Assignment 2: Constraint Satisfaction

Niranjan Balsubramanian, Tianyi Zhao, and Tao Sun

CSE 352 Spring 2021

1 Due Date and Collaboration

- The assignment is due on **Mar 16 at 11:59 pm**. You have a total of four extra days for late submission across all the assignments. You can use all four days for a single assignment, one day for each assignment – whatever works best for you. Submissions between fourth and fifth day will be docked 20%. Submissions between fifth and sixth day will be docked 40%. Submissions after sixth day won't be evaluated.
- You can collaborate to discuss ideas to understand the concepts and math.
- You should NOT collaborate at the code or pseudo-code level. This includes all implementation activities: design, coding, and debugging.
- You should NOT not use any code that you did not write to complete the assignment.
- The homework will be cross-checked. **Do not cheat at all! It's worth doing the homework partially instead of cheating and copying your code and get 0 for the whole homework.**

2 Goals

The main goal of this assignment is to 1) understand the impact of the enhancements to depth-first search with backtracking 2) learn the details of min-conflicts algorithm, which we did not see in class.

In this programming assignment you will build a program that can solve graph coloring problems. Just as a reminder, the graph coloring problem is one where you are given a set of regions (variables), a set of colors (domains of these variables), and the adjacency information for these regions (constraints that connect the variables as a constraint graph).

3 Implementation

Your program will solve the graph coloring problem using Depth first with backtracking and Min-Conflicts Local Search.

3.1 Notes on Min-Conflicts Local Search

Min-conflicts Local Search is a very simple and fast heuristic method to solve constraint satisfaction problems (CSP). You can find many online resources:

1. `https://en.wikipedia.org/wiki/Min-conflicts_algorithm`
2. `http://pages.cs.wisc.edu/~bgibson/cs540/handouts/csp.pdf`

One drawback of Min-Conflicts algorithm is that it might get stuck in a local optimum or might not find a solution even if one exists. You need to figure out some solutions when your algorithm encounters these issues.

You need to resart from a different random state. In this case, you should log the steps taken to solve from the current start state but also the total steps from the beginning including all the abandoned search steps.

3.2 Input File Format

There are N variables, numbered 0 to N-1, and M constraints connecting them. There are also K possible colors for each of variables, so the domain for all variables is the same. In the input, we first give them N, M and K in the first line, separated by a **single** whitespace. Then on each following line, we give constraints one by one by giving them U and V, separated by a **single** whitespace, as two variables whose colors should not be the same. There should be M lines after the first line containing the constraints.

Input File Content:

$N\ M\ K$

$v_0\ \ u_0$ \\ variables v_0 and u_0 should not have the same color.

$v_1\ \ u_1$

...

$v_{M-1}\ \ u_{M-1}$

3.3 Out File Format

You will output the colors for each variable one per line. The color for variable 0 should be in line 1 and variable 1 in line 2 and so on. If there is no answer, simply write "No answer" in output file.

Output file content

c_0
c_1
...
c_{n-1}

3.4 Sample Problems

The sample input and output is shown below:

Input	Output
3 2 2 0 2 0 2	0 0 1

We will provide you with two sets of sample problems (easy and hard). Below you can find if each sample is solvable or not. **The hard tests are given to optimize your implementation and polish your code, the final tests used for evaluating your codes are going to be easier than the hard ones here.**

backtrack_easy	backtrack_hard	minconflict_easy	minconflict_hard
solvable	solvable	solvable	solvable*

*: It's hard to find the solution for this case with the minconflict algorithm in given time, so if you didn't find it, there is no problem. "No answer" is also acceptable for this test.

3.5 Performance Comparison

You are provided a script to generate more CSP problems. To generate solvable problem

python CSPGenerator.py <N> <M> <K> <output_file_path>

where <N>, <M>, <K> are the same as above. output_file_path is output file path for test problem. This script can not guarantee to generate test problem

for given parameters. The generated csp are exclusive, so the script would fail if M is too large.

To generate random problem (which may not be solvable and we will not evaluate on)

python CSPGenerator.py <N> <M> <K> <output_file_path> 0

You are required to generate more test problems by yourself and compare the performance of three algorithms on them. Specifically,

1. Define 5 sets of (N, M, K) parameters with increasing problem sizes. For example, you can fix $K = 4$, $M = N^2/4$, and choose $N = [20, 50, 100, 200, 400]$. You are free to choose other settings for the parameters, but make sure your choices show some trends that you want to discuss.

2. For each parameter set, you need to compute the number of states explored by DFSB, DFSB++, the number of steps in MinConflicts algorithm and the actual time for the three algorithms (3 numbers of states/steps plus 3 actual times).

3. To report your result, you are required to repeat each parameter set 20 times (which means 100 test problems in total), and report the mean and stddev for the above 6 variables (the number of states ... etc) on 5 parameter sets. An exemplary table is shown below:

parameter set	number of states	actual time
N=20, K=4, M=100	mean±std	
N=50, K=4, M=625		
N=100, K=4, M=2500		
N=200, K=4, M=10000		
N=400, K=4, M=40000		

Table 1: Exemplary table for DFSB algorithm

3.6 Requirements

1. You must implement the algorithms from scratch in **Python 3.7+**.
2. You CANNOT use external libraries that provide search related capabilities directly.
3. You CAN use native python libraries that provide support for data structures (e.g., priority queues) but your program should run on its own.
4. You CANNOT use existing implementations of any kind for development or reference.
5. You CAN use pseudo-code or descriptions of the algorithms for reference but not actual code.

6. You CAN discuss with your friends about which algorithms to use and to understand the algorithms but not share or discuss code.

7. All code will be tested by running through plagiarism detection software.

8. You should include cite any web resources or friends you used/discussed with for pseudo-code or the algorithm itself. Failure to do so will result in an F.

9. Your algorithm should be documented at a method level briefly so they can be read and understood by the TAs.

4 What should you turn in?

- README - Code Implementation details. Please include your Python version or any other information that is necessary for TAs to run your code.

- Code - You should turn in your source code. Your source code can be structured however you like but it needs to have two application files: **dfsb.py** and **minconflicts.py**. Your solvers should be able to take the input file specified in the format above and output a coloring consistent with the constraints.

  1. dfsb.py – This should run in two modes. a) Plain DFS-B and b) DFS-B with variable, value ordering + AC3 for constraint propagation. A sample execution of dfsb.py should be as below:

  python dfsb.py <INPUT_FILE> <OUTPUT_FILE> <MODE_FLAG>.

  <MODE_FLAG> can be either 0 (plain DFS-B) or 1 (improved DFS-B).

  2. minconflicts.py – This should run the MinConflicts local search algorithm. You may have to adjust the MinConflicts algorithm to avoid plateaus or getting stuck in local depressions. A sample execution of minconflicts.py should be as below:

  python minconflicts.py <INPUT_FILE> <OUTPUT_FILE>.

- Four output files - You should turn in the output, named input_file-output.txt, of your program on the four input files shipped as part of the assignment.

  The backtrack_∗ files are to be used as inputs to dfsb.py in the improved mode.
  The minconflict_∗ files are to be used as inputs to minconflicts.py.

  e.g., for *backtrack_easy* as input you should output in a *backtrack_easy-output.txt* file.

- Report (PDF) – This should be a 3-4 page report, named SBUID-LastName-FirstName-A2.pdf, which includes three sections:

    1. Briefly explain how each method works including pseudo code for DFSB, DFSB++, and MinConflicts
    2. Tables describing the performance of the algorithms (DFSB, DFSB++, and MinConflicts) on your generated problems. Please refer to section 3.5.
    3. Explain the observed performance differences.

Please put all these files in the same folder. Your submission should be one zip file named SBUID-LastName-FirstName-A2.zip, containing all source code and report under a single folder named SBUID-LastName-FirstName-A2, with no other folders in the zip file. Please note that the file name for the source codes must be exactly the same as instructed.

5 Grading

Here is a coarse grained grading rubric.

- DFSB implementation (40 points)

    1. As part of evaluation of your implementation, your code will be run against several test cases. The time limit for solving these tests is 60 seconds. The limit is not strict, which means that we will select test cases our own code can solve in no more than 30 seconds.
    2. Detail comments in source codes (10 points)
        - Comments that serve as code documentation for readers.
        - Explain the purpose of lines/functions in the code.
        - No need to have comments on every single lines but main steps that explain what you are doing.

- MinConflicts implementation (40 points) – Same as above.

- Report (20 points).