Lecture 11 — Shortest Paths, Graph Representations Revisited

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

Lectured by Guy Blelloch — October 4, 2011

Today:

- Using Array Sequences to represent graphs
- Continuation of Bellman-Ford's Algorithm (see last lecture notes)
- All pairs shortest paths

1 Representing Graphs With Arrays

Graphs can be represented with an array implementation of sequences instead of tables and sets. The benefit is that we can improve asymptotic performance of certain algorithms, but the cost is that this representation is less general, requiring us to restrict the names of vertices to integers in a fixed range. This can become inconvenient in graphs that change dynamically but is typically OK for static graphs. Furthermore, in the functional setting, we have to be careful since by default, updating an array requires copying the whole array; this is because of the data persistency requirement. This means that if we make an update, we also have to keep the old version. This also means we cannot simply overwrite a value in an existing array. Here, we will describe the interface for a single-threaded sequence that supports efficient updates on the most recent version but works correctly on any version. This is similar to some of the ideas you covered in 15-150.

We refer to the type of this sequence as a `stseq` and it supports the following interface.

	Work	Span
`fromSeq(S) : α seq → α stseq` Converts from a regular sequence to a stseq.	$O(\|S\|)$	$O(1)$
`toSeq(ST) : α stseq → α seq` Converts from a stseq to a regular sequence.	$O(\|S\|)$	$O(1)$
`nth ST` i `: α stseq → int → α` Returns the i^{th} element of ST. Same as for seq.	$O(1)$	$O(1)$
`update` (i,v) S `: (int × α) → α stseq → α stseq` Replaces the i^{th} element of S with v.	$O(1)$	$O(1)$
`inject` I S `: (int × α) seq → α stseq → α stseq` For each $(i,v) \in I$ replaces the i^{th} element of S with v.	$O(\|I\|)$	$O(1)$

The costs for `nth`, `update` and `inject` assume the user is using the most recent version. Here we will not define the costs unless using the most recent versions.

Now let's say that we have a graph $G = (V, E)$ where $V = \{0, 1, \ldots, n-1\}$. We will refer to such a graph 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

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

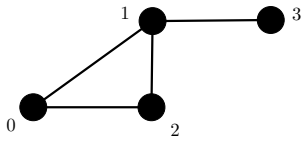
is equivalent to the sequence

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

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 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. This is what was needed for the homework, for example. We can represent the IL graph as a `(int seq) seq`. We can represent the parent mapping P as a `(int option) stseq`. The option is used to indicate whether there is a parent yet. The locations in P with $\mathit{SOME}(v)$ are the visited vertices. We can represent the frontier as an integer sequence containing all the vertices in the frontier. We make P of type `stseq` since it gets updated with changes which are potentially small compared to its length. 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⟨⟨(u, v) : u ∈ G[v]⟩ : v ∈ F⟩   % neighbor edges of frontier
7              val P' = inject(N, P)                          % new parents added
8              val F' = ⟨u : (u, v) ∈ N ∧ P'[u] = v⟩           % remove duplicates
9          in BFS'(P', F') end
10   val P_init = ⟨if (v = s) then SOME(s) else NONE
11                 : v ∈ ⟨0, ..., |G| − 1⟩⟩
12 in BFS'(toSTSeq(P_init), ⟨s⟩)
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	$O(\sum_{v\in F} \lvert N_G(v)\rvert)$	$O(\log n)$	$O(\sum_{v\in F} \lvert N_G(v)\rvert)$	$O(\log n)$
7	$O(\sum_{v\in F} \lvert N_G(v)\rvert)$	$O(1)$	$O(n)$	$O(1)$
8	$O(\sum_{v\in F} \lvert N_G(v)\rvert)$	$O(\log n)$	$O(\sum_{v\in F} \lvert N_G(v)\rvert)$	$O(\log n)$
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

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

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

```
1  fun DFS(G : set table, s : key) =
2  let fun DFS'(X : set, v : key) =
3        if (v ∈ X) then X
4        else iterate DFS' (X ∪ {v}) (G_v)
5  in DFS'({s}, s) end
```

And the version using sequences.

```
1  fun DFS(G : (int seq) seq, s : int) =
2  let
3     fun DFS'(X : bool stseq, v : int) =
4        if (X[v]) then X
5        else iterate DFS' (update(X, v, true)) (G[v])
6     val X_init = ⟨(v = s) : v ∈ ⟨0, ..., |G| − 1⟩⟩
7  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 All Pairs Shortest Paths

The all pairs shortest path (APSP) problem is: given a weighted directed graph $G = (V, E)$, find the shortest weighted path between ever pair of vertices in V.

It turns out that if the edges all have non-negative weights, then for sparse graphs, the fastest way to solve the APSP problem is simply by solving the (non-negative weight) SSSP problem using each of the

vertices as a source. All these can be run in parallel, making the all-pairs problem much more parallel than the single-source problem. If we use the implementation of Dijkstra's algorithm described earlier, and with $n = |V|$ and $m = |E|$, then the overall work and span is:

$$\begin{aligned} W(m,n) &= \sum_{s \in V} O(m \log n) \\ &= O(mn \log n) \\ S(m,n) &= \max_{s \in V} O(m \log n) \\ &= O(m \log n) \end{aligned}$$

What about for the general case allowing for negative weights? Later in the course we will see that we can use matrix multiply to solve the APSP problem in $O(n^3 \log n)$ work and $O(\log^2 n)$ span. This is very parallel, but for $m \ll n^2$, it is much more costly than using multiple instances of Dijkstra's algorithm.

For sparse graphs, there is actually another solution that works with negative weights and matches the cost of using multiple instances of Dijkstra's algorithm. The approach boils down to first using Bellman-Ford to convert the graph into one that has no negative weights, and then using Dijkstra's algorithm in parallel across the vertices. The graph is converted in a way such that the shortest path taken between every pair of vertices does not change although the weights of the edges on the path might. Before going further, it is worth thinking about what systematic changes on edges will not change the shortest path taken between two vertices. You might think of simply increasing all the weights on all the edges equally. Unfortunately, this does not work.

Exercise 1. *Come up with a small example that shows that increasing the weight of all edges in the graph equally, changes the shortest paths.*

We could consider multiplying all the weights by any positive number. This will not affect the paths taken, but unfortunately, it does not help us since the negative edges will remain negative. What if we take a vertex and increase every out edge by some constant p_v and decrease every in-edge by the same constant. Does this maintain the shortest paths? Let us consider the possible cases. Firstly, if a path goes through v, then the weight of the path is decreased by v on the edge into v and increased by the same amount on the edge out of v. Therefore, the weight of the path is not changed. Secondly, a path could start at v. In this case all paths out of v increased by the same amount so the shortest path is not affected (although its weight will be). Similarly, for the paths that finishes at v: all paths will be decreased by the same amount. Therefore, this transformation would appear not to change the shortest path between any pair of vertices.

If we can modify the in- and out- weights of one vertex, then we can modify them all. This suggests an idea of assigning values to all vertices and adjusting the edges accordingly. More specifically, we'll assign a real valued "potential" to each vertex, which we indicate as $p : v \to \mathbb{R}$. Now each directed edge (u, v) will be reweighted so that its new weight is

$$w'(u,v) = w(u,v) + p(u) - p(v).$$

(i.e. we add the potential of u going out of u, and subtract the potential of v coming in to v. This leads to the following lemma:

Lemma 2.1. *Given a weighted directed graph $G = (V, E, w)$ with weight function $w : E \to \mathcal{R}$, and "potential" function $p : v \to \mathbb{R}$, then for a graph $G' = (V, E, w')$ with weights*

$$w'(u, v) = w(u, v) + p(u) - p(v),$$

we have that for every path from s to t,

$$W_{G'}(s, t) = W_G(s, t) + p(s) - p(t)$$

where $W_G(s, t)$ is the weight of the path from s to t in graph G.

Proof. Each s-t path is of the form $\langle v_0, v_1, ..., v_k \rangle$, where $v_0 = s$ and $v_l = t$. For every vertex $v_i, 0 < i < k$ in the path the conversion to w' removed p_{v_i} from the weight when entering v_i and added it back in when leaving, so they cancel (the overall sum is a telescoping sum). Therefore, we need only consider the two endpoints, giving the desired result. □

This lemma shows that the potential weight transformation does not affect the shortest paths since all paths between the same two vertices will be affected by the same amount (i.e. $p(s) - p(t)$ for vertices s and t).

Now the question is whether we can pick potentials so that they get rid of all negative weight edges. The answer is yes. The trick is to add a source vertex and link it to all other vertices V with an edge weight of zero. Now we find the shortest path from s to all vertices using the original weights. This can be done with a SSSP algorithm that allows negative weights (e.g. Bellman-Ford). Now we simply use the distance from s as the potential.

Exercise 2. *Show that the this potential guarantees that no edge weight will be negative.*

This together with a parallel application of SSSP with non-negative weights (indicated as SSSP$^+$) gives us our desired algorithm. In particular:

```
1  fun APSP(G) =
2  let
3     val G' = G  with a new source s and 0 weight edges from s to each v ∈ V
4     val D = SSSP(G', s)
5     val W'' = {(u, v) ↦ w(u, v) + D(u) − D(v) : (u, v) ∈ E_G'}
6     val G'' = (V_G', E_G', W'')
7  in
8     {v ↦ SSSP+(G'', v) : v ∈ V}
9  end
```

We now consider the work and span of this algorithm. If Bellman Ford is used for SSSP and Dijkstra for SSSP$^+$ then we can just add the work and span of Bellman Ford to the work and span we analyzed earlier for the parallel application of Dijkstra. The cost of the parallel application of Dijkstra dominates giving $O(mn \log n)$ work and $O(m \log n)$ span.