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.

This report describes our submission for assignment 3 in the CSE 537 course on arti

cial intelligence. Assignment 3 requires us to implement sudoku solvers with different approaches. The approaches include backtracking, backtracking with MRV, backtracking with MRV and forwarding checking, backtracking with MRV and constraint propagation, and minConflict. 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 status of sudoku board when we searched solutions with approaches.
This board data structure supports the operations on the board such as put a value, rollback a value and check if the board is at a solution state, and also supports the constraints related operations for different approaches.

2.2 backtracking

For backtracking, the algorithm would choose one arbitrary unassigned position and used the edge constraints to get valid values. With traverse all these valid values and with a depth first search with backtracking, it tries to find a solution if there is one.
The code in gameNode.py is:

```
    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]
```

And the code to use the above method to search with backtracking method is:

```
def backtracking(filename):
    ###
    # use backtracking to solve sudoku puzzle here,
    # return the solution in the form of list of
    # list as describe in the PDF with # of consistency
    # checks done
    ###
    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", 0)

def backtracking_helper(node):
    node.num_checks += 1
    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 function count_contraints for MRV approach.

```
    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)])
```

And the code for backtracking + MRV is:

```
def backtrackingMRV(filename):
    ###
    # use backtracking + MRV to solve sudoku puzzle here,
    # return the solution in the form of list of
    # list as describe in the PDF with # of consistency
    # checks done
    ###
```

```
    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", 0)

def backtrackingMRV_helper(node):
    node.num_checks += 1
    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 + fwd

We added the function forward_checking in gameNode for forward checking in this approach.

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

And the code for backtracking + MRV + fwd is:

```
def backtrackingMRVfwd(filename):
    ###
    # use backtracking +MRV + forward propogation
    # to solve sudoku puzzle here,
    # return the solution in the form of list of
    # list as describe in the PDF with # of consistency
    # checks done
    ###
    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", 0)

def backtrackingMRVfwd_helper(node):
    node.num_checks += 1
    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 function propagate_constraints in class gameNode for the constraint propagation purpose in this approach.

```
    def propagate_constraints(self):
        # AC-3
        queue = [(pos1, pos2) for pos1, pos2 in
            product(self.get_positions(), self.get_positions()) if pos1 != pos2]
        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
```

And the code of backtracking + MRV + CP is:

```
def backtrackingMRVcp(filename):
    ###
    # use backtracking + MRV + cp to solve sudoku puzzle here,
    # return the solution in the form of list of
    # list as describe in the PDF with # of consistency
    # checks done
    ###
    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", 0)

def backtrackingMRVcp_helper(node):
    node.num_checks += 1
    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 backtrackingMRV_helper(node):
            return True
        node.board = backup_board
    return False
```

2.6 minConflict

We added count_conflicts in class gameNode to get the minimum conflict counts and used it later in the minConflict method.

```
    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 code of minConflict is:

```
def minConflict(filename):
    ###
    # use minConflict to solve sudoku puzzle here,
    # return the solution in the form of list of
    # list as describe in the PDF with # of consistency
    # checks done
    ###
    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", 0)

def minConflict_helper(node, max_steps=3000):
    # 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):
        node.num_checks += 1
        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 showed in table **??**.

3.1.2 Manual Cases

We added several manual cases to compare the results.

3.2 Benchmarks

	bt	bt + MRV	bt + MRV + fwd	bt + MRV + cp	minConflict
Empty	392	82	49	82	NA
Empty(3×4)	7426	145	95	145	40
Case1	210	77	74	75	1
Case2	46019	2482	2458	1	1
Case3	0	1	1	1	1
Case4	0	1	1	1	1
Case5	0	1	1	1	1

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" node to expand in the progress as we expected and we learnt from the lectures.

4.2 Hanging