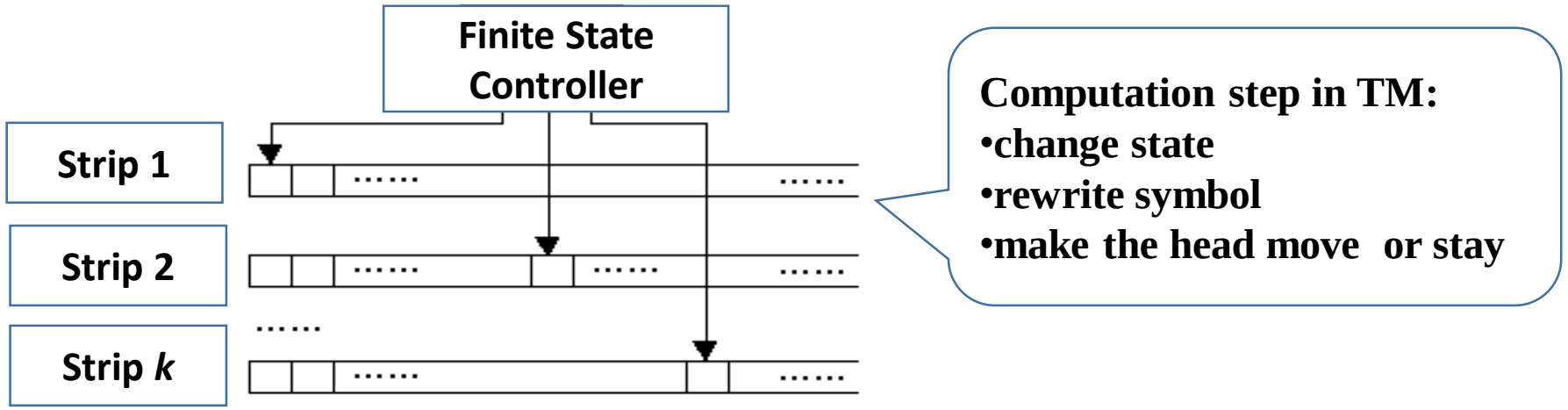
Lecture 11 NPC problems

- NP-complete (NPC) problems
- Reductions
- Approximation Algorithms

Computation Model: Turing machine

- A k-strip TM is a tuple $< Q, \quad T, \quad I, \quad \delta, \quad b, \quad q_0, \quad q_f >$:
  - Q is a set of finite states
  - T is a set of finite symbols
  - I is a set of input symbols and $I \subseteq T$
  - b is a blank symbol
  - q_0 is an initial state
  - q_f is a final state
  - δ is a transfer function: $Q \times T^k \rightarrow Q \times T \times \{L, \quad R, \quad S\}$

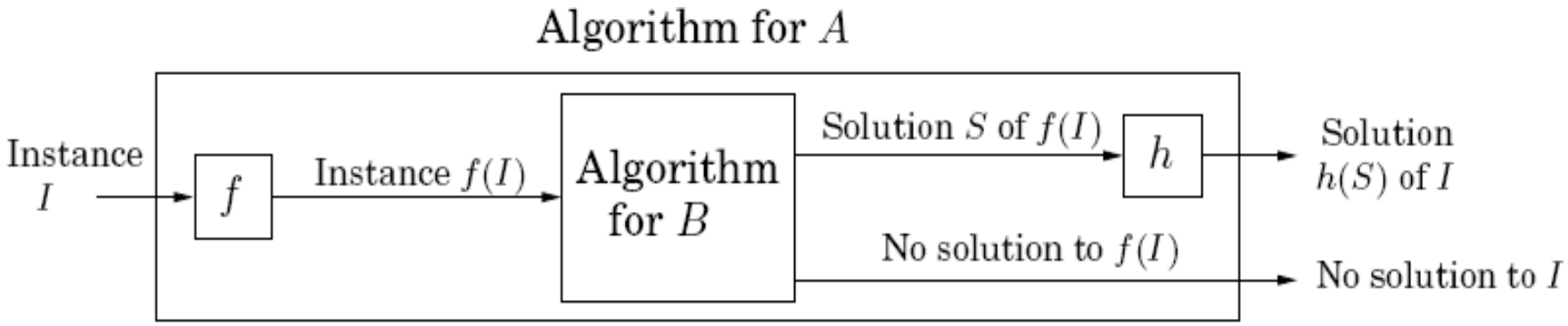


NP-complete problems

- **P (polynomial) and NP (nondeterministic polynomial)**
  - NP: the class of all problems that can be solved in polynomial time on nondeterministic TM.
  - P: the class of all problems that can be solved in polynomial time on deterministic TM
  - P ≠ NP?
    - Most researchers think so
    - But no one can prove it

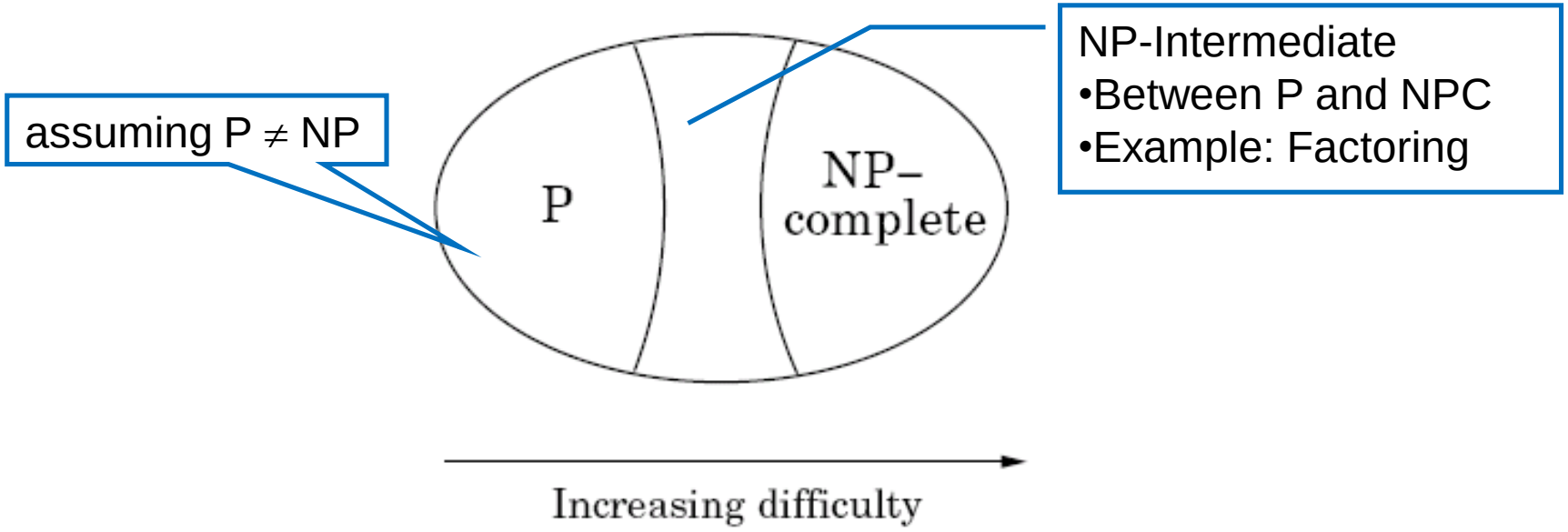
NP-complete problems

- Reduction
  - A *reduction* from *A* to *B* is *f* + *h*
    - Both *f* and *h* are polynomial-time algorithms
    - *f* transforms any instance *I* of *A* into an instance *f*(*I*) of *B*
    - *h* maps any solution *S* of *f*(*I*) back into a solution *h*(*S*) of *I*

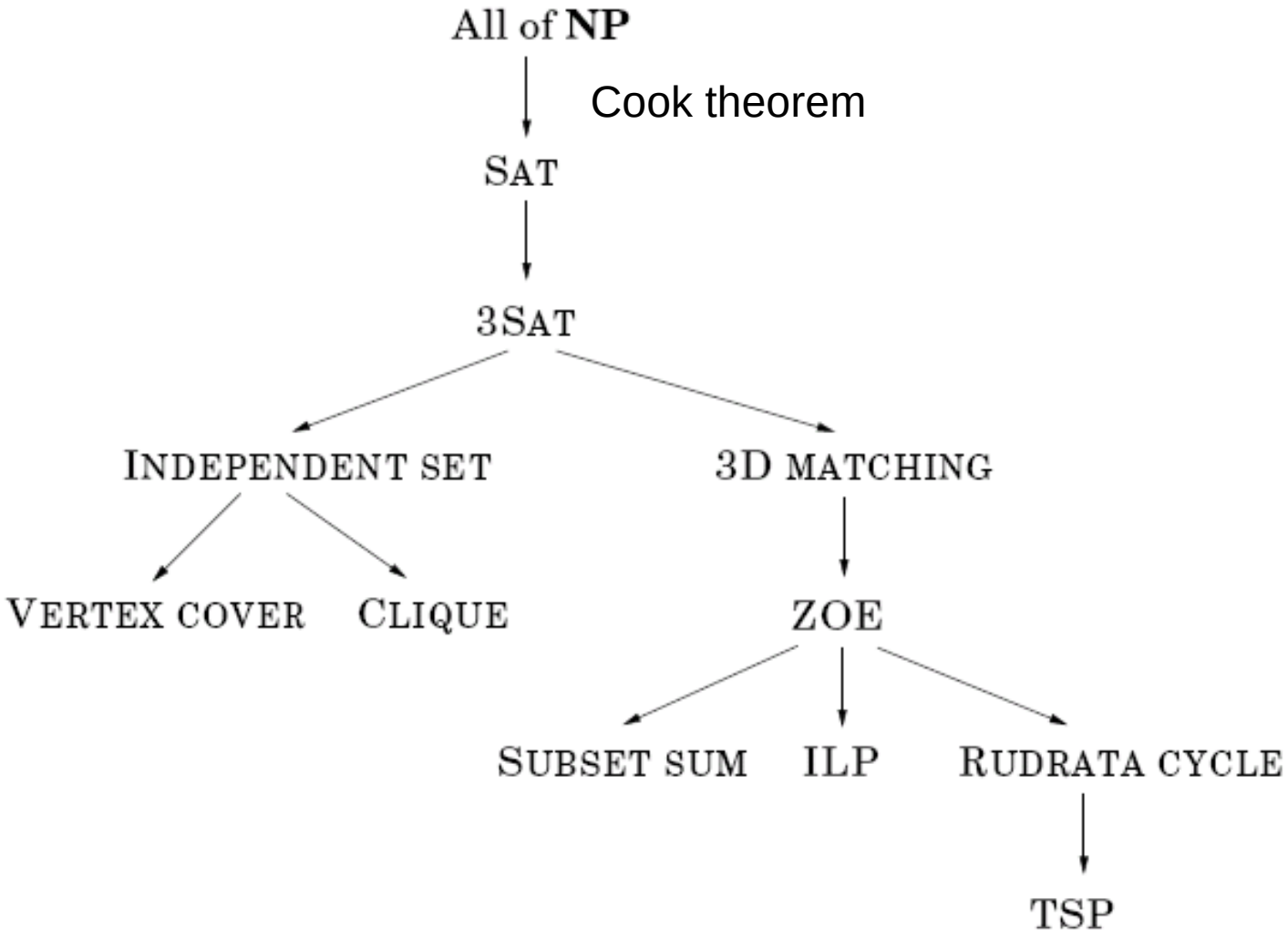


NP-complete problems

- NP-complete (NPC)
  - The class of the hardest problems (NPC $\subseteq$ NP)
  - All other problems reduce to a NPC problem ($\forall np \propto_P npc$)
  - If one of NPC has a polynomial time algorithm, then every problem in NP can be solved in polynomial time, so P = NP



- Reduction tree



Example 1

- SAT → 3SAT
  - Reduction: any clause

$$(a_1 \vee a_2 \vee \ldots \vee a_k), k > 3 \longrightarrow (a_1 \vee a_2 \vee y_1)(\bar{y}_1 \vee a_3 \vee y_2)(\bar{y}_2 \vee a_4 \vee y_3)\ldots(\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$$

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \Longleftrightarrow \left\{ \begin{array}{c} \text{there is a setting of the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)\ (\bar{y}_1 \vee a_3 \vee y_2)\ \cdots\ (\bar{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Left satisfied

→ Some a_i must be true

→Set $y_1,\ldots, y_{i-2}$ to true and the rest to false $(\ldots(\neg y_{i-2} \vee a_i \vee y_{i-1})(\neg y_{i-1} \vee a_{i+1} \vee y_i)\ldots)$

→ Right satisfied

Right satisfied

→ left satisfied (One of $a_1,\ldots, a_k$ must be true)

→Otherwise (All of $a_1,\ldots, a_k$ are false)

- y_1 must be true, y_2 must be true, …, y_{k-3} must be true $(\neg y_{k-4} \vee a_{k-2} \vee y_{k-3})$
- False the last clause (Right unsatisfied)

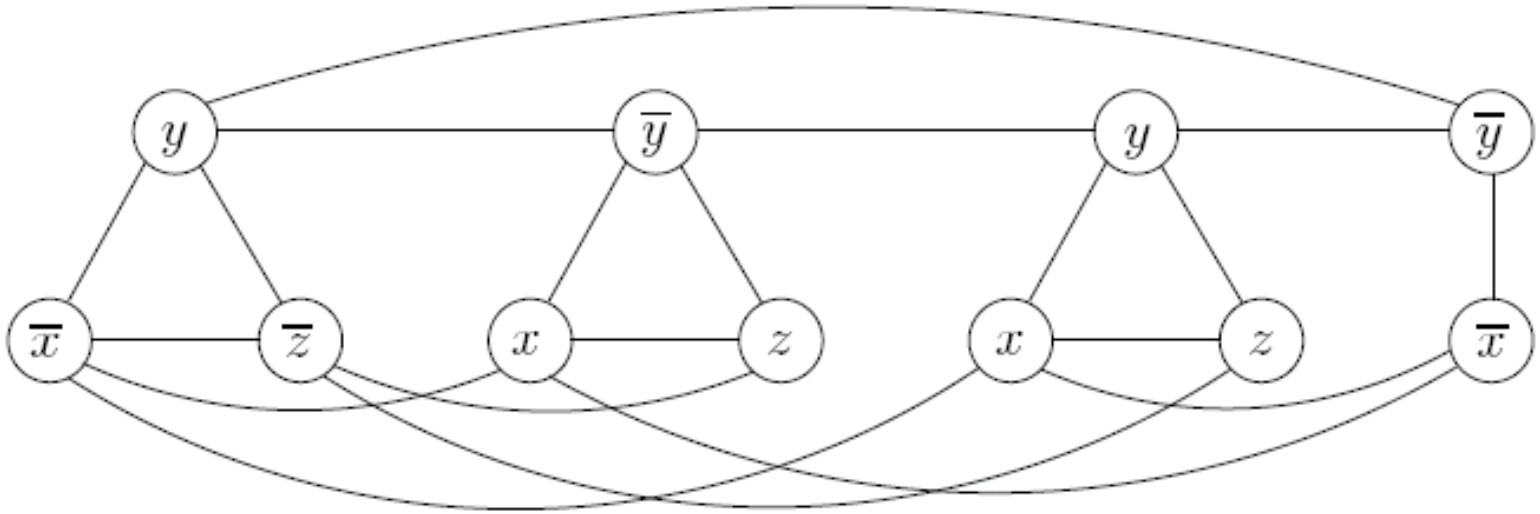
Example 2

- 3SAT → INDEPENDENT SET
  - *3SAT*: a set of clauses, each with **three or fewer** literals, find a satisfying truth assignment
  - *INDEPENDENT SET*: find a set of *g* pairwise **non-adjacent** vertices
  - Reduction maps an instance *I* of 3SAT into an instance (*G*, *g*) of INDEPENDENT SET:
    - Graph *G* has a triangle for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, and has additional edges between any two vertices that represent opposite literals
    - The goal *g* is set to the number of clauses

Example 2

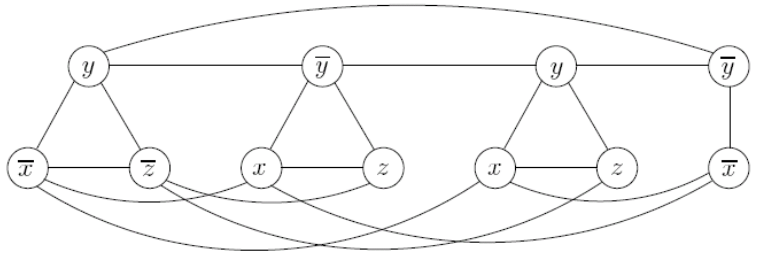
- 3SAT → INDEPENDENT SET

The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



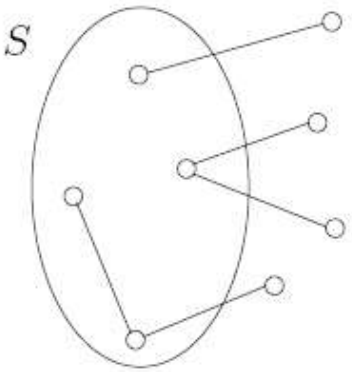
Example 2

The graph corresponding to $(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y})$.



- 3SAT → INDEPENDENT SET
  - Given an independent set *S* of *g* vertices in *G*, it is possible to efficiently recover a satisfying truth assignment to *I*
    - assign *x* a value of true if *S* contains a vertex labeled *x*;
    - assign *x* a value of false if *S* contains a vertex labeled $\overline{x}$;
    - assign true or false to *x* if *S* contains neither *x* or $\overline{x}$;
  - If graph *G* has no independent set of size *g*, then the Boolean formula *I* is unsatisfiable
    - Prove the contrapositive: *I* is satisfiable→ independent set of size *g*
      - For each clause, pick any literal whose value under the satisfying assignment is true, and add it to *S*

Example 3



- INDEPENDENT SET → VERTEX COVER
  - Reduction: a set of nodes *S* is a vertex cover of graph $G = (V, E)$ (*S* touches every edge in *E*) if and only if the remaining nodes, *V*-*S*, are an independent set of *G*
  - An instance (G, g) of INDEPENDENT SET reduces to a vertex cover of *G* with $|V|$-*g* nodes

- INDEPENDENT SET → CLIQUE
  - Reduction: a set of nodes *S* is an independent set of *G* if and only if *S* is a clique of $\bar{G} = (V, \bar{E})$,where $\bar{E}$ contains those unordered pairs of vertices that are not in *E*

Exercise 4

Give a simple reduction from 3D MATCHING to SAT, and another from RUDRATA CYCLE to SAT. (*Hint:* In the latter case you may use variables x_{ij} whose intuitive meaning is "vertex i is the jth vertex of the Hamilton cycle"; you then need to write clauses that express the constraints of the problem.)

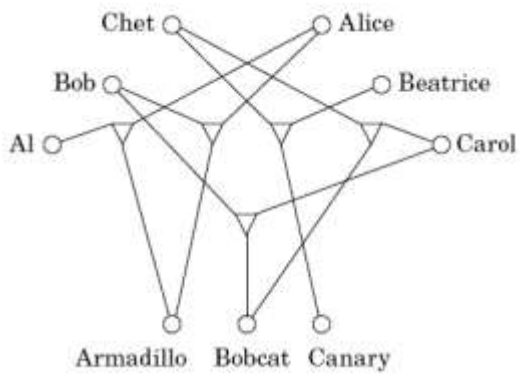
Exercise 4

3D-MATCHING to SAT

We have a variable x_{bgp} for each given triple (b, g, p). We interpret $x_{bgp} = \texttt{true}$ as choosing the triple (b, g, p). Suppose for boy b, $(b, g_1, p_1), \ldots, (b, g_k, p_k)$ are triples involving him. Then we add the clause $(x_{bg_1p_1} \vee \ldots \vee x_{bg_kp_k})$ so that at least one of the triples is chosen. Similarly for the triples involving each girl or pet. Also, for each pair of triples involving a common boy, girl or pet, say (b_1, g, p_1) and (b_2, g, p_2), we add a clause of the form $(\bar{x}_{b_1gp_1} \vee \bar{x}_{b_1gp_1})$ so that at most one of the triples is chosen.

The total number of triples, and hence the number of variables, is at most n^3. The first type of clauses involve at most n^2 variables and we add $3n$ such clauses, one for each boy, girl or pet. The second type of clauses involve triples sharing one common element. There are $3n$ ways to choose the common element and at most n^4 to choose the rest, giving at most $3n^5$ clauses. Hence, the size of the new formula is polynomial in the size of the input.

If there is a matching, then it must involve n triples $(b_1, g_1, p_1), \ldots, (b_n, g_n, p_n)$. Setting the variables $x_{b_1g_1p_1}, \ldots, x_{b_ng_np_n}$ to `true` and the rest to `false` gives a satisfying assignment to the above formula. Similarly, choosing only the triples corresponding to the `true` variables in any satisfying assignment must correspond to a matching for the reasons mentioned above. Hence, the formula is satisfiable if and only if the given instance has a 3D matching.



Exercise 4

RUDRATA CYCLE to SAT

We introduce variables x_{ij} for $1 \leq i, j \leq n$ meaning that the ith vertex is at the jth position in the Rudrata cycle. Each vertex must appear at some position in the cycle. Thus, for every vertex i, we add the clause $x_{i1} \vee x_{i2} \vee \ldots \vee x_{in}$. This adds n clauses with n variables each.

Also, if the ith vertex appears at the jth position, then the vertex at $(j+1)$th position must be a neighbor of i. In other words, if u, v are *not* neighbors, then either u appears at the jth position, or v appears at the $(j+1)$th position, but not both. Thus for every $(u, v) \notin E$ and for all $1 \leq j \leq n$, add the clause $(\bar{x}_{uj} \vee (\bar{x}_{v(j+1)})$. This adds at most $O(n^2) \times n = O(n^3)$ clauses with 2 variables each.

Using the "meanings" of the clauses given above, it is easy to see that every satisfying assignment gives a Rudrata cycle and vice-versa.

Approximation Algorithms and Schemes

Let C_{opt} be the cost of the optimal algorithm for a problem of size n. An approximation algorithm for this problem has an approximation ratio $\varrho(n)$ if, for any input, the algorithm produces a solution of cost C such that:

$$max(\frac{C}{C_{opt}}, \frac{C_{opt}}{C}) \leq \varrho(n)$$

Such an algorithm is called a $\varrho(n)$-approximation algorithm.

An approximation scheme that takes as input $\epsilon > 0$ and produces a solution such that $C = (1 + \epsilon)C_{opt}$ for any fixed ϵ, is a $(1 + \epsilon)$-approximation algorithm.

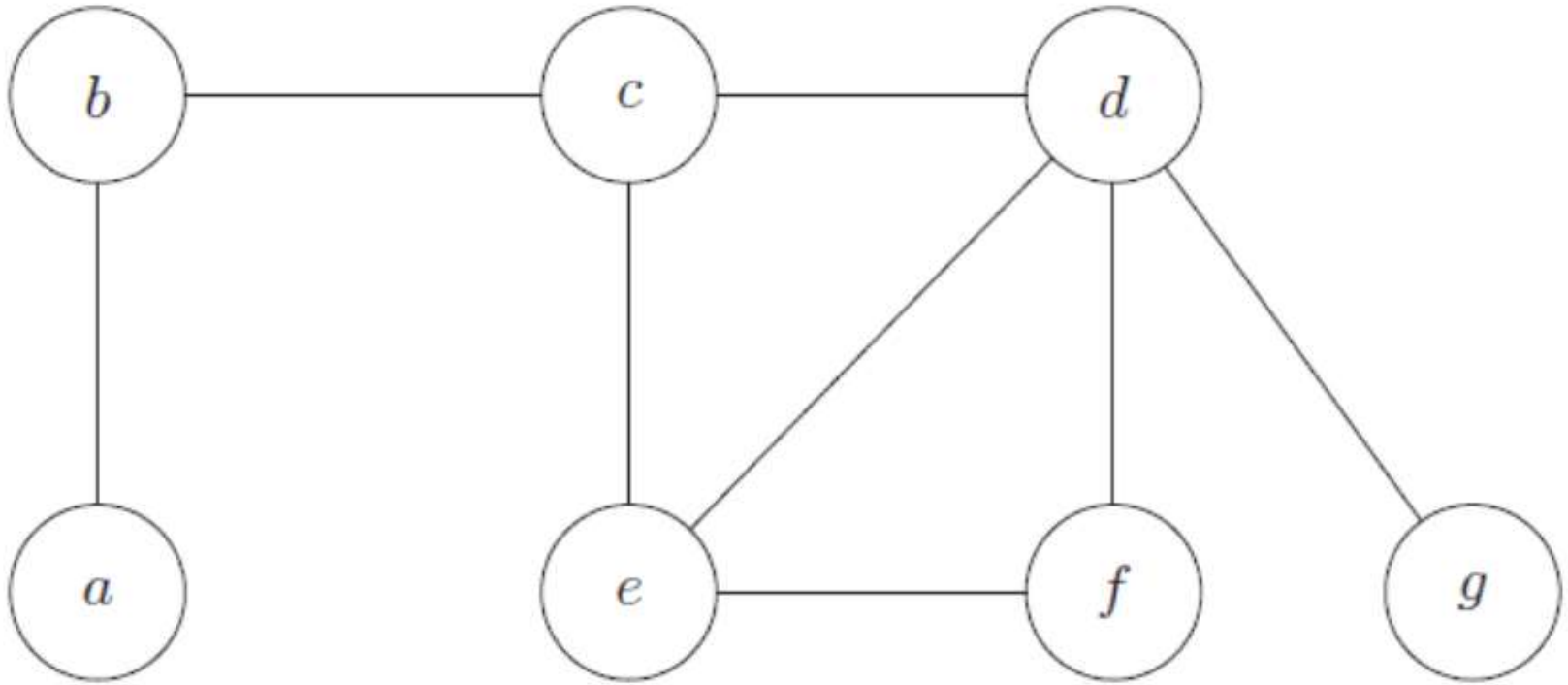
Vertex Cover

Given an undirected graph $G(V, E)$, find a subset $V' \subseteq V$ such that, for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$ (or both). Furthermore, find a V' such that $|V'|$ is minimum. This is an NP-Complete problem.

Approximation Algorithm For Vertex Cover

Here we define algorithm *Approx_Vertex_Cover*, an approximation algorithm for Vertex Cover. Start with an empty set V'. While there are still edges in E, pick an edge (u, v) arbitrarily. Add both u and v into V'. Remove all edges incident on u or v. Repeat until there are no more edges left in E. *Approx_Vertex_Cover* runs in polynomial time.

Take for example the following graph G:



Approx_Vertex_Cover could pick edges (b, c), (e, f) and (d, g), such that $V' = \{b, c, e, f, d, g\}$ and $|V'| = 6$. Hence, the cost is $C = |V'| = 6$. The optimal solution for this example is $\{b, d, e\}$, hence $C_{opt} = 3$.

Claim: *Approx_Vertex_Cover* is a 2-approximation algorithm.

Proof: Let $U \subseteq E$ be the set of all the edges that are picked by *Approx_Vertex_Cover*. The optimal vertex cover must include at least one endpoint of each edge in U (and other edges). Furthermore, no two edges in U share an endpoint. Therefore, $|U|$ is a lower bound for C_{opt}. i.e. $C_{opt} \geq |U|$. The number of vertices in V' returned by *Approx_Vertex_Cover* is $2 \cdot |U|$. Therefore, $C = |V'| = 2 \cdot |U| \leq 2C_{opt}$. Hence $C \leq 2 \cdot C_{opt}$. $\square$

Set Cover

Given a set X and a family of (possibly overlapping) subsets $S_1, S_2, \cdots, S_m \subseteq X$ such that $\cup_{i=1}^{m} S_i = X$, find a set $P \subseteq \{1, 2, 3, \cdots, m\}$ such that $\cup_{i \in P} S_i = X$. Furthermore find a P such that $|P|$ is minimum.

Set Cover is an NP-Complete problem.

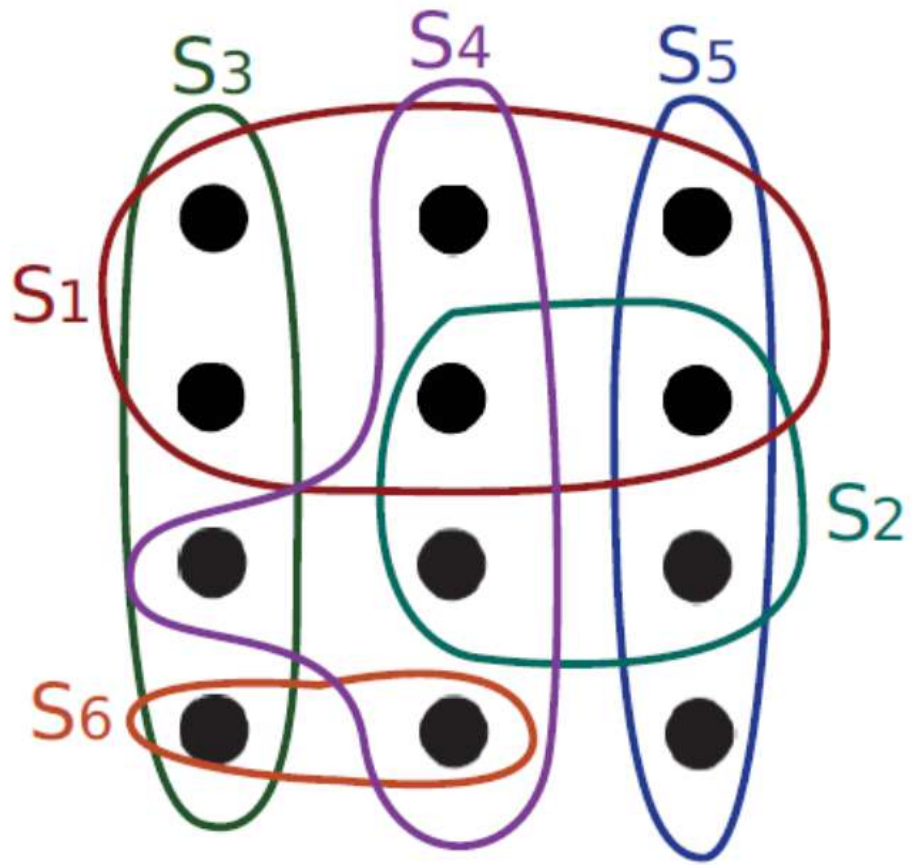
Approximation Algorithm for Set Cover

Here we define algorithm *Approx_Set_Cover*, an approximation algorithm for Set Cover. Start by initializing the set P to the empty set. While there are still elements in X, pick the largest set S_i and add i to P. Then remove all elements in S_i from X and all other subsets S_j. Repeat until there are no more elements in X. *Approx_Set_Cover* runs in polynomial time.

Repeat until all elements of B are covered:

Pick the set S_i with the largest number of uncovered elements.

In the following example, each dot is an element in X and each S_i are subsets of X.



Approx_Set_Cover selects sets S_1, S_4, S_5, S_3 in that order. Therefore it returns $P = \{1, 4, 5, 3\}$ and its cost $C = |P| = 4$. The optimal solution is $P_{opt} = \{S_3, S_4, S_5\}$ and $C_{opt} = |P_{opt}| = 3$.