CSOR W4231.002 Lecture 7: Mar 24, 2014 Spring 2014

Eleni Drinea

In this lecture, we first introduce graphs and discuss how to represent graphs in a computer. Then we discuss the Breadth-First-Search (BFS) algorithm, which is an algorithm for searching a graph, that is, follows the edges of a graph so that it visits its vertices.

1 Graphs

Definition 1 *A directed graph consists of a finite set of vertices V and a set of directed edges E. An edge is an ordered pair of vertices (u, v).*

So in mathematical terms, a directed graph $G = (V, E)$ is a binary relation $E \subseteq V \times V$ on a finite set V. An undirected graph is the special case of a directed graph where $(u, v) \in E$ if and only if $(v, u) \in E$. In this case, we may indicate an edge between vertices u and v as the unordered pair $\{u, v\}$.

We use the following notational conventions for a graph $G = (V, E)$. Circles denote **vertices** (also called nodes). Lines denote **edges**. If the edges have arrows, then the graph is **directed**. Otherwise, it is **undirected**. Numbers on the edges denote **edge weights**. If there are no numbers, then the graph is unweighted and the weight of every edge is by default 1.

In undirected graphs, we define the **degree** of a vertex u, denoted by $deg(u)$, to be the number of edges incident to vertex u. In directed graphs, we define (a) the **in-degree** of a vertex u, denoted by $indeg(u)$, to be the number of edges entering vertex u; and (b) the **out-degree** of a vertex u, denoted by $outdeg(u)$, to be the number of edges leaving vertex u.

Finally, we usually denote the number of vertices by n and the number of edges by m, thus $|V| = n$ and $|E| = m$.

1.1 Why are graphs useful?

Graphs may be used to model various situations. Some examples are provided below.

Transportation networks: Vertices represent cities and edges represent highways connecting the cities; or, vertices are airports and edges indicate nonstop flights between airports. In both case, edges may have weights, that is, the distances between cities, or costs of flights, and we may want to find (a) whether it is possible to reach a vertex t from a vertex s and (b) what is the shortest (or cheapest) path to do so.

Today we will solve this problem for a fixed vertex s and unweighted graphs. We will later solve this problem for general edge weights and all pairs (s, t) in the graph.

Information networks: The World Wide Web can naturally be viewed as a directed graph: vertices correspond to web pages and there is an edge from vertex u to vertex v when there is a hyperlink from webpage u to webpage v. Modeling the web as a graph and understanding its structure allows us to infer other useful information such as identifying important pages.

Wireless networks: Vertices are devices sitting at locations in physical space and there is a directed edge from device u to device v if v is close enough to u to hear from it.

Dependency networks: In our examples so far, vertices and edges correspond to quite concrete entities. However there could be more abstract relationships: for example, given a list of functions in a large program we might want to check in what order to test our functions. Here we could think of functions corresponding to nodes and there is a directed edge from function f to function h if h calls f. So f should be tested before h. We will see how to tackle this problem in the next lecture.

1.2 Useful definitions

We begin with a review of several useful definitions.

- A **path** is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that consecutive vertices are adjacent (edge $(x_i, x_i + 1) \in E$ for all $1 \leq i \leq n - 1$).
- A path is **simple** when all vertices are distinct.
- A **cycle** is a simple path that ends where it starts, that is, $x_n = x_1$.
- The **distance** between between u and v is the length of the shortest path from u to v.
- An undirected graph is **connected** when there is a path between every pair of vertices.
- The **connected component** of a node u in an undirected graph is the set of all nodes in the graph reachable by a path from u.
- A directed graph is **strongly connected** when, for every pair of vertices u, v, there is a path from u to v and a path from v to u.
- The **strongly connected component** of a node u in a directed graph is the set of nodes v such that there is a path from u to v and a path from v to u.

1.3 Trees

Definition 2 *A tree is a connected acyclic graph (undirected graphs); or, a rooted graph such that there is a unique path from the root to any other vertex (all graphs).*

We can think of a tree as the minimal connected graph. It is the most widely encountered special type of graph. The following lemma summarizes important properties of trees.

Lemma 1 *If G is an undirected graph and any two of the following properties are true, then the third property is true and G is a tree:*

1. *G is connected;*
2. *G is acyclic;*
3. *$|E| = |V| - 1$.*

1.4 Bipartite graphs

In numerous situations including social networks and coding theory, it is natural to create graphs where the vertices are partitioned into two subsets such that there are no edges between vertices in the same subset. Such a graph is called bipartite and often denoted as $G = (X \cup Y, E)$, where X, Y are the two sets of vertices and E is the set of edges such that every edge has one endpoint in X and one endpoint in Y.

For example, suppose that we have 5 people and 5 jobs and for every person we have a list of the jobs he/she qualifies for. We can naturally form the bipartite graph in Figure 1, where $X = \{A, B, C, D, E\}$ and $Y = \{1, 2, 3, 4, 5\}$. We want to find a feasible one-to-one matching of people to jobs if one exists; this is also called a perfect matching.

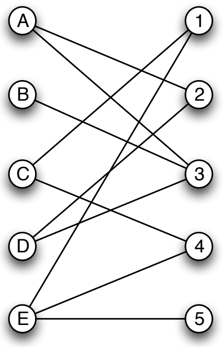

Figure 1: The graph of people and jobs. An edge in the graph denotes that a person qualifies for a job.

In this example, such a matching is not possible: consider subset $V_1 = \{A, B, D\} \subset X$. These three people qualify for the subset of jobs $\{2, 3\}$ only. Therefore we cannot match this subset of people in a one-to-one way with any subset of jobs, much less match the set X of all the people with jobs.

We will cover algorithms that can be used to find perfect matchings when we discuss max flow.

Bipartite graphs have a number of useful properties so we want to know when a graph is bipartite. Often, as in the example above, the graph is bipartite by construction. Other times it may not be obvious so we will need check for bipartiteness. In a later section we will discuss an algorithm to accomplish this task.

1.5 Running time of algorithm graphs

A graph algorithm is **linear** when it runs in time $O(|V| + |E|) = O(n + m)$.

For connected graphs, a lower bound for m is $\Omega(n)$. If the graph is simple, that is, if it has no multi-edges (multiple edges between the same pair of vertices) or self-loops (edges that start and end at the same vertex), an upper bound for m is $O(n^2)$. Hence for connected simple graphs, a linear algorithm runs in $O(n + m) = O(m)$ time.

A theorem that proves useful in arguing about the running time of graph algorithms is the following.

Theorem 1 *In any graph, the sum of the degrees of all vertices is equal to twice the number of the edges.*

Proof. Every edge is incident to two vertices, thus contributes 2 to the sum of the degrees of all vertices. $\square$

2 Representing graphs in a computer

There are two ways to represent a graph in a computer.

1. **Adjacency matrix:** An $n \times n$ matrix A such that $A[i, j] = 1$ if $(i, j) \in E$ and 0 otherwise.
   - **Advantages** of adjacency matrix:
     1. Check whether edge $(i, j) \in E$ in constant time.
     2. Easy to adapt so that $A[i, j]$ contains the weight of the edge.
     3. Suitable for dense graphs where $m = \Theta(n^2)$.

- **Drawbacks** of adjacency matrix:
  1. Requires $\Omega(n^2)$ space even if the graph is sparse, that is, even for $m = O(n)$.
  2. Precludes the possibility for a linear time algorithm for such graphs (at least in the cases where all matrix entries must be examined).

2. **Adjacency list:** An alternative representation for $G = (V, E)$ is as follows. We say that a vertex j is adjacent to a vertex i when $(i, j) \in E$. The adjacency list for vertex i is the list of vertices that are adjacent to vertex i. Maintain an array with n entries where each entry $A[i]$ points to the adjacency list of vertex i. (Instead of storing the actual vertices in each adjacency list, we may keep pointers to these vertices).

   The space required by this representation is the space required for the array of n pointers plus the sum of the lengths of all adjacency lists. If G is directed, we maintain the list of vertices that have incoming edges from v and the list of vertices that have outgoing edges to v. Thus the adjacency list of v contains $outdeg(v) + indeg(v)$ vertices. So total space is $\sum_v outdeg(v) + indeg(v) = 2m$. If G is undirected, the adjacency list of v contains $deg(v)$ vertices and total space is $\sum_v deg(v) = 2m$.

   So the total space required by the adjacency list representation is $O(n + m)$ and we will use this representation for all our algorithms that take linear or near-linear time.

   - **Advantages** of adjacency list:
     1. Requires space $O(n + m)$, thus allocates no unnecessary space.
   - **Drawbacks** of adjacency list:
     1. Searching for an edge in an adjacency list can take $O(n)$ time (why?).

To conclude, we use adjacency matrix when we need determine quickly whether an edge is in the graph, the graph is dense or the graph is small (it is a simpler representation). Otherwise, we use adjacency list.

3 BFS

Recall our transportation network example: we want to find if vertex t is reachable from vertex s. More generally, we may want to find all vertices t reachable from a specific vertex s in a graph.

This problem is also referred to as **s-t connectivity**, because it asks whether there is a path from vertex s to vertex t for every t in the graph.

One natural algorithm for this problem is Breadth-First-Search (BFS), which, in the case of unweighted graphs, also returns the shortest path from vertex s to every reachable vertex t.

The main idea of BFS is to explore the graph starting from a vertex s **outward in all possible directions**, adding nodes one "layer" at a time. That is, first add all nodes that have an incoming edge from s to form the first layer (these nodes are at distance 1 from s). Then add all nodes that have an incoming edge from a node in the first layer; these form the second layer. And so on and so forth.

For example, for the graph in Figure 3(a), BFS yields the tree in Figure 2(c).

3.1 Properties of the layers of the BFS tree

More formally the layers in BFS are formed as follows.

- Layer L_0 consists of s.
- Layer L_1 consists of all nodes that have an incoming edge from s.

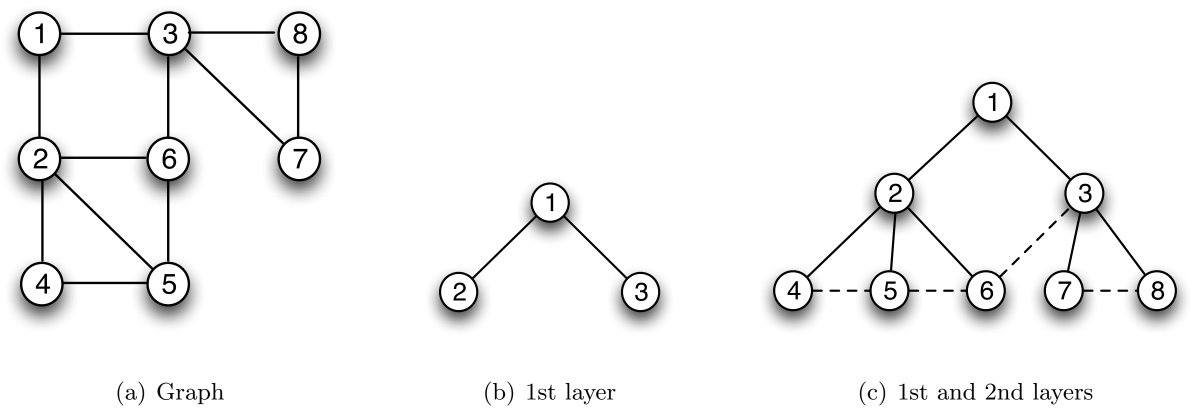

Figure 2: The construction of the breadth-first search tree for the graph in Figure 3(a), layer by layer. Solid edges belong to the tree while dotted edges belong to the graph but not to the tree.

- Having defined layers $L_0, L_1, \ldots, L_i$, layer L_{i+1} consists of all nodes that do not belong to a previous layer and have an edge from a node in layer L_i.

Since the graph produced as above is rooted at s and there is a path from s to every node in the graph, it is a tree. A node v reachable from s is added to the BFS tree when at some layer L_i a node u is explored such that for the first time there is an edge (u, v) in the graph. Then v is added to layer L_{i+1} and u becomes the parent of v in the BFS tree since u is responsible for "discovering" v. Also, a graph node fails to appear in the BFS tree rooted at s because there is no path from s to that node. Thus the BFS tree contains the set of nodes reachable from s.

A property of all BFS trees is that non-tree edges in the graph are either edges between nodes in the same layer or edges between nodes in adjacent layers.

Claim 1 *Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge in G. Then i and j differ by at most 1.*

Proof. W.l.o.g. suppose that $j \geq i$. At layer L_i, when x is explored

- either y is discovered then and thus added to the tree at layer L_{i+1}; or,
- y has already been discovered, thus y appears in the tree at some layer L_k with $k < i + 1$.

Hence y appears at layer L_{i+1} at the latest. It follows that $j \leq i + 1$. □

Claim 2 *For all nodes reachable by s, L_i contains the set of nodes that are at distance i from s.*

Proof. The proof is by induction on $0 \leq i \leq n - 1$. Conventionally, the distance from s of a node that is not reachable by s is ∞.

- **Induction Basis:** True for layer L_0.
- **Induction hypothesis:** Suppose that the vertices at layer L_i are the vertices that are at distance i from s for some $i > 0$.
- **Induction step:** The only vertices added to L_{i+1} are those that have an edge from some node in L_i and do not have an edge from any node in any previous L_k with $k < i$. It follows that the distance from s to such a vertex is $i + 1$.

□

Hence BFS computes shortest paths from s to every node reachable by s, when the graph is unweighted.

Note that the order in which the vertices in the graph are visited matters for the final tree but **not** for the distances computed.

3.2 Implementing BFS

To store the discovered nodes at layer L_i (with the goal to explore them later), we use a queue. A queue is a FIFO (First-In First-Out) data structure, where elements are added to the end of the queue, and extracted from the head of the queue. A queue may be implemented as a double-linked list so there are explicit pointers to the head and the end of the queue. Then enqueue and dequeue operations take constant time.

The following pseudocode computes shortest paths (distances) from the source s to every node reachable by s. It also stores the parent of every node in the BFS tree.

```
BFS( $G = (V, E), s \in V$ )
  array $distance[V]$ initialized to $\infty$
  array $discovered[V]$ initialized to 0
  queue $q$
  $discovered[s] = 1$
  $distance[s] = 0$
  $parent[s] = NIL$
  enqueue($q, s$)
  while  $size(q) > 0$  do
      $u$ =dequeue($q$)
      for $(u, v) \in E$ do
          if $discovered[v] = 0$  then
              $discovered[v] = 1$
              $distance[v] = distance[u] + 1$
              $parent[v] = u$
              enqueue($q, v$)
          end if
      end for
  end while
```

The running time of this algorithm is $O(n + m)$ since every node is enqueued at most once and the total time spent in the for loop is $\sum_{u \in V} deg(u) = O(m)$.

4 Applications of BFS

4.1 Finding connected components in undirected graphs

BFS naturally produces the connected component containing vertex s, that is, the set of nodes reachable from s. We denote this set by $R(s)$. Given $R(s)$, answering s-t connectivity is easy: simply check if t belongs to $R(s)$.

To compute all the connected components in an undirected graph, we first observe how $R(u)$ and $R(v)$ compare, for any pair of vertices u and v in G.

Fact 1 *For any two vertices u and v their connected components are either the same or disjoint.*

Proof. Consider any two nodes u, v, such that there is a path between u and v: then $R(u) = R(v)$ since any node x in $R(v)$ is reachable from u (how?) and every node y in $R(u)$ is reachable from v.

Now consider any two nodes u, v such that there is no path between them: their connected components are disjoint. If not, there is a node w that belongs to both components. Thus there is a path between u and w and a path between v and w. Then there is a path between u and v, which contradicts our hypothesis. □

Given Fact 1, the following algorithm computes all the connected components of a graph.

1. Start with an arbitrary node u and run BFS
2. Continue with any node v that has not been visited by the BFS on u and run BFS on v.
3. Iterate until all nodes have been visited.

4.2 Testing bipartiteness

Suppose we need to answer whether an input graph $G = (V, E)$ is bipartite or not, that is, whether we can partition V into two subsets X and Y such that all edges have one endpoint in X and one endpoint in Y.

This problem is equivalent to the following problem: *given a graph G, is it possible to color the vertices in G with 2 colors so that no edge has two endpoints with the same color?*

The two problems are equivalent because a graph is bipartite if and only if it is 2-colorable: we can always 2-color a bipartite graph by coloring the vertices in X red and the vertices in Y white. And, if a graph is 2-colorable, then it is bipartite.

To answer whether a graph is not bipartite, we need understand the structural properties of such graphs. The simplest graph that is not bipartite —equivalently, cannot be 2-colored— is a triangle, that is, a cycle of length 3. More generally, if a graph contains a cycle of odd length then it cannot be bipartite: there is no way to 2-color such a cycle, let alone the whole graph. Hence

Fact 2 *If a graph contains an odd cycle then the graph is not bipartite.*

The following algorithm checks for bipartiteness by attempting to 2-color its input graph G.

1. Start BFS from any vertex s; color s red.
2. Color white all vertices in the first layer of the BFS tree. If there is an edge between two vertices in layer 1, stop and declare the graph not bipartite.
3. Otherwise, continue from layer 1, coloring red the vertices in even layers and white in odd layers.

The output of this algorithm is as follows: either it 2-colored all layers of the graph without finding an edge between vertices in the same layer so it declares the graph bipartite; or, it stopped at some level because an edge was found between two vertices of the same level, and declared the graph non-bipartite.

We will now show that this algorithm correctly determines whether a graph G is bipartite or not by showing that it declares a graph non-bipartite only when it detects an odd-length cycle in G.

Claim 3 *Let G be a connected graph, and let L_1, L_2, … be the layers produced by BFS starting at node s. Then exactly one of the following two is true.*

- *There is no edge of G joining two nodes of the same layer. In this case, G is bipartite and we cannot find an odd length cycle.*

- *There is an edge of G joining two nodes of the same layer. In this case, G contains an odd length cycle and hence is not bipartite.*

Proof. First consider the case where no edge of G joins two nodes of the same layer. Since the graph edges are either edges between nodes in adjacent layers in the BFS tree or edges between nodes in the same layer, in this case there are only edges between nodes in adjacent layers in the BFS tree. Our coloring procedure gives nodes in adjacent layers different colors, hence it produces a valid 2-coloring of the whole graph. Therefore the graph is bipartite.

Now consider the case where there is an edge of G joining two nodes u and v in the same layer. Obviously G is not 2-colorable by our algorithm since nodes in the same layer are assigned the same color so there is an edge in G that has two endpoints with the same color. So our algorithm declares G non-bipartite. Can we show existence of an odd cycle to prove that G is indeed non-bipartite (and thus, not 2-colorable by any algorithm)?

Let z be the lowest common ancestor of u and v in the BFS tree rooted at s (z might be s). Suppose z is at layer L_i with $i < j$. Then consider the following cycle in the graph G: first follow the path of BFS tree edges from z to u; then follow edge (u, v); and then follow the path of BFS tree edges from v to z. These paths form a cycle that starts and ends in z and consists of $(j - i) + 1 + (j - i) = 2(j - i) + 1$ edges, hence has odd length. Therefore G is not bipartite. $\square$

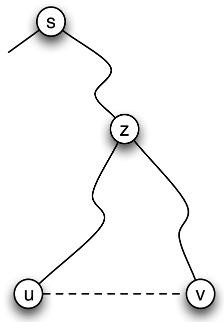

Figure 3: The path from z to u to v and back to z is an odd-length cycle.

Corollary 1 *A graph is bipartite if and only if it contains no odd length cycle.*

4.3 Finding the strongly connected component of a vertex

In directed graphs, the notion of connectivity is more subtle: saying that a graph is connected when there is a path from u to v but no path from v to u is misleading. We say that G is **strongly connected** if for every pair of vertices u and v, there is a path from u to v and a path from v to u.

Running BFS from s returns the set T of nodes that are reachable from s in G. However we do not know if there are paths **from** these nodes **to** s. To compute the set of nodes that can reach s, consider the graph G^r where the directions of the edges in G are reversed. Then, if there is a path from a node u to s in G, there is a path from s to u in G^r. Thus if we run BFS in G^r starting from s, we will obtain the set T' of vertices that have paths to s in G.

The strongly connected component of s consists of the vertices that belong to both T, T'.