Linear time algorithm for testing Satisfiability of Propositional Horn Formulae

Sai Sanjeev Balakrishnan and
Shreepranav Varma

27^{th} May 2021

REFERENCES:

LINEAR TIME ALGORITHM FOR TESTING THE SATISFIABILITY OF PROPOSITIONAL HORN FORMULAE
- By William F. Dowling and Jean H Gallier

Horn Formulae

A Horn formula is a formula which in CNF has at most one positive literal per clause. In other words, if a Horn clause is defined to be a disjunction of literals in which at most one literal is positive, then a Horn formula is a conjunction of Horn clauses.

Examples

$(P_1) \wedge (\neg P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_2 \vee P_4)$
$(\neg P_1 \vee \neg P_2) \wedge (\neg P_2 \vee P_3) \wedge (P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_3 \vee P_5)$

Horn Formulae

Remark

Note that each Horn clause is equivalent to an implication where the antecedent is a conjunction of literals. For example, the Horn formula
$(P_1) \wedge (\neg P_1 \vee \neg P_2 \vee P_3) \wedge (\neg P_2 \vee \neg P_4)$
is equivalent to
$(True \implies P_1) \wedge ((P_1 \wedge P_2) \implies P_3) \wedge ((P_2 \wedge P_4) \implies False)$

Satisfiability of Horn Formulae

When we look for a minimal satisfying assignment, in the sense that the satisfying assignment gives True to as few propositional letters as possible, it is intuitive that a propositional letter is made true iff it is on the right hand side of an implication and that all propositional letters in the left hand side have already been assigned true.
Indeed, this is the strategy we shall employ to check the satisfiability of a Horn formula. In order to do this though, we shall first construct an edge-labelled directed graph from a given Horn Formula.

Graph associated with a Horn Formula

Let A be a Horn Formula. Then we construct graph G_A from A as follows:

- G_A has a vertex for each propositional letter in A, as well as two additional vertices – T for *True* and F for *False*.

Graph associated with a Horn Formula

Let A be a Horn Formula. Then we construct graph G_A from A as follows:

- G_A has a vertex for each propositional letter in A, as well as two additional vertices – T for *True* and F for *False*.
- For each Horn Clause in A consisting of a single positive literal P_i, there is an edge from T to P_i, labelled by the clause number.

Graph associated with a Horn Formula

- For each Horn Clause in A consisting only of negative literals $\neg P_{i_1}, \neg P_{i_2} \ldots \neg P_{i_k}$, there is an edge from each P_{i_j} in the clause to F, labelled by the clause number.

Graph associated with a Horn Formula

- For each Horn Clause in A consisting only of negative literals $\neg P_{i_1}, \neg P_{i_2} \ldots \neg P_{i_k}$, there is an edge from each P_{i_j} in the clause to F, labelled by the clause number.
- For each Horn Clause in A consisting of the negative literals $\neg P_{i_1}, \neg P_{i_2} \ldots \neg P_{i_k}$ and the positive literal P_l, there is an edge from each P_{i_j} occurring negatively in the clause to P_l, labelled by the clause number.

Example 1

$$P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$$

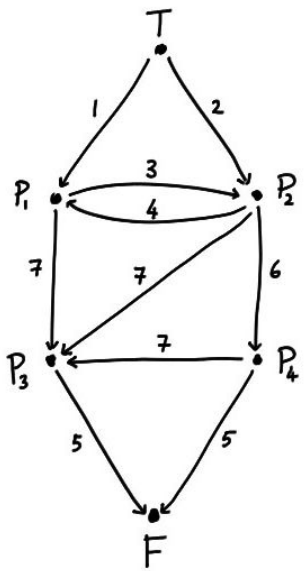


Figure: Graph associated with the formula

Example 2

$$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$$

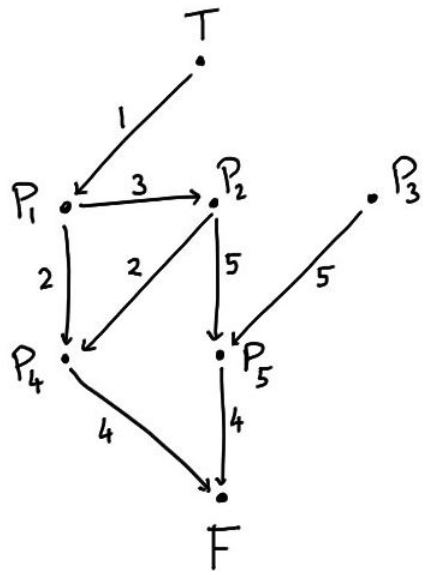


Figure: Graph associated with the formula

Graph Pebblings

Given propositional letter Q, we say that there is a pebbling from T to Q in graph G_A if:

- There is an edge from T to Q in the Graph G_A, or

Graph Pebblings

Given propositional letter Q, we say that there is a pebbling from T to Q in graph G_A if:

- There is an edge from T to Q in the Graph G_A, or
- For some Horn clause numbered n, there is a pebbling from T to P_i for all vertices P_i such that there is an edge labelled by n from P_i to Q.

Example 1

$$P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$$

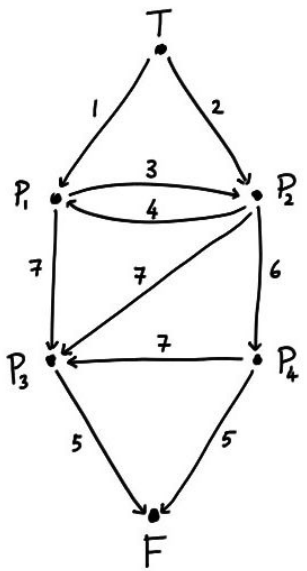


Figure: Graph associated with the formula

Example 2

$$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$$

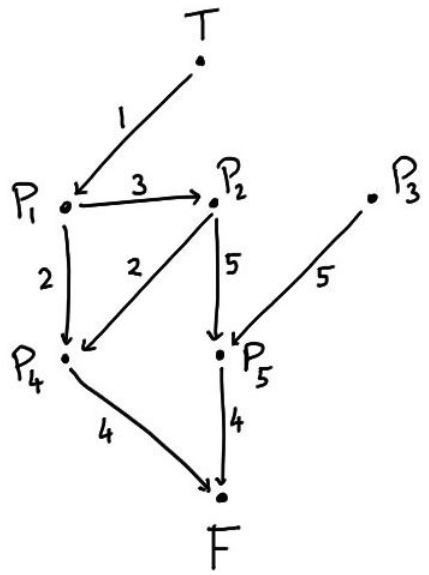


Figure: Graph associated with the formula

Length of a Pebbling

The length of a pebbling from T to Q in graph G_A is defined recursively as follows:

- The length is 1 if there is an edge from T to Q in the Graph G_A

Length of a Pebbling

The length of a pebbling from T to Q in graph G_A is defined recursively as follows:

- The length is 1 if there is an edge from T to Q in the Graph G_A
- If for some Horn clause $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k} \vee Q)$ numbered n, if the length of pebbling from T to P_{i_j} is d_j for each P_{i_j} in the clause, then the length of the pebbling to Q is defined to be $1 + max\{d_1, \ldots, d_k\}$

Soundness

Theorem

Let A be a Horn formula, and let G_A be its associated graph. If there is a pebbling of F from T in G_A, then A is unsatisfiable.

Proof

Suppose for the sake of contradiction that v is a satisfying assignment. By induction on length of pebblings, we shall show that if there is a pebbling from T to Q, then v must evaluate Q to *True*. Clearly this is trivial for length 1. Now suppose there is a pebbling of length d to Q through the clause $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k} \vee Q)$. Then there are pebblings of length $d-1$ or lower to each P_{i_j}.

Soundness

Proof (Contd.)

By induction hypothesis, each P_i must be evaluated to *True* by v, and since the Horn clause has to be satisfied, even Q must be evaluated to *True* by v.
Now, since there is a pebbling to F, there must be some Horn clause $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k})$ in which there are pebblings to each P_{i_j}. However this means that v evaluates each P_{i_j} to *True*. Hence, v cannot possibly satisfy $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k})$. This gives us our contradiction.

Completeness

Theorem

Let A be a Horn formula and G_A be its associated graph. If there is no pebbling of F from T, then A is satisfiable.

Proof

Define valuation v as follows – v evaluates P to *True* iff there is a pebbling of P from T. We shall show that this satisfies all the Horn clauses in A.

- If P is a Horn clause in A, then there is a pebbling of length 1 from T to P, so v evaluates P to *True* and thus this clause is satisfied.

Completeness

Proof (Contd.)

- If $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k}) \vee Q$ is a Horn clause, if even one of the P_{i_j} is evaluated to *False* by v, then the clause is obviously satisfied. So assume each P_{i_j} is evaluated to *True*, i.e., there is a pebbling of each P_{i_j} from T. Hence, there is a pebbling of Q from T, so v evaluates Q to *True* and the clause is satisfied.

Completeness

Proof (Contd.)

- If $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k}) \vee Q$ is a Horn clause, if even one of the P_{i_j} is evaluated to *False* by v, then the clause is obviously satisfied. So assume each P_{i_j} is evaluated to *True*, i.e., there is a pebbling of each P_{i_j} from T. Hence, there is a pebbling of Q from T, so v evaluates Q to *True* and the clause is satisfied.
- If $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k})$ is a Horn clause, we claim that at least one of the P_{i_j} must be evaluated to *False*. If not, then there must be a pebbling of each P_{i_j} from T, giving a pebbling of F from T. This contradicts our assumption. Hence, v must satisfy the clause.

Example 1

$$P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$$

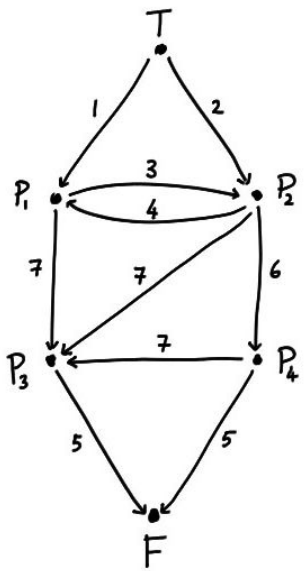


Figure: Graph associated with the formula

Example 2

$$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$$

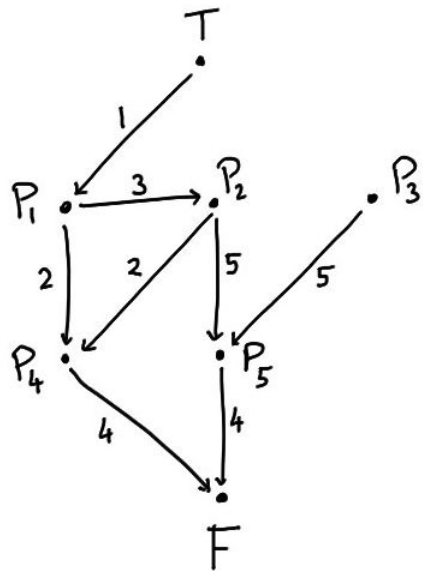


Figure: Graph associated with the formula

Naïve Algorithm

The basic goal of each algorithm we discuss is to find all vertices to which pebblings from T exist. The naïve approach is to run through all clauses to build up on the existing pebblings.

- If we reach F, then we immediately return UNSAT.
- If no new vertices are reached in this iteration, then clearly subsequent iterations won't build up on the pebblings either, so we have found all vertices to which pebblings from T exist.
- However, if we reach new vertices during this iteration, we set it them *True* and we now have to run through all Horn clauses from the beginning again, since these new pebblings could now possibly be extended by any of the Horn clauses.

Naïve Algorithm

To implement this, we first set the value of all propositional letters to *False*. We keep track of two Boolean flags – *consistent* and *change*, each set to *True*. We iterate only as long as both these flags have been set to *True*. During each iteration we set *change* to *False*, and for each Horn clause, we do the following:

Naïve Algorithm

To implement this, we first set the value of all propositional letters to *False*. We keep track of two Boolean flags – *consistent* and *change*, each set to *True*. We iterate only as long as both these flags have been set to *True*. During each iteration we set *change* to *False*, and for each Horn clause, we do the following:

- If all the negative literals have already been set to *True*, and the positive literal is still *False*, we change its value to *True*. We also set *change* to *True*.

Naïve Algorithm

To implement this, we first set the value of all propositional letters to *False*. We keep track of two Boolean flags – *consistent* and *change*, each set to *True*. We iterate only as long as both these flags have been set to *True*. During each iteration we set *change* to *False*, and for each Horn clause, we do the following:

- If all the negative literals have already been set to *True*, and the positive literal is still *False*, we change its value to *True*. We also set *change* to *True*.
- If all the negative literals have already been set to *True* and there is no positive literal in the clause, we set *consistent* to *False*.

Naïve Algorithm

To implement this, we first set the value of all propositional letters to *False*. We keep track of two Boolean flags – *consistent* and *change*, each set to *True*. We iterate only as long as both these flags have been set to *True*. During each iteration we set *change* to *False*, and for each Horn clause, we do the following:

- If all the negative literals have already been set to *True*, and the positive literal is still *False*, we change its value to *True*. We also set *change* to *True*.
- If all the negative literals have already been set to *True* and there is no positive literal in the clause, we set *consistent* to *False*.

At the end, if *consistent* is *True* we return SAT, and otherwise we return UNSAT.

Example 1

$P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$

Iteration	Letters set to *True*	*change*	*consistent*
1	P_1, P_2, P_4, P_3	*True*	*True*
2	P_1, P_2, P_4, P_3	*False*	*False*

Example 2

$$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$$

Iteration	Letters set to *True*	*change*	*consistent*
1	P_1, P_2	*True*	*True*
2	P_1, P_2, P_4	*True*	*True*
3	P_1, P_2, P_4	*False*	*True*

Naïve Algorithm

Why exactly is this algorithm bad?

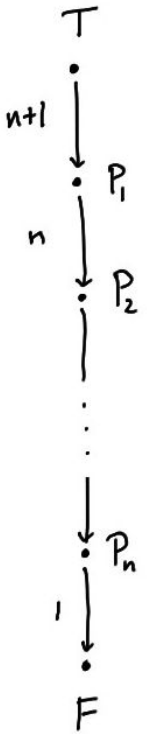
Clearly this algorithm checks the same Horn clause multiple times. However, only one of these checks does something fruitful. In fact, there is no point in checking a Horn clause before we know that all its negative literals have been set to *True* and its positive literal has not yet been. This redundancy can be used to explain why this approach is rather slow.

But how slow is it?

An example in the next slide shows that this algorithm could take up to quadratic time in the worst case.

Naïve Algorithm

$$\neg P_n \wedge (\neg P_{n-1} \vee P_n) \wedge \ldots \wedge (\neg P_2 \vee P_3) \wedge (\neg P_1 \vee P_2) \wedge P_1$$



Linear time algorithm

- We can give a linear time algorithm, by modifying the first algorithm slightly and implementing it better
- Assume formula has K distinct propositional letters - $P_1, ..P_K$
- For each positive literal P - compute the list $clauselist[P]$ of all basic horn propositions in which P occurs as a negative literal
- Construct array *numargs* and *poslitlist* of dimension M where M is the number of basic horn propositions. We will also maintain a queue for implementing the algorithm
- The algorithm will be linear in N, which is the number of total occurrences of literals

Linear time algorithm

- *numargs* array is such that $numargs[n]$ is the number of negative literals in the clause number n that have current truth value **false**
- *poslitlist* array is such that $poslitlist[n]$ is the positive literal occurring in clause n, if any
- We say that a basic horn clause C_n is *ready to be processed* if $numargs[n] = 0$
- Consider the following example : $P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \vee (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$

Example:

$P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \vee (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$

P	*clauselist*[P]
P_1	$[C_3, C_7]$
P_2	$[C_4, C_6, C_7]$
P_3	$[C_5]$
P_4	$[C_5, C_7]$

n	*numargs*[n]
1	0
2	0
3	1
4	1
5	2
6	1
7	3

n	*poslitlist*[n]
1	P_1
2	P_2
3	P_2
4	P_1
5	–
6	P_4
7	P_3

Algorithm contd.

- We also maintain a queue. This queue will contain the basic horn clauses that are ready to be processed.
- These clauses will be such that all the letters occurring negatively in them have been set to true, and thus the remaining positive literal will be forced to be set true now.
- This queue is thus updated when a new positive literal is set to true.
- This queue is initialised to contain the horn clauses which consist of a single positive literal.
- In our running example, the queue will be initialised to $[P_1, P_2]$

Algorithm contd.

- *oldnumclause* holds the size of this queue. $oldnumclause = 2$ in our example.
- Now we perform a for loop running over this queue if the positive literal in them has not yet been set to true.
- Let *clause1* be he current head of the queue that is popped and let $nextpos = poslitlist[\text{clause1}]$ be the positive literal in clause1. In our example, $nextpos = P_1$. Set this to true.
- Now, clause1 is there on the queue because *nextpos* was set to true, or has to be set to True now, which happened because all the negative literals were set to true, i.e P_1 is set to true.
- Consider the list $clauselist[nextpos]$ which contains the list of clauses which contains *nextpos* negatively. $clauselist[P_1] = [C_3, C_7]$

Algorithm contd.

- Now we iterate over the list *clauselist*[*nextpos*]. For each *clause2* in *clauselist*[*nextpos*], we reduce *numargs*[*clause2*] by 1.
- If *numargs*[*clause2*] $= 0$, then it means that all letters occurring negatively are set to True, and thus ready to be *processed*. If it contains a positive literal $n =$ *poslitlist*[*clause2*], then add it to the queue to be processed next time. Else, return UNSAT.
- Renew the queue, reset *oldnumclause* to length of the queue. Continue this till the queue becomes empty, in which case, the formula is SAT.

Application to example

Example : $P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \wedge (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$
$clauselist[P_1]$: $numargs[C_3] = 0$ and $poslitlist[C_3] = P_2$, enter C_3 into newqueue. $numargs[C_7] = 2$
$clauselist[P_2]$: $[C_4, C_6, C_7]$; $numargs[C_4] = 0$, $poslitlist[C_4] = P_1$, add C_4 into newqueue. $numargs[C_6] = 0$, $poslitlist[C_6] = P_4$ add C_6 into newqueue and set P_4 to true. $numargs[C_7] = 1$
newqueue $= [C_3, C_4, C_6]$ and oldnumclause $=$ newnumclause $= 3$.
Now we iterate over newqueue.

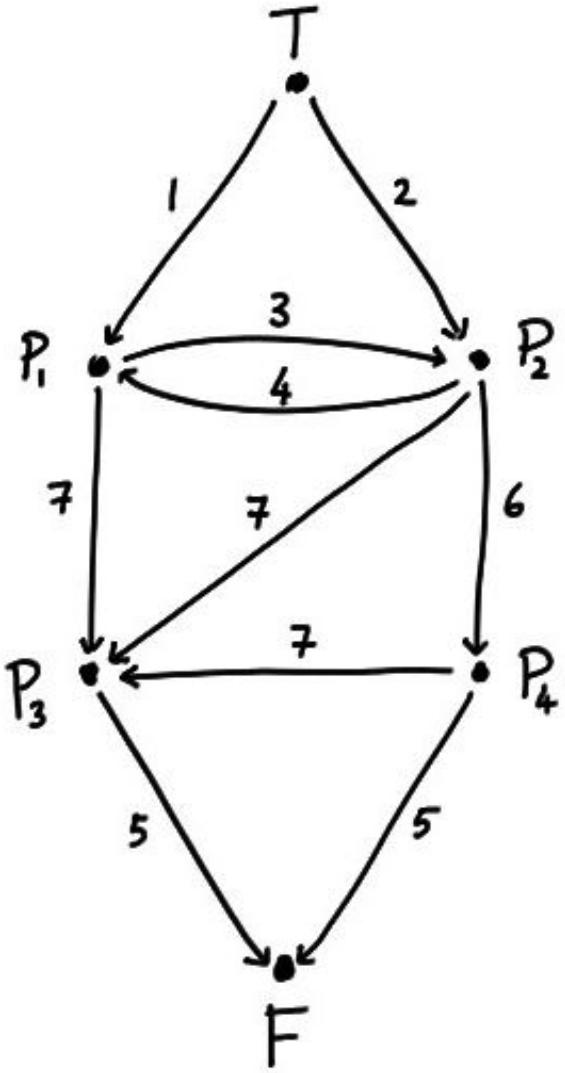
Application continued

Example : $P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \vee (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$
$poslitlist[C_3] = P_2$ which is already set to true.
$poslitlist[C_4] = P_1$ which is already set to true.
$poslitlist[C_6] = P_4$, set this to true and consider $clauselist[P_4] = [C_5, C_7]$, $numargs[C_5] = 1$ and $numargs[C_7] = 0$ and $poslitlist[C_7] = P_3$ and hence put C_7 into newqueue.
Now, newqueue $= [C_7]$ and $poslitlist[C_7] = P_3$, set this to true, then $clauselist[P_3] = [C_5]$ and $numargs[C_5] = 0$, but $poslitlist[C_7] = []$, hence UNSAT.

Complexity of the algorithm

- Assume that there are K propositional letters P_1,P_2,...P_K
- *numargs* and *poslitlist* can be initialized in linear time, ie $O(N)$ where N is the number of occurrences of literals.
- Every horn clause is *processed* only once in the entire algorithm, ie when the positive literal in it is set to true.
- For every such clause, when it is processed, for all the clauses in *clauselist*[*nextpos*], we decrement *numargs*[clause] by 1, which corresponds to the "*deletion*" of negative occurrences of *nextpos*.
- Once the above step is done, we **never** revisit any literal, positive or negative, which contains *nextpos*.
- Hence, every time *processing* is done, we consider disjoint occurrences of literals, and number of such occurences is at most N, hence running time is $O(N)$.

Graphical analysis



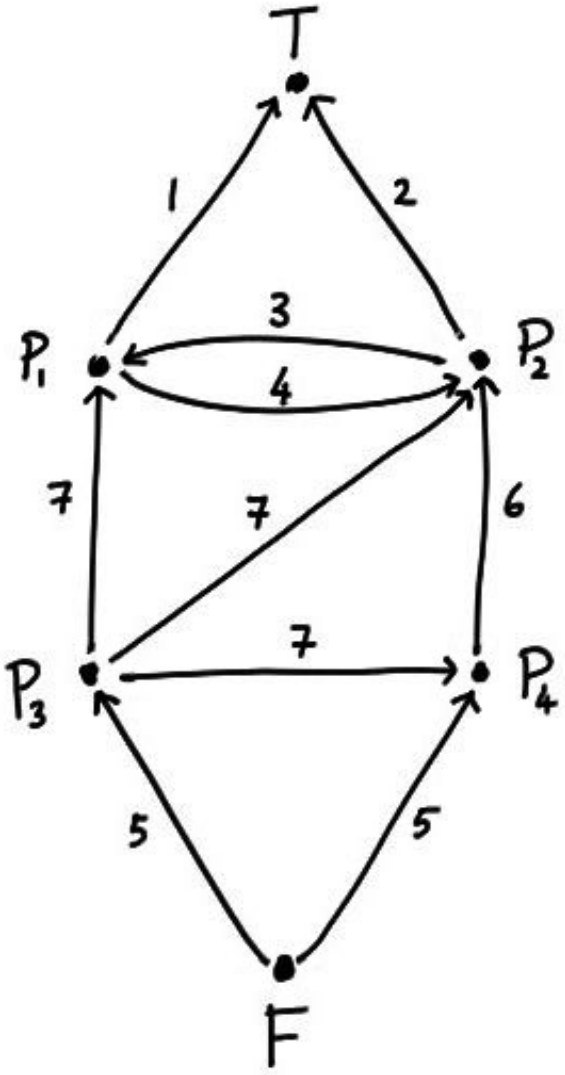
Motivation for another linear time algorithm

- We thus basically try to find a pebbling from true to false in a breadth-first fashion in the previous algorithm.
- For the next algorithm we try to find a pebbling from true to false, by evaluating what's necessary to have the desired pebbling. Thus we now move from False to True in the graph, contrary to the previous algorithm.
- This approach is called the "call-by-need" graph algorithm.

Idea for the next linear time algorithm

- We recursively find whether all P_i are forced to be set True, which will force Q to be True. For this purpose, we consider a new graph, G'_A which is obtained from the earlier graph by reversing its edges.
- Thus to find out whether Q should be True, it is sufficient to visit all the nodes reachable from Q.
- However this time, we cannot avoid visiting some nodes more than once, but however, we would like to minimize the number of visits to nodes.
- We consider the same running example as before : $P_1 \wedge P_2 \wedge (\neg P_1 \vee P_2) \vee (\neg P_2 \vee P_1) \wedge (\neg P_3 \vee \neg P_4) \wedge (\neg P_2 \vee P_4) \wedge (\neg P_1 \vee \neg P_2 \vee \neg P_4 \vee P_3)$

Graph for this algorithm in our example



Algorithm 3

- We "mark" edges and allow a visit to a node provided *either there is some unmarked incoming edge to it, or one of its immediate successors has some unmarked outgoing edge*.
- Each edge of the graph will have a field *visited*,and each node of the graph will have a field *marked* which is a counter holding the number of non-visited outgoing edges from the node and is decremented every time such an edge is visited.
- We also use the lists *clauselist* and *numargs* used earlier. Again, as before, once a propositional letter is set to True, the entries in *numargs* corresponding to the clauses in *clauselist* are decremented by 1.
- Once a propositional letter is set to True, in order to ensure we don't visit it anymore, we set a field *computed* to True.

Algorithm 3

- The graph will be implemented as an array of linked lists, each entry in the array being a record corresponding to a node of the graph, and each linked list being the list representing all edges having that node as the source.
- For each node, we create a list *successors* consisting of records (one for each label in the set of all outgoing edges with this node as the source)
- Each record contains a label number i and a pointer to the list of target nodes of all edges with source labelled i.

Graph initialization

- Every horn clause consisting of a single positive literal, the *val* field of the corresponding node is True and False for all other nodes.
- The *visited* field of every edge is set of False and the *computed* of every node P is set to the sum of the negative literals in all basic clauses containing P.

How the algorithm proceeds

- We start from the node false. For each node, we traverse through the list *successors* and check if there is any node whose *computed* field is not yet set to True. Pick one such node.
- If the computed field of all nodes in *successors* are set to True, this corresponds to *numargs* being 0. Set *computed* to True for this node. Also update *numargs*
- Consider the edge joining the two nodes and make the field *visited* to True corresponding to that edge. Also reduce the field *marked* by 1. Next, for the chosen node, you perform the recursion.
- If one of the nodes in the successors list is the node True, then for the present node, set the propositional letter to True and set *computed* to True.Also update *numargs*

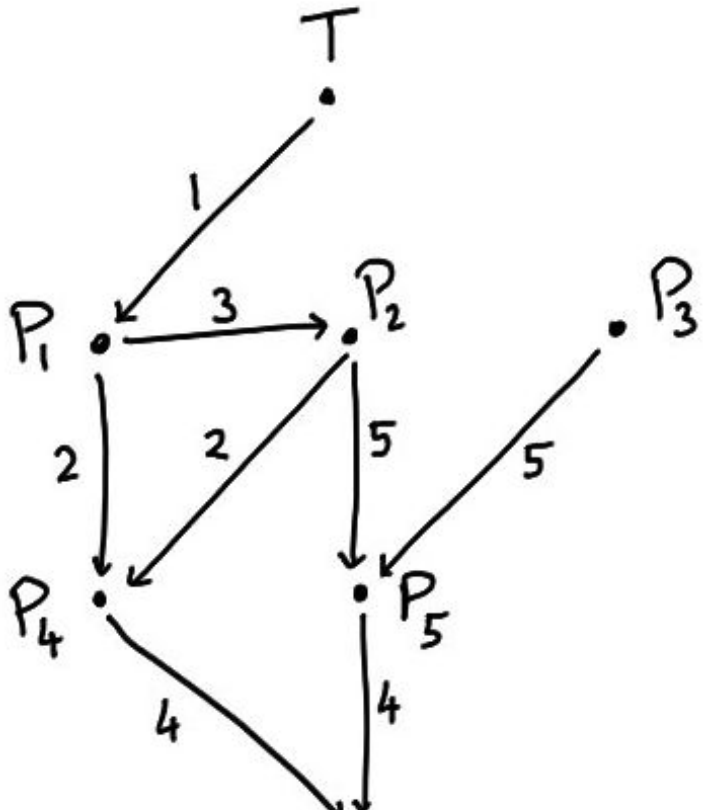
Complexity of the algorithm

- Assume for now that the graph can be built in linear time.
- In the worst case scenario, for every basic horn formula, the positive literal in it and the targets of the edges, which correspond to the negative literals in the horn formula are visited once. This corresponds to N visits plus the starting node "False"
- The truth value of every node is computed exactly once as we use the field *computed* and set it to True once its computed.
- Updating the *numargs* array every time a propositional letter is set True, as earlier will be $O(N)$ since we will be disjointly deleting negative occurrence of that letter in the formula.
- Hence, the traversal is $O(N)$, combined with building graph taking $O(N)$, the whole algorithm is again linear.

Another example

$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$

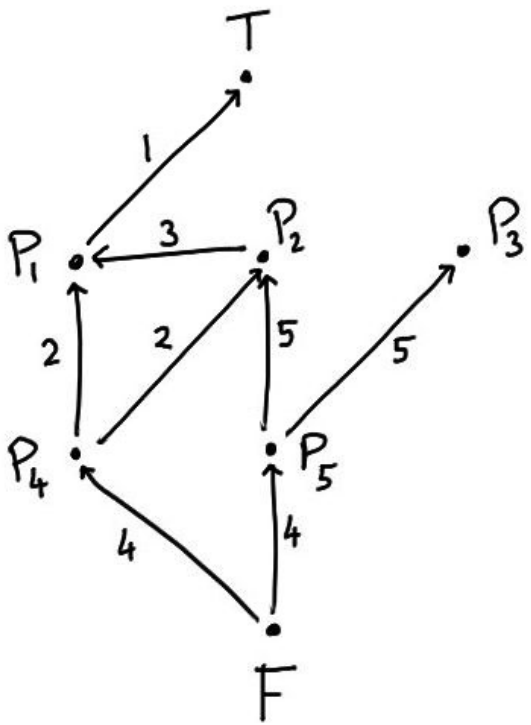
This is the graph for algorithm 2



Another example

$P_1 \wedge (\neg P_1 \vee \neg P_2 \vee P_4) \wedge (\neg P_1 \vee P_2) \wedge (\neg P_4 \vee \neg P_5) \vee (\neg P_2 \vee \neg P_3 \vee P_5)$

This is the graph for algorithm 3



Algorithm Buildgraph

- We represent the graph using adjacency lists, i.e., to each vertex A we associate the list of all vertices to which edges go out to from A. Since the vertices are just labelled $P_1, \ldots, P_n$, we can just use an array (of size n+2) of lists containing integers, with T corresponding to index 0, P_i corresponding to index i and F corresponding to index $n+1$.

Algorithm Buildgraph

- We represent the graph using adjacency lists, i.e., to each vertex A we associate the list of all vertices to which edges go out to from A. Since the vertices are just labelled $P_1, \ldots, P_n$, we can just use an array (of size n+2) of lists containing integers, with T corresponding to index 0, P_i corresponding to index i and F corresponding to index $n+1$.

- Note that this is where we use the crucial assumption that the propositional letters are precisely $P_1, P_2, \ldots, P_n$, and not some arbitrary letters. If the letters were indeed arbitrary, then setting up a mapping from the letters to a range of integers would itself take longer than linear time.

Algorithm Buildgraph

For each Horn clause we encounter while travelling from left to right, if the clause is of the form:

- P_i, then append 0 to the list at index i.
- $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k})$, then append each i_j to the list at index $n+1$.
- $(\neg P_{i_1} \vee \ldots \vee \neg P_{i_k} \vee P_t)$, then append each i_j to the list at index t.

Algorithm Buildgraph

- Note that figuring out which type a Horn Clause belongs to requires only to pass through it once, and is thus linearly proportional to its length.
- Each subcase looks at every literal at most once, and thus this is also linearly proportional to the length of the Horn Clause.
- In conclusion, the buildgraph algorithm gives an adjacency list representation of the graph G'_A in time proportional to the length of the Horn formula, i.e., the number of occurrences of literals in the formula.