GR2: 2-SAT

Notes for CS-8803-GA: Introduction to Graduate Algorithms

Georgia Tech (Dr. Eric Vigoda), Fall 2017

as recorded by Brent Wagenseller

We will look at the **satisfiability problem** (aka **SAT** problem), which will be used later in our NP-completeness study.

**Boolean formula**

Variables: x_1, x_2,\dots,x_ntaking values True or False

Literals: x_1,\overline{x_1},x_2,\overline{x_2},\dots,x_n,\overline{x_n}.

There are 2n literals, for example () with the bar variable indicating it’s the opposite (so if x = True, bar x = False).

Operators: \vee=OR, \wedge=AND.

Clause is an OR of several literals. Example: (\overline{x_3}\vee x_5\vee\overline{x_2}\vee x_1).

Boolean formula fis in **conjunctive normal form** (**CNF**) means that it is the AND of msuch clauses. Example: (x_2)\wedge(\overline{x_3}\vee x_5\vee\overline{x_2}\vee x_1)\wedge(\overline{x_2}\vee\overline{x_1}).

In the above example, there are 3 clauses, and each clause is the OR of several literals. For example, in order to satisfy the clause , x3 would have to be False, x5 would have to be True, x2 would have to be false, or x1 would have to be True.

In the example, if all of the clauses are satisfied, the formula is satisfied. Any formula can be convered to the CNF form, but the size of the formula may ‘blow up’.

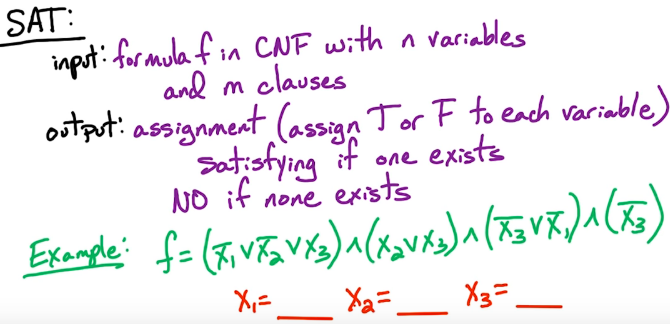
**SAT**

SAT seems to be some algorithm that is fully comprised of Boolean TRUE and FALSE values. There are clauses that are comprised of ‘literals’ (xn variables that can be True or False; they can be negated and they are usually joined by ORs in the clause while the clauses are joined with an AND. One type – 2-SAT, which means no more than 2 literals in each caluse – can be solved in polynomial time using Strongly Connected Components.

Input: given formula fin CNF with nvariables and mclauses.

Output: assignment satisfying fif one exists, and NO if no satisfying assignment exists (so it seems it either outputs a correct set of True/False for each variable, or NO if its impossible).

Quiz:

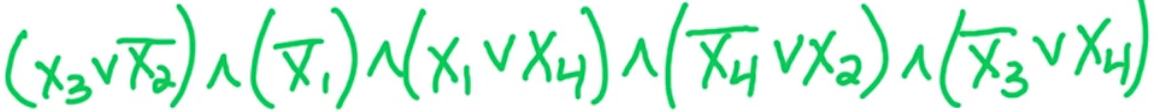


Answer: There is only one answer: x1 = F, X2 = T, x3 = F

k-SAT: same as SAT but the input fhas clauses with \le kliterals in each (so, for example, if k = 3 we would have a bunch of clauses with no more than 3 literals in a clause). We’ll see that SAT is NP-complete and k-SAT is NP-complete for each k\ge 3.

**Simplifying Input**

Consider input f for 2-SAT:



Simplify:

1. Remove unit clauses (clauses with 1 literal). In our case, we see that if there is any hope of making this true x1 must be False; so now we can remove the clause that completely relies on x1 (with the stipulation now that x1 is False).
2. Satisfy it (that is to say, do what we did above and make the unit clause evaluate to True)
3. Remove clauses containing x1 & drop bar(x1)
   1. We can eliminate x1 from any clause IF it evaluates to false as its useless; IF x1 evaluates to True for that clause we can eliminate the entire clause; we are left with



* 1. Now notice that we now have another unit clause: x4. We repeat steps 1 - 3

1. If we are left with an empty set, the SAT has been satisfied OR we end up with a formula where all clauses are of size exactly 2 (as we eliminated the ones and threes were never possible to begin with)

Formula f is only satisfiable if f’ (that is to say, the reduced formula above) is satisfiable.

Now, for the algorithm we write to solve, we can assume that the input to the 2-SAT problem will be of size exactly 2.

The setup is we now have a formula f with all clauses of size 2, n variables, m clauses. We want to convert this logic problem into a graph problem. To do this, we will take this and make a directed graph. The vertices of the graph will correspond to the variables in the formula; there are n variables and 2N literals; ultimately, we will have 2n vertices, one vertex for each literal. We will also have 2m edges in the graph; 2 edges for each clause; each clause has 2 ‘implications’, so we need an edge for each.

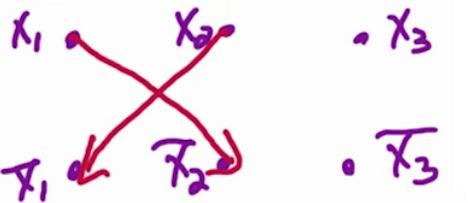
Consider this (new) formula to graph:



This has 3 variables and 3 clauses; our graph will have 6 vertices – one for each possible literal:



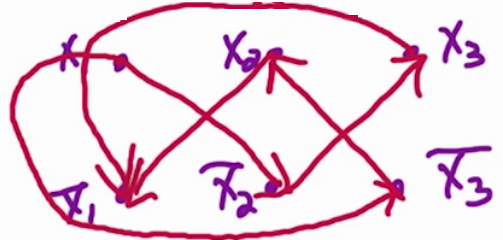
Lets look at the first clause “bar(x1) ⋁ bar(x2)”. For the time being, lets set x1= T; this means that, for the first clause, the first literal is not satisfied, so for the clause to be satisfied, bar(x2) MUST be True (or, rather, x2 MUST be False). Now if instead of x1= T, we set x2 = T, this would mean x1 MUST be False. These relationships can be captured in a graph:



Each of these edges is known as ….. an **implication**.

I believe the graphs only represent Trues; so for example, the arrow from x1 to bar(x2) means “If x1 is True, bar(x2) MUST be True”.

We can use this logic to draw the rest of the graph:



This graph is saying that

In general, if we have (α,β), we will have bar(α) → β and bar(β) → α

**Claim**: 2-SAT has a polynomial-time algorithm.

Take input ffor 2-SAT. We’ll assume all clauses have size exactly 2. What about clauses of size 1?  A clause of size 1 is called a `unit clause’.

For f, take a unit clause, say literal a_i:

* satisfy this clause by setting a_i=T
* remove all clauses containing a_i(these are all satisfied)
* drop any occurrences of \overline{a_i}.

Let f'be the resulting formula.  
**Observation**: fis satisfiable \Leftrightarrowf'is satisfiable.

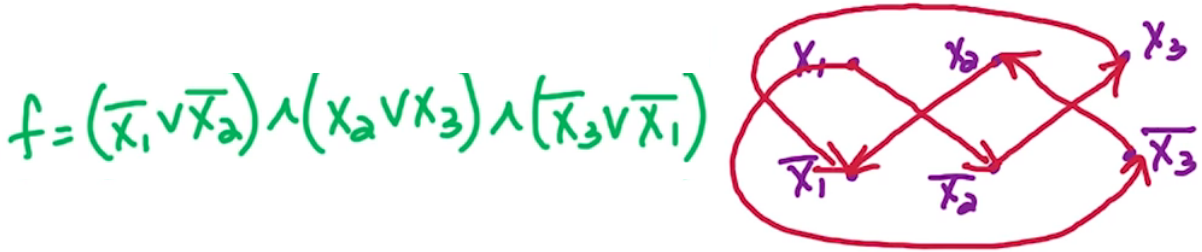
Thus we can repeat the above procedure until all unit clauses are eliminated.  Then we are left with a formula where all clauses are of size exactly 2, or we have some empty clauses (the clauses were never eliminated but the literals were dropped).  If we have an empty clause then the formula is not satisfiable, so we just output NO.

Take fwith clauses of size exactly 2, and nvariables and mclauses. Create a directed graph as follows:

* 2nvertices corresponding to x_1, \overline{x_1},x_2,\overline{x_2},\dots,x_n,\overline{x_n}
* 2medges corresponding to 2 “implications” per clause
* example: (x_i\vee\overline{x_j})
  + if x_i=Fthen we need to set x_j=Fto satisfy this clause.
  + if x_j=Tthen we need x_i=T.
  + so we include edges \overline{x_i}\rightarrow\overline{x_j}and x_j\rightarrow x_i.
* In general for clause (\alpha\vee\beta), add edges \overline{\alpha}\rightarrow\betaand \overline{\beta}\rightarrow\alpha.

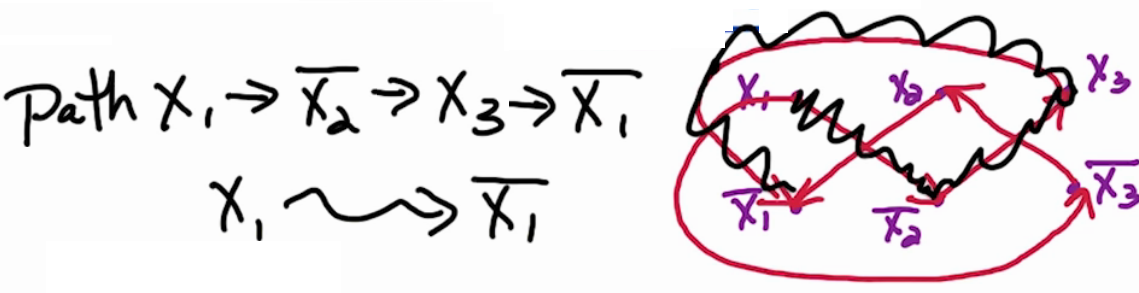
**Example**: f=(\overline{x_1}\vee\overline{x_2})\wedge(x_2\vee x_3)\wedge(\overline{x_3}\vee\overline{x_1}). (see PDF for the graph). Note path x_1\rightarrow\overline{x_2}\rightarrow x_3\rightarrow\overline{x_1}. So if x_1=Tthen we need \overline{x_1}=T (i.e., x_1=F), contradiction. But if x_1=Fthen we can set x_2=Tand x_3be anything to satisfy f.

This example – and its associated graph - is below:



Again, I believe the graphs only represent Trues; so for example, the arrow from x1 to bar(x2) means “If x1 is True, bar(x2) MUST be True”.

One interesting path is we can go from x1 to bar(x1); observe:



The x1 ↝ bar(x1) (and notice, this squiggly is how to gloss over in-between vertexes) indicates that “if x1 is True, bar(x1) MUST be True” which is impossible via contradiction. If we look at x1 = False, there are no edges out of that node. This all means that we cannot depend on x1 in any capacity as it will toggle other critical switches. NOTE: if we ALSO had bar(x1) ↝ (x1), the entire formula is not solvable. So if there are BOTH paths for one x leading to its opposite, the formula is not satisfiable.

To expand on this, if x1 ↝ bar(x1) and bar(x1) ↝ (x1) are in the same SCC, the formula is not satisfiable; this is true for ANY variable. If no variables lie in the same SCC as its negation, the formula is satisfiable.

**Note**: a path x_i \leadsto x_jmeans setting x_i=Twe must also set x_j=T. Thus if there’s a path x_i\leadsto\overline{x_i}then we can’t set x_i=Tand satisfy f. Similarly if there’s a path \overline{x_i}\leadsto x_i. Therefore if x_iand \overline{x_i}are in the same SCC, then fis unsatisfiable.

**Lemma**: fis satisfiable  \Leftrightarrow\ \forall_i, x_i and \overline{x_i}are in different SCC’s.; in other words, if for all i, x1 & \overline{x_i}are in different SCCs, then f is satisfiable

We just argued that if \exists_iwhere x_iand \overline{x_i}are in the same SCC, then fis unsatisfiable. Thus we proved \Rightarrow. We now need to show \Leftarrowwhich we’ll do by giving an algorithm.

**Key idea**: take a sink SCC Sand set S=T(by satisfying all literals in S). Note that Shas no outgoing edges so no implications from setting S=T. But we’ve also set \overline{S}=F(does \overline{S}have incoming edges?)

Take a source SCC S'and set S'=F. Note that S'has no incoming edges so we’ll never be forced to set S'to T. But we’ve also set \overline{S'}=T(does \overline{S'}have outgoing edges?)

**Lemma**: Sis a sink SCC \Leftrightarrow\ \overline{S}is a source SCC.

So we take a sink SCC S:

* set S=T(and thus \overline{S}=F)
* remove Sand \overline{S}
* repeat until empty graph

This is valid because no variable x_ihas x_iand \overline{x_i}in the same SCC. Just need to prove the lemma now.

**Claim**: \alpha\leadsto\beta\Leftrightarrow\overline{\beta}\leadsto\overline{\alpha}. (\leadsto means a path)

**Proof**: for \Rightarrow: suppose path \alpha\leadsto\betais \gamma_0\rightarrow\gamma_1\rightarrow\cdots\rightarrow\gamma_lwhere \gamma_0=\alphaand \gamma_l=\beta. Note that the edge \gamma_i\rightarrow\gamma_{i+1}comes form the clause (\overline{\gamma_i}\vee\gamma_{i+1})and the other edge is \overline{\gamma_{i+1}}\rightarrow\overline{\gamma_i}. Thus we have the path \overline{\gamma_l}\rightarrow\overline{\gamma_{i-1}}\rightarrow\cdots\rightarrow\overline{\gamma_0}where \overline{\gamma_l}=\overline{\beta}and \overline{\gamma_0}=\overline{\alpha}.

For \Leftarrow, similarly we can construct a path \alpha\leadsto\betafrom a path \overline{\beta}\leadsto\overline{\alpha}.  \Box

Take SCC S. For \alpha,\beta\in S, there are paths \alpha\leadsto\betaand \beta\leadsto\alpha. Thus there’s also paths \overline{\beta}\leadsto\overline{\alpha}and \overline{\alpha}\leadsto\overline{\beta}. So \overline{\alpha}and \overline{\beta}are in the same SCC S'. Therefore \overline{S}is a SCC. So S is a SCC \Leftrightarrow\ \overline{S}is a SCC.

Moreover, \overline{S}has no incoming edges (so source SCC) if and only if S has no outgoing edges (sink SCC). This proves the lemma.

2-SAT Algorithm

2-SAT(f):

Step 0: Simplify f so all unit clauses are eliminated

#we are guaranteed that all clauses are exactly of size 2

Step 1: Construct graph G for f

Step 1.a: Run SCC on G

#we now have a SCC with the components in topological order

Step 2: Take a sink SCC S

#this will be the last component on the list; call this S

Step 2.a: Set S = True and !S = False

#satisfy all the individual literals in S and unsatisfy in !S

Step 2.b: Remove S and !S

Step 2.c: Repeat until empty