One of the most common operations in graphs is finding shortest paths between vertices. This section discusses three algorithms: 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 traverse a graph. It is similar to depth-first search, but while DFS goes as deep as quickly as possible, BFS explores a broad neighborhood before venturing deeper. The starting point is a vertex \( s \). An example is shown in Figure 90. As before, we call an edge a tree edge if it is traversed by the algorithm. The tree edges define the breadth-first tree, which we can use to redraw the graph in a hierarchical manner, as in Figure 91. In the case of an undirected graph, no non-tree edge can connect a vertex with an ancestor in the breadth-first tree. Why?

We use a queue to turn the idea of BFS into an algorithm. First, the graph and the queue are initialized.

```plaintext
forall vertices \( i \) do
    \( V[i].d = -1 \)
endfor;

\( V[s].d = 0; \)

MAKE_QUEUE; ENQUEUE(\( s \)); BFS.
```

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

```plaintext
void BFS
while queue is non-empty do
    \( i = \text{DEQUEUE}; \)
    forall each neighbor \( j \) of \( i \) do
        if \( V[j].d = -1 \) 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 \( i \) to \( s \). 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 \( w : E \to \mathbb{R}_+ \). 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 92 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 so this is true.

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 \( d[i] \) for the priority of vertex \( i \). First we initialize the graph and the priority queue.

\[
V[s].d = 0; \quad V[s].\pi = -1; \quad \text{INSERT}(s);
\]
\[
\text{forall vertices } i \neq s \text{ do } \quad V[i].d = \infty; \quad \text{INSERT}(i) \quad \text{endfor.}
\]

After initialization, the priority queue stores the source \( s \) 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.

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

Table 10 illustrates the algorithm by showing the information in the priority queue after each iteration of the while-loop operating on the graph in Figure 92. 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 11. We get the best result with Fibonacci heaps for which the total running time is \( O(n \log n + m) \).

**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.

**Shortest Path Lemma.** At every moment in time

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

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

**Proof.** Invariant (A) 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 (B) 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 \( i \) to \( s \). Because of Invariant (A), there must be at least one other unmarked vertex on the shortest path. Let the last such vertex be \( y \), as in Figure 93. But then \( V[y].d < V[i].d \), which is a contradiction to the choice of \( i \).

We used Invariant (B) to prove Invariant (A) and Invariant (A) to prove Invariant (B). 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 \( (A_k) \) we need \( (B_k) \) and to prove \( (B_k) \) we need \( (A_{k-1}) \). 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.

**All-pairs Shortest Paths.** We can run Dijkstra’s algorithm \( n \) times, once for each vertex being the source, and thus get the distance between every pair of vertices. The running time is \( O(n^2 \log n + nm) \), which is \( O(n^3) \) unless the graph is sparse. Cubic running time can be achieved with a much simpler algorithm using the adjacency matrix to store distances. The idea is to go through \( n \) iterations, 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 \( i, j \) contains only vertices of index \( k \) or less.

```plaintext
for k = 0 to n - 1 do
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
        endfor
    endfor
endfor.
```

The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is \( O(n^3) \). We illustrate the algorithm by showing the adjacency or distance matrix of the graph in Figure 92 before the algorithm in Table 12 and after one iteration of the algorithm in Table 13.

**Figure 93:** The vertex \( y \) is the last unmarked vertex on the dashed path that connects \( i \) to \( s \).

<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>10</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>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>5</td>
<td>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 12: 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>10</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>5</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>14</td>
<td>15</td>
<td>10</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>0</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>10</td>
<td>6</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 13: Matrix of distances after one iteration of the outermost for-loop.