Recitation 8

Graphs and BFS

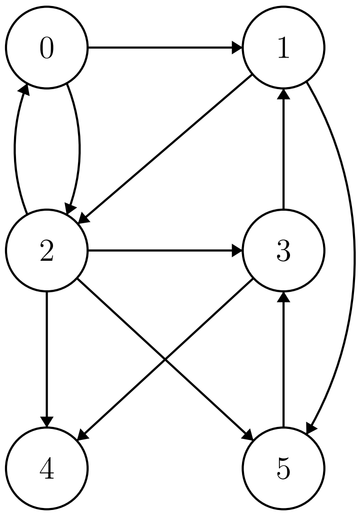
8.1 Announcements

- *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****.*



Edge Set:

$$\big\{(0,1),(0,2),(1,2),(1,5),(2,0),(2,3)\\ (2,4),(2,5),(3,1),(3,4),(5,3)\big\}$$

Adjacency Table:

$$\big\{0 \mapsto \{1,2\}, 1 \mapsto \{2,5\}, 2 \mapsto \{0,3,4,5\},\\ 3 \mapsto \{1,4\}, 4 \mapsto \{\}, 5 \mapsto \{3\}\big\}$$

Adjacency Sequence:

$$\big\langle \langle 1,2\rangle, \langle 2,5\rangle, \langle 0,3,4,5\rangle, \langle 1,4\rangle, \langle\rangle, \langle 3\rangle \big\rangle$$

Task 8.2. *Implement the function*

```
val adjTable : (vertex * vertex) Table.Seq.t
             → Table.Set.t Table.t
```

where (`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.

Algorithm 8.3. *Constructing an adjacency table.*

```
fun adjTable S =
  Table.map Table.Set.fromSeq (Table.collect S)
```

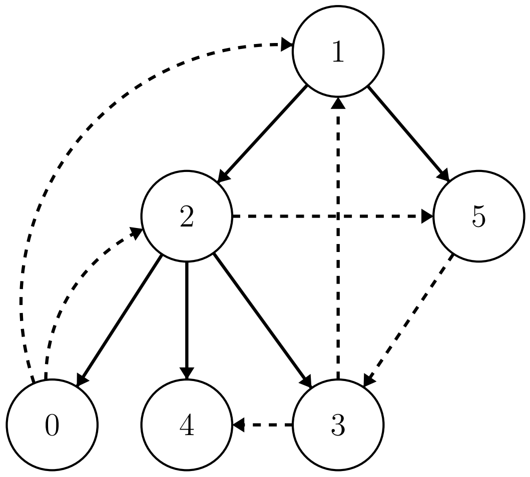
The `Table.collect` incurs a cost of $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span. The `Table.map Table.Set.fromSeq` incurs a work cost of $O(\sum_i |S_i| \log |S_i|)$ where S_i is the sequence of neighbors of the i^{th} vertex, therefore $|S_i| \leq |S|$ and $|S| = \sum_i |S_i|$. That gives us $O(\sum_i |S_i| \log |S_i|) = O(|S| \log |S|)$ work. The span is clearly $O(\log^2 |S|)$.

Hence the work and span of (`adjTable` S) are $O(|S| \log |S|)$ and $O(\log^2 |S|)$, respectively.

8.3 BFS

8.3.1 An Example

Task 8.4. *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.*



Note that we could have chosen $(5, 3)$ as a tree edge instead of $(2, 3)$. Either edge is valid; as long as we don't choose *both* as tree edges, we're golden!

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 $\boxed{}$ and (`SOME` x) as $\boxed{x}$.

Algorithm 8.5. *Computing BFS trees on adjacency sequences.*

```
fun BFS $(G, s)$ =
  let
    fun BFS' $(X_i, F_i)$ =
      if $|F_i| = 0$ then STSeq.toSeq $X_i$ else
      let
        val $N_i$ =
          Seq.flatten $\langle\langle(u, \boxed{v}) : u \in G[v] \mid X_i[u] = \boxed{\phantom{x}}\rangle : v \in F_i\rangle$
        val $X_{i+1}$ = STSeq.inject $(X_i, N_i)$
        val $F_{i+1}$ = $\langle u : (u, v) \in N_i \mid X_{i+1}[u] = \boxed{v}\rangle$
      in
        BFS' $(X_{i+1}, F_{i+1})$
      end

    val init = STSeq.fromSeq $\langle\boxed{\phantom{x}} : 0 \le i < |G|\rangle$
    val $X_0$ = STSeq.update (init, $(s, \boxed{s})$)
    val $F_0$ = $\langle s\rangle$
  in
    BFS' $(X_0, F_0)$
  end
```

Task 8.6. *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 $i = 0, 1, 2, 3$.*

i	X_i	F_i	N_i
0	$\langle\boxed{}, \boxed{1}, \boxed{}, \boxed{}, \boxed{}, \boxed{}\rangle$	$\langle 1\rangle$	$\langle(2, \boxed{1}), (5, \boxed{1})\rangle$
1	$\langle\boxed{}, \boxed{1}, \boxed{1}, \boxed{}, \boxed{}, \boxed{1}\rangle$	$\langle 2, 5\rangle$	$\langle(0, \boxed{2}), (4, \boxed{2}), (3, \boxed{2}), (3, \boxed{5})\rangle$
2	$\langle\boxed{2}, \boxed{1}, \boxed{1}, \boxed{5}, \boxed{2}, \boxed{1}\rangle$	$\langle 0, 4, 3\rangle$	$\langle\rangle$
3	$\langle\boxed{2}, \boxed{1}, \boxed{1}, \boxed{5}, \boxed{2}, \boxed{1}\rangle$	$\langle\rangle$	*(nonexistent)*

Task 8.7. *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).*

Let's break down the code, line-by-line. We write $||F|| = \sum_{v \in F}(1 + d_G^+(v))$.

- Line 7: $O(||F_i||)$ work, $O(\log n)$ span.
- Line 8: $O(||F_i||)$ work, $O(1)$ span.
- Line 9: $O(||F_i||)$ work, $O(\log n)$ span.
- Line 14: $O(n)$ work, $O(1)$ span.
- Lines 15,16: $O(1)$ work, $O(1)$ span.

There are two important observations to make here:

1. no vertex is ever in a frontier more than once, and
2. the number of rounds of BFS is upper bounded by $d + 1$. (There could be a vertex d hops away from the source, and each round progresses by exactly one hop. The "+1" comes from the final round which verifies that the frontier is empty, then exits).

We can now show that

$$\sum_{i=0}^{d} ||F_i|| \leq \sum_{v} (1 + d_G^+(v)) = n + m.$$

Therefore the total work is

$$O\left(n + \sum_{i=0}^{d-1} ||F_i||\right) = O(n + m)$$

and the span is $O(d \log n)$.

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 $S$ = $\langle i : 0 \leq i < n \rangle$
2 val $S_0$ = STSeq.fromSeq $S$
3 val $S_1$ = STSeq.update ($S_0$, (0, 42))
4 val $S_2$ = STSeq.update ($S_1$, (1, 43))
5 val $S_3$ = STSeq.update ($S_1$, (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!

```
val $S$ = STSeq.fromSeq $\langle i : 0 \leq i < n \rangle$
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.