Recitation 9 — Graphs and Matrices

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

October 26, 2011

Today's Agenda:

- Announcements
- Path Sharing
- Set Cover
- Matrices and Graphs

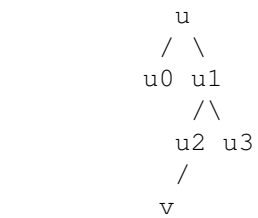
1 Announcements

- Exam 2 will be Thursday, November 10 at 7pm, in Doherty 2210.
- Emam 2 review will be Wednesday, November 9 at 7pm, in HH B103
- Assignment 7 will be out on Friday. It's a practice exam, but this time you'll hand it in for credit. Next week we will add a few additional problems on the newer material.
- Questions about homework or class?

2 Path Sharing

Let's say we have a tree stored as a table mapping nodes to their parents, and we want to compute the path from the root to a given vertex. (This should look familiar from hw4.)

Example tree:



Here's one way to do it.

```
(* findPath : 'a table -> 'a -> ('a * 'a) seq *)
let fun findPath (T:'a table) v =
          case (Table.find T v)
           of NONE => Seq.empty
            | SOME u => Seq.append(findPath u, Seq.singleton(u, v))
```

Q: What is the cost implication of this function?

A: Seq.append using an arraySequence implementation has $O(n)$ work and $O(1)$ span.

If $|V| = n$ and the length of the path from v to the root is p, then work for `findPath'` using append is

$$W(p) = W(p-1) + O(p + \log n) = O(p^2 + p \log n)$$

Here's a better way:

```
let fun findPath T v =
  let fun findPath' T v path =
          case (Table.find T v)
           of NONE => path
            | SOME u => (findPath u (u,v)::path)
  in
     Seq.fromList (findPath' T v [])
  end
```

Now the work for `findPath` using cons is

$$W(p) = W(p-1) + O(\log n) = O(p \log n)$$

Moral: Avoid using an $O(n)$ operation inside a loop when there is an $O(1)$ alternative!

Note that `Seq.append`, per se, is not necessarily always bad. You can imagine that there may be an implementation of sequences where append is cheap.

OK, so the cost of using cons vs append may be old hat. But there is a more subtle issue that arises when using append. Consider what happens when you are creating the tree and, instead of mapping each node to its parent, you want each node to map to the whole path to (or from) the root. In hw4 using append seems tempting, because you can keep the paths in the correct order, from the root down to each vertex, and now in `findPath` you don't need to reverse the path (or convert it to a sequence) and the work is $O(logn)$ independent of the path length!

Q: What is the problem besides the work cost?

Let's see what happens when keep the paths as linked lists (we'll only maintain the list of vertices) and we use either append or cons.

Recall that we are using a vTable implementation to maintain the tree.

Q: How much *space* will the two versions use?

A: Consider append first.

You're all seasoned C hackers by now, so let's think about this problem in terms of pointers into memory.

We have the following table:

$$\begin{aligned}
u &\mapsto (u \to nil) \\
u0 &\mapsto (u \to u0 \to nil) \\
u1 &\mapsto (u \to u1 \to nil) \\
u2 &\mapsto (u \to u1 \to u2 \to nil) \\
u3 &\mapsto (u \to u1 \to u3 \to nil) \\
v &\mapsto (u \to u1 \to u2 \to v \to nil)
\end{aligned}$$

Q: Notice that even though the paths for $u1$, $u2$, $u3$, v all start with $[u \to u1]$ they cannot share that subpath. Why?

A: The obvious answer is that we cannot have two arrows pointing out from a single linked list node. But even if there is no branching, we would have a problem. For example, when we add v to the tree, we cannot append a v to the path at $u2$ (even destructively) without making a copy of the list, because otherwise the list at $u2$ would also change.

With append, each vertex makes a *new copy* for its parent's path.

Therefore, the tree will use $O(n^2)$ space!

Q: What does the table look like if we use cons?

$$\begin{aligned} u &\mapsto (u \to nil) \\ u0 &\mapsto (u0 \to (u\texttt{'s list})) \\ u1 &\mapsto (u1 \to (u\texttt{'s list})) \\ u2 &\mapsto (u2 \to (u1\texttt{'s list})) \\ u3 &\mapsto (u3 \to (u1\texttt{'s list})) \\ v &\mapsto (v \to (u2\texttt{'s list})) \end{aligned}$$

This time each entry in the table is a single element that links to the linked list of its parent. For example, the linked list at $u2$ shares the linked list at $u1$ as does the linked list at $u3$. They all refer to the same linked list from $u1$ to u. The link structure truly is a tree with pointers toward the root. Each element points to its parent and many elements can point to the same parent.

Because "cons" is a *data constructor*, it doesn't compute something and make a copy – it has a pointer directly to that data.

Now the table uses only $O(n)$ space.

3 Set Cover

Q: Who remembers how the set cover problem is defined?

A:

- **Instance:** A "universe" U and a family set $\mathcal{F} \subseteq \mathcal{P}(U)$
- **Solution:** A subset $\mathcal{C}$ of $\mathcal{F}$ such that for all $x \in U$, there exists an $X \in \mathcal{C}$ such that $x \in X$.

Q: The above spec describes a **valid solution** to the problem. How would we describe an *optimal* solution?

A: If $\mathcal{C}$ is a *valid* solution, it is optimal iff for all valid solutions $\mathcal{C}'$, $|\mathcal{C}'| \geq |\mathcal{C}|$.

Q: What was our greedy algorithm from class?

A: Set up a bipartite graph with one side $= \mathcal{F}$ (each vertex is a set among the sets in a family), one side $= U$ (each vertex represents an element in the universe), and draw edges if a set contains an element. Each round, pick the set in $\mathcal{F}$ with the highest degree and remove it and its neighbors from the working graph; continue until U is empty.

Q: Does this create a valid solution?

A: Yes (by end condition).

Q: Why isn't it optimal? Can anyone see how to construct a counterexample?

Here's an instance of the set cover problem for which the greedy method doesn't find the optimal cover (draw this both as points and circles, and a bipartite graph):

```
A ---> 1, 2, 3, 4
B ---> 3, 4, 5
C ---> 1, 2, 6
```

A has highest degree; after we remove it:

```
B ---> 5
C ---> 6
```

So we get $\mathcal{C} = \{A, B, C\}$ even though the optimal is $\{B, C\}$.

In fact, the set cover problem is **NP**-complete, so we shouldn't expect a simple greedy algorithm to solve it.

For a minimization problem (i.e., we try to minimize an objective), an *α-approximation algorithm* is one that comes up with answers that are at most $\alpha\times$ optimal. We claimed in class that this algorithm is a $(1 + \ln n)$-approximation; let's take a closer look.

The number of sets we wind up with at the end is the same as the number of "rounds" in the algorithm. This is a contraction algorithm; we make one recursive call per round on a smaller problem. The question is: how much smaller?

Let's suppose `opt` is the number of sets in the optimal cover. Note that the optimal algorithm does *not* have to proceed in rounds like our greedy algorithm. But the sets in an optimal solution have the property that, on average, a set covers $n/\texttt{opt}$ elements. This means that there is a set Y in this optimal solution that covers at least $n/\texttt{opt}$ elements. (This is an averaging argument you have seen before in 251.)

Notice that thus far, we haven't used any property about the greedy algorithm. Here's where the greedy nature of the algorithm comes in: Our algorithm picks the set with the highest degree (i.e., the largest coverage). This set that we pick necessarily covers more elements than Y; remember Y was also a candidate. Therefore, we have

$$R(n) \leq R(n - n/\texttt{opt}) + 1 \leq$$

which we know from the Karp-Upfal-Wigderson (KUW) lemma from last week (KUW) is at most $\texttt{opt} \cdot H_n$.

Therefore, the greedy algorithm computes an H_n-approximation.

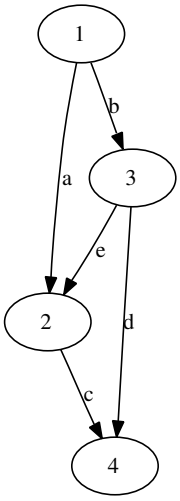
4 Matrices and Graphs

We saw in class that sparse matrices are the same as graphs. Let's see how we can use sparse matrices to solve a particular graph problem.

Recall the single-source shortest path (SSSP) problem:

- **Instance:** A weighted, directed graph $G = (V, E)$ and a source vertex $s \in V$
- **Solution:** A mapping from each vertex $v \in V$ to the distance of the shortest path from s to v

Here's an example to stare at while you think:



Q: If all the weights are non-negative, what algorithm can we use?

A: Dijkstra

Q: What is the inductive hypothesis for Dijkstra?

A: Given the SP distances to the k vertices nearest to s, find the distances of the $k+1$ vertices nearest to s. (Inherently sequential)

Q: When there are negative edge weights, what algorithm do we need to use?

A: Bellman-Ford

Q: Who remembers how Bellman-Ford works, at a high level? What is its inductive hypothesis?

A: We increase the length (number of edges) of the paths by at most one each iteration.

That is, given the SP distances to the vertices that have paths that are at most i "hops" from s, find the SP distances to the vertices that have paths that are at most $i+1$ hops from s. For comparison, BFS/Dijkstra do not revisit vertices, whereas Bellman-Ford does.

Let $\mathbf{d}_v^i = \delta^i(s,v), \forall v \in V$ be the shortest path distance from s to v that uses at most i edges. This could be infinity if v is not reachable from s within i edges.

Q: How do we find $\mathbf{d}_v^{i+1}, \forall v \in V$?

A: Consider $v \in V$. We need to add one more edge to paths of length i so that we get to v, and then find the minimum of these.

Q: Which vertices do we need to consider?

A: The in-neighbors of v. Then

$$\mathbf{d}_v^{i+1} = \min_{u \in N_{\mathbf{G}}^-(v)} \{\mathbf{d}_u^i + w(u,v)\}$$

where $N_G^-(v)$ is the set of in-neighbors of v in the graph G.

Hmm. But getting in-neighbors of v is awkward, as we often represent a graph as a table of out-neighbors of u.

Q: How should we represent the graph using tables?

A: Use `transpose`(G), fond memories of Exam 1.

Now let's consider how we can use a matrix representation of graph to implement the Bellman-Ford algorithm. Recall that a matrix is essentially the type `'a seq seq` where we will consider each (inner) sequence as a row of the matrix.

Q: First, how would we represent the graph above as a dense matrix? We'll later use a sparse matrix representation.

A: For $|V| = n$, make an $n \times n$ matrix $\mathbf{G}$ such that $\mathbf{G}_{uv} = w(u, v)$ if defined and ∞ otherwise.

```
  | 1    2    3    4
--------------------
1| ?    a    b   inf
 |
2| inf  ?    inf  c
 |
3| inf  e    ?    d
 |
4| inf  inf inf  ?
```

Q: What should we put for $\mathbf{G}_{vv}$?

A: It depends on the problem we want to solve!

First, let's look at how Bellman-Ford works, when we are given the graph $\mathbf{G}$ as a matrix. As before we'll use $\mathbf{G}^T$, the transpose of $\mathbf{G}$. Let $\mathbf{d}^i$ be a vector of shortest path distances of length i from s.

Q: What operation computes $\mathbf{d}_v i + 1$ in terms of our matrix $\mathbf{G}^T$ and vector $\mathbf{d}$?

$$\mathbf{d}_v^{i+1} = \min_{u \in N_G^-(v)} \{\mathbf{d}_u^i + w(u, v)\}$$

A: It is the dot product of row $\mathbf{G}_v^T$ and $\mathbf{d}^i$, where instead of summing the products, we take the min of sums. (Remember, $\mathbf{G}_{vu}^T = w(u, v)$ and the row $\mathbf{G}_v^T$ has edge weights for the in-neighbors of v.)

Q: How do we find $\mathbf{d}^{i+1}$?

$$\mathbf{d}^{i+1} = \texttt{mvmult}(\mathbf{G}^T, \mathbf{d}^i)$$

Q: How should we initialize the $\mathbf{d}^0$ vector?

A: In our example,

```
< 0 inf inf inf >
```

Q: What should the matrix diagonals $\mathbf{G}_{ii}$ be for this problem?

A: 0. If we made them ∞, we would capture shortest paths of *exactly* i hops; 0 lets us stay where we are for free, capturing "at most" i hops as needed.

Let's see what $\mathbf{G}^T$ looks like for our example graph:

```
| 0 inf inf inf|   | 0 |   | min(0,inf,inf,inf)  |
| a  0   e  inf| * |inf| = | min(a, a, b+e, inf) |
| b inf  0  inf|   |inf|   | min(b, inf,b, inf)  |
|inf c   d   0 |   |inf|   | min(inf,a+c,d+b,inf)|

= <0, min(a,b+e), b, min(a+c,d+b)>
```

4.1 ML Code

```
fun BF'(G, s) =
  let
    val d = tabulate (fn i=> if i=s then 0 else inf) (numRows G)
    val G' = transpose(G)
  in
    iter (fn (d,i) => mvmult (G', d)) d d
  end
```

4.2 Cost

Q: What's the cost of this algorithm? A: $O(n^3)$

Q: Can we do better? A: Yes, if we use sparse matrix for $\mathbf{G}$.

Q: That begs the question, what are the "zeros" of $\mathbf{G}$?

A: The ∞ values since $\min(\infty, a) = \min(a, \infty) = a$.

Q: How many non-zeros are in $\mathbf{G}$?

A: $|E| + |V|$ for the $w(u, v)$ entries and the diagonal. If we use a compressed sparse row representation for $\mathbf{G}$ using an array implementation of sequences, then a sparse matrix vector multiplication takes $O(nz(\mathbf{G}))$ work and $O(log|V|)$ span.

Since we iterate $|V|$ times, the cost of Bellman-Ford is $O(|V||E|)$ work and $O(|V|log|V|)$ span, which is the same bounds as using `vTable` representation with an array sequences implementation.