CS2 Algorithms and Data Structures Note 11

Breadth-First Search and Shortest Paths

In this last lecture of the CS2 Algorithms and Data Structures thread we will consider the problem of computing distances and shortest paths between vertices of a graph. We will start with a simple algorithm based on breadth-first search and then consider the so-called Floyd-Warshall algorithm.

11.1 BFS Revisited

Remember that a breadth-first search starting at some vertex v of a graph G first visits v, then all neighbours of v, then their neighbours, et cetera. In LN9, we saw an implementation of BFS. For your convenience, I have included it again in this lecture note (see Algorithms 11.1 and 11.2). Disregard line 2 of bsf and line 8 of bfsFromVertex for now; they record some additional information that will be useful later.

Algorithm bfs(G)

1. Initialise Boolean array $visited$ by setting all entries to FALSE
2. Initialise vertex array $parent$ by setting all entries to NIL
3. Initialise *Queue* Q
4. **for all** $v \in V$ **do**
5. **if** $visited[v] =$ FALSE **then**
6. bfsFromVertex(G, v)

Algorithm 11.1

The analysis of bfs can be done very similarly to the analysis of dfs last lecture. Let G be a graph with n vertices and m edges. If we disregard the time spent executing the subroutine bfsFromVertex, bfs(G) requires time $\Theta(n)$. Now let us consider the execution of bfsFromVertex(G, v) for all v at once. Lines 1 and 2 are executed at most once for each vertex, thus overall they require time $O(n)$. The crucial observation is that Lines 3–9 are executed exactly once for each vertex, because each vertex v will enter the queue once. Then $visited[v]$ will be set to TRUE, and after v is dequeued it can never enter the queue again. The execution

Algorithm bfsFromVertex(G, v)

1. $visited[v] = \text{TRUE}$
2. Q.enqueue(v)
3. **while not** Q.isEmpty() **do**
4. $\quad v \leftarrow Q$.dequeue()
5. $\quad$ **for all** w adjacent to v **do**
6. $\quad\quad$ **if** $visited[w] = \text{FALSE}$ **then**
7. $\quad\quad\quad visited[w] = \text{TRUE}$
8. $\quad\quad\quad parent[w] = v$
9. $\quad\quad\quad Q$.enqueue(w)

Algorithm 11.2

of lines 3–9 requires time $\Theta(\text{out-degree}(v))$. Thus overall, bfs requires time

$$\Theta(n) + O(n) + \sum_{v \in V} \Theta(\text{out-degree}(v)) = \Theta(n + m),$$

just like dfs (which, of course, is no big surprise given how similar bfs and dfs are).

So what is the array $parent$ good for? For each vertex w, it contains a reference to the vertex through which the BFS entered w. If we follow these references back, we will arrive at the vertex the BFS started at. Thus for every vertex w we can reconstruct the *path* from the start vertex to w that the BFS took. This can be quite useful.

In the following, I am assuming that you know the basic terminology of *trees* and *forests*. If you don't, go to the library and look it up.

As with a DFS, a BFS starting at some vertex v explores the graph by building up a tree that contains all vertices that are reachable from v and all edges that are used to enter these vertices. We call this tree a *BFS tree*. A complete BFS exploring the full graph (and not only the part reachable from a vertex v) builds up a collection of trees, or forest, called a *BFS forest*. The array $parent$ can be seen as a compact representation of this forest: The edges of the forest are those going from $parent[v]$ to v, for all vertices v. Since all vertices of the graph are vertices of the BFS forest, there is no need to store the vertices separately.

11.2 Shortest Path Problems

Recall that the *length* of a path in a graph is the number of edges in the path. A *shortest path* from a vertex v to a vertex w in a graph G is a path from v to w of minimum length. The *distance* from v to w, denoted by $d(v, w)$, is the length of

a shortest path from v to w. If there is no path from v to w, then we define the length to be ∞ ("infinity").

The most basic shortest path problem is the following *single pair shortest path problem*: The input consists of a graph G and two vertices v, w, and the problem is to find a shortest path from v to w in G. But often, we do not only want to find a shortest path between two particular vertices, but shortest paths either from one vertex to all other vertices or between all pairs of vertices. The *single source shortest paths* problem takes as input a graph G and a vertex v and asks for shortest paths from v to all other vertices. The *all pairs shortest paths problem* takes as input a graph G and asks for shortest paths between all pairs of vertices. It turns out that the single pair shortest path problem cannot be solved significantly more efficiently than the single source shortest paths problem. Therefore, we will concentrate on the single source shortest paths problem and the all pairs shortest paths problem.

For the single source shortest paths problem and the all pairs shortest paths problem the question arises how we represent the shortest paths. If we store each path separately, for example in a linked list, then we use a lot of space; up to $\Theta(n^2)$ for the single source shortest paths and $\Theta(n^3)$ for the all pairs shortest paths. Fortunately, there are always families of shortest paths that can be represented more compactly. Observe that if P is a shortest path from v to w and x is a vertex appearing on P, then the segment of P connecting v with x is a shortest path from v to x. This means that a shortest path from v to w already contains shortest paths to all other vertices appearing on the path, and there is no need to store these shorter paths separately. Moreover, shortest paths may share vertices. Suppose that we have a shortest path P_1 from v to w_1 and a shortest path P_2 from v to w_2, and that a vertex x appears on both P_1 and P_2 (see Figure 11.3(a)). Then instead of storing both paths, we can just store the segment of one of the two paths connecting v with x and the remaining two segments connecting x with w_1 and w_2, respectively (as in Figure 11.3(b)). If we start with a family of shortest paths from a vertex v to all other vertices of the graph and eliminate duplicate vertices in the way just described, we'll end up with a tree whose root is v and whose vertices are all vertices that are reachable from v. All paths in this tree from the root v to the other vertices are shortest paths in the graph. We call such a tree a *shortest path tree*.

Storing a shortest path tree only requires space $O(n)$. Indeed, as in Algorithms 11.1 and 11.2 above, we can store such a tree quite efficiently in a *parent array*. For each vertex appearing in the tree (except v itself) this array stores its parent; the entries for all other vertices are just NIL. We can represent shortest paths between all pairs of vertices by a *parent matrix*, whose rows are parent arrays for all vertices.

Often, we are also interested in the distances between vertices. We can store the distances of all vertices from a source v in a *distance array* and the distances between all pairs of vertices in a *distance matrix*.

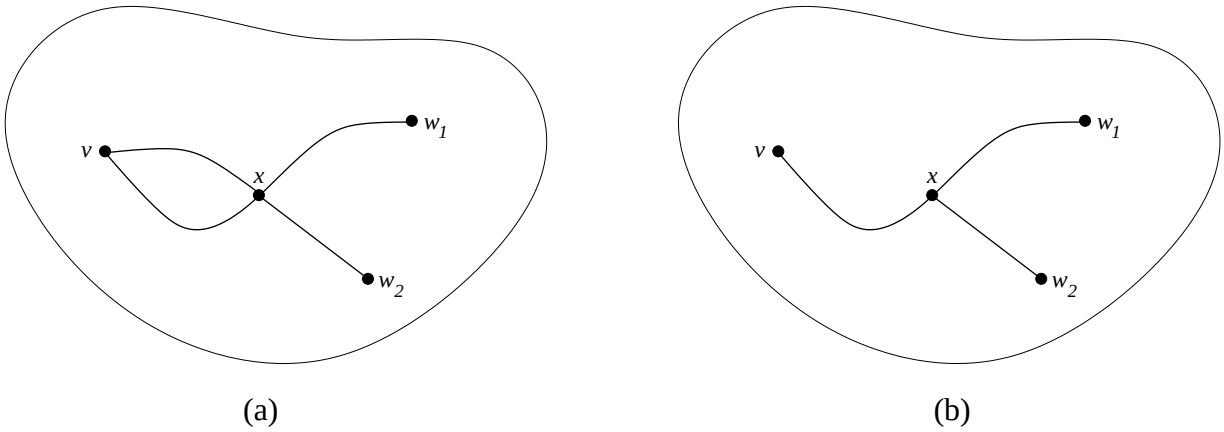

Figure 11.3.

11.3 Computing Distances and Shortest Paths by BFS

Shortest path problems can be solved using BFS because a BFS starting at a vertex v visits the vertices in the order of increasing distances from v. Indeed, a BFS tree is always a shortest path tree. Thus we can simply solve the single source shortest path problem by a BFS starting at the source v. Algorithm 11.4 does this and returns the result in the form of a parent array. In addition, the algorithm also computes a distance array, just to illustrate how this can be done.

Clearly, the asymptotic worst-case running time of bfsSSSP is the same as that of bfs, i.e., $\Theta(n+m)$.

We can solve the all pairs shortest paths problem by executing bfsSSSP(G, v) for all vertices v. The running time of this algorithm will be $\Theta(n(n+m)) = \Theta(n^2+n\cdot m)$. Note that this is in $O(n^3)$, because $m \leq n^2$.

11.4 The Floyd-Warshall Algorithm

The Floyd-Warshall Algorithm is an algorithm for the all pairs shortest paths problem that works completely differently than the BFS-based algorithm described in the previous section.

Let $G = (V, E)$ be a graph with vertices numbered 0 to $n-1$. In the following discussion, we will identify vertices with their numbers. In other words, we assume that $V = \{0, \ldots, n-1\}$. To explain how the algorithm works, it is best to consider the problem of computing the *distance matrix* of G first, i.e., the matrix $D = (d_{ij})_{0 \leq i,j \leq n-1}$ with d_{ij} being the distance from vertex i to vertex j.

Lemma 11.5. *For all i, j,*

$$d_{ij} \leq n-1.$$

Algorithm bfsSSSP(G, v)

```
1.  Initialise array distance by setting all entries to ∞
2.  Initialise array parent by setting all entries to NIL
3.  Initialise Queue Q
4.  distance[v] = 0
5.  Q.enqueue(v)
6.  while not Q.isEmpty() do
7.      v ← Q.dequeue()
8.      for all w adjacent to v do
9.          if distance[w] = ∞ then
10.             parent[w] = v
11.             distance[w] = distance[v] + 1
12.             Q.enqueue(w)
13. return parent
```

Algorithm 11.4

I leave the proof of this lemma as an exercise.

We first consider a simple but inefficient matrix based algorithm for the problem of computing a distance matrix. For $0 \leq i, j \leq n-1$ and $k \geq 0$, let

$$d_{ij}^{\leq k} = \begin{cases} d_{ij} & \text{if } d_{ij} \leq k, \\ \infty & \text{otherwise.} \end{cases}$$

Let $D^{\leq k} = (d_{ij}^{\leq k})_{0 \leq i,j \leq n-1}$; we may call $D^{\leq k}$ the *distance up to* k matrix of G. Note that $D^{\leq 1}$ is the matrix obtained from the adjacency matrix of G by replacing all diagonal entries by 0 and all 0s that are not on the diagonal by ∞. Moreover, note that $D^{\leq (n-1)}$ is equal to the full distance matrix D, because by Lemma 11.5 we have $d_{ij} \leq n-1$ for all i, j.

The crucial observation is that for $k \geq 2$ and $0 \leq i, j \leq n-1$, there is a path of length k from vertex i to vertex j if, and only if, for some vertex ℓ, there is a path of length $(k-1)$ from i to ℓ and an edge from ℓ to j. It follows that

$$d_{ij}^{\leq k} = \min_{0 \leq \ell \leq n-1} \big(d_{i\ell}^{\leq (k-1)} + d_{\ell j}^{1} \big). \tag{11.1}$$

Here we let $d + \infty = \infty + d = \infty + \infty = \infty$ for all integers $d \geq 0$. The recurrence (11.1) for the $d_{ij}^{\leq k}$ yields a simple algorithm for computing the distance matrix of a graph (Algorithm 11.6).

The matrix $D^{\leq 1}$ is essentially the adjacency matrix of the graph. It can therefore easily be computed in time $\Theta(n^2)$, both if the graph is given in adjacency

Algorithm simpleDist(G)

1. Compute $D^{\leq 1}$
2. **for** $k = 2$ **to** $n-1$ **do**
3. **for** $i = 0$ **to** $n-1$ **do**
4. **for** $j = 0$ **to** $n-1$ **do**
5. $d_{ij}^{\leq k} \leftarrow \infty$
6. **for** $\ell = 0$ **to** $n-1$ **do**
7. **if** $d_{i\ell}^{\leq (k-1)} + d_{\ell j}^{\leq 1} < d_{ij}^{\leq k}$ **then**
8. $d_{ij}^{\leq k} \leftarrow d_{i\ell}^{\leq (k-1)} + d_{\ell j}^{\leq 1}$
9. **return** $D^{\leq n-1}$

Algorithm 11.6

matrix representation and if it is given in adjacency list representation. Therefore, the running time of simpleDist is $\Theta(n^4)$ — which is rather bad.

The Floyd-Warshall algorithm uses a beautiful trick to make essentially the same idea work faster. The *interior vertices* of a path are all its vertices except the two endpoints. For $-1 \leq k \leq n$ and $0 \leq i, j \leq n-1$, let $d_{ij}^{(k)}$ be the minimum length of a path from i to j whose interior vertices are all in $\{0, \ldots, k\}$ (i and j themselves may be larger than k, though). If no such path exists, we let $d_{ij}^{(k)} = \infty$. Moreover, let $D^{(k)} = (d_{ij}^{(k)})_{0 \leq i,j \leq n-1}$.

Example 11.7. Consider the graph in Figure 11.8. We have

$$D^{(-1)} = \begin{pmatrix} 0 & \infty & \mathbf{1} & \infty & \mathbf{1} & \mathbf{1} & \infty \\ \mathbf{1} & 0 & \infty & \infty & \infty & \infty & \infty \\ \infty & \mathbf{1} & 0 & \infty & \infty & \mathbf{1} & \infty \\ \infty & \mathbf{1} & \infty & 0 & \infty & \infty & \mathbf{1} \\ \mathbf{1} & \infty & \infty & \infty & 0 & \mathbf{1} & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & \mathbf{1} & \infty & \mathbf{1} & 0 \end{pmatrix},$$

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 1 & \infty & 1 & 1 & \infty \\ 1 & 0 & \mathbf{2} & \infty & \mathbf{2} & \mathbf{2} & \infty \\ \infty & 1 & 0 & \infty & \infty & 1 & \infty \\ \infty & 1 & \infty & 0 & \infty & \infty & 1 \\ 1 & \infty & \mathbf{2} & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & \infty & 1 & 0 \end{pmatrix},$$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & 1 & \infty & 1 & 1 & \infty \\ 1 & 0 & 2 & \infty & 2 & 2 & \infty \\ \mathbf{2} & 1 & 0 & \infty & \mathbf{3} & 1 & \infty \\ \mathbf{2} & \mathbf{1} & \mathbf{3} & 0 & \mathbf{3} & \mathbf{3} & 1 \\ 1 & \infty & 2 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & \infty & 1 & 0 \end{pmatrix},$$

$$D^{(2)} = \begin{pmatrix} 0 & \mathbf{2} & 1 & \infty & 1 & 1 & \infty \\ 1 & 0 & 2 & \infty & 2 & 2 & \infty \\ 2 & 1 & 0 & \infty & 3 & 1 & \infty \\ 2 & 1 & 3 & 0 & 3 & 3 & 1 \\ 1 & \mathbf{3} & 2 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \infty & \infty & \infty & 1 & \infty & 1 & 0 \end{pmatrix},$$

$$D^{(3)} = \begin{pmatrix} 0 & 2 & 1 & \infty & 1 & 1 & \infty \\ 1 & 0 & 2 & \infty & 2 & 2 & \infty \\ 2 & 1 & 0 & \infty & 3 & 1 & \infty \\ 2 & 1 & 3 & 0 & 3 & 3 & 1 \\ 1 & 3 & 2 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ \mathbf{3} & \mathbf{2} & \mathbf{4} & 1 & \mathbf{4} & 1 & 0 \end{pmatrix},$$

$D^{(5)} = D^{(4)} = D^{(3)}$,

$$D^{(6)} = \begin{pmatrix} 0 & 2 & 1 & \infty & 1 & 1 & \infty \\ 1 & 0 & 2 & \infty & 2 & 2 & \infty \\ 2 & 1 & 0 & \infty & 3 & 1 & \infty \\ 2 & 1 & 3 & 0 & 3 & \mathbf{2} & 1 \\ 1 & 3 & 2 & \infty & 0 & 1 & \infty \\ \infty & \infty & \infty & \infty & \infty & 0 & \infty \\ 3 & 2 & 4 & 1 & 4 & 1 & 0 \end{pmatrix},$$

Observe that $D^{(n-1)} = D$ and $D^{(-1)} = D^{\leq 1}$. Moreover, for $0 \leq i, j, k \leq n-1$ we have

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}. \tag{11.2}$$

To see this, let P be a path of minimum length from i to j whose interior vertices are in $\{0, \ldots, k\}$. Then:

- Either vertex k does not appear on P at all, in which case P is actually a path whose interior vertices are in $\{0, \ldots, k-1\}$, and its length is $d_{ij}^{(k-1)}$,
- or vertex k does appear exactly once on P, in which case it can be split up into a path P_1 from i to k and a path P_2 from k to j. The interior vertices of both P_1 and P_2 are in $\{0, \ldots, k-1\}$, because k only appears once on P. Thus the length of P_1 is $d_{ik}^{(k-1)}$ and the length of P_2 is $d_{kj}^{(k-1)}$, and the length of P is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

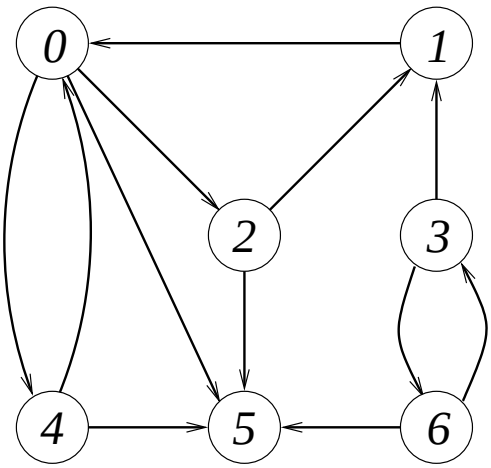

Figure 11.8.

Note that vertex k cannot appear more than once on a shortest path.

The recurrence (11.2) is used in the Floyd-Warshall algorithm (Algorithm 11.9). Note that, compared to simpleDist, we have saved one loop. This brings the running time down to $\Theta(n^3)$. As stated, the algorithm only computes a distance matrix. It is quite easy, however, to compute a parent matrix representing the shortest paths between all pairs from a distance matrix. It is also possible to enhance the algorithm so that it computes another matrix representing the paths (in a different form than a parent matrix does, though).

There is a JAVA implementation of the Floyd-Warshall algorithm available from the CS2 lecture notes web page. To save memory space, in this implementation we do not compute separate matrices for $D^{(-1)}, D^{(0)}, \ldots, D^{(n-1)}$, but just overwrite the previous matrix in each step of the iteration over k. (Why is this correct?)

Still, you may wonder what the point of the Floyd-Warshall algorithm is. It's running time is $\Theta(n^3)$, which is not better than the running time of a simple BFS-based algorithm for the all pairs shortest paths problem. Actually, it is worse for sparse graphs. An advantage of the Floyd-Warshall algorithm is its simplicity, which is particularly appealing if the input graph is represented by an adjacency matrix. A consequence of this simplicity is a small constant factor hidden in the big-Θ of the $\Theta(n^3)$ running time. A big advantage the Floyd-Warshall algorithm has over the BFS-based algorithm is that it also works for weighted graphs.

11.5 Shortest Paths in Weighted Graphs

Recall that in a *weighted graph* each edge has a real number called its *weight* assigned to it. In the setting of a shortest path problem, weights represent *distances*. For example, if the graph represents a map, then the weight of an edge

Algorithm floydWarshall(G)

1. Compute $D^{(-1)}$
2. **for** $k = 0$ **to** $n-1$ **do**
3. **for** $i = 0$ **to** $n-1$ **do**
4. **for** $j = 0$ **to** $n-1$ **do**
5. $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$
6. **if** $d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \geq d_{ij}^{(k-1)}$ **then**
7. $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$
8. **else**
9. $d_{ij}^{(k)} \leftarrow d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
10. **return** $D^{(n-1)}$

Algorithm 11.9

representing a segment of a road may be the length of the road segment. It is not hard to imagine that most "real-life" shortest path problems are set in this weighted scenario.

The *length* of a path in a weighted graph is the sum of the weights of its edges. With this notion of length, we can define the various shortest path problems as before.

The Floyd-Warshall algorithm works in the weighted case without any changes if we slightly re-define the matrix $D^{(-1)} = (d_{ij}^{(-1)})_{0 \leq i,j \leq n-1}$ by letting

$$d_{ij}^{(-1)} = \begin{cases} 0 & \text{if } i = j, \\ w & \text{if } i \neq j, \text{ and there is an edge from } i \text{ to } j, \\ & \text{and the weight of this edge is } w, \\ \infty & \text{otherwise.} \end{cases}$$

BFS-based shortest path algorithms do *not* work in the weighted case, as the following simple example illustrates: Consider the graph in Figure 11.10. Clearly the shortest path from vertex 0 to vertex 2 is the path through vertex 1 (of length 1.1). However, a BFS starting at vertex 0 would find the direct path from 0 to 2 first.

There is an algorithm for the single source shortest path problem in weighted graphs, known as *Dijkstra's algorithm*, whose basic mechanism is similar to breadth-first search. Instead of a queue, Dijkstra's algorithm uses a priority queue storing items whose elements are vertices and whose keys are the distances computed so far. But the ADS strand of CS2 ends here ...

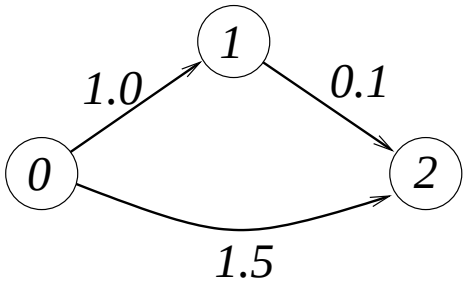

Figure 11.10.

Exercises

1. Explain why the asymptotic running time of DFS and BFS is $\Theta(n^2)$ if the input graph is given in adjacency matrix representation.

2. Run the Floyd-Warshall algorithm on the weighted graph diplayed in Figure 11.11. Compute all the matrices $D^{(i)}$, for $-1 \leq i \leq n-1$.

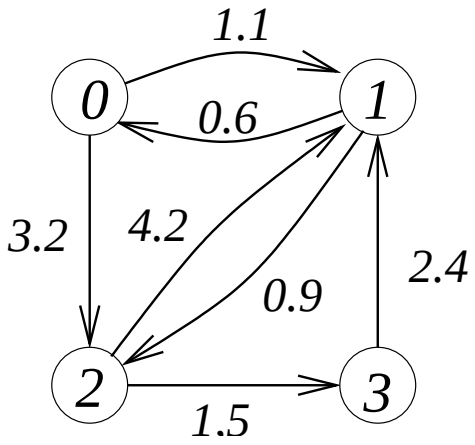

Figure 11.11.

Don Sannella