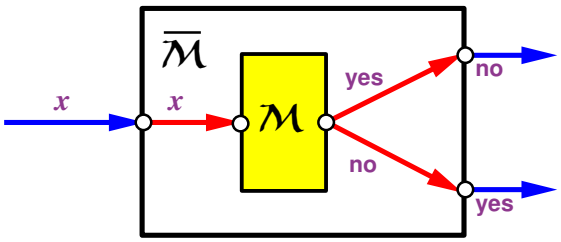
The Complement Classes

- **Given a complexity class C, its complement, denoted as co-C, is the set of languages that are complements of the languages in C.**
- Consider the set Reachability. It contains all triples (G, a, b) that have directed paths from a to b in G.
- The complement of Reachability is the set that contains all triples (G, a, b) such that there are not directed paths from a to b in directed graph G.
- **If C is a deterministic time or deterministic space complexity class, then C = co-C.** Hence, **P = co-P** and **PSPACE = co-PSPACE**.



Let $L \in$ **C**. Then, there is a Turing machine $\mathcal{M}$ that *halts* within its resource bound for every string, whether in L or not. The complement Turing machine $\bar{\mathcal{M}}$ is constructed above. $\bar{\mathcal{M}}$ accepts string $x \in$ **co-C** if and only if $\mathcal{M}$ rejects x. Hence, $\mathcal{M}$ decides **C** while $\bar{\mathcal{M}}$ decides **co-C**.

The Complement Classes: co-NP

- **The class co-NP contains all languages whose complements are in NP**. More precisely, these are the problems that we can quickly guess a "no" answer and quickly verify it.
- **Example 1:**
  1. Consider the Hamiltonian-Circuit problem (HC). It is easy to guess a set of vertices and verify that if they form a directed circuit.
  2. The *complement Hamiltonian-Circuit* problem (co-HC) asks if a given directed graph contains **no** Hamiltonian-circuit. Because the complement of HC is in **NP**, co-HC is in **co-NP**.
  3. Note that it is not easy to show co-HC in **NP**. (Actually, we do not know if it is true!) Given a directed graph, a single guess of a list of vertices that does not form a circuit **does not** guarantee the given graph has no Hamiltonian-Circuit!

- **Example 2:**
  1. **VALIDITY of Boolean Expressions:** Is a given Boolean expression a tautology (*i.e.*, always satisfiable)?
  2. The complement of this problem is: Is there an assignment of truth values so that the result is false?
  3. Thus, the complement is in **NP**. Compare with the SAT problem and you will see the similarity.
  4. Therefore, the Validity of Boolean Expression is in **co-NP**.
  5. Note that we do not know if this problem is in **NP** because only one guess that can satisfy the given expression does not mean the expression is a tautology.
- **Conclusion:** Problems in **co-NP** are those ones that can quickly be **disqualified**.

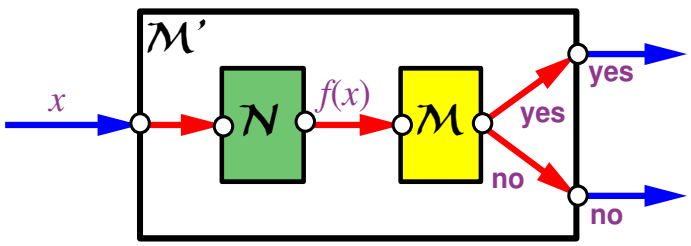
Some Simple Results

1. **If $A \leq_{\mathbf{p}} B$ and B is in $\mathbf{P}$ (*resp.*, $\mathbf{NP}$) A is also in $\mathbf{P}$ (*resp.*, $\mathbf{NP}$).**

   (a) If $A \leq_{\mathrm{p}} B$, there is a polynomial-time computable function f such that $x \in A$ if and only if $f(x) \in B$.

   (b) If B is in $\mathbf{P}$ (*resp.*, $\mathbf{NP}$), there is a deterministic (*resp.*, non-deterministic) Turing machine $\mathcal{M}$ that accepts B and a polynomial $p(x)$ such that for every $x \in B$ there is an accepting computation in $\leq p(|x|)$ steps.

   (c) Let $\mathcal{N}$ be a Turing machine that computes f. Construct a new Turing machine $\mathcal{M}'$ as follows: $\mathcal{M}'$ takes x, sends it to $\mathcal{N}$ which delivers $f(x)$, and sends $f(x)$ to $\mathcal{M}$.



   (d) Thus, the total number of steps is still polynomial. Consequently, $L \in \mathbf{P}$ if $\mathcal{M}$ is deterministic or $L \in \mathbf{NP}$ if $\mathcal{M}$ is non-deterministic.

2. **If $A \leq_{\mathbf{p}} B$, then $\bar{A} \leq_{\mathbf{p}} \bar{B}$.** As a result, we have **co-P = P $\leq_{\mathrm{p}}$ co-NP**

   (a) Since $A \leq_{\mathrm{p}} B$, there is a polynomial-time computable function f such that $x \in A \Leftrightarrow f(x) \in B$.

   (b) Since $x \notin A \Leftrightarrow f(x) \notin B$, we have $x \in \bar{A} \Leftrightarrow f(x) \in \bar{B}$.

   (c) Therefore, **co-P $\leq_{\mathrm{p}}$ co-NP**.

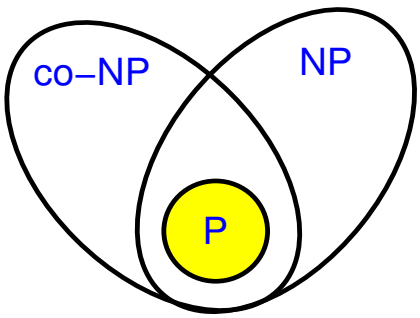
   (d) Since **P = co-P**, we also have **P $\leq_{\mathbf{p}}$ co-NP**.

3. **co-P $\subseteq$ co-NP**

   (a) If $L \in$ **co-P**, then $\bar{L} \in$ **P**.

   (b) Since **P $\subseteq$ NP**, we have $\bar{L} \in$ **NP** and hence $L \in$ **co-NP**.

4. **Since P = co-P, we have P $\subseteq$ NP $\cap$ co-NP as shown below.**



   We do not know if **P = NP** nor **NP = co-NP**. However, if **P = NP**, then **P = NP = co-NP**.

5. **NP = co-NP if and only if the complement of an NP-complete language is in NP.**

   More precisely, **NP** = **co-NP** if and only if we can find a L in **NP**-complete such that $\bar{L} \in \mathbf{NP}$ holds.

   (a) ($\Rightarrow$) If **NP** = **co-NP**, then every **NP**-complete language L is also in **co-NP**. Because $\bar{L} \in \mathbf{co\text{-}NP}$, we have $L \in \mathbf{NP}$.

   (b) ($\Leftarrow$) We need to prove $\mathbf{NP} \subseteq \mathbf{co\text{-}NP}$ and $\mathbf{co\text{-}NP} \subseteq \mathbf{NP}$. Suppose L is **NP**-complete and $\bar{L} \in \mathbf{NP}$.

   - $\mathbf{NP} \subseteq \mathbf{co\text{-}NP}$
     - Let $Q \in \mathbf{NP}$.
     - Since L is **NP**-complete, $Q \leq_{\mathrm{p}} L$, and $\bar{Q} \leq_{\mathrm{p}} \bar{L}$.
     - Since $\bar{Q} \leq_{\mathrm{p}} \bar{L}$ and $\bar{L} \in \mathbf{NP}$, $\bar{Q} \in \mathbf{NP}$.
     - Therefore, $Q \in \mathbf{co\text{-}NP}$ and $\mathbf{NP} \subseteq \mathbf{co\text{-}NP}$.
   - $\mathbf{co\text{-}NP} \subseteq \mathbf{NP}$
     - Let $Q \in \mathbf{co\text{-}NP}$. Thus, $\bar{Q} \in \mathbf{NP}$.
     - Since L is **NP**-complete, we have $\bar{Q} \leq_{\mathrm{p}} L$ and $Q \leq_{\mathrm{p}} \bar{L}$.
     - Since $\bar{L} \in \mathbf{NP}$, we have $Q \in \mathbf{NP}$ and $\mathbf{co\text{-}NP} \subseteq \mathbf{NP}$.

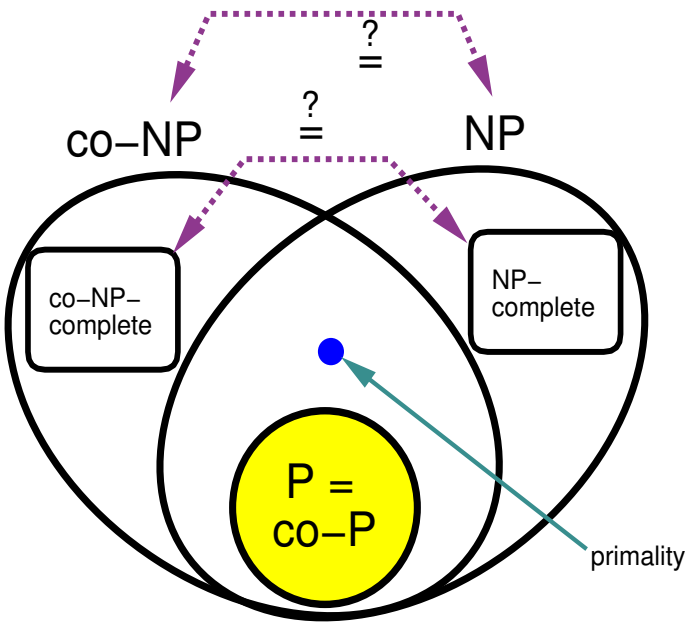
Co-NP-Completeness

- **co-NP-Hard: A language L is co-NP-hard if for every language $Q \in$ co-NP, $Q \leq_{\mathbf{p}} L$.**
- **co-NP-complete: A language L is co-NP-complete if and only if $L \in$ co-NP and L is co-NP-hard.**
- The classes of **NP**-complete and **co-NP**-complete languages are denoted as **NP-complete** and **co-NP-complete**, respectively.
- **$L \in$ NP-complete $\Leftrightarrow$ $\bar{L} \in$ co-NP-complete.**
  1. Let $L \in$ **NP-complete**. We shall prove $\bar{L} \in$ **co-NP-complete**.
  2. For every $Q \in$ **co-NP**, we have $\bar{Q} \in$ **NP**.
  3. Since L is **NP**-complete, $\bar{Q} \leq_{\mathrm{p}} L$ and $Q \leq_{\mathrm{p}} \bar{L}$.
  4. Therefore, $\bar{L}$ is **co-NP**-complete. The other direction is similar.
- **NP = co-NP $\Leftrightarrow$ a co-NP-complete language is in NP $\Leftrightarrow$ co-NP-complete $\cap$ NP $\neq \emptyset$.**

  This result was proved earlier.

The Class NP ∩ co-NP

- **Primality: Given an n-bit integer, is it a prime number?**
- **Primality** is in **NP** ∩ **co-NP**. (V. R. Pratt)
- We do not know if **Primality** is in **P** or in **NP-complete** or **co-NP-complete**. However, a quantum computer can find a non-trivial factor, if there is any, in $O(n^3)$ time with $O(1)$ certainty.
- We have the following relation so far:



- We do not know if **NP** = **co-NP** or **NP-complete** = **co-NP-complete**.
- Prove yourself **co-NP** ⊆ **PSPACE**.

The Counting-Reachability Theorem

- **Given a directed graph $G = (V, E)$ and a vertex a, the number of vertices reachable from a can be computed by a non-deterministic Turing machine in space $O(\log_2 |V|)$.** (Immerman-Szelepscényi).
- Define $\mathsf{Reachable}(d)$ to be the set of vertices that are *reachable* from the given vertex a with a path of length $\leq d$. We will only use $|\mathsf{Reachable}(k)|$, the number of elements in the set rather than the set itself.
- The basic idea is to compute $\mathsf{Reachable}(d)$ from $\mathsf{Reachable}(d-1)$. This is illustrated below.

```
|Reachable(0)| := 1;
FOR d := 1 TO |V| - 1 DO
   Compute |Reachable(d)| from |Reachable(d-1)|;
```

The above computes all vertices that are reachable with a path of length d from those vertices that are reachable with a path of length $d-1$. More precisely, **we find a path of length $d-1$ and "extend" it to a path of length d.**

- How do we compute Reachable(d) from Reachable($d-1$)? The following is a possible way:

```
Count := 0;
FOR v := 1 TO |V| DO
    IF v is in Reachable(d) THEN
        Count := Count + 1;
```

We go through *every* vertex to find out if some of them can be reached from a with a path of length $\leq d$. If one is found, variable Count is increased by one. We have not yet discussed how to compute Reachable(d) from Reachable($d-1$); but, this will become clear soon. So far, we have a program like the following:

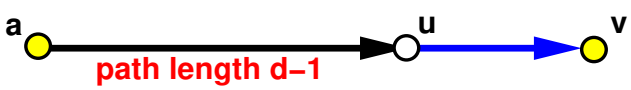
```
|Reachable(0)| := 1;
FOR d := 1 TO |V|-1 DO
   BEGIN
      Count := 0;
      FOR v := 1 TO |V| FO
         IF v is in Reachable(d) THEN
            Count := Count + 1;
   END;
```

The *unfinished* part is marked in blue.

- Now, it comes the hard part. We design a function `v-in-Reachable-d()` for the test "v **is in** Reachable(d)."

```
BOOLEAN FUNCTION  v-in-Reachable-d(v,d,|Reachable(d-1)|)
BEGIN
   New_Count := 0;
   Test_Result := FALSE;
   FOR u := 1 TO |V| DO
      IF ((u is in Reachable(d-1)) THEN
         BEGIN
            New_Count := New_Count + 1;
            IF ((u = v) OR (u ⟶ v is an edge)) THEN
               Test_Result := TRUE;
         END;
   IF (New_Count < |Reachable(d-1)|) THEN
      GIVE-UP  /* Turing machine says "NO" */
   ELSE
      Test_Result := TRUE;
   RETURN  Test_Result;
END FUNCTION   v-in-Reachable-d;
```

  - Variable `New_Count` re-counts all vertices that can be reached from a with a path of length $\leq d-1$.
  - Find all vertices u in Reachable($d-1$) such that $u = v$ holds or $u \rightarrow v$ is an edge. Thus, a path of length $d-1$ is "extended" to a path of length d that can reach v.



  - If the "re-count" disagrees with Reachable($d-1$), stops and says "no" because the non-deterministic guess (see below) is not quite reliable.

- Finally, how do we implement "u **is in** Reachable$(d-1)$"? We should not use the same technique, because we would enter a recursion that will break the $\log_2 |V|$ space limit. Instead, we use a non-deterministic guess. The following function `u-in-Reachable-d-1()` implements the test "u **is in** Reachable$(d-1)$":

```
BOOLEAN FUNCTION  u-in-Reachable-d-1(a,u,d)
BEGIN
   Start := a;
   FOR p := 1 TO d-1 DO
      BEGIN
         Guess non-deterministically End;
         IF (NOT((Start = End) OR
                 (Start --> End is an edge))) THEN
            GIVE-UP; /* Turing machine says "no"*/
         Start := End;
      END;
   IF (End = u) THEN
      RETURN  TURE
   ELSE
      GIVE-UP; /* Turing machine says "no" */
END FUNCTION  u-in-Reachable-d-1;
```

  - This function builds a path of length $\leq d-1$ non-deterministically by guessing all vertices between a and u.
  - If such path exists, this function returns `TRUE` which means u is in Reachable$(d-1)$; otherwise, gives up and the Turing machine says "no."

- **Space Usage:**
  1. There is no recursion and the number of variables used is a constant.
  2. Since a variable is in the range of 0 to $|V|$, it uses $\log_2 |V|$ tape squares (*i.e.*, bits).
  3. Therefore, the total space consumption is $O(\log_2 |V|)$.

- **Correctness: The algorithm correctly computes Reachable(d) for all d**
  - The algorithm is obviously correct when $d = 0$.
  - Suppose $d \geq 1$.
  - Consider a successful computation, a computation that is not rejected by the test

    `New_Count < |Reachable(d-1)|`

    This means we assume the non-deterministic guesses in `u-in-Reachable-d-1()` build good paths!
  - This is the *only* place a computation may fail given successful guesses in function `u-in-Reachable-d-1()`. Hence, all $u \in$ Reachable($d-1$) were verified successfully, `Test_Result` accurately records whether $v \in$ Reachable(d), and variable `Count` in the "main" program (shown below) is increased correctly. Thus, this program is correct.

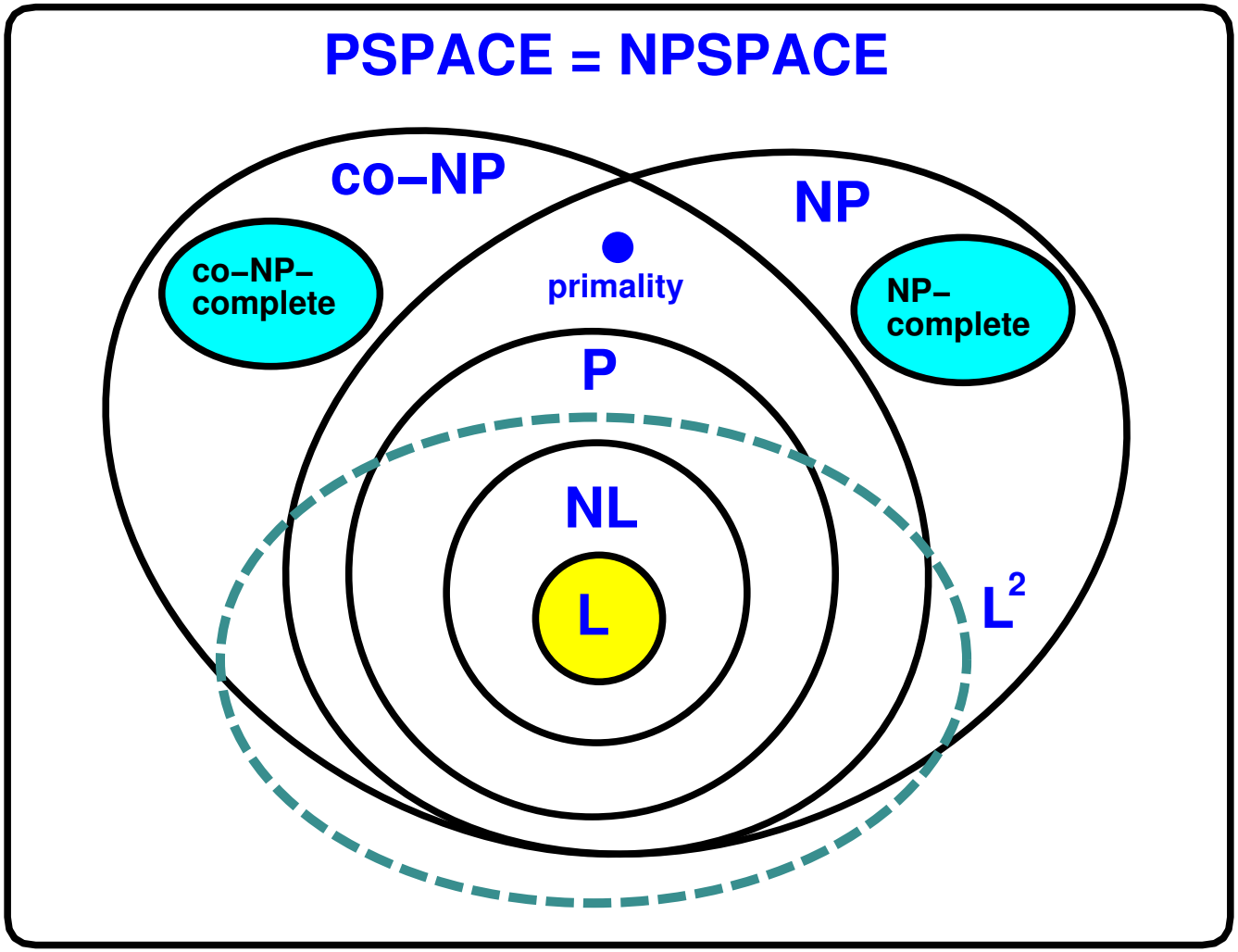
```
|Reachable(0)| := 1;
FOR d := 1 TO |V|-1 DO
   BEGIN
      Count := 0;
      FOR v := 1 TO |V| FO
         IF v is in Reachable(d) THEN
            Count := Count + 1;
   END;
```

  - Finally, it is easy to see that there is at least one correct guess (*i.e.*, successful computation). Thus, our proof completes.

NSPACE = co-NSPACE

- **NSPACE**$(f(n))$ = **co-NSPACE**$(f(x))$ **for** $f \geq \log_2$
- **NSPACE**$(f(x))$ contains all languages that are accepted by a non-deterministic Turing machine with a space bound $O(f(x))$, and **co-NSPACE**$(f(x))$ is the complement of **NSPACE**$(f(x))$.
- **NSPACE**$(f(n)) \subseteq$ **co-NSPACE**$(f(x))$ for $f \geq \log_2$
  1. Let $L \in$ **NSPACE**$(f(n))$ and $\mathcal{M}$ a non-deterministic Turing machine that accepts strings of L with a space bound $O(f(|x|))$, where x is an input string.
  2. We shall construct a Turing machine $\bar{\mathcal{M}}$ that accepts $\bar{L}$ with the same space bound.
  3. Given a $x \in L$, construct the configuration graph of $\mathcal{M}$ running x.
  4. As for $\bar{\mathcal{M}}$, x is on the input tape and the computations of $\mathcal{M}$ are simulated on one of $\bar{\mathcal{M}}$'s work tapes.

5. $\bar{\mathcal{M}}$ runs a slightly modified version of the **Counting-Reachability** algorithm:
   - If `v-in-Reachable-d()` returns TRUE for a vertex v that is a halting accepting configuration of $\mathcal{M}$, then $\bar{\mathcal{M}}$ halts and rejects the string.
   - If the "main" program completes successfully without rejecting any string, then $\bar{\mathcal{M}}$ halts and accepts the input string because every possible accepting computation for the input string has been examined and none of them is accepting.
   - $\bar{\mathcal{M}}$ is non-deterministic.
6. The space used by $\bar{\mathcal{M}}$ is the space needed for **Counting-Reachability**, which is $O(\log_2 |V|)$, where V is the set of vertices of the configuration graph. Since the configuration graph has $O(k^{f(n)+\log_2 n})$ vertices and since $f(n) \geq \log_2 n$, the configuration graph has $O(k^{f(n)})$ vertices and the space bound of $\bar{\mathcal{M}}$ is $O(\log_2(k^{f(n)})) = O(f(n))$.
7. As a result, $\bar{L} \in$ **co-NSPACE**$(f(n))$ and hence **NSPACE**$(f(n)) \subseteq$ **co-NSPACE**$(f(x))$

- A similar reasoning gives the other direction.

What we have so far