CMSC 641: Algorithms

NP Completeness

Review: Dynamic Programming

- When applicable:
  - Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - Overlapping subproblems: few subproblems in total, many recurring instances of each
  - Basic approach:
    - Build a table of solved subproblems that are used to solve larger ones
    - *What is the difference between memoization and dynamic programming?*
      - *Memoization employs a top-down strategy*
      - *Advantage:Selective subproblems*
      - *Disadvantage: Overhead of recursion*

Review: Greedy Algorithms

- A *greedy algorithm* always makes the choice that looks best at the moment
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
    - Yes: fractional knapsack problem
    - No: playing a bridge hand
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Review: Activity-Selection Problem

- The *activity selection problem*: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, starting and ending at different times
  - Your goal: get onto as many rides as possible
- Naïve first-year CS major strategy:
  - Ride the first ride, get off, get on the very next ride possible, repeat until carnival ends
- *What is the sophisticated third-year strategy?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - **s_i = start time of activity i**
    - **f_i = finish time of activity i**
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is the optimal substructure for this problem?*

Review: Activity-Selection

- Formally:
  - Given a set S of n activities
    - s_i = start time of activity i
    - f_i = finish time of activity i
  - Find max-size subset A of compatible activities
  - Assume activities sorted by finish time
- *What is an optimal substructure for this problem?*
  - If k is the activity in A with the earliest finish time, then $A - \{k\}$ is an optimal solution to $S^* = \{i \in S: s_i \geq f_k\}$

Review: Greedy Choice Property For Activity Selection

- Dynamic programming? Memoize? Yes, but…
- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice ⇒ globally optimal solution
  - Pick the activity a* with the earliest finish time
  - if S is an activity selection problem sorted by finish time, then ∃ optimal solution $A \subseteq S$ such that $\{a^*\} \in A$
    - Sketch of proof: if ∃ optimal solution B that does not contain {a*}, we can always replace first activity b* in B with {a*} (*Why?*). Same number of activities, thus optimal.

Koustuv Dasgupta

Review: The Knapsack Problem

- The *0-1 knapsack problem*:
  - A thief must choose among n items, where the ith item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust
- *What greedy choice algorithm works for the fractional problem but not the 0-1 problem?*
  - *Value per pound :* v_i / w_i

Koustuv Dasgupta

NP-Completeness

- Some problems are *intractable*: as they grow large, we are unable to solve them in reasonable time
- What constitutes reasonable time? Standard working definition: *polynomial time*
  - On an input of size n the worst-case running time is $O(n^k)$, for some constant k
  - Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \lg n)$
  - Not in polynomial time: $O(2^n)$, $O(n^n)$, $O(n!)$

Koustuv Dasgupta

Polynomial-Time Algorithms

- Are some problems solvable in polynomial time?
  - Of course: every algorithm you have studied provides polynomial-time solution to some problem
  - We define **P** to be the class of problems solvable in polynomial time
- Are all problems solvable in polynomial time?
  - No: Turing's "Halting Problem" is not solvable by any computer, no matter how much time is given
    - Undecidable problems
  - Such problems are clearly intractable, not in **P**

Koustuv Dasgupta

NP-Complete Problems

- The *NP-Complete* problems are an interesting class of problems whose status is unknown
  - No polynomial-time algorithm has been discovered for an NP-Complete problem
  - No super polynomial lower bound has been proved for any NP-Complete problem, either
- We call this the *P = NP question*
  - Biggest open problem in CS ?
  - Maybe … who am I to say

Koustuv Dasgupta

An NP-Complete Problem: Hamiltonian Cycles

- An example of an NP-Complete problem:
  - A *Hamiltonian cycle* of an undirected graph is a simple cycle that contains every vertex
  - The Hamiltonian-cycle problem: given a graph G, does it have a Hamiltonian cycle?
  - *Describe a naïve algorithm for solving the Hamiltonian-cycle problem. Running time?*

Koustuv Dasgupta

P and NP

- As mentioned, **P** is set of problems that can be solved in polynomial time
- **NP** (*nondeterministic polynomial time*) is the set of problems that can be solved in polynomial time by a *nondeterministic* computer
  - *What is that?*

Nondeterminism

- Think of a non-deterministic computer as a computer that magically "guesses" a solution, then has to verify that it is correct
  - If a solution exists, computer always guesses it
  - One way to imagine it: a parallel computer that can freely spawn an infinite number of processes
    - Have one processor work on each possible solution
    - All processors attempt to verify that their solution works
    - If a processor finds one, we have a working solution
  - So: **NP** = problems *verifiable* in polynomial time

P and NP

- Summary so far:
  - **P** = problems that can be solved in polynomial time
  - **NP** = problems for which a solution can be verified in polynomial time
  - Unknown whether **P = NP** (most suspect not)
- Hamiltonian-cycle problem is in **NP**:
  - Cannot solve in polynomial time
  - Easy to verify solution in polynomial time (*How?*)

NP-Complete Problems

- We will see that NP-Complete problems are the "hardest" problems in NP:
  - If any *one* NP-Complete problem can be solved in polynomial time…
  - …then *every* NP-Complete problem can be solved in polynomial time…
  - …and in fact *every* problem in **NP** can be solved in polynomial time (which would show **P = NP**)
  - Thus: solve Hamiltonian-cycle in $O(n^{100})$ time, you've proved that **P = NP**. Retire rich & famous.

Reduction

- The crux of NP-Completeness is *reducibility*
  - Informally, a problem P can be reduced to another problem Q if *any* instance of P can be "easily rephrased" as an instance of Q, the solution to which provides a solution to the instance of P
    - *What do you suppose "easily" means?*
    - This rephrasing is called *transformation*
  - Intuitively: If P reduces to Q, P is "no harder to solve" than Q

Reducibility

- An example:
  - P: Given a set of Booleans, is at least one TRUE?
  - Q: Given a set of integers, is their sum positive?
  - Transformation: $(x_1, x_2, \ldots, x_n) = (y_1, y_2, \ldots, y_n)$ where $y_i = 1$ if x_i = TRUE, $y_i = 0$ if x_i = FALSE
- Another example:
  - Solving linear equations is reducible to solving quadratic equations
    - *we can easily use a quadratic-equation solver to solve linear equations*

Using Reductions

- If P is *polynomial-time reducible* to Q, we denote this $P \leq_p Q$
- Definition of NP-Complete:
  - If P is NP-Complete,
    then $P \in$ **NP** and all problems $R \in$ **NP** are reducible to P
  - Formally: $R \leq_p P$, $\forall R \in$ **NP**
- If $P \leq_p Q$ and P is NP-Complete, Q is also NP-Complete
  - This is the ***key idea*** for proving NP-completeness

Proving NP-Completeness

- *What steps do we have to take to prove a problem Q is NP-Complete?*
  - Pick a known NP-Complete problem P
  - Reduce P to Q
    - Describe a transformation that maps instances of P to instances of Q, s.t. "yes" for Q = "yes" for P
    - Prove the transformation works
    - Prove it runs in polynomial time
  - prove $Q \in$ **NP** (verification step)
    - What if you can't ?

Directed Hamiltonian Cycle ⇒ Undirected Hamiltonian Cycle

- *What was the Hamiltonian cycle problem ?*
- For the next trick, I will reduce the *directed Hamiltonian cycle* problem to the *undirected Hamiltonian cycle* problem
  - *Which variant am I proving NP-Complete?*
  - *What transformation do I need to effect?*

Transformation: Directed ⇒ Undirected Ham. Cycle

- Transform graph G = (V, E) into G' = (V', E'):
  - Every vertex v in V transforms into 3 vertices v^1, v^2, v^3 in V' with edges (v^1,v^2) and (v^2,v^3) in E'
  - Every directed edge (v, w) in E transforms into the undirected edge (v^3, w^1) in E' (draw it)
  - *Can this be implemented in polynomial time?*
  - *Argue that a directed Hamiltonian cycle in G implies an undirected Hamiltonian cycle in G'*
  - *Argue that an undirected Hamiltonian cycle in G' implies a directed Hamiltonian cycle in G*

Undirected Hamiltonian Cycle

- Thus we can reduce the directed problem to the undirected problem
- *What's left to prove the undirected Hamiltonian cycle problem is NP-Complete?*
- *Argue that the problem is in* **NP**

Hamiltonian Cycle ⇒ TSP

- The well-known *traveling salesman problem*:
  - Optimization variant: a salesman must travel to n cities, visiting each city exactly once and finishing where he begins. How to minimize travel time?
  - Model as complete graph with cost $c(i,j)$ to go from city i to city j
- *How would we turn this into a decision problem?*
  - A: ask if $\exists$ a TSP with cost $< k$

Hamiltonian Cycle ⇒ TSP

- The steps to prove TSP is NP-Complete:
  - Prove that TSP ∈ **NP** (*Argue this*)
  - Reduce the undirected Hamiltonian cycle problem to the TSP
    - So if we had a TSP-solver, we could use it to solve the Hamiltonian cycle problem in polynomial time
    - *How can we transform an instance of the Hamiltonian cycle problem to an instance of the TSP?*
      - *Given an instance G= (V,E) of HAM-CYCLE, form the complete graph G' = (V', E') :*
        - *with edges (i,j)*
        - *c(i,j) = 0 if (i,j) ∈ E, 1 otherwise*
    - *Can we do this in polynomial time?*
  - *G has a HAM-CYCLE iff G' has a TSP tour of cost at most zero*

The TSP

- Random asides:
  - TSPs (and variants) have enormous practical importance
    - E.g., for shipping and freighting companies
    - Lots of research into good approximation algorithms
  - Recently made famous as a DNA computing problem

The SAT Problem

- One of the first problems to be proved NP-Complete was *satisfiability* (SAT):
  - Given a Boolean expression on *n* variables, can we assign values such that the expression is TRUE?
  - Ex: $((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$
  - **Cook's Theorem:** The satisfiability problem is NP-Complete
    - **Proof: not here**

Conjunctive Normal Form

- Even if the form of the Boolean expression is simplified, the problem may be NP-Complete
  - *Literal*: an occurrence of a Boolean or its negation
  - A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    - Ex: $(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_3)$
  - *3-CNF*: each clause has exactly 3 distinct literals
    - Ex: $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee x_4) \wedge (\neg x_5 \vee x_3 \vee x_4)$
    - Notice: true if at least one literal in each clause is true

The 3-CNF Problem

- Satisfiability of Boolean formulas in 3-CNF form (the *3-CNF Problem*) is NP-Complete
  - Proof: Nope
- The reason we care about the 3-CNF problem is that it is relatively easy to reduce to others
  - Thus by proving 3-CNF NP-Complete we can prove many seemingly unrelated problems NP-Complete

3-CNF → Clique

- *What is a clique of a graph G?*
- A subset of vertices fully connected to each other, i.e. a complete subgraph of G
- The *clique problem*: how large is the maximum-size clique in a graph?
- *Can we turn this into a decision problem?*
- Yes, we call this the *k-clique problem*
- *Is the k-clique problem within **NP**?*

3-CNF → Clique

- *What should the reduction do?*
- A: Transform a 3-CNF formula to a graph, for which a *k*-clique will exist (for some *k*) iff the 3-CNF formula is satisfiable

3-CNF → Clique

- The reduction:
  - Let $B = C_1 \wedge C_2 \wedge \ldots \wedge C_k$ be a 3-CNF formula with *k* clauses, each of which has 3 distinct literals
  - For each clause put a triple of vertices in the graph, one for each literal
  - Put an edge between two vertices if they are in different triples and their literals are *consistent*, meaning not each other's negation
  - Run an example:
    $B = (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee y \vee z)$

3-CNF → Clique

- Prove the reduction works:
  - If B has a satisfying assignment, then each clause has at least one literal (vertex) that evaluates to 1
  - Picking one such "true" literal from each clause gives a set V' of *k* vertices. V' is a clique (*Why?*)
  - If G has a clique V' of size k, it must contain one vertex in each triple (clause) (*Why?*)
  - We can assign 1 to each literal corresponding with a vertex in V', without fear of contradiction

Clique → Vertex Cover

- A *vertex cover* for a graph G is a set of vertices that are incident to every edge in G
- The *vertex cover problem*: what is the minimum size vertex cover in G?
- Restated as a decision problem: does a vertex cover of size *k* exist in G?
- vertex cover is NP-Complete

Clique → Vertex Cover

- First, show vertex cover in **NP** (*How?*)
- Next, reduce *k*-clique to vertex cover
  - The *complement* G_C of a graph G contains exactly those edges not in G
  - Compute G_C in polynomial time
  - G has a clique of size *k* iff G_C has a vertex cover of size $|V| - k$
- Example

Clique → Vertex Cover

- Claim: If G has a clique of size *k*, G_C has a vertex cover of size $|V| - k$
  - Let V' be the *k*-clique
  - Then V - V' is a vertex cover in G_C
    - Let (u,v) be any edge in G_C
    - Then *u* and *v* cannot both be in V' (*Why?*)
    - Thus at least one of *u* or *v* is in V-V' , so edge (u, v) is covered by V-V'
    - Since true for *any* edge in G_C, V-V' is a vertex cover

Vertex Cover → Clique

- Claim: If G_C has a vertex cover $V' \subseteq V$, with $|V'| = |V| - k$, then G has a clique of size k
  - For all $u,v \in V$, if $(u,v) \in G_C$ then $u \in V'$ or $v \in V'$ or both (*Why?*)
  - Contrapositive: if $u \notin V'$ and $v \notin V'$, then $(u,v) \in E$
  - In other words, all vertices in V-V' are connected by an edge, thus V-V' is a clique
  - Since $|V| - |V'| = k$, the size of the clique is k

Koustuv Dasgupta

General Comments

- Literally hundreds of problems have been shown to be NP-Complete
- Some reductions are profound, some are comparatively easy, many are easy once the key insight is given
- You can expect a simple NP-Completeness proof on the final

Koustuv Dasgupta

Other NP-Complete Problems

- *Subset-sum*: Given a set of integers, does there exist a subset that adds up to some target *T*?
- *0-1 knapsack*: when weights not just integers
- *Hamiltonian path*: Obvious
- *Graph coloring*: can a given graph be colored with *k* colors such that no adjacent vertices are the same color?
- Etc…
- *Don't forget to look up Garey & Johnson – Computers and Intractability : A guide to the Theory of NP-completeness*

Koustuv Dasgupta