Report on Graph Coloring using CSP and Min-conflicts Local Search

Some common methods used to solve CSP are:

1) Depth First search with Backtracking (DFSB)
2) Depth First search with Backtracking + Variable, value ordering + AC3 for constraint propagation (DFSB++)
3) Using Local search algorithms like MinConflict algorithm with some added noise to surpass local minima and plateaus

Let us see how each of these algorithms work:

DFSB: A Depth-first search algorithm that chooses values one variable at a time and backtracks when a variable has no legal values left to assign is called DFSB. Here, we first select an unassigned variable, then choose a value from its domain and check if it is consistent with its constraints. If it is consistent, we finalise this assignment and recurse for the next assignment. If it is inconsistent, we choose a different value for the variable. If we do not find any consistent value for a variable, we backtrack and change the assignment for a previous variable.

```
function BACKTRACKING-SEARCH(csp) {
# returns a solution, or failure
        return BACKTRACK({ }, csp)
}

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
                        result ← BACKTRACK(assignment, csp)
                        if result ≠ failure then
                                return result
                        remove {var = value} from assignment

        return failure
}
```

DFSB++: This is an improvement over DFSB algorithm. It uses the Most Constrained Variable and Least Constrained Value heuristics to perform variable, value ordering while selecting a value to be assigned to a variable. A variable in a CSP is said to be arc-consistent if every value in its domain satisfies the variable's binary constraints. This helps us prune the domains of variables before-hand, thus reducing the number of backtracking required. For this purpose, we use the AC3 algorithm before we finalise each assignment.

```
function BACKTRACKING-SEARCH(csp) {
# returns a solution, or failure
        return BACKTRACK({ }, csp)
}

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

        var ← SELECT-MOST-CONSTRAINED-VARIABLE(csp)
        for each value in LEAST-CONSTRAINED-VALUE(var,assignment,csp) do
                if value is consistent with assignment then
                        # Check arc consistency for all variables
                        consistent ← AC3(csp,var,value)
                        if consistent = failure then
                                continue
                        add {var = value} to assignment
                        result ← BACKTRACK(assignment, csp)
                        if result≠ failure then
                                return result
                        remove {var = value} from assignment

        return failure
}

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 = 0 then
                                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
}
```

MinConflict Algorithm: Local search algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local optima" they are equipped with various heuristics for randomizing the search. We will be using the random walk heuristic to avoid local optima and plateaus. Their stochastic nature generally voids the guarantee of "completeness" provided by the systematic search methods.
Min-conflicts heuristics chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

```
function MIN-CONFLICTS(csp,max steps,p) {
# returns a solution or failure
# inputs: csp, a constraint satisfaction problem
            max steps, the number of steps allowed before giving up
            p is the probability of randomly choosing a value for a variable

        current ← an initial complete assignment for csp using greedy approach
        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
                if probability p verified then
                        value ←a value v randomly chosen from domain of var
                else
                        value ←the value v for var that minimizes CONFLICTS(var,v,current,csp)
                set var = value in current

        return failure
}
```

Performance Table:
Let us look at the performance of these algorithms given the inputs:
- ➢ Backtrack_easy
- ➢ Backtrack_hard

- MinConflict_easy
- MinConflict_hard

Algorithm	Input	Time taken	No. of search steps + arc pruning steps
DFSB	Backtracking_easy	0.00091	12
DFSB	Backtracking_hard	NA	NA
DFSB++	Backtracking_easy	0.0015	36
DFSB++	Backtracking_hard	14.252	5980
MinConflict	MinConflict_easy	0.0096	103
MinConflict	MinConflict_hard	20.99	10000

Performance Observations:

Backtracking_easy : DFSB & DFSB++ give comparable results for easy inputs as number of steps involved for such inputs are less. However, the extra steps for DFSB ++ is due to the arc pruning steps.

Backtracking_hard: DFSB++ outperforms DFSB as it has improved heuristics like MCV, LCV and AC3 checks involved. DFSB goes on infinitely for this input as the number of permutations for each node are huge.

MinConflict: Local search algorithms like minconflict can also be used successfully to solve CSP.

For easy inputs, Minconflict gives results in equivalent time as the DFSB. However, we need to optimize the minconflict algorithm to accommodate more randomisation heuristics for it work for large inputs.