Mitchell Chan 10753135
Peter Chung 38777124

CPSC322
Assignment 2

1.

a. ***Functions***

i) All = HC && BH && RA && GD

ii)
```
define notClose(O2)
    (assuming that x & y are within domain)
    if (O2.X - 1 = O1.X && O2.Y - 1 = O1.Y)
        not close
    elif (O2.X + 1 = O1.X && O2.Y + 1 = O1.Y)
        not close
    elif (O2.X - 1 = O1.X && O2.Y + 1 = O1.Y)
        not close
    elif (O2.X + 1 = O1.X && O2.Y - 1 = O1.Y)
        not close
    elif (O1.locn = O2.locn)
        not close
```

iii)
```
define close(O2)
    (assuming that x & y are within domain)
    if (O2.X = O1.X && O2.Y = O1.Y + 1)
        close
    elif (O2.X = O1.X && O2.Y = O1.Y - 1)
        close
    elif (O2.X + 1 = O1.X && O2.Y = O1.Y)
        close
    elif (O2.X - 1 = O1.X && O2.Y = O1.Y)
        close
```

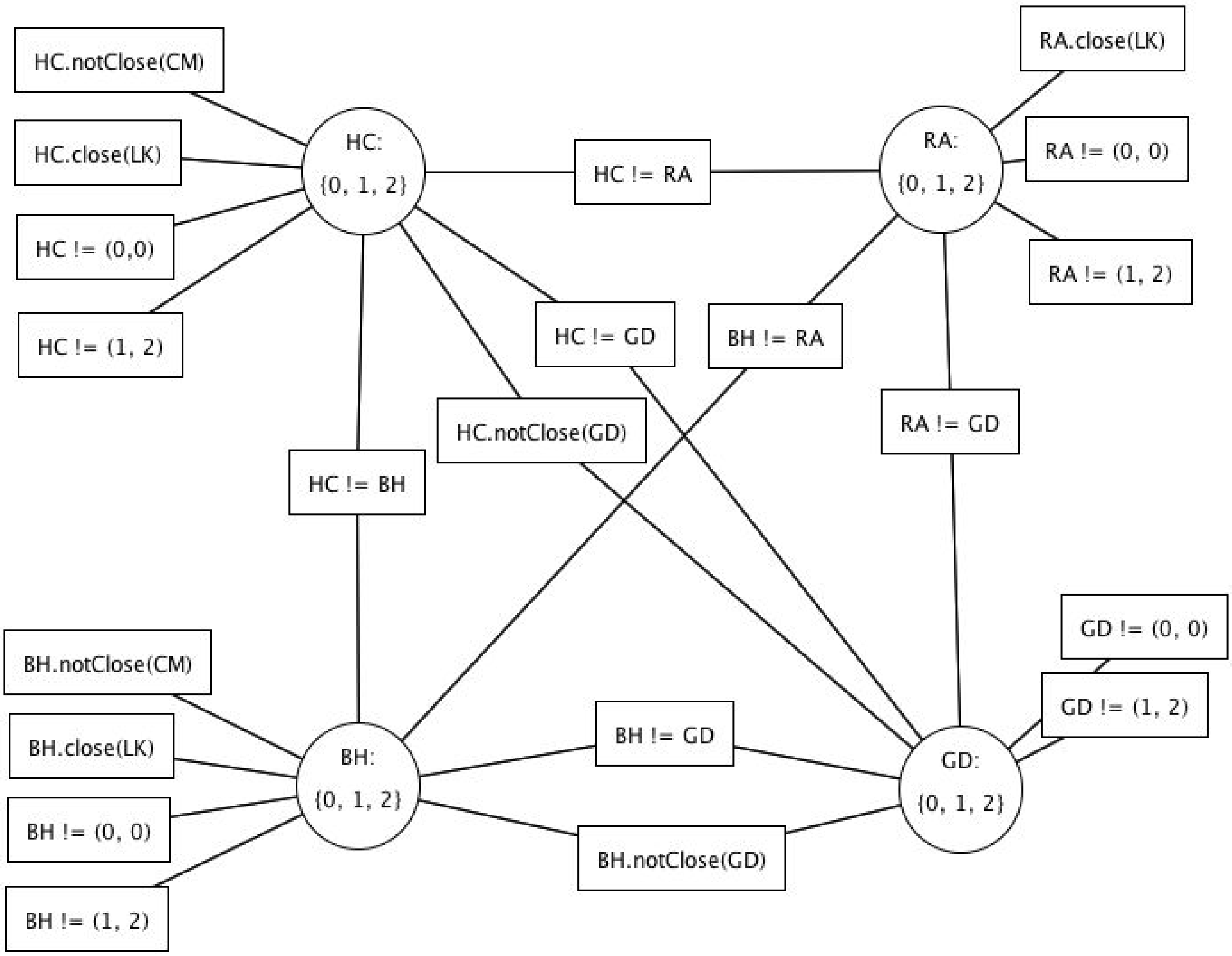
Variables: HC, BH, RA, GD, CM, LK

Domains: 0, 1, 2 for row (X) and 0,1,2 for column (Y)

Constraints: Obj1.locn != Obj2.locn
HC.notClose(CM)
BH.notClose(CM)
HC.notClose(GD)
BH.notClose(GD)
RA.close(LK)
HC.close(LK)
BH.close(LK)
0 <= X <= 2
0 <= Y <= 2
All != (0, 0)

All != (1, 2)

b.


HC.notClose(CM)
HC.close(LK)
HC != (0,0)
HC != (1, 2)
HC:
{0, 1, 2}
HC != RA
RA.close(LK)
RA:
{0, 1, 2}
RA != (0, 0)
RA != (1, 2)
HC != GD
BH != RA
HC.notClose(GD)
RA != GD
HC != BH
GD != (0, 0)
GD != (1, 2)
BH.notClose(CM)
BH.close(LK)
BH != (0, 0)
BH != (1, 2)
BH:
{0, 1, 2}
BH != GD
GD:
{0, 1, 2}
BH.notClose(GD)

Mitchell Chan 10753135
Peter Chung 38777124

2.

a. Written in java, iterative approach. See the attached pdf for the code snippet.
4 total successes found, correct values for each variable:

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

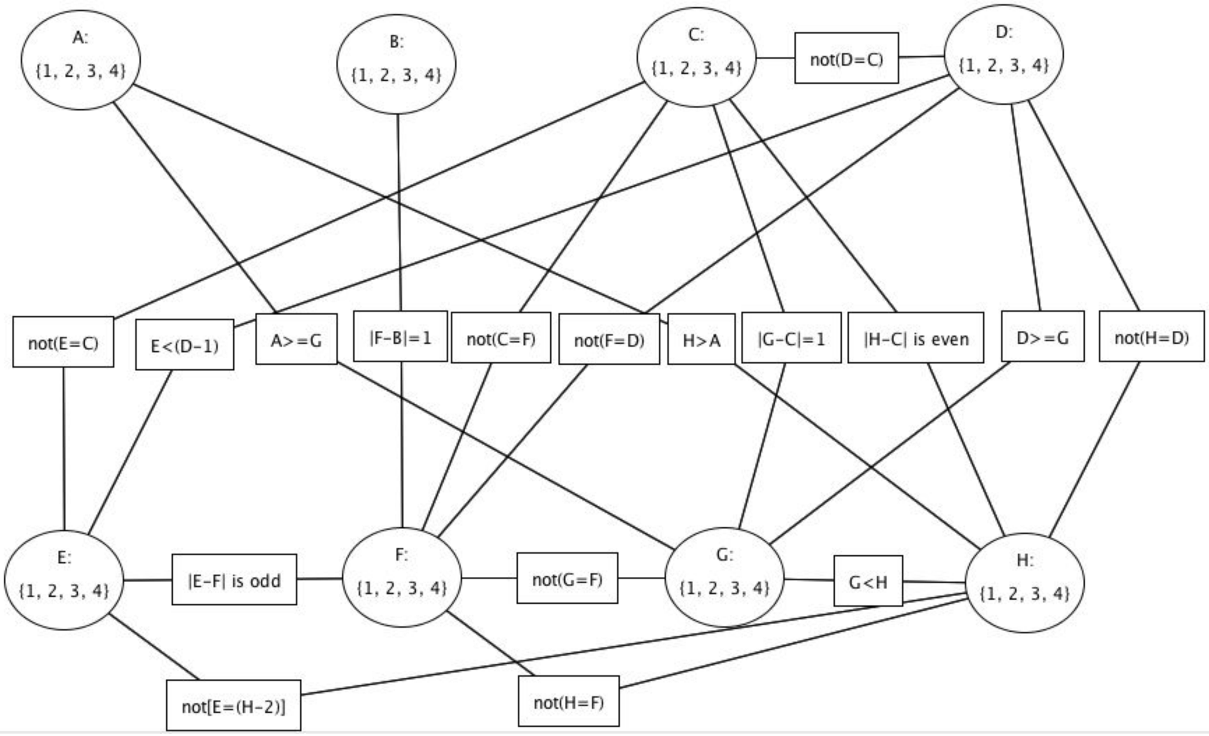
Total number of failures : 65532

b. A good heuristic would be the following: whenever a variable is assigned a value, we want to keep track of 2 things. One is the number of constraint not being satisfied due to the new assignment of that variable and second, the number of adjustments required for other variables to now satisfy all the constraints (for the case where there is none, this could have a really large heuristic value hence we don't take this route).

c. Above heuristic would be a good approach because, by assigning a really huge heuristic for impossible value assignments, we can filter out the subset of assignments to variables where they would never satisfy the constraints UNLESS they were given some exact values.

3.

a.



Step 1

Select Arc (G, G<H)

Inconsistent

Remove 4 from G because 4 in G not less than an element in H

Step 2

Select Arc (H, G<H)

Inconsistent

Remove 1 from H because 1 in H not greater than an element in G

Step 3

Select Arc (C, |G-C|=1)

Consistent

No deletion

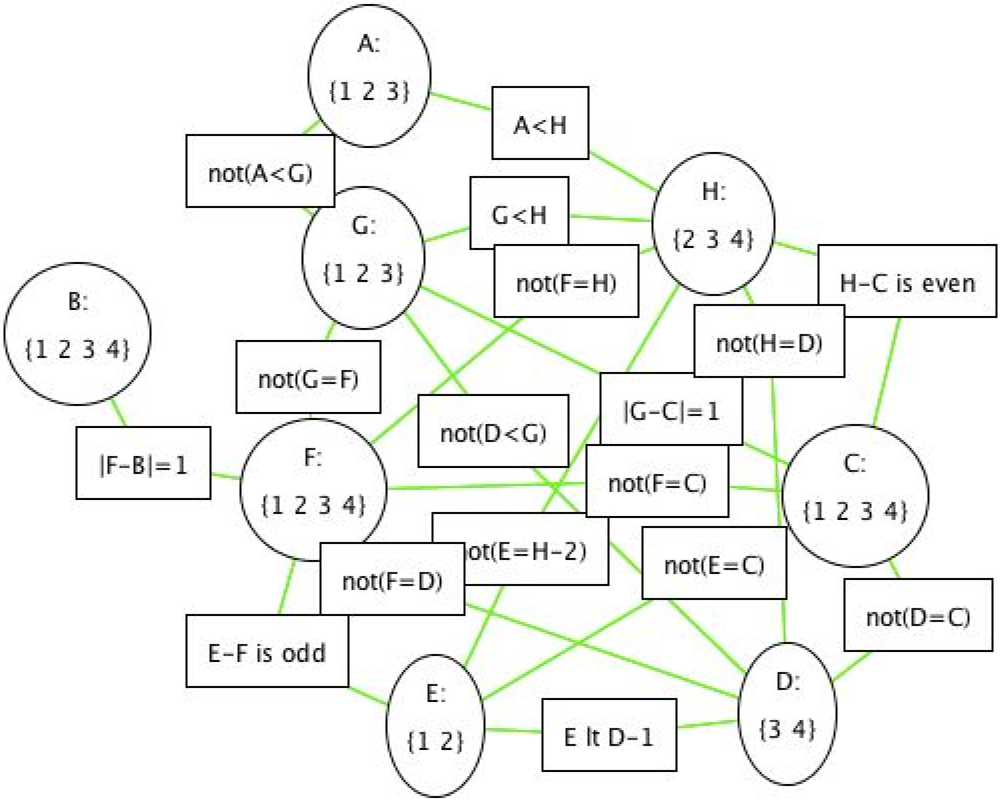
Step 4

Select Arc (G, |G-C|=1)

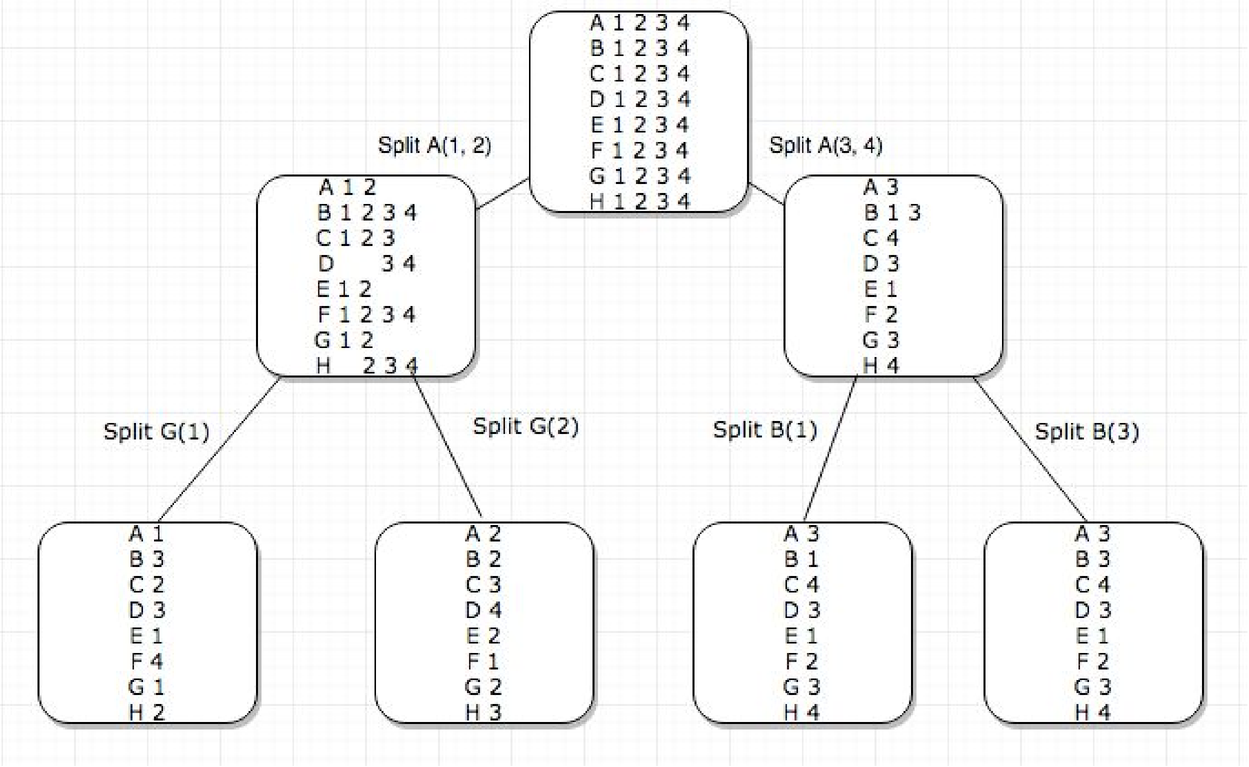
Consistent

No deletion

Mitchell Chan 10753135
Peter Chung 38777124



b.

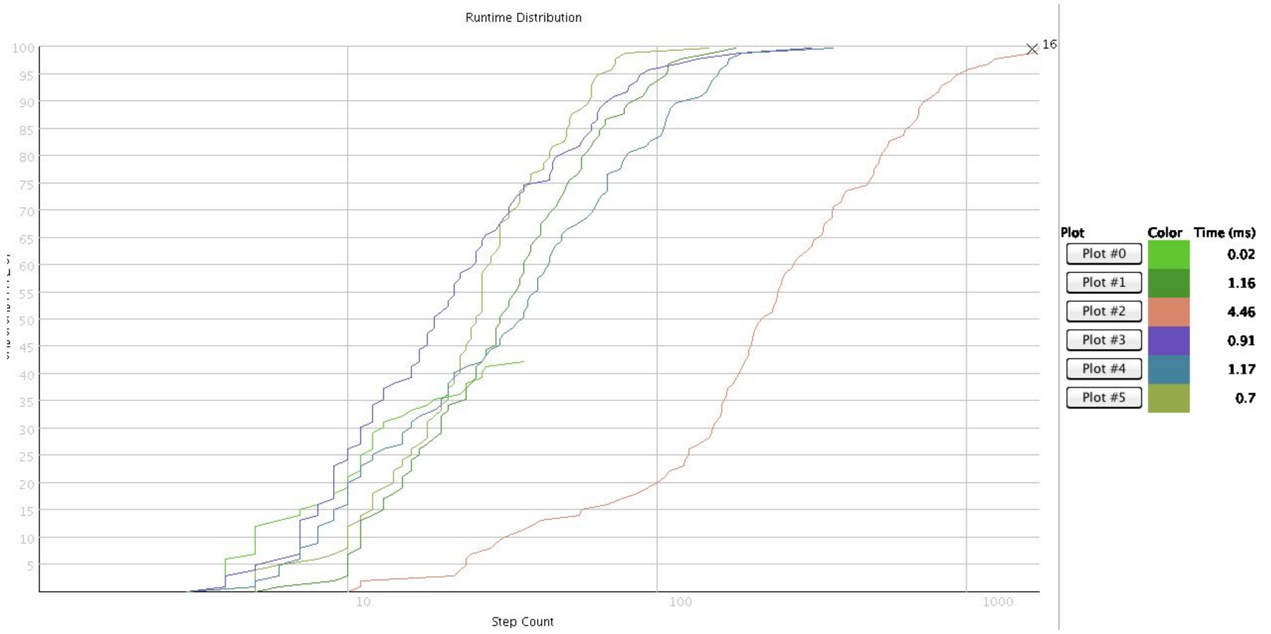


c. If Domain splitting is applied, we don't need to run search again when some domains have more than one value for a solution (since we will be just taking a union of all these as a solution). The time complexity for domain splitting is O(n^2d^3) wherelse the time complexity for DFS with pruning is O(d^n) hence much faster if we use domain splitting.
However domain splitting requires more memory usage because now we need to keep all the CSPs where DFS with pruning does not.

4.

a.	***A***	***B***	***C***	***D***	***E***	***F***	***G***	***H***	***Conflicts***
Initialized	3	3	3	2	2	1	2	4	4
Step 1	3	3	3	2	2	4	2	4	5
Step 2	3	3	3	2	2	4	2	3	3
Step 3	3	3	3	2	2	1	2	3	3
Step 4	3	2	3	2	2	1	2	3	2
Step 5	2	2	3	2	2	1	2	3	1

b.



Plot	Color	Time (ms)
Plot #0	🟩	0.02
Plot #1	🟩	1.16
Plot #2	🟧	4.46
Plot #3	🟪	0.91
Plot #4	🟦	1.17
Plot #5	🟨	0.7

i. **Plot 0**

Largest mean # of steps (1145) and only a 43% success rate. Was the only instance where there is no case where it would be the case where reached 100% successful runs If run 8 or less steps, Plot 0(Light Green) better option than Plot 3(purple). At crossover point, Plot 3 becomes better algorithm to use

ii. **Plot 1**

Mean # of steps was 40. Runs would be 100% successful in 180 steps or less. Becomes better to use Plot 1 than Plot 3(Purple) when run is longer than 150 steps.

iii. **Plot 2**

Took the longest out of all of the runs that met success. Mean # of steps 333.5. Runs would be 100% successful when reach 1700 steps. Longest mean run time of solved runs(slowest)

iv. **Plot 3**

50% best node/50% Random Red Node. Mean # of steps 33. Runs would be 100% successful in 315 steps or less. Has best median performance

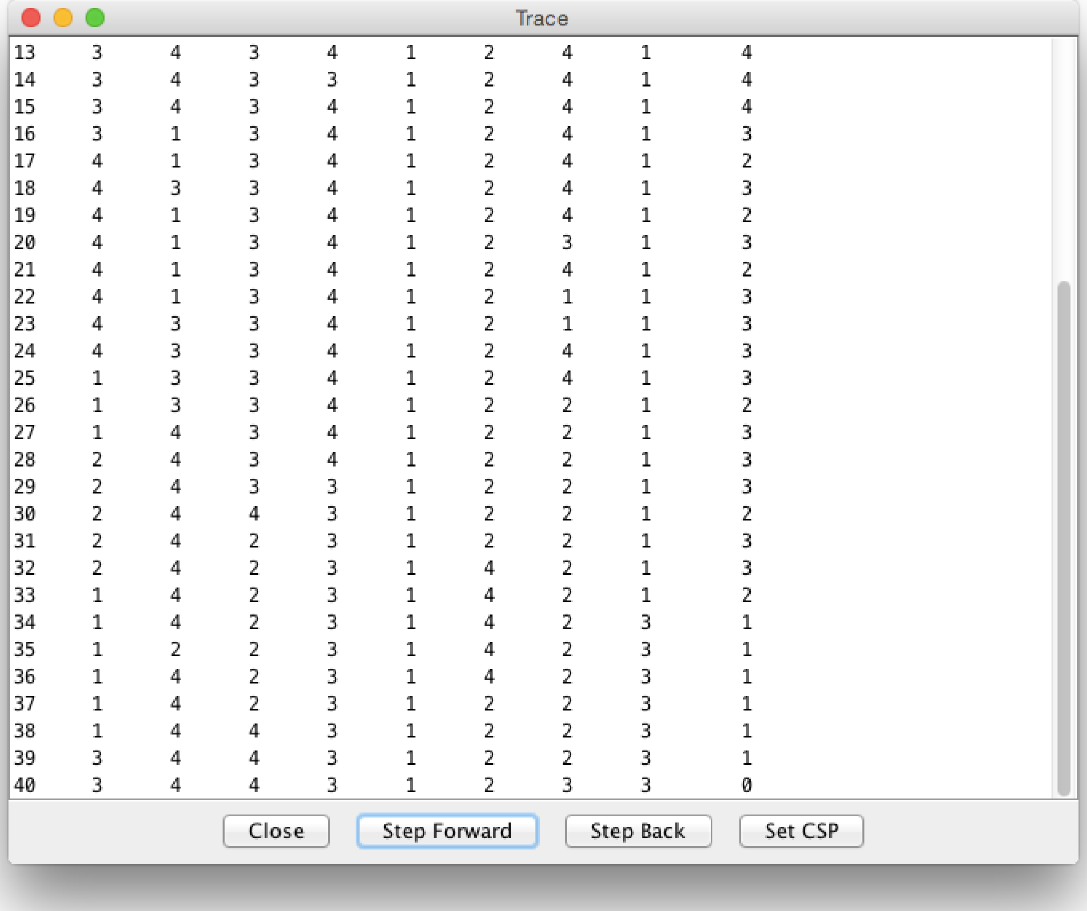
Plot 4

75% best node/25% Random Red Node.Mean # of steps 51.83. Runs would be 100% successful in 370 steps or less.

Plot 5

25% best node/75% Random Red Node.Mean # of steps 30.19. Runs would be 100% successful in 150 steps or less. Best of the probabilistic mix of i and ii. Becomes better option than Plot 3(purple) when a run is longer than 55 steps. Best overall performing - makes sense since most likely to pick value with minimal conflicts.

c.



```
13    3    4    3    4    1    2    4    1     4
14    3    4    3    3    1    2    4    1     4
15    3    4    3    4    1    2    4    1     4
16    3    1    3    4    1    2    4    1     3
17    4    1    3    4    1    2    4    1     2
18    4    3    3    4    1    2    4    1     3
19    4    1    3    4    1    2    4    1     2
20    4    1    3    4    1    2    3    1     3
21    4    1    3    4    1    2    4    1     2
22    4    1    3    4    1    2    1    1     3
23    4    3    3    4    1    2    1    1     3
24    4    3    3    4    1    2    4    1     3
25    1    3    3    4    1    2    4    1     3
26    1    3    3    4    1    2    2    1     2
27    1    4    3    4    1    2    2    1     3
28    2    4    3    4    1    2    2    1     3
29    2    4    3    3    1    2    2    1     3
30    2    4    4    3    1    2    2    1     2
31    2    4    2    3    1    2    2    1     3
32    2    4    2    3    1    4    2    1     3
33    1    4    2    3    1    4    2    1     2
34    1    4    2    3    1    4    2    3     1
35    1    2    2    3    1    4    2    3     1
36    1    4    2    3    1    4    2    3     1
37    1    4    2    3    1    2    2    3     1
38    1    4    4    3    1    2    2    3     1
39    3    4    4    3    1    2    2    3     1
40    3    4    4    3    1    2    3    3     0
```

Chose method ii. For the longest time, either had same number of conflicts or even increased. When the number of conflicts was constant, believe the algorithm was stagnating. At this point, was worried that a solution wasn't going to be found because got caught in one region. Solution finally reached after 40 steps. By choosing any variable that is involved in an unsatisfied constraint, have equal probability of choosing a variable that only has a single, unsatisfied constraint, many unsatisfied constraints, or any number in between. Best case scenario, we choose the variable that has the least number of unsatisfied constraints every time.

d. In case of annealing, you start from very random and over time, it becomes less random and more likely that you are to follow a scoring function. If we were to constantly choose the next value at random, this would completely defeat the purpose of annealing. By selecting the next best value that has the fewest unsatisfied constraints, this follows the annealing schedule.
In stochastic beam search, the neighbours with lower heuristics are chosen more frequently. This means attaching a weight to constraints. The heuristic of any variable is the sum of its weights of unsatisfied constraints. Lower the heuristic, higher the probability of being chosen - this maintains "randomness" but at the same time allows some diversity

e. In part (b), the method that provided the best performance was a probabilistic mix of (i) and (ii) with more weight on random red nodes. If random resets weren't allowed and you found yourself in a situation where you were stuck at a local maxima, plateau, or shoulder, there is potential to be stuck there indefinitely. We eliminate local maxima by injecting randomness; Stochastic Local Search is taking local search and adding randomness which enables you to get out of these local maxima. Without random resets, the only other options are **i)** Random Steps or **ii)**Hill Climbing. Both of these can be extremely time-consuming, especially if stuck at a local maxima.

Mitchell Chan 10753135
Peter Chung 38777124

APPENDIX

Code for Q2

```
public static List<List<Integer>> satisfy(){
    List<List<Integer>> satisfyingResults = new ArrayList<>();

    boolean isSatisfied = false;
    for (int A = 1; A <= 4; A++ ){
      for(int B = 1; B <= 4; B++){
        for(int C = 1; C <= 4; C++){
          for(int D = 1; D <= 4; D++){
            for(int E = 1; E <= 4; E++){
              for(int F = 1; F <= 4; F++){
                for(int G = 1; G <= 4; G++){
                  for(int H = 1; H <= 4; H++){
                    if(A >= G){
                      if(A < H){
                        if(Math.abs(F-B) == 1){
                          if(G < H){
                            if(Math.abs(G-C) == 1){
                              if(Math.abs(H-C) % 2 == 0){
                                if(H != D){
                                  if(D >= G){
                                    if(D != C){
                                      if(E != C){
                                        if(E < D -1){
                                          if(E != H - 2){
                                            if(G != F){
                                              if(H != F){
                                                if(C != F){
                                                  if(D != F){
                                                    if(Math.abs(E-F) % 2 == 1){

                                                      isSatisfied = true;
                                                      List<Integer> res = new
ArrayList<>();

                                                      res.add(A);
                                                      res.add(B);
                                                      res.add(C);
                                                      res.add(D);
                                                      res.add(E);
                                                    res.add(F);
                                                    res.add(G);
                                                    res.add(H);
                                                  satisfyingResults.add(res);
                                                  }
                                                }
                                              }
                                            }
                                          }
                                        }
                                      }
                                    }
                                  }
                                }
                              }
                            }
                          }
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }

  return satisfyingResults;
}
```