Suheyla Iyimaya
Niranjan Balasubramanian
CSE 352
03/16/2021

Assignment 2: Constraint Satisfaction

1. **Briefly explain how each method works including pseudo code for DFSB, DFSB++, and MinConflicts**

**DFSB**

```
def dfsb_plain(csp): returns True if a solution is found. False otherwise
   if assignment_is_complete(variables):
       return True
   variable = select_unassigned_variable(variables)
   for each color in variable’s domain do:
       if is_consistent(color, variable):
           variable.value = color
           result = dfsb_plain(csp)
           if result is not None:
               return result
           variable.value = -1 #unassign the variable’s color
   return None
```

`assignment_is_complete(variables):` checks if each variable in assignment has an assigned value. Returns True if yes, False otherwise

`select_unassigned_variable(variables):` Iterates through the variables list in csp and returns the first unassigned variable

`is_consistent(color, variable):` checks if the csp is consistent. If the given color is same as any of the variable's neighbor's color, returns False. If all the neighbors of the variable has a different value then the given color, returns True

## DFSB++

```
def dfsb_improved(csp): returns True if a solution is found. False otherwise
    if assignment_is_complete(variables):
        return True
    variable = select_most_constrained_variable(variables)
    order_domain_values(variable, csp)
    for each color in variable.domain:
        if is_consistent(color, variable):
            variable.value = color
            variable.domain = [color]
            initialize_queue(csp)
            ac_3(csp)
            result = dfsb_improved(csp)
            if result:
                return True
            variable.value = -1.  #unassign the variable's color
            resetDomains(csp)
    return False
```

assignment_is_complete(variables): checks if each variable in assignment has an assigned value. Returns True if yes, False otherwise

select_most_constrained_variable(variables): selects and returns  the most constrained variable (the variable with the least number of values in its domain)

order_domain_values(variable, csp): orders the domain of the given variable from least constraining value to most constraining value by assuming the value for the variable and using the csp to check how many values remain for the other variables.

is_consistent(color, variable): checks if the csp is consistent. If the given color is same as any of the variable's neighbor's color, returns False. If all the neighbors of the variable has a different value then the given color, returns True

initialize_queue(csp): This method adds all the arcs to csp's queue. If A and B are adjacent, A->B and B->A are arcs which are added as [A, B] and [B, A] to the queue. The method makes sure to add the arcs added have unassigned tails.

ac_3(csp): This method implements the Arc Consistency algorithm which prunes the current set of possibilities. For all possible colors of tail, checks if there is a consistent(different) color in head. If this fails, it removed that color from tail's domain.

resetDomains(csp): this method resets the domain of unassigned variables (by setting all the unassigned variables to [0, 1, 2] if K = 3)

## MinConflicts

```
# returns True if a solution is found. False otherwise
def minconflictsAlgorithm(max_steps, csp):
    for i from 0 to max_steps
        if is_solution(variables):
            return True
        variable = pickVariable(csp)
        # if you assigned the minConflict value of each
          variable, restart from a new random assignment to
          avoid local depression
        if variable is None:
            csp = assign_random_values(csp)
            reset_minconflict_vals(csp)
            variable = pickVariable(csp)
        value = pickValue(variable, csp)
        # Add some randomness helps! (Simulated Annealing)
          Rather than restrictedly choosing good moves, we will
          sometimes also allow bad moves to avoid being stuck
        keep_randomness = randint(0, 50)
        if keep_randomness < 35:
            variable.value = value
        else:
            variable.value = randint(0, num_colors)
    return False
```

`is_solution(variables):` if all the given variable's values are consistent (no neighbor two neighbors have the same color) return true, false otherwise

`assign_random_values(csp):` assign random values for each variable in csp

`reset_minconflict_vals(csp):` for each variable in csp, reset its current minimum conflict value

`pickVariable(csp):` randomly pick a variable that violates the constraints (if the variable has the same color as any of its neighbors). If all the minconflict values are assigned for each variable, returns none, otherwise returns a variable that violates constrains.

`pickValue(variable, csp):` Finds and returns the value that minimize the total number of violated constraints (over all variables) (if assigning a color gives the most number of options for other variables, return that variable)

2. **Tables describing the performance of the algorithms (DFSB,DFSB++, and MinConflicts) on your generated problems.**

| DFSB | | |
|---|---|---|
| parameter set | number of states | actual time (ms) |
| N=20, K=4, M=100 | 1091.5 ± 5.9160 | 0.33637 ± 1.4400 |
| N=50, K=4, M=625 | 266.5 ± 5.9160 | 0.1306 ± 0.46566 |
| N=100, K=4, M=2500 | 261.5 ± 5.91607 | 0.2213 ± 0.8055 |
| N=200, K=4, M=10000 | 1486.5 ± 5.9160 | 1.558 ± 6.6055 |
| N=400, K=4, M=40000 | 566.5 ± 5.9160 | 1.3398 ± 5.0598 |

| DFSB++ | | |
|---|---|---|
| parameter set | number of states | actual time (ms) |
| N=20, K=4, M=100 | 29.5 ± 5.9160 | 0.343 ± 1.4875 |
| N=50, K=4, M=625 | 59.5 ± 5.9160 | 3.0218 ± 13.4113 |
| N=100, K=4, M=2500 | 109.5 ± 5.9160 | 3.0353 ± 13.408 |
| N=200, K=4, M=10000 | 209.5 ± 5.9160 | 3.0960 ± 13.393 |
| N=400, K=4, M=40000 | No Answer | No Answer |

| MinConflicts | | |
|---|---|---|
| parameter set | number of states | actual time (ms) |
| N=20, K=4, M=100 | 255.5 ± 5.9160 | 0.4302 ± 1.7703 |
| N=50, K=4, M=625 | 1021.5 ± 5.9160 | 3.1338 ± 13.3849 |
| N=100, K=4, M=2500 | No Answer | No Answer |
| N=200, K=4, M=10000 | No Answer | No Answer |
| N=400, K=4, M=40000 | No Answer | No Answer |

3. **Explain the observed performance differences.**

The test cases above use K = 4, M = $N^2/4$, and N = [20, 50, 100, 200, 400]. For simple DFSB, DFSB++ and MinConflicts, standard deviation of depth we find the solution is the same for all the test cases. For DFSB and DFSB++, mean of number of states keeps increasing, but for MinConflicts the number of states changes without a pattern since we deal with randomness instead of systematic search. We can also realize that DFSB++ takes longer than DFSB to solve the problems since it does pruning. Pruning is helpful for solving harder problems but since the time complexity of DFSB++ is greater than DFSB, it makes sense to see that DFSB takes less time to find a solution for smaller sets. In addition, MinConflicts could not find a solution for the last 3 use cases in less than 60 seconds so the table is filled with “No Answer”.