Exact Algorithms for NP-Complete Problems

The Vertex Cover Problem

Algorithms: Design and Analysis, Part II

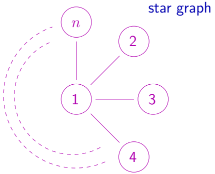
The Vertex Cover Problem

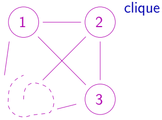
Input: An undirected graph $G = (V, E)$.

Goal: Compute a minimum-cardinality vertex cover – a subset $S \subseteq V$ that contains at least one endpoint of each edge of G.

Quiz

Question: What is the minimum size of a vertex cover of a star graph with n verities and a clique with n vertices respectively?





A) 1 and $n-1$

B) 1 and n

C) 2 and $n-1$

D) $n-1$ and n

Fact: In general, Vertex Cover is an NP-complete problem.

Strategies for NP-Complete Problems

(1) Identify computationally tractable special cases

- Trees [application of dynamic programming - try it!]
- Bipartite graphs [application of the maximum flow problem]
- When the optimal solution is "small" ($\approx \log n$ or less)

(2) Heuristics (e.g., via suitable greedy algorithms)

(3) Exponential time but better than brute-force search [coming up next]

Exact Algorithms for NP-Complete Problems

Smarter Search for Vertex Cover

Algorithms: Design and Analysis, Part II

The Vertex Cover Problem

Given: An undirected graph $G = (V, E)$.

Goal: Compute a minimum-cardinality vertex cover (a set $S \subseteq V$ that includes at least one endpoint of each edge of E).

Suppose: Given a positive integer k as input, we want to check whether or not there is a vertex cover with size $\leq k$. [Think of k as "small"]

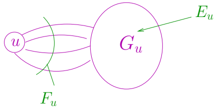
Note: Could try all possibilities, would take $\approx \binom{n}{k} = \Theta(n^k)$ time.

Question: Can we do better?

A Substructure Lemma

Substructure Lemma: Consider graph G, edge $(u, v) \in G$, integer $k \geq 1$. Let $G_u = G$ with u and its incident edges deleted (similarly, G_v). Then G has a vertex cover of size $k \iff G_u$ or G_v (or both) has a vertex cover of size $(k-1)$

Proof: $(\Leftarrow)$ Suppose G_u (say) has a vertex cover S of size $k-1$. Write E $= E_u$ (inside G_u) $\cup$ F_u (incident to u)



Since S has an endpoint of each edge of E_u, $S \cup \{u\}$ is a vertex cover (of size k) of G.

$(\Rightarrow)$ Let $S =$ a vertex cover of G of size k. Since (u, v) an edge of G, at least one of u, v (say u) is in S. Since no edges of E_u incident on u, $S - \{u\}$ must be a vertex cover (of size $k-1$) of G_u. QED!

A Search Algorithm

[Given undirected graph $G = (V, E)$, integer k]

[Ignore base cases]

(1) Pick an arbitrary edge $(u, v) \in E$.

(2) Recursively search for a vertex cover S of size $(k - 1)$ in G_u (G with u + its incident edges deleted).
If found, return $S \cup \{u\}$.

(3) Recursively search for a vertex cover S of size $(k - 1)$ in G_v.
If found, return $S \cup \{v\}$.

(4) FAIL. [G has no vertex cover with size k]

Analysis of Search Algorithm

Correctness: Straightforward induction, using the substructure lemma to justify the inductive step.

Running time: Total number of recursive calls is $O(2^k)$ [branching factor ≤ 2, recursion depth $\leq k$] (formally, proof by induction on k)

- Also, $O(m)$ work per recursive call (not counting work done by recursive subcalls)

$\Rightarrow$ Running time $= \boxed{O(2^k m)}$

Polynomial-time as long as $k = O(\log n)$

Remains feasible even when $k \approx 20$

Way better than $\Theta(n^k)$!

Exact Algorithms for NP-Complete Problems

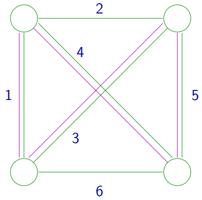
The Traveling Salesman Problem

Algorithms: Design and Analysis, Part II

The Traveling Salesman Problem

Input: A complete undirected graph with nonnegative edge costs.

Output: A minimum-cost tour (i.e., a cycle that visits every vertex exactly once).



$OPT = 13$

Brute-force search: Takes $\approx n!$ time
[tractable only for $n \approx 12, 13$]

Dynamic Programming: Will obtain $O(n^2 2^n)$ running time
[tractable for n close to 30]

Tim Roughgarden

A Optimal Substructure Lemma?

Idea: Copy the format of the Bellman-Ford algorithm.

Proposed subproblems: For every edge budget $i \in \{0, 1, \ldots, n\}$, destination $j \in \{1, 2, \ldots, n\}$, let
L_{ij} = length of a shortest path from 1 to j that uses at most i edges.

Question: What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

A) There is a super-polynomial number of subproblems

B) Can't efficiently compute solutions to bigger subproblems from smaller ones

C) Solving all subproblems doesn't solve original problem

D) Nothing!

A Optimal Substructure Lemma II?

Proposed subproblems: For every edge budget $i \in \{0, 1, \dots, n\}$, destination $j \in \{1, 2, \dots, n\}$, let
L_{ij} = length of shortest path from 1 to j that uses exactly i edges.

Question: What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

A) There is a super-polynomial number of subproblems
B) Can't efficiently compute solutions to bigger subproblems from smaller ones
C) Solving these subproblems doesn't solve original problem
D) Nothing!

A Optimal Substructure Lemma III?

Proposed subproblems: For every edge budget $i \in \{0, 1, \ldots, n\}$, destination $j \in \{1, 2, \ldots, n\}$, let
L_{ij} = length of shortest path from 1 to j with exactly i edges and no repeated vertices

Question: What prevents using these subproblems to obtain a polynomial-time algorithm for TSP?

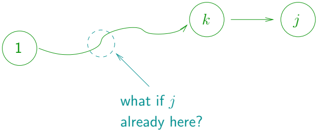
A) There is a super-polynomial number of subproblems
B) Can't efficiently compute solutions to bigger subproblems from smaller ones
C) Solving these subproblems doesn't solve original problem
D) Nothing!

A Optimal Substructure Lemma III? (con'd)

Hope: Use the following recurrence: $L_{ij} = \min_{k \neq 1,j} \{ L_{i-1,k} + c_{kj} \}$

shortest path from 1 to k, $(i-1)$ edges no repeated vertices

cost of final hop



Problem: What if j already appears on the shortest $1 \to k$ path with $(i-1)$ edges and no repeated vertices?
$\Rightarrow$ Concatenating (k_{ij}) yields a second visit to j (not allowed)

Upshot: To enforce constraint that each vertex visited exactly once, need to remember the <u>identities</u> of vertices visited in subproblem.

Exact Algorithms for NP-Complete Problems

A Dynamic Programming Algorithm for TSP

Algorithms: Design and Analysis, Part II

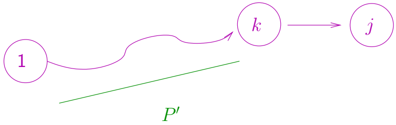
The Subproblems

Moral of last video: To enforce constraint that each vertex visited exactly once, need to remember the identities of vertices visited in a subproblem. [But not the order in which they're visited]

Subproblems: For every destination $j \in \{1, 2, \ldots, n\}$, every subset $S \subseteq \{1, 2, \ldots, n\}$ that contains 1 and j, let
$L_{S,j}$ = minimum length of a path from 1 to j that visits precisely the vertices of S [exactly once each]

Optimal Substructure

Optimal Substructure Lemma: Let P be a shortest path from 1 to j that visits the vertices S (assume $|S| \geq 2$) [exactly once each]. If last hop of P is (k, j), then P' is a shortest path from 1 to k that visits every vertex of $S - \{j\}$ exactly once. [Proof = straightforward "cut+paste"]



Corresponding recurrence:

$L_{S,j} = \min_{k \in S, k \neq j} \{L_{S-\{j\},k} + c_{kj}\}$

["size" of subproblem = $|S|$]

A Dynamic Programming Algorithm

Let A = 2-D array, indexed by subsets $S \subseteq \{1, 2, \ldots, n\}$ that contain 1 and destinations $j \in \{1, 2, \ldots, n\}$

Base case:

$$A[S,1] = \begin{cases} 0 \text{ if } S = \{1\} \\ +\infty \text{ otherwise} \end{cases}$$ [no way to avoid visiting vertex (twice)]

For $m = 2, 3, \ldots, n$ [m = subproblem size]

For each set $S \subseteq \{1, 2, \ldots, n\}$ of size m that contains 1

For each $j \in S, j \neq 1$

$A[S,j] = \min_{k \in S, k \neq j} \{A[S - \{j\}, k] + c_{kj}\}$ [same as recurrence]

Return $\min_{j=2,\ldots,n}\{\underbrace{A[\{1,2,\ldots,n\},j]}_{\text{min cost from 1 to } j \text{ visiting everybody once}} + \underbrace{c_{j1}}_{\text{cost of final hop of tour}}\}$

Running time: $O(\underbrace{n}_{\text{choices of } j} \cdot \underbrace{2^n}_{\text{choices of } S}) \; O(n) = O(n^2 2^n)$

choices of j · choices of S = # of subproblems

work per subproblem

Tim Roughgarden