Lecture 10 — Shortest Weighted Paths I

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

Lectured by Margaret Reid-Miller — 16 February 2012

Today:

- Representing Graphs with Arrays
- Weighted Graphs
- Priority First Search
- Shortest Weighted Paths

1 Representing Graphs With Arrays

When graphs are implemented using tables and sets, the cost of finding and updating a vertex in the table is $(logn)$ work and span. We can improve the asymptotic performance of certain algorithms if these operations are constant work. An array is an obvious data structure that has constant lookup, but in a functional setting the cost of changing a value in the array requires copying the whole area in order to maintain data persistency.

Single-threaded sequences introduce in the previous lecture supported constant-work for each lookup and element update, as long as the operations always occurred on the most recent version of the sequence. Although the costs of these operations can be greater when applied to an earlier version of a sequence, the operations still work correctly. But, as single-threaded sequences use mutation, they appears functional only to a sequential observer and are unsafe for parallel observers. They include, however, a parallel update operation `inject`, so we can still get parallel performance.

To take advantage of array's faster access and update, we can use sequences to represent graphs. But the cost is that this representation is less general, requiring the names of vertices to be restricted to integers in a fixed range. This restriction can become inconvenient in graphs that change dynamically but is typically fine for static graphs.

We refer to a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$ as an *integer labeled* (IL) graph. For such an IL graph, an α `vertexTable` can be represented as a sequence of length n with the values stored at the appropriate indices. In particular, the table

$$\{(0 \mapsto a_0), (1 \mapsto a_1), \cdots, (n-1 \mapsto a_{n-1})\}$$

is equivalent to the sequence

$$\langle a_0, a_1, \cdots, a_{n-1} \rangle ,$$

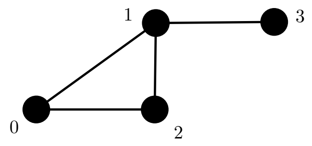
using standard reductions between sequences and sets. If we use an array representation of sequences, then this gives us constant work access to the values stored at vertices. We can also represent the set

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

of neighbors of a vertex as an integer sequence containing the indices of those neighbors. Therefore, instead of using an `set table` to represent a graph we can use a

`(int seq) seq`,

For example, the following undirected graph:



would be represented as

$$G = \langle\langle 1,2\rangle, \langle 0,2,3\rangle, \langle 0,1\rangle, \langle 1\rangle\rangle.$$

Let's consider how this affects the cost of BFS. We consider the version of BFS that returns a mapping from each vertex to its parent in the BFS tree. We represent the IL graph as a `(int seq) seq`. Notice that the graph itself does not use `stseq`, as we do not change the graph. Maintaining the set of encountered vertices, however, does change during the course of the algorithm. Therefore, we will use an `(int option) stseq` to represent these encountered vertices. The option is `NONE` if the vertex has not been encountered, and `SOME`(v) if it has been encountered. Each time we encounter a vertex, we map it to its parent in the BFS tree; the value v in `SOME`(v) is its parent vertex. Since the final result of this sequence is the maps each vertex to its parent in the BFS tree, we refer to it as P instead of X. 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.

Then the algorithm is:

```
1   fun BFS($G$ : (int seq) seq, $s$ : int) =
2   let
3     fun BFS'($P$ : int option stseq, $F$ : int seq) =
4       if $|F| = 0$ then toSeq($P$)
5       else
6         let val $N$ = flatten$\langle\langle (u,v) : u \in G[v]\rangle : v \in F\rangle$      % neighbor edges of frontier
7             val $P'$ = inject($N, P$)                                  % new parents added
8             val $F' = \langle u : (u,v) \in N \wedge P'[u] = v\rangle$         % remove duplicates
9         in BFS'($P', F'$) end
10    val $P_{init}$ = ⟨if ($v = s$) then SOME($s$) else NONE
11                : $v \in \langle 0, \ldots, |G|-1\rangle\rangle$
12  in BFS'(toSTSeq($P_{init}$), $\langle s\rangle$)
13  end
```

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

	P : `stseq`		P : `seq`	
line	work	span	work	span
6	$O(\sum_{v \in F_i} \|N_G(v)\|)$	$O(d \log n)$	$O(\sum_{v \in F} \|N_G(v)\|)$	$O(\log n)$
7	$O(\sum_{v \in F_i} \|N_G(v)\|)$	$O()$	$O(n)$	$O(1)$
8	$O(\sum_{v \in F_i} \|N_G(v)\|)$	$O(\log n)$	$O(\sum_{v \in F} \|N_G(v)\|)$	$O(\log n)$
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). Note that the total across rounds is calculated using the fact that every vertex appears in a frontier at most once so that

$$\sum_{i=0}^{d} \sum_{v \in F_i} |N_G(v)| \le |E| = m.$$

We can do a similar transformation to DFS. Here is our previous version.

```
fun DFS($G$ : set table, $s$ : key) =
let fun DFS'($X$ : set, $v$ : key) =
      if ($v \in X$) then $X$
      else iterate DFS' ($X \cup \{v\}$) ($G_v$)
in DFS'({}, $s$) end
```

And the version using sequences.

```
fun DFS($G$ : (int seq) seq, $s$ : int) =
let
  fun DFS'($X$ : bool stseq, $v$ : int) =
    if ($X[v]$) then $X$
    else iterate DFS' (update($X$, $v$, true)) ($G[v]$)
  val $X_{init}$ = $\langle$false : $v \in \langle 0, \ldots, |G|-1 \rangle\rangle$
in DFS'($X_{init}$, $s$) end
```

If we use an `stseq` for X (as indicated in the code) then this algorithm uses $O(m)$ work and span. However if we use a regular sequence, it requires $O(n^2)$ work and $O(m)$ span.

2 Weighted Graph Representation

There are a number of ways to represent weights in a graph. More generally, we might want to associate any sort of value with the edges. That is, we have a "label" function of type $w : E \to$ `label` where `label` is the type of the label.

The first representation we consider translates directly from viewing edge labels as a function. We keep a table that maps each edge (a pair of vertex identifiers) to its label (or weight). This would have type

(`label` *vertexVertexTable*)

That is, the keys are pairs of vertices (hence *vertexVertexTable*), and the values are labels.

Another way to associate values with edges is to use a structure similar to the adjacency set representation for unweighted graphs and to piggyback labels on top of it. In particular, instead of associating a set of neighbors with each vertex, we can have a table of neighbors that maps each neighbor to its label (or weight). It would have type:

$$(\texttt{label}\ \mathit{vertexTable})\ \mathit{vertexTable}.$$

3 Priority First Search

Generalizing BFS and DFS, *priority first search* visits the vertices in some priority order. This priority order can either be static and decided ahead of time, or can be generated on the fly by the algorithm. To apply priority first search, we only need to make sure that at every step, we have a priority value for all the unvisited vertices adjacent to the visited vertices. This allows us to pick the best (highest priority) among them. When we visit a vertex, we might update the priorities of the remaining vertices. One could imagine using such a scheme for exploring the web so that the more interesting part can be explored without visiting the whole web. The idea might be to rank the outgoing links based on how interesting the tag on the link appears to be. Then, when choosing what link to visit next, choose the best one. This link might not be from the page you are currently on.

Many famous graph algorithms are instances of priority first search. For example, Dijkstra's algorithm for finding single-source shortest paths (Sssp) from a single source on a weighted graph and Prim's algorithm for finding Minimum Spanning Trees (Mst).

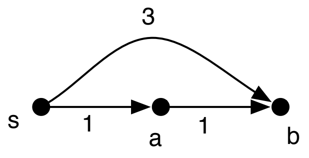
3.1 Shortest Weighted Paths

The single-source shortest path (Sppp) problem is to find the shortest (weighted) path from a source vertex s to every other vertex in the graph. We'll need a few definitions to describe the problem more formally. Consider a graph (either directed or undirected) graph $G = (V, E)$. A *weighted graph* is a graph $G = (V, E)$ along with a weight function $w : E \to \mathbb{R}$ that associates with every edge a real-valued weight. Thus, the *weight of a path* is the sum of the weights of the edges along that path.

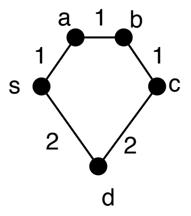
Problem 3.1 (The Single-Source Shortest Path (Sssp) Problem)**.** Given a weighted graph $G = (V, E)$ and a source vertex s, the *single-source shortest path (SSSP) problem* is to find the shortest weighted path from s to every other vertex in V.

We will use $\delta_G(u, v)$ to indicate the weight of the shortest path from u to v in the weighted graph G. Dijkstra's algorithm solves the Sssp problem when all the weights on the edges are non-negative. Dijkstra's is a very important algorithm both because shortest paths have many applications but also because it is a very elegant example of an efficient greedy algorithm that generates optimal solutions on a nontrivial task.

Before describing Dijkstra's algorithm, we would like to understand why we need a new algorithm for the weighted case. Why, in particular, doesn't BFS work on weighted graphs? Consider the following directed graphs on 3 nodes:



In this example, BFS would visit b then a. This means when we visit b, we assign it an incorrect weight of 3. Since BFS never visit it again, we'll never update it to the correct value. Note that we cannot afford to visit it again as it will be much more expensive. This problem still persists even if the graph is geometric (and satisfies the triangle inequality), as the following example shows:



But why does BFS work in unweighted case? The key idea is that it works outwards from the source. For each frontier F_i, it has the correct unweighted distance from source to each vertex in the frontier. It then can determine the correct distance for unencountered neighbors that are distance one further away (on the next frontier).

Let's consider using a similar approach when the graphs has non-negative edge weights. Starting from the source vertex s, for which vertex can we safely say we know its shortest path from s? The vertex v that is the closest neighbor of s. There could not be a shorter path to v, since such a path would have to go through one of the neighbors that is further away from s and that path cannot get shorter because none of the edge weights are shorter. More generally, if we know the shortest path distances for a set of vertices, how can we determine the shortest path to another vertex?

Let's think inductively. Consider the vertices sorted by their shortest path distance from s. That is, let $X = \langle v_1, v_2, ...v_n \rangle$ be the vertices in $G(V, E)$ sorted by $\delta(s, v_i)$. Now suppose you have found $\delta(s, v_i)$ for the first k vertices. That is, we know $\delta(s, v_i)$ for the subset $X_k = \langle v_1, v_2, ...v_k \rangle$. How do you know which vertex is v_{k+1}, and how can you find $\delta(s, v_{k+1}$? It must be the vertex that is not in X_k but is the next closest to s.

What do we know about the shortest path from s to v_{k+1}? Suppose the vertex u is the node just before v_{k+1} in the shortest path from s to v_{k+1}. Since all the edge weights are non-negative, u must be closer or at least as close to s as v_{k+1} is. Therefore u must be in X_k because otherwise v_{k+1} could not be closest vertex outside X_k. Then it must be the case that v_{k+1} is a neighbor of the set X_k. That is, the shortest path from s to v_{k+1} is a known shortest path extended by a single edge.

Next time we will consider how we can find v_{k+1} efficiently and give the complete algorithm.

4 SML code

Here we present the SML code for BFS using sequences.

```
functor SeqBFS(STSeq : ST_SEQUENCE) =
struct
  open STSeq
  type vertex = int
  type graph = (int seq) seq

  fun N(G : graph , F : int seq) =
  let
    fun UV v = Seq.map (fn u => (u, SOME(v))) (Seq.nth G v)
  in
    Seq.flatten(Seq.map UV F)
  end

  fun BFS(G : graph , s : vertex) =
  let
    val n = Seq.length G

    fun BFS'(P : int option stseq, F : int seq) =
      if (Seq.length(F) = 0)
      then toSeq(P)
      else let
        val Nbr = Seq.filter (fn (u,v) => (nth P u) = NONE) (N(G, F))
        val P' = inject Nbr P
        val F' = Seq.filter (fn (u,v) => (nth P' u) = v) Nbr
        val F'' = Seq.map (fn (u,v) => u) F'
      in BFS'(P', F'') end

    val P = Seq.tabulate (fn i => if (i=s) then SOME(s) else NONE) n

  in
    BFS'(fromSeq(P), Seq.singleton(s))
  end

end
```