COEN279 - Design and Analysis of Algorithms

NP-Complete

Dr. Tunghwa Wang

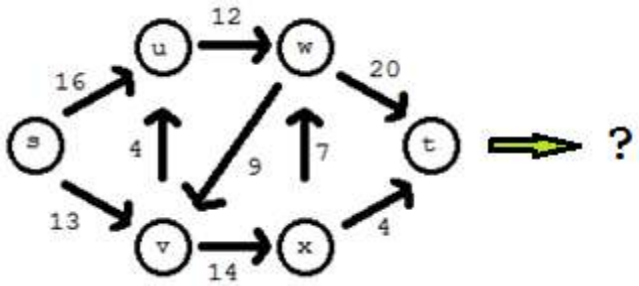
Fall 2016

Announcement

- Grader: Xiaoyu Li
  - Email: natalienew2016@gmail.com
  - Future submissions to her: CC to me
    - Filename should contain your name and type of homework: for example, TunghwaWang-Bonus2
    - You may resubmit, same file names: new file replaces the old one.
- Programming assignment #3:
  - Due 11/17 11:59PM:
    - When you run “perl Submit P3”, you should see the message of successful submission only to twang1@cse.scu.edu.
- Bonus Assignment #11:
  - It is online now.
  - Due 11/19 09:00AM

Ford-Fulkerson Algorithm

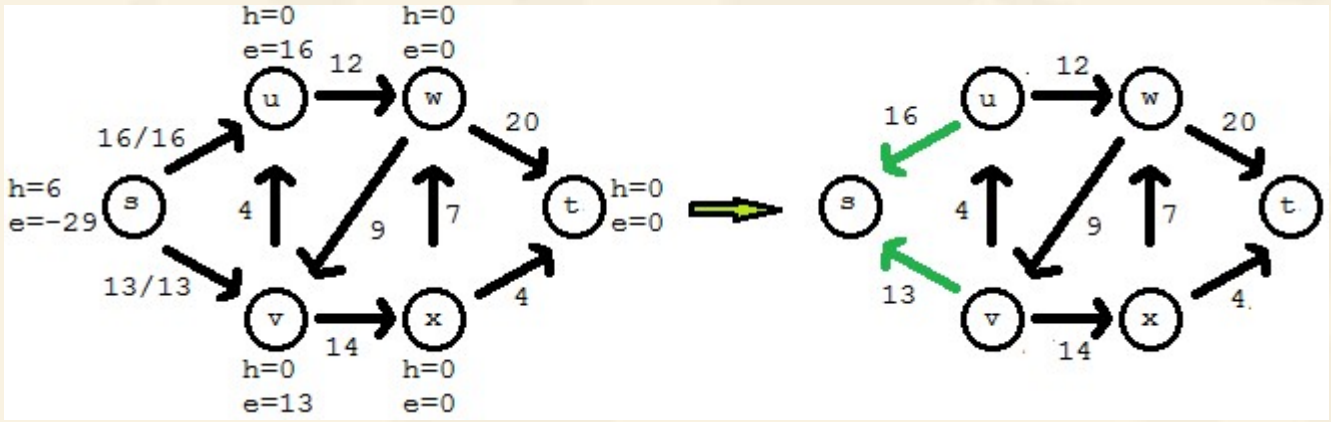
- Example:
  - After initialization:



  - For any given flow network G, what is $G_{f=0}$?
    - Since nothing needs to be done when f=0, it si G.

Push-Relabel Algorithm

- Example:
  - After initialization:



  - LHS is the given G.
  - RHS is the corresponding $G_{f=0}$.
  - Initial available operations:
    - Relabel u: its height becomes 1.
    - Relabel v: its height becomes 1.
  - Afterwards:
    - Push u and push v are possible.
    - More other push and relabel operations will be made possible.

$O(2^n)$ Complexity

- Exponential complexity:
  - Super-polynomial complexity.
    - $O(n^c)$: no matter how big is the constant c, it is still polynomial.
      - Computation complexity class P: This class of problems are solvable in polynomial time complexity.
    - Computation complexity class NP:
      - Formal definition is Non-deterministic Polynomial time complexity.
      - Equivalently, this class of problems have to be verifiable (with answer 'yes') by deterministic polynomial time complexity.
  - Obviously, $P \subseteq NP$:
    - However, we do not know whether P = NP.
    - Computing Fibonacci numbers: It is not an NP problem.
      - Recursion: $O(2^n)$ time complexity
      - With O(n) space complexity trade-off: O(n) time complexity.

$O(2^n)$ Complexity

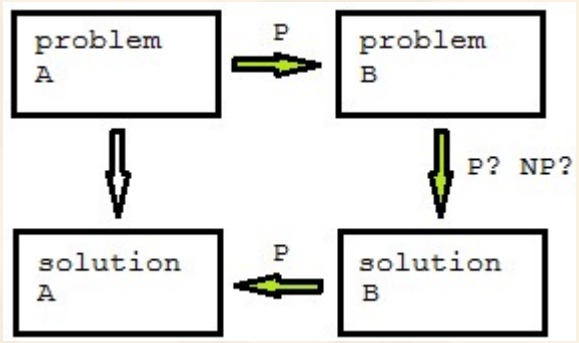
- Exponential complexity:
  - Computation complexity class co-NP:
    - Equivalently, this class of problems have to be verifiable with answer 'no' by deterministic polynomial time complexity.
    - We do not know whether NP = co-NP.
- NP is hard:
  - Computation complexity is not polynomial or better.
    - We focus on hardness.
  - It is very close from the problem we are dealing with:
    - Shortest vs. longest paths of a graph:
      - Shortest: O(|V||E|)
      - Longest: NP.
    - Euler tour vs. Hamiltonian cycle of a graph:
      - Euler tour: O(|E|)
      - Hamiltonian cycle: NP.

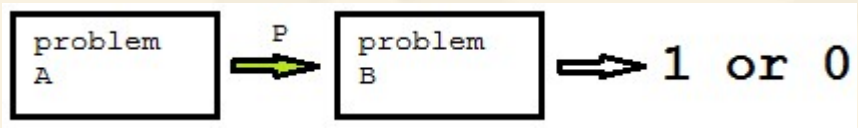
NP-Complete

- It is NP hard:
  - However, it is as hard as all other problems in this class.
    - Any problem is as hard as others.
    - If any one can be solved in polynomial time, then all problems in this class can be solved in polynomial time. However,
      - Up to date, we still do not have such solution.
      - We cannot prove that it cannot be solved in polynomial time either.
- To prove that a problem is NP-complete:
  - We focus on how hard it is.
  - Key points:
    - Decision problems vs. optimization problems:
      - Usually an optimization problem can be abstracted into a decision problem that is no harder than original optimization problem.
    - Reductions:
      - Polynomial time to transform on problem into another and vice versa.
    - First NP-complete problem?

NP-Complete

- Polynomial-time transformations:





  - We only need to prove that there is polynomial transformation time and leave everything else to computation theorist.
    - If problem B has polynomial-time solution, then problem A is solved in polynomial time by performing transforming problem A to problem B; solve problem B; transforming from solution B to solution A.
    - If solution B is simple answer such as ‘yes’ or ‘no’, then here is even no need to transform solution B back to solution A.

Encoding

- Encoding of set S is a function S→N:
  - Binary encoding:
    - We call a problem whose instance set is the set of binary strings a concrete problem.
    - For an abstract decision problem Q, mapping an instance set I to {0, 1}, an encoding e: I→{0, 1}* can induce a related concrete decision problem, e(Q).
  - A Function f: {0, 1}* → {0, 1}* is polynomial-time computable if there exists a polynomial-time algorithm A such that, given any input $x \in \{0, 1\}^*$, produces as output f(x).
    - For some set I of problem instances, two encodings e_1 and e_2 are polynomially related if there exist two polynomial computable functions f_{12} and f_{21} such that for any instance $i \in I$, $f_{12}(e_1(i)) = e_2(i)$ and $f_{21}(e_2(i)) = e_1(i)$.
    - Let Q be an abstract problem on an instance set I, and e_1 and e_2 be polynomially related encodings on I. Then $e_1(Q) \in P$ if and only if $e_2(Q) \in P$.

Formal Language Framework

- Based on formal-language theory:
  - A language L over alphabet Σ:
    - Σ is a finite set of symbols.
    - Σ^*, including empty string ε, is the set of all stings over Σ.
    - A language L over alphabet Σ is a subset of Σ^*.
      - Empty language $\varnothing$
  - Operations:
    - Set operations can be applied to a language.
    - String operations can be applied to a language.
    - Closure of a language L, denoted as $L^* = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots$
      - $L^0 = \{\varepsilon\}$
      - $L^k = L \mid L \mid L \mid \ldots \mid L$ (k times)

Formal Language Framework

- Based on formal-language theory:
  - The set of instances for any decision problem Q is simply the set Σ^* where $\Sigma = \{ 0, 1 \}$:
    - A language L over $\Sigma = \{ 0, 1 \}$ where $L = \{ x \in \Sigma^* : Q(x) = 1 \}$.
      - Q accepts a string $x \in \Sigma^*$ if $Q(x) = 1$.
      - Q rejects a string $x \in \Sigma^*$ if $Q(x) = 0$.
      - We say that language L is accepted by algorithm Q.
    - However, even L is accepted by algorithm Q, an $x \notin L$ as an input may not be rejected.
      - Q(x) may never stop.
  - A language L is accepted in polynomial time by an algorithm Q:
    - There exists a constant k such that for any length-n string $x \in L$, algorithm Q accepts x in time $O(n^k)$.

Formal Language Framework

- Based on formal-language theory:
  - A language L is decided by an algorithm Q:
    - Q(x) = 1 for all $x \in L$.
    - Q(x) = 0 for all $x \notin L$.
  - A language L is decided in polynomial time by an algorithm Q:
    - There exists a constant k such that for any length-n string $x \in \Sigma^*$, algorithm Q correctly decides whether $x \in L$ in time $O(n^k)$.
  - Thus for algorithm Q:
    - To accept language L:
      - Produce answers for all $x \in L$.
    - To decide language L:
      - Produce answers for all $x \in \Sigma^*$.

Formal Language Framework

- Based on formal-language theory:
  - A complexity class is a set of languages satisfying a membership measure:
    - Running time
    - Memory utilized
    - Else
  - Complexity class P:
    - P = { L $\subseteq$ { 0, 1 }*: there exists an algorithm Q that decides L in polynomial time }
    - P = { L $\subseteq$ { 0, 1 }*: there exists an algorithm Q that accepts L in polynomial time }
      - →: It is trivial by definition.
      - ←: We can say a new Q' exists, though we have no idea how to construct such algorithm, based on Q

Formal Language Framework

- Based on formal-language theory:
  - A verification algorithm:
    - x: an input string
    - y: a binary string (certificate)
    - A verification algorithm Q(x, y) works on the given input string x and certificate string y.
  - Algorithm Q(x, y) verifies the input string x if there exists a certificate y such that Q(x, y) = 1.
    - A language L verified by verification algorithm Q is L = { x $\in$ { 0, 1 }*: there exists y $\in$ { 0, 1 }* such that Q(x, y) = 1 }
  - Complexity class NP:
    - NP = { L $\subseteq$ { 0, 1 }*: there exists a verification algorithm Q and constant c such that Q verifies L in polynomial time }
      - L = { x $\in$ { 0, 1 }*: there exists y $\in$ { 0, 1 }* with $|y| = O(|x|^c)$ such that Q(x, y) = 1 }

Formal Language Framework

- Based on formal-language theory:
  - Complexity class co-NP:
    - Co-NP = { $L \subseteq \{ 0, 1 \}^*$: there exists a verification algorithm Q and constant c such that Q verifies $\{ 0, 1 \}^* - L$ in polynomial time }
  - Possible relations of P, NP, co-NP:
    - P = NP = co-NP: however we can neither prove nor disprove.
    - $P \subset NP$ = co-NP: however we can neither prove nor disprove.
    - NP ≠ co-NP and $P = NP \cap$ co-NP: however we can neither prove nor disprove.
    - NP ≠ co-NP and $P \subset NP \cap$ co-NP: however we can neither prove nor disprove.
  - Suspected relations of P, NP, NPC:
    - $P \subset NP$
    - $NPC \subset NP$
    - $P \cap NPC = \varnothing$
    - Thus most likely NP ≠ co-NP and $P \subset NP \cap$ co-NP

NP-Complete

- We'll only focusing on how to prove a problem is NP-complete problem.
  - $L_1 \leq_p L_2$: A language L_1 is polynomial time reducible to another language L_2 if and only if
    - There exists a polynomial-time computable function f: { 0, 1 }* → { 0, 1 }* such that for all $x \in \{ 0, 1 \}^*$, $x \in L_1$ if and only if $f(x) \in L_2$.
    - f is the reduction function
    - Polynomial-time algorithm Q that computers f is the reduction algorithm.
  - If $L_1 \subseteq \{ 0, 1 \}^*$ and $L_2 \subseteq \{ 0, 1 \}^*$ are languages such that $L_1 \leq_p L_2$, then $L_2 \in P$ implies $L_1 \in P$.
    - A language L is NP-complete if
      - $L \in NP$
      - $L' \leq_p L$ for every $L' \in NP$.
    - A language L is NP-hard if second condition is met but first condition not met.

NP-Complete

- We'll only focusing on how to prove a problem is NP-complete problem.
  - NPC problems are hardest NP problems.
  - NP-hard problems are harder than NP, thus NPC.
- We only know this:
  - If any NP-complete problem is polynomial-time solvable, then P = NP. Equivalently, if any problem in NP is not polynomial-time solvable, then no NP-complete problem is polynomial-time solvable.

Backup Slides