Lecture 17: Breadth-First Search (BFS)

built on 2019/06/18 at 13:07:54

1 Graph Traversal

A few lectures ago, we saw several ways to traverse/search a tree. We can generalize some of those ideas to a more general structure—the graph.

Graph searching or traversal is one of the most fundamental tasks on graphs. One starts at some vertex, or a set of vertices, and visits new vertices until there is nothing left to search. To perform a search, it is important to be systematic because we need discipline to make sure that we visit all vertices that we can reach and that we do not visit the same vertex multiple times.

This generally requires recording what vertices we have already visited so we don't visit them again.

A main standard method for searching graphs is breadth-first search (BFS). This specifies a particular order in which vertices of a graph are visited. In the end, we'll have visited every vertex reachable from the starting point.

2 Breadth First Search

Breadth-first search (BFS) can be applied to solve a variety of problems including:

- finding all the vertices reachable from a vertex v;
- finding if an undirected graph is connected;
- finding the shortest path from a vertex v to all other vertices;
- determining if a graph is bipartite;
- bounding the diameter of an undirected graph; and
- many more.

It can be applied to both directed and undirected graphs. We'll only focus on undirected graphs in this lecture.

To discuss BFS, we'll need a few definitions:

- Let $R_G(u)$ indicate all the vertices that can be *reached* from u in a graph G (i.e., vertices v for which there is a path from u to v in G).
- The *distance* $\delta_G(u, v)$ from a vertex u to a vertex v in a graph G is the shortest path (minimum number of edges) from u to v.

Hence, for any starting vertex s, every vertex $v \in R_G(s)$ has a finite distance value, i.e., $\delta_G(s, v) < \infty$.

2.1 Basic Idea

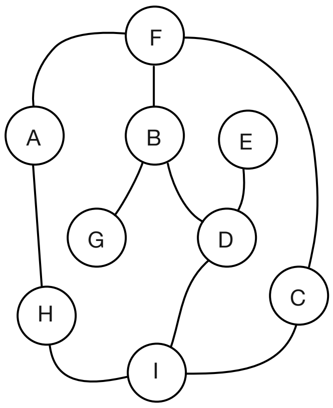
To gather some intuition for BFS, we'll consider the following strategy. We start at a *source* vertex. We will then explore the graph level by level:

- The 1st level to visit is all vertices that are reachable within exactly 1 step from s, i.e., all the neighbors of s. Let F_1 be the set of these vertices at level 1. Then, $F_1 = N_G(\{s\})$.
- The 2nd level to visit is all vertices that are reachable within exactly 2 steps from s, i.e., all the neighbors of F_1 except things already in F_1 and s. I.e., $F_2 = N_G(F_1)\backslash(F_1 \cup \{s\})$.
- ...
- The i-th level to visit is all vertices that are reachable within exactly i steps from s. That is, all the neighbors of F_{i-1} except those that we have encountered previously. Hence,

$$F_i = N_G(F_{i-1})\backslash X_{i-1} \quad \text{where} \quad X_{i-1} = \{s\} \cup F_1 \cup F_2 \cup \cdots \cup F_{i-1}.$$

  We call X_i the set of all encountered vertices at the start of step i.

Let's look at an example to see how this works in action. Consider the following graph, where we start searching from F.



Level	F_i	X_i
$i = 0$	$\{F\}$	$\{F\}$
$i = 1$	$\{A, B, C\}$	$\{F, A, B, C\}$
$i = 2$	$\{G, D, I, H\}$	$\{F, A, B, C, G, D, I, H\}$
$i = 3$	$\{E\}$	$\{F, A, B, C, G, D, I, H, E\}$

A few relationships can be observed here:

- As we said before, F_i is the set of new vertices that are exactly distance i from the starting point s.
- We call these vertices the *frontier* vertices because they are at the forefront of the search. Indeed, they are contained in the set of all encountered vertices, i.e., $F_i \subseteq X_i$.

Turing this idea into pseudocode (technically this is valid Python), we have:

```
def bfs(G, s):
    # F, X are the frontiers F_i and the encountered vertices X_i for i.
    i = 0
    F = { s }    # create a set with just s
    X = { s }
    while F != emptyset:   (**)
        nextF = neighbors(G, F) - X
        nextX = X + nextF

        X, F, i  = nextX, nextF, i + 1
    return X, i
```

Figure 1 schematically depicts a run of BFS on an undirected graph where s is the central vertex. Initially, X_0 and F_0 are the single source vertex s (which is the only vertex that is a distance 0 from s). It should be noted that this is the only time $F_i = X_i$ in general. X_1 is all the vertices that have distance at most 1 from s, and F_1 contains those vertices that are on the inner concentric ring, a distance exactly 1 from s. The outer concentric ring contains vertices in F_2, which are a distance 2 from s. The neighbors $N_G(F_1)$ are the central vertex and those in F_2. Notice that some vertices in F_1 share the same neighbors, which is why $N_G(F)$ is defined as the *union* of neighbors of the vertices in F to avoid duplicate vertices.

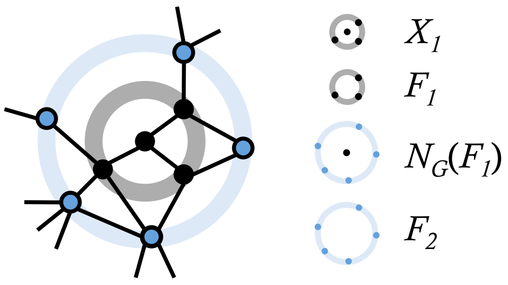


Figure 1: BFS on an undirected graph with the source vertex at the center

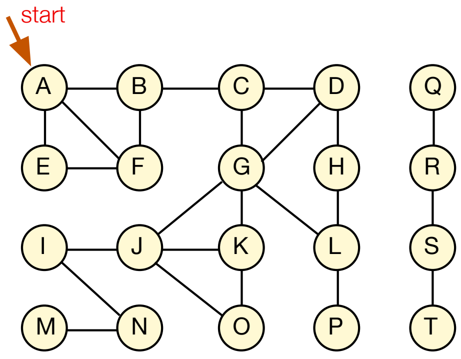
We can prove the following lemma that relates i to what's in X and F using induction:

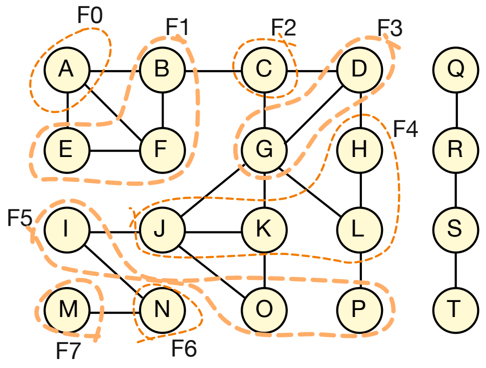
Lemma 2.1 *In algorithm BFS, on line (**) where the while condition is tested, we have* $X = \{v \in V_G | \delta_G(s, v) \leqslant i\}$ *and* $F = \{v \in V_G | \delta_G(s, v) = i\}$.

Proof: This can be proved by induction on the step i. For the base case (the initial call) we have $X = F = \{s\}$ and $i = 0$. This is true since only s has distance 0 from s. For the inductive step we note that, if all vertices F at step i have distance i from s, then a neighbor of F must have minimum path of length $d \leqslant i + 1$ from s—since we are adding just one more edge to the path. However, if a neighbor of F has a path $d < i + 1$ then it must be in X, by the inductive hypothesis so it is not added to `nextF`. Therefore F on $i + 1$ will contain vertices with distance exactly $d = i + 1$ from s. Furthermore since the neighbors of F are unioned with X, X at step $i + 1$ will contain exactly the vertices with distance $d \leqslant i + 1$. ■

Several things follow from this lemma: First, the algorithm returns all reachable vertices $R_G(s)$. We note that if a vertex v is reachable from s, then it must have distance $d = \delta(s, v) < \infty$, meaning it must be discovered at some point by BFS, in particular in F_d. Furthermore, for any vertex v, $\delta(s, v) < |V|$, so the algorithm will terminate in at most $|V|$ steps.

Let's try another example:





2.2 Java Implementation

Converting the idea above into Java is pretty straightforward:

```java
import java.util.*;

public class BasicBFS {

    static Set<Integer> nbrs(UndirectedGraph G, Set<Integer> vtxes) {
        Set<Integer> union = new HashSet<>();
        for (Integer src : vtxes) {
            for (Integer dst : G.adj(src))
                union.add(dst);
        }
        return union;
    }
    static Set<Integer> bfs(UndirectedGraph G, int s) {
        // What can be reached 0 hops away? only s itself
        Set<Integer> frontier = new HashSet<>(Arrays.asList(s));
        Set<Integer> visited = new HashSet<>(Arrays.asList(s));

        while (!frontier.isEmpty()) {
            frontier = nbrs(G, frontier);
            frontier.removeAll(visited);  // nbrs(frontier) - visited
            visited.addAll(frontier);  // the i-th position is what's reached at i hops
        }

        return visited;
    }
}
```

3 BFS Extensions

So far we have specified a routine that returns a list of vertex sets reachable at different distances. Often, we would like to know more, such as the distance of each vertex from s, or the shortest path from s to some vertex v. It is easy to extend BFS for these purposes.

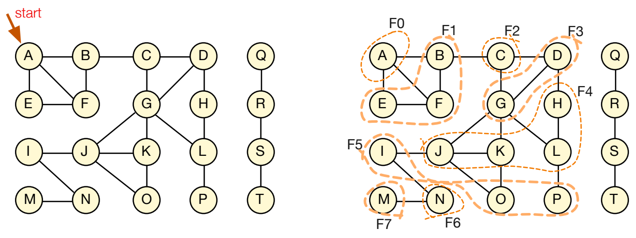
The main idea is when we compute the neighbors of the current frontier vertices, we'll remember which vertex we use to get there. Hence, the neighbor function will return a map return instead of a set, like so:

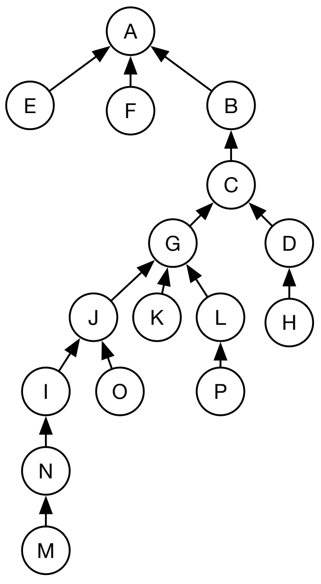
```
static Map<Integer, Integer> nbrs(UndirectedGraph G, Set<Integer> vtxes) {
  Map<Integer, Integer> union = new HashMap<>();
  for (Integer src : vtxes) {
    for (Integer dst : G.adj(src))
      union.put(dst, src);
  }
  return union;
}
```

While the structure of the bfs code will largely remain the same, it will have to be updated to keep track of parent information as the following code shows:

```
static Map<Integer, Integer> bfs(UndirectedGraph G, int s) {
    // What can be reached 0 hops away? only s itself
    Map<Integer, Integer> frontier = new HashMap<>();
    Map<Integer, Integer> visited = new HashMap<>();

    frontier.put(s, null); visited.put(s, null);

    while (!frontier.isEmpty()) {
        frontier = nbrs(G, frontier.keySet());
        // remove all that have been visited
        frontier.keySet().removeAll(visited.keySet());
        visited.putAll(frontier);
    }

    return visited;
}
```

By doing so, we're generating a parent-child relation between all the nodes reachable from the source. This parent-child relation can be seen as a tree, known as a *BFS tree*. For example, running the extended BFS code on the "mesh" graph below gives us the tree on the right.





Exercise: How to report an actual shortest path from the source to a given vertex?

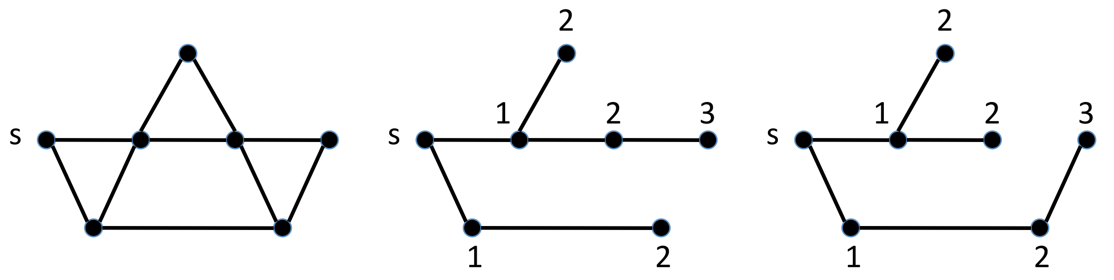


Figure 2: An undirected graph and two possible BFS trees with distances from s

3.1 Running Time

The cost of `BFS` is dominated by the cost of `nbrs`(G, F) over all frontiers. A careful analysis of the cost of `BFS` is a bit tricky because the cost of `nbrs`(G, F) depends on the graph structure. For a $G = (V, E)$ one needs to keep track of the size of the frontiers and the neighbor sets and then realize that the aggregate sizes are function of $n = |V|$ and $m = |E|$. We start by analyzing the cost of `nbrs`(G, F) for some frontier F, and then consider the total cost over all frontiers.

Using our adjacency table representation, we know that `G.adj`(v) returns, in $O(1)$ time, the neighbors of v (as an iterable). By examining the code of `nbrs(G, F)`, we know that calls put for each neighbor v of $u \in F$. This means each $u \in F$ calls `put` a total of $\deg(u)$ times. Since each `put` on a `HashMap` runs in $O(1)$ time, this means the total cost is $\sum_{v \in F} \deg(u)$.

Using the definition of F_i and X_i from the previous lecture, we have that for every i, the cost is $O(\sum_{v \in F_{i-1}} \deg(v))$. Hence, if BFS terminates when $i = d$ (i.e., $F_i = \emptyset$), we will have that the total running time is big O of

$$\sum_{i=1}^{d} \sum_{v \in F_{i-1}} \deg(v).$$

To continue, we make an observation that if a vertex w is encountered in BFS, w belongs to a unique F_i. This means that the F_i's are disjoint. Therefore,

$$\sum_{i=1}^{d} \sum_{v \in F_{i-1}} \deg(v) \leqslant \sum_{v \in V} \deg(v) = 2m = O(m).$$

Therefore, BFS has running time $O(m)$.