CSCI 4470/6470 Algorithms, Fall 2015

Lecture Note V

Liming Cai, Department of Computer Science, UGA

December 7, 2015

Part VII. Selected Topics

Part VII. Selected Topics

Chapter 34 NP-Completeness

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models
   - nondeterministic computation $=$ certificate $+$ verification

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models
   - nondeterministic computation = certificate + verification
3. NP-completeness framework

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models
   - nondeterministic computation = certificate + verification
3. NP-completeness framework
   - reduction, polynomial-time reduction

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models
   - nondeterministic computation = certificate + verification
3. NP-completeness framework
   - reduction, polynomial-time reduction
4. NP-completeness proof

Part VII. Selected Topics

Chapter 34 NP-Completeness

1. Intractable problems
   - decision versions of optimization problems
2. Nondeterministic computational models
   - nondeterministic computation = certificate + verification
3. NP-completeness framework
   - reduction, polynomial-time reduction
4. NP-completeness proof
   - NP-complete problems, reduction techniques.

Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST

Chapter 34. NP-Completeness

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$,

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.
- why would a time $O(n^{100})$-time algorithm be attractive?

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.
- why would a time $O(n^{100})$-time algorithm be attractive?

  only theoretical?

Chapter 34 NP-Completeness

1. Intractable problems

- we have seen many problems solvable in polynomial time
  e.g., sorting, SCC, MST
- there are problems that do not seem to have polynomial time algorithms
  i.e., not solvable in time $O(n \log n)$, $O(n^3)$, or $O(n^{100})$.
- why would a time $O(n^{100})$-time algorithm be attractive?

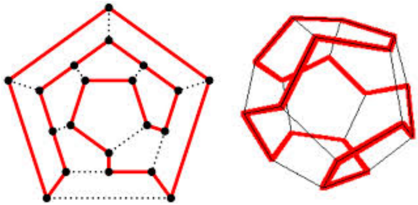
  only theoretical? practical significance as well

Chapter 34. NP-Completeness

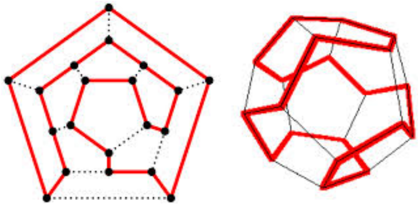
Define: a Hamiltonian cycle in a graph is a circular path going **through every vertex exactly once**.

Chapter 34. NP-Completeness

Define: a Hamiltonian cycle in a graph is a circular path going **through every vertex exactly once**.

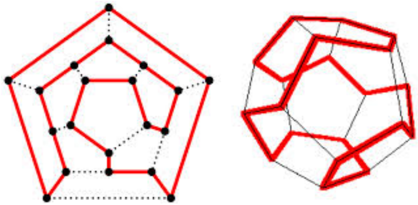


Define: a Hamiltonian cycle in a graph is a circular path going **through every vertex exactly once**.

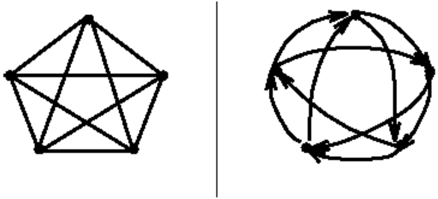


Different from Eulerian cycle that goes **through every edge exactly once**.

Define: a Hamiltonian cycle in a graph is a circular path going **through every vertex exactly once**.



Different from Eulerian cycle that goes **through every edge exactly once**.



Chapter 34. NP-Completeness

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;

Travel Salesman Problem (TSP)
Input: an edge-weighted graph $G = (V, E)$;
Output: a Hamiltonian cycle of the minimum weight sum.

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of $(v_1, v_2, \ldots, v_n)$ or simply a permutation of $(1, 2, \ldots, n)$, where $|V| = n$.

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of $(v_1, v_2, \dots, v_n)$ or simply a permutation of $(1, 2, \dots, n)$, where $|V| = n$. so the problem has time

upper bound $O(n!|E|)$, exponential time.

Travel Salesman Problem (TSP)
Input: an edge-weighted graph $G = (V, E)$;
Output: a Hamiltonian cycle of the minimum weight sum.

- intuitively, a circular path is a permutation of $(v_1, v_2, \ldots, v_n)$ or simply a permutation of $(1, 2, \ldots, n)$, where $|V| = n$. so the problem has time

upper bound $O(n!|E|)$, exponential time.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \geq n \times (n-1) \cdots \times \frac{n}{2} \geq (\frac{n}{2})^{\frac{n}{2}}$$

- all known algorithms (solving TSP) are of exponential-time.

Chapter 34. NP-Completeness

Chapter 34. NP-Completeness

Instead of considering

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Chapter 34. NP-Completeness

Instead of considering

Travel Salesman Problem (TSP)
  Input: an edge-weighted graph $G = (V, E)$;
  Output: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-Cycle Weight Decision (HCW)
  Input: an edge-weighted graph $G = (V, E)$, a weight value K;
  Output: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially,

Chapter 34. NP-Completeness

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP

Chapter 34. NP-Completeness

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\Longrightarrow$ P-time algorithms for HCW,

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\Longrightarrow$ P-time algorithms for HCW, **why?**

Chapter 34. NP-Completeness

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\Longrightarrow$ P-time algorithms for HCW, **why?**

How to prove:

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
  INPUT: an edge-weighted graph $G = (V, E)$;
  OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
  INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
  OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\Longrightarrow$ P-time algorithms for HCW, **why?**

How to prove: P-time algorithms for TSP

Chapter 34. NP-Completeness

Instead of considering

TRAVEL SALESMAN PROBLEM (TSP)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: a Hamiltonian cycle of the minimum weight sum.

We may consider a related problem:

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- HCW appears to "easier" than TSP as an H-cycle is not produced in the answer.
- However, HCW may not be "easier"

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Trivially, P-time algorithms for TSP $\Longrightarrow$ P-time algorithms for HCW, **why?**

How to prove: P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW ?

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for $\mathrm{TSP} \Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$ "YES".
2. mark all edges in G as "unvisited";

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
   pick an "unvisited" edge (u, v), mark it "visited";

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for $\mathrm{TSP} \Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
      pick an "unvisited" edge (u, v), mark it "visited";
      let $G' = G - \{(u, v)\}$;

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
   pick an "unvisited" edge (u, v), mark it "visited";
   let $G' = G - \{(u, v)\}$;
   if $A(G', k_{min}) =$ "YES"

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
      pick an "unvisited" edge (u, v), mark it "visited";
      let $G' = G - \{(u, v)\}$;
      if $A(G', k_{min}) =$ "YES"
         then $G = G'$;

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".

2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
     pick an "unvisited" edge (u, v), mark it "visited";
     let $G' = G - \{(u, v)\}$;
     if $A(G', k_{min}) =$ "YES"
       then $G = G'$;
       else mark (u, v) "critical";

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for $\mathrm{TSP} \Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".

2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
      pick an "unvisited" edge (u, v), mark it "visited";
      let $G' = G - \{(u, v)\}$;
      if $A(G', k_{min}) =$ "YES"
         then $G = G'$;
         else mark (u, v) "critical";
   return (all "critical" edges)

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$ "YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
      pick an "unvisited" edge (u, v), mark it "visited";
      let $G' = G - \{(u, v)\}$;
      if $A(G', k_{min}) =$ "YES"
         then $G = G'$;
         else mark (u, v) "critical";
   return (all "critical" edges)

- show algorithm B runs in P-time.

Chapter 34. NP-Completeness

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for TSP $\Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$ "YES/ "NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

  1. on input G, **for** every possible values of K, **call** $A(G, K)$;
     remember the smallest k_{min} such that $A(G, k_{min}) =$ "YES".
  2. mark all edges in G as "unvisited";
     while there are "unvisited" edges in G
        pick an "unvisited" edge (u, v), mark it "visited";
        let $G' = G - \{(u, v)\}$;
        if $A(G', k_{min}) =$ "YES"
           then $G = G'$;
           else mark (u, v) "critical";
     return (all "critical" edges)

- show algorithm B runs in P-time. How to make Step 1 P-time?

Theorem 1: HCW is solvable in P-time **if and only if** TSP is solvable in P-time.

Proof: (P-time algorithms for $\mathrm{TSP} \Longleftarrow$ P-time algorithms for HCW)

- assume P-time algorithm A for HCW such that $A(G, K) =$"YES/"NO"
- construct a P-time algorithm $B(G)$ for TSP to behave as follows:

1. on input G, **for** every possible values of K, **call** $A(G, K)$;
   remember the smallest k_{min} such that $A(G, k_{min}) =$"YES".
2. mark all edges in G as "unvisited";
   while there are "unvisited" edges in G
   pick an "unvisited" edge (u, v), mark it "visited";
   let $G' = G - \{(u, v)\}$;
   if $A(G', k_{min}) =$ "YES"
   then $G = G'$;
   else mark (u, v) "critical";
   return (all "critical" edges)

- show algorithm B runs in P-time. How to make Step 1 P-time?

Theorem 1 says problems HCW and TSP are "polynomially equivalent".

Consider another related problem:

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;

Chapter 34. NP-Completeness

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- Which problem is seemingly "easier"?

Consider another related problem:

H-CYCLE DECISION (HC)
 INPUT: an edge-weighted graph $G = (V, E)$;
 OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

H-CYCLE WEIGHT DECISION (HCW)
 INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
 OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- Which problem is seemingly "easier"?

Theorem 2: HCW is P-time solvable **if and only if** HC is P-time solvable.

Consider another related problem:

H-CYCLE DECISION (HC)
INPUT: an edge-weighted graph $G = (V, E)$;
OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

H-CYCLE WEIGHT DECISION (HCW)
INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

- Which problem is seemingly "easier"?

Theorem 2: HCW is P-time solvable **if and only if** HC is P-time solvable.

Can you prove it?

Consider another related problem:

H-CYCLE DECISION (HC)
 INPUT: an edge-weighted graph $G = (V, E)$;
 OUTPUT: "YES" if and only there is a Hamiltonian cycle in G.

Compared with

H-CYCLE WEIGHT DECISION (HCW)
 INPUT: an edge-weighted graph $G = (V, E)$, a weight value K;
 OUTPUT: "YES" if and only there is a Hamiltonian cycle of weight $\leq K$ in G.

• Which problem is seemingly "easier"?

Theorem 2: HCW is P-time solvable **if and only if** HC is P-time solvable.

Can you prove it?

Theorem 2 says problems HCW and HC are "polynomially equivalent".

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

Max Independent Set (MaxIS)
Input: graph $G = (V, E)$;

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

Max Independent Set (MaxIS)
Input: graph $G = (V, E)$;
Output: an independent set of vertices of the maximum size;

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

MAX INDEPENDENT SET (MAXIS)
INPUT: graph $G = (V, E)$;
OUTPUT: an independent set of vertices of the maximum size;

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

MAX INDEPENDENT SET (MAXIS)
INPUT: graph $G = (V, E)$;
OUTPUT: an independent set of vertices of the maximum size;

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

MAX INDEPENDENT SET (MAXIS)
INPUT: graph $G = (V, E)$;
OUTPUT: an independent set of vertices of the maximum size;

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

Theorem 4: MAXIS is P-time solvable **if and only if** IS is P-time solvable.

Corollary 3: Problems TSP, HCW, and HC are all "polynomially equivalent".

There are other problems that have the similar situation.

MAX INDEPENDENT SET (MAXIS)
INPUT: graph $G = (V, E)$;
OUTPUT: an independent set of vertices of the maximum size;

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

Theorem 4: MAXIS is P-time solvable **if and only if** IS is P-time solvable.

Can you prove the theorem?

Similarly,

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;
OUTPUT: a vertex cover set of vertices of the minimum size;

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;
OUTPUT: a vertex cover set of vertices of the minimum size;

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;
OUTPUT: a vertex cover set of vertices of the minimum size;

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has a vertex cover of size $\leq k$.

Chapter 34. NP-Completeness

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;
OUTPUT: a vertex cover set of vertices of the minimum size;

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has a vertex cover of size $\leq k$.

Theorem 5: MINVC is P-time solvable **if and only if** VC is P-time solvable.

Similarly,

MIN VERTEX COVER (MINVC)
INPUT: graph $G = (V, E)$;
OUTPUT: a vertex cover set of vertices of the minimum size;

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has a vertex cover of size $\leq k$.

Theorem 5: MINVC is P-time solvable **if and only if** VC is P-time solvable.

Can you prove the theorem?

Conclusions:

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established between different decision problems,

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established between different decision problems, e.g.,

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established between different decision problems, e.g.,

   Corollary 6: VC is P-time solvable **if and only if** IS is P-time solvable.

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established between different decision problems, e.g.,

   Corollary 6: VC is P-time solvable **if and only if** IS is P-time solvable.

3. However, "Polynomial equivalency" does not tell us the tractability of the problems.

Conclusions:

1. "Polynomial equivalency" can be established between optimization problems and decision problems.

   To study tractability of optimization problems, often it suffices to investigate decision problems.
   (Decision problems are also called *languages*.)

2. "Polynomial equivalency" can also be established between different decision problems, e.g.,

   Corollary 6: VC is P-time solvable **if and only if** IS is P-time solvable.

3. However, "Polynomial equivalency" does not tell us the tractability of the problems.

4. We need a rigorous framework to study tractability via the notion "Polynomial equivalency".

Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Deterministic algorithms

2. Nondeterministic algorithms

Deterministic algorithms

- Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.

Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Deterministic algorithms

- Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.
- All algorithms we have seen so far are deterministic.

Chapter 34. NP-Completeness

2. Nondeterministic algorithms

Deterministic algorithms

- Given input data, a deterministic algorithm has its every step completely determined by the algorithm and data.
- All algorithms we have seen so far are deterministic.
- Every deterministic algorithm can be unfolded into a linear sequence of steps (when the input is given).

$M = -\infty$
$n = 3$
$i = 1$
check $1 \leq 3$
check $-\infty < 10$
$M = 10$
$i = 2$
check $2 \leq 3$
check $10 < 30$
$M = 30$
$i = 3$
check $3 \leq 3$
check $30 < 20$
$i = 4$
check $4 \leq 3$
return (30)

Unfolded when input $L = (10, 30, 20)$

MaxOfList(L)
$M = -\infty$
$n = length(L)$
for $i = 1$ **to** n
  if $M < L[i]$
    $M = L[i]$
return (M)

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps;

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices;

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers "YES" if one execution path leads to "YES".

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers "YES" if one execution path leads to "YES".
- the running time is the number of steps on a **longest path**.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers "YES" if one execution path leads to "YES".
- the running time is the number of steps on a **longest path**.
- if running time is n, there may be $\geq 2^n$ paths.

Chapter 34. NP-Completeness

A deterministic algorithm can be thought of a **linear path** of steps; each vertex uniquely determines its successor step.

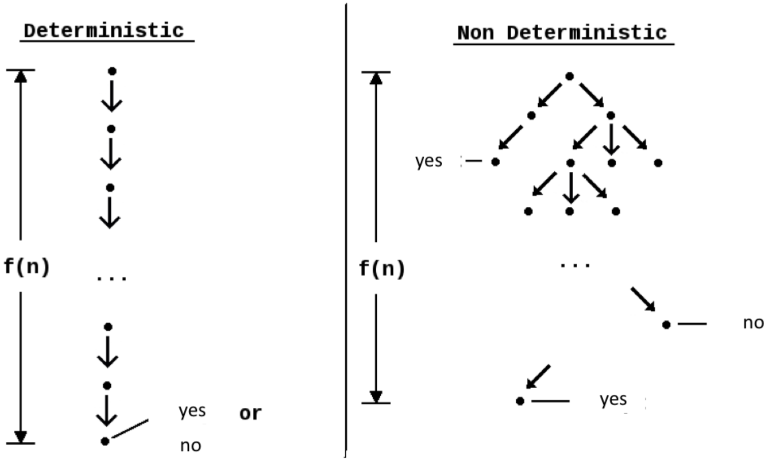
- the running time is the number of steps on the path.

In a nondeterministic algorithm, when unfolded, there may be more than one possible successor.

- a nondeterministic algorithm can be thought of a **tree** of steps.
- each step has more than one *nondeterministic* choice.
- a path from root to a leaf is a sequence of nondeterministic choices; thus a nondeterministic execution of the algorithm.
- algorithm answers "YES" if one execution path leads to "YES".
- the running time is the number of steps on a **longest path**.
- if running time is n, there may be $\geq 2^n$ paths.

Let us call this **tree model** of nondeterministic algorithms.

Chapter 34. NP-Completeness



Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE

Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

Use nondeterministic algorithms to solve problem Hamiltonian Cycle in polynomial time.

(1) starting from any vertex v in the graph;

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors which has not been chosen;

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors which has not been chosen;

let the newly picked vertex be v, **go to** step (2)

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors which has not been chosen;

let the newly picked vertex be v, **go to** step (2)

(3) if all vertices have been chosen,
return "YES" if their edges form an H-cycle;
return "NO" if their edges do NOT form an H-cycle;

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors which has not been chosen;

let the newly picked vertex be v, **go to** step (2)

(3) if all vertices have been chosen,
return "YES" if their edges form an H-cycle;
return "NO" if their edges do NOT form an H-cycle;

- The algorithm will answer "YES" iff there is a H-cycle in G.

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors which has not been chosen;

let the newly picked vertex be v, **go to** step (2)

(3) if all vertices have been chosen,
return "YES" if their edges form an H-cycle;
return "NO" if their edges do NOT form an H-cycle;

- The algorithm will answer "YES" iff there is a H-cycle in G. Because each path try one permutation of vertices.

Use nondeterministic algorithms to solve problem HAMILTONIAN CYCLE in polynomial time.

(1) starting from any vertex v in the graph;

(2) nondeterministically choose one of its (at most $n-1$) neighbors
which has not been chosen;

let the newly picked vertex be v, **go to** step (2)

(3) if all vertices have been chosen,
return "YES" if their edges form an H-cycle;
return "NO" if their edges do NOT form an H-cycle;

- The algorithm will answer "YES" iff there is a H-cycle in G.
  Because each path try one permutation of vertices.
- The algorithm runs in polynomial time as each path takes $O(n)$ steps.

Chapter 34. NP-Completeness

Problems like INDEPENDENT SET, VERTEX COVER, HCW can all be solved with nondeterministic algorithms in polynomial time.

Problems like INDEPENDENT SET, VERTEX COVER, HCW can all be solved with nondeterministic algorithms in polynomial time.

Can you prove the claim?

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$

Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

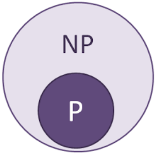
- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$



Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

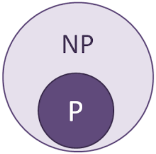
- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

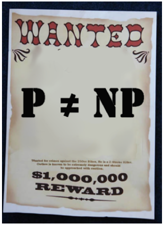
Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$





Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

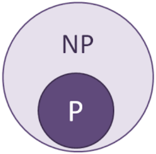
- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

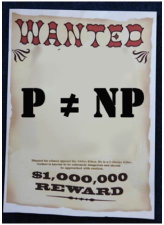
Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

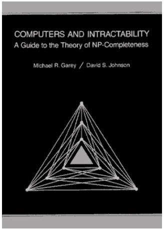
- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$







Chapter 34. NP-Completeness

Definition: $\mathcal{P}$ is the class of languages (i.e., decision problems) that can be solved by deterministic polynomial-time algorithms.

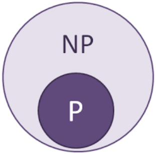
- class $\mathcal{P}$ contains problems like $\textsc{Reachability}$ and many others.

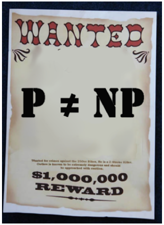
Definition: $\mathcal{NP}$ is the class of languages (i.e., decision problems) that can be solved by nondeterministic polynomial-time algorithms.

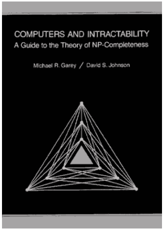
- class $\mathcal{NP}$ contains problems like VC, HC, IS and many others.

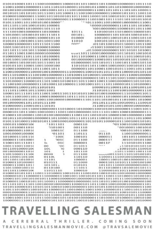
Because every deterministic algorithm is a special case of a nondeterministic algorithm,

$$\mathcal{P} \subseteq \mathcal{NP}$$

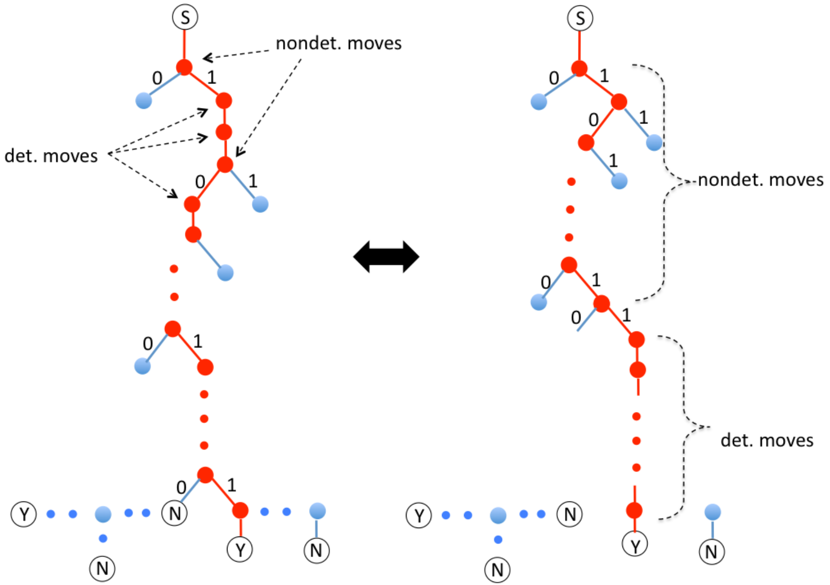








Chapter 34. NP-Completeness



Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string:

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)
- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices
  (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string:
  0 for branching left, 1 for right.

- we can assume the algorithm does **all** nondeterministic choices **before** other operations.

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices
  (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string:
  0 for branching left, 1 for right.

• we can assume the algorithm does **all** nondeterministic choices **before**
other operations. So we can model the computation as

We consider the **tree model** of nondeterministic algorithms.

- we may assume each step has exactly 2 nondeterministic choices (5 choices can be simulated with 4 nondeterministic steps)

- each nondeterministic path can be represented with a binary string: 0 for branching left, 1 for right.

- we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

• we may assume each step has exactly 2 nondeterministic choices
(5 choices can be simulated with 4 nondeterministic steps)

• each nondeterministic path can be represented with a binary string:
0 for branching left, 1 for right.

• we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

(1) first choose a binary string nondeterministically, and

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

• we may assume each step has exactly 2 nondeterministic choices
(5 choices can be simulated with 4 nondeterministic steps)

• each nondeterministic path can be represented with a binary string:
0 for branching left, 1 for right.

• we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

(1) first choose a binary string nondeterministically, and
(2) follow the specified path deterministically

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

• we may assume each step has exactly 2 nondeterministic choices
(5 choices can be simulated with 4 nondeterministic steps)

• each nondeterministic path can be represented with a binary string:
0 for branching left, 1 for right.

• we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

(1) first choose a binary string nondeterministically, and
(2) follow the specified path deterministically

The binary string is called **certificate** or **witness**;
The deterministic computation part is called **verification**.

Chapter 34. NP-Completeness

We consider the **tree model** of nondeterministic algorithms.

• we may assume each step has exactly 2 nondeterministic choices
(5 choices can be simulated with 4 nondeterministic steps)

• each nondeterministic path can be represented with a binary string:
0 for branching left, 1 for right.

• we can assume the algorithm does **all** nondeterministic choices **before** other operations. So we can model the computation as

(1) first choose a binary string nondeterministically, and
(2) follow the specified path deterministically

The binary string is called **certificate** or **witness**;
The deterministic computation part is called **verification**.

Deterministic algorithms are when the certificate is empty.

Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_{Π}, and a constant $c > 0$, such that

Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_Π, and a constant $c > 0$, such that

(1) if x is a positive instance of Π, **there is** a binary string y of length n^c,
$A_\Pi(x, y) =$ "YES";

Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_Π, and a constant $c > 0$, such that

(1) if x is a positive instance of Π, **there is** a binary string y of length n^c,
$A_\Pi(x, y) =$ "YES";

(2) if x is a negative instance of Π, **for all** binary string y of length n^c,
$A_\Pi(x, y) =$ "NO";

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_Π, and a constant $c > 0$, such that

(1) if x is a positive instance of Π, **there is** a binary string y of length n^c,
$A_\Pi(x, y) =$ "YES";
(2) if x is a negative instance of Π, **for all** binary string y of length n^c,
$A_\Pi(x, y) =$ "NO";
and A_Π runs in time $O(n^c)$.

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_Π, and a constant $c > 0$, such that

(1) if x is a positive instance of Π, **there is** a binary string y of length n^c,
$A_\Pi(x, y) =$ "YES";
(2) if x is a negative instance of Π, **for all** binary string y of length n^c,
$A_\Pi(x, y) =$"NO";
and A_Π runs in time $O(n^c)$.

We call y a **certificate**/**witness** and A_Π the **verification algorithm**.

Chapter 34. NP-Completeness

Alternative view of nondeterministic polynomial-time computation

Every nondeterministic polynomial time computation is

- to nondeterministically choose a binary string of a polynomial length,
- then to compute deterministically in polynomial time.

Let $\Pi \in \mathcal{NP}$. Then there is a **deterministic** algorithm A_Π, and a constant $c > 0$, such that

(1) if x is a positive instance of Π, **there is** a binary string y of length n^c,
$A_\Pi(x, y) =$ "YES";
(2) if x is a negative instance of Π, **for all** binary string y of length n^c,
$A_\Pi(x, y) =$ "NO";
and A_Π runs in time $O(n^c)$.

We call y a **certificate**/**witness** and A_Π the **verification algorithm**.

$\mathcal{P}$ is defined with certificate $y = \epsilon$, i.e., empty string.

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$.

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L,

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L$$

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y,$$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \le |x|^c,$$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what?

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x, y|$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x, y| = |x| + |y|$

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time m^d

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n + n^c)$

Chapter 34. NP-Completeness

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n + n^c)^d$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n + n^c)^d \leq (2n^c)^d$

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n+n^c)^d \leq (2n^c)^d = O(n^{dc})$,

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n+n^c)^d \leq (2n^c)^d = O(n^{dc})$, also polynomial time of $n = |x|$.

Definition of $\mathcal{NP}$ in terms of languages:

Let $L \subseteq \{0,1\}^*$ be a language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

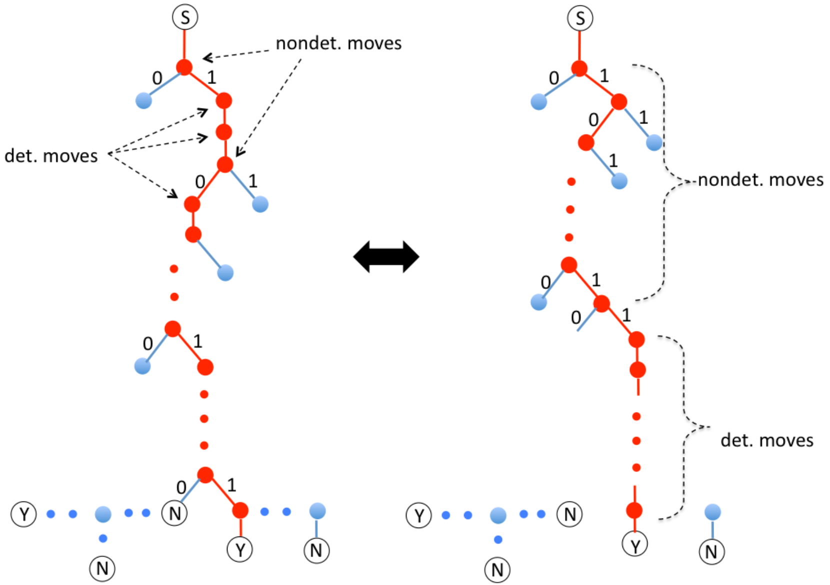
and A_L runs in polynomial time.

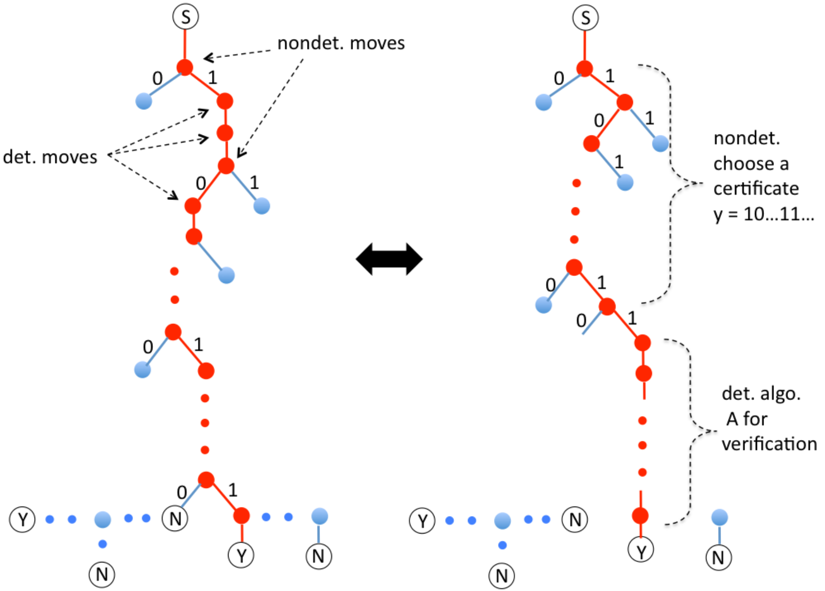
A_L runs in polynomial time of what? in $m = |x,y| = |x| + |y| \leq n + n^c$.

So if A_L runs in polynomial time $m^d \leq (n+n^c)^d \leq (2n^c)^d = O(n^{dc})$, also polynomial time of $n = |x|$.

Class $\mathcal{P}$ is defined with certificate $y = \epsilon$, i.e., empty string.

Chapter 34. NP-Completeness




S
nondet. moves
det. moves
0
1
Y
N
nondet.
choose a
certificate
y = 10...11...
det. algo.
A for
verification

Proof that $\mathrm{HC} \in \mathcal{NP}$.

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"}$$

We can design that

- certificate y represents a sequence of ordered vertices;

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{“YES”}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \iff \exists y, |y| \leq |G|^c, A(G, y) = \text{“YES”}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Details:

- $y = B_1 B_2 \ldots B_n$, where B_i is a binary representation of some vertex in G;

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{“YES”}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Details:

- $y = B_1 B_2 \ldots B_n$, where B_i is a binary representation of some vertex in G; How many bits does B_i need?

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{“YES”}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Details:

- $y = B_1 B_2 \dots B_n$, where B_i is a binary representation of some vertex in G; How many bits does B_i need? $\lceil \log_2 n \rceil$

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{"YES"}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Details:

- $y = B_1 B_2 \dots B_n$, where B_i is a binary representation of some vertex in G; How many bits does B_i need? $\lceil \log_2 n \rceil$
- whether y forms a H-cycle can be verified in time

Proof that $\mathrm{HC} \in \mathcal{NP}$.

We need to show there is a deterministic algorithm A and a constant $c > 0$, such that for any G,

$$G \in \mathrm{HC} \Longleftrightarrow \exists y, |y| \leq |G|^c, A(G, y) = \text{“YES”}$$

We can design that

- certificate y represents a sequence of ordered vertices;
- algorithm A is to verify that y does form a H-cycle.

Details:

- $y = B_1 B_2 \ldots B_n$, where B_i is a binary representation of some vertex in G; How many bits does B_i need? $\lceil \log_2 n \rceil$
- whether y forms a H-cycle can be verified in time $O(|E|)$

Chapter 34. NP-Completeness

exercises:

Chapter 34. NP-Completeness

exercises:

Proof that Independent Set $\in \mathcal{NP}$.

Proof that Vertex Cover $\in \mathcal{NP}$.

Notes

exercises:

Proof that INDEPENDENT SET $\in \mathcal{NP}$.

Proof that VERTEX COVER $\in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$.

exercises:

Proof that $\textsc{Independent Set} \in \mathcal{NP}$.

Proof that $\textsc{Vertex Cover} \in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$.

2. there is a difference between deciding $x \in L$ and checking $A_L(x, y) = 1$.

exercises:

Proof that $\textsc{Independent Set} \in \mathcal{NP}$.

Proof that $\textsc{Vertex Cover} \in \mathcal{NP}$.

Notes

1. to prove a language is in the class $\mathcal{NP}$ by no mean to prove that the language can be solved in polynomial time. Instead, it only shows the language is in the class $\mathcal{NP}$.

2. there is a difference between deciding $x \in L$ and checking $A_L(x, y) = 1$.

3. as between convicting a suspect vs checking an evidence against the suspect.

3. NP-Completeness Framework

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

$$\overline{L} = \{x : x \notin L\}$$

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

$$\overline{L} = \{x : x \notin L\} \text{ called } \textit{complement} \text{ of } L$$

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

$$\overline{L} = \{x : x \notin L\} \text{ called } \textit{complement} \text{ of } L$$

$$L \cup \overline{L} = \{0,1\}^*$$

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

$$\overline{L} = \{x : x \notin L\} \text{ called } \textit{complement} \text{ of } L$$

$$L \cup \overline{L} = \{0,1\}^* = \mathcal{U},$$

3. NP-Completeness Framework

The notion of reduction (i.e., transformation) between languages

- We use languages for decision problems.
- A language contains positive instances of the corresponding decision problem.
- Define

$$\overline{L} = \{x : x \notin L\} \text{ called } \textit{complement} \text{ of } L$$

$$L \cup \overline{L} = \{0,1\}^* = \mathcal{U}, \text{ called } \textit{universe}$$

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$,

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

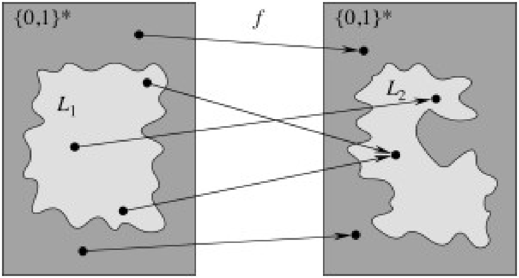
$$x \in L_1 \Longleftrightarrow f(x) \in L_2$$

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

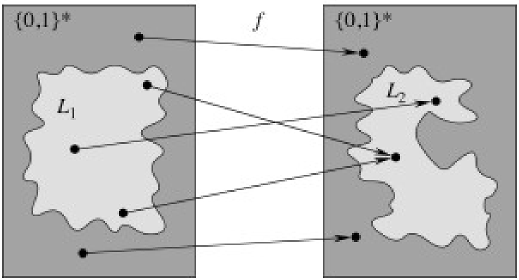


Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$



That is,

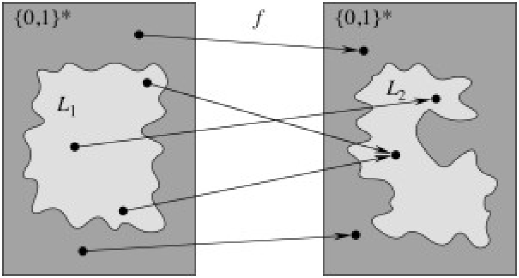
$x \in L_1 \implies f(x) \in L_2$;

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$



That is,

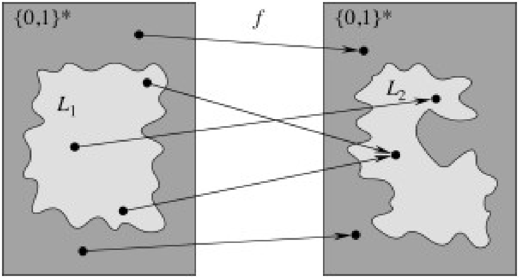
$x \in L_1 \implies f(x) \in L_2$; $x \in \overline{L_1} \implies f(x) \in \overline{L_2}$;

Chapter 34. NP-Completeness

Let L_1 and L_2 are two languages over the alphabet $\{0, 1\}$.

A reduction from L_1 to L_2, denoted as $L_1 \leq L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$



That is,

$x \in L_1 \Longrightarrow f(x) \in L_2$; $\qquad x \in \overline{L_1} \Longrightarrow f(x) \in \overline{L_2}$;

Chapter 34. NP-Completeness

Two example problems:

Chapter 34. NP-Completeness

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;

Chapter 34. NP-Completeness

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

and

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

and

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;

Two example problems:

Independent Set (IS)
Input: graph $G = (V, E)$, integer k;
Output: "YES" if and only if G has an independent set of size $\geq k$.

and

Vertex Cover (VC)
Input: graph $G = (V, E)$, integer k;
Output: "YES" if and only if G has a vertex cover of size $\leq k$.

Chapter 34. NP-Completeness

Two example problems:

Independent Set (IS)
  Input: graph $G = (V, E)$, integer k;
  Output: "YES" if and only if G has an independent set of size $\geq k$.

and

Vertex Cover (VC)
  Input: graph $G = (V, E)$, integer k;
  Output: "YES" if and only if G has a vertex cover of size $\leq k$.

Consider their corresponding languages:

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

and

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

Two example problems:

INDEPENDENT SET (IS)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has an independent set of size $\geq k$.

and

VERTEX COVER (VC)
INPUT: graph $G = (V, E)$, integer k;
OUTPUT: "YES" if and only if G has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Two example problems:

Independent Set (IS)
Input: graph $G = (V, E)$, integer k;
Output: "YES" if and only if G has an independent set of size $\geq k$.

and

Vertex Cover (VC)
Input: graph $G = (V, E)$, integer k;
Output: "YES" if and only if G has a vertex cover of size $\leq k$.

Consider their corresponding languages:

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Chapter 34. NP-Completeness

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof:

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof: we use the fact that complement set of an independent set is a vertex cover in the same graph

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof: we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping f that maps instance $\langle G, k\rangle$ to instance $\langle G, |G| - k\rangle$,

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof: we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping f that maps instance $\langle G, k\rangle$ to instance $\langle G, |G| - k\rangle$, i.e.,

$$f(\langle G, k\rangle)$$

Chapter 34. NP-Completeness

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof: we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping f that maps instance $\langle G, k\rangle$ to instance $\langle G, |G|$ - $k\rangle$, i.e.,

$$f(\langle G, k\rangle) = \langle G, |G| \text{ - } k\rangle$$

This is a reduction from L_{IS} to L_{VC} because

$$G \text{ has an i.s. of size } \geq k \iff G \text{ has an v.c. of size } \leq |G|\text{-}k$$

$$L_{IS} = \{\langle G, k\rangle : G \text{ has an independent set of size } \geq k\}$$

$$L_{VC} = \{\langle G, k\rangle : G \text{ has a vertex cover of size } \leq k\}$$

Because the two problems are very relevant to each other, we have:

Theorem: $L_{IS} \leq L_{VC}$

Proof: we use the fact that complement set of an independent set is a vertex cover in the same graph

We construct a mapping f that maps instance $\langle G, k\rangle$ to instance $\langle G, |G|$ - $k\rangle$, i.e.,

$$f(\langle G, k\rangle) = \langle G, |G| \text{ - } k\rangle$$

This is a reduction from L_{IS} to L_{VC} because

$$G \text{ has an i.s. of size } \geq k \iff G \text{ has an v.c. of size } \leq |G|\text{-}k$$

So $L_{IS} \leq L_{VC}$.

Chapter 34. NP-Completeness

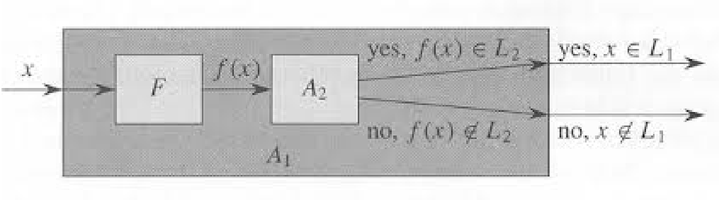
An important motivation for reduction:

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;

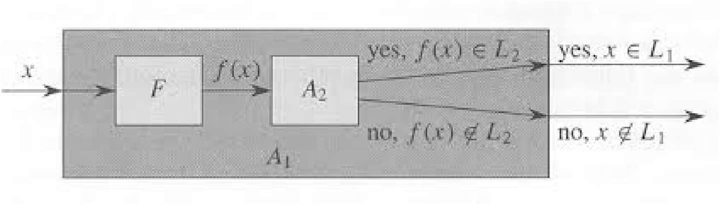


where algorithm F computes the reduction f,

Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;

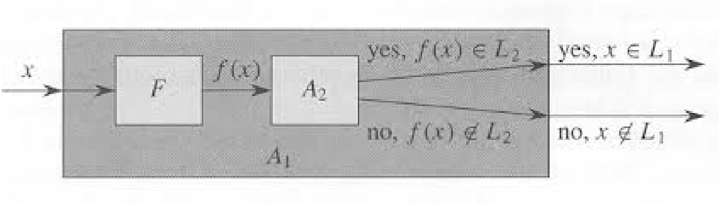


where algorithm F computes the reduction f,
and algorithm A_2 solves for L_2

Chapter 34. NP-Completeness

An important motivation for reduction:

- a reduction transforms instances of the first problem to the instances of the second problem;
- algorithms solving the second problem can be used to solve the first;



where algorithm F computes the reduction f,
and algorithm A_2 solves for L_2

So the combined algorithm (gray-color box) solves for L_1.

Chapter 34. NP-Completeness

Formally,

Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0,1\}^* \rightarrow \{0,1\}^*$,

Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0,1\}^* \to \{0,1\}^*$, such that

Chapter 34. NP-Completeness

Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$,
is some mapping function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, such that for any $x \in \{0,1\}^*$,

Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any $x \in \{0, 1\}^*$,

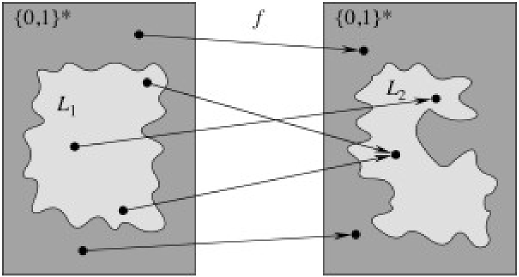
$$x \in L_1 \iff f(x) \in L_2$$

Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$, is some mapping function $f : \{0,1\}^* \rightarrow \{0,1\}^*$, such that for any $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

where f can be computed in time $O(|x|^c)$ for some fixed $c > 0$.

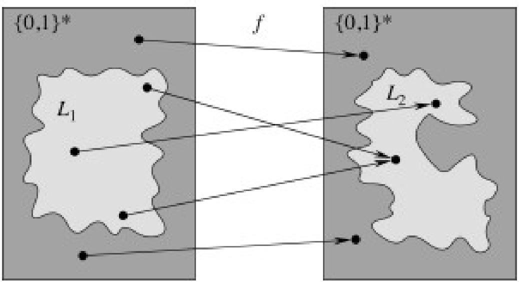


Formally,

A polynomial-time reduction from L_1 to L_2, denoted as $L_1 \leq_p L_2$,
is some mapping function $f : \{0,1\}^* \to \{0,1\}^*$, such that for any $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

where f can be computed in time $O(|x|^c)$ for some fixed $c > 0$.



For example, $L_{IS} \leq_p L_V$.

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

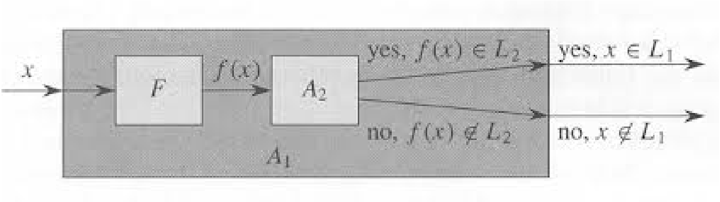
We need to show that the gray box runs in polynomial time

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:

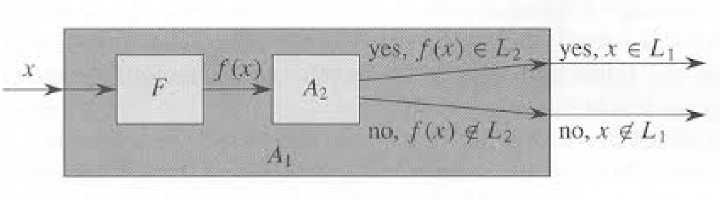


Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:

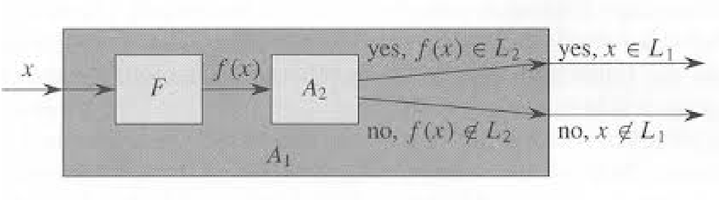


Total time is the sum of time for F and time for A_2.

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

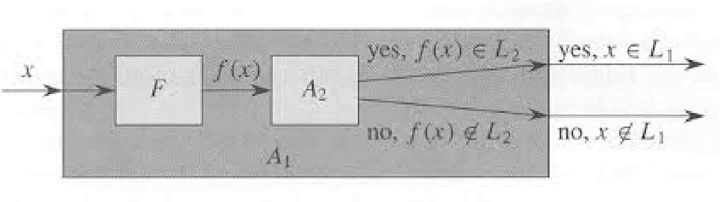
$$O(|x|^c) + O(|f(x)|^d)$$

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

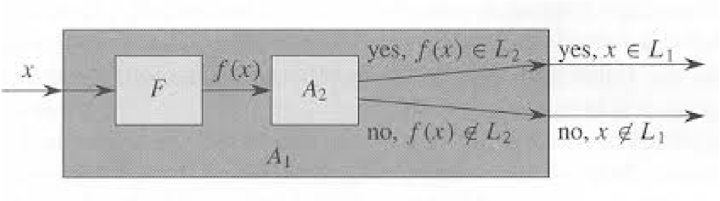
$O(|x|^c) + O(|f(x)|^d)$ now, what is the length of $f(x)$?

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

$$O(|x|^c) + O(|f(x)|^d) \text{ now, what is the length of } f(x)?$$

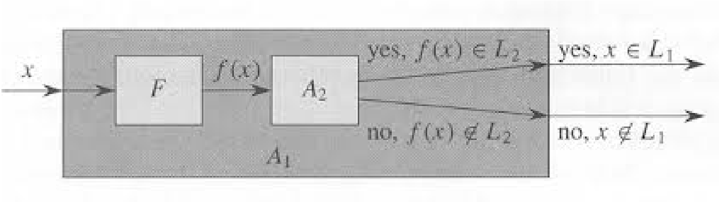
Because F runs in time $O(|x|^c)$,

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

$$O(|x|^c) + O(|f(x)|^d) \text{ now, what is the length of } f(x)?$$

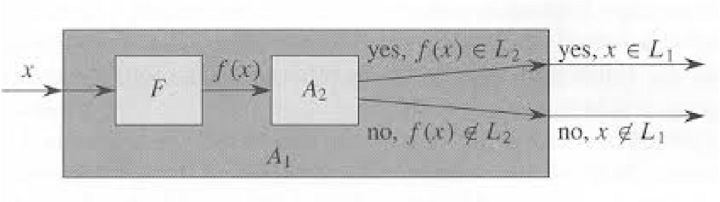
Because F runs in time $O(|x|^c)$, the number of bits outputted by F is $O(|x|^c)$.

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

$$O(|x|^c) + O(|f(x)|^d) \quad \text{now, what is the length of } f(x)?$$

Because F runs in time $O(|x|^c)$, the number of bits outputted by F is $O(|x|^c)$.
So

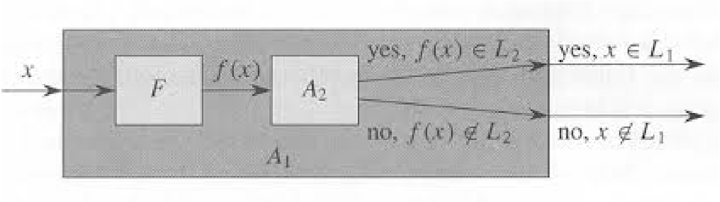
$$O(|x|^c) + O(|f(x)|^d)$$

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

$$O(|x|^c) + O(|f(x)|^d)$$ now, what is the length of $f(x)$?

Because F runs in time $O(|x|^c)$, the number of bits outputted by F is $O(|x|^c)$.
So

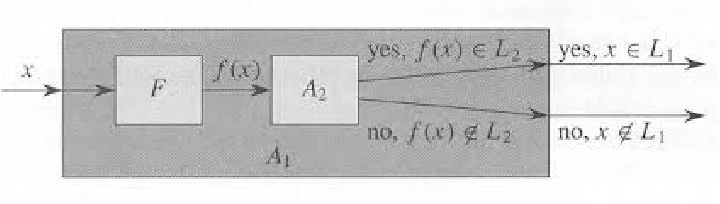
$$O(|x|^c) + O(|f(x)|^d) = O(|x|^c + O((|x|^c)^d)$$

Chapter 34. NP-Completeness

Theorem: Let $L_1 \leq_p L_2$. If $L_2 \in \mathcal{P}$, then $L_1 \in \mathcal{P}$.

Proof: Assume algorithm F computes f, and algorithm A_2 solves for L_2

We need to show that the gray box runs in polynomial time
if both F and A_2 runs in polynomial time:



Total time is the sum of time for F and time for A_2.

$O(|x|^c) + O(|f(x)|^d)$ now, what is the length of $f(x)$?

Because F runs in time $O(|x|^c)$, the number of bits outputted by F is $O(|x|^c)$.
So

$$O(|x|^c) + O(|f(x)|^d) = O(|x|^c + O((|x|^c)^d) = O(|x|^{cd})$$

Theorem: Polynomial-time reductions compose (are transitive). That is

If $L_1 \leq_p L_2$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is

If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$,

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$;

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1$$

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2$$

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1$

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.
But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \le_p L_2$ and $L_2 \le_p L_3$, then $L_1 \le_p L_3$.

Proof. Assume functions f for $L_1 \le_p L_2$; function h for $L_2 \le_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \le L_3$.
But we need to show the reduction is $\le_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.
But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm H computes h: $H(y) = h(y)$ in time $O(|y|^d)$

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.
But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm H computes h: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f)$ = time of F and time of H

$$= O(|x|^c) + O(|f(x)|^d)$$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$, then $L_1 \leq_p L_3$.

Proof. Assume functions f for $L_1 \leq_p L_2$; function h for $L_2 \leq_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \leq L_3$.
But we need to show the reduction is $\leq_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm H computes h: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f) =$ time of F and time of H

$$= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d)$$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \le_p L_2$ and $L_2 \le_p L_3$, then $L_1 \le_p L_3$.

Proof. Assume functions f for $L_1 \le_p L_2$; function h for $L_2 \le_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \le L_3$.
But we need to show the reduction is $\le_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm H computes h: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f) =$ time of F and time of H

$$= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d) = O(|x|^c) + O(|x|^{cd})$$

Chapter 34. NP-Completeness

Theorem: Polynomial-time reductions compose (are transitive). That is
If $L_1 \le_p L_2$ and $L_2 \le_p L_3$, then $L_1 \le_p L_3$.

Proof. Assume functions f for $L_1 \le_p L_2$; function h for $L_2 \le_p L_3$.

for every $x \in \{0,1\}^*$,

$$x \in L_1 \iff f(x) \in L_2 \iff h(f(x)) \in L_3$$

That is $x \in L_1 \iff h(f(x)) \in L_3$

So composite function $(h \circ f)$ realizes reduction $L_1 \le L_3$.
But we need to show the reduction is $\le_p$, i.e., a polynomial time reduction.

Assume that algorithm F computes f: $F(x) = f(x)$ in time $O(|x|^c)$
and algorithm H computes h: $H(y) = h(y)$ in time $O(|y|^d)$

Let $y = f(x)$, the total time for computing $(h \circ f)$ = time of F and time of H

$$= O(|x|^c) + O(|f(x)|^d) = O(|x|^c) + O((|x|^c)^d) = O(|x|^c) + O(|x|^{cd})$$

So $L_1 \le_p L_3$.

Some conclusions:

Some conclusions:

- Using $\leq_p$, languages in $\mathcal{NP}$ can be ordered partially;

Some conclusions:

- Using $\leq_p$, languages in $\mathcal{NP}$ can be ordered partially;
- If those languages at the end of a $\leq_p$ chain have polynomial-time algorithms, so does every language on the chain.

Some conclusions:

- Using $\leq_p$, languages in $\mathcal{NP}$ can be ordered partially;
- If those languages at the end of a $\leq_p$ chain have polynomial-time algorithms, so does every language on the chain.
- Informally, those at the end of a $\leq_p$ chain are called NP-hard.

Definition 1: L is NP-hard

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$,

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?
- If L is NP-hard

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?
- If L is NP-hard and $L \leq_p L'$,

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?
- If L is NP-hard and $L \leq_p L'$, then L' is NP-hard.

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?
- If L is NP-hard and $L \leq_p L'$, then L' is NP-hard.
  Proof?

Chapter 34. NP-Completeness

Definition 1: L is NP-hard if for every language $L' \in \mathcal{NP}$, $L' \leq_p L$.

Definition 1: L is NP-complete if (1) L is NP-hard and (2) $L \in \mathcal{NP}$.

Properties of NP-hard problems

- If L is NP-hard and $L \in \mathcal{P}$, then $\mathcal{P} = \mathcal{NP}$.
  Proof?
- If L is NP-hard and $L \leq_p L'$, then L' is NP-hard.
  Proof?

How to prove a language is NP-hard?

4. NP-Completeness Proofs

4. NP-Completeness Proofs

To prove a language L is NP-complete,

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard.

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP},\ L' \leq_p L$$

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP},\ L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to L.

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP},\ L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to L.
- Instead, formulate a generic language that represents all languages in NP

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP},\ L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to L.
- Instead, formulate a generic language that represents all languages in NP and prove that every language in $\mathcal{NP}$ can be reduced to the generic language in polynomial time.

4. NP-Completeness Proofs

To prove a language L is NP-complete, we need to show it is NP-hard. That is, we need to show

$$\text{for every language } L' \in \mathcal{NP},\ L' \leq_p L$$

- Apparently, it is not possible to enumerate all languages in NP and prove that everyone is polynomial-time reducible to L.
- Instead, formulate a generic language that represents all languages in NP and prove that every language in $\mathcal{NP}$ can be reduced to the generic language in polynomial time.
- To obtain such a generic language, we need to consider the definition of languages in NP.

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L,

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y,$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c,$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x, y) = 1$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \Longleftrightarrow \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

$$\Updownarrow$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

$$\Updownarrow$$

$$x \in L \iff f(x) \in L_{tbd}$$

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x,y) = 1$$

$$\Updownarrow$$

$$x \in L \Longleftrightarrow f(x) \in L_{tbd}$$

where L_{tbd} is a language to be defined.

Chapter 34. NP-Completeness

Recall the definition of languages in $\mathcal{NP}$:

Let $L \subseteq \{0,1\}^*$ be any language in the class $\mathcal{NP}$. Then there is a **deterministic** algorithm A_L, and a constant $c > 0$, such that, for every $x \in \{0,1\}^*$,

$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

and A_L runs in polynomial time.

The "iff" relationship looks a little like the relationship in a reduction

$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1$$

$$\Updownarrow$$

$$x \in L \iff f(x) \in L_{tbd}$$

where L_{tbd} is a language to be defined.

Can we identify L_{tbd} and f ?

Chapter 34. NP-Completeness

Again we examine

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \tag{1}$$

Chapter 34. NP-Completeness

Again we examine

$$x \in L \Longleftrightarrow \exists y, |y| \leq |x|^c, A_L(x, y) = 1 \tag{1}$$

- A_L is a deterministic algorithm can be implemented with a boolean circuit B_L with two sets of input gates $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ such that

$$A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \tag{2}$$

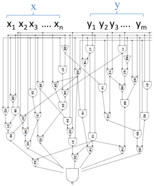
Chapter 34. NP-Completeness

Again we examine

$$x \in L \iff \exists y, |y| \leq |x|^c, A_L(x,y) = 1 \tag{1}$$

- A_L is a deterministic algorithm can be implemented with a boolean circuit B_L with two sets of input gates $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ such that

$$A_L(x,y) = 1 \text{ if and only if } B_L(x,y) = 1 \tag{2}$$



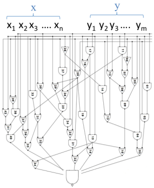
Chapter 34. NP-Completeness

Again we examine

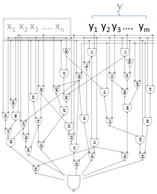
$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1 \tag{1}$$

- A_L is a deterministic algorithm can be implemented with a boolean circuit B_L with two sets of input gates $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_m$ such that

$$A_L(x,y) = 1 \text{ if and only if } B_L(x,y) = 1 \tag{2}$$



B_L



C_L^x

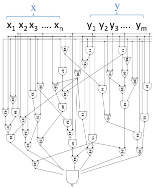
Chapter 34. NP-Completeness

Again we examine

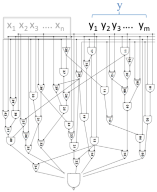
$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x, y) = 1 \tag{1}$$

- A_L is a deterministic algorithm can be implemented with a boolean circuit B_L with two sets of input gates $x = x_1 x_2 \dots x_n$ and $y = y_1 y_2 \dots y_m$ such that

$$A_L(x, y) = 1 \text{ if and only if } B_L(x, y) = 1 \tag{2}$$



B_L



C_L^x

- Because x is given, circuit B_L can be made into circuit C_L^x such that

$$B_L(x, y) = 1 \text{ if and only if } C_L^x(y) = 1 \tag{3}$$

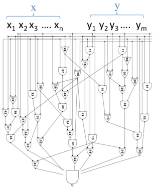
Chapter 34. NP-Completeness

Again we examine

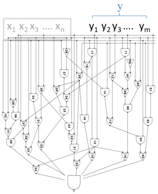
$$x \in L \iff \exists y, |y| \le |x|^c, A_L(x,y) = 1 \qquad (1)$$

- A_L is a deterministic algorithm can be implemented with a boolean circuit B_L with two sets of input gates $x = x_1 x_2 \dots x_n$ and $y = y_1 y_2 \dots y_m$ such that

$$A_L(x,y) = 1 \text{ if and only if } B_L(x,y) = 1 \qquad (2)$$



B_L



C_L^x

- Because x is given, circuit B_L can be made into circuit C_L^x such that

$$B_L(x,y) = 1 \text{ if and only if } C_L^x(y) = 1 \qquad (3)$$

- From (1), (2), and (3), we have

$$x \in L \iff \exists y \, C_L^x(y) = 1 \qquad (4)$$

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

Now we have

$$x \in L \iff \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;

Now we have

$$x \in L \Longleftrightarrow \exists y \, C_L^x(y) = 1 \qquad (5)$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

- Define the following language:

$$CSAT = \{C : \text{ circuit } C \text{ is satisfiable } \}$$

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

- Define the following language:

$$CSAT = \{C : \text{ circuit } C \text{ is satisfiable } \}$$

- From (4), we have

$$x \in L \Longleftrightarrow C_L^x \in CSAT \tag{6}$$

Chapter 34. NP-Completeness

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

- Define the following language:

$$CSAT = \{C : \text{ circuit } C \text{ is satisfiable } \}$$

- From (4), we have

$$x \in L \Longleftrightarrow C_L^x \in CSAT \tag{6}$$

It remains to be shown

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

- Define the following language:

$$CSAT = \{C : \text{ circuit } C \text{ is satisfiable } \}$$

- From (4), we have

$$x \in L \Longleftrightarrow C_L^x \in CSAT \tag{6}$$

It remains to be shown

- that reducing **algorithm** A_L to **circuit** B_L is valid;

Now we have

$$x \in L \Longleftrightarrow \exists y\, C_L^x(y) = 1 \tag{5}$$

- Define: a boolean circuit C is **satisfiable** if there exists at least one set of values y to its input gates such that $C(y) = 1$.

  e.g., $C(x_1, x_2) = (\neg x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ is satisfiable;
  but $D(x_1, x_2) = (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$ is not!

- Define the following language:

$$CSAT = \{C : \text{ circuit } C \text{ is satisfiable } \}$$

- From (4), we have

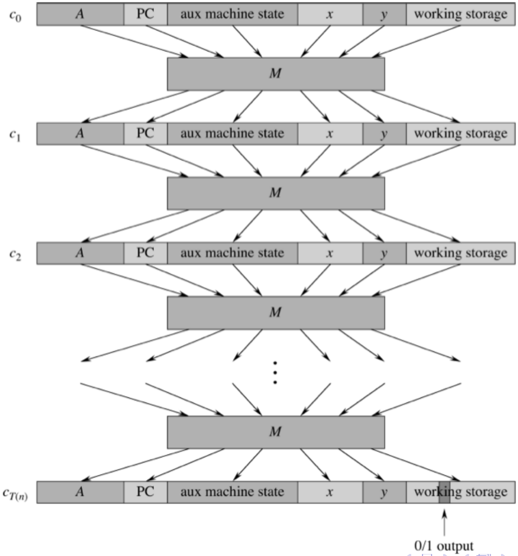
$$x \in L \Longleftrightarrow C_L^x \in CSAT \tag{6}$$

It remains to be shown

- that reducing **algorithm** A_L to **circuit** B_L is valid; and
- that the reduction can be done in **polynomial time**.

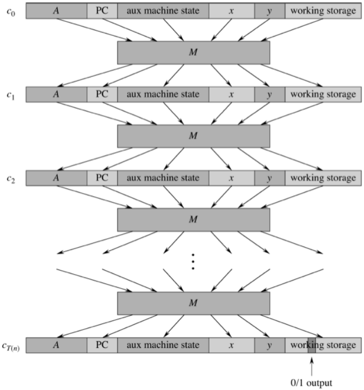
Unfold deterministic polynomial-time algorithm $A(x, y)$ with input $\langle x, y \rangle$

Unfold deterministic polynomial-time algorithm $A(x, y)$ with input $\langle x, y \rangle$



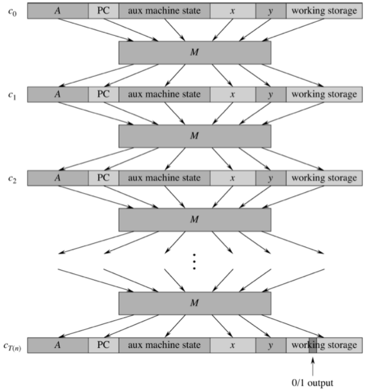
The algorithm is physically implemented with a boolean circuit

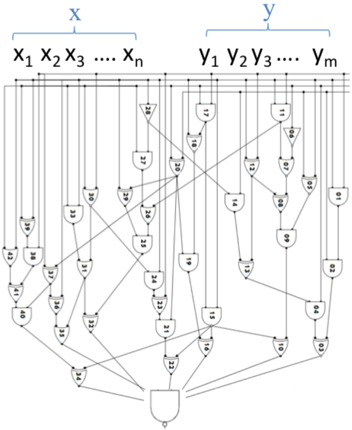
The algorithm is physically implemented with a boolean circuit



Chapter 34. NP-Completeness

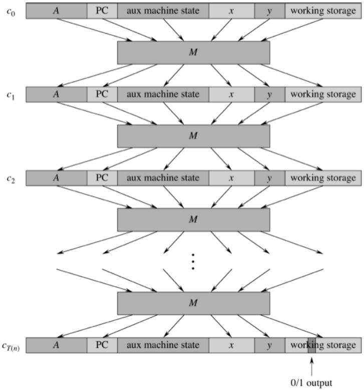
The algorithm is physically implemented with a boolean circuit

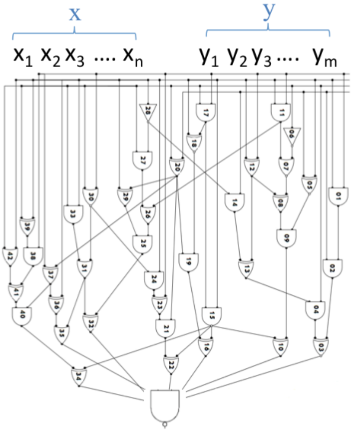




Chapter 34. NP-Completeness

The algorithm is physically implemented with a boolean circuit





And the circuit can be built from the algorithm in polynomial time.

The above discussion shows that L_{CSAT} is NP-hard.

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

Proof: It suffices to show the $CSAT$ is in NP. (Can you prove this?)

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

Proof: It suffices to show the $CSAT$ is in NP. (Can you prove this?)

Actually, the following language SAT was first proved to be NP-complete [Cook'71]
`https://www.cs.toronto.edu/~sacook/homepage/1971.pdf`

$$SAT = \{\phi : \text{ CNF boolean formula } \phi \text{ is satisfiable}\}$$

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

Proof: It suffices to show the $CSAT$ is in NP. (Can you prove this?)

Actually, the following language SAT was first proved to be NP-complete [Cook'71]
`https://www.cs.toronto.edu/~sacook/homepage/1971.pdf`

$$SAT = \{\phi : \text{ CNF boolean formula } \phi \text{ is satisfiable}\}$$

Cook's Theorem: SAT is NP-complete.

Chapter 34. NP-Completeness

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

Proof: It suffices to show the $CSAT$ is in NP. (Can you prove this?)

Actually, the following language SAT was first proved to be NP-complete [Cook'71]
`https://www.cs.toronto.edu/~sacook/homepage/1971.pdf`

$$SAT = \{\phi : \text{ CNF boolean formula } \phi \text{ is satisfiable}\}$$

Cook's Theorem: SAT is NP-complete.

Cook's reduction: characterizing a polynomial-time computation on nondeterministic Turing machine with a boolean formula,

The above discussion shows that L_{CSAT} is NP-hard.

Theorem: Language $CSAT$ is NP-complete.

Proof: It suffices to show the $CSAT$ is in NP. (Can you prove this?)

Actually, the following language SAT was first proved to be NP-complete [Cook'71] `https://www.cs.toronto.edu/~sacook/homepage/1971.pdf`

$$SAT = \{\phi : \text{ CNF boolean formula } \phi \text{ is satisfiable}\}$$

Cook's Theorem: SAT is NP-complete.

Cook's reduction: characterizing a polynomial-time computation on nondeterministic Turing machine with a boolean formula, such that a nondeterministic path leading to the accept state **corresponds** to an assignment to the variables making the the formula TRUE.

It is very easy to convert a boolean formula to a boolean circuit. So

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

On the other hand,

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

On the other hand,

Theorem: $CSAT \leq_p SAT$.

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

On the other hand,

Theorem: $CSAT \leq_p SAT$.

how to convert a circuit to a boolean formula (from network to tree)?

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

On the other hand,

Theorem: $CSAT \leq_p SAT$.

how to convert a circuit to a boolean formula (from network to tree)?
simply replicating gates may blow-up the size of formula to exponential!

It is very easy to convert a boolean formula to a boolean circuit. So

Theorem: $SAT \leq_p CSAT$.

On the other hand,

Theorem: $CSAT \leq_p SAT$.

how to convert a circuit to a boolean formula (from network to tree)?
simply replicating gates may blow-up the size of formula to exponential!

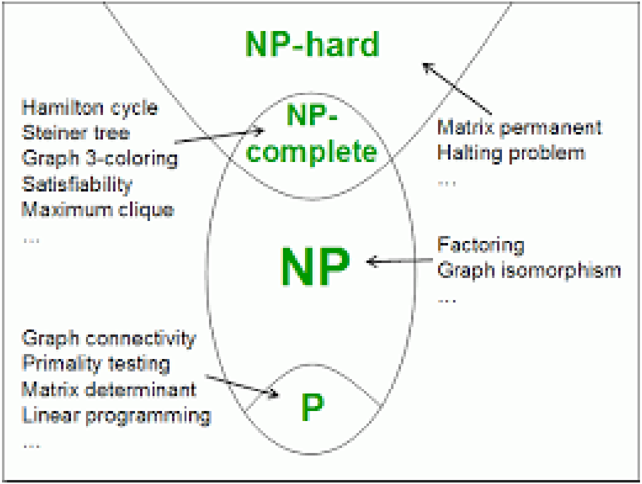
Will be discussed in CSCI 8470 Advanced Algorithms.

Chapter 34. NP-Completeness

Landscape of NP problems and beyond

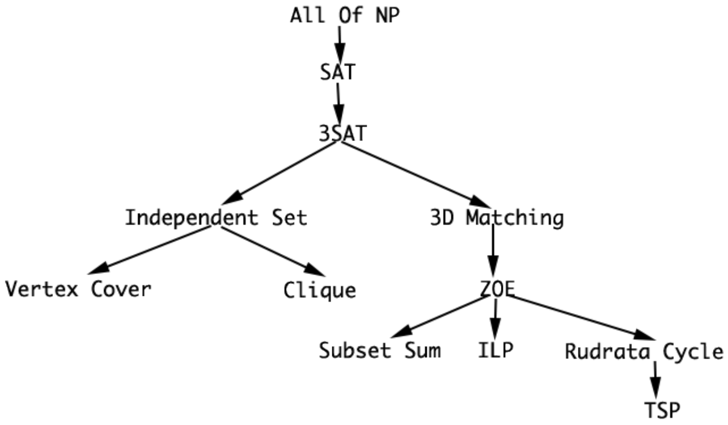
Chapter 34. NP-Completeness

Landscape of NP problems and beyond



Chapter 34. NP-Completeness

Many problems/languages have been proved NP-complete (Karp70s)



Examples of reduction techniques

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p 3\mathrm{SAT}$

(z)

Examples of reduction techniques

Example 1: $\mathrm{SAT} \le_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p 3\mathrm{SAT}$

$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$
(y, z)

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$
$$(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$
$$(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)$$
$$(x, y, z)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$
$$(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)$$
$$(x, y, z) \Longrightarrow (x, y, z)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$
$$(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)$$
$$(x, y, z) \Longrightarrow (x, y, z)$$
$$(y, z, u, v)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$
\begin{aligned}
&(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)\\
&(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)\\
&(x, y, z) \Longrightarrow (x, y, z)\\
&(y, z, u, v) \Longrightarrow (y, z, x_1) \wedge (\neg x_1, u, v)
\end{aligned}
$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

$$(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)$$
$$(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)$$
$$(x, y, z) \Longrightarrow (x, y, z)$$
$$(y, z, u, v) \Longrightarrow (y, z, x_1) \wedge (\neg x_1, u, v)$$
$$(y, z, u, v, w)$$

Examples of reduction techniques

Example 1: $\mathrm{SAT} \leq_p \mathrm{3SAT}$

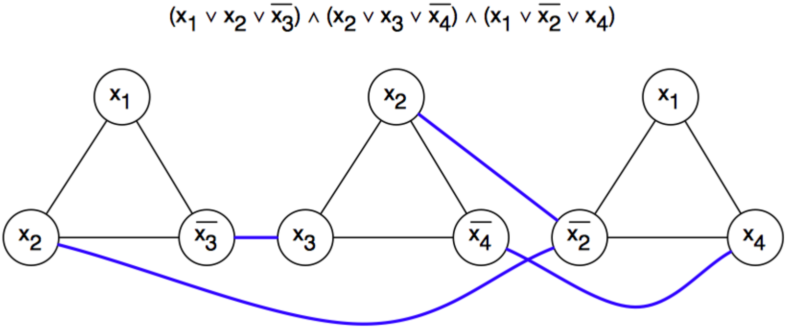
$$
\begin{aligned}
&(z) \Longrightarrow (z, x_1, x_2) \wedge (z, x_1, \neg x_2) \wedge (z, \neg x_1, x_2) \wedge (z, \neg x_1, \neg x_2)\\
&(y, z) \Longrightarrow (y, z, x_1) \wedge (y, z, \neg x_1)\\
&(x, y, z) \Longrightarrow (x, y, z)\\
&(y, z, u, v) \Longrightarrow (y, z, x_1) \wedge (\neg x_1, u, v)\\
&(y, z, u, v, w) \Longrightarrow (y, z, x_1) \wedge (\neg x_1, u, x_2) \wedge (\neg x_2, v, w)
\end{aligned}
$$

Chapter 34. NP-Completeness

Chapter 34. NP-Completeness

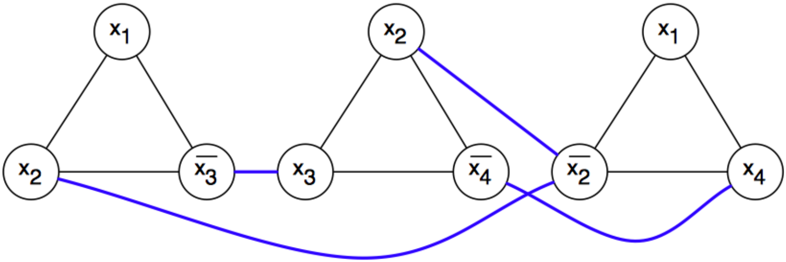
Example 2: $3\mathrm{SAT} \leq_p \mathrm{IS}$

Example 2: $3\mathrm{SAT} \leq_p \mathrm{IS}$



Example 2: $3\mathrm{SAT} \leq_p \mathrm{IS}$

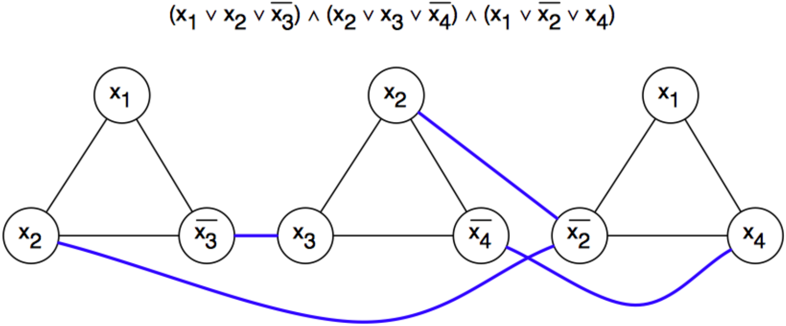
$$(x_1 \vee x_2 \vee \overline{x_3}) \wedge (x_2 \vee x_3 \vee \overline{x_4}) \wedge (x_1 \vee \overline{x_2} \vee x_4)$$



An assignment TRUE to one literal in each clause

Chapter 34. NP-Completeness

Example 2: $3\mathrm{SAT} \leq_p \mathrm{IS}$



An assignment TRUE to one literal in each clause corresponds to an independent set in the transformed graph.

Summary

Summary

Scope of the Final Exam

Summary

Scope of the Final Exam

- Minimum spanning tree

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

  generic, Kruskal's and Prim's

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

  generic, Kruskal's and Prim's

- Shortest path (single source and all pairs)

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

  generic, Kruskal's and Prim's

- Shortest path (single source and all pairs)

  concept/properties of shortest path, greedy algorithms, relaxation technique

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

  generic, Kruskal's and Prim's

- Shortest path (single source and all pairs)

  concept/properties of shortest path, greedy algorithms, relaxation technique

  single source: Bellman-Ford, Shortest-path-DAG, Dijkstra's

Summary

Scope of the Final Exam

- Minimum spanning tree

  concept/properties of MST, greedy algorithms,

  generic, Kruskal's and Prim's

- Shortest path (single source and all pairs)

  concept/properties of shortest path, greedy algorithms, relaxation technique

  single source: Bellman-Ford, Shortest-path-DAG, Dijkstra's

  all pairs: DP, Floyd-Warshall

Summary

Summary

Scope of the Final Exam (cont')

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

  non-deterministic computation, certificate, checker,

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

  non-deterministic computation, certificate, checker,

  definitions of NP class, proof that a language is in NP

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

  non-deterministic computation, certificate, checker,

  definitions of NP class, proof that a language is in NP

  reduction, polynomial-time reduction, properties

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

  non-deterministic computation, certificate, checker,

  definitions of NP class, proof that a language is in NP

  reduction, polynomial-time reduction, properties

  definition of NP-hard, NP-complete languages, properties

Summary

Scope of the Final Exam (cont')

- NP-completeness theory

  non-deterministic computation, certificate, checker,

  definitions of NP class, proof that a language is in NP

  reduction, polynomial-time reduction, properties

  definition of NP-hard, NP-complete languages, properties

  NP-completeness proofs (simple, assembly of previous known reductions)

More abou algorithms and complexity theory

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions
   - complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$.

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions
   - complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$.
2. Coping with the intractability

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions
   - complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$.
2. Coping with the intractability
   - Exact algorithms via parameterization (parameterized computation)

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions
   - complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$.
2. Coping with the intractability
   - Exact algorithms via parameterization (parameterized computation)
   - Randomized algorithms (Monte Carlo algorithms)

More abou algorithms and complexity theory

What will be in CSCI 8470 Advanced Algorithms

1. More landscapes of intractability problems
   - polynomial-time reductions
   - complexity classes beyond $\mathcal{P}$ and $\mathcal{NP}$.
2. Coping with the intractability
   - Exact algorithms via parameterization (parameterized computation)
   - Randomized algorithms (Monte Carlo algorithms)
   - Approximation algorithms (an introduction)