Recitation 8

Graphs and BFS

8.1 Announcements

- *RangeLab* has been released, and is due **Thursday night**. It’s worth 125 points.
- *BridgeLab* will be released on **Thursday**.
8.2 Graph Representations

**Task 8.1.** Write the representation of the following graph

1. as an **edge set**
   (use the pair \((x, y)\) to indicate a directed edge from \(x\) to \(y\)),
2. as an **adjacency table**, and
3. as an **adjacency sequence**.

![Graph Diagram]

**Task 8.2.** Implement the function

\[
\text{val adjTable} : (\text{vertex} \times \text{vertex}) \text{ Table.Seq.t} \rightarrow \text{Table.Set.t Table.t}
\]

where \((\text{adjTable } S)\) converts the “edge set” \(S\) into an adjacency table.\(^a\) Analyze the work and span of your implementation, assuming tables/sets implemented as treaps.

Assume Table ascribes to TABLE where type Key.t = vertex.

\(^a\)In this context, we represent an “edge set” simply as an unordered sequence of directed edges.
8.3 BFS

8.3.1 An Example

**Task 8.3.** Run BFS on the example graph from the previous section, starting at vertex 1. Draw the resulting BFS tree. Draw tree edges as solid lines and non-tree edges as dashed lines.

8.3.2 Implementation

Consider the following code, which computes the BFS tree of an enumerated graph represented by an adjacency sequence. For brevity, we’ll write None as □ and (SOME x) as x.

**Algorithm 8.4.** Computing BFS trees on adjacency sequences.

```
fun BFS (G, s) = 
  let 
    fun BFS' (X, F) = 
      if |F| = 0 then STSeq.toSeq X else 
      let 
        val N = 
          Seq.flatten ⟨⟨(u, v) : u ∈ G[v] | X[u] = □⟩ : v ∈ F⟩ 
        val X' = STSeq.inject (X, N) 
        val F' = ⟨u : (u, v) ∈ N | X'[u] = v⟩ 
      in 
        BFS' (X', F') 
      end 
    in 
      BFS' (X0, F0) 
  end
```

**Task 8.5.** Execute this code on the example graph given in the first section, starting with vertex 1 as the source. Trace the process by writing down the values X_i, F_i, and N_i for i = 0, 1, 2, 3.

**Task 8.6.** Analyze the work and span of this implementation in terms of n (the number of vertices), m (the number of edges), and d (the diameter of the graph).
8.4 Bonus: Single-Threaded Sequences

A single-threaded sequence is basically an ephemeral sequence wrapped up in a (seemingly) purely functional interface. By “ephemeral”, we mean the opposite of “persistent”: ephemeral data structures allow destructive modifications to their contents. For example, imagine an array. When we call STSeq.update or STSeq.inject, we are destructively modifying this array. This is why STSeq.update and STSeq.inject are so fast: there is no need to copy an entire sequence.

Now, consider the following code.

1 val S = ⟨i : 0 ≤ i < n⟩
2 val S0 = STSeq.fromSeq S
3 val S1 = STSeq.update (S0, (0, 42))
4 val S2 = STSeq.update (S1, (1, 43))
5 val S3 = STSeq.update (S1, (2, 44))

On lines 3 and 4, we destructively modify $S_0$ to create $S_1$, then destructively modify $S_1$ in order to create $S_2$. So, what happens on line 5, when we attempt to perform another update on $S_1$, which is currently an “old” version?

In this scenario, in order to appear persistent, $S_1$ has to rebuild itself. That is, it has to replay every update which happened since the “origin,” which in this case is line 2. Note that this could be arbitrarily expensive!

From the perspective of ensuring certain cost bounds, the “correct” usage of a single-threaded sequence is identical to how one would use an array: that is, you can only operate on the most recent version.

We call these sequences “single-threaded” since they should only be modified by a single thread at a time. For example, the following code has a nasty race condition!

1 val S = STSeq.fromSeq ⟨i : 0 ≤ i < n⟩
2 val (A, B) = (STSeq.update (S, (0, 42)) || STSeq.update (S, (0, 43)))

In summary,

- Single-threaded sequences are essentially arrays which have been made persistent.
- It is cheap to modify the most recent version of an st-sequence. (Updates are constant-time, injections are linear in the number of updates.)
- It is expensive to modify old versions of an st-sequence. In general, using an old version which is $i$ steps away from its origin will require an additional $\Omega(i)$ work and span. For the sake of cost analysis, you should never modify an old version.
- A single-threaded sequence should never be modified by two parallel threads.