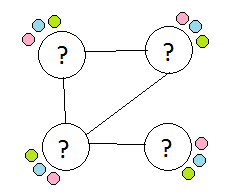
CSPs

Local search seems promising when you know very little about the problem, and the problem is too large to explore, but as you know, it takes a LOT of time. What if you could cast the problem into something more concise, with an algorithmic way to solve for the problem, instead of a brute force search, or a local search? Enter CSPs.

Propositional logic is extremely powerful as a problem modelling tool (if you ask any Algorithms enthusiast, they’ll tell you about how every hard problem can basically be converted into a CSP). It is by no means fast (the run time is still exponential), but if an exact solution is what you’re looking for, this is as good as it gets.

Revisiting Graph Colouring



For today’s lab, you will model the simple graph colouring problems from the previous lab as a CSP and solve it using the DPLL algorithm. We have provided a python script that contains the DPLL solver. You can run it using:

> *python dpll.py < input.txt*

The input to the DPLL algorithm is a standard format:

|  |
| --- |
| N  <C1>  <C2>  <CN> |

The first line is the number of clauses in the formula. Each subsequent line represents a clause, and each clause is a set of integers representing the literals in the clause. Negations are represented as negative integers.

For example, the formula ϕ = (X1 ∨ X2 ∨ ¬X3) ∧ (X1 ∨ X3) ∧ (¬X2 ∨ X3) Will be represented as:

|  |
| --- |
| 3  1 2 -3  1 3  -2 3 |

The two graph colouring test cases are in the same format as the previous lab. Convert the G(V,E) representation into a CNF formula and save it to a file for use with the dpll.py script.

The output is the truth assignments for every variable.

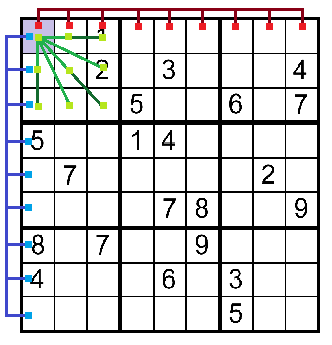
For example if a satisfying assignment is X1=0, X2=1, X3=1, the output will be:

|  |
| --- |
| 1 -1  2 1  3 1 |

NOTE: for gc\_4\_1 use #colours=2 and for gc\_20\_1 use #colours=3 (these are the optimum numbers of colours. In practice, you’d start at 1 colour, and keep calling a CSP for iteratively larger numbers of colours until you get a satisfying assignment)

You need to output the final colouring for each graph.

Sudoku



The rules of Sudoku are nearly identical to the rules of graph colouring. For example, ‘every element in a row should have different numbers’ is equivalent to ‘all elements of a row are neighbors and must have different colours’.

As a second exercise, cast the partially completed Sudoku instance given in the file sudoku.txt into a graph, and use the parsing program you developed in the previous section to cast the graph into a satisfiability problem. (You could just directly derive the clauses instead of constructing the graph, but it’s more fun this way)

*Input format: 9 rows of 9 space-separated integers representing filled locations, and ‘.’ Representing unfilled locations.*

HINT: The literals corresponding to every element that is already partially filled out must be forced to their appropriate truth assignment as part of the formula itself. After you construct the graph-derived formula, you have to append a new set of clauses to accommodate this.

You need to output the final, completed Sudoku grid.