Shortest Path Problem

- **Statement** :

  Finding a path between 2 vertices in a graph s.t. the sum of the weights of its constituent edges is **minimized**

- The problem can be defined for undirected or directed graphs.

- Input: A weighted $G := (V, E)$

  s.t. $\forall \; (v_i, v_j) \in E$

  $\exists \; c_{ij} \in \mathbb{R}_+$

- Problem: For a path

  Goal $P = (v_1, v_2, \ldots v_n)$

  minimize $\sum_{i=1}^{n} c_{i, i+1}$

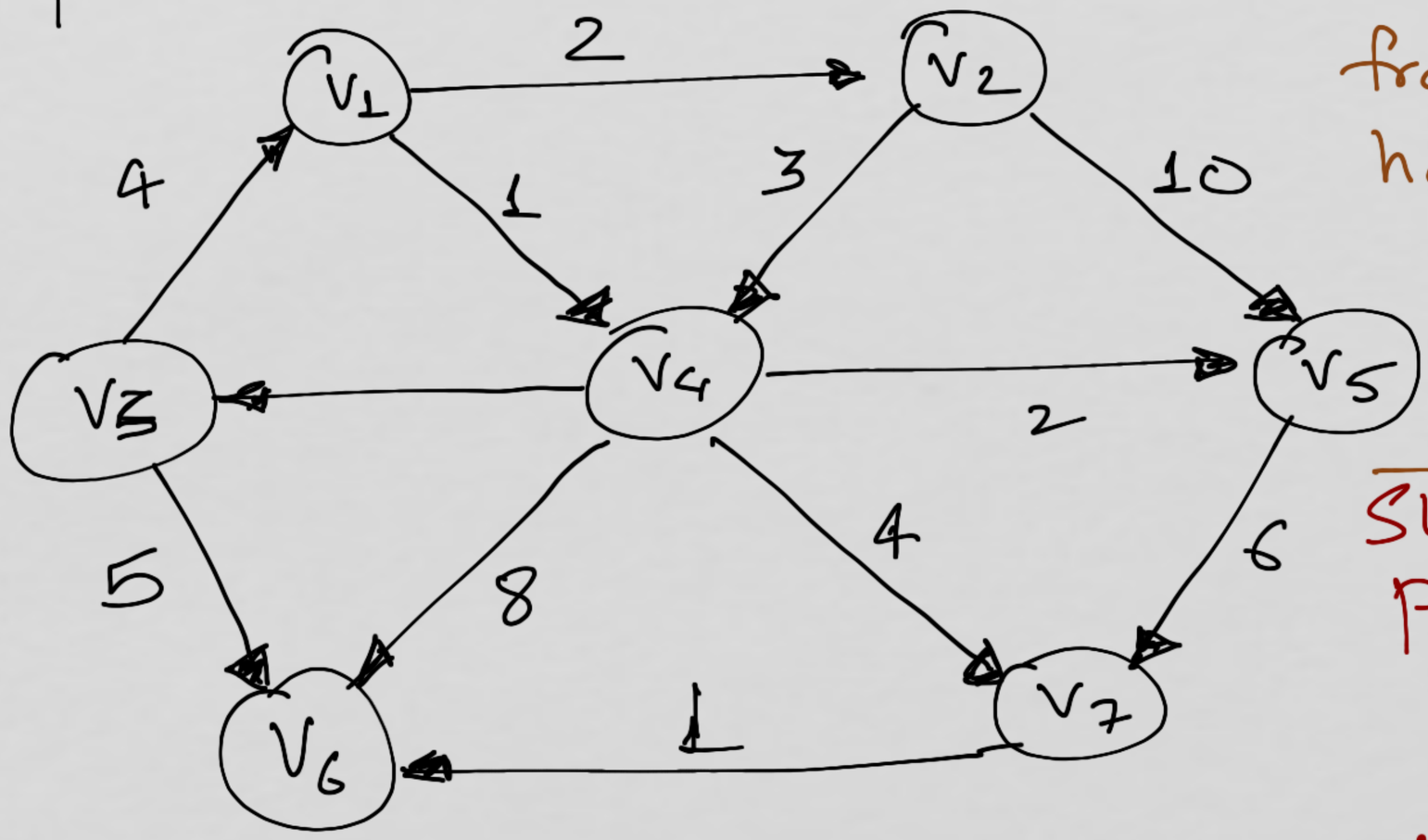
Variations

- Single-source shortest path
  → Given a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G.

- All-pairs shortest path problem
  → finding shortest paths between every pair of vertices in G.

Applications : Google maps, Routing protocols,

Example



Shortest path from V_1 to V_6 has a cost 6

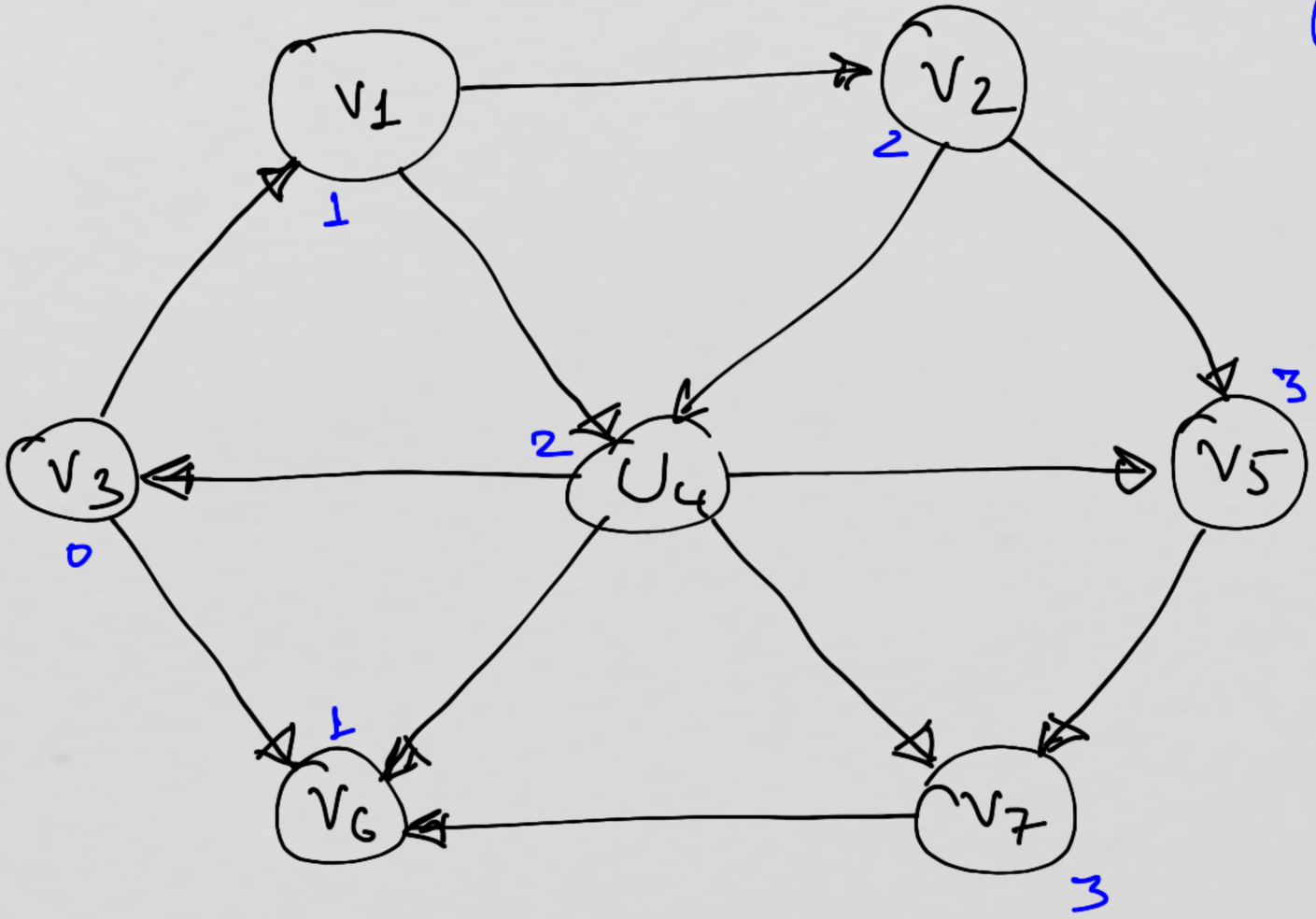
→ $V_1 \rightarrow V_4 \rightarrow V_7 \rightarrow V_6$

Shortest Unweighted path → 2

Negative weighted Edges can create complications

↳ We will come to them later!

Unweighted Shortest Paths



Thus, all we require is BFS traversal from the source vertex with Extra book-Keeping.

Only interested in examining # of edges in the path.

Let $s = v_3$

- $v_3 \longrightarrow v_3$
  $= 0$ Edges
- Path of length 1
  $v_3 \longrightarrow v_1$
  $v_3 \longrightarrow v_6$
- Path of length 2
  $v_3 \longrightarrow v_1 \longrightarrow v_2$
  $v_3 \longrightarrow v_1 \longrightarrow v_4$
- Path of length 3
  $v_3 - v_1 - v_2 - v_5$
  $v_3 - v_1 - v_4 - v_7$

The additional book-keeping

d_v → holds the distance of vertex v from source s

- Initialise → all are unreachable Except s (to itself)

Alg 1

```
for (currDist = 0; currDist < NumVertex;
                   currDist ++)
    for each vertex v
        if (!K[v] and d[v] = currDist)
            K[v] = True;
            foreach w adjacent to v
                if d[w] = ∞
                    d[w] = currDist + 1
                    P[w] = v
```

Runtime

$O(|V|^2)$

Remove the inefficiency the way we did for Topological Sort!

Need not search all vertices
→ But only those which have d[v] = currDist or d[v] = currDist + 1

Only this needs to be processed so maintain a queue

These get added to the rear of the Queue

Alg 2

```
Q.Enq(s)
while ( ! Q.Empty() )
    v = Q.deq()
    K[v] = True;
    for each w adjacent to v
        if d[w] = ∞
            d[w] = d[v]+1
            P[w] = v
            Q.enq(w)
```

Runtime Complexity

$O(V+E)$

as long as adjacency lists are used!

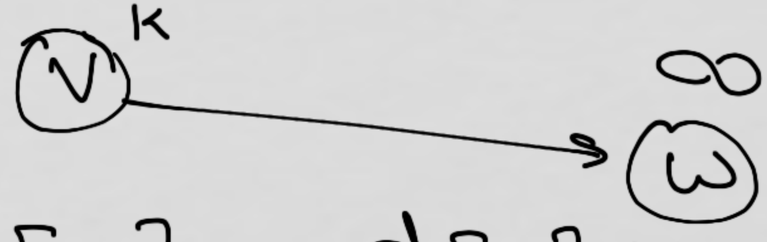
Weighted Graphs

- The problem becomes a bit harder but we can use the ideas from the unweighted case
- Dijkstra's Algorithm (circa 1956)

Dijkstra's Algo

- Example of a greedy algorithm
- At each stage, the alg. chooses an unvisited vertex with the smallest d[v] and declares the shortest path from s ⟶ to ⟶ v is known
- The remainder of a stage is focused on updating d[w] where w : adj(v)

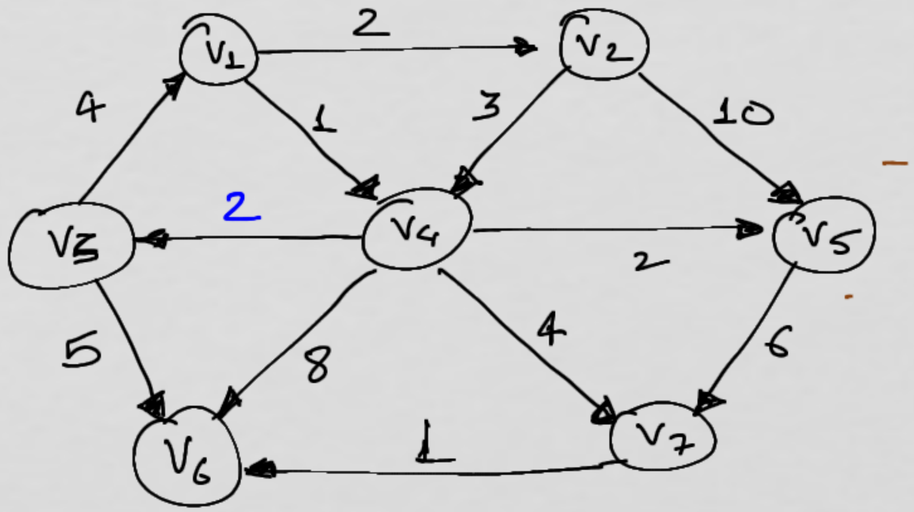
— Note in the unweighted case



$$d[w] = d[v] + 1$$

— In the weighted case this could become

$$d[w] = d[v] + c_{vw}$$



$S = v_1$

$v_1 \to v_2 = 2$

$v_1 \to v_4 = 1$ → choose the min

v_4 is marked

$v_1 \to v_4 v_3 = 3$

$v_1 \to v_4 v_5 = 3$

$v_1 \to v_4 v_L = 9$

$v_2 \to v_4 v_7 = 5$

v_2 is marked

$v_1 \to v_2 v_4 = 5$ → Not updated ∵ already known

$v_1 \to v_2 v_5 = 12$ → Not adjusted

We now choose v_5

$v_L \to v_{5,3} v_7 = 9$ Not adjusted

at cost 3 [i.e through v_2 & v_4]

$v_1 \to v_{3,3} v_6 = 8 < 9$ so it is updated

v_3 is selected at 3

Pseudo Code

```
Dijkstra(G, source):
    for each v ∈ V:
        d[v] = ∞
        P[v] = 1   // i.e. undefined
        add v to Q
    d[source] = 0
    while (Q is not empty):
        u = vertex in Q with min d[u]
        remove u from Q
        for each neighbor v of u in Q
            alt = d[u] + C_u,v
            if (alt < d[v]):
                d[v] = alt
                P[v] = u
```

Runtime Analysis

- It is over a BFS
- Outermost loop will run for $O(|E| + |V| \cdot T_{em})$
- T_{em}
  - Time to extract min $d[u]$
  - If implemented as an array or list $T_{em} \in O(|V|)$

$\therefore T \in O(|V|^2)$

Dijkstra's Alg Ver 2

- Idea: - Do not fill Q with all $v \in V$, initially. Instead, start with inserting only source to Q.
  - Later, if (alt < d[v]) check succeeds, then **add v to Q** if the node is not already in Q, Else it is **decrease priority!**
  - Implement Q as a min-heap!

Changes in the algorithm

```
for each neighbor v of u in Q
    alt = d[u] + c_{u,v}
    if (alt < d[v]):
        d[v] = alt
        P[v] = u
        if v not in Q:
            add v to Q
        Else
            decrease-key (Q,v)
```

Runtime Analysis

- $T(\text{Dijkstra's Alg}) = \Theta(|E| \cdot T_{dk} + |V| \cdot T_{em})$
  - T_{dk}: Time to decrease key
  - T_{em}: Time to extract minimum
- $T_{dk} = T_{em} \in \Theta(1)$ for min-Heaps
- $T(\text{Dijkstra's Alg}) = \Theta((|E| + |V|) \log |V|)$

for connected graphs [i.e. graph must have at least $|V|-1$ edges]

$= \Theta(|E| \log |V|)$

Using more advanced data structures such as fibonacci heaps

$= \Theta(|E| + |V| \log |V|)$

Proof of Correctness

- Let $\delta[v]$ = shortest path from s to v
- Lemma 1: if $d[v] = \delta[v]$ at any stage of the algorithm then $d[v] = \delta[v]$ for the rest of the algorithm

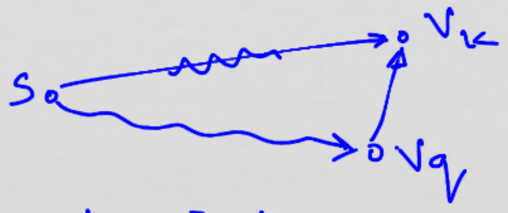
Proof: Trivial [the check in the algorithm will never succeed]

- Thm 1: Let $\langle s = v_1, v_2, \ldots v_n \rangle$ denote the sequence of vertices extracted from the heap Q. When v_i is extracted, $d[v_i] = \delta[v_i]$

Proof: . WLOG, every v is either reachable by s through a finite path or unreachable

Base
- Clearly $d[s] = \delta[s] = 0$

Hypothesis
- Assume the hypothesis is true for the first $k-1$ vertices
i.e. when v_i is deleted from Q
$d[v_i] = \delta[v_i]$



$d[v_k]$ through q
$= d[v_q] + \min C_{q,k}$
$\Rightarrow$ If the above was $\delta[v_k]$ then $\delta[v_k] > d[v_q]$

Induction Step
- if the shortest path from s to v_k contains vertices from the set $\{v_1, v_2, \ldots v_{k-1}\}$ then $d[v_k] = \delta[v_k]$
[why?] → I.H., Lemma 1
- Assume $\delta[v_k] < d[v_k]$
i.e. $\exists$ v_q which is unvisited on the shortest path from s to v_k.
By I.H. the shortest path to v_q through visited nodes only is $d[v_q] = \delta[v_q]$
Min cost from s to v_k via v_q is then $d[v_q]$ + min cost from v_q to v_k
∵ No edges have zero-costs → ①
Also note alg picked v_k over v_q ⇒ $d[v_k] < d[v_q]$
[But from ① and the assumption $d[v_q] \geq \Delta = \delta[v_k] < d[v_k]$

Contradiction

Bellman-Ford

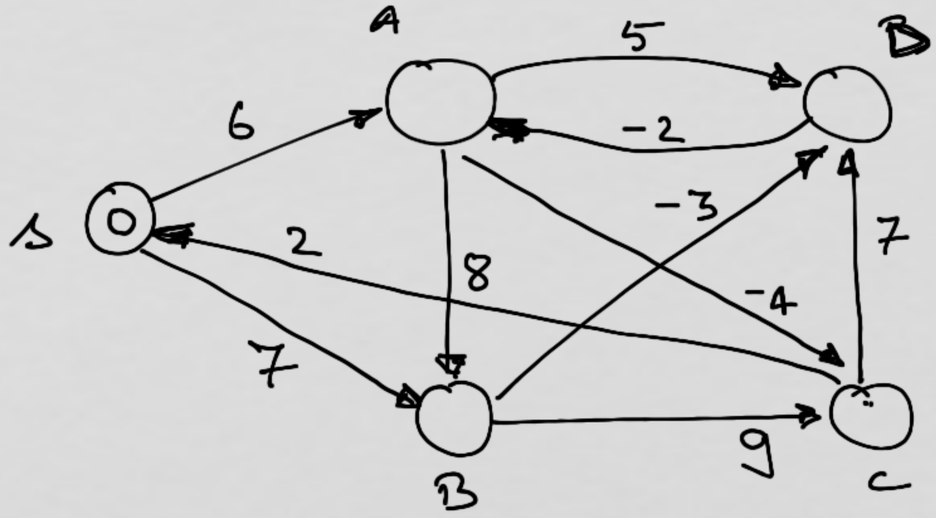
- A SSSP algorithm
- Slower than Dijkstra's Alg
- Takes care of negative-edge weights
- Dijkstra's Alg → uses priority Q to greedily choose the closest vertex
- Bellman-ford → simply looks at all Edges & repeats the process for |V|-1 times
- Time Complexity is a casuality

Algorithm

```
Bellman-ford ( source, V, E):

    // initialise d[v] , p[v]

    // initialise d[s] = 0

    Repeat |V|-1 times :
        for each (u,v) ∈ E with weight c
            alt = d[u] + c
            if (alt < d[v]):
                d[v] = alt
                p[v] = u

    // Code for finding a
    // negative-weight cycle

// Key condn : d[u] + c_uv < d[v]
```

why?

Longest path w/o acycle is |V|-1; Edges must be scanned that many times



	A	B	C	D
Δ	6	7	-	-
→A	6	7	2	11
→B	6	7	2	4
→ C	6	7	2	4
→ D	?	7	2	4

```
//Check for -ive cycles
foreach (u,v) ∈ E do
    if d[u] + cuv < d[v]
        P[v] = u
        // create visited array
        visited[v] = True
        while ! visited[u]:
            visited [u] = True
            u = P[v]
        ncycle = [u]
        v = P[u]
        while (v != u):
            ncycle = concatenate
                     (v, ncycle)
            v = P[v]
        Throw "G contains a
               -ive cycle"
```

DA

P[a] = d

visited

[a] = T

u = P[d]