Jessica Lin

CSE 352 – AI

Assignment 1 – Search Report

1. Method Description

1.1.DFSB

The method for DFSB takes in an *assignments* and *csp* variable, which represent the assigned values and the neighbors for each variable.

First, *dfsb* checks if a solution is reached by making sure that for each assigned value, there isn't a neighboring variable with the same value. If a solution has not yet been reached, then a random unassigned variable is selected.

For each of the valid values in the variable's domain, if by assigning that variable, the constraints are still met, then the variable is assigned that value. The function *dfsb* is then recursively called to select the next unassigned variable. If when the function call is returned and is not None, then a solution has been reached and is returned.

Otherwise, the value is unassigned, and the next possible value is attempted.

```
def select_unassigned_variable(assignment, csp):
    unassigned <- list of unassigned variables
    return random(unassigned)

def dfsb(assignment, csp):
    if is_solution(assignment, csp):
        return assignment
    var <- select_unassigned_variable(assignment, csp)
    for each value in domain[var]:
        if value is valid given constraints:
            assignment[value] □ value
            result <- dfbs(assignment, csp)
            if result is solution:
                return result
            remove [val, value] from assignment
    return None
```

1.2. DFSB++

The method for DFSB++ takes in an *assignments* and *csp* variable, which represent the assigned values and the neighbors for each variable.

First, *dfsb++* checks if a solution is reached by making sure that for each assigned value, there isn't a neighboring variable with the same value. If a solution has not yet been reached, then a random unassigned variable is selected.

Among the unassigned variables, the one that is the most constrained is selected. The most constrained variable is the one with the least number of domains. With the selected variable, the domain is sorted by least constrained values. How constrained a value is depends on how whether the value is also in the neighboring domains. A copy of the current state of all the variable domains is made.

For each of the valid values, if by assigning that variable, the constraints are still met, then the value is assigned to that variable.

The arc consistency function is then called in order to prune the domains. In this function, we add the arcs of the assigned variable and its neighbors to a queue. While this queue is not empty, any inconsistent values in the domain are removed. If any values are removed, then the arcs are added to the queue.

The function *dfsb++* is then recursively called to select the next unassigned variable. If when the function call is returned and is not None, then a solution has been reached and is returned. Otherwise, changes to the domain are undone.

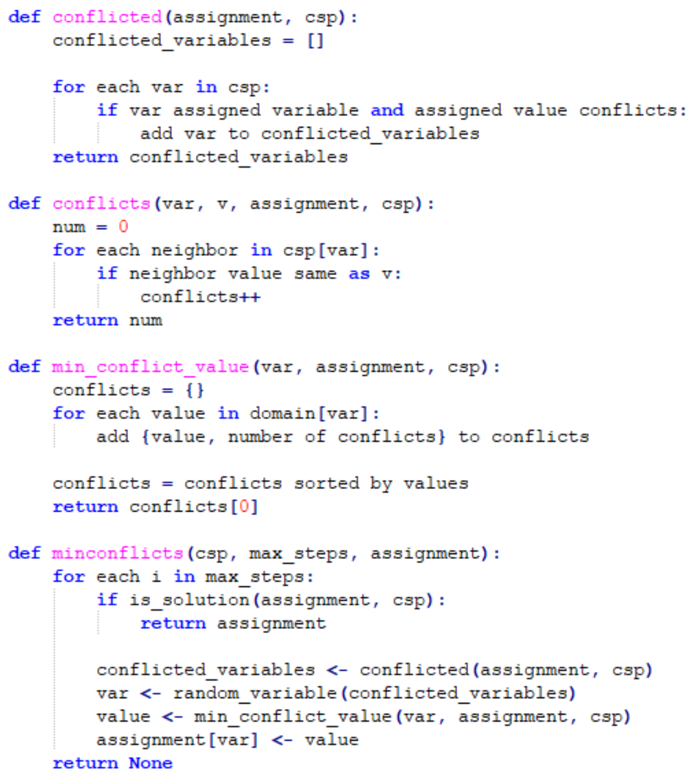
```
def ac_3(csp, variable):
    queue <- list of [variable, neighbor] and [neighbor, variable] pairs

    while queue is not empty:
        x_i, x_j = queue.pop()
        if remove_inconsistent_values(x_i, x_j):
            for each x_k in csp[x_i]:
                queue.append([x_k,x_i])

def remove_inconsistent_values(x_i, x_j):
    removed <- False
    for each value in domain[x_i]:
        if value is inconsist:
            remove value from domain of x_i
            removed <- True
    return removed

def most_constrained_variable(assignment, csp):
    most_constrained_variable <- None
    min_value <- inf
    for each variable in csp:
        if variable is more constrained than current most_constrained_variable:
            most_constrained_variable <- variable
            min_value <- number of values in domain[variable]
    return most_constrained_variable

def least_constrained_values(variable, assignment, csp):
    count = dictionary containing values in domain[variable]

    for each var in csp[variable]:
        for each value in domain[var]:
            increment count[value] if value in dictionary

    return keys of count sorted by value

def dfsb++(assignment, csp):
    if is_solution(assignment, csp):
        return assignment

    var <- most_constrainted_variable(assignment, csp)
    values <- least_constrainted_values(var, assignment, csp)
    domain_copy = deepcopy(domain)

    for each value in values:
        if is_valid(assignment, csp, var, value):
            assignment[var] <- value
            domain[var] <- [value]

            ac_3(csp, var)
            result <- dbfs++(assignment, csp)

            if success:
                return result
        reset domain
    return None
```

1.3. MinConflicts

The method for *minconflicts* takes in an *assignment*, *csp*, and *max_steps* variable, which represents the assigned values, the neighbors for each variable, and the maximum number of iterations.

Minconflicts loops *max_steps* times. At each iteration, it's checked if a solution has been reaches by making sure that for each assigned value, there isn't a neighboring variable with the same value. If a solution has not yet been reached, then a random unassigned variable is selected.

Next, with the current state, a list of variables that conflict with the constraints is determined. If there's no conflicting values, then a random variable that hasn't been assigned yet is selected. Otherwise, a random variable from the list of conflicting variables is selected. Finally, value that will conflict the least is selected to be assigned to the variable.



```
def conflicted(assignment, csp):
    conflicted_variables = []

    for each var in csp:
        if var assigned variable and assigned value conflicts:
            add var to conflicted_variables
    return conflicted_variables

def conflicts(var, v, assignment, csp):
    num = 0
    for each neighbor in csp[var]:
        if neighbor value same as v:
            conflicts++
    return num

def min_conflict_value(var, assignment, csp):
    conflicts = {}
    for each value in domain[var]:
        add {value, number of conflicts} to conflicts

    conflicts = conflicts sorted by values
    return conflicts[0]

def minconflicts(csp, max_steps, assignment):
    for each i in max_steps:
        if is_solution(assignment, csp):
            return assignment

        conflicted_variables <- conflicted(assignment, csp)
        var <- random_variable(conflicted_variables)
        value <- min_conflict_value(var, assignment, csp)
        assignment[var] <- value
    return None
```

2. Performance

2.1. DFSB

Parameter Set	Number of States	Actual Time
N=20, K=4, M=100	215.65 ± 343.38	0.003969901586356627
N=50, K=4, M=625	613.8 ± 647.80	0.011070263385772706
N=100, K=4, M=2500	857.3 ± 838.74	0.024009406566619873
N=200, K=4, M=10000	698 ± 744.99	0.036639511585235596
N=400, K=4, M=40000	947.55 ± 530.82	0.07724903821945191

2.2. DFSB ++

Parameter Set	Number of States	Actual Time
N=20, K=4, M=100	20.65 ± 1.53	0.0013566136360168457
N=50, K=4, M=625	50 ± 0	0.007840752601623535
N=100, K=4, M=2500	100 ± 0	0.039805269241333006
N=200, K=4, M=10000	200 ± 0	0.23693499565124512
N=400, K=4, M=40000	400 ± 0	3.0118372559547426

2.3. MinConflicts

Parameter Set	Number of States	Actual Time
N=20, K=4, M=100	203.15 ± 235.55	0.0026965498924255537
N=50, K=4, M=625	317.55 ± 310.9	0.008189439773555957
N=100, K=4, M=2500	608.95 ± 643.36	0.03266298770904541
N=200, K=4, M=10000	1715.2 ± 1662.51	0.2115374445915222
N=400, K=4, M=40000	2724.5 ± 2338.14	1.2442355155944824

3. Observed Performance Difference

Between all three algorithms, dfsb++ visits the least number of states for all parameter sets. However, between dfsb and minconflicts, when there's a smaller number of variables, minconflict visits less states than dfsb. As the number of variables grow, minconflicts begins to visit more states than dfsb. For both dfsb and minconflicts, however, there is also a large fluctuation on the number of states visited. This may be due to the degree of randomness the algorithms have when choosing an unassigned variable. The dfsb++ algorithm, on the other hand, visits the least number of states compared to the other two algorithms. The number of states visited in dfsb++ appears to be around the same as the total number of variables in the problem.

For both minconflicts and dfsb, the algorithms run faster than dfbs++ as the number of constraints grows closer to $\frac{N^2}{4}$. However, when there's less constraints with a larger number of variables, both minconflicts and dfsb have difficulty solving the problem in a reasonable time, although minconflicts runs faster than dfsb. On the other hand, compared to the other two algorithms, dfsb++ is able to solve the algorithms at a faster rate. The difference is due to the fact that dfsb++ does more work per state than the other two algorithms. For each state, dfsb++ performs arc consistency and pruning.

4. Citations

[1] MinConflicts: https://en.wikipedia.org/wiki/Min-conflicts_algorithm

[2] DFSB and DFSB++:
https://piazza.com/class_profile/get_resource/kkkgroo574w5gb/km12tqtchle3ux