Lecture 6 Notes (March 3, 2025)

Sam Westrick

`s.westrick@nyu.edu`

Graphs

Graphs are commonly used to model relationships between people, objects, places, etc.

A few examples:

- **Friend graph (social network)**: vertices represent people, and edges indicate that two people are friends. This is an example of a **symmetric** relationship, commonly modeled by an "undirected" edge.
- **Follower graph (social network)**: vertices represent people, and an edge $p_1 \to p_2$ indicates that person p_1 follows p_2, e.g., on social media. This is an example of an **asymmetric** relationship, commonly modeled by a "directed" edge.
- **Road network**: vertices represent places, and edges represent roads. Depending on the circumstances, the edges might be either undirected or directed.

Depending on the specific relationship being modeled, we might also associate *weights* with vertices and/or edges. For example, an edge that represents a road might be annotated with a distance; a vertex that represents a city might be annotated with its population.

You are likely familiar with a variety of sequential algorithms on graphs, such as sequential breadth-first and depth-first search, Kruskal's minimum spanning tree algorithm, Dijkstra's shortest path algorithm, A* search, etc.

These algorithms can be used to implement a variety of interesting queries. For example, on a follower graph, we might be interested in identifying tight-knit groups of people where every person in the group both (1) follows someone else in the group, and (2) is followed-by someone else in the group. Such groups are commonly called *strongly connected components* and these can easily be identified with a particular flavor of depth-first search.

Many real-world graphs are quite large, however, which make sequential computation impractically slow. In this lecture, we will discuss techniques for implementing **parallel** graph algorithms.

Efficient Memory Representations for Graphs

Perhaps the simplest representation for a graph is an **adjacency matrix**. This is a square matrix, with one row and one column for every vertex. An element at position (i, j) in the matrix encodes information about an edge between vertices i and j. This could be a boolean (indicating the presence or lack of an edge), or a weight, etc.

For a graph with N vertices, adjacency matrices require $O(N^2)$ space to store in memory.

However, in practice, many real-world graphs are highly **sparse**, meaning that the total number of edges in the graph is relatively small. For example, in a friendship graph, each individual person might be friends with a few hundred (or even a few thousand) people, but this is approximately 7 orders of magnitude smaller than the total number of people in the world (8.2 billion, as of March 2025).

Let M be the number of edges, and N the number of vertices. Often, we say that a graph is sparse if $M \leq c \cdot N$ for some reasonably small constant c. For sparse graphs, adjacency matrices are incredibly wasteful representations, requiring $O(N^2)$ space instead of $O(N)$.

Therefore, we often use **sparse representations** to store graphs in memory. Such representations only require $O(N + M)$ space.

One such sparse representation is the **adjacency list** or **adjacency sequence**, where for each vertex we only store its immediate neighbors. (If there is an edge between u and v, we say that u and v are neighbors.)

Vertices are commonly given the names 0 to $N - 1$, and then we can store the adjacency sequence simply as a (jagged) 2-dimensional sequence. For example, the adjacency sequence $\langle\langle 1, 2\rangle, \langle 3\rangle, \langle\rangle, \langle 0, 2\rangle\rangle$ represents a graph with 4 vertices (numbered 0 through 3) with the following directed edges: $0 \to 1, 0 \to 2, 1 \to 3, 3 \to 2, 3 \to 0, 3 \to 2$.

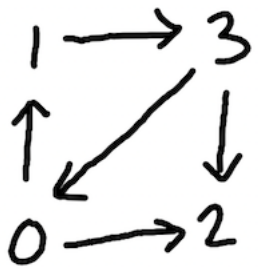

Figure 1: Example graph

Adjacency sequences have an outer sequence of length N, and the sum of the lengths of the inner sequences comes out to M in total, so the total space requirement is $O(N + M)$. Also, notice that we can query the neighbors of any vertex in constant time.

Compressed Sparse Rows (or Columns)

Another common representation for graphs is called the **compressed sparse row** representation, or sometimes **compressed sparse column**. (The analogy to rows and columns comes from the adjacency matrix interpretation.) These are abbreviated **CSR** and **CSC**.

Essentially, the CSR representation is the same as the adjacency sequence, except the (inner) neighbor sequences have been flattened into a single 1D sequence of length M. This is called the edge array, or neighbor array. We then separately store the offset of where the neighbors for each vertex begin. This offsets array is length N (or sometimes $N + 1$, i.e., all prefix sums over the neighbor lengths, which is convenient for computing the degrees of vertices).

For example, the adjacency sequence $\langle\langle 1, 2\rangle, \langle 3\rangle, \langle\rangle, \langle 0, 2\rangle\rangle$ can be compressed into the following CSR representation. Here we store $N + 1$ offsets; notice that the last offset value is equal to the number of edges.

edges: $\langle 1, 2, 3, 0, 2\rangle$

offsets: $\langle 0, 2, 3, 3, 5\rangle$

In this representation, we can still query the neighbors of a vertex in constant time: the neighbors of a vertex v are just the subsequence edges$[i...j]$ where $i =$ offsets$[v]$ and $j =$ offsets$[v + 1]$.

One warning: the difference between **CSR** and **CSC** is not important conceptually, but sometimes in practice these are accidentally used interchangeably, which is a common source of bugs. Getting the two mixed up might result in accidentally swapping the direction of all of the edges in a directed graph.

Edge List (a.k.a. Edge Set)

It is important to mention one more sparse graph representation that you might commonly see in practice: the edge list or edge set. The idea is simply to write down a list of all of the edges. This might be sorted or unsorted. Clearly, this representation requires only $O(M)$ space.

Including Weights

In edge-weighted graphs (where each edge is annotated with a weight), we can replace every edge with a tuple that includes the weight.

Summary of Sparse Representations

Here are a few possible sparse representations that you might commonly see or might like to use in practice.

	Unweighted	Weighted

adjacency sequence	`type v = int (* vertices *)`
	`type graph = v seq seq`
		`type v = int (* vertices *)`
		`type w = ... (* weights *)`
		`type graph = (v * w) seq seq`

CSR/CSC	`type v = int (* vertices *)`
	`type graph = v seq * int seq`
		`type v = int (* vertices *)`
		`type w = ... (* weights *)`
		`type graph = (v * w) seq * int seq`

edge set	`type v = ... (* vertices *)`
	`type graph = (v * v) seq`
		`type v = ... (* vertices *)`
		`type w = ... (* weights *)`
		`type graph = (v * v * w) seq`

Connections between Matrices and Graphs

Often, algorithms on graph can be re-interpreted as algorithms on matrices, and vice-versa. The idea is to imagine the graph as an adjacency matrix, regardless of whether or not the graph is physically stored as one.

For example, in a directed weighted graph, suppose we wanted to compute the average weight of every vertex's neighbors.

Here is one possible graph-based implementation, with a graph represented as an adjacency sequence.

```
type vertex = int
type weight = real
type graph = (vertex *real) seq seq (* adjacency sequence *)

fun average_neighbor_weights (G: graph) : real seq =
  Parallel.tabulate (0, Seq.length G) (fn u =>
    let
      val u_nbrs = Seq.nth G u
      val w_total = Seq.reduce (op+) 0.0 (Seq.map (fn (v, w) => w) u_nbrs)
    in
      w_total / Real.fromInt (Seq.length u_nbrs)
    end)
```

But we could have equivalently computed the same result by converting the graph G to a matrix M_G where

$$M_G[u,v] = \begin{cases} \text{weight}(u,v) & \text{if } (u,v) \text{ is an edge in } G \\ 0 & \text{otherwise} \end{cases}$$

and then multiplying M_G^{T} with a vector d^{-1} where $d^{-1}[v] = 1/\text{degree}(v)$, where the **degree** of a vertex is its number of neighbors.

Parallel Breadth-First Search

Let's now develop a particular parallel algorithm, specifically a parallel implementation of breadth-first search.

You may already be familiar with sequential breadth-first search using a queue, where we iteratively (1) dequeue a vertex v, (2) check if v has already been visited, and if not, (3) "visit" v and then enqueue all of its neighbors. This algorithm is entirely sequential.

A parallel breadth-first search can visit many vertices all at the same time. The idea is, on round i, to visit *all* vertices that are i edge-hops away from the source vertex.

An implementation is shown below. This particular implementation outputs a sequence of "parents", where we keep track of which edge was used to visit each vertex. In particular, if edge (u,v) is used to visit v, then we set the parent of v to be u. Note that there can be many possible parents of a vertex; this implementation picks one arbitrarily.

The code operates in rounds by calling the recursive function `loop(P,F)`, where:

- P is the current sequence of parents. We mark unvisited vertices v with $P[v] = -1$. All visited vertices will have their parent set to be another vertex. Initially, all vertices are unvisited except for the source. By convention, we'll set the parent of the source vertex to be itself.
- F is the current "frontier" of vertices that were visited on the previous round.

The `loop` continues until the frontier F is empty. On each round, we begin by constructing a sequence of "backwards edges" called `BENV`, where for each edge $u \in F$ and each edge (u,v), we put the backwards edge (v,u) in `BENV` if v has not yet been visited.

Next, we "inject" the sequence `BENV` into the array of parents, producing a new array of parents P' for the next round. For each tuple (v,u) in `BENV`, this updates $P[v]$ with the value u. (In other words, "inject" is just a bulk update on a sequence.) This effectively selects a parent for each vertex v, choosing arbitrarily if there are multiple choices (v,u_1) and (v,u_2) in `BENV`. Details of the `Seq.inject` function are described in more detail below.

Finally, we do one more pass over `BENV` to produce a sequence of vertices v which were visited on this round; this is the next frontier. Note that on this step, we could have just done `Seq.map (fn (v, u) => v) BENV`, but this would result in duplicates if there are multiple backwards edges (v,u_1) and (v,u_2) for the same vertex v. Instead, we first filter the sequence to select only the backwards edges that were implicitly selected by the injection. That is, we keep (v,u) if $P'[v] = u$. This will ensure that the new frontier has no duplicates.

```
type vertex = int
type graph = vertex seq seq

(* breadth-first search of G starting with source vertex s *)
fun bfs(G: graph, s: vertex) : vertex seq =
let
  fun loop(P: vertex seq, F: vertex seq) =
    if Seq.length F = 0 then P else
    let
      fun benv_from(u) =
        Seq.map (fn v => (v, u))
          (Seq.filter (fn v => Seq.nth P v = ~1) (Seq.nth G u))
      val BENV: (vertex * vertex) seq =
        Seq.flatten (Seq.map benv_from F)
      val P' = Seq.inject BENV P
      val F' = Seq.map (fn (v, _) => v)
        (Seq.filter (fn (v, u) => Seq.nth P' v = u) BENV)
    in
      loop(P', F')
    end
  (* initial parents P and frontier F, for just the source vertex *)
  val init_P = Parallel.tabulate (0, Seq.length G) (fn v =>
    if v = s then s else ~1)
  val init_F = Seq.singleton s
in
  loop(init_P, initF)
end
```

Understanding `Seq.inject`

The function `Seq.inject` takes a sequence of tuples `(i, x)` and a underyling sequence `S` and performs the bulk updates `S[i] := x` for each tuple `(i,x)`, all in parallel. This is also sometimes called a **scatter**. The updated sequence is returned as a result.

```
val inject: (int * 'a) -> 'a seq -> 'a seq
```

In implementing `inject U S`, there are a few interesting questions to consider:

1. How does `inject` handle duplicates (two elements (i, x_1) and (i, x_2) in U with the same index i)?
2. Does `inject` modify the input S?

Let's consider the first question. So far in this course, we have focused on deterministic parallelism, and in particular, purely functional parallelism, where the output is always the same (regardless of the number of processors used, etc.) However, for practical efficiency of `inject`, we find ourselves in a situation where non-determinism may be desirable. We can implement `inject` by performing all updates concurrently; if there are many concurrent updates at the same index, then one of these updates will non-deterministically occur last, and this is the update that will be observable in the output. The alternative would be to perform another pass over the tuples (i, x) to deduplicate (for example, you could sort the tuples (i, x) by their indices i, and then deduplicate by removing adjacent tuples that have the same index.) However, this requires an additional pass over the data. So, perhaps the non-determinism of the more efficient implementation is acceptable.

Next let's consider the second question. If we didn't modify the input S, then the minimum possible work for `inject U S` would be $O(|S| + |U|)$, because we would have to copy S before updating the copy. However, in the case of the BFS above, as we will see below, we can obtain an asymptotic reduction in the total amount of work required for the BFS by instead directly modifying the input S and only performing $O(|U|)$ work, i.e., work linear in the number of updates performed.

Note in particular that, for `S' = inject U S`, as long as we never access `S` again and only instead access `S'`, then modifying `S` in-place is safe.

For the cost analysis below, we will use the cost specification where `inject U S` requires $O(|U|)$ work and $O(\log |U|)$ span.

BFS Cost Analysis

The total work for one round of BFS is, asymptotically:

$$||F|| \triangleq \sum_{u \in F} \left(1 + \sum_{v \in G[u]} 1 \right)$$

We can confirm this by stepping through the code and analyzing each of the calls to the various sequence functions. Intuitively, a constant amount of work is required for every edge (u, v) where $u \in F$; additionally, a constant amount of work is required per vertex $u \in F$. The quantity defined above, $||F||$, captures both of these costs. Note that $||F||$ is also an upper bound on the size of `BENV`, and this is the number of updates performed by the `inject`. The final line of each round, which constructs `F'`, also performs work linear in the size of `BENV`. Altogether, then, we have that the cost per round (on inputs P and F) is bounded by $O(||F||)$.

If F_i is the frontier at round i, then the total cost just sums over all rounds. Next, we can use a meta argument: we know that this implementation of BFS will "visit" every vertex at most once. In other words, for any vertex u there exists exactly one F_i such that $u \in F_i$. From this, it is clear that:

$$\text{work of BFS} \;=\; O\left(\sum_i ||F_i|| \right) = O(N + M)$$

.

We can similarly analyze the span. Each round of BFS requires $O(\text{polylog}(||F_i||))$ span, which can be upper bounded by $O(\text{polylog} M) = O(\text{polylog} N)$. If D is the number of rounds, then the total span is $O(D \text{ polylog} N)$.

An important point is that this quantity D will vary significantly by graph. Some graphs (such as a long chain) could have $D \approx N$. Other graphs, such as real-world social network graphs, typically have very small values of D. As long as D is reasonably small, this parallel implementation of BFS will have a significant amount of parallelism.

(The value D is closely related to a concept called the **diameter** of a graph, which is well-studied in graph theory.)