CS170–Spring 2015 — Solutions to Homework 3

Kevin Chau, SID 23816929, `cs170-pz`

Collaborators: None

1. Pattern matching, with tolerance for noise

(a) O(mn) algorithm for pattern matching
Input: s[0...m-1] a string of m characters, t[0...n-1] a string of n characters where $m < n$, and k maximum errors.
Output: a list of indices in t that match s within tolerance k.
Algorithm: for each index i in the range [0, n-1], if t[i] = s[0] initialize an error counter to zero. Loop over the characters of s with an index j in the range [0, m-1], and for every case where t[i + j] is not equal to s[j] increase the error counter. If after comparing all s[j] in the inner most loop the error counter is less than or equal to the tolerance k, append the index i to the output list of indices. After looping through all n indices of string t, return the output list of matching indices.

(b)

2. Polynomial from roots

1. Main Idea: An algorithm for computing the coefficient representation of a polynomial p from its n roots $r_1, ..., r_n$ where $p(r_i) = 0$. A naive algorithm can do this by multiplying all its factors $(x - r_i)$ in $O(n^2 \log n)$ time by having to do n polynomial multiplications via the O(nlogn) FFT based algorithm. The algorithm presented here has a running time strictly less than $O(n^2)$.

2. Pseudocode: Divide the roots into n/2 pairs. For each pair of roots (r_i, r_{i+1}), multiply $q = (x - r_i)$ and $p = (x - r_{i+1})$ using the FFT algorithm. Specifically, evaluate p and q into the FFT basis with FFT(p) and FFT(q), take their dot product in value representation, and then use the IFFT to recover the coefficient representations of p(x)q(x). After multiplying all pairs of root factors, again divide the resulting polynomials into pairs and iteratively apply FFT multiplications on the polynomial pairs. At the last step there will be a two polynomials of degree n/2 that can be FFT multiplied in $O(n \log n)$ time. After applying the inverse FFT to recover the coefficients of $\prod\limits_i (x - r_i)$, return the coefficients in an array.

3. Proof of correctness: The algorithm is guaranteed to return a polynomial where the coefficient of the highest degree term is 1. This is because each pairwise multiplication of the factors $(x - r_i), (x - r_{i+1})$ results in a polynomial whose highest degree coefficient is 1, and each successive pairwise multiplication will have the same result. Thus the final polynomial will have a leading coefficient of 1. Next we show that the algorithm returns the correct polynomial. It returns the correct polynomial because by definition it performs the multiplication $\prod\limits_i (x - r_i)$. It does so in a tree like structure, and since multiplcation is associative and the FFT is guanranteed to correctly multiply polynomials we know that the algorithm does this product of root factor polynomials correctly.

4. Running Time: $O(n(\log n)^2)$

5. Justification: We can think of this algorithm as traversing a recursion tree backwards. Instead of starting from a problem of size n and breaking the problem down into small sub problems, this algorithm starts with many small sub problems and works its way up to one large problem. However, the tree structure is essentially the exact same as a divide and conquer recursion. At every level, the problem size is doubled and the number of problems is halved, and so we have $\log n$ levels. This is just the reverse of a simple recursion where the problem size halves and the number of problems doubles. Thus we can just use master theorem, with the recurrence relation $T(n) = 2T(n/2) + O(n \log n)$. Here we used the fact that the FFT takes $O(n \log n)$ time to multiply polynomials for each sub problem. There is a general form of master theorem not cited in the textbook that says if the recurrence is $T(n) = aT(n/b) + O(n^c (\log n)^k)$ where $c = \log_b a$, then $T(n) = O(n^c (\log n)^{k+1})$ (see wikipedia on Master Theorem). In this case $\log_b a = \log_2 2 = 1 = c$, so we can apply the theorem to get $O(n(\log n)^2)$.

3. Triple Sum

1. Main Idea: An algorithm that determines if an array A[0...n-1] with A[i] in the range [-10n,10n] has indices i, j, k such that A[i]+A[j]+A[k] $= 0$.
2. Pseudocode:
3. Proof of Correctness:
4. Running Time:
5. Justification:

4. Cycle Detection

1. Main Idea: This algorithm detects whether a particular edge e = (u,v) in is contained within a cycle in the undirected graph G in linear time with the number of vertices $|V|$ and edges $|E|$.

2. Pseudocode:
   Input: A graph G with vertices V and edges E. A particular edge e = (u,v).
   Output: True if e is in a cycle in G , False otherwise.
   cycleDetect(G, e(u,v)):

```
(a) visited(v) = True
(b) explore(G,u)
(c) for each edge (v,w): if w is not equal to u and visited(w)==True, return True.
(d) else return false

subroutine explore(G,v):
visited(v) = True
for each edge (v,u) in E: if not visited(u): explore(u)
```

3. Proof of Correctness: The algorithm starts by marking one of the two vertices in e as visited; call this vertex v. It then calls the explore subroutine on the unvisted vertex u. The result of exploring u is that all vertices reachable from it will be marked True. Then the algorithm looks to see if any of v's neighbors has been visited. If one of v's neighbors other than u had been visited when we explored u, then its visited flag would be marked True. Any vertex that is reachable by both u and v with disjoint paths (ie paths that do not both have the edge e(u,v)) is a vertex in a cycle with edge e. Suppose our algorithm does indeed find a vertex w that was marked True by explore(u). Then the edge (v,w) and the path from u to w are guaranteed to be disjoint paths by the fact that we marked v before exploring u. Thus the algorithm looks for exactly this vertex w to find whether or not there is a cycle and so the algorithm is correct.

4. Running Time: $O(\|V\| + \|E\|)$

5. Justification: The subroutine explore(G,v) from the textbook runs in $O(\|V\| + \|E\|)$ and is only called on one vertex in this algorithm. The algorithm also checks the boolean value for all of v's neighbors. In the worst case graph, v could have at most $\|V\|$ neighbors. This means that the algorithm has a running time of $O(\|V\| + \|V\| + \|E\|) = O(2\|V\| + \|E\|)$ which just reduces down to $O(\|V\| + \|E\|)$. Thus the algorithm is linear in the number of vertices and edges.

5. Bipartite Graphs

(a) (a) Main Idea: A linear time algorithm for determining whether an undirected graph G=(V,E) is bipartite. This means that the graph can be partitioned into two set V_1 and V_2, where u,v are elements of V_1 if and only if (u,v) is not in E, and a,b are elements of V_2 if and only if (a,b) is not an edge in E.

(b) Pseudocode: The algorithm is just a modified depth-first search using a coloring modification to determine if a graph is bipartite. Color the starting node of DFS with WHITE. While visiting each of the root's children, color them BLACK. For each time DFS visits a child node, color it the opposite color as its parent (if parent was colored BLACK when visited, color the child WHITE as its been visited). For each edge e = (u,v) in E, check the color of u and the color of v. If u and v have the same color, then the algorithm returns FALSE; the graph is not bipartite. If for all edges e in E, color(u)≠color(u), then the graph is bipartite and the algorithm returns True.

(c) Proof of correctness: The two-coloring graph problem is just a way to assign nodes in a bipartite graph to a set. All nodes of one color are in one bipartitioned set and all nodes of the other color are in the other bipartitioned set. If a graph is indeed bipartite, then the algorith will be able to two-color the graph without any neighboring repeats. If the graph is not bipartite, we must show that the algorithm will not be able to two-color the graph successfully. After coloring all the nodes, the graph will check all the edges in the graph. If any edge has vertices with the same coloring, then the algorithm will find such edge and return that the graph is not bipartite.

(d) Running Time: $O(\|V\| + \|E\|)$

(e) Justification: DFS takes $O(\|V\| + \|E\|)$ to traverse a graph. Looking at the color of every vertex in each edge e in E takes $O(\|E\|)$ time. DFS is only called ones and looping over the edges is only performed once, so the run times add together and the running time of the algorithm is just $O(\|V\| + \|E\|)$.

(b) First we prove that if a graph is bipartite, it has no odd-length cycles. Suppose for the purpose of contradicion that the graph G has one single odd-lengthed cycle. Every other vertex in the cycle is in the same set, since they do not share an edge. If the cycle is an odd length k, then the first and last vertex in the cyle should be in the same set (since $v_1, v_3, ..., v_k$ are all in the same set). However, v_1 and v_k share an edge, since they are the first and last point in the cycle, which contradicts the fact that they are in the same bipartite set. Thus G not bipartite if it has an odd length cycle. Now we must prove that if G has no cycles of odd length, it is bipartite. Using a DFS coloring algorithm, we can see how this works. If there is a backedge in the DFS tree such that its vertices are the same color, that backedge must be contained in an odd length cycle since we are two coloring with children having opposite color from their parent nodes. Thus for G to have no odd length cycles, it must have no back edges that have same colored vertices. Thus the graph can be two colored and G is in fact bipartite.

(c) A graph with exactly one odd-length cycle needs at most 3 colors. This is easy to prove. Color the first 3 vertices in the cycle different colors A,B, and C. Then color the remaining vertices in the cycle as a two color problem. Since the remaining number of vertices after coloring the first 3 is an even number, we can guarantee that we can two color the rest of the cycle without having two consecutive colors next to the first vertex colored A or the third vertex colored C. The rest of the parts of the graph can just be two colored since they contain no odd length cycles and are therefore bipartite.