Most of NP-completeness is not concerned with algorithms for decision problems, but with relationships between decision problems. In particular, how an algorithm for one decision problem reduces to (can be used to solve) another decision problem.

Figure corresponds to the notation: \( A \leq_P B \)
6. (10 pts.) Using the Bellman-Ford algorithm as a subroutine, write an algorithm in pseudocode to determine if a directed graph $G$ contains a cycle. What is the running time of your algorithm?

**Cycle-Test($G$)**

$G' = G$

Add a new vertex $s$ to $G'$

for each vertex $v \in G.V$, add edge $(s, v)$ to $G'$

for each edge $(u, v) \in G'.E$, $w(u, v) = \neg 1$

return $\neg$Bellman-Ford($G'$, $w$, $s$)

The running time is $\Theta(VE)$, the time to run the Bellman-Ford algorithm.

7. (20 pts.) In pseudocode, write an algorithm to count all the simple paths in a graph, including paths of length 0. Don’t worry about creating an efficient algorithm. Hint: Write it recursively with one parameter equal to the vertices not in the current path. What is the running time of your algorithm?

**Count-Paths($G$)**

$count = 0$

for each vertex $v \in G.V$

\[ count = count + \text{Count-Paths-Recurse}(G, v, G.V - \{v\}) \]

return $count$

**Count-Paths-Recuse($G, u, S$)**

$count = 1$  // count the path that made it here

for each vertex $v \in G.Adj[u]$

\[ \text{if } v \in S \]

\[ count = count + \text{Count-Paths-Recuse}(G, v, S - \{v\}) \]

return $count$

The number of simple paths is at most $V + V \cdot (V - 1) + V \cdot (V - 1) \cdot (V - 2) + \ldots + V!$. This sum is less than $3(V!)$. Each call involves an $O(V)$ loop, so the running time is $O(V \cdot V!)$. 


$A = \text{Does a directed graph } G \text{ contain a cycle?}$

$B = \text{Given a directed, weighted graph } G, \text{ and a source vertex } s, \text{ does } s \text{ have a path to a negative cycle?}$

\[ \begin{array}{c|c|c|c}
\text{instance } \alpha \text{ of } A & \text{polynomial-time reduction algorithm} & \text{instance } \beta \text{ of } B & \text{polynomial-time algorithm to decide } B \\
\hline
\text{yes} & \text{yes} & \text{no} & \text{no} \\
\hline
\text{need to add weights} & \text{Bellman-Ford} (G',w',s) & \text{need to arrange them properly} & \\
\end{array} \]

\[ A = \text{Does a 3SAT formula } \phi \text{ satisfiable?} \]

\[ B = \text{Does a undirected } G \text{ contain a clique of size } k? \]

\[ \begin{array}{c|c|c|c|c|c}
\text{instance } \alpha \text{ of } A & \text{polynomial-time reduction algorithm} & \text{instance } \beta \text{ of } B & \text{polynomial-time algorithm to decide } B \\
\hline
\text{yes} & \text{yes} & \text{no} & \text{no} \\
\hline
\text{create a vertex} & \text{choose } k & \text{vertices from number of} & \text{vertices from} & \text{connected if} \text{ literals don't contradict} \\
\text{each time a literal} & \text{the same clause clauses} & \text{different clauses} & \text{not connected} & \\
\text{appears,} & \text{not connected} & \text{not connected} & \text{not connected} & \\
\end{array} \]
Problem Reduction Sequence

Any NP problem

CIRCUIT-SAT

SAT

3-CNF-SAT

3SAT

CLIQUE

3SAT ≤ P CLIQUE

VERTEX-COVER

CLIQUE ≤ P

VERTEX-COVER

SUBSET-SUM

3SAT ≤ P

SUBSET-SUM

CLIQUE

HAM-CYCLE

TSP

CS 3343 Analysis of Algorithms

NP-Completeness – 5
We define 3-CNF satisfiability using the following terms. A literal in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in conjunctive normal form, or CNF, if it is expressed as an AND of clauses, each of which is the OR of one or more literals. A boolean formula is in 3-conjunctive normal form, or 3-CNF, if each clause has exactly three distinct literals.

For example, the boolean formula

\[(x_1 \lor \neg x_1 \lor \neg x_2) \land (x_3 \lor x_2 \lor x_4) \land (\neg x_1 \lor \neg x_3 \lor \neg x_4) \land C_4 \land C_5 \land \ldots\]

is in 3-CNF. The first of its three clauses is \(x_1 \lor \neg x_1 \lor \neg x_2\), which contains the three literals \(x_1\), \(\neg x_1\), and \(\neg x_2\).

In 3-CNF-SAT, we are asked whether a given boolean formula \(\phi\) in 3-CNF is satisfiable. \(\text{Is there an assignment that makes } \phi \text{ true?}\)

- \(x_1 \lor \neg x_1 \lor \neg x_2\) is a clause
- \(x_3 \lor x_2 \lor x_4\) and \(\ldots\)
  - each clause has contains 3 literals
- \(x_1\) is a literal
- \(\neg x_1\) is a literal
- a literal is a variable or its negation
A clique in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in $E$. In other words, a clique is a complete subgraph of $G$. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph. As a decision problem, we ask simply whether a clique of a given size $k$ exists in the graph. The formal definition is:

$$\text{CLIQUE} = \{(G, k) : G \text{ is a graph containing a clique of size } k\}$$

---

**Diagrams:**
- Clique of size 3
- Clique of size 4
Reducing 3SAT to CLIQUE

\[ \phi = C_1 \land C_2 \land C_3 \]

\[ C_1 = x_1 \lor \neg x_2 \lor \neg x_3 \]

\[ C_2 = \neg x_1 \lor x_2 \lor x_3 \]

\[ C_3 = x_1 \lor x_2 \lor x_3 \]