14 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 57. As

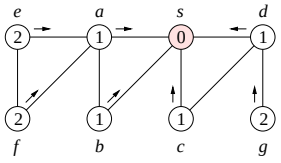

Figure 57: A sample graph with eight vertices and ten edges labeled by breath-first search. The label increases from a vertex to its successors in the search.

before, we call and 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 58. 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 the idea into an algorithm.

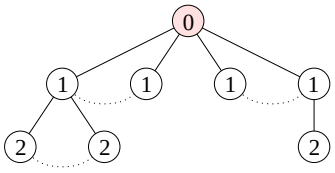

Figure 58: The tree edges in the redrawing of the graph in Figure 57 are solid, and the non-tree edges are dotted.

First, the graph and the queue are initialized.

```
forall vertices i do V[i].d = −1 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 neighbors j of i do
      if V[j].d = −1 then
        V[j].d = V[i].d + 1; V[j].π = 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 a simple, 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 59 illustrates the problem by showing the shortest paths to the source s. In

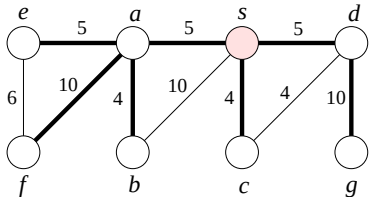

Figure 59: The bold edges form shortest paths and together the shortest path tree with root s. It differs by one edge from the breadth-first tree shown in Figure 57.

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.

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.

$V[s].d = 0$; $V[s].\pi = -1$; INSERT(s);
`forall` vertices $i \neq s$ `do`
  $V[i].d = \infty$; INSERT(i)
`endfor`.

After initialization the priority queue stores s with priority 0 and all other vertices with priority ∞.

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`
  $i =$ EXTRACTMIN; mark i;
  `forall` neighbors j of i `do`
    `if` j is unmarked `then`
      $V[j].d = \min\{w(ij) + V[i].d, V[j].d\}$
    `endif`
  `endfor`
`endwhile`.

Table 3 illustrates the algorithm by showing the information in the priority queue after each iteration of the while-loop operating on the graph in Figure 59. 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 4. We get the best result with Fibonacci heaps for which the total running time is O($n \log n + m$).

s	**0**							
a	∞	5	**5**					
b	∞	10	10	9	**9**			
c	∞	**4**						
d	∞	5	5	**5**				
e	∞	∞	∞	10	10	**10**		
f	∞	∞	∞	15	15	15	**15**	
g	∞	∞	∞	∞	15	15	15	**15**

Table 3: Each column shows the contents of the priority queue. Time progresses from left to right.

	array	heap	F-heap
EXTRACTMINS	n^2	$n \log n$	$n \log n$
DECREASEKEYS	m	$m \log m$	m

Table 4: Running time of Dijkstra's algorithm for three different implementations of the priority queue holding the yet unmarked vertices.

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) $V[j].d$ 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.

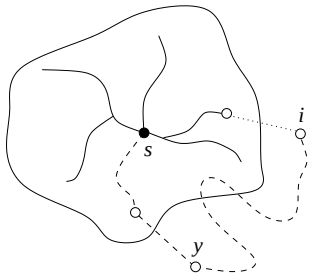

Figure 60: The vertex y is the last unmarked vertex on the hypothetically shortest, dashed path that connects i to s.

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 is at least one other unmarked vertex on the shortest path. Let the last such vertex be y, as shown in Figure 60. But then $V[y].d < V[i].d$, which is a contradiction to the choice of i. □

We used (B) to prove (A) and (A) to prove (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 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. 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.

```
for k = 1 to n do
  for i = 1 to n do
    for j = 1 to n do
      A[i,j] = min{A[i,j], A[i,k] + A[k,j]}
    endfor
  endfor
endfor.
```

The only information needed to update $A[i, j]$ during the k-th iteration of the outer for-loop are its old value and values in the k-th row and the k-th column of the prior adjacency matrix. This row remains unchanged in this iteration and so does this column. We therefore do not have to use two arrays, writing the new values right into the old matrix. We illustrate the algorithm by showing the adjacency, or distance matrix before the algorithm in Figure 61 and after one iteration in Figure 62.

	s	*a*	*b*	*c*	*d*	*e*	*f*	*g*
s	0	5	10	4	5			
a	5	0	4			5	10	
b	10	4	0					
c	4			0	4			
d	5			4	0			10
e		5				0	6	
f		10				6	0	
g					10			0

Figure 61: Adjacency, or distance matrix of the graph in Figure 57. All blank entries store ∞.

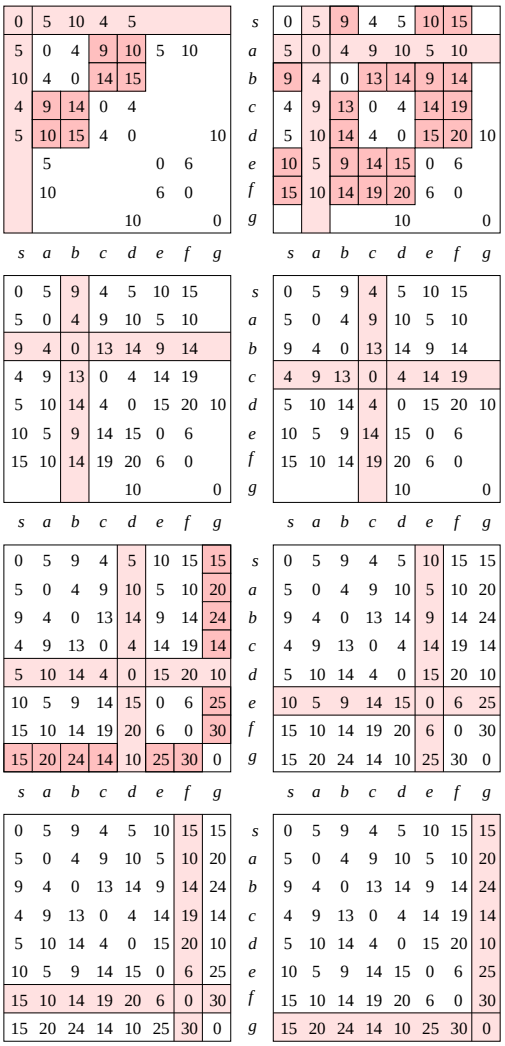

Figure 62: Matrix after each iteration. The k-th row and colum are shaded and the new, improved distances are high-lighted.

The algorithm works for weighted undirected as well as for weighted directed graphs. Its correctness is easily verified inductively. The running time is $O(n^3)$.