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 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 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, the 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 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. This is done by simply adding the frontier to the visited vertices, *i.e*, $X_{i+1} = X_i \cup F_i$. To generate the next set of frontier vertices the search simply 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 $N_G(F) = \cup_{v \in F} N_G(v)$.

Here 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.

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

```
1  function BFS(G = (V, E), s) =
2  let
3    % requires: X = {u ∈ V | δ_G(s,u) < i} ∧
4                 F = {u ∈ V | δ_G(s,u) = i}
5    % returns: (R_G(s), max{δ_G(s,u) : u ∈ R_G(s)})
6    function BFS'(X, F, i) =
7      if |F| = 0 then (X, i)
8      else let
9          X' = X ∪ F          % Visit the Frontier
10         N = N_G(F)          % Determine the neighbors of the frontier
11         F' = N \ X'         % Remove vertices that have been visited
12     in BFS'(X', F', i + 1) % Next level
13     end
14 in BFS'({}, {s}, 0)
15 end
```

Recall that $N_G(F) = \bigcup_{v \in F} N(v)$, where $N(v)$ are the neighbors of v. If we are using a adjacency table representation of the graph, this can be calculated 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. 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? For the graph in the figure, which vertices are in X_2?

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 only s has distance 0 from s and no vertex has distance less than 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 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

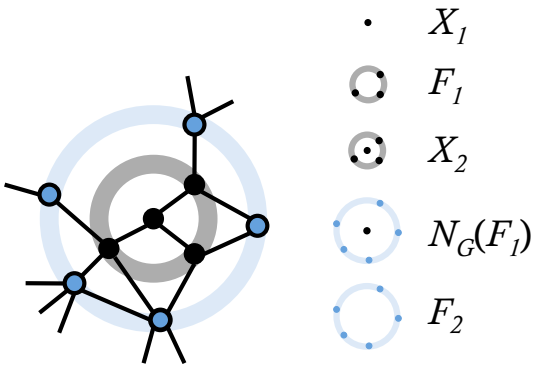

Figure 1: BFS on an undirected graph with the source vertex at the center

all vertices of distance no more than $i+1$ 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, 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 u vertex with distance $\delta(s,u) = d-1$. Therefore BFS will not terminate without finding it. Furthermore, for any 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 a routine 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 $\delta_G(s,v)$.

```
1  function BFS(G,s) = let
2    function BFS'(X, F, i) =
3      if |F| = 0 then X
4      else let
5          X' = X ∪ {v ↦ i : v ∈ F}
6          F' = N_G(F) \ domain(X')
7      in BFS'(X', F', i+1) end
8  in BFS'({}, {s}, 0) end
```

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

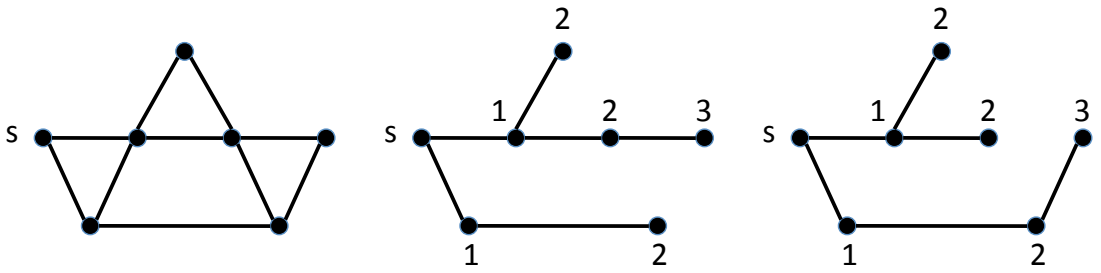

Figure 2: An undirected graph and two possible BFS trees with distances from s

$F' = N(F)/X'$ 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. In the appendix we include sample SML code that generates the parents in this way. The ideas is that both the visited vertices X and the frontier F are table that maps the given vertex sets to their parent. Finding the next visited set is easy since it is just a union (or a merge on tables). Finding the next frontier requires tagging each neighbor with where it came from and then merging the results. This merge has decide how to break ties since many vertices in the frontier might have the same neighbor. In the code the first vertex is taken. See the code if you are interested, or need something like this for your homework.

To find $N_G(F)$, the code in the appendix takes the union of the sets of neighbors (one set for each $v \in F$). This time, for each $v \in F$, we generate a table $\{u \mapsto v : u \in N(v)\}$ that maps each neighbor of v back to v. Then instead of union, we use merge to combine all these neighbor tables. In terms of our ML library this would look something like

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., simply adding up the costs across a sequence of rounds of an algorithm.

Since BFS works in a sequence of rounds, one per level, we can add up the work and span across the levels. However, the work done on each level varies since it depends on the size of the frontier on 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 also are only processed exactly once. If we can calculate the cost per edge W_e and per vertex W_v for processing a frontier, 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 span per level S_l and multiply it by the number of levels $S = S_l d$, where $d = \max_{v \in V} \delta(s, v)$ is the number of levels.

If we use the tree representations 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 lets 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 outedges 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
$X \cup F$	$O(\|F\| \log n)$	$O(\log n)$
$N_G(F)$	$O(\|\|F\|\| \log n)$	$O(\log^2 n)$
$N \setminus F$	$O(\|\|F\|\| \log n)$	$O(\log n)$

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`))`

The work for extract is bounded by $O(|F| \log n)$. For the cost of the union we can use Lemma 2.1 from lecture 6. In particular merge satisfies the conditions of the Lemma, therefore the work is bound by

$$W(\texttt{reduce union \{\}}\ F_{ngh}) = O\left(\log |F_{ngh}| \sum_{\texttt{ngh} \in F_{ngh}} (1 + |\texttt{ngh}|)\right) = O\left(\log n \cdot ||F||\right)$$

where $F_{ngh} =$ `Table.extract(`G, F`)`, and span is bounded by

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

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

Now we see that work per edge is $O(\log n)$ since on level i we process $||F_i||$ edges and every out edge is only processed once.

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 to represent the graph and single threaded (ST) sequences to mark the visited vertices. The advantage is that it runs in $O(m)$ total work and $O(d \log n)$ 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, giving constant work access to the values stored at vertices (if using array sequences). The neighbors of each vertex can also be represented as an array sequence containing the integer indices of their neighbors. An IL graph can therefore be implemented with type:

```
(int seq) seq.
```

We consider the version of BFS that returns a mapping from each vertex to its parent in the BFS tree. We maintain the set of visited vertices using an `(int option) stseq`. This allows us to update which vertices have been visited in constant work per update. In this sequence the option `NONE` indicates the vertex has not been visited, and `SOME`(v) indicates it has been and its parent is v. Each time we visit a vertex, we map it to its parent in the BFS tree; the value v in `SOME`(v) is its parent vertex. 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.

```
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 P
5      else let
6          N = flatten ⟨ ⟨(u, SOME(v)) : u ∈ G[v] ⟩ : v ∈ F ⟩   % neighbors of the frontier
7          XF' = stSeq.inject(F, XF)                            % new parents added
8          F' = ⟨ u : (u, v) ∈ N | XF'[u] = v ⟩                 % remove duplicates
9      in BFS'(XF', F') end
10    X_0 = stSeq.toSTSeq(⟨ NONE : v ∈ ⟨0, ..., |G| − 1⟩ ⟩)
11 in
12   BFS'(stSeq.update(s, SOME(s), X_0), ⟨s⟩)
13 end
```

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.

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 longest path length from s to any other reachable vertex). 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
```

Generating a BFS Tree. The following code generates a BFS tree. It represents the visited set X and the frontier F as table that map each vertex in the visited set or frontier to their parent in the BFS tree (i.e. who visited them). The function `outEdges` returns the out edges of v or an empty set if v is not found. The $N_G(F)$ function not only returns the neighbors for every vertex $v \in F$, but also tags each neighbor with the vertex v they came from. In particular `tagNeighbors` tags all neighbors with v, and then a `Table.reduce` is used to merge all the tables of tagged neighbors. The `merge` used in the reduce takes the first argument if there are two equal keys, which happens in BFS when a vertex has multiple potential parents.