Class Meeting #9

COS 226 — Spring 2018

Mark Braverman

Time and location

The midterm is during lecture on

Monday, April 30, 11–12:20pm
Wednesday, May 2, 11–12:20pm

The exams will start and end promptly, so please do arrive on time.
Bring a charged laptop to the programming exam. Reboot it before coming to class, and open the SDE.

The rooms are

Precepts P01, P02, P02A, P05, P05A: McDonnell 02 (new room)

Precepts P03, P04, P04A, P04B, P05B : Friend 101 (traditional lecture room)

Rules

- Closed book, closed note.
- You may bring one 8.5-by-11 sheet (two sides) with notes in your own handwriting to the exam.
- No electronic devices (including calculators, laptops, and cell phones). Headphones attached to audio devices are also prohibited.
- Discussing the exam with others until solutions are posted is a violation of the Honor Code.

Consider the 4-SUM problem: *Given N integers, do any 4 of them sum up to exactly 0?*

(a) Consider the following brute-force solution (we ignore integer overflow).

```
public static fourSum(int[] a) {
   int N = a.length;
   for (int i = 0; i < N; i++)
      for (int j = i+1; j < N; j++)
         for (int k = j+1; k < N; k++)
            for (int l = k+1; l < N; l++)
               if (a[i] + a[j] + a[k] + a[l] == 0) return true;
   return false;
}
```

What is the order of growth of the worst-case running time? Circle the best answer.

N $N \log N$ N^2 N^3 N^4 2^N

(b) Design an algorithm for 4-SUM that runs in $O(N^2)$ time and uses $O(N^2)$ memory. Assume that you have access to a hashing-based symbol table that can `put()` and `get()` integer keys in constant time per operation.

Consider the 4-SUM problem: *Given N integers, do any 4 of them sum up to exactly 0?*

(a) N^4

(b) For each i and j, put the integer key $a[i] + a[j]$ in a hash table. Assoicate with each integer key the list of all pairs of indices that sum to that value.

For each k and l, check whether the key $-(a[k] + a[l])$ is in the hash table. If so, scan the list of all pairs of indices taht sum to that value. If such a pair exists whose indices i and j are disjoint from k and l, then we have four distinct array entries that sum to exactly 0.

The first phase takes $O(N^2)$ time and $O(N^2)$ space.

The second phase can be inefficient because searching through the list can take too long (if too many of the entries at the beginning of the list have indices that are not disjoint from k and l). For example, if the array contains $20, 20, 20, \ldots, 20, 10, -40, 30, 0$, then the list of all pairs that sum to 30 will be $20 + 10, 20 + 10, 20 + 10, \ldots 20 + 10, 30 + 0$. Thus, in the second pass, the algorithm will waste alot of time trying to find a match for 10-40 because it is not disjoint from all of the 20+10 entries.

To avoid this bottleneck, preprocess the original array so that at most 4 copies of each value remain. Now, when scanning the list for a sum that complements k and l, only scan the first 9 entries in the list for the sum $-(a[k] + a[l])$. Ther can be at most 4 pairs of indices in the list with k as one of the indices and at most 4 pairs with l as one of the indices. By the 9th entry, we will have found a pair of indices disjoint from k and l (or we will have exhausted the list). Thus, we only need to do a constant amount of work for each k and l.

Given a directed graph with V vertices and E edges, design an efficient algorithm to find a directed cycle with the minimum number of edges (or report that the graph is acyclic).

Your answer will be graded on correctness, efficiency, clarity, and succinctness. For full credit, your algorithm should run in $O(EV)$ time and use $O(E+V)$ space. Assume $V \leq E \leq V^2$.

Solution

(a) The critical observation is that the shortest directed cycle is a shortest path (number of edges) from s to v, plus a single edge $v \rightarrow s$.

```
For each vertex s:
   * Use BFS to compute shortest path from s to each other vertex.
   * For each edge v->s entering s, consider cycle formed by
     shortest path from s to v (if the path exists) plus the edge v->s.
Return shortest overall cycle.
```

(b) The running time is $O(EV)$.
The single-source shortest path computation from s takes $O(E+V)$ time per using BFS. Finding all edges entering s takes $O(E+V)$ time by scanning all edges (though a better way is to compute the reverse graph at once and access the adjacency lists). We must do this for each vertex s. Thus, the overall running time is $O(EV)$.

(c) The memory usage is $O(E+V)$.
BFS uses $O(V)$ extra memory and we only need to run one at a time. (A less efficient solution is to compute a V-by-V table containing the shortest path from v to w for every v and w. This uses $O(V^2)$ memory.)

Given a *directed* graph G with positive edge weights and a *landmark* vertex x, your goal is to find the length of the shortest path from one vertex v to another vertex w that passes through the landmark x.

(For example, Federal Express packages are routed through x = Atlanta.)

(a) Describe a $O(E \log V)$ algorithm for the problem. Justify briefly why your proposed algorithm is correct.

(b) Now suppose that you will perform many such shortest path queries for the same landmark x, but different values of v and w. Describe how to build a data structure in $O(E \log V)$ time so that, given the data structure, you can process each query in *constant time.*

(a) Compute the shortest path from v to x using Dijkstra's algorithm. Then compute the shortest path from x to w using Dijkstra's algorithm. Concatenate the two paths.

Correctness follows since all of the edge weights are positive: if the shortest landmark path used a non-shortest path from v to x, we could shorten it by substituting a shortest path from v to x. The same argument applies to the path from x to w.

(b) Pre-compute the following two quantities. Here x is fixed, and we compute the quantity for every vertex u.

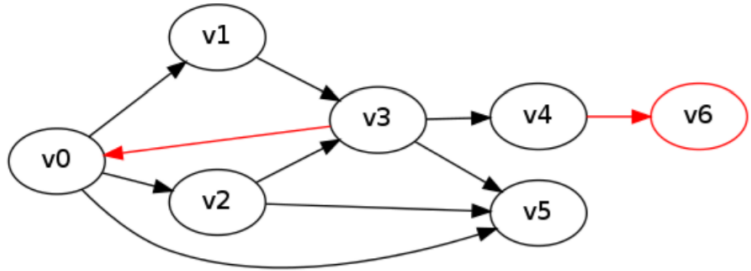
- $\bar{d}(u, x)$ = length of the shortest path from u to x.
- $d(x, u)$ = length shortest path from x to u.

Use Dijkstra's algorithm (with x as the source) to compute $d(x, u)$. This computes $d(x, u)$ for every vertex u in $O(E \log V)$ time. Use Dijkstra's algorithm on the reverse graph $\bar{G}$ (with x as the source) to compute $\bar{d}(u, x)$.

To process a shortest landmark path query from v to w, return $\bar{d}(v, x) + d(x, w)$.

(S'13 Final)

In this improved algorithm we start by setting pathCount inside each node to 0. When processing an edge $a \rightarrow b$, `if distTo[b] = distTo[a] + 1`, we increment `pathCount[b]` by `pathCount[a]` (since vertex a provides a new set of shortest `distTo[a]`). If `distTo[b] < distTo[a] + 1`, we've found a new shortest path, and set `pathCount[b]` to `pathCount[a]`. If `distTo[b] > distTo[a] + 1`, we do nothing since the path(s) under consideration is too long to be considered.



For example, for the first graph above, our graph state would be given by:

	distTo	edgeTo	pathCount
v0	0	(don't care)	1
v1	1	(don't care)	1
v2	1	(don't care)	1
v3	2	(don't care)	2
v4	3	(don't care)	2
v5	1	(don't care)	1
v6	∞	(don't care)	0

As an example, when the edge from $v4 \rightarrow v6$ is processed, the algorithm will see that `distTo[v4] < distTo[6] + 1`, and thus `pathCount[v6]` will be set equal to `pathCount[v4]`.

In data compression, a set of binary code words is *prefix-free* if no code word is a prefix of another. For example, $\{01, 10, 0010, 1111\}$ is prefix free, but $\{01, 10, 0010, 10100\}$ is not because 10 is a prefix of 10100.

(a) Design an efficient algorithm to determine if a set of binary code words is prefix-free.

Your answer will be graded on correctness, efficiency, clarity, and succinctness.

What is the order of growth of the worst-case running time of your algorithm as a function of N and W, where N is the number of binary code words and W is the total number of bits in the input?

Solution

(a) Insert all of the codewords into a binary trie, marking the terminating nodes. The set of string is not prefix-free if when inserting a codeword (i) you pass through a marked node (an existing codeword is a prefix of the codeword you are inserting) or (ii) the node you mark is not a leaf node (the codeword you're inserting is a prefix of an existing codeword).

O(N W)

A *tandem repeat* of a base string `b` within a string `s` is a substring of `s` consisting of at least one consecutive copy of the base string `b`. Given `b` and `s`, design an algorithm to find a tandem repeat of `b` within `s` of maximum length.

For example, if `s` is `"abcabcababcaba"` and `b` is `"abcab"`, then `"abcababcab"` is the tandem substring of maximum length (2 copies).

Your answer will be graded on correctness, efficiency, clarity, and succinctness. Let M denote the length of `b` and let N denote the length of `s`. For full credit, your algorithm should take time proportional to $M + N$.

* consider R to be constant

This problem is a generalization of substring search (is there at least one consecutive copy of b within s?) so we need an algorithm that generalizes substring search.

Create the Knuth-Morris-Pratt DFA for k copies of b, where $k = \lfloor N/M \rfloor$. Now, simulate DFA on input s and record the largest state that it reaches. From this, we can identify the longest repeat.

What's next

Algorithms and related courses:

- COS326: functional programming
- COS340: reasoning about computation
- COS324: intro to machine learning
- COS423: theory of algorithms
- COS521: advanced algorithms
- ORF307: optimization

Many COS courses have just COS226 as prerequisite (but best to take COS217 asap).