CS 170 Dis 02

Released on 2018-09-10

1 Cubed Fourier

(a) Cubing the 9^{th} roots of unity gives the 3^{rd} roots of unity. Next to each of the third roots below, write down the corresponding 9^{th} roots which cube to it. The first has been filled for you. *We will use ω_9 to represent the primitive 9^{th} root of unity, and ω_3 to represent the primitive 3^{rd} root.*

$\omega_3^0 : \omega_9^0,$,

$\omega_3^1 :$, ,

$\omega_3^2 :$, ,

(b) You want to run FFT on a degree-8 polynomial, but you don't like having to pad it with 0s to make the (degree+1) a power of 2. Instead, you realize that 9 is a power of 3, and you decide to work directly with 9th roots of unity and use the fact proven in part (a). Say that your polynomial looks like $P(x) = a_0 + a_1x + a_2x^2 + \ldots + a_8x^8$. **How do you split $P(x)$ to use the fact proven in part (a) to your advantage?** Provide either the polynomial, or explain how the vector can be divided to recurse on. *Recall that for the FFT algorithm shown in the book, we split a given polynomial $Q(x) = A_e(x^2) + xA_o(x^2)$, and we define what $A_e(x^2)$ and $A_o(x^2)$ are. Correspondingly, in lecture you saw the $\vec{a}$ split into $\vec{a}_{even}$ and $\vec{a}_{odd}$.*

Solution:

(a) $\omega_3^0 : \omega_9^0, \omega_9^3, \omega_9^6$
$\omega_3^1 : \omega_9^1, \omega_9^4, \omega_9^7$
$\omega_3^2 : \omega_9^2, \omega_9^5, \omega_9^8$

(b) Let $P(x) = P_1(x^3) + xP_2(x^3) + x^2P_3(x^3)$
where $P_1(x^3) = a_0 + a_3x^3 + a_6x^6$.
and $P_2(x^3) = a_1 + a_4x^3 + a_7x^6$.
and $P_3(x^3) = a_2 + a_5x^3 + a_8x^6$.

2 Vandermonde Matrices

Recall that a square Vandermonde matrix is of the following form:

$$\begin{bmatrix} 1 & \alpha_1 & \alpha_1^2 & \ldots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \ldots & \alpha_2^{n-1} \\ & & \ldots & & \\ 1 & \alpha_n & \alpha_n^2 & \ldots & \alpha_n^{n-1} \end{bmatrix}$$

Some real matrices have the nice property that $M^{-1} = cM^T$ for some constant c, or even that $M^{-1} = M$. Show that if M is a *real* n-by-n Vandermonde matrix and $n > 2$, then M^{-1} is not equal to cM^T for some constant c. (Hint: It suffices to show that MM^T is not a diagonal matrix, i.e. at least one of its off-diagonal entries is non-zero).

(Why are we asking you to show this? As seen in the textbook, both evaluating a polynomial at n points and going from the value of a polynomial at n points to its coefficients were equivalent to solving for either x or b in the equality $Mx = b$, where M is a Vandermonde matrix, x is a vector of the polynomial's coefficients, and b is the value of the polynomial at the distinct points $\alpha_1, \alpha_2 \dots \alpha_n$, using the same α_i that define the Vandermonde matrix. In particular, for FFT the matrix M used has the nice property that its conjugate M^* is proportional to its inverse M^{-1}. This exercise shows that if we want to use a matrix of only real values in FFT, we won't be able to achieve this nice property, which is what allows inverse-FFT to look so similar to FFT.)

Solution:

As the hint suggests, if M^{-1} were equal to cM^T, then $MM^T = \frac{1}{c}MM^{-1} = \frac{1}{c}I$, which is a diagonal matrix. So showing MM^T is not a diagonal matrix suffices.

Now, for a Vandermonde matrix M, if any $\alpha_i = \alpha_j$ then $(MM^T)_{ij}$ is equal to $1 + \alpha_i^2 + \alpha_i^4 + \dots$ which is non-zero since α_i is real. So we only worry about the case where the α_i are distinct. Then, we claim that since there are at least three distinct α_i, there exists some i, j such that $\alpha_i\alpha_j \neq -1$. This is because if $\alpha_i\alpha_j = -1$ and $\alpha_i\alpha_k = -1$, then $\alpha_j\alpha_k$ cannot be equal to -1 since this implies $(\alpha_i\alpha_j)(\alpha_i\alpha_k)(\alpha_j\alpha_k) = (\alpha_i\alpha_j\alpha_k)^2 = -1$, i.e. the square of a real number is negative, a contradiction. Now for this i, j pair, $(MM^T)_{ij}$ is equal to $1 + (\alpha_i\alpha_j) + (\alpha_i\alpha_j)^2 + \dots + (\alpha_i\alpha_j)^{n-1}$. If $\alpha_i\alpha_j = 1$, then this is equal to n, i.e. not zero. Otherwise, this equals $\frac{1-(\alpha_i\alpha_j)^n}{1-\alpha_i\alpha_j}$. Since $\alpha_i\alpha_j \neq 1$ and $\alpha_i\alpha_j$ is real and $n > 2$, this term is non-zero because $\alpha_i\alpha_j = -1$.

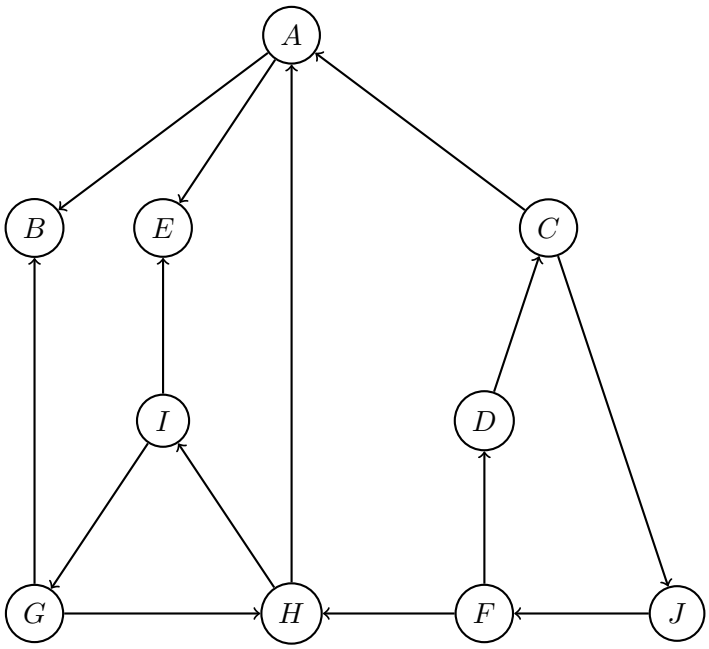
3 Graph Traversal

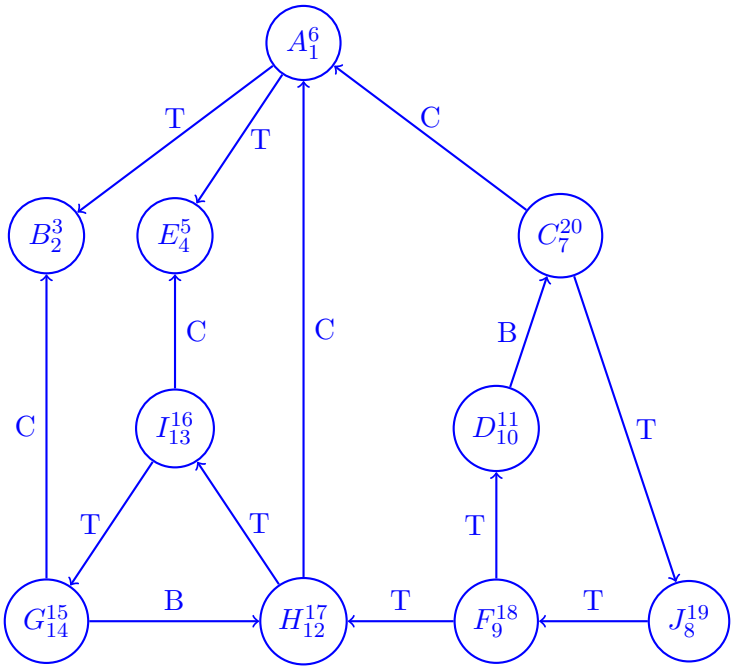
(a) For the directed graph above, perform DFS starting from vertex A, breaking ties alphabetically. As you go, label each node with its pre- and post-number, and mark each edge as **T**ree, **B**ack, **F**orward or **C**ross.

(b) What are the strongly connected components of the above graph?

(c) Draw the DAG of the strongly connected components of the graph.

Solution:



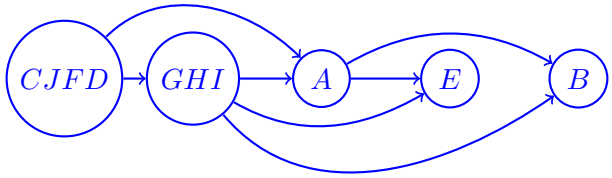


(a)

(b)

$$\{A\}, \{B\}, \{E\}, \{G, H, I\}, \{C, J, F, D\}$$

(c)



4 Short Answer

For each of the following, either prove the statement is true or give a counterexample to show it is false.

(a) If (u, v) is an edge in an undirected graph and during DFS, $\text{post}(v) < \text{post}(u)$, then u is an ancestor of v in the DFS tree.

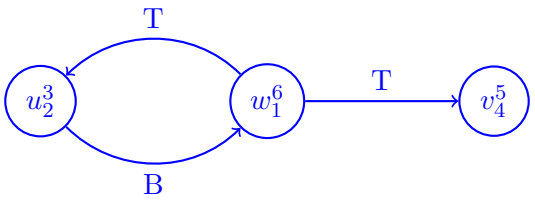
(b) In a directed graph, if there is a path from u to v and $\text{pre}(u) < \text{pre}(v)$ then u is an ancestor of v in the DFS tree.

(c) In any connected undirected graph G there is a vertex whose removal leaves G connected.

Solution:

(a) True. There are two possible cases: $\text{pre}(u) < \text{pre}(v) < \text{post}(v) < \text{post}(u)$ or $\text{pre}(v) < \text{post}(v) < \text{pre}(u) < \text{post}(u)$. In the first case, u is an ancestor of v. In the second case, v was popped off the stack without looking at u. However, since there is an edge between them and we look at all neighbors of v, this cannot happen.

(b) False. Consider the following case:



(c) True. Remove a leaf of a DFS tree of the graph.

5 True Source

Design an efficient algorithm that given a directed graph G determines whether there is a vertex v from which every other vertex can be reached. (Hint: first solve this for directed acyclic graphs. Note that running DFS from every single vertex is not efficient.)

Solution:

We provide two solutions below that both run in linear time (there may be many more).

Solution 1: In directed acyclic graphs, this is easy to check. We just need to see if the number of source nodes (zero indegree) is 1 or more than 1. Certainly if it is more than 1, there is no true source, because one cannot reach either source from the other. But if there is only 1, that source can reach every other vertex, because if v is any other vertex, if we keep taking one of the incoming edges, starting at v, we have to either reach the source, or see a repeat vertex. But the fact that the graph is acyclic means that we can't see a repeat vertex, so we have to reach the source. This means that the source can reach any vertex in the graph.

Now for general graphs, we first form the SCCs, and the metagraph. Now if there is only one source SCC, any vertex from it can reach any other vertex in the graph, but if there are more than one source SCCs, there is no single vertex that can reach all vertices.

Finding the SCCs/metagraph can be done in $O(|V| + |E|)$ time via DFS as seen in the textbook, and counting the number of sources in the metagraph can also be done in $O(|V|+|E|)$ time by just computing the in-degrees of all vertices using a single scan over the edges.

Solution 2: There is an alternative solution which avoids computing the metagraph altogether. The solution is to run DFS once on G to form a DFS forest. Now, let v be the root of the last tree that this run of DFS visited. Run DFS starting from v to determine if every vertex can be reached from v. If so, output v, if not, output that no true source exists.

It suffices to show no other vertex besides v can be a true source. In this case, if we determine v is not a true source, then saying there is no true source is correct. (Of course, if we find v is a true source, outputting it is also correct)

If there is a true source u, v can't reach u because v is not a true source, so the DFS must explore u before v. So nothing visited after v can be a true source, since v is the last root and thus all vertices visited after v are reachable from v. But every vertex visited before v must not be able to reach v, because otherwise the DFS would have taken a path from one of those vertices to v and thus v would not be a root in the DFS forest. So nothing visited before v can be a true source, since nothing visited before v can reach v. Thus v is the only candidate for a true source.

Since the algorithm just involves running DFS twice, it runs in linear time.

6 Path Problems on DAGs

Let G be a directed, acyclic graph.

(a) Give an efficient algorithm to compute the number of edges in the longest path in G.

(b) Give an efficient algorithm that takes G and two vertices s, t in its input and computes the number of paths from s to t.

Solution: For both parts of the problem, the key is to use the fact that G is a DAG to linearize G, and then solve a subproblem for vertices in G in the order given by the linearization. In solving the subproblems for later vertices, we can reuse the solution to the subproblem for earlier vertices.

(a) For this part, we solve the subproblem "what is the longest path in G that ends with i" for all vertices i in the order given by the linearization. If i is a source, then the only

path that ends in i is the empty path, of length 0. Otherwise, for any "parent" j of i (i.e. any j such that $(j,i) \in E$) any path ending in j can be extended into a path ending in i by appending the edge (j,i). In particular, the longest path that ends in i can be formed by taking the longest path ending in some parent j of i, and appending (j,i) to that path. This suggests the following algorithm:

LongestPath(G)

Let $v_1, \dots v_n$ be the linearization of G
Let L be a length n array (We will store the length of the longest path to v_i as $L[i]$)
for $i = 1, 2, \dots n$ **do**
  if v_i is a source vertex **then**
    $L[i] = 0$
  else
    $L[i] = 1 + \max_{j:(v_j,v_i)\in E} L[j]$
return the largest element of L

The correctness of the algorithm is given by the fact the longest path that ends in i is formed by taking the longest path ending in some parent j of i, and appending (j,i) to that path, as we argued above. Once we have the length longest path to all i, we just take the maximum to get the longest path in the entire graph. The algorithm can be implemented in linear time because linearization can be done in linear-time, we do n iterations in the for loop, and each edge participates in one iteration, so the for loop takes $O(n+m)$ time.

(b) For this part, we solve the subproblem "how many paths from s end in i" for all i. For any source besides s, the answer is 0. For s, the answer is 1, the empty path. For any other vertex i, any path from s to one of i's parents j can be extended into a path from s to i by adding the edge (j,i). In particular, all paths from s to i must go through some parent of i, so to count the number of paths from s to i we can just add up the number of paths from s to all of i's parents. This suggests the following algorithm:

CountPaths(G, s, t)

Let $v_1, \dots v_n$ be the linearization of G
Let L be a length n array (We will store the number of paths from s to v_i as $L[i]$)
for $i = 1, 2, \dots n$ **do**
  if $v_i = s$ **then**
    $L[i] = 1$
  else
    $L[i] = \sum_{j:(v_j,v_i)\in E} L[j]$ (if v_i is a source, this will set $L[i] = 0$)
return $L[t]$

The correctness of the algorithm is given by the fact that the number of paths from s to i is just the number of paths from s to all of i's parents, as we argued above. The algorithm can be implemented in linear time because linearization can be done in linear-time, we do n iterations in the for loop, and each edge participates in one iteration, so the for loop takes $O(n+m)$ time.