Coping with NP-completeness: Special Cases

Alexander S. Kulikov

Steklov Institute of Mathematics at St. Petersburg
Russian Academy of Sciences

Advanced Algorithms and Complexity
Data Structures and Algorithms

The fact that a problem is **NP**-complete does not exclude an efficient algorithm for special cases of the problem.

Outline

1. 2-Satisfiability
2. Independent Sets in Trees

This part

- Striking connection between strongly connected components of a graph and formulas in 2-CNF
- A linear time algorithm for 2-SAT

2-Satisfiability (2-SAT)

Input: A set of clauses, each containing at most two literals (that is, a 2-CNF formula).

Output: Find a satisfying assignment (if exists).

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$
- $(x \vee y)(\overline{z})(z \vee \overline{x})(\overline{y})$ is unsatisfiable

Example

- $(x \vee y)(\overline{z})(z \vee \overline{x})$ is satisfied by $x = 0, y = 1, z = 0$
- $(x \vee y)(\overline{z})(z \vee \overline{x})(\overline{y})$ is unsatisfiable
- $(x \vee y)(x \vee \overline{y})(\overline{x} \vee y)(\overline{x} \vee \overline{y})$ is unsatisfiable

- Consider a clause $(\ell_1 \vee \ell_2)$

- Consider a clause $(\ell_1 \vee \ell_2)$
- Essentially, it says that ℓ_1 and ℓ_2 cannot be both equal to 0

- Consider a clause $(\ell_1 \vee \ell_2)$
- Essentially, it says that ℓ_1 and ℓ_2 cannot be both equal to 0
- In other words, if $\ell_1 = 0$, then $\ell_2 = 1$ and if $\ell_2 = 0$, then $\ell_1 = 1$

Definition

Implication is a binary logical operation denoted by $\Rightarrow$ and defined by the following truth table:

x	y	$x \Rightarrow y$
0	0	1
0	1	1
1	0	0
1	1	1

Definition

For a 2-CNF formula, its implication graph is constructed as follows:

- for each variable x, introduce two vertices labeled by x and $\overline{x}$;
- for each 2-clause $(\ell_1 \vee \ell_2)$, introduce two directed edges $\overline{\ell}_1 \to \ell_2$ and $\overline{\ell}_2 \to \ell_1$
- for each 1-clause (ℓ), introduce an edge $\overline{\ell} \to \ell$

Definition

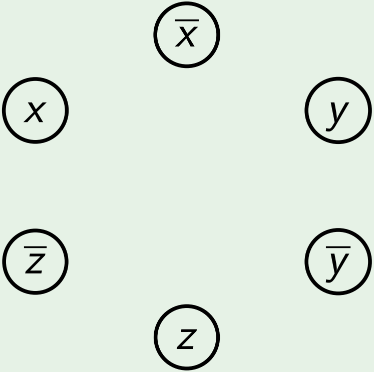
For a 2-CNF formula, its implication graph is constructed as follows:

- for each variable x, introduce two vertices labeled by x and $\overline{x}$;
- for each 2-clause $(\ell_1 \vee \ell_2)$, introduce two directed edges $\overline{\ell}_1 \to \ell_2$ and $\overline{\ell}_2 \to \ell_1$
- for each 1-clause (ℓ), introduce an edge $\overline{\ell} \to \ell$

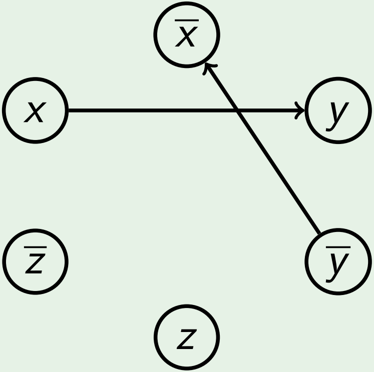
Encodes all implications imposed by the formula.

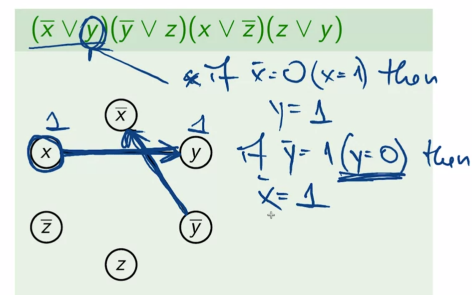
$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

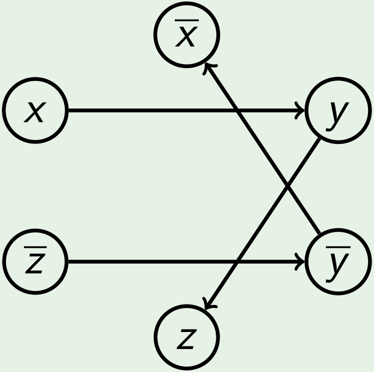


$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

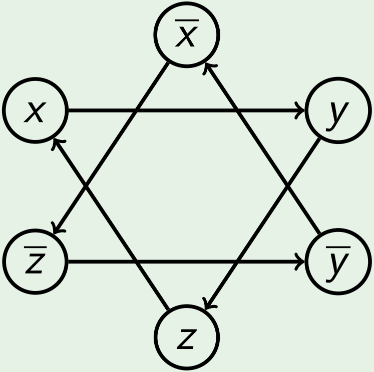




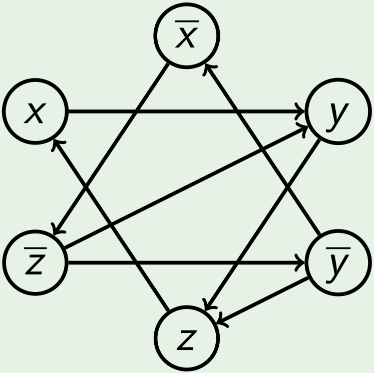
$(\overline{x} \lor y)(\overline{y} \lor z)(x \lor \overline{z})(z \lor y)$



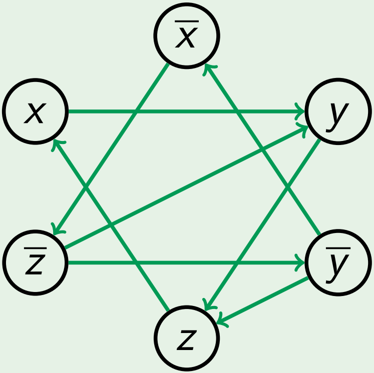
$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$



$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$

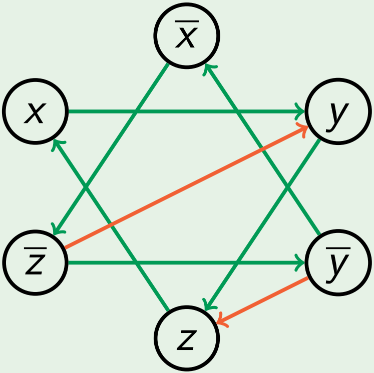


$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$



$x = 1, y = 1, z = 1$

$$(\overline{x} \vee y)(\overline{y} \vee z)(x \vee \overline{z})(z \vee y)$$



Two edges over here are falsified meaning beginning of this edge is 1 while the end is 0

Meaning truth (1) does not imply false (0)

$$x = 0, y = 0, z = 0$$

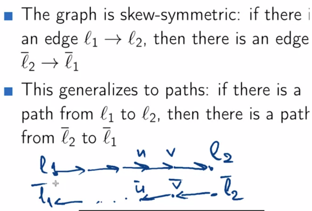
Thus, our goal is to assign truth values to the variables so that each edge in the implication graph is "satisfied", that is, there is no edge from 1 to 0.

Skew-Symmetry

- The graph is skew-symmetric: if there is an edge $\ell_1 \to \ell_2$, then there is an edge $\overline{\ell}_2 \to \overline{\ell}_1$

Skew-Symmetry

- The graph is skew-symmetric: if there is an edge $\ell_1 \to \ell_2$, then there is an edge $\overline{\ell}_2 \to \overline{\ell}_1$
- This generalizes to paths: if there is a path from ℓ_1 to ℓ_2, then there is a path from $\overline{\ell}_2$ to $\overline{\ell}_1$



Transitivity

Lemma

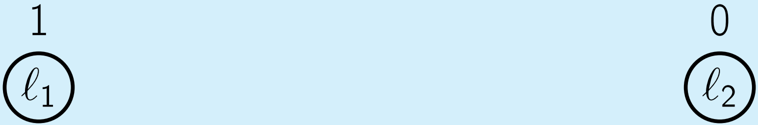
If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

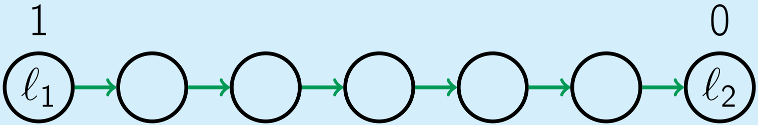


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

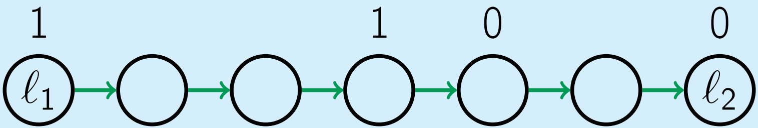


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof

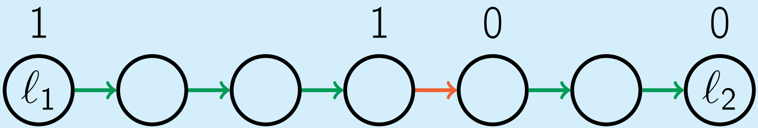


Transitivity

Lemma

If all the edges are satisfied by an assignment and there is a path from ℓ_1 to ℓ_2, then it cannot be the case that $\ell_1 = 1$ and $\ell_2 = 0$.

Proof



Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value

Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value
- In particular, if a SCC contains a variable together with its negation, then the formula is unsatisfiable

Strongly Connected Components

- All variables lying in the same SCC of the implication graph should be assigned the same value
- In particular, if a SCC contains a variable together with its negation, then the formula is unsatisfiable
- It turns out that otherwise the formula is satisfiable!

2SAT(2-CNF F)

```
construct the implication graph G
find SCC's of G
for all variables x:
  if x and x̄ lie in the same SCC of G:
    return "unsatisfiable"
find a topological ordering of SCC's
for all SCC's C in reverse order:
  if literals of C are not assigned yet:
    set all of them to 1
    set their negations to 0
return the satisfying assignment
```

Note that every step of this Algo takes linear time hence this is a linear time algorithm

CNF Formula is unsatisfiable if its implication graph contains a variable implied to its negation in strongly connected component.

`2SAT(2-CNF` F`)`

```
construct the implication graph G
find SCC's of G
for all variables x:
  if x and x̄ lie in the same SCC of G:
    return "unsatisfiable"
find a topological ordering of SCC's
for all SCC's C in reverse order:
  if literals of C are not assigned yet:
    set all of them to 1
    set their negations to 0
return the satisfying assignment
```

Running time: $O(|F|)$

Lemma

The algorithm `2SAT` is correct.

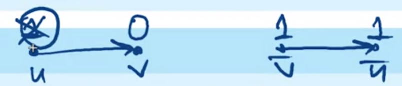
Proof

- When a literal is set to 1, all the literals that are reachable from it have already been set to 1 (since we process SCC's in reverse topological order).

Lemma

The algorithm 2SAT is correct.

Proof



If !v and !u are assigned 1 then u and v has to be 0. u = 1 (as shown) is a contradiction

- When a literal is set to 1, all the literals that are reachable from it have already been set to 1 (since we process SCC's in reverse topological order).

As we are assigning values in reverse topological order we always guarantee that the assigned value is 1 on the right side of implication graph (v is always 1 in u --> v)

- When a literal is set to 0, all the literals it is reachable from have already been set to 0 (by skew-symmetry).

Does this mean even the edge from 0 to 1 is not present ?

Outline

1. 2-Satisfiability
2. Independent Sets in Trees

Planning a company party

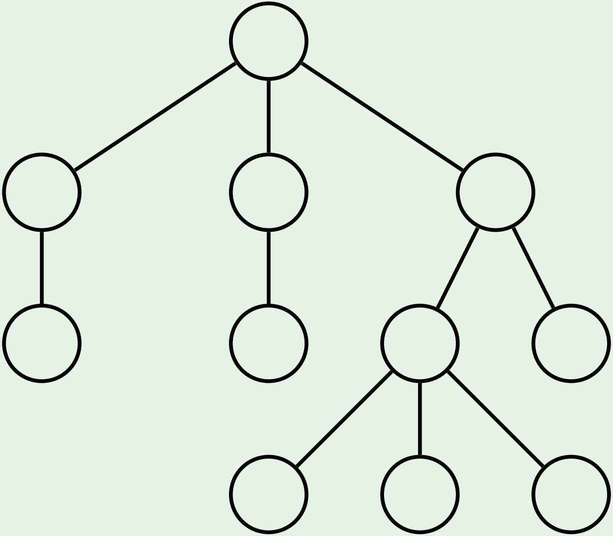
You are organizing a company party. You would like to invite as many people as possible with a single constraint: no person should attend a party with his or her direct boss.

Maximum independent set in a tree

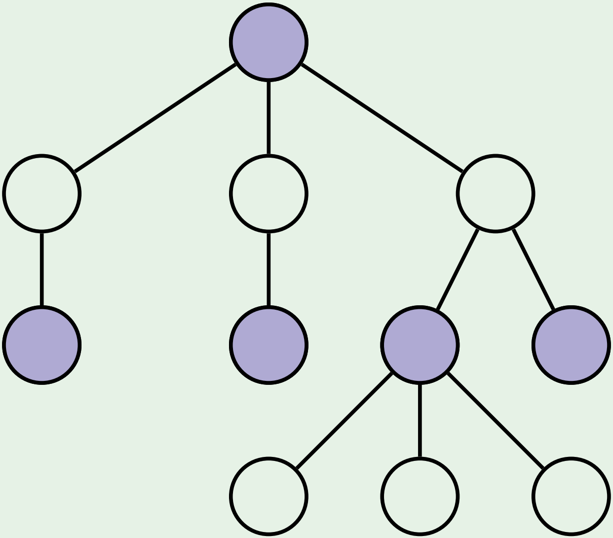
Input: A tree.

Output: An independent set (i.e., a subset of vertices no two of which are adjacent) of maximum size.

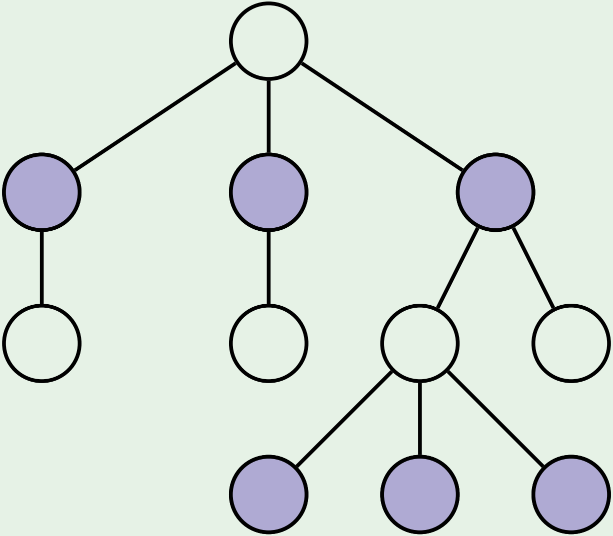
Example



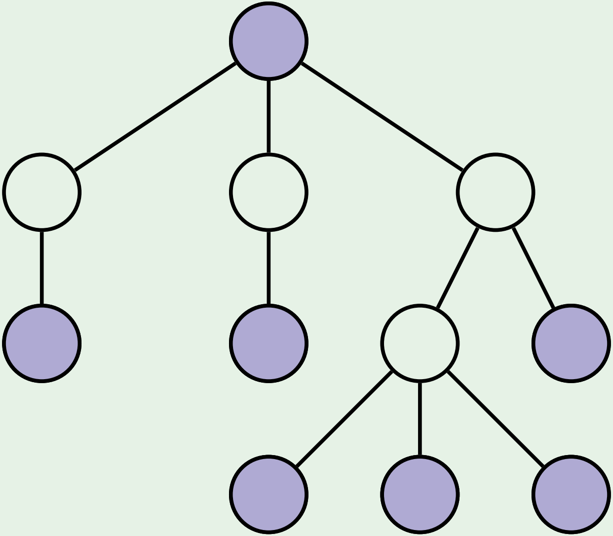
Example



Example

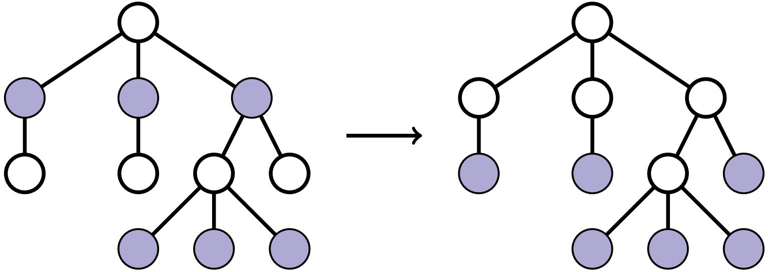


Example



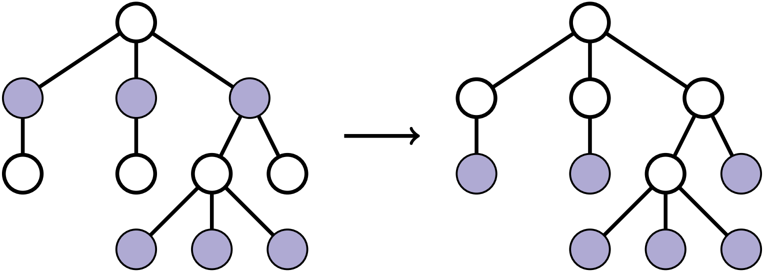
Safe move

For any leaf, there exists an optimal solution including this leaf.



Safe move

For any leaf, there exists an optimal solution including this leaf.



It is safe to take all the leaves.

PartyGreedy(T)

```
while T is not empty:
  take all the leaves to the solution
  remove them and their parents from T
return the constructed solution
```

PartyGreedy(T)

```
while T is not empty:
  take all the leaves to the solution
  remove them and their parents from T
return the constructed solution
```

For each vertex you keep track of number of its children's. You actually do not remove any vertex from the tree itself but you keep the track of current number of children's for each vertex so when you remove a vertex you decrease by one the number of children of it's parent and you also maintain a queue that contains the vertices that does not have any children so at each iteration you know all the leaves of the current tree these are exactly the vertices with no childrens

Running time: $O(|T|)$ (for each vertex, maintain the number of its children).

Planning a company party

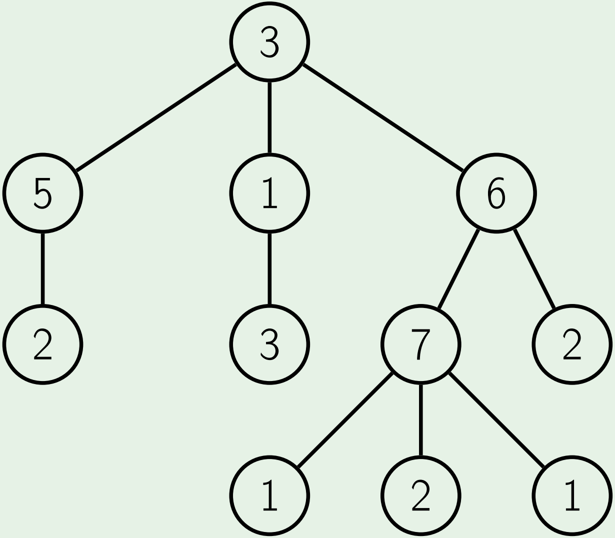
You are organizing a company party again. However this time, instead of maximizing the number of attendees, you would like to maximize the total fun factor.

Maximum weighted independent set in trees

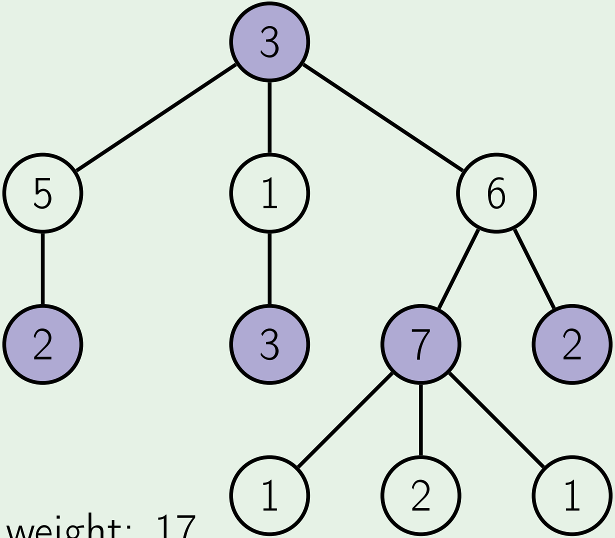
Input: A tree T with weights on vertices.

Output: An independent set (i.e., a subset of vertices no two of which are adjacent) of maximum total weight.

Example

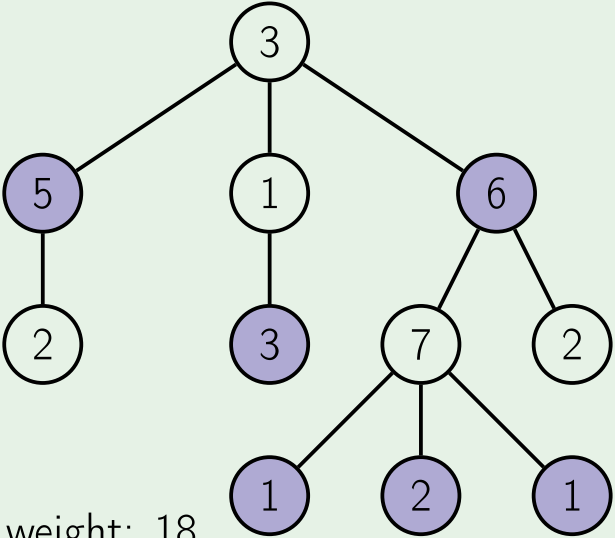


Example



total weight: 17

Example



total weight: 18

Subproblems

- $D(v)$ is the maximum weight of an independent set in a subtree rooted at v

Subproblems

- $D(v)$ is the maximum weight of an independent set in a subtree rooted at v
- Recurrence relation: $D(v)$ is

$$\max \left\{ w(v) + \sum_{\substack{\text{grandchildren} \\ w \text{ of } v}} D(w), \sum_{\substack{\text{children} \\ w \text{ of } v}} D(w) \right\}$$

There are basically two cases either we include v into solution or we do not include it
If we include it then it contributes it's weight to the total fun factor but we cannot take any of it's children's in the solution as then it won't be an independent set. However, we can take anything from subtree's root at its grandchildren.

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
```

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m1 ← w(v)
    for all children u of v:
      for all children w of u:
        m1 ← m1 + FunParty(w)
```

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m1 ← w(v)
    for all children u of v:
      for all children w of u:
        m1 ← m1 + FunParty(w)
    m0 ← 0
    for all children u of v:
      m0 ← m0 + FunParty(u)
```

Running time O(|T|)

Function FunParty(v)

```
if D(v) = ∞:
  if v has no children:
    D(v) ← w(v)
  else:
    m_1 ← w(v)
    for all children u of v:
      for all children w of u:
        m_1 ← m_1 + FunParty(w)
    m_0 ← 0
    for all children u of v:
      m_0 ← m_0 + FunParty(u)
  D(v) ← max(m_1, m_0)
return D(v)
```

Value returned here if already computed before

Question

We've seen that the independent set problem can be solved in polynomial time for trees.

We'll show now that it can be also solved in polynomial time for bipartite graphs. We know that the complement of any independent set in a graph is its vertex cover, so we will compute a minimum vertex cover of a graph. For this, we use the König's theorem that states that the number of vertices in a minimum vertex cover of a **bipartite graph** is equal to the number of edges in a maximum matching in the graph. The proof of this theorem is constructive, meaning that it provides a polynomial time algorithm that transforms a given maximum matching into a minimum vertex cover. In turn, a maximum matching can be found using flow techniques.

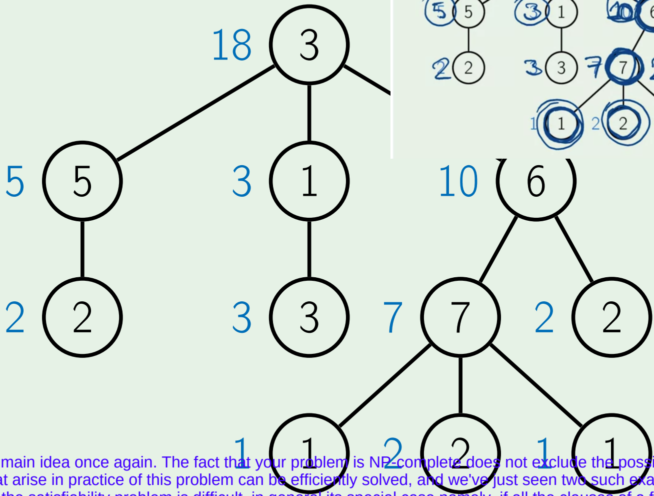
Is this argument correct?

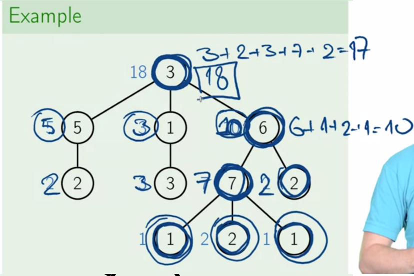
- (•) Yes, it is correct.
- ( ) No, it has a flaw.

✓ **Correct**
That's right!

Running time is O(|T|) because for each vertex there is just one serious call to fun party. By serious I mean a call which actually is inside this IF loop which gives rise to some computation, because after the first time when we are inside this loop for the vertex v, we will store the value in D(v) and then for any further call for FunParty(v) we just return the value immediately.

Example





Let me remind you the main idea once again. The fact that your problem is NP-complete does not exclude the possibilities that some special cases that arise in practice of this problem can be efficiently solved, and we've just seen two such examples. Despite of the fact that the satisfiability problem is difficult, in general its special case namely, if all the clauses of a formula contain at most two literals can be easily solved in minimal time. The second example was about independent set. This problem is hard in general, so given a graph, it is difficult to implement an algorithm which always finds an optimum size independent set of a graph. But, if you know that all of your graphs that are arrive in your application are trees,then it is easier to implement a linear time algorithm that will find an optimal answer quickly.