Lecture 9

Shortest Paths*

One of the most common operations in graphs is finding shortest paths between vertices. This section discusses three algorithms for this problem: breadth-first search for unweighted graphs, Dijkstra’s algorithm for weighted graphs, and the Floyd-Warshall algorithm for computing distances between all pairs of vertices.

Breadth-first search. We call a graph connected if there is a path between every pair of vertices. A (connected) component is a maximal connected subgraph. Breadth-first search, or BFS, is a way to search a graph. It is similar to depth-first search, but while DFS goes as deep as quickly as possible, BFS is more cautious and explores a broad neighborhood before venturing deeper. The starting point is a vertex \( s \). An example is shown in Figure 1. As before, we call an edge a tree edge if it is traversed by the algorithm. The tree edges define the BFS tree, which we can use to redraw the graph in a hierarchical manner, as in Figure 2. In the case of an undirected graph, no non-tree edge can connect a vertex to an ancestor in the BFS tree. Why? We use a queue to turn

![Figure 1: A sample graph with eight vertices and ten edges labeled by breath-first search. The label increases from a vertex to its successors in the search.](image1)

![Figure 2: The tree edges in the redrawing of the graph in Figure 1 are solid, and the non-tree edges are dotted.](image2)

*These notes are extracted from the lecture notes on “Design and Analysis of Algorithms”, Fall 2005 by Herbert Edelsbrunner. Notes on Bellman-Ford and Johnson’s algorithms are added by Vladimir Kolmogorov.
the idea into an algorithm. First, the graph and the queue are initialized.

forall vertices \( i \) do 
\( V[i].d = \infty \) endfor;
\( V[s].d = 0 \);
MAKEQUEUE; ENQUEUE(s); SEARCH.

A vertex is processed by adding its unvisited neighbors to the queue. They will be processed in turn.

```void SEARCH
while queue is non-empty do
    i = DEQUEUE;
    forall outgoing edges \( ij \) do
        if \( V[j].d = \infty \) then
            \( V[j].d = V[i].d + 1; V[j].\pi = i; \)
            ENQUEUE(j)
        endif
    endfor
endwhile.
```

The label \( V[i].d \) assigned to vertex \( i \) during the traversal is the minimum number of edges of any path from \( s \) to \( i \). In other words, \( V[i].d \) is the length of the shortest path from \( s \) to \( i \). The running time of BFS for a graph with \( n \) vertices and \( m \) edges is \( O(n+m) \).

**Single-source shortest path.** BFS can be used to find shortest paths in unweighted graphs. We now extend the algorithm to weighted graphs. Assume \( V \) and \( E \) are the sets of vertices and edges of an undirected graph with a positive weighting function \( w : E \rightarrow \mathbb{R}_+ \). The length or weight of a path is the sum of the weights of its edges. The distance between two vertices is the length of the shortest path connecting them. For a given source \( s \in V \), we study the problem of finding the distances and shortest paths to all other vertices. Figure 3 illustrates the problem by showing the shortest paths to the source \( s \). In the non-degenerate case, in which no two paths have the same length, the union of all shortest paths to \( s \) is a tree, referred to as the shortest path tree. In the degenerate case, we can break ties such that the union of paths is a tree.

![Figure 3: The bold edges form shortest paths and together the shortest path tree with root s. It differs by one edge from the breadth-first tree shown in Figure 1.](image-url)
As before, we grow a tree starting from $s$. Instead of a queue, we use a priority queue to determine the next vertex to be added to the tree. It stores all vertices not yet in the tree and uses $V[i].d$ for the priority of vertex $i$. First, we initialize the graph and the priority queue.

\[
V[s].d = 0; \text{ INSERT}(s);
\]

forall vertices $i \neq s$ do
\[
V[i].d = \infty; \text{ INSERT}(i)
\]
endfor.

After initialization the priority queue stores $s$ with priority 0 and all other vertices with priority $\infty$.

**Dijkstra’s algorithm.** We mark vertices in the tree to distinguish them from vertices that are not yet in the tree. The priority queue stores all unmarked vertices $i$ with priority equal to the length of the shortest path that goes from $i$ in one edge to a marked vertex and then to $s$ using only marked vertices.

while priority queue is non-empty do
\[
i = \text{EXTRACTMIN}; \text{ mark } i;
\]
forall neighbors $j$ of $i$ do
\[
\text{if } j \text{ is unmarked then}
\]
\[
V[j].d = \min \{w(ij) + V[i].d, V[j].d\}
\]
\[
\text{DECREASEKEY}(j)
\]
endif
endfor
endwhile.

Table 1 illustrates the algorithm by showing the information in the priority queue after each iteration of the while-loop operating on the graph in Figure 3. The marking mechanism is not necessary but clarifies the process. The algorithm performs $n$ **EXTRACTMIN** operations and at most $m$ **DECREASEKEY** operations. We compare the running time under three different data structures used to represent the priority queue. The first is a linear array as originally proposed by Dijkstra, the second is a heap, and the third is a Fibonacci heap. The results are shown in Table 2. We get the best result with Fibonacci heaps for which the total running time is $O(n \log n + m)$.

<table>
<thead>
<tr>
<th></th>
<th>$s$</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$\infty$</td>
<td>10</td>
<td>10</td>
<td>$\infty$</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 1: Each column shows the contents of the priority queue. Time progresses from left to right.
Correctness. It is not entirely obvious that Dijkstra’s algorithm indeed finds the shortest paths to s. To show that it does, we inductively prove that it maintains the following two invariants. At every moment in time

(A) $V[i].d$ is the length of the shortest path from s to i, for every marked vertex i.

(B) $V[j].d$ is the length of the shortest path from s to j that uses only marked vertices other than j, for every unmarked vertex j.

PROOF. Invariant (B) is true at the beginning of Dijkstra’s algorithm. To show that it is maintained throughout the process, we need to make sure that shortest paths are computed correctly. Specifically, if we assume Invariant (A) for vertex i then the algorithm correctly updates the priorities $V[j].d$ of all neighbors j of i, and no other priorities change.

At the moment vertex i is marked, it minimizes $V[j].d$ over all unmarked vertices j. Suppose that, at this moment, $V[i].d$ is not the length of the shortest path from s to i. Because of Invariant (B), there is at least one other unmarked vertex on the shortest path. Let the last such vertex be y, as shown in Figure 4. But then $V[y].d < V[i].d$, which is a contradiction to the choice of i.

We used (A) to prove (B) and (B) to prove (A). To make sure we did not create a circular argument, we parametrize the two invariants with the number k of vertices that are marked and thus belong to the currently constructed portion of the shortest path tree. To prove (Bk) we need (Ak) and to prove (Ak) we need (Bk−1). Think of the two invariants as two recursive functions, and for each pair of calls, the parameter decreases by one and thus eventually becomes zero, which is when the argument arrives at the base case.
Bellman-Ford algorithm The Dijkstra algorithm works correctly only if all edge weights are non-negative. Now suppose that we have a directed graph in which some weights are allowed to be negative. The goal is to compute shortest paths from the source $s$ to all other vertices. Assume that the graph does not have negative-weight cycles reachable from $s$. Then the shortest distances can be computed via the Bellman-Ford algorithm:

\[
\begin{align*}
\text{set } & V[s].d = 0 \text{ and } V[i].s = \infty \text{ for nodes } i \neq s \\
\text{for iter } = 1 \text{ to } n - 1 \text{ do} & \\
& \text{for each edge } (i, j) \text{ do} \\
& \quad V[j].d = \min\{V[j].d, V[i].d + w(i, j)\} \\
& \text{endfor} \\
& \text{endfor} \\
\end{align*}
\]

If the graph does contain a negative cycle reachable from $s$ then shortest paths from $s$ to nodes in this cycle are not well-defined. Let us show the algorithm’s correctness, assuming that there are no such cycles. We claim that the following holds:

- At any point $V[i].d$ is greater or equal than the shortest distance from $s$ to $i$.
- After $k$ iterations $V[i].d$ is smaller or equal than the shortest distance from $s$ to $i$ of a path of length at most $k$.

This can be proved using induction on $k$ (we omit details). The length of a shortest path from $s$ to any other vertex does not exceed $n - 1$; therefore, after $n - 1$ iterations we obtain correct distances. The algorithm’s complexity is $O(mn)$.

We can also use this algorithm to test whether the graph contains a negative cycle reachable from $s$. Namely, after $n - 1$ iterations we can run one more iteration of the Bellman-Ford algorithm; the distances will change if and only if the graph contains a negative cycle reachable from $s$.

All-pairs shortest paths: Floyd-Warshall algorithm We can run Dijkstra’s algorithm $n$ times, once for each vertex as the source, and thus get the distance between every pair of vertices. The running time is $O(n^2 \log n + nm)$ which, for dense graphs, is the same as $O(n^3)$. Cubic running time can be achieved with a much simpler algorithm using the adjacency matrix to store distances. This algorithm also works for negative weights, assuming that there are no negative cycles in the graph.

The idea is to iterate $n$ times, and after the $k$-th iteration, the computed distance between vertices $i$ and $j$ is the length of the shortest path from $i$ to $j$ that, other than $i$ and $j$, contains only vertices of index $k$ or less.

\[
\begin{align*}
\text{for } k = 1 \text{ to } n \text{ do} & \\
& \text{for } i = 1 \text{ to } n \text{ do} \\
& \quad \text{for } j = 1 \text{ to } n \text{ do} \\
& \quad \quad A[i, j] = \min\{A[i, j], A[i, k] + A[k, j]\} \\
& \quad \text{endfor} \\
& \text{endfor} \\
& \text{endfor} \\
\end{align*}
\]

We note that the $k$-th iteration of the outer for-loop leaves row $k$ and column $k$ unchanged. We therefore do not have to use two arrays, computing the new from the
Table 3: Adjacency, or distance matrix of the graph in Figure 1. All blank entries store \( \infty \).

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td></td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td></td>
<td>4</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td>5</td>
<td></td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>10</td>
<td></td>
<td></td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Matrix of distances after one iteration of the outermost for-loop. The new information is shown in boldface.

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>9</td>
<td>14</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td></td>
<td></td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

undisturbed old matrix. We illustrate the algorithm by showing the adjacency, or distance matrix before the algorithm in Table 3 and after one iteration in Table 4. The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is \( O(n^3) \).

**Summary**  
The following table summarizes different algorithms and their complexities.

<table>
<thead>
<tr>
<th></th>
<th>non-negative weights</th>
<th>arbitrary weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>single source</td>
<td>Dijkstra ( O(n \log n + m) )</td>
<td>Bellman-Ford ( O(mn) )</td>
</tr>
<tr>
<td>all pairs</td>
<td>Floyd-Warshall ( O(n^3) )</td>
<td>Floyd-Warshall ( O(n^3) )</td>
</tr>
<tr>
<td></td>
<td>repeated Dijkstra ( O(n^3 \log n + mn) )</td>
<td>Johnson ( O(n^2 \log n + mn) )</td>
</tr>
</tbody>
</table>

Johnson’s algorithm combines the ideas of Bellman-Ford and Dijkstra algorithms. First, add a new node \( s \) to the input graph \( G \), connect it to all other vertices via zero-weight edges, and compute shortest distances \( h(i) \) from \( s \) to \( i \) using Bellman-Ford algorithm. Node \( s \) is then removed.

Now let us define new edge weights \( A'[i, j] = A[i, j] + h(i) - h(j) \). These weights will be non-negative, assuming that there are no negative cycles in \( G \). Call the new graph \( G' \). Next, compute shortest distances \( d'(s, t) \) from \( s \) to \( t \) in \( G' \) using \( n \) applications of the Dijkstra algorithm. The shortest distances in the original graph \( G \) are now given by \( d(s, t) = d'(s, t) - h(s) + h(t) \). This follows from the following observation: for fixed \( s \) and \( t \), the lengths of any \( s \)-\( t \) paths in \( G \) and in \( G' \) differ by a constant, and so shortest paths in \( G \) and in \( G' \) coincide.