CSE 6140/ CX 4140
Computational Science and Engineering
ALGORITHMS

Project

Instructor: Xiuwei Zhang
Assistant Professor
School of Computational Science and Engineering

CSE 6140

PROJECT

Minimum Vertex Cover Problem

- MINIMUM VERTEX COVER: Given a graph G = (V, E) , **find the smallest** subset of vertices S ⊆ V such that for each edge at least one of its endpoints is in S?
- VERTEX COVER: Given a graph G = (V, E) and an integer k, **is there a subset** of vertices S ⊆ V such that **|S| ≤ k**, and for each edge, at least one of its endpoints is in S?

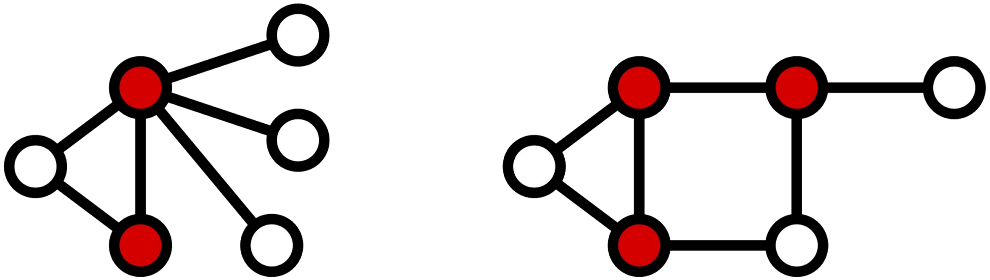


Image from https://en.wikipedia.org/wiki/Vertex_cover

Timeline

- Oct. 22: release of project description and data
- Oct. 30: groups should be finalized
- Nov. 20: partial report due
- Dec. 4: final deliverables due

These are hard deadlines with no grace period.

Teams

- Group of up to 4
- If have not formed your groups, consider using Piazza to finalize.
- You have until Oct. 30 11:59pm to submit your group members and implementation language, via Canvas (0 pts, but required)

  LastName1, First Name1

  LastName2, First Name2

  …

  Python
- If you are really having problem finding groups, indicate that in your submission, we could randomly assign you to a group.
- We may restructure/merge some groups, if needed.
- Groups of different sizes will be evaluated in the same way.

4 Algorithms

- Branch-and-Bound (**1** alg)
  - How do you branch/expand on each node?
  - How do you bound on each node?
- Approximation Algorithm (**1** alg)
- Local Search (**2** different algs)
  - Choose a method to select a starting solution
  - Choose a neighborhood
  - Choose a method to explore search space
    - Hill Climbing, Simulated Annealing, Iterated Local Search, Tabu Search, Genetic Alg

Executable

- Format of executable should follow what is specified in Project Description
- Output:
  - Solution files
  - Solution trace files for BnB and LS
  - Format should conform with Project description

Deliverables

- Partial report/check in on Nov 20 (5/50 points)
  - You should have be able to parse input and produce output
  - You should have at least 2 approaches working by then
  - Report best quality within x (eg. x=10) mins for each benchmark instance
- Report (the bulk of your grade)
  - Written like a proper academic paper
  - Describes your approaches and choices, Data structures, Worst-case running time
  - For each instance, report results on all ALGs in terms of relative optimality gap with max *x (*around *10)* mins time cutoffs
  - Does the empirical running time match the worst-case complexity? How does the empirical relative error compare to the approximation guarantee? How do methods compare?
  - For more details check "project description"

Competition

Optional competition for a small number of bonus points

Two independent algorithm tracks:
(1) branch-and-bound
(2) local search

For each track, bonus points will be assigned to top performing submissions as follows:
1st place - 3 pts
2nd place - 2 pts
3rd place - 1 pt

A team can earn up to a maximum of 6 pts in the competition.

Submissions that fail to follow the instructions will not be considered.

Team Participation

- Each team will submit one submission
- Each individual will submit (through Canvas quiz) a thoughtful and honest evaluation of the contributions of your group members, including yourself.
- For each individual, include a score from 1 to 9 indicating your evaluation of their work. All scores for one team should sum up to 10.
- You may also include any clarifying comments, etc. (especially for low scores).
- Scores for individuals will be adjusted by teammates' evaluation

CSE 6140/ CX 4140
Computational Science and Engineering
ALGORITHMS

Test 2 review

Instructor: Xiuwei Zhang
Assistant Professor
School of Computational Science and Engineering

Test 2 scope

- The dynamic programming chapter
- The NP-completeness chapter
- Some basic concepts introduced in the lecture on Monday in "coping with NPC" chapter

Dynamic Programming [KT 6]

- Show problem has optimal substructure: the optimal solution can be constructed from optimal solutions to subproblems.
- Define the recurrence (make sure to include base cases, and show where is the score being optimized)
- Show subproblems are overlapping, i.e., subproblems may be encountered many times, but the total number of distinct subproblems is polynomial
- Construct an algorithm that computes the optimal solution to each subproblem only once, and reuses the stored result all other times
- Show that time and space complexity is polynomial

Algorithmic Paradigms

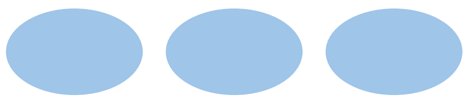
- Greedy. Build up a solution incrementally, myopically optimizing some local criterion. (not trying all options but can prove that greedy choice results optimal solution at the end)

- Divide-and-conquer. Break up a problem into *non-overlapping* sub-problems, solve each sub-problem independently, and combine solution to sub-problems to form solution to original problem.

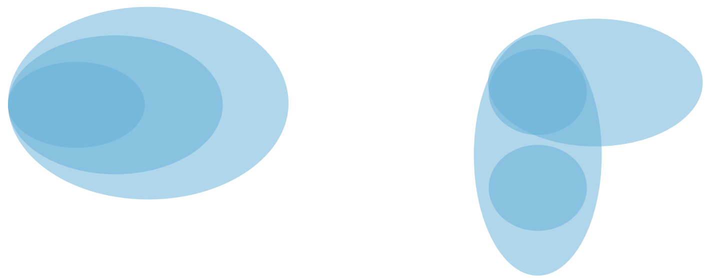
- **Dynamic programming.** Break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger sub-problems from smaller subproblems, (*reusing* solutions of encountered subproblems as much as possible).

Subproblems

Divide and conquer



Greedy or Dynamic Programming



Dynamic programming: algorithms which systematically search all possibilities (thus guaranteeing correctness) while storing results to avoid recomputing (thus providing efficiency).

Greedy vs Dynamic Programming

	Greedy	Dynamic programming
Optimal substructure	the optimal solution can be constructed from optimal solutions to subproblems	
Optimality	Does not guarantee optimality	Guarantees optimality; equivalent to exhaustive search; efficient because of the reuse of subproblems
	Makes decisions based on local subproblem; once a choice is made, it is not changed	Makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution

Dynamic Programming

- Top-down DP = Memoization
  - Design a recursive algorithm
  - Store result for each subproblem when you first compute it
  - Check for existing result for a subproblem, before doing any extra work

- Bottom-up DP = Iterative DP
  - Determine dependency between a problem and its subproblems
  - Determine an order in which to compute subproblems so that you always have what you need already available
  - Fill in the table of results in the determined order (using FOR loops)

Steps in DP

1. Prove optimal substructure
2. Formulate the answer as a recurrence relation or recursive algorithm (base cases, top case to solve problem)
3. Show that the number of different instances/subproblems of your recurrence is bounded by a polynomial (overlapping subproblems)
4. (Top-down) Use recursion, save time by saving results for subproblems in a cache the first time you encounter them
5. (Bottom-up) Specify an order of evaluation for the recurrence so you always have what you need (you have to be careful about this)
6. Running time – how many subproblems and how much time to spent on each subproblems
7. Space – how many subproblems, how much space per each
8. Retrieve the solution - back-tracing

Problems

- Fixed number of choices
  - Weighted Interval Scheduling O(n log n) [KT 6.1]
  - Longest Common Subsequ. [CLRS 15.4] + Seq. Alignment [KT 6.6]
  - Coin changing pb O(nS) [BRV 4.1] and Knapsack O(nW) [KT 6.4] – pseudo-polynomial
  - All-pairs shortest paths – Floyd-Warshall $O(V^3)$ [CLRS 25.2]
- Multiway choice
  - RNA secondary structure [KT 6.5]
- Solved exercises [KT 6] ex1, [CLRS 15.2] Matrix Chain Product

Steps in DP

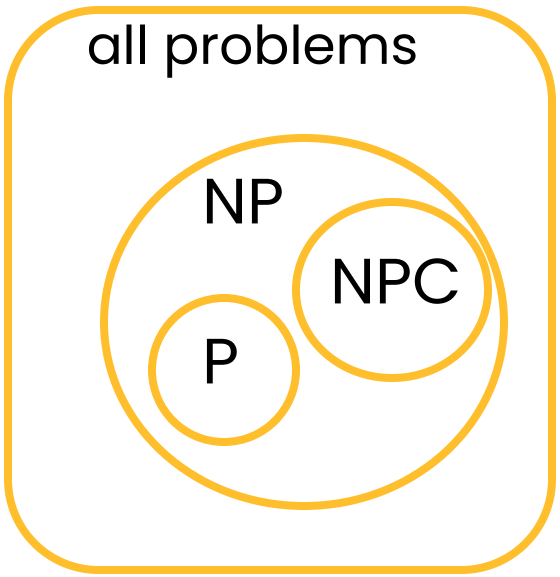
1. Prove optimal substructure Week6_Lecture1_Part1of3
2. Formulate the answer as a recurrence relation or recursive algorithm (base cases, top case to solve problem)
3. Show that the number of different instances/subproblems of your recurrence is bounded by a polynomial (overlapping subproblems)
4. (Top-down) Use <u>recursion</u>, save time by saving results for subproblems in a cache the first time you encounter them
5. (Bottom-up) Specify an <u>order</u> of evaluation for the recurrence so you always have what you need (you have to be careful about this)
6. Running time – how many subproblems and how much time to spent on each subproblems
7. Space – how many subproblems, how much space per each
8. Retrieve the solution - back-tracing Week6_Lecture2_Part1of3

NP-completeness [KT 8]

- Problems
  - Decision problems (yes/no)
  - Optimization problems (solution with best score)
- P
  - Decision problems (decision problems) that can be solved in polynomial time
  - be solved “efficiently”
- NP
  - Decision problems whose “YES” answer can be verified in polynomial time, if we already have candidate solution

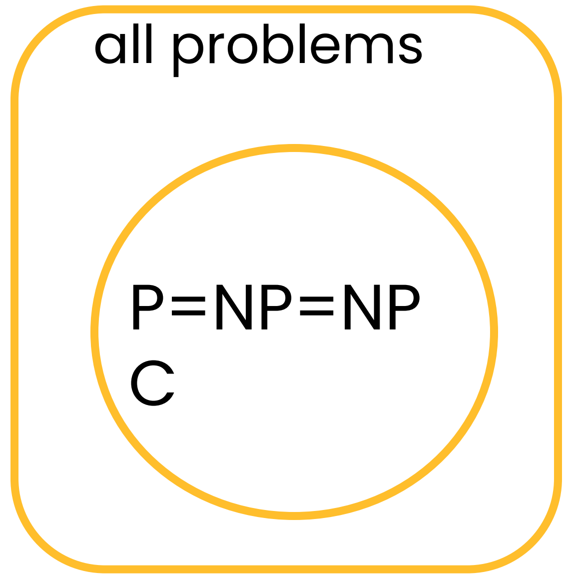
Possible Worlds

$P \neq NP$



or

$P = NP$

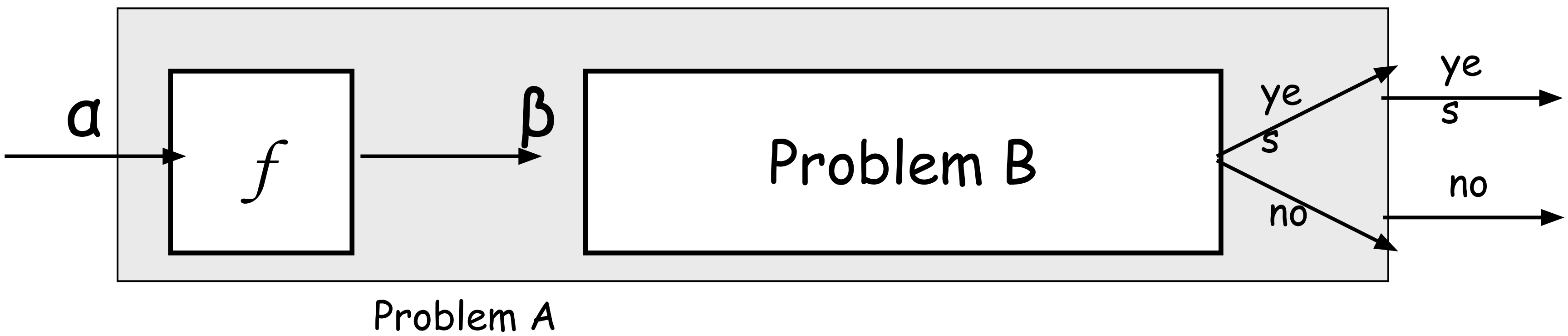


NPC: NP-complete

Reductions

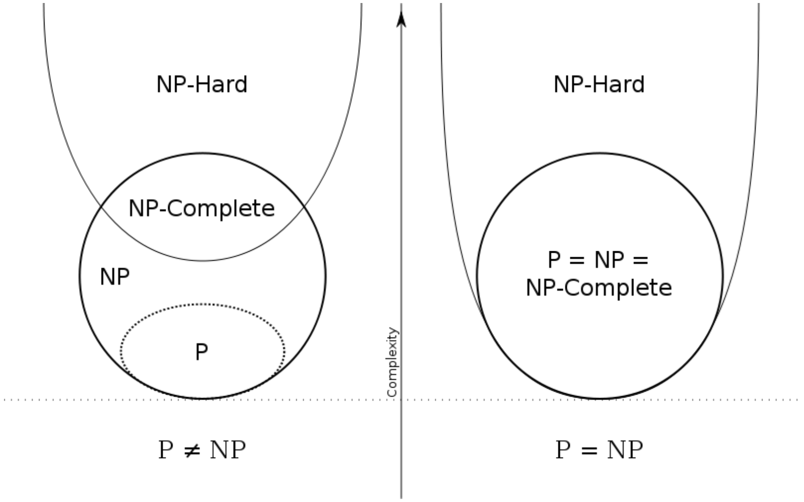
- **"A ≤ B":** Reduction from A to B is showing that we can solve A using the algorithm that solves B

- If we have an oracle for solving B, then we can solve A by making polynomial number of computations and <u>polynomial number of calls to the oracle</u> for B (Cook)
- **Idea:** transform the inputs of A to inputs of B (<u>single call to oracle</u>) (Karp)



NP-completeness

- NP-hard
  - NP-hard problems are at least as hard as NP problems
  - A problem is NP-hard iff a polynomial-time algorithm for it implies a polynomial-time algorithm for every problem in NP
- NP-complete
  - A problem is NP-complete if it is NP-hard, and it is in NP



Establishing NP-Completeness

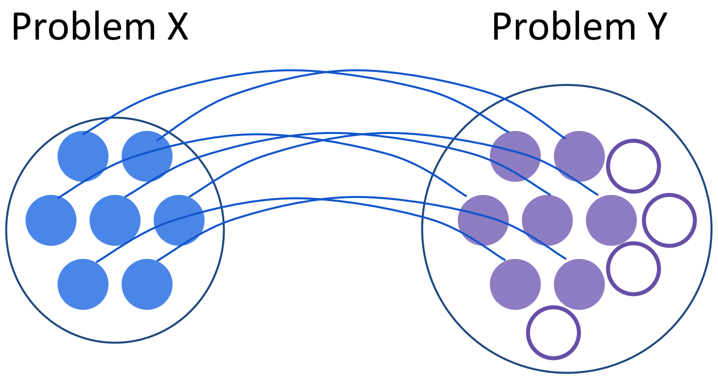
- Recipe to establish NP-completeness of problem Y.

  - Step 1. Show that Y is in NP.
    - Describe how a potential **solution**/witness will be represented
    - Describe a **procedure to check** whether the potential witness is a correct solution to the problem instance, and argue that this procedure takes **polynomial time**

  - Step 2. Choose an NP-complete problem X.

  - Step 3. Prove that $X \leq_P Y$ (X is **poly-time reducible** to Y).
    - (3a) Describe a **procedure f that converts** the inputs i of X to inputs of Y in **polynomial time**
    - (3bc) Show that the reduction is correct by showing that $X(i) = YES \Leftrightarrow Y(f(i)) = YES$ (**if and only if**, proof in both directions)

Prove that $X \leq_P Y$



X is NP-Complete. Y is at least as difficult as X. Y is NP-complete.

NP-completeness proofs: Step 3

- (Step 1 usually straightforward, and for Step 2 we may give you suggestions of problems to pick.)
- Step 3 of the proof: $X \leq_P Y$
  - Let I_1 be **any** instance of X (i.e., a graph, a set of numbers…)
  - Transform I_1 into an instance I_2 of problem Y
  - Check whether this transformation takes a polynomial time
  - Suppose I_1 has a solution (i.e., 3SAT is satisfiable). Then prove that I_2 also has a solution (this is why you constructed I_2 this way, so this is usually the "easy" way)
  - Suppose I_2 has a solution (the **particular** instance of problem Y that you created from I_1). Then, show that it implies that I_1 has a solution (i.e., exhibit the 3SAT instance I_1 is satisfiable)
- Hence, if you have an algorithm to solve pb Y, you can solve pb X using this algorithm on transformed instance I_2. X is easier than Y.

Summary of some NPc problems