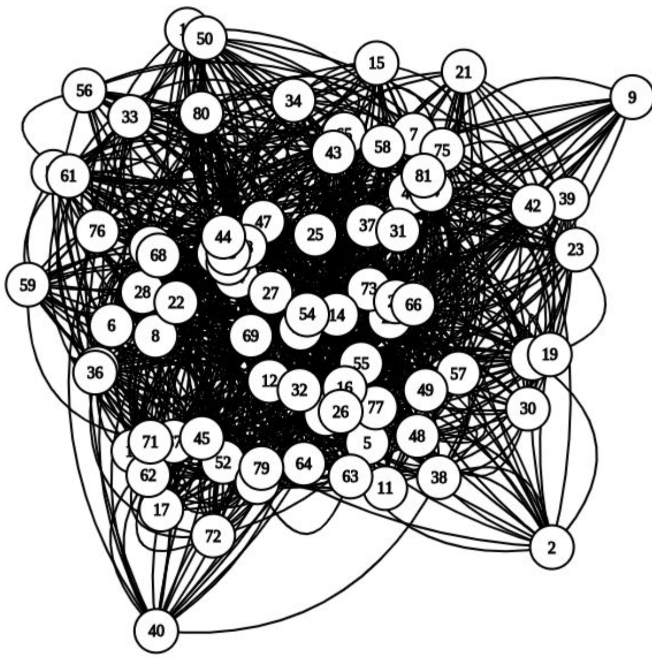
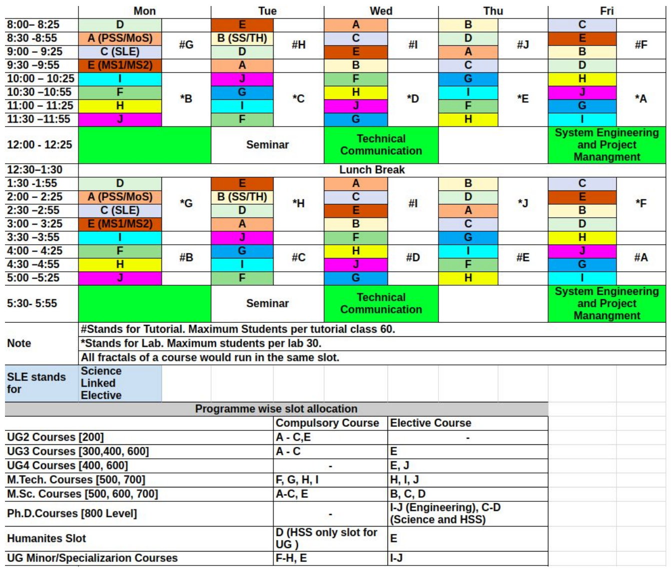
CS323 ARTIFICIAL INTELLIGENCE

Solving Graph Coloring Problem with AI for Practical Applications

Kunal Tawatia





	Mon		Tue		Wed		Thu		Fri	
8:00– 8:25	D		E		A		B		C	
8:30 -8:55	A (PSS/MoS)	#G	B (SS/TH)	#H	C	#I	D	#J	E	#F
9:00 – 9:25	C (SLE)		D		E		A		B	
9:30 –9:55	E (MS1/MS2)		A		B		C		D	
10:00 – 10:25	I	*B	J	*C	F	*D	G	*E	H	*A
10:30 –10:55	F		G		H		I		J	
11:00 – 11:25	H		I		J		F		G	
11:30 –11:55	J		F		G		H		I	
12:00 - 12:25			Seminar		Technical Communication				System Engineering and Project Manangment	
12:30–1:30	Lunch Break									
1:30 -1:55	D	*G	E	*H	A	#I	B	*J	C	*F
2:00 – 2:25	A (PSS/MoS)		B (SS/TH)		C		D		E	
2:30 –2:55	C (SLE)		D		E		A		B	
3:00 – 3:25	E (MS1/MS2)		A		B		C		D	
3:30 –3:55	I		J		F		G		H	
4:00 – 4:25	F	#B	G	#C	H	#D	I	#E	J	#A
4:30 –4:55	H		I		J		F		G	
5:00 –5:25	J		F		G		H		I	
5:30- 5:55			Seminar		Technical Communication				System Engineering and Project Manangment	
Note	#Stands for Tutorial. Maximum Students per tutorial class 60.									
	*Stands for Lab. Maximum students per lab 30.									
	All fractals of a course would run in the same slot.									
SLE stands for	Science Linked Elective									

Programme wise slot allocation		
	Compulsory Course	Elective Course
UG2 Courses [200]	A - C,E	-
UG3 Courses [300,400, 600]	A - C	E
UG4 Courses [400, 600]	-	E, J
M.Tech. Courses [500, 700]	F, G, H, I	H, I, J
M.Sc. Courses [500, 600, 700]	A-C, E	B, C, D
Ph.D.Courses [800 Level]	-	I-J (Engineering), C-D (Science and HSS)
Humanites Slot	D (HSS only slot for UG)	E
UG Minor/Specializarion Courses	F-H, E	I-J

Introduction

In our academic registration, during the start of every session, we have to register for some compulsory courses and choose certain electives. The choices are completely bounded by the slots assigned to each course and there are cases when we miss out on our favorite courses because the slot was clashing with some other course on our list. These slots further decide many things, among which is the schedule of exams and daily academic timetable. The lesser the slots the quicker would the schedules be and will save us valuable time. The existing process of slot distribution is dynamic, tedious, and no individual request of a student can be entertained.

This problem is an optimization instance of class CSP where we have to assign a minimum number of slots to the given courses such that there is no student or faculty with more than one course in a single slot. The problem is fairly novel in the sense that we have never discussed to automate this problem on our campus. The problem is NP-Hard and we would have to check for all possible combinations of assignments of slots to courses to verify for an optimal answer, and still, this is done manually in our case, presently for 238 courses and 2000+ students and faculties.

In our project, we have made a system, which takes the list of courses, a list of conflicting choices, and a partial assignment of slots (possibly empty) as an input, and gives an assignment of slots as output. We solve this problem as an instance of the Graph - Coloring Problem (GCP) and we have tested our approach on other applications of graph coloring, like solving the Sudoku puzzle, as well.

Problem Definition

Definition and Structure

We are given a graph $G = (V, E)$ with a set of n vertices representing variables, $V = \{V_1, V_2, \ldots, V_n\}$, and a set of m undirected unweighted edges representing the constraints, $E \subset V \times V$, A *proper k-coloring* of G is a complete assignment of colors(values) to the vertices(variables) such that there are less than or equal to k unique colors, and no two vertices with an edge in between have the same color. We aim to solve for the minimum such k.

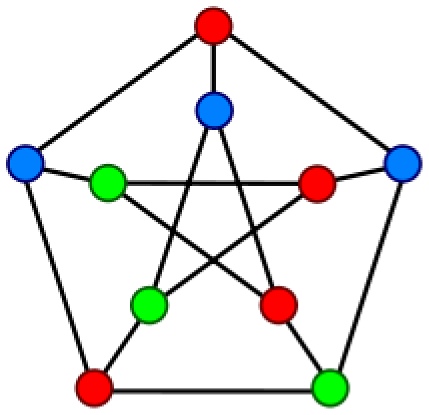


Fig. A proper 3-coloring for a Petersen Graph.
Source: Wikipedia

In terms of our slot-assignment problem, we have V to be the set of courses, i.e V = { CS323, CS311, HSL4010, … }, two courses (vertices) u and v have an edge i.e (u, v) ∈ E if there exists a student or faculty such that he/she has opted for both the courses u and v. A *proper k-coloring* of this graph *G* is a complete assignment of slots to the courses such that there are less than or equal to k unique slots, and no two courses with an edge in between have the same color.

Similarly, in terms of the sudoku problem, the vertices are the empty grids to fill and there is an edge between two grids if they belong to the same row or the same column or the same box. Here, instead of minimizing the k, we check whether there exists a k-coloring, and k here is also defined to be the length of a side in the square matrix i.e 9 in a standard puzzle.

Search

* *Backtracking:*

We form a search tree where each state is one valid assignment i.e all the assigned variables are satisfying constraints (there exists no such edge where its two vertices (u, v) have the same assigned color). The transition step from any state would be to assign one unassigned variable with a non-conflicting value less than or equal to ‘k’ (so the derived state is also a valid assignment). We do not care about the path (i.e how we assigned the variables) in this search and thus cost function is not necessary.

The start state would be an assignment with all the variables unassigned i.e {0, 0, …, 0}. This start state is valid from the definition. And, in our goal test, we check whether all variables are validly assigned i.e have a non-conflicting value which is less than or equal to ‘k’. The state which passes our goal test is our goal state.

If we can find any goal state, we say that a proper k-coloring for the graph exists.

**** Simulated Annealing:***

We define each state to be one complete assignment i.e all variables are assigned with some value less than or equal to k. We define our loss function H(x), which takes a state as an input and gives the number of conflicting edges as output i.e count of such edges where two vertices u, v have the same assigned value.

Our objective is to minimize the loss function. In one transition we can assign one uniformly randomly picked node with a uniformly randomly picked value. The transition depends on `delta` and `temperature`, of which, we have control over the `temperature`

If we can find a state g with H(g) = 0, we say that a proper k-coloring for the graph exists and it's the assignment followed by g. H(g) = 0 signifies that no edges are conflicting and thus our constraints satisfy the whole graph.

Background survey

Although this problem is NP-Hard, there exist certain polynomial-time algorithms that only work on a specific family of graphs. For example, To check for a 2-coloring, we can check whether the graph is bipartite or not, this can be done in linear time. Similarly, optimal coloring for some other families, like planar graphs, can be computed with pseudo polynomial-time algorithms using dynamic programming as well.

There are naive brute force approaches using DFS (DFS for CSPs) for exact optimal answers. Although they work in exponential time, we can manage that for smaller graphs.

Backtracking approaches with various strategies are widely used for these problems and we see in our project that we also use some of those approaches after optimizing them with data structures to reduce run time.

We also have evidence for stochastic search used in CSPs, simulated annealing approach defined in the previous section is one such search method. We have used that method and experimented on the parameters to fit our case for better results.

Discussion

** Describe the novelty in terms of your problem and/or solution*

The existing slot-scheduling method on our campus is manual. Students face a loss when they don't have the right choices to choose from, their requests are not entertained (unless it's a mass request), some students also have to switch their courses later in the semester if a course is no longer available in their slot. Solving this problem with the solutions discussed in the course is fairly novel and interesting.

The concept of the solutions we propose in this project (backtracking, simulated annealing) is *not* novel. Rather, we have optimized backtracking with the use of a unique combination of data structures (discussed in the next section) to give us a better runtime and so that we can explore a larger search space. In simulated annealing, we have tweaked the parameters to perform well in our specific cases of graphs (course selection and sudoku graphs, discussed later).

** Why do you consider your solution an engineering-based solution*

By definition, "Engineering is the use of scientific principles to design and build machines, structures, and other items." (Wikipedia), I think that we have used scientific principles (concepts used in AI) and designed a system which best suits our needs and helps us resolve a tedious task. So, Yes, that's the reason why I consider my solution to be "engineering-based".

** Discuss the scale and feasibility of the problem and the proposed solution*

The problem we aimed for, is a part of the "scheduling" class of problems that are solved by graph coloring. The *scale* of this problem can be very huge, like in network analysis we try to solve server scheduling, and a problem can also be seen in daily life, like TA scheduling for different classes.

The proposed solution is *scalable* because it is generic. We only take the structure of the graph as input, and that structure can be of any *scale* (minding the limits of time and memory in our machines).

The problem is NP-Hard and thus any solution we propose is limited in *feasibility*. However, the generic solution we have proposed gives a solution in 100 trials of simulated annealing of 10000 iterations and/or a 0.5s run of our backtracking search. These limits are dynamic and well optimized, so we can argue that they are feasible.

** Validation and Demonstration*

The types of graphs we validate our solution on, are the ones specific to the problem we have aimed at. For example, for the course-scheduling algorithm, we have processed our solution on the graph of 250 courses which are divided into several levels (100, 200, ..., 700) and then we have 2000 students which are divided into levels and they can add a conflict between 5 courses of some certain levels. In the case of the sudoku-solving algorithm, we have processed our solution on the graph of 81 nodes which have 20 edges each. Hence, our solution, at least, proves to be valid for the problem we solve.

The demonstration of these is shown in other sections.

Algorithm

Backtracking

This is a deterministic algorithm that is guaranteed to find an optimal solution, limitations are that it might not find a solution in the allowed time. So, we want to optimize this approach to maximize the expansion of our search tree. Instead of naive backtracking, we have used the method of forward-checking (arc-consistency heuristic wasn't used because of the heavy computation it required), and with that, "MRV ordering" was followed.

Optimizations:

The following are used in the transition step of our search:

* `update`, change a value : we have to assign a value to a variable, v, and update the domain of its neighbours. We can assign in O(1) and update the domain in O(deg(v)), but to maintain a priority queue structure for the next step, we have to assign in O(log(n)) and update neighbour's domain in O(deg(v) * log(n)).

* `most_constrained_variable`, returns the next variable to go to: this can be done with a linear search over all unassigned variables which take O(n), but since we maintain a priority queue we can do this step in O(1).

* `remove`, unassign a node to go back: we have to assign 0 to a variable, v, and update the domain of its neighbours. We can assign in O(1), but to manage a priority queue we do it in O(log(n)). To refresh the domain of neighhbours we earlier had to check for all next neighbours to reconstruct it which took O(deg(v) * max(deg(v_neighbour))), but, we here have used a stack structure which allows us to fetch the value of domain which were there earlier, this allows us again to update the domain in O(deg(v)), but again here to manage a priority queue we do it in O(deg(v) * log(n)).

Other than these, We also have used bitmasking for managing domain rather than a `set` or `list`, which allows us to operate on it faster.

This method we termed as "optimization" is actually just an alternative and it can be seen that existing algorithm works better than this in a dense graph, more details on this and complexity (for all implementations) and examples are mentioned in the coming sections.

Simulated Annealing

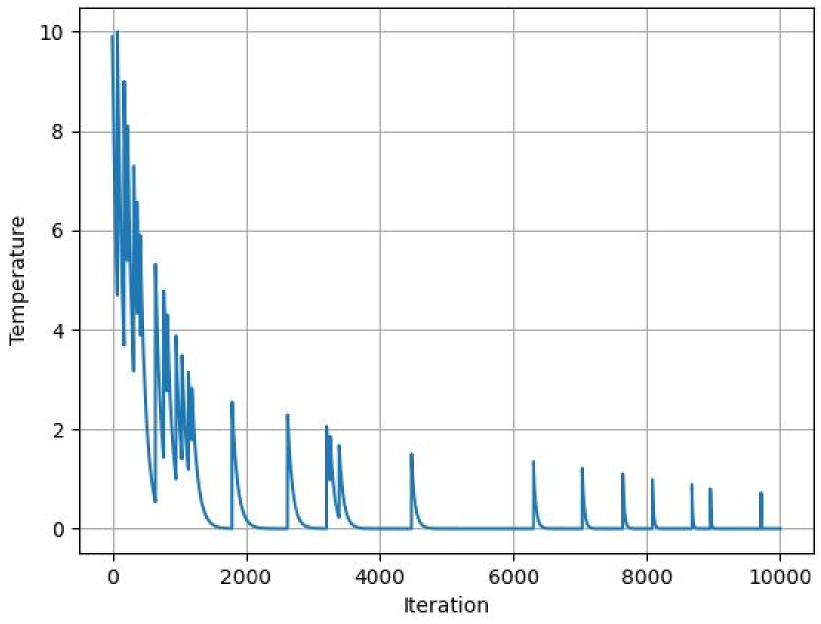
This is a stochastic algorithm that is not necessarily guaranteed to find an optimal solution, but at times it performs better than backtracking in terms of time. We can have greater contnrol over this with the help of parameterslimitations are that it might not find a solution in the allowed time. This method performs better in bigger graphs. This search method is already defined in previous sections.

Optimization:

* `update`, change a value: We have to change the value in a node, v, and recalculate the loss function for the updated assignment, this takes O(n + m) time all over again. To

optimise this, we calculate the previous conflicts of `v` and new conflicts of `v` in O(deg(v)) time and we are able to then calculate new loss function, $H(x_{new}) = H(x_{prev})$ + new_conflicts - prev_conflicts.

We have set total trials limit (number of random walks) to be 100 and iterations in every walk to be 10000, we keep initial temperature 10 and another variable max_temperature = 10, if we face 50 rejected iterations in a sequence then we are stuck in a local minima and we change the temperature back to max_temperature and max_temperature after this reduces to 90%. And also in every iteration temperature reduces by a factor of (1- 100 / (remaining iterations)).



This graph shows a random instance of our temperature function. This is a simple heuristic i have used after numerous experiments with temperature transition function.

Chromatic Number

The above two algorithms are to search whether there exists a k-coloring or not. But our goal is to find the smallest k, for that, we observe that if our answer is O then (O +

i)-coloring exists for i >= 0, and for i < 0 (O + i)-coloring doesn't exist. Thus to find O we can do a binary search for range (0, n] and find the best k.

To process for any middle value of range, we can use backtracking with some time limit to terminate the search (0.5s in our case) or we can use the simmulated annealing which terminates on its own (but we could tweak with number of trials for better results).

Algorithm Complexity

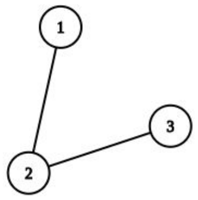
Backtracking

Our depth to goal in worse case would be n, since initially we can have all unassigned variable and at every step we assign a single variable unless all variables are validly assigned. The branching factor at a node in worse case can be k (maximum number of variables to assign). O((n^k) * J) where J is the time complexity of taking n steps, i.e time spent in updating each value and choosing the next node. In naive case J is O(n*n), but we discussed in Algorithm section how we have an alternative of O(log(n) * m) for n steps. Now if m = n*n, then this becomes O(log(n) * n^2) which is actually worse, but this case happens only in very dense graph, which our course-scheduling graph is not. Regarding the space complexity, it takes O(n) for call stack, and in our alternative case the factor is just added with another O(n) for priority queue and O(m) for the domain stacks mentioned, thus in our case the space complexity is O(n + m).

Simulated Annealing

Let the number of trials be T, the number of iterations per trial be I. Every iteration takes O(deg(v)) in our case, thus the total complexity comes out to be O(T* I * max(deg(v))) which can also be written as O(T * I * m), (m was the number of edges in our graph). Auxiliary space complexity here for this algorithm is O(n) since it only keeps into account the value of each variable. Time complexity grows linearly with respect to the number of constraints, and Space complexity grows linearly with respect to the number of variables.

Worked Example



This above is a simple graph with variables {1, 2, 3} and we have to find the chromatic number for this graph, we implement a binary search on the range (0, 3] for k.

We initially pick k = GIF ((0 + 3) / 2) = 1,

Let’s implement a backtracking approach to check for 1-coloring on this graph, The initial node would be 1 we assign a value of 1 and reduce the domain of 2 -> {}, since the domain is empty we refuse this assignment and traverse back. Variable 1 has no other value to try, and we failed the search, we return to main with no solution.

Now the range for binary search would become (1, 3],

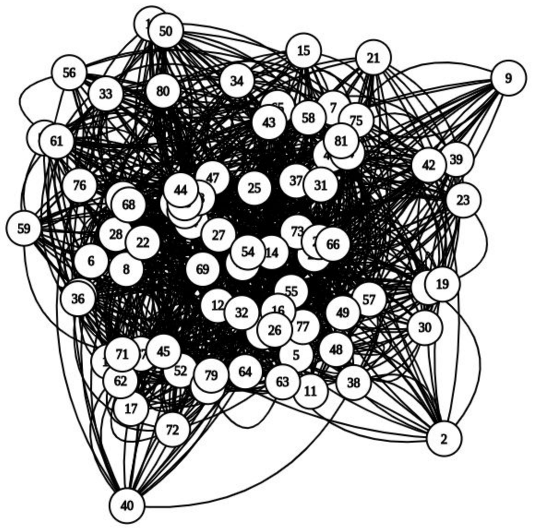
We now pick k = GIF ((0 + 3) / 2) = 2, and with backtracking The initial node would be 1 we assign a value of 1 and reduce the domain of 2 -> {2}. The domain for 2 is the smallest we would assign values to 2 now, we assign 2 to variable 2, and further domain for 3 -> {1}, since this is the only variable left, we would choose this and assign 1 to variable 3. We finally reached the goal node and backtracking was successful.

Now since the binary search range has become (1, 2]. We can say that answer would be 2.

Thus there exists a proper 2-coloring for this graph (which is minimum by our algorithm). And the final assignment would be {1,2,1}.

Experiments

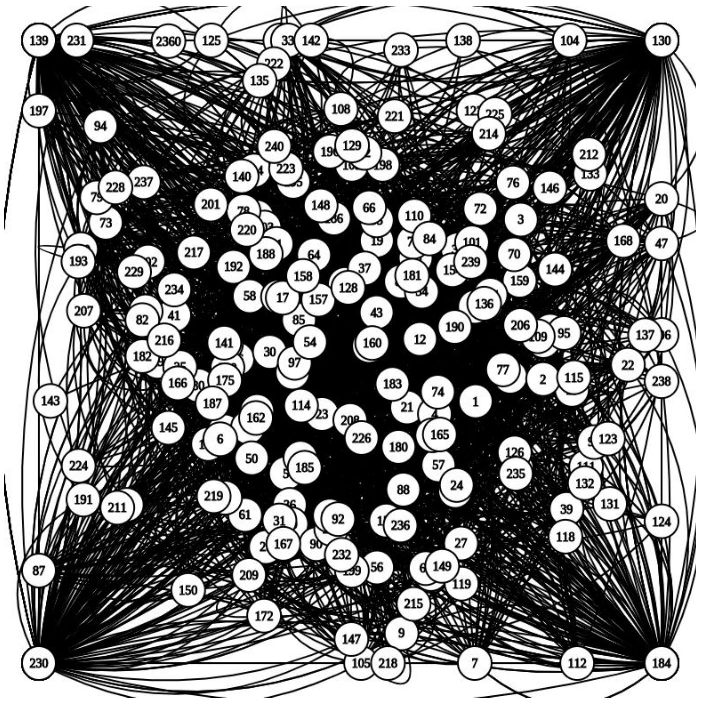
We experimented variants of our algorithms to check for 9-coloring on the sudoku dataset of 1 million puzzles provided on *kaggle*. We have seen that our simulated annealing algorithm passes ~99% of the cases (tried only 10000 for this) and takes on an average 0.0059s per puzzle, the backtracking naive version passes 100% cases and takes 0.000036s per puzzle on average, and our backtracking with strategies takes 0.000029s on average and also passes 100% cases. We can see that there is no much difference in the versions of backtracking here because sudoku graph is a dense graph where each node has a 20 edges (as shown in the figure below).



After this experiment we move onto the examples of our courses-scheduling problem, we try to imitate the graph of academic registration choices by taking examples of courses of different levels, and creating stub UG, PG and then taking their random choice (the protocol for this can be found in `dataset.py` file). The following graph is an example of one input.

It is interesting to note that such a dense graph can be assigned with just 8 slots (using advanced backtracking version, the naive version gives 12 and simulated annealing give 9-11 slots randomly).

If we face the current scenario of manual assignments, in this trimester there are 238 courses and a whooping 15 slots are used to cover them.



Summary and Conclusions

In our project, we have discussed how we make a system which takes a graph of variables and constraints, and which tries to solve for the k-coloring problem on the graph. This solution can also help us schedule our courses into slots for better management during academic registration with the help of AI. Students would have broader choices to choose from and that too without any constraint of slot mismatching. In this scheduling process the user can also feed the initial values for any course individually i.e he/she has the power of pre-painting any slot on any schedule.

This solution is feasible since it can give a correct schedule in matter of seconds and the computation can be done on regular machines. Although, If we give more time to our backtracking approached then they can sure explore a larger state space and then it might give a better coloring (it’s a no loss situation, you’ll always find a solution better than the previous one).

"The inspiration of this problem comes from EEL4330 which I couldn't choose since it clashes with CS311 in slot E." -Kunal Tawatia

References

1. Graph Coloring [Wikipedia]
2. Graph Coloring [Brilliant Wiki]
3. Graph Visualiser [CSAcademy]
4. Sudoku Dataset [Kaggle]