Recitation 5 — Graphs and StSeqs

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2013)

February 13, 2013

1 Announcements

- How did Assignment 3 go?
- Assignment 4 has been released.
- Questions?

2 Graph Representations

Suppose you're given an undirected graph $G = (V, E)$, where V is a set of vertices (also known as nodes) and $E \subseteq \binom{V}{2}$ is a set of edges. Notice that in this definition, an edge $e = \{x, y\}$ is a set of size two representing the endpoints. Thus, the edge $\{x, y\}$ represents the same edge as $\{y, x\}$. Following this, we can think of representing a graph simply as an edge set. But the edge set representation doesn't provide an efficient means to access the neighbors of a vertex. That's why in class we looked at a different representation which we call an adjacency table. In the following, we'll look at a simple problem that will illustrate the differences in complexity between these two common representations.

Consider writing a simple routine to compute the degree of a vertex v. First, how would we do it in the edge set representation? Let the edge set E be represented as a set.

```
1  function degree E v =
2    let
3       nbrs = filter (fn (x,y) => v=x orelse v=y) E
4    in
5      size nbrs
6    end
```

What's the cost of this function? Linear work and $O(\log n)$ span.

Now suppose we're given a graph represented in the edge set form. Can we convert it into an adjacency table? What is the type of an adjacency table? We use `int set IntTable.table`.

```
1  function makeAdjTable E =
2    let
3       biDir = flatten (map (fn (x,y) => fromList [(x,y), (y,x)]) (toSeq E))
4       nbrsSeq = collect Int.compare biDir
5    in
6      Table.fromSeq (map (fn (u, nbrs) => (u, Set.fromSeq nbrs)) nbrsSeq)
7    end
```

What's the cost of `makeAdjTable`? $O(n \log n)$ work and $O(\log^2 n)$ span.

Finally, in this adjacency table representation, it's super easy to compute the degree.

```
1  function degree' G v =
2    let
3       nbrs = find G v
4    in
5      size nbrs
6    end
```

What's the cost? $O(\log n)$ cost (both work and span).

3 Friends of Friends

Suppose we have just founded a new social networking site and naturally we have represented the network of friends as a graph. We are using the adjacency table representation of the graph. As discussed in class, an adjacency table is a table of sets where each element in the table maps a vertex to a set of its neighbors.

Suppose we want to find friends of our friends. These would be the neighbors of our neighbors. That is, find all vertices that have a distance of two from a source vertex v. We will need to introduce another function `tabulate` that can be found in the Table ADT. It allows us to perform a `map` over a set, but it creates a table, where the keys are from the set and values are the result of applying the map function. (Recall, there is no `map` for sets, because the results of the map may not result in unique values.) `Table.tabulate` has signature:

```
Table.tabulate : (Table.key -> 'a) -> Table.set -> Table.table
```

For example if we had a set $S = \{3, 4, 5\}$ and applied

```
val squares = Table.tabulate (fn x => x*x) S
```

then `squares` $= \{3 \mapsto 9, 4 \mapsto 16, 5 \mapsto 25\}$.

To find friends of friends we might write:

```
1  type graph = Set.set Table.table
2  function FoF (G : graph) v =
3  let
4    function neighbors v = Table.find G v
5    Ngh = neighbors v
6    NghNgh = Table.tabulate neighbors Ngh
7  in
8    Seq.filter (fn x => x <> v) (Table.reduce Table.Set.union {} NghNgh)
9  end
```

In Line 5 we find the neighbors of v and in Line 6 we find the neighbors' neighbors, which now is a table mapping each neighbor to its neighbors. To flatten these sets of neighbors we do a `reduce` using set union for combining so that duplicates are removed. Note that the friends of our friends might not include our friends.

For example, let's say we start with the graph:

$$G = \{\text{“a”} \mapsto \{\text{“b”}, \text{“c”}\}, \text{“b”} \mapsto \{\text{“a”}, \text{“d”}\}, \text{“c”} \mapsto \{\text{“a”}, \text{“d”}, \text{“e”}\}, \text{“d”} \mapsto \{\text{“b”}, \text{“c”}\}, \text{“e”} \mapsto \{\text{“c”}\}\}$$

Now we have:

```
 FoF G "a"
⇒  {"d","e"}
```

with intermediate results:

$\texttt{Ngh} = \{\text{“b”},\text{“c”}\}$

$\texttt{NghNgh} = \{\text{“b”} \mapsto \{\text{“a”},\text{“d”}\}, \text{“c”} \mapsto \{\text{“a”},\text{“d”},\text{“e”}\}\}$

The `reduce` with `union` will combine the two sets, and applying a `filter` gives the answer.

4 Representing Graphs With Array Sequences

When graphs are implemented using tables and sets, the cost of finding a vertex in the table is $(\log n)$ work and span (recall that $n = |V|$). We can improve the asymptotic performance of certain algorithms if the search is constant work. An array sequence is an obvious data structure that has lookup with constant work.

To take advantage of the faster access of sequences we can use sequences to represent graphs. But the drawback is that this representation is less general, requiring the vertex names be restricted to integers in a fixed range. This restriction can become inconvenient in graphs that change dynamically but typically is 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

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

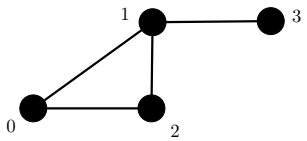
using standard reductions between sequences and sets. If we use an array representation of sequences, then this gives us constant work access to the values stored at vertices. We can also represent the set of neighbors of a vertex as an integer sequence containing the indices of those neighbors. Therefore, instead of using a

```
set table
```

to represent a graph we can use a

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

For example, the following undirected graph:



would be represented as

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

The question is what are the costs of various operations on such a representation. We covered the costs for an adjacency table representation that we covered in class? What might they be using adjacency sequences?

	adj table		**adj seq**	
	work	*span*	*work*	*span*
`isEdge`$(G,(u,v))$	$O(\log n)$	$O(\log n)$	$O(d_G^+(u))$	$O(\log d_G^+(u))$
`map over all edges`	$O(m)$	$O(\log n)$	$O(m)$	$O(1)$
`map over neighbors of` v	$O(\log n + d_G^+(v))$	$O(\log n)$	$O(d_G^+(v))$	$O(1)$
$d_G^+(v)$	$O(\log n)$	$O(\log n)$	$O(1)$	$O(1)$

5 Reviewing `stseq`

Single-threaded sequences introduced 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 appear 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.

	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.	$O(\|I\|)$	$O(1)$

5.1 Discussing complexity of `stseq` operations

```
1  let
2     a = ⟨2,5,1,7⟩
3  in
4        b = stseq.fromSeq a
5        c = update (0,3) b
6        d = update (1,2) c
7        d′ = update (3,12) c
8        e = nth c 5
9        f = nth d′ 5
10 end
11
```

Line 6: O(1)

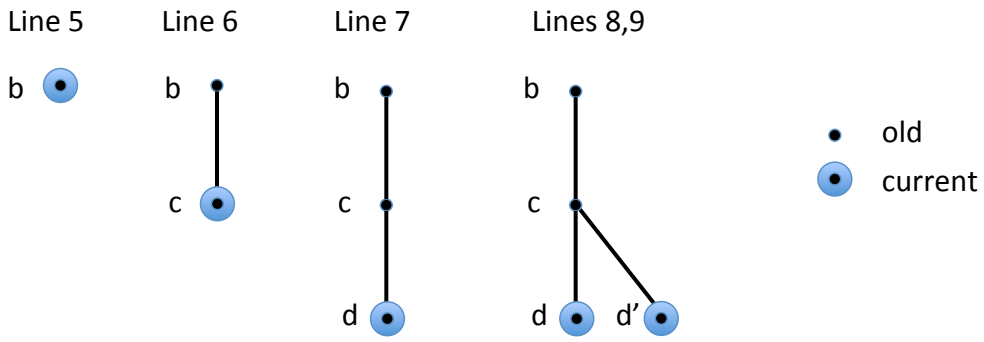
Line 7: O(possibly large number)

Line 8: O(large number)

Line 9: O(1)

Updating the current version takes constant work. However, updates can take a long time if the update is not made to the most recent version of the stseq. This is why the `update` from line 7 could also take a long time. Similarly `nth` can take a long time if not applied on the most recent version, as in line 8. When operating on the current version we can just look up the value in the current copy, which is up to date, as is the case in lines 6 and 9. When operating on an old version, however, we have to go back to the original copy and then check all the changes in the change log to see if any have modified the location we are asking about. This can be expensive. This is why updating and reading the current version is cheap ($O(1)$ work) while working with an old version is expensive.

The figure below illustrates which versions of the `stseq`s are current and old for each line of code above:



5.2 Marking my friends friend's with StSeqs

We started with an example of finding the neighbors of the neighbors of a vertex v in a graph represented as a `set table`. The expensive operation was taking the union of all the sets of neighbors. Instead, here we represent the graph using array sequences, and use a Boolean `stseq` to mark which vertices are the neighbors of neighbors. We assume that all values are initially `false`.

```
1  type graph = (int Seq.seq) Seq.seq

2  function FoF (flags : bool StSeq.stseq) (G : graph) (v : int) =
3  let
4    function neighbors v = Seq.nth G v
5     NghNgh = Seq.flatten(Seq.map neighbors (neighbors v))
6     I = Seq.map (fn i ⇒ if i <> x then (i,true) else (i,false)) NghNgh
7  in
8    StSeq.inject I flags
9  end
```

In this code the `flags` argument is a Boolean `stseq` of length $|V|$. The sequence `NghNgh` contains all neighbors' neighbors of v, but can have repeats. For each, it writes `true` to that vertex' index in the `flags` sequence using `inject`. Now the expensive operation is flatten which uses $O(d^+(v) + \sum_{u \in N^+(v)} d^+(u))$ work and has $O(\log d^+(v))$ span.