CSC2512
Advanced Propositional Reasoning

CSC2512: #SAT (Model Counting)

- A SAT formula can have many satisfying models. #**SAT** is the problem of counting the number of satisfying models.
  - Subsumes **SAT** --- the formula is unsatisfiable iff the number of models is zero.
  - **#SAT** is a **#P** complete problem.
- If each model is assigned a different weight, then #SAT becomes the problem of summing the weights of the satisfying models.
- With a weighted model counter one can answer any probabilistic query over a discrete probability space.

CSC2512: #SAT

DPLL simplest version.

```
DPLL($\phi$)
if $\phi$ has no clauses, output "satisfiable" and HALT
else-if $\phi$ does not contain an empty clause then
    choose a variable $x$ that appears in $\phi$
    Call DPLL($\phi|_{x=0}$)
    Call DPLL($\phi|_{x=1}$)
return
```

CSC2512: #SAT

#DPLL simplest version.
Returns #models/2^n where n = # variables in the formula.

```
#DPLL($\phi$)
if $\phi$ has no clauses, return 1
else-if $\phi$ has an empty clause, return 0
else
    Choose a variable $x$ that appears in $\phi$
    return #DPLL($\phi|_{x=0}$) $\times \frac{1}{2}$ + #DPLL($\phi|_{x=1}$) $\times \frac{1}{2}$
```

CSC2512: #SAT

E.g. (x,y), (x,-y) = 2/4 models

#DPLL({(x,y), (x,-y)})

➔ #DPLL({(x,y), (x,-y)}|x=1)

= #DPLL({})

🡐 1

➔ #DPLL({(x,y), (x,-y)}|x=0)

= #DPLL({(y),(-y)})

🡐 0

🡐 1*1/2 + 0*1/2 = 1/2

CSC2512: #SAT

#DPLL simplest version.
Runs in time 2^n

We can do much better by recognizing that the same formula often arises along different branches.

E.g., (-z, r,-x) (z,-r, -x) : The formula (-x) arises when z=1, r=0, as well as when z=0, r=1

How frequently we get the same formula depends on (a) the connectivity of the formula (branch-width) and (b) which variables we branch on.

CSC2512: #SAT

```
#DPLLSimpleCache($\phi$)
If InCache($\{\phi\}$), return
else
    Pick a variable $v$ in $\phi$
    $\phi^{-} = \phi|_{v=0}$
    #DPLLSimpleCache($\phi^{-}$)
    $\phi^{+} = \phi|_{v=1}$
    #DPLLSimpleCache($\phi^{+}$)
    AddToCache$\left(\phi, \begin{array}{c} \text{GetValue}(\{\phi^{-}\}) \times \frac{1}{2} \\ + \text{GetValue}(\{\phi^{+}\}) \times \frac{1}{2} \end{array}\right)$
    return
```

CSC2512: #SAT

In #DPLLSimpleCache we always cache the # of models for each solved subformula, and look up in the cache the current formula to see if we have already solved it.

There exists an execution of #DPLLSimpleCache that runs in time $2^{O(\mathbf{w} \log n)}$ where **w** is the branch-width of the CNF instance.

Note $2^{O(\mathbf{w} \log n)} = n^{O(w)}$

If the branch-width is small this is much smaller than $2^{O(n)}$

To get a “fast” execution we must follow the variable ordering arising from the small branch-width.

This shows that the same sub-formula can show up many times!

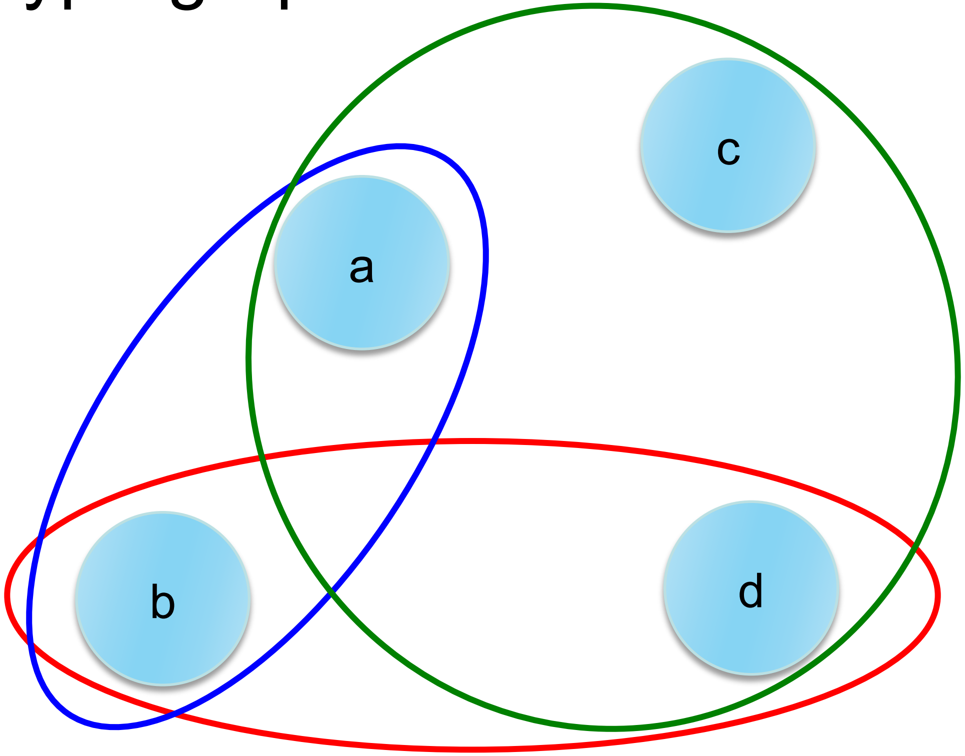
CSC2512: Branch-Width

Branch-width and and Tree-width are metrics that measure how tree-like a graph is. Branch-width of a CNF instance. CNF-Hypergraph:

Nodes are the variables.
Each clause yields a hyperedge over its variables:

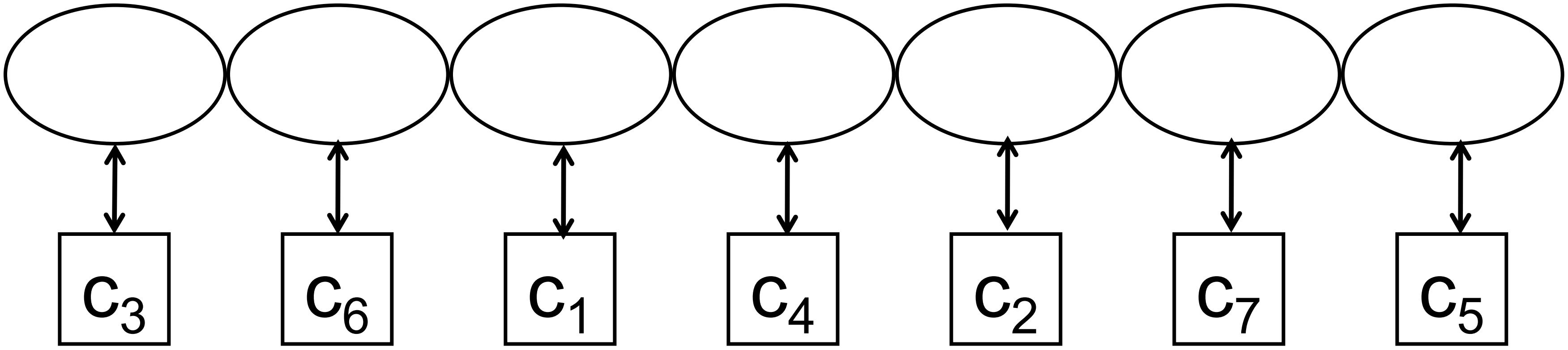
(a,-b) (b,-d) (a,c,d)

Ignore polarity of literals



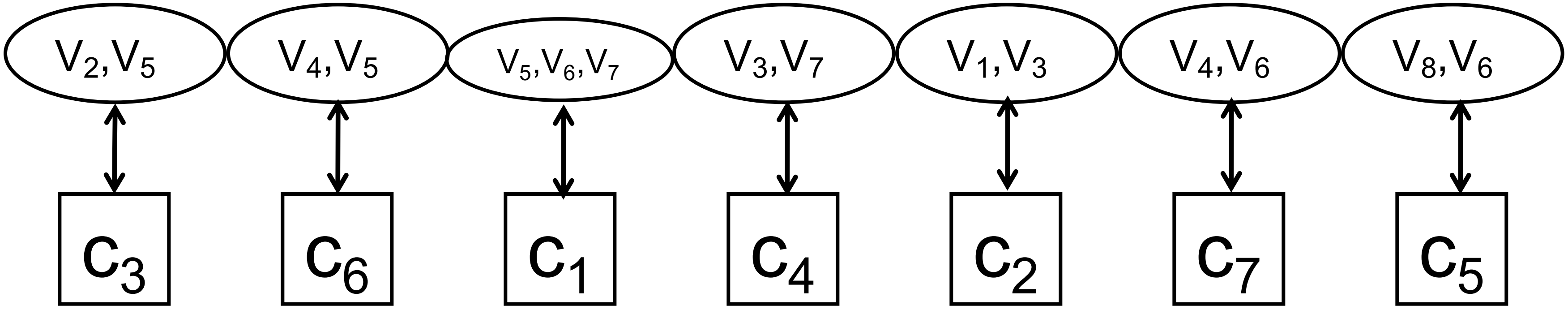
CSC2512: Branch-Width

Associate the clauses with leaves of a binary tree…in any order.



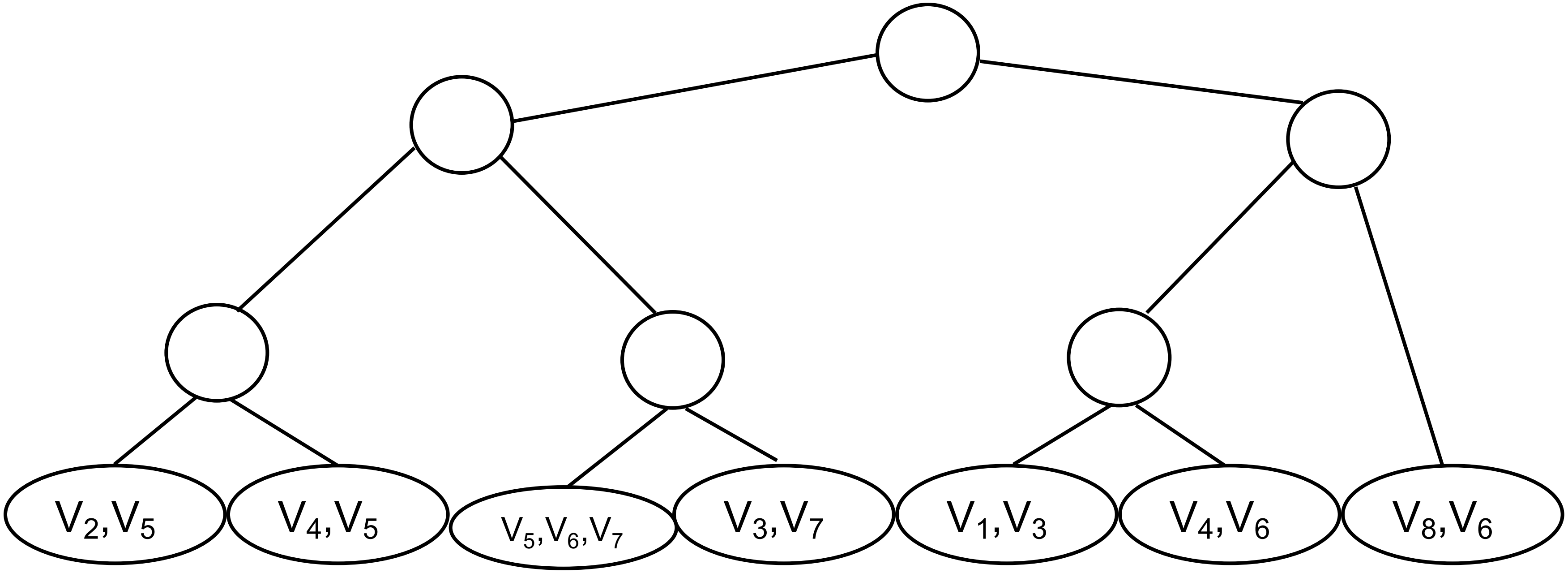
CSC2512: Branch-Width

Label each leaf with the variables of the clause.



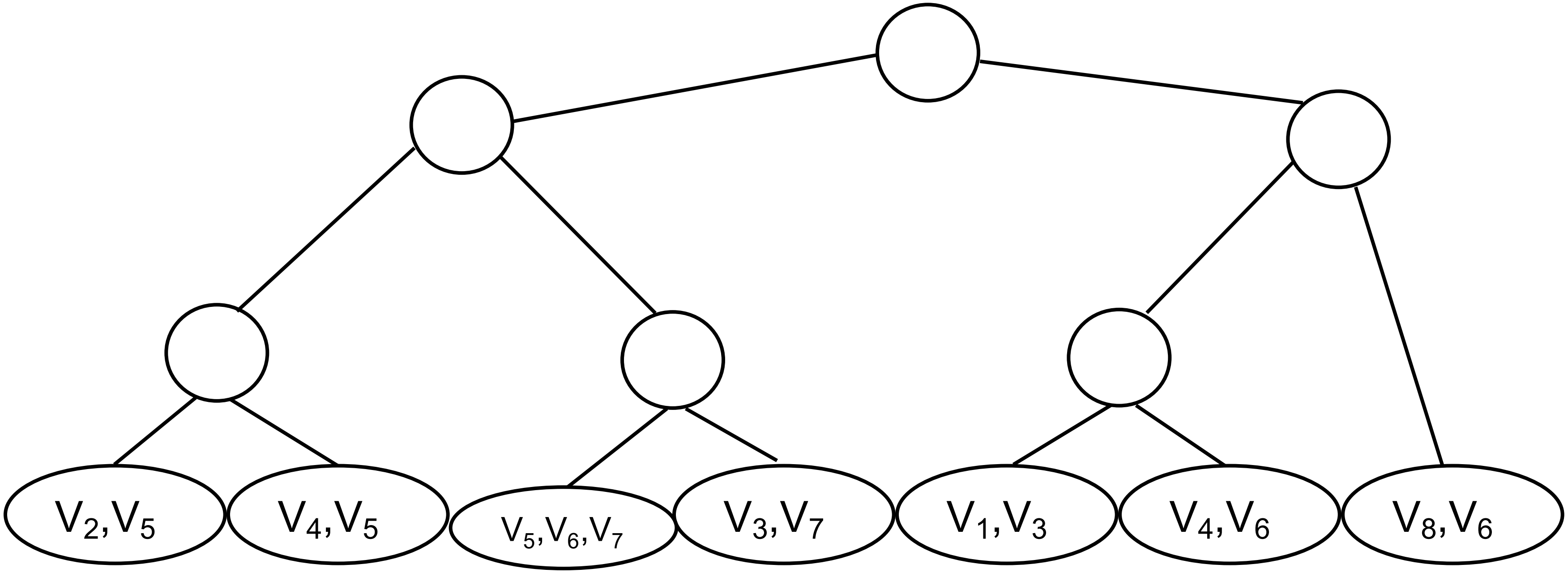
CSC2512: Branch-Width

- Build a binary tree on top of these nodes.



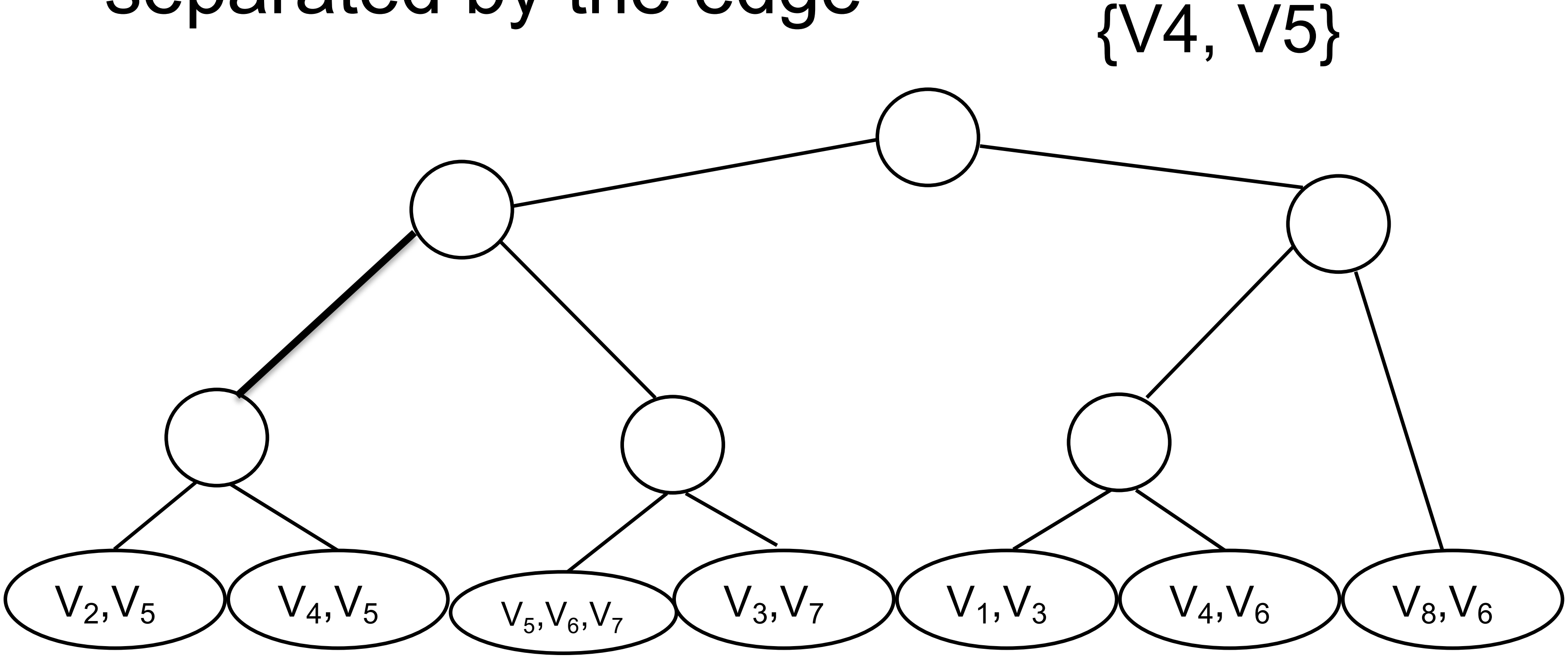
CSC2512: Branch-Width

- Label each edge of the tree with the variables that are in both subgraphs separated by the edge



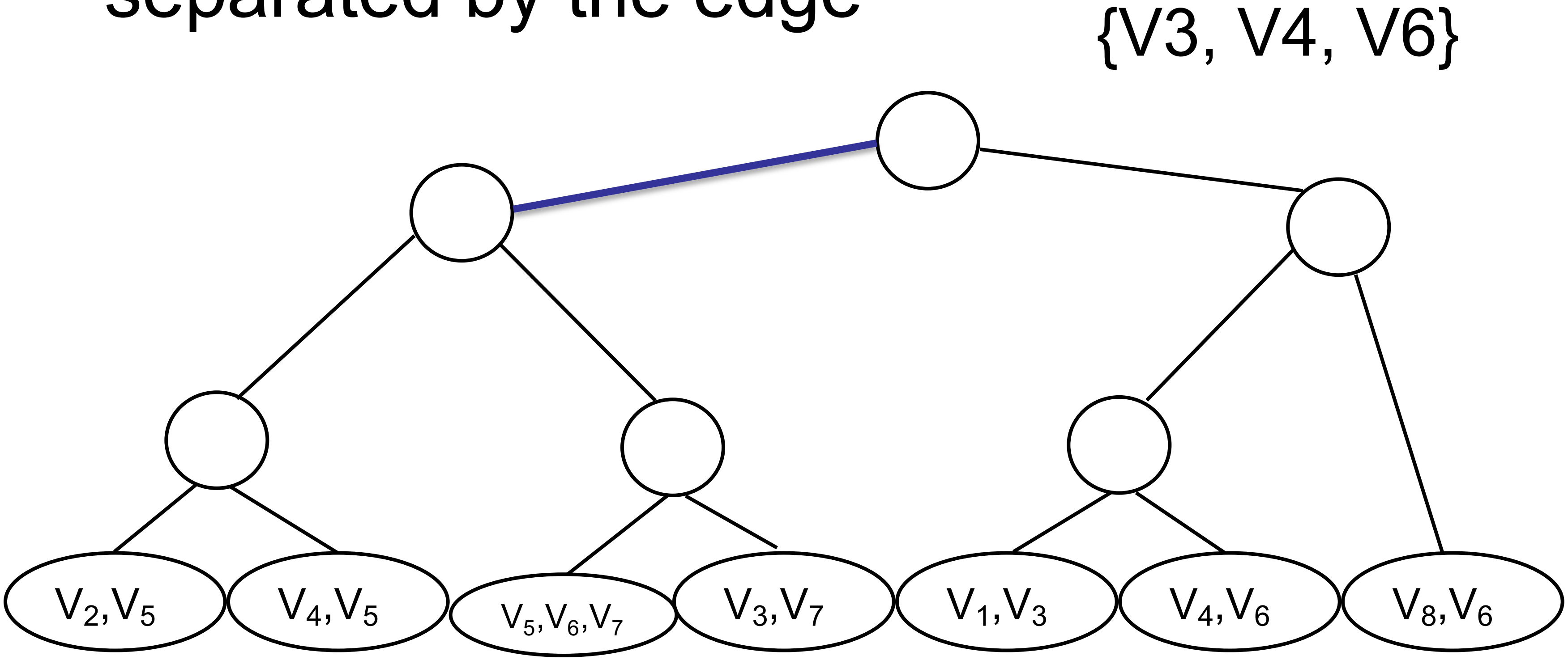
CSC2512: Branch-Width

- Label each edge of the tree with the variables that are in both subgraphs separated by the edge



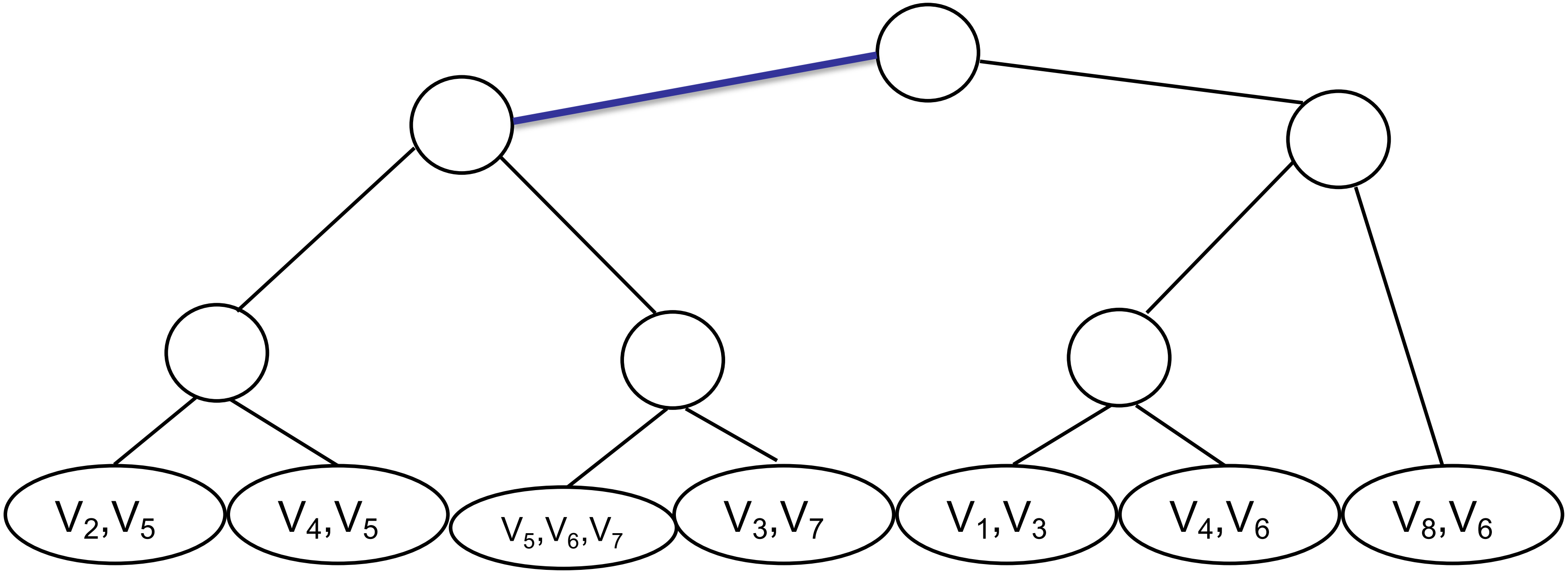
CSC2512: Branch-Width

- Label each edge of the tree with the variables that are in both subgraphs separated by the edge



CSC2512: Branch-Width

- The width of this particular branch decomposition is the size of the largest edge label.

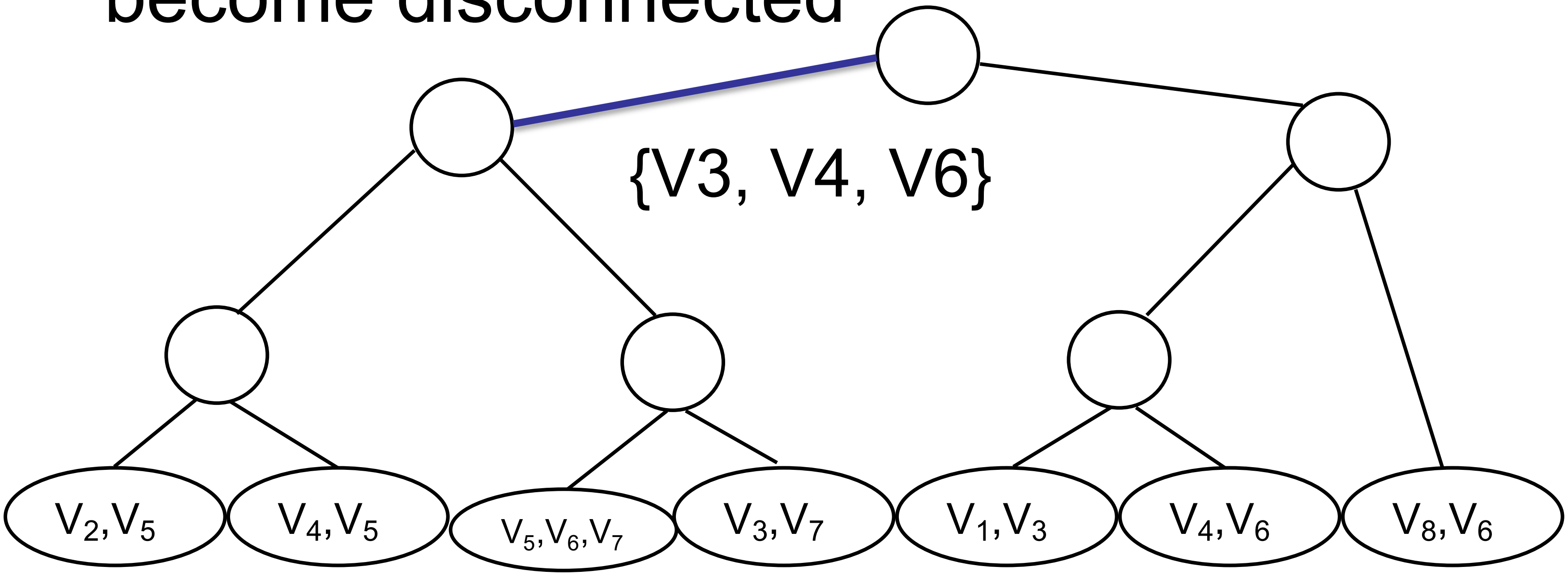


CSC2512: Branch-Width

- Consider all possible branch-decompositions (different orderings of the clause leaves, different binary tree above).
- Each branch-decomposition has a different width.
- The **branch-width** of the CNF is the MIN width over all different branch-decompositions.

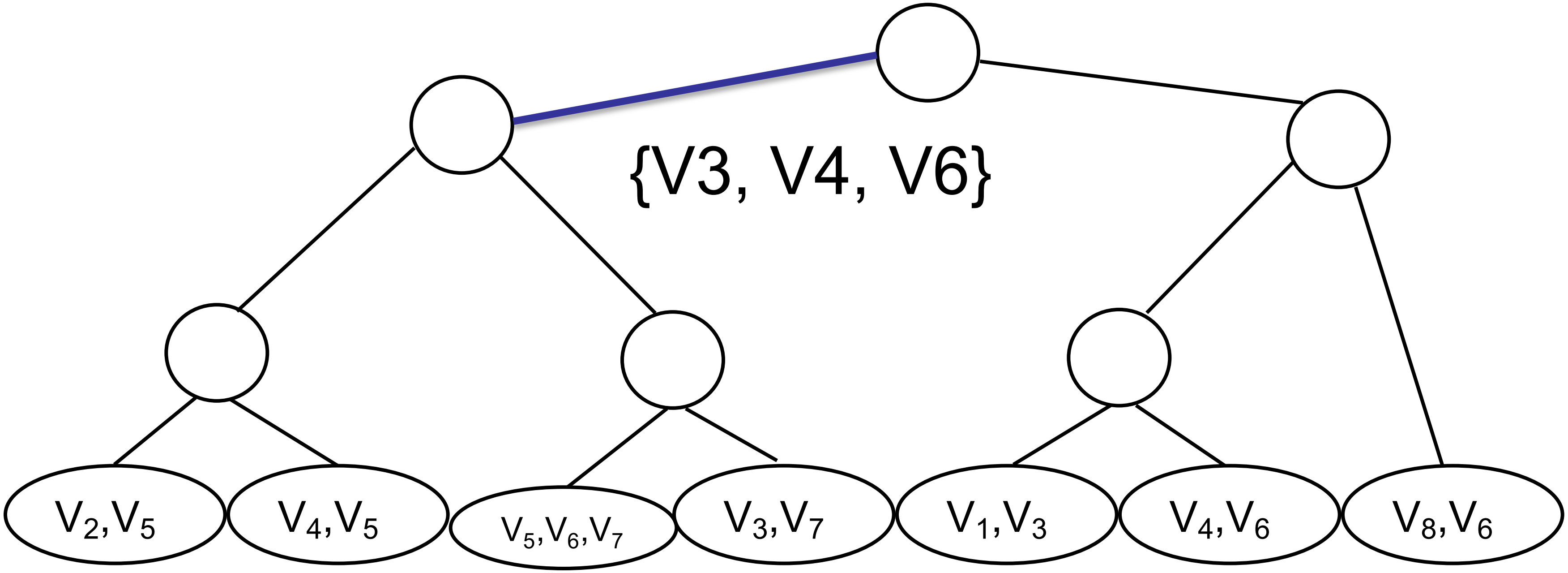
CSC2512: Branch-Width

- Notice that when we assign all of the variables of an edge label the set of clauses in the two parts of the tree become disconnected



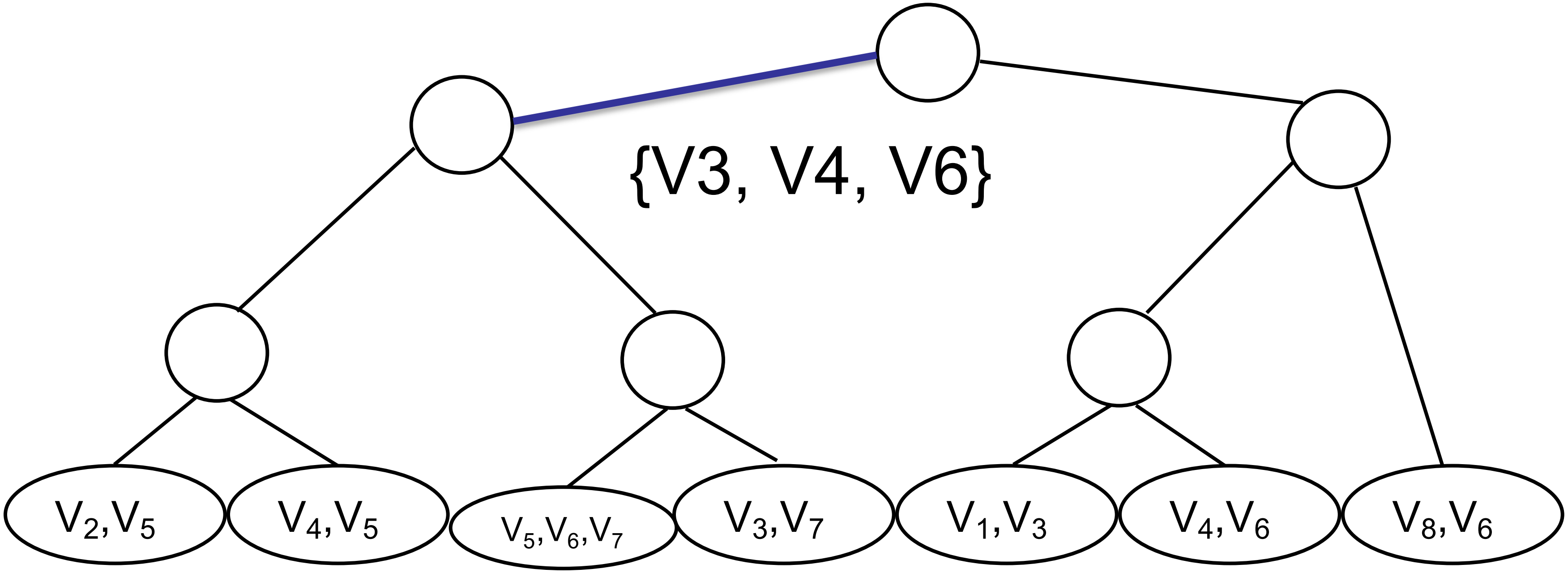
CSC2512: Branch-Width

- That is they will no longer share any variables.



CSC2512: Branch-Width

- So the $2^{O(w\ldots)}$ arises from the fact that the largest edge label has w variables and there are only 2^w different ways to assign them. Thus the clauses in the two subtrees can only be in 2^w different configurations.



CSC2512: #SAT (Model Counting)

- Components. If the CNF can be partitioned into k parts C1, …, Ck, where each part shares on variables with any other part, then each Ci is called a component.

- #DPLL(CNF) = #DPLL(C1) * #DPLL(C2) … * #DPLL(Ck)

- That is we can count the models in each part and multiply to obtain the model count of the whole formula: any satisfying assignment for C1, C2, ... Ck can be combined to form a satisfying assignment for CNF.

- Typically, the input CNF has only one component…it is fully connected.

- This observation gives rise to an algorithm that searches over the current set of components.

CSC2512: #SAT

#DPLL algorithm with component caching

```
#DPLLCache($\Phi$)
If InCache($\Phi$), return
else
    $\Phi$ = RemoveCachedComponents($\Phi$)
    Pick a variable $v$ in some component $\phi \in \Phi$
    $\Phi^{-}$ = ToComponents($\phi|_{v=0}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^{-}$)
    $\Phi^{+}$ = ToComponents($\phi|_{v=1}$)
    #DPLLCache($\Phi - \{\phi\} \cup \Phi^{+}$)
    AddToCache$\left(\phi, \begin{array}{l} \text{GetValue}(\Phi^{-}) \times \frac{1}{2} \\ + \text{GetValue}(\Phi^{+}) \times \frac{1}{2} \end{array}\right)$
    return
```

CSC2512: #SAT (Model Counting)

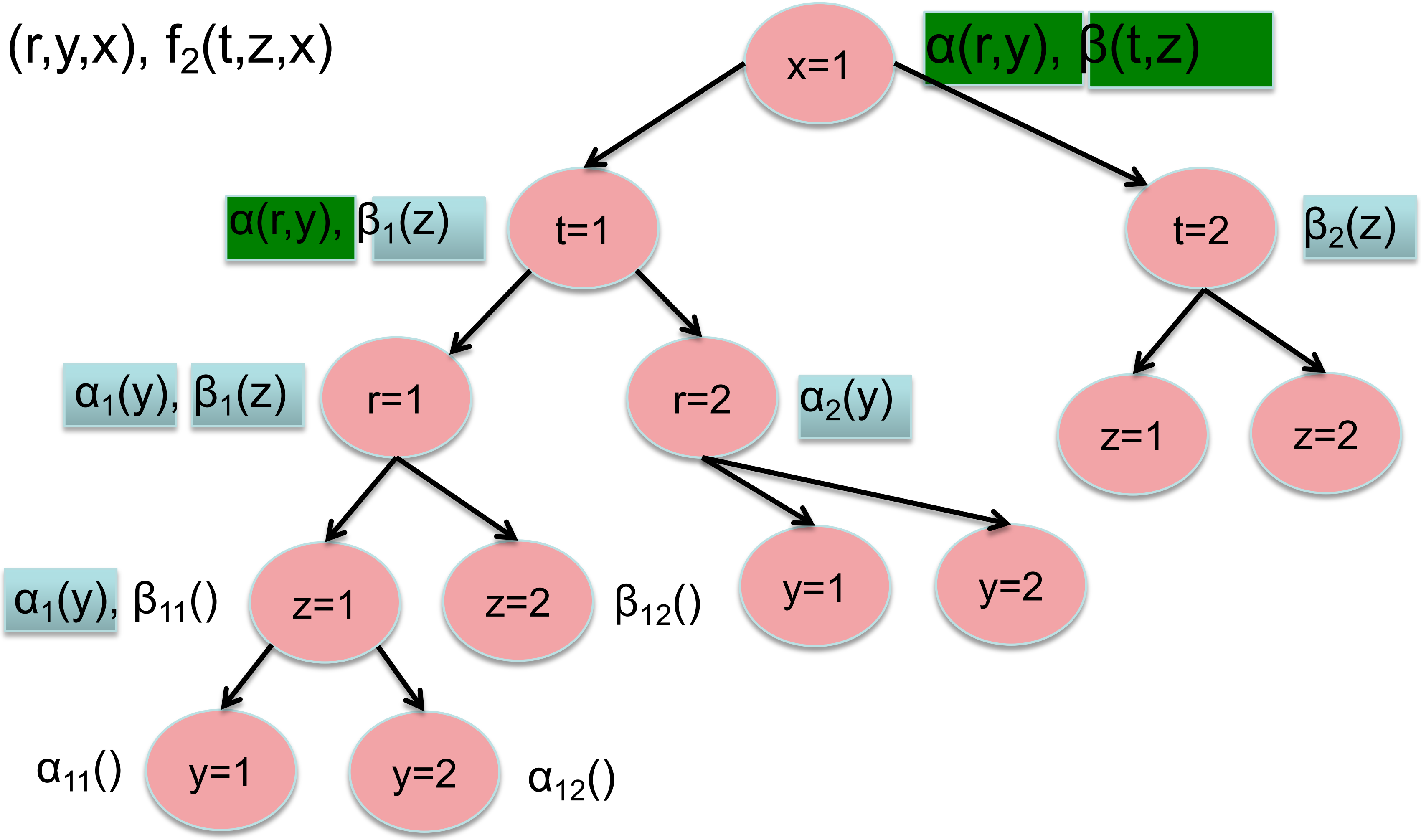
- Initially the cache is empty, at termination the cache contains the input formula and its model count.
- The initial call is given the set of components of the input formula (usually just one component)
- If all components of the input are in the cache we return.
- Otherwise remove all known components.
- Pick a component and a variable of that component to branch on.
- Note that GetValue is passed a set of components. It looks up the value of each component in the cache and returns the product of these values.

CSC2512: #SAT (Model Counting)

- Branch on that variable and compute the new set of components that arise from that branching.
- Solve each value of the variable recursively.
- On return the value of the component we branched on is known, and recursively all other components have been solved and their model counts put in the cache.
- Now all components in input set are solved and we can return.

Search with Components

$f_1(r,y,x), f_2(t,z,x)$



CSC2512: #SAT (Model Counting)

- Solving with components + clause learning
- Can we also use clause learning---yes.
- If c is a learnt clause and F is the input formula then F ⊨ c, and π is a model of F iff it is a model of F ∧ c. So adding learnt clauses does not change the set of models or the model count.
- Learnt clauses span components. So the components of F ∧ c are typically a subset of the components of F. However, if F = $\Phi_1 \land \Phi_2 \land \ldots \land \Phi_k$ where the Φ_i are components. Then F is equivalent to

$$\Phi_1 \land \Phi_2 \land \ldots \land \Phi_k \land c$$

$$\equiv (\Phi_1 \land c) \land (\Phi_2 \land c) \land \ldots \land (\Phi_k \land c)$$

CSC2512: #SAT (Model Counting)

- So we can still solve each component independently, using the set of learnt clauses while solving each component.
- Furthermore if $\Phi_i \models c$ then F ⊨ c so any new clauses learnt while solving Φ_i are also valid learnt clauses for F.
- Bottom line, learnt clauses come along for the ride and are used to enhance unit propagation. **But components are computed ignoring the connections generated by the learnt clauses.**

CSC2512: #SAT (Model Counting)

- However, if among set of components $\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_k$ one of them is unsatisfiable, then clause learning can cause incorrect model counts to be computed for the other components. So we are not allowed to cache computed component values if one of the components is unsat.

CSC2512: #SAT (Model Counting)

- **F = A** ∧ **B** with A = (x,-a,y),(x,-z,b)(a,b,c)
  B = (-y,d)(-d,e)(-e,z)
- **F** ⊧ c = (x,-a,b)
- Under the assignment π: x=0, y=1, z=0
  A|π = (a,b,c)
  B |π = (d)(-d,e)(-e) (unsat)
  c | π = (-a,b)
- A|π and B |π are components while c is over the variables of A. However, the model count of (A ∧ c)|π = 5 while the model count of A|π = 7

CSC2512: #SAT (Model Counting)

- But this problem does not occur when all components are satisfiable (have at least one model).
- So we have to alter #DPLLcache to
  - perform unit prop after each decision variable has been set.
  - compute 1-UIP clauses and backtrack when prop finds a conflict.
  - reduce the current components by the newly implied literals.
  - If we return from a recursive call because of a conflict we do not store anything in the cache.

CSC2512: #SAT (Model Counting)

- Preprocessing is very useful for #SAT-–any simplifications impact the entire search tree.
- Nevertheless exact model counting is hard.
- Recent interest in approximate model counting and random sampling of solutions.

CSC2512: #SAT (Model Counting)

- Readings for next time

1. Preprocessing for Propositional Model Counting, Jean-Marie Lagniez and Pierre Marquis, AAAI-2014
2. Constrained Sampling and Counting: Universal Hashing Meets SAT Solving. Kuldeep S. Meel,Moshe Y. Vardi, Supratik Chakraborty, Daniel J. Fremont, Sanjit A. Seshia, Dror Fried, Alexander Ivrii and Sharad Malik, Beyond NP. Papers from the 2016 AAAI Workshop, Phoenix, Arizona
3. On Computing Minimal Independent Support and Its Applications to Sampling and Counting
   Alexander Ivrii, Sharad Malik, Kuldeep S. Meel, and Moshe Y. Vardi Constraints 21(1), 2016.

CSC2512: #SAT (Model Counting)

- Now some a brief overview of Approximate Model Counting (more info in reading #2 and #3)
- Consider a CNF formula F with a set of models M (satisfying truth assignments). We want to sample uniformly at random from M.
  - The models correspond to atomic events in a finite probability distribution and we want to sample these atomic events randomly.
  - Note that F is a **compact** way of representing the exponential sized set of models M.

CSC2512: #SAT (Model Counting)

- By simply sampling from M, we are sampling from a **uniform** distribution over the models.
- In general, we would want each atomic event to have a different probability, and we would want to sample a model m ∈ M according to this distribution.
  - That is, we would want the probability of obtaining m as our sample to be equal to its probability.
- This issue can be handled by adding more variables to F and setting those variables according to the weight of each model m. This gives us the formula F'' which is a superset of F.

CSC2512: #SAT (Model Counting)

- F'' has new variables and clauses, and has the property that for every model **m** of F can be extended to a number of models of F''
- F'' is specified so that the number of models of **m** in F'' is proportional to the probability of **m.**
- So high probability models get more models in F''.
- Ultimately, sampling uniformly at randomly of F'' allows us to sample according the probabilities over F.

CSC2512: #SAT (Model Counting)

- So how do we sample randomly from the models of F.
- The idea is related to hashing
- A Hash function maps [1—n] to [1—m] (e.g., a set of size n to a set of size m (typically n and m are powers of two), and it is designed to do so uniformly, so that approximately the same number of items map to each value m.
- Let H(n,m) be a family of different Hash functions from n to m.

CSC2512: #SAT (Model Counting)

- Consider the uniform distribution over H(n,m)
- The family H(n,m) is called **universal** if
  forall i, j ∈[1—n] with i ≠ j we have
  $$\Pr[h(i) = h(j)] \leq 1/m$$
  where h is selected from H(n,m) under the uniform distribution.
- We also need the notion of **r-universal.** A family of hash functions H(n,m,r) is called **r-universal** if for all distinct $x_1, x_2, \ldots, x_r \in [1—n]$ and all $a_1, a_2, \ldots, a_r \in [1—m]$ we have
  $$P[h(x_1)= a_1 \wedge \ldots \wedge h(x_r)= a_r] = m^{-r}$$
  where h is selected from H(n,m,r) at random.

CSC2512: #SAT (Model Counting)

- One prominent approach to approximate sampling of the models of a SAT formula F is to use a XOR class of hash functions:

  $H_{xor}(2^n,2^m)$ (i.e., n-bits to m bits)

  $\{h \mid h(x)[i] = a_{i,0} \oplus a_{i,1} * x[1] \oplus \ldots \oplus a_{i,n} * x[n]\}$

  where $\oplus$ is XOR and $a_{i,j}$ is 0 or 1. That is, each bit of m-bit output is computed by XORing a base 0/1 ($a_{i,0}$) along with all bits of the n-bit input multiplied by 0 or 1.

- So by choosing $a_{i,j}$ randomly we can choose a hash function h from H_{xor}
- h(x)=y can be computed by $y = Ax \oplus \mathbf{a_0}$ where A is the martix of $a_{i,j}$ values and $\mathbf{a_0}$ is the vector of base bits.

CSC2512: #SAT (Model Counting)

- It has been shown that H_{xor} is 3-universal.

- Then the basic idea is to constraint the input formula F with a randomly selected hash function h ∈ H_{xor} constrained to map to a randomly selected m-bit.
  - That is, we only admit models of F that are mapped to this specific m-bit number by the hash function.
  - This should reduce the number of models by a factor of 1/m and should randomly select which models remain.
- Then with this constrained version of F we count the number of models. That count N* is an estimate of F's original model count.

CSC2512: #SAT (Model Counting)

- However, there are a number of practical impediments:
  - Counting 1/m the models of F might be too many.
    - We can iteratively add more random hash functions constrained to map to random m-bits.
    - Each such hash function cuts down the number of models by a factor of 1/m
    - When we have added a sufficient number we can more easily count the models remaining.

CSC2512: #SAT (Model Counting)

- Counting is still hard, so we can go further
  - Add a sufficient number of hash functions so that there the formula becomes unsat (no models left). If we added k hash functions constraints, and got UNSAT but k-1 was SAT then we can estimate that the number of models is in the range

    m^k and m^{k-1}

  - Under this scheme we don’t have to count we only have to SAT solve.

CSC2512: #SAT (Model Counting)

- SAT solving with these hash function constraints can be very hard. So methods have been found to use shorter hash function constraints (hash functions over fewer variables) and still obtain the probabilistic guarantees we want.