// earlier in lecture 15 we saw BFS being used for shortest paths

Single Source Shortest Paths

Paths in Graphs

"directed graph"

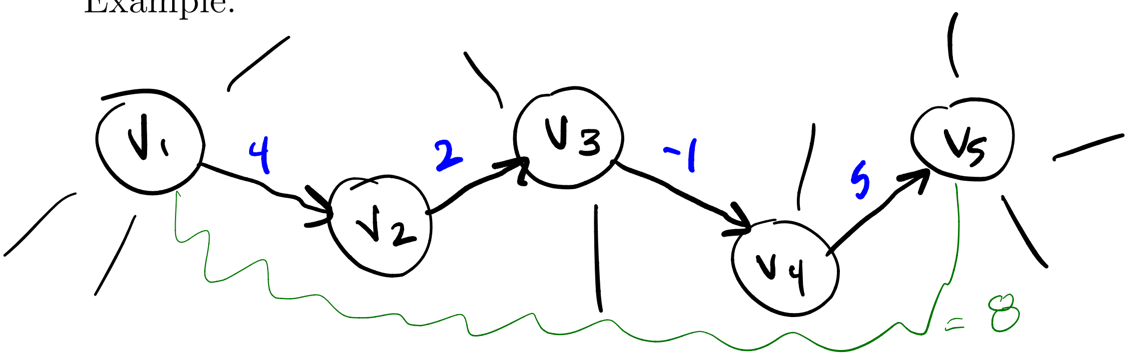
Consider a digraph $G = (V, E)$ with edge-weight function $w : E \to \mathbb{R}$. The weight of path $p = v_1, v_2, \ldots, v_k$ is defined to be

$$w(p) = \underbrace{\sum_{i=1}^{k-1} w(v_i, v_{i+1})}$$

– sum of all edge weight

= weight

Example:



$\Rightarrow$ 4 + 2 + (-1) + 5 = 10

- This is cost of 1 path, we have another path that has $w(p) = 8$.

$\Rightarrow$ We pick the path with smallest $w(P)$

Shortest Paths

A **shortest path** from u to v is a path of minimum weight from u to v. The **shortest path weight** from u to v is defined as

$\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$

(Note: $\delta(u, v) = \infty$ if no path from u to v exists)

Optimal Substructure

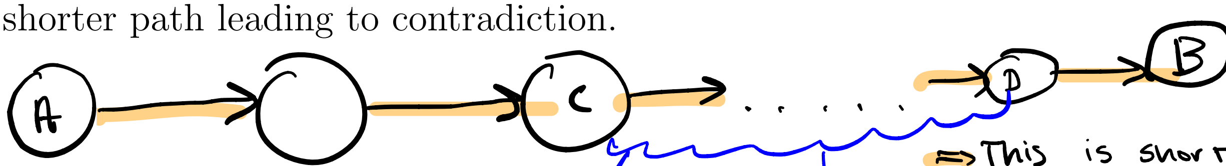
D.P.

∴ Find shortest paths recursively

Some useful theorems: For calculating shortest paths

- **Theorem**. A subpath of a shortest path is a shortest path.

Proof. Suppose it wasn't a shortest path. Cut and paste to create a shorter path leading to contradiction.



⇒ This is shortest path from (A to B)

- Theorem above states this is the shortest path from (C to D). No shorter path exists

- (C to D) is a subpath

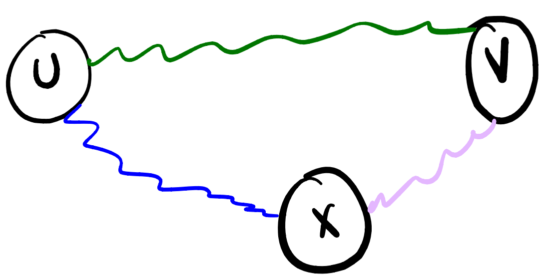
- **Theorem**. For all $u, v, x \in V$, it is the case that $\underbrace{\delta(u, v)}$ $\leq \delta(u, x) + \delta(x, v)$

↳ Defined as shortest path weights from (u to v)

∴ Squiggles represent shortest paths between vertices

Proof.

- Triangle inequality

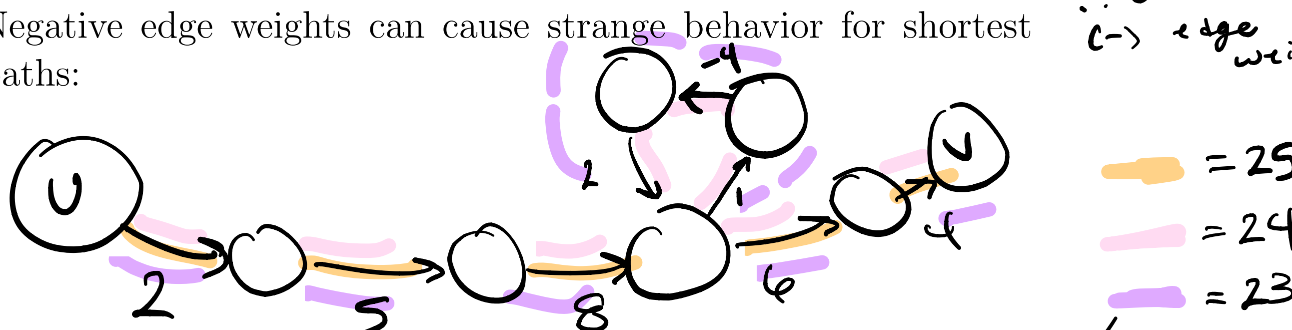


⇒ $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$

∴ can't be faster to go (u, x), then (x, v)

Many algorithms assume that there are no negative edge weights. Negative edge weights can cause strange behavior for shortest paths:

∴ only 1 (-) edge weight



∴ goes through loop twice

⇒ Shortest path will infinitely loop

↳ No shortest path exists from (u to v)

Single-source Shortest Paths

Problem. From a given source vertex $s \in V$, find the shortest the shortest-path weights $\delta(s, v)$ for all for all $v \in V$.

Assumption: All edge weights $w(u, v)$ are nonnegative. It follows that all shortest-path weights must exist (i.e., no negative weight cycles).

Idea: Greedy Algorithm

- Maintain a set S of vertices whose shortest-path weights from s are known (Let $d[v] = \delta(s, v)$)
- At each step add to S the vertex $v \in V - S$ whose distance estimate from s is minimal
- Update the distance estimates of vertices adjacent to v.

Dijkstras algorithm

```
Set Q = V
Set d[v] = ∞ for all v ∈ V
Set d[s] = 0 for your starting point s ∈ V
while Q is not empty do
    u = ExtractMin(Q)
    for each v ∈ Adj[u] do
        if v ∈ Q and d[u] + w(u, v) < d[v] then
            d[v] = d[u] + w(u, v)      //Decreases key in priority queue Q
            π[v] = u
        end if
    end for
end while
```

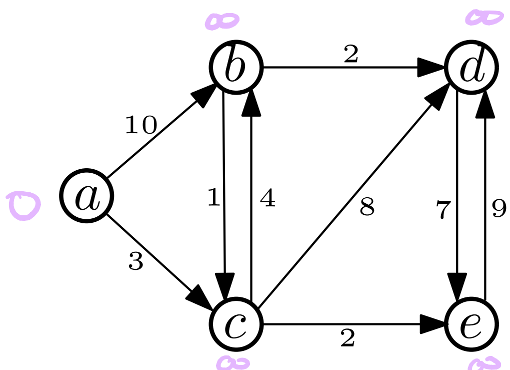
compute: shortest path from start point to all other places of graph

SP := Shortest path

Dijkstras Example (1)

Known SPs

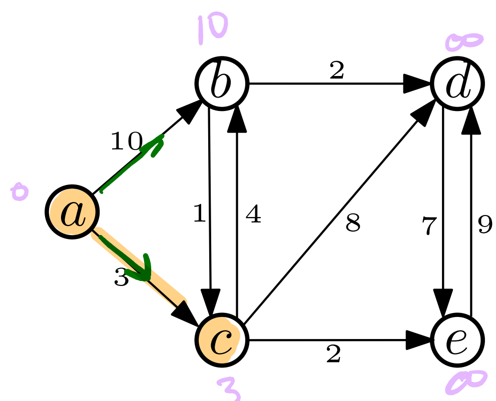
S = { }



	A	B	C	D	E
$Q:$	$\cancel{0}$	∞	∞	∞	∞

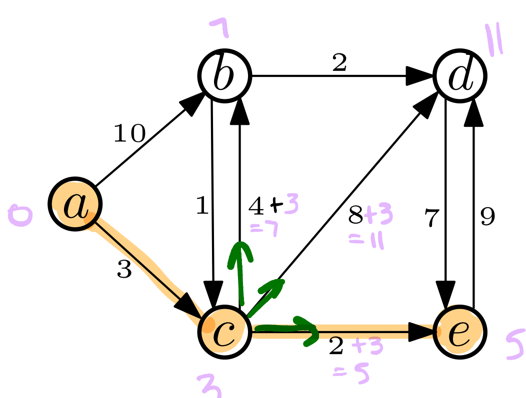
Priority queue

S = {A}



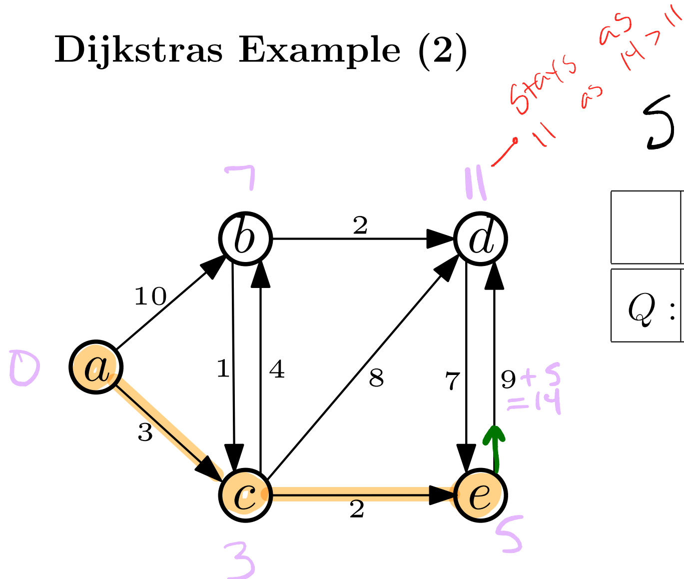
	A	B	C	D	E
$Q:$	$-$	10	$\cancel{3}$	∞	∞

S = {A, C}



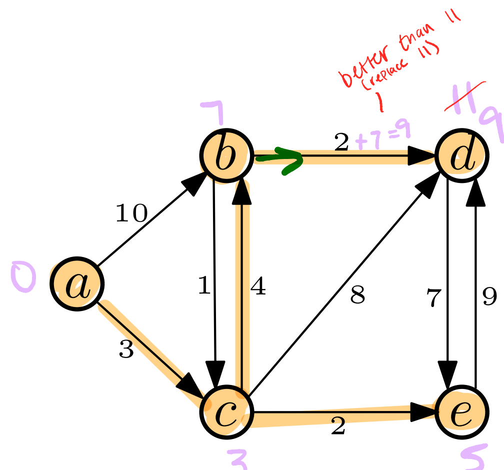
	A	B	C	D	E
$Q:$	$-$	7	$-$	11	$\cancel{5}$

Dijkstras Example (2)



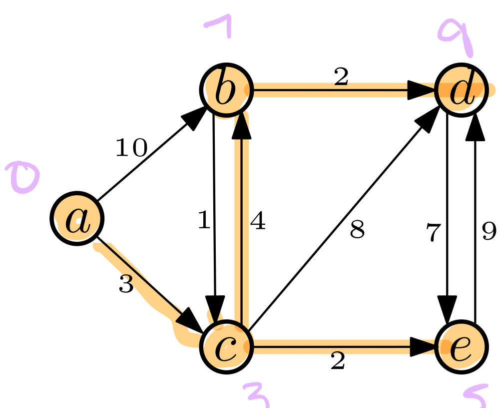
S = { A, C, E }

	A	B	C	D	E
Q :	–	7	–	11	–



S = { A, C, E, B }

	A	B	C	D	E
Q :	–	–	–	9	–



S = { A, C, E, B, d }

	A	B	C	D	E
Q :	–	–	–	–	–

∴ Forms a Tree of connections that tells us what edges to use to get to diff. locations

Dijkstras Algorithm Runtime

Here's our pseudocode for Dijkstras Algorithm:

Set $Q = V$
Set $d[v] = \infty$ for all $v \in V$
Set $d[s] = 0$ for your starting point $s \in V$
while Q is not empty **do**
    $u = ExtractMin(Q)$
    for each $v \in Adj[u]$ **do**
        if $v \in Q$ and $d[u] + w(u, v) < d[v]$ **then**
            $d[v] = d[u] + w(u, v)$ //Decreases key in priority queue Q
            $\pi[v] = u$
        end if
    end for
end while

Compare this to Prim's Algorithm:

$Q = V$
Set $key[v] = \infty$ for all $v \in V$
Set $key[s] = 0$ for some arbitrary $s \in V$
while Q is not empty **do**
    $u = ExtractMin(Q)$
    for each $v \in Adj[u]$ **do**
        if $v \in Q$ and $w(u, v) < key[v]$ **then**
            $key[v] = w(u, v)$ //Decreases key in priority queue Q
            $\pi[v] = u$
        end if
    end for
end while

Same as Prim's Algorithm:

Q	$T_{EXTRACT-MIN}$	$T_{DECREASE-KEY}$	Total Runtime
array	$O(\|V\|)$	$O(1)$	$O(\|V\|^2)$
binary heap	$O(\log\|V\|)$	$O(\log\|V\|)$	$O(\|V\|\log\|V\|+\|E\|\log\|V\|)$
Fibonacci heap (not covered)	$O(\log\|V\|)$	$O(1)$	$O(\|V\|\log\|V\|+\|E\|)$