Programming Abstraction in C++

Eric S. Roberts and Julie Zelenski

Stanford University

2010

Chapter 7. Backtracking Algorithms

Outline

Outline

Introduction

Backtracking problem:

While solving a problem, you make decisions along the way. Reaching a dead end, you want to backtrack to an early point and try an alternative choice.

Examples: Sudoku, maze

Introduction

Backtracking problem:

While solving a problem, you make decisions along the way. Reaching a dead end, you want to backtrack to an early point and try an alternative choice.

Examples: Sudoku, maze

Recursive insight:

A backtracking problem has a solution if and only if at least one of the smaller backtracking problems that results from making each possible initial choice has a solution.

Outline

Maze problem

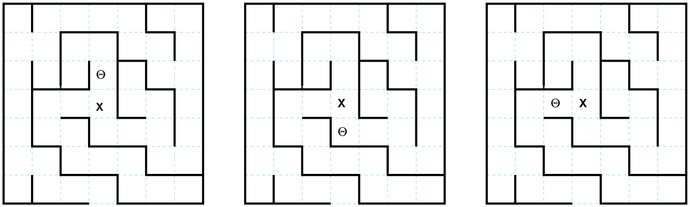
An iterative solution:

The right-hand rule:

```
Put your right-hand against a wall.
while (you have not escaped from the maze) {
    Walk forward keeping your right hand on a wall.
}
```

Maze problem (cont.)

Recursive decomposition: three subproblems



Maze problem (cont.)

Stopping points (simple cases):

1. The current square is outside the maze, the maze is solved.
2. The current square is marked, the maze is unsolvable.

Maze Problem (cont.)

```
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;
}
```

Maze Problem (cont.)

```
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;
}
```

Question: Why `UnmarkSquare`?

Maze problem (cont.)

The `mazelib.h` interface provides an appropriate data structure for the maze. An abstraction layer for the main program to access the information it needs to solve the maze problem. For example, where the walls are, whether a square is marked, if the current square is outside the maze. (Figure 7-2, pp. 240-241)

Maze problem (cont.)

The `mazelib.h` interface provides an appropriate data structure for the maze. An abstraction layer for the main program to access the information it needs to solve the maze problem. For example, where the walls are, whether a square is marked, if the current square is outside the maze. (Figure 7-2, pp. 240-241)

```
int main() {
    ReadMazeMap(MazeFile);
    if (SolveMaze(GetStarPosition())) {
        cout << "The marked squares show a solution path." << endl;
    } else {
        cout << "No solution exists." << endl;
    }
    return 0;
}
```

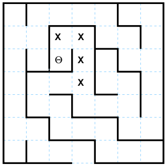
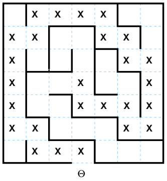
Maze problem (cont.)

Not convinced?

Maze problem (cont.)

Not convinced?

```
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;
}
```

Outline

Two-player games

Examples: tic-tac-toe, chess

Two-player games

Examples: tic-tac-toe, chess

The minimax strategy:

Finding the position (state) that leaves your opponent with the worst possible best move, that is, the move that minimizes your opponent's maximum opportunity.

Two-player games

Examples: tic-tac-toe, chess

The minimax strategy:

Finding the position (state) that leaves your opponent with the worst possible best move, that is, the move that minimizes your opponent's maximum opportunity.

Question: Why not maxmize my opportunity?

Two-player games

Find a good move:

```
for (each possible move) {
    Evaluate the position that results from making the move.
    If the resulting position is bad, return the move.
}
Return a sentinel value indicating that no good move exists.
```

Game of nim

The game begins with a pile of $n = 13$ coins.

1. On each turn, players take either one, two, or three coins from the pile and put them aside.
2. The one forces the opponent to take the last coin wins.

Game of nim

The game begins with a pile of $n = 13$ coins.

1. On each turn, players take either one, two, or three coins from the pile and put them aside.
2. The one forces the opponent to take the last coin wins.

Bad position

Your turn and there is only one coin on the pile.

Good position

Your turn and there are two, three, or four coins on the pile, because you can find a move so that the position that results from the move is bad.

Game of nim (cont.)

If you find yourself with five coins on the pile, you are in a bad position. Why?

Game of nim (cont.)

If you find yourself with five coins on the pile, you are in a bad position. Why?

Because all your three possible moves will result a good position. In other words, you can’t find a good move, so that the position that results from the move is bad.

Game of nim (cont.)

If you find yourself with five coins on the pile, you are in a bad position. Why?

Because all your three possible moves will result a good position. In other words, you can't find a good move, so that the position that results from the move is bad.

See a mutual recursion?

A mutual recursion

```
int FindGoodMove(int nCoins) {
   for (int nTaken = 1; nTaken <= MAX_MOVE; nTaken++) {
      if (IsBadPosition(nCoins - nTaken)) return nTaken;
   }
   return NO_GOOD_MOVE;
}
```

```
bool IsBadPosition(int nCoins) {
    if (nCoins == 1) return true;
    return (FindGoodMove(nCoins) == NO_GOOD_MOVE);
}
```

Game tree

In general, to minimize your opponent’s maximum opportunity, we need a quantitative messurement.

Game tree

In general, to minimize your opponent's maximum opportunity, we need a quantitative messurement.

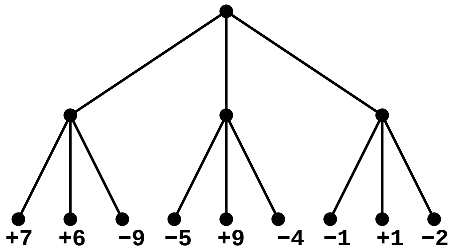
Game tree

- Each node represents a position. The root represents the initial position.
- Each branch (edge) represents a move.
- Each leaf node is assigned a numerical score.

Find a move to minimize your opponent's highest score.

Example

A two-level game tree from your point-of-view.



Following the minimax strategy, what is your best initial move?

Implementing the minimax algorithm

To make the implementation general

- It must be possible to limit the depth of the recursive search. For many games, such as chess, it is prohibitively expensive to search the entire game tree.
- It must be possible to assign ratings to moves and positions. So we have a quantitative measurement for comparing moves.

Implementing the minimax algorithm (cont.)

```
moveT FindBestMove(stateT state, int depth, int & rating) {
    Vector<moveT> moveList;
    GenerateMoveList(state, moveList);
    int nMoves = moveList.size();
    if (nMoves == 0) Error("No moves available");
    moveT bestMove;
    int minRating = WINNING_POSITION + 1;
    for (int i = 0; i < nMoves && minRating != LOSING_POSITION; i++) {
        moveT move = moveList[i];
        MakeMove(state, move);
        int curRating = EvaluatePosition(state, depth + 1);
        if (curRating < minRating) {
            bestMove = move;
            minRating = curRating;
        }
        RetractMove(state, move);
    }
    rating = -minRating;
    return bestMove;
}
```

Implementing minimax algorithm (cont.)

```
int EvaluatePosition( stateT state, int depth) {
    int rating;
    if (GameIsOver(state) || depth >= MAX_DEPTH) {
        return EvaluateStaticPosition(state);
    }
    FindBestMove(state, depth, rating);
    return rating;
}
```

Implementing minimax algorithm (cont.)

```
int EvaluatePosition( stateT state, int depth) {
    int rating;
    if (GameIsOver(state) || depth >= MAX_DEPTH) {
        return EvaluateStaticPosition(state);
    }
    FindBestMove(state, depth, rating);
    return rating;
}
```

Data structures not game specific, to be defined by a particular game:

`stateT` and `moveT`

Example. Tic-tac-toe game

Tic-tac-toe game. Figure 7-6, pp. 258-268.

```
struct stateT {
    Grid<char> board;
    playerT whoseTurn;
    int turnsTaken;
}
```

`board`: 3-by-3 grid of `char`, `"X"`, `"O"`, or `" "`
`whoseTurn`: `Human` or `Computer` (enumerated type)
`turnsTaken`: count for the number of occupied squares

Example. Tic-tac-toe game

Tic-tac-toe game. Figure 7-6, pp. 258-268.

```
struct stateT {
    Grid<char> board;
    playerT whoseTurn;
    int turnsTaken;
}
```

`board`: 3-by-3 grid of `char`, `"X"`, `"O"`, or `"  "`
`whoseTurn`: `Human` or `Computer` (enumerated type)
`turnsTaken`: count for the number of occupied squares

```
typedef int moveT;
```

represents the nine squares.