ARTIFICIAL INTELLIGENCE

ASSIGNMENT-1

RUPINDER GHOTRA

20460938

Question 1 (part -2)

a) What is the time and space complexity of IDA *?

Solution:

Introduction:

The IDA* is actually a hybrid of both Iterative Deepening and A* Algorithm.

Iterative deepening A* (IDA*) is an optimal search algorithm with the performance properties of **A*—it is complete and optimal**—and the space requirements of **DFS—(essentially) linear in depth**. The main idea of iterative deepening A*is to repeatedly search in depth-first fashion, over sub graphs with f-cost less than α, less than 2α, less than 3α, and so on, until a goal is found, **where α is a lower bound on the cost between nodes and their successors throughout the search space**: i.e., $\alpha \leq c(n, m)$, for all n, m 2 ε $\delta(n)$.

Recall that the space complexity of ID is O(bd), where d is the depth of the goal node. Similarly, the space complexity of IDA* is $O(bC^*/\alpha)$, where C* is the optimal cost. Whereas the Space complexity of A* algorithm is $O(b^d)$, which is much expensive. This is due to that A* keep the record of each and every visited node.

Time Complexity:

Case I :When Step cost is constant:

For calculating the time complexity of IDA* we can consider it as equivalent to ID algorithm as it work in the same fashion if the f- cost is equal. So, the nodes on the bottom level are expanded once, those on the next to bottom level are expanded twice, and so on, up to the root of the search tree, which is expanded d+1 times .So the total number of expansions in an iterative deepening search is **$O(b^d)$.**

Comparison with A*

Where as the Time complexity of A* would surely be dependent on the type of heuristic we are applying. For problems with constant step cost the growth in run time as a function of the optimal solution depth d is analyzed in terms of absolute or relative error of the heuristic. The absolute error is defined as $\Delta = h^* - h$, where h* is the actual cost of getting from root to goal and relative error is defined as
$\varepsilon = (h^* - h)/h^*$.
For constant step cost the time complexity would be $O(b^{\varepsilon d})$,where d is the solution depth.

Now, for almost all the heuristics the absolute error is proportional to h*. Which means that ε is either constant or growing. As suppose h*-h is proportional to h* by a factor of 3. Which result in the value of **ε = (h*-h)/h* = (3h*)/h* = 3.**

So, the time complexity of A* would be O(b^d) as we can ignore ε.

<u>Case II:</u>

The time complexity of IDA*, however, can exceed that of A*.
In particular, in search spaces where the step - cost is different at every state, only one additional state is expanded during each iteration. In such a search space, if A* expands N nodes, IDA_*expands 1+. . .+N = **O(N^2)** nodes. The typical solution to this problem is to fix an increment β> α such that several nodes n have cost $f_i < f(n) \leq f_i + \beta$, where fi is the ith incremental value of the step-cost. This strategy reduces search time, since the total number of iterations is proportional to 1/β< 1/α, and returns solutions that are at worst β-optimal: i.e., if the algorithm returns m*, then g(m*) < C*+ β.

b) Is IDA* complete?

Solution:
<u>Completeness:</u> Completeness of heuristic/algorithm may be defined as if the algorithm guaranteed to find a solution.

Now, If suppose C* is the cost of the optimal solution path, then we can say that:

- IDA* expands all the nodes with f(n) < C*, that's is obvious because no heuristic can over estimate the cost of path ,it will always be certainly less than C*.
- IDA* might expand some nodes right on the GOAL path i.e. f(n) = C* before selecting a goal node.

So**, Completeness clearly requires that there be only finitely many nodes with cost less than or equal to C*,** a condition that will be true if all step costs exceed some finite γ and if b is finite.

Because there could me infinite node within the optimal path cost value and that will put the IDA* into infinite loop. Similarly, if the step comes out to be zero which is practically impossible than also IDA* won't give any results.

Conclusion: IDA* would give the result no matter what!!! But may take a little bit longer.

c) Is IDA* Optimal?

Solution:

OPTIMALITY: Optimality of any heuristic may be defined as answer to the question that does the strategy finds the optimal solution. The path cost function measures solution quality, and an optimal solution has the lowest path cost among all solutions.

So, the next step is to prove that whenever IDA* selects a node n for expansion, the optimal path to that node has been found.

We all know that these heuristics works in the condition that f is non decreasing along any path .

Since Iterative Deepening A* performs a series of **depth-first searches**, its memory requirement is linear with respect to the maximum search depth. In addition, if the heuristic function is admissible, IDA* finds an optimal solution. Finally, by an argument similar to that presented for DFID, IDA* expands the same number of nodes, asymptotically, as A* on a tree, provided that the number of nodes, asymptotically, as A* on a tree, provided that the number of nodes grows exponentially with solution cost. These costs, together with the optimality of A*, imply that IDA* is asymptotically optimal in time and space over all heuristic search algorithms that find optimal solutions on a tree. Additional benefits of IDA* are that it is much easier to implement, and often runs faster than A*, since it does not incur the overhead of managing the open and closed lists.

Conclusion: IDA* would give the optimal solution.

In Summary:

Criteria	IDA*
Time	$O(N^2)$,if step costs differ at all states ,when if A* expands N nodes.
Space	$O(bC^*/\alpha)$,if f is monotonically non-decreasing in depth and if C^*
Completeness	**YES,** if there do not exist ∞- many nodes n such that $f(n) < f^* + \alpha$ is the optimal cost.
β-Optimality	YES, if h is admissible and g is monotonically non decreasing at depth.

Question 1(1):

Solution:

Generalization: As given 8 puzzle problem, here we have branching factor, b = 3;

- When empty tile is in middle = 4 moves
- When empty tile in corner = 2 moves
- When empty tile is on edge = 3 move

So, average branching factor will be 3.

This will certainly lead to 3^{22} exhaustive tree search required to solve this problem. But we can cut short this kind of problem by applying some nice heuristic.

For the heuristic to be admissible it should never over estimate the path cost.

1) Misplaced Tile Heuristic:
   h_1 = the number of misplaced tiles. For the given puzzle two tiles i.e. "5 & 6" are misplaced, so the state would have h_1 = 2.

   As, h_1 = 0 + 0 + 0 + 0 +1 +1 +0 +0 = **2.**

   This heuristic is obviously admissible heuristic because it is clear that any tile that is out of place must be moved at least once.

2) Manhattan Distance:
   H_2 = the sum of the distances of the tiles from the goal position. The distance will be sum of horizontal and vertical distance. Here, h_2 = 2.

   As, h2 = 0 + 0 + 0 + 0 +1 +1 +0 +0 = **2.**

   H2 is also admissible because all any move can do is move one tile one step closer to the goal.

   Also, true solution cost here is 26.

From the above two observation, it looks like both the heuristic will perform equally with A*.But this not true in reality as h2 ≥ h1 no matter what. In other words h2 always dominate h1.

Suppose there is another 8-puzzle tile as follow:

7	6	
4	3	1
2	5	8

Initial State

1	2	3
8		4
7	6	5

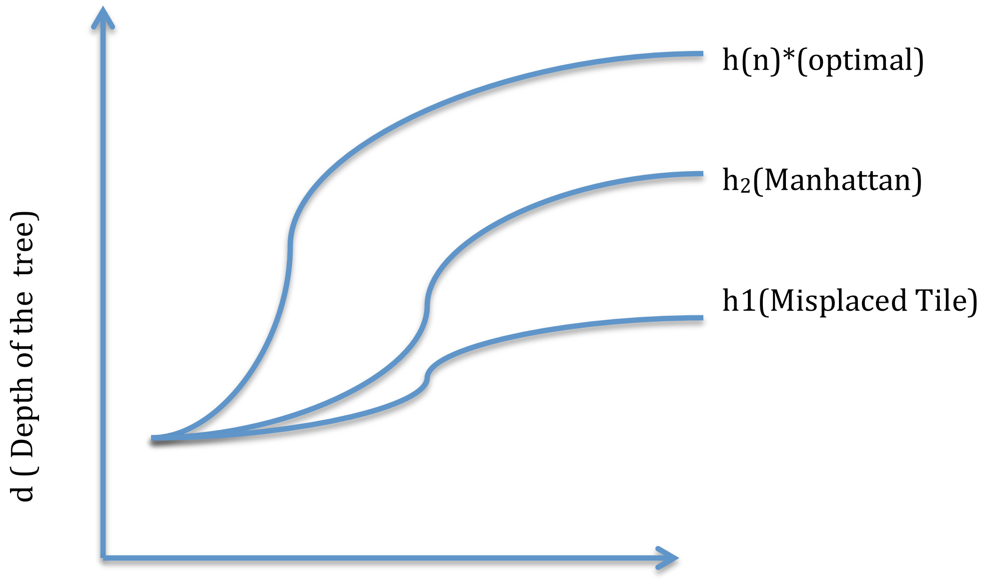
Goal State

Here,
H1 = 8 (as all of them are misplaced).

H2 = 3+ 3+ 2+ 2+ 1+ 2+ 2+ 3 = 18.

So, there it seems like H1 is better than h2 as it is giving us lesser moves or less cost than h2. Here, h2 is dominating h1. Domination translates directly into efficiency: A* using h2 will never expand more nodes than using h1(except for some nodes where f(n) = C*). This is same as saying that every node with h(n) < C* will surely be expanded. But because h2 is at least as big as h1 for all nodes, every node that is surely expanded by A* search with h2 will also surely be expanded with h1.

Hence, it is generally better to use a heuristic function with higher under estimate value. In other words, it is good to get the heuristic that gives the highest possible under estimate of cost.



b*(Branching Factor) {Graph not precise ,just to demonstrate}

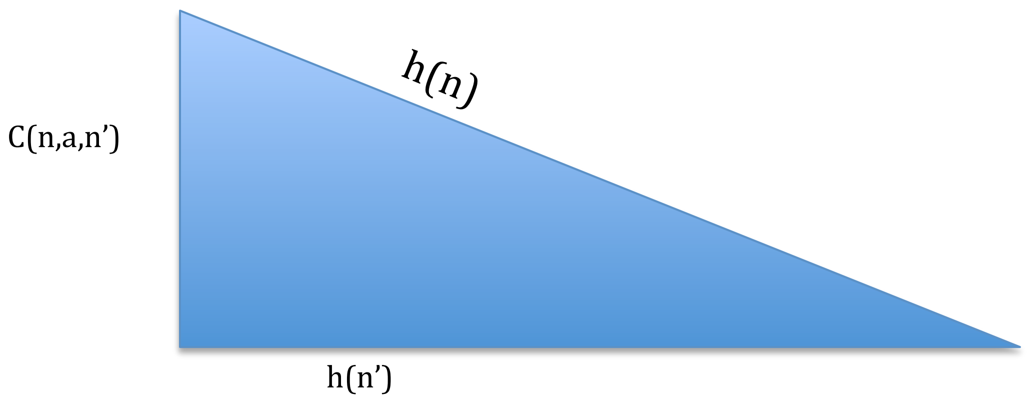
This is clear that we must have to choose that heuristic which is closer to the optimal heuristic that

Question 1 (1)

b)

Solution: Both h1 and h2 heuristics are admissible as they never over estimate the cost of reaching to the goal.

Consistent: For any heuristic to be consistent it should follow the triangle equality. Which is that sum of two sides will always be greater than or equal to the sum of the third side. This fact's analogy to the heuristic is as under:



This means that:

$h(n) \leq c(n,a,n') + h(n')$.

So, heuristic h(n) is consistent if, for every node n and every successor n' and n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'.

<u>Proof of consistency of h1(Missing Tiles heuristic)</u> :

- $c(n,a,n') = 1$ for any action *a*
- Claim: $h_1(n) \leq h_1(n') + c(n,a,n') = h_1(n') + 1$
  - Now, no move (action) can get more than one misplaced tile into place.
  - Also, no move can create more than one new misplaced tile.
  - Hence, the above follows. I.e. **h_1 is consistent**.

<u>Proof of consistency of h2 (Manhattan Distance) :</u>

- c(n,a,n') = 1 for any action a.

- Claim: $h_1(n) \le h_1(n') + c(n,a,n') = h_1(n') + 1$.
  - Now, if we view the above moving tiles as making right angled triangle than it will **prove the above heuristic as consistent**.
  - Because in right angled triangle $\mathbf{a + b \ge c}$

Sudoku Solver using CSP formulation

This program is to solve a Sudoku puzzle using backtracking search with combination of forward checking and heuristics.

Three heuristics are used:
1. Most constrained variable
   This heuristic is first applied to choose the variable has the fewest "legal" moves.
2. Most constraining variable
   If there is a tie when applying the first heuristic, this heuristic is used to choose the variable with the most constraints on remaining variables.
3. Least constraining value
   This heuristic is applied to sort domain values of the selected variable so that the value which leaves the fewest values in the remaining variables is searched first.

Source Code

sudoku.py	The sudoku solver program implemented in Python (version 2.7)

Test Case Files

easy.txt	The easy puzzle
medium.txt	The medium puzzle
difficult.txt	The difficult puzzle
evil.txt	The evil puzzle

Running Instruction

Usage:
```
    python sudoku.py [-f|-h] <puzzle_file>
```

Where:
```
    <puzzle_file>    the file containing the sudoku puzzle to solve
    -f    solve the puzzle using backtracking search with forward checking
    -h    solve the puzzle using backtracking search with forward checking
          and the 3 heuristics
```

Examples:
    To find solution for the puzzle stored in easy.txt file with only backtracking search, use the following command:

```
        python sudoku.py easy.txt
```

    To find the solution using backtracking with forward checking, use the following command:

```
        python sudoku.py -f easy.txt
```

    And to apply both forward checking and the three heuristics, use the following command:

```
        python sudoku.py -h easy.txt
```

Format of input file:
    Each line of the input file represents a row in the puzzle which is a 9-character string containing digits (1..9) or dash (-) character indicating blank cells.
    Line starting with # is considered as comments and not a part of the puzzle.

Question 2

Solving Sudoku puzzle as a constraint satisfaction problem (CSP)

Variables

A sudoku puzzle contains 9x9 cells to fill values. A variable is identified by the location(row index and column index) of the corresponding cell. Therefore, we have 81 variables:

$X_{11}, X_{12}, ..., X_{19}$
$X_{21}, X_{22}, ..., X_{29}$
...
$X_{91}, X_{92}, ..., X_{99}$

Domains

Each cell can be filled in a number from 1 to 9. All variables have the same domain: **{1, 2, ..., 9}**

Constraints

A Sudoku puzzle has three kind of constraints:

- Row Constraint: Every cell in the same row must have a different value.

X_{11} <> X_{12} <> ...<> X_{19}
X_{21} <> X_{22} <> ...<> X_{29}
...
X_{91} <> X_{92} <> ...<> X_{99}

- Column Constraint: Every cell in the same column must have a different value.

X_{11} <> X_{21} <> ...<> X_{91}
X_{12} <> X_{22} <> ...<> X_{92}
...
X_{19} <> X_{29} <> ...<> X_{99}

- Block Constraint: Every cell in the same 3-by-3 block must have a different value.

X_{11} <> X_{12} <> ...<> X_{33}
X_{14} <> X_{15} <> ...<> X_{36}
...
X_{77} <> X_{78} <> ...<> X_{99}

Goal

The goal is to assign a value to every unassigned variable such that all constraints are satisfied.

Performance Table

	B		B+FC		B+FC+H	
	Time (sec)	**Nodes**	**Time (sec)**	**Nodes**	**Time (sec)**	**Nodes**
Easy	0.12	22725	0.12	2549	0.08	51
Medium	0.03	4141	0.02	483	0.07	51
Difficult	0.56	118339	0.54	13172	0.09	58
Evil	0.57	116529	0.54	12973	0.12	77

Performance Discussion

The above performance table shows comparison of algorithm performance in base of:

1. Using the same searching strategy with different problem size.

In the first comparison, the implemented algorithm takes more time to search as well as more nodes to find solution for problem with bigger size. However the is an exception of the Medium test case. Because the backtracking search is a depth first search, some particular problem might result in less depth level.

2. Using different searching strategy with the same problem.

In the second comparison, the implemented algorithm has the best performance when using backtracking with forward checking and heuristics. The searching time as well as searched node is reduced if using forward checking. And the searching nodes is significant reduced if using heuristics.

Forward checking reduces the size domain values of variables that reduces the number of searched nodes.

Heuristics help to choose better variables and values. The backtracking search is depth first search. For better choosing searching brand, the solution can be reached with less searched nodes, therefore, reduce searching time.

TEST CASES:

1) Difficult:

```
2--9-4---
346-8----
9--362---
--3----5-
8-5---1-9
-6----3--
---219--6
----4-938
---8-7--2
```

2) Easy

```
9-4----58
---1---97
--18--4-6
---6---4-
-5-4-9-1-
-9---1---
5-6--43--
34---8---
78----9-4
```

3) Medium

```
--5-2---4
---9-1--6
94--7--2-
48-7---93
---------
75---4-18
-3--1--59
5--2-8---
6---5-3--
```

4) Evil

```
-2---86--
8----7---
-73-4----
2----5-9-
5--6-9--4
-9-3----5
----9-31-
---2----7
--24---8-
```

```
"""
This program is to solve a Sudoku puzzle using backtracking search with
combination of forward checking and heuristics.

Three heuristics are used:
1. Most constrained variable
   This heuristic is first applied to choose the variable has the fewest
   "legal" moves.
2. Most constraining variable
   If there is a tie when applying the first heuristic, this heuristic is used
   to choose the variable with the most constraints on remaining variables.
3. Least constraining value
   This heuristic is applied to sort domain values of the selected variable
   so that the value which leaves the fewest values in the remaining variables
   is searched first.

Usage:
    python sudoku.py [-f|-h] <puzzle_file>

Where:
    <puzzle_file>    the file containing the sudoku puzzle to solve
    -f    solve the puzzle using backtracking search with forward checking
    -h    solve the puzzle using backtracking search with forward checking
          and the 3 heuristics

Examples:
    To find solution for the puzzle stored in easy.txt file with only
    backtracking search, use the following command:

        python sudoku.py easy.txt

    To find the solution using backtracking with forward checking, use the
    following command:

        python sudoku.py -f easy.txt

    And to apply both forward checking and the three heuristics, use the
    following command:

        python sudoku.py -h easy.txt

Format of input file:
    Each line of the input file represents a row in the puzzle which is a
    9-character string containing digits (1..9) or dash (-) character
    indicating blank cells.
    Line starting with # is considered as comments and not a part of the
    puzzle.
"""
import sys
from os import times

class Sudoku:
    """Represent a Sudoku puzzle as a 2-dimension array where each element is
    an int value from 0 to 9. Zero indicates blank cells.
    """
    def __init__(self, filename):
        """Create a new Sudoku puzzle by loading its initial state from the
        given file."""
        self.sudoku = []

        # open the puzzle file
        sudoku_file = open(filename, "U")
        # Read each line
        for line in sudoku_file:
            line = line.strip()
            if line.startswith('#'):
                continue    # ignore comments

            # Each line is a 9-character string containing digits (1..9) or
            # dash (-) character which indicates blank cells.
            row = [int(c) if c.isdigit() else 0
                   for c in line.strip()]
            self.sudoku.append(row)

        sudoku_file.close()

    def get(self, row, col):
        """Return the value at the given location (row, col)"""
        return self.sudoku[row][col]

    def set(self, row, col, value):
        """Set the value at the given location (row, col)"""
        self.sudoku[row][col] = value

    def get_row(self, row):
        """Return values in of the row at the given index (0..8)"""
        return self.sudoku[row]

    def get_col(self, col):
        """Return values in of the column at the given index (0..8)"""
        return [self.sudoku[row][col] for row in range(9)]

    def get_block(self, block_row, block_col):
        """Return values in of the block at the given index (0..2, 0..2)."""

        # The location of the starting cell of the block
        start_row = block_row * 3
        start_col = block_col * 3

        # Return values of 9 cells in the given block
        return [self.sudoku[start_row + row][start_col + col]
                for row in range(3)
                for col in range(3)]

    def print_state(self):
        """Print the current state of the puzzle"""
        for row in self.sudoku:
            for cell in row:
                if cell > 0:
                    print cell,
                else:
```

```
                print '-',
            print
        print

class SudokuCSP:
    """
    Represents Sudoku puzzle as a Constraint satisfaction problem (CSP) with
    the following characteristics:
    Variables:    {X(0,0), .., X(8,8)}
    Domains:      {1, .., 9}
    Constraints:
        Row Constraint: for each row i, i = 0..8:
                        every cell in the row has different value
        Col Constraint: for each column i, i = 0..8:
                        every cell in the column has different value
        Block Constraint: for each block (i,j) i,j = 0..2:
                        every cell in the block has different value
    Goals: Every variable is assigned a value such that all constraints are
    satisfied.
    """

    DOMAIN_VALUES = [1,2,3,4,5,6,7,8,9]

    def __init__(self, assignment, forwardchecking, heuristics):
        self.do_forwardchecking = forwardchecking
        self.use_heuristics = heuristics

    def is_complete(self, assignment):
        """Test whether the given assignment is a complete sudoku state."""

        # Test row constraints
        for i in range(9):
            values = assignment.get_row(i)
            if 0 in values:
                return False
            if sorted(values) != SudokuCSP.DOMAIN_VALUES:
                return False

        # Test col constraints
        for i in range(9):
            values = assignment.get_col(i)
            if 0 in values:
                return False
            if sorted(values) != SudokuCSP.DOMAIN_VALUES:
                return False

        # Test block constraints
        for i in range(3):
            for j in range(3):
                values = assignment.get_block(i, j)
                if 0 in values:
                    return False
                if sorted(values) != SudokuCSP.DOMAIN_VALUES:
                    return False

        return True
    def select_unassigned_var(self, assignment):
        """
        Select unassigned variables in the given sudoku assignment.
        Return the location of the first cell having value of zero if no
        heuristic is used. Otherwise, 3 heuristics are applied to choose
        among unassigned variables.
        """
        unassignments = []

        for row in range(9):
            for col in range(9):
                if assignment.get(row, col) == 0:
                    if not self.use_heuristics:
                        # return the first unassigned variable found
                        return (row, col)
                    else:
                        unassignments.append((row, col))

        if self.use_heuristics:
            # use heuristics to choose one of unassigned variables
            return self.apply_heuristics(assignment, unassignments)

        return None # all variables have been assigned

    def apply_heuristics(self, assignment, var_list):
        """
        Apply most constrained variable and most constraining variable
        heuristics to choose one of unassigned variables.
        """

        # For choosing the variable which has the fewest "legal" moves
        min_choices = 10    # we will have maximum o 9 values
        min_var = None

        for var in var_list:
            # Get number of values of the domain of the variable
            choices = len(self.get_domain_values(assignment, var))

            if choices < min_choices:
                # Save the most constrained variable
                min_var = var
                min_choices = choices

            elif choices == min_choices:
                # There is a tie.
                # Apply the most constraining variable heuristic
                min_var = self.most_constraining_variable(assignment,
                                                          min_var, var)

        return min_var

    def most_constraining_variable(self, assignment, var1, var2):
        """Choose between two variables that has the most constraints on
        remaining variables."""
        count1 = self.count_constraints(assignment, var1)
        count2 = self.count_constraints(assignment, var2)
        return var1 if count1 < count2 else var2

    def count_constraints(self, assignment, var):
        """Count number of constraints between the given variable and other
        unassigned variables."""
        row, col = var

        count = 0

        # Count variables which is row-constraint with the given variable
        for i in range(9):
            if i != col and assignment.get(row, i) == 0:
                count += 1
        # Count variables which is col-constraint with the given variable
        for i in range(9):
            if i != row and assignment.get(i, col) == 0:
                count += 1
        # Count variables which is block-constraint with the given variable
        start_i = row // 3 * 3
        start_j = col // 3 * 3
        for i in range(start_i, start_i + 3):
            for j in range(start_j, start_j + 3):
                if i != row and j != col and assignment.get(i, j) == 0:
                    count += 1

        return count

    def get_domain_values(self, assignment, var):
        """Return domain values (sorted) of the given variable."""

        if self.do_forwardchecking:
            row, col = var

            # get values which are used by the cells on the same row, column,
            # or block
            row_values = assignment.get_row(row)
            col_values = assignment.get_col(col)
            block_values = assignment.get_block(row // 3, col // 3)

            values = [val for val in SudokuCSP.DOMAIN_VALUES
                      # exclude values which is inconsistent with the constraints
                      if val not in row_values and val not in col_values
                      and val not in block_values]

            if self.use_heuristics:
                # Apply least-constraining value heuristic
                values.sort(lambda x, y:
                            cmp(self.least_constraining_value_count(assignment,
                                var, x),
                                self.least_constraining_value_count(assignment,
                                    var, y)))

            return values

        else:
            # Without forward checking, all variables have the same domain
            return SudokuCSP.DOMAIN_VALUES

    def least_constraining_value_count(self, assignment, var, value):
        """Count number of values for the remaining variables to choose if
        the given variable is set with the given value"""
        row, col = var

        row_values = assignment.get_row(row)
        col_values = assignment.get_col(col)
        block_values = assignment.get_block(row // 3, col // 3)

        count = 0
        for val in SudokuCSP.DOMAIN_VALUES:
            if val != value and val not in row_values and \
                val not in col_values and val not in block_values:
                count +=1

        return count

    def test_constraints(self, assignment, var, value):
        """Test if the given value of the given variable is consistent with
        all constraints."""
        row, col = var # Get row, column index of the variable

        # Test row constraint
        if value in assignment.get_row(row):
            return False
        # Test col constraint
        if value in assignment.get_col(col):
            return False
        # Test block constraint
        if value in assignment.get_block(row // 3, col // 3):
            return False

        return True

    def add_var(self, assignment, var, value):
        """Add the value to the specified variable of the assignment"""
        row, col = var
        assignment.set(row, col, value)

    def remove_var(self, assignment, var, value):
        """Remove the value from the specified variable of the assignment"""
        row, col = var
        assignment.set(row, col, 0)

nodes_count = 0 # For counting number of nodes

def recursive_backtrack(assignment, csp):
    """
    Do a recursive backtracking search on the given assignment using the given
    CSP specification. The csp is an object which the following methods:

        is_complete(assignment):
            Test whether the assignment is complete
        select_unassigned_var(assignment):
            Select an unassigned variables of the assignment
            Return an object to identify the variable

        get_domain_values(assignment, var):
            Return domain values for the given variable of the assignment.

        test_constraints(assignment, var, value):
            Test if the given value of the variable is consistent with the
            assignment according to the constraints.

        add_var(assignment, var, value):
            Add the given value to the variable of the assignment

        remove_var(assignment, var, value)
            Remove the value of the variable from the assignment

    Return a solution, or False indicating failure
    """
    global nodes_count

    if csp.is_complete(assignment):
        # The goal is reached, return the solution
        return assignment

    # Select an unassigned variables from the assignment
    var = csp.select_unassigned_var(assignment)

    # Try all values in the domain of the selected variable
    for value in csp.get_domain_values(assignment, var):
        nodes_count += 1

        # Test if value is consistent wit assignment according to
        # Constraints(csp)
        if csp.test_constraints(assignment, var, value):
            csp.add_var(assignment, var, value)

            # Recursive search for solution
            result = recursive_backtrack(assignment, csp)
            if result != False:
                # Solution found
                return result

            # Remove the tested value of the variable
            csp.remove_var(assignment, var, value)

    # Reached here, all possible values tested with no solution found
    return False


def main():
    """
    Main function to read command line parameter for the puzzle, solve and
    print out the solution (if any) including performance data (time, nodes).
    """
    # input file containing the initial puzzle state
    sudoku_file = None
    # do backtracking with forward checking
    do_forwardchecking = False
    # backtracking search with forward checking and the 3 heuristics
    use_heuristics = False

    # get command line parameters
    for arg in sys.argv[1:]:
        if arg == "-f":
            do_forwardchecking = True
        elif arg == "-h":
            use_heuristics = True
            do_forwardchecking = True
        else:
            sudoku_file = arg

    if not sudoku_file:
        print "Usage: python sudoku.py [-f|-h] <puzzle_file>"
        return

    # read the puzzle file for the initial state
    sudoku = Sudoku(sudoku_file)

    print "Initial State:"
    sudoku.print_state()

    print "Backtracking search",
    if do_forwardchecking:
        print "with forward checking",
        if use_heuristics:
            print "and 3 heuristics",
    print

    # Create the CSP from the given puzzle with optional forward checking and
    # heuristics.
    csp = SudokuCSP(sudoku, do_forwardchecking, use_heuristics)

    # Find the solution using backtracking search
    start = times() # Record the start time
    solution = recursive_backtrack(sudoku, csp)
    end = times() # Record the end time

    # Print the solution if any
    if solution:
        print "Solution Found:"
        solution.print_state()
    else:
        print "Solution not found"

    # compute the time to complete the puzzle
    time = end[0] - start[0]
    print "Puzzle is solved in", time, "seconds with",
    print nodes_count, "nodes searched."
```

```
if __name__ == '__main__':
    main()
```

REFERENCES:

1) http://en.wikipedia.org/wiki/Heuristic.

2) http://www.cs.brown.edu/courses/cs141/amy_notes/astar.pdf

3) Russell and Norvig : A Modern Approach(3rd Edition)