Breadth-First Search

Breadth-first search (BFS) is another fundamental graph search algorithm. Like DFS, it can explore all vertices reachable from a given starting vertex. However, BFS explores vertices in a different order, which makes it useful for finding *shortest paths* in unweighted graphs.

Recall that DFS explores as deeply as possible before backtracking, so the first path it discovers to a vertex is not necessarily a shortest path. In contrast, BFS explores vertices in *increasing order of their distance* from the source.

We will begin with the familiar sequential BFS algorithm from 15-122, then reorganize it to expose opportunities for parallelism. Finally, we will show that we can implement both an imperative and a purely functional parallel version by using the data structures and algorithms we have developed throughout this course.

1 Sequential Breadth-First Search

Like DFS, Breadth-first search (BFS) explores a graph starting from a source vertex s. Unlike DFS, which explores one path as deeply as possible before backtracking, BFS explores vertices in increasing order of distance from a source vertex s: first the vertices at distance 0, then those at distance 1, then distance 2, and so on.

Definition: Shortest Path Distance

In an **unweighted graph** G, the **shortest-path distance** $\delta_G(s, v)$ is the minimum number of edges in any path from s to v. If v is unreachable from s, we define $\delta_G(s, v) = \infty$.

The key property of BFS is that *the first time a vertex is discovered, the algorithm has found a shortest path to it.* Thus, one of BFS's main applications is to compute the shortest-path distance from s to every vertex in an unweighted graph.

Implementing BFS with a queue. The classic implementation of BFS, which you have already seen in 15-122, uses a **FIFO queue** (first-in, first-out). Initially, the algorithm enqueues only the source vertex s. It then repeatedly dequeues a vertex u, examines its outgoing edges, and enqueues any vertices that have not yet been discovered.

Because the queue processes vertices in the order they were discovered, vertices closer to s are processed before those farther away.

Algorithm: Sequential Breadth-First Search

```
fun bfs(adj: UnweightedAdjList, s: int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]   // Pretend that ∞ is an int

  Q: queue<int> = empty  // Assume an O(1) queue implementation
  Q.enqueue(s)
  dist[s] ← 0

  while not Q.is_empty():
    u = Q.dequeue()
    for v in adj[u]:
      if dist[v] == ∞:
        dist[v] ← dist[u] + 1
        Q.enqueue(v)

  return dist
```

We use a distance of ∞ to represent undiscovered vertices. When a vertex v is first discovered via an edge (u, v), we set its distance to dist$[u]+1$, where u is the vertex from which it was discovered. Since BFS explores vertices in distance order, this is exactly the shortest-path distance $\delta_G(s, v)$.

Notice that unlike in DFS, this algorithm marks vertices as discovered (by setting their distance) when they are *discovered*, rather than when they are visited. In DFS, these two events happen back-to-back, so the distinction is not meaningful (a vertex, when discovered, is immediately recurred on and visited). For BFS however, it matters. This ensures that each vertex is enqueued at most once, even if several vertices have edges to it. Not doing this would increase the cost of the algorithm due to having to process duplicates.

Theorem: Cost of Sequential BFS

On a graph with n vertices and m edges, sequential BFS performs $O(n+m)$ work and has $O(n+m)$ span.

Each vertex enters the queue at most once, and each adjacency list is examined at most once. Assuming constant-time queue operations, the total cost is therefore $O(n+m)$.

This implementation is inherently sequential: vertices are processed one at a time from the queue. In the next section, we will reorganize the algorithm to make its opportunities for parallelism more apparent.

2 Layered Breadth-First Search

Recall that our original idea for BFS was to explore vertices in increasing order of their distance from the source. The FIFO queue achieves this implicitly: at any point during BFS, the queue contains vertices from at most two consecutive distances, corresponding to the current distance and the next distance.

To make this structure explicit, we can reorganize BFS to process an entire **layer** at a time, rather than processing vertices individually from a queue.

Definition: BFS Frontier

The **frontier** F_i is the vertices whose shortest-path distance from the source s is exactly i:

$$F_i = \{v \in V : \delta_G(s, v) = i\}.$$

In particular, $F_0 = \{s\}$. To compute F_{i+1} from F_i, we examine the outgoing edges of all vertices in F_i and collect their previously undiscovered neighbors. This gives us a **layered BFS** algorithm: Maintain the current frontier and construct the next frontier by examining its outgoing edges.

Algorithm: Layered Sequential Breadth-First Search

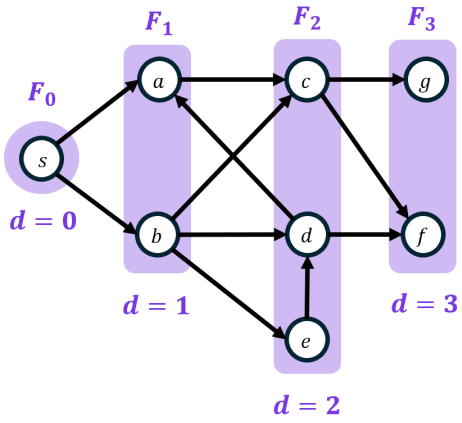
```
fun bfs(adj: UnweightedAdjList, s: int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[s] ← 0
  frontier = [s]
  d = 0

  while |frontier| > 0:
    next_frontier = []
    for u in frontier:
      for v in adj[u]:
        if dist[v] == ∞:
          dist[v] ← d+1
          next_frontier.append(v)

    frontier = next_frontier
    d ← d+1

  return dist
```

The following diagram illustrates how BFS partitions the reachable vertices into distance layers.



Here, d denotes the distance of the *current* frontier. At each iteration, we discover vertices at distance $d+1$, add them to the next frontier, and then advance to that frontier.

The algorithm is still sequential, but it explicitly expresses the computation of one frontier in terms of the previous frontier. This will make it much easier to parallelize.

Theorem: Correctness of Layered BFS

For every vertex v, the algorithm computes

$$\text{dist}[v] = \delta_G(s, v),$$

where $\delta_G(s, v) = \infty$ if v is unreachable from s.

Proof. We prove correctness by induction on the distance d. At the beginning of each iteration of the loop, the following invariants hold:

1. `frontier` contains exactly the vertices at distance d from s, i.e., F_d.
2. Every vertex at distance at most d has its correct distance recorded in `dist`, while all other vertices still have distance ∞.

Initially, $d = 0$ and the frontier contains only s, whose distance is 0, so both invariants hold. Now suppose both invariants hold at the beginning of an iteration. For every vertex u in the current frontier, we examine its outgoing edges (u, v).

If v has not yet been discovered, then there is a path of length $d+1$ from s to v. Furthermore, v cannot have a shorter path, since every vertex at distance at most d has already been discovered. Thus, assigning $\text{dist}[v] = d+1$ is correct. Conversely, every vertex v at distance $d+1$ must have a predecessor u at distance d along some shortest path. Since u belongs to the current frontier, the algorithm will examine (u, v) and discover v.

Therefore, `next_frontier` contains exactly F_{d+1}, and all vertices at distance at most $d+1$ have their correct distances recorded. The invariants are preserved when we advance to the next iteration. When the algorithm terminates, every reachable vertex has been discovered with its correct shortest-path distance. Unreachable vertices retain distance ∞. □

Theorem: Cost of Layered BFS

On a graph with n vertices and m edges, layered sequential BFS performs $O(n+m)$ work and has $O(n+m)$ span.

Each reachable vertex belongs to exactly one frontier, so its adjacency list is processed exactly once. Consequently, the algorithm examines at most m edges and processes at most n vertices, giving $O(n+m)$ total work. Since all the loops are sequential, the span is also $O(n+m)$.

3 Imperative Parallel Breadth-First Search

The layered version of BFS exposes an opportunity for parallelism that was previously hidden by the queue version. Although each frontier depends on the previous one, the vertices *within* a frontier can examine their outgoing edges in parallel. However, we cannot simply replace the nested loops with parallel loops.

Consider two vertices u_1 and u_2 in the current frontier, both of which have an edge to the same undiscovered vertex v: If both vertices attempt to discover v simultaneously, they may both update dist[v] and append v to the next frontier. This creates a data race. The challenge is therefore to construct the next frontier in parallel, without duplicate discoveries or concurrent writes to the same memory location.

Constructing the next frontier. We can solve this problem using parallel sequence operations. Given the current frontier:

1. Use `map` and `flatten` to collect all outgoing neighbors.
2. Use `filter` to remove vertices that have already been discovered.
3. Remove duplicates from the remaining vertices to obtain the next frontier.
4. Assign distances to the vertices in the next frontier in parallel.

One simple way to remove duplicates is by sorting the sequence and then applying `unique`:

```
remove_duplicates = fn(S) => unique(sort(S))
```

This operation costs $O(|S| \log |S|)$ work and $O(\log^2 |S|)$ span due to the cost of sorting.

Algorithm: Imperative Parallel Breadth-First Search

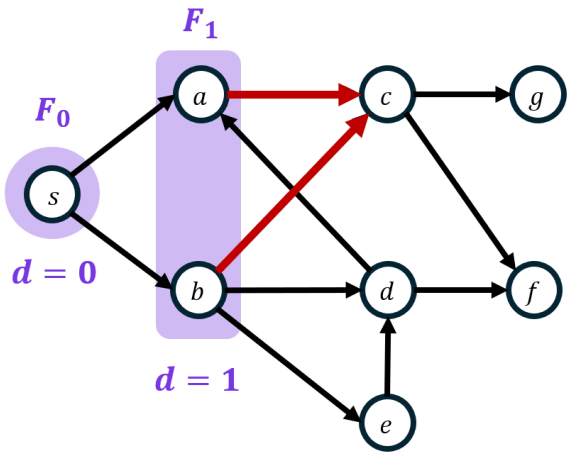
```
fun bfs(adj: UnweightedAdjList, s: int) -> sequence<int>:
  n = |adj|
  dist = [∞ for _ in 0...n-1]
  dist[s] ← 0
  frontier = [s]
  d = 0

  while |frontier| > 0:
    neighbors = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => dist[v] == ∞, neighbors)
    next_frontier = remove_duplicates(unvisited)

    parallel for v in next_frontier:
      dist[v] ← d+1

    frontier = next_frontier
    d ← d+1

  return dist
```



The diagram above illustrates the challenge of building the next frontier in parallel. Notice that `unvisited` may contain the same vertex multiple times, since several vertices in the current frontier might share an outgoing neighbor.

The important point is that all filtering and deduplication happens *before* we update any distances. Once we have constructed `next_frontier`, each vertex appears exactly once, so the parallel distance assignments cannot write to the same location.

The algorithm is correct by the same reasoning as layered sequential BFS: each round discovers exactly the previously undiscovered vertices at distance $d+1$ from the source.

3.1 Cost Analysis

Definition: Frontier Volume

For a frontier F_i, we define its **volume** as

$$\|F_i\| = \sum_{u \in F_i} (1 + d_G^+(u)).$$

This counts both the vertices in the frontier and their outgoing edges.

The volume measures the amount of data processed when constructing the next frontier. An important observation is that the total volume over all frontiers is linear in the graph's size.

Lemma: Total Frontier Volume

For the frontiers generated by BFS,

$$\sum_{i=0}^{D} \|F_i\| = O(n+m),$$

where D is the largest finite distance from the source to any reachable vertex.

Proof. Each reachable vertex belongs to exactly one frontier, so the total number of vertices counted is at most n. Likewise, each outgoing edge is processed only when its source vertex belongs to the current frontier. Thus, each directed edge is counted at most once. For undirected graphs, each edge may be counted twice, which does not change the asymptotic bound. □

Theorem: Cost of Imperative Parallel BFS

Let D be the largest shortest-path distance from s to any reachable vertex. Imperative parallel BFS costs $O((n+m)\log(n+m))$ work and $O((D+1)\log^2(n+m))$ span.

Proof. Consider the computation of the next frontier from F_i. The number of neighbors is at most $\|F_i\|$, so the `map`, `flatten`, and `filter` operations cost $O(\|F_i\|)$ work and $O(\log\|F_i\|)$ span.

The most expensive operation is `remove_duplicates`, which sorts the sequence. It costs $O(\|F_i\|\log\|F_i\|)$ work and $O(\log^2\|F_i\|)$ span. Since $\|F_i\| = O(n+m)$, the work of one layer is

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

Summing over all frontiers, using the frontier-volume lemma, we get

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

There are $D+1$ nonempty frontiers since the distances range from 0 through D. Each round has $O(\log^2(n+m))$ span, so the total span is $O((D+1)\log^2(n+m))$. □

4 Functional Parallel Breadth-First Search

Our parallel BFS algorithm processes each frontier using parallel sequence operations, but it still relies on *parallel mutation* to update the distance array. Can we eliminate this mutation and obtain a purely functional parallel BFS? The answer is yes, and the key idea is to replace the mutable distance array with a **persistent dictionary** mapping vertices to their shortest-path distances. We will use a treap-based[1] dictionary:

```
type dist_dict = treap_dict<int, int>
```

Unlike the array implementation, we do not need a special value such as ∞ to represent undiscovered vertices. Instead, a vertex is undiscovered exactly when its key is absent from the dictionary. Most of the next-frontier computation remains unchanged: we collect the neighbors of the current frontier, filter out previously discovered vertices, and remove duplicates.

The main difference is how we record the newly discovered distances. Rather than writing to a shared array, we:

[1]This would technically make all of our cost bounds randomized and hold with high probability, so for our cost analysis we will pretend its an AVL Tree or Red-Black Tree to get deterministic bounds. We will stick with Treaps for the pseudocode though since that's what we have actually covered in this course.

1. Construct a dictionary containing the distances of all vertices in the next frontier.
2. Use **`union`** to combine this persistent dictionary with the existing distances, obtaining a new persistent dictionary.

Algorithm: Functional Parallel Breadth-First Search

```
type dist_dict = treap_dict<int, int>

fun bfs(adj: UnweightedAdjList, s: int) -> dist_dict:

  fun explore(frontier: sequence<int>,
              dist: dist_dict, d: int) -> dist_dict:
    if |frontier| == 0:
      return dist

    neighbors = flatten(map(fn u => adj[u], frontier))
    unvisited = filter(fn v => dist.find(v) == NONE, neighbors)
    next_frontier = remove_duplicates(unvisited)

    new_dists = dist_dict.from_sequence(
      map(fn v => (v, d+1), next_frontier))
    next_dist = union(dist, new_dists)

    return explore(next_frontier, next_dist, d+1)

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

The parameter d is the distance of the current frontier. Each recursive call constructs the frontier at distance $d+1$ and a new dictionary containing its distances.

Because the dictionary is persistent, the operation `union` produces a new dictionary without mutating the existing one. Furthermore, the dictionaries `dist` and `new_dists` have disjoint keys, since the next frontier contains only previously undiscovered vertices. So we do not have to worry about tie-breaking the values. Thus, we have obtained a purely functional algorithm: each recursive call produces new values rather than modifying shared mutable state!

4.1 Cost Analysis

Theorem: Cost of Functional Parallel BFS

Let D be the largest shortest-path distance from the source s to any reachable vertex. Functional parallel BFS costs $O((n+m)\log(n+m))$ work and $O((D+1)\log^2(n+m))$ span, assuming `dist_dict` is a balanced BST type.

Proof. The computation of `neighbors` and `next_frontier` are the same as the imperative versions. Computing `unvisited` costs an additional $O(\log n)$ factor due to the dictionary `find`, so we pay $O(\|F_i\|\log n)$ additional work. The non-trivial additional costs come from constructing `new_dists` and computing the union with the existing distances.

Let $k = |F_{i+1}| \leq \|F_i\|$ be the number of vertices discovered in round i. Constructing a dictionary from k key-value pairs costs $O(k \log k)$ work and $O(\log^2 k)$ span. Then, recall that the union of two balanced-BSTs of sizes $a \leq b$ can be computed with $O(a \log b)$ work and $O(\log b)$ span.

Since `new_dists` has k elements and `dist` has at most n elements, the union therefore costs at most $O(k \log n)$ work and $O(\log n)$ span. Every reachable vertex, except the source, belongs to exactly one newly constructed frontier. Therefore, summing over all rounds, the we pay

$$\sum_{i=0}^{D} (|F_{i+1}| + \|F_i\|) \log n = O((n+m) \log n).$$

additional work. The additional span per round is $O(\log^2 n)$, giving $O((D+1)\log^2 n)$ additional span overall. Both are within the bounds for imperative parallel BFS. Thus, the functional version has the same asymptotic cost. □

4.2 Summary

We have developed four implementations of BFS, each building on the previous one. All four compute the same shortest-path distances, but differ in how they organize the computation and expose parallelism.

Version	**Key Idea**	**Work**	**Span**
Classic BFS	FIFO queue	$O(n+m)$	$O(n+m)$
Layered BFS	Explicit frontiers	$O(n+m)$	$O(n+m)$
Imperative parallel BFS	Parallel frontiers	$O((n+m)\log(n+m))$	$O((D+1)\log^2(n+m))$
Functional parallel BFS	Persistent BST dictionary	$O((n+m)\log(n+m))$	$O((D+1)\log^2(n+m))$

In the span bounds, D is the maximum shortest-path distance from the source to any reachable vertex. The key insight is that BFS has sequential dependencies *between* frontiers, but substantial opportunities for parallelism *within* each frontier. This makes parallel BFS particularly effective on graphs with wide frontiers and small distances from the source.

Furthermore, by using persistent BST dictionaries, we can achieve the same asymptotic work and span bounds without relying on shared mutable state.

Remark: Can Parallel BFS Be Work-Efficient?

Sequential BFS costs $O(n+m)$ work. Our parallel algorithms incur an extra logarithmic factor of work because they use sorting and/or a balanced binary search tree.

It is in fact possible to achieve $O(n+m)$ work using more sophisticated techniques. For example, atomic compare-exchange operations can ensure that each vertex is discovered by only one processor. However, this requires atomic synchronization primitives and assumptions about their cost under contention; ordinary concurrent writes to the same location would be data races.

Our implementation avoids these complications by using parallel sequence operations at the expense of a logarithmic factor in work.