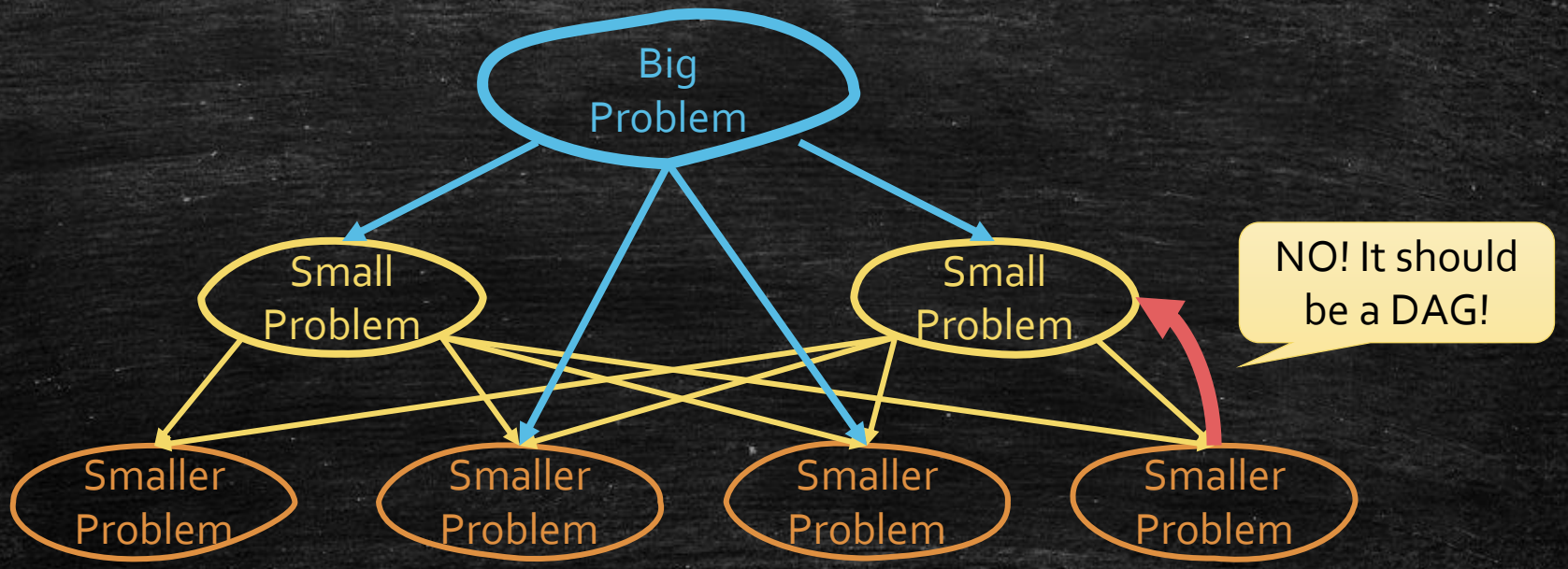
Dynamic Programming

DP on graphs

Dynamic Programming



A simpler guideline

- Find subproblems.
- Check whether we are in a **DAG** and find the **topological order** of this DAG. (Usually, by hand.)
- Solve & store the subproblems by the topological order.

Understand Bellman-Ford as A DP

Bellman-Ford

Function bellman_ford(G, s)
    $dist[s] = 0, dist[x] = \infty$ for other $x \in V$
    while $\exists dist[x]$ is updated
        for each $(u, v) \in E$
            $dist[v] = \min\{dist[v], dist[u] + d(u, v)\}$

Lemma 1

After k rounds, $dist(v)$ is the shortest distance of all **k-edge-path (path with at most k edges)**.

Define subproblems

- $dist[k, v]$: the shortest distance from s to v among all **k-edge-path (path with at most k edges)**.

Observation 2

The shortest distance of all **$|V|$-edge-path** can not be shorter than the shortest distance of all **$(|V| - 1)$ –edge-path** unless there is a **Negative Cycle.**

Bellman-Ford

function bellman_ford(G, s)
    $dist[0, s] = 0, dist[0, x] = \infty$ for other $x \in V$
    for $k = 1$ to $|V|$
        for each $(u, v) \in E$
            $dist[k, v] = \min\{dist[k-1, v], dist[k-1, u] + d(u, v)\}$

Solving Subproblems

- $dist[k, v] = \min\{dist[k-1, v], dist[k-1, u] + d(u, v)\}$

$f[k, v]$	s	v_2	v_3	v_4	v_5	v_6	v_7	…	$v_{\|V\|}$
0	0	∞	∞	∞	∞	∞	∞	∞	∞
1									
2									
3						$f[k, v]$			
…									
$\|V\|$									

All Pair Shortest Path

- **Input:** A directed graph $G(V,E)$, and a weighted function $d(u,v)$ for all $(u,v) \in E$.
- **Output:** Distance $dist(u,v)$, for **all vertex pair** u, v.

What can we do?

- Naïve Plan:
  - Run $|V|$ times Bellman-Ford
  - $O(|V|^2|E|)$
- Improve it by an integrated DP!
  - Floyd-Warshall Algorithm!
  - $O(|V|^3)$
  - History from Wikipedia:

History and naming [edit]

The Floyd–Warshall algorithm is an example of dynamic programming, and was published in its currently recognized form by Robert Floyd in 1962.[3] However, it is essentially the same as algorithms previously published by Bernard Roy in 1959[4] and also by Stephen Warshall in 1962[5] for finding the transitive closure of a graph,[6] and is closely related to Kleene's algorithm (published in 1956) for converting a deterministic finite automaton into a regular expression.[7] The modern formulation of the algorithm as three nested for-loops was first described by Peter Ingerman, also in 1962.[8]

Define subproblems

- Bellman-Ford: $dist[k, v]$: the shortest distance from s to v among all **k-edge-path (path with at most k edges)**.
- A very natural generalization!
- Natural Generalization: $dist[k, u, v]$: the shortest distance from u to v among all **k-edge-path (path with at most k edges)**.

Natural Generalization

- Natural Generalization: $dist[k, u, v]$: the shortest distance from u to v among all **k-edge-path (path with at most k edges)**.
- Transfer:
  - $dist[k, u, v] = \min_{(s,v)\in E}\{dist[k-1, u, s] + d(s, v)\}$
- Time:
  - $|V|$ rounds
  - In one round, an edge can be used to update $|V|$ distance.
  - Totally $O(|V|^2|E|)$!

No improvement!

Solving Subproblems

- $dist[k, u, v] = \min_{(s,v) \in E} \{dist[k-1, u, s] + d(s, v)\}$

$k-1$	v_1	v_2	v_3	…	$v_{\|V\|}$
v_1					
v_2					
v_3					
v_4					
…					
$v_{\|V\|}$					

k	v_1	v_2	v_3	…	$v_{\|V\|}$
v_1					
v_2					
v_3					
v_4			$f[k, u, v]$		
…					
$v_{\|V\|}$					

The point for improvement

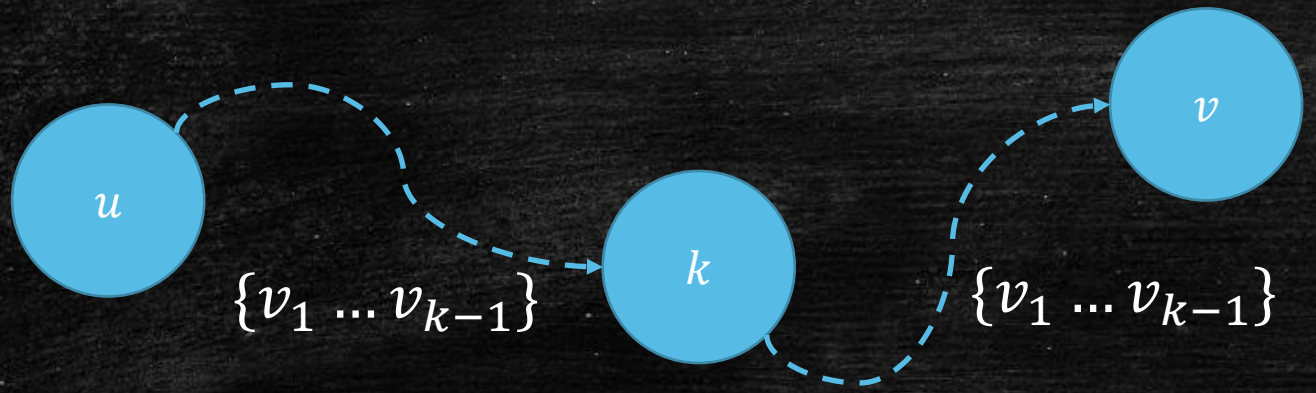
- Transfer:
  - $dist[k, u, v] = \min_{(s,v)\in E}\{dist[k-1, u, s] + d(s, v)\}$
- Problem
  - We know all states of $dist[k-1, u, v]$.
  - We only use $dist[k-1, u, s]$ to update $dist[k-1, u, v]$. (start from u)
  - Different start points are independent.
  - The same as run $|V|$ times Bellman-Ford.

Floyd-Warshall: Subproblems

- Natural Generalization: $dist[k, u, v]$: the shortest distance from u to v among all **k-edge-path (path with at most k edges)**.
- Floyd-Warshall: $dist[k, u, v]$: the shortest distance from u to v that **only across inter-vertices in $\{v_1 \dots v_k\}$.**
- Remark:
  - We can label vertices from 1 to $|V|$.
  - $dist[0, u, v]$ is exactly $d(u, v)$ or ∞. (allow 0 inter-vertex)
  - $dist[|V|, u, v]$ is exactly what we want!

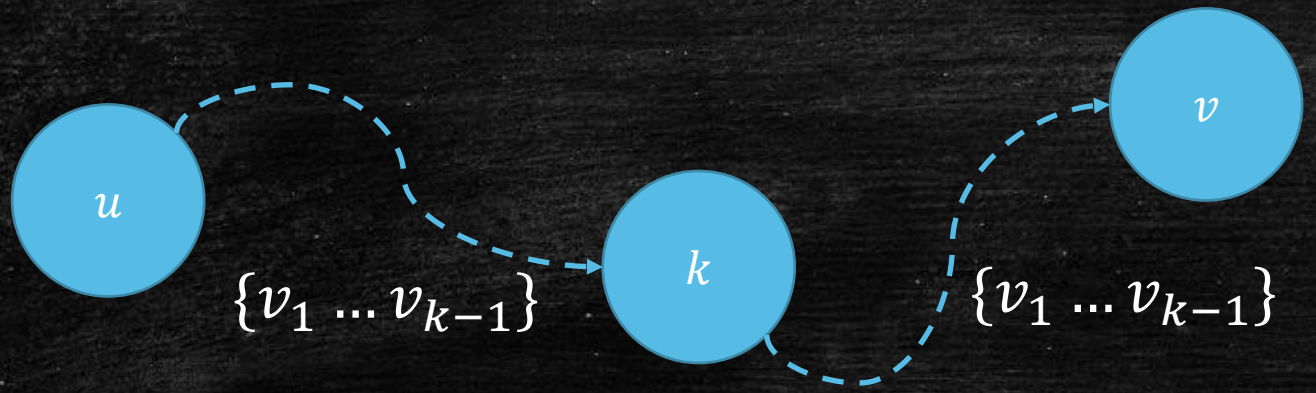
Key Fact

- If there is no negative cycle, v_k only appear once.
- Focusing on positive edge weight.
- v_k on any $dist[k', u, v]$ for $k' \geq k$.



Floyd-Warshall: Solving Subproblems

- $dist[k, u, v]$: the shortest distance from u to v that only **across inter-vertices in $\{v_1 \dots v_k\}$.**
- Solve $dist[k, u, v]$ (give addition power k to all pairs)
  - Case 1: the shortest path do not go across k.
  - Case 2: the shortest path go across k.
  - $dist[k, u, v] = \min\{dist[k-1, u, v], dist[k-1, u, k] + dist[k-1, k, v]\}$



Solving Subproblems

- $dist[k, u, v] = \min\{dist[k-1, u, v], dist[k-1, u, k] + dist[k-1, k, v]\}$

$k-1$	v_1	v_2	v_3	…	$v_{\|V\|}$
v_1					
v_2					
v_3					
v_4					
…					
$v_{\|V\|}$					

k	v_1	v_2	v_3	…	$v_{\|V\|}$
v_1					
v_2					
v_3					
v_4			$f[k, u, v]$		
…					
$v_{\|V\|}$					

DAG and Topological

- $dist[k, u, v]$ only depends
  - $dist[k-1, u, v]$
  - $dist[k-1, u, k]$
  - $dist[k-1, k, v]$
- We initialize $dist[0, u, v] = d(u, v)$ for all (u, v).
- Solve them from $k = 1$ to n is a topological order.
- Running Time: $3 \cdot O(|V| \cdot |V| \cdot |V|)$

Floyd-Warshall

Floyd-Warshall $O(|V|^3)$

function floyd_warshall(G)
    $dist[0, u, v] = d(u, v)$ for all $(u, v) \in E$, $dist[0, u, v] = \infty$ otherwise.
    for $k = 1$ to $|V|$
        for $u = 1$ to $|V|$
            for $v = 1$ to $|V|$
                $dist[k, u, v] = \min\{dist[k-1, u, v], dist[k-1, u, k] + dist[k-1, k, v]\}$

Floyd-Warshall: a simpler implement

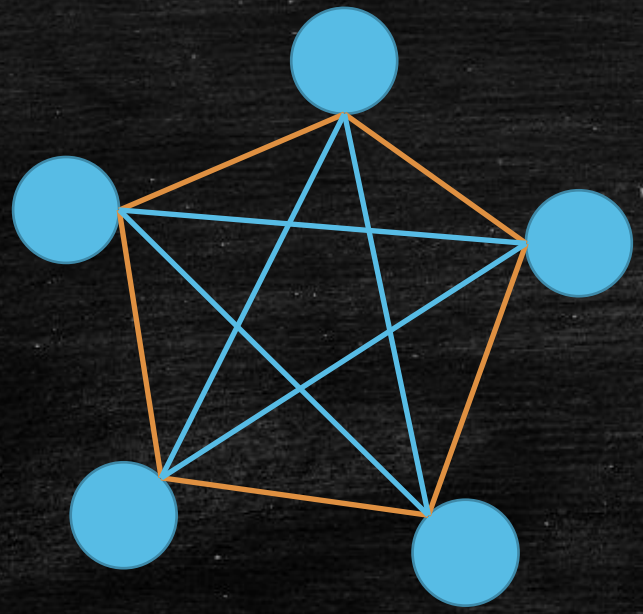
Floyd-Warshall

function floyd_warshall(G)
    $dist[u, v] = d(u, v)$ for all $(u, v) \in E$, $dist[u, v] = \infty$ otherwise.
    for $k = 1$ to $|V|$
        for $u = 1$ to $|V|$
            for $v = 1$ to $|V|$
                $dist[u, v] = \min\{dist[u, v], dist[u, k] + dist[k, v]\}$

$O(|V|^3)$ running time but $O(|V|^2)$ space! Why it is correct?

Traveling Salesman Problem (TSP)

- **Input:** A complete weighted undirected graph G, such that $d(u, v) > 0$ for each pair u, v $(u \neq v)$.
- **Output:** the cycle of n vertices with the minimum weight.

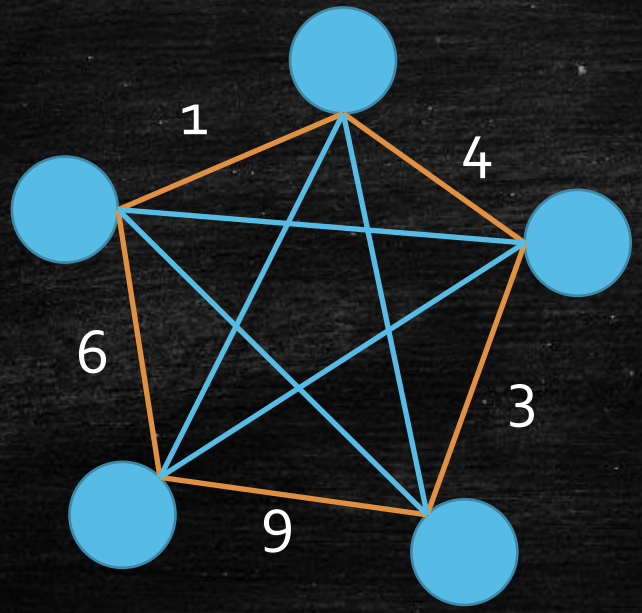


How to brute-force

What is the time complexity?

TSP vs. Shortest Path

- TSP
  - **Output:** the cycle of n vertices with the minimum weight.
- All Pair Shortest Path
  - **Output:** the minimum weight path from u to v.

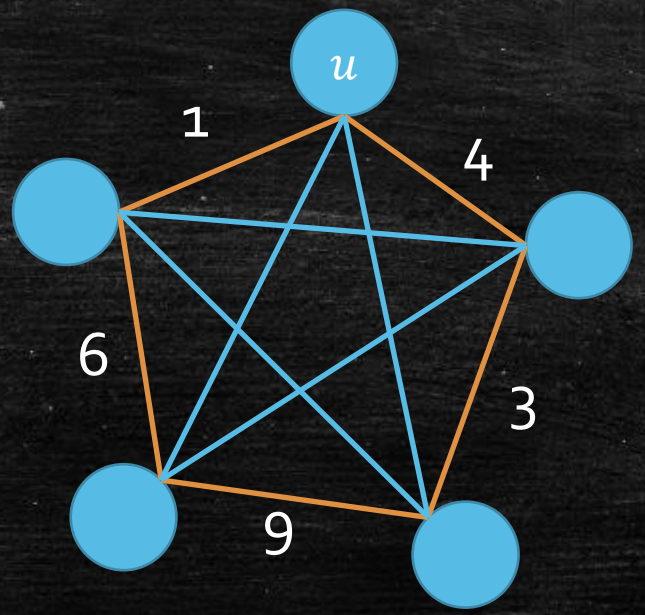


Subproblems in Shortest Path Problem

- $f[k, u, v]$
  - The shortest path from u to v, with inter-vertex chosen in $v_1 \ldots v_k$.
- Is it good for TSP?
- What we should do now?

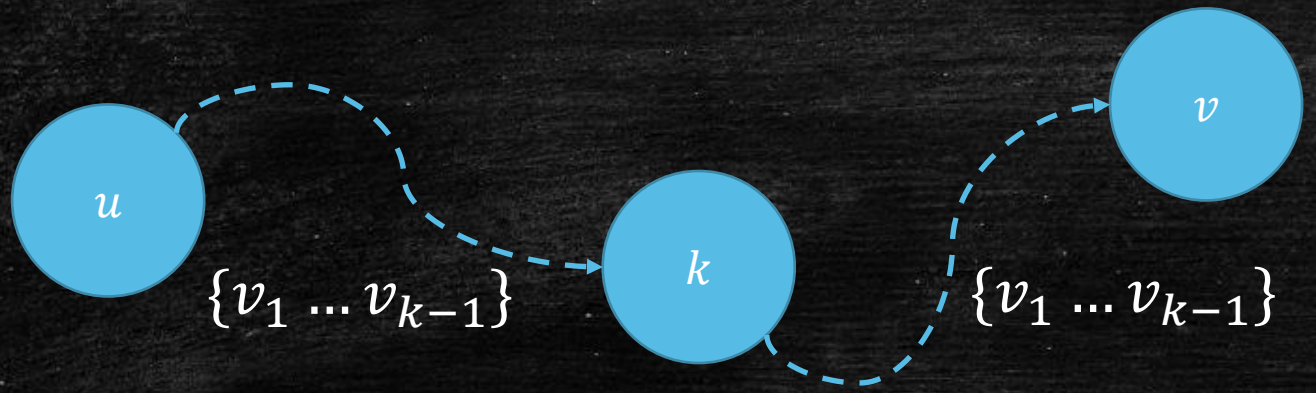
Plan A

- $f[k, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $v_1 \dots v_k$ except u and v.
- How to solve TSP?
  - $\min_u f[|V|, u, u]$ is what we want!
- How to solve $f[k, u, v]$?



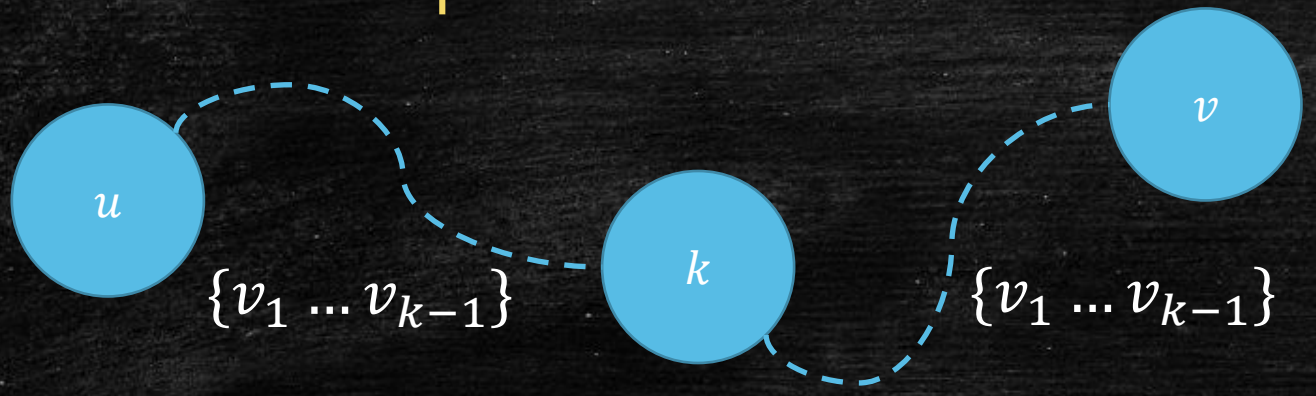
Floyd-Warshall: Solving Subproblems

- $dist[k, u, v]$: the shortest distance from u to v that only **across inter-vertices in $\{v_1 \dots v_k\}$.**
- Solve $dist[k, u, v]$ (give addition power k to all pairs)
  - Case 1: the shortest path do not go across k.
  - Case 2: the shortest path go across k.
  - $dist[k, u, v] = \min\{dist[k-1, u, v], dist[k-1, u, k] + dist[k-1, k, v]\}$



Plan A: Subproblem Definition

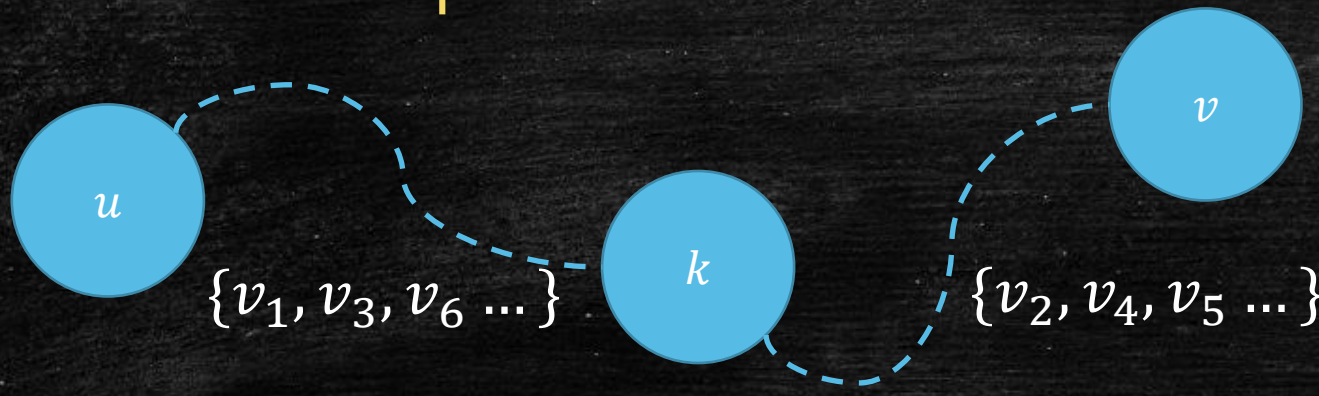
- $f[k, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $v_1 \dots v_k$ except u and v.
  - $\min_u f[|V|, u, u]$ is what we want!
- How to solve $f[k, u, v]$?
- What is the problem now?



Two sub paths can not contain same vertices.

Plan A: Why it is not enough?

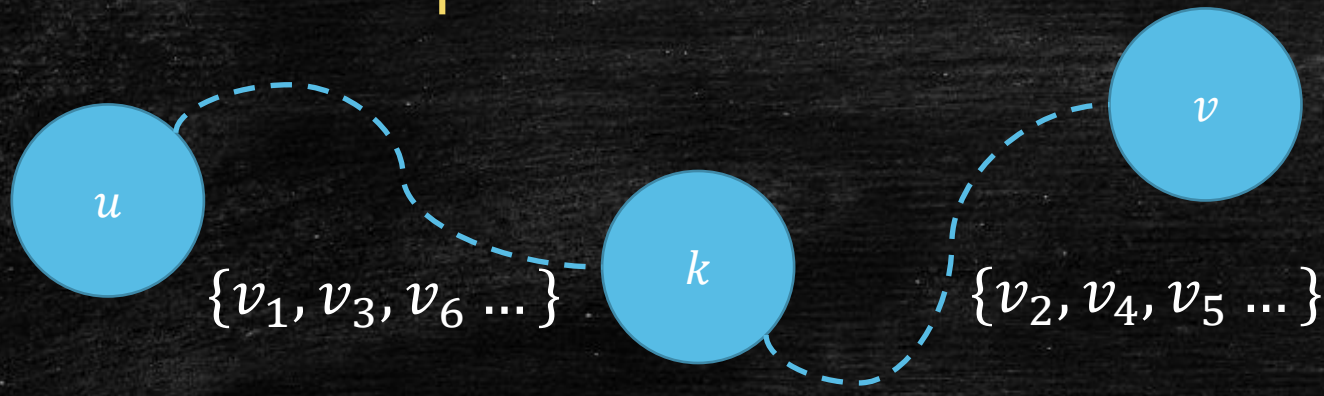
- $f[k, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $v_1 \dots v_k$ except u and v.
  - $\min_u f[|V|, u, u]$ is what we want!
- How to solve $f[k, u, v]$?
- What is the problem now?



We need to know
- what vertices $u \to k$ use?
- what vertices $k \to v$ use?

Plan A: Why it is not enough?

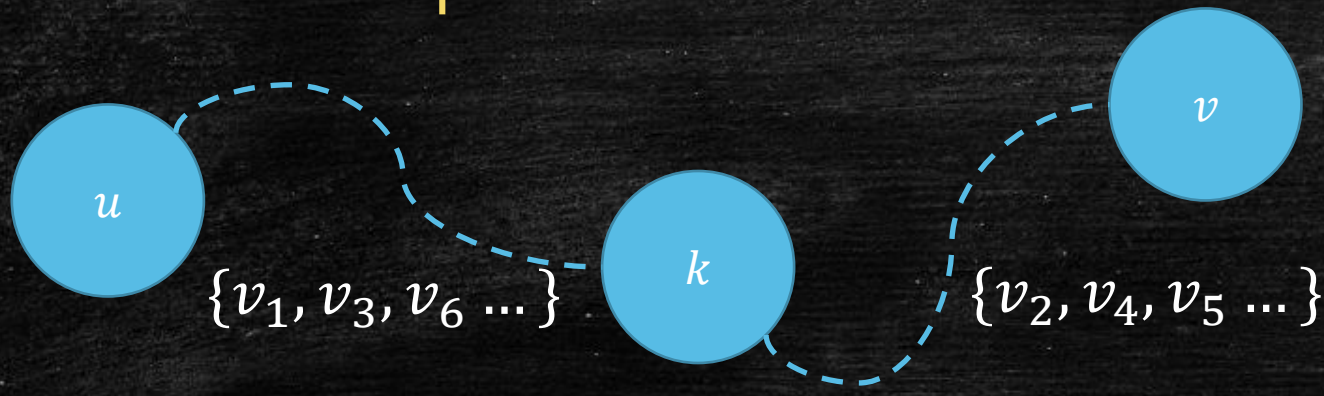
- $f[k, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $v_1 \dots v_k$ except u and v.
  - $\min_u f[|V|, u, u]$ is what we want!
- How to solve $f[k, u, v]$?
- What is the problem now?



We need to know
- what vertices $u \to k$ use?
- what vertices $k \to v$ use?

Plan A: Why it is not enough?

- $f[k, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $v_1 \dots v_k$ except u and v.
  - $\min_u f[|V|, u, u]$ is what we want!
- How to solve $f[k, u, v]$?
- What is the problem now?



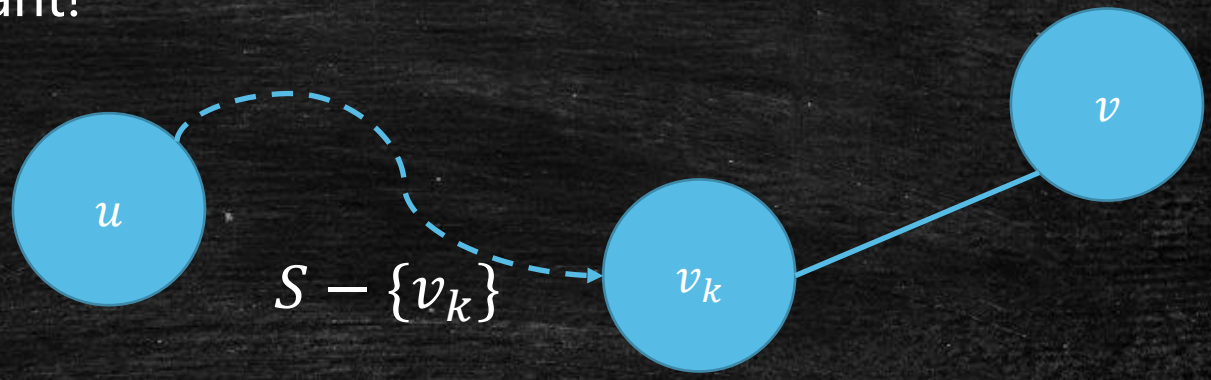
We need to know
- what vertices $u \to k$ use?
- what vertices $k \to v$ use?

We do not solve the subproblem $u \to k$ with $\{v_1, v_3, v_6 \dots\}$

Do you know how to fix?

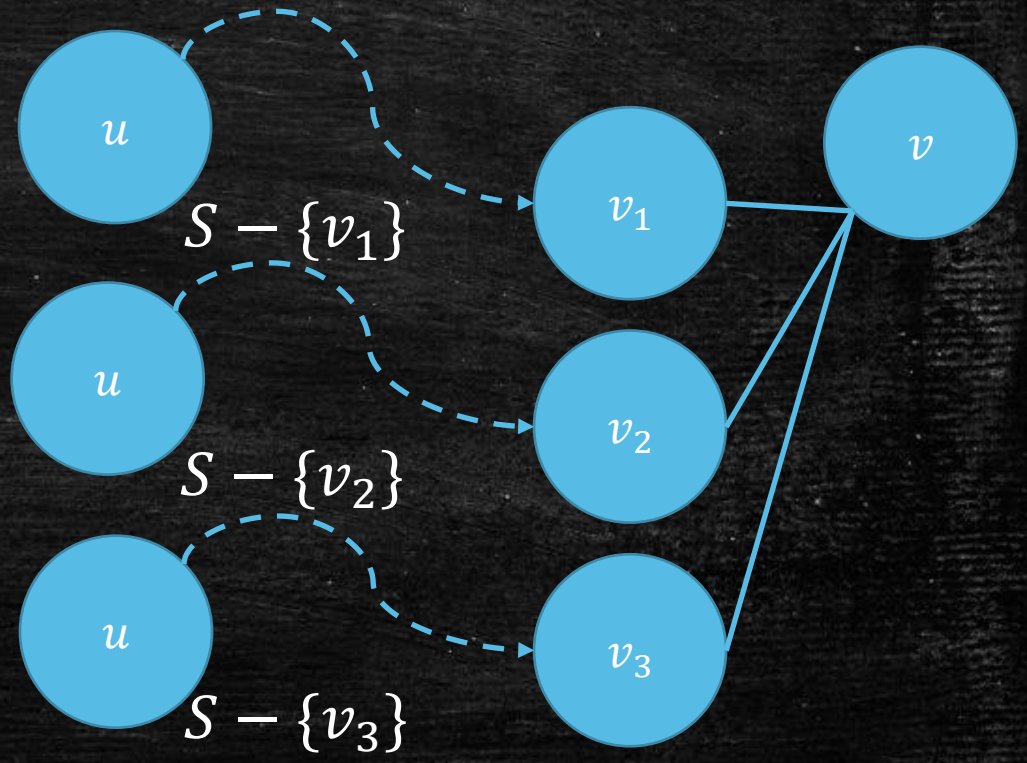
Plan B: Subproblem Definition

- $f[S, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $S \subset V$ except u and v.
  - $\min_u f[V, u, u]$ is what we want!
- How to solve $f[S, u, v]$?



Plan B: Solving Subproblems

- $f[\boldsymbol{S}, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $\boldsymbol{S} \subset \boldsymbol{V}$ except u and v.
  - $\min_u f[\boldsymbol{V}, u, u]$ is what we want!
- How to solve $f[\boldsymbol{S}, u, v]$?
- $f[S, u, v] = \min_{k \in S} f[S - \{k\}, u, k] + d(k, v)$

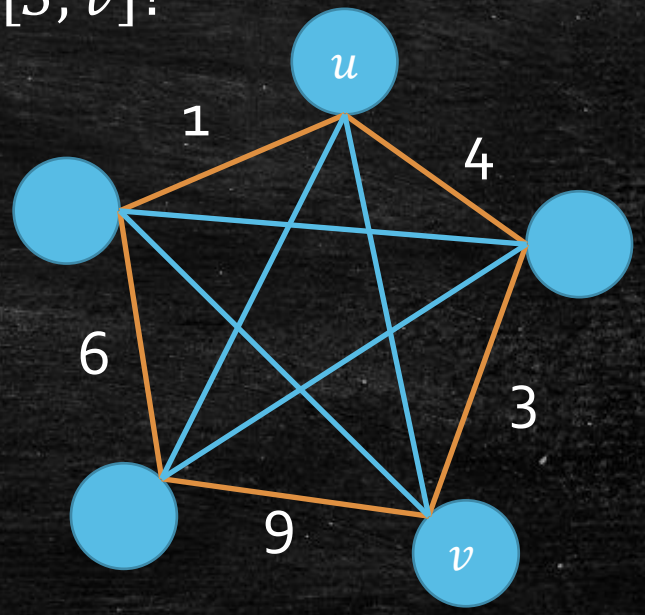


Summarize

- $f[S, u, v]$
  - The shortest path from u to v with inter-vertex **exactly** $S \subset V$ except u and v.
  - $\min_u f[V, u, u]$ is what we want!
- $f[S, u, v] = \min_{k \in S} f[S - \{k\}, u, k] + d(k, v)$
- Do you know the topological order of the DP?

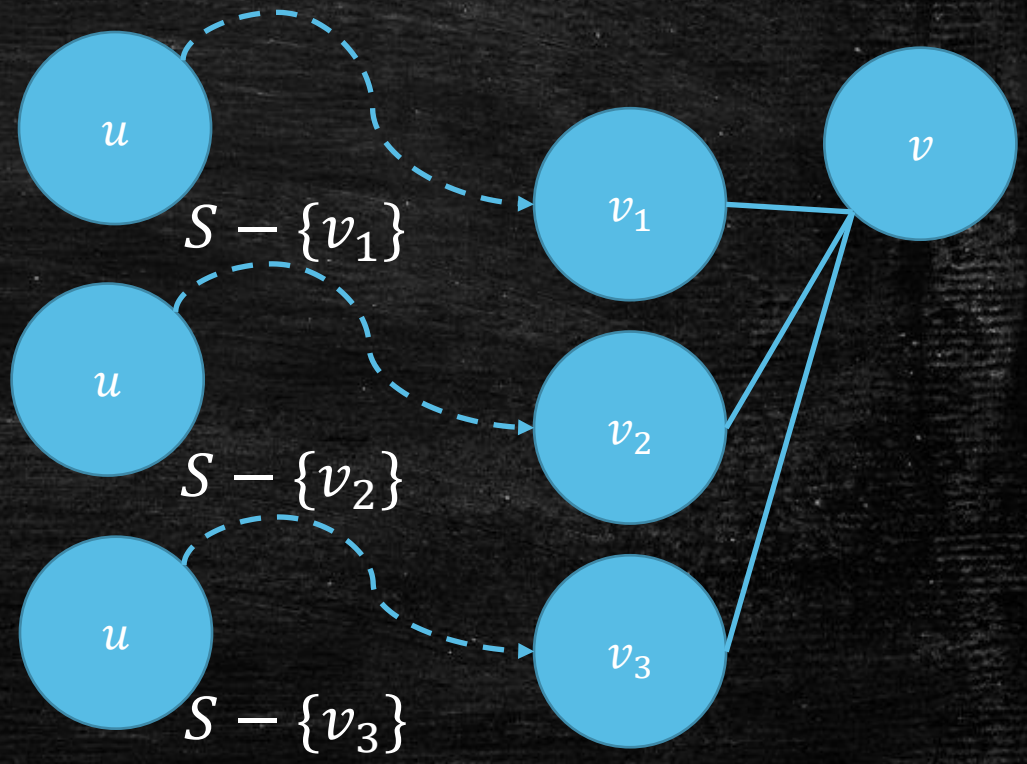
A little improvement:

- $\forall u, v, f[V, u, u] = f[V, v, v]$!
- We only need to know one fixed $f[V, u, u]$.
- Can we fix an arbitrary u and only solve $f[S, v]$?



Solve $f[S, v]$!

- $f[S, u, v] = \min_{k \in V} f[S - \{k\}, u, k] + d(k, v)$
- $f[S, u, v]$ only comes from $f[S - \{k\}, u, k]$.
- It is enough for us to only record $f[S, v]$.
- $f[S, v] = \min_{k \in V} f[S - \{k\}, k] + d(k, v)$.



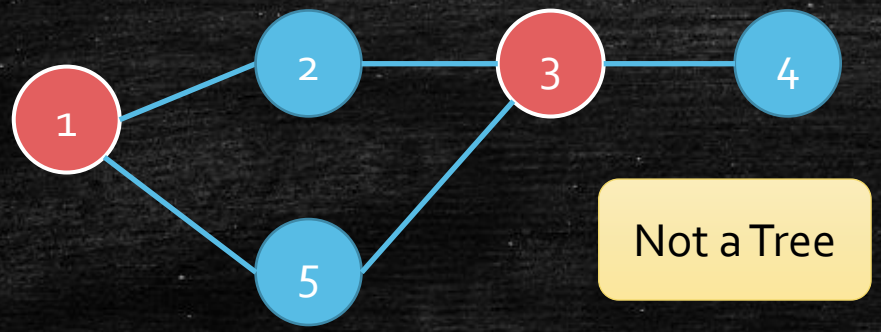
Time Complexity

- Time complexity $(n = |V|)$?
  - $O(n2^n)$ subproblems.
  - $O(n)$ solving.
  - $\boldsymbol{O(n^2 2^n)}$ totally!
- Comparing to Brute-force
  - Brute-force: $\boldsymbol{O(n!)}$
  - Do you know why $O(n^2 2^n)$ is better than $O(n!)$?
  - Do you know why DP is better than brute-force?
- Do you know how to implement $f[S, v]$?
  - S is a set.
  - Use $v(s) \in (0 \dots 2^n)$ to represent!

Solve Problems on Trees

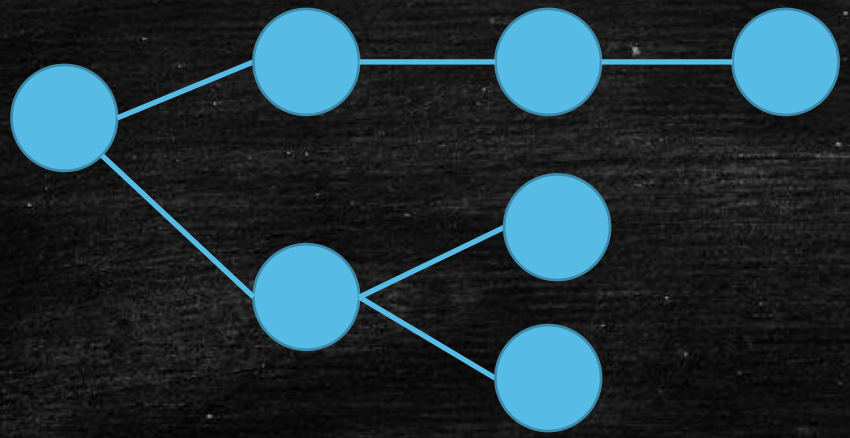
Maximize Independent Set on Trees

- **Input:** an undirected tree $G = (V, E)$.
- **Output:** an independent set with maximum cardinality (number of vertices)
- **Independent Set:** a set S of vertices:
  - $\forall u, v \in S$, we have $(u, v) \notin E$



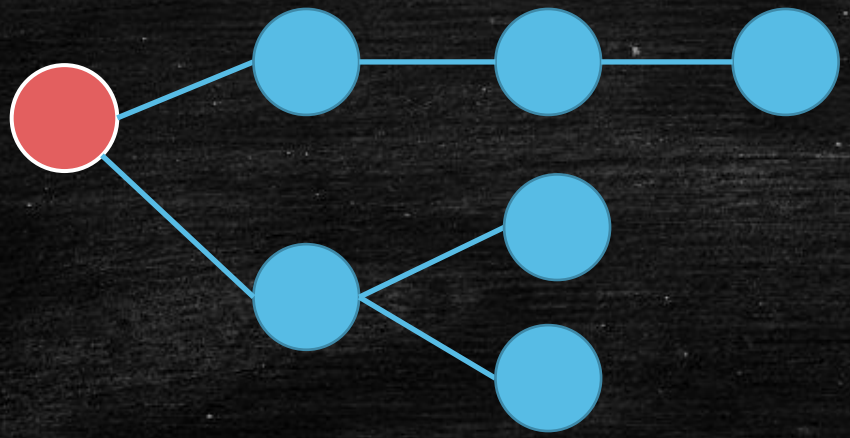
Maximize Independent Set ~~on Trees~~

- Maximize Independent Set ~~on Trees~~ is NP-hard.
- Is the tree special case easier?



Solve it Recursively

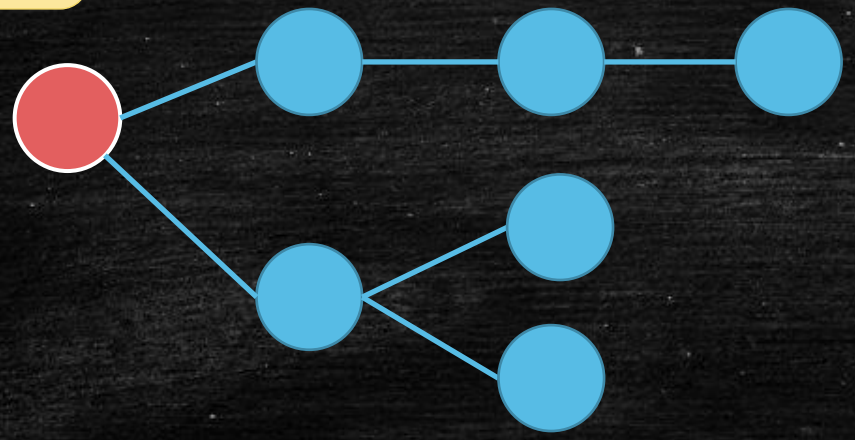
- Start from the root
  - Case 1: We choose the root, what happens?
  - Case 2: We do not choose the root, what happens?



Start from recursive

- Start from the root
  - Case 1: We choose the root, what happens?
  - Case 2: We do not choose the root, what happens?

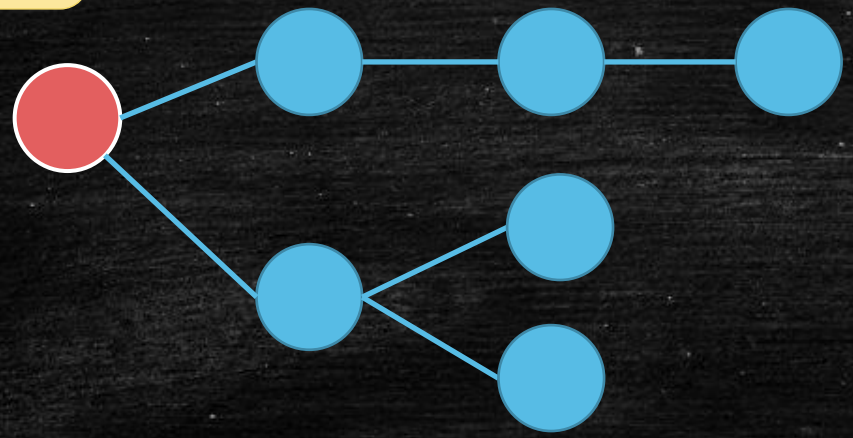
Case 1: We can not choose its children.



Start from recursive

- Start from the root
  - Case 1: We choose the root, what happens?
  - Case 2: We do not choose the root, what happens?

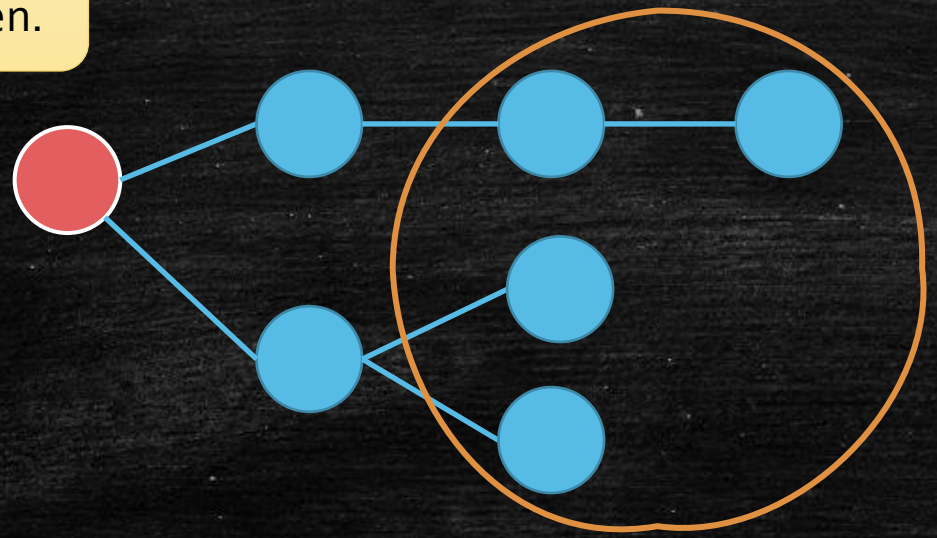
Case 2: We can choose its children.



What subproblems do we need to solve?

Case 1: We can not choose its children.

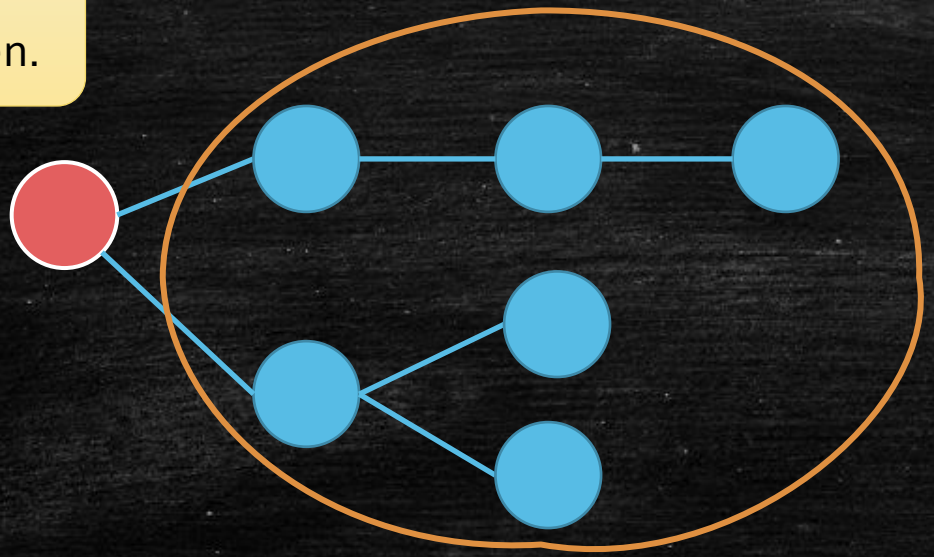
We need to know the max independent set here.



What subproblems do we need to solve?

Case 2: We can choose its children.

We need to know the max independent set here.

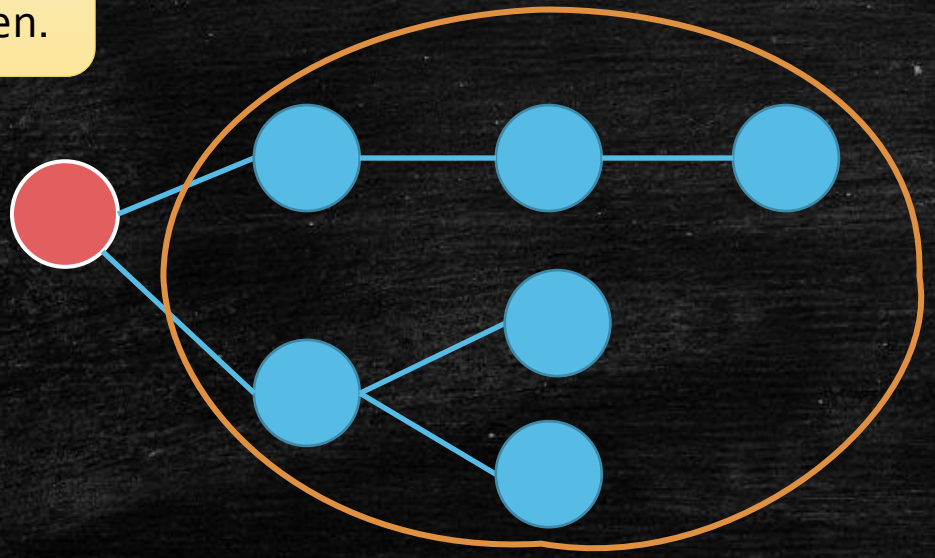


How to define subprobelms?

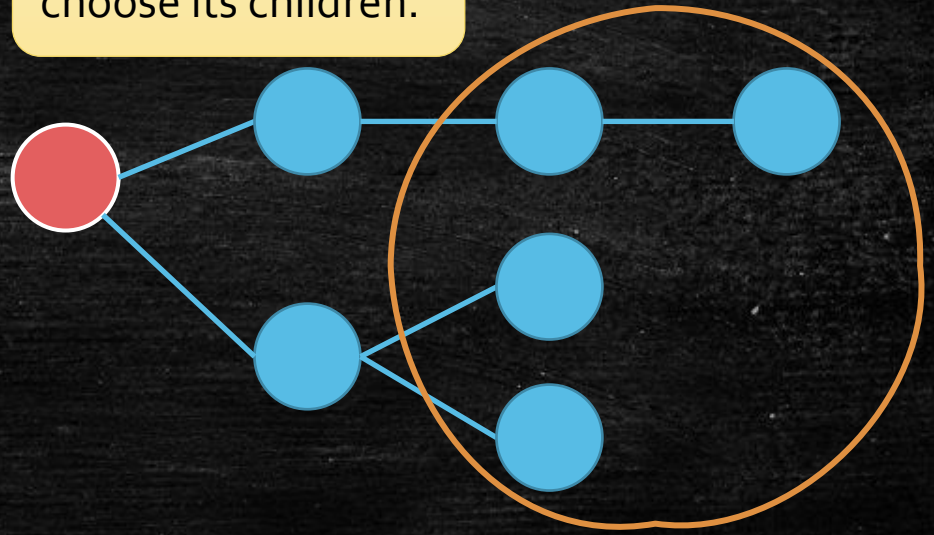
Subproblem Definition

- Subproblem $f[v]$: the maximized size of independent set of the subtree rooted at v.

Case 2: We can choose its children.



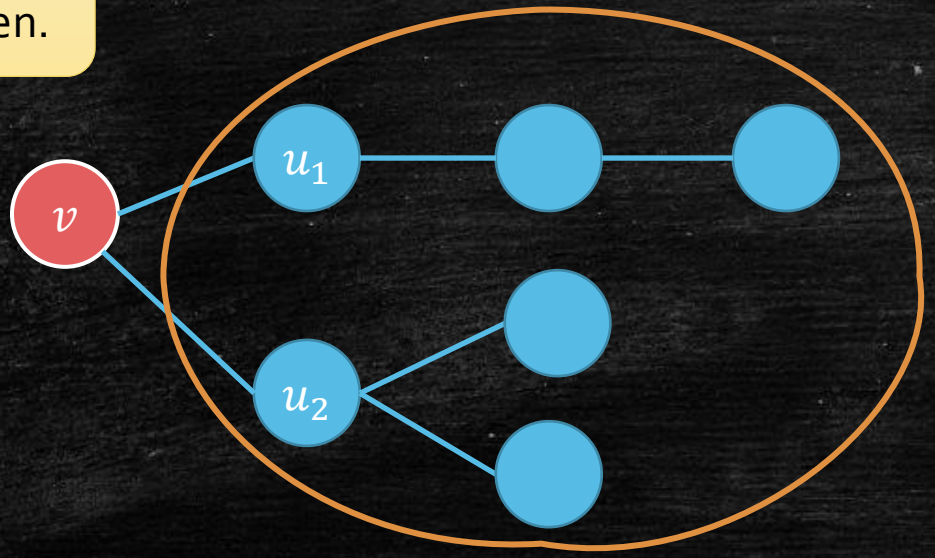
Case 1: We can not choose its children.



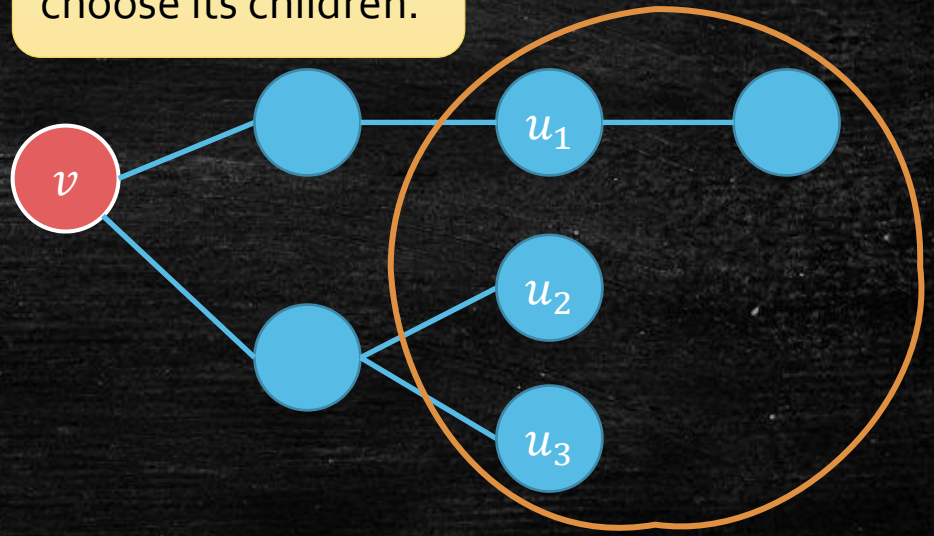
Subproblem Solving

- Subproblem $f[v]$: the maximized size of independent set of the subtree rooted at v.
- $f[v] = \max\{\sum_{u \in children(v)} f[u], \sum_{u \in grandchildren(v)} f[u] + 1\}$

Case 2: We can choose its children.



Case 1: We can not choose its children.



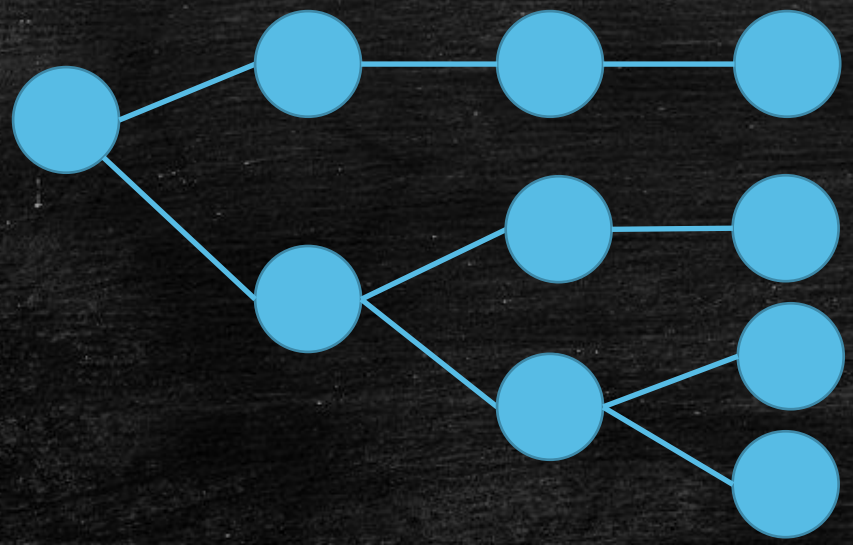
Running Time

- $f[v] = \max\{\sum_{u \in children(v)} f[u], \sum_{u \in grandchildren(v)} f[u] + 1\}$
- Looks $O(n^2)$.
  - We have n subproblems.
  - Each take $O(n)$ times.
- But it is $O(n)$.
  - Each of its children and its grandchildren cost one.
  - On other words, each vertex only need to pay one for its **parent** and one for its **grandparent**.
  - Totally $O(n)$.
  - Question: how to find a bottom-up order?

What is the topological order of the DP?

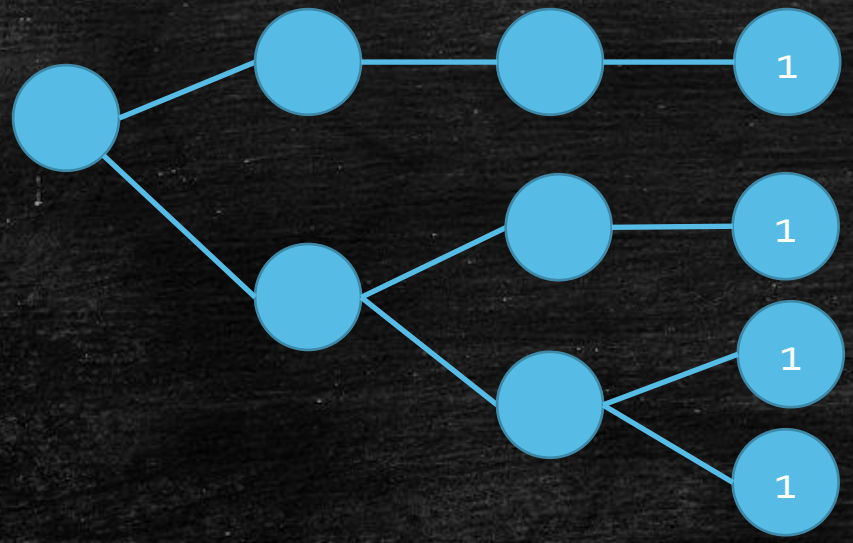
It is also a greedy algorithm!

- Try to solve it bottom-up!



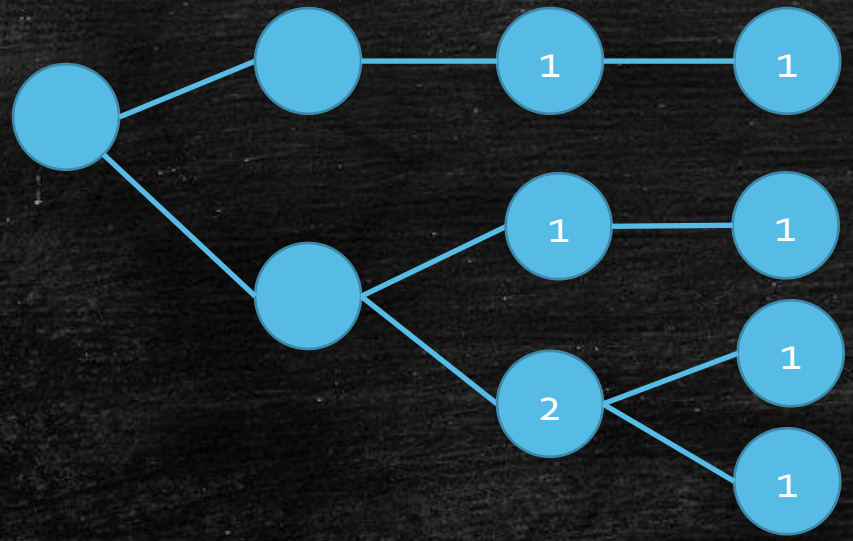
It is also a greedy algorithm!

- Try to solve it bottom-up!



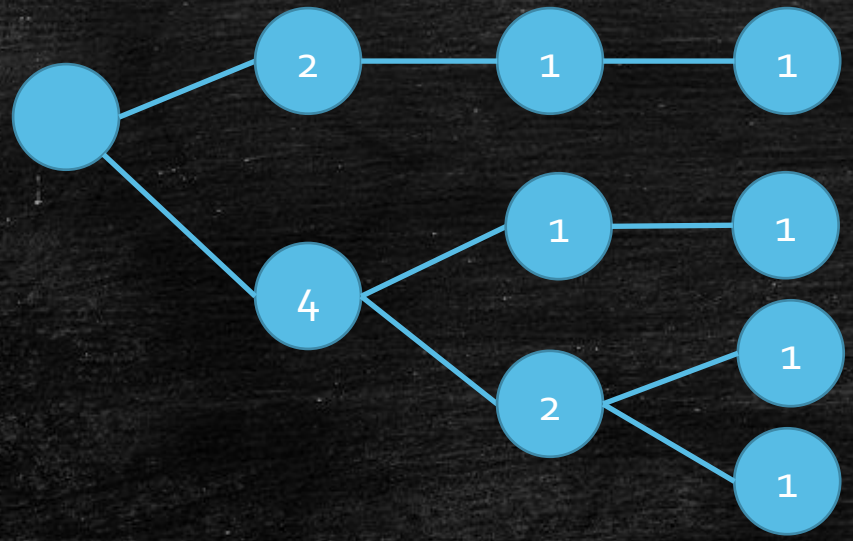
It is also a greedy algorithm!

- Try to solve it bottom-up!



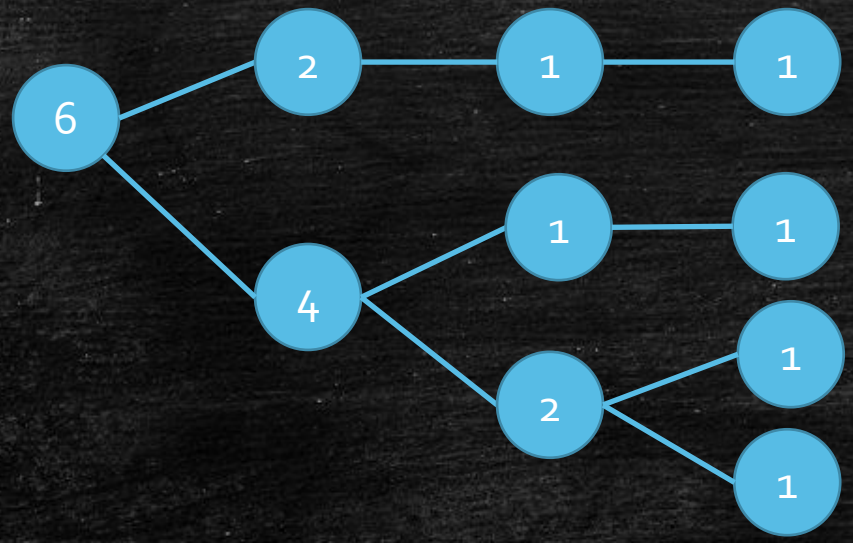
It is also a greedy algorithm!

- Try to solve it bottom-up!



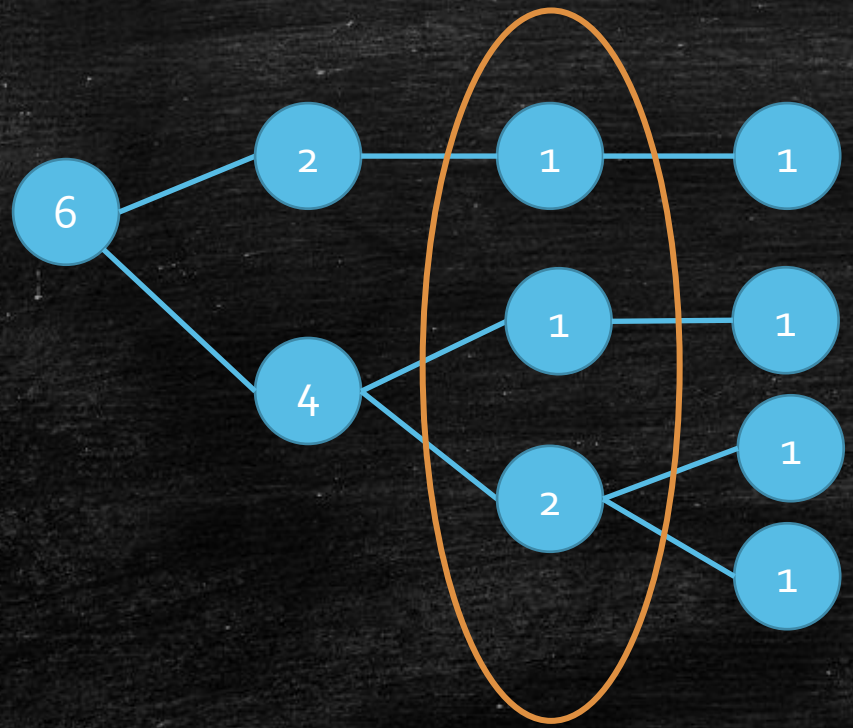
It is also a greedy algorithm!

- Try to solve it bottom-up!



It is also a greedy algorithm!

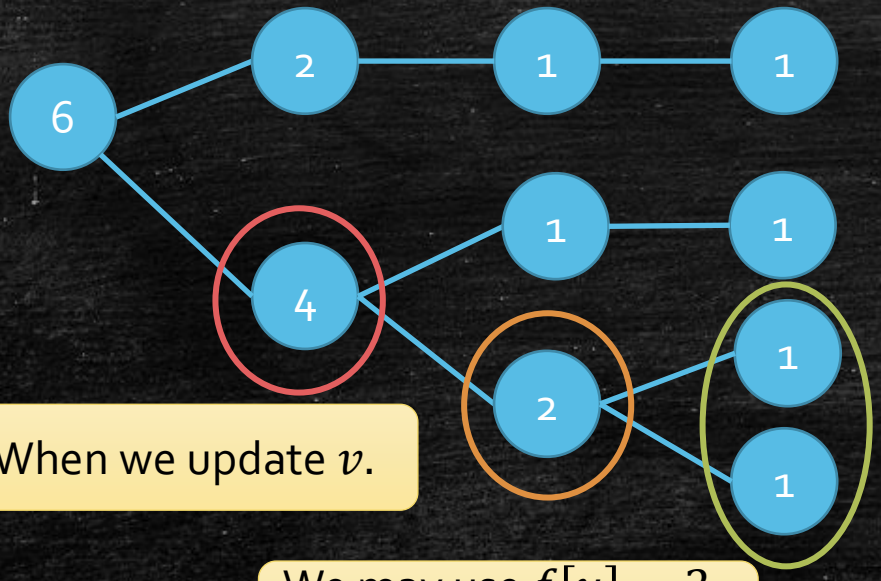
- Try to solve it bottom-up!



They are useless!

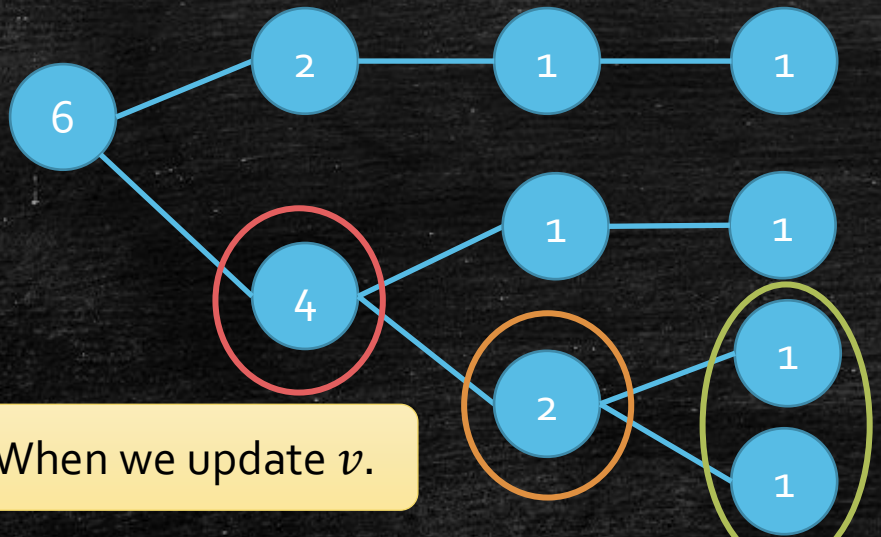
It is also a greedy algorithm!

- Try to solve it bottom-up!



It is also a greedy algorithm!

- Try to solve it bottom-up!
- $\sum_{u \in children(v)} f[u]$



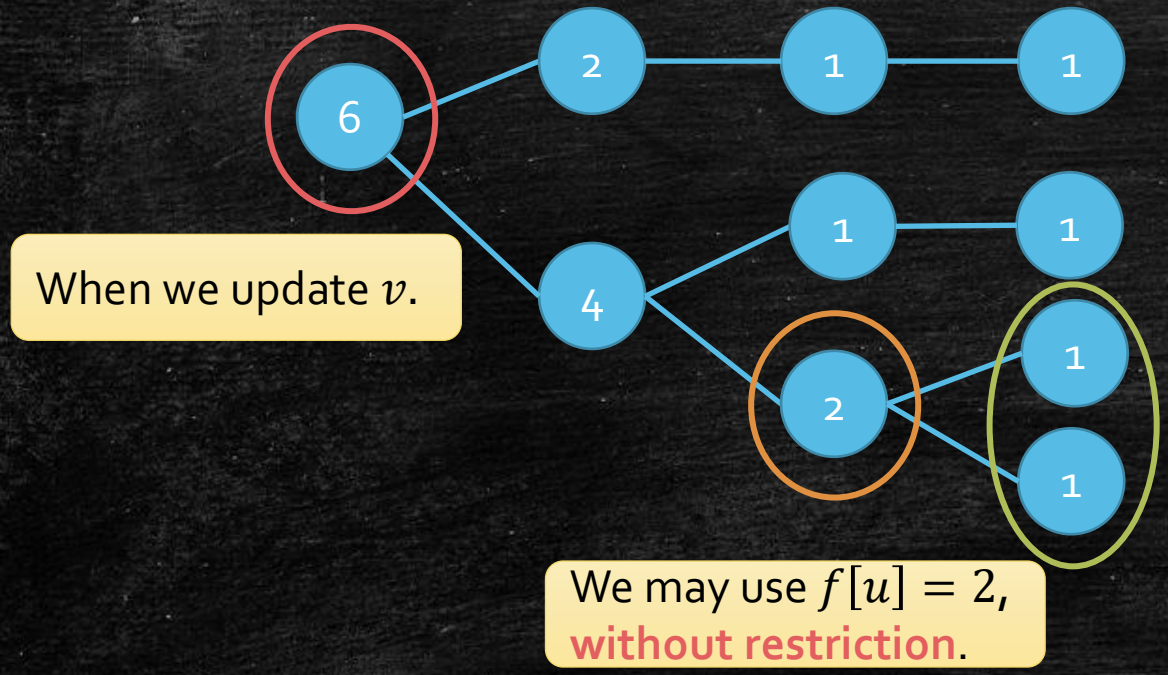
When we update v.

We may use $f[u] = 2$, with restriction.

Why not use $f[u_1] + f[u_2]$ without restriction?

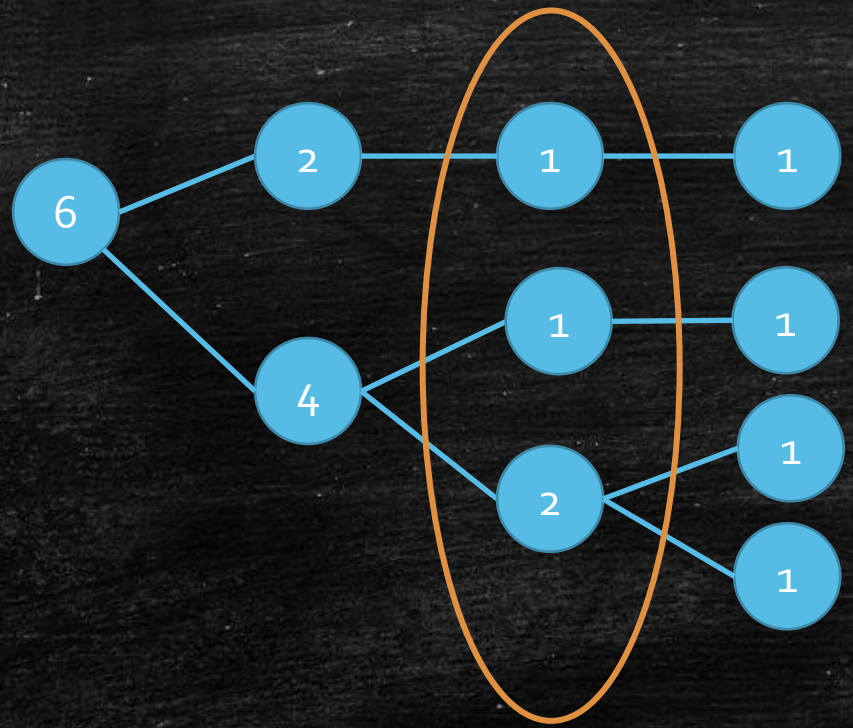
It is also a greedy algorithm!

- Try to solve it bottom-up!
- $\sum_{u \in grandchildren(v)} f[u] + 1$



It is also a greedy algorithm!

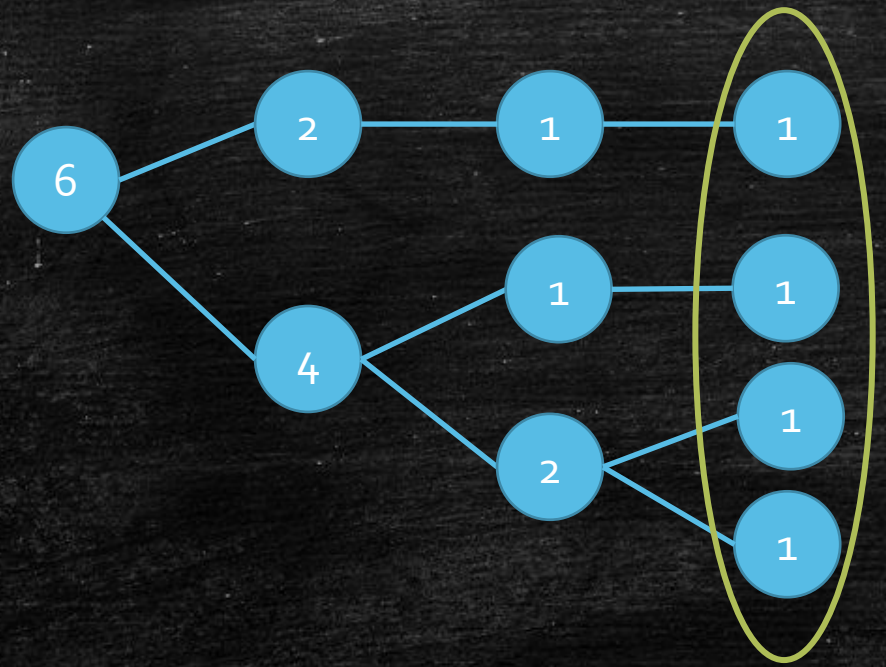
- Try to solve it bottom-up!



They are useless!

It is also a greedy algorithm!

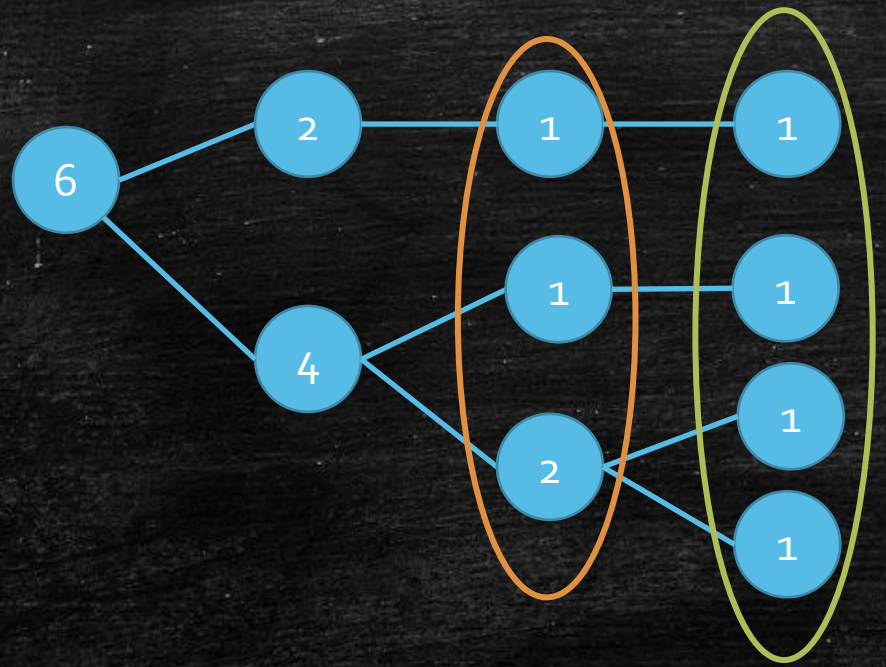
- Try to solve it bottom-up!



They are super useful!
They can update their ancestor **without restriction**!

It is also a greedy algorithm!

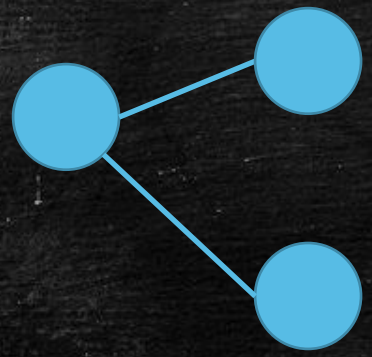
- Try to solve it bottom-up!

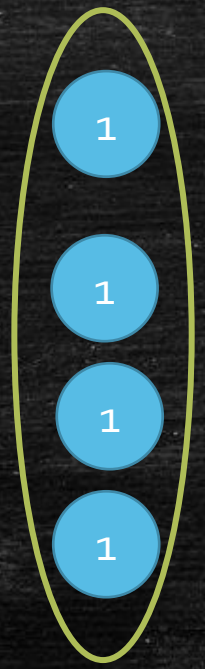


It is same to say, we choose Green and remove Orange.

It is also a greedy algorithm!

- Try to solve it bottom-up!

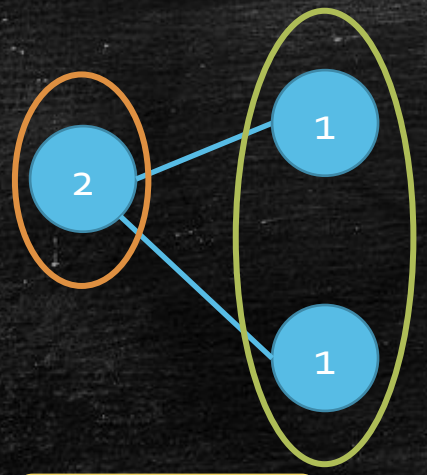




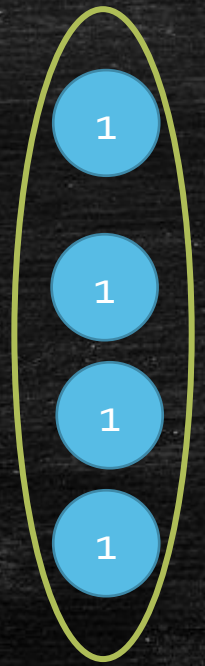
It is same to say, we choose Green and remove Orange.

It is also a greedy algorithm!

- Try to solve it bottom-up!



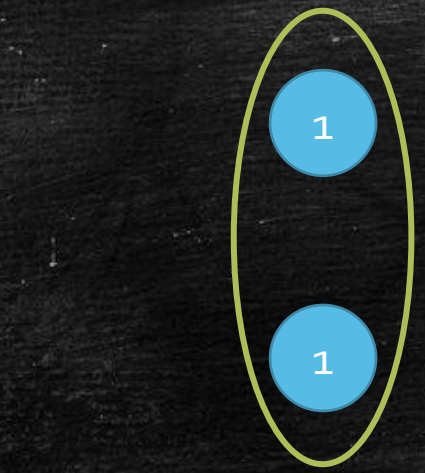
Do it again



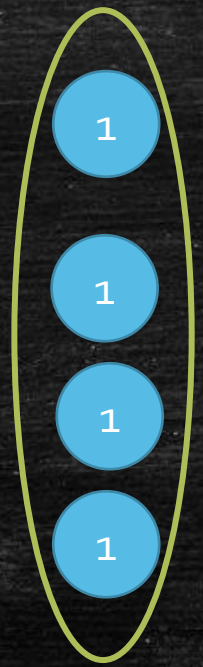
It is same to say, we choose Green and remove Orange.

It is also a greedy algorithm!

- Try to solve it bottom-up!



Do it again



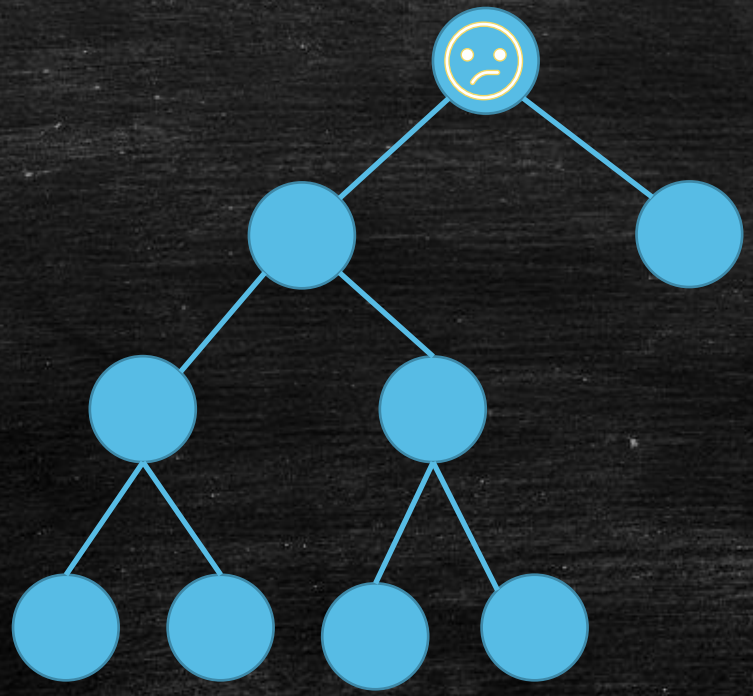
It is same to say, we choose Green and remove Orange.

The Greedy Algorithm

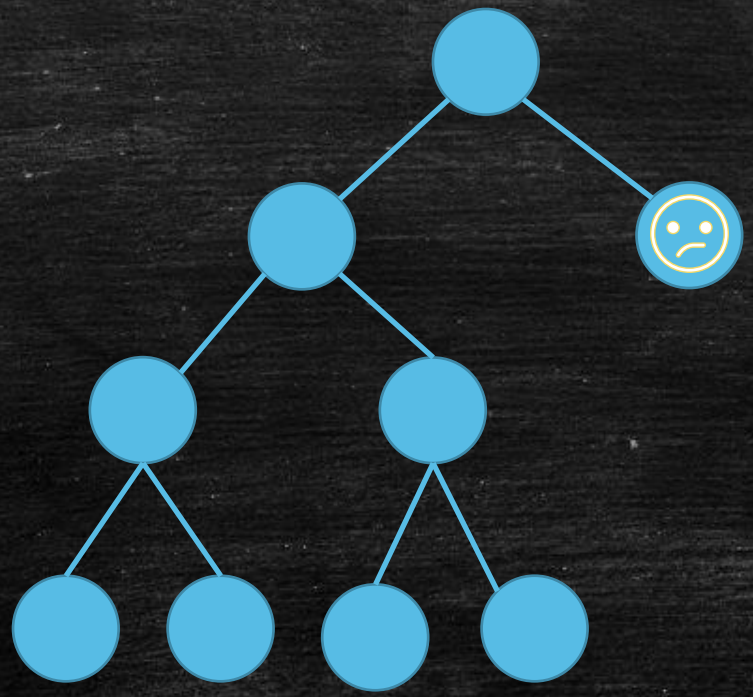
1. Choose all Leaves.
2. Remove all leaves' parents.
3. Repeat 1 again.

- How to implement it in $O(|V|)$?

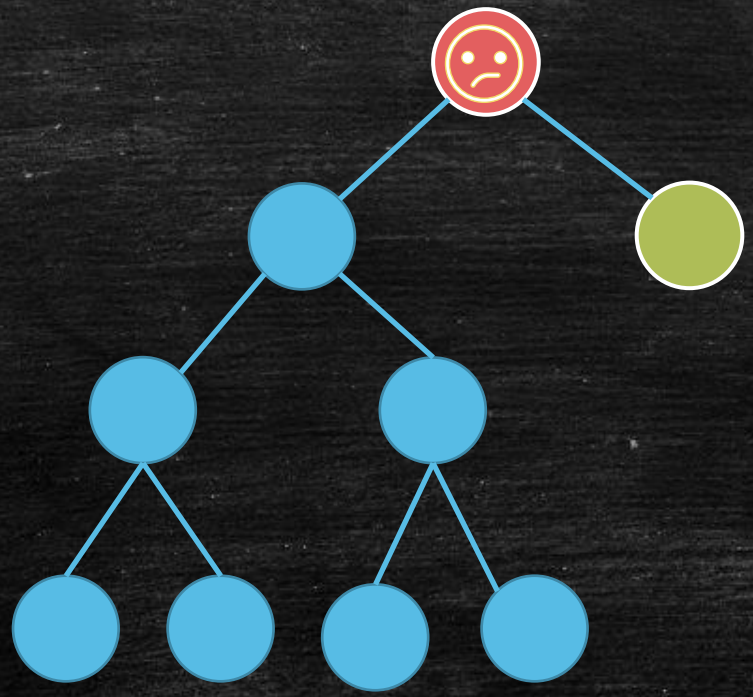
Greedy Algorithm Implementation



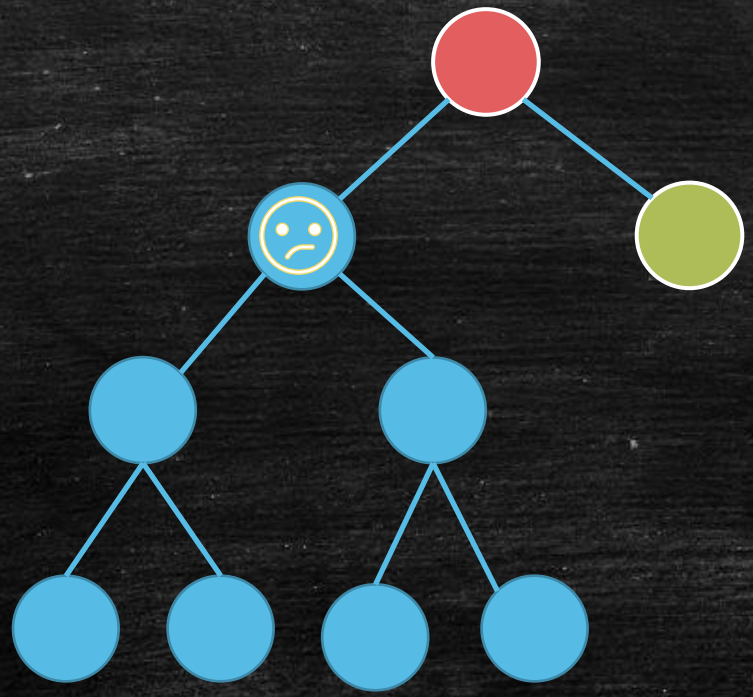
Greedy Algorithm Implementation



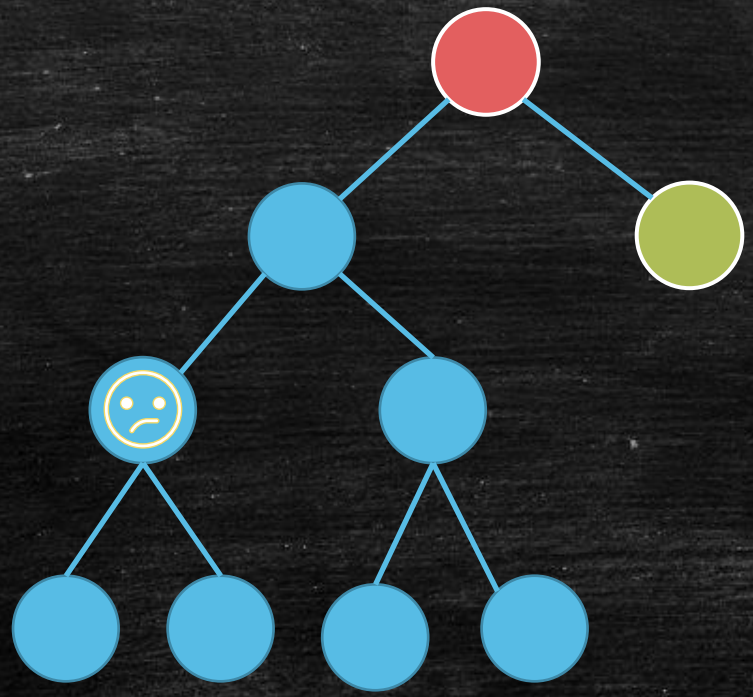
Greedy Algorithm Implementation



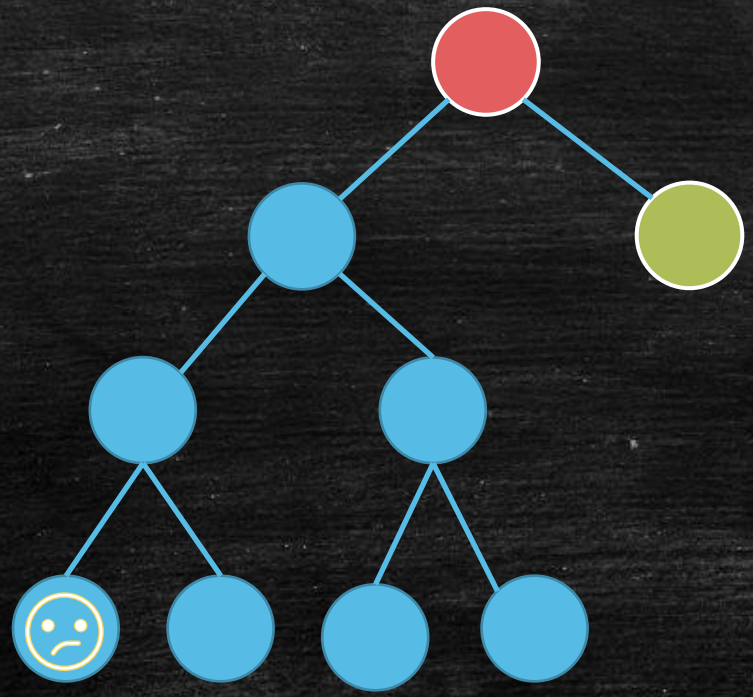
Greedy Algorithm Implementation



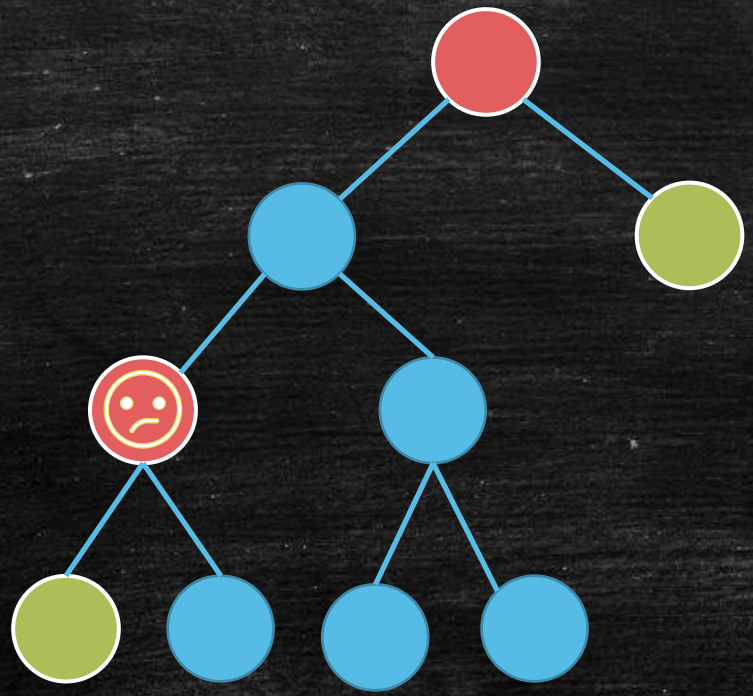
Greedy Algorithm Implementation



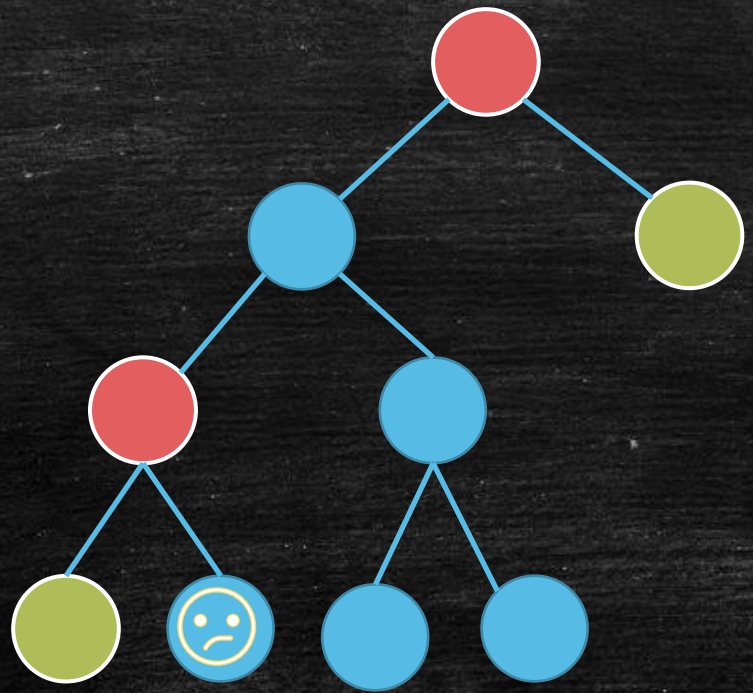
Greedy Algorithm Implementation



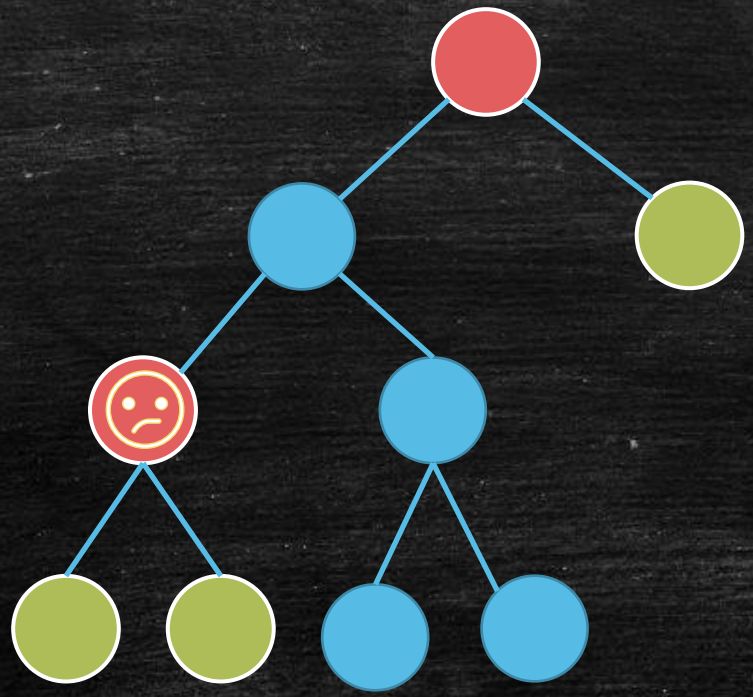
Greedy Algorithm Implementation



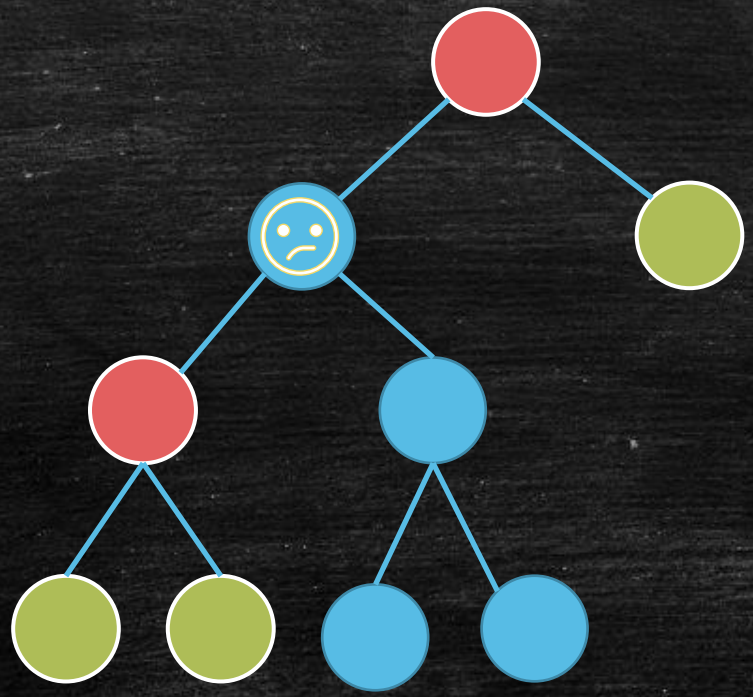
Greedy Algorithm Implementation



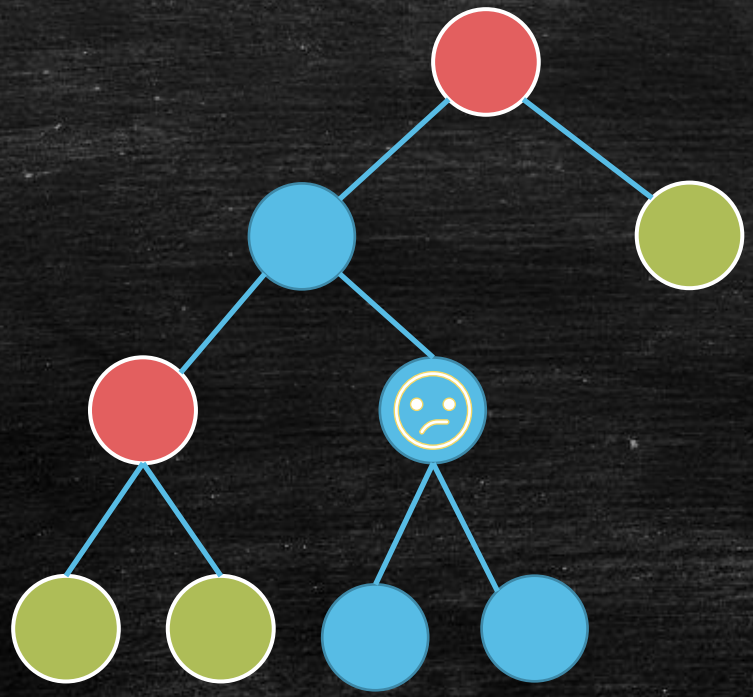
Greedy Algorithm Implementation



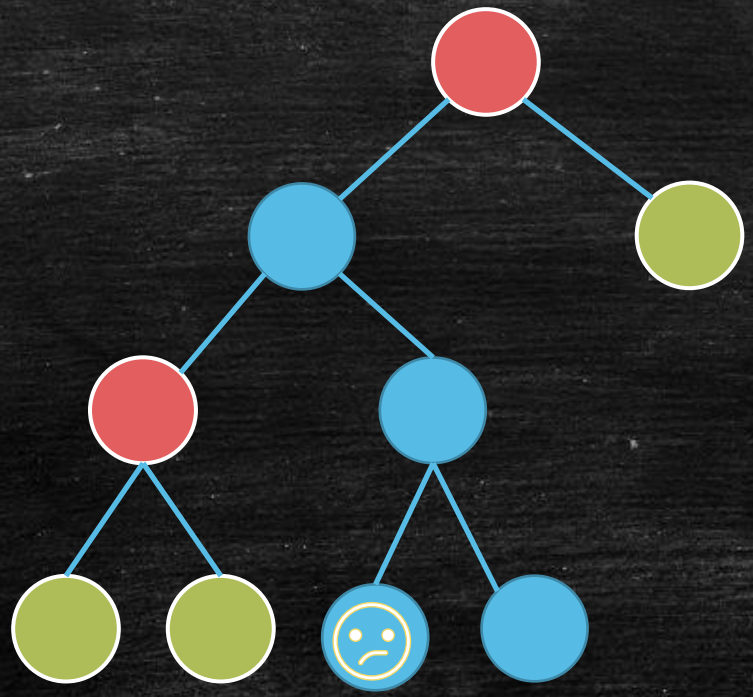
Greedy Algorithm Implementation



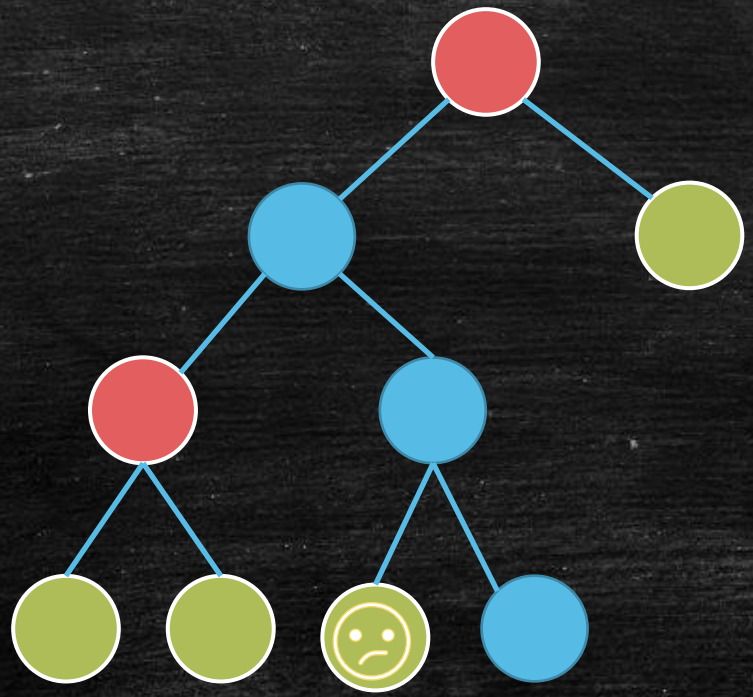
Greedy Algorithm Implementation



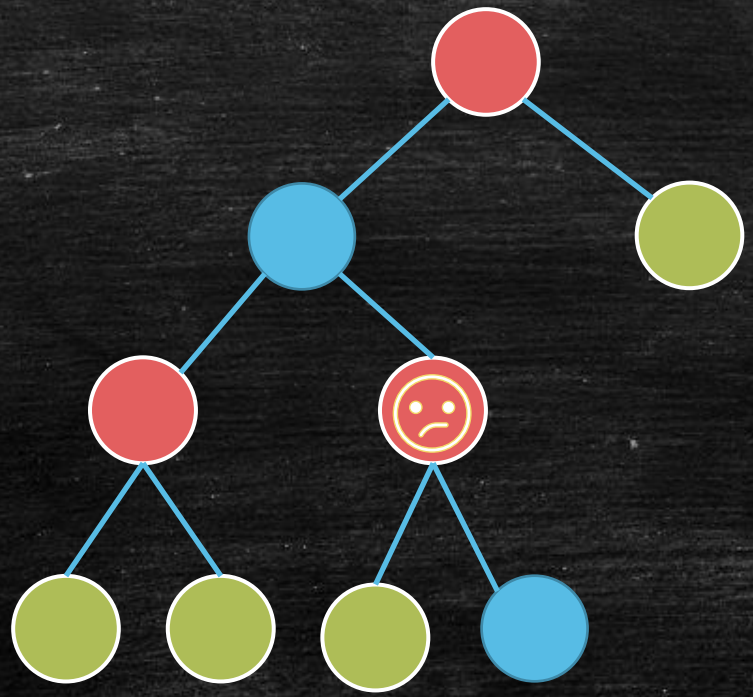
Greedy Algorithm Implementation



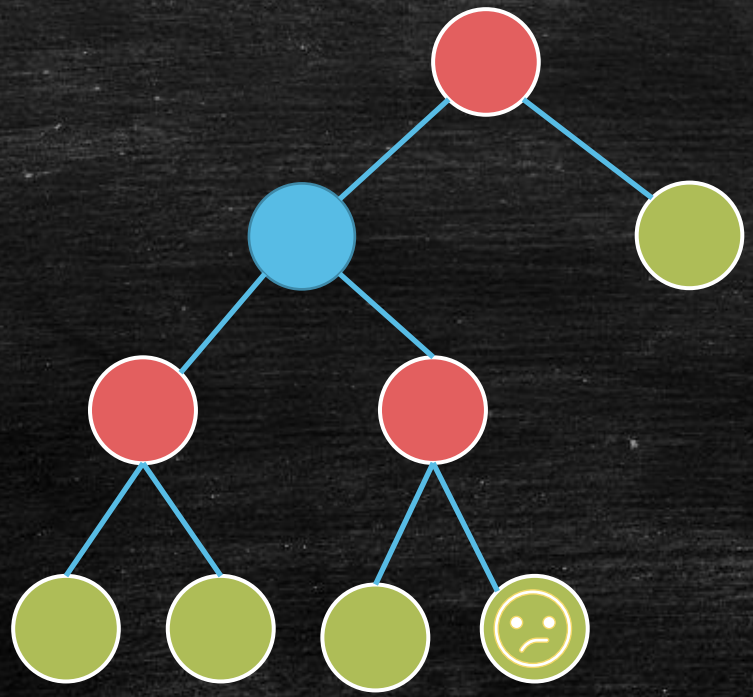
Greedy Algorithm Implementation



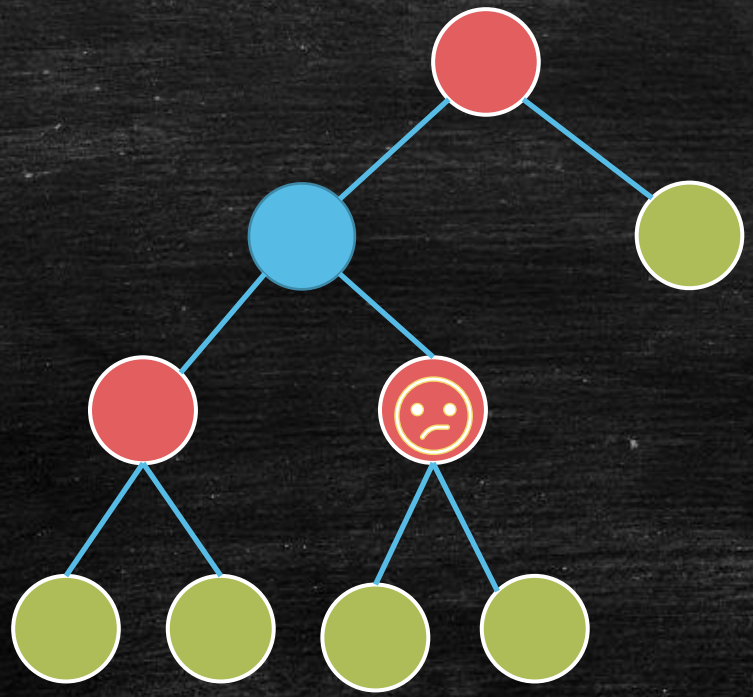
Greedy Algorithm Implementation



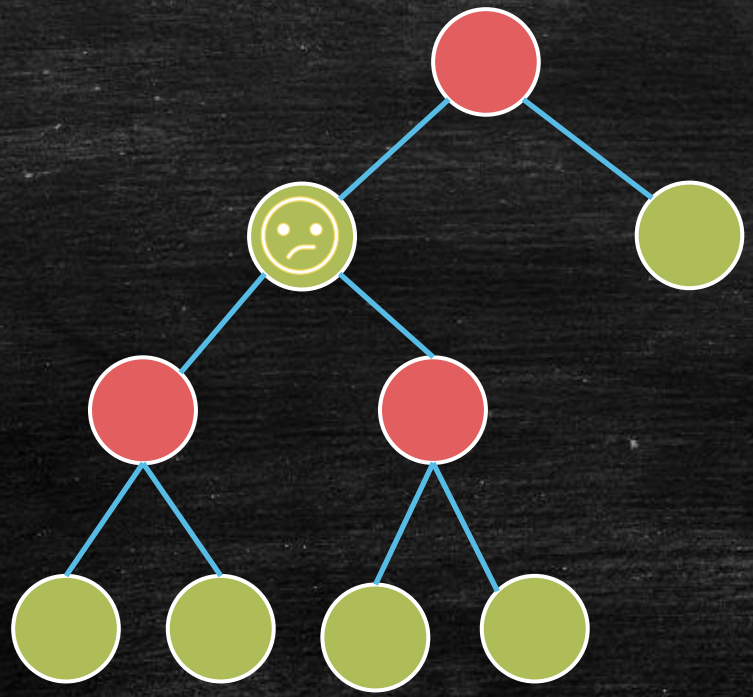
Greedy Algorithm Implementation



Greedy Algorithm Implementation

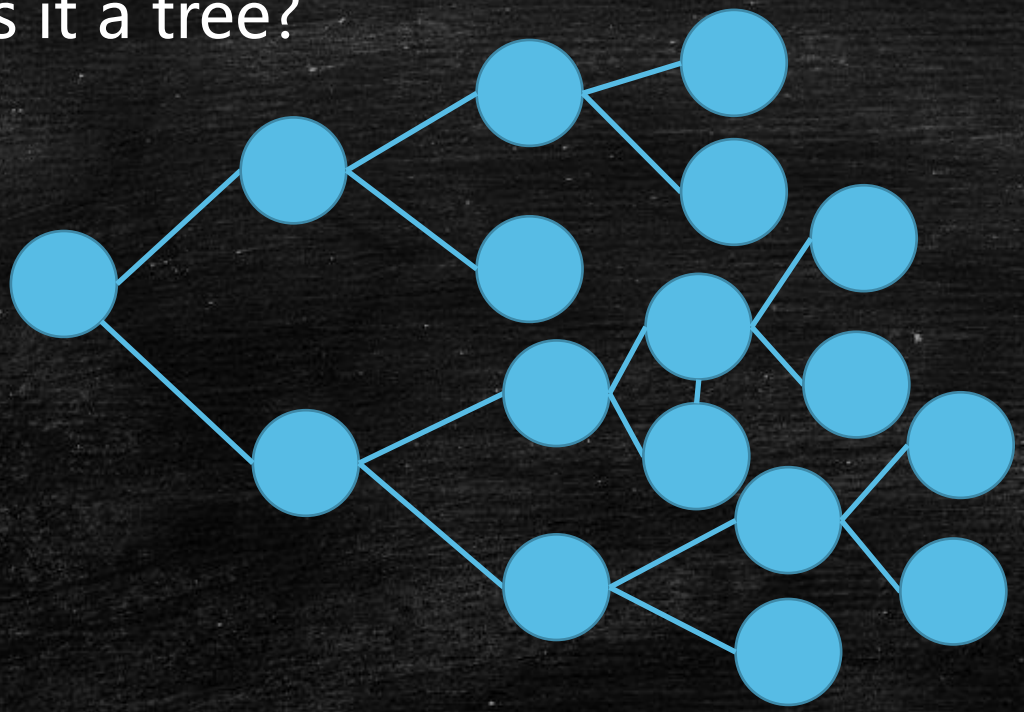


Greedy Algorithm Implementation



Independent Set on General Graphs

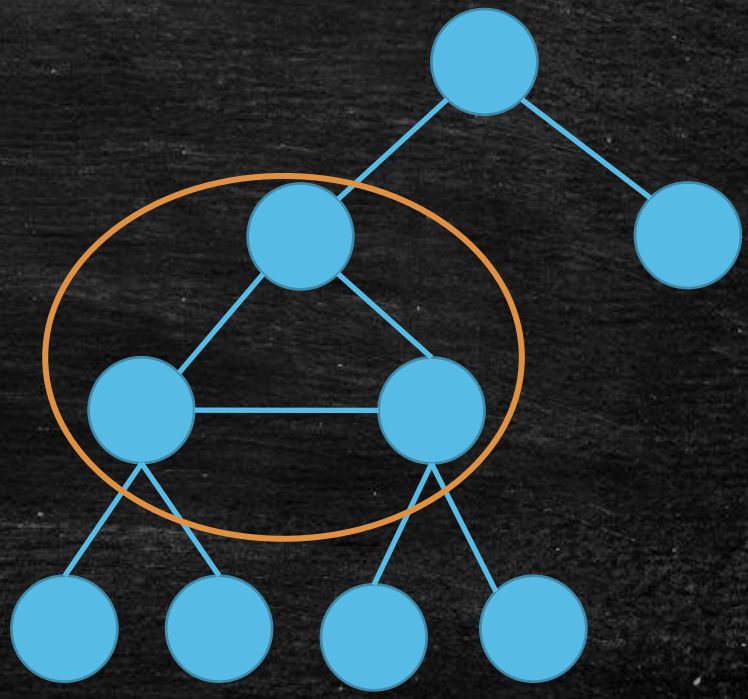
- Look at the following graph.
- Is it a tree?



It looks like a tree!

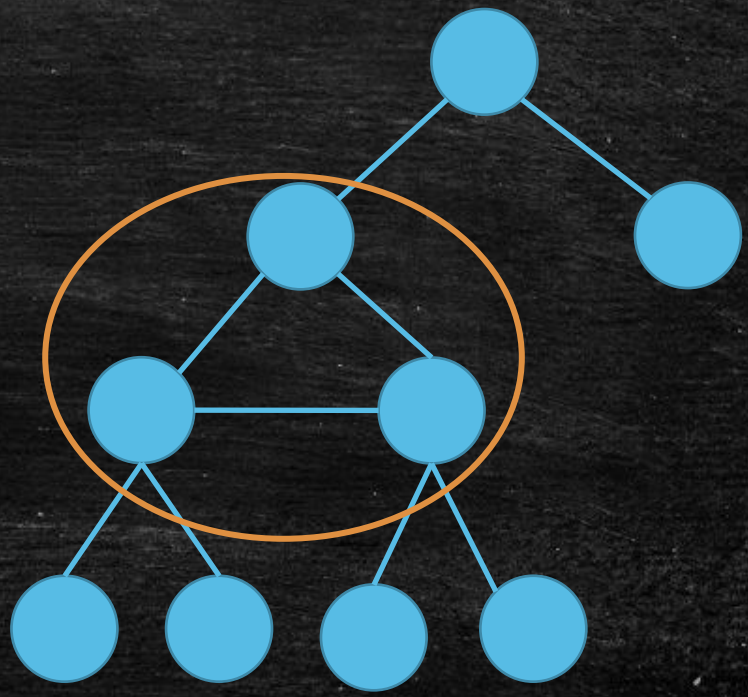
A Tree-like Graph

- It is a tree if we view the triangle as a **super node**.
- Question: Can we still use DP?
- Consider the subtree rooted at the **orange super node**.
  - Case 1: We choose the **super node**, what happens?
  - Case 2: We do not choose the **super node**, what happens?



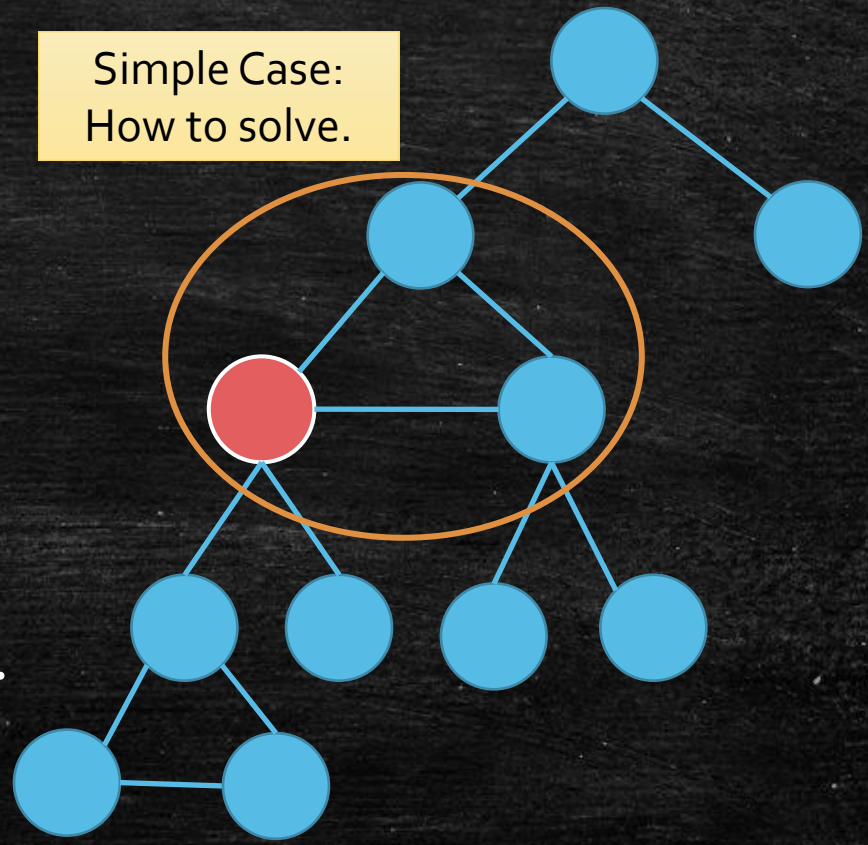
“Choose” a Super Node.

- “Choose” a Super Node.
- How many cases?
- We have at most 2^3 possible way!
- Different way means different restriction for next level selection.



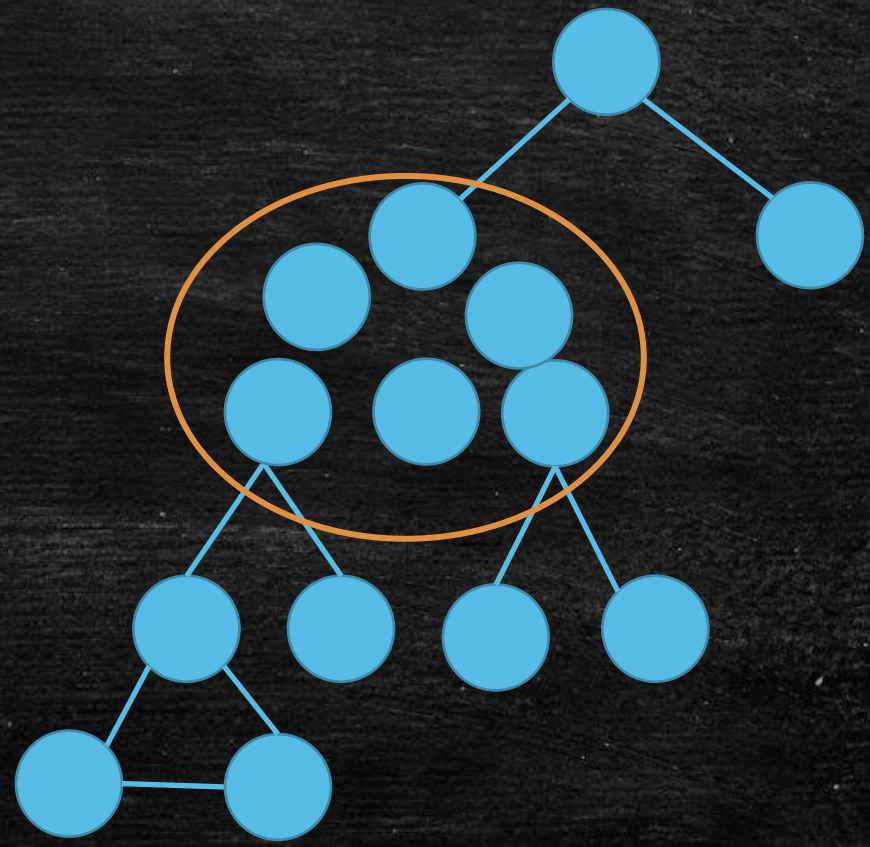
“Choose” a Super Node.

- “Choose” a Super Node.
- How many cases?
- We have at most 2^3 possible way!
- Different way means different restriction for next level selection.
- We can design a DP for $f[i, way]$.
- Time: $O(2^{3+3} \cdot n)$



“Choose” a Super Node.

- What if super node has k vertices?
- We have at most 2^k possible way!
- Time: $O\left(2^{O(k)} \cdot n\right)$
- It hold when the largest super node has k vertices!



Treewidth (Idea)

- The best way to make a graph to tree-like!
- Best: minimize the number of vertices (k) in the largest super node.
- Tree-width: $k - 1$
  - Cycle: 2
  - Clique: n-1
  - Tree: 1
  - series-parallel graphs: ≤ 2
- Many Optimization Problem in these graphs can use tree DP to get $O\left(2^{O(k)} \cdot poly(n)\right)$.

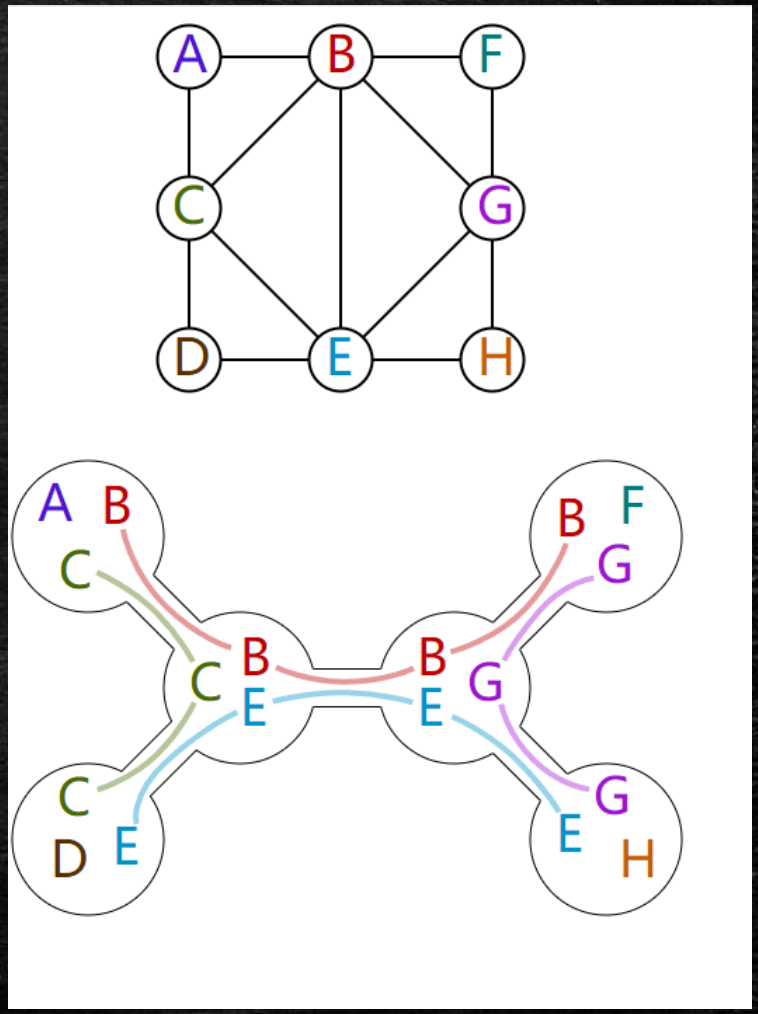
Fixed-Parameter Tractable

- $O(f(k) \cdot n^c)$
- $f(k)$ do not need to be polynomial.
- Many Optimization Problem in these graphs can use tree DP to get $O\left(2^{O(k)} \cdot poly(n)\right)$!
- They are FPT in terms of the treewidth!
- Compare to Approximation Algorithms!

Tree Decomposition and Treewidth

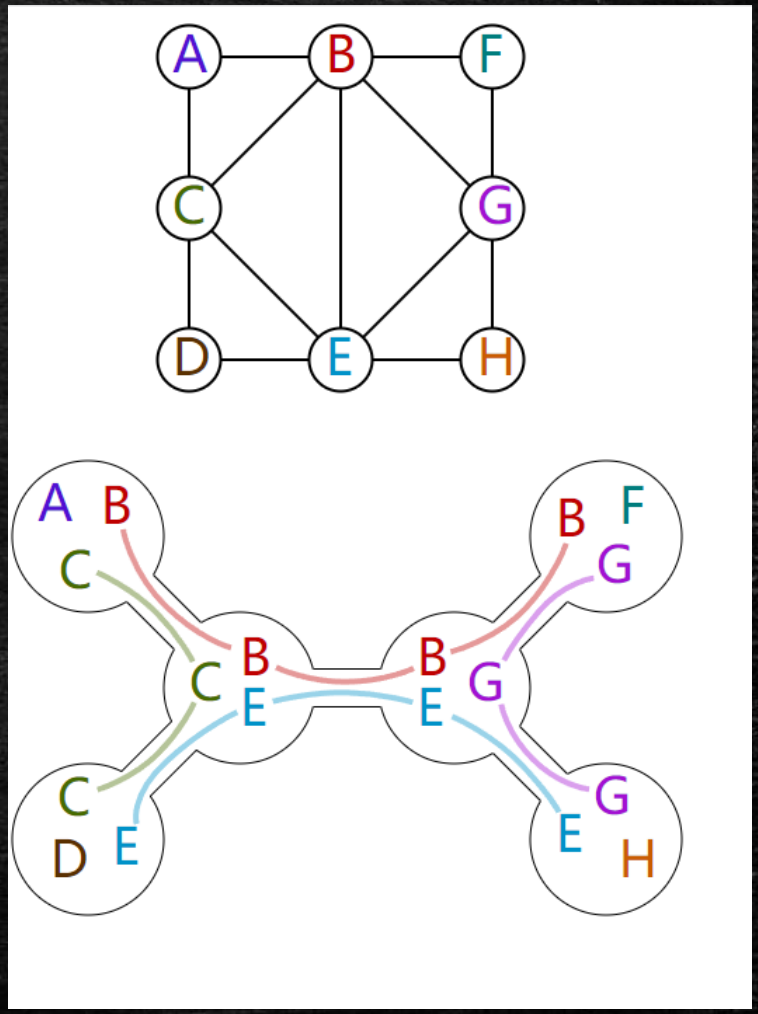
- A tree decomposition
- A Tree node: a **Bag** of vertices.
- Requirement
  - Any vertex must be included in one **Bag**.
  - The **Bags** contains u are connected.
  - An edge (u, v): A bag contains both u and v.

- Treewidth: Size of largest bag – 1.



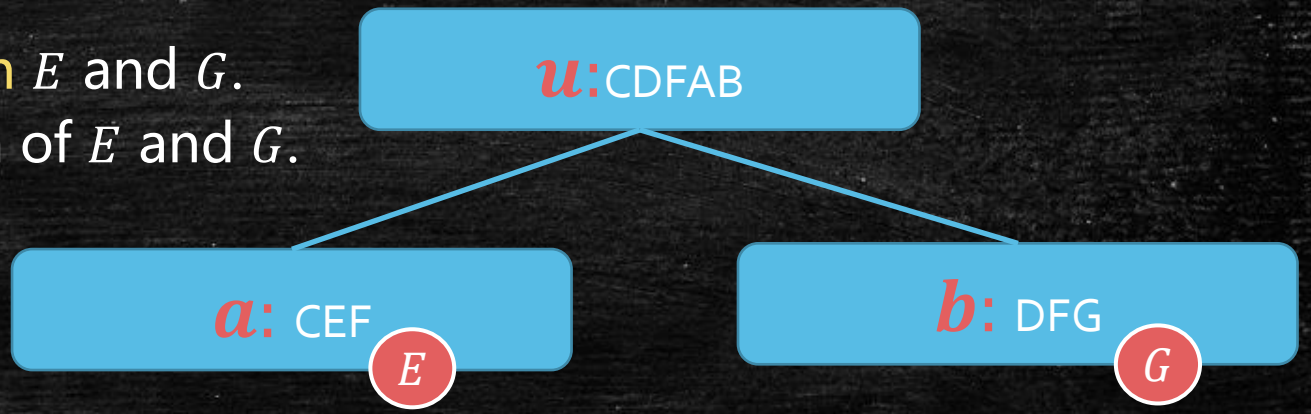
Nice Tree Decomposition

- Number of **Bags** at most $4n$.
- Binary Tree.
- **Theorem:** If G has a tree decomposition with treewidth k, then it admits a nice tree decomposition with treewidth k.
- It is enough to remember we are talking about a tree with size $O(n)$.

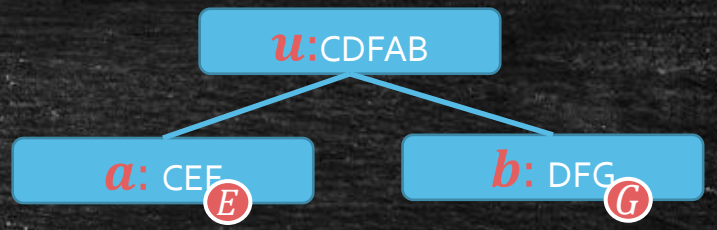


Separation Property

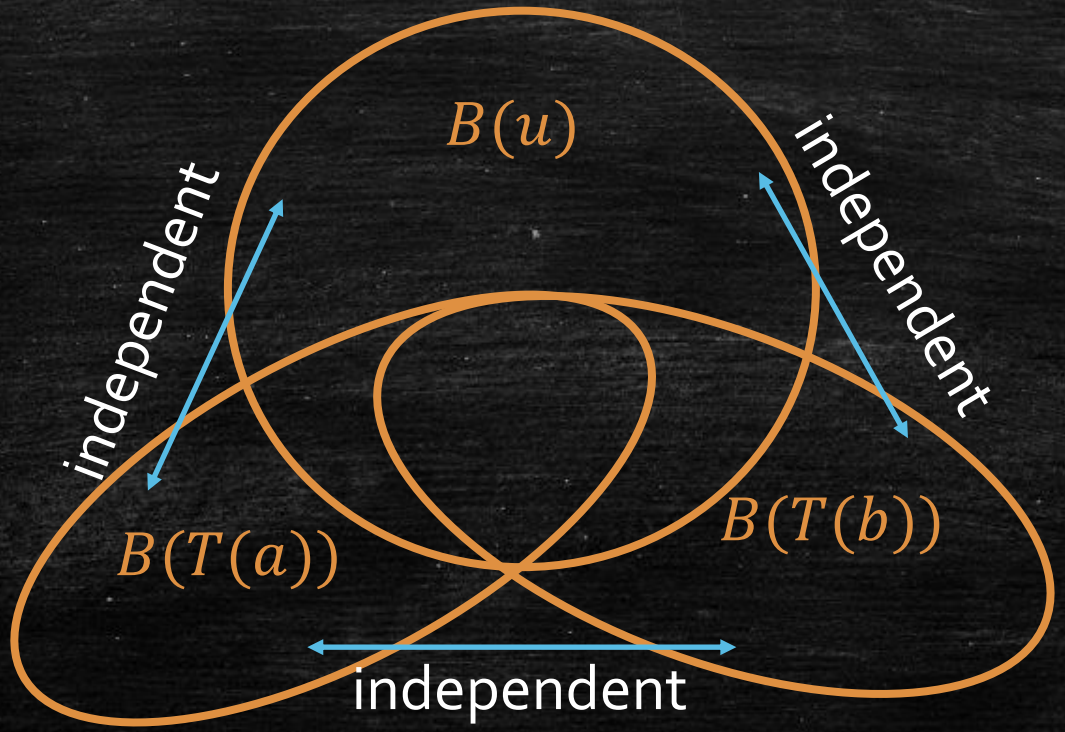
- $B(u)$ separates the two subtree $T(a)$ and $T(b)$.
- $B\big(T(a)\big) - B(u)$ and $B\big(T(b)\big) - B(u)$ are independent!
- Notation: $B(u), T(u), B(T(u))$.
- Example:
  - If (E, G) is an edge.
  - There is a Bag contains both E and G.
  - But u blocks the connection of E and G.



Separation Property

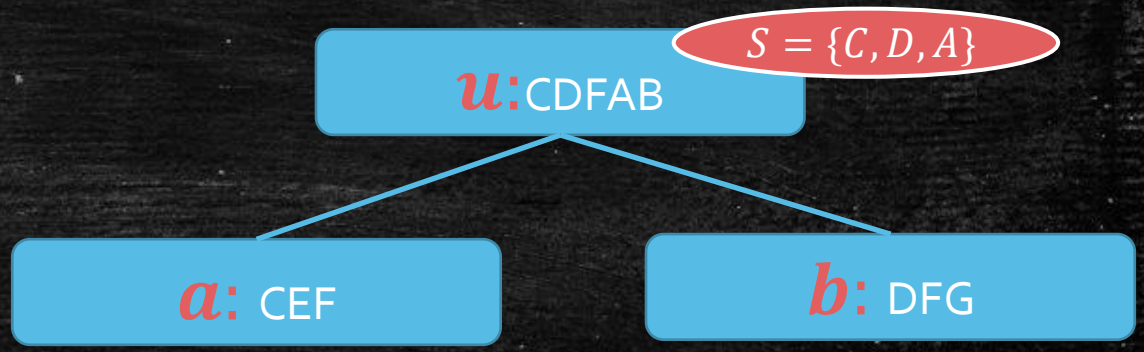


- $B(u)$ separates the two subtree $T(a)$ and $T(b)$.
- $B\big(T(a)\big) - B(u)$ and $B\big(T(b)\big) - B(u)$ and are independent!



DP on tree decomposition

- Solve max independent set on a graph with treewidth k.
- Subproblem $f(S, u)$: The size of max independent set $I \subset B(T(u))$, where $I \cap B(u) = S$.
- $x \in S \rightarrow choosed; x \in B(u) - S \rightarrow not\ choosed$.
  - $C \in I$
  - $D \in I$
  - $A \in I$
  - $F \notin I$
  - $B \notin I$

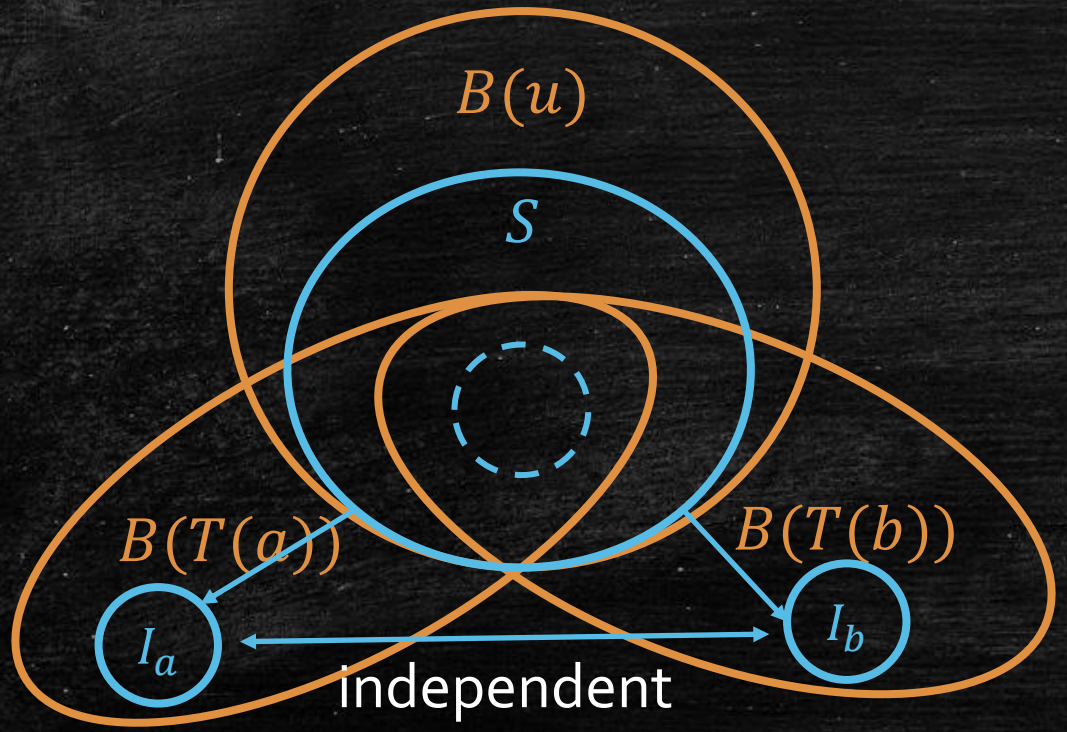


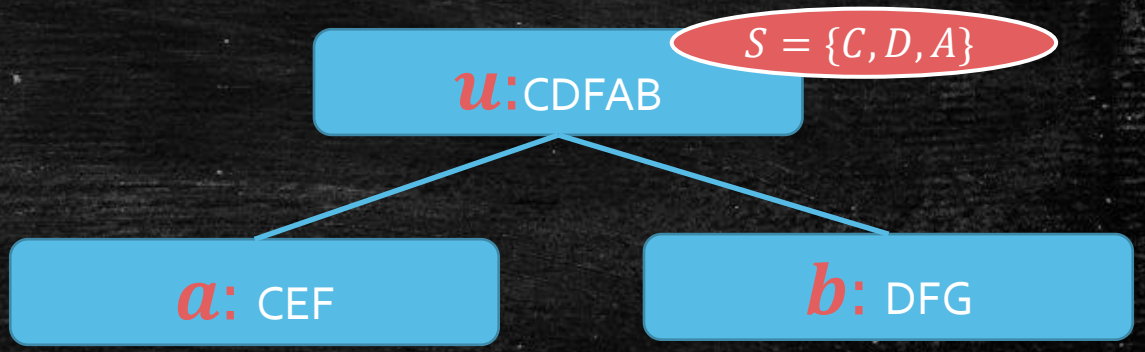
DP on tree decomposition

- Solve max independent set on a graph with treewidth k.
- $f(S, u)$: The size of max independent set $I \subset B(T(u))$, where $I \cup B(u) = S$.
- How to solve $f(S, u)$?
- Bottom-up!
- Base Case (Leaves)
  - $f(S, u) = |S|$ if $|S|$ is independent.
  - $f(S, u) = 0$ if $|S|$ is not independent.
- Output: the largest $f(S, r)$ for root r!

DP on tree decomposition

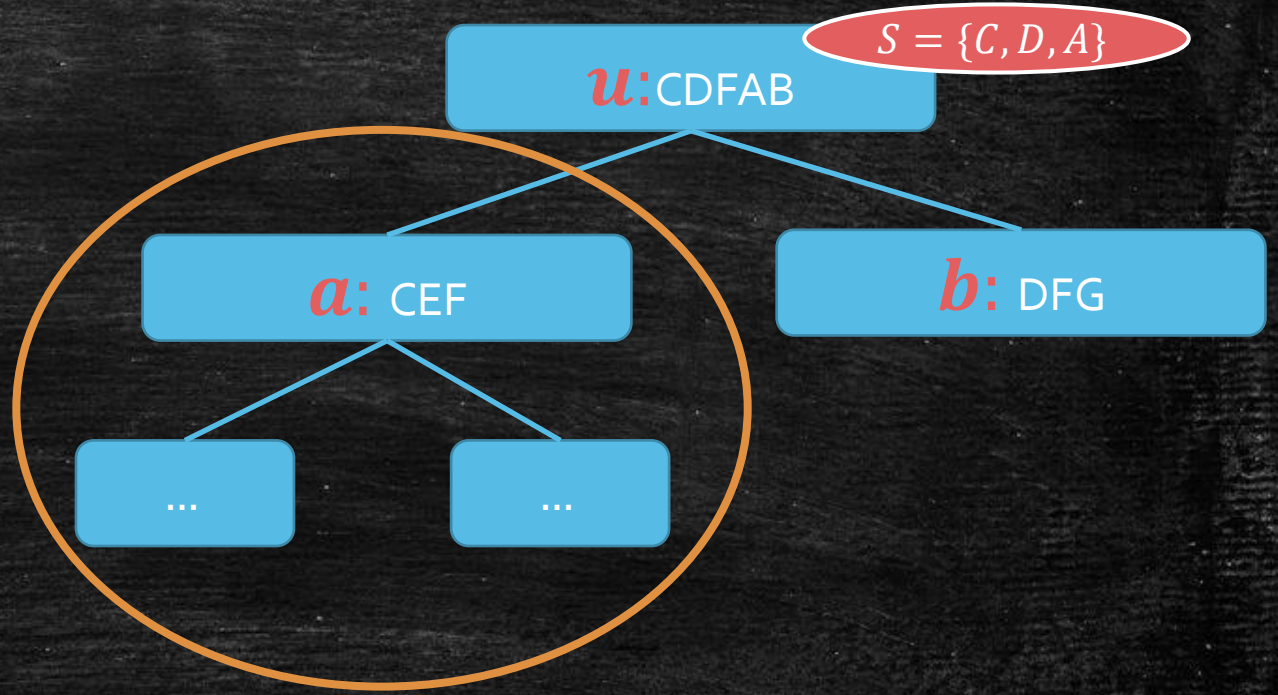
- How to solve $f(S, u)$?
  - If $|S|$ is not independent → $f(S, u) = 0$.
  - How to grow S as large as possible?

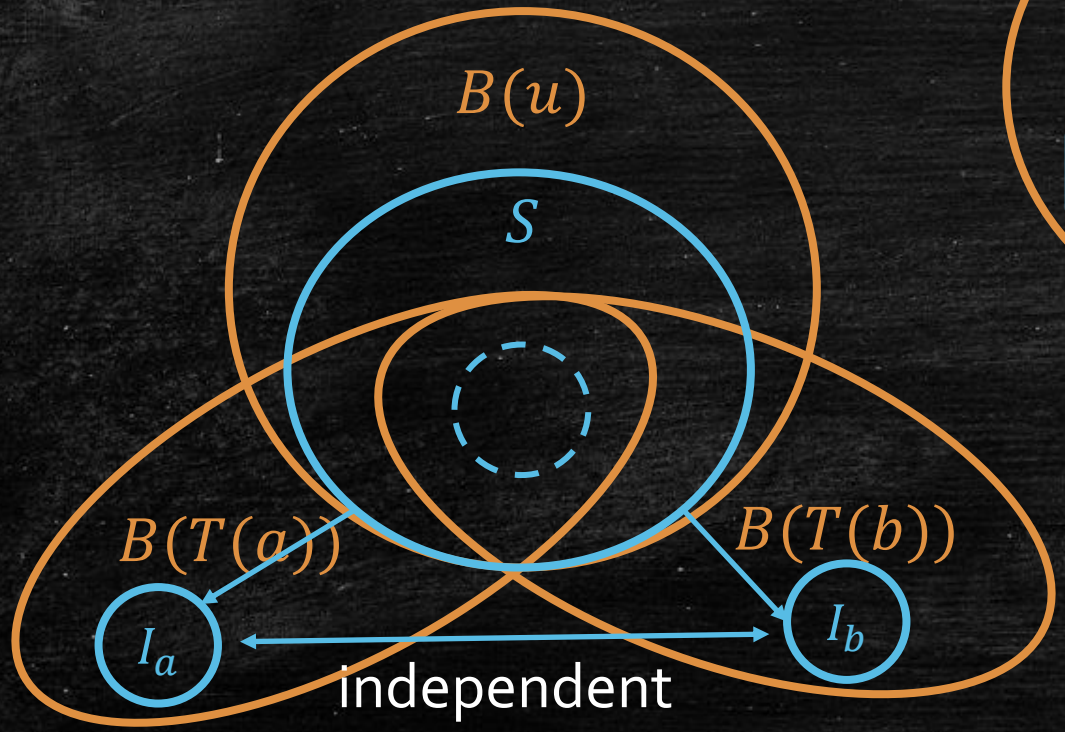




Grow in $T(a)$ first.

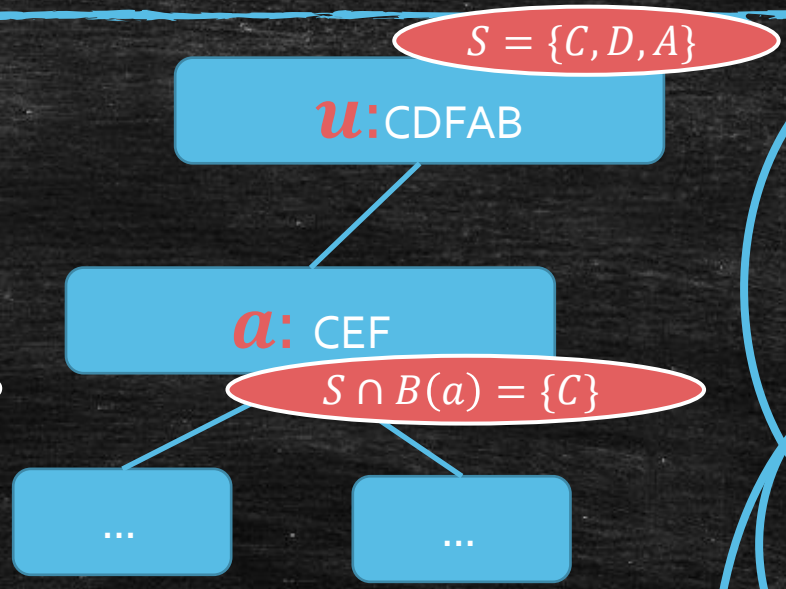
- How large I_a we can get?

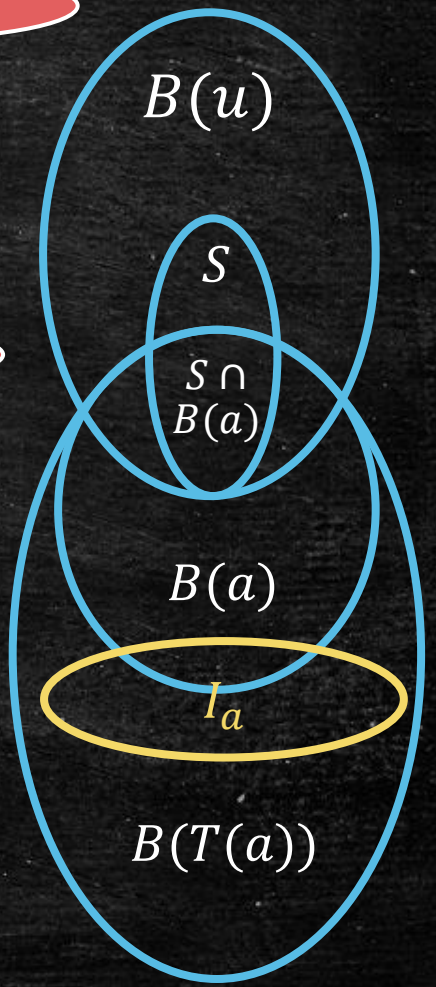




Question: How to calculate?

- How large I_a we can get?
- Requirement
  - I_a is independent with S.
  - ? I_a is independent with $S - B(a)$?
  - ✓ I_a is independent with $S \cap B(a)$?

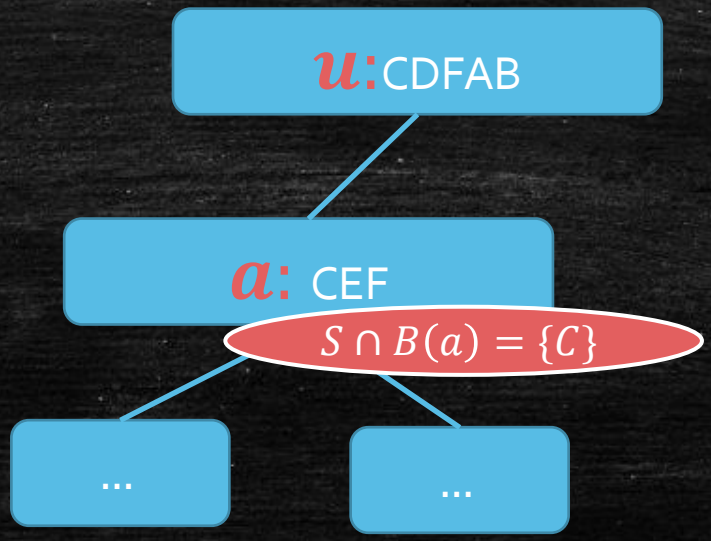


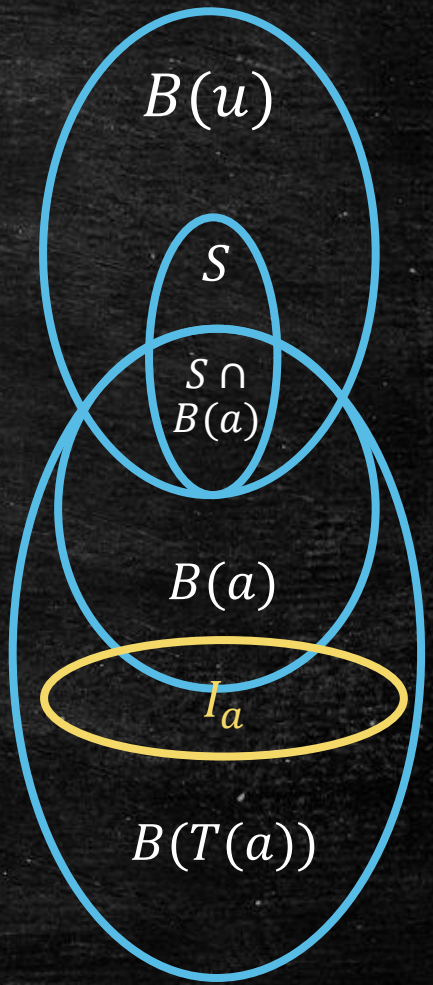


- Find a maximized independent set growing from $S \cap B(a) \rightarrow$ maximized $I_a \cup (S \cap B(a))$.
- Question: is it solved in previous subproblems?
  - Candidate: $f(S \cap B(a), a)$

Question: How to calculate?

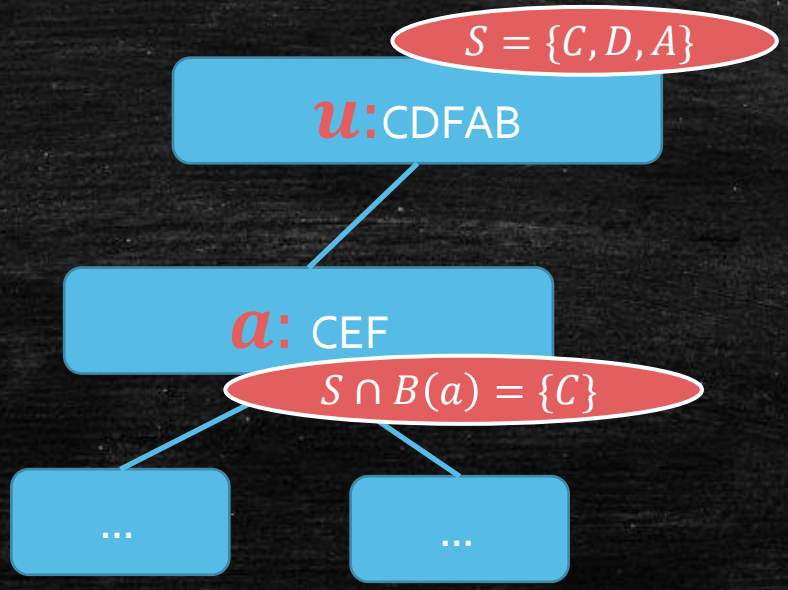
- Find a maximized independent set growing from $S \cap B(a) \rightarrow$ maximized $I_a \cup S \cap B(a)$.
- Question: is it solved in previous subproblems?
- Cases of a
  - Choose $\{C\}$ in $B(a)$.
  - Choose $\{C, E\}$ in $B(a)$.
  - Choose $\{C, F\}$ in $B(a)$.
  - Choose $\{E\}$ in $B(a)$.
  - Choose $\{C, E, F\}$ in $B(a)$.

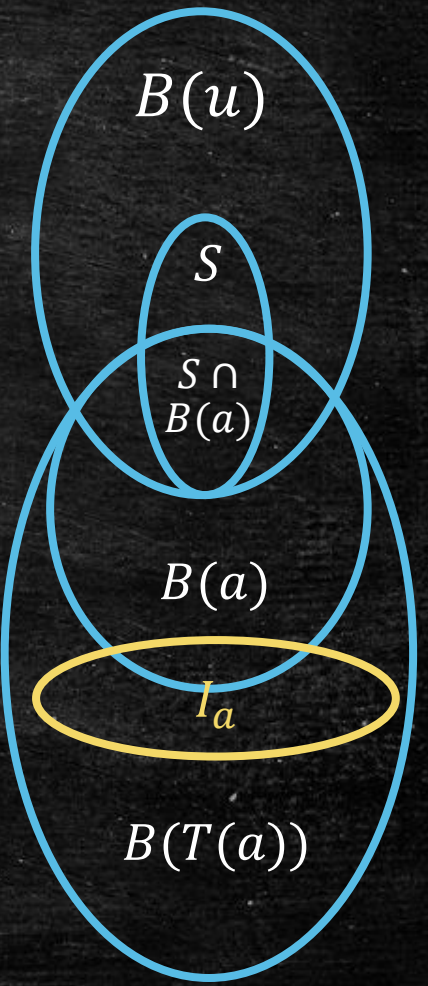




Question: How to calculate?

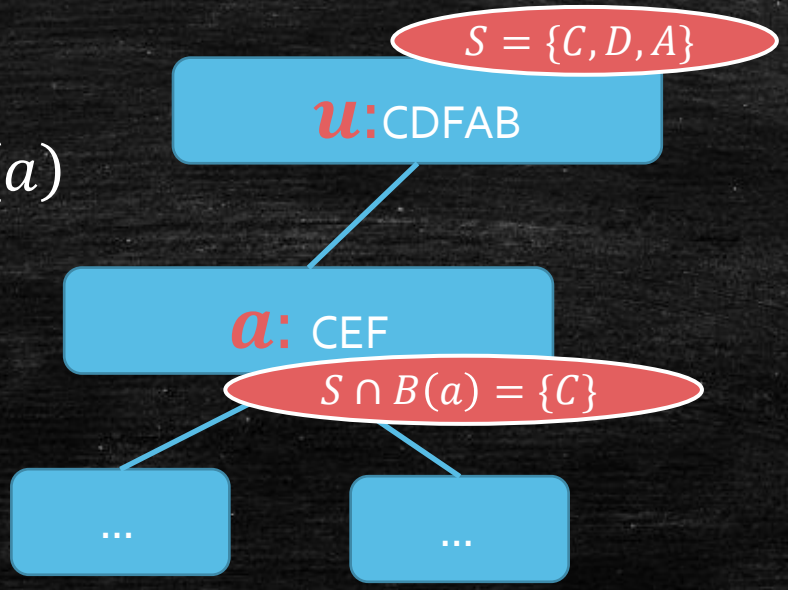
- Find a maximized independent set growing from $S \cap B(a) \rightarrow$ maximized $I_a \cup S \cap B(a)$.
- Question: is it solved in previous subproblems?
- Cases of a
  - Choose $\{C\}$ in $B(a)$.
  - Choose $\{C, E\}$ in $B(a)$.
  - Choose $\{C, F\}$ in $B(a)$.
  - Choose $\{E\}$ in $B(a)$.
  - Choose $\{C, E, F\}$ in $B(a)$.
- Choices of a
  - $S' \cap B(u) = S \cap B(a)$

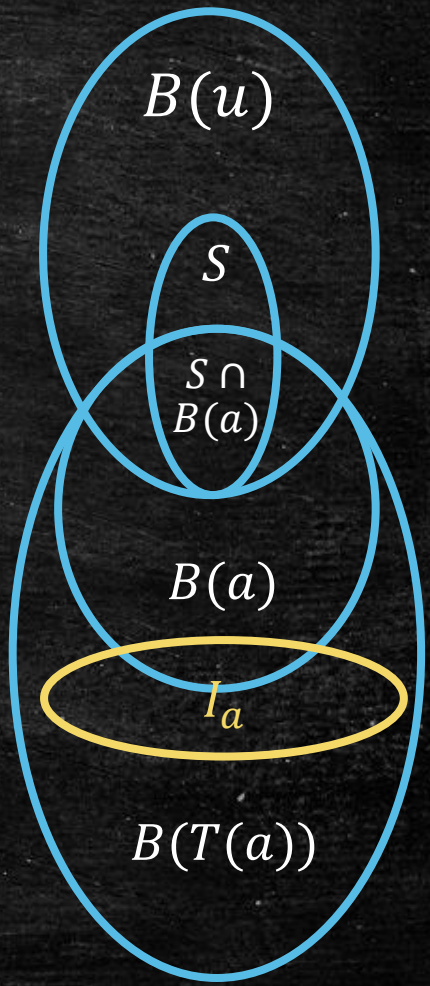




Question: How to calculate?

- Find a maximized independent set growing from $S \cap B(a) \rightarrow$ maximized $I_a \cup S \cap B(a)$.
- Question: is it solved in previous subproblems?
- Choices of a
  - $S' \cap B(u) = S \cap B(a)$
- MIS growing from $S \cap B(a)$
  - $\max_{s' \subset B(a), S' \cap B(u) = S \cap B(a)} f(S', a)$
  - $I_a = \max f(S', a) - |S \cap B(a)|$





Do the same thing for b!

- Choices of b
  - $\forall S' \subset B(b)$
  - $S' \cap B(u) = S \cap B(b)$
- MIS growing from $S \cap B(b)$
  - $\max_{s' \subset B(b), S' \cap B(u) = S \cap B(b)} f(S, b)$
  - $I_b = \max f(S, b) - |S \cap B(b)|$

Combining

- Grow s as large as possible in every direction!
- $f(S,u) =$

$$|S| + \sum_{a \in child(u)} \left(\max_{S' \in B(a), S' \cap B(u) = S \cap B(a)} f(S', a) - |S \cap B(a)| \right)$$

- Subproblem number: $O(2^k n)$
- Transfer Cost: $O(2^k)$
- Totally $O(4^k n)$

What we get!

- Tree-width: $k - 1$
  - Cycle: 2
  - Clique: n-1
  - Tree: 1
  - series-parallel graphs: ≤ 2
- $O(n)$ algorithm for Cycle, Tree, and series-parallel graphs!