Shortest Paths and Dijkstra's Algorithm

We have now seen breadth-first search and depth-first search, which allow us to explore graphs and determine which vertices are reachable from a particular source. Today, we are interested in finding *shortest paths* between vertices. Depth-first search can find paths that are unnecessarily long, so it is not suitable for this task. Breadth-first search, on the other hand, finds shortest paths in *unweighted* graphs, i.e., paths that use as few edges as possible.

However, in a weighted graph, the shortest path is not necessarily the one that uses the fewest edges. This motivates the study of shortest-path algorithms for weighted graphs, which minimize the total weight of the edges along a path.

1 Shortest Path Problems

Path lengths and weights. Previously, we have mostly been working with unweighted graphs. We will now consider **weighted graphs**, in which each edge has an associated real-valued weight. It is important to distinguish two properties of a path:

Definition: Path Length and Weight

- The **length** of a path is the number of edges in the path.
- The **weight** of a path is the sum of the weights of its edges.

For unweighted graphs, these are the same thing if we set every edge weight to 1. A shortest path in an unweighted graph minimizes its length, which is exactly what BFS computes. For weighted graphs, we instead want to minimize the path's weight, which may be different from minimizing its length.

Note that some books use the word "length" to refer to the sum of edge weights in a weighted graph. To avoid ambiguity, we will consistently use *length* for the number of edges and *weight* for the sum of edge weights.

Shortest-path weights. Given a weighted graph G and two vertices u and v, we want to find a path from u to v with minimum possible weight. We first introduce some notation.

Definition: Shortest path weight

Given a weighted graph G, we write

$$\delta_G(u, v)$$

(or simply $\delta(u, v)$ when G is clear) to denote the minimum weight of all paths between u and v. If v is not reachable from u, we define $\delta_G(u, v) = \infty$.

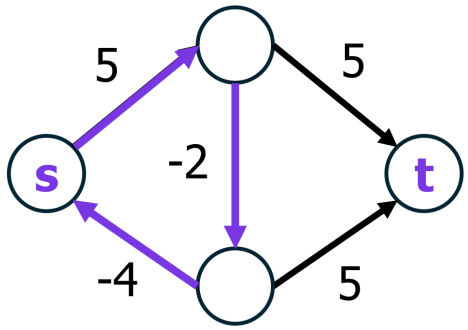
For the rest of our discussion, we will generally assume that graphs are weighted and directed unless stated otherwise.

One subtlety is that shortest paths are not always well-defined when negative edge weights are allowed. In particular, a reachable negative-weight cycle may allow us to decrease a path's weight without bound.

1.1 Negative weights and cycles

Should we allow edges to have negative weights? There is nothing inherently wrong with negative edge weights: a path containing negative-weight edges still has a well-defined minimum weight.

However, **negative-weight *cycles*** introduce a problem. Suppose there is a negative-weight cycle that is reachable from s and from which we can reach t. We could construct an s-t path that traverses this cycle any number of times. Each additional traversal would decrease the total weight of the path!



For example, in the graph above, we can traverse the purple cycle as many times as we would like, and each such traversal contributes -1 to the weight of the path. We can therefore make an s-t path with any arbitrarily low weight.

Consequently, there **does not exist** a minimum-weight s-t path: its weight could be made arbitrarily negative. We say that the shortest-path weight is *undefined* in this situation. Notice that a negative-weight cycle elsewhere in the graph does not necessarily affect the shortest path from s to t. The cycle must be reachable from s and be able to reach t.

We will first study algorithms that require non-negative edge weights, then return to algorithms that allow negative weights and can detect negative-weight cycles in the next lecture.

Can shortest paths contain cycles? Consider what happens when a path contains a cycle:

- A **positive-weight cycle** cannot belong to a shortest path, because removing the cycle would strictly decrease the path's weight.
- A **zero-weight cycle** could belong to a shortest path, but removing it preserves the total weight.
- A **negative-weight cycle** would allow us to decrease the path's weight by traversing it again, so no shortest path would exist between those endpoints.

This gives us the following useful property:

Theorem: Simple Shortest Paths

If a shortest path from s to t exists, then there is a **simple** shortest path from s to t, i.e., one that contains no repeated vertices.

To see why, take any shortest path and remove any cycles it contains. Such cycles cannot have positive weight, since removing them would produce a shorter path, and they cannot have negative weight, since then no minimum-weight path would exist. Therefore, any cycles must have weight zero, and removing them preserves the path's weight.

In particular, on a graph with n vertices, a simple path can contain at most $n-1$ edges. This observation will become important when we develop the Bellman–Ford algorithm.

1.2 Shortest Path Properties

Triangle Inequality A fundamental property of shortest paths is the **triangle inequality**.

Theorem: Triangle Inequality

For any vertices s, u, t for which the shortest-path weights are well-defined,

$$\delta(s,t) \leq \delta(s,u) + \delta(u,t).$$

Proof. Consider taking a shortest path from s to u, followed by a shortest path from u to t. This gives us a route from s to t whose weight is $\delta(s,u)+\delta(u,t)$. Since $\delta(s,t)$ is the minimum weight of any route from s to t, it cannot be larger than this value. □

In other words, traveling from s to t via another vertex u can never give a lower weight than the shortest possible path from s to t. This property holds even when some edges have negative weights, provided the shortest-path weights are well-defined.

Subpaths Property One final important property that we will use extensively next lecture is that every portion of a shortest path must itself be a shortest path.

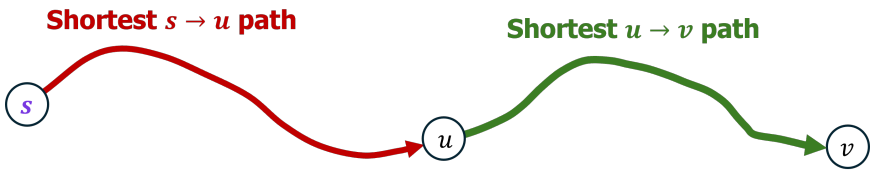
Theorem: Subpaths Property

Any contiguous subpath of a shortest path is itself a shortest path between its endpoints.

In particular, if u lies on a shortest path from s to t, then

$$\delta(s,t) = \delta(s,u) + \delta(u,t).$$

Proof. Suppose a shortest path from s to t passes through vertices u and v, but the portion from u to v is not a shortest path. We could replace that portion with a shorter path from u to v, producing a path from s to t with strictly smaller weight. This contradicts the assumption that the original path was shortest. □



This complements the triangle inequality, which gives an upper bound and holds for any intermediate vertex u, by showing that for particular vertices u, the inequality is exact.

The subpaths property holds even in the presence of negative edge weights, provided the shortest path exists. We will make extensive use of this property in the next lecture to develop shortest-path algorithms based on dynamic programming.

1.3 Variants of the shortest-path problem

There are three main types of shortest-path problems that we might want to solve on a graph:

Problem: Shortest Path Problems

- **Single-pair shortest path:** Given vertices s and t, find a shortest path from s to t.
- **Single-source shortest paths (SSSP):** Given a source vertex s, find the shortest paths from s to every other vertex in the graph.
- **All-pairs shortest paths (APSP):** Find the shortest paths between every pair of vertices s, t in the graph.

We will generally focus on computing the *weights* of the shortest paths, rather than explicitly constructing the paths themselves. For most shortest-path algorithms it is straightforward to also recover the actual paths by recording additional information, such as the predecessor of each vertex on a shortest path.

These problems are closely related; we can solve a single-pair problem using a single-source algorithm, and we can solve an all-pairs problem by running a single-source algorithm once for each vertex. However, there is clearly redundancy in doing so, so it is worthwhile to explore and design more efficient algorithms for each problem.

Different kinds of graphs. There are also many variations in the properties of the graphs on which we want to find shortest paths. For example:

- **Edge weights:** Are the edges unweighted, restricted to non-negative weights, or allowed to have negative weights? Are the weights integers or arbitrary real numbers?
- **Direction:** Is the graph directed or undirected?
- **Graph structure:** Does the graph have special properties, such as being acyclic or planar?
- **Graph density:** Is the graph sparse, with relatively few edges, or dense, with many edges?

Different combinations of these properties can lead to very different shortest-path algorithms and cost bounds. In general, unweighted shortest paths are easiest (BFS), non-negative weights are harder, and negative weights introduce further complications. Integer weights can be easier than arbitrary real weights. Similarly, undirected graphs and special graph structures, such as acyclic or planar graphs, can admit faster algorithms.

For example, single-source shortest paths on undirected graphs with positive integer weights can be solved in $O(n+m)$ work, just as efficiently as BFS! Similarly, on planar graphs with non-negative real weights, shortest paths can be computed in $O(n)$ work.

In contrast, for general directed graphs with real weights, we don't know any linear-work algorithms. Unlike BFS, we cannot simply process vertices in order of their path length. Instead, a natural approach is to repeatedly select a vertex with the smallest distance. This requires efficiently maintaining candidates ordered by path weight, typically using a priority queue or similar data structure which will add a logarithmic cost per operation.

We will now study **Dijkstra's algorithm**, the canonical algorithm for single-source shortest paths in directed graphs with non-negative real-valued edge weights. In the following lecture, we will develop algorithms that can also handle negative weights, such as the Bellman–Ford and Floyd–Warshall algorithms.

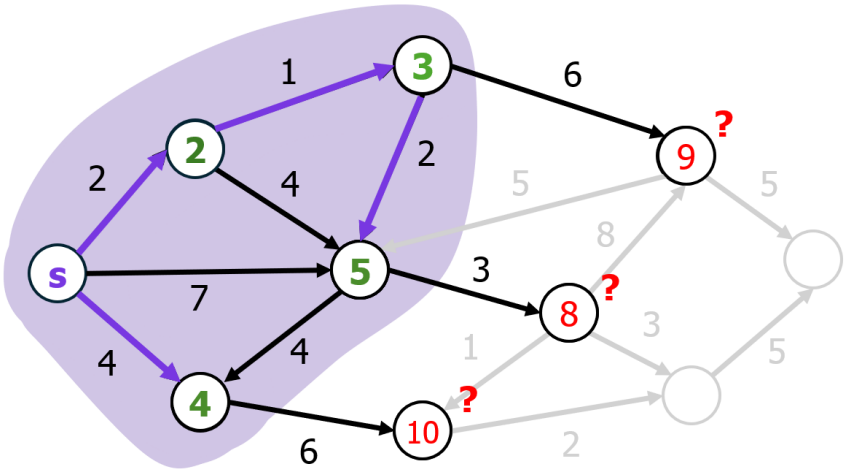
2 Dijkstra's Algorithm

The first algorithm we will look at is **Dijkstra's algorithm**, which solves the single-source shortest paths problem on graphs with *non-negative* edge weights. It is a sequential, greedy algorithm that returns the shortest-path weight from a given source s to every vertex in the graph.

The greedy idea. Dijkstra's algorithm is based on the following observation: if we already know the shortest paths from s to some vertices, we can find another shortest path by extending one of those paths by a single edge and choosing the shortest such extension.

Intuitively, this works because any alternative path would have to pass through some vertex whose shortest path we have not yet found. **Since all edge weights are non-negative, extending such a path cannot make its weight smaller**. Thus, the smallest candidate path must already be a shortest path. Note, very critically, that this property is only true because the graph contains no negative edge weights. Negative weights completely ruin this property, hence why it is generally more difficult to find shortest paths when they are present.

For example, consider the following graph. Suppose we already know the shortest paths to the vertices in the purple region, whose distances are labeled in green and whose shortest-path edges are highlighted in purple.

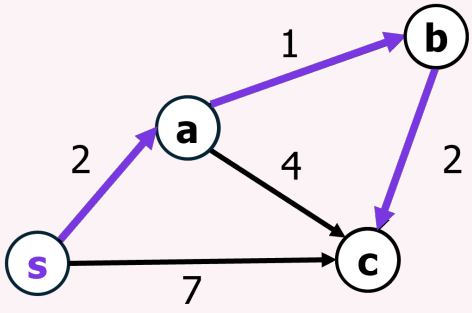


The algorithm considers extending each known shortest path by one additional edge. Among these candidate paths, the shortest reaches the vertex labeled with a distance estimate of 8, using the edge of weight 3 from the vertex whose distance is 5.

This is **guaranteed** to be a shortest path: any alternative route through the undiscovered region would first have to reach a vertex whose current distance estimate is at least 9. Since all remaining edge weights are non-negative, such a route cannot improve upon 8.

Remark: Distance estimates can improve many times

It is important to notice that the shortest path to a vertex **is not necessarily the first path to it that is discovered**. Consider the following vertices of the larger graph from above.



The vertex labeled c is discovered immediately as a neighbour of s, but the weight of the corresponding path would be 7. The algorithm then visits a at a distance of 2 and finds a *shorter* path to vertex c of total weight $2+4=6$. This is lower but still not the shortest path. After subsequently visiting b at distance 3, the algorithm then finds the path $s \to a \to b \to c$ with weight $2+1+2=5$, which ends up being the shortest path.

Thus, the distance estimate of c goes from ∞ to 7 to 6 to 5 before c itself is finally visited, at which point its distance estimate is discovered to be its true shortest path weight.

2.1 Implementation

Dijkstra's algorithm maintains an array `dist` of distance estimates and an array `finalized` indicating which vertices have been finalized, i.e., which vertices distances are known to be their exact shortest path weight.

At each step, the algorithm selects the vertex with the smallest distance estimate among those whose distances have not yet been finalized. It is therefore natural to use a **priority queue**, which allows us to efficiently retrieve the minimum element.

We represent the weighted graph using an adjacency list, where each entry (v, w) in `adj[u]` represents an edge from u to v with weight w. The priority queue then stores pairs (d, u), representing candidate paths to vertex u with weight d, ordered by their weights.

Algorithm: Dijkstra's Algorithm

```
type WeightedAdjList = sequence<sequence<(int, real)>>

fun dijkstra(adj: WeightedAdjList, s: int) -> sequence<real>:
  n = |adj|
  dist = [0 if u == s else ∞ for u in 0...n-1]
  finalized = [False for u in 0...n-1]

  pq: priority_queue<(real,int)> = empty
  pq.insert((0, s))

  while not pq.is_empty():
    d, u = pq.delete_min()

    if not finalized[u]:
      finalized[u] ← True
      for v, w in adj[u]:
        new_dist = d + w
        if new_dist < dist[v]:
          dist[v] ← new_dist
          pq.insert((new_dist, v))

  return dist
```

Updating distance estimates. When Dijkstra's algorithm finalizes a vertex u, it explores all if its outgoing edges. For each edge (u, v) with weight w, we have discovered a candidate path to v of weight $\text{dist}[u] + w$. If this improves the current estimate of v, we update `dist[v]` and insert the new candidate into the priority queue. This operation is called **relaxing** an edge.

As we saw in the earlier example, a vertex can have its estimate improved several times before being finalized. In our implementation, each improvement inserts another entry into the priority queue. For example, the vertex c from our earlier graph would have entries $(7, c)$, $(6, c)$, and $(5, c)$ in the priority queue simultaneously.

This is not a problem. Since the priority queue returns the smallest candidate first, $(5, c)$ will

be removed before the larger entries, and c will be finalized with the correct distance. When the outdated entries are eventually removed, the `finalized[c]` conditional check causes the algorithm to ignore them.

Remark: Alternate Implementations

An alternative to allowing multiple candidate paths for the same vertex to exist in the priority queue simultaneously is to maintain exactly one entry per vertex and *update its priority* whenever we find a shorter path. This would require a priority queue supporting an operation such as `decrease_key`.

Although this approach is common in textbooks, most standard-library priority queues do not support `decrease_key` (e.g., neither C++'s `std::priority_queue` nor Python's `heapq` support such an operation). Our implementation instead uses only `insert` and `delete_min`, which are supported by virtually every priority queue implementation.

Sequential execution. Lastly, notice that the algorithm uses mutable arrays and a mutable priority queue. The vertices are finalized one at a time, and the choice of the next vertex depends on the distance estimates produced by earlier iterations. Thus, Dijkstra's algorithm is entirely sequential. We will see some shortest path algorithms next lecture which expose opportunities for parallelism. The tradeoff, however, is that they cost much more work.

2.2 Analysis of Dijkstra's Algorithm

Theorem: Cost of Dijkstra's Algorithm

On a graph with n vertices and m edges, with no multi-edges, Dijkstra's algorithm takes $O(n + m \log n)$ work and span. If $m = \Omega(n)$, this simplifies to $O(m \log n)$.

Proof. The cost of Dijkstra's algorithm is dominated by the priority queue operations, together with the initialization of the arrays. An efficient priority queue supports `insert` and `delete_min` in $O(\log(|Q|))$ work, where $|Q|$ is the number of elements currently in the queue.

How many priority queue operations do we perform? Each vertex is finalized at most once, and when it is finalized, we examine each of its outgoing edges. An edge (u, v) can cause at most one insertion into the priority queue, when we discover a shorter path to v.

Therefore, there are at most $m + 1$ insertions (including the initial insertion of the source), and at most as many deletions. Furthermore, the priority queue can never contain more than $m + 1$ elements. Each priority queue operation thus costs $O(\log(m))$ work. Since the graph has no multi-edges, $m \leq n^2$, so this is $O(\log n)$. The total work for all priority queue operations is therefore $O(m \log n)$.

Finally, initializing the arrays costs $O(n)$ work, hence the overall work is

$$O(n + m \log n).$$

Since the algorithm is entirely sequential, its span is the same as its work. □

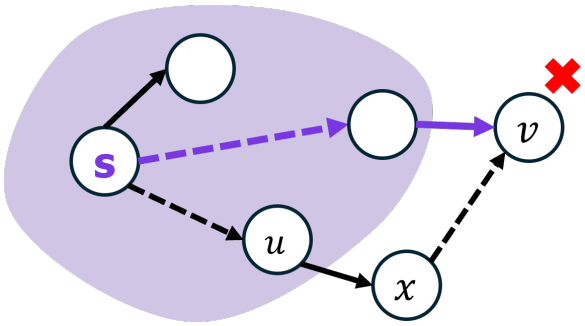
Theorem: Correctness of Dijkstra's Algorithm

On a directed graph with non-negative edge weights, Dijkstra's algorithm correctly computes the shortest-path weights from the source s to every vertex.

Proof. Assume for the sake of contradiction that Dijkstra's algorithm finalizes some vertex with an incorrect distance. Let v be the *first* such vertex. Since every finite distance estimate corresponds to the weight of some path from s, our estimates can never be smaller than the actual shortest-path weights. Therefore, we must have

$$\text{dist}[v] > \delta(s, v).$$

Consider a shortest path from s to v, and let x be the *first vertex on this path that has not yet been finalized* when v is selected. Let u be its predecessor on the path.



By our choice of v, every vertex finalized before v has the correct distance. In particular, u has already been finalized with $\text{dist}[u] = \delta(s, u)$.

When the algorithm explored u, it considered the edge (u, x). By the subpaths property, the path from s to x is itself shortest, so relaxing this edge ensures that $\text{dist}[x] = \delta(s, x)$. Furthermore, **since all edge weights are non-negative**, the prefix of the shortest path ending at x cannot have greater weight than the entire path:

$$\delta(s, x) \leq \delta(s, v).$$

But v was selected as the unfinalized vertex with the smallest distance estimate. Therefore,

$$\underbrace{\text{dist}[v] \leq \text{dist}[x]}_{\text{Greedy choice}} = \underbrace{\delta(s, x) \leq \delta(s, v)}_{\text{since non-negative weights}} < \text{dist}[v],$$

a contradiction! Thus, every vertex is finalized with its correct shortest-path distance. Any vertex that is unreachable from s is never discovered and retains distance ∞, which is also correct. □