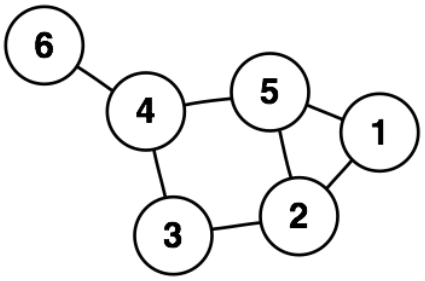
Discussion Problems: "SAT Encodings"
Declarative Methods (JHU 601.432/632)
Prof. Jason Eisner

1. Given an undirected graph G, the "Minimum Vertex Cover" problem asks you to find the smallest set S of vertices such that every edge of G has at least one endpoint in S. Continue reading for an example.

   (a) [8 + 4 **extra credit** points] Minimum Vertex Cover can be encoded as a weighted MAX-2-CNF-SAT problem. Give the encoding in the case of the graph G below.[1] Be sure to specify the clause weights. For extra credit, do it (correctly) without using any infinite weights.



   (b) [4 points] Under your encoding, give the assignment to your SAT variables that corresponds to the solution $\{2, 4, 5\}$, which has the minimum size of 3.[2]

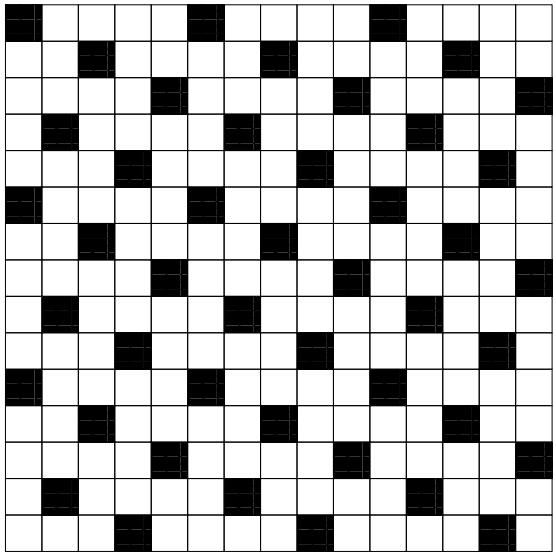
   (c) [5 points] It turns out that Minimum Vertex Cover is NP-complete. (More precisely, given a graph G and an integer k, it is NP-complete to determine whether there exists a vertex cover of size $\leq k$.) Use this fact to argue that MAX-2-CNF-SAT is probably much harder than 2-CNF-SAT. Your answer should be a few sentences long.

[1] Thanks to Wikipedia.

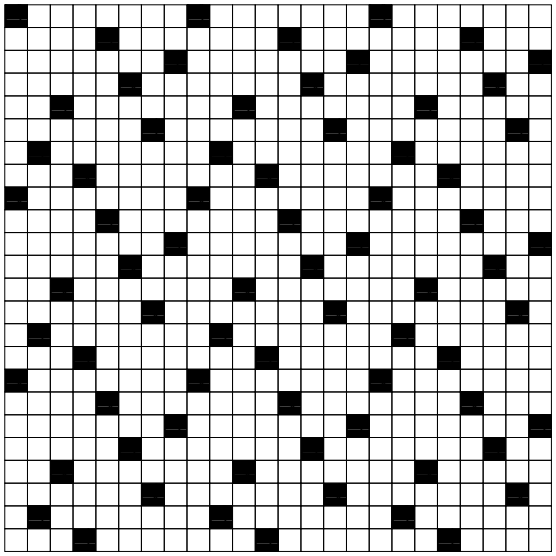
[2] Another solution of size 3 is $\{1, 2, 4\}$. A larger solution is $\{1, 3, 5, 6\}$; this is not a *minimum* vertex cover.

2. Don't panic: the figures below are not crossword puzzles! They are bathroom floors. They show some nice patterns for black-and-white floor tiles on an *infinite* rectangular grid. (Not shown here in full—but you can see how you'd extend the patterns forever, if you happened to be laying floor tile in an infinitely large bathroom.)

n=5



n=8



You will notice that in each infinite row, every n^{th} tile is black (and the rest are white). The same is true in each infinite column, and also along each infinite rightward diagonal, and also along each infinite leftward diagonal. When a pattern has all 4 of these properties, we call it a "royal pattern."

Observe that for $n = 1$, the only royal pattern is an all-black floor.

Given $n > 1$, you would like to use a SAT solver to find such patterns (if they exist).

(a) [3 points] Let's start with a simpler problem. Suppose we have 3 adjacent tiles, arranged like this: | A | B | C |. Let the variables A, B, C represent the propositions that tiles A,B,C are black, respectively.

Write a DNF-SAT expression in terms of A, B, C to state that *exactly one* of the 3 tiles is black. It should be clear how to generalize your solution to a sequence of more than 3 adjacent tiles (exactly one of which must be black).

(b) [2 points] Generalizing your solution to question 2a, how many DNF clauses are needed for n adjacent tiles? (Answer exactly, not with $O(\ldots)$ notation.)

(c) [3 points] Now write a CNF-SAT expression for the same thing. (It does not have to be 3-CNF-SAT.) Again, it should be clear how to generalize your solution to more than 3

tiles.

(d) [2 points] Generalizing your solution to question 2c, how many CNF clauses are needed for n adjacent tiles? (Answer exactly, not with $O(\ldots)$ notation.)

(e) [4 points] Now let's go to the full royal pattern problem. You want to reduce it to CNF-SAT. The interesting issue here is that you want to construct an infinite royal pattern, but SAT solvers can't deal with infinite formulas.

First, let's see how the steps should fit together. The job of your encoder is to take an integer n as input, and to output some CNF-SAT formula of finite size. What then is the job of the SAT solver, and what is the job of your decoder?

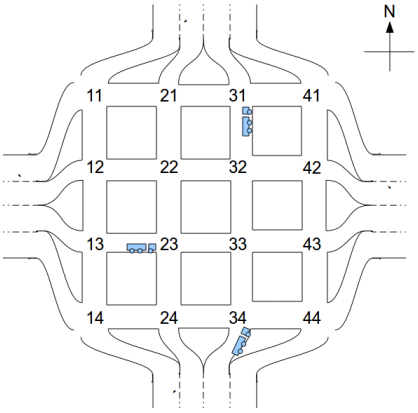
(f) [6 points] Now think of an encoding scheme. For large n, **exactly how many variables** and **exactly how many clauses** will be used in the CNF formula that your encoder produces?

Write your two answers as functions of n (again, do **not** use $O(\ldots)$ notation). Explain briefly in English.

(g) [3 points] What should your SAT solver return for $n = 2$? Why?

The following setup will be used for the next several questions. A mistake in one question should not hurt your answers to the other questions, so don't give up partway through.

Thruville's streets are organized as a $k \times k$ grid, with k east-west streets and k north-south streets. All streets are 2-way. The picture shows the case $k = 4$ (but you could imagine solving this problem for $k = 100$, with many vehicles).



You are a traffic engineer. Each intersection has an ordinary traffic light that you control in discrete time units. For example, at time 7, your light at intersection 32 will allow either *east-west* traffic or *north-south* traffic. You can control this by setting the variable 7L32 to be *true* or *false*, respectively.

Your goal is to maximize traffic flow over the next T time steps, in a sense to be defined later. The current time is 1. Your job is to pick the values of all of the $t\mathrm{L}xy$ variables for $t \in \{1, 2, \ldots, T\}$, $x \in \{1, 2, \ldots, k\}$, $y \in \{1, 2, \ldots k\}$.

This is a planning problem—much like the register allocation problem from lecture. For example, if there is a lot of eastbound traffic on 2nd St., you'd prefer to time the traffic lights so that each light will turn green by the time a vehicle reaches it. However, there are tradeoffs, since those east-west green lights would block north-south traffic.

Thanks to advanced sensors, you know the current location and direction of all vehicles in the city (i.e., at time 1). You can use this information, timing your lights to help the *actual* current traffic! The picture shows a vehicle heading east that needs to pass next through intersection 23. Your sensors record this as a value of *true* for the variable 1E23 ("E" for "eastbound"). Each intersection has "E," "W", "N", and "S" variables for the four directions.

Each city block is short and has room for at most one vehicle in each direction, so it is okay that 1E23 is encoded as boolean ("is there a vehicle there?") rather than integer ("how many vehicles are there?").

3. [5 points] Since you are doing your planning at time 1, you do *not* (yet) have access to the sensor variables at future times, such as 2E33.

   But for planning purposes, you assume that vehicles will behave deterministically: that they will always stop at a red light, and will always go straight through a green light if they can (no turns!).

   Stopping at a red light means that if 1E23 = *true* and 1L23 = *false*, then 2E23 = *true* also.

   Going through a green light is a bit more complicated because of traffic flow. If 1E23 = *true* and 1L23 = *true*, then ordinarily the 1E23 vehicle leaves its current position E23 and moves to the next position E33. (The rate of travel is 1 block per time step.) But remember, blocks are short: we cannot have two vehicles at position E33. So the 1E23 vehicle can only move if any existing 1E33 vehicle can move out of its way. An existing 1E33 vehicle may not be able to move because it faces a red light—or recursively, because an 1E43 vehicle in front of it can't move.

   At any rate, this deterministic behavior motivates a system of constraints that fully predicts all of the sensor variables at time $t+1$, based on the sensor *and* lights variables from time t.[3]

   Think of this system of constraints. Then, to prove you've got it, write down the specific constraints that determine whether 2E33 is true, based on the variables at time 1. *You do not have to put your constraints into CNF* (e.g., you can use $\leftrightarrow$).

   > *Hint*: Introduce some new helper variables at time 1 that indicate whether a vehicle can move. For example, M1E33 should be true if and only if a 1E33 vehicle could move ahead. Write the constraints on 2E33 in terms of some of these helper variables. But make sure to also write the relevant constraints on the helper variables that you use.

4. [4 points] Cars may exit Thruville to the surrounding highways. Your sensor 1N31 is *true*, according to the picture, so there is currently a car that wants to exit by going north through intersection 31. It will do so as soon as it gets a green light.

   Your goal is to minimize the total travel time of all cars, on the assumption that all cars want to drive straight through Thruville—no stopping and no turning.[4]

[3] You can't predict whether new cars will arrive from the surrounding highways, but just assume for planning purposes that they won't. Your sensor 1N34 = *true* indicates that at time 1 (right now), there is a car that wants to enter by coming north through intersection 34. This car will have to wait for a green light. You may assume, however, that no new cars from outside will replace it once it has entered. In other words, just assume that the only cars that will show up are the ones you already know about at time $t = 1$.

[4] I'll discuss in the solution handout how to improve this goal and avoid these assumptions, but for now just take them as given.

Since the cars' start times are fixed, this is equivalent to getting them out of Thruville as quickly as possible. That is, you want to minimize the sum of the exit times of all cars. In other words, you will be rewarded whenever a car leaves Thruville, but you will be rewarded 1 point more if it leaves at $t = 3$ than if it leaves at time $t = 4$.

Write down the soft constraints that reward cars for leaving north via intersection 31. (The constraints for the other intersections would be similar.) For each of these soft constraints, give the weight for MAX-SAT.

5. [4 points] How should you set T?

6. [5 points] Suppose you have a solver for the Thruville problem. (It might call a SAT solver, or it might be specialized for the traffic planning problem.) The solver input consists of the grid size k, the planning length T, and all the sensor values at time 1. The output consists of a schedule for the lights at all times $t = 1, 2, \ldots, T$.

   You want to use the solver to solve the Megaville problem instead. Megaville is just like Thruville, but it has longer blocks. Each block can accommodate up to 3 vehicles in each direction, not just 1. It also takes 3 time steps for a vehicle to get from one intersection to the next, not just 1.

   Describe a simple problem reduction that will let you use the Thruville solver to correctly solve the Megaville problem. Explain why it works.