CSE 352 A2 Report

Jiaxin Dong

111658846

In both of these two algorithm, DFBS or MinConflict, the basic code structure for describing problem are the same. We use a class which called Graph to describe coloring map into a undirected graph by using a number to describe node counts and a adj. list for edges. The only function in this class is add constrains, which add an edge to current graph object's adj. list

.

Part 1 DFSB & DFSB++

DFSB or DFSB++ using a plain backtrack algorithm which is:

Check if answer is correct: if correct output the answer

Select a candidate from rest of the nodes

Create a loop in every possible action on candidate and recursively do this function on it.

With DFSB++, we using Arc3 to pre-eliminate some of the impossible solution (prune). It will go forward to neighbor node to see if all color in current node can satisfy binary constraint, in this case, no same color. We implement this in our plain DFBS: each time it choose a candidate, we forward and using arc consistent to prune out some of the colors, so we don't have that much color choices to run.

Part 2 MinConflict

This algorithm is not useful as the DFSB. The reason why is this algorithm depend too much on randomization and local optimum. The algorithm working as this:

We first randomly generate a color for every nodes to form a “answer vector”.

While the vector is not true answer:

    Pick a node that has violation with it’s neighbor

    Find colors that has least conflict

        If we only have one color then we put this value in the node we choose

        If we get many colors when we randomly choose on to apply to the node

This algorithm has a problem on finding colors and assign to the node we select. The way it choose color is not based on what this node actually need but it looks over whole problem and generate an answer than can minimize global conflict. This will not get to an answer we want but a local optimum and it will keep looping in that local optimum and will never get the answer.

Part 3 Table

The reasonable time I set is 65 seconds and I can’t get both hard case in reasonable time

M = N^2/6

Alone with increasing of N and M, the runnable function become more rare. So the std and mean are relatively huge/small due to less samples.

DFSB:

Param Set	No. states explored	Time(s)
N=20, M=66, K=4	1008 +- 286	0.01695 +- 0.0034
N=50, M=416, K=4	148173 +- 1248	6.80936 +- 3.7547
N=100, M=1666, K=4	289413 +- 829	24.87745 +- 10.8756
N=200, M=6666, K=4	245593 +- 8975	37.94430 +- 6.4368
N=400, M=26666, K=4	202937 +- 56548	57.91330 +- 2.7564

DFSB++:

Param Set	No. states explored	Time
N=20, M=80, K=4	638 +- 49	0.27526 +- 0.0539
N=50, M=416, K=4	No answer	Out of 65s
N=100, M=1666, K=4	No answer	Out of 65s
N=200, M=6666, K=4	No answer	Out of 65s
N=400, M=26666, K=4	No answer	Out of 65s

MinConflict:

Param Set	No. states explored	Time
N=20, M=80, K=4	22 +- 3	0.00898 +- 0.0014
N=50, M=416, K=4	No answer	Out of 65s
N=100, M=1666, K=4	No answer	Out of 65s
N=200, M=6666, K=4	No answer	Out of 65s
N=400, M=26666, K=4	No answer	Out of 65s

Above Minconflict is using the same test cases with DFSB, test cases are huge for this algorithm to come out answers.

MinConflict:

Param Set	No. states explored	Time
N=20, M=80, K=4	33 +- 6	0.01695 +- 0.0068
N=40, M=100, K=4	108 +- 26	0.11768 +- 0.0394
N=60, M=130, K=4	119 +- 13	0.15259 +- 0.0248
N=80, M=180, K=4	141 +- 25	0.33607 +- 0.0846
N=100, M=200, K=4	72 +- 20	0.25561 +- 0.1064

Part 4 Observation

DFSB++ will not perform faster than plain DFSB than we thought, because the time we use on checking arc consistency is way longer than running the pruned solutions. I'm running a test case that have N=20 M=80 K=4 with plain DFSB can be in 0.025 sec, but in DFSB++ it can run around 1 sec. So by using DFSB++ our algorithm is perform nearly 30-40 times slower. If plain DFSB can't get answer in 1-2s of a case, then DFSB++ can't get answer in 65s.

With minConflict I sometime can't get answer from it, so I changed the basic structure of the algorithm so when it can't get answer from the random vector we generate in N*90 times, it will generate another random vector to keep searching for N*90 times. It will generate total of N vectors if it can't find any solution. This algorithm can easily run into a local optimum and stuck in it.