Veronica Child & Liv Phillips
Artificial Intelligence
Prof. Howald
3/9/18

Assignment #4 Write-Up

Introduction

We used Python to build an algorithm that provides a mapping between actions and locations that satisfy a given set of constraints on a floor . Our algorithm performs a brute force search of all possibilities and also uses the most constraining edge optimization. We represented the floor that the HeatMiser operates on as a graph with each room as a node with its adjacent neighbors stored in a list.

CSP Formalization

HeatMiser Action Problem

- *Set of variables*, *X*: all of the rooms in the floor, specifically
    - Office 1, Office 2, Office 3, Office 4, Office 5, Office 6
    - Warehouse 1, Warehouse 2, Warehouse 3, Warehouse 4
- *Set of domains, D*:

  (a) change the temperature

  (b) change the humidity

  (c) pass through a location.
- *Set of constraints, C*:

  (a) whatever action that HeatMiser takes in a given location, the same action cannot be taken in an adjacent location

  (b) across multiple runs of HeatMiser, every location has the possibility of having its temperature or humidity changed given constraint (a).
- *Scope*: adjacent rooms

Analysis: Brute Force

Our brute force algorithm runs until each room has had each value associated with it in a valid assignment. In running, it performs the following actions:

1. Checks if all rooms have had each of the three possible values associated with it in at least one of the valid assignments produced thus far. If so, **break**. Else:
2. Randomly initialize in a room -> **current_room**
3. While not all rooms have actions associated with them:
    a. If there are actions that haven't been tried in **current_room**, then:

i. Randomly assign **current_room** one of those actions (fitting within the constraints). Add this action the history of **current_room.**
ii. Push **current_room** onto a stack.
iii. If there are neighbors of **current_room** that don't currently have actions associated with them:
    1. For all neighbors of **current_room** that don't currently have an action associated with them, push them onto the stack, too.
    2. Pop a neighbor from the stack, set this as **current_room**, and return to step **(3)**
iv. Else, if there are no neighbors left, check if all rooms have actions associated with them. If so, **break**. If not:
    1. Increment the number of *failures* that the algorithm has encountered, and go back to step **(3).**

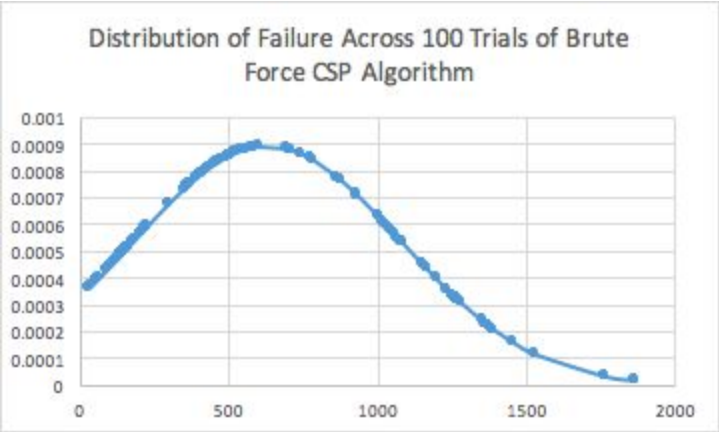
b. Else, if there are no actions left that haven't been tried in **current_room**, clear the action history of **current_room**, and increment the number of *failures* that the algorithm has encountered. Then, pop a room from the stack. Set this as **current_room**, and return to step **(3)**.

4. Go through each room in the valid assignment. If the action associated with the room has never been associated with it before, add the room to the appropriate list. Return to step **(1)**.

We can analyze the complexity of a constraint satisfaction problem as a search problem that incorporates backtracking. Backtracking, as mentioned above, is essentially a form of depth-first search with one variable assigned per node in our domain. In our HeatMiser problem, variables are colors and our domain of nodes are the different rooms. Given n variables and a domain size of d, the branching factor, the number of children at each node of our search, is nd. Our CSP has a branching factor of dn = 10 * 3 = 30 at the first level, and $(n-1)d$ = 9 * 3 = 27 at the next, which continues until all d items in the domain have been assigned a valid action. As a result, there are only d^n = 3^10 = 59049 complete assignments, which are evaluations that include all variables. However, our algorithm yields that there are only 6 solutions, which are evaluations that do not violate any constraints.

We ran our brute force search 100 times and tracked iterations -- how many runs our HeatMiser had to do before every room had been passed through, had its temperature changed, and had its humidity changed at least once, and failures -- the number of times the HeatMiser had to

backtrack during a round. The average number of iterations was 3.85 and the average number of failures was 627.27 (see fig 1).



(Figure 1: Distribution of Failures Across 100 Trials of Brute Force CSP Algorithm)

Analysis: Optimization

To optimize our algorithm, we implemented most constraining variable, which allows for better selection of the next variable that our algorithm should change. Before running, our optimized algorithm assigns constraint variables to each room, based on how many edges the room has. Then, our optimized algorithm runs until each room has had each value associated with it in a valid assignment. In running, it performs the following actions:

1. Checks if all rooms have had each of the three possible values associated with it in at least one of the valid assignments produced thus far. If so, **break**. Else:
2. Initialize in the most constrained room -> **current_room**
3. While not all rooms have actions associated with them:
    a. If there are actions that haven't been tried in **current_room**, then:
        i. Randomly assign **current_room** one of those actions (fitting within the constraints). Add this action the history of **current_room.**
        ii. Push **current_room** onto a stack.
        iii. If there are neighbors of **current_room** that don't currently have actions associated with them:
            1. For all neighbors of **current_room** that don't currently have an action associated with them, push them onto the stack, too, in order from least constrained to most constrained.
            2. Pop a neighbor from the stack, set this as **current_room**, and return to step **(3)**

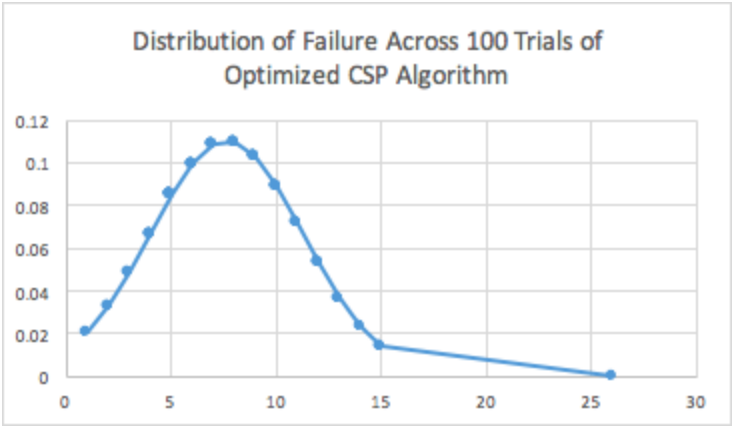
iv. Else, if there are no neighbors left, check if all rooms have actions associated with them. If so, **break**. If not:

1. Increment the number of *failures* that the algorithm has encountered, and go back to step **(3).**

b. Else, if there are no actions left that haven't been tried in **current_room**, clear the action history of **current_room**, and increment the number of *failures* that the algorithm has encountered. Then, pop a room from the stack. Set this as **current_room**, and return to step **(3)**.

4. Go through each room in the valid assignment. If the action associated with the room has never been associated with it before, add the room to the appropriate list. Return to step **(1)**.

As with brute force, we ran our optimized algorithm 100 times and tracked iterations and failure. For our optimized algorithm, the average number of iterations was 3.79 and the average number of failures was 7.63 (see fig 2). However, the optimized algorithm does have added complexity in the sense that there are more factors to keep track of, using more memory. Additionally, the algorithm must do extra computations to assign constraint factors to each room, which also increases the algorithm's complexity.



(Figure 2: Distribution of Failures Across 100 Trials of Optimized CSP Algorithm)

Recommendation

If "hardcoded with the results of [our] analysis (i.e. all possible states)", means that we could just give the six solutions we found to our HeatMiser, and tell it to go through a combination of them until all rooms had experienced all values, then this would obviously be the most efficient solution. However, this is clearly only valuable for the very specific arrangement of rooms that we have been

given. Obviously, hardcoding would fail on any other combination of room arrangement, or with any added values. Thus, while hard-coding would be most efficient for this very specific example, the optimized results would be much better for generalizability. Likewise, if any component of the brute force search would be involved in hard-coding, then the optimized algorithm would be a large improvement.