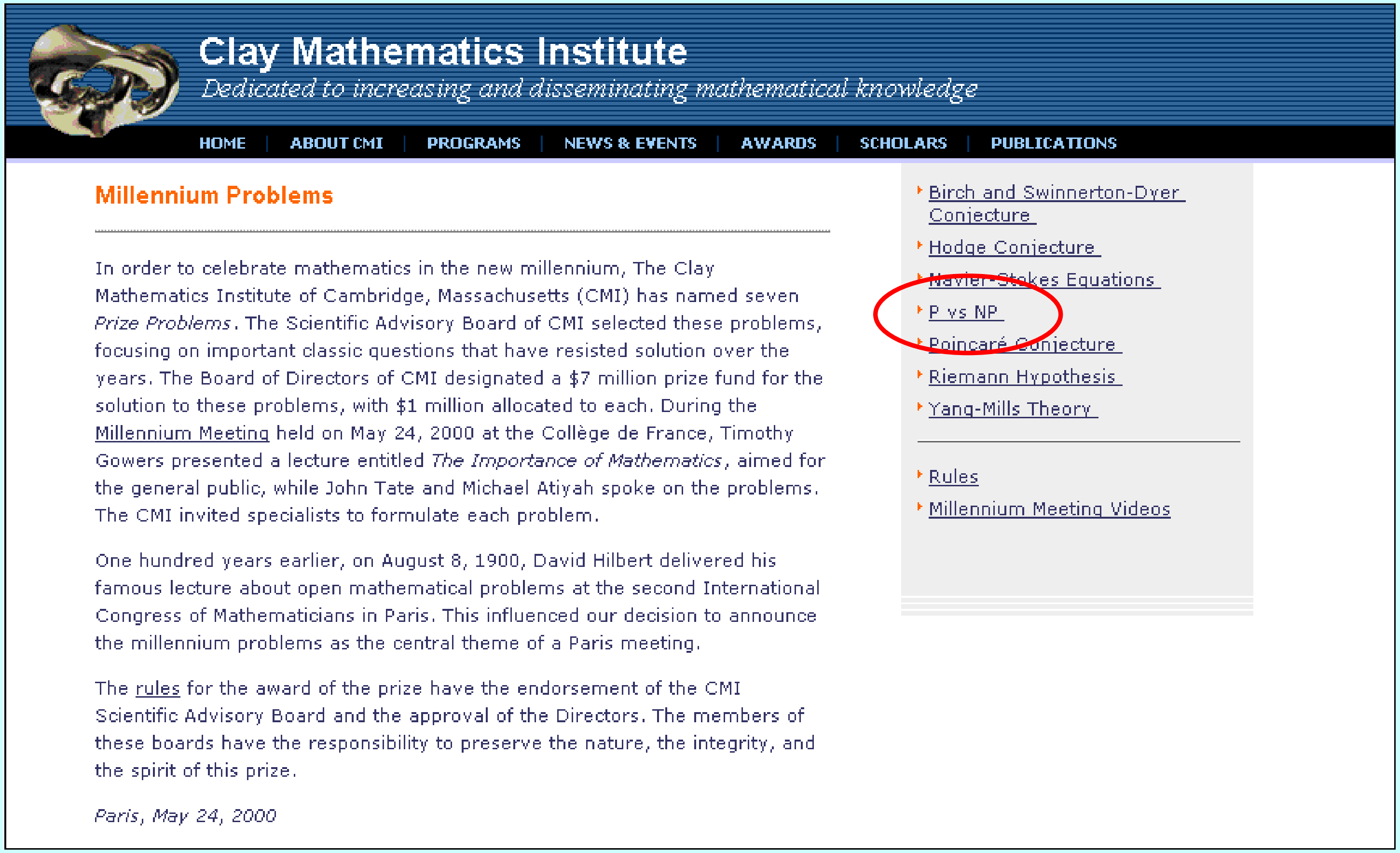
- The recursive decomposition of a maze generates a series of independent submazes; the goal is to solve any one of them.
- If you had a multiprocessor computer, you could try to solve each of these submazes in parallel. This strategy is analogous to cloning yourself at each intersection and sending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



- Is this parallel strategy more efficient?

The P = NP Question

- The question of whether a parallel solution is fundamentally faster than a sequential one is related to the biggest open problem in computer science, for which there is a $1M prize.



Clay Mathematics Institute

Dedicated to increasing and disseminating mathematical knowledge

HOME | ABOUT CMI | PROGRAMS | NEWS & EVENTS | AWARDS | SCHOLARS | PUBLICATIONS

Millennium Problems

In order to celebrate mathematics in the new millennium, The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) has named seven *Prize Problems*. The Scientific Advisory Board of CMI selected these problems, focusing on important classic questions that have resisted solution over the years. The Board of Directors of CMI designated a $7 million prize fund for the solution to these problems, with $1 million allocated to each. During the Millennium Meeting held on May 24, 2000 at the Collège de France, Timothy Gowers presented a lecture entitled *The Importance of Mathematics*, aimed for the general public, while John Tate and Michael Atiyah spoke on the problems. The CMI invited specialists to formulate each problem.

One hundred years earlier, on August 8, 1900, David Hilbert delivered his famous lecture about open mathematical problems at the second International Congress of Mathematicians in Paris. This influenced our decision to announce the millennium problems as the central theme of a Paris meeting.

The rules for the award of the prize have the endorsement of the CMI Scientific Advisory Board and the approval of the Directors. The members of these boards have the responsibility to preserve the nature, the integrity, and the spirit of this prize.

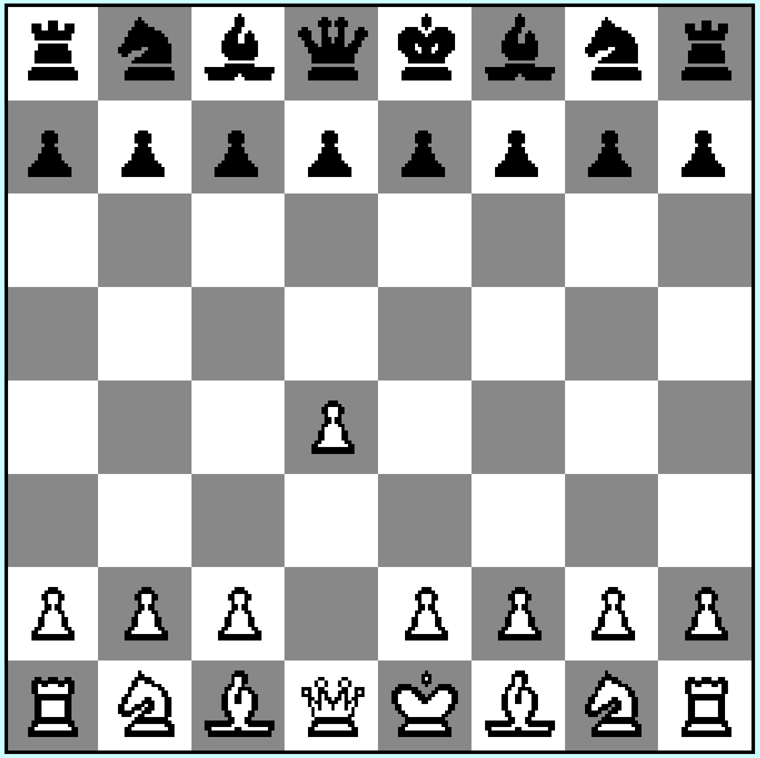
Paris, May 24, 2000

- Birch and Swinnerton-Dyer Conjecture
- Hodge Conjecture
- Navier-Stokes Equations
- P vs NP
- Poincaré Conjecture
- Riemann Hypothesis
- Yang-Mills Theory

- Rules
- Millennium Meeting Videos

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.

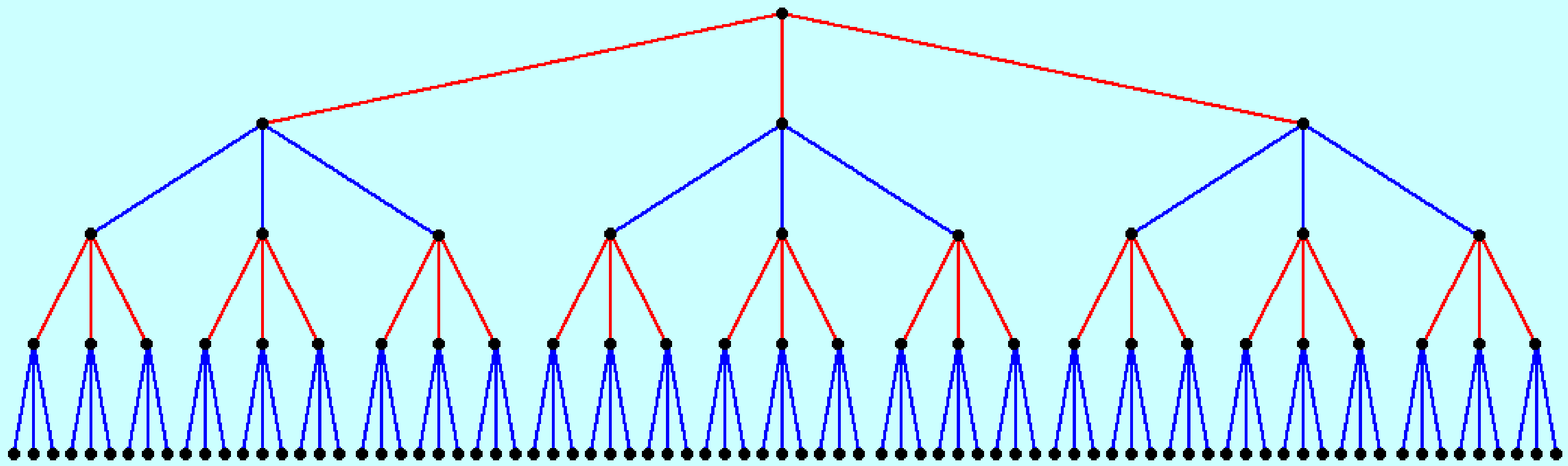


Positions evaluated: ~ 10^{53}

. . . *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 . . .



The Essential Idea about Recursive Games

- If you take nothing else away from the discussion of games in Chapter 6, you should make sure you understand the recursive definition of the following terms:
  - A ***good move*** is one that leaves your opponent in a bad position.
  - A ***bad position*** is one that has no good moves.
- That single idea is the essence of the minimax strategy, which has made it possible for computers, for example, to beat the world chess champion.

The End