Some graph algorithms

Djikstra’s algorithm

- This algorithm is used to find the shortest path between two points.

- Applications of this algorithm (or its variations):
  - Finding a route between two points on a map
    - Google maps likely uses some variation on this (A* algorithm? … don’t know)

  - Finding a route in networking (packet routing)

  - Etc.

Djikstra’s algorithm

- It is a greedy algorithm (we will see this when we look at the steps)

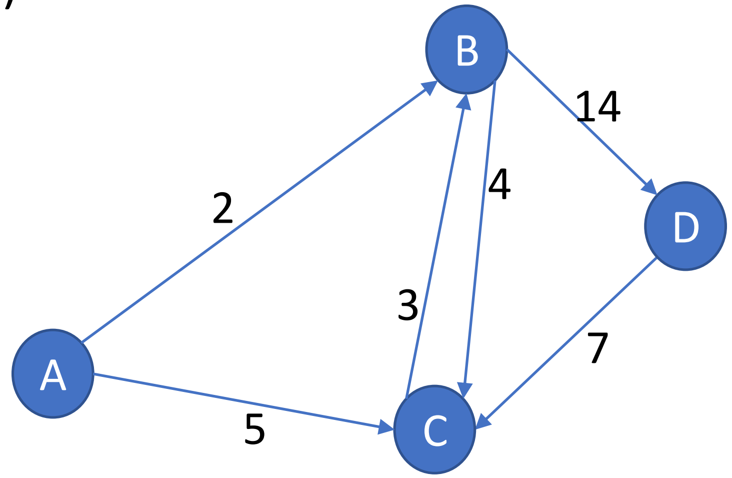
- However, in spite of being greedy, it does give us the shortest path between two vertices.

Djikstra's shortest path algorithm

- This algorithm finds the shortest path from a given vertex to *all* other vertices in the graph.
  - The graph is
    - a *directed weighted* graph,
    - with no negative weights
    - It *can* have cycles

- At a high level, the algorithm goes sort of like this:
  1. Pick ur starting vertex (we will pick **A**).
  2. In the table, initialize the distance from this vertex as 0 (obvious)
  3. And initialize distance of all other vertices to infinity

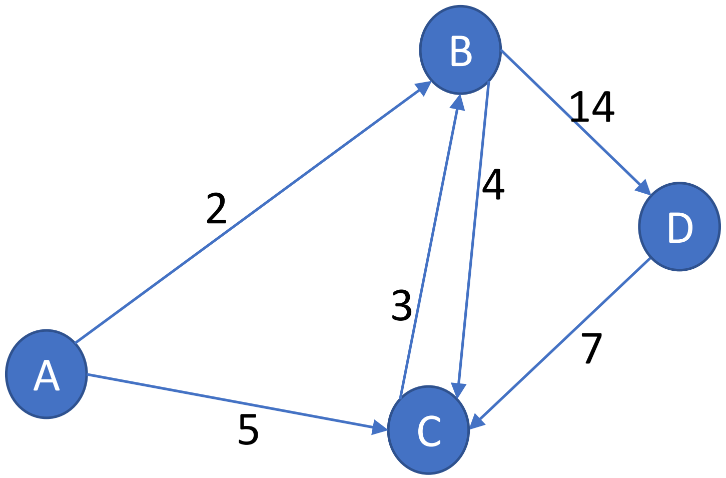
Vertex	Distance from starting vertex	Prev vertex
A	0	
B	inf	
C	inf	
D	inf	



Djikstra's shortest path algorithm

- At a high level, the algorithm goes sort of like this:
  1. Pick ur starting vertex (we will pick **A**).
  2. In the table, initialize the distance from this vertex as 0 (obvious)
  3. And initialize distance of all other vertices to infinity.
  4. currentVertex = startingVertex.
  5. Add currentVertex to visited set. { **A** }
  6. Update distance entry of neighbors of currentVertex(**A**) with their distance to the starting vertex (if new distance is smaller).
     - For B, that is 2 and for C that is 5
  7. Now, find the *cheapest unvisited* vertex.
  8. This would be B. Make it the currentVertex.
  9. Loop back to step 5 (next steps with B as currentVertex in next slide)

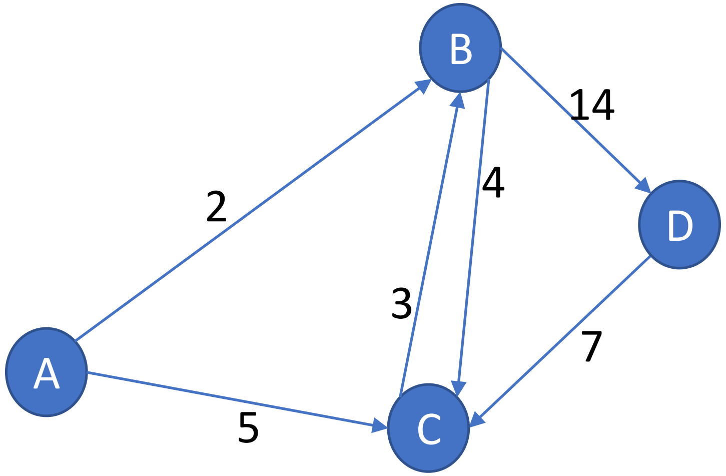
Vertex	Distance from starting vertex	Prev vertex
A	0	
B	~~inf~~ 2	**A**
C	~~inf~~ 5	**A**
D	inf	



Djikstra's shortest path algorithm

- At a high level, the algorithm goes sort of like this:
  1. Pick ur starting vertex (we will pick **A**).
  2. In the table, initialize the distance from this vertex as 0 (obvious)
  3. And initialize distance of all other vertices to infinity.
  4. currentVertex = startingVertex.
  5. Add currentVertex to visited set. { A , **B**}
  6. Update distance entry of neighbors of currentVertex (**B**) with their distance to the starting vertex (if new distance is smaller).
     - For B, that is 2 and for C that is 5
     - For C, that is 6 and for D that is 16
       - Don't update C's distance, as 6 > 5.
       - Update D's distance
  7. Now, find the *cheapest unvisited* vertex.
  8. This would be **C**. Make it the currentVertex.
  9. Loop back to step 5. (next steps with **C** as currentVertex in next slide)

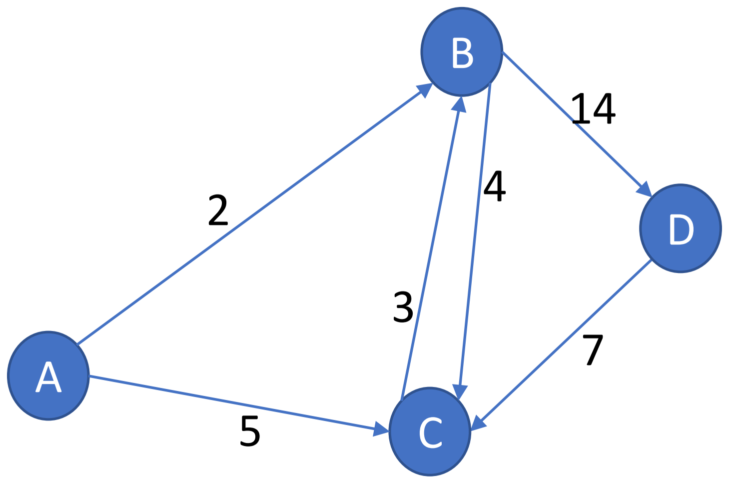
Vertex	Distance from starting vertex	Prev vertex
A	0	
B	2	A
C	5	A
D	~~inf~~ 16	**B**



Djikstra’s shortest path algorithm

- At a high level, the algorithm goes sort of like this:
  1. Pick ur starting vertex (we will pick **A**).
  2. In the table, initialize the distance from this vertex as 0 (obvious)
  3. And initialize distance of all other vertices to infinity.
  4. currentVertex = startingVertex.
  5. Add currentVertex to visited set. { A, B , **C**}
  6. Update distance entry of neighbors of currentVertex with their distance to the starting vertex (if new distance is smaller).
     - For B, that is 2 and for C that is 5
     - For C, that is 6 and for D that is 16
     - For B, that is 8
       - Dont update distance for B, as it currently has a smaller value (2)
  7. Now, find the *cheapest unvisited* vertex.
  8. This would be **D**. Make it the currentVertex.
  9. Loop back to step 5. (next steps with **D** as currentVertex in next slide)

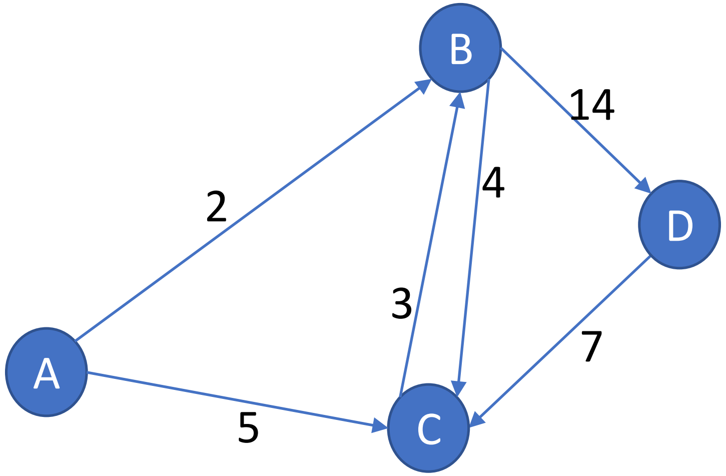
Vertex	Distance from starting vertex	Prev vertex
A	0	
B	2	A
C	5	A
D	~~inf~~ 16	B



Djikstra's shortest path algorithm

- At a high level, the algorithm goes sort of like this:
  1. Pick ur starting vertex (we will pick **A**).
  2. In the table, initialize the distance from this vertex as 0 (obvious)
  3. And initialize distance of all other vertices to infinity.
  4. currentVertex = startingVertex.
  5. Add currentVertex to visited set. { A, B, C , **D**}
  6. Update distance entry of neighbors of currentVertex with their distance to the starting vertex (if new distance is smaller).
     - For B, that is 2 and for C that is 5
     - For C, that is 6 and for D that is 16
     - For B, that is 8
     - For C, that is 23
       - Dont update distance for C, as it currently has a smaller value (5)
  7. Now, find the *cheapest unvisited* vertex.
  8. This would be **no unvisited vertices left. We are done.**

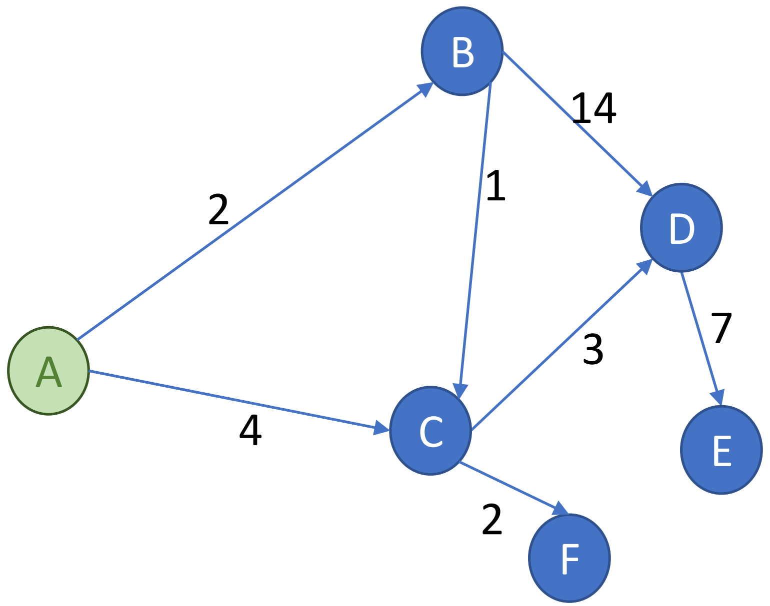
Vertex	Distance from starting vertex	Prev vertex
A	0	
B	2	A
C	5	A
D	~~inf~~ 16	B



Another graph example

- A is the start vertex

Vertex	Distance from starting vertex	Prev vertex
A	0	
B	~~inf~~ 2	A
C	~~inf~~ ~~4~~ 3	~~A~~ B
D	~~inf~~ ~~16~~ 6	~~B~~ C
E	~~inf~~ 13	D
F	~~inf~~ 5	C



Pseudocode

Lets look at some pseudocode corresponding to what we have seen so far:

visited : array representing if a vertex has been visited or not. So, *vertex*[v] true means visited, false means not visited.

dist : array of int representing distance from source vertex. So, dist [v] is distance of vertex v from source vertex s

prev: array containing previous vertex. So, *prev*[v] is the vertex we visited before coming to v, or predecessor of v.

Initialize: A function that

- Initializes *dist* array to infinity
- Initializes *prev* array to null or undefined

```
Dijkstra()  {
        Initialize();
        currentVertex = source;
        dist [ source ] = 0;

        while ( unvisited vertices are there )
        {
                visited [ currentVertex ] = true;
                for each neighborVertex of currentVertex  {
                        newDist = dist [ currentVertex ]  +  Length ( currentVertex,  neighborVertex );
                        if ( newDist  <   dist [ neighborVertex ] )
                                dist [ neighborVertex ] = newDist;
                                prev [ neighborVertex ] = currentVertex;
                }
                currentVertex = GetUnvisitedVertexWithSmallestDist();
        }
```

- Complexity is $O(|E| + |V|^2)$
    - This is if we use a list or array for holding the vertices when looking for next vertex with minimum distance to source.
    - A better data structure to use for getting next vertex with minimum distance to source would be a ? [redacted]
        - Complexity in this case would be [redacted]

Dijkstra and BFS

- If we make all weights on a graph the same (say, 1), then BFS and Dijkstra will behave the same, because we essentially now have an unweighted graph.

Other algorithms

- **Floyd-Warshall**
  - Shortest path between all pairs of vertices in a graph
  - Can detect a negative weight cycle.
  - O($|V|^3$)

- **Bellman-Ford**
  - Alternative to Dijkstra if graph has negative edge weights.
  - Can detect a negative weight cycle.
  - O(|V| |E|)