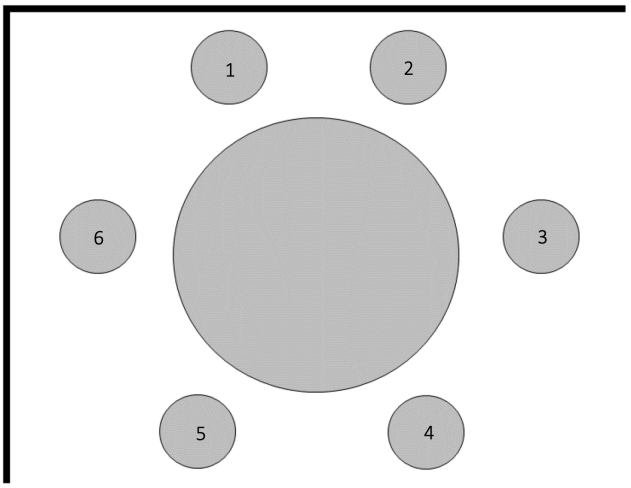
David Pérez Carrasco | 25984360
Lucas Amar | 37082823
21/07/2022

CPSC 322 Assignment 2

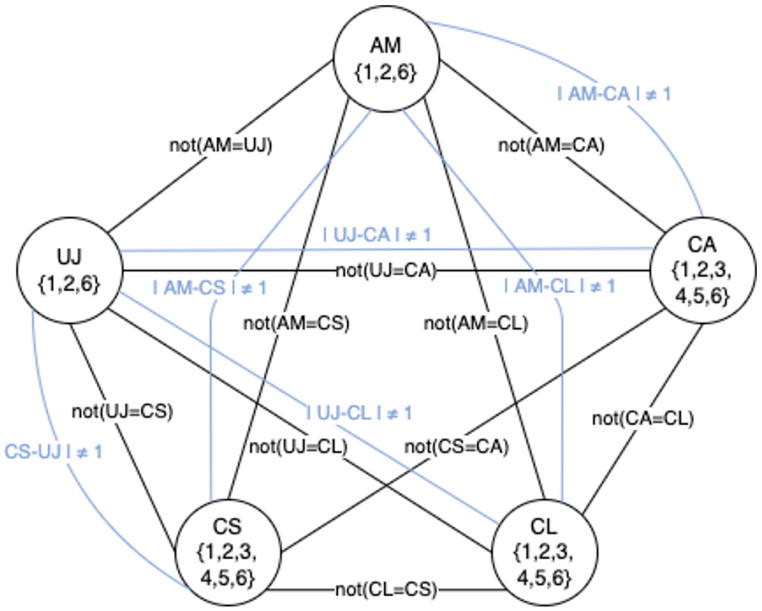
Question 1 [20 points] A Wedding Reception

1. CSP Seating arrangement – Assigning seat numbers as follows:



a. Make a CSP
   i. Variables : {UJ, AM, CA, CL, CS}
      1. Where UJ = Uncle Joe, AM = Aunt Marge, CA = Cara, CL = Claude, CS = Capt. Sweatpants
   ii. Domain : D_i = {1,2,3,4,5,6}
   iii. Constraints :
      1. UJ≠ {AM|CA|CL|CS}
      2. AM≠ {CA|CL|CS}
      3. CA≠ {CL|CS}
      4. CL≠ {CS}
      5. UJ≠ {3|4|5}
      6. AM ≠ {3|4|5}
      7. CA≠ {UJ+1|UJ-1|AM+1|AM -1}
      8. CL≠ {UJ+1|UJ-1|AM+1|AM -1}
      9. CS≠ {UJ+1|UJ-1|AM+1|AM -1}

b. Make a constraint network



Question 2 [35 points] CSP - Search

(a) [25 points] CSP Search Algorithm

Solutions found:

1. a=2, b=3, c=2, d=4, e=1, f=4, g=1, h=2
2. a=3, b=2, c=3, d=4, e=2, f=1, g=2, h=3
3. a=4, b=1, c=4, d=3, e=1, f=2, g=3, h=4
4. a=4, b=3, c=4, d=3, e=1, f=2, g=3, h=4

Number of failing branches= 1266

Code:

```
using Printf



function CSP_Search()

    fails = 0 #number of fails

    sols = 0 #number of solutions

    solutions = zeros(Int,(4^8,8)) #a matrix to store the solutions



    for a in 1:4

        @printf("A=%d ",a) #drawing the tree

        for b in 1:4

            @printf("B=%d ",b)

            for c in 1:4

                @printf("C=%d ",c)

                for d in 1:4

                    @printf("D=%d ",d)

                    if d != c #stating the constraints, if any is not fulfilled, the branch stops growing

                        for e in 1:4

                            @printf("E=%d ",e)

                            if e != c && e < d-1

                                for f in 1:4

                                    @printf("F=%d ",f)

                                    if (e-f)%2 != 0 && abs(f-b) ==1 && f!=c && d!=f-1

                                        for g in 1:4

                                            @printf("G=%d ",g)

                                            if a>g && abs(g-c) == 1 && g!=f d>= g

                                                for h in 1:4

                                                    @printf("H=%d ",h)

                                                    if a<=h && abs(h-c)%2 == 0 && h!=f && h!=d && g<h && e!= h-2

                                                        sols+=1

                                                        @printf("solution\n") #deport a success, all the constraints fulfilled

                                                        solutions[sols,:] = [a, b, c, d, e, f, g, h] #store the solution
```

```
                                        else
                                            fails+=1
                                            @printf("failure\n") #deport a failure
                                        end
                                        if h!= 4 @printf("                          ") end #assuring the tree structure
                                    end
                                else
                                    fails+=1
                                    @printf("failure\n")
                                end
                                if g!= 4 @printf("                      ") end
                            end
                        else
                            fails+=1
                            @printf("failure\n")
                        end
                        if f!= 4 @printf("                  ") end
                    end
                else
                    fails+=1
                    @printf("failure\n")
                end
                if e!= 4 @printf("               ") end
            end
        else
            fails+=1
            @printf("failure\n")
        end
        if d!= 4 @printf("            ") end
    end
    if c!= 4 @printf("        ") end
end
if b!= 4 @printf("    ") end
end
return solutions[1:sols,:], fails #we return our solutions in the matrix and the number of failures
end


#We get the output of the algorithm and report appropriately the solutions found and the number of failing branches
(sols, fails) = CSP_Search()
if size(sols)[1]>1
    for i in 1:size(sols)[1]
        @printf("\nSolution number %d:\n", i)
```

```
        @printf("a=%d, b=%d, c=%d, d=%d, e=%d, f=%d, g=%d, h=%d\n", sols[i,1], sols[i,2], sols[i,3], sols[i,4], sols[i,5], sols[i,6],
sols[i,7], sols[i,8])

    end

    @printf("Number of failures = %d\n", fails)

else

    @printf("\nThere are no solutions\n")

    @printf("Number of failing branches = %d\n", fails)

end
```

We did the program in Julia and it returned the search tree, solutions and number of failures reported above.

We know we could have done the print inside the function, but considered it better to return all the data and properly report the outcome outside the function.

(b) [5 points] Simple Variable heuristic

The degree heuristic that we are going to use is based on the Minimum remaining values (MRV), which consists in ordering all the variables choosing the variables with the fewest possible values first, in other words, assigning most constrained variable first, so the simple variable heuristic selection will be based on the number of constraints of each variable. However, we noticed that there are several variables with the same number of constraints, but we are still selecting the variables with the fewest possible values first, as MRV, which implies that when there is a draw we select first the variable whose constraints are more restrictive. Hence, if two variables have the same number of constraints, we will select the one that has more equality constraints, without counting the odd/even ones, (a=x), if there is still a draw then we will chose the one with more inequality constraints (a>x) or (a>b), then the one with more “greater/less than or equal to”, (a>=x) or (a>=b), then the one with more odd/even constraints and finally the one with more non-equality constraints (a!=b). So, for instance, f and h have both 6 constraints, but we will choose first f since it has a equality constraint (|f-b| = 1), and c, d and g have also 5 constraints but G and C both have an equality constraint and D does not, and G have two inequalities without the equal sign and C have none, so we will choose first G, then C and then D.

If we had to write a heuristic function, it would be something like f(n) = n_constraints, where n_constraints is the number of constraints of the variable n and we select first the variables with greater f(n), and in the case there is a draw then we use g(n) = n_equalities, where n_equalities is the number of equality constraint of the variable n, and we would choose the one with greater g(n), and if there is still a draw then we would do the same with inequalities, inequalities or equalities, even/odd constraints and inequalities.

Variable ordering obtained: F, H, G, C, D, E, A, B

Number of failing consistency checks: 177

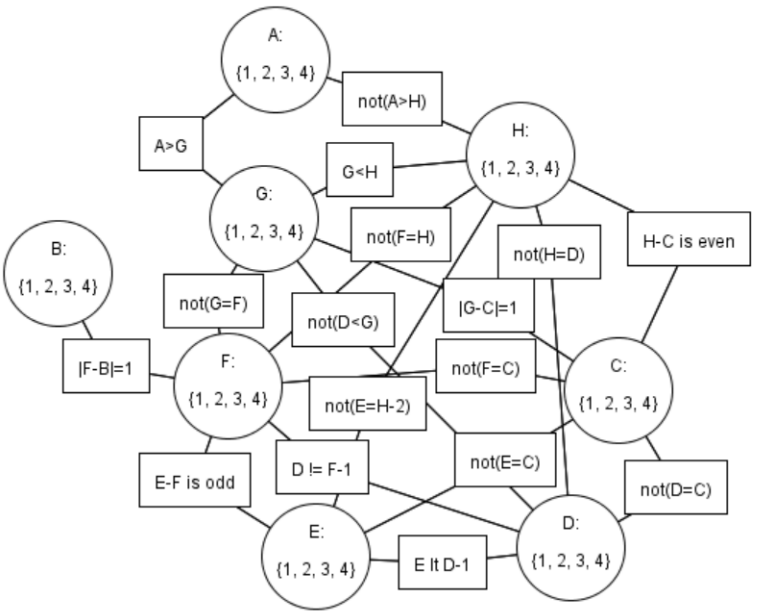
(c) [5 points] Why (b) heuristic is reasonable?

The chosen heuristic is reasonable because in order to reduce the tree branches we have to check first the variables that have more constraints, because then we will cut many possible combinations the earlier possible. The main fact that proves that this will be a reasonable heuristic is that its main functionality is pruning impossible assignments fairly early, avoiding to check impossible branches that would be checked in other order. Since the definition of MRW says, we choose the variable with fewer possible values first, making the tree the maximum smaller possible since the beginning.

Question 3 (16 points) CSP – Arc Consistency

(a) [5 points] Arc Consistency Graph

Initial Constraint Graph:



First 4 steps:

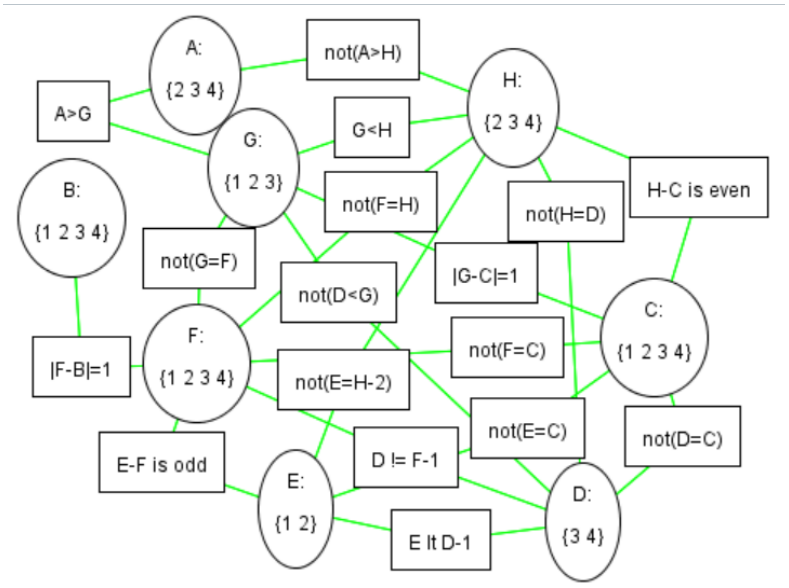
1st Step: 1 is removed from the domain of H by arc (H, G<H)

2nd Step: 4 is removed from the domain of G by arc (G, G<H)

3rd Step: No element is removed from the domain of C by Arc (C, |G-C|=1)

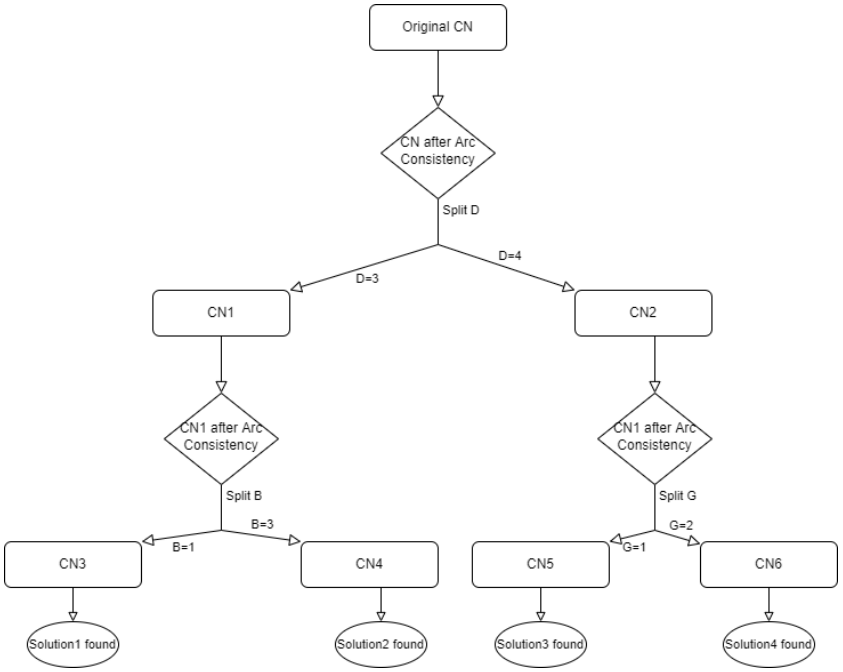
4th Step: No element is removed from the domain of G by Arc (G, |G-C|=1)

Final Constraint Graph:



(b) [5 points] Arc Consistency with Domain Splitting

Tree of splits:



Solution1 found: G = 3, H = 4, C = 4, D = 3, E = 1, F = 2, A = 4, B = 1

Solution2 found: G = 3, H = 4, C = 4, D = 3, E = 1, F = 2, A = 4, B = 3

Solution3 found: G = 1, H = 2, C = 2, D = 4, E = 1, F = 4, A = 2, B = 3

Solution4 found: G = 2, H = 3, C = 3, D = 4, E = 2, F = 1, A = 3, B = 2

(c) [5 points] Trade-off between DFS with pruning and arc consistency with domain splitting in terms of time/space complexity.

First, in terms of time, DFS is exponentially worse than Arc consistency, since it has to check every single node in the tree, while Arc Consistency removes all the impossible cases before checking every possible nodes, hence the number of nodes and constraints that DFS will check will be significantly greater than the one in Arc Consistency with domain splitting. Moreover, we have to consider that DFS is also reliant on a good degree heuristic, while Arc Consistency will get the best result by itself in all the cases. However, when it comes to space complexity, DFS with pruning is better than Arc Consistency with domain splitting, since DFS will have occupied space for at most the number of nodes in the last level, while Arc Consistency with domain Splitting has to consider all these nodes but some of the nodes of the previous levels as well, since it will have to come back to find another solution in the other domain splits. So in overall terms, DFS will be better for space complexity but worse for time complexity.

Question 4 (27 points) CSP – Stochastic Local Search

a. Values for each variable as {A,B,C,D,E,F,G,H}:

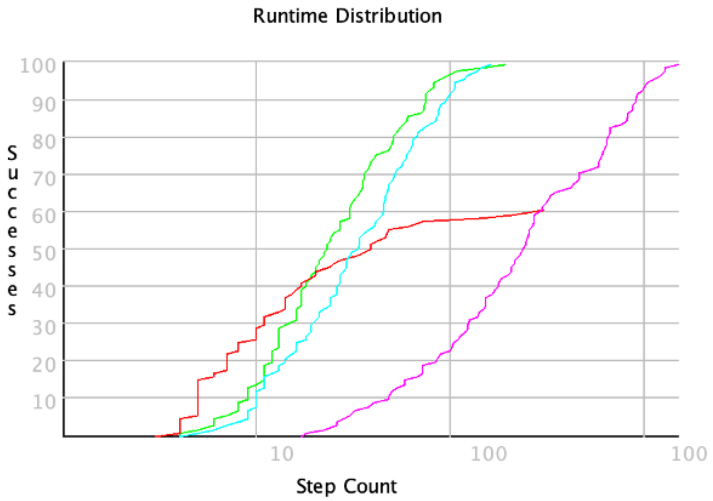
   a. Initialized values: {4,1,2,4,1,3,3,4} – 4 Conflicts

   b. Step 1: F = 2 (2 Conflicts)

   c. Step 2: D = 3 (1 Conflict)

   d. Step 3: C = 2 (0 Conflicts)

   e. No more steps, there are no more conflicts



b. In the graph curves for algorithms (i-iv) are represented as follows:

   i. Red Maximum number of unsatisfied constraints & best value

   ii. Cyan Random node with unsatisfied constraints & best value

   iii. Fuchsia Random node & best value

   iv. Green 25% max number of unsatisfied constraints, 75% random unsatisfied node & best value

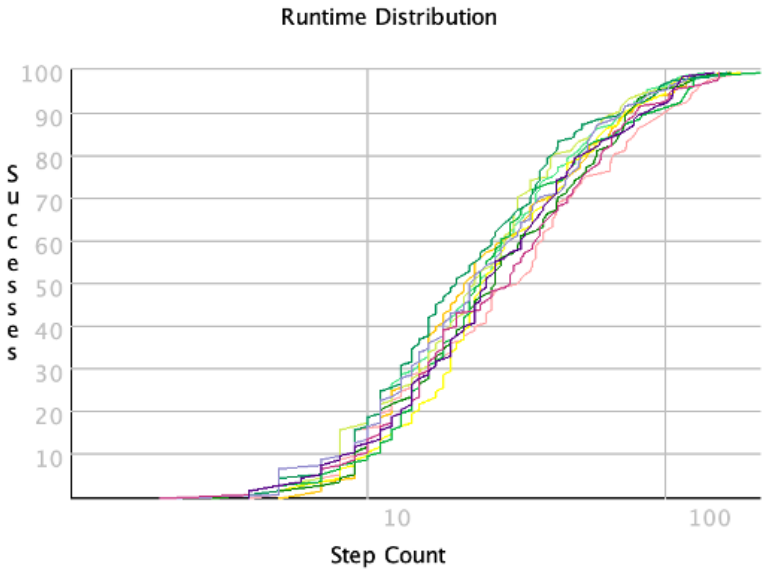
   b. As we can see, on the first 29 steps, the red curve is more efficient to solve the CSP. This is because when a solution is a few steps away, picking the path with the least number of unsatisfied constraints will get you to the solution quicker. However, it will not always work, as for some solutions you need to go to a state with less satisfied constraints, to reach the goal.

   c. For steps greater than 29, picking a random unsatisfied node seems to be the best option, given that it is more likely that when solving this one, you are not over optimizing and will reach a possible solution, while still making sure that you are changing values for a node that currently is unsatisfied. As seen on the graph this will solve almost all problems.

   d. Picking a random node each time will clearly be the least efficient method to find a solution out of the four, given it is highly likely (specially when close to a

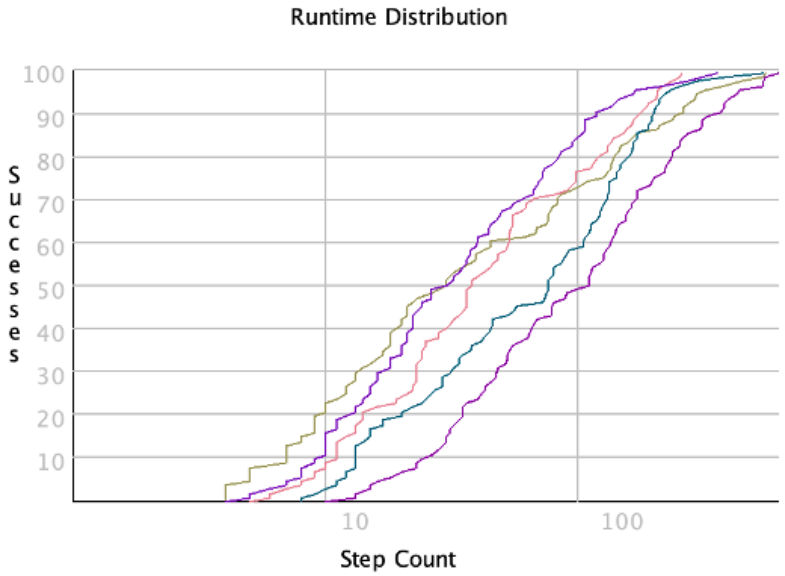
solution) that a node with a valid value will be changed, and consequently, a state with more unsatisfied constraints will be reached. However, it is still a relatively complete solution given that it will reach a valid solution approximately 98% of the time.

e. Finally, the mix I made will clearly be a mix of the red and cyan curves. It will be more efficient than the cyan curve given that it will pick the node with the most unsatisfied constraints 25% of the time, therefore tackling the larger problem, however it will not get stuck like the red curve because 75% of the time it will pick another unsatisfied node, and from there will be able to solve the problem in a way that doesn't over optimize the solution. I made the combination of these 2 because clearly the best line that could have been made from the first 3 is following the red one until the 29th step and then picking the cyan one. Thus, I decided to combine those 2.



c. I have chosen to run the algorithm selecting 25% the node with the maximum number of unsatisfied constraints and 75% a random node with unsatisfied constraints, and the best value. After running it 10 times, I got this graph:

a. Clearly, we can see that the curves follow the same trend, which is to be expected as it is the same algorithm that is being run. However, since initialization of the CSP is random, it will almost never be repeated. There are 8 variables each having a domain of size 4. Therefore, there are 4096 different initializations. Hence different problems will have to be solved each time, which explains why there is a slight variation in the curves, but why they still clearly follow the same trend.



d. To explain this, I have run various algorithms with different combinations of node with maximum unsatisfied nodes and random nodes as follows:

   i. Light purple (1st from the right) 20% max – 80% random

   ii. Dark green (2nd from the right) 40% max – 60% random

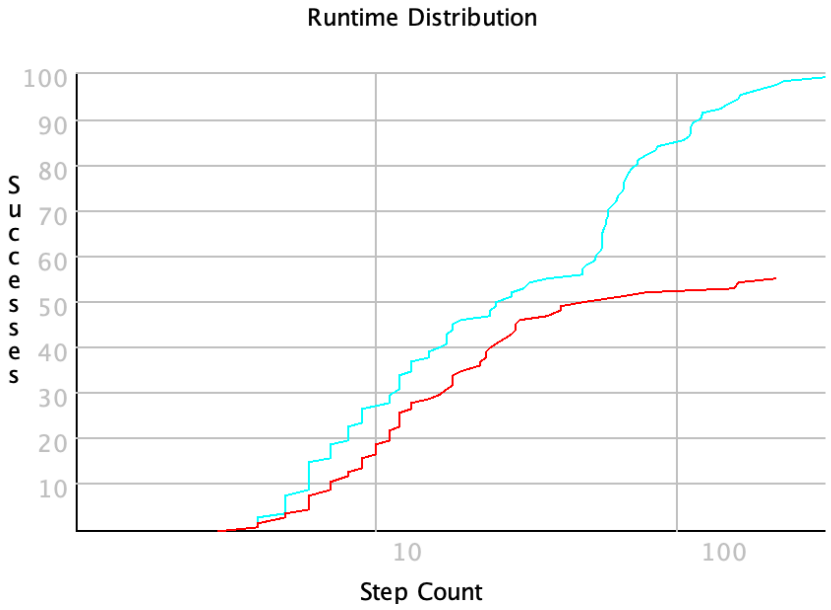
   iii. Pink (3rd from the right) 60% max – 40% random

   iv. Dark purple (4th from the right) 80% max – 20% random

   v. Greenish (leftmost curve) 95% max – 5% random

   b. With this, I have shown that a combination of the 2 is important. Picking more random than the node with the maximum number of unsatisfied nodes will result in a less efficient algorithm than finding a balance between the 2. We can see this clearly as, while moving from 20%-80% towards 80%-20% the algorithm clearly gets more efficient while still being complete. However, as seen from the greenish line, when we exaggerate the ratio and get to 95%-5%, after 25 steps the algorithm struggles to find efficient solutions. Therefore, it is very important to pick the value that results in the fewest unsatisfied nodes, but it is also important to not make it choose that node too often, otherwise it over optimizes and may get stuck in local "ditches", causing it to not find solutions quickly.

e. As we can see in the graph below, there isn't much of a difference between the 2 curves. The green curve is the one with no random restarts and the yellow one restarts every 50 steps. This is because I chose to apply method 4 from section b. Given that 75% of the time it will pick a random unsatisfied node to solve for, adding randomness into this model will not be the cause of a large difference. This model was already not getting stuck or over optimizing compared to the others in b, and clearly does not need random restarts every 50 steps to find solutions more efficiently.


Runtime Distribution
Successes
100
90
80
70
60
50
40
30
20
10
10
100
Step Count