# **Shortest Path Algorithms**

**Types of Shortest Path Problems**

* A common graph operation is finding the shortest path between two vertices.
* The methodology to find the shortest path depends on the **type of graph**.
* There are two general types of shortest path algorithms:

1. **Shortest Path in Unweighted Graphs**
   1. **Unweighted Shortest-Path Problem**
      1. *O*(|*E*|+|*V*|) time
2. **Shortest Path in Weighted Graphs**
   1. **Weighted Shortest-Path Problem** assuming there ***are no negative edges***
      1. *O*(|*E*| log |*V*|) time
   2. **Weighted Shortest-Path Problem** assuming there ***are negative edges***
      1. *O*(|*E*| · |*V*|) time
   3. **Weighted Shortest-Path Problem** assuming an ***acyclic graph***
      1. *O*(|*V*|) time

## **Path Length**

**Path Definition**

* A ***path*** between two vertices is a sequence of edges that begins at one vertex and ends at another vertex.
* A ***path*** is a list of vertices **{*v*1, *v*2, *v*3,...,*vN}***such that

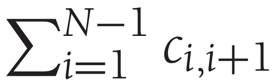
(*vi*, *vi*+1) ∈ *E* for 0 ≤ *i* < *N*

**Unweighted Shortest Path**

* The ***unweighted path* *length*** of a path is the **number of edges on the path**, namely, N − 1.

**Weighted Path Length**

* Each edge (*vi*, *vj*) in a weighted graph has an associated cost *ci*,*j* to traverse the edge.
* The ***weighted path length*** (total cost) of a path **{*v*1, *v*2, *v*3,...,*vN}***is



## **Single-Source Shortest-Path Problem**

### **Problem Statement**

* Given as input a weighted graph, *G* = (*V*, *E*), and a distinguished vertex, *s*, find the shortest weighted path from *s* to every other vertex in *G*.
* We can find the shortest path from ***vSource* to *vDestination***
* Finding the path from *s* to one vertex is just as fast than finding the path from *s* to all vertices.
* If we want the shortest path from all sources, the best algorithm is to rerun the single-source algorithm for all source vertices.

## **Unweighted Shortest Path**

* Our goal is to find the ***shortest path*** in an ***unweighted graph***.
* The shortest path can be found by counting the ***minimum number of edges*** between two vertices.
* The ***unweighted shortest-path*** problem is actually a ***special case of the weighted shortest-path problem***, since we could assign all **edges a weight of 1**.
* There are different techniques we can use to accomplish this task. One such technique for searching a graph is known as **breadth-first search**.

### **Breadth-First Search Technique**

* **Breadth-First Search** operates by processing vertices in layers:
  + The vertices closest to the *s* are evaluated first
  + The most distant vertices are evaluated last.

### **Example Graph**

* Input parameters
  + starting vertex, ***s***
  + unweighted graph, ***G***
* Goal
  + Find the ***unweighted shortest path*** from ***s***to all other vertices
* Suppose we choose *s* to be *v*3.

A picture containing watch, old

Description automatically generated

* Immediately, we can tell that the shortest path from *s* to *v*3 is then a path of length 0.
* We can mark this information, obtaining the graph in Figure 9.11.

A picture containing watch, old

Description automatically generated

* Now we can start looking for all vertices that are a distance 1 away from *s*.
* These can be found by looking at the vertices that are adjacent to *s*.
* If we do this, we see that *v*1 and *v*6 are one edge from *s*.
* This is shown in Figure 9.12.

A picture containing watch, old

Description automatically generated

* We can now find vertices whose shortest path from *s* is exactly 2, by finding all the vertices adjacent to *v*1 and *v*6 (the vertices at distance 1), whose shortest paths are not already known (a vertex we are trying to reach may have already been visited).
* This search tells us that the shortest path to *v*2 and *v*4 is 2.
* Figure 9.13 shows the progress that has been made so far.

A picture containing watch, old

Description automatically generated

* Finally, we can find, by examining vertices adjacent to the recently evaluated *v*2 and *v*4, that *v*5 and *v*7 have a shortest path of three edges.
* All vertices have now been calculated, and so Figure 9.14 shows the final result of the algorithm.

A picture containing old, watch

Description automatically generated

* For each vertex, we will keep track of three pieces of information.

1. First, we will keep its distance from *s* in the entry ***dv***.

Initially all vertices are unreachable except for *s*, whose path length is 0.

Table

Description automatically generated

Initial configuration of table used in unweighted shortest-path computation

1. The entry in ***pv***is the bookkeeping variable, which will allow us to print the actual paths.
2. The entry ***known***is set to true after a vertex is processed.

Initially, all entries are not *known,* including the start vertex.

When a vertex is marked *known,* we have a guarantee that no cheaper path will ever be found, and so processing for that vertex is essentially complete.

* The basic algorithm can be described in Figure 9.16.
* The algorithm in Figure 9.16 mimics the diagrams by declaring as *known* the vertices at distance *d* = 0, then *d* = 1, then *d* = 2, and so on, and setting all the adjacent vertices *w* that still have *dw* = ∞ to a distance *dw* = *d* + 1.
* By tracing back through the *pv* variable, the actual path can be printed.
* We will see how when we discuss the weighted case.
* The running time of the algorithm is *O*(|*V*|2), because of the doubly nested for loops. An obvious inefficiency is that the outside loop continues until NUM\_VERTICES − 1, even if all the vertices become *known* much earlier. Although an extra test could be made to avoid this, it does not affect the worst-case running time, as can be seen by generalizing what happens when the input is the graph in Figure 9.17 with start vertex *v*9.
* We can remove the inefficiency in much the same way as was done for topological sort.
* At any point in time, there are only two types of *unknown* vertices that have *dv* ̸= ∞.
* Some have *dv* = currDist and the rest have *dv* = currDist + 1.
* Because of this extra structure, it is very wasteful to search through the entire table to find a proper vertex.
* A very simple but abstract solution is to keep two boxes.
* Box #1 will have the unknown vertices with *dv* = currDist, and box #2 will have *dv* = currDist + 1. The test to find an appropriate vertex *v* can be replaced by finding any vertex in box #1. After updating *w* (inside the innermost if block), we can add *w* to box #2. After the outermost for loop terminates, box #1 is empty, and box #2 can be transferred to box #1 for the next pass of the for loop.
* We can refine this idea even further by using just one queue. At the start of the pass, the queue contains only vertices of distance currDist. When we add adjacent vertices of distance currDist + 1, since they enqueue at the rear, we are guaranteed that they will not be processed until after all the vertices of distance currDist have been processed. After the

1. We know that we can immediately access all vertices adjacent to the source vertex with a path of length one.
2. We then search the neighbors of all of these vertices to find paths of length two.
3. … and, so on, until all edges have been considered.

* This is much the same as a level-order traversal for trees.
* We can extend the idea to a graph
  + Instead of “child” in tree, use "neighbor" in graph (adjacent)
  + Keep track of paths
  + Find BFS ordering in those graphs
* For now, suppose we are interested only in the length of the shortest paths, not in the actual paths themselves.
* Therefore, we need to keep track of each path we traverse down.
* This is a matter of simple bookkeeping.

***Pseudocode for the algorithm***

Mark each vertex as unvisited.  
Put the source vertex on a queue q of vertices to visit.

Mark the source vertex as visited.  
While q is not empty:

Remove vertex v from front of queue.  
For each edge (**v**, **wi**) for which **v** is the source:

If **wi** has not been visited:

The shortest path to **wi** from the source goes through **v**.

The distance to **wi** is 1 plus the distance to **v**.  
Enqueue **wi** on q.  
Mark **wi** as visited.

***Runtime of Algorithm***

* Assuming that the graph is connected, each vertex is placed on the queue exactly once.
* (We mark the vertex as visited immediately after we put it on the queue and this prevents it from every being placed on the queue again.)
* When we examine each vertex, we consider each of its edges exactly once.
* This implies that the algorithm requires O(n + e) time for an adjacency list, where
  + n is the number of vertices

and

* + e is the number of edges
* This is O(e) if the graph is connected, since there must be at least O(n) edges in that case.
* For an adjacency matrix, this requires O(n2) time.

**Weighted Shortest Path Algorithms - *Smallest sum of edge weights***

* As opposed to unweighted graphs, which count the ***minimum*** ***number of edges*** to find the shortest pathbetween two given vertices, the shortest path in a weighted graph is the path that has the ***smallest sum of edge weights***.
* The **sum** of the weights of the edges of a path is called the path’s **length** or **weight** or **cost**.
* An edge ‘weight’ can be defined in many different ways
  + Distance in miles
  + Cost of each flight in dollars
  + Duration of each flight in hours

**Example**

* Consider a map of airline routes.
* A weighted directed graph can represent this map:
  + The vertices are cities
  + The edges indicate existing flights between cities
  + The edge weights represent the mileage between cities (vertices); as such, the weights are not negative.
* What is the shortest path from vertex 0 to vertex 1 in the graph in Figure 20-24a?
  + Though the most straightforward approach seems to be between vertex 0 -> 1 because it only requires one ‘step’ or a single ‘traversal’, it is not the shortest path because the path has a cost of 8.
  + The actual shortest path is from vertex 0 -> 4 -> 2 -> 1, with a cost of 7. While this may have 3 ‘steps’ it requires the least ‘cost’.

Diagram

Description automatically generated

# **Dijkstra’s Algorithm**

* The following algorithm, which is attributed to E. Dijkstra, determines the shortest paths between a given origin and *all* other vertices. The algorithm uses a set *vertexSet* of selected vertices and an array *weight,* where *weight*[*v*] is the weight of the shortest (cheapest) path from vertex 0 to vertex *v* that passes through vertices in *vertexSet*.

If *v* is in *vertexSet,* the shortest path involves only vertices in *vertexSet*. However, if *v* is not in *vertexSet,* then *v* is the only vertex along the path that is not in *vertexSet.* That is, the path ends with an edge from a vertex in *vertexSet* to *v*.

Initially, *vertexSet* contains only vertex 0, and *weight* contains the weights of the single-edge paths from vertex 0 to all other vertices. That is, *weight*[*v*] equals *matrix*[0][*v*] for all *v*, where *matrix* is the adjacency matrix. Thus, initially *weight* is the first row of *matrix*.

After this initialization step, you find a vertex *v* that is not in *vertexSet* and that minimizes *weight*[*v*]. You add *v* to *vertexSet*. For all (unselected) vertices *u* not in *vertexSet*, you check the values *weight*[*u*] to ensure that they are indeed minimums. That is, can you reduce *weight*[*u*]—the weight of a path from vertex 0 to vertex *u*—by passing through the newly selected vertex *v*?

To make this determination, break the path from 0 to *u* into two pieces and find their weights as follows:

*weight*[*v*] = weight of the shortest path from 0 to *v*

*matrix*[*v*][*u*] = weight of the edge from *v* to *u*

Then compare *weight*[*u*] with *weight*[*v*] + *matrix*[*v*][*u*] and let

*weight*[*u*] = the smaller of the values *weight*[*u*] and *weight*[*v*] + *matrix*[*v*][*u*]