The 2-MAXSAT Problem Can Be Solved in Polynomial Time

Yangjun Chen
Department of Applied Computer Science
University of Winnipeg

Outline

- Motivation
  - Satisfiability problem
  - 2-MAXSAT problem
- Algorithm for finding most satisfied clauses
  - *p*-graphs and *p**-graphs
  - Trie-like graphs
  - Layered representation of trie-like graphs
- 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:

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

  Problem: find a truth assignment for c_{ij}'s such that each C_i evaluates to *true*.

  The problem is NP-complete.

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 is still NP-hard.

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 ($= c_{i1} \vee c_{i2}$) contains at most two literals.

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

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

Algorithm

- 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.

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

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

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

Algorithm

- Proposition: C has at lease n^* satisfied clauses if and only if D has at least n^* satisfied conjunctions.
- 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.

Algorithm

- **Consider**

$$D = D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5 \vee D_6$$

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

We can find an assignment $\sigma = \{c_1 = 1, c_2 = 0, c_3 = 0, c_4 = 1, c_5 = 1, c_6 = 1\}$ for D. Under σ, three conjunctions: D_1, D_3, and D_5 are satisfied. In fact, they are a maximum set of satisfied conjunctions.

Algorithm

- **Rewrite *D* as:**

$D_1 = c_1 \wedge \neg c_3 \wedge c_5$

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

where $(c_j, *) = c_j \vee \neg c_j =$ ***true***.

Variable Sequences

- **Variable sequences, in which negative literals are removed**

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

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	3/6	3/6	5/6	6/6	5/6

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

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

Sorted Attribute Sequences

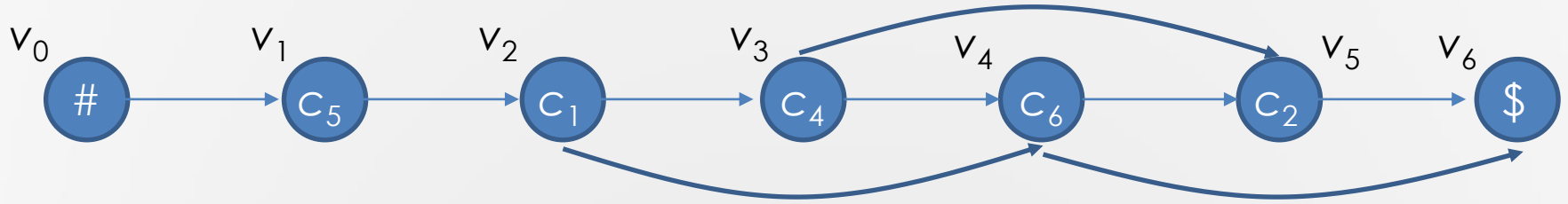
Sorted attribute sequence:

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

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.

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



Each span is represented by a sub-path covered by it: $(c_4, *) = \langle v_2, v_3, v_4 \rangle$

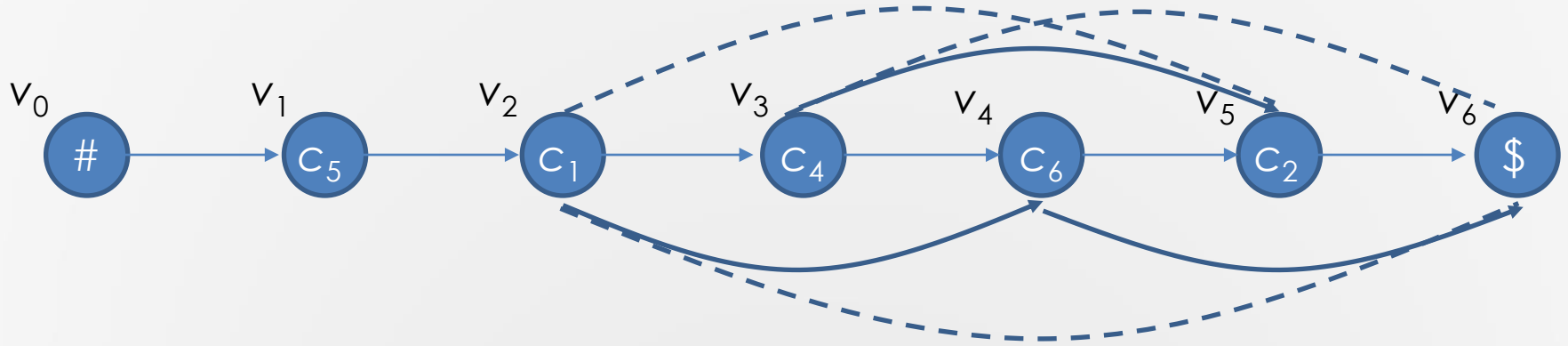
P*-Graphs

- 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, $\langle v_2, v_3, v_4 \rangle$ and $\langle v_3, v_4, v_5 \rangle$ are overlapped. $\langle v_3, v_4, v_5 \rangle$ and $\langle v_4, v_5, v_6 \rangle$ are also overlapped. But $\langle v_2, v_3, v_4 \rangle$ and $\langle v_4, v_5, v_6 \rangle$ not. Here, we notice that the overlapped spans imply the consecutive '*'s, just like $\langle v_2, v_3, v_4 \rangle$ and $\langle v_3, v_4, v_5 \rangle$, which correspond to two consecutive '*'s: $(c_4, *)$ and $(c_6, *)$.
- 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.

*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 $P^* = p \cup S^*$.

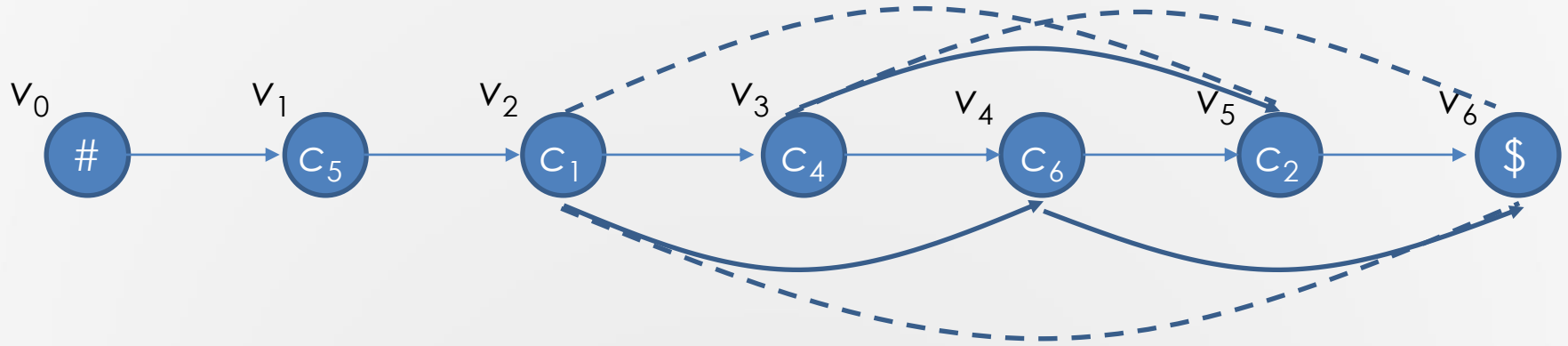
$D_1 = \#.c_5.\ c_1.(c_4, *).(c_6, *).(c_2, *).\$$



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.

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

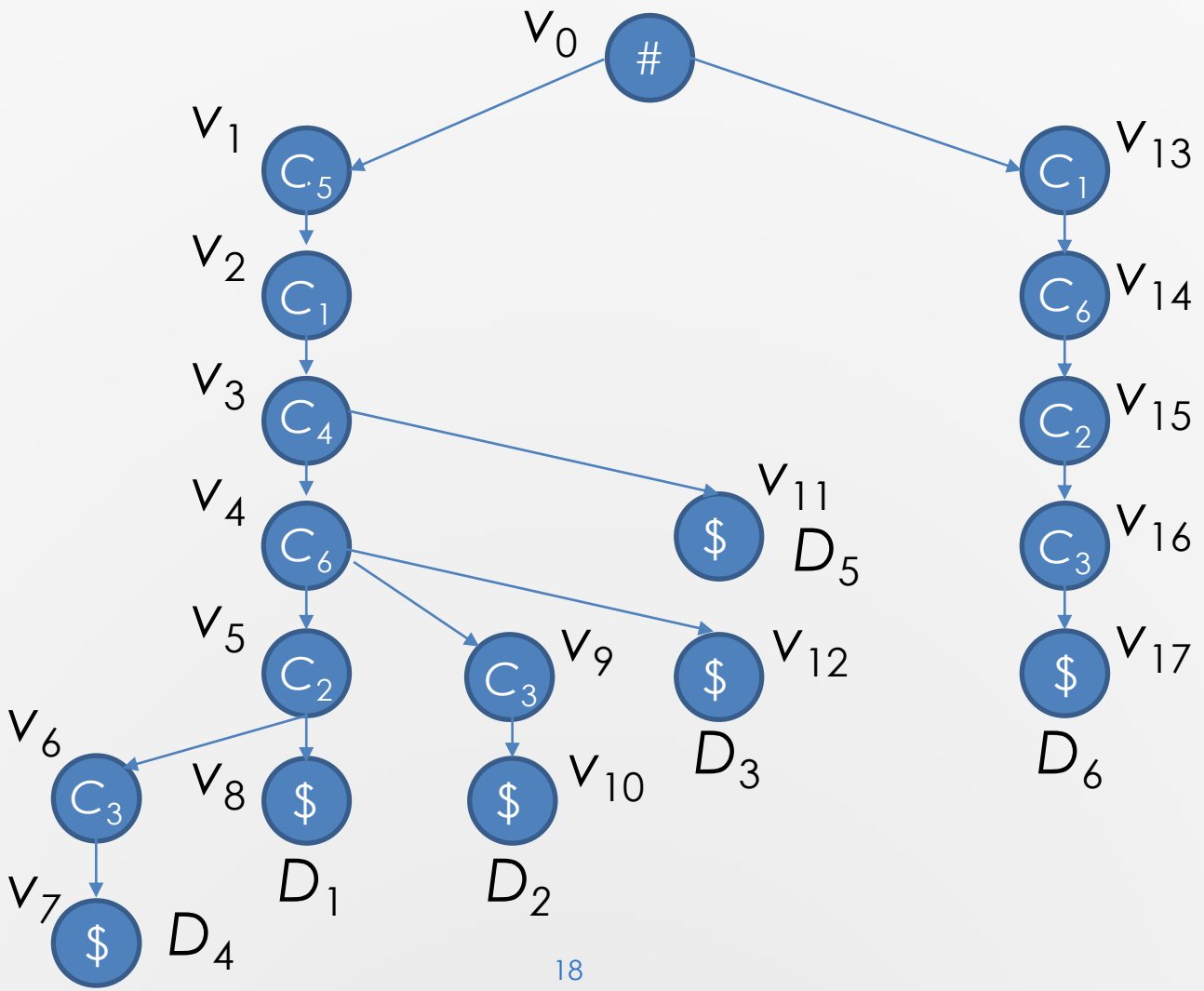


Trie over Main Paths

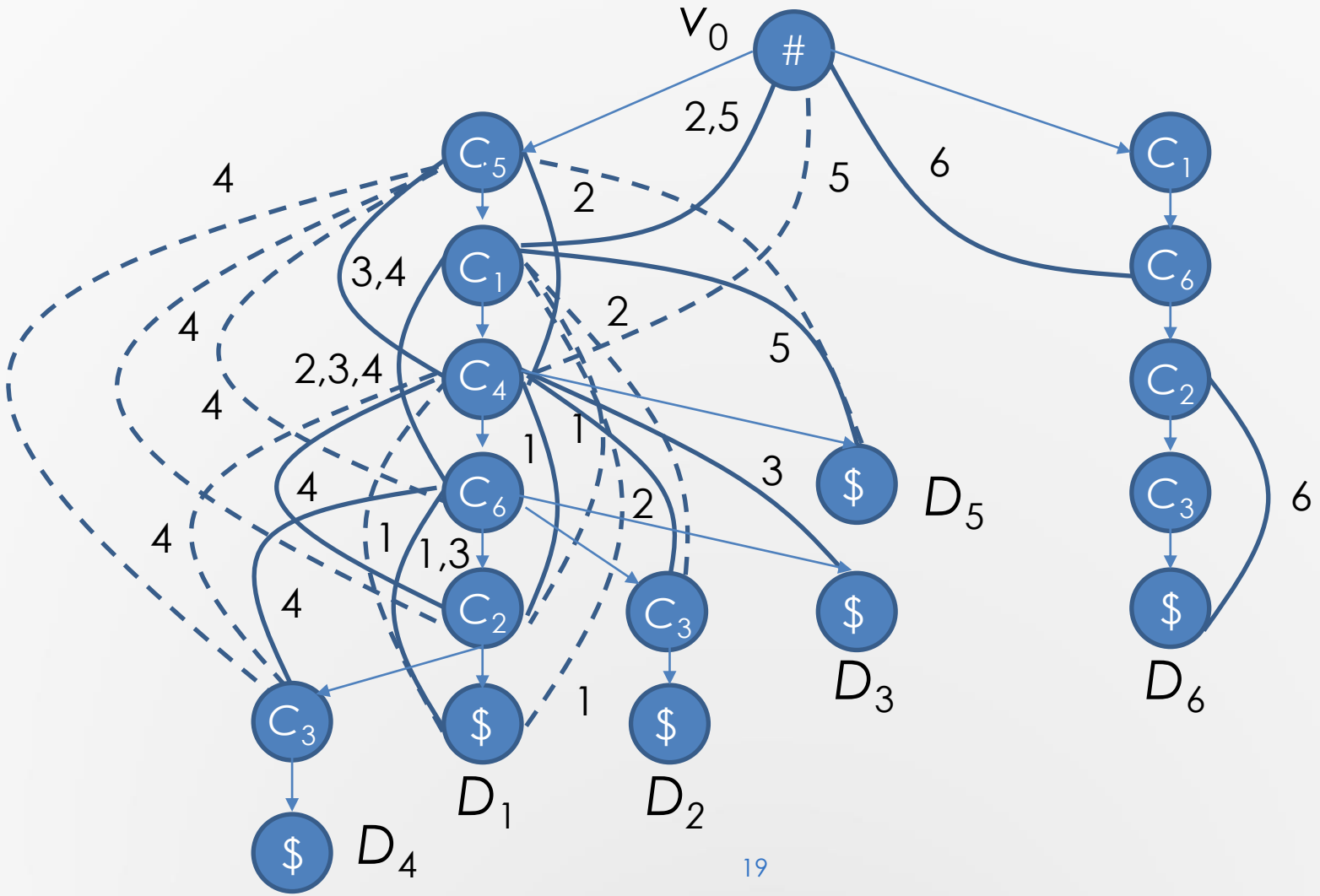
- Let $p_1, p_2, \ldots, p_n$ be the main paths in $P_1^*, P_2^*, \ldots, P_n^*$. A trie over $R = \{p_1, p_2, \ldots, p_n\}$, denoted as $T = trie(R)$, is defined as follows.

- 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 $trie(R)$.

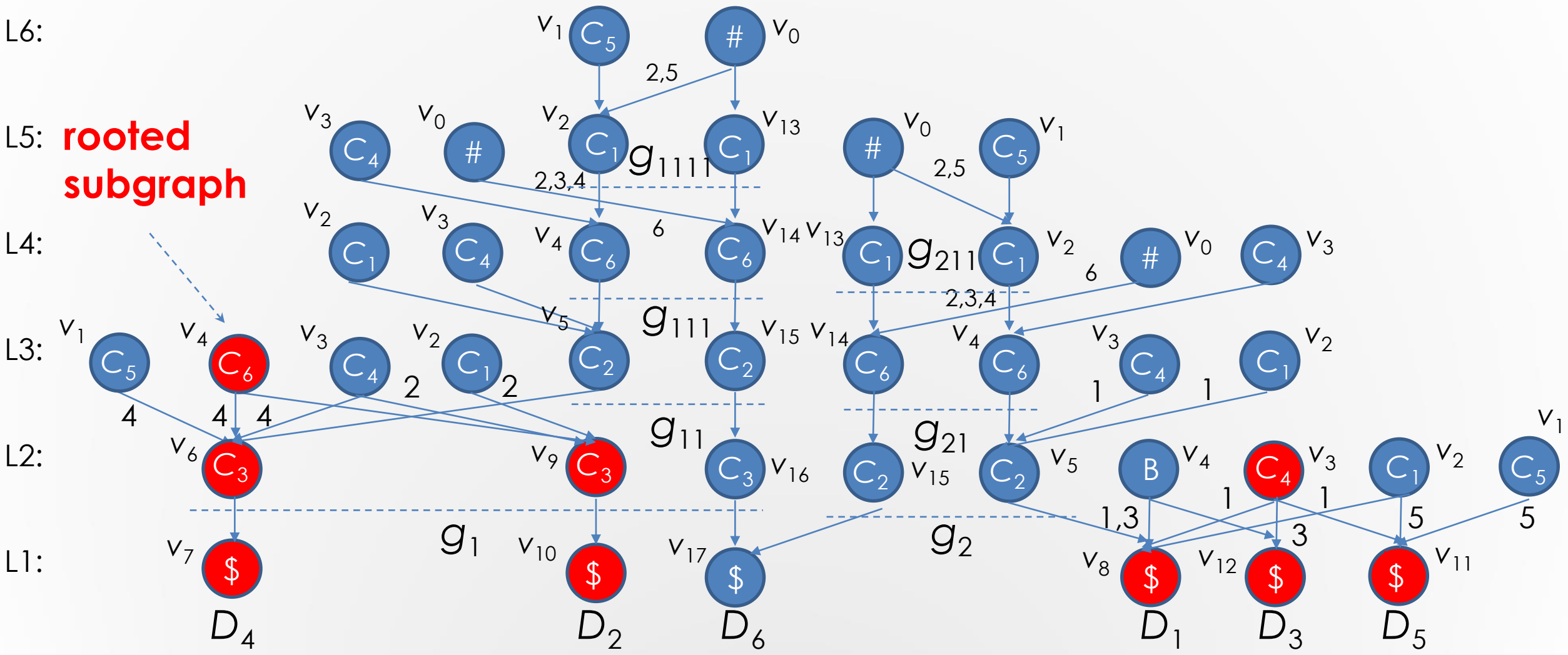
Trie over Main Paths



Trie-like Graphs



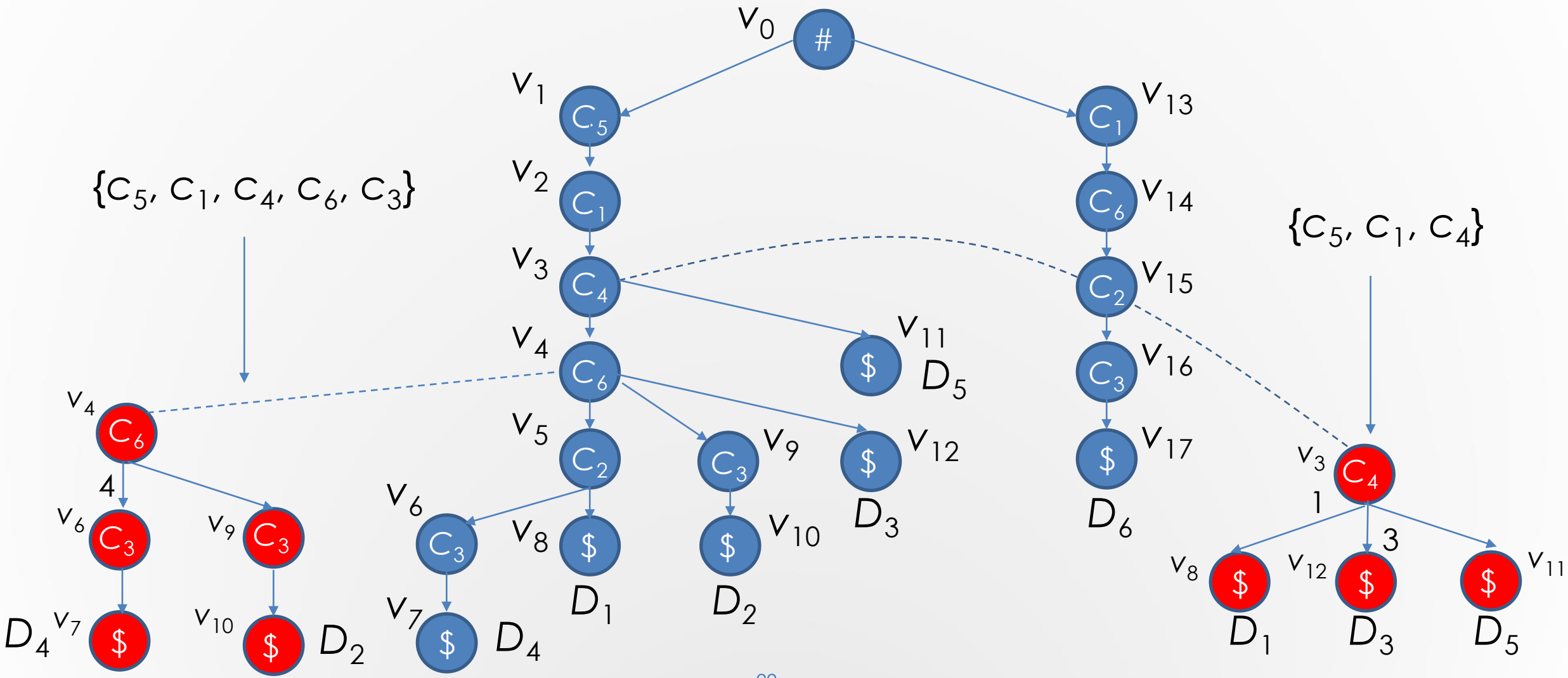
Layered Graphs



How to Find Answers

- Each node without parents in a layered graph is called a root.
- All the nodes reachable from a root make up a rooted subgraph.
- Each rooted subgraph corresponds to a subset of queries satisfied by a certain package.

How to Find Answers



Algorithm

Algorithm 1: *SEARCH*(*G*)

Input: a trie-like graph *G*
Output: a most popular package

1. *G′* :={all leaf nodes of *G*}; *g* := {all leaf nodes of *G*};
2. push(*S*, *g*);
3. **while** *S* is not empty **do**
4. *g'* := pop(*S*);
5. find the parents of each node in *g'*; add them to *G′*;
6. divide all such parent nodes into several groups: g_1, g_2, ..., g_k such that all the nodes in a group with the same label;
7. **for** each $j \in \{1, ..., k\}$ **do**
8. **if** $|g_j| > 1$ **then**
9. push(*S*, g_j);
10. return *findPackage*(*G′*);

Algorithm

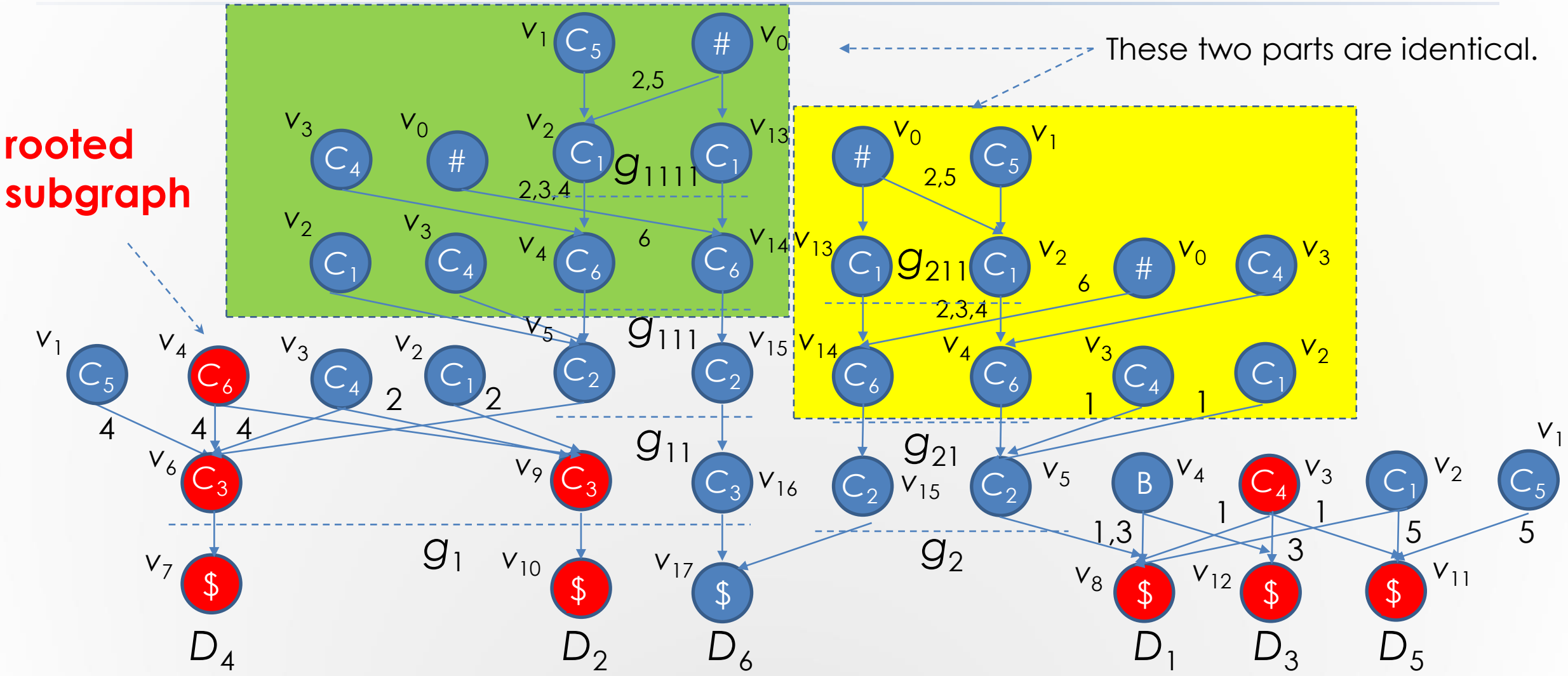
Algorithm 2: *findSubset*(*G'*)

Input: a layered graph *G'*

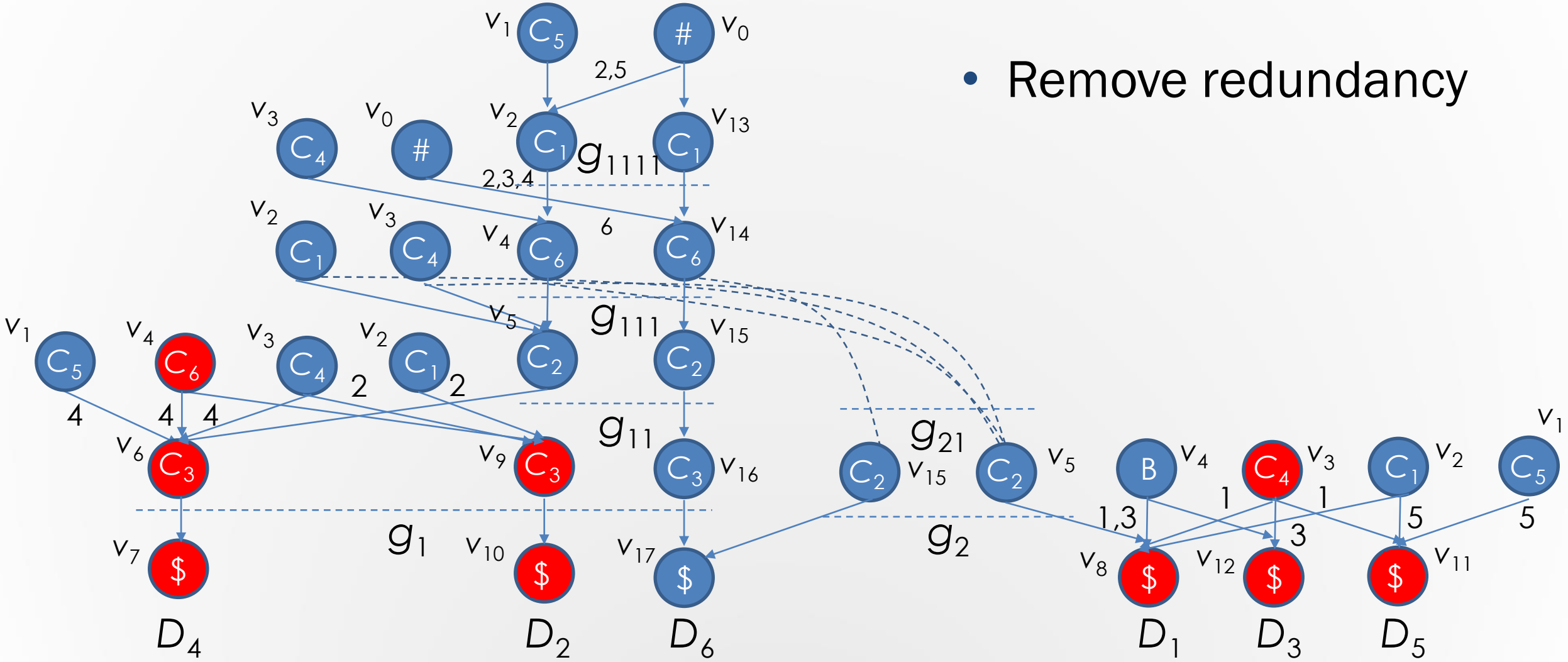
Output: a largest subset of conjunctions satisfying a certain truth assignment

1. (*u*, s, *f*) := (*null*, 0, *Φ*); (* find a package for a maximum subset of queries. *)
2. **for** each rooted subgraph G_v **do**
3. determine the subset *Q'* of satisfied conjunctions in G_v;
4. **if** |*Q'*| > s **then**
5. *u* := *v*; s := |*Q'*|; *f* := *Q'*;
6. return (*u*, s, *f*);

Further Improvement



Further Improvements



- Remove redundancy

Time Complexity Analysis

The total running time of the algorithm consists of four 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, $\tau_1 = O(nm)$.
- 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 generate its layered representation. Since in this process, each edge in G is accessed once and the number of all edges is bounded by $O(nm^2)$, we have $\tau_3 = O(nm^2)$.
- The fourth part τ_4 is the time for checking all the rooted subgraphs. Since each level in the layered representation G' of G has at most $O(nm)$ nodes, we have $O(nm^2)$ nodes in G' in total. In addition, the number of edges in each rooted subgraph G' is bounded by $O(nm)$, the cost of this part of computation is bounded by $O(n^2m^3)$.
- Thus, the total running time of our algorithm is

$$\sum_{i=1}^{4} \tau_i = O(nm) + (O(nm\log m) + O(nm^2) + O(n^2m^3) = O(n^2m^3).$$

Conclusion

- Main contribution
  - *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*-hard, our algorithm is in fact a proof of *P* = *NP*.

Thank you!