CAS CS 440
Lec 11

CSP II (cont.)

1. Review

DFS-Search interface function

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

DFS helper function (to do the actual DFS)

```
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
```

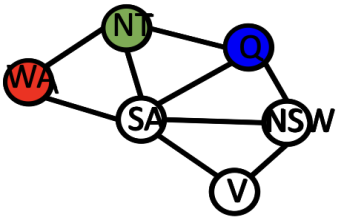
   a.
   b. Place holder
      i. Select-Unassigned-variable(csp) → give a variable in csp
      ii. Inference → discuss later
      iii. Order-Domain-Values
2. BackTrack
   a. Let's pretend no inference
   b. Static ordering of variables & domains
   c. Variable Order: [Q → NT → V → T →> WA → SA → NSW]
   d. Domain Order: [R → G → B]
   e. AC-3 only removes values in domain if it breaks constraint (the function might simply return the original graph → every value is consistent)
   f. Step by step
      i. Assign Red to Q and call backtrack
      ii. Try to assign Red to NT → break constraint (backtrack states no) → returns back to the previous state
      iii. Try to assign Green to NT → break consistent (backtrack states yes)
      iv. Try to assign Red to V → cont.

3. CSP Heuristics
    a. Heuristics in past problems
        i. Domain-specific knowledge
        ii. Task Engineering
        iii. If you change the world, you have to remake the heuristic function (cannot reuse the same heuristic)
    b. Heuristics in CSPs
        i. More abstract
        ii. For any problem, you can model CSP
        iii. There is no world-specific CSP (more powerful)
        iv. Apply to all CSPs
    c. You want the DFS to get solution fast → use heuristic (but want to perform better in the average case)
        i. Similar to Dijkstra and A* (A* is no different than Dijkstra in worst case but better in average case)
    d. Variable ordering
        i. Select-unassigned-variable
        ii. Goal: Prune the tree (as fast as possible)
            1. Rather than rely on a fixed ordering variables
            2. Pick new variable based on what other's have already been chosen
            3. Pick variable with smallest domain remaining (everything probably relies on the guy with smallest domain remaining)
                a. Minimum remaining values (MRV) heuristic
                b. This is not static ordering (based on where I am on the tree, domains might have different sizes compared to the same variable on upper and lower parts of the tree)
                c. Fail-first heuristic
            4. Degree heuristic
                a. Pick variable involved in the most constraints (the impact of setting variable can be measured by the number of degree)
            5. Can combine multiple heuristics
                a. Use MRV and settle ties with degree heuristic
                b. Least Constraining Value (LCV) heuristic:
                    i. Prefers domain values that affect the neighbor domains the least (if the neighbors or other variables are not blue, choose blue first)
                    ii. Fail-last heuristic

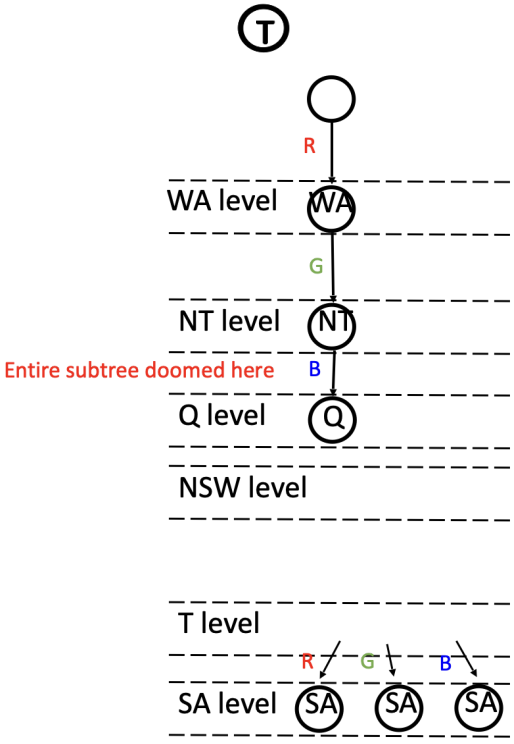
    e. Value ordering
        i. Order-domian-values
            1. Assign the most problematic variables first and find the solution
            2. Choose a value that is the most “flexible”
4. Inference During Search
    a. When I make n inference, spread that inference to the tree
    b. AC-3:
        i. If I choose blue, my neighbors can’t be blue, so do not bother to give them blue
        ii. When a domain was pruned:
            1. Propagate those changes to neighbors
        iii. This is an inference procedure!
    c. How can we leverage inference to prune during the search?
        i. Whenever we make an assignment:
            1. Infer domain reductions on neighboring variables!
5. Forward Checking
    a. A flavor of inference (the INFERENCE procedure)
        i. Whenever we add $X_i = v$ to assignment
        ii. Make neighbors X_j of X_i arc consistent with X_i
        iii. If one variable has value of Red, prune the variable red from its neighbors (do not consider the option of red to its neighbors)
        iv. If domain only has one constraint, it has to been assigned that value (assignment comes out for free) → it sometimes accelerates DFS search
    b. Note: this is useless if you ran AC-3 as a preprocessing step
        i. Very useful interference if you choose not to run AC-3 beforehand!
6. Maintaining Arc Consistency (MAC)
    a. Another flavor of inference (the INFERENCE procedure)
    b. It does everything that forward-checking does but forward-checking doesn’t do everything that MAC does
        i. Whenever we add $X_i = v$ to assignment
        ii. Call AC-3!
            1. Instead of the queue initially populated with all edges in the CSP:
                a. Queue initially populated with all edges to unassigned neighbors of X_i
                b. If the domain is removed from a variable, check whether the neighbors of that variable is arc consistent with that variable
                c. MAC says something changed; let’s propagate further
    c. Note: also useless if you do AC-3 as a preprocessing step
        i. Super useful if you don’t to AC-3 beforehand!

ii. Strictly more powerful than forward checking!
    1. MAC does everything forward checking does
    2. Forward checking does not do everything MAC does!

7. Better Backtracking
    a. We're running a DFS search
    b. When the recursive call fails:
        i. We return to the most recent DFS call
        ii. Try something else
        iii. "chronological backtracking"
    c. The problem is that the subtree we are exploring may already be doomed
        i. May have been doomed long ago
        ii. Continuing to explore this subtree is wasteful!
        iii. Goal: backtrack to the variable who doomed us!



d.



e.

f. Entire subtree doomed but was found later at the end (have to go all the way back)
    i. Going back one step back is not ideal (takes a lot of time) → forward checking wouldn't make it this bad
    ii. Want to jump all the way back to where it was doomed

8. How to Know When We're Doomed?
    a. Idea: Keep track of values that conflict with a variable
        i. Conflict set for SA = {WA=R, NT=G, Q=B}
        ii. When we get to SA and need to backtrack:
            1. Go to most recent addition to the conflict set for SA!
            2. DFS has to know to continue to skip all the way up
            3. Potentially skip over lots of other decision points in the tree!
    b. How can we compute these conflict sets?
        i. Forward checking gives them to us for free!
        ii. Assign $X_i = v$ and call forward checking (config of INFERENCE):
            1. Whenever we delete from X_j's domain: add $(X_i = v)$ to X_j's conflict set!
    c. But doesn't forward checking prune before we get here?
        i. Yes
        ii. Every branch pruned by forward checking is pruned by backjumping
        iii. "simple" backjumping is redundant when we use forward checking or MAC
9. How to Compute Conflict Sets?
    a. The idea of backtracking is still good:
        i. When you run into a conflict: don't chronologically backtrack
        ii. Backtrack to the source of the problem (the doomed point)
    b. The idea is not to forget future problems:
        i. Dooming a branch is not (necessarily) the fault of one individual variable
            1. It is not one constraint's fault, rather it is a combination of the constraints
        ii. Multiple variables together doom a branch
    c. Whenever we backjump:
        i. Use forward checking to compute conflict sets
        ii. When we backjump, modify conflict sets this way
        iii. Let's say the variable we backjump to is X_j
        iv. We came from X_i
        v. X_j's conflict set should "absorb" X_i's conflict set: (since it is not only that variable's problem but is a combination)
            1. There is no solution from X_j onward given the preceding assignments to X_j's new conflict set
        vi. $conf(X_j) = conf(X_j) \cup conf(X_i) - \{X_j\}$
        vii. Don't waste time changing the wrong variables
        viii. Avoid running into the same problem again
        ix. The variable might have affected other variables (or the variable at last) but it is not purely on my shoulder