Lecture 8 — Graph Search and BFS

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2012)

Lectured by Margaret Reid-Miller — 9 February 2012

Today:

- Graph Search
- Breadth First Search

1 Graph Search

One of the most fundamental tasks on graphs is searching a graph by starting at some vertex, or set of vertices, and visiting new vertices by crossing (out) edges until there is nothing left to search. In such a search we need to be systematic to make sure that we visit all vertices that we can reach and that we do not visit vertices multiple times. This will require recording what vertices we have already visited so we don't visit them again. Graph searching can be use to determine various properties of graphs, such as whether the graph is connected or whether it is bipartite, as well as various properties relating vertices, such as whether a vertex u is reachable from v, or finding the shortest path between vertices u and v. In the following discussion we use the notation $R_G(u)$ to 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).

There are three standard methods for searching graphs: breadth first search (BFS), depth first search (DFS), and priority first search. All these methods visit every vertex that is reachable from a source, but the order in which they visit the vertices can differ. Search methods often divide the vertices into tree sets: vertices already visited, vertices encountered but yet visited (frontier), and vertices not yet encountered. Vertices are visited only once, but may be encountered multiple times.

All search methods when starting on a single source vertex generate a rooted *search tree*, either implicitly or explicitly. This tree is a subset of the edges from the original graph. In particular a search always visits a vertex v by entering from one of its neighbors u via an edge (u, v). This visit to v adds the edge (u, v) to the tree. These edges form a tree (i.e., have no cycles) since no vertex is visited twice and hence there will never be an edge that wraps around and visits a vertex that has already been visited. We refer to the source vertex as the *root* of the tree. Figure 1 gives an example of a graph along with two possible search trees. The first tree happens to correspond to a BFS and the second to a DFS.

Graph searching has played a very important role in the design of sequential algorithms, but the approach can be problematic when trying to achieve good parallelism. Depth first search (DFS) has a wealth of applications, but it is inherently sequential. Because of this, one often uses other techniques in designing good parallel algorithms. We will cover some of these techniques in upcoming lectures. Breadth first search (BFS), on the other hand, can be parallelized effectively as long as the graph is shallow (the longest shortest path from the source to any vertex is reasonably small). In fact, the depth of the graph will show up in the bounds for span. Fortunately many real-world graphs

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

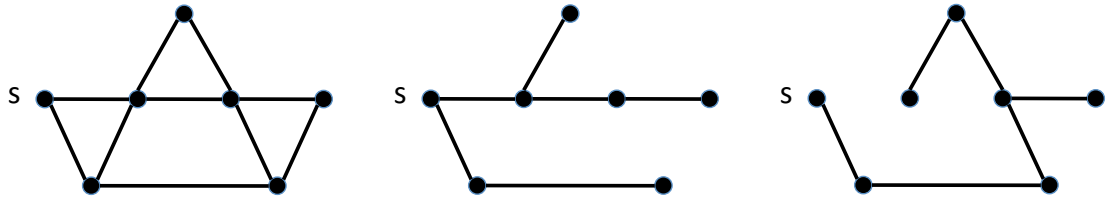

Figure 1: An undirected graph and two possible search trees.

are shallow. But if we are concerned with worst-case behavior over any graph, then BFS is also sequential.

2 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 graph. In the following discussion when we write "(out-)neighbors of v" we mean, in the undirected case, simply the neighbors of v, and in the directed case, the neighbors that can be reached from out-going edges of v. Similar is the meaning of (in-)neighbor. 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.

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 encountered so that it does not visit them more than once. Let's call the set of all encountered vertices at the start of step i, X_i. On each step the search also needs to keep the set of new vertices that are exactly distance i from s. We refer to these as the *frontier* vertices $F_i \subseteq X_i$. To generate the next set of frontier vertices the search simply takes the neighborhood of F and removes any vertices that have already been encountered, *i.e.*, $N_G(F) \setminus X$. 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 that returns the set of vertices reachable from a vertex s as well as the shortest distance to the furthest reachable vertex.

```
1  fun BFS(G,s) =
2  let
3     % requires: X = {u ∈ V_G | δ_G(s,u) ≤ i} ∧ F = {u ∈ V_G | δ_G(s,u) = i}
4     % returns: (R_G(v), max{δ_G(s,u) : u ∈ R_G(v)})
5     fun BFS'(X,F,i) =
6           if |F| = 0 then (X,i)
7           else BFS'(X ∪ N_G(F), N_G(F) \ X, i+1)
8  in BFS'({s},{s},0)
9  end
```

Figure 2 illustrates BFS on an undirected graph where s is the central vertex. Initially, X_0 and F_0 are the single source vertex s, as it is the only vertex that is a distance 0 from s. It is the only time $F = X$. 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. 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?

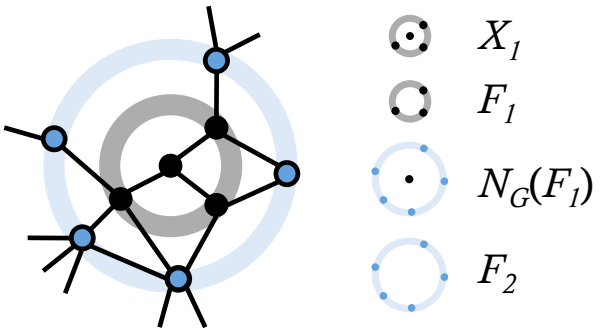

Figure 2: BFS on an undirected graph with the source vertex at the center

The SML code for the algorithm is given in the appendix at the end of these notes. It uses a table that maps each vertex to a set that contains its (out-)neighbors. The function `N(G,F)` first uses `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 applies the function f to the table's values. Therefore, the result of `reduce` is union of all of F's neighbors, thereby removing any duplicate neighbors. The function `BFS'(G,F)` simply mirrors the pseudo-code above.

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

Lemma 2.1. *In algorithm BFS when calling BFS'(X,F,i), we have* $X = \{v \in V_G \mid \delta_G(s,v) \leq i\} \wedge F = \{v \in V_G \mid \delta_G(s,v) = i\}$

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 $d \leq i + 1$ 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 F'. Therefore F on step $i + 1$ 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 $d \leq i + 1$. □

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 $|V|$ steps.

2.1 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. It is easy to extend BFS for these purposes. For example the following algorithm returns a table mapping every reachable vertex to its shortest path from s.

```
1  fun BFS(G, s) =
2  let
3    fun BFS'(X, F, i) =
4      if |F| = 0 then X
5      else let
6        val F' = N_G(F) \ {v : (v ↦ _) ∈ X}
7        val X' = X ∪ {v ↦ (i + 1) : v ∈ F'}
8      in BFS'(X', F', i + 1)end
9  in BFS'({s ↦ 0}, {s}, 0)
10 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 3). We note that to generate the pointers to parents requires that we not only find the next frontier $F' = N(F)/X$ 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 3 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. 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

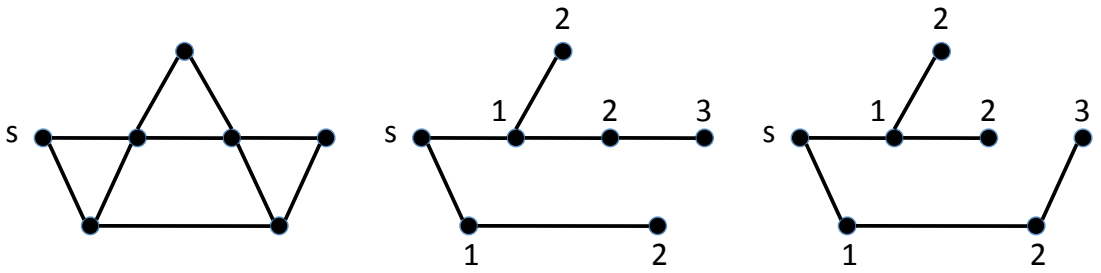

Figure 3: An undirected graph and two possible BFS trees with distances from s

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  fun N(G : 'a graph, F : set) =
2  let
3    fun N'(v) = Table.map (fn l ⇒ (v, l)) (getNbrs G v)
4    val nghs = Table.tabulate N' F
5  in
6    Table.reduce merge {} nghs
7  end
```

This code uses a more general adjacency table representation of a graph. Each vertex v maps to a neighbor table that maps each u, a neighbor to v, to a label; for example, the label could be an edge weight. The function `getNbrs` returns v's neighbor table or empty, and then $N'(v)$ augments the labels with the parent v. The result `nghs` is a table mapping each $v \in F$ to its augmented neighbor table. Finally, `reduce` merges all these neighbor tables into one table. Here `merge = (Table.merge (fn (x,y) => x))`, which merges two tables and when a key appears in both tables, takes the value from the first (left) table. Using either value would work. In effect, when a vertex u has several possible parents, the merge selects one of them to be the parent in the final shortest path tree.

2.2 BFS Cost

The cost of BFS is dominated by the cost of $N(G,F)$ over all frontiers. A careful analysis of the cost of BFS is a bit tricky because the cost of $N(G,F)$ 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 $N(G,F)$ for some frontier F, and then consider the total cost of $N(G,F)$ over all frontiers.

The cost `getNbrs` in Line 3 is the cost of (Table.find) which has $O(\log n)$ work and span. The cost of `Table.map` is $O(|N(v)|)$ work and $O(1)$ span. Therefore, function $N'(v)$ in Line 3 has $O(|N(v)|)$ work and $O(\log n)$ span. On Line 4 this function is applied across the whole frontier. It's work is therefore $W(F) = O(\sum_{v \in F} |N(v)|)$ and $S(F) = O(\max_{v \in F} \log n) = O(\log n)$.

Finally to analyze the reduce in Line 6 we use Lemma 2.1 from lecture 5. In particular merge satisfies the conditions of the Lemma, therefore the work is bound by

$$W(\texttt{reduce merge \{\} nghs}) = O\left(\log|\texttt{nghs}| \sum_{ngh \in \texttt{nghs}} |ngh|\right) = O\left(\log|F| \sum_{v \in F} |N(v)|\right)$$

and span is bounded by

$$S(\texttt{reduce merge \{\} nghs}) = O(\log^2 n)$$

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

Next we consider the total cost of $N(G,F)$ across all frontiers. Let's assume that the frontiers are $(F_0, F_1, \ldots, F_d)$ where d is the depth of the shortest path tree. Now we note that every vertex will only appear in one frontier and every (directed) edge will only be counted once. For the overall work done by $N(G,F)$ across the whole algorithm we have:

$$\begin{aligned}
W &= O\left(\sum_{i=[0,\ldots,d]} \log|F_i| \sum_{v \in F_i} |N(v)|\right) \\
&= O\left(\log n \sum_{i=[0,\ldots,d]} \sum_{v \in F_i} |N(v)|\right) \\
&= O\left(\log n \sum_{v \in V} |N(v)|\right) \\
&= O\left((\log n)|E|\right) \\
&= O(m \log n)
\end{aligned}$$

And for span:

$$\begin{aligned}
S &= O\left(\sum_{i=[0,\ldots,d-1]} \log^2 n\right) \\
&= O(d \log^2 n)
\end{aligned}$$

Similar arguments can be used to bound the cost of the rest of BFS across all frontiers.

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.

3 SML Code

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

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

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

end
```