CPSC 322 Assignment 1

Jose Abraham Torres Juarez - 79507828
Gustavo Martin - 62580121

1 Question 1 (21points): Allocating Developments Problem

(a) Represent this problem as a CSP

(a) **Variables:**

$$\{hc \text{ (Housing complex)}, bh \text{ (Big Hotel)}, ra \text{ (Recreation Area)}, gd \text{ (Garbage dump)}\}$$

(b) **Domains:**

$$\{(0,0),(0,1),(0,2),(1,0),(1,1),(1,2),(2,0),(2,1),(2,2)\}$$

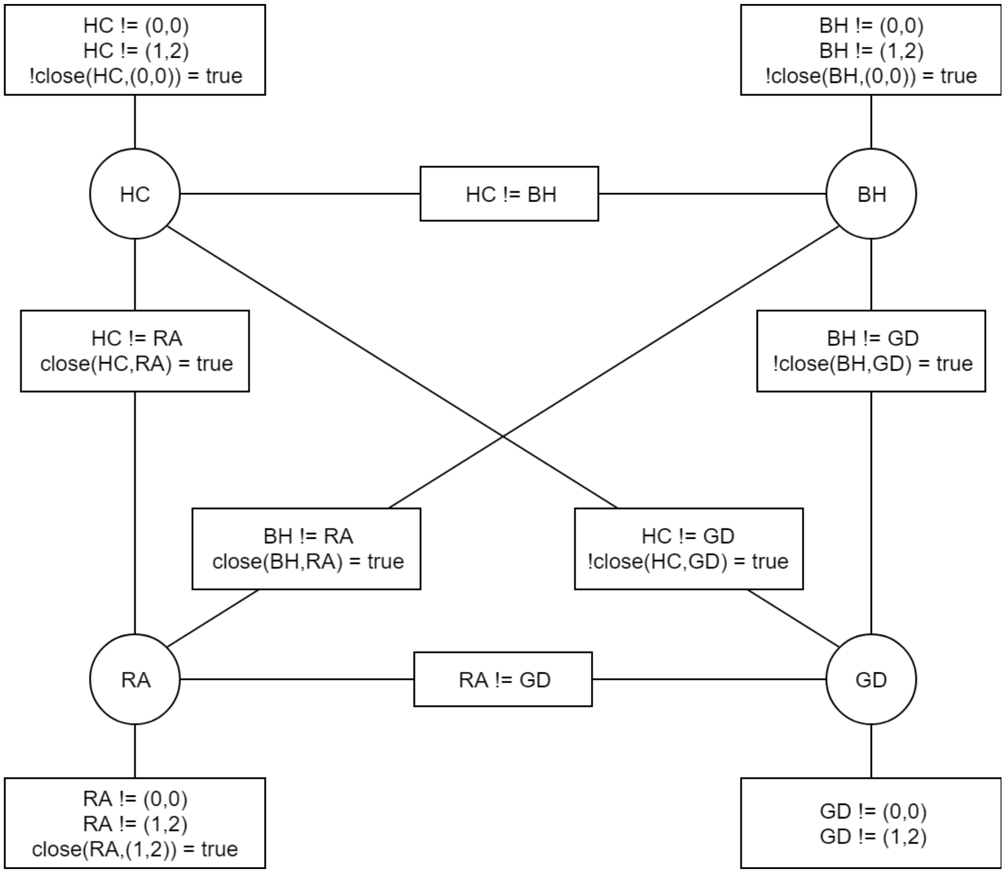
(c) **Constraints:**

$$\{hc \neq ra, hc \neq dg, hc \neq bh, bh \neq ra, bh \neq gd, ra \neq gd,$$
$$hc \neq (0,0), bh \neq (0,0), ra \neq (0,0), gd \neq (0,0),$$
$$hc \neq (1,2), bh \neq (1,2), ra \neq (1,2), gd \neq (1,2),$$
$$!Close(hc,(0,0)), !Close(bh,(0,0)), !Close(hc,gd), !Close(bh,gd)$$
$$Close(ra,(1,2)), Close(hc,ra), Close(bh,ra)\}$$

For our constraint definitions we use the following functions:

```
Function Close (A, B) // Returns true if the A and B coordinates share an edge
    // Check if $||A - B||_1 == 1$, the Manhattan distance must be 1
    return $(|A[0] - B[0]| + |A[1] - B[1]|) == 1$
```

(b) Draw a constraing graph for this problem


HC != (0,0)
HC != (1,2)
!close(HC,(0,0)) = true
BH != (0,0)
BH != (1,2)
!close(BH,(0,0)) = true
HC
HC != BH
BH
HC != RA
close(HC,RA) = true
BH != GD
!close(BH,GD) = true
BH != RA
close(BH,RA) = true
HC != GD
!close(HC,GD) = true
RA
RA != GD
GD
RA != (0,0)
RA != (1,2)
close(RA,(1,2)) = true
GD != (0,0)
GD != (1,2)

2 CSP - Search

(a) Show how search can be used to solve this problem, using the variable ordering A, B, C, D, E, F, G, H.
Number of fails: 1278
Number of solutions: 4
Solutions:

A	B	C	D	E	F	G	H
3	3	4	3	1	2	3	4
3	1	4	3	1	2	3	4
2	2	3	4	2	1	2	3
1	3	2	3	1	4	1	2

See **Appendix A** for the implementation of this `CSP`.

(b) Is it possible to generate a smaller tree?
Yes by selecting in different order the variables we assign.

The heuristic we applied to our variable selection, was use first the variables that appear in more constraints. By using this heuristic the tree had only **153** failing branches which is much better than **1278** from the previous approach with alphabetical ordering.

The new order of the variable selection is:

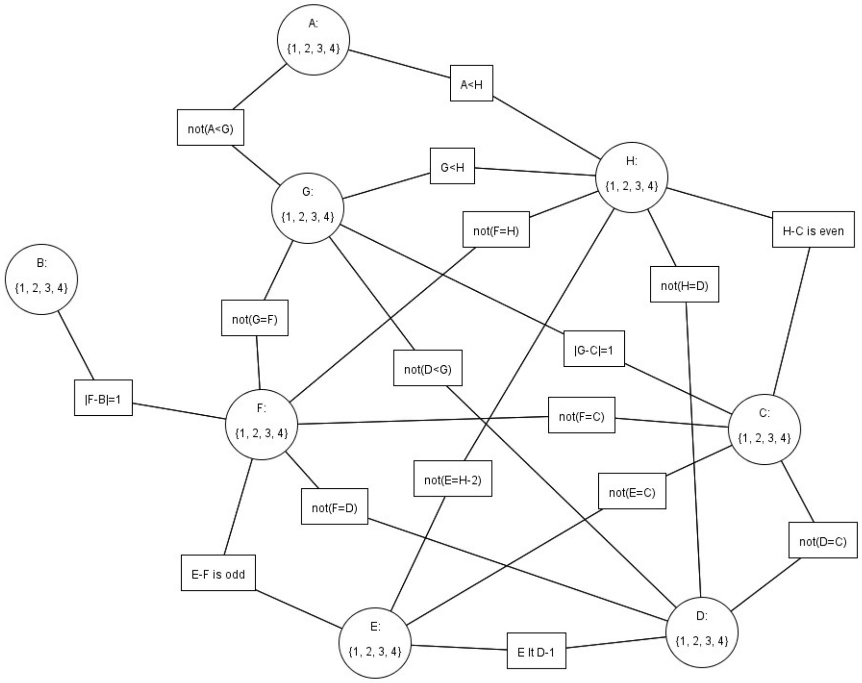
$$\{H, G, F, E, D, C, B, A\}$$

(c) Explain why you expect the heuristic in part (b) to be good The branches in the search tree stop being searched if they are a failure or a solution, for this problem we know that a solution will always be at the end on the tree, so we need to improve the chance of having failures on shallow branches this will prevent us from doing unncessary processing and will output a small tree.

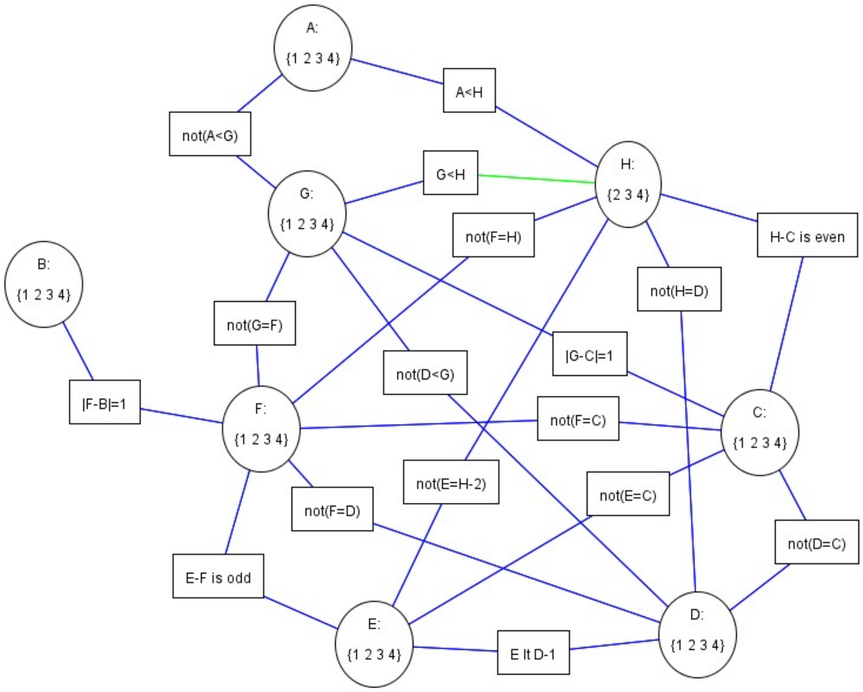
By using the heuristic of assigning a value first to the variables that appear the most in the constraints, we increase the chances for having a failing consistency check.

3 CSP - Arc Consistency

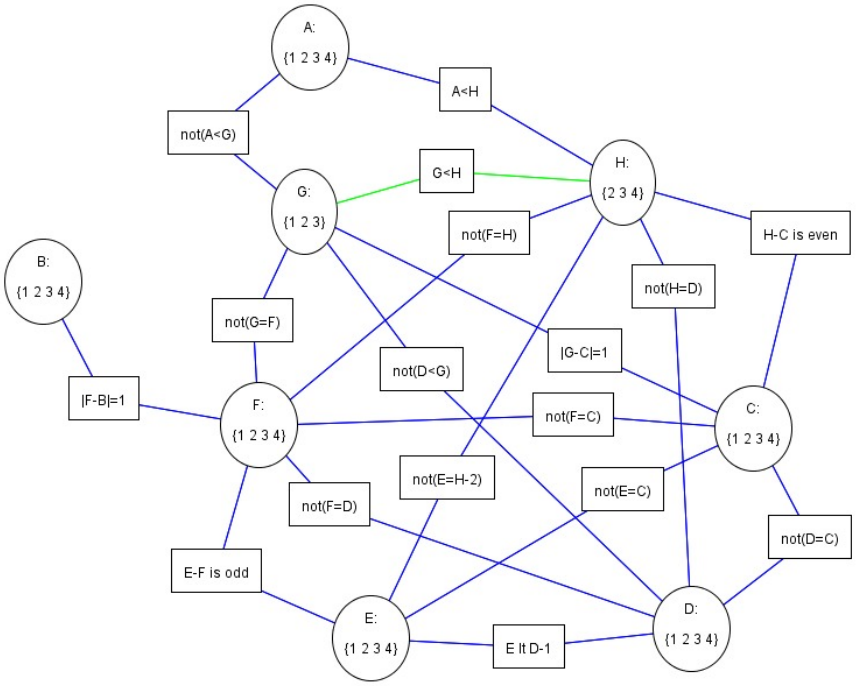
(a) Show how arc consistency can be used to solve the scheduling problem in Question 2



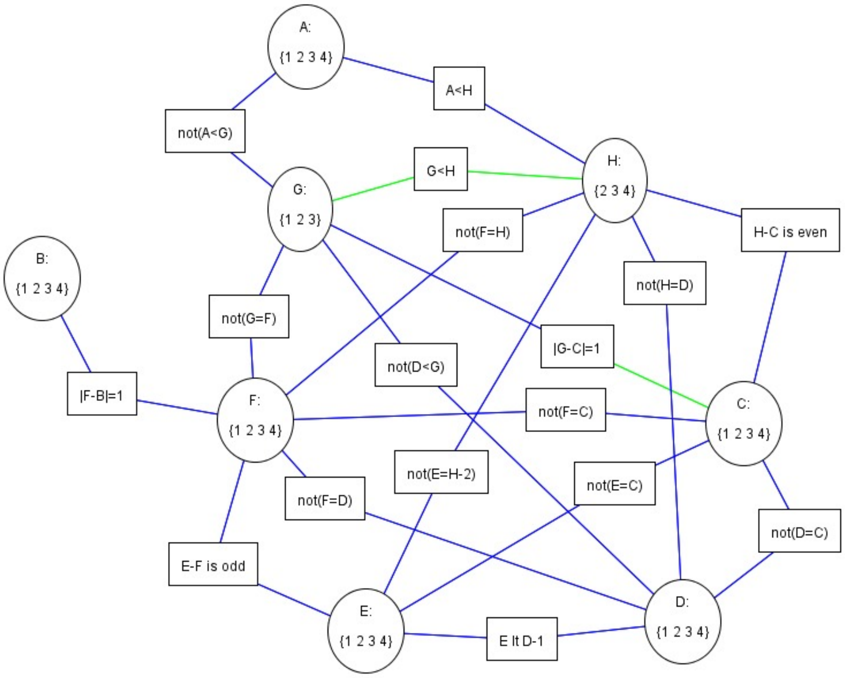
Initial state



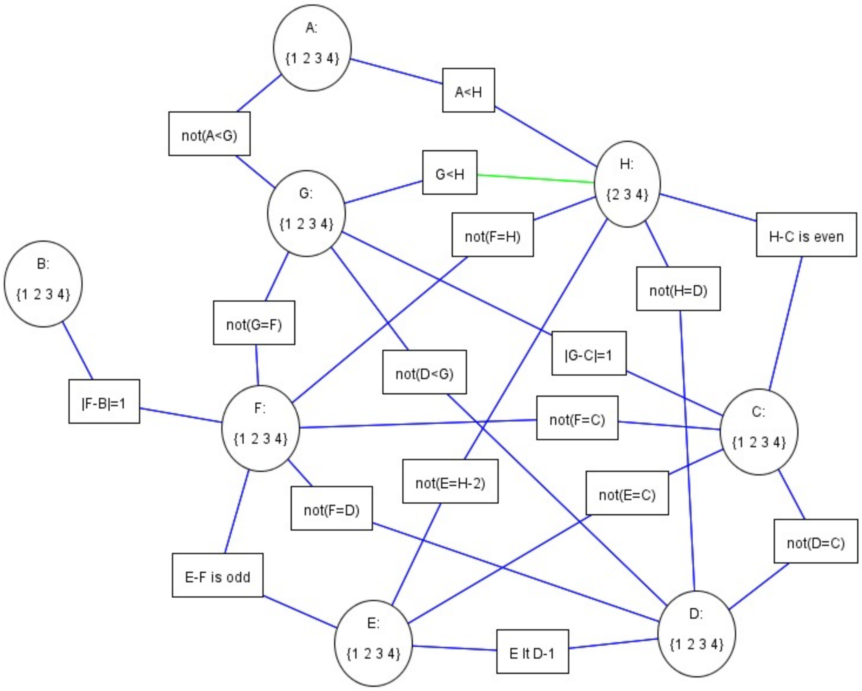
Step 1: arc $(H, G < H)$ was selected, 1 was removed from the domain of H.



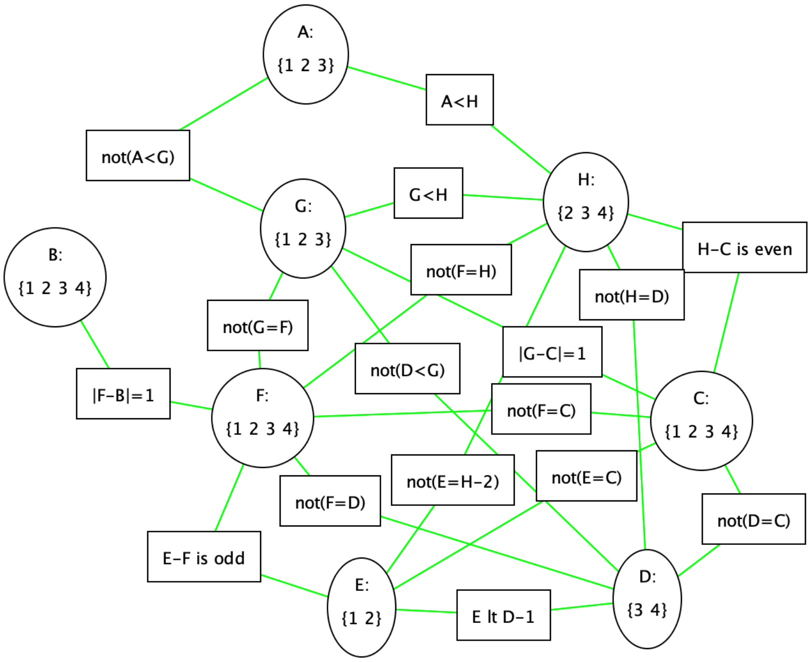
Step 2: arc $(G, G < H)$ was selected, 4 was removed from the domain of G.



Step 3: arc $(C, |G - C| = 1)$ was selected, nothing was removed.

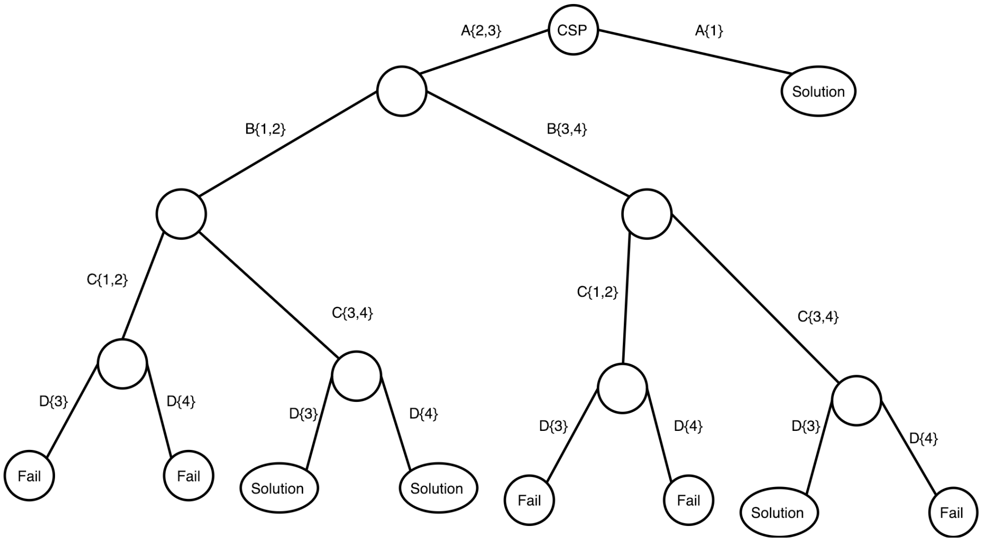


Step 4: arc $(G, |G-C|=1)$ was selectec, nothing was removed.



Final state

(b) Use domain splitting to solve this problem



Domain splitting graph

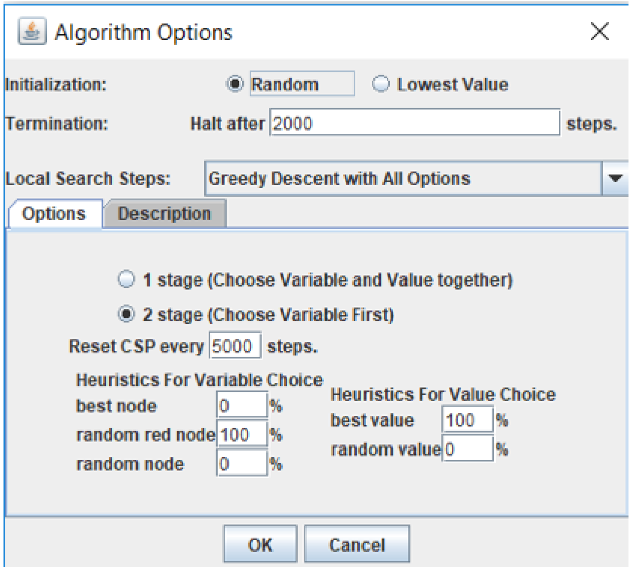
(c) Constraint satisfaction problems can become extremely large and complex. Given the choice between DFS with pruning and arc consistency with domain splitting, what (qualitatively) is the tradeoff between the two methods in terms of time/space complexity?

DFS with pruning is gonna have less space complexity than arc consistency with domain splitting, because on DFS at worst you keep track of the value assigned to all eight variables, whereas with arc consistency with domain splitting we have to track the domain of every variable separately.

On the other hand the domain splitting method works faster and arrive to a solution in fewer steps rather than using DFS. We can recall that in the subsection 2.a, the implementation of dfs with pruning had to do **1278** failing branches to get all the solutions, even by using a heuristic for selecting the variable ordering in subsection 2.b got **153** failing branches. The domain splitting method only had **5** failing branches as we can see on the image above.

4 CSP - Stochastic Local Search

(a) For one particular run, make SLS select any variable that is involved in an unsatisfied constraint, and select a value that results in the minimum number of unsatisfied constraints. Report the initialized values. At each step, report which variable is changed, its new value, and the resulting number of unsatisfied arcs. (You only need to do this for 5 steps).


Algorithm Options

Initialization: ● Random ○ Lowest Value

Termination: Halt after 2000 steps.

Local Search Steps: Greedy Descent with All Options

○ 1 stage (Choose Variable and Value together)

● 2 stage (Choose Variable First)

Reset CSP every 5000 steps.

Heuristics For Variable Choice

best node 0 %

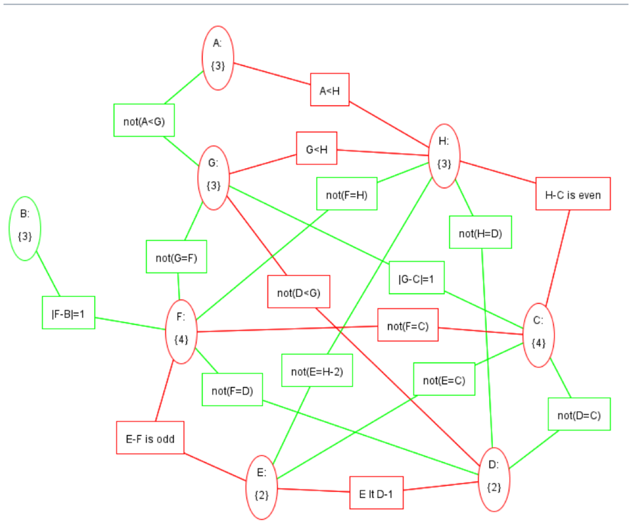
random red node 100 %

random node 0 %

Heuristics For Value Choice

best value 100 %

random value 0 %



Steps:

(1) Variable A changes value to 1, 14 unsatisfied arcs.

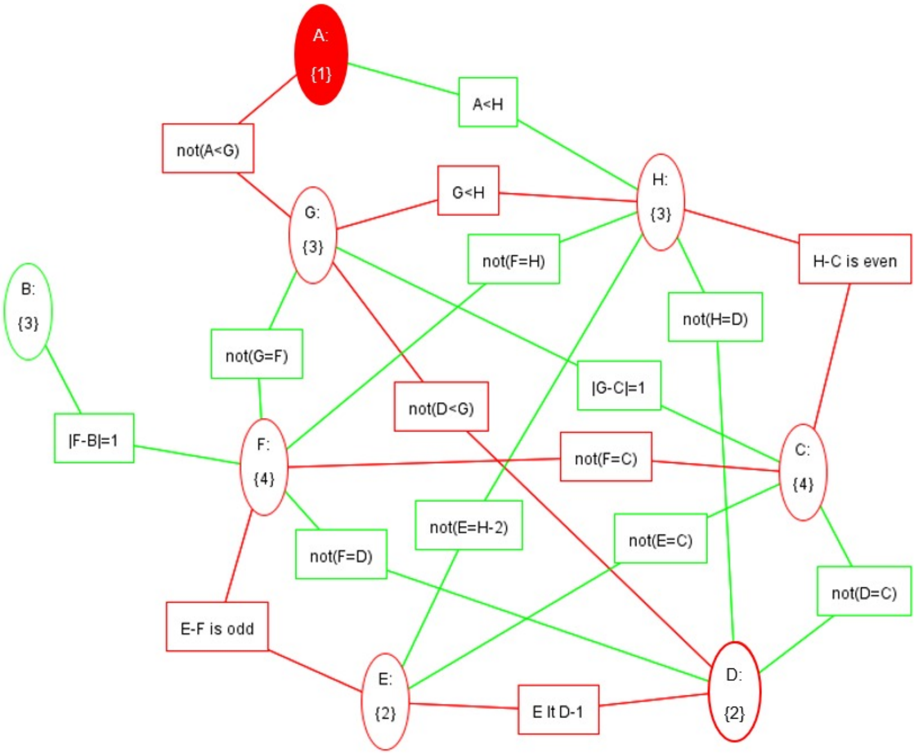
(2) Variable D changes value to 1, 14 unsatisfied arcs.

(3) Variable C changes value to 3, 12 unsatisfied arcs.

(4) Variable D changes value to 4, 10 unsatisfied arcs.

(5) Variable D changes value to 1, 12 unsatisfied arcs.

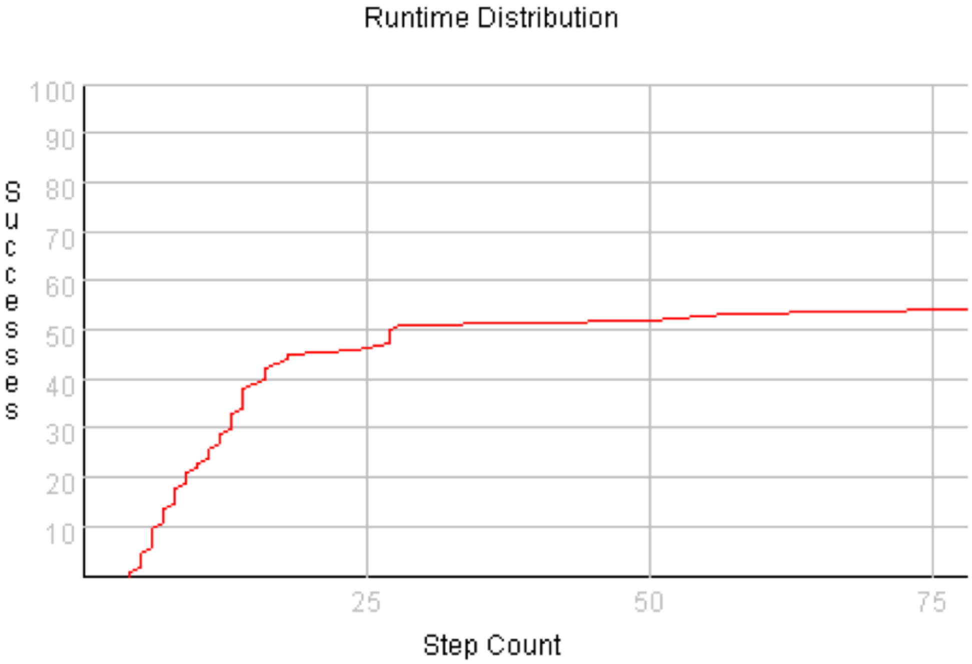
Final state:


A:
{1}
A<H
not(A<G)
G<H
H:
{3}
G:
{3}
not(F=H)
H-C is even
B:
{3}
not(H=D)
not(G=F)
|G-C|=1
not(D<G)
|F-B|=1
F:
{4}
not(F=C)
C:
{4}
not(E=H-2)
not(E=C)
not(F=D)
not(D=C)
E-F is odd
E:
{2}
E lt D-1
D:
{2}

(b) Using batch runs, compare and explain the result of the following settings:

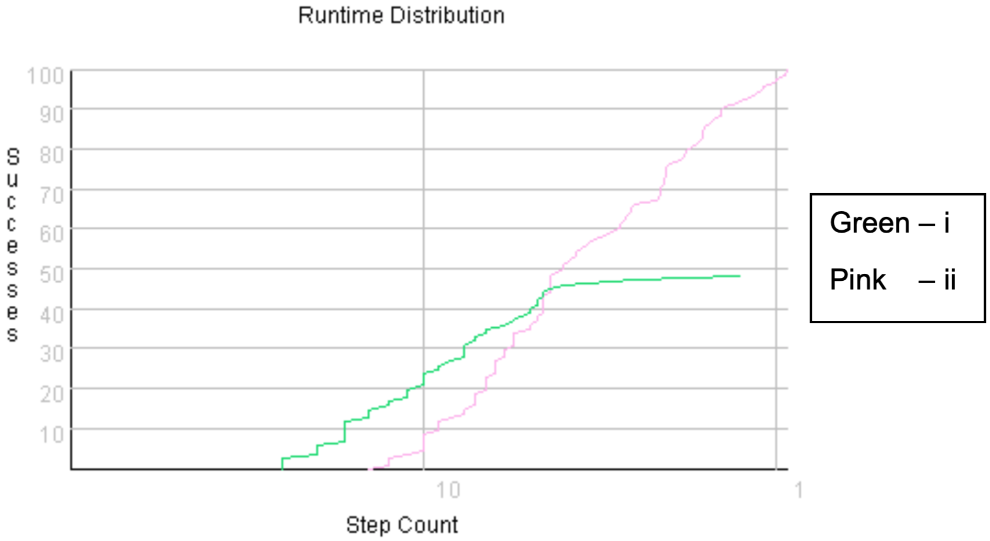
(i) Select a variable involved in the maximum number of unsatisfied constraints, and the best value.

If we select the best variable (Maximum amount of constraints) and the best value (Maximum constraints solved) the algorithm is going to get stuck 50% of the find and don't find a solution. This is because the solution can get stuck in a local minimum. If a solution is found, in the majority of the cases it will be before step 25.

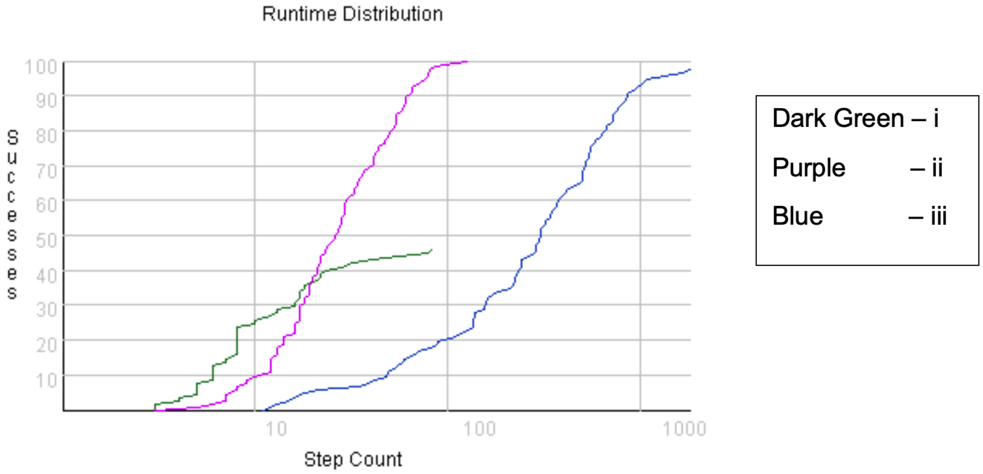


(ii) Select any variable that is involved in unsatisfied constraints, and the best value.

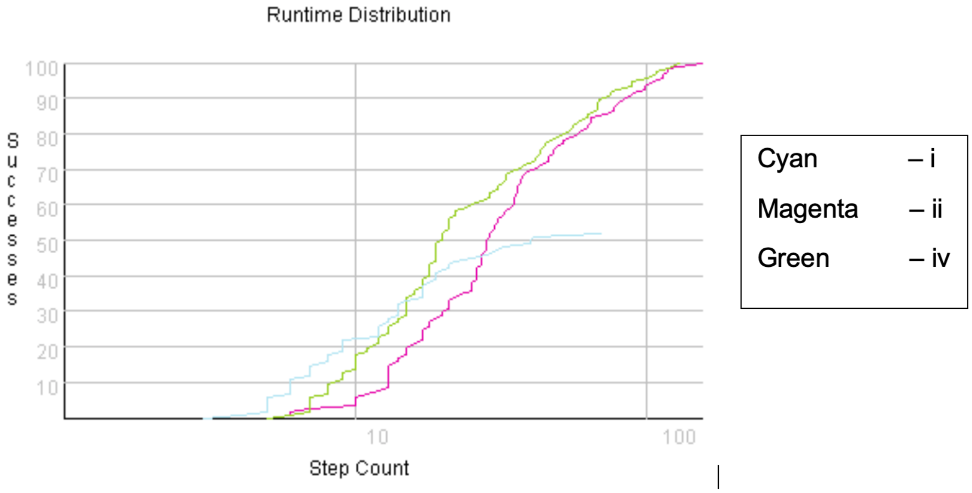
This method is slightly less efficient than using the node with the most amount of constraints when it finds a solution, but this algorithm will always find a solution.



(iii) Select a variable at random, and the best value.
This method is very inefficient compared to the previous methods used, it might take a large number of steps before finding a solution.

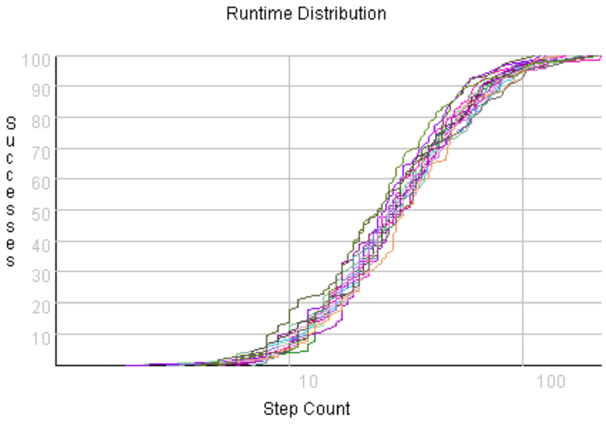


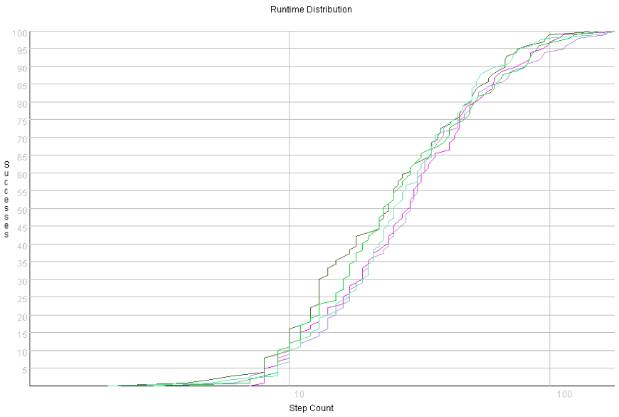
(iv) A probabilistic mix of i. and ii.Try a few probabilities and report on the best one you find
We found that the best results are achieved when you use the best node around 40%–60% of the time. For this comparison we select 50% of the time the best node and 50% a random node with unsatisfied constraints. The results found where that the efficiency is between method 1 and method 2 and this hybrid method will always find a solution.



(c) Pick one of the methods(ii)-(iv)from part (b) and run it repeatedly (5 runs should be enough, but you may do more runs if you wish). State which method you chose. Give a screenshot of the results; then think about, describe and explain whatyou observe.

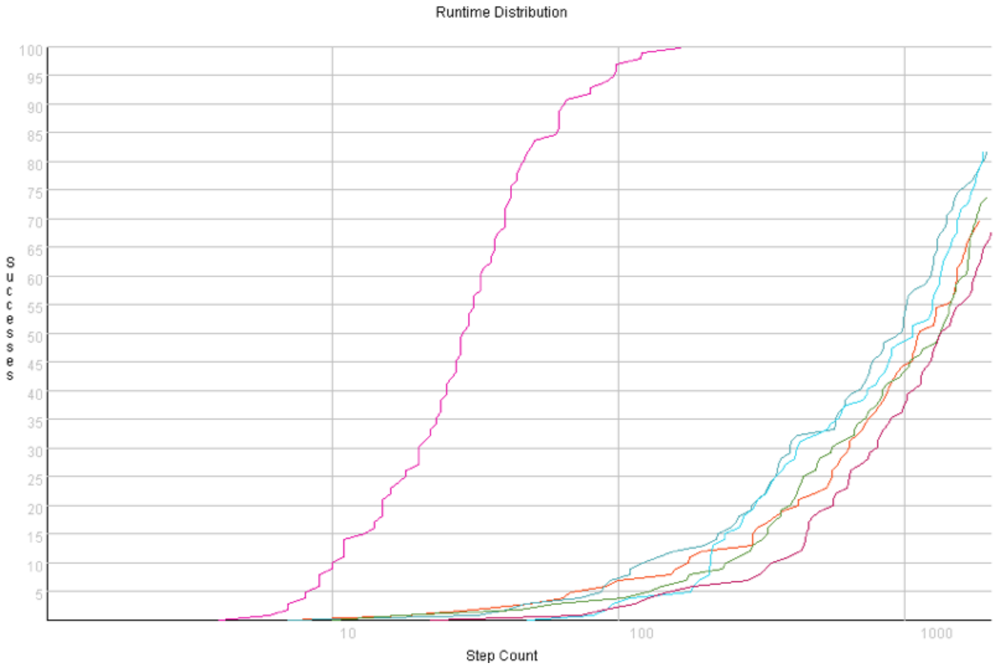
Results are consistent even though we are choosing random node with unsatisfied constraints, this is probably because even though we choose a random node, we are always choosing the value that will try to satisfy the maximum amount of constraints.





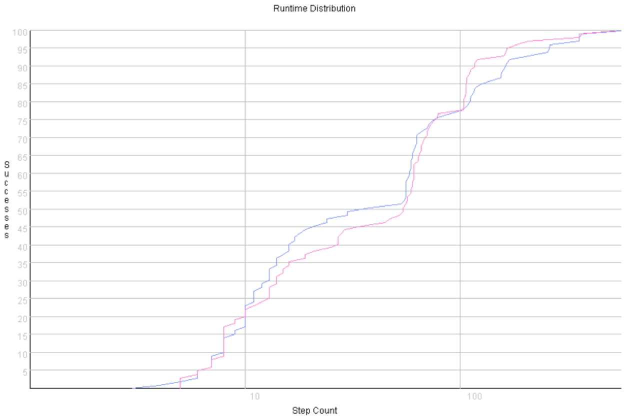
(d) How important is it to choose the value that results in the fewest unsatisfied constraints as opposed to choosing a value at random? Justify your answer with evidence.

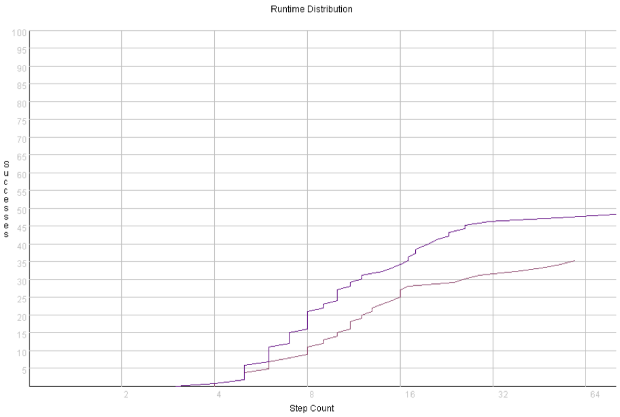
If we don't select the best value then our algorithm will rely just on randomness to find a solution, it will greatly impact efficiency and completeness. As evidence we provide a graph comparing method 2(purple) and just choosing a random value. We can not only see that method 2 is far more efficient, but also that choosing random values will greatly decrease our probability of success.



(e) For the best variable/best value method from part(b), allow random resets (for example, after 50 steps). Give a screenshot comparing the results to what happens if random resets are not allowed, and

explain how this affects the performance of the algorithm, and why.
If we allow random restarts at step 50 then we let the algorithm to get unstuck from local minima. On the graph on the left where we have random restarts enabled we can see that efficiency starts to stagnate at around step 40, then a random restart occurs, and our algorithm can continue finding a solution. We can probably decrease the step in where it restarts to improve efficiency. If we don't have random restarts, then the algorithm would not find a solution every time as we can see in the right graph.





A Appendix: CSP as Search implementation

```python
import sys
# Helper class to represent a variable assignment
class Node:
    def __init__(self, name, value=None, depth = 0, values = []):
        self.adjacents = []
        self.name = name
        self.value = value
        self.depth = depth
        self.values = values.copy()
        if depth > 0:
            self.values[depth-1] = value
    # Returns the name and value of the node, i.e., A = 1
    def description(self):
        if self.depth == 0:
            return ""
        return "{}={}".format(self.name,self.value)
    # Links node to another one
    def link(self, node):
        self.adjacents.append(node)

"""
 Solve the CSP
"""
def solveCSP(useHeuristic = False, file = None):
    # Problem configuration
    Domain = [1,2,3,4]
    varNames = ['A','B','C','D','E','F','G','H']
    constraints = {
        'AG':lambda A, G : A >= G,
        'AH':lambda A, H : A < H,
        'FB':lambda F, B : abs(F-B) == 1,
        'GH':lambda G, H : G < H,
        'GC':lambda G, C : abs(G-C) == 1,
        'HC':lambda H, C : abs(H-C) % 2 == 0,
        'HD':lambda H, D : H != D,
        'DG':lambda D, G : D >= G,
        'DC':lambda D, C : D != C,
        'EC':lambda E, C : E != C,
        'ED':lambda E, D : E < D-1,
        'EH':lambda E, H : E != H-2,
        'GF':lambda G, F : G != F,
        'HF':lambda H, F : H != F,
        'CF':lambda C, F : C != F,
        'DF':lambda D, F : D != F,
        'EF':lambda E, F : abs(E-F) % 2 == 1,
    }
    mapVarNames = {'A':0,'B':1,'C':2,'D':3,'E':4,'F':5,'G':6,'H':7}
    varN = len(varNames)

    # Helper that check if constraints are valid for the provided values
    def checkConstraints(values):
        for key, constraint in constraints.items():
            X = values[mapVarNames[key[0]]]
            Y = values[mapVarNames[key[1]]]
        if X == None:
            continue
        if Y == None:
            continue
        if not constraint(X,Y):
            return False
    return True

# Helper that generates a new order for the variables to assign
def varNamesByHeuristic():
    heuristics = {}
    newNames = []
    mapNewNames = {}
    for i in range(varN):
        varName = varNames[i]
        heuristics.update({varName:0})
        for key, constraint in constraints.items():
            if varName in key:
                heuristics[varName] += 1
    i = 0
    for key in sorted(heuristics.keys(), reverse=True):
        mapNewNames.update({key: i})
        newNames.append(key)
        i += 1

    return newNames, mapNewNames

# CSP Solve setup
root = Node("Root", values=[None]*varN)
frontier = [root]
solutions = []
fails = 0
# For formatting correctly the tree structure
newLine = False
# If heurisitc is needed, generate new ordering for the variables
if useHeuristic:
    varNames,mapVarNames = varNamesByHeuristic()
# If [file] is provided, the tree description will be written on a file with the name [file]
f = None
if file != None:
    f = open(file,"w+")

while(len(frontier)>0):
    n = frontier.pop()
    # Format the node description (Variable=Value)
    nTabs = 1
    if newLine:
        nTabs = n.depth
        newLine = False
    description = "{}{}".format("\t"*nTabs, n.description())
    # Write to terminal or the file
    if file != None:
        f.write(description)
    else:
        print(description, end="")
        # Check if the current assigned variables are valid with constraints
        if not checkConstraints(n.values):
            # Write to terminal or the file
            if file != None:
                f.write("failure\n")
            else:
                print("failure")
            fails += 1
            newLine = True
            continue

        # Check if we hit a solution
        if n.depth == varN:
            # Write to terminal or the file
            if file != None:
                f.write("solution\n")
            else:
                print("solution")
            solutions.append(n.values)
            newLine = True
            continue

        # Add to frontier the next variable, assigning every posible domain value
        for i in range(len(Domain)):
            new = Node(varNames[n.depth], Domain[i], n.depth+1, n.values)
            n.link(new)
            frontier.append(new)
    f.close()
    return varNames, solutions, fails

if __name__ == "__main__":
    if len(sys.argv)>1:
        # Use the heuristic for variable selection
        variables, solutions, fails = solveCSP(True, "../data/q2_tree_h.txt")
    else:
        # Use alphabetical ordering for variable selection
        variables, solutions, fails = solveCSP(False, "../data/q2_tree.txt")
    varN = len(variables)

    print("\nNumber of fails: {}".format(fails))
    print("\nNumber of Solutions: {}".format(len(solutions)))
    # Print the solutions
    print("\nSolutions:\n")
    for i in range(varN):
        print("\t{}".format(variables[i]),end="")
    print()
    for i in range(len(solutions)):
        for j in range(varN):
            print("\t{}".format(solutions[i][j]),end="")
        print()
```