Nondeterministic Algorithms

Jerry Huang
Justin Zhang

May 13 2016

1 Introduction

A non-deterministic algorithm formally speaking is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. This basically boils down to using random number generation when solving problems.

2 Example Problem (192C)

I have an undirected graph consisting of n nodes, numbered 1 through n. Each node has at most two incident edges. For each pair of nodes, there is at most an edge connecting them. No edge connects a node to itself.
I would like to create a new graph in such a way that:

- The new graph consists of the same number of nodes and edges as the old graph.
- The properties in the first paragraph still hold.
- For each two nodes u and v, if there is an edge connecting them in the old graph, there is no edge connecting them in the new graph.

Help me construct the new graph, or tell me if it is impossible.
Input

The first line consists of two space-separated integers: n and m ($1 \leq m \leq n \leq 105$), denoting the number of nodes and edges, respectively. Then m lines follow. Each of the m lines consists of two space-separated integers u and v ($1 \leq u, v \leq n; u \neq v$), denoting an edge between nodes u and v.
Output

If it is not possible to construct a new graph with the mentioned properties, output a single line consisting of -1. Otherwise, output exactly m lines. Each

line should contain a description of edge in the same way as used in the input format. (TLE = 3 seconds)

This problem is pretty difficult if you attempt to find a deterministic solution to this problem. However, we can easily construct a non-deterministic algorithm via by randomly shuffling the nodes.

```
x = [1, 2, 3, ..., n]
fail = False
while True:
    shuffle(x)
    for i = 1 to m:
        if the edge (x[i], x[(i+1)%n]) is in input:
            fail = True
            # fail this iteration, repeat the entire procedure

    if not fail:
        # we obtain a solution!
        for i = 1 to m:
            print x[i], x[(i+1)%n]
        break
```

3 Primality Testing

Given a natural number n larger than two, determine whether it is prime.
A nondeterministic algorithm for this problem called the Fermat primality test is the following based on Fermat's little theorem:

1. Pick a random integer a such that $2 \leq a \leq n-2$
2. If $a^{n-1} \neq 1 \pmod{n}$, return answer composite
3. Repeat until close to time bound
4. Return answer as prime if $a^{n-1} \neq 1 \pmod{n}$ is never met

The runtime is $O(a * \log^2 n * \log\log n * \log\log\log n)$.

4 Reservoir Sampling

Reservoir sampling is a family of randomized algorithms for randomly choosing a sample of k items from a list S containing n items, where n is either a very large or unknown number. Typically n is large enough that the list doesn't fit into main memory. The naive algorithm is $O(k^2)$, which is pretty bad. To make it $O(n)$, we

```
# Initialize reservoir with the first k elements from stream
for i in range(k):
    reservoir[i].append(stream[i]);

# Iterate from the (k+1)th element to nth element
```

```
for i in range(k+1,n):
    # Pick a random index from 0 to i.
    int j = randint(0,i);

    # If the randomly  picked index is smaller than k, then replace
    # the element present at the index with new element from stream
    if j < k:
      reservoir[j] = stream[i];
```

5 Monte Carlo

Given an unsorted array A of n numbers and $\epsilon > 0$, compute an element whose rank (position in sorted A) is in the range $[(1–\epsilon) * \frac{n}{2}, (1+\epsilon) * \frac{n}{2}]$.
Following steps represent an algorithm that is $O((\log n) * (\log \log n))$ time and produces incorrect result with probability less than or equal to $2/n^2$.

1. Randomly choose k elements from the array where k=c log n (c is some constant)
2. Insert then into a set.
3. Sort elements of the set.
4. Return median of the set i.e. $\frac{k}{2}$th element from the set

6 Simulated Annealing

Simulated annealing (SA) is a probabilistic technique for approximating the global optimum of a given function. In brief, SA is an optimization on naive hill climbing. A solution is optimized, step-by-step, by repeatedly choosing a "neighboring" solution. At each step, the "neighboring" solution is randomly chosen, but less optimal solutions are weighted less and more optimal ones. This is to avoid local minimums.

Let's apply this to the Travelling Salesman Problem (TSP): given a set of points, find the most optimal tour. A basic SA algorithm may be:

1. Choose a random tour.
2. Randomly choose a neighboring tour – one in which two cities's visiting order are swapped
3. If this tour is more optimal, choose it
4. If not, choose this tour with a probability defined by a function of its cost and the temperature of the annealing process
5. Lower the temperature value, and repeat step two.

7 Random Problems

1. (Petr Mitrichev Contest 10 Problem C) There is a 6 by 6 square, with each unit square being either radioactive (1) or not (0). First, it sends the radioactivity of the unit square where it has landed. Then, it moves randomly into one of the adjacent unit squares, and sends the radioactivity of the unit square he's arrived to. Then he makes another random move, and so on. All in all, the transmission from the rover that has made n steps is a string of length $2n + 1$ and looks like 0L1U1R0D0L1L1U0. Unfortunately, the transmission comes to the Earth damaged — some of its characters are unrecognizable and thus replaced by question marks (?), and we also don't know where in the region the rover has landed. Output 6 lines with 6 characters each, 1 denoting the radioactive unit squares, and 0 denoting non-radioactive ones. If there are several possible solutions for which the transmission in the input is possible, output any.

2. (Google Code Jam 2016 Problem C) You have brought along J different jackets ($1 \dots J$), P different pairs of pants ($1 \dots P$), and S different shirts ($1 \dots S$). You have at least as many shirts as pairs of pants, and at least as many pairs of pants as jackets. ($J \leq P \leq S$.) Every day, you will pick one jacket, one pair of pants, and one shirt to wear as an outfit. If the Fashion Police find out that you have worn the exact same outfit twice, you will immediately be taken to the Fashion Jail! You will also immediately be taken to Fashion Jail if they find out that you have worn the same two-garment combination more than K times in total. A combination consists of a particular jacket worn with a particular pair of pants, a particular jacket worn with a particular shirt, or a particular pair of pants worn with a particular shirt. For example, in the set of outfits (jacket 1, pants 2, shirt 3) and (jacket 1, pants 1, shirt 3), the combination (jacket 1, shirt 3) appears twice, whereas the combination (pants 1, shirt 3) only appears once. You will wear one outfit per day. Can you figure out the largest possible number of days you can avoid being taken to Fashion Jail and produce a list of outfits to use each day?

3. Solve Mincut with worst case complexity as $O(V^4)$. This is called Karger's algorithm.