Solving 2-MAXSAT in Polynomial Time: a Proof of $P = NP$

Yangjun Chen
Department of Applied Computer Science
University of Winnipeg, Canada
www.acs.uwinnipeg.ca/ychen2

Outline

- Motivation
  - Satisfiability problem
  - 2-MAXSAT problem
- Algorithm for finding a truth assignment satisfying most clauses
  - Transformation of *CNF* to *DNF*
  - *p*-graphs and *p**-graphs
  - Trie-like graphs
  - Reachable subsets through spans (*RS*) and Algorithm
- Time complexity analysis
- Conclusion

Motivation

- **Satisfiability problem**

  $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$ ← Conjunctive normal form (*CNF*)

  where each C_i is a clause of the form:

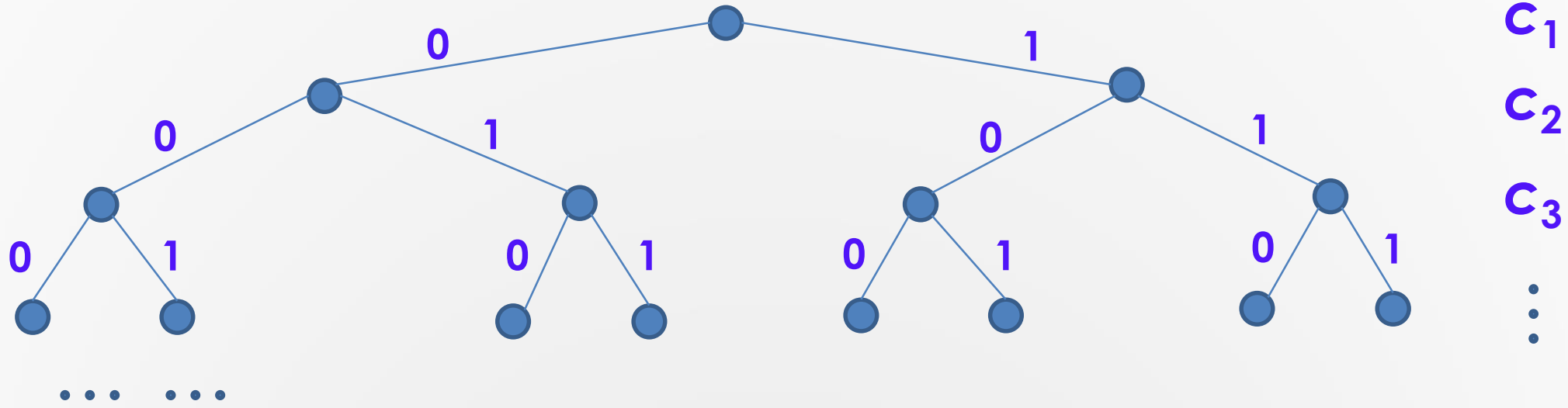
  $C_i = (c_{i1} \vee \ \ldots \vee c_{im})$.

  Each c_{ij} is a literal (a Boolean variable or a negative variable)

- **Problem: find a truth assignment for c_{ij}'s such that each C_i evaluates to *true*. The problem is NP-complete.**

Motivation

- We need exponential time to solve the problem.



Time complexity: $O(2^n)$

NP – non-deterministic polynomial time

P – polynomial time

Motivation

- **3-sat: each clause contains at most three literals.**

  3-sat is still *NP*-complete.

  Karp, Richard M. (1972). "Reducibility Among Combinatorial Problems" (PDF). In R. E. Miller; J. W. Thatcher; J.D. Bohlinger (eds.). *Complexity of Computer Computations*. New York: Plenum. pp. 85–103.

- **2-sat: each clause contains at most two literals.**

  2-sat can be solved in linear time.

  https://cp-algorithms.com/2SAT.html.

Motivation

- **Maximum satisfiability problem (MAXSAT)**
  - An optimization version of satisfiability that seeks a truth assignment to maximize the number of satisfied clauses
  - The MAXSAT is NP-hard.

- **2-MAXSAT problem (MAXSAT)**
  - A simplified version of MAXSAT with each C_i containing at most two literals.
  - The 2-MAXSAT problem is *NP*-complete.

2-MAXSAT

- **2-MAXSAT is *NP*-complete**
  - **The 3-sat problem can be reduced to 2-MAXSAT in polynomial time.**

M. R. Garey, D. S. Johnson, and L. Stockmeyer, Some simplified NP-complete graph problems, Theoret. Comput. Sci., (1976), pp. 237-267.

Main Idea of Our Algorithm

- **Main idea**
  - Transform the 2-MAXSAT problem to a problem which can be solved in polynomial time.
  - Consider a logic formula in *CNF*:

$$C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

with each C_i ($= c_{i1} \vee c_{i2}$) containing at most two literals.

  - Define a logic expression in disjunctive normal form (*DNF*):

$$D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \ldots \vee D_{n1} \vee D_{n2}$$

Main Idea of Our Algorithm

- **Main idea**
  - Transform the 2-MAXSAT problem to a problem which can be solved in polynomial time.
  - Consider a logic formula in *CNF*:

$$C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

with each C_i ($= c_{i1} \vee c_{i2}$) containing at most two literals.

Main Idea

- Define a disjunctive normal form (*DNF*):

$$D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \ \ldots \vee D_{n1} \vee D_{n2}$$

where

$$D_{i1} = c_{i1} \wedge x_i,$$

$$D_{i2} = c_{i2} \wedge \neg x_i, \text{ and}$$

x_i (i = 1, …, n) is a new (Boolean) variable.

$x_1 \quad x_2 \quad \ldots \quad x_n$

$$C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$$

$$C_i = c_{i1} \vee c_{i2} \ (i = 1, \ldots, n)$$

- We can prove that C has at lease n^* satisfied clauses if and only if D has at least n^* satisfied conjunctions.

Main Idea

- **Proposition: C has at lease *n** satisfied clauses if and only if *D* has at least *n** satisfied conjunctions.**
- **Y. Chen,** Solving 2-MAXSAT in Polynomial Time: a Proof of *P* = *NP*, arXiv:2304.12517v26 [cs.CC] 15 Sept 2026. (arXiv paper)
- **New problem: find a truth assignment that maximizes the number of satisfied conjunctions in**

$$D = D_1 \vee D_2 \vee \ \ldots \vee D_n$$

where $D_i = c_{i1} \wedge c_{i2} \wedge \ldots \wedge c_{im}$.

If we can solve the above problem in polynomial time, then we can also solve the 2-MAXSAT problem in polynomial time.

Main Idea

Example. Consider the following formula:

$$C = C_1 \wedge C_2 \wedge C_3$$
$$= (c_1 \vee c_2) \wedge (c_2 \vee \neg c_3) \wedge (c_3 \vee \neg c_1)$$

$$D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \vee D_{31} \vee D_{32}$$
$$= D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5 \vee D_6$$
$$= (c_1 \wedge c_4) \vee (c_2 \wedge \neg c_4) \vee$$
$$(c_2 \wedge c_5) \vee (\neg c_3 \wedge \neg c_5) \vee$$
$$(c_3 \wedge c_6) \vee (\neg c_1 \wedge \neg c_6)$$

We can find an assignment $\sigma = \{c_1 = 1, c_2 = 1, c_3 = 1\}$ for C. Under σ, a maximum of 3 clauses: C_1, C_2, and C_3 are satisfied.

We can also find an assignment $\sigma' = \{c_1 = 1, c_2 = 1, c_3 = 1, c_4 = 1, c_5 = 1, c_6 = 1\}$ for D. Under σ', a maximum of 3 conjunctions: D_1, D_2, and D_3 are satisfied.

Main Idea

- Introducing a notation $(c_j, *) = c_j \vee \neg c_j = \mathit{true}$ to represent a missing variable
- Rewrite $D = D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5 \vee D_6$

$$D = (c_1 \wedge (c_2, *) \wedge (c_3, *) \wedge c_4 \wedge (c_5, *) \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge \neg c_4 \wedge (c_5, *) \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge (c_4, *) \wedge c_5 \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge (c_2, *) \wedge \neg c_3 \wedge (c_4, *) \wedge \neg c_5 \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge (c_2, *) \wedge c_3 \wedge (c_4, *) \wedge (c_5, *) \wedge c_6) \vee$$
$$(\neg c_1 \wedge (c_2, *) \wedge (c_3, *) \wedge (c_4, *) \wedge (c_5, *) \wedge \neg c_6)$$

Variable Sequences

- Variable sequences, in which negative literals are removed

$D = (c_1 \wedge (c_2, *) \wedge (c_3, *) \wedge c_4 \wedge (c_5, *) \wedge (c_6, *)) \vee$

$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge (c_5, *) \wedge (c_6, *)) \vee$

$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge (c_4, *) \wedge c_5 \wedge (c_6, *)) \vee$

$((c_1, *) \wedge (c_2, *) \wedge (c_4, *) \wedge (c_6, *)) \vee$

$((c_1, *) \wedge (c_2, *) \wedge c_3 \wedge (c_4, *) \wedge (c_5, *) \wedge c_6) \vee$

$(\wedge(c_2, *) \wedge (c_3, *) \wedge (c_4, *) \wedge (c_5, *))$

conjunctions	Variable sequence
$D1$	$c_1.(c_2, *).(c_3, *).c_4.(c_5, *).(c_6, *)$
$D2$	$(c_1*).\ c_2.(c_3, *).(c_5, *).(c_6, *)$
$D3$	$(c_1, *).\ c_2.(c_3, *).(c_4, *).c_5.(c_6, *)$
$D4$	$(c_1, *).(c_2, *).(c_4, *).(c_6, *)$
$D5$	$(c_1, *).(c_2, *).c_3.(c_4, *).(c_5, *).c_6$
$D6$	$(c_2, *).(c_3, *).(c_4, *).(c_5, *)$

Why to remove negative literals?

Variable Sequences

Why to remove negative literals?

- **What we want is to construct a graph for each D_j such that each root-to-leaf path represents a truth assignment under which D_j evaluates to *truth*.**
- **By this truth assignment, any variable on this path has value *true*. Otherwise, its value is *false*.**

Sorted Variable Sequences

Sorted variable sequence:

- Compute the appearance frequencies of variables in variable sequences

variable	c_1	c_2	c_3	c_4	c_5	c_6
frequency	5/6	6/6	5/6	5/6	5/6	5/6

- Global ordering of variables such that the most frequent variable appears first:

$$c_2 \longrightarrow c_3 \longrightarrow c_1 \longrightarrow c_4 \longrightarrow c_5 \longrightarrow c_6$$

Sorted Attribute Sequences

Sorted attribute sequence:

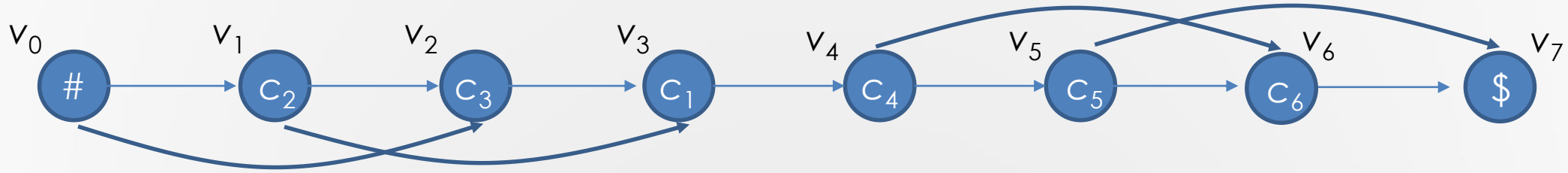
conjunctions	Variable sequence	Sorted variable sequence
*D*1	$c_1.(c_2, *).(c_3, *).c_4.(c_5, *).(c_6, *)$	$\#(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$
*D*2	$(c_1*).\ c_2.(c_3, *).(c_5, *).(c_6, *)$	$\#c_2.(c_3, *).(c_1,*).\ (c_5, *).(c_6, *).\$$
*D*3	$(c_1, *).\ c_2.(c_3, *).(c_4, *).c_5.(c_6, *)$	$\#c_2.(c_3, *).(c_1, *).\ (c_4, *).c_5.(c_6, *).\$$
*D*4	$(c_1, *).(c_2, *).(c_4, *).(c_6, *)$	$\#(c_2, *).(c_1, *).(c_4, *).(c_6, *).\$$
*D*5	$(c_1, *).(c_2, *).c_3.(c_4, *).(c_5, *).c_6$	$\#(c_2, *).c_3.(c_1, *).(c_4, *).(c_5, *).c_6.\$$
*D*6	$(c_2, *).(c_3, *).(c_4, *).(c_5, *)$	$\#(c_2, *).(c_3, *).(c_4, *).(c_5, *).\$$

Represent each D_j as a graph

p-Graphs

Definition Let $r = d_0 d_1 \ldots d_k d_{k+1}$ be a variable sequence representing a conjunction as described above (with $d_0 = \#$ and $d_{k+1} = \$$). A *p*-graph over r is a directed graph, in which there is a node for each d_j ($j = 0, \ldots, k + 1$); and an edge for (d_j, d_{j+1}) for each $j \in \{0, \ldots, k\}$. In addition, there may be an edge from d_j to d_{j+2} for each $j \in \{0, \ldots, k - 1\}$ if d_{j+1} is a pair of the form $(a, *)$, where a is a variable. Each off-path edge is called a span.

$D_1 = \#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$

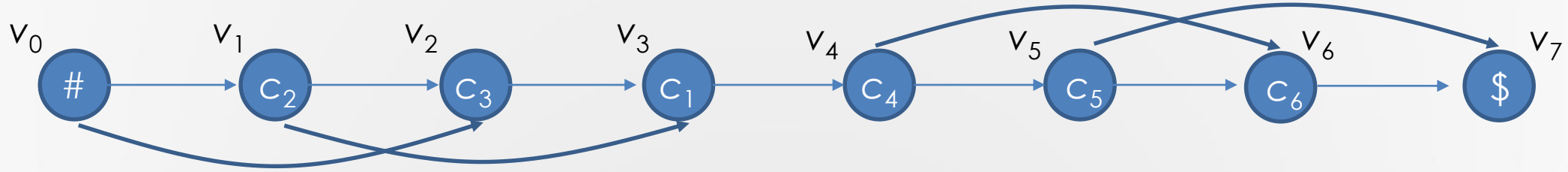


Each span is represented by a sub-path covered by it: $(c_2, *) = \langle v_0, v_1, v_2 \rangle$.

Overlapped spans

- Let $s_1 = \langle v_1, ..., v_k \rangle$ and $s_2 = \langle u_1, ..., u_l \rangle$ be two spans attached on a same path. We say, s_1 and s_2 are overlapped, if $u_1 = v_j$ for some $v_j \in \{v_1, ..., v_{k-1}\}$, or if $v_1 = u_{j'}$ for some $u_{j'} \in \{u_1, ..., u_{l-1}\}$.
- For example, in the above figure, $\langle v_0, v_1, v_2 \rangle$ and $\langle v_1, v_2, v_3 \rangle$ are overlapped. $\langle v_4, v_5, v_6 \rangle$ and $\langle v_5, v_6, v_7 \rangle$ are also overlapped. Here, we notice that the overlapped spans imply the consecutive '*'s, just like $\langle v_0, v_1, v_2 \rangle$ and $\langle v_1, v_2, v_3 \rangle$, which correspond to two consecutive '*'s: $(c_2, *)$ and $(c_3, *)$.

$D_1 = \#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$



Transitive closure of overlapped spans

- The overlapped spans exhibit some kind of *transitivity*. That is, if s_1 and s_2 are two overlapped spans, the $s_1 \cup s_2$ must be a new, but bigger span, representing some more consecutive '*'s. Applying this operation to all the spans over a *p*-path, we will get a *transitive closure* of overlapped spans.

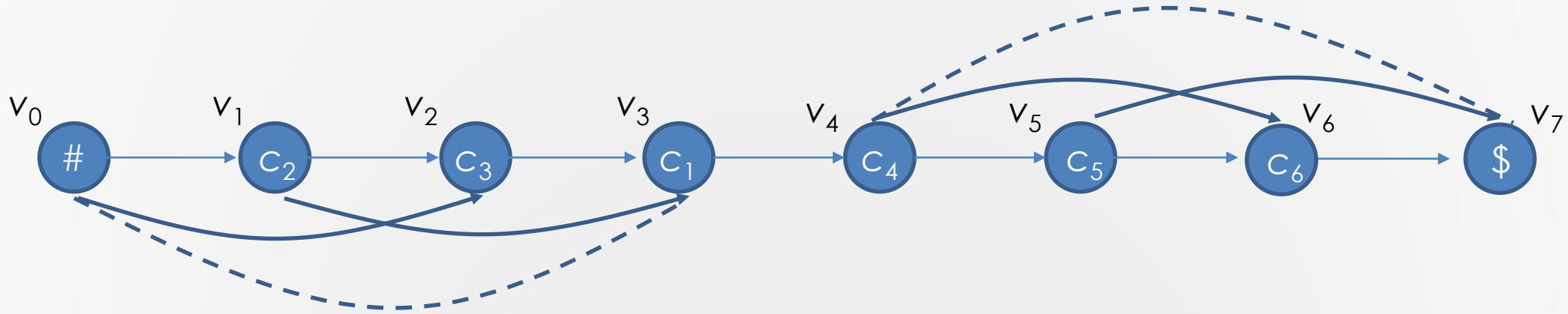
Definition Let S be the set of all spans over the main path p for a certain conjunction. The transitive closure of S, denoted as S*, is another set of spans S* = $\{s_1, s_2, ..., s_l\}$ for some *l*, which contains the whole S and is with each s_i satisfying one of the following two conditions:

1. $s_i \in S$, or
2. There exist *j*, *k* ($\neq i$) such that s_j and s_k are overlapped and $s_i = s_j \cup s_k$.

P*-Graph

Definition Let *P* be a *p*-graph. Let *p* be its main path and S be the set of all spans over *p*. Denote by S* the 'transitive closure' of S. Then, the *p**-graph with respect to *P* is the union of *p* and S*, denoted as *P** = *p* ∪ S*.

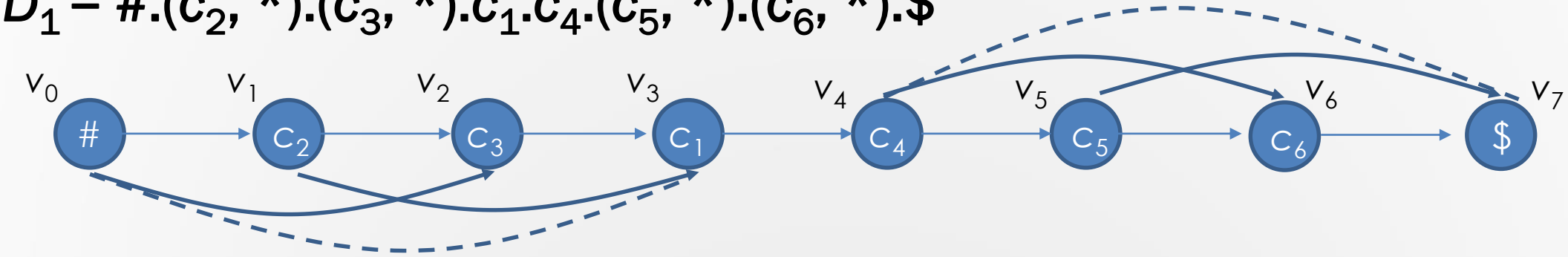
$D_1 = \#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$



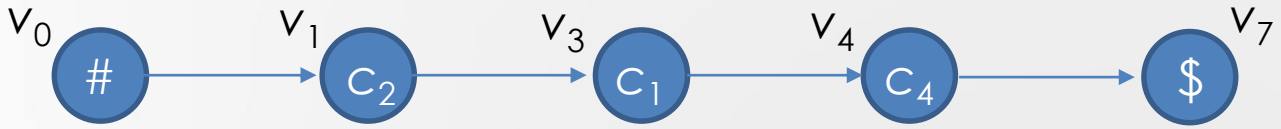
P*-Graph

Lemma Let P^* be a p^*-graph for a variable sequence D_i in D. Then, each path from # to $ in P^* represents a truth assignment, satisfying D_i.

$D_1 = \#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$



a root-to-leaf path:



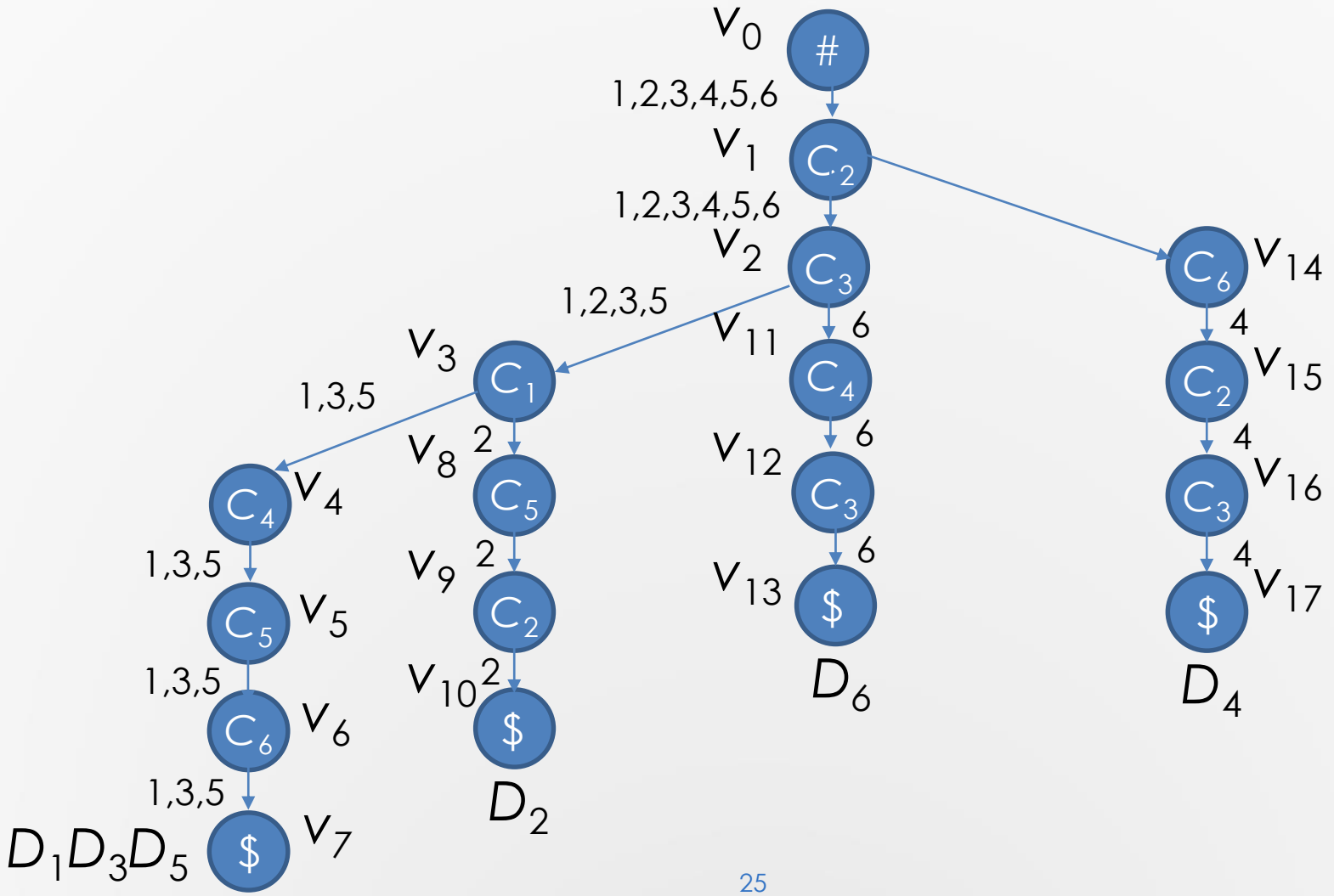
Corresponding truth assignment:

$c_1 = 1, c_2 = 1, c_3 = 0, c_4 = 1, c_5 = 0, c_6 = 0$

Trie over Main Paths

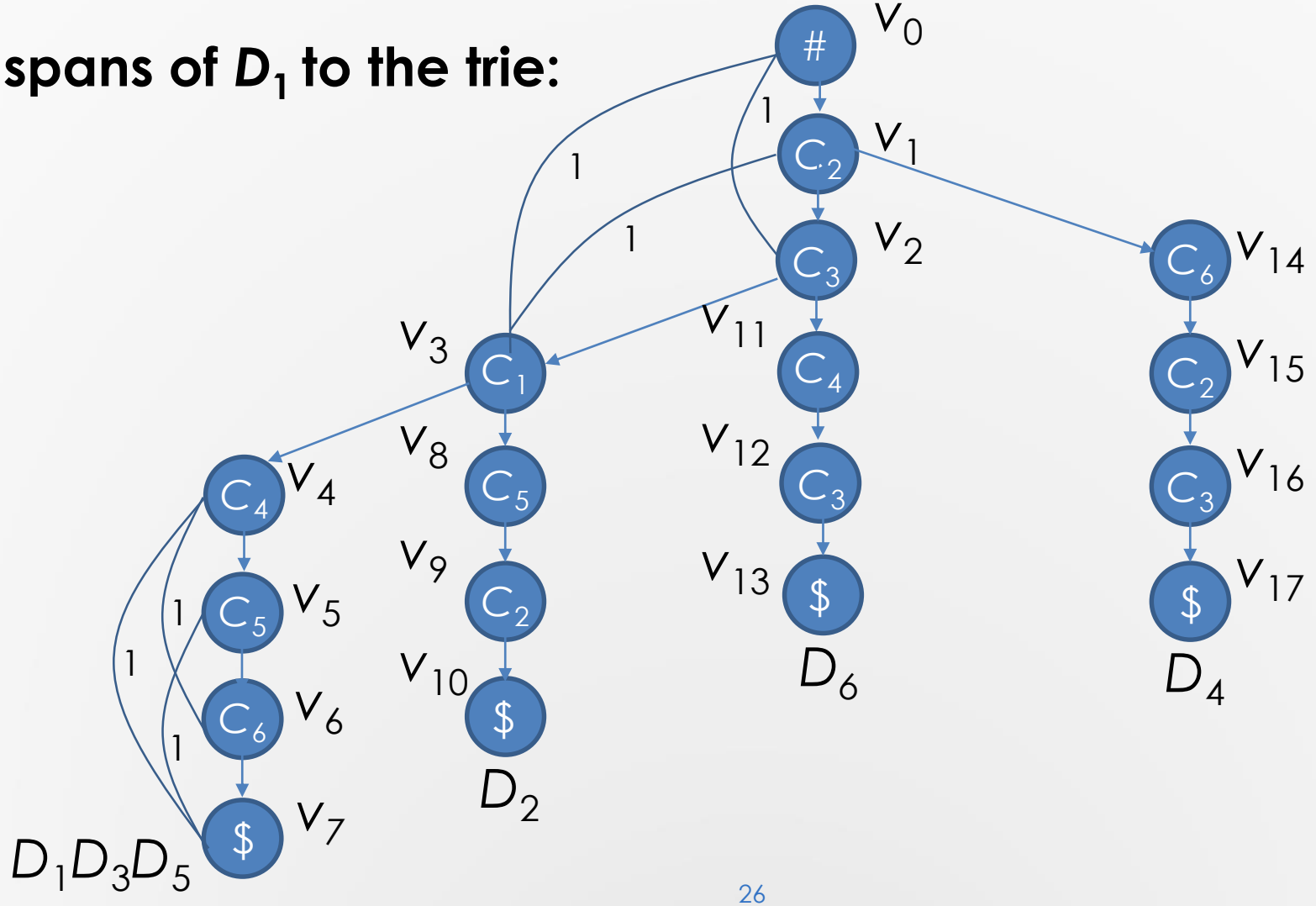
- Let $p_1, p_2, \ldots, p_n$ be the main paths in $P_1^*, P_2^*, \ldots, P_n^*$. A trie over $R = \{p_1, p_2, \ldots, p_n\}$, denoted as $T = trie(R)$, is defined as follows.
- If $|R| = 0$, $trie(R)$ is, of course, empty. For $|R| = 1$, $trie(R)$ is a single node. If $|R| > 1$, R is split into m (possibly empty) subsets $R_1, R_2, \ldots, R_m$ so that each R_j ($j = 1, \ldots, m$) contains all those sequences with the same first attribute name. The tries: $trie(R_1), trie(R_2), \ldots, trie(R_m)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth attribute (along the global ordering of attributes). They are then connected from their respective roots to a single node to create $trie(R)$.

Trie over Main Paths

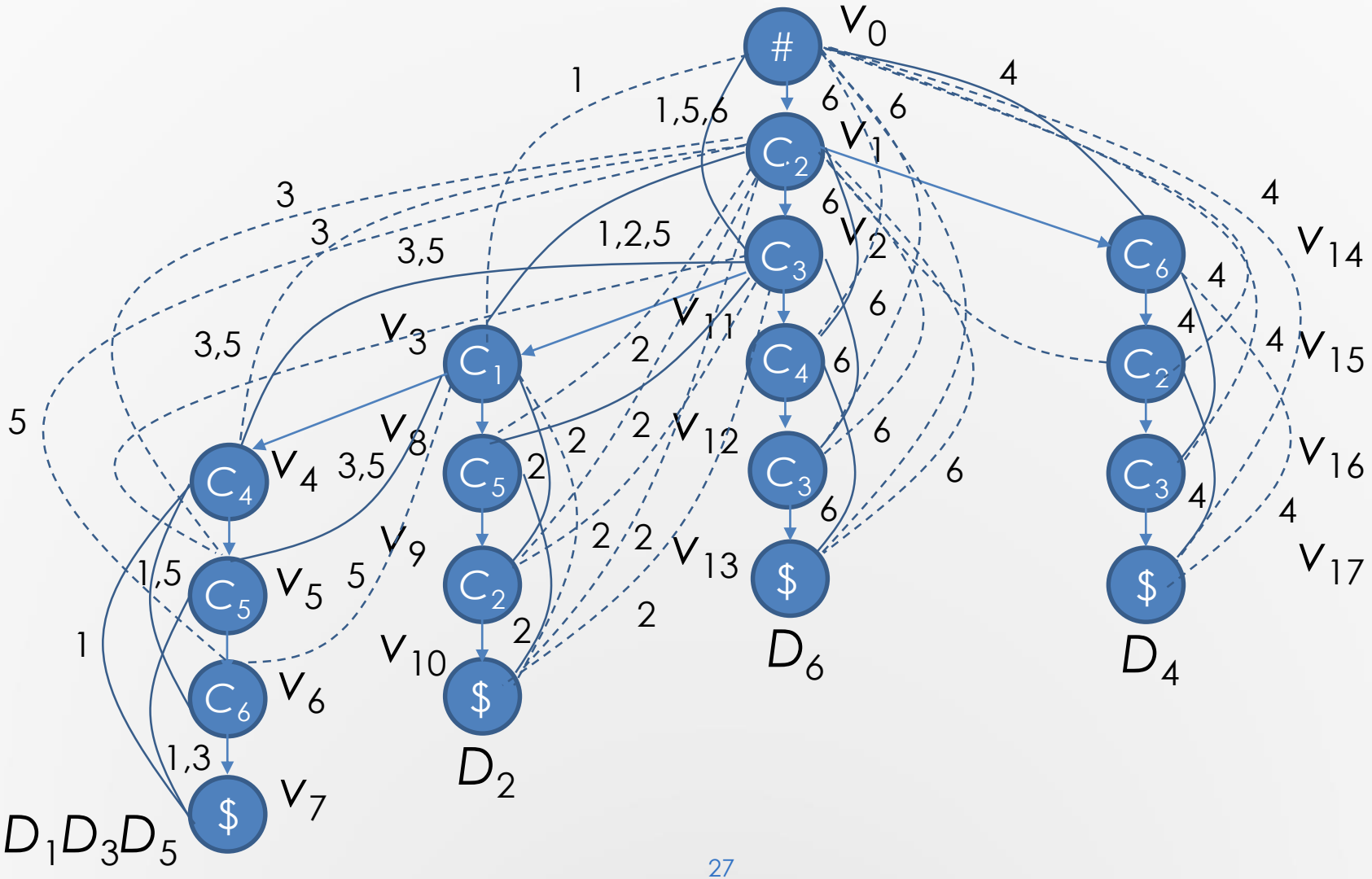


Trie over Main Paths

add the spans of D_1 to the trie:



Trie-like Graph

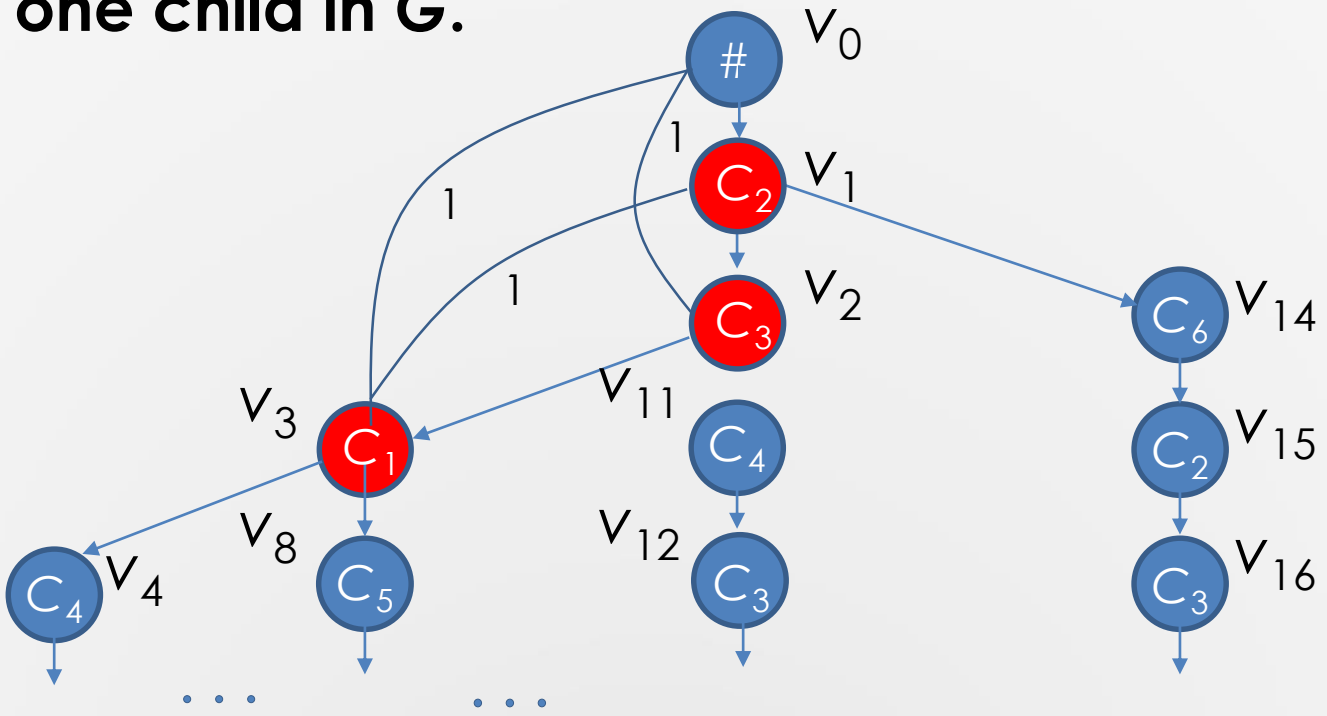


Branching Nodes

Branching nodes: nodes having more than one child in *T*.

Example: v_1, v_2, v_3.

But v_0, is not a branching node since it has only one child in *T* even though it has more than one child in *G*.



Reachable Subsets through Spans

Definition Let *v* be a branching node. Let *u* be a node on the tree path (in *T*) from root to *v* (not including *v* itself). A reachable subset of *u* through spans are all those nodes with a same label *c* in different subgraphs in *G*[*v*] (subgraph of *G* rooted at *v*) and reachable from *u* through a span, denoted as $RS_s^{v,u}[c]$, where *s* is a set containing all the labels associated with the corresponding spans.

Reachable Subsets through Spans

More about *RS*:

- **Any RS with |RS| = 1 will not be considered since it will not lead to new answers.**
- **Definition of the *RS* for anchor node which is a branching node itself is a little bit different. Specifically, each of its RSs is defined to be a subset of nodes reachable from u via a span or via a tree edge.**

Reachable Subsets through Spans

Example

$RS^{v_3,v_2}_{\{2,3,5\}}[c_5] = \{v_5, v_8\}$

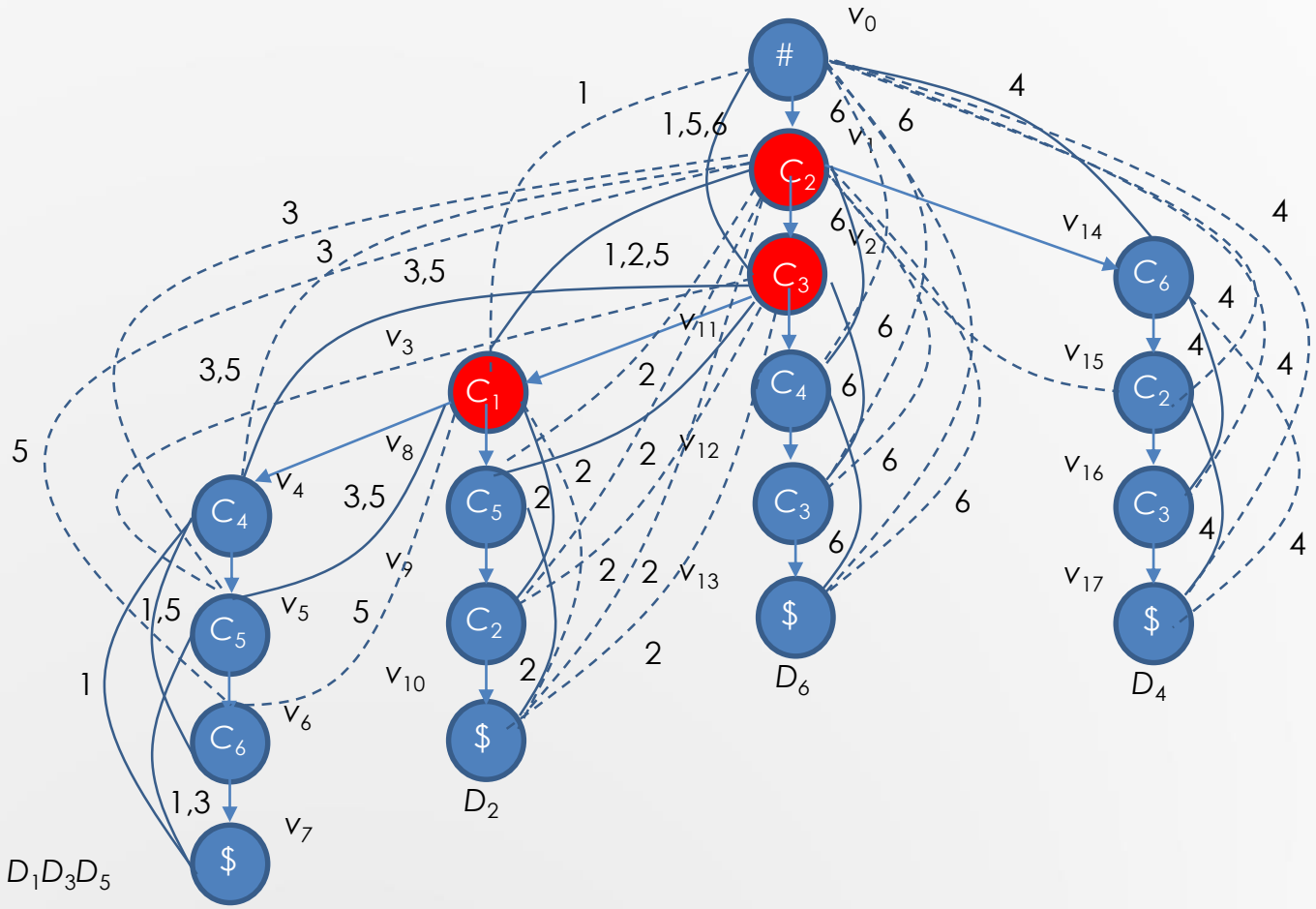
Due to spans: $v_2 \xrightarrow{3,5} v_5$

$v_2 \xrightarrow{2} v_8$

$RS^{v_3,v_2}_{\{2,5\}}[c_6] = \{v_6, v_9\}$

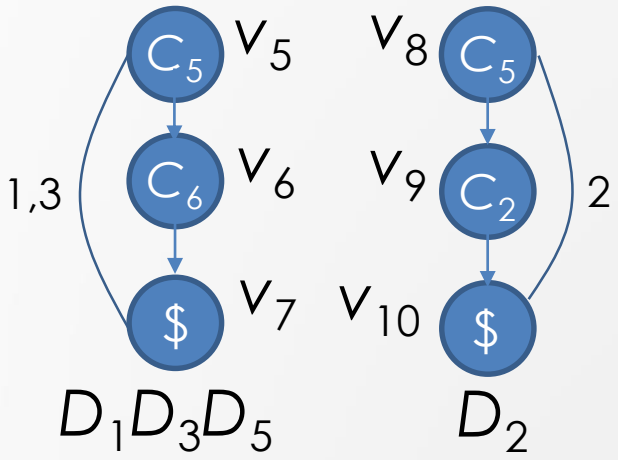
Due to spans: $v_2 \xrightarrow{5} v_6$

$v_2 \xrightarrow{2} v_9$

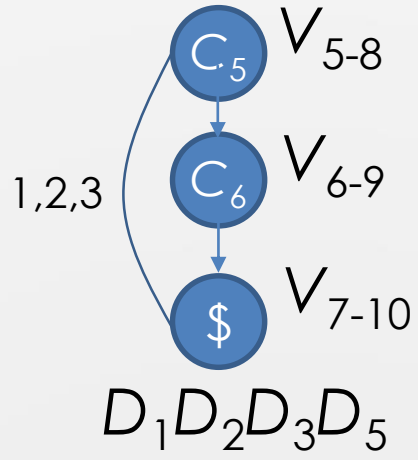


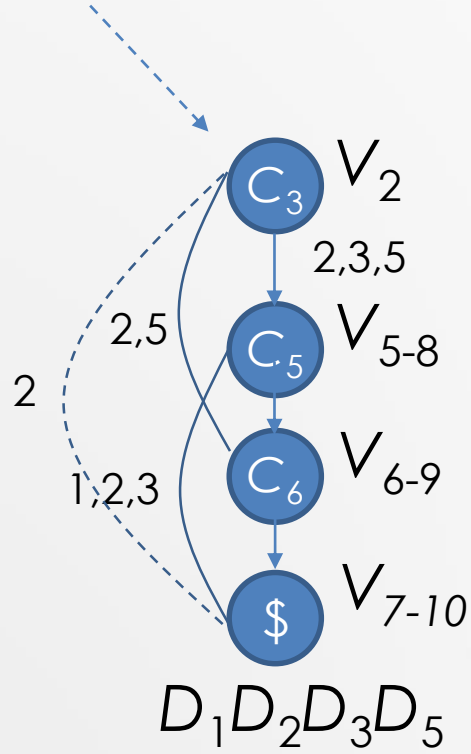
Trie-like Subgraphs

$RS^{v_3,v_2}_{\{2,3,5\}}[c_5] = \{v_5, v_8\}$



Add the corresponding anchor node as the virtual *root* of the trie-like subgraph





Main Steps

To find all the maximum subsets of conjunctions, which can be satisfied by a same truth assignment, the following steps will be performed.

- **Search G bottom-up (in postorder).**
- **For each branching node, a set of RS for different anchor nodes will be created.**
- **For each RS, a trie-like subgraph will be generated. (This can be done by a recursive call of the algorithm).**
- **Recursively explore the trie-like subgraph to generate some more subsets of satisfiable conjunctions.**
- **For each satisfiable subsets, trace the corresponding path to find its truth assignment.**

Algorithm

Algorithm 1: 2-MAXSAT(C)

Input : a logic formula *C* in *CNF* with each clause in C containing at most two literals.
Output: a largest subset of clauses satisfying a certain truth assignment.

1. transform *C* to another formula *D* in *DNF*;
2. let $D = D_1 \vee \ldots \vee D_n$;
3. for $i = 1$ to n do
4. construct a p^*-graph $\boldsymbol{P}_i^*$ i for D_i ;
5. construct a trie-like graph *G* over $\boldsymbol{P}_1^*$, ..., $\boldsymbol{P}_n^*$;
6. *R* := *SEARCH*(*G*);
7. return the result calculated in terms of *R*;

Algorithm

Algorithm 2: *SEARCH*(*G*)

Input : a trie-like subgraphs *G*.
Output: a largest subset of conjunctions satisfying a certain truth assignment.

1. if *G* is a single *p**-graph then
2. R' := subset associated with the leaf node;
3. R := *merge*(R, R');
4. return R;
5. for each leaf node *v* in *G* do
6. let R' be the subset associated with v;
7. R := *merge*(R, R');
8. let $v_1, v_2, \ldots, v_k$ be all branching nodes in postorder;
9. for i = 1 to k do
10. let $RS_1, \ldots, RS_l$ be all the *RS*s constructed for v_i ;
11. for j = 1 to l do
12. construct a trie-like graph *D* in terms of *RS*j ;
13. let u_j be the corresponding anchor node;
14. D' := $\{u_j\} \cup D$;
15. R' := *SEARCH*(D');
16. R := *merge*(R, R');
17. return R;

Time Complexity Analysis

The total running time of the algorithm consists of three parts.

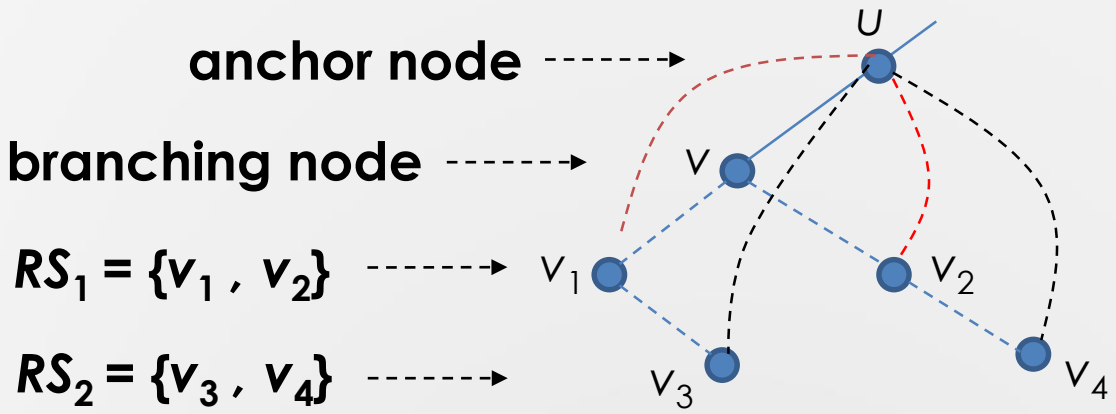
- The first part τ_1 is the time for computing the frequencies of variable appearances in variable sequences. Since in this process each variable in a sequence is accessed only once, $\tau_1 = O(nm)$.

Time Complexity Analysis

- The second part τ_2 is the time for constructing a trie-like graph *G* for *Q*. This part of time can be further partitioned into three portions.
  - τ_{21}: Time for sorting variable sequences in *D*. It is obviously bounded by O(*nm* log *m*).
  - τ_{22}: Time for constructing *p**-graphs for each of conjunctions. Since for each conjunction a transitive closure over its spans should be first created and needs $O(m^2)$ time, this part of cost is bounded by $O(nm^2)$.
  - τ_{23}: Time for merging all *p**-graphs to form a trie-like graph *G*, which is also bounded by $O(nm^2)$.

Time Complexity Analysis

- The third part τ_3 is the time for searching G to find a maximum subset of conjunctions satisfied by a certain truth assignment.
  - In the whole working process, we may have at most O(nm) branching nodes.
  - For each branching node, we may have at most m anchor nodes (along a path in G). Corresponding to each anchor node, we may have O(m) recursive calls with each corresponding to a different label (a variable). So, for each branching node, we may have at most O(m^2) recursive calls.



Time Complexity Analysis

- The third part τ_3 is the time for searching G to find a maximum subset of conjunctions satisfied by a certain truth assignment.
  - In the whole working process, we may have at most $O(nm)$ branching nodes.
  - For each branching node, we may have at most m anchor nodes (along a path in G). Corresponding to each anchor node, we may have $O(m)$ recursive calls with each corresponding to a different label (a variable). So, for each branching node, we may have at most $O(m^2)$ recursive calls.
  - By each recursive call, we need $O(nm^2)$ time to construct a trie-like subgraph.
  - Therefore, time complexity is bounded by $O(n^2m^5)$.
- This time complexity can be reduced to $O(n^2m^4)$ by removing some redundancy.

Conclusion

- **Main contribution**
  - **Transformation of *CNF* to *DNF***
  - ***p*-graphs, used to represent each conjunction in a *DNF***
  - ***p**-graphs, used to represent all those truth assignment of a conjunction D_i, under which D_i evaluates to *true***
  - **Trie-like graph, constructed by mering all *p**-graphs**
  - **Search a trie-like graph to get the answer in polynomial time**
  - **Since the 2-MAXSAT is *NP*-complete, our algorithm is in fact a proof of *P* = *NP*.**

Thank you!