Algorithms. A Collection of Problems

Zhengyang Song

April 29, 2017

1 Longest path in a DAG

We are given a directed acyclic graph G and two specific vertices s and t in G. Each edge in the graph has a length. Design a polynomial time algorithm that finds the longest path from s to t. (if there is not path from s to t, your algorithm should be able to detect the fact.)

Solution: First do a topology sort in polynomial time. If t is before s, return no path. We now only care about the edges between s and t, discard all the others.

Then run a DFS to enumerate the graph.

2 Finding the maximum area polygon

We are given a unit circle and n points on the circle. Design a polynomial time algorithm that, given a number $m < n$, finds m points (out of n points) such that the area of the polygon formed by the m points is maximized.

Solution: We can use a dynamic programming method. Suppose the polygon is rooted at 0, then we let $dp[m][i]$ be the maximum area we can get for a m-gon with the largest index vertex i.

$$dp[m][i] = max(\{dp[m-1][j] + area(0, i, j) | j < i\})$$

Then the answer for root 0 is just

$$answer = max(\{dp[n][i] | 1 \leq i \leq n\})$$

Then we enumerate all the possible root position (rather than 0) in time $O(n)$. That completes our polynomial algorithm.

3 Longest palindrome subsequence

A palindrome is a nonempty string over some alphabet that reads the same forward and backward. For example, aaaabaaaa, 00000, abcddcba are all palindrome. Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. Your algorithm should run in time better than $O(n^3)$.

Solution: Substrings are consecutive subsequences. We can use dynamic programming.

Let $dp[i][j]$ be the length of maximum subsequence from s[i:j+1]. Then we have

$$dp[i][i] = 1$$

$$dp[i][j] = 0, \forall i > j$$

$$dp[i][j] = max(dp[i+1][j], dp[i][j-1]) \text{ if } s[i] \neq s[j]$$

$$dp[i][j] = max(dp[i+1][j], dp[i][j-1], dp[i+1][j-1] + 2) \text{ if } s[i] == s[j]$$

Note dp array is in size $O(n^2)$, thus our algorithm running time.

4 Matrix-chain multiplication

We state the matrix-chain multiplication problem as follows: given a chain of matrices $A_1 A_2 \dots A_n$ where A_i has dimension $p_{i-1} \times p_i$. (Note the number of columns of A_i must be the same as the number of rows of A_{i+1}). Assume multiplying a $x \times y$ matrix with a $y \times z$ matrix takes xyz scalar multiplications. Design a polynomial time to find a fully parenthesization of the product $A_1 \dots A_n$ in a way that minimizes the number of scalar multiplications.

Solution: For a $x \times y$ matrix and a $y \times z$ matrix, one element in the result matrix needs y scalar multiplication, and there are $x \times z$ such elements. That is where the $x \times y \times z$ comes from.

For a computation with n matrices, we need $n+1$ numbers to store the state, i.e., $\{p_0, p_1, \dots, p_n\}$. Each multiplication is to pick one number p_k, remove it from the list, and add $p_{k-1} \times p_k \times p_{k+1}$ to the total cost. When the computation ends, there will only be two number left, which is $\{p_0, p_n\}$.

We use $dp[i][j]$ to denote the minimum cost when we do the multiplication operations on the subsequence $\{p_i, p_{i+1}, \dots, p_j\}$. By enumerating the last elements left between p_i and p_j, we have

$$dp[i][j] = \min\{dp[i][k] + dp[k][j] + p_i \times p_k \times p_j | k \in \{i+1, \dots, j-1\}$$

where the base case is simply

$$dp[i][i+1] = 0, \forall i \in \{0, 1 \dots, n-1\}$$

It can be shown that this algorithm works in $O(n^2)$.

5 Viterbi algorithm

We can use dynamic programming on a directed graph $G(V; E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds produced by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.

1. Describe a polynomial time algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \dots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label s, if such path exists. Otherwise, the algorithm should return NO-SUCH-PATH.

   Solution: We can just use a BFS-like algorithm. We start with an queue with a single element v_0. Then for each label in the sequence, we pop all the vertices. For each of the popped vertex, we examine the edges connected. If any edge is labeled with σ_i, we push the end vertex into the queue. During the process, we mark down the ancestor of each vertex. If any time the queue is empty, we return NO-SUCH-PATH. At last, if there are still any vertex in the queue, we can reconstruct the origin path based on the ancestor marked by the vertices.

2. Now, suppose that every edge (u, v) has an associated nonnegative probability $p(u, v)$ of traversing the edge, and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to be the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a random walk beginning at v_0 will follow the specific path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u. Extend your answer to part (a) so that if a path is returned, it is a most probable path starting at v_0 and having label s. Analyze the running time of your algorithm.

   Solution: The difference is that we store the maximum likelihood and the corresponding ancestor when we expand a node to its descendent. The running time is the same as before (?).

6 Edit distance

In order to transform one string x to a target string y,we can perform various edit operations. Our goal is,given x and y,to produce a series of edits that change x to y. We may choose from among edit operations:

- Insert a letter, (e.g., changing 100 to 1001 takes one insertion)
- Delete a letter, (e.g., changing 100 to 10 takes one deletion)
- Replace a letter by another (e.g., you need do one replacement to change 100 to 000).

Design an efficient algorithm that finds a series of edit operations that change x to y and the total number of edits is minimized.

Solution: Denote $dp[i][j]$ as the optimal number of operations to transform $x[:i]$ to $y[:j]$. Then we have

$$dp[i][j] = dp[i-1][j-1], \text{if } x[i] = y[j]$$

$$dp[i][j] = 1 + \min(dp[i][j-1], dp[i-1][j], dp[i-1][j-1]), \text{if } x[i] \neq y[j]$$

$$dp[i][j] = i, \text{if } j = 0$$

$$dp[i][j] = j, \text{if } i = 0$$

The answer is just $dp[len(x)][len(y)]$.

7 Four Russians Speedup

8 Solve the following two recurrences

- $T(n) = 2T(n/2) + n \log n$

  Solution: Using master theorem where $a = 2, b = 2, c = 1, k = 1$, we can get

  $$T(n) = n \log^2 n$$

- $T(n) = 2T(\sqrt{n}) + \log n$

  Solution: Let $n_1 = \log n$, $T_1(n_1) = T(2^{n_1}) = T(n)$, then

  $$T_1(n_1) = T(n) = 2T(\sqrt{n}) + \log n = 2T(2^{n_1/2}) + n_1 = 2T_1(n_1/2) + n_1$$

  Using master theorem with $a = 2, b = 2, k = 0$, we have

  $$T_1(n_1) = n_1 \log n_1$$

  Thus

  $$T(n) = \log n \log \log n$$

9 Maximal Common Subsequence

Solution: We use dp[i][j] to denote the answer for $a_1 \dots a_i$, $b_1 \dots b_j$. Then the outer wrapper is two loops (one for a and one for b), which makes the time complexity $O(mn)$. Then inner is just

$$dp[i][j] = \max(dp[i-1][j], dp[i-1][j-1] + 1) \text{ if } a[i] == b[j]$$

$$dp[i][j] = dp[i-1][j] \text{ if } a[i] != b[j]$$

Also note that when we update the i-th line of dp, we only use the information of the $i-1$-th line. That makes the space complexity $O(m+n)$.

10 Stick Game

Solution: We just maintain a bool array $win[i]$, which means if there are i sticks left, then the next player will win. Then we show how to compute $win[n]$ in polynomail time.

We initialize

$$win[0] = false, win[1] = true, win[2] = false, win[3] = true, win[4] = true$$

Then the recursion is as

$$win[i] = true \text{ if } win[i-1] = false \text{ or } win[i-4] = false$$

$$win[i] = false \text{ if } win[i-1] = true \text{ and } win[i-4] = true$$

11 Mongo Matrix

$$A[i,j] + A[k,l] \leq A[i,l] + A[k,j], \forall i < k \text{ and } j < l$$

$$\begin{pmatrix} 10 & 9 & 12 & 10 \\ 11 & 10 & 13 & 10 \\ 9 & 8 & 0 & 7 \\ 11 & 10 & 11 & 8 \end{pmatrix}$$

Let $f(i)$ be the column index of the leftmost minimum element of row i.

1. Show that $f(1) \leq f(2) \leq \cdots \leq f(m)$.
2. Given an $m \times n$ Monge matrix, design an algorithm that computes $f(1), \ldots, f(m)$ in $O(m + n \log m)$ time.

Solution:

1. For $i < j$, we want to prove $f(i) \leq f(j)$. Suppose otherwise, we have $f(i) = s > f(j) = t$. Then

   $$A[i,s] < A[i,t]$$
   $$A[j,t] \leq A[j,s]$$
   $$A[i,s] + A[j,t] < A[i,t] + A[j,s]$$

   It contradicts with the definition of Monge matrix.

2. We use divide-and-conquer. We first find $f(\lfloor m/2 \rfloor)$ in time $O(n)$. Then besed on the above conclusion, we can safely drop the bottom left quarter and top right quarter. Do the same operation for the top left quarter and bottom right quarter. Note that this procedure will end in $O(\log m)$ rounds. In each round, the total time cost for all the subtasks are $O(n)$. Furthermore, output all the results will cost time $O(m)$. That leads us to the time complexity of $O(m + n \log m)$.

12 Unit tasks scheduling

13 Coin changing

14 Schedule to minimize completion time

15

16

17

Given an undirected graph $G(V, E)$, a feedback set is a set $X \subset V$ with the property that $G - X$ has no circles. The undirected feedback set problem asks whether G contains a feedback set of size at most k. Show that the problem is NP-complete.

18 Rearrangeable Matrix

19 Turan's bound

Solution: Here we show how to get a indepent set with $|V| \geq \sum \frac{1}{deg(v)+1}$, then $\alpha(G) \geq |V|$.

Everytime, we select the vertex that with the least degree, put it into the indepent set, then delete it and all its neighbors. By doing each step of this, we add 1 to the final result, we distribute this to the $d+1$ nodes, which is

$$1 = (d+1)\frac{1}{d+1} \geq \frac{1}{d+1} + \frac{1}{v_1+1} + \frac{1}{v_2+1} + \cdots + \frac{1}{v_d+1}$$

20 Multiple Interval Scheduling

21

22 Densest k-Subgraph

Solution: Does G contain a subgraph with exactly k vertices and at least y edges?

First prove it is NP, then prove it is NP-hard.

For an instance (G, k) of CLIQUE, where $G = (V, E)$, we construct an instance of Dense Subgraph $(G, k, k(k-1)/2)$ in constant time.

23 Maximum coverage

Solution: Vertex Cover.

Also, we can reducing vertex cover to hitting set.

24 Polynomial Multiplication

$$P(x) = \prod_{i=1}^{n}(a_i x + b_i)$$

25 Longest increasing subsequence

Design a polynomial time algorithm that finds an longest increasing subsequence in the given sequence S.

Solution: We use $dp[i][j]$ to denote the answer of longest increasing subsequence starting from $s[i]$, with the constrain that the first character must be larger than $s[j]$. Suppose the suffix of s is from 1 to n, and $dp[i][0]$ means there is no constrain. So our goal is to get $dp[1][0]$.

The transfer between states are as follows:

- $dp[n+1][j] = 0$
- $dp[i][j] = max(1 + dp[i+1][i], dp[i+1][j])$, if $s[i] >= s[j]$ and $i <= n$
- $dp[i][j] = dp[i+1][j]$, if $s[i] < s[j]$ and $i <= n$

Note the dp only has $O(n^2)$ different states, that makes our algorithm polynomial.

26

1. Give a strategy to win this game in a finite number of moves.
2. Show that there is a strategy that guarantees a win if the number of cups N is a power of 2.
3. Show that there is not a strategy that guarantees a win if the number of cups N is not a power of 2.

27

28

A Hamitonian cycle is a cycle that goes through all vertices exactly once. Prove: Every graph with $n \geq 3$ vertices and minimum degree at least $\lceil n/2 \rceil$ has a Hamitonian cycle.

29 Catch a car without knowing anything

$$p + vt$$

Solution: Since we know the position of the car at time t is $p+vt$, so the problem is that whether we can determine p and v in finite time. That is, if we query position $p_0 + v_0 t$ at time t, and the car is not there, then we can rule out the combination (p_0, v_0). Furthermore, if we check (p_0, v_0) at time t, then we can check $(-p_0, v_0)$, $(p_0, -v_0)$, $(-p_0, -v_0)$ at the next three continous time points. So the problem is now turned into whether we can find a way to enumerate $\{(p, v)|p, v \in N\}$ so that a fixed pair (p_0, v_0) can always be encountered in finite time.

Just recall the halting problem and the diagonalization method, i.e., we put all (p, v) onto a two dimensional coordinate, then we enumerate alongside the diagonals: $(0, 0)$, $(0, 1)$, $(1, 0)$, $(0, 2)$, $(1, 1)$, $(2, 0)$, ... It can be easily shown that this will complete our algorithm.

30 Turan's bound

Solution: Here we show how to get a indepent set with $|V| \geq \sum \frac{1}{deg(v)+1}$, then $\alpha(G) \geq |V|$.

Everytime, we select the vertex that with the least degree, put it into the indepent set, then delete it and all its neighbors. By doing each step of this, we add 1 to the final result, we distribute this to the $d + 1$ nodes, which is

$$1 = (d+1)\frac{1}{d+1} \geq \frac{1}{d+1} + \frac{1}{v_1+1} + \frac{1}{v_2+1} + \cdots + \frac{1}{v_d+1}$$

31 Traveling salesman in the unit square

1. $$|v_1v_2|^2 + \cdots + |v_{n-1}v_n|^2 + |v_nv_1|^2 \leq 4$$

2. **Solution:** Cauchy's Inequality.

$$\begin{aligned} & (|v_1v_2| + \cdots + |v_{n-1}v_n| + |v_nv_1|)^2 \\ \leq\ & (1^2 + 1^2 + \cdots + 1^2) \cdot (|v_1v_2|^2 + \cdots + |v_{n-1}v_n|^2 + |v_nv_1|^2) \\ =\ & n \cdot (|v_1v_2|^2 + \cdots + |v_{n-1}v_n|^2 + |v_nv_1|^2) \\ \leq\ & n \cdot 4 \end{aligned}$$

32

1. **Solution:** BFS
2. **Solution:** BFS
3.

33

34

Initially, we have n empty bins. In each round, we throw a ball into a uniformly random chosen bin. Let T be number of rounds needed such taht no bin is empty. Show that $E[T] = nH_n$ where $H_n = \sum_{i=1}^{n} \frac{1}{i}$.

Solution: We denote E_m as the expected number of rounds needed when there are still m bins are left empty.

$$E_m = \frac{m}{n}(1 + E_{m-1}) + \frac{n-m}{n}(1 + E_m)$$
$$E_0 = 0$$

Then we have

$$E_m = E_{m-1} + \frac{n}{m}$$

$$\begin{aligned} E_n &= E_{n-1} + \frac{n}{n} \\ &= E_{n-2} + \frac{n}{n-1} + \frac{n}{n} \\ &= E_0 + \frac{n}{1} + \cdots + \frac{n}{n} \\ &= nH_n \end{aligned}$$

35

Show that finding a min-cost matching with exactly k edges in a bipartite graph can be solved in polynomial time.

Solution: We can turn this into a minimum-cost maximum-flow probelm. Add a source s and a sink t, together with a s' and t', where there is an edge with capacity k from s to s', and an edge with capacity 1 from s' to all the vertices in U. The same is with t' and t. The edges in E are all assigned with a capacity 1. All newly added edges are assigned with a cost 0. Then we just need to compute the minimum-cost maximum-flow of this new graph G', where a polynomial algorithm exists.

36

1. In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

   Solution: Suppose the set of edges in the maximum matching is M, the set of vertices in the minimum vertex cover is K. Then we want to prove $|K| \geq |M|$ and $|K| \leq |M|$.

   $|K| \geq |M|$: No vertex in a vertex cover can cover more than one edge of M, that is, the number of vertex cover cannot be less than $|M|$.

   $|K| \leq |M|$: Here we will find a vertex cover K with exactly $|M|$ vertices.

2. In any bipartite graph, the number of vertices in a maximum independent set equals the total number of vertices minus the number of edges in a maximum matching.

   Solution: A set of vertices is a vertex cover if and only if its complement is an independent set.

37

38

References