Recitation 13

Hashing

13.1 Announcements

- *DPLab* has been released, and is due **Monday, April 18**. It's worth 140 points.
- *PASLLab* will be released on Monday, April 18.

13.2 Removing Duplicates

Removing duplicates is a crucial substep of many interesting algorithms. For example, in BFS, consider the step where we construct a new frontier. One viable method would to be to generate the sequence of all out-neighbors, and then remove duplicates:

$$F' = \texttt{removeDuplicates}\ \left\langle v : u \in F, v \in N_G^+(u) \right\rangle$$

So, how fast is it to remove duplicates? Can we do it in parallel?

13.2.1 Sequential

Before we think about parallelism, we should acquaint ourselves with a good sequential algorithm solving the same problem. This way, we know what to shoot for in terms of work bounds, since we want our parallel algorithm to be asymptotically work-efficient.

Task 13.1. *Describe a sequential algorithm which performs expected $O(n)$ work to remove duplicates from a sequence of length n. Also argue that $\Omega(n)$ work is necessary in order to solve this problem, and conclude that your algorithm is asymptotically optimal.*

Hint: try hashing elements one at a time.

We can iterate left-to-right across the sequence, maintaining a set of all elements seen so far. At each element, we check to see if it's present in the set. If it is, we ignore it. If it isn't, we insert it into the set and also write it to the output. Using a hash set, we can check for membership and do insertions both in expected constant time. So, this algorithm has expected $O(n)$ work.

Removing duplicates requires at least $\Omega(n)$ work because we have to inspect every element. Here's a sketch of a proof by contradiction: suppose there was an algorithm for removing duplicates which used only $o(n)$ work. Then there must be at least one element of the input which the algorithm did not inspect. If the algorithm would blindly include this element in the output, then we can adversarily choose that element to be a duplicate. If instead the algorithm blindly excludes the element from the output, then we can adversarily choose the element to be distinct. Thus clearly the given algorithm does not properly remove duplicates.

Since $\Omega(n)$ work is necessary and our algorithm has $O(n)$ work, we know it is asymptotically optimal in terms of runtime.

13.2.2 Parallel

Task 13.2. *Implement a function*

$$\textbf{val}\ \mathtt{removeDuplicates}\ :\ (\alpha \times \mathtt{int} \to \mathtt{int}) \to \alpha\ \mathtt{Seq.t} \to \alpha\ \mathtt{Seq.t}$$

where $(\mathtt{removeDuplicates}\ h\ S)$ *retuns a sequence of all unique elements of* S*, given that* $h(e, m)$ *hashes the element* e *to a uniform random integer in the range* $[0, m)$ *(thus the probability of collision for any two distinct elements is* $1/m$*).*

Hint: as a first attempt, try simultaneously hashing as many elements as possible all at the same time. What do you do when elements collide?

We can use `inject` to simultaneously insert as many keys as possible into an initially empty hash table T of some size $m > n$ (we'll decide on a value for m later). Specifically, for every $0 \le i < |S|$, we attempt to insert the pair $(i, S[i])$ at the location $T[h(S[i], m)]$.

Every pair $(i, S[i])$ then compares itself against the value $(j, S[j])$ stored at $T[h(S[i], m)]$. If $i = j$, then $S[i]$ is "accepted," and will be included in the output. Otherwise, if $S[i] \ne S[j]$, then $S[i]$ is passed on to be retried in the next round. We continue retrying until no elements remain.

Why is this algorithm correct? Consider some key k; we never discard k until k is accepted, so we only need to argue that we never accept the same key twice. Consider some round and two indices $i \ne j$ such that $S[i] = S[j]$. If $S[i]$ is accepted then $S[j]$ won't be (since $i \ne j$), and furthermore it won't be retried on the next round. Therefore $S[j]$ will never be accepted.

How many elements are retried each round? Consider:

$$\begin{aligned}\textbf{Pr}\left[S[i] \text{ is retried}\right] &= \textbf{Pr}\left[\exists j. S[i] \ne S[j] \wedge h(S[i], m) = h(S[j], m)\right] \\ &\le \sum_{S[i] \ne S[j]} \textbf{Pr}\left[h(S[i], m) = h(S[j], m)\right] \\ &\le \frac{|S|}{m}\end{aligned}$$

If we chose $m = 3|S|/2$, then $|S|/m = 2/3$, and by linearity of expectation, we have that the number of retried elements is at most $2|S|/3$ in expectation.

Since we need $O(|S|)$ work and $O(\log |S|)$ span on each round, we expect a logarithmic number of rounds with a geometrically decreasing input. We've seen such recurrences before; they solve to expected linear work and log-squared span.

Algorithm 13.3. *Removing duplicates with hashing.*

```
fun removeDuplicates $S$ =
   if $|S| = 0$ then $\langle\rangle$ else
   let
      val $E$ = Seq.enum $S$

      val $m$ = $\frac{3|S|}{2}$
      val base = $\langle \mathit{NONE} : 0 \le i < m \rangle$
      val updates = $\langle (h(k,m),\ \mathit{SOME}(i,k)) : (i,k) \in E \rangle$
      val $T$ = Seq.inject (base, updates)

      fun accept $(i,k)$ = $\big(T[h(k,m)] = \mathit{SOME}\,(i,k)\big)$
      val $A$ = $\langle k : (i,k) \in E \mid \mathit{accept}(i,k) \rangle$

      fun retry $k$ = let val $\mathit{SOME}(\_, k')$ = $T[h(k,m)]$ in $k \ne k'$ end
   in
      Seq.append ($A$, removeDuplicates $\langle k \in S \mid \mathit{retry}(k) \rangle$)
   end
```