Lecture 8 — Graph Search and BFS

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2011)

Lectured by Guy Blelloch — September 22, 2011

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 u and v. In the following discussion we use the notation $R_G(v)$ to indicate all the vertices that can be *reached* from v in a graph G (i.e., vertices u for which there is a path from v to u 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. 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) 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) 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 are shallow, but if we are concerned with worst-case behavior over any graph, then BFS is also sequential.

```
        o
       / \
S o---o---o---o
   \ /     \ /
    o-------o

        o
       /
S o---o---o---o
   \
    o-------o

        o
       / \
S o   o   o---o
   \       \
    o-------o
```

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

1.1 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 graph. In the following discussion 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 v and explore the graph level by level, first visiting all vertices that are the (out) neighbors of v (i.e. have distance 1 from v), then vertices that have distance two from v, 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.

The BFS approach therefore works as follows. As with all the search approaches, the approach needs to keep track of what vertices have already been visited so that it does not visit them more than once. Let's call the set of all visited vertices at the end 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 v. We refer to these as the *frontier* vertices $F_i \subset X_i$. To generate the next set of frontier vertices the search simply takes the neighborhood of F and removes any vertices that have already been visited, *i.e.*, $N_G(F) \setminus X$. Recall that for a vertex u, $N_G(u)$ are the neighbors of u in the graph G (the out-neighbors for a directed graph) and for a set of vertices U, that $N_G(U) = \cup_{u \in U} N_G(u)$.

Here is pseudocode for a BFS algorithm that returns the set of vertices reachable from a vertex v as well as the furthest distance to any vertex that is reachable.

```
fun $\mathit{BFS}(G, s) =$
let
   % requires: $X = \{u \in V_G \mid \delta_G(s,u) \leq i\} \wedge F = \{u \in V_G \mid \delta_G(s,u) = i\}$
   % returns: $(R_G(v), \max\{\delta_G(s,u) : u \in R_G(v)\})$
   fun $\mathit{BFS}'(X, F, i) =$
         if $|F| = 0$ then $(X, i)$
         else $\mathit{BFS}'(X \cup N_G(F),\ N_G(F) \setminus X,\ i+1)$
in $\mathit{BFS}'(\{s\}, \{s\}, 0)$
end
```

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

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* $X = \{u \in V_G \mid \delta_G(v,u) \leq i\} \wedge F = \{u \in V_G \mid \delta_G(v,u) = i\}$

Proof. This can be proved by induction on the step i. For the base case (the initial call) we have $X = F = \{v\}$ and $i = 0$. This is true since only v has distance 0 from v. For the inductive step we note that, if all vertices F at step i have distance i from v, then a neighbor of F must have minimum path of length $d \leq i+1$ from v—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 v. 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 u is reachable from u and has distance $d = \delta(v,u)$ then there must be another x vertex with distance $\delta(v,x) = d-1$. Therefore BFS will not terminate without finding it. Furthermore for any vertex u $\delta(v,u) < |V|$ so the algorithm will terminate in at most $|V|$ steps.

So far we have specified a routine that returns the set of vertices reachable from v 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 v, or the shortest path from v to some vertex u. 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 v.

```
         o
        / \
v o---o---o---o          Original Graph
   \ /     \ /
    o-------o

         2
         o
     1 /  2   3
v o---o---o---o        BFS tree with distances from v
   \
    o-------o
    1       2

         2
         o
     1 /  2   3
v o---o---o   o        anoher BFS tree with distances from v
   \         /
    o-------o
    1       2
```

Figure 2: An undirected graph and two possible BFS trees.

```
fun $BFS(G, s) =$
let
   fun $BFS'(X, F, i) =$
      if $|F| = 0$ then $X$
      else let
         val $F' = N_G(F) \setminus \{v : (v, \_) \in X\}$
         val $X' = X \cup \{(v, i+1) : v \in F'\}$
      in $BFS'(X', F', i+1)$end
in $BFS'(\{(v, 0)\}, \{v\}, 0)$
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 neighborhood $F' = N(F)/V$ 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 when generating the neighbors of F to instead of taking the union of the neighbors, to instead for each $v \in F$ generate a table $\{(u, v) : u \in N(v)\}$ and merge all these tables. In our ML library this would look something like

```
1  fun $N(G : \texttt{'a graph},\ F : \texttt{set}) =$
2  let
3    fun $N'(v) = \texttt{Table.map}$ (fn $\Rightarrow w \Rightarrow (v, w))$ $(\texttt{getEdges}\ G\ v)$
4    val $\texttt{nghs} = \texttt{Table.tabulate}\ N'\ F$
5  in
6    $\texttt{Table.reduce merge}\ \{\}\ \texttt{nghs}$
7  end
```

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. See the specification of `Table.merge` from Lecture 6.

We can now analyze the cost of $N(G, F)$. For a graph $G = (V, E)$ we assume $n = |V|$ and $m = |E|$. The cost of the function $N'(v)$ in Line 3 is simply $O(|N(v)|)$ work and $O(\log n)$ span. On Line 4 this function is applied across the whole frontier. Its 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

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

For the span we have $S(\texttt{reduce merge}\ \{\}\ \texttt{nghs}) = O(\log^2 n)$ since each merge has span $O(\log n)$ and the reduction tree is bounded by $\log n$ depth.

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

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

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.

2 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(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
```