Breadth-First Search

Breadth-First Search (BFS) is another approach to graph search. It can solve some of the same problems that DFS can, but there are also some important problems that only BFS can solve (and some that only DFS can solve, meaning both are useful in their own scenarios).

We'll discuss BFS in the context of searching from a specific starting vertex s in a graph G, where the key most common application is finding *unweighted shortest paths from a given start vertex*.

1 "Classic" (Sequential) Breadth-First Search

Breadth-first search (BFS) explores a graph starting from a source vertex s. Unlike DFS, which visits vertices in a depth-first order, i.e., all reachable vertices from a vertex are visited before any unreachable ones, BFS explores the vertices in **distance order**. That is, it visits all vertices of distance i from the vertex s before it visits any vertices of distance $i+1$ and so on.

This makes it useful for computing **shortest paths**, which we will continue to study in the next few lectures. Specifically, in an unweighted graph, BFS computes the **shortest-path distance** $\delta_G(s, v)$ from s to every vertex v, where the distance is the minimum number of edges in a path from s to v. If v is unreachable from s, the distance is ∞.

The textbook standard algorithm for BFS is typically implemented using a FIFO (first in first out) queue. The queue starts with the source vertex s, and then visits vertices in their order that they are added to the queue—closer to s first, further away later.

When a vertex is visited, it adds its unvisited neighbors to the queue, much like how in DFS, when a vertex is visited, it recursively visits its unvisited neighbors. So, the algorithms operate much alike, the only difference being the order in which vertices are processed.

Algorithm: Sequential Breadth-First Search

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun bfs(adj : adjacency_list, s : int) -> sequence<int>:
  n = |adj|
  visited = [False for _ in range(n)]
  dist    = [∞     for _ in range(n)]

  Q = empty queue
  Q.enqueue(s)
  dist[s] ← 0
  visited[s] ← True
  while not Q.is_empty():
    u = Q.dequeue()
    for v in adj[u]:
      if not visited[v]:
        visited[v] ← True
        dist[v] ← dist[u] + 1
        Q.enqueue(v)

  return dist
```

Theorem: Cost of Classic BFS

On a graph with n vertices and m edges, "Classic" BFS costs $O(n+m)$ (work and span).

Note that this version of the algorithm is inherently sequential since it processes vertices from the queue one at a time, so there are very few opportunities for parallelism. In the next section, we will work towards modifying the algorithm to make it more parallelizable.

2 Layered Breadth-First Search

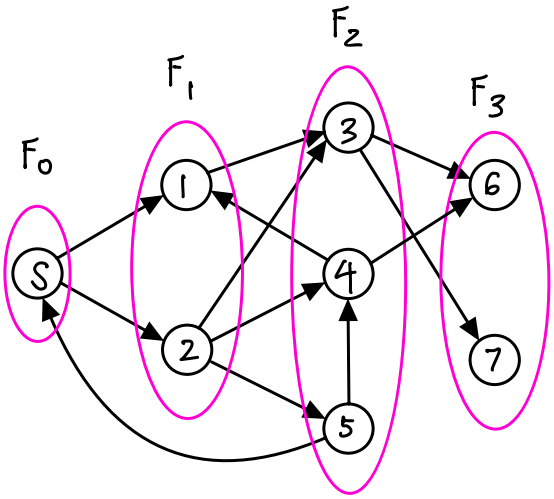
Although not so obvious from the classic implementation, conceptually, BFS constructs a sequence of *frontiers*

$$F_0, F_1, F_2, \ldots$$

where

- $F_0 = \{s\}$,
- F_i contains exactly the vertices at distance i from s.

Each iteration (or *round*) computes F_{i+1} from F_i. We can reorganize the computation to explicitly compute each frontier as a function of the previous frontier, rather than processing vertices from the queue one by one. This is effectively the same algorithm, but we are processing "in bulk" from the queue—instead of one by one, we process all of the vertices at distance i in bulk and obtain all the vertices of distance $i+1$.



This algorithm is still sequential, but it is much easier to parallelize, and we will make it parallel in the next section.

Algorithm: Layered Sequential Breadth-First Search

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun bfs(adj : adjacency_list, s: int) -> sequence<int>:
  n = |adj|
  visited = [False for _ in 0...n-1]
  dist =    [∞    for _ in 0...n-1]

  d = 1
  frontier = [s]
  dist[s] ← 0
  visited[s] ← True
  while |frontier| > 0:
    next_frontier = []
    for u in frontier:
      for v in adj[u]:
        if not visited[v]:
          visited[v] ← True
          dist[v] ← d
          next_frontier.append(v)
    d ← d+1
    frontier = next_frontier
  return dist
```

Theorem: Correctness of Layered BFS

The `dist` sequence computed by this algorithm satisfies: For all vertices v, if v is reachable from s then `dist[v]` is the length of the shortest path from s to v. If v is not reachable from s, then `dist[v]` $= \infty$.

Proof. The proof is by induction. The following conditions hold at the beginning of the loop:

1. `frontier` is a list of all vertices in G such that $\delta_G(s,v) = d-1$
2. for all v in G, `visited[v]` is true if and only if $\delta_G(s,v) \leq d-1$

These statements clearly hold at the beginning when $d = 1$, since `frontier` is `[s]` and `dist[s]` $= 0$. In the loop, any vertex which is a neighbor of a vertex in `frontier` that is not marked visited is marked visited, and its distance is set to d. We know that this assignment is correct, because there is an edge (u,v) in G with u in `frontier` so there is a path to v of length d. But there cannot be a shorter path from s to v because `visited[v]` was false.

Finally, this process is exhaustive, i.e., it finds all vertices at distance d from s. Because if a vertex v is distance d from s, then by virtue of the path from s to v, the vertex before v on that path must have distance $d-1$ to s. And the algorithm would have found it.

By the end of this pass `next_frontier` must contain all the vertices at distance d from s, and these vertices have all been appropriately marked. This completes the induction proof. It follows that at the end `dist[v]` $= \delta_G(s,v)$. □

Theorem: Cost of Layered BFS

On a graph with n vertices and m edges, Layered BFS costs $O(n+m)$ (work and span).

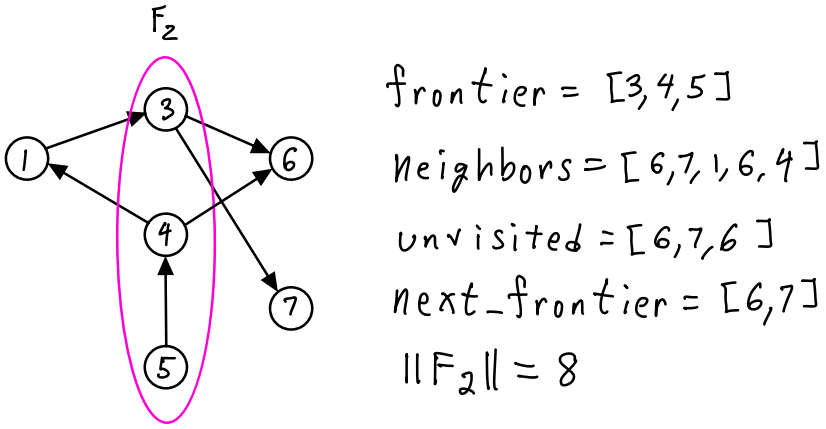
3 Imperative Parallel Breadth-First Search

One big advantage of the layered approach is that it is now much easier to see an opportunity for parallelism. Although each layer can only be computed after the previous layer has been computed, so there's no parallelism there, *within* each layer we can do all of the computations in parallel.

At a high level, the idea is that given a frontier, one can compute the next frontier by taking all of the neighbors of the current frontier that have not already been visited. This can be computed in parallel by flattening the sequences of all neighbors of the frontier and then filtering out those that have already been visited.

Algorithm: (Imperative) Parallel Breadth-First Search

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun bfs(adj : adjacency_list, s : int) -> sequence<int>:
  n = |adj|
  visited = [False for _ in range(n)]
  dist    = [∞     for _ in range(n)]

  dist[s] ← 0
  frontier = [s]
  d = 1
  while |frontier| > 0:
    neighbors = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => not visited[v], neighbors)
    next_frontier = unique(sort(unvisited))  // remove duplicates

    parallel for v in next_frontier:
      visited[v] ← True
      dist[v] ← d

    d += 1
    frontier = next_frontier

  return dist
```



The diagram above shows an example of what happens in the first three lines of the while loop above.

Notice that this version of the algorithm relies on mutation in parallel, so we must be careful! To avoid a data race, we need to make sure that `visited[v]` and `dist[v]` are not written multiple times in parallel. We ensure this by de-duplicating the frontier using `sort` plus `unique` (where `unique` removes *adjacent* duplicate elements).

For each vertex in the next frontier, we can then safely write `visited[v]` and `dist[v]` in parallel. Since the computation of the frontier itself is also highly parallel using parallel sequence algorithms, we have essentially fully parallelized each frontier computation.

Theorem: Cost of (Imperative) Parallel BFS

On a simple graph, imperative parallel BFS costs $O((m+n)\log n)$ work and $O(D \cdot \log^2 n)$ span, where D is the diameter of the graph (the maximum length of a shortest path).

Proof. Let F_i be the frontier of distance i. We will introduce the notation

$$\|F\| = \sum_{x \in F} (d_G^+(x) + 1),$$

to mean the "size" of the frontier. Note that this is more than just the number of vertices in the frontier, but rather it also counts the number of *outgoing edges*. This is important since when we process a frontier, we have to look at all of its outgoing edges to find the new frontier. So this quantity is essentially the total amount of "stuff" that gets processed during the computation of the next frontier.

Given a frontier F_i, to compute the next frontier F_{i+1}, the algorithm considers all neighbors of the current frontier, filters the already-visited ones, and then de-duplicates them and writes to their distances. The work of these steps is dominated by de-duplication which requires sorting, hence it costs

$$O\big(\|F_i\| \log(\|F_i\|)\big).$$

Since $\|F_i\| \le m$ and in a simple graph $m \le n^2$, we know $\log(\|F_i\|) = O(\log n)$, and hence the work of one round is

$$O\big(\|F_i\| \log n\big).$$

Summing over all rounds, the total work is therefore

$$\sum_{i=0}^{D} O\big(\|F_i\| \log n\big)$$

Now importantly, observe that at each round, the outgoing edges of the frontier are unique: since the frontiers partition the vertices by their distance and each edge only has one start vertex, each edge appears in at most one frontier calculation[1]. Therefore $\sum \|F_i\| = n + m$ and the total work is

$$O((m+n)\log n).$$

The span of each round is again dominated by the cost of sorting, which costs $O(\log^2 n)$ span, and since the graph by assumption has diameter D, the algorithm will terminate after D rounds, hence the total span is $O(D \cdot \log^2 n)$. □

[1] If the graph is undirected, each edge could appear in at most two frontier calculations, but that's a constant so the analysis still works out

4 Functional Parallel Breadth-First Search

As our final variant, we show how to eliminate the need for mutation in the (imperative) parallel breadth-first search implementation above. The key change needed is a way to maintain the distance values across each frontier computation without storing them in a (mutable) array.

In this version, much of the code stays the same. The frontiers can still be represented as sequences because they are disjoint at each iteration and hence there is no need to mutate them; they are entirely replaced at each iteration. The key change is `dist`, which is a BST-based dictionary that maps vertices to their distance from s. Any vertex that has not been visited is not in `dist`, and hence `dist` also plays the role of `visited`.

At each frontier computation, we can add all of the new distances in bulk using `union`, which is efficient since the cost of union depends predominantly on the cost of the *smaller* tree, not the combined cost of the trees.

Algorithm: Functional Parallel Breadth-First Search

```
// Assume an unweighted, enumerable graph
type adjacency_list = sequence<sequence<int>>

fun bfs(adj : adjacency_list, s : int) -> BstDict<int,int>:
  n = |adj|

  fun explore(frontier : sequence<int>,
              dist     : BstDict<int, int>, d : int)
    -> BstDict<int, int>:

    if |frontier| == 0: return dist

    neighbors = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => dist.find(v) == NONE, neighbors)
    next_frontier = unique(sort(unvisited))  // remove duplicates

    d⁺ = BstDict::from_seq(map(fn v => (v, d), next_frontier))
    next_dist = union(dist, d⁺)

    return explore(next_frontier, next_dist, d+1)

  return explore([s], {s:0}, 1)
```

It remains to analyze the work and span of this algorithm. This will require carefully analyzing the cost of the BST operations.

Theorem: Cost of Functional Parallel BFS

On a simple graph, functional parallel BFS costs $O(m \log n)$ work and $O(D \cdot \log^2 n)$ span, where D is the diameter of the graph (the maximum length of a shortest path).

Proof. Note that the cost of computing the next frontier remains the same since that code is no different from the imperative version. The only change is the computation of the updated distance dictionary, which is accomplished using `union`. So, it remains to analyze that.

Consider the union operation, which is used in the line `union(dist, d⁺)`. Recall that the union of two sets of size a and b where $1 \leq a \leq b$ is:

$$O\left(a + a \log\left(\frac{b}{a}\right)\right).$$

Therefore, denoting `|dist|` as d and `|d⁺|` as d^+, the cost of the union operation is at most the minimum of:

$$O\left(d + d \log\left(\frac{d^+}{d}\right)\right) \qquad \text{or} \qquad O\left(d^+ + d^+ \log\left(\frac{d}{d^+}\right)\right).$$

Since the work is the *minimum* of these two, to simplify, we can just be pessimistic and assume that d^+ is smaller. If our assumption is wrong, the work only goes *down*, so this is a valid upper bound on the work.

Now note that d^+ is exactly the vertices of the next frontier, and that d at worst contains every vertex in the graph and hence is at most n, so the work is at most

$$\sum_{i=0}^{D} O\left(\|F_i\| + \|F_i\| \log\left(\frac{n}{\|F_i\|}\right)\right).$$

Since $\log(n/d^+) \leq \log n$, we have that the work of the unions is at most

$$\sum_{i=0}^{D} O\left(\|F_i\| + \|F_i\| \log\left(\frac{n}{\|F_i\|}\right)\right) \leq \sum_{i=0}^{D} O\left(\|F_i\| \log n\right) = O((m+n)\log n).$$

Hence the work is no greater than the imperative version. The span of `union` is at most $O(\log n)$ and hence the span is also unaffected. □