COMPSCI 330: Design and Analysis of Algorithms **Feb 21, 2019**

Lecture 11: Graph Algorithms II

Lecturer: Rong Ge *Scribe: Haoming Li*

Overview

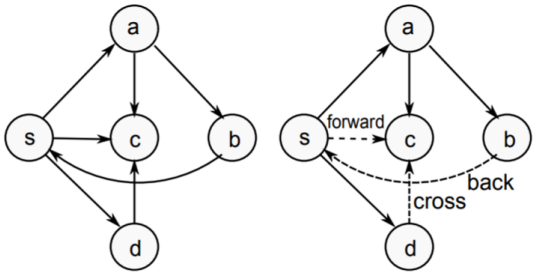
In this lecture, we compare breath-first search (BFS) and depth-first search (DFS), two ways to traverse a graph, and study their applications.

11.1 Applications of Depth-First Search

In the previous lecture, we detailed the DFS algorithm and introduced the notion of pre-order and post-order. To present our first application of DFS, we first observe that the edges we traverse as we execute a DFS can be classified into four types. During a DFS execution, the classification of edge (u, v), depends on whether we have visited v before in the DFS and if so, the relationship between u and v.

1. If v is visited for the first time as we traverse the edge (u, v), then the edge is a *tree edge*.
2. Else, v has already been visited:
   (a) If v is an ancestor of u, then edge (u, v) is a *backward edge*.
   (b) Else, if v is a descendant of u, then edge (u, v) is a *forward edge*.
   (c) Else, if v is neither an ancestor or descendant of u, then edge (u, v) is a *cross edge*.

Consider the following example: given directed graph on the left.



The tree formed by the solid edges on the right is a DFS tree, constructed by a DFS algorithm starting from vertex s. The solid edges are tree edges. (s, c) is a forward edge, (b, s) is a backward edge and (d, c) is a cross edge.

Similar to the struture of the DFS tree and the pre-/post-order, edge types also depend on the choices made in the DFS algorithm. We are now ready to introduce the first application of DFS: cycle finding.

11.1.1 Cycle Finding

In short, graph G has a cycle if and only if DFS finds at least one backward edge. The algorithm is precisely defined below.

Algorithm: DFS_visit(u)

Mark u as visited;
Mark u as in stack;
for *each edge* (u, v) **do**
  if *IF* v *is in stack* **then**
    (u, v) is a backward edge, found a cycle
  end
  if v *is not visited* **then**
    DFS_visit(v)
  end
end

Algorithm: DFS

for $u = 1$ *to* n **do**
  DFS_visit(u)
end

To prove the correctness of this algorithm, we first prove a lemma.

Lemma 11.1 *Suppose when DFS visits u, there is a path from u to v, and u is the only visited vertex on the path. Then u is on the stack when v is visited.*

Proof:

We prove this by induction on the length of the path.

Induction hypothesis: Suppose when DFS visits u, there is a path from u to v of length at most l, and u is the only visited vertex on the path, then u is on the stack when v is visited.

Base case: When $l = 1$, (u, v) is an edge and v is not visited when u is visited. We know the sequence of following 3 event: by the DFS_visit() algorithm, we must 1) visit u, then 2) consider edge (u, v), then 3) DFS_visit(u) returns. When edge (u, v) is considered:

- If v is already visited, then DFS_visit(v) happens between 1) and 2). In this time interval, u is always on the stack.
- If v is not visited, immediately call DFS_visit(v) so v is still visited while u is on stack.

Inductive step: Suupose IH is true for $l = k$. Consider w, the vertex right before v on the u-v path. We know there is a path from u to w of length k, and u is still the only visited vertex on the path. By IH, we know u is on the stack when w is visited, by design of algorithm, we know the sequence of the following five events: 1) visit u, then 2) visit w, then 3) edge (w, v) considered, then 4) DFS_visit(w) returns, then 5) DFS_visit(u) returns. When (w, v) is considered:

- If v is already visited, DFS_visit(v) happens between 1) and 3). u is on the stack for the entire time interval.
- If v is not visited, call DFS_visit(v) immediately, so u is still on the stack when v is visited.

■

Now that the lemma is proven, we are ready to prove the correctness of cycle finding.

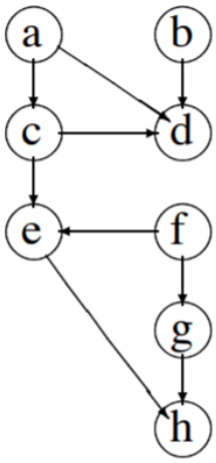
Proof:

Assume there is a cycle $(v_1, \ldots, v_k)$. Assume w.o.l.g. that v_1 is the first vertex visited in the cycle. By the lemma, we know v_1 is still on the stack when DFS visits v_k. Therefore, when edge (v_k, v_1) is considered, it must be a backward edge.

■

11.1.2 Topological Sort

Given a directed acyclic graph, we want to output an ordering of vertices such that all edges are from an earlier vertex to a later vertex. For example, consider the graph below.



One possible topological ordering would be f, g, b, a, c, d, e, h. This is, in fact, the reverse of the post-order output by a DFS. In fact, every reverse post-order is a valid topological sort. We will prove this by proving the following lemma.

Lemma 11.2 *For every edge (u, v), u must be later than v in post-order.*

Proof: Assume towards contradiction that there is an edge (u, v) where u is before v in post order.

- If u is visited before v. By the base case of the previous lemma, u is on the stack when v is visited. Therefore, u is after v in post order.
- If u is visited after v, then 1) visit v, then 2) visit u, then 3) DFS_visit(u) returns, then 4) DFS_visit(v) returns is the only possible sequence of events. This means when u is visited, v is on the stack. By DFS algorithm, there is a path from v to u, but (u, v) is also an edge. So the v-u path and edge (u, v) form a cycle. This contradicts with the assumption that the graph is acyclic.

■

11.2 Breath-First Search

Another way to traverse a graph is via Breath-First Search (BFS). In BFS, at any vertex, we visit its neighbors first (before its neighbors' neighbors.)

Algorithm: BFS_visit(u)

```
Mark u as visited;
Put u into a queue;
while queue is not empty do
    Let x be the head of the queue;
    for all edges (x, y) do
        if y has not been visited then
            Add y to the queue;
            Mark y as visited;
        end
    end
    Remove x from the queue;
end
```

Algorithm: BFS

```
for u = 1 to n do
    BFS_visit(u);
end
```

11.3 Applications of Breath-First Search

To introduce an application of BFS, we first introduce the idea of a *BFS tree*. If y is added to the queue while examining x, then (x, y) is an edge in the BFS tree.

11.3.1 Shortest Path

Given a graph as well as pair of edges, we want to find the path between them that minimizes the number of edges. BFS helps us to find exact that.

Lemma 11.3 *From starting point u, BFS finds the shortest path from u to every v reachable from u.*

Proof: We will again prove this by induction.

Induction Hypothesis: BFS finds shortest path from u to every v at a distance $\leq l$.

Base case: When $l = 1$, (u, v) is an edge. Since BFS first considers all neighbors of u, (u, v) will be considered and BFS finds the shortest path.

Inductive Step: Assume IH is true for $l = k$. Consider a vertex v at distance $k + 1$ to u. The shortest path from u to v has length $k + 1$. Consider w, the vertex immediately before v on the shortest path. The distance from u to w is k. By IH, BFS finds shortest path from u to w. Now consider the time w is processed in BFS.

- If v is already in the queue, v is added to the queue by a vertex w' that is processed before w. By design, $dis(u, w') \leq dis(u, w) = k$. Therefore, BFS finds a path of length $\leq k + 1$.
- If v is not in the queue, BFS will add v to the queue and find a path of length $k + 1$.

■