Lecture 7 — Graph Representations, Sets, and Tables

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

Lectured by Kanat Tangwongsan — February 7, 2012

Material in this lecture:
- Graph Representations
- Sets and Tables

1 How should we represent a graph?

How we want to represent a graph largely depends on the operations we intend to support. To begin this lecture, let's look at a few questions we want to be able to answer efficiently about a graph. As a running example, suppose you just founded a new social networking site. Naturally, the network of friends is represented as a graph, and you're interested in supporting the following operations:

(1) Is x a friend of y?

(2) For each user x, how many friends does x have?

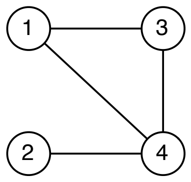
(3) Find friends of x who are not in group Y.

(4) Find friends of friends of x.

(5) Find the diameter of this graph.

Traditionally, there are 4 main representations, all of which assume that vertices are numbered from $1, 2, \ldots, n$ (or $0, 1, \ldots, n-1$):

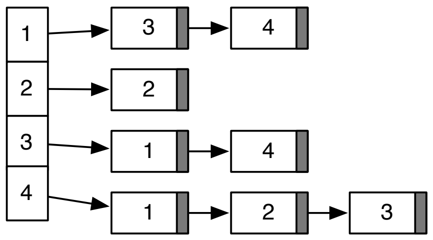
- **Adjacency matrix.** An $n \times n$ matrix of binary values in which location (i, j) is 1 if $(i, j) \in E$ and 0 otherwise. Note that for an undirected graph the matrix is symmetric and 0 along the diagonal.



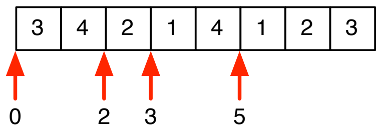
$$\begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

- **Adjacency list.** An array A of length n where each entry $A[i]$ contains a pointer to a linked list of all the out-neighbors of vertex i. In an undirected graph with edge $\{u, v\}$ the edge will appear in the adjacency list for both u and v.

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



- **Adjacency array.** Similar to an adjacency list, an adjacency array keeps the neighbors of all vertices, one after another, in an array `adj`; and separately, keeps an array of indices that tell us where in the `adj` array to look for the neighbors of each vertex.



- **Edge list.** A list of pairs $(i, j) \in E$.

Since operations on lists are inherently sequential, in this course, we are going to raise the level of abstraction so that parallelism is more natural. At the same time, we also want to loosen the restriction that vertices need to be labeled from 1 to n and instead allow for any labels. Conceptually, though, the representations we describe are not much different from adjacency lists and edge lists. We are just going to think parallel and base our view on data types that are parallel.

When we make decisions about how to represent graphs, we should consider take into account the costs of basic operations on graphs that we intend to support. The following operations are common among standard graph algorithms:

1. $\deg(v)$ — finding the degree of a vertex;
2. *Is* $\{u, v\} \in E(G)$*?* — finding if an edge is in the graph;
3. *Map over edges* — mapping or iterating over all edges in the graph; and
4. *For each* $v \in N(v)$ — mapping or iterating over all neighbors of a given vertex.

In a dynamically changing graph, we will also want to insert and delete edges.

Idea #1: Suppose we want a representation that directly follows the mathematical definition of a graph and simply uses a set of edges $E \subseteq \binom{V}{2}$. A moment's thought shows that we can actually support these basic operations rather efficiently using sequences. Sets are such an important and useful concept; we should create an ADT for it and see how to best support set operations.

2 An Abstract Data Type for Sets

Sets undoubtedly play an important role in mathematics and are often needed in the implementation of various algorithms. Whereas a sequence is an ordered collection, a set is its *unordered* counterpart.

Most programming languages either support sets directly (e.g., Python and Ruby) or have libraries that support them (e.g., in the C++ STL library and Java collections framework). They sometimes have more than one implementation of sets. For example, Java has sets based on hash tables and balanced trees. Unsurprisingly, the set interface in different libraries and languages differ in subtle ways. So, when using one of these interfaces you should always read the documentation carefully.

We now define an abstract data type for sets. The definition follows the mathematical definition of sets from set theory and is purely functional. In particular, when updating a set (e.g. with insert or delete) it returns a new set rather than modifying the old set.

Definition 2.1. For a universe of elements $\mathbb{U}$ (e.g. the integers or strings), the SET abstract data type is a type $\mathbb{S}$ representing the powerset of $\mathbb{U}$ (i.e., all subsets of $\mathbb{U}$) along with the following functions:

`empty`	:	$\mathbb{S}$	=	$\emptyset$
`size`(S)	:	$\mathbb{S} \to \mathbb{Z}_{\geq 0}$	=	$\lvert S \rvert$
`singleton`(e)	:	$\mathbb{U} \to \mathbb{S}$	=	$\{e\}$
`filter`(f, S)	:	$((\mathbb{U} \to \{\mathtt{T}, \mathtt{F}\}) \times \mathbb{S}) \to \mathbb{S}$	=	$\{s \in S \mid f(s)\}$
`find`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \{\mathtt{T}, \mathtt{F}\}$	=	$\lvert\{s \in S \mid s = e\}\rvert = 1$
`insert`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \cup \{e\}$
`delete`(S, e)	:	$\mathbb{S} \times \mathbb{U} \to \mathbb{S}$	=	$S \setminus \{e\}$
`intersection`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cap S_2$
`union`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \cup S_2$
`difference`(S_1, S_2)	:	$\mathbb{S} \times \mathbb{S} \to \mathbb{S}$	=	$S_1 \setminus S_2$

where $\mathbb{Z}_{\geq 0}$ are the non-negative integers.

We write this definition to be generic and not specific to Standard ML. In our library, the type $\mathbb{S}$ is called `set` and the type $\mathbb{U}$ is called `key`, the arguments are not necessarily in the same order, and some of the functions are curried. For example, the interface for find is `find` : `set` $\to$ `key` $\to$ `set`. Please refer to the documents for details. In the pseudocode, we will give in class and in the notes we will use standard set notation as in the right hand column of the table above.

A Note about `map`. You may notice that the interface does not contain a `map` function. If we try to generalize the notion of `map` from sequences, a `map` function does not make sense in the context of a set: if we interpret map to take in a collection, apply some function to each element and return a collection of the same structure. Consider a function that always returns 0. Mapping this over a set would return all zeros, which would then be collapsed into a singleton set, containing exactly 0. Therefore, such a `map` would allow reducing the set of arbitrary size to a singleton, which doesn't match the `map` paradigm (which traditionally preserves the structure and size).

2.1 Cost Model

So far, we have laid out a semantic interface, but before we can put it to use, we need to worry about an implementation and how costly each operation is. The most common efficient ways to implement sets are either using hashing or balanced trees. They have various tradeoffs in cost. For simplicity, we'll consider a cost model based on a balanced-tree implementation[1]. We will cover how to implement these set operations when we talk about balanced trees later in the course. For now, a good intution to have is that we use a comparison function to keep the elements in sorted order in a balanced tree. Since this requires comparisons inside the various set operations, the cost of these comparisons affects the work and span. For this, we'll assume that compare has C_w work and C_s span.

	Work	*Span*
`size`(S) `singleton`(e)	$O(1)$	$O(1)$
`filter`(f,S)	$O\left(\sum_{e\in S} W(f(e))\right)$	$O\left(\log\|S\| + \max_{e\in S} S(f(e))\right)$
`find`(S,e) `insert`(S,e) `delete`(S,e)	$O(C_w \cdot \log\|S\|)$	$O(C_s \cdot \log\|S\|)$
`intersection`(S_1,S_2) `union`(S_1,S_2) `difference`(S_1,S_2)	$O\left(C_w \cdot m \cdot \log(1+\frac{n}{m})\right)$	$O\left(C_s \cdot \log(n+m)\right)$

where $n = \max(|S_1|,|S_2|)$ and $m = \min(|S_1|,|S_2|)$.

The bounds for `intersection`, `union`, and `difference` deserve further discussion. At a glance, the work bounds for `intersection`, `union`, and `difference` might seem a bit funky. These turn out to be both the theoretical upper and lower bounds for any comparison-based implementation of sets. We will get to this later.

Notice that when the two sets have roughly the same length (say, $n \le \alpha m$, where $\alpha \ge 1$ is a constant), the work is simply

$$O(C_w \cdot m \cdot \log(1+\alpha)) = O(C_w \cdot n).$$

This should not be surprising, because it corresponds to the cost of merging two approximately equal length sequences (effectively what these operations have to do). As you'll soon figure out (Homework 3), this can be done in linear work.

Moreover, you should also observe that in the case that one of the sets is a singleton, then the work is $O(\log n)$.

Buy in Bulk and Save. On inspection, the functions `intersection`, `union`, and `difference` are simply the "parallel" counterparts of the functions `find`, `insert`, and `delete`, to wit:

- `intersection` — search for multiple elements instead of one.

[1]dealing with hash tables in a functional setting where data needs to be persistent is somewhat complicated.

- `union` — insert multiple elements.
- `difference` — delete multiple elements.

In fact, it is easy to implement `find`, `insert`, and `delete` in terms of the others.

$$\begin{aligned}\texttt{find}(S,e) &= \texttt{size}(\texttt{intersection}(S,\texttt{singleton}(e))) = 1\\ \texttt{insert}(S,e) &= \texttt{union}(S,\texttt{singleton}(e))\\ \texttt{delete}(S,e) &= \texttt{difference}(S,\texttt{singleton}(e))\end{aligned}$$

Since `intersection`, `union`, and `difference` can operate on multiple elements they are well suited for parallelism, while `find`, `insert`, and `delete` have no parallelism. Consequently, in designing parallel algorithms it is good to think about how to use `intersection`, `union`, and `difference` instead of `find`, `insert`, and `delete` if possible. For example, one way to convert a sequence to a set would be to insert the elements one by one, which can be coded as

```
fun fromSeq(S) = Seq.iter (fn (S',e) => Set.insert e S') Set.empty S
```

However, the above is sequential. To do it in parallel we could instead do

```
fun fromSeq(S) = Seq.reduce Set.union Set.empty (Seq.map Set.singleton S)
```

Exercise 1. *What is the work and span of the first version of* `fromSeq`.

Exercise 2. *Show that on a sequence of length n the second version of* `fromSeq` *does* $O(C_w n \log n)$ *work and* $O(\log^2 n)$ *span.*

3 Tables: Associating Each Element With A Value

Suppose we want to extend sets so that each element is associated with a payload. A table is an abstract data type that stores for each key data associated with it. The table ADT supplies operations for finding the value associated with a key, for inserting new key-value pairs, and for deleting keys and their associated value. Tables are also called dictionaries, associative arrays, maps, mappings, and functions (in set theory). Given our focus on parallelism, the interface we will discuss also supplies "parallel" operations that allow the user to insert multiple key-value pairs, to delete multiple keys, and to find the values associated with multiple keys.

In this class, the notation we are going to be using is

$$\{(k_1 \mapsto v_1), (k_2 \mapsto v_2), \ldots, (k_n \mapsto v_n)\},$$

where we have *keys* and *values*—and each key k_i is associated with the value v_i. Mathematically, a table is simply a set of pairs and can therefore be written as a set of ordered pairs $\{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$. Our notation choice is largely to better identify when tables are being used.

As with sets, tables are commonly used in many applications. Most languages have tables either built in (e.g. dictionaries in Python, Perl, and Ruby), or have libraries to support them (e.g. map in the C++ STL library and the Java collections framework). We note that the interfaces for these languages and libraries have common features but typically differ in some important ways, so be warned. Most do not support the "parallel" operations we discuss. Again, here we will define tables mathematically in terms of set theory before committing to a particular language.

Formally, a table is set of key-value pairs where each key appears only once in the set. Such sets are called *functions* in set theory since they map each key to a single value. We will avoid this terminology so that we don't confuse it with functions in a programming language. However, note that the `(find T)` in the interface is precisely the "function" defined by the table `T`. In fact it is a *partial function* since the table might not contain all keys and therefore the function might not be defined on all inputs. Here is the definition of a table.

Definition 3.1. For a universe of keys $\mathbb{K}$, and a universe of values $\mathbb{V}$, the TABLE abstract data type is a type $\mathbb{T}$ representing the powerset of $\mathbb{K} \times \mathbb{V}$ restricted so that each key appears at most once (i.e., any set of key-value pairs where a key appears just once) along with the following functions:

$$
\begin{array}{lcll}
\texttt{empty} & : & \mathbb{T} & = \emptyset \\
\texttt{size}(T) & : & \mathbb{T} \to \mathbb{Z}_{\geq 0} & = |T| \\
\texttt{singleton}(k,v) & : & \mathbb{K} \times \mathbb{V} \to \mathbb{T} & = \{(k,v)\} \\
\texttt{filter}(f,T) & : & ((\mathbb{V} \to \{\texttt{T},\texttt{F}\}) \times \mathbb{T}) \to \mathbb{T} & = \{(k,v) \in T \mid f(v)\} \\
\texttt{map}(f,T) & : & ((\mathbb{K} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T}) \to \mathbb{T} & = \{(k, f(k,v)) \mid ((k,v) \in T)\} \\
\texttt{insert}(f,T,(k,v)) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times (\mathbb{K} \times \mathbb{V}) \to \mathbb{T} & = \\
 & & \forall k \in \mathbb{K}, \begin{cases} (k, f(v,v'))\} & (k,v') \in T \\ (k,v) & (k,v') \notin T \end{cases} & \\
\texttt{delete}(T,k)) & : & \mathbb{T} \times \mathbb{K} \to \mathbb{T} & = \{(k',v) \in T | k \neq k'\} \\
\texttt{find}(T,k) & : & \mathbb{T} \times \mathbb{K} \to (\mathbb{V} \cup \perp) & = \begin{cases} v & (k,v) \in T \\ \perp & otherwise \end{cases} \\
\texttt{merge}(f,T_1,T_2) & : & (\mathbb{V} \times \mathbb{V} \to \mathbb{V}) \times \mathbb{T} \times \mathbb{T} \to \mathbb{T} & = \\
 & & \forall k \in \mathbb{K}, \begin{cases} (k, f(v_1,v_2)) & (k,v_1) \in T_1 \wedge (k,v_2) \in T_2 \\ (k,v_1) & (k,v_1) \in T_1 \\ (k,v_2) & (k,v_2) \in T_2 \end{cases} & \\
\texttt{extract}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \{(k,v) \in T | k \in S\} \\
\texttt{erase}(T,S) & : & \mathbb{T} \times \mathbb{S} \to \mathbb{T} & = \{(k,v) \in T | k \notin S\}
\end{array}
$$

where $\mathbb{S}$ is the powerset of K (i.e., any set of keys) and $\mathbb{Z}_{\geq 0}$ are the non-negative integers.

Distinct from sets, the `find` function does not return a Boolean, but instead it returns the value associated with the key k. As it may not find the key in the table, its result may be bottom ($\perp$). For this reason, in the Table library, the interface for find is `find` : `'a table` $\to$ `key` $\to$ `'a option`, where `'a` is the type of the values.

Unlike sets, when we insert an element, we can't simply ignore that element if it is already present—their values might be different. For this reason, the `insert` function takes a function $f : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ as an argument. The purpose of f is to specify what to do if the key being inserted already exists in the table; f is applied to the two values. This function might simply return either its first or second argument, or it can be used, for example, to add the new value to the old one. The parallel counterpart of `find` is the `merge` function, which takes a similar function since it also has to consider the case that an element appears in both tables.

We also introduce new pseudocode notation for `map` and `filter` on tables:

$$\{(k \mapsto f(v) \,|\, (k \mapsto v) \in T\}$$

is equivalent to $\texttt{map}(f, T)$ and

$$\{(k \mapsto v) \in T \,|\, p(v)\}$$

is equivalent to $\texttt{filter}(p, T)$.

The costs of the table operations are very similar to sets.

	Work	*Span*
$\texttt{size}(T)$ $\texttt{singleton}(k, v)$	$O(1)$	$O(1)$
$\texttt{filter}(f, T)$	$O\left(\sum_{(k,v)\in T} W(f(v))\right)$	$O\left(\log\|T\| + \max_{(k,v)\in T} S(f(v))\right)$
$\texttt{map}(f, T)$	$O\left(\sum_{(k,v)\in T} W(f(k,v))\right)$	$O\left(\max_{(k,v)\in T} S(f(k,v))\right)$
$\texttt{find}(S, k)$ $\texttt{insert}(T, (k, v))$ $\texttt{delete}(T, k)$	$O(C_w \log\|T\|)$	$O(C_s \log\|T\|)$
$\texttt{extract}(T_1, T_2)$ $\texttt{merge}(T_1, T_2)$ $\texttt{erase}(T_1, T_2)$	$O\left(C_w m \log(1 + \frac{n}{m})\right)$	$O\left(C_s \log(n + m)\right)$

where $n = \max(|T_1|, |T_2|)$ and $m = \min(|T_1|, |T_2|)$.

As with sets there is a symmetry between the three operations `extract`, `merge`, and `erase`, and the three operations `find`, `insert`, and `delete`, respectively, where the prior three are effectively "parallel" versions of the earlier three.

We note that, in the SML Table library we supply, the functions are polymorphic (accept any type) over the values but not the keys. In particular the signature starts as:

```
signature TABLE =
sig
  type 'a table
  type 'a t = 'a table
  structure Key : EQKEY
  type key = Key.t
  structure Seq : SEQUENCE
  type a seq = 'a Seq.seq
  type set = unit table
  ...
  val find : 'a table -> key -> 'a option
  ...
```

The `'a` in `'a table` refers to the type of the value. The key type is fixed to be `key`. Therefore there are separate table structures for different keys (e.g. `IntTable`, `StringTable`). The reason to do this is because all the operations depend on the key type since keys need to be compared for a tree implementation, or hashed for a hash table implementation. Also note that the signature defines `set` to be a `unit table`. Indeed a set is just a special case of a table where there are no values.

In the SML Table library, we supply a `collect` operation that takes a sequence of key-value pairs and produces a table that maps every key in S to all the values associated with it in S, gathering all the values with the same key together in a sequence. This is equivalent to using a sequence collect followed by a `Table.fromSeq`. Alternatively, it can be implemented as

```
fun collect(S) =
let
  val S' = ⟨ {k ↦ ⟨ v ⟩} : (k, v) ∈ S ⟩
in
  Seq.reduce (Table.merge Seq.append) {} S'
end
```

Exercise 3. *Figure out what this code does.*

4 Case Study: Building and Searching An Index

Imagine that you want to generate an index of the sort that Google, Bing, or Twitter creates, so that a user can make word queries and find the documents in which certain words occur. For the purpose of this lecture, we will only consider logical queries on words involving *and*, *or*, and *andnot*. For example, a query might look like

“CMU” *and* “fun” *and* (“courses” *or* “clubs”)

and it would return a list of entries that match the query (*i.e.*, contain the words “CMU”, “fun” and either “courses” or “clubs”).

These kinds of searchable indexes predate the web by many years. The Lexis system, for searching law documents, for example, has been around since the early 70s. Now searchable indexes are an integral part of most mailers and many operating systems. By default, Google supports queries with *and* and *adjacent to* but with an advanced search you can search with *or* as well as *andnot*.

Say you want to index tweets—and you want to support the following interface on tweets:

```
signature INDEX = sig
  type word = string
  type tweetId = string
  type 'a seq
  type index
  type docList

  val makeIndex : (tweetId * string) seq -> index
```

```
    val find : index -> word -> tweetList
    val And : tweetList * tweetList -> tweetList
    val AndNot : tweetList * tweetList -> tweetList
    val Or : tweetList * tweetList -> tweetList
    val size : tweetList -> int
    val toSeq : tweetList -> tweetId seq
  end
```

The input to makeIndex is a sequence of pairs each consisting of a tweet identifier and the contents of the tweet. For example, this might look like:

$T = \langle$ (“@kanat_101”, “grading was fun”),
(“@vincent_102”, “I had a fun time in 210 class today”),
(“@jon_103”, “campus food :(and i’m stuck grading”),
(“@margaret_104”, “In 297 today I was reading my email”),
(“@grant_105”, “I had fun at nick’s party”),
(“@alex_106”, “clubbing was no fun, but more fun than 210”),
(“@bill_107”, “these people don’t have a life”),
$\ldots\rangle$

We can make an index from these tweets:

```
val $f$ = find (makeIndex($T$))
```

In addition to making the index, this partially applies find on it. We can then use this index for various queries, for example:

```
toSeq(And($f$ "fun", Or($f$ "class", $f$ "210"))
  ⇒ ⟨"vincent", "alex"⟩

size($f$ "fun")
  ⇒ 4
```

We can implement this interface very easily using sets and tables. The makeIndex function can be implemented as follows.

```
fun makeIndex(tweets) =
let
  fun tagWords(name, str) = $\langle (w, \texttt{name}) : w \in \texttt{tokens(str)} \rangle$
  val Pairs = flatten $\langle$ tagWords($d$) : $d \in$ tweets $\rangle$
in
  $\{ w \mapsto \texttt{Set.fromSeq}(d) : (w \mapsto d) \in \texttt{Table.collect(Pairs)} \}$
end
```

Assuming that all tokens are constant length, the cost of makeIndex is dominated by the collect, which is basically a sort. It therefore has $O(n \log n)$ work and $O(\log^2 n)$ span assuming the words have constant length. The rest of the interface can be implemented as

```
fun find(T, v) = Table.find(T, v)
fun And(s_1, s_2) = $s_1 \cap s_2$
fun Or(s_1, s_2) = $s_1 \cup s_2$
fun AndNot(s_1, s_2) = $s_1 \setminus s_2$
fun size(s) = $|s|$
fun toSeq(s) = Set.toSeq(s)
```

Note that if we do a `size(f "red")` the cost is only $O(\log n)$ work and span. It just involves a search and then a length.

If we do `And(f "fun", Or(f "courses", f "classes"))`, we have the following the worst-case cost: the work is $O(\texttt{size(f "fun")} + \texttt{size(f "courses")} + \texttt{size(f "classes")})$ and the span is $O(\log |index|)$.

5 ...Back to Graphs

With an ADT for sets, we can implement an *edge set* representation directly. The representation is similar to an edge list representation mentioned before, but it abstracts away from the particular data structure used for the set—the set could be based on a list, an array, a tree, or a hash table.

If we use the balanced-tree cost model for sets, for example, then determining if an edge is in the graph requires much less work than with an edge list— only $O(\log n)$ instead of $\Theta(n)$ (i.e. following the list until the edge is found). The problem with edge sets, as with edge lists, is that they do not allow for an efficient way to access the neighbors of a given vertex v. Selecting the neighbors requires considering all the edges and picking out the ones that have v as an endpoint. Although with an edge set (but not an edge list) this can be done in parallel with $O(\log m)$ span, it requires $\Theta(m)$ work even if the vertex has only a few neighbors.

More efficient neighbor access. In our second representation, we aim to get more efficiency in accessing to the neighbors of a vertex. This representation, which we refer to as an *adjacency table*, is a table that maps every vertex to the set of its neighbors. Simply put, it is an edge-set table.

Therefore, in this representation, accessing the neighbors of a vertex v is cheap: it just requires a lookup in the table. Assuming the balanced-tree cost model for tables, this can be done in $O(\log n)$ work and span. Once the neighbor set has been pulled out, mapping a constant work function over the neighbors can be done in $O(d_G(v))$ work and $O(1)$ span. Looking up if an edge is in the graph requires the same work and span as with edge sets: $O(\log n)$. This is because we can first look up one side of the edge in the table and then the second side in the set that is returned. Note that an adjacency list is a special case of adjacency table where the table of vertices is represented as an array and the set of neighbors is represented as a list.

Friends of friends? Say our graph is a `unit table table`.

```
fun friends user = Table.find G user
fun FoF user =
  let val friends = friends user
      val f' = Table.map (fn (v,_) => Table.find G v) friends
  in Table.reduce Table.merge {} f'
  end
```

Cost Summary. The costs assuming the tree cost model for sets and tables can be summarized in the following table assuming the function being mapped uses constant work and span:

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

6 Teaser — How many "hops" do you need to go from Amy to Betsy?

With these operations, can you come up with an algorithm that computes the shortest-path distance—in terms of edge count—from point A to point B? We'll see this next time.