Assignment 2: Constraint Satisfaction
CSE 352 Spring 2021

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

DFSB

The (plain) depth-first search algorithm with backtracking, goes down the branch of a tree one variable at a time. If there is a dead-end, the algorithm goes back up the branch to the last variable whose value can be changed without violating any constraints and change (backtracking). After the value is changed, the process starts over. The algorithm is complete and will find a solution if one exists. It will expand the entire (finite search space if necessary).

In the case of the constraint satisfaction problem of graph coloring problem, each variable has a domain of values. The values are the possible colors. Constraints are given as a set of two variables and they are not allowed to have the same value (color).

For the pseudo-code, there is a helper function (backtracking-search) that calls the recursive DFSB function (recursive-backtracking). The helper function passes an empty assignment and the CSP to the DFSB function. The CSP is an object that stores the graph's variables, constraints (given to us), and domain for each variable. In the recursive DFSB function, it first checks if the assignment is complete - which means that each variable is assigned a correct value (colors) and follows the constraints. If it isn't, it goes to the next part of the algorithm. "Var" is assigned a selected unassigned variable (select-unassigned-variable). An unassigned variable is one that exists in CSP but not in the assignment as it has not been assigned a value yet. In plain DFSB, it just selects the first in the ordered list of variables provided by the CSP (no optimization). For each value in order domain values (order-domain-values) (iterate through each possible value for var). In plain DFSB, it just takes the default order provided by the CSP. If the value is consistent with the assignment given the constraints from CSP (valid value for the variable), add "var" with this value to the assignment (this variable assigned the color), "result" is assigned another call to the recursive DFSB function (recursive-backtracking) (with the updated assignment, go through this process again for the next variable), if "result" doesn't return failure then return "result" (solution was found), (otherwise, solution was not found, it will go to the next part of the algorithm) remove "var" from the assignment (so that it can go through another assignment down the tree). Outside the for each value loop, return failure (no solution was found after exploring the entire tree).

Sources used: http://pages.cs.wisc.edu/~bgibson/cs540/handouts/csp.pdf, Lecture Slides

DFSB++

The plain DFSB algorithm is not that efficient and can be improved, which is what the DFSB++ algorithm does. Improvements to the backtracking efficiency are made by “general-purpose heuristics that can give huge gains in speed”. It does so by addressing the following: deciding which variable should be assigned next, deciding what order should a variable’s values be tried, and being able to detect inevitable failure early on. DFSB++ uses variable, value ordering + AC3 for constraint propagation.

In this algorithm, selecting the unassigned variable (select-unassigned-variable) will use the most constrained variable, also called the Minimum Remaining Values (MRV) heuristic and degree heuristic. Ordering the domain values (order-domain-values) will use the Least Constraining Value (LCV) heuristic. The algorithm also uses forward checking and AC3 (arc consistency) to trim variable domains - so assigning a variable domain with fewer values (colors) to choose from will prove to be quicker. The algorithm uses constraint propagation by calling the inference function (inference).

In the pseudo-code, it is the same as plain DFSB with some additions mentioned in the previous paragraph. Selecting an unassigned variable (select-unassigned-variable) with MRV chooses the variable with the fewest legal values = choosing the variable with the most constraints. This tries to cut off the search as soon as possible. When there is a tie between variables, the degree heuristics decide how to break the tie. The degree heuristic chooses the variable with the highest degree - the one with the most constraints provided by CSP.

Ordering domain values (order-domain-values) with LCV, given a variable it will choose the least constraining value = the value that rules out the fewest values in the remaining variables (unassigned variables). This tries to pick values the best first.

In the algorithm after the variable with it’s value is added to the assignment - the domain for the variable is set to just that value, then “inferences” is assigned a boolean value, from the inference function (inference). Inference will call the forward checking and ac3 (arc consistency) heuristics. Inside the if value is consistent with assignment then, If inferences does not return failure, it adds the inferences to the assignment - the unassigned variables that have a domain with only one value in it (the only option for value), which is consistent given the assignment and constraints. Then “result” is assigned another call to the recursive DFSB function (recursive-backtracking). If “result” is not a failure then return “result” (solution was found). Inside the if inferences deos not return a failure and after the code mentioned in the previous sentence, remove “var” with the value from assignment, remove the variables added by inferences from the assignment, reset “var” and all inferenced variables removed from the assignment to default domain (all possible values).

The inference function (inference) calls forward checking and AC3, and returns forward checking (returns boolean) and AC3 (returns boolean).

Forward checking keeps track of remaining legal values for all unassigned variables and terminates the search when any variable has no legal values. So, iterating through each variable in the assignment, “var”, check its neighbors. If “var” neighboring variable’s domain contains the value of “var”, remove it from the neighbor’s domain.

Forward checking propagates information from assigned to unassigned variables but doesn't provide early detection for all failures - unlike AC3. Constraint propagation repeatedly enforces constraints locally with its neighbors.

AC3 handles the situation where when you delete a value from the variable’s domain, it will check all the variables connected to that variable. If any of them change, delete inconsistent values connected to them. In the pseudo-code, the AC3 function has a local variable, “queue” - a queue of arcs, initially all the arcs in CSP. Arcs are one-directional so for each constraint (x, y) x<->y, there are 2 arcs, x->y and y->x for all the initial arcs in CSP. While the queue is not empty, pop the first element out of the queue (xi, xj). If remove-inconsistent-values(xi, xj), for each xk in neighbors of xi, add (xk, xi) to the queue. This is so that if you remove anything from a variable, you will need to add all the arcs (affected neighbors) that go into that variable back into the queue. Removing the inconsistent values (remove-inconsistent-values) will prune domain of xi based on xj. X Y is consistent if for every value x at X there is some allowed y, i.e., there is at least 1 value of Y that is consistent with x.

Removing the inconsistent values (remove-inconsistent-values) returns true if it succeeds. The function starts with “removed’ being assigned false (return removed at the end of the function). For each x in domain xi (for each value of xi domain) check if no value in y In domain xj allows (x,y) to satisfy the constraint xi <-> xj. If is done by if len(xj.domain) == 1 and x in xj.domain. If there doesn’t exist a value in xj domain that satisfies the constraint, then delete x (color value) from xi domain. Return “removed”.

Sources used: http://pages.cs.wisc.edu/~bgibson/cs540/handouts/csp.pdf, Lecture Slides

MinConflicts

The MinConglicts algorithm utilizes set max_steps and randomness avoid being stuck in local minima. We will call MinConflicts multiple times in a while loop for multiple tries. The while loop exists if MinConflicts returns a solution or over 60 seconds have passed.

MinConflicts, for i to max_steps, if current_state is a solution of CSP then return current_state (if a solution was found, return it and exit). Otherwise, set “var” to a randomly chosen variable from the set of conflicted variables, Conflicted[csp]. The conflicted variables are found by going through each constraint in CSP, if the two

variable values are the same, then both variables are conflicted and added to the array of conflicted variables. Then set “value” with the value v for var that minimizes conflicts. Conflicts are found by conflicts function which goes through each value of “var” - and counts how many conflicts the value assigned to “var” will cause. This is calculated by iterating through each constraint in CSP, check both variables in the constraint if the variable is in the assignment and is the same as the value of “var” then increase the counter by 1. The values are sorted by the least conflicts first. Then set “var” with the value in current_state.

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

DFSB algorithm

parameter set	number of states (mean± std)	actual time in ms (mean ±std)
N=20, K=4, M=100	211.15 ± 283.01948510194217	18.95675 ± 19.21173264570377
N=50, K=4, M=625	418.75 ± 512.8978430140071	136.63995 ± 166.60199076856796
N=100, K=4, M=2500	305.95 ± 252.9719091956758	360.5672 ± 337.3840686218788
N=200, K=4, M=10000	299.85 ± 127.96556897113089	1158.7319499999997 ± 634.6796568208726
N=400, K=4, M=40000	815.5 ± 715.8453596604286	12994.120849999998 ± 13639.1552366259

DFSB++ algorithm

parameter set	number of states (mean± std)	actual time in ms (mean ±std)
N=20, K=4, M=100	5.85 ± 3.8151360602073865	9.874050000000002 ± 3.5723671125518584
N=50, K=4, M=625	4.35 ± 0.48936048492959283	38.65345 ± 11.101439008074584
N=100, K=4, M=2500	4.8 ±	313.30410000000003 ±

	0.41039134083406165	72.71842823839882
N=200, K=4, M=10000	5.0 ± 0.0	2990.74625 ± 118.5068090030784
N=400, K=4, M=40000	5.0 ± 0.0	30498.091650000003 ± 1576.5542458673087

MinConflicts algorithm

parameter set	number of states (mean± std)	actual time in ms (mean ±std)
N=20, K=4, M=100	341.45 ± 566.3897184429593	39.88660000000001 ± 55.51525994800652
N=50, K=4, M=625	161.2 ± 32.21081871212316	122.67064999999998 ± 44.677204708388636
N=100, K=4, M=2500	367.0 ± 58.77342757976332	1043.1487499999998± 190.36275280987948
N=200, K=4, M=10000	846.2 ± 226.19451340546888	10309.6277 ± 2953.3861019602414
N=400, K=4, M=40000	No answer	No answer

3. Explain the observed performance differences.

The observation differences is that DFSB runs the slower than DFSB++ at smaller parameter sets, but takes longer time for larger parameter sets. This is caused because while DFSB++ will go through less states than DFSB but does more work for each state with its heuristics - MRV, LCV, Forward Checking, AC3. The work needed to do for arc consistency is $O(n^2 d^3)$ which would take time. DFSB (plain) has no heuristics implemented. MinConflicts goes through more states than DFSB++ because MinConflicts decides many things randomly. It keeps trying random assignments and going through it and minizing conflicts until it finds a solution. However, when MinConflicts does find a solution, it is found rather quickly, it is faster than DFSB and DFSB++ for larger sets.