# Recursive Backtracking

Eric Roberts
CS 106B
January 23, 2015

# 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 CE

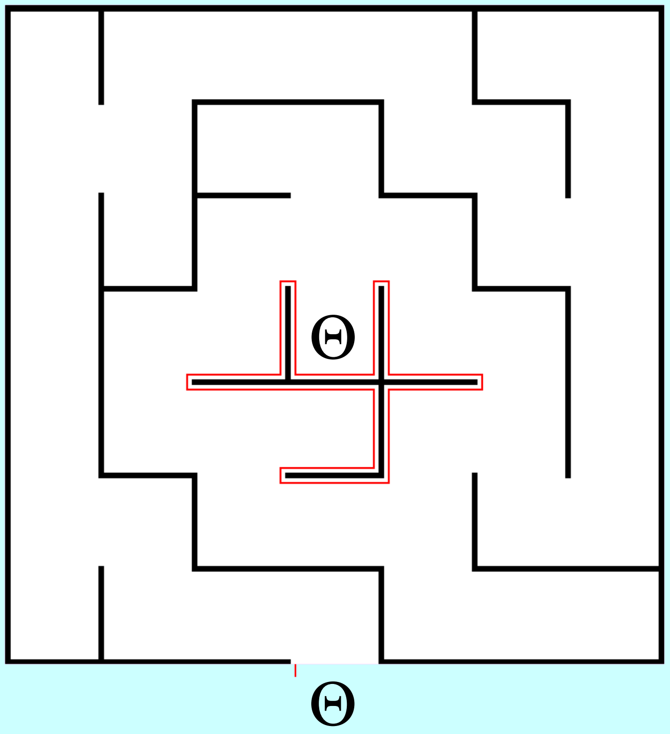
. . . When Minos, willing to conceal the shame
That sprung from the reports of tatling Fame,
Resolves a dark inclosure to provide,
And, far from sight, the two-form'd creature hide.

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 . . .

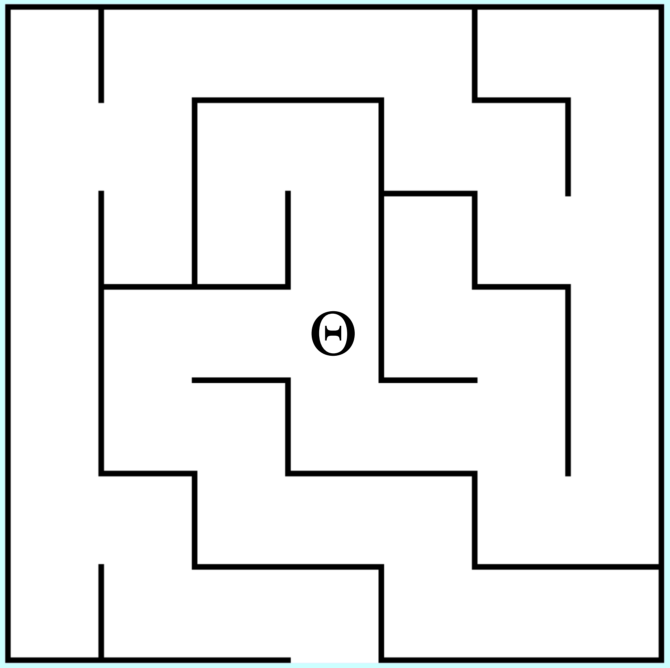
# The Right-Hand Rule

- 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.



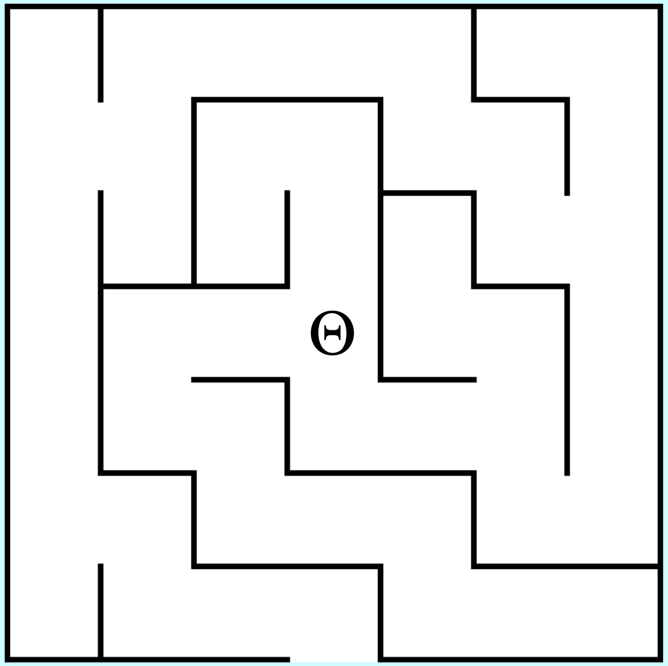
# A Recursive View of Mazes

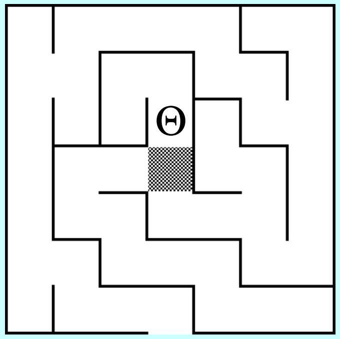
- It is also possible to solve a maze recursively. Before you can do so, however, you have to find the right recursive insight.
- Consider the maze shown at the right. How can Theseus transform the problem into one of solving a simpler maze?
- The insight you need is that a maze is solvable only if it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

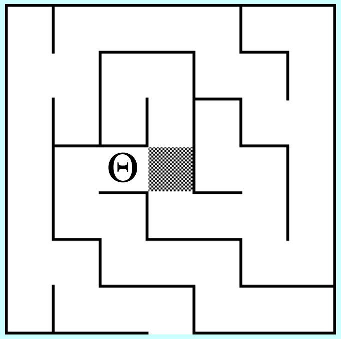


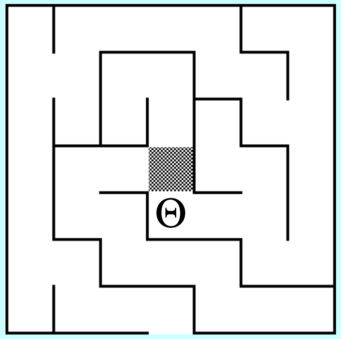
# A Recursive View of Mazes

- 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









# 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 **solveMaze** Function

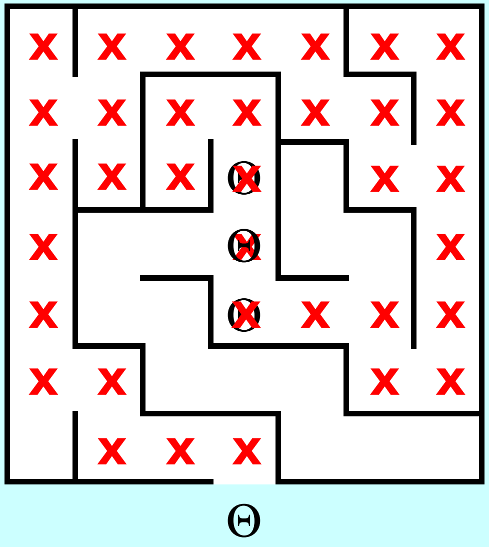
```
/*
 * Function: solveMaze
 * Usage: solveMaze(maze, start);
 * ------------------------------
 * Attempts to generate a solution to the current maze from the specified
 * start point.  The solveMaze function returns true if the maze has a
 * solution and false otherwise.  The implementation uses 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) | |



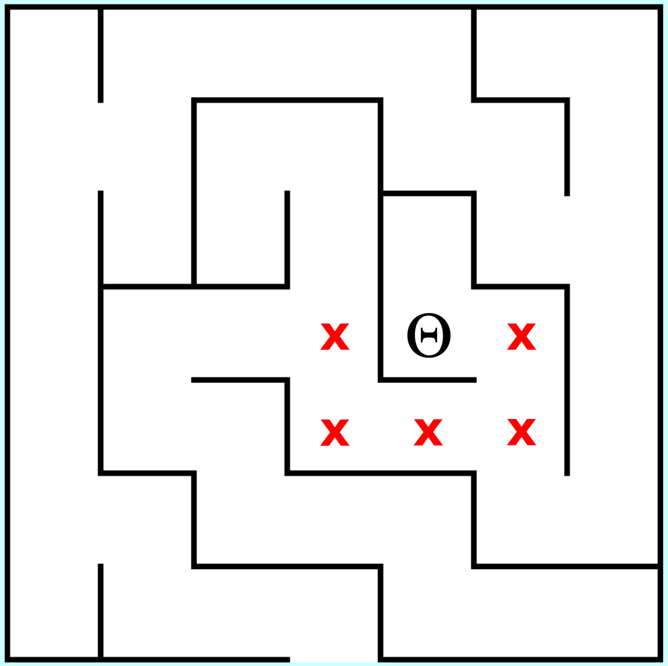
*Don't follow the recursion more than one level.*
*Depend on the recursive leap of faith.*

# 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:

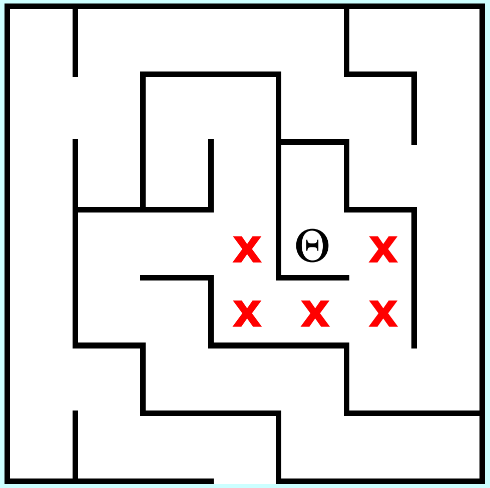


- How does the algorithm keep track of the "big picture" of what paths it still needs to explore?

# Each Frame Remembers One Choice

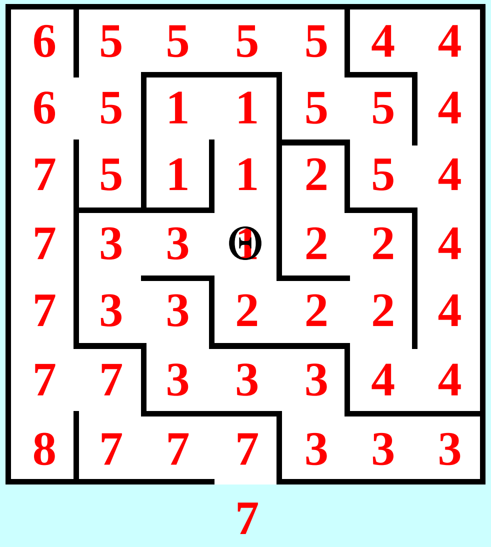
```
bool solveMaze(Maze & maze, Point start) {
 bool solveMaze(Maze & maze, Point start) {
  bool solveMaze(Maze & maze, Point start) {
   bool solveMaze(Maze & maze, Point start) {
    bool solveMaze(Maze & maze, Point start) {
     bool solveMaze(Maze & maze, Point start) {
        if (maze.isOutside(start)) 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) | |



# Recursion and Concurrency

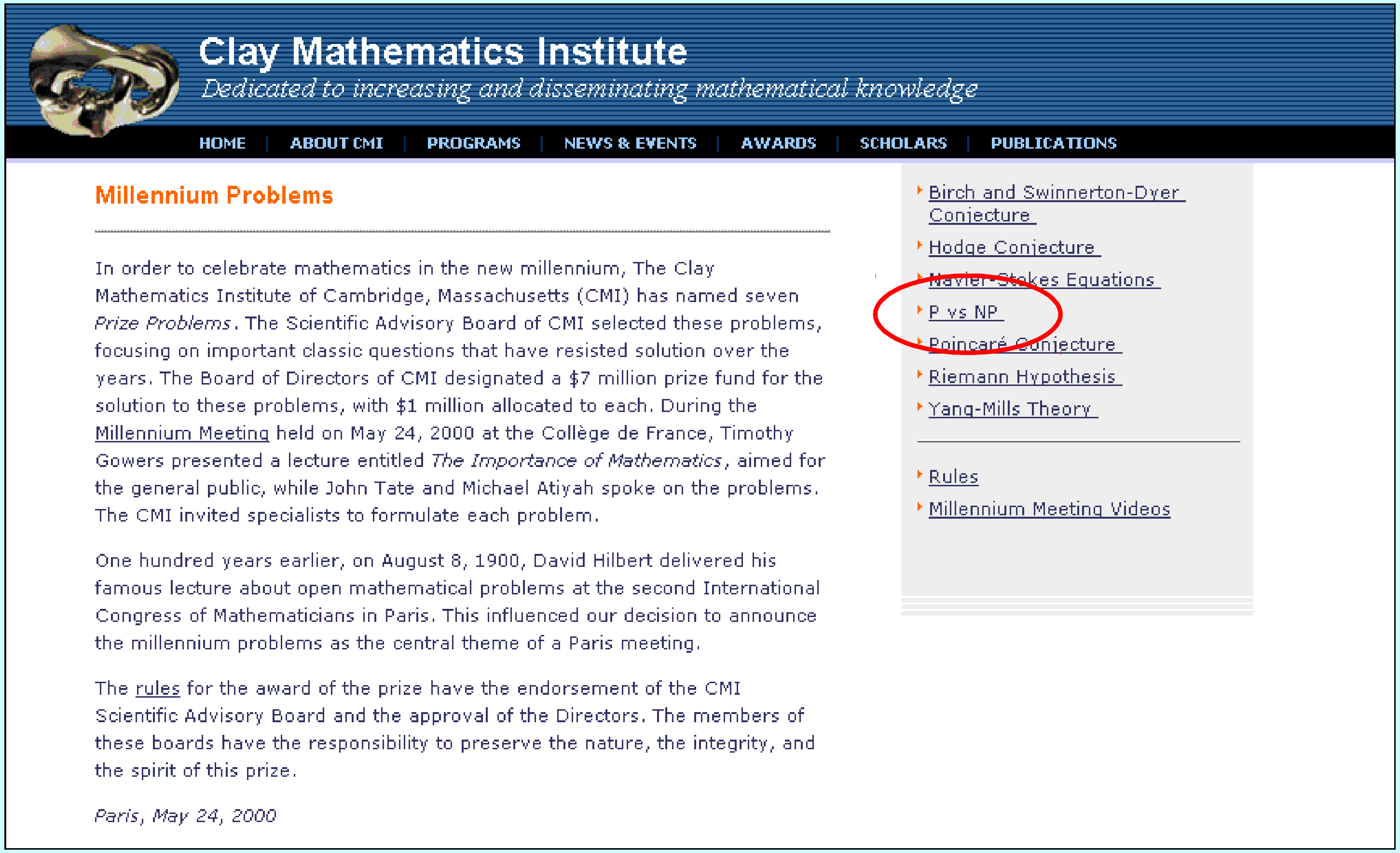
- The recursive decomposition of a maze generates a series of independent submazes; the goal is to solve any one of them.
- If you had a multiprocessor computer, you could try to solve each of these submazes in parallel. This strategy is analogous to cloning yourself at each intersection and sending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

# Exercise: Keeping Track of the Path

- As described in exercise 3 on page 418, 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.

The End