Report

Simple DFS with backtracking:

The algorithm was used almost exactly as described in the textbook.

The code was modified to avoid using any performance enhancers such as variable/value ordering, or heuristics.

A simple pseudocode is as follows:

```
function BACKTRACK (assignment, csp) returns success, or failure
    if assignment is complete then return success
    var ←select first unassigned variable in the given list
    for each value in possible_values do:
        add {var = value} to assignment
        if value is consistent with assignment then
            result<-BACKTRACK (assignment, csp)
            if result is success:
                return success
    remove {var = value} from assignment
    return failure
```

The simple backtrack is a direct implementation of DFS. We first check if we have a complete and valid assignment for all the variables. If so, we have our result. Or else, we keep searching the tree/graph till we have our assignments. We select the first unassigned variable, and iterate through each of its possible values. If a value is consistent with the assignments, we travel deeper, now adding the current assignment to the existing ones. If after looking at every single situation, we don’t have an answer, we return failure.

Simple DFS with backtracking and Arc Consistency:

```
function BACKTRACK (assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
```

```
    var ←SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES (var, assignment, csp) do
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences ←INFERENCE (csp, var , value)
            if inferences   = failure then
                add inferences to assignment
                result ←BACKTRACK (assignment, csp)
                if result   = failure then
                    return result
        remove {var = value} and inferences from assignment
    return failure
```

The pseudocode is the same as given in the book. I have used the same structuring for the code.

The remaining functions is as mentioned below:

```
function SELECT-UNASSIGNED-VARIABLE (csp, assignments), returns a variable
    until a variable is found:
        select the variable with the smallest set of remaining values in its domain
        if the variable is unassigned:
            return the variable
        else:
            select the next variable with the smallest set of remaining values in its domain
```

```
function ORDER-DOMAIN-VALUES (csp, assignments, variable, domain_vals), returns a sorted list
    for each value in domain_values of variable do:
        check for the number of reductions caused in domains of variables that are neighbors
        of current variable by selecting the current domain_value
```

create a tuple with the value and number of reductions

add the tuple to the list

sort the list in ascending order of the number of reductions caused

return the list of tuples

function AC-3(csp) returns false if an inconsistency is found and true otherwise

inputs: csp, a binary CSP with components (X, D, C)

local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do

(Xi, Xj)←REMOVE-FIRST(queue)

if REVISE(csp, Xi, Xj) then

if size of Di = 0then return false

for each Xk in Xi.NEIGHBORS - {Xj} do

add (Xk, Xi) to queue

return true

function REVISE(csp, Xi, Xj) returns true iff we revise the domain of Xi

revised ←false

for each x in Di do

if no value y in Dj allows (x ,y) to satisfy the constraint between Xi and Xj then

delete x from Di

revised ←true

return revised

This version of the DFS backtrack has optimized selection of values and variables.

We choose the variable that has the least number of valid domain values.

For that we just keep iterating through the list of variable and find one with the minimum length of its domain values and still isn't assigned a value.

Once we have our variable, we then look through its domain values (that remain after pruning).

We then sort this list in order of the number of unassigned variables whose domains are reduced.

Once we have our value and variable, we then repeat the steps of the simple dfsb but, now before we call the recursive function, or assign the value, we call a function that makes the entire thing arc-consistent. The arc consistent algorithm will fix the domains of all variables that might be affected by the assignment of the value to the variable. Once we have established arc consistency, we proceed as before.

If we cannot establish arc consistency, there is no need to visit this particular subtree, so we simply move on.

function MIN-CONFLICTS(csp,max steps) returns a solution or failure

inputs: csp, a constraint satisfaction problem

max steps, the number of steps allowed before giving up

current ←an initial complete assignment for csp

for i = 1 to max steps do

if current is a solution for csp then return current

var ←a randomly chosen conflicted variable from csp.VARIABLES

value←the value v for var that minimizes CONFLICTS(var, v, current , csp), chosen with probability p

OR

Value<-random value v for var with probability 1-p

set var =value in current

return failure

Minconflicts at its start simply assigns a random value to all its variables.

We then (for a fixed number of steps) keep assigning those variables, for which the assignment is not consistent. At each step, we select a random variable from the list of inconsistent variables and check for its assignment. Through simply randomizing, we arrive at a complete and consistent assignment for all the variables.

Now the problem here is that sometimes we can get caught in a local minimum. To remedy this, I simply added another random component to the algorithm, namely, RANDOM WALK.

At each assignment step, the value to assign is chosen by a probability. We assign a random value with probability p. And, with probability 1-p, we assign it a value that reduces (by the highest value) the number of conflicts corresponding to the given CSP.

PERFORMANCES:

Algorithm	Input	Time Taken	Info	Result
DFS_Plain	**Backtrack_easy**	**0.0**	**9 calls**	**Success**
DFS_Plain	**Minconflicts_easy**	**0.004010677337646484**	**1506 calls**	**Success**
DFS_AC3	**Bactrack_easy**	**0.0009682178497314453**	**9 calls, 13 prunes**	**Success**
DFS_AC3	**Bactrack_hard(modified)**	**0.6376965045928955**	**501 calls, 1187 prunes**	**Success**
DFS_AC3	**Minconflicts_easy**	**0.0030090808868408203**	**26 calls, 36 prunes**	**Success**
DFS_AC3	**Minconflicts_hard**	**0.10631084442138672**	**274 calls, 2417 prunes**	**Success**
Minconflicts	**Backtrack_easy**	**0.0009744167327880859**	**56**	**Success**
Minconflicts	**Minconflicts_easy**	**0.002004384994506836**	**68 steps**	**Success**
Minconflicts	**Minconflicts_hard**	**32.26969003677368**	**250000 steps**	**No answer**

Looking at the results, it is clear to see why in some cases a random approach such as that of Minconflixts in useful. When in the worst case, even with effective pruning, we have to traverse many paths to get the correct output, we can simple randomize the assignments and see if we can reach the result faster, which is true in the case of the Minconflicts_easy input.

The downside of using a random algorithm is that it is easy to get stuck and not be able to move past a local minimum, even with a random walk addition. We are unable to get a result even in 250000 steps. But a deliberate search method such as a DFS with backtracking and heuristics such as AC3, we can arrive at a solution as can be seen from the result above.

Also, when we use a random algorithm, it can be more expensive than to just use a simple straightforward algorithm in the case of a simple input.

For very simple input, we should just brute force the algorithm instead of trying to optimize it. Optimization could involve unnecessary calculations which might not be needed such as the prunes.