DFS vs. BFS; Optimal-cost paths

Kathi Fisler and Milda Zizyte

October 21, 2022

Objectives

By the end of this lecture, you will know:

- How order of vertices considered matters when traversing graphs
- The abstractions of *Stack* and *Queue*
- How to compute paths between vertices
- How to compute shortest (or *least costly* paths between vertices

1 DFS vs BFS (Exploring vertices in a different order)

1.1 Exploring in a different order

Where in the code does the "deep" decision we discussed in the last lecture get made? It's in the lines where we `addLast` to `toCheck` and `removeLast` from `toCheck`. By putting the vertices reachable from the current vertex at the *back* of the `toCheck` list, and by removing also from the back of this list, we guarantee that we will visit them before we visit vertices that are already pending in the list.

1.2 Breadth-First Search

So what if we made a different decision there, and took the new vertices off the *front* of the list rather than the back? In other words, what if our **`while`** loop looked like:

```
while (! toCheck.isEmpty()) {
    Vertex<T> checkingVertex = toCheck.removeFirst(); // this line changes!
    if (dest.equals(checkingVertex)) {
      return true;
    }
    for (Vertex<T> neighbor : checkingVertex.toVertices) {
      if (!visited.contains(neighbor)) {
        visited.add(neighbor);
        toCheck.addLast(neighbor);
      }
    }
```

Now which order would we visit the vertices in when searching for a Boston-Hartford route?

We'd still start at Boston, putting Providence and Worcester in the `toCheck` list (in that order). When processing Providence, we would put any neighbor Vertices that we hadn't visited *after* Worcester in the list (in this case, there aren't any Providence neighbors we haven't visited, but that's besides the point). In this way, we wouldn't get stuck in a deep search without also making progress on the other paths.

This version is called *breadth-first search* (BFS), because it explores the breadth of next vertices before moving onto their next vertices. This implies that you first check all vertices reachable in one step from the starting point, then all vertices reachable in two steps from the starting point, then all vertices reachable in three steps, and so on.

If we were using breadth-first search, would we still need a visited list? Wouldn't we eventually find the route, before getting stuck in an infinite loop? If there is a route, you would find it without going into an infinite loop. But what if there is no route (as with Hartford to Boston)? Then, you would get stuck in an infinite loop if you didn't have a visited list.

Furthermore, the visited list is still useful for time-efficiency. When your graph has multiple paths of different length to the same vertex from your starting point, you'd end up exploring that same vertex for each path, rather than only once.

Before we move on to our third graph algorithm, we will wrap up our discussion on BFS and DFS. '

1.3 A word on data structures

The difference between our `LinkedList`-based DFS and BFS implementations was that, for DFS, we used `addLast` and `removeLast`, whereas for BFS, we used `addLast` and `removeFirst`.

For DFS, we could have very well used `addFirst` and `removeFirst`. The key idea is that we wanted to remove the *latest* element we added. The key abstraction here is that we were using a Last-In-First-Out (LIFO) add and remove strategy. A LIFO is also called a *stack*, a common data structure in computer science. You will often see the add method of a stack called *push* and the remove method called *pop*.

Similarly, for BFS, the key idea is that we wanted to remove the very *earliest* element we added. This is called a First-In-First-Out (FIFO). A FIFO is also alled a *queue*, another common data structure in computer science. You might sometimes see *offer* or *add* used to add elements to a queue, and *poll* or *remove* used to remove elements from a queue.

Note that, in Java, `LinkedLists` offer you use of all of these methods (like `push` and `pop`). That is because LinkedLists can be used to implement stacks or queues, just like in our `canReach` code above, when these are viewed as interfaces!

Being comfortable with the distinction between stacks and queues will come in handy when studying algorithms in computer science.

1.4 When to use BFS or DFS

Which algorithm (BFS or DFS) should you use in practice? It depends on context.

- If your goal is to find the shortest path length, use BFS. Since BFS checks all nodes at each distance from the starting node before looking at any node at distance + 1, if there are two paths of different lengths to the same node, BFS detects the shortest one first.
- If your goal is detect cycles, use DFS. As soon as DFS tries to process a node that is already in the visited list, you know you have a cycle in the graph. Cycle-detection ends up being a key component of some advanced computing applications.
- Also, given the shape of your particular graph and characteristics of your route queries, one of BFS or DFS might perform better in practice. For example, if your graph is very wide (each vertex has a lot of neighbors) and you know the path you're looking for exists deep in the tree, it will save memory to use DFS.

1.5 Tracking Routes

The code we wrote for `canReachDFS` will tell us *whether* a route exists between two vertices, but not which vertices to take to get between vertices. To produce the route, we also need to record the vertex through which each vertex entered the `toCheck` list.

To do this, we will create a data structure to store information about how the computation had progressed so far. Specifically, we will create a `HashMap` that maps each vertex to the vertex that added it to `toCheck` – essentially, we are keeping track of where a given vertex came from. For this reason, let's call this `HashMap` `cameFrom`. The new code is on lines 5 and 20:

```
 1    // in the Graph class
 2  public List<Vertex<T>> routeDFS (Vertex<T> source, Vertex<T> dest) {
 3    LinkedList<Vertex<T>> toCheck = new LinkedList<Vertex<T>>();
 4    HashSet<Vertex<T>> visited = new HashSet<Vertex<T>>();
 5    HashMap<<Vertex<T>, Vertex<T>> cameFrom = new HashMap<<Vertex<T>,Vertex<T>>();
 6
 7    toCheck.addLast(source);
 8    visited.add(source);
 9
10    while (! toCheck.isEmpty()) {
11      Vertex<T> checkingVertex = toCheck.removeLast();
12      if (dest.equals(checkingVertex)) {
13        // backtrack through cameFrom and return route
14      }
15      for (Vertex<T> neighbor : checkingVertex.toVertices) {
16        if (!visited.contains(neighbor)) {
17          visited.add(neighbor);
18          toCheck.addLast(neighbor);
19          cameFrom.put(neighbor, checkingVertex); // can get to neighbor from
20                                                  // checkingVertex
21        }
22      }
23    }
24    // return empty list or throw exception
25  }
```

Now, consider running this new code to find the route between Boston and Hartford. For the following, we use `[]` to denote the contents of a list (in order). We use `{}` to denote the contents of a `HashSet` when just elements are listed, and to denote the contents of a `HashMap` when key-value pairs are listed (e.g. `{key1 -> value1, key2 -> value2}`).

Before entering the **`while`** loop,

```
toCheck is [bos]
visited is {bos}
cameFrom is {}
```

The first time through the **`while`** loop, we remove `bos` from `toCheck`, we add its neighbors to `toCheck` and update `cameFrom`:

```
  toCheck is [pvd, wos]
  visited is {bos, pvd, wos}
  cameFrom is {pvd -> bos, wos -> bos}
```

The second time through the **`while`** loop, we remove `wos` from `toCheck` (remember that we are removing from the end of the list), and loop through its neighbors

```
  toCheck is [pvd, har]
  visited is {bos, pvd, wos, har}
  cameFrom is {pvd -> bos, wos -> bos, har -> wos}
```

Finally, we remove `har` from `toCheck` and see that we have arrived at the destination. Then, starting with `har`, we can backtrack to `bos` by following the vertices we came from using `cameFrom` (pseudocode provided):

```
retlist = []
checking = dest
add dest to the front of retlist
while (checking != source):
  checking = cameFrom.get(checking) (find out how we got to checking)
  add checking to the front of retlist
return retlist
```

For the `bos` to `har` route, `checking` is initialized to `har` and `retlist` becomes [`har`]. The next time through the loop, `checking` becomes `wos` (because `cameFrom` maps `har` to `wos`) and `retlist` becomes [`wos`, `har`]. The subsequent time through the loop, `checking` becomes `bos` and `retlist` becomes [`bos`, `wos`, `har`], and we terminate the loop because we have backtracked all the way to Boston (the source). We can then return `retlist` as the path that got us from Boston to Hartford.

2 Finding Shortest (cheapest) Paths: Dijkstra's Algorithm

Now assume that we care about not just finding some path, but about finding a good path. There are many definitions of good paths. One of them might be a path that cost the least, or that had the shortest total distance. For this, we want to use an algorithm called Dijkstra's Algorithm.

Dijkstra's algorithm assumes that our graphs have costs or weights on the edges. Without these, there's no useful notion of a "shortest" path. Here's a different graph, this times with edge weights. In this graph, we will also assume that edges are bidirectional (meaning that you could travel them in either direction). We call this an *undirected* graph, where the previous graph we were working with was *directed*. The algorithm we highlight here will work with directed and undirected graphs.

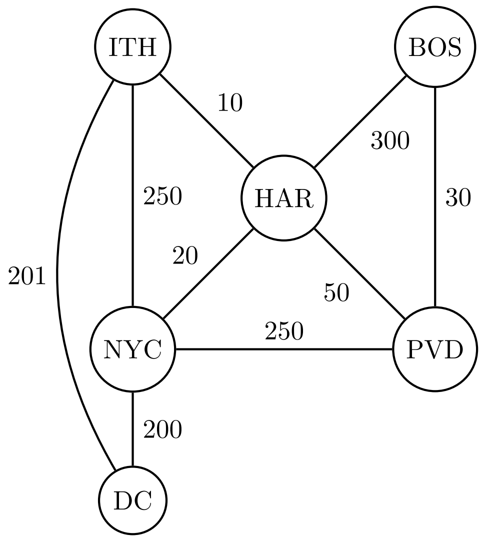

Figure 1: An example weighted graph. Edges are undirected; travel is possible in both directions at the same cost.

How do we think about finding short/cheap paths? A naive approach would be to take the cheapest edge each time. However, this might lead to a case where the solution is actually wrong! Consider getting from

Boston to NYC in this way: we would first try going to Providence, then to Hartford, and then end up in Ithaca. But from Ithaca, the only remaining paths are really expensive – we would end up paying a cost of 201 to go to DC, and then 200 more to finally get to NYC. This is clearly not the optimal path – by looking at this graph, the actual optimal route is from Boston to Providence to Hartford to NYC, which a cost of 100.

Another approach might be to enumerate all possible paths between Boston and NYC (for example, by writing a DFS or BFS that doesnt immediately terminate), and then computing their costs and choosing the lowest one, but this turns out to not be optimally efficient.

To conceptualize a better algorithm, think back to our discussion of how we might add items to the `toCheck` list. Shortest paths involve some sort of optimization, or priority of which nodes to consider. In this case, we want to check nodes in an order based on the cheapest way to make progress in our search for a path.

For example, if we start at Boston, we can compute the cost to get to Hartford (300) and Providence (30), and then select the cheaper of the two and update the costs of its neighbors accordingly. In this example, we would first consider Providence, and compute a new cost of getting to Hartford (30 + 50), which is cheaper than the previous cost we had. Thus, we would update our knowledge of the cheapest route (so far) to Hartford.

The data structure we will use to select the next vertex to consider is called a *Priority Queue*. You worked with Priority Queues in lab, and the main idea is that you can assign values of comparison to every item in a priority queue and get the item with the smallest value of comparison in *log(N)* time.

For our value of comparison, we will use the smallest computed distnce (so far) from the source vertex. Initially, all vertices are infinitely far from the starting vertex (except the start vertex itself).

After initialization, we should see the following estimates in `toCheckQueue`:

`[BOS:0, PVD:inf, HAR:inf, ITH:inf, NYC:inf, DC:inf]`

The algorithm removes `BOS` from the priority queue; our estimate of 0 for reaching Boston is optimal (we mark this in these notes with `[X]`, meaning we won't have to revisit `BOS`). Now, for each of Boston's neighbors (`HAR` and `PVD`), we check to see if we can do better than the current estimate. Since the current estimates are both infinity, and Boston has finite-cost edges to both cities, we can improve both estimates.

1. `[BOS:0 [X], PVD:30, HAR:300, ITH:inf, NYC:inf, DC:inf]`

Now the algorithm removes `PVD`: 30 is the optimal path length from Boston to Providence. We can improve estimates for 2 of Providence's neighbors: Hartford and New York City. Both now equal the optimal cost to reach Providence (30) plus the length of the direct edge from Providence (50 and 250 respectively):

2. `[BOS:0 [X], PVD:30 [X], HAR:80, ITH:inf, NYC:280, DC:inf]`

The algorithm continues until it runs out of nodes on the priority queue, updating estimates as follows:

3. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90, NYC:100, DC:inf]`

4. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100, DC:291]`

5. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100 [X], DC:291`

6. `[BOS:0 [X], PVD:30 [X], HAR:80 [X], ITH:90 [X], NYC:100 [X], DC:291[X]`

Notice that this algorithm actually gives us the cost of the shortest route between Boston (the starting vertex) and every other vertex in the graph, not just a specific one in question.

2.1 Pseudocode

Along with maintaining the priority queue, we also want to maintain the same `cameFrom` HashMap that we discussed for DFS, so that we can backtrack to the source from any destination. The way we update `cameFrom` is by changing the entry for a city every time we find a more optimal route to that city. Assuming V is the collection of vertices:

```
toCheckQueue = V (prioritized on routeDist)
cameFrom = empty map

for v in V:
  v.routeDist = inf
source.routeDist = 0

while toCheckQueue is not empty:
  checkingV = toCheckQueue.removeMin()
  for neighbor in checkingV's neighbors:
     if checkingV.routeDist + cost(checkingV, neighbor) < neighbor.routeDist:
        neighbor.routeDist = checkingV.routeDist + cost(checkingV, neighbor)
        cameFrom.add(neighbor -> checkingV)
        toCheckQueue.decreaseValue(neighbor)

backtrack from dest to source through cameFrom
```

Stop and think: Why is it that we do not have to revisit a vertex after removing it from the priority queue?

Reason: At the time we remove it from the queue, we have computed the cost of the optimal path to that vertex from the starting vertex. We can reason about this by contradiction: if there existed a better path that we could compute to that vertex after we had already removed it, that path would have to go through a vertex with a shorter distance to the source than it. But no such vertex exists in the queue, because we remove vertices from the queue in order of shortest/cheapest. Thus, we ensure that Dijkstra's algorithm computes the optimal paths. Mathematically rigorous formal proofs of this idea exist and have been used to prove Dijkstra's algorithm correct.