Sudoku

Jordi Cortadella
Department of Computer Science

Solving a Sudoku

Input

\[
\begin{array}{ccc}
5 & 3 & 7 \\
6 & 1 & 9 & 5 \\
9 & 8 & 6 \\
6 & 8 & 6 & 3 \\
4 & 8 & 3 & 1 \\
2 & 2 & 6 \\
6 & 2 & 8 \\
4 & 1 & 9 & 5 \\
8 & 7 & 9 \\
\end{array}
\]

• Is it correct (conflict free)?
• Is it complete (no empty cells)?
• If incomplete, can we find a solution?

Sudoku: data structures

// A 9x9 matrix to store values.
// The matrix contains values in \{0,\ldots,9\},
// where 0 means “empty”.

```
typedef vector< vector<int> > Grid;
```

Detection of conflicts

Similar matrices can be used for column and square conflicts

```
// A 9x10 Boolean matrix to indicate used values.
// The 0-column is not used.

typedef vector< vector<bool> > Used;
```
Sudoku: data structures

```cpp
struct Sudoku {
    Grid G;       // the main grid
    Used Rows;    // conflicts for rows
    Used Columns; // conflicts for columns
    Used Squares; // conflicts for squares
};
```

// Post: The Sudoku initialized as empty.
void initSudoku(Sudoku& S) {
    S.G = Grid(9, vector<int>(9, 0));
    S.Rows = Used(9, vector<bool>(10, false));
    S.Columns = Used(9, vector<bool>(10, false));
    S.Squares = Used(9, vector<bool>(10, false));
}

Sudoku S;

row \( r \), column \( c \), value \( v \)

// Pre: \( S \) is a Sudoku partially filled in.
// Post: Cell \((r, c)\) is filled in with value \( v \) if no conflict is produced. The Sudoku is not changed in case of conflict.
// Returns true if no conflict, or false otherwise.
bool writeCell(Sudoku& S, int r, int c, int v) {
    int sq = 3*(r/3) + c/3;
    if (S.Rows[r][v] or S.Columns[c][v] or S.Squares[sq][v]) return false;
    S.G[r][c] = v;
    return true;
}
```

Sudoku: indexing the Used matrices

For row conflicts \( \rightarrow S\.Rows[r][v] \)

For column conflicts \( \rightarrow S\.Columns[c][v] \)

For square conflicts \( \rightarrow S\.Squares[3\cdot(r/3) + c/3][v] \)

---

Reading a Sudoku

```
5 3 7 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

```
5 3 7 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

```
5 3 7 1 9 5 6 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

```
5 3 7 1 9 5 6 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

---

```
5 3 7 1 9 5 6 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

```
5 3 7 1 9 5 6 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

```
5 3 7 1 9 5 6 1 9 5 9 8 6 1 4 8 6 3 7 2 6 6 2 8 2 8 5 4 1 5 8 7 9
```

---

```
5 3 0 7 0 0 0 0 0
6 0 0 1 9 5 0 0 0
0 9 8 0 0 0 0 6 0
8 0 0 0 6 0 0 3 0
4 0 0 8 0 3 0 1 0
7 0 0 0 2 0 0 6 0
0 6 0 0 0 2 8 0 0
0 0 0 4 1 9 0 0 5
0 0 0 8 0 0 7 9
```
Reading a Sudoku

// Pre: the input has 81 digits in {'0',...,'9'}
// Post: The Sudoku S has been read from cin.
// Returns true if the Sudoku is correct, or false otherwise.
bool readSudoku(Sudoku& S) {
    initSudoku(S); // empty Sudoku
    for (int r = 0; r < 9; ++r) { // Read all rows and columns
        for (int c = 0; c < 9; ++c) {
            char digit;
            cin >> digit;
            int n = int(digit - '0'); // Convert to int
            if (n != 0 and not writeCell(S, r, c, n)) return false;
        }
    }
    return true; // Correct sudoku
}

Is the Sudoku complete?

// Pre: S is a correct Sudoku.
// Returns true if the Sudoku is complete (no empty cells),
// or false otherwise.
bool completeSudoku(const Sudoku& S) {
    for (int r = 0; r < 9; ++r) {
        for (int c = 0; c < 9; ++c) {
            if (S.G[r][c] == 0) return false; // Empty cell
        }
    }
    return true; // Complete Sudoku
}

Main program

// The program reads a Sudoku and reports
// whether it is correct and complete.
int main() {
    Sudoku S;
    if (not readSudoku(S)) {
        cout << “The Sudoku is incorrect.” << endl;
        return 1;
    }
    if (not completeSudoku(S)) {
        cout << “The Sudoku is not complete.” << endl;
        return 1;
    }
    cout << “The Sudoku is complete and correct.” << endl;
}
Main program

// The program reads a Sudoku and tries to solve it.
// In case it is solvable, it writes a solution.

int main() {
    Sudoku S;
    if (not readSudoku(S)) {
        cout << “The Sudoku is incorrect.” << endl;
        return 1;
    }

    if (not solveSudoku(S)) {
        cout << “The Sudoku has no valid solution.” << endl;
        return 1;
    }

    writeSudoku(S);
}

Write a Sudoku

// Pre: S is a complete Sudoku.
// Post: The Sudoku has been printed into cout.

void writeSudoku(const Sudoku& S) {
    for (int r = 0; r < 9; ++r) {
        for (int c = 0; c < 9; ++c) cout << S.G[r][c];
        cout << endl;
    }
}

Number of Sudokus

6,670,903,752,021,072,936,960 \approx 6.67 \times 10^{21}

Solving a Sudoku

Doing progress from a partially filled Sudoku
Solving a Sudoku

Doing progress from a partially filled Sudoku

Erasing a cell

// Pre: The Sudoku S has a value in cell (r,c).
// Post: Cell (r,c) has been erased.

void eraseCell(Sudoku& S, int r, int c) {
    int v = S.G[r][c];  // Gets the value
    S.G[r][c] = 0;  // Erases the value
    // Cleans the conflict matrices for the value
    int sq = 3*(r/3) + c/3;
}

Recursive Sudoku

// Pre: S has been filled in up to cell (r,c) without conflicts,
without including cell (r,c).
// Returns true if the Sudoku is solvable with the
pre-filled cells, or false otherwise.
// Post: S contains a solution if the Sudoku is solvable.
// S is not modified if the Sudoku is unsolvable.

bool solveSudokuRec(Sudoku& S, int r, int c);

Recursive Sudoku

Basic case: (r=9, c=0)

– The Sudoku has been completed without conflicts.
A solution has been found! We are done (return true).

Cells are filled in by rows, starting from cell (0,0).
Recursive Sudoku

Simple case: cell is not empty
– Don’t touch and solve from the next cell

```
4 4
```

Recursive Sudoku

Difficult case: cell is empty
– Write 1 in \((r,c)\) and check for conflicts.
  If no conflict, solve from next cell. If successful, done! (return true)
– else, erase 1, write 2 in \((r,c)\) and check for conflicts.
  If no conflict, solve from next cell. If successful, done! (return true)
– ...
– else, erase 8, write 9 in \((r,c)\) and check for conflicts.
  If no conflict, solve from next cell. If successful, done! (return true)
– else, failure (return false).

```
? ?
```

Recursive Sudoku

```cpp
bool solveSudokuRec(Sudoku& S, int r, int c) {
  if (r == 9) return true; // Yupee! (Sudoku completed)

  int next_r = r + c/8;   // Next row (increase when c=8)
  int next_c = (c + 1)%9; // Next column (0 if c+1=9)

  // If cell is not empty, don’t touch and go to next cell
  if (S.G[r][c] != 0) return solveSudokuRec(S, next_r, next_c);

  // Try all possible values from 1 to 9
  for (int v = 1; v <= 9; ++v) {
    if (writeCell(S, r, c, v)) {
      if (solveSudokuRec(S, next_r, next_c)) return true; // Yupee!
      eraseCell(S, r, c); // Backtrack
    }
  }
  return false;
}
```

```
int main() {
  Sudoku S;
  if (not readSudoku(S)) {
    cout << “The Sudoku is incorrect.” << endl;
    return 1;
  }
  if (not solveSudoku(S)) {
    cout << “The Sudoku cannot be solved.” << endl;
    return 1;
  }
  writeSudoku(S);
}
```

Back to main program

// Pre: S is a correct and possibly incomplete Sudoku.
// Returns true if the Sudoku is solvable, or false otherwise.
// Post: If solvable, the S contains a solution.
bool solveSudoku(Sudoku& S) {
  return solveSudokuRec(S, 0, 0);
}
The 8 queens puzzle

Place eight queens on an 8×8 chessboard so that no two queens threaten each other.

Generalization to n queens

n=6

n=13

n=25

Source: http://www.7sudoku.com/very-difficult

Try from here
Solving a maze

Summary

• **Backtracking** is a common technique used to solve constraint satisfaction problems.

• Strategy: build partial solutions and backtrack when some constraint is not satisfied.

• Backtracking avoids the exhaustive enumeration of all possible solutions.