CS512 LECTURE NOTES - LECTURE 22

1 NP Complete Problems

1.1 The CNFSAT problem

We will use a version of the satisfiability problem where the given boolean formula is a conjunction of disjunctions, also called *Conjunctive Normal Form* (CNF). So the boolean formula is seen as a collection of k Clauses separated by "*and*" operators:

$$\phi(x_1, \ldots, x_n) = C_1 \wedge C_2 \wedge \ldots \wedge C_k$$

where each clause is a collection of literals separated by "*or*" operators:

$$C_j = (l_{j1} \vee l_{j2} \vee \ldots \vee l_{jr_j})$$

and each literal is a variable x_i or the negation of a variable $\neg x_i$.

An example of a boolean formula in CNF is:

$$\phi = (x_1 \vee \bar{x_2} \vee \bar{x_4} \vee x_5) \wedge (\bar{x_1} \vee x_2 \vee x_3 \vee \bar{x_4} \vee x_5) \wedge (x_1 \vee \bar{x_3} \vee x_5) \wedge (x_2 \vee \bar{x_4} \vee x_5)$$

Notice that in the reduction that we made from Turing Machine to SAT, we only used CNF formulas, so the reduction that was given was actually $\Pi_x \leq_P CNFSAT$, where Π_x is a problem in NP.

1.2 The 3SAT problem

The 3SAT problem is similar to the CNFSAT problem but each clause has at most three literals. The decision problem consists of finding an assignment of the variables such that the boolean formula ϕ is satisfied.
Theorem The 3SAT problem is NP-complete.

Proof.

1. 3SAT is in NP:
   Given a certificate for 3SAT in the form of a truth assignment of the variables it is easy to verify that in each clause there is at least one literal that is true. This verification can clearly be done in polynomial time with respect to the number of clauses and literals.

2. 3SAT is in NP-hard:

   We will prove this by providing a reduction from CNFSAT.

   Theorem CNFSAT$\leq_p$3SAT

   Proof.

   Given an instance of CNFSAT:

   A boolean formula with k clauses and n variables

   we will transform one clause

   $$(l_1 \vee l_2 \vee l_3 \vee l_4 \vee \dots l_r)$$

   into several clauses with at most three literals each by creating additional variables $y_1, \dots, y_{r-3}$:

   $$(l_1 \vee l_2 \vee y_1) \wedge (\bar{y_1} \vee l_3 \vee y_2) \wedge (\bar{y_2} \vee l_4 \vee y_3) \wedge \dots \wedge (\bar{y_{r-3}} \vee l_{r-1} \vee l_r)$$

   Whe now have to show there is a satisfying assignment for

   $$(l_1 \vee l_2 \vee l_3 \vee l_4 \vee \dots l_r)$$

   $\Leftrightarrow$ there is a satisfying assignment for

   $$(l_1 \vee l_2 \vee y_1) \wedge (\bar{y_1} \vee l_3 \vee y_2) \wedge (\bar{y_2} \vee l_4 \vee y_3) \wedge \dots \wedge (\bar{y_{r-3}} \vee l_{r-1} \vee l_r)$$

   - ($\Rightarrow$) Assume that there is a satisfying assignment for

     $$(l_1 \vee l_2 \vee l_3 \vee l_4 \vee \dots l_r)$$

     $\Rightarrow$ at least one literal is true, $l_i = T$ This means that the literals to the right and left in the clause where l_i appear can be false (F), and then these y's can be propagated to the previous and next clauses as True, so all the clauses will be satisfied:

$$(l_1 \vee l_2 \vee y_1) \wedge (\bar{y_1} \vee l_3 \vee y_2) \wedge (\bar{y_2} \vee l_4 \vee y_3) \wedge \ldots \wedge (\bar{y_{r-3}} \vee l_{r-1} \vee l_r)$$

for instance, if l_3 is true then $\bar{y_1}$ and y_2 can be F, which means that y_1 is true, and clause 1 is satisfied, and $\bar{y_2}$ is true, and clause 3 is satisfied, and so on.

- ($\Leftarrow$) Assume that all the clauses in

  $$(l_1 \vee l_2 \vee y_1) \wedge (\bar{y_1} \vee l_3 \vee y_2) \wedge (\bar{y_2} \vee l_4 \vee y_3) \wedge \ldots \wedge (\bar{y_{r-3}} \vee l_{r-1} \vee l_r)$$

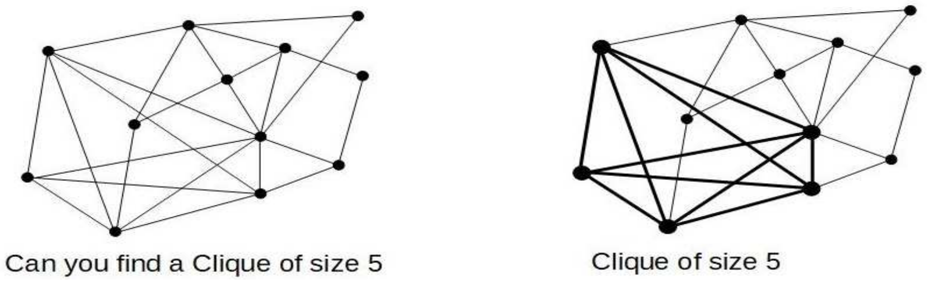
  are satisfied, but all $l_i = F \ \forall i = 1 \ldots r$, then y_1 must be True for the first clause to be satisfied, which implies that $\bar{y_1} = F$, so y_2 must be true, and so on, until we have that y_{r-3} must be true, which implies that $\bar{y_{4-3}}$ is false, and since l_{r-1} and l_r are false, then the last clause is false. Which contradicts our assumption that all the clauses are satisfied. Therefore, at least one literal must be true.

$\square$

1.3 The Clique problem

Given an undirected graph $G = (V, E)$ the goal of this problem is to find a set of vertices $V' \subseteq V$ that induce a complete graph of maximum size. The decision version of this problem can be written as:
Input: $G = (V, E), \ \ k \in \mathbb{N}$
Question: Is there a $V' \subseteq V$ such that $|V'| = k$ and $\forall \ u, v \in V'$, $(u, v) \in E$.

The following figure illustrates a clique of size 5.



Theorem: Clique is NP-Complete

Proof:

1. Clique is in NP:

   Verifying that a given set of vertices $V' \subseteq V$ is a clique can be done in time $O(|V|^2)$ since we just need to check that $\forall u, v \in V'$, $(u, v) \in E$.

2. Clique is NP-hard:

   To show that clique is np-hard we will provide a reduction from 3SAT:

   Given a 3SAT formula $\phi(x_1, \ldots, x_n)$ consisting of k clauses $C_1, C_2, \ldots C_k$ where each clause has at most 3 literals, $C_i = (l_{i,1}, l_{i,2}, l_{i,3})$ and where each literal is either a variable or its negation

   <u>We reduce it to an instance of Clique:</u>

   We build a graph where there is a vertex for each literal:

   $V = \{l_{i,j} | 1 \leq i \leq k,\ 1 \leq j \leq |C_i| \leq 3\}$

   ($|C_i|$ is the number of literals in clause C_i).

   and an edge connecting literals from **different** clauses as long as they are logically compatible, i.e. a literal is not adjacent to its negation.

   $E = \{(l_{i,j}, l_{r,s}) | i \neq r\ \wedge l_{i,j} \neq \neg l_{r,s}\}$

   Example:

   Given the boolean formula

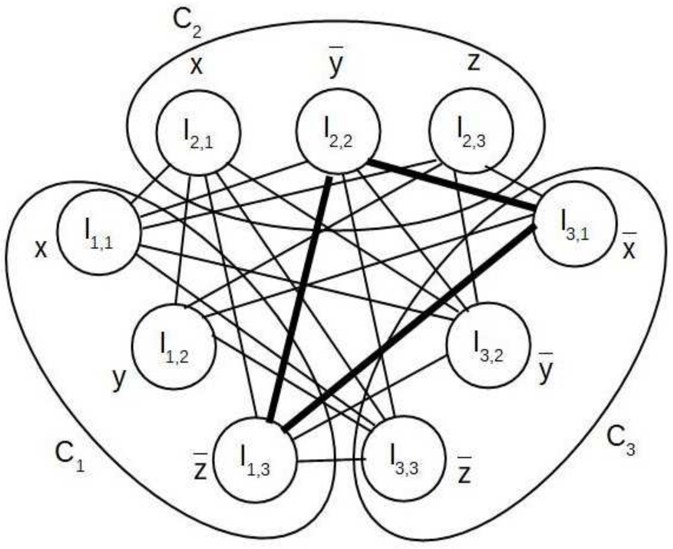
   $$\phi(x, y, z) = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

   we have the following clauses:

   $$C_1 = (l_{1,1} \vee l_{1,2} \vee l_{1,3}) = (x \vee y \vee \neg z)$$

$$\begin{aligned} C_2 &= (l_{2,1} \vee l_{2,2} \vee l_{2,3}) = (\neg x \vee \neg y \vee \neg z) \\ C_3 &= (l_{3,1} \vee l_{3,2} \vee l_{3,3}) = (x \vee \neg y \vee z) \end{aligned}$$

Which we use to construct the graph $G = (V, E)$ based on the description given above:



Notice that in this example there are several cliques of size 3, one of them is denoted by thick edges.

Theorem: $3SAT \leq_p clique$

Proof:

We have to show that $\phi(x_1, \ldots, x_n)$ is satisfiable if and only if $G = (V, E)$ (as described above) has a clique of size k

- ($\Rightarrow$) Assume that $\phi(x_1, \ldots, x_n)$ is satisfiable, then at least one literal in each clause must be true. Pick one true literal from each clause, $l_{1,t_1}, l_{2,t_2}, \ldots, l_{k,t_k}$. We claim that the set of vertices corresponding to those literals $V' = \{l_{1,t_1}, l_{2,t_2}, \ldots, l_{k,t_k}\}$, is a clique of size k. To prove this, let us take two arbitrary vertices from V', i.e. $l_{i,t_i}, l_{j,t_j} \in V'$, since the literals corresponding to these vertices are true, one cannot be the negation of the other one, i.e. $l_{i,t_i} \neq \neg l_{j,t_j}$, therefore $(l_{i,t_i}, l_{j,t_j}) \in E$.

$\square$

- ($\Leftarrow$) Assume that $G = (V, E)$ has a clique of size k, then the k literals corresponding to each vertex can be set to true, with no conflict since they are all neighbors of each other (and so none can be the negation of another one). Since (at least) one literal in each clause is True, then the entire boolean formula is true, and the assignment is built by setting the variable that corresponds to each of the true literals to its corresponding value.

$\square$

1.4 The Vertex Cover problem

The input to the vertex cover problem is an undirected graph $G = (V, E)$ and an integer k. The decision problem is to determine if G has a vertex cover V' of size k.

A vertex cover of G is a subset $V' \subseteq V$ such that every edge has at least one endpoint in V'. Formally, $\forall (u, v) \in E$, $u \in V'$ or $v \in V'$.
Theorem. The Vertex Cover problem is NP-complete.
Proof

1. Vertex Cover is in NP:
   Given a certificate V' it can easily be verified that $V' \subseteq V$, and that $\forall (u, v) \in E$, $u \in V'$ or $v \in V'$. In particular if the set V' is given as an array vp of $0, 1$ where $vp_i = 1$ iff $i \in V'$, and the edges are given in an adjacency list, the verification process can be done in $O(|E|)$ time, which clearly is polynomial.

2. Vertex Cover is NP-hard:

   In order to prove this we need to show that a known NP-hard problem can be polynomially reduced to Vertex cover.

   Theorem: Clique $\leq_p$ Vertex Cover

   Proof

   Given an instance of Clique:

   An undirected graph $G = (V, E)$ and an integer k

   We reduce it to an instance of Vertex cover:

   $G^c = (V, E^c)$ $k' = |V| - k$

   We have to show that G has a clique of size k if and only if G^c has a vertex cover of size $|V| - k$

- ($\Rightarrow$) Assume that G has a clique $V' \subset V$ such that $|V'| = k$.

  We start by assuming that (u, v) is an edge in G^c

  $$\begin{aligned} \text{Let } (u,v) \in E^c &\Rightarrow (u,v) \notin E \\ &\Rightarrow \text{both } u \text{ and } v \text{ cannot be in the clique} \\ &\Rightarrow u \notin V' \text{ or } v \notin V' \\ &\Rightarrow u \in V - V' \text{ or } v \in V - V' \end{aligned}$$

  Therefore $V - V'$ satisfies the definition of vertex cover, and since $|V'| = k$ we get that $|V - V'| = |V| - k$, i.e. $V - V'$ is a vertex cover of size $|V| - k$

- ($\Leftarrow$) Assume that G^c has a vertex cover V' of size k.

  We start by assuming (again) that $(u, v) \in E^c$, and since V' is a vertex cover, we have

  $$\text{Let } (u, v) \in E^c \quad \Rightarrow \quad u \in V' \text{ or } v \in V'$$

  Taking the contrapositive of this last proposition:

  $$u \notin V' \text{ and } v \notin V' \quad \Rightarrow \quad (u, v) \notin E^c$$

  Which is equivalent to

  $$u \in V - V' \text{ and } v \in V - V' \quad \Rightarrow \quad (u, v) \in E$$

  So, if two vertices u, v are such that $u \in V - V'$ and $v \in V - V' \Rightarrow (u, v) \in E$, we have that $V - V'$ satisfies the definition of clique.
  Therefore $V - V'$ is a clique of size $|V| - |V'| = |V| - k$

$\square$

1.5 The Subset Sum problem

An instance of the subset sum problem consists of a set of positive integers $S \in \mathrm{N}$ and an integer k.

The problem is to find a subset $S' \subseteq S$ such that the sum of the elements in S' add up exactly to k, i.e.

$$\sum_{i \in S'} i = k$$

We will show that this problem is NP-complete, however, first we will show that there is a Dynamic programming algorithm that can solve this problem:

Dynamic Programming Algorithm for Subset Sum

For this simple algorithm we will use a boolean array to store the desired values. The number of elements in the array will be equal to the sum of all the elements in S.

Suppose that $S = \{3, 6, 7, 10\}$, $k = 19$.

Let us initialize our array such that it has a 1 in position 0 and 0 everywhere else. Meaning that without any elements in S the sum that we can get is 0.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Let us store in an array (row one of the matrix) those numbers that can be obtained by adding **only the first element in** $3 \in S$, to do this, we find the 1 in the array and put a 1 three positions to the right.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

We repeat this process with the second element $6 \in S$, find all the 1s and put a 1 six positions to the right:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Now our array contains a 1 in those positions that represents sums of subsets of $\{3, 6\}$.

We repeat this process with the third element $7 \in S$, find all the 1s and put a 1 seven positions to the right:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0	0	1	0	0	1	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0

Now our array contains a 1 in those positions that represents sums of subsets of $\{3, 6, 7\}$.

Finally, we repeat this process with the fourth element $10 \in S$, find all the 1s and put a 1 ten positions to the right:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
1	0	0	1	0	0	1	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0	0	1	0	0	1

Now our array contains a 1 in those positions that represents sums of subsets of $\{3, 6, 7, 10\}$.

We can now see that since there is a 1 in position $k = 19$, then there is a subset of elements in S that adds up to 19, so we answer **YES**.

So we can write the algorithm:

```
Algorithm: Dynamic Programming Algorithm for Subset Sum
Input:
    $S[1], \ldots, S[n]$: array $S$ or positive integers
    $k$ integer
$sum = \sum_{i=1}^{n} s[i]$
$a[0] = 1$
for $j = 1$ to $sum$
    $a[j] = 0$
for $i = 1$ to $n$
    for $j = 1$ to $sum$
        if $a[j] = 1$ and $j + s[i] \leq sum$
            $a[j + s[i]] = 1$
if $a[k] = 1$ return true
return false
```

Analysis

This dynamic programming algorithm takes $O(n \sum_{i=1}^{n} s[i])$ time.

The problem is that $\sum_{i=1}^{n} s[i]$ is a value computed from input elements, which can be as long as 2^r, where r is the number of bits required to write the numbers represented in the array. Therefore the algorithm runs in exponential time with respect to the size of the input. This types of algorithms are called pseudopolynomial time algorithms.

Theorem. The Subset Sum Problem is NP-complete.

Even though subset sum has a pseudopolynomial time algorithm, it can be proven to be NP-complete.

Proof

1. Subset Sum is in NP:

   Notice that given a certificate $S' \subseteq S$ it requires polynomial time to verify that the sum of the elements in S' is equal to k. Notice that the sums can be performed in linear time with respect to the length of the input (number of bits representing the numbers in the set).

2. Subset Sum is NP-hard:

   Theorem: 3SAT $\leq_p$ Subset Sum

   Proof

   Given an instance of 3SAT:

   A boolean formula $\varphi(x_1, x_2, \ldots, x_n)$ on n variables. Assume that the formula has k clauses.

   <u>We reduce it to an instance of Subset sum:</u>

In what follows, we will create integers, but we will address the positions of the digits in string-like fashion. Each string has $n + k$ characters.

- The first n characters of the string will represent the variables $x_1, x_{,2}, \ldots, x_n$
- The last k characters of the string will represent the clauses.

We create two strings for each variable in the following way:

- v_i: We put a 1 in the position of variable i and a 1 the position of each clause in which variable x_i appears **unnegated**.
- v_i': We put a 1 in the position of variable i and a 1 the position of each clause in which variable x_i appears **negated**.

Every other position has a 0

We will need to add two extra strings as slack for each clause C_j:

- s_j put a 1 in the position of clause j
- s'_j put a 2 in the position of clause j

Every other position has a 0

The target value will be set to a string containing 1 in each variable position and 4 in each clause position.

Example

Given the boolean formula

$$\varphi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$$

where the clauses are:

$$\begin{aligned} C_1 &= x_1 \vee \neg x_2 \vee \neg x_3 \\ C_2 &= \neg x_1 \vee \neg x_2 \vee \neg x_3 \\ C_3 &= \neg x_1 \vee \neg x_2 \vee x_3 \\ C_4 &= x_1 \vee x_2 \vee x_3 \end{aligned}$$

Following the two steps above we create two integers (strings) for each variable:

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
$v_1 =$	1	0	0	1	0	0	1
$v_1' =$	1	0	0	0	1	1	0
$v_2 =$	0	1	0	0	0	0	1
$v_2' =$	0	1	0	1	1	1	0
$v_3 =$	0	0	1	1	1	0	0
$v_3' =$	0	0	1	0	0	1	1
$k =$	1	1	1	4	4	4	4

Since the sum of the variable positions is 1 then we have to select one of the two possible integers representing either the negated or the unnegated variable, but not both.

Notice that for a formula to be satisfiable, each one of the clauses must be true, i.e. at least one literal in each clause must be true. So the possible values of the sum of the clause positions are 1, 2 and 3. How can we get to 4?, we can add two slack strings for each clause, with values 1 and 2 so that we can make up the sum of 4.

Slack integers:

	x_1	x_2	x_3	C_1	C_2	C_3	C_4
s_1	0	0	0	1	0	0	0
s'_1	0	0	0	2	0	0	0
s_2	0	0	0	0	1	0	0
s'_2	0	0	0	0	2	0	0
s_3	0	0	0	0	0	1	0
s'_3	0	0	0	0	0	2	0
s_4	0	0	0	0	0	0	1
s'_4	0	0	0	0	0	0	2

Proof (cont'd)

We now show that ϕ is satisfiable if and only if there is a subset of $S' \subset \{v_1, v'_1, v_2, v'_2, \ldots, v'_n, s_1, s'_1, \ldots, s'_k\}$ such that it adds exatly up to $k = 1 \ldots 14 \ldots 4$.

- ($\Rightarrow$) Assume that ϕ is satisfiable. Then there exists truth an assignment of the variables $x_1, \ldots, x_n$ such that at least one literal in each clause is true. We will pick the following integers to be in S':
  - if $x_i = T$ in the satisfying assignment, then $v_i \in S'$
  - if $x_i = F$ in the satisfying assignment, then $v'_i \in S'$

  Notice that the sum of each one of the variable positions is equal to 1, and the sum of each one of the clause positions is at least 1, so we add the required slack variables so that the sum of each clause position is equal to 4.
- ($\Leftarrow$) Assume that there is a set S' such that its elements add up to $1111 \ldots 11444 \ldots 4$.

  Then it contain have exactly one integer with a one in a variable position. We can build an assignment of the variables of the boolean formula in the following way:
  - if $v_i \in S'$ then set $x_i = T$
  - if $v'_i \in S'$ then set $x_i = F$

  Since the sum of each clause position must be at least equal to 1 (there is no way to add up to 4 unless the sum without the slack values equals at least 1 because slack values can only be 1 or 2 in each clause position), then at least one literal of each clause must be true, therefore the formula is satisfiable.

$\square$

2 What to do when a problem is NP-Complete

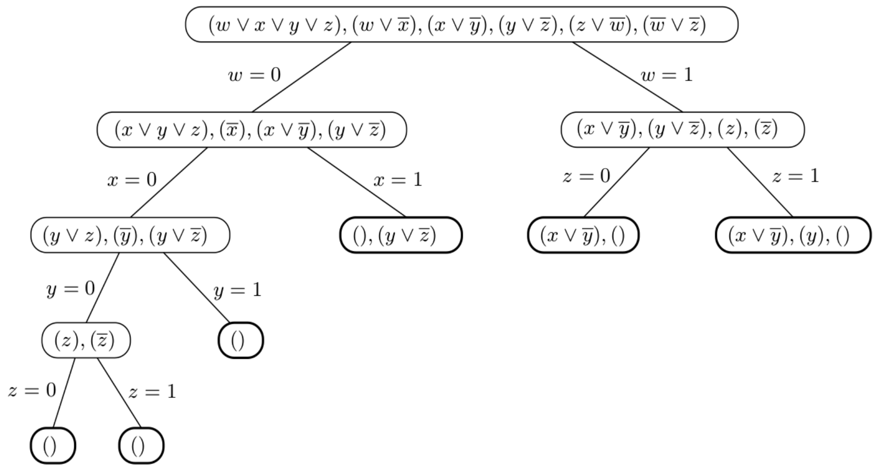
There are several alternatives when we find ourselves with trying to solve an NP-complete problem:

2.1 Backtracking

We think about an NP-Complete problem as that of exploring a tree of options. We will prune those branches that do not lead to solution hoping to reduce the search space as much as possible. Useful in the case of decision problems

```
Backtracking($P_0$)
Let $L = \{P_0\}$ where $P_0$ is the original problem
while $L$ is not empty
    $P$ = extract from $L$
    expand $P$ into subproblems $P_1, \ldots, P_k$
    for each $P_i$ $(i = 1, \ldots k)$
        If answer is "yes" then return true
        If answer is "no" do nothing
        else $L = L \cup \{P_i\}$
return "no"
```

Example:



$(w \vee x \vee y \vee z), (w \vee \overline{x}), (x \vee \overline{y}), (y \vee \overline{z}), (z \vee \overline{w}), (\overline{w} \vee \overline{z})$
$w = 0$
$w = 1$
$(x \vee y \vee z), (\overline{x}), (x \vee \overline{y}), (y \vee \overline{z})$
$(x \vee \overline{y}), (y \vee \overline{z}), (z), (\overline{z})$
$x = 0$
$x = 1$
$z = 0$
$z = 1$
$(y \vee z), (\overline{y}), (y \vee \overline{z})$
$(), (y \vee \overline{z})$
$(x \vee \overline{y}), ()$
$(x \vee \overline{y}), (y), ()$
$y = 0$
$y = 1$
$(z), (\overline{z})$
$()$
$z = 0$
$z = 1$
$()$
$()$

2.2 Branch and Bound

In the case of optimization problems, the algorithm is similar, but we keep track of:

- A current `best_so_far` value
- For each subproblem in the tree, we compute a lower bound of its solution (in the case of a minimization problem)

If we are dealing with a minimization problem then we keep searching only if the `lowerbound` $<$ `best_so_far`.

So the branch and bound algorithm ends up being something like:

```
Branch and Bound($P_0$)
Let $L = \{P_0\}$ where $P_0$ is the original problem
best_so_far=$\infty$
while $L$ is not empty
    $P$ = extract from $L$
    expand $P$ into subproblems $P_1, \dots, P_k$
    for each $P_i$ $(i = 1, \dots k)$
        If $P_i$ is a complete solution: update best_so_far
        else if lowerbound($P_i$)<best_so_far then $L = L \cup \{P_i\}$
return best_so_far
```

2.3 Approximation Algorithms

For some NPO problems it might be possible to find a polynomial time algorithm that provides an approximation to its solution. Let us say that given an instance I of a problem Π

- $Opt(I)$ is the optimum solution of instance I
- $Alg(I)$ is the solution computed by the given approximation algorithm

In the case of a minimization problems it is clear that

$$Alg(I) \geq Opt(I)$$

Therefore

$$\frac{Alg(I)}{Opt(I)} \geq 1$$

There are some cases where and algorithm can be found such that

$$\frac{Alg(I)}{Opt(I)} \leq \rho$$

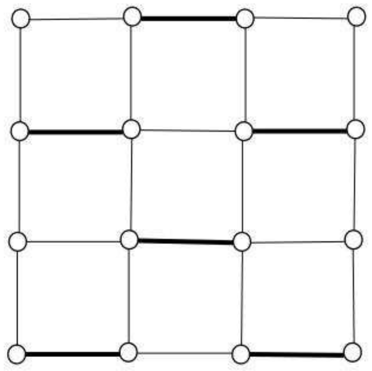
where $\rho \geq 1$ is a constant.

Exampe

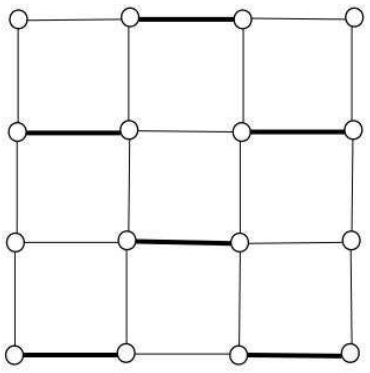
Let us take Vertex Cover again as an example and let us see its relationship with Maximal matching problem.

- Vertex Cover: Find a set of vertices $V' \subseteq V$ such that $\forall (u, v) \in E$, $u \in V'$ or $v \in V'$
- Maximal matching: Find a set of edges $M \in E$ with no common vertices but such that given any other edge $(u, v) \in E$, each one of its end points belongs to an edge in M. So it is not possible to extend M further (that is why it is maximal)

The figure shows a maximal matching for a graph:



Important Properties



1. Each edge of a *maximal matching* must be covered by one of its end-points by any *vertex cover*.
2. If S contains both end-points of a maximal matching, S must be a vertex cover. If this did not happen the matching would not be maximal.

So we have the following algorithm:

```
ApproximateVertexCover(G)

Find a maximal matching M (can be done in polynomial time)
Let S be the set of all the end points of M
return |S| (as an approximation to the max Vertex Cover)
```

Again by the properties given:

1. Each edge of a *maximal matching* must be covered by one of its end-points by any *vertex cover*:

   This includes the optimum vertex cover, so we have:

   $$opt(I) \geq |M|$$

   Where $opt(I)$ is the optimum vertex cover of instance I

2. If S contains both end-points of a maximal matching, S must be a vertex cover.

   The approximation algorithm given above returns:

   $$Alg(I) = 2|M|$$

Which implies that

$$\begin{aligned} |M| &= \frac{Alg(I)}{2} \\ \Rightarrow \quad opt(I) &\geq \frac{Alg(I)}{2} \\ \Rightarrow \quad \frac{Alg(I)}{opt(I)} &\leq 2 \end{aligned}$$

So we have an approximation algorithm for the Vertex Cover problem.

2.4 Approximability classes

These classes apply for NPO problems:

- **APX:** Class of problems that accepts a polynomial time approximation algorithm with fixed constant ratio.
- **APX-hard:** Π is in APX-hard iff there is a PTAS reduction $\leq_{PTAS}$ from every problem in APX to it.
- **APX-complete:** problems in APX that are APX-complete
- **PTAS:** Polynomial time approximation Scheme. Approximable within $1 + \epsilon$ in polynomial time. Notice that $PTAS \subseteq APX$
- **FPTAS:** Admits a fully polynomial time approximation scheme (similar to a pseudopolynomial time algorithm).