601.432/632 Declarative Methods - Prof. Jason Eisner
Answers to "SAT Encodings" discussion problems

1.
(a) We have two types of constraints:
- hard: each edge is included in the cover
- soft: the cover contains the fewest amount of nodes.

The edge constraints can be written as:
(infinity, X1 v X2)
(infinity, X1 v X5)
(infinity, X2 v X3)
(infinity, X2 v X5)
(infinity, X3 v X4)
(infinity, X4 v X5)
(infinity, X4 v X6)

And the minimality constraint can be written as:
(1, ~X1)
(1, ~X2)
(1, ~X3)
(1, ~X4)
(1, ~X5)
(1, ~X6)

[EC] Replace each infinity by 7 (or any number > 6).
Then the solver will in fact never violate any of the hard constraints, since that would cost 7 points, whereas just making all the variables true (for a vertex cover of size 6) would only cost a total of 6 points.

(b) X1=0, X2=1, X3=0, X4=1, X5=1, X6=0

(c) 2-CNF-SAT is in P, i.e. it can be solved deterministically in poly time. In the exercise, we showed a reduction of Minimum Vertex Cover to MAX-2-CNF-SAT. So, MAX-2-CNF-SAT is at least as hard as Minimum Vertex Cover, which is NP-complete. So, if we could solve MAX-2-CNF-SAT efficiently, then we could solve Minimum Vertex Cover efficiently, which would imply NP = P.

2.
(a) We want to encode the statement:
"tile A is black" OR "tile B is black" OR "tile C is black"
that is,
(A ^ ~B ^ ~C) v (~A ^ B ^ ~C) v (~A ^ ~B ^ C)

(b) Assuming the scheme above, n.

(c) We can rewrite the statement in 2(a) as:
"at least one tile is black" AND
"no two tiles can be black at the same time"

Consider tiles A,B and note that
~(A ^ B) iff (~A v ~B)
so we conclude that the CNF statement is
(A v B v C) ^ (~A v ~B) ^ (~A v ~C) ^ (~B v ~C).

(d) Assuming the scheme above, $1 + choose(n,2) = 1 + n(n-1)/2$.

Note: If we wanted 3-CNF-SAT, we would need to break the first "long" clause into several 3-clauses.

(e) The SAT solver should find an assignment of variables to the finite formula.
The decoder should turn that assignment into an infinite grid.

(f) Any royal pattern is necessarily an $n \times n$ grid, repeated forever in all directions by concatenation. For this to yield a

royal pattern, the $n \times n$ grid must have exactly one black square in each of the n rows, in each of the n columns, and on each of the n rightward diagonals and n leftward diagonals.

But wait: aren't there more than n rightward diagonals, and aren't some of them shorter than length n (so that they don't necessarily deserve a black square)?

Well, it turns out that you have to interpret diagonals as "wrapping around." For example, the "exactly one black square" constraints in the 4 x 4 grid below should be on the rows ABCD, EFGH, IJKL, MNOP, the columns AEIM, BFJN, CGKO, DHLP, the wrapping-around rightward diagonals AFKP, BGLM, CHIN, and EJOD, and the wrapping-around leftward diagonals DGJM, HKNA, LOBE, PCFI.

```
A B C D
E F G H
I J K L
M N O P
```

The wrapping around is justified because if you tile the infinite plane with this grid, you'd see that 1/4 of the infinite rightward diagonals ran as ...EJODEJODEJOD..., just as 1/4 of the infinite rows have the form ...ABCDABCDABCD...

In short, there are n^2 variables, with $4n$ "exactly one" constraints each of which is implemented by $1 + n(n-1)/2$ clauses (assuming the answer in d. above). So the total number of constraints is:

$$4n + 2n^2(n-1) = 2n^3 - 2n^2 + 4n = (2n)(n^2 - n + 2)$$

Note: If we were constraining along ordinary diagonals of the $n \times n$ grid rather than wrapped diagonals, then finding such grids would be equivalent to the n-queens problem: place n queens on an $n \times n$ chessboard so that no queen can take any other queen. That's why I decided to call these "royal patterns." We'll probably look at the n-queens problem during our next unit.

(g) Let's start by drawing all 2 x 2 grids with the row and column constraints satisfied:

```
X .     and     . X
. X             X .
```

We can see that these patterns are in contradiction with the diagonal requirement, so the SAT solver should return UNSAT.

3. We will define the time t helper variables and time t+1 sensor variables using iff constraints (denoted as <->):

This captures the deterministic behavior of the cars. When we write

X <-> some formula

then the value of X is completely determined by the value of that formula.

There are a few reasonable solutions, depending on exactly how you define the helper variables. In understanding the solutions below, notice that any instance of (~X v Y) could also be written as (X -> Y).

Recall that we are interested in the truth value of 2E33.

Version 1:

Let M1E23 be a helper variable that describes:

- There is an 1E23 vehicle

```
        AND
        - It can move ahead,
        that is:
            M1E23 <-> 1E23 ^ 1L23 ^ (~1E33 V M1E33),
        with M1E33 defined similarly

    Then, 2E33 is true iff:
        - There is an 1E33 vehicle that cannot move
        OR
        - The vehicle 1E23 can move ahead (i.e., M1E23),
        that is:
            2E33 <-> (1E33 ^ ~M1E33) v M1E23

Version 2: M1E23 means that *if* there is an
1E23 vehicle, it can move ahead.  This is the logical sense of
"if," so M1E23 will be true when there is no 1E23 vehicle.
Equivalently, M1E23 means "there is not a stuck 1E23 blocking the
way."

2E33 <-> ~M1E33 v (1E23 ^ M1E23)
        "2E33 iff 1E33 is stuck or 1E23 moves."
M1E23 <-> ~1E23 v (1L23 ^ M1E33)
        "1E23 is non-stuck iff it doesn't exist or it has a green
       light and 1E33 isn't stuck in front of it."
M1E33 defined similarly.

Version 3: M1E23 means that the conditions are right for a 1E23
vehicle to move ahead. However, it doesn't care whether there is
such a vehicle.

2E33 <-> (1E33 ^ ~M1E33) v (1E23 ^ M1E23)
    "2E33 iff 1E33 is stuck or 1E23 moves."
M1E23 <-> 1L23 ^ (~1E33 v M1E33)
    "The conditions are right for a 1E23 vehicle to move iff it has
    a green light and IE33 isn't stuck in front of it."
M1E33 defined similarly.

Remarks on how this pattern would generalize to positions other
than 2E33:

-   In the constraints for north-south cars, the L variables (such as
    1L23) would be negated.

-   In each version above, M1E23 has a clause saying "nothing is
    stuck in front."  When defining variables like M1E43 at the edge
    of the grid, nothing can possibly be stuck in front, so this
    clause can just be deleted. (Otherwise it would have to refer to
    M1E53, which doesn't exist!)

Remark on the use of <-> constraints:

There is an alternative setting in which you control the cars
as well as the lights.  So you are allowed to delay one car
to improve the traffic flow for other cars, or just because
you can.  In that case, instead of writing
    M1E23 <-> ...
you can write
    M1E23  -> ...

Now M1E23 is no longer deterministic: you can control it in the
assignment and decide whether to make the car move.  However,
there's still a -> constraint: if you make it move (make M1E23 true),
then you must have a green light and no car in front of you.

Remark on how to avoid these deterministic predictions:
```

You could create some other variables that affect decisions of the vehicles. (For example, if R2E33 is true, then a 2E33 vehicle that can move at all will turn right instead of going straight.) Different random settings of these variables will give rise to different scenarios, with different predicted sensor variables and different optimal plans.

You can then use a MAX-SAT solver to find a plan for the lights that will minimize the average travel time across 30 random scenarios. This is known as "roll-out" because it guesses the future by rolling dice (it is used for computer backgammon, for example).

To find such a plan, make 30 copies of the problem, each with its own sensor variables for times $t > 2$, and each with its own random settings that determine how those sensor variables play out. But have all these copies share the same set of known sensor variables at time t = 1, and the same lights variables. That way, you are looking for a single assignment to the lights variables that achieves a good total MAX-SAT score, summed over the 30 copies. (The sum is 30 times the average, so maximizing the sum is equivalent to maximizing the average.)

4. A constraint that says
"a car should leave northward via intersection 31 at time t"
will have the form
"a car is waiting to go north and has a green light",
that is:
(tN31 ^ tL31)

(Some of you wrote tN31 -> ~tL31, but that constraint would also be satisfied whenever tN31 was false, which does not not necessarily mean that a car was exiting the city.)

Or if you have defined helper variables as in version 1 of the answer to question 5, you could just write
MtN31 <-> "a car moves northbound"

What is the weight of this constraint? A reasonable answer (adequate for purposes of the exam) is that you give it a weight of 1+T-t. Then you are rewarded with:
T points for leaving at time 1,
T-1 points for leaving at time 2,
...
1 point for leaving at time T (still better than 0!)

However, a better scheme would be to add at least (C-1)(T-1) to all of these weights, where C is the total number of cars. Then you get

(C-1)(T-1) + T points for leaving at time 1,
(C-1)(T-1) + T-1 points for leaving at time 2,
...
(C-1)(T-1) + 1 points for leaving at time T.

This ensures that the MAX-SAT solution will find a T-step plan that actually gets all C cars out of Thruville, assuming that T is large enough to make this possible (see question 7).

Why? Because for every car that you don't get out, you are now giving up more than (C-1)(T-1) points. Might that loss be worth it in order to speed up the other cars? Never! Even if it allowed you to massively speed up all of the other C-1 cars, getting them all out at time 1 instead of time T, then that would only gain

(C-1)(T-1) points.

Of course, (C-1)(T-1) is just the minimum thing to add that will have this effect. You could just as well add a larger number. For example, $4k^2 T$ is simple to compute and write and is guaranteed to be > (C-1)(T-1). Or you could just add 9999999, hope it's big enough, and pray that no one ever reads your code. :-)

Remark on soft versus hard constraints:

As an alternative to manipulating the weights like this, you could just write hard constraints that says you must get all C cars out of Thruville. In our setting, these constraints are pretty easy: just say each position in the grid is empty at time T, e.g., ~TE23 "no car waiting at time T to go east through intersection 23" However, again you'd have to make the weights on these constraints sufficiently high to turn them into hard constraints.

Remark on realism:

Consider the generalization where you deal with continuous traffic flow, with new cars entering all the time. In that case, it is harder to ensure that all of the cars that are in the grid at time 1 are out by time T, since there are other new cars in the grid. You'd have to make your encoding keep track of individual cars.

(It's not enough to still just reward arbitrary cars for leaving. Why not? Because then if there's lots of east-west traffic, you'd have an incentive to maximize your score by leaving all of the east-west lights green all the time, and to heck with the poor north-south car that never gets through This problem is known as "starvation," and a good traffic engineer would like to control the lights such that no one starves.)

Another remark on realism:

As a real traffic engineer, you wouldn't just try to get the cars out as quickly as possible. You'd also have some soft constraints against stopping. If a car is going to traverse 3 blocks in 6 time steps, then "drive drive drive wait wait wait" (1 stop) is much better than "drive wait drive wait drive wait" (3 stops). It saves gasoline, reduces accidents, and makes driving easier and more pleasant for the driver.

5. Remember that in the register allocation problem, we set T large enough that we knew it was at least possible to satisfy the hard constraints. Thus, the MAX-SAT solver would certainly be able to produce a decodable solution, and the only question is whether it produces one that does well on the soft constraints.

   In the Thruville problem, we might like to set T large enough that we can get all the cars out. For example, $T = 2*(k+1)$ is big enough, since we can then get all the cars out by leaving the north-south lights green for k+1 steps and then turning the east-west lights green for k+1 steps.

   This also gives us enough lookahead to do well with respect to footnote 4.

6. Add more intersections to Megaville so that each Megaville block turns into 3 blocks (with 2 new intersections along the block).

   You can do this by tripling the number of east-west streets and also the number of north-south streets. This reduces your k by k Megaville problem to a 3k by 3k Thruville problem. You haven't

put any traffic on the new streets---they just serve to cut each Megaville block into thirds.

What about the traffic lights at the 2 new intersections you've inserted into each Megaville block? They don't matter---the Thruville solver will always be able to give an immediate green light to a car passing through one of the new intersections, because there is never any competing traffic trying to cross that intersection on the new extra street.

(Some of you wanted to force the Thruville solver to make these lights permanently green. That would indeed be a valid solution, but remember this is a problem reduction---you are using an existing Thruville solver and can't force it to do any such thing. All you can do is give it a problem and let it pick the lights. Fortunately you don't need any more control than that.)

How fast is this? Well, suppose the Thruville solver runs in polynomial time, such as O(k^d) for a Thruville problem of size k, where d is a constant. Then it will be able to solve a Megaville problem of size k in time O((3k)^d) = O(3^d k^d) = O(k^d) (the constant 3^d is absorbed into the big-O notation).