Suggestions from the Project 2.1 Consultations

CS3243 AY24/25 Semester 1

3 perspectives in optimizing CSP for Project 2.1

- **Optimize CSP, a theoretical overview**
- **Optimize domain updates**
- **Optimal data structures for storing values/variables/constraints**

Ways to optimize CSP in theory

- Value ordering heuristics – LCV
- Variable ordering heuristics – MRV & degree heuristic
- Forward checking (FC)
- AC3 (not required for this project)

Comments:

1. AC3 is a preprocessing step. The other three occur within the recursion.
2. The logic for checking consistency is involved in multiple places:
   - Answer: LCV, FC, and also the moment right after assignment is made.

This implies: We can do them **simultaneously**! ⇒ If we have implemented forward checking correctly, then we **need not check consistency again** after we perform an assignment.

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

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
      if value is consistent with assignment then
          add {var = value} to assignment
          inferences ← INFERENCE(csp, var, assignment)
          if inferences ≠ failure then
              add inferences to csp
              result ← BACKTRACK(csp, assignment)
              if result ≠ failure then return result
              remove inferences from csp
          remove {var = value} from assignment
  return failure
```

The consistency checking in this line is not needed

A note on LCV

- This heuristic essentially requires that for each value in the domain of the current variable, we check it against every other value from each constraint that the current variable is involved in.
- This is a **<u>triply nested loop</u>**! It is very expensive to implement (indeed!), as right now for each value you arrive at, you impose a triply nested loop structure onto that node. (*plus its logic largely overlaps with consistency checking*)
- **Is LCV essential?** You see, the philosophy behind LCV is that we aim to place larger subtree to be closer to the root node, **but it is just an ordering**, it does not remove any values from the domain, nor does it eliminate any combinations. Worst case scenario we still have to traverse through every node.
- **<u>LCV is not necessary for this project.</u>**

How about degree heuristic & MRV then?

- Again, it is about the trade off between the time it could potentially save versus the time it takes to find that ordering.
- Similar to LCV, it simply imposes **an ordering that greedily search for the next unassigned variable** that will give the smallest subtree (domain size corresponds to the number of branches).
- **Benefit**: <u>Potentially</u> avoid some massive subtrees.
- **Cost**: It is used each time we want to select the next variable to be assigned. Can be implemented in O(n) time, where n refers to the number of variables.
- **Overall**: Since cost to implement is relatively cheap, and the benefit could be potentially substantial, we can indeed consider using MRV. (*degree heuristic is not essential as it is just a tie-breaker condition*).

Forward checking

- What FC does: Given a newly assigned variable together with the newly assigned value, we traverse through the involved constraints (with unassigned variables) and remove the inconsistent values in those domains.
- Given a variable, and its assigned value, forward checking can be done in a **doubly nested loop** (i.e. a for loop inside a for loop). (Provided you have stored constraint functions and assigned values in an optimal way)
- Somewhat superior compared to LCV and MRV as it *actually* prunes away inconsistent values, so it actually shrinks the search space even in the worst case scenario.
- Once we locate an inconsistent value when traversing through the doubly nested loop, how to **remove that in an efficient manner**?

(see next page)

Optimal way to update domain

- To remove the inconsistent value during forward checking, **list.remove()** is O(n) in python, this is undesirable!! Can we do any better?
- Yes! We can store those invalid values and valid values **separately**, we store those valid values to be used for the next level down the recursion, and we store the invalid values to add them back to the domain when we come back from the deeper recursion (it comes back after *encountering failure*). (Which data structure can help to achieve this functionality? I hope the answer is obvious.)
- Note that in the previous point, it removes the values without invoking the list.remove() function. This ensures the forward checking's time complexity remains at O(M*N), where M is the number of constraints, N is the domain size.

Ways to optimize CSP regarding efficient DS

Hints on potential Data Structures to be used:

- Look up time for constraint function involved given a pair (variable_1, variable_2) can be done O(1).
- Look up time to check if a variable is assigned can be done O(1).
- Loop up time to check the previously assigned value given the variable can be done in O(1).