Recitation 9

Graph Search: BFS and DFS

9.1 Announcements

- *BridgeLab* has been released, and is worth 140 points. The due dates are a bit wonky because of Spring Break: the written section is due at Friday at 5pm, while the programming portion is due Sunday night.

- *ShortLab* will be released on **Friday**.
9.2 DFS Trees and Numberings

Task 9.1. Starting at vertex 1, execute DFS on the following graph, visiting vertices in increasing order. Trace the process by doing each of the following.

1. Draw the resulting DFS tree. Draw tree edges as solid lines, and include non tree edges in your drawing as dashed lines.
2. Classify each non tree edge as one of forward, back, or cross.
3. Label each vertex with its discovery and finish times.

Task 9.2. Suppose DFS is run on a directed graph, and consider some edge $(x, y)$. Using the discovery and finish times of $x$ and $y$, attempt to classify this edge as one of tree, forward, back, or cross.
9.2.1 Higher-Order DFS

Recall the following code from the textbook:

Algorithm 9.3. Directed, generalized DFS.

```
1 directedDFS (revisit, discover, finish) (G, Σ₀, s) =
2   let
3       DFS p ((X, Σ), v) =
4       if (v ∈ X) then (X, revisit (Σ, v, p)) else
5         let
6             Σ' = discover (Σ, v, p)
7             X' = X ∪ {v}
8             (X'', Σ'') = iterate (DFS v) (X', Σ') (N⁺_G(v))
9             Σ''' = finish (Σ', Σ'', v, p)
10        in
11        (X'', Σ''')
12      end
13    in
14      DFS s (({}, Σ₀), s)
15  end
```
9.3  BFS

9.3.1  An Example

Task 9.6. Run BFS on the example graph above, starting at vertex 1. Draw the resulting BFS tree. Draw tree edges as solid lines and non-tree edges as dashed lines.
9.3.2 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 9.7. Computing BFS trees on adjacency sequences.**

1. `fun BFS (G, s) =`
2. `let`
3. `fun BFS' (X_i, F_i) =`
4. `if |F_i| = 0 then STSeq.toSeq X_i else`
5. `let`
6. `val N_i = Seq.flatten ⟨(u, v) : u ∈ G[v] | X_i[u] = []⟩ : v ∈ F_i⟩`
7. `val X_{i+1} = STSeq.inject (X_i, N_i)`
8. `val F_{i+1} = {u : (u, v) ∈ N_i | X_{i+1}[u] = [v]}`
9. `in`
10. `BFS' (X_{i+1}, F_{i+1})`
11. `end`
12. `end`
13. `val init = STSeq.fromSeq ⟨[] : 0 ≤ i < |G|⟩`
14. `val X_0 = STSeq.update (init, (s, [s]))`
15. `val F_0 = ⟨s⟩`
16. `in`
17. `BFS' (X_0, F_0)`
18. `end`

**Task 9.8.** 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 9.9.** 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).