# 14 Shortest Paths (November 8)

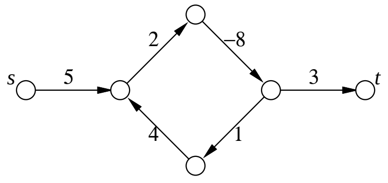
## 14.1 Introduction

Given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, a *source* $s$ and a *target* $t$, we want to find the shortest directed path from $s$ to $t$. In other words, we want to find the path $p$ starting at $s$ and ending at $t$ minimizing the function

$$w(p) = \sum_{e \in p} w(e).$$

For example, if I want to answer the question 'What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?', we might use a graph whose vertices are cities, edges are roads, weights are driving times, $s$ is Champaign, and $t$ is Columbus.[1] The graph is directed since the driving times along the same road might be different in different directions.[2]

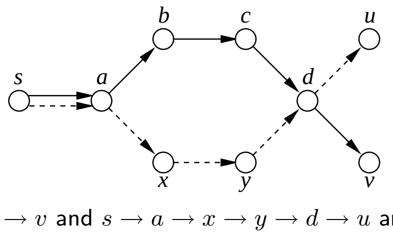
Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, since the presence of a negative cycle might mean that there is no shortest path. In general, a shortest path from $s$ to $t$ exists if and only if there is *at least one* path from $s$ to $t$, but there is no path from $s$ to $t$ that touches a negative cycle. If there is a negative cycle between $s$ and $t$, then se can always find a shorter path by going around the cycle one more time.



There is no shortest path from $s$ to $t$.

Every algorithm known for solving this problem actually solves (large portions of) the following more general *single source shortest path* or *SSSP* problem: find the shortest path from the source vertex $s$ to *every* other vertex in the graph. In fact, the problem is usually solved by finding a *shortest path tree* rooted at $s$ that contains all the desired shortest paths.

It's not hard to see that if the shortest paths are unique, then they form a tree. To prove this, it's enough to observe that sub-paths of shortest paths are also shortest paths. If there are multiple shortest paths to the same vertices, we can always choose one path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices $u$ and $v$ that diverge, then meet, then diverge again, we can modify one of the paths so that the two paths only diverge once.

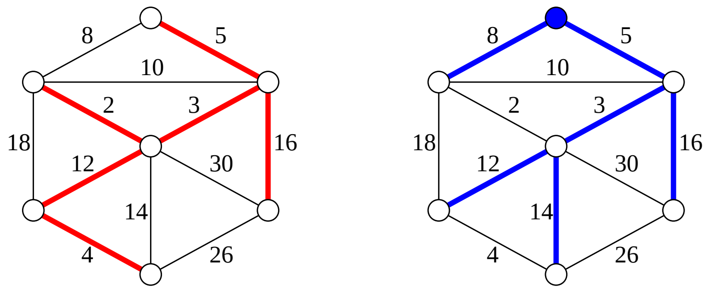


If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are both shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path.

[1]West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.

[2]There is a speed trap on I-70 just inside the Ohio border, but only for eastbound traffic.

I should emphasize here that shortest path trees and minimum spanning trees are usually very different. For one thing, there is only one minimum spanning tree, but in general, there is a different shortest path tree for every source vertex.



A minimum spanning tree and a shortest path tree (rooted at the topmost vertex) of the same graph.

All of the algorithms I'm describing in this lecture also work for undirected graphs, with some slight modifications. Most importantly, we must specifically prohibit alternating back and forth across the same undirected negative-weight edge. Our unmodified algorithms would interpret any such edge as a negative cycle of length 2.

## 14.2 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm. Each vertex $v$ in the graph stores two values, which describe a *tentative* shortest path from $s$ to $v$.

- $\text{dist}(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path.
- $\text{pred}(v)$ is the predecessor of $v$ in the tentative shortest $s \rightsquigarrow v$ path.

Notice that the predecessor pointers automatically define a tentative shortest path tree. We already know that $\text{dist}(s) = 0$ and $\text{pred}(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $\text{dist}(v) = \infty$ and $\text{pred}(v) = \text{NULL}$ to indicate that we do not know of *any* path from $s$ to $v$.

We call an edge $u \to v$ *tense* if $\text{dist}(u) + w(u \to v) < \text{dist}(v)$. If $u \to v$ is tense, then the tentative shortest path $s \rightsquigarrow v$ is incorrect, since the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX($u \to v$):
    $\text{dist}(v) \leftarrow \text{dist}(u) + w(u \to v)$
    $\text{pred}(v) \leftarrow u$
```

If there are no tense edges, our algorithm is finished, and we have our desired shortest path tree.

The correctness of the relaxation algorithm follows directly from three simple claims:

1. If $\text{dist}(v) \neq \infty$, then $\text{dist}(v)$ is the total weight of the predecessor chain ending at $v$:
$$s \to \cdots \to \text{pred}(\text{pred}(v)) \to \text{pred}(v) \to v.$$
This is easy to prove by induction on the number of relaxation steps. (Hint, hint.)

2. If the algorithm halts, then $\text{dist}(v) \leq w(s \rightsquigarrow v)$ for *any* path $s \rightsquigarrow v$. This is easy to prove by induction on the number of edges in the path $s \rightsquigarrow v$. (Hint, hint.)

3. The algorithm halts if and only if there is no negative cycle reachable from $s$. The 'only if' direction is easy—if there is a reachable negative cycle, then after the first edge in the cycle is relaxed, the cycle *always* has at least one tense edge. The 'if' direction follows from the fact that every relaxation step reduces either the number of vertices with $\text{dist}(v) = \infty$ by 1 or reduces the sum of the finite shortest path lengths by some positive amount.

I haven't said anything about how we detect which edges can be relaxed, or what order we relax them in. In order to make this easier, we can refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a 'bag' of vertices, initially containing just the source vertex $s$. Whenever we take a vertex $u$ out of the bag, we scan all of its outgoing edges, looking for something to relax. Whenever we successfully relax and edge $u \to v$, we put $v$ in the bag.

```
InitSSSP($s$):
  dist($s$) ← 0
  pred($s$) ← Null
  for all vertices $v \neq s$
      dist($v$) ← $\infty$
      pred($v$) ← Null
```

```
GenericSSSP($s$):
  InitSSSP($s$)
  put $s$ in the bag
  while the bag is not empty
      take $u$ from the bag
      for all edges $u \to v$
          if $u \to v$ is tense
              Relax($u \to v$)
              put $v$ in the bag
```

Just as with graph traversal, using different data structures for the 'bag' gives us different algorithms. There are three obvious choices to try: a stack, a queue, and a heap. Unfortunately, if we use a stack, we have to perform $\Theta(2^V)$ relaxation steps in the worst case! (This is a problem in the current homework.) The other two possibilities are much more efficient.

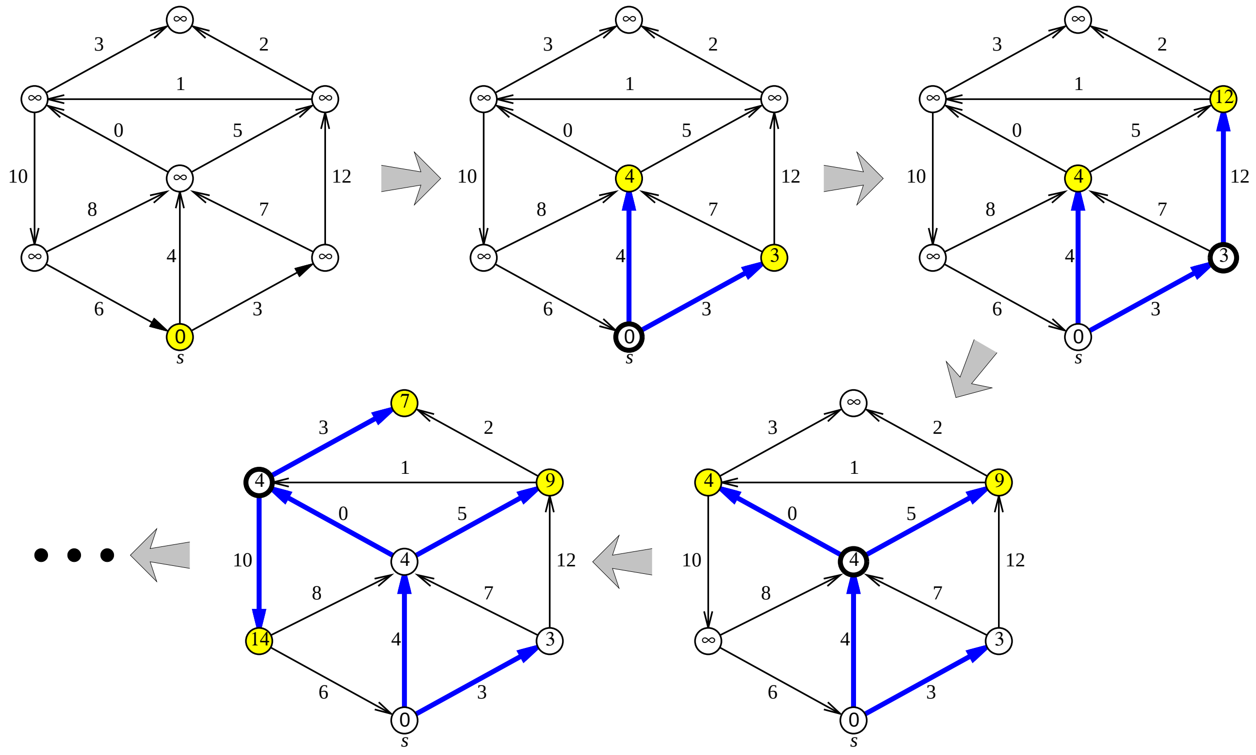
## 14.3 Dijkstra's Algorithm

If we implement the bag as a heap, where the key of a vertex $v$ is $\text{dist}(v)$, we obtain an algorithm first published by Leyzorek, Gray, Johnson, Ladew, Meaker, Petry, and Seitz in 1957, and then later independently rediscovered by Edsger Dijkstra in 1959. (A similar algorithm was also described by Dantzig in 1958.)

Dijkstra's algorithm, as it is universally known, is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from $s$. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from $s$, the algorithm performs a DecreaseKey operation every time an edge is relaxed. Thus, the algorithm performs at most $E$ DecreaseKeys. Similarly, there are at most $V$ Insert and ExtractMin operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boxed{O(E + V \log V)}$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here) is still *correct* if there are negative edges[3], but the worst-case running time could be exponential. (I'll leave the proof of this unfortunate fact as an extra credit problem.)

[3]The version of Dijkstra's algorithm presented in CLRS gives incorrect results for graphs with negative edges.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

## 14.4 The $A^*$ Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the $A^*$ algorithm, is frequently used to find a shortest path from a single source node $s$ to a single target node $t$. $A^*$ uses a black-box function GuessDistance$(v, t)$ that returns an estimate of the distance from $v$ to $t$. The only difference between Dijkstra and $A^*$ is that the key of a vertex $v$ is $\text{dist}(v) + \text{GuessDistance}(v, t)$.

The function GuessDistance is called *admissible* if GuessDistance$(v, t)$ never overestimates the actual shortest path distance from $v$ to $t$. If GuessDistance is admissible and the actual edge weights are all non-negative, the $A^*$ algorithm computes the actual shortest path from $s$ to $t$ at least as quickly as Dijkstra's algorithm. The closer GuessDistance$(v, t)$ is to the real distance from $v$ to $t$, the faster the algorithm. However, in the worst case, the running time is still $O(E + V \log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, $A^*$ can be used to solve puzzles (15-puzzle, Freecell, Shanghai, Minesweeper, Sokoban, Atomix, Rush Hour, Rubik's Cube, ...) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

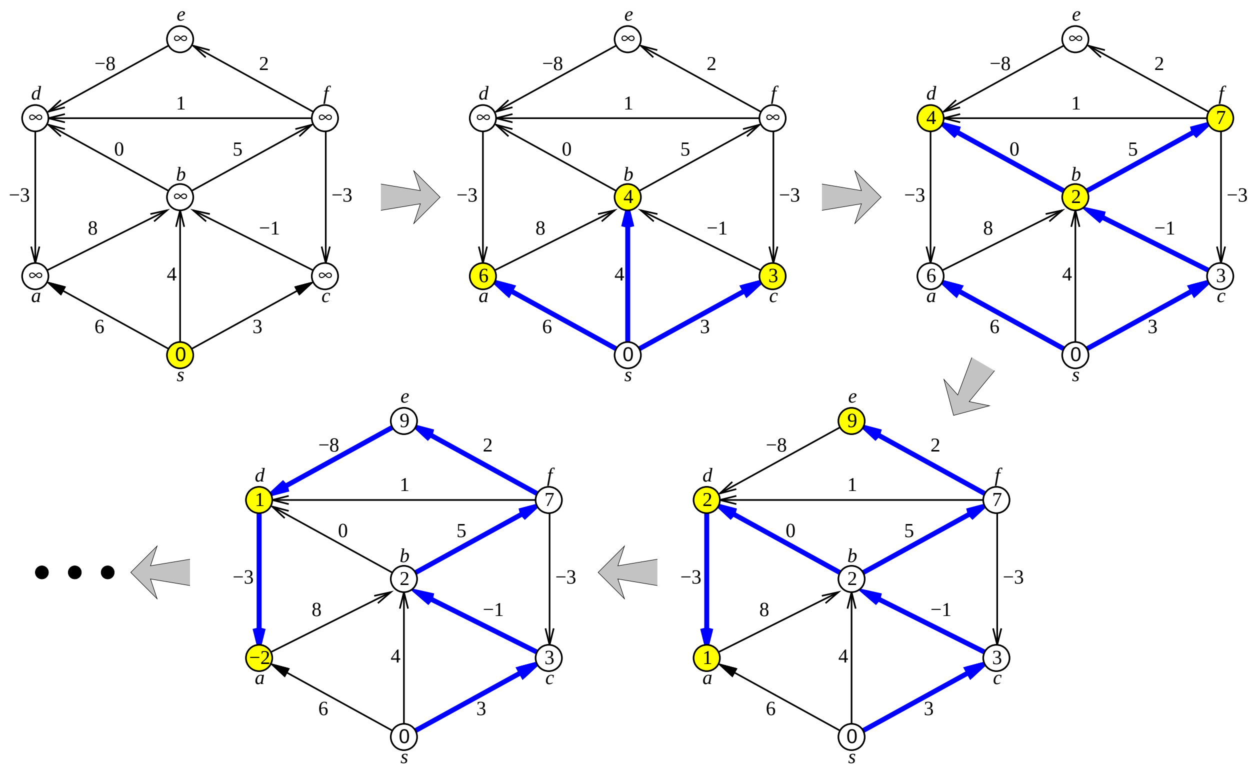
## 14.5 Shimbel's Algorithm ('Bellman-Ford')

If we replace the heap in Dijkstra's algorithm with a queue, we get an algorithm that was first published by Shimbel in 1955, then independently rediscovered by Moore in 1957, by Woodbury and Dantzig in 1957, and by Bellman in 1958. Since Bellman used the idea of relaxing edges, which was first proposed by Ford in 1956, this algorithm is usually called 'Bellman-Ford'. Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges, however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

The easiest way to analyze the algorithm is to break the execution into phases. Before we even begin, we insert a token into the queue. Whenever we take the token out of the queue, we begin a new phase by just reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex $s$. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant:

> At the end of the $i$th phase, for every vertex $v$, $\text{dist}(v)$ is less than or equal to the length of the shortest path $s \rightsquigarrow v$ consisting of $i$ or fewer edges.

Since a shortest path can only pass through each vertex once, either the algorithm halts before the $V$th phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boxed{O(VE)}$.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\, b\, c \diamond d\, f\, b \diamond a\, e\, d \diamond d\, a \diamond \diamond$, where $\diamond$ is the token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

Now that we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably. Instead of using a queue to perform a partial breadth-first search of the graph in each phase, let's just scan through the adjacency list directly and try to relax every edge in the graph.

```
SHIMBELSSSP($s$)
  INITSSSP($s$)
  repeat $V$ times:
      for every edge $u \to v$
          if $u \to v$ is tense
              RELAX($u \to v$)
  for every edge $u \to v$
      if $u \to v$ is tense
          return 'Negative cycle!'
```

This is closer to how CLRS presents the 'Bellman-Ford' algorithm. The $O(VE)$ running time of this version of the algorithm should be obvious, but it may not be clear that the algorithm is still correct. To prove correctness, we just have to show that our earlier invariant holds; as before, this can be proved by induction on $i$.

## 14.6 Greedy Shortest Paths?

Here's another algorithm that fits our generic framework, but which I've never seen analyzed.

Repeatedly relax the tensest edge.

Specifically, let's define the 'tenseness' of an edge $u \to v$ as follows:

$$\text{tenseness}(u \to v) = \max\{0, \text{dist}(v) - \text{dist}(u) - w(u \to v)\}$$

(This is defined even when $\text{dist}(v) = \infty$ or $\text{dist}(u) = \infty$, as long as we treat $\infty$ just like some indescribably large but finite number.) If an edge has zero tenseness, it's not tense. If we relax an edge $u \to v$, then $\text{dist}(v)$ decreases by the edge's tenseness.

Intuitively, we can keep the edges of the graph in some sort of heap, where the key of each edge is its tenseness. Then we repeatedly pull out the tensest edge $u \to v$ and relax it. Then we need to recompute the tenseness of other edges adjacent to $v$. Edges leaving $v$ possibly become more tense, and edges coming into $v$ possibly become less tense. So we need a heap that efficiently supports the operations INSERT, EXTRACTMAX, INCREASEKEY, and DECREASEKEY.

If there are no negative cycles, this algorithm eventually halts with a shortest path tree, but how quickly? Can the same edge be relaxed more than once, and if so, how many times? Is it faster if all the edge weights are positive? Hmm....