CS61B, Spring 2003 **Discussion #9** **Amir Kamil**
UC Berkeley **3/17/03**

Topics: Backtracking

1 Backtracking

Consider the maze in figure 1. You probably solved it immediately, but can you describe the algorithm you use to solve it? On a larger maze, the actual algorithm you use would be cleare, and is called *backtracking*. This is where you try a path until you get stuck, then retrace your steps until you can follow another path, and repeat. Eventually, you will either find a path that leads to the destination, or try all paths and decide that no solution exists. Figure 2 shows an example use of backtracking on a simpler maze.

In order to describe the backtracking algorithm in detail, let's divide each path into a series of steps. A step consists of moving one unit in a certain direction, in this case up, down, left, or right. Then the algorithm would be:

```
while not at destination do:
  if there is a legal direction to move in such that you haven't been to the cell in that
direction then:
    Move in that direction.
  else:
    Retrace your steps until such a direction exists.
  fi
od
```

The problem here, is how would you know which steps you need to retrace? There are two solutions to this problem, to store all the steps that you have taken, in order, or to use recursion (which implicitly stores all steps). Recursion is the easier solution, so let's rewrite the algorithm using recursion, and a little more formally. We will use a `Maze` object that stores the maze and also keeps track of our current position.

```
boolean solve(Maze m):
  if m.isSolved() then:
    return true;
  fi;
  for each move dir (up, down, left, and right) do:
    m.move(dir);
    if solve(m) then:
      return true;
    fi;
```

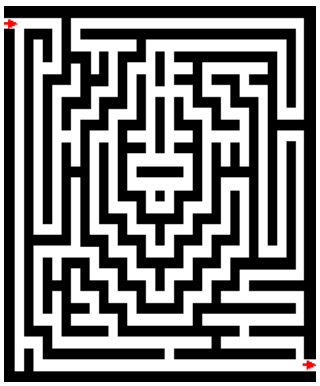

Figure 1: A maze.

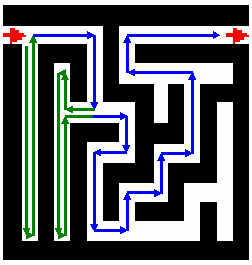

Figure 2: An example of backtracking on a maze. The backtracking is shown in green.

```
    m.unmove(dir);
  od;
  return false;
```

There are a couple of things to notice about this algorithm. The first is that it is necessary to be able to undo moves. Moves are only undone after all subsequent moves have been tried, and no solution has been found. If you trace out the execution of this arrangement, you will see that the result is retracing your steps until a new move can be made. The second is that the algorithm only determines if a solution exists, and does not give the actual path. This can be remedied using the following observation. If the recursive call `solve(m)` returns true, this means that the move that we just did leads to a solution. So we output that move. There is a small problem here, though. The resulting moves are printed in reverse order, since they are printed in the order in which the recursive calls return. So to resolve this problem, we'll store the moves in a stack and print them at the end, instead of immediately outputting them. The result is in figure 3.

Depending on the order in which directions are explored, this backtracking algorithm may result in a longer than necessary solution. Figure 4 shows the solution given by the above algorithm applied to figure 1 using the order *down*, *up*, *right*, *left*. As you can see, this does not result in the shortest solution.

```
void solve(Maze m):
  Create a new Stack s;
  if solveHelper(m, s) then:
    while s is not empty do:
      output s.pop();
    od;
  else:
    output "No solution exists.";
  fi;

boolean solveHelper(Maze m, Stack s):
  if m.isSolved() then:
    return true;
  fi;
  for each move dir (up, down, left, and right) do:
    m.move(dir);
    if solve(m) then:
      s.push(dir);
      return true;
    fi;
    m.unmove(dir);
  od;
  return false;
```

Figure 3: A recursive backtracking algorithm for solving mazes.

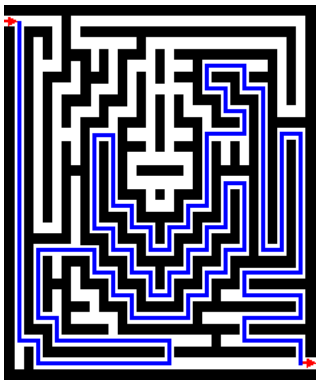

Figure 4: A possible solution to a maze using backtracking.

The execution of the algorithm can be thought of as forming a *tree*. The original maze would be at the root of the tree, and its children would be each maze that results from a single legal move from the original maze. Similarly, their children would be each maze that results from a single legal move from the parent maze. At the leaves would be mazes that either are solved or that have reached a deadend, since in either case no more moves would be made. Figure 5 shows part of the tree for this maze. These leaves would not necessarily be at the same height. For example, the leftmost leaf shown in the figure would be at a depth of 10, and the rightmost at a depth of 44. The depth corresponds to the number of moves it takes to get to that leaf.

The algorithm we wrote traverses this tree *depth first*, meaning that it would follow one branch of the tree all the way to the bottom before trying the other branches. This is why the computed solution is not the shortest. If it happens that the first branch results in a solution at depth 100, but the second results in a solution at depth 10, the algorithm would find the first solution because it does a depth first traversal.

Alternatively, we could do a *breadth first traversal*. In such a treversal, the tree is traversed level by level. In other words, all nodes in the tree at depth 1 are checked first, then all nodes at depth 2, and so on. This will result in a solution of shortest length, since the solution at the least depth is found first. Unfortunately, it is practically impossible to write a recursive breadth first traversal. It must be implemented using iteration and a queue.

The backtracking algorithm can work on all single-player games in which the solution consists of a sequence of moves, with only minor modifications. There is, however, a drawback to the algorithm. Because of the tree structure that the recursive backtracking calls produce, the algorithm can potentially take exponential time to run. This can be avoided using heuristics, but in the worst case it would either still be exponential or have a possibility of not finding a solution where one exists.

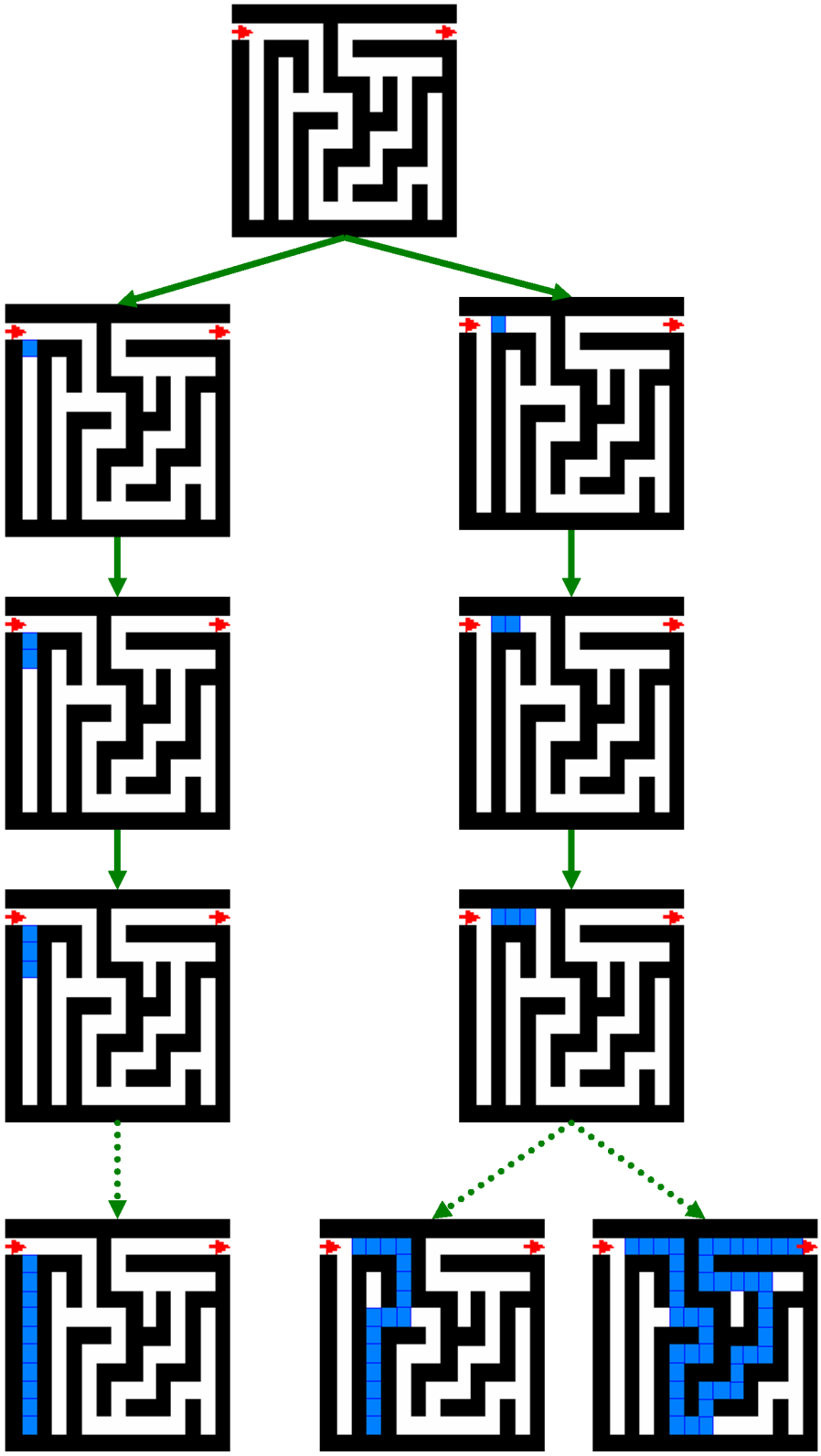

Figure 5: Part of the implicit tree formed in running the backtracking algorithm.