Graph Applications: Dijkstra's Shortest Path Algorithm

CS 112 Spring 2020 – Apr 9

Shortest Path

Numerous graph applications need to know what is the shortest path from point A to point B.

If the graph does not have edge weights, this is not a hard problem to solve (think of using DFS or BFS)

But when the graph has edge weights (such as road maps with towns for vertices and distances in miles between towns), then it becomes a much harder problem

A brute force approach might find all possible paths between A and B, then find which one is the shortest, but this will be prohibitively slow for real world graphs with tens of thousands of vertices and edges

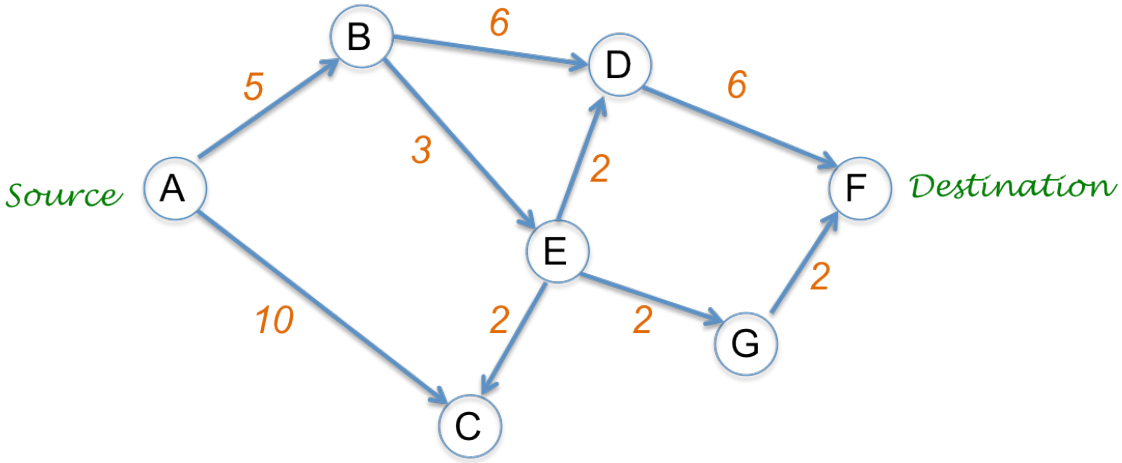
Dijkstra

Edsger Dijkstra came up with an algorithm to solve this problem in an efficient way (various other approaches had been proposed, but none had been fast enough to be practical)

Dijkstra's algorithm works on any graph, directed or undirected, with positive edge weights (real world graphs don't have negative weights)

Shortest Path Problem

We want to find the shortest path from point A to point F. In other words, A is the the *source* and F is the *destination*.



There are several paths from A to F:

A→B→D→F
(length 17)

A→B→E→D→F
(length 16)

A→B→E→G→F
(length 12)

Dijkstra's Algorithm

Dijkstra came up with an algorithm that is not brute force. Instead, he proposed an ingenious approach that proceeds by trial and error!

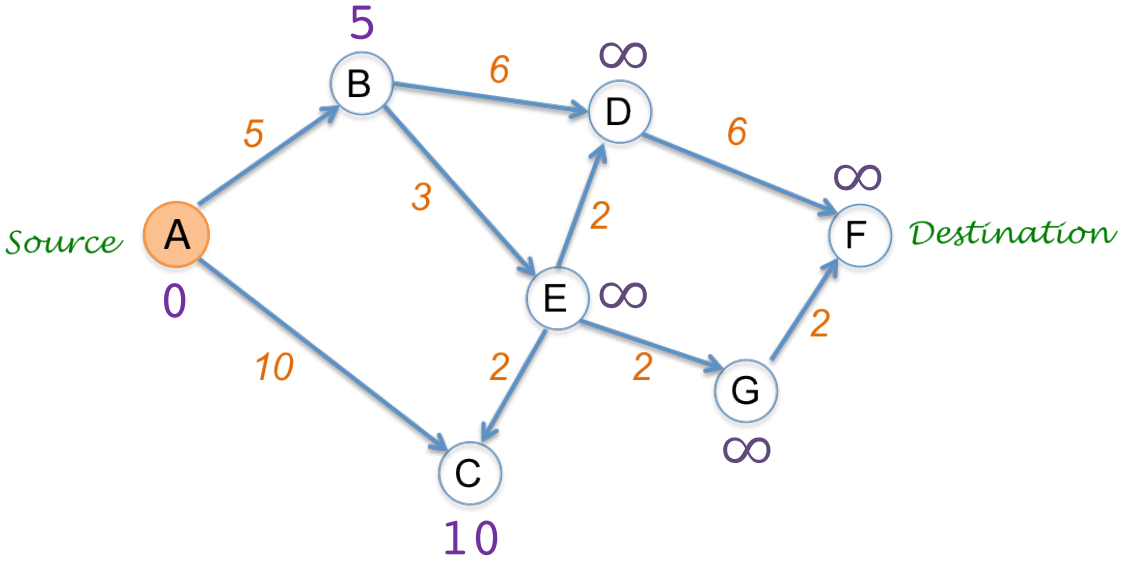
His approach was this:

- Start at the source, and work through the graph by increments
- In each increment, estimate the next best step to take
- But if the estimate is off, then it is corrected in a future step

This is a so-called greedy approach, because it goes for instant gratification, then repairs the damage if any, later. And surprisingly, against all intuition, this algorithm works correctly, and is much faster than the brute force approach.

Dijkstra's Algorithm

Imagine that you are starting at the source vertex A, about to embark on a journey that will eventually get you to F, along the shortest path.

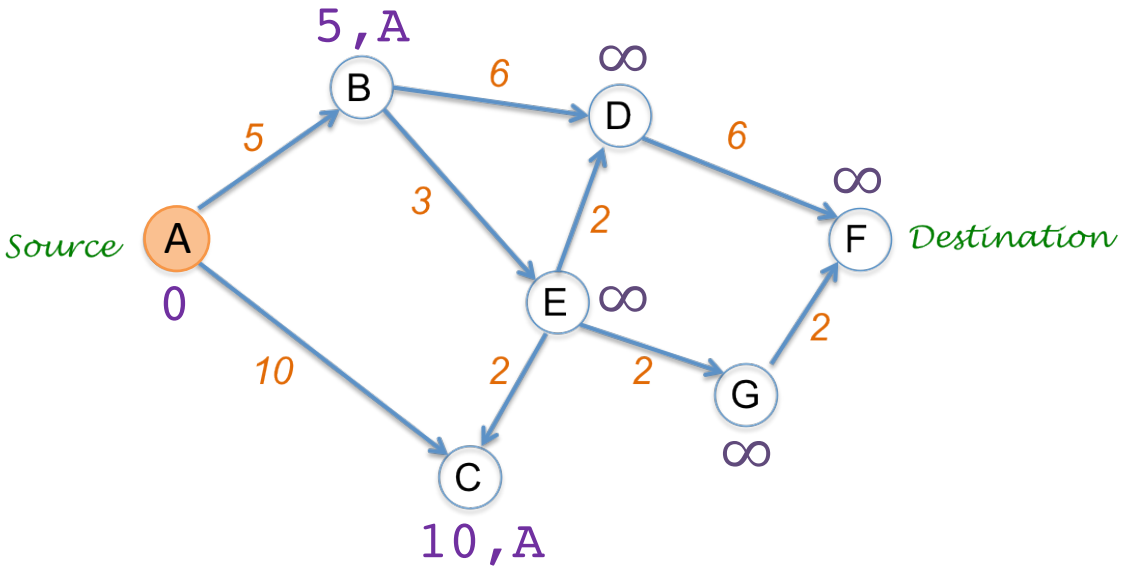


At the heart of the algorithm is the notion of the current distance of every vertex from the source.

Initially, positioned at A, the distance of A from itself is 0. You can get to B with one hop, traveling a distance of 5. Or, you can reach C in one hop, traveling a distance of 10.

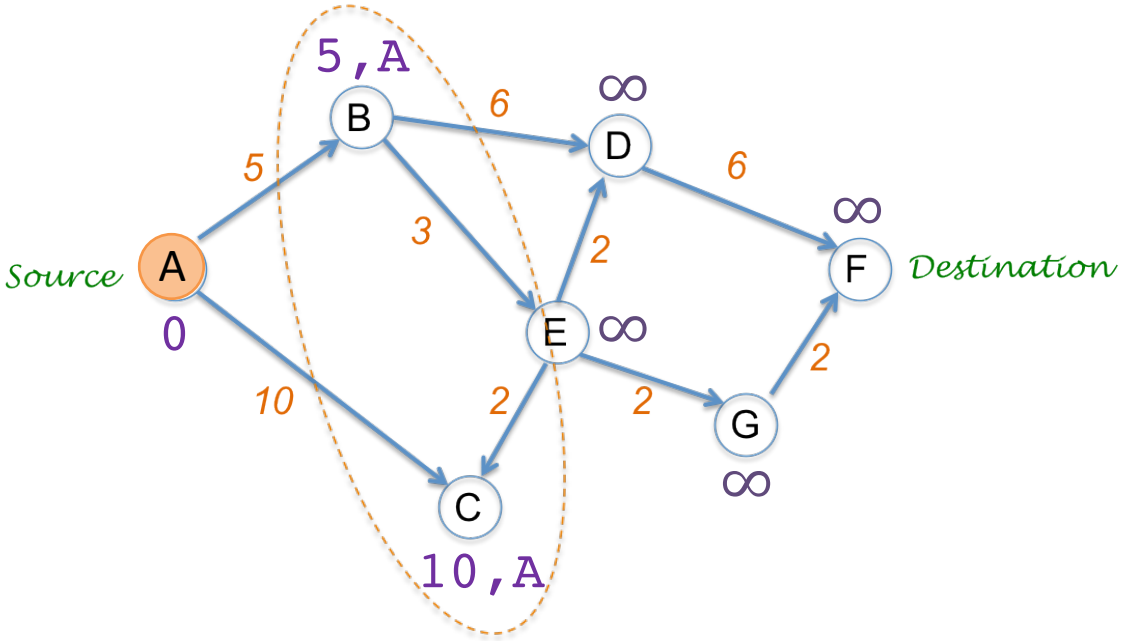
The other vertices are not reachable from A in a single hop. In other words, they are not neighbors of A, and because you can only see the neighbors, the other distances may as well be infinity.

Dijkstra's Algorithm



In addition to keeping the current distance of a vertex from the source, we will also keep the previous vertex from which we got to this vertex. We got to B from A, so we record A as B's previous vertex. Similarly for C. This will help us flesh out the sequence of edges in the shortest path when we hit the destination.

Dijkstra's Algorithm



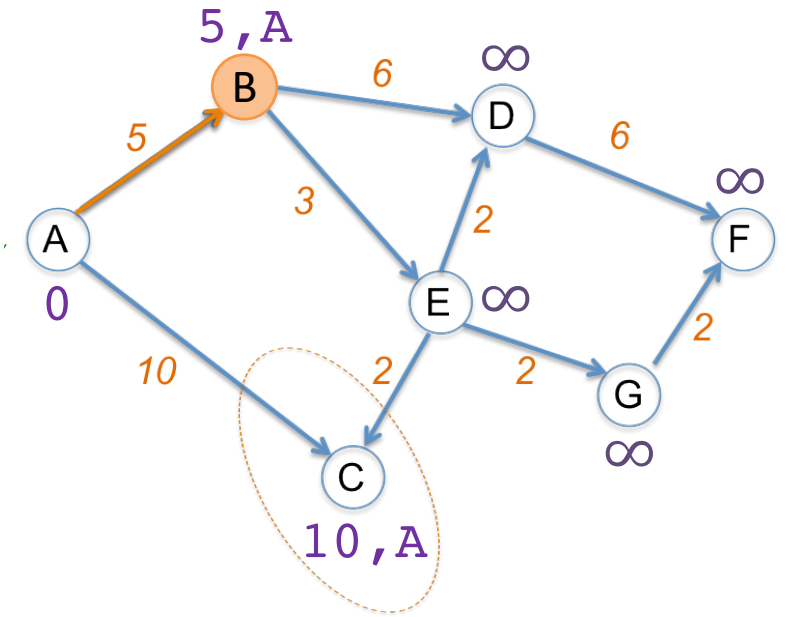
This initial step of looking at neighbors of A sets up what's called the fringe, which is the set of vertices B and C, both reachable from A with one hop.

To advance in the graph, you pick the vertex from the fringe that has the minimum distance. This is the vertex B. You remove B from the fringe, and go from A to B.

Iteration 1: Remove B from fringe, and advance to B

Now here's the neat thing. Dijkstra's algorithm says that when a vertex is taken out of the fringe, the shortest path to it has been found!

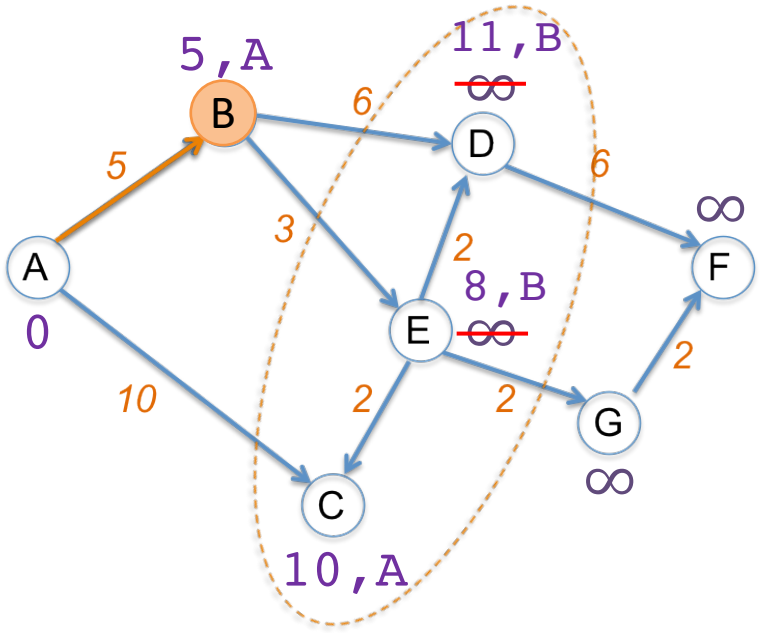
Here, you have found the shortest path to B, which is the edge A to B, and if the objective had been to find the shortest path from A to B, then you're all done!



But what if the graph were different, and there was another path that wound through other vertices and eventually landed at B, with an even shorter distance?

The magic of Dijkstra's approach is that this is guaranteed NOT to happen. When you remove a vertex from the fringe, you are guaranteed to have found the shortest path to it, and you will NOT in the future find a shorter path.

After removing B from the fringe, and advancing to B



From B, we can get to D and E

1. The weight of the edge B-D is 6, and the distance to B from the source is 5, so adding these we get 11, the distance of D from A.

2. The infinity distance of D is replaced with 11, since it is less, and we record the previous vertex to be B.

3. Similarly, the distance of E is computed as 8, and its previous vertex is set to B.

4. Since D and E have now been found to be reachable from A, they are both added to the fringe.

Keeping track of progress in a table

As we repeat the steps of the algorithm, it would be helpful to track progress in a table, as shown here after the initial, or zeroth step, and the first step that we just completed.

Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞

This table is NOT maintained by the algorithm.
It's just a way for us to track progress so we understand what actions the algorithm performs in each step.

The third through last columns show the current shortest distance of each vertex from A.

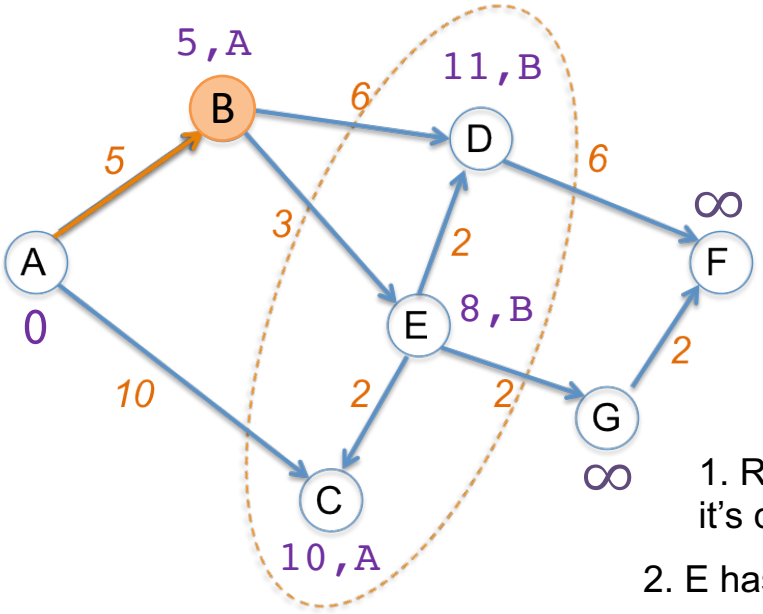
B and C are in the fringe after the initial step.

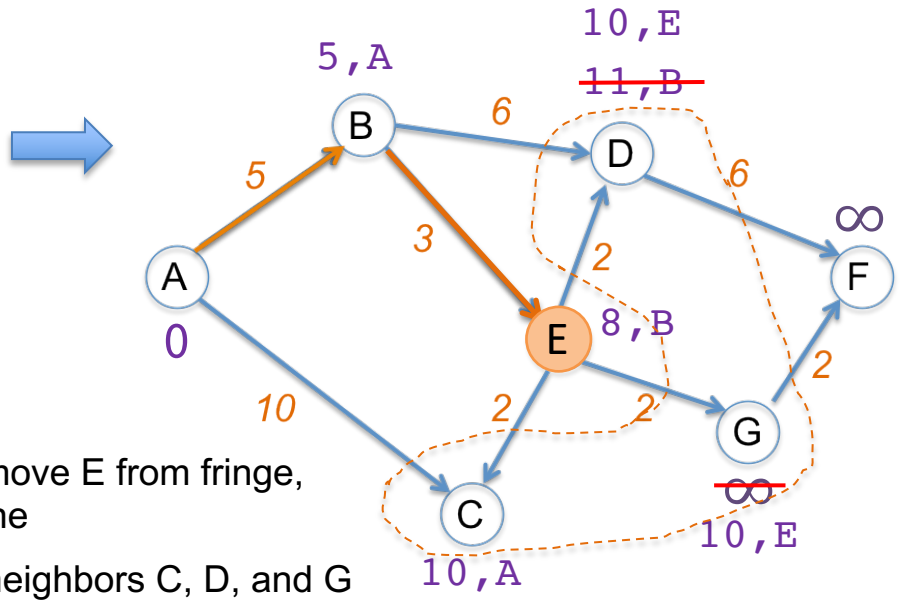
In the first iteration, or step 1, B is picked from the fringe, and is "done", meaning the shortest path to B from A has been found.

The B column effectively goes out of consideration, since B's distance will not change after this point.

The distances of D and E are lowered to 11 and 8 respectively, and the fringe now contains C, D, and E.

Iteration 2: Of all vertices in fringe, E has smallest distance





1. Remove E from fringe, it's done

2. E has neighbors C, D, and G

3. C's new distance (if the path were to go through E) is 10. It's not less than the current distance 10, so no change

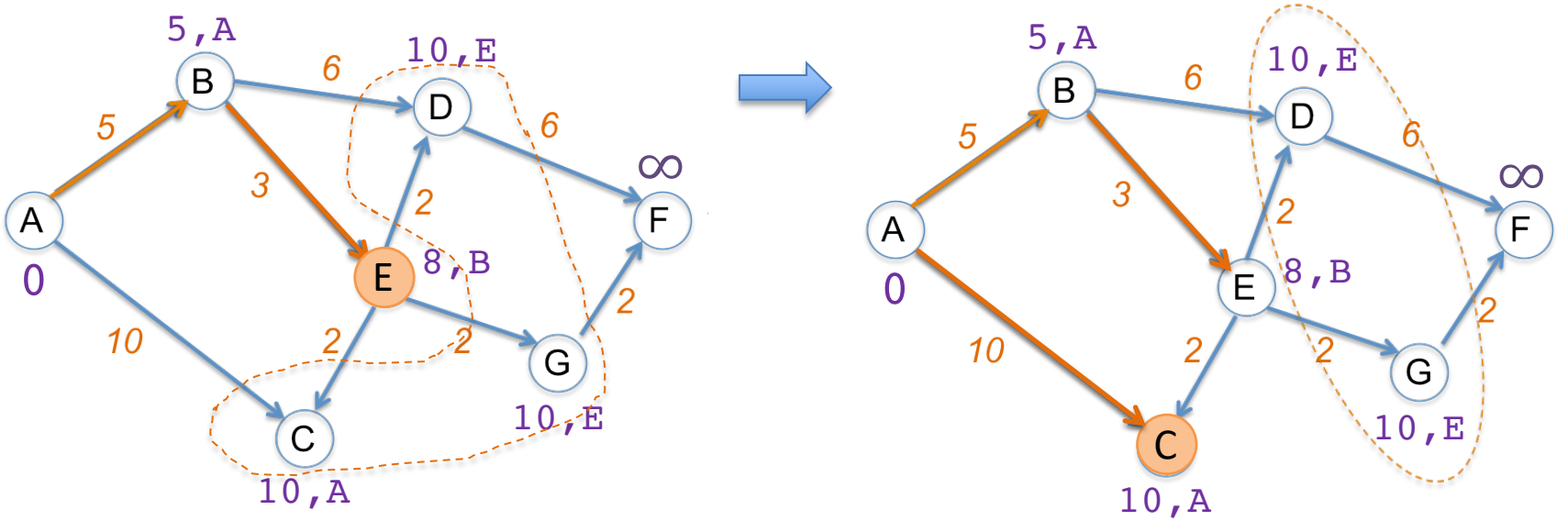
4. D's new distance = E's distance + weight of edge E-D = 8+2 = 10. This is less than the current distance of 11, so update to 10, and set previous vertex to E

5. G's current distance is infinity, so it is replaced by the new distance of 10 (previous vertex E), and G is added to the fringe

Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞
2	E		10	10		∞	10

Iteration 3: Fringe vertices C,D,G are tied for distance

Since all fringe vertices have the same distance, any of them can be picked arbitrarily – we will find a shortest path, no matter what we pick

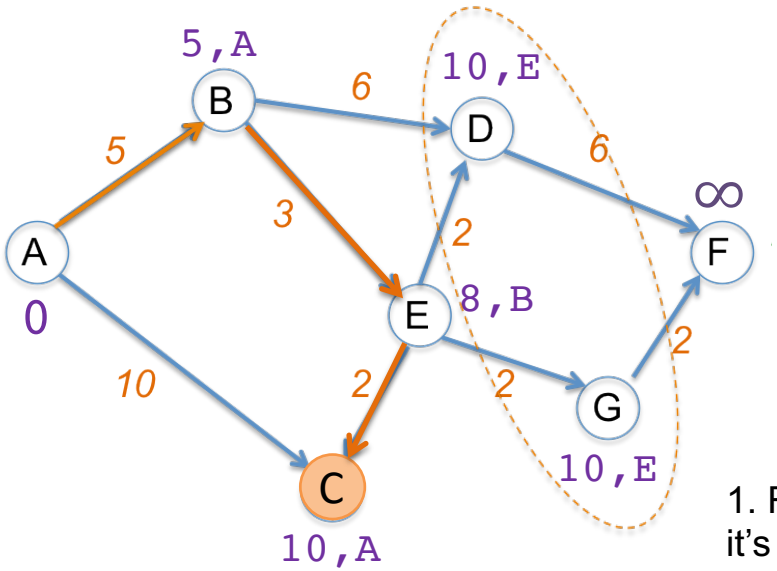


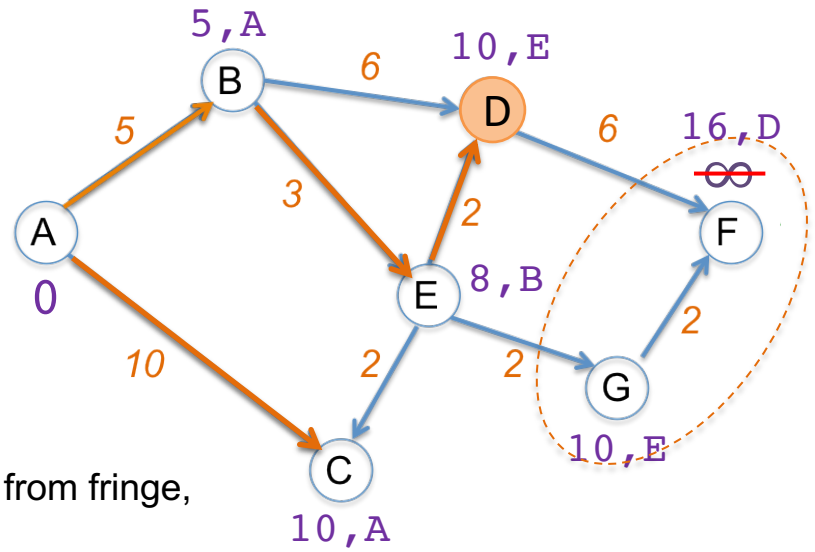
1. Remove C from fringe, it's done. It does not have any neighbors

Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞
2	E		10	10		∞	10
3	C			10		∞	10

Iteration 4: Fringe vertices D,G are tied for distance

Again, both fringe vertices, D and G, have the same distance, so we can pick either arbitrarily – we will still find a shortest path





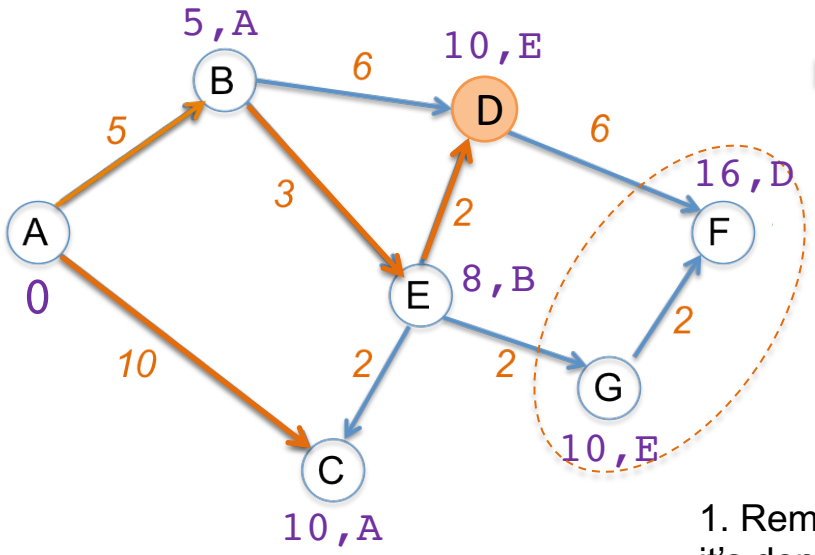
1. Remove D from fringe, it's done

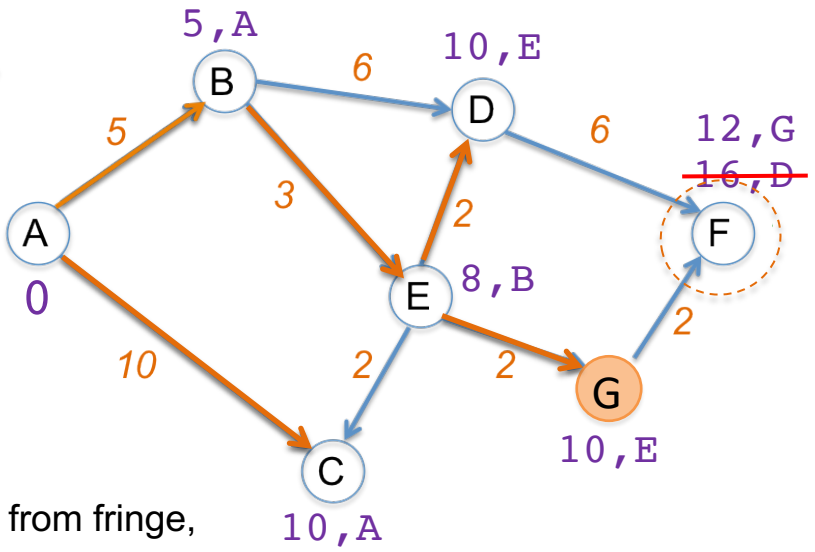
2. D has one neighbor, F

3. Distance of F can be reduced from infinity to 16, with previous vertex D, and added to the fringe

Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞
2	E		10	10		∞	10
3	C			10		∞	10
4	D					16	10

Iteration 5: Of fringe vertices G and F, G has smaller distance





1. Remove G from fringe, it's done

2. G has one neighbor, F

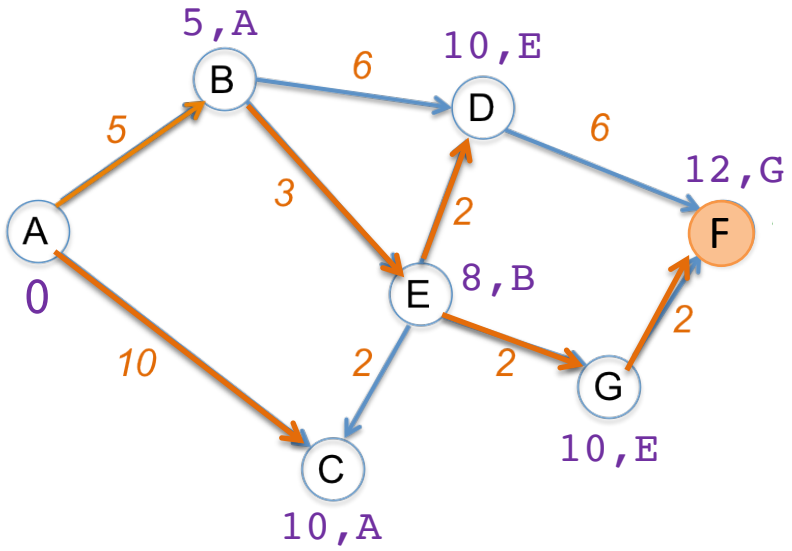
3. F's new distance = G's distance + weight of edge G--F = 10+2 = 12. This is less than the current distance of 16, so update to 12, and set previous vertex to G

4. F is the only vertex in the fringe

Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞
2	E		10	10		∞	10
3	C			10		∞	10
4	D					16	10
5	G					12	

Iteration 6: Last iteration, remove solitary vertex F from fringe

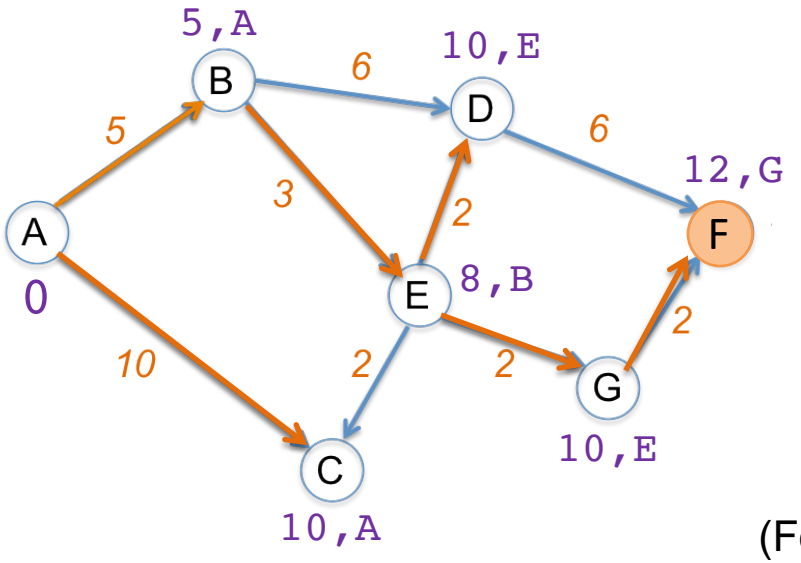
We have found a shortest path to F



Step	Done	D[B]	D[C]	D[D]	D[E]	D[F]	D[G]
0	A	5	10	∞	∞	∞	∞
1	B		10	11	8	∞	∞
2	E		10	10		∞	10
3	C			10		∞	10
4	D					16	10
5	G					12	
6	F						

The distance of the shortest path to F is at readily available.
But how to get the actual path, i.e. sequence of edges from A to F?

Tracing the shortest path



distance

0	5	10	10	8	12	10
A	B	C	D	E	F	G

previous

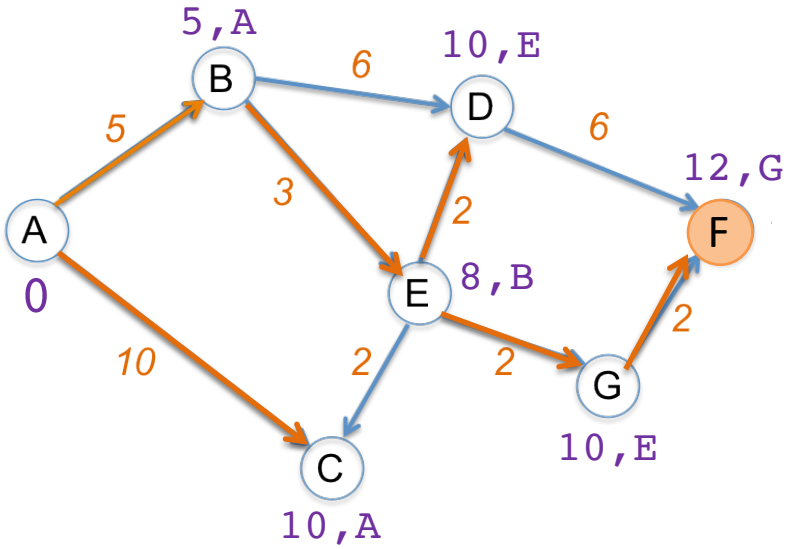
A	A	A	E	B	G	E
A	B	C	D	E	F	G

(For the arrays vertex labels are used for readability. In an implementation, vertex numbers will be used.)

The sequence of previous vertices going back from F is like a trail of breadcrumbs we can use to trace the shortest path to A

But in the end we will need to write out the path from A to F – we can achieve this by pushing vertices along the reverse path on stack, then popping from stack in the forward sequence

Tracing the shortest path



Pop all stack contents and write out the path:

A→B→E→G→F

previous

A	A	A	E	B	G	E
A	B	C	D	E	F	G

6. Stop. previous[A] = A, the source vertex

A
B
E
G
F

5. Push previous[B] = A

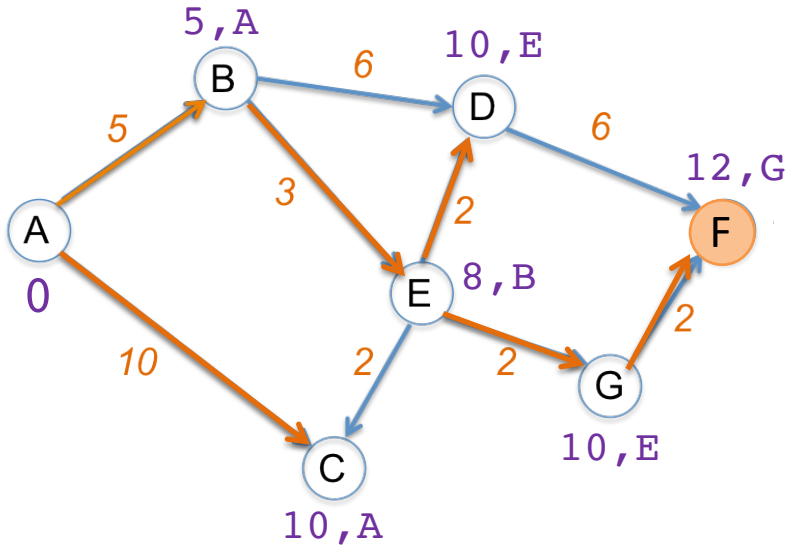
4. Push previous[E] = B

3. Push previous[G] = E

2. Push previous[F] = G

1. Push F

Dijkstra's Algorithm is a "single source" algorithm

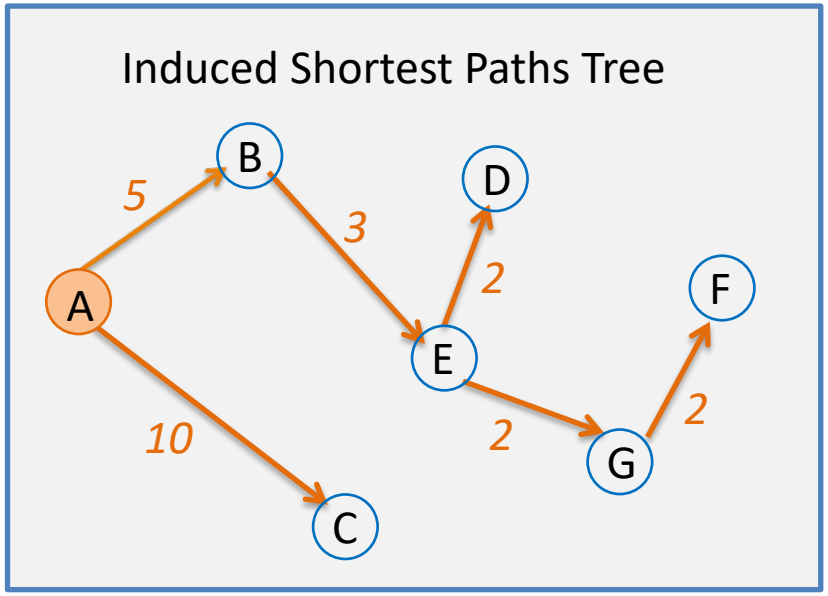


This seems redundant, but there is a big benefit. If the entire previous array contents, or the induced shortest paths tree, is stored in a file, we would have shortest paths to all vertices from the source

Then, any time we need to get the shortest distance to any vertex starting at the same source (such as A), we can simply look up the stored results, which would be much faster than running the algorithm over again.

Dijkstra's algorithm will run until all vertices are done, and the fringe is empty, even if the destination vertex was already encountered.

In our example, the destination could have been any vertex, and the algorithm would still have run until the last vertex in the fringe, F, was removed.



If the source (or graph) changes, then run the algorithm again all the way through, and store the induced shortest paths tree

Dijkstra's Shortest Paths Algorithm

```
Let s be the source vertex

Initial (step 0):
for all vertices other than s, set distance to infinity

for each neighbor vertex, v, of s:
    - Set distance of v to weight of the edge s→v, i.e.
            d(v) = wt(s,v)
    - Add v to the fringe

Iterate:
while the fringe is not empty do

   - Remove the minimum distance vertex, say m, from the fringe
     (it is done, the shortest path to it has been found)

   - for each neighbor, w, of m do
       - if d(w) is infinity:
            d(w) = d(m) + wt(m,w)
            Add w to the fringe

       - otherwise:
            d(w) = min(d(w), d(m)+wt(m,w))
```