4.4.3 On Weighted Graph: Dijkstra's

If the given graph has edges with *different*[15] weights, the fast $O(V+E)$ and simple BFS does not work. This is because there can be a 'longer' path (in terms of number of vertices and edges involved in the path) that has smaller total weight than the 'shorter' path found by BFS. For example, in Figure 4.16—left, the shortest path from source vertex 0 to vertex 3 is not via direct edge $0 \rightarrow 3$ with weight 7 that is normally found by BFS, but a 'detour' path: $0 \rightarrow 1 \rightarrow 3$ with smaller total weight $2 + 3 = 5$ (see Figure 4.18—right).

To solve the SSSP problem on weighted graph, we use a *greedy* Edsger Wybe *Dijkstra*'s algorithm. There are several ways to implement this classic algorithm mentioned in various textbooks, e.g., [5, 35, 6]. In fact, Dijkstra's original paper that describes this algorithm [8] did not describe a specific implementation. We present two versions below.

On Non-Negative Weighted Graph: Original Dijkstra's

Dijkstra's algorithm starts with the standard initial condition for all SSSP algorithm. At the beginning, we only know `dist[s] = 0` (the shortest path from s to s itself is clearly 0) while `dist[u]` $= \infty$ for all other V-1 vertices that are not s. Dijkstra's algorithm uses a Priority Queue (`pq`) data structure of vertex information pair (`dist[u]`, u) to dynamically order (sort) the pairs by non-decreasing `dist[u]` values (vertex number u is unique). We insert V vertex information pairs of all V vertices into `pq` upfront and this already takes $O(V \log V)$ so far.

Dijkstra's algorithm will then process these vertices greedily: the vertex with the shortest `dist[u]` first (also see Section 3.4.1 about greedy algorithm with `pq` and **Exercise 4.4.3.4*** for a proof of correctness of this greedy strategy). Obviously at the start, the source vertex s (with the smallest possible `dist[s] = 0`) will be processed first while the rest (currently with unknown/infinitely large shortest path distance values) will be behind in `pq`. Then, Dijkstra's algorithm tries to relax each neighbor v of $u = s$. The `relax(u, v, w_u_v)` operation sets `dist[v] = min(dist[v], dist[u]+w_u_v)`. This opens up the possibilities of other shorter paths from vertex v to some other vertices as the shortest path distance values from source vertex s to v, i.e., `dist[v]`, will be lowered from initially ∞ into a (much) lower number. We also update (lower) that information in `pq` and let `pq` dynamically (re-)order the vertices based on non-decreasing `dist[u]` values.

Unfortunately, C++ STL `priority_queue`/Java `PriorityQueue`/Python `heapQ`—that has a Binary Heap data structure internally—does not have *built-in* capabilities (yet) to alter the key values *after* they are inserted into `pq`. Fortunately, we can get around this issue by using C++ STL `set`/Java `TreeSet`/OCaml `Set`—internally a balanced Binary Search Tree data structure—instead. With[16] C++ STL `set`/Java `TreeSet`/OCaml `Set`, we can update (lower) old (higher `dist[u]`, u) into new (lower `dist[u]`, u) by first deleting old (higher `dist[u]`, u) in $O(\log V)$ time and re-inserting new (lower `dist[u]`, u) also in $O(\log V)$ time.

Dijkstra's algorithm then repeats the same process until `pq` is empty: it greedily takes out vertex information pair (`dist[u]`, u) from the front of `pq` and relax each outgoing edge $u \rightarrow v$ of u, updating (lowering) `dist[v]` and the associated pair in `pq` if the edge relaxation is successful.

As each of the V vertices and each of the E edges are processed just once, the time complexity of Dijkstra's algorithm is $O((V+E)\log V)$. The extra $O(\log V)$ is for `pq` operations

[15]We have shown in Section 4.4.2 that the SSSP problem on weighted graph with constant weight C on all edges or weighted graph with only 0/1-weighted edges are still solvable with BFS.

[16]As of year 2020, Python standard library does not have built-in balanced BST equivalent yet. Hence, if you are a Python user, please use the Modified Dijkstra's version instead.

(we enqueue/dequeue V vertices into/from `pq`, respectively and we update (lower) shortest path values at most E times. Note: $O(\log E) = O(\log V^2) = O(2 \times \log V) = O(\log V)$).

To strengthen your understanding about this Dijkstra's algorithm, we show a step by step example of running this Dijkstra's algorithm on a small weighted graph and source vertex $s = 0$. Take a careful look at the content of `set<ii> pq` at each step.

1. Figure 4.16—left: At the beginning, only `dist[s] = dist[0] = 0`, `set<ii> pq` initially contains $\{(0, 0), (\infty, 1), (\infty, 2), (\infty, 3), (\infty, 4)\}$.

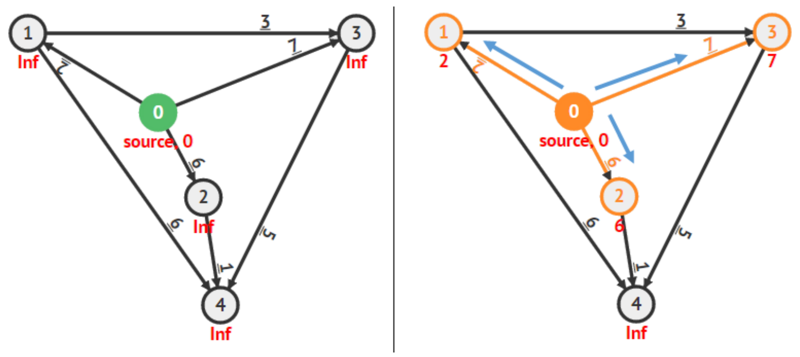

Figure 4.16: Dijkstra's Animation on a Weighted Graph (from UVa 00341 [44]), Steps 1+2

2. Figure 4.16—right: Dequeue the vertex information pair at the front of `pq`: (0, 0). Relax edges incident to vertex 0 to get `dist[1] = 2`, `dist[2] = 6`, and `dist[3] = 7`. While doing this, we simultaneously update (lower) the keys in `set<ii> pq`. `set<ii> pq` now contains $\{(2, 1), (6, 2), (7, 3), (\infty, 4)\}$.
3. Figure 4.17—left: Dequeue the vertex information pair at the front of `pq`: (2, 1). Relax edges incident to vertex 1 to get `dist[3] = min(dist[3], dist[1]+w(1,3)) = min(7, 2+3) = 5` and `dist[4] = 8` and update the keys in `pq`. `set<ii> pq` now contains $\{(5, 3), (6, 2), (8, 4)\}$. By now, edge $0 \rightarrow 3$ is not going to be part of the SSSP spanning tree from $s = 0$.

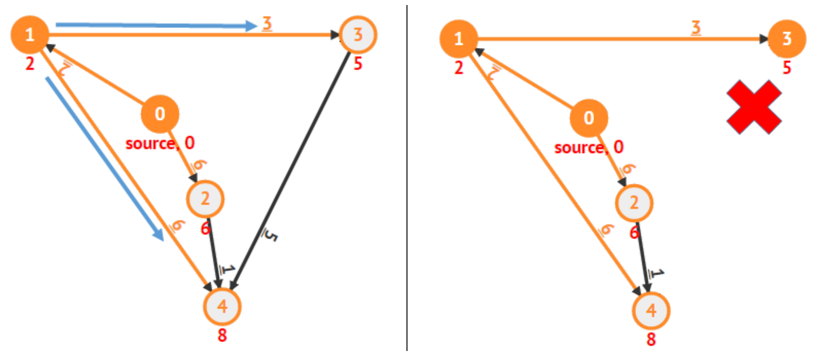

Figure 4.17: Dijkstra's Animation, Steps 3+4

4. Figure 4.17—right: We dequeue (5, 3) and try to do `relax(3, 4, 5)`, i.e., 5+5 = 10. But `dist[4] = 8` (from path $0 \rightarrow 1 \rightarrow 4$), so `dist[4]` is unchanged. `set<ii> pq` now contains $\{(6, 2), (8, 4)\}$. By now, edge $3 \rightarrow 4$ is also not going to be part of the SSSP spanning tree from $s = 0$.
5. Figure 4.18—left: We dequeue (6, 2) and do `relax(2, 4, 1)`, making `dist[4] = 7`. The shorter path from 0 to 4 is now $0 \rightarrow 2 \rightarrow 4$ instead of $0 \rightarrow 1 \rightarrow 4$. `set<ii> pq` now contains $\{(7, 4)\}$. By now, edge $1 \rightarrow 4$ is also not going to be part of the SSSP spanning tree from $s = 0$.

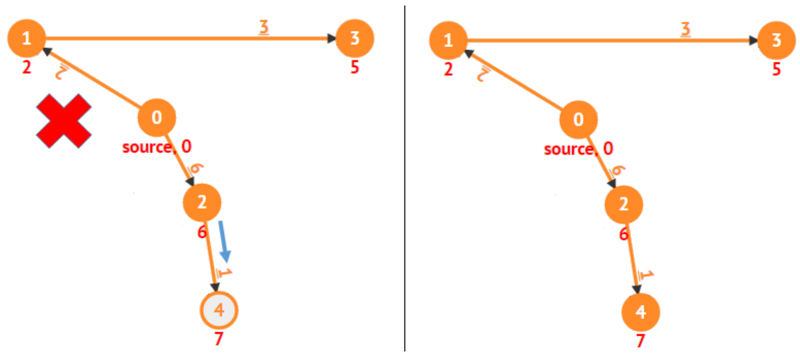

Figure 4.18: Dijkstra's Animation, Steps 5+6

6. Figure 4.18—right: Finally, (7, 4) is processed but nothing changes.
   `set<ii> pq` is now empty and Dijkstra's algorithm stops here.
   The final SSSP spanning tree describes the shortest paths from s to other vertices.

Our short C++ code is shown below and it looks very similar to Prim's algorithm and BFS code shown in Section 4.3.3 and 4.4.2, respectively. We call this implementation the *Original* Dijkstra's algorithm as we will *modify* them in the next subsection.

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;                  // INF = 1e9 here
  set<ii> pq;                                    // balanced BST version
  for (int u = 0; u < V; ++u)                    // dist[u] = INF
    pq.emplace(dist[u], u);                      // but dist[s] = 0

  // sort the pairs by non-decreasing distance from s
  while (!pq.empty()) {                          // main loop
    auto [d, u] = *pq.begin();                   // shortest unvisited u
    pq.erase(pq.begin());
    for (auto &[v, w] : AL[u]) {                 // all edges from u
      if (dist[u]+w >= dist[v]) continue;        // not improving, skip
      pq.erase(pq.find({dist[v], v}));           // erase old pair
      dist[v] = dist[u]+w;                       // relax operation
      pq.emplace(dist[v], v);                    // enqueue better pair
    }
  }

  for (int u = 0; u < V; ++u)
    printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
```

On Non-Negative Cycle Graph: Modified Dijkstra's

There is another way to implement Dijkstra's algorithm, especially for those who insist to use C++ STL `priority_queue`/Java `PriorityQueue`/Python `heapq` even though it does not have *built-in* capabilities (yet) to alter the key values *after* they are inserted into Priority Queue. Dijkstra's algorithm will only *lower* `dist[u]` values and never increase the values. This one sided update has an alternative Priority Queue solution.

To differentiate this Dijkstra's implementation with the previous one (called the *Original* Dijkstra's algorithm), we call this version as the *Modified* Dijkstra's algorithm.

Modified Dijkstra's algorithm works 99% similar with the Original Dijkstra's algorithm as it also maintains a Priority Queue (`pq`) that stores the same vertex information pairs. But this time, `pq` only contains one item initially: the base case $(0, s)$ which is true for the source vertex s. Then, Modified Dijkstra's implementation repeats the following similar process until `pq` is empty: it greedily takes out vertex information pair (d, u) from the front of `pq`. If the shortest path distance from s to u recorded in d is greater than `dist[u]`, it ignores u; otherwise, it processes u. The reason for this special check is shown below.

When this algorithm process u, it tries to relax each neighbor v of u. Every time it successfully relaxes an edge $u \rightarrow v$, it will *always enqueue* a pair (newer/shorter distance from s to v, v) into `pq` and will *always leave the inferior pair* (older/longer distance from s to v, v) inside `pq`. This is called as 'Lazy Deletion' and it causes *more than one copy* of the same vertex in `pq` with *different distances* from the source. That is why we have to process only the *first dequeued* vertex information pair which has the correct/shortest distance (other copies will have the outdated/longer distance). This Lazy Deletion technique works as the `pq` update operations in Modified Dijkstra's only *lower* the `dist[u]` values.

On non-negative weighted graph, the time complexity of this Modified Dijkstra's is identical with the Original Dijkstra's. Again, each vertex will only be processed once. Each time a vertex is processed, we try to relax its neighbors once (total E edges). Because of the Lazy Deletion technique, we may have up to $O(E)$ items in the `pq` at the same time, but this is still $O(\log E) = O(\log V)$ per each dequeue or enqueue operations. Thus, the time complexity remains at $O((V + E) \log V)$.

To strengthen your understanding about this Modified Dijkstra's algorithm, we show a *similar* step by step example of running this Modified Dijkstra's implementation on the same small weighted graph and $s = 0$. Just take a careful look at the content of `priority_queue<ii> pq` at each step that is different with the Original Dijkstra's version.

1. Figure 4.16—left: At the beginning, only `dist[s] = dist[0] = 0`,
   `priority_queue<ii> pq` initially contains {(0, 0)}.
2. Figure 4.16—right: Dequeue the vertex information pair at the front of `pq`: (0, 0).
   Relax edges incident to vertex 0 to get `dist[1] = 2`, `dist[2] = 6`, and `dist[3] = 7`.
   We always enqueue new vertex information pair upon a successful edge relaxation.
   `priority_queue<ii> pq` now contains {(2, 1), (6, 2), (7, 3)}.
3. Figure 4.17—left: Dequeue the vertex information pair at the front of `pq`: (2, 1).
   Relax edges incident to vertex 1 to get `dist[3] = min(dist[3], dist[1]+w(1,3)) = min(7, 2+3) = 5` and `dist[4] = 8` and immediately enqueue two more pairs in `pq`.
   `priority_queue<ii> pq` now contains {(5, 3), (6, 2), (7, 3), (8, 4)}.
   See that we have 2 entries of vertex 3 in `pq` with increasing distance from s. We do not immediately delete the inferior pair (7, 3) from the `pq` and rely on future iterations of our Modified Dijkstra's to correctly pick the one with minimal distance later, which is pair (5, 3). This is called as 'lazy deletion'.
   By now, edge $0 \rightarrow 3$ is not going to be part of the SSSP spanning tree from $s = 0$.
4. Figure 4.17—right: We dequeue (5, 3) and try to do `relax(3, 4, 5)`, i.e., 5+5 = 10.
   But `dist[4] = 8` (from path $0 \rightarrow 1 \rightarrow 4$), so `dist[4]` is unchanged.
   `priority_queue<ii> pq` now contains {(6, 2), (7, 3), (8, 4)}.
   By now, edge $3 \rightarrow 4$ is also not going to be part of the SSSP spanning tree from $s = 0$.
5. Figure 4.18—left: We dequeue (6, 2) and do `relax(2, 4, 1)`, making `dist[4] = 7`.
   The shorter path from 0 to 4 is now $0 \rightarrow 2 \rightarrow 4$ instead of $0 \rightarrow 1 \rightarrow 4$.
   `priority_queue<ii> pq` now contains {(7, 3), (7, 4), (8, 4)} (2 entries of vertex 4).
   By now, edge $1 \rightarrow 4$ is also not going to be part of the SSSP spanning tree from $s = 0$.

6. Figure 4.18—right: We do several bookkeeping at this step.
   We dequeue (7, 3) but ignore it as we know that its `d > dist[3]` (i.e., 7 > 5). This is when the actual deletion of the inferior pair (7, 3) is executed rather than at step 3 previously. By deferring it until now, the inferior pair (7, 3) is now located at the front of `pq` for the standard $O(\log V)$ deletion of C++ STL `priority_queue` to work.
   `priority_queue<ii> pq` now contains only {(7, 4), (8, 4)}.
   We then dequeue (7, 4) and process it, but nothing changes.
   `priority_queue<ii> pq` now contains only {(8, 4)}.
   Finally, we dequeue (8, 4) but ignore it again as its `d > dist[4]` (i.e., 8 > 7).
   `priority_queue<ii> pq` is now empty and the Modified Dijkstra's stops here.
   The final SSSP spanning tree describes the shortest paths from s to other vertices.

Our short C++ code is shown below and it is very identical with the Original Dijkstra's version. The main difference is the way both variants use Priority Queue data structures.

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;                 // INF = 1e9 here
  priority_queue<ii, vector<ii>, greater<ii>> pq;
  pq.emplace(0, s);

  // sort the pairs by non-decreasing distance from s
  while (!pq.empty()) {                         // main loop
    auto [d, u] = pq.top(); pq.pop();           // shortest unvisited u
    if (d > dist[u]) continue;                  // a very important check
    for (auto &[v, w] : AL[u]) {                // all edges from u
      if (dist[u]+w >= dist[v]) continue;       // not improving, skip
      dist[v] = dist[u]+w;                      // relax operation
      pq.emplace(dist[v], v);                   // enqueue better pair
    }
  }

  for (int u = 0; u < V; ++u)
    printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
```

Source code: `ch4/sssp/dijkstra.cpp|java|py|ml`

SSSP on Weighted Graph Variants

All SSSP on unweighted graph variants discussed in Section 4.4.2 are also applicable on weighted graph too, i.e., the SSSDSP variant (but only on non-negative weighted graph), the SDSP variant, the MSSP variant, Shortest Path Reconstruction, including solving the 0/1-weighted graph variant using the (slightly) slower Dijkstra's algorithm instead of BFS+deque. Next, we will discuss one other variant that is specific for weighted graphs.

SSSP on Non-Negative Cycle Graph

If the input graph has at least one (or more) negative edge weight(s), the Original Dijkstra's algorithm [5, 35, 6] will likely produce wrong answer as such negative edge weights violate the assumption required for the greedy algorithm to work (see **Exercise 4.4.3.4***). In Figure 4.19—left, we have a graph with one negative edge weight but no negative weight cycle—keep an eye on vertex 4 and edge $3 \to 4$.

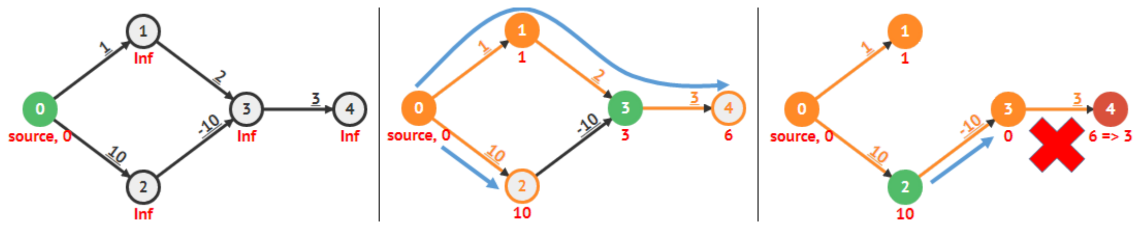

Figure 4.19: Original Dijkstra's Fails on a Negative Weight Graph, $s = 0$

In Figure 4.19—middle, we see that the Original Dijkstra's *wrongly* propagates shortest path distance from $0 \to 1 \to 3$ to vertex 4, causing vertex 4 to believe that the shortest path from source vertex 0 is $0 \to 1 \to 3 \to 4$ with value 6. In Figure 4.19—right, we see that the very last `relax(2, 3, -10)` operation causes the shortest path from source vertex 0 to vertex 3 to change into $0 \to 2 \to 3$ with value 10+(-10) = 0. Vertex 4 has no way to know this mistake as the Original Dijkstra's will stop as soon as the last vertex 2 is processed.

Note that if you run our current implementation of Original Dijkstra's on a graph like in Figure 4.19, you will get undefined behavior because C++ STL `set` encounters a problem when trying to erase the old vertex information pair. In the example above, `pq.find({dist[3], 3})` or `pq.find({10, 3})` will return `pq.end()` as pair `{10, 3}` is already processed and is no longer in the Priority Queue (`set`). Trying to erase this pair via the chained operation `pq.erase(pq.find({dist[3], 3}))` causes undefined behavior.

However, the Modified Dijkstra's algorithm will work just fine, albeit slower. This is because Modified Dijkstra's algorithm will keep inserting new vertex information pair into `pq` every time it manages to do a successful relax operation. Figure 4.19—middle and Figure 4.20—left depicts the same situation after identical initial steps between the Original and the Modified Dijkstra's. However, the next few actions of Modified Dijkstra's are different. Figure 4.20—middle, we see that vertex 3 is re-enqueued into `pq`. Figure 4.20—right, we see that vertex 3 now *correctly* propagates shortest path distance $0 \to 2 \to 3$ to vertex 4, causing vertex 4 to now have the correct shortest path of $0 \to 2 \to 3 \to 4$ of value 3.

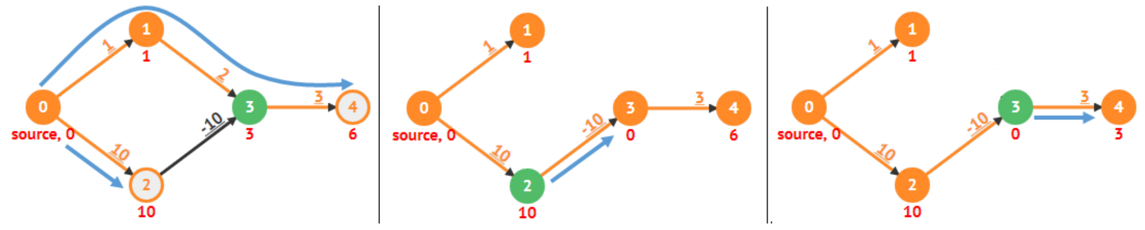

Figure 4.20: Modified Dijkstra's Can Work on a Non-Negative Cycle Graph, $s = 0$

If the weighted graph has no negative (weight) *cycle*, Modified Dijkstra's algorithm will keep propagating the shortest path distance information until there is no more possible relaxation (which implies that all shortest paths from the source have been found). However, when given a graph with negative weight *cycle*, the Modified Dijkstra's algorithm will hopelessly trapped in an infinite loop. Example: See the graph in Figure 4.22. Cycle $1 \to 2 \to 3 \to 1$ is a negative cycle with weight 15 + 0 + (-42) = -27. Modified Dijkstra's will keep looping forever as it is always possible to continue relaxing the edges along a negative cycle.

On graph with (a few) negative weight edges but no negative cycle, Modified Dijkstra's runs slower than $O((V+E)\log V)$ due to the need of re-processing already processed vertices but the shortest paths values will eventually be correct, unlike the Original Dijkstra's that stops after at most $O((V+E)\log V)$ operations but gives wrong answer on such a graph.

In either case, the early termination technique when the destination vertex t is also given in the SSSDSP variant will not work on such a graph.

However on an extreme case, we can actually setup a graph that has negative weights but no negative cycle that can significantly slow down Modified Dijkstra's algorithm, see Figure 4.21[17]. On such test case like in Figure 4.21, Modified Dijkstra's will first take the bottom path $0 \to 2 \to 4 \to 6 \to 8 \to 10$ with cost $0+0+0+0+0 = 0$ before finding $0 \to 2 \to 4 \to 6 \to 8 \to 9 \to 10$ with lower cost $0+0+0+0+1+(-2) = -1$ and so on until it explores all 2^5 possible paths from vertex 0 to vertex 10. It terminates with the correct answer of path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7 \to 8 \to 9 \to 10$ with cost -31. Each additional triangle (two more vertices and three more edges) in such a graph increases the runtime by twofold. Hence, Modified Dijkstra's can be made to run in exponential time. The difficulty of this test case for Modified Dijkstra's is best appreciated using a live animation so please also check VisuAlgo for the animation.

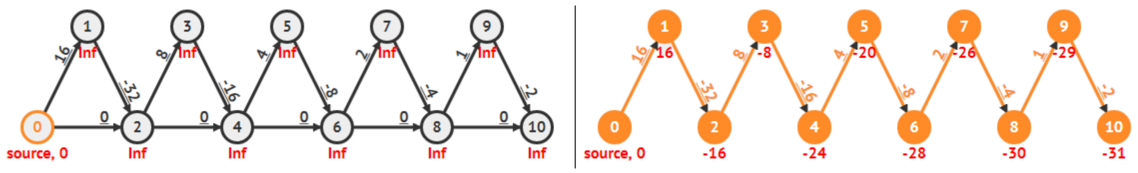

Figure 4.21: Modified Dijkstra's Can Be Made to Run in Exponential Time

---

Exercise 4.4.3.1: The source code for the Original Dijkstra's algorithm shown above uses `set<ii>` instead of `multiset<ii>`. What if there are two (or more) different vertices that have similar shortest path distance values from the source vertex s?

Exercise 4.4.3.2: The source code for the Modified Dijkstra's algorithm shown above uses `priority_queue<ii, vector<ii>, greater<ii>> pq;` to sort pairs of integers by increasing distance from source s. Can we get the same effect without defining comparison operator for the `priority_queue`? Hint: We have used similar technique with Kruskal's algorithm implementation in Section 4.3.2.

Exercise 4.4.3.3: The source code for the Modified Dijkstra's algorithm shown above has this important check `if (d > dist[u]) continue;`. What if that line is removed? What will happen to the Modified Dijkstra's algorithm?

Exercise 4.4.3.4*: Prove the correctness of Dijkstra's algorithm (both variants) on *non-negative* weighted graphs!

Exercise 4.4.3.5*: Dijkstra's algorithm (both variants) will run in $O(V^2 \log V)$ if run on a *complete* non-negative weighted graph where $E = O(V^2)$. Show how to modify Dijkstra's implementation so that it runs in $O(V^2)$ instead such complete graph! Hint: Avoid PQ.

Profile of Algorithm Inventor

Edsger Wybe Dijkstra (1930-2002) was a Dutch computer scientist. One of his famous contributions to computer science is the shortest path-algorithm known as **Dijkstra's algorithm** [8]. He does not like 'GOTO' statement and influenced the widespread deprecation of 'GOTO' and its replacement: structured control constructs. One of his famous Computing phrase: "two or more, use a for".

[17]This test case is contributed by a Competitive Programming Book reader: Francisco Criado.

4.4.4 On Small Graph (with Negative Cycle): Bellman-Ford

To solve the SSSP problem in the potential presence of negative weight *cycle(s)*, we can use the more generic (but slower) Bellman-Ford algorithm. This algorithm was invented by Richard Ernest *Bellman* (the pioneer of DP techniques) and Lester Randolph *Ford*, Jr (the same person who invented Ford-Fulkerson method for the Network Flow problem—discussed in Book 2). The main idea of this algorithm is simple: relax all E edges (in arbitrary order) V-1 times!

Initially `dist[s] = 0`, the base case. If we relax an edge (s, u), then `dist[u]` will have the correct value. If we then relax an edge (u, v), then `dist[v]` will also have the correct value. If we have relaxed all E edges V-1 times, then the shortest path from the source vertex to the furthest vertex from the source (which will be a simple path with V-1 edges) should have been correctly computed (see **Exercise 4.4.4.1*** for proof of correctness). The basic Bellman-Ford C++ code is very simple, simpler than BFS and Dijkstra's code:

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;                     // INF = 1e9 here
  for (int i = 0; i < V-1; ++i)                     // total O(V*E)
    for (int u = 0; u < V; ++u)                     // these two loops = O(E)
      if (dist[u] != INF)                           // important check
        for (auto &[v, w] : AL[u])                  // C++17 style
          dist[v] = min(dist[v], dist[u]+w);
```

The complexity of Bellman-Ford algorithm is $O(V^3)$ if the graph is stored as an Adjacency Matrix or $O(VE)$ if the graph is stored as an Adjacency List or Edge List. This is simply because if we use Adjacency Matrix, we need $O(V^2)$ to enumerate all the edges in our graph whereas it is just $O(E)$ using either Adjacency List or Edge List. Both time complexities are (much) slower compared to Dijkstra's and this is one of the main reason why we don't normally use Bellman-Ford to solve standard SSSP on weighted graph.

For some improvement, we can add a Boolean flag `modified = false` in the outermost loop (the one that repeats all E edges relaxation V-1 times). If at least one relaxation operation is done in the inner loops (the one that explores all E edges), set `modified = true`. We immediately break the outermost loop if variable `modified` is still false after all E edges have been examined. If this no-relaxation happens at the (outermost) loop iteration i, then there will be no further relaxation in iteration $i+1$, $i+2$, ..., $i = V$-1 either. This way, the time complexity of Bellman-Ford becomes $O(kV)$ where k is the number of iteration of the outermost loop. Note that k is still $O(V)$ though.

Bellman-Ford will never be trapped in an infinite loop even if the given graph has negative cycle(s). In fact, Bellman-Ford algorithm can be used to detect *the presence* of negative cycle (e.g., UVa 00558 - Wormholes) although such SSSP problem is ill-defined.

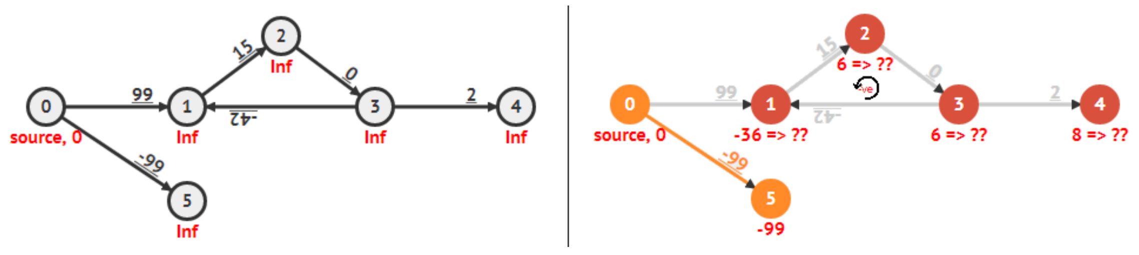

Figure 4.22: Bellman-Ford can detect the presence of negative cycle (UVa 00558 [44])

It can be proven (see **Exercise 4.4.4.1***) that after relaxing all E edges V-1 times, we should have solved the SSSP problem, i.e., we cannot relax any more edge. As the corollary: if we can still relax an edge, there must be at least one negative cycle in our weighted graph. This is a useful feature of the Bellman-Ford algorithm.

For example, in Figure 4.22—left, we see a simple graph with a negative cycle. After 1 pass, `dist[1] = 72` and `dist[2] = dist[3] = 114`. After V-1 = $6-1$ = 5 passes, `dist[1] = -36` and `dist[2] = dist[3] = 6` and Bellman-Ford algorithm stops. However, as there is a negative cycle, we can still do successful edge relaxations, e.g., we can still relax `dist[2] = -36+15 = -21`. This is lower than the current value of `dist[2] = 6`. The presence of a negative cycle (of weight 15+0-42 = -27) causes the vertices reachable from this negative cycle to have ill-defined shortest paths information. This is because one can simply traverse this negative cycle infinite number of times to make all reachable vertices from this negative cycle to have negative infinity shortest paths information. Notice that in Figure 4.22—right, vertex 4 is affected by the negative cycle whereas vertex 5 is not. The additional code to check for negative cycle *after* running the $O(VE)$ Bellman-Ford is shown below.

Our more complete Bellman-Ford C++ code is shown below. It shows Bellman-Ford with optimization and additional negative cycle check[18].

```
// inside int main()
  vi dist(V, INF); dist[s] = 0;                  // INF = 1e9 here
  for (int i = 0; i < V-1; ++i) {                // total O(V*E)
    bool modified = false;                       // optimization
    for (int u = 0; u < V; ++u)                  // these two loops = O(E)
      if (dist[u] != INF)                        // important check
        for (auto &[v, w] : AL[u]) {             // C++17 style
          if (dist[u]+w >= dist[v]) continue;    // not improving, skip
          dist[v] = dist[u]+w;                   // relax operation
          modified = true;                       // optimization
        }
    if (!modified) break;                        // optimization
  }

  bool hasNegativeCycle = false;
  for (int u = 0; u < V; ++u)                    // one more pass to check
    if (dist[u] != INF)
      for (auto &[v, w] : AL[u])                 // C++17 style
        if (dist[v] > dist[u]+w)                 // should be false
          hasNegativeCycle = true;               // if true => -ve cycle

  if (hasNegativeCycle)
    printf("Negative Cycle Exist\n");
  else {
    for (int u = 0; u < V; ++u)
      printf("SSSP(%d, %d) = %d\n", s, u, dist[u]);
  }
```

Source code: `ch4/sssp/bellman_ford.cpp|java|py|ml`

[18]There is another algorithm that can do negative cycle check: the $O(V^3)$ Floyd-Warshall algorithm applications that is discussed in Section 4.5.3.