COMPSCI 230: Discrete Mathematics for Computer Science

Recitation 4: Proving Properties of Algorithms

Spring 2020 *Created By: David Fischer*

1. Preamble and Attendance - 5 minutes Make sure you are signed up for my digital hand! As the semester gets busier, we will be relying on it more and more during office hours, and might not have time to take out of office hours to help you get it set up. Today we examine how these proof techniques actually help us as computer scientists - number theorists are not the only people who care about rigorous proofs!

2. Bubble Sort

   Theorem 1. *Bubble sort always terminates.*

   Proof. We proceed with a direct proof. Recall the definition of the number of *inversions* in the list, namely, the count of $i < j$ such that $L[i] > L[j]$, where L is the list of numbers to be sorted. First, note that when bubble sort performs a swap, the number of inversions in the list is reduced by one. The following lemmas are helpful.

   Lemma 1. *Bubble sort terminates iff the list is sorted.*

   Proof. Suppose the list is sorted. Then, bubble sort does not perform a swap. Thus, bubble sort termintates. Suppose bubble sort terminates. Then, on the last iteration bubble sort does not find a pair $L[i]$ $L[i+1]$ such that $L[i] > L[i+1]$. So, the elements of L are such that $L[i] \leq L[i+1] \forall i \in [1, n-1]$, and so the list is sorted. □

   Lemma 2. *If an iteration of bubble sort does not perform a swap, then the list is sorted.*

   Proof. Suppose not. Then there is an unsorted list such that an iteration of bubble sort does not reduce the number of inversions. Since this list is unsorted, there is at least one inversion: $\exists i, j \in [n].(i < j) \wedge L[i] > L[j]$. Consider the largest i and smallest j in relation to that i which satisfy the previous; denote these i^* and j^*. Note that $j^* = i^* + 1$. (this is fairly evident, but for completeness we will prove it) Suppose not. Then $j^* >= i^* + 2$. However, consider $i^* + 1$. Either $L[i^* + 1] \geq L[i^*]$ or $L[i^* + 1] < L[i^*]$. In the former case, then $i^* + 1$ is a number greater than i^* which satisfies $L[i^* + 1] > L[j^*]$, and so we have a contradiction on the definition of i^*. In the latter, then $i + 1$ is a number smaller than j^* that satisfies $L[i^*] > L[i^* + 1]$, which contradicts our definition of j^*. So, $j^* = i^* + 1$. Then, bubble sort will compare $L[i^*]$ against $L[j^*]$ and will find that $L[j^*] < L[i^*]$, and so will swap the two. This contradicts our assumption that the iteration of bubble sort does not perform a swap. □

   The unsorted list has some number of inversions in it at the start. The number of inversions is bounded (namely, by $\binom{n}{2}$ where n is the length of L). Each iteration of bubble sort reduces the number of inversions by at least one, or if it does not, then the list is sorted. Therefore, the number of iterations of bubble sort is bounded by n, and so bubble sort terminates. □

3. Graphs

We begin with a few definitions that should be review from 201. Note that we slightly abuse notation by conditioning on whether a vertex x is a member of a set of edges (*explored*).

- A *graph* is a set of vertices $V = \{v_1, v_2, ..., v_n\}$ along with a set of edges $E = \{(v_i, v_j)\}$ for some $i, j \in [1, n]$.
- Breadth First Search is an algorithm which explores a graph using the following method. The algorithm is simplified to some extent for ease of illustration.

Algorithm 1: $BFS\_EXPLORE(v, explored, queue)$

Result: Shortest path tree T
for $(u \in V | (v, u) \in E)$ **do**
  if $u \notin explored$ **then**
    $explored := explored \cup \{(v, u)\}$;
    $queue.enqueue(u)$;
  end
end
$newv = explored.pop()$;
$EXPLORE(newv, explored, queue)$;
return *explored*

- Draw a simple example of BFS to hammer in the intuition.
- Depth First Search is an algorithm which explores a graph using the following method. The algorithm is simplified to some extent for ease of illustration.

Algorithm 2: $DFS\_EXPLORE(v, explored)$

Result: Shortest path tree T
for $(u \in V . (v, u) \in E)$ **do**
  if $u \notin explored$ **then**
    $explored := explored \cup \{(v, u)\}$;
    EXPLORE(u,*explored*);
  end
end
return *explored*

- Draw a simple example of DFS to hammer in the intuition.

Theorem 2. *The output of BFS is a rooted tree (that is, an undirected, connected, and acyclic graph).*

Proof. We start by showing that T is a connected subgraph of G. Suppose not, then there is either a node in T which is not in G, or T is not connected. Clearly, the first case leads to a contradiction of our definition of u in the for each statement of our algorithm. For the second, this implies there are two vertices w and x in T such that there is no path from w to x in T. However, there is a path from w to its parent, and to its parent, and etc until we reach

the root v. Thus, there is a path p_1 from w to v, and by the same reasoning there is a path p_2 from x to v. Then, $p_1 \cup p_2$ is a path from w to x, and so this contradicts our assumption that such a path does not exist. Now we must show that T is acyclic. Suppose T contained a cycle C of length n, that is a set of edges $\{(a_i, a_{i+1})\} \forall i \in [1, n]$ such that $a_{n+1} = a_1$. At some point, BFS must have reached a state where $\{(a_1, a_2), (a_2, a_3), ..., (a_{n-1}, a_n)\} \subseteq explored$. Note that we can relabel the vertices without loss of generality; the point is that there is one edge in the cycle not in the explored set. Now, note that the if statement will prevent the edge (a_n, a_1) from being added to the explored set. Thus, this cycle cannot be in T. So, T is an undirected, acyclic, connected subgraph of G, and so is a tree. □

Theorem 3. *Consider the rooted tree T which is output from $BFS(v)$ performed on an undirected graph $G = (V, E)$. For each other $u \in T$ such that $u \neq v$, there is only one unique path $p = (v, u_1, u_2, ..., u_{n-1}, u)$ of length n from v to u.*

Proof. Given an output tree T from BFS, suppose there were two paths in T. Then, there would be a cycle, which contradicts Theorem 2. □

Theorem 4. *Consider the rooted tree T which is output from $BFS(v)$ performed on an undirected graph $G = (V, E)$. For each vertex $u \in V$, $u \neq v$, the unique path in T from v to u is the shortest path from v to u in G.*

Proof. Consider the root v and an arbitrary node u in T. Let l be the length of the path from v to u in T. Then, suppose for the sake of contradiction there was a path in G of length $l - 1$. Without loss of generality, let this path be $p = (v, u_1, u_2, ..., u_{l-2}, u)$; note that this path is not in T. Then, note that u_i is the parent of u_{i+1}, and v is the parent of u_1 and u_{l-2} is the parent of u. Then, BFS would in fact act by adding (v, u_1), and then (u_1, u_2), and so on, until it added (u_{l-2}, u). Thus, in fact the path in G with a shorter length than the path in T would be traversed first, and so T would have that path in it, not the longer path. Note that this proof is informal; the formal proof leads into a discussion of induction, which we will explore next week. □

4. Graph Coloring

   (a) First, a few definitions.

   - A *coloring* of a graph $G = (V, E)$ is a function from each vertex to a set of colors C, $f : V \to C$. A coloring is *valid* is a coloring such that for every edge (u, v) in the graph the two vertices at either end of the edge map to different colors, i.e. $f(u) \neq f(v)$.
   - A graph $G = (V, E)$ is *bipartite* iff the set of vertices can be separated into two sets S and $S' = V \setminus S$ such that all edges "go across" the intersection: i.e. $\forall (u, v) \in E$, we have $u \in S \wedge v \in S'$.

   Theorem 5. *Given a cycle with an odd number of edges, $Y = (V, E)$ such that $|V|\%2 = 1$, no valid coloring exists that uses only two colors (i.e. the range of the function f has cardinality two).*

   Proof. Let $X = (V, E)$ be a cycle with m such that m is odd. Let $f : V \to C$ be a valid coloring, and suppose $C = \{c_1, c_2\}$ so that $|C| = 2$. We will show $\exists (u, v) \in E. f(u) = f(v)$ by contradiction. Suppose not. Consider an arbitrary vertex $v_1 \in V$, and label the vertices in an arbitrary direction around the cycle as $v_2, v_3, ..., v_m$ so that $(v_i, v_i + 1) \in E \forall i \in [1, m - 1]$, and $(v_m, v_1) \in E$. Suppose w.l.o.g. $f(v_n) = c_1$ for all odd n; all v_n with odd n have the same coloring, or else the coloring is invalid. Then, note that v_1 and v_m have the same coloring. Since $(v_1, v_m) \in E$, this proves that f is in fact not a valid covering. □

(b) Recall the definition of a coloring and BFS from last recitation and class. Consider the following algorithm to define a coloring on a graph $G = (V, E)$, given that the size of the range is two (i.e. to define a two-coloring $f : V \to C$, where $C = \{c_1, c_2\}$). Pick an arbitrary starting vertex v. Perform $BFS(v)$, with the following modification to the $EXPLORE(v, explored)$ subroutine:

Algorithm 3: $2COLOR\_BFS(v, queue, explored, f)$

```
Result: Coloring function f
for (u ∈ V|(v, u) ∈ E) do
    if u ∉ explored then
        explored := explored ∪ {u};
        if f(v) == c_1 then
            f(u) := c_2;
        else
            f(u) := c_1;
        end
    end
end
if !queue.empty() then
    2COLOR_BFS(queue.dequeue(), queue, explored, f);
end
return f
```

Theorem 6. *This algorithm is correct on a graph G if and only if G is bipartite.*

The backward direction $\leftarrow$ relies on induction, and so we leave the proof for next week.

Lemma 3. $\rightarrow$ *If the algorithm works, then the graph is bipartite.*

Proof. Suppose the algorithm works on some graph $G = (v, E)$. Then, define a set of nodes $A = \{n \in V | f(n) = c_1\}$, and $B = \{n \in V | f(n) = c_2\}$. Suppose for the sake of contradiction there were an edge between two nodes v_1 and v_2 in A (w.l.o.g.). Then, the algorithm would have set $f(v_1) \neq f(v_2)$, which contradicts our definition of A. □