Explanation of DFSB:

```
function search(csp):
    return backtrack({}, csp)


def backtrack(assignment, csp):
    if assignment is complete:
        return assignment
    else:
        var = Select_Unassigned_Var(csp)
        for each value in Domain_Value(var, csp):
            if  value is consistent with assignment given Constraints(csp):
                add{var = value} to assignment
                if (backtrack(assignment, csp) != failure):
                    return assignment
                remove{var = value} from assignment
        return failure
```

Plain DFSB loops through every value in the domain for each variable, and it tries to find out a solution that has no conflict with the constraints.

Explanation of DFSB++:

```
function search_plus(csp):
    return backtrack_plus({}, csp)


def backtrack_plus(assignment, csp):
    if assignment is complete:
        return assignment
    else:
        var = Select_Unassigned_MCVar(csp)
        # values are ordered such that LCValue comes first
        for each value in Order_Domain_Value(var, csp):
            if  value is consistent with assignment given Constraints(csp):
                AC-3(var, csp)
                add{var = value} to assignment
                if (backtrack_plus(assignment, csp) != failure):
                    return assignment
                remove{var = value} from assignment
        return failure
```

DFSB++ is similar to DFSB, except that DFSB++ picks MCVar for var in each step, and it searches the values in an order that from LCValue first and MCValue last. Also, before assigning a consistent value to var and going to the next recursive step, DFSB++ performs AC-3 for the current csp to rule out some values in the domain of the neighbors of current var.

Explanation of Min-Conflicts:

```
function search_minconflicts(csp, max_steps):
    use random method to generate start_state
    return backtrack_plus(csp, max_steps,  start_state)


function minconflicts(csp, max_steps, current_state)
   for i=1 to max_steps do
       if current_state is a solution of csp then return current_state
       var <-- a randomly chosen variable from the set of conflicted variables CONFLICTED[csp]
       value <-- the value v for var that minimizes CONFLICTS(var,v,current_state,csp)
       pr = random_number_from_1_to_10
       if pr <= 7:
           set var = value in current_state
       else:
           set var = a random value in the domain in current_state
   return failure
```

Min-Conflicts starts with a random generated state (assignment). In each step, a variable is chosen in random, and the value chosen should minimized the conflicts in the state (assignment). To adjust the algorithm to avoid plateaus or getting stuck in local depressions, this value will be assigned to the variable with a probability of 70%; and a random value will be assigned to the variable with a probability of 30%. If no solution was found after max_steps, return failure. In my implementation, I set max_steps to be 100*N.

Performance Table

	DFSB-easy	DFSB-hard	DFSB++-easy	DFSB++-hard	MinConflicts-easy	MinConflicts-hard
time	1ms	Not converging	1ms	1727ms	3ms	172ms
Search steps	8	-------------	8	990	113	1051
Prune calls	-----------	-------------	13	15328	-----------------	-----------------

Difference Observation:

DFSB and DFSB++ have similar performance on the easy case. However, DFSB is not converging for the hard case. The reason for this is that I did not set a max_steps (the search steps allows before giving up). So, while DFSB is searching a result for the hard case, it can go to a very deep level of the states tree, while that could be the wrong trace.

For the Min-Conflicts, it will always return a solution for the easy case. Since 100*N random searches should be enough with high probability to find out a solution with respect to the M constraints and the randomly generated start_state for the easy case. However, for the hard case, the start_state could be totally messed up, and the 100*N search steps might not be enough to find out a solution. So Min-Conflicts might fail with a higher probability for the hard case.