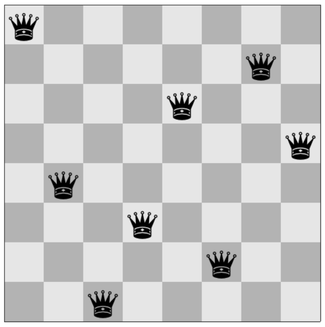
CS357: First Home Assignment
– Satisfiability –

This assignment is intended to be solved **individually**, but discussion via Piazza is encouraged. Submit your report and any implementation files in an archive via email to `zeljic@stanford.edu` with subject CS357 - Assignment 1. The deadline is **Sunday October 20th**. For programming tasks in the assignment you are free to use your programming language of choice.

1. N-Queens puzzle

The n-queens puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens threaten each other. Two queens threaten each other if they share a column, row or a diagonal. Below you can see an example solution, for $n = 8$:



1. Describe the high-level constraints of the problem.
2. Consider how each kind of constraint is encoded into propositional logic when using:
   a) sparse encoding and
   b) log-encoding

A **sparse encoding** uses a Boolean variable for every queen k at every position

i, j:

$$v_{i,j}^k = \begin{cases} True, \text{if } k^{th} \text{ queen is at position i,j on the board} \\ False, \text{otherwise} \end{cases} \quad , 1 \leq i, j, k \leq n$$

A **log-encoding** represents each queen as a pair of integer positions x and y, so for queen k at position i and j: $v_x^k = i$ and $v_y^k = j, 1 \leq i, j, k \leq n$. Integer values would be encoded into propositional logic using their binary representation, which takes $O(log(n))$ propositional variables, hence the name of the encoding.

1.1. Answer the following questions:

1. How many variables and clauses would be needed to represent the n-queen problem using each encoding?
2. What are the advantages and disadvantages of each encoding?
3. Are there any *redundant* constraints in the problem formulation?
4. Are there any *implicit* constraints that can be added to the encoding?
5. *Symmetry breaking* - solutions that represent rotations of another solution are not interesting. What kind of constraints can be added to avoid symmetric solutions?

1.2. Implementation

Implement a program, that encodes n-queens puzzle of into SAT using an encoding of your choice, where n is the input parameter. The program should generate a SAT formula in the DIMACS-cnf format.

2. Implement DPLL algorithm

Implement the DPLL algorithm in a language of your choice. Your implementation should take a file name pointing to a file in DIMACS-cnf format as input. The program should parse the input file and solve the problem by applying the rules of the DPLL algorithm. The code should be well documented.

For the report, note all the different design choices that you had to make during the implementation.

3. Tying it all together

1. Use your implementation of the DPLL algorithm to solve the n queens problem for various sizes of n. What is the highest n of the puzzle that your implementation can solve within 5 minutes?

2. Play around with the implementation and the encoding and document how different choices impact the performance. Can you improve the size of n the your implementation can solve within 5 minutes by modifying either the encoding or the program? Discuss your observations in the report.

While playing around, it might be interesting to look at the number of times different DPLL rules are applied. In particular, the number of split rule applications is a good metric for the size of the search space explored.

4. (Optional) Phase transition

Given a number of variables n and a ratio of clauses to variables r, generate random 3-SAT formulas. For a formula with n variables, the number of clauses should be $n \times r$. The distribution of variables and polarities in the generated formula should be uniform, meaning that all variables are equally likely (i.e. $1 : n$) and for each variable occurrence positive and negative literals are equally likely (i.e. $1 : 2$).

Choose a number of variables that your implementation can solve, based on the experiments with the encoding of the n-queens puzzle. For the chosen number of variables n, vary the values of the clause-to-variables ratio from 2 to 8, in increments of 0.2. For each value of the ratio generate 25 random 3-SAT formulas and solve them with some timeout (10 minutes should be sufficient). Count the number of SAT and UNSAT formulas for each value of the clause-to-variable ratio and plot the average run-times (or alternatively average number of splits), for each value of the ratio. What can you observe? Do you have an intuitive explanation for the observations?

Instead of using your implementation you can use a state-of-the-art SAT solver, but the size of formulas to be generated will need to be considerably larger, and even then you might need to plot different run time statistics to observe the effect.