CSE 537 Assignment 3 Report: Sudoku

Remy Oukaour
SBU ID: 107122849
remy.oukaour@gmail.com

Jian Yang
SBU ID: 110168771
jian.yang.1@stonybrook.edu

Sunday, October 25, 2015

1 Introduction

This report describes our submission for assignment 3 in the CSE 537 course on artificial intelligence. Assignment 3 requires us to implement some different algorithms to solve Sudoku games: variations on backtracking (simple, MRV, MRV with forwarding checking, and MRV with constraint propagation), and min-conflicts. In this report, we discuss the implementation details and performance of our solutions.

2 Implementation

2.1 Data structures

We implemented one class named *gameNode* in *gameNode.py* to contain the state of a Sudoku board in the search tree used by our algorithms.

The *gameNode* data structure supports operations such as setting a value, rolling back a value, and checking if the board is solved. (The *solved* method for performing that check also increments the node's num_checks counter.) It also supports the constraint-related operations for different algorithms.

Two important methods of the *gameNode* class are *get_neighbors* and *get_valid_moves*:

```
def get_neighbors(self, pos):
    i, j = pos
    bi = i // self.M * self.M
    bj = j // self.K * self.K
    neighbors = set()
    for k in xrange(self.N):
        neighbors.add((i, k))
        neighbors.add((k, j))
        neighbors.add((bi + k // self.K, bj + k % self.K))
    neighbors.remove(pos)
    return list(neighbors)
```

```
def get_valid_moves(self, pos):
    if self[pos].value():
        return []
    invalid_moves = {self[n].value() for n in self.get_neighbors(pos)}
    return [m for m in self.get_values() if m not in invalid_moves]
```

2.2 Backtracking

For backtracking, the algorithm chooses one arbitrary unassigned position and uses the edge constraints to get valid values. It traverses all these valid values, and using a backtracking depth-first search, it tries to find a solution if one exists.

The implementation of simple backtracking search in *csp.py* is:

```
def backtracking(filename):
    node = gameNode.gameNode()
    if not node.load_game(filename):
        return ("Error: Fail Load", 0)
    if backtracking_helper(node):
        return node.solution()
    return ("Error: No Solution", node.num_checks)

def backtracking_helper(node):
    if node.solved():
        return True
    unassigned = node.get_unassigned_positions()
    if not unassigned:
        return False
    pos = unassigned[0]
    for move in node.get_valid_moves(pos):
        domain = node[pos].domain
        node[pos] = move
        if backtracking_helper(node):
            return True
        node[pos] = domain
    return False
```

2.3 Backtracking + MRV

We added the *count_contraints* method to the *gameNode* class:

```
def count_constraints(self, pos, value=None):
    if value is None:
        value = self[pos].value()
    return len([n for n in self.get_neighbors(pos)
        if not self[n].value() and value in self.get_valid_moves(n)])
```

The implementation of backtracking + MRV in *csp.py* is:

```
def backtrackingMRV(filename):
    node = gameNode.gameNode()
    if not node.load_game(filename):
        return ("Error: Fail Load", 0)
    if backtrackingMRV_helper(node):
        return node.solution()
    return ("Error: No Solution", node.num_checks)

def backtrackingMRV_helper(node):
    # Based on Figure 6.5 from page 215 of the textbook
    if node.solved():
        return True
    unassigned = node.get_unassigned_positions()
    if not unassigned:
```

```
        return False
    mrv_pos = min(unassigned, key=lambda p: len(node.get_valid_moves(p)))
    lcv_moves = sorted(node.get_valid_moves(mrv_pos),
        key=lambda m: node.count_constraints(mrv_pos, m))
    for move in lcv_moves:
        domain = node[mrv_pos].domain
        node[mrv_pos] = move
        if backtrackingMRV_helper(node):
            return True
        node[mrv_pos] = domain
    return False
```

2.4 Backtracking + MRV + FC

We added the *forward_checking* method to the *gameNode* class:

```
def forward_checking(self, pos):
    # Use AC-3 with limited queue
    queue = [(n, pos) for n in self.get_neighbors(pos)
        if not self[n].value()]
    return self._ac3(queue)

def _ac3(self, queue):
    # Based on Figure 6.3 from page 209 of the textbook
    while queue:
        pos1, pos2 = queue.pop()
        if self[pos2].value() in self[pos1].domain:
            self[pos1].domain -= self[pos2].domain
            if not self[pos1].domain:
                return False
            for pos3 in set(self.get_neighbors(pos1)) - {pos2}:
                queue.append((pos3, pos1))
    return True
```

The implementation of backtracking + MRV + forward checking in *csp.py* is:

```
def backtrackingMRVfwd(filename):
    node = gameNode.gameNode()
    if not node.load_game(filename):
        return ("Error: Fail Load", 0)
    if backtrackingMRVfwd_helper(node):
        return node.solution()
    return ("Error: No Solution", node.num_checks)

def backtrackingMRVfwd_helper(node):
    # Based on Figure 6.5 from page 215 of the textbook
    if node.solved():
        return True
    unassigned = node.get_unassigned_positions()
    if not unassigned:
        return False
    mrv_pos = min(unassigned, key=lambda p: len(node.get_valid_moves(p)))
    lcv_moves = sorted(node.get_valid_moves(mrv_pos),
        key=lambda m: node.count_constraints(mrv_pos, m))
    for move in lcv_moves:
```

```
        backup_board = copy.deepcopy(node.board)
        node[mrv_pos] = move
        if (node.forward_checking(mrv_pos) and
            backtrackingMRVfwd_helper(node)):
            return True
        node.board = backup_board
    return False
```

2.5 Backtracking + MRV + CP

We added the *propagate_constraints* method to the *gameNode* class:

```
def propagate_constraints(self, updated_pos):
    # Use AC-3 with all constrained pairs
    queue = []
    pos1 = updated_pos
    for pos2 in self.get_neighbors(pos1):
        queue.append((pos1, pos2))
    return self._ac3(queue)

def _ac3(self, queue):
    # Based on Figure 6.3 from page 209 of the textbook
    while queue:
        new_queue = []
        for q in queue:
            pos1, pos2 = q
            if self[pos1].value() in self[pos2].domain:
                self[pos2].domain.remove(self[pos1].value())
                if not self[pos2].domain:
                    return False
                for pos3 in set(self.get_neighbors(pos2)):
                    if pos3 != pos1:
                        new_queue.append((pos2, pos3))
        queue = new_queue
    return True
```

The implementation of backtracking + MRV + constraint propagation in *csp.py* is:

```
def backtrackingMRVcp(filename):
    node = gameNode.gameNode()
    if not node.load_game(filename):
        return ("Error: Fail Load", 0)
    if backtrackingMRVcp_helper(node):
        return node.solution()
    return ("Error: No Solution", node.num_checks)

def backtrackingMRVcp_helper(node):
    # Based on Figure 6.5 from page 215 of the textbook
    if node.solved():
        return True
    unassigned = node.get_unassigned_positions()
    if not unassigned:
        return False
    mrv_pos = min(unassigned, key=lambda p: len(node.get_valid_moves(p)))
    lcv_moves = sorted(node.get_valid_moves(mrv_pos),
```

```
        key=lambda m: node.count_constraints(mrv_pos, m))
    for move in lcv_moves:
        backup_board = copy.deepcopy(node.board)
        node[mrv_pos] = move
        if node.propagate_constraints() and backtrackingMRVcp_helper(node):
            return True
        node.board = backup_board
    return False
```

2.6 Min-conflicts

We added the *count_conflicts* method to the *gameNode* class:

```
def count_conflicts(self, pos, value=None):
    if value is None:
        value = self[pos].value()
    return len([n for n in self.get_neighbors(pos)
        if self[n].value() == value])
```

The implementation of min-conflicts in *csp.py* is:

```
def minConflict(filename):
    node = gameNode.gameNode()
    if not node.load_game(filename):
        return ("Error: Fail Load", 0)
    if minConflict_helper(node):
        return node.solution()
    return ("Error: No Solution", node.num_checks)

def minConflict_helper(node, max_steps=10000):
    # Based on Figure 6.8 from page 221 of the textbook
    # initial complete assignment
    # greedy minimal-conflict values for each variable
    for pos in node.get_positions():
        if not node[pos].given:
            node[pos] = min(node.get_values(),
                key=lambda m: node.count_conflicts(pos, m))
    for _ in xrange(max_steps):
        if node.solved():
            return True
        con_pos = random.choice(node.get_conflicted_positions())
        lcv_move = min(node.get_values(),
            key=lambda m: node.count_conflicts(con_pos, m))
        node[con_pos] = lcv_move
    return False
```

3 Results

3.1 Test cases

3.1.1 Empty Input

As a given input in the assignment, we use an empty board to test our algorithms. The result is shown in table 1.

3.1.2 Manual Cases

We added several manual test cases (including the ones provided by Caleb on Piazza) to compare the results. And one of them has conflict input, we added one conflict check at the beginning of all methods based on that.

3.2 Benchmarks

	BT	BT+MRV	BT+MRV+FC	BT+MRV+CP	Min-conflicts
Empty (9×9)	392	82	49	49	>10K
Empty(12×12)	7426	145	95	95	40
board1	210	77	74	75	>10K
board2	45804	2480	2458	234	>10K
board3	N/A	248	244	59	>10K
board4	1	1	1	1	1

Table 1: Expanded node counts for all algorithms in different test cases

4 Conclusion

4.1 Optimizations in backtracking

In the test result, we could see that with different extra optimization methods to choose the next backtracking node worked much better than the simple backtracking with arbitrary node sequence. The methods really chose "better" nodes to expand in the progress as we expected and we learnt from the lectures.

4.2 Hanging

The min-conflicts algorithm sometimes does not solve a board, even with a high iteration limit. We suspect that this is because it uses random choices to solve the board, and sometimes those choices are not useful ones.

4.3 Best algorithm

We recommend using backtracking search with the MRV heuristic and constraint propagation. This algorithm made the fewest consistency checks in our testing. MRV + forward checking is the second best one. Min-heuristics sometimes made even fewer consistency checks than BT+MRV+FC, but other times it would fail to solve the board.