Lecture 10 — BFS

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

Lectured by Umut Acar — February 14, 2013

Material in this lecture:
- Breadth First Search
- Unweighted Shortest Paths

1 Breadth First Search

The first graph search approach we consider is breadth first search (BFS). 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, partitioning graphs, and as a subroutine for finding the maximum flow in a flow network (using Ford-Fulkerson's algorithm). BFS, as with the other graph searches, can be applied to both directed and undirected graphs. In the directed case we only consider the outgoing arcs when searching.

The idea of *breadth first search* is to start at a *source* vertex s and explore the graph outward in all directions level by level, first visiting all vertices that are the (out-)neighbors of s (i.e. have distance 1 from s), then vertices that have distance two from s, then distance three, etc. It should be clear that a vertex at distance $i+1$ must have an (in-)neighbor from a vertex a distance i. Therefore, if we know all the vertices at distance i, then we can find the vertices at distance $i+1$ by just considering their (out-)neighbors.

As with all the search approaches, BFS needs to keep track of which vertices have already been visited so that it does not visit them more than once. We will refer to all the visited vertices at the start of level i as X_i. Since on level i we visit vertices at a distance i away, the vertices in X_i are exactly those with distance less than i from the source. On each level the search also maintains a frontier. At the start of level i the frontier F_i contains all unvisited neighbors of X_i, which is all vertices in the graph with distance exactly i from s.

In BFS on each level we visit all vertices in the frontier. This differs from DFS which only visits one. What we do when we visit depends on the particular application of BFS. But for now we assume we simply mark the vertices as visited. We mark newly visited vertices by simply adding the frontier to the previously visited vertices, *i.e*, $X_{i+1} = X_i \cup F_i$. To generate the next set of frontier vertices, the search takes the neighborhood of F and removes any vertices that have already been visited, *i.e.*, $F_{i+1} = N_G(F) \setminus X_{i+1}$. Recall that for a vertex v, $N_G(v)$ are the neighbors of v in the graph G (the out-neighbors for a directed graph) and for a set of vertices F, that $N_G(F) = \bigcup_{v \in F} N_G(v)$.

Below is pseudocode for a BFS algorithm just described. It returns the set of vertices reachable from a vertex s as well as the shortest distance to the furthest reachable vertex. We define $\delta_G(s,u)$ to

[†] Lecture notes by Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan.

be the shortest distance from vertex s to vertex u, and $R_G(s)$ the set of vertices reachable from s in the graph G.

```
function BFS($G = (V, E)$,  $s$) =
let
  % requires:  $X = \{u \in V \mid \delta_G(s,u) < i\} \wedge$
                $F = \{u \in V \mid \delta_G(s,u) = i\}$
  % returns:  $(R_G(s),\ \max\{\delta_G(s,u) : u \in R_G(s)\})$
  function BFS'($X$,  $F$,  $i$) =
    if $|F| = 0$ then $(X, i)$
    else let
      $X' = X \cup F$            % Visit the Frontier
      $N = N_G(F)$               % Determine the neighbors of the frontier
      $F' = N \setminus X'$      % Remove vertices that have been visited
    in BFS'($X'$,  $F'$,  $i+1$) % Next level
    end
in BFS'({},  {$s$},  0)
end
```

If we are using a adjacency table representation of the graph, we can find $N_G(F)$, all the neighbors of the frontier F, as

```
function $N_G(F)$ = Table.reduce Set.Union {} (Table.extract($G, F$))
```

The full SML code for the algorithm is given in the appendix at the end of these notes.

Figure 1 illustrates BFS on an undirected graph where s is the central vertex. Initially, X_0 is empty and F_0 is the single source vertex s, as it is the only vertex that is a distance 0 from s. X_1 is all the vertices that have distance less than 1 from s (just 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. For the graph in the figure, which vertices are in X_2?

Exercise 1. *In general, from which frontiers could the vertices in $N_G(F_i)$ come when the graph is undirected? What if the graph is directed?*

To prove that the algorithm is correct we need to prove the assumptions that are stated in the algorithm. In particular:

Lemma 1.1. *In algorithm BFS, when calling BFS'(X, F, i), we have $X = \{v \in V_G \mid \delta_G(s,v) < i\} \wedge F = \{v \in V_G \mid \delta_G(s,v) = i\}$*

Proof. This can be proved by induction on the level i. For the base case (the initial call) we have $X_0 = \{\}$, $F_0 = \{s\}$ and $i = 0$. This is true since no vertex has distance less than 0 from s and only s

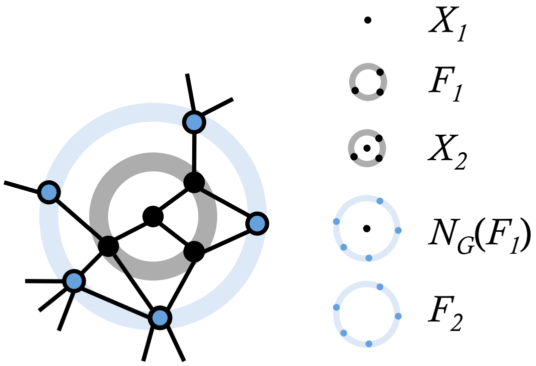

Figure 1: BFS on an undirected graph with the source vertex at the center

has distance 0 from s. For the inductive step we assume the claims are correct for i and want to show it for $i+1$. For X_{i+1} we are simply taking the union of all vertices at distance less than i (X_i) and all vertices at distance exactly i (F_i). So this union must include exactly the vertices a distance less than $i+1$. For F_{i+1} we are taking all neighbors of F_i and removing the X_{i+1}. Since all vertices F_i have distance i from s, by assumption, then a neighbor v of F must have $\delta_G(s,v)$ of no more than $i+1$. Furthermore, all vertices of distance $i+1$ must be reachable from a vertex at distance i. Therefore, the neighbors of F_i contain all vertices of distance $i+1$ and only vertices of distance at most $i+1$. When removing X_{i+1} we are left with all vertices of distance $i+1$, as needed. □

To argue that the algorithm returns all reachable vertices, we note that if a vertex v is reachable from s and has distance $d = \delta(s,v)$ then there must be another vertex u with distance $\delta(s,u) = d-1$. Therefore, BFS will not terminate without finding v. Furthermore, for any reachable vertex v, $\delta(s,v) < |V|$ so the algorithm will terminate in at most $|V|$ rounds (levels).

1.1 BFS extensions

So far we have specified an algorithm that returns the set of vertices reachable from s and the longest length of all shortest paths to these vertices. 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, i.e., the actual sequence of vertices in the path. It is easy to extend BFS for these purposes. For example the following algorithm returns a table mapping every reachable vertex v to $\delta_G(s,v)$.

```
function BFS$(G,s)$ = let
  function BFS$'(X, F, i)$ =
    if $|F| = 0$ then $X$
    else let
        $X' = X \cup \{v \mapsto i : v \in F\}$
        $F' = N_G(F) \setminus \mathtt{domain}(X')$
    in BFS$'(X', F', i+1)$ end
in BFS$'(\{\}, \{s\}, 0)$ end
```

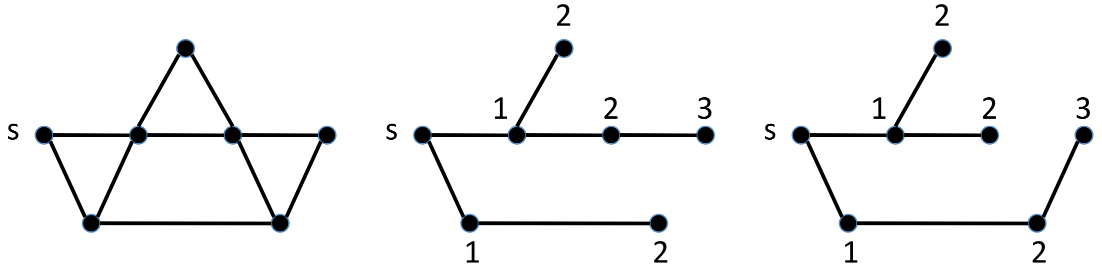

Figure 2: An undirected graph and two possible BFS trees with distances from s

To report the actual shortest paths one can generate a shortest path tree, which can be represented as a table mapping each reachable vertex to its parent in the tree. Then one can report the shortest path to a particular vertex by following from that vertex up the tree to the root (see Figure 2). We note that to generate the pointers to parents requires that we not only find the next frontier $F' = N_G(F)/X'$ of the frontier on each level, but that we identify for each vertex $v \in F'$ one vertex $u \in F$ such that $(u, v) \in E$. There could be multiple such edges to the vertex v. Indeed, Figure 2 shows two possible trees that differ in what the parent is for the vertex at distance 3.

There are various ways to identify the parent. One is to post-process the result of the BFS that returns the distance. In particular, for every vertex we can pick one of its (in-)neighbors with a distance one less than itself.

Another way is to identify the parent when generating the neighbors of F. The ideas is that both the visited vertices X and the frontier F are tables that map each of its vertices to a parent vertex. Finding the next visited table is easy since it just merges two tables X and F. Finding the next frontier requires tagging each neighbor from where it came and then merging the results. That is, for each $v \in F$, it generates a table $\{u \mapsto v : u \in N(v)\}$ that maps each neighbor of v back to v. When merging these tables, it has decide how to break ties since vertices in the frontier might have the several (parent) neighbors. The code takes the first vertex.

1.2 BFS Cost

So far in the class we have mostly calculated costs using recurrences. This works well for divide-and-conquer algorithms, but, as we will see, most graph algorithms do not use divide-and-conquer. Instead, for many graph algorithms we can calculate costs by counting, i.e., adding the costs across a sequence of rounds of an algorithm. Different rounds can take a different amount of work or span.

Since BFS works in a sequence of rounds, one per level, we can add up the work and span across the levels. The problem, however, is that the work done at each level varies, since it depends on the size of the frontier at that level—in fact it depends on the number of outgoing edges from the frontier for that level. What we do know, however, is that every reachable vertex only appears in the frontier exactly once. Therefore, all their out-edges are processed exactly once only. If we can calculate the cost per edge W_e and per vertex W_v regardless of the level, then we can simply multiply these by the number of edges and vertices giving $W = W_v n + W_e m$ (recall that $n = |V|$ and $m = |E|$).

For the span we can determine the largest span per level S_l and multiply it by the number of levels $d = \max_{v \in V} \delta(s, v)$, giving $S = S_l d$,.

If we use the tree representation of sets and tables, we can show that the work per edge and per vertex is bounded by $O(\log n)$ and the span per level is bounded by $O(\log^2 n)$. Therefore we have:

$$\begin{aligned} W_{BFS}(n,m,d) &= O(n \log n + m \log n) \\ &= O(m \log n) \\ S_{BFS}(n,m,d) &= O(d \log^2 n) \end{aligned}$$

We drop the $n \log n$ term in the work since for BFS we cannot reach any more vertices than there are edges.

Now let's show that the work per vertex and edge is $O(\log n)$. We can examine the code and consider what is done on each level. In particular the only non-trivial work done on each level is the union $X' = X \cup F$, the calculation of neighbors $N = N_G(F)$ and the set difference $F' = N \setminus F$. The cost of these will depend on the size of the frontier, and in fact in the number of out-edges from the frontier. We will use $||F||$ to denote the number of out-edges for a frontier plus the size of the frontier, i.e., $||F|| = \sum_{v \in F}(1 + d_G^+(v))$. The costs for each level are as follows

	Work	Span
$X \cup F$	$O(\lvert F\rvert \log n)$	$O(\log n)$
$N_G(F)$	$O(\lvert\lvert F\rvert\rvert \log n)$	$O(\log^2 n)$
$N \setminus F$	$O(\lvert\lvert F\rvert\rvert \log n)$	$O(\log n)$

The first and last lines fall directly out of the cost spec for the set interface. The second line is a bit more involved. Recall that it is implemented as

function $N_G(F) =$ `Table.reduce Set.Union {} (Table.extract(`G, F`))`

Let $G_F =$ `Table.extract`(G, F). The work to find G_F is bounded by $O(|F| \log n)$. For the cost of the union we can use Lemma 2.1 from lecture 6. In particular, union satisfies the conditions of the Lemma. Therefore, the work is bound by

$$W(\texttt{reduce union } \{\}\ G_F) = O\left(\log |G_F| \sum_{v \mapsto N(v) \in G_F} (1 + |N(v)|)\right) = O\left(\log n \cdot ||F||\right)$$

and span is bounded by

$$S(\texttt{reduce union } \{\}\ G_T) = O(\log^2 n)$$

since each union has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

Now we see that work per vertex and edge is $O(\log n)$, since each vertex and its out-edges appear on only one frontier and on level i we process $||F_i||$ vertices and their out-edges.

Notice that span depends on d. In the worst case $d \in O(n)$ and BFS is sequential. As we mentioned before, many real-world graphs are shallow, and BFS for these graphs has good parallelism.

2 BFS with Single Threaded Sequences

Here we consider a version of BFS that uses sequences instead of sets and tables. The advantage is that it runs in $O(m)$ total work and $O(d \log n)$ span, a factor of $O(logn)$ less in both work and span.

We refer to a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ as an *integer labeled* (IL) graph. For an IL graph we can use the sequences to represent a graph. Each vertex identifier is an integer and the data for the vertex is stored at that index in the sequence. In this way, if the sequence is array-based, looking up a vertex is only constant work. At each index we store the neighbors of the vertex, where the neighbors can be represented as an array sequence containing the neighbors' integer identifiers. An IL graph can therefore be implemented with type:

`(int seq) seq.`

Because the graph does not change during BFS, this representation is efficient.

The set of visited vertices X, however, does change during the course of the algorithm. Therefore, we use a single threaded (ST) sequence of length $|V|$ to mark which vertices have been visited. By using inject, we can mark vertices in constant work per update. We can use either a boolean to indicated whether a vertex has been visited on not, or an `(int option)` if we want to map each vertex to its parent. In this case the option `NONE` indicates the vertex has not been visited, and `SOME`(v) indicates it has been visited and its parent is v. Each time we visit a vertex, we map it to its parent in the BFS tree. As the updates to this sequence are potentially small compared to its length, using an `stseq` is efficient. On the other hand, because the set of frontier vertices is new at each level, we can represent the frontier simply as an integer sequence containing all the vertices in the frontier, allowing for duplicates.

To simplify the algorithm we change the invariant a bit. In particular on entering `BFS'` the sequence `XF` contains parent pointers for both the visited and the frontier vertices instead of just for the visited vertices. F is an integer sequence containing the frontier.

```
1   function BFS($G$ : (int seq) seq, $s$ : int) =
2   let
3     function BFS$'$(XF : int option stseq, $F$ : int seq) =
4       if $|F| = 0$ then stSeq.toSeq XF
5       else let
6           $N$ = flatten $\langle \langle (u, \texttt{SOME}(v)) : u \in G[v] \rangle : v \in F \rangle$     % neighbors of the frontier
7           XF$'$ = stSeq.inject($F$, XF)                                  % new parents added
8           $F' = \langle u : (u, v) \in N \mid \texttt{XF}'[u] = v \rangle$                     % remove duplicates
9       in BFS$'$(XF$'$, $F'$) end
10     $X_0$ = stSeq.toSTSeq($\langle \texttt{NONE} : v \in \langle 0, \ldots, |G| - 1 \rangle \rangle$)
11  in
12    BFS$'$(stSeq.update($s$, SOME($s$), $X_0$), $\langle s \rangle$)
13  end
```

All the work is done in lines 6, 7, and 8. Also note that the `stSeq.inject` on line 7 is always applied to the most recent version. We can write out the following table of costs:

	XF : stseq		XF : seq	
line	work	span	work	span
6	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
7	$O(\|\|F_i\|\|)$	$O(1)$	$O(n)$	$O(1)$
8	$O(\|\|F_i\|\|)$	$O(\log n)$	$O(\|\|F_i\|\|)$	$O(\log n)$
total across all d rounds	$O(m)$	$O(d \log n)$	$O(m + nd)$	$O(d \log n)$

where d is the number of rounds (i.e. the shortest path length from s to the reachable vertex furthest from s). The last two columns indicate the costs if XF was implemented as a regular array sequence instead of an stSeq. The big difference is the cost of `inject`. As before the total work across all rounds is calculated by noting that every out-edge is only processed in one frontier, so $\sum_{i=0}^{d} ||F_i|| = m$.

3 SML Code

Basic BFS. The following SML code for BFS mirrors the pseudo-code in the notes. It uses a table that maps each vertex to a set that contains its (out-)neighbors. The function **function** N G F implements $N_G(F)$ by first using `extract` to get a table with only the vertices in F. That is, the resulting table maps each vertex in F to its neighbors. Next, it combines all the neighbors of F into a single set. Recall that `Table.reduce` f combines the values in the table with the function f.

```
functor TableBFS(Table : TABLE) =
struct
  structure Set = Table.Set
  type vertex = Table.key
  type graph = Set.set Table.table

  fun N (G : graph) (F : Set.set) =
    Table.reduce Set.union Set.empty (Table.extract (G, F))

  fun BFS_reachable (G : graph, s : vertex) =
  let
    (* Require: X = {v in V_G | delta_G(s,v) < i} and
     *          F = {v in V_G | delta_G(s,v) = i}
     * Return: (R_G(s), max {delta_G(s,v) : v in R_G(s)}) *)
    fun BFS’ (X : Set.set, F : Set.set, i : int) =
      if  (Set.size F = 0) then (X, i)
      else let
        val X’ = Set.union (X, F)
        val F’ = Set.difference (N G F, X’)
      in
         BFS’(X’, F’, i+1)
      end
  in
    BFS’(Set.empty, Set.singleton s, 0)
  end
end
```