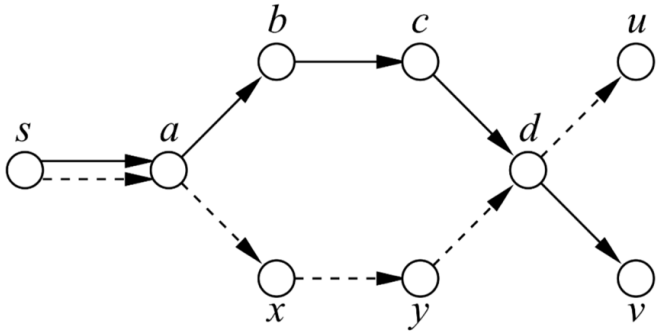
Complexity & Algorithms, Spring 2026

Shortest Paths

Single source shortest paths (SSSP)

Some notes:

Why a tree?



If $s \to a \to b \to c \to d \to v$ and $s \to a \to x \to y \to d \to u$ are shortest paths, then $s \to a \to b \to c \to d \to u$ is also a shortest path.

What about negative cycles or edges?

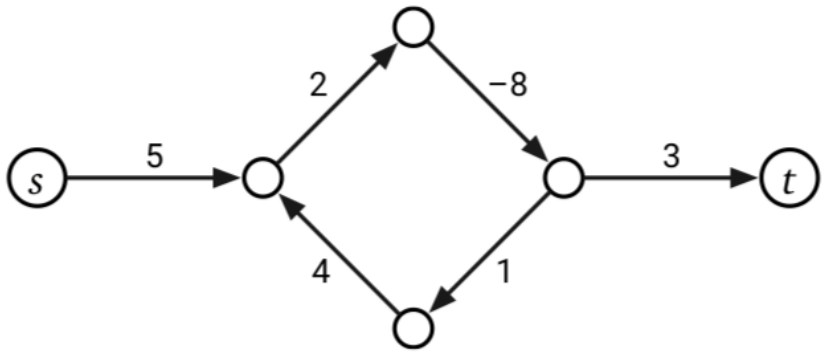
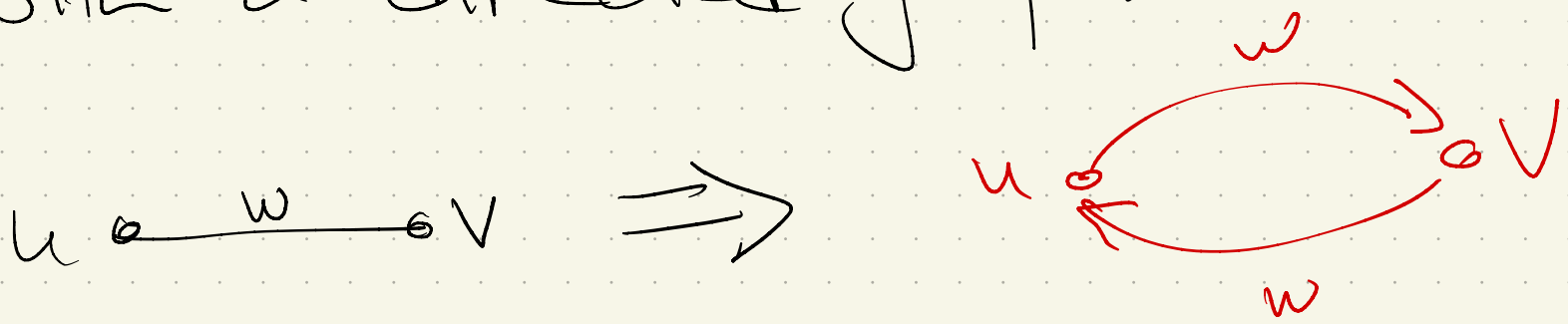


Figure 8.3. There is no shortest walk from s to t.

Also: If undirected, can simulate with a directed graph:



Unless you have negative edges.
(It gets wierd.)

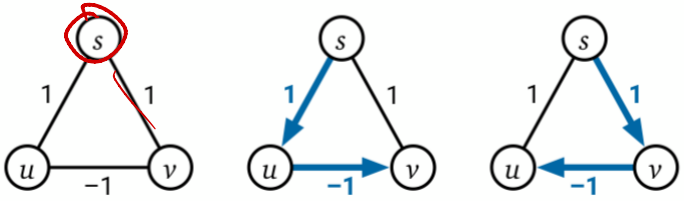
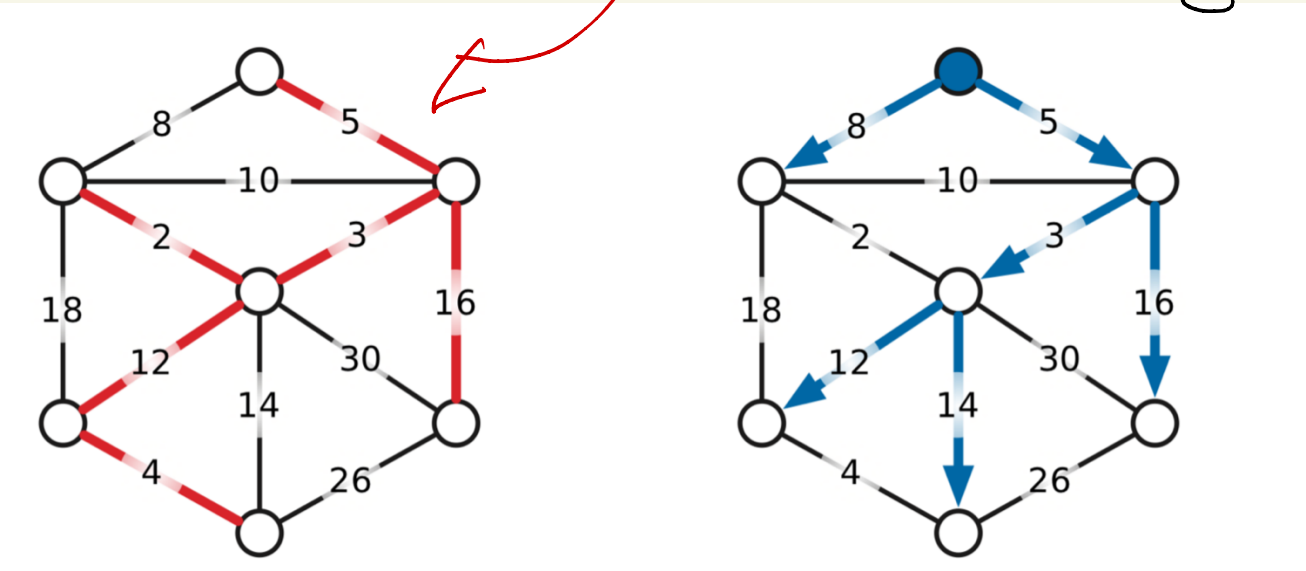


Figure 8.4. An undirected graph where shortest paths from s are unique but do not define a tree.

Important to realize:

SSSP ≠ Minimum Spanning Tree



Why? based on source vs global min object

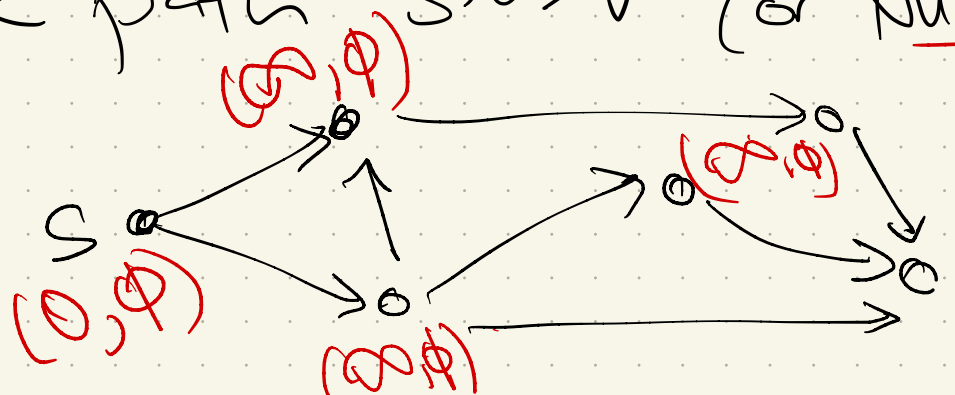
Computing a SSSP.

(Ford 1956 & Dantzig 1957)

Each vertex will store 2 values.
(Think of these as tentative shortest paths.)

- dist(v) is length of tentative shortest path $s \rightsquigarrow v$ (or ∞ if don't have an option yet)
- pred(v) is the predecessor of v on that tentative path $s \rightsquigarrow v$ (or NULL if none)

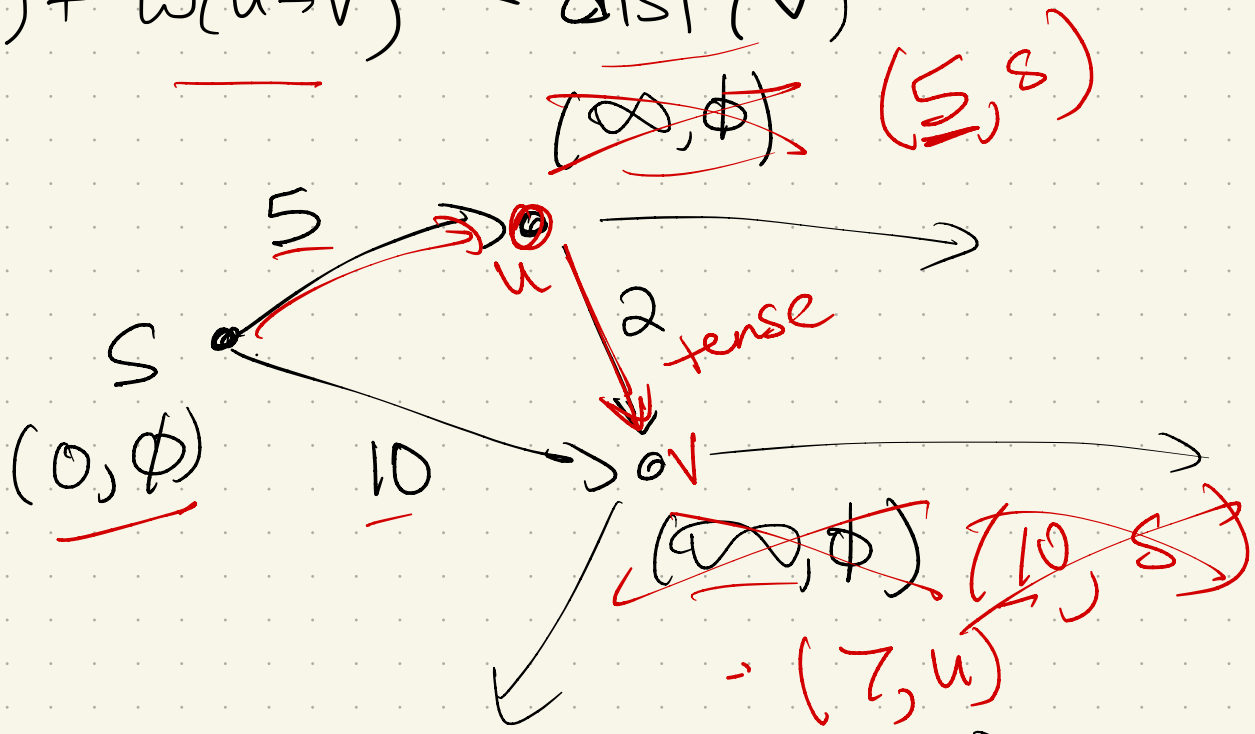
Initially:



We say an edge $\overrightarrow{uv}$ is tense if

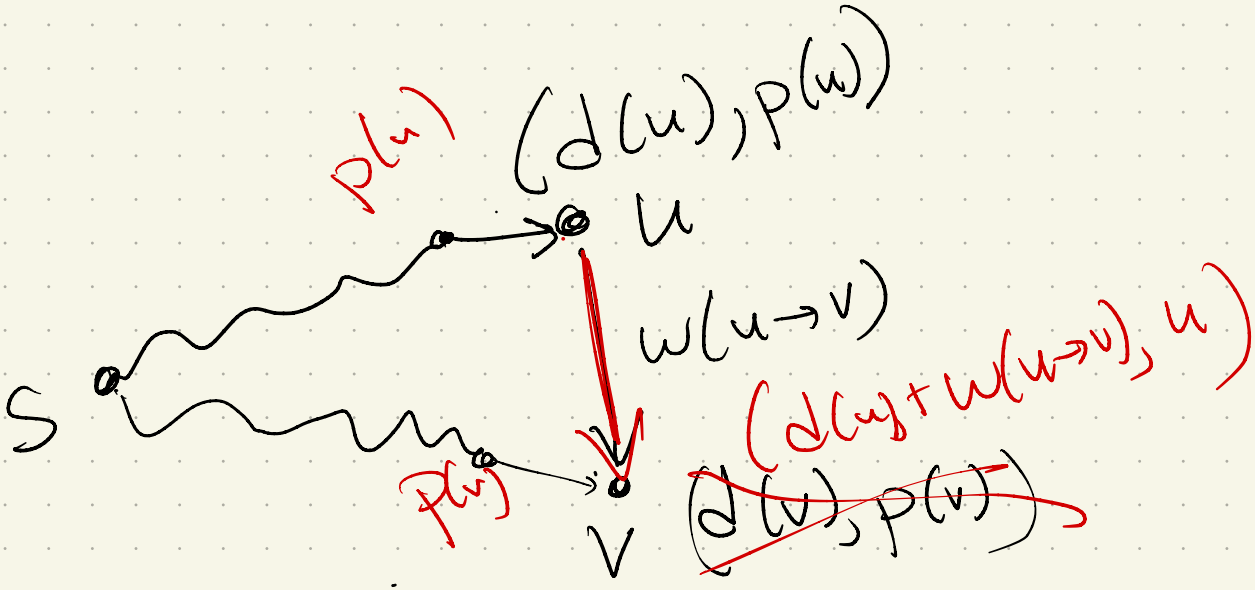
$$dist(u) + w(u \to v) < dist(v)$$

Initially:



Here:

In general:



Key idea of all algorithms:
Find tense edges & relax them:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

Then:

```
INITSSSP(s):
  dist(s) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← NULL
```

```
GENERICSSSP(s):
  INITSSSP(s)
  put s in the bag
  while the bag is not empty
      take u from the bag
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              put v in the bag
```

Claim: At any point in time, dist(v) is either ∞ or the length of some $s \rightsquigarrow v$ walk.
(which uses $\leq i$ edges, i = loop iteration)

Proof: Induction on while loop iterations.

Base case: loop iteration 1

at beginning, s has dist = 0 +
all others = ∞

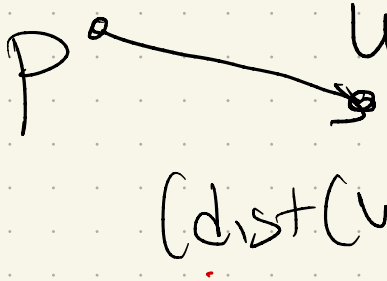
at end, s has dist = 0 still,
+ all neighbors u now have
dist(u) = w(s→u), which is a
length 1 walk. (Others are ∞)

Ind hyp:

In iteration $k-1$, the claim is true: all are $= \infty$, or some $s \rightsquigarrow v$ walk

Ind Step:

In iteration k: At beginning, we take out some vertex u.



By IH, dist(u) is the weight of some $s \rightsquigarrow u$ walk.

At end, all nbrs v of u are either unchanged (& so by IH are still either ∞ or length of $s \rightsquigarrow v$ walk)

or $u \to v$ was tense, so

now $dist(v) = dist(u) + w(u \to v)$.

Since $dist(u)$ is a $s \rightsquigarrow u$ walk, then $dist(v)$ is weight of the walk $(s \rightsquigarrow u) + (u \to v)$, which is a walk with one more edge at end.

(All other vertices are unchanged, so by IH are still ∞ or a $s \rightsquigarrow v$ walk.) ∎

Dijkstra's algorithm:

Algorithm 2: Dijkstra's Algorithm

Input: Digraph $G = (V, E)$ with edge-weights $w_e \geq 0$ and source vertex $s \in G$

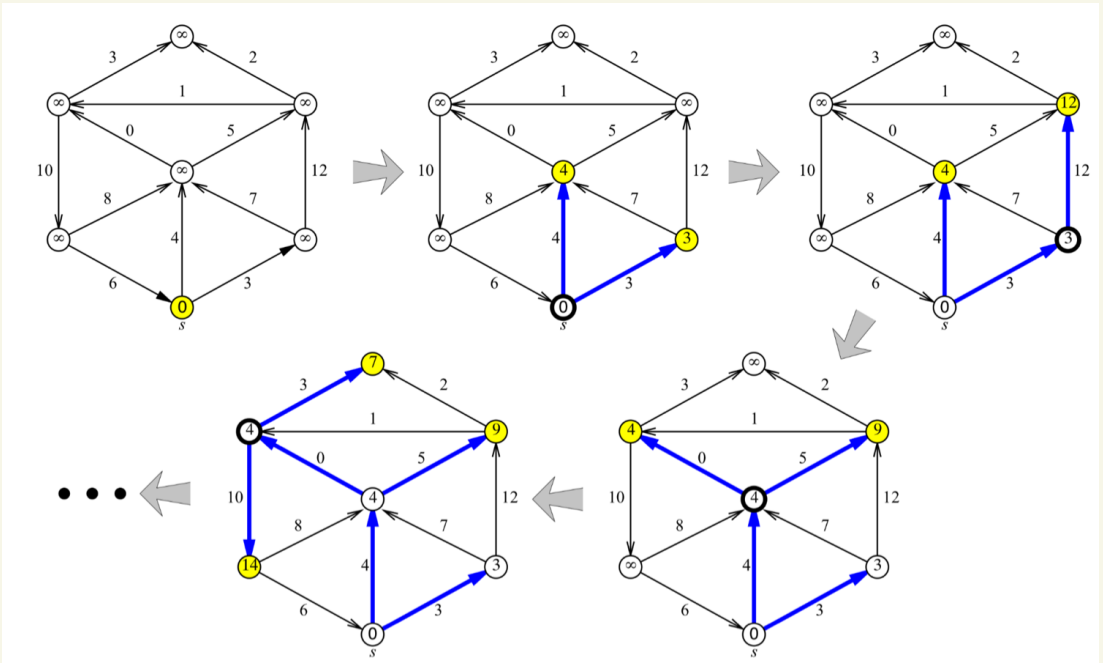
Output: The shortest-path distances from s to each vertex

```
2.1 add s to heap with key 0
2.2 for v ∈ V \ {s} do
2.3 |   add v to heap with key ∞
2.4 while heap not empty do
2.5 |   u ← deletemin
2.6 |   for v a neighbor of u do
2.7 |   |   key(v) ← min{ key(v), key(u) + w_uv }      // relax uv
```

min heap:

Extract min

replace old value if tense

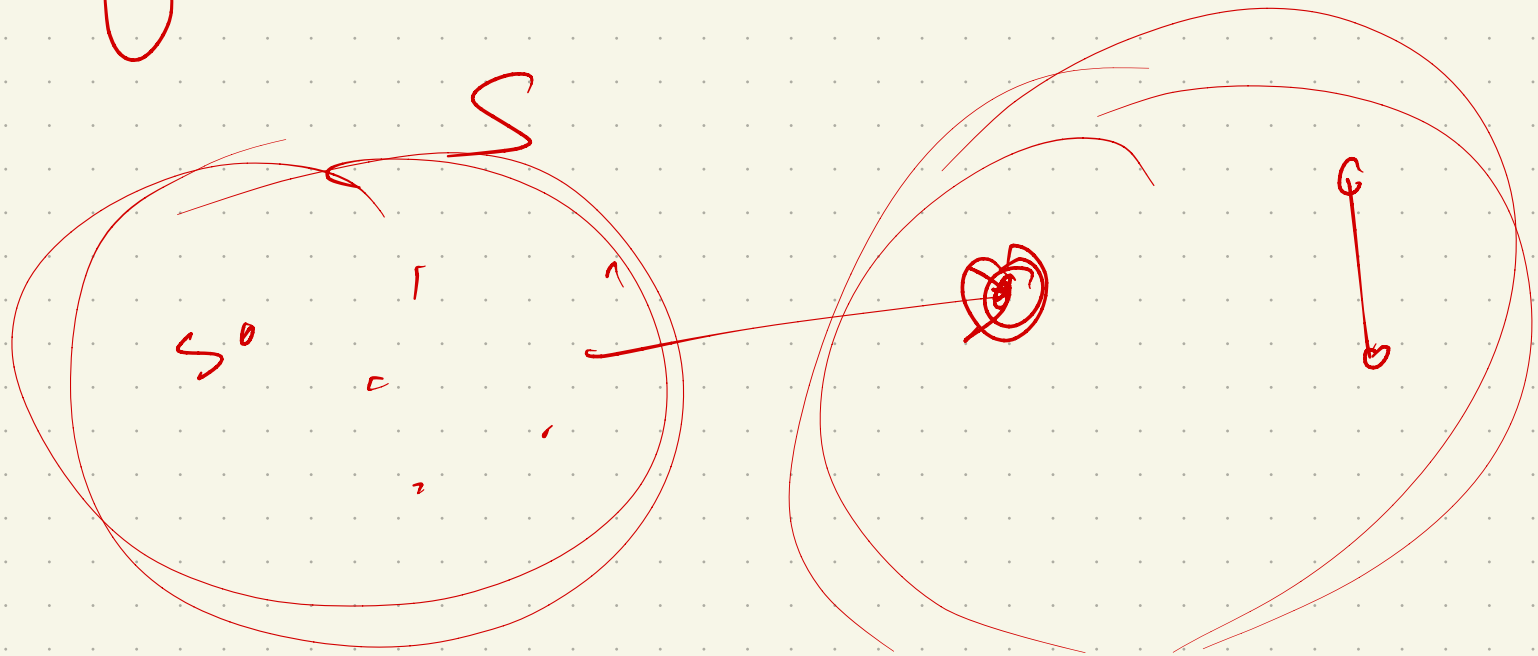


Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

Logic:

at each iteration, grab closest vertex.

All others can only have greater distance



Runtime:

Algorithm 2: Dijkstra's Algorithm

Input: Digraph $G = (V, E)$ with edge-weights $w_e \geq 0$ and source vertex $s \in G$

Output: The shortest-path distances from s to each vertex

```
2.1 add s to heap with key 0
2.2 for v ∈ V \ {s} do
2.3 |   add v to heap with key ∞
2.4 while heap not empty do
2.5 |   u ← deletemin
2.6 |   for v a neighbor of u do
2.7 |   |   key(v) ← min{ key(v), key(u) + w_uv }      // relax uv
```

$V \log V \approx$ time to do anything in heap

→ How many times can loop repeat?

V times, since we never readd vertex

Each time: alter key + deleteMin

$O(\log_2 V)$ binary heap $O(\log_2 V)$

V log V ← initialize

repeat V times:

delete ← log V

$d(x)$ → for each incident edge

alterkey ← log V

$$\sum_{x \in V} d(x) \cdot \log V = E \log V$$

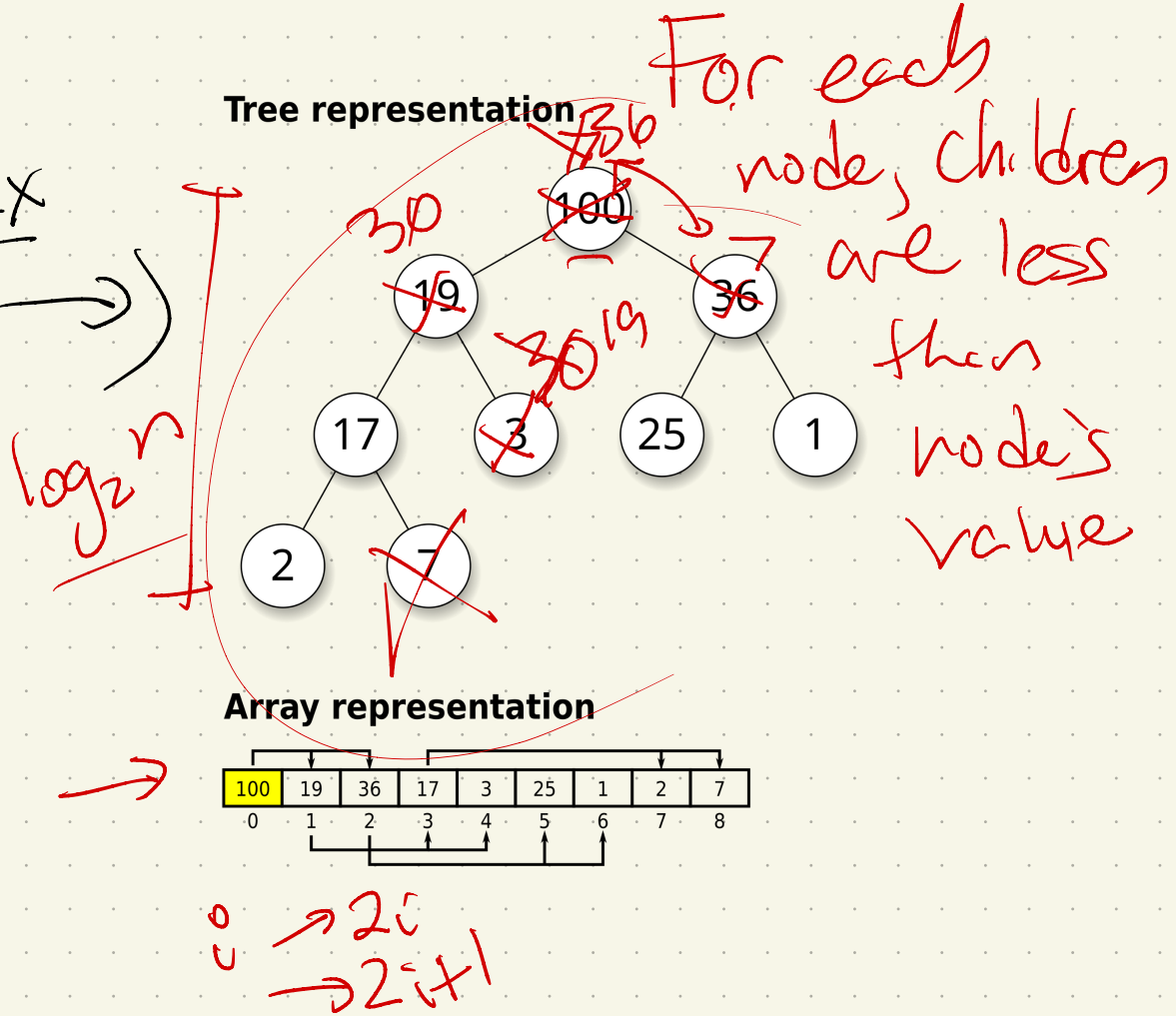
$O(E \log V)$

Other data structures:

Heaps are classic. → Priority queue

You probably saw binary heaps:

(This is a max binary heap →)



For each node, children are less than node's value

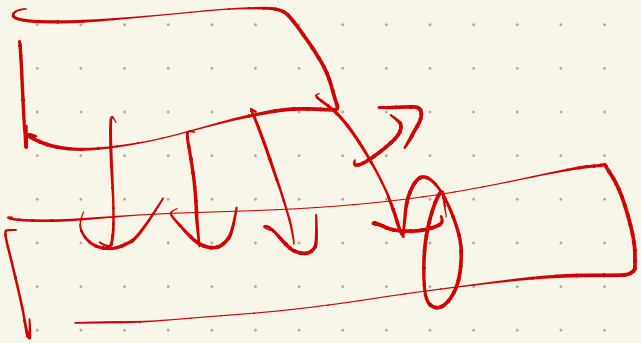
Next time?

More data structures!

- Fibonacci Heaps
- Van Emde Boas → probably a bit too long...

→ Binomial heap ☆

Amortized analysis



Negative edges:

We can still use Dijkstra idea, but the problem is that things can get "readded":

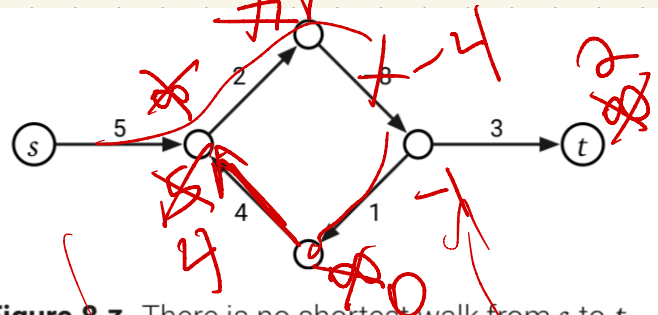


Figure 8.3. There is no shortest walk from s to t.

```
InitSSSP(s):
    dist(s) ← 0
    pred(s) ← Null
    for all vertices v ≠ s
        dist(v) ← ∞
        pred(v) ← Null
```

```
GenericSSSP(s):
    InitSSSP(s)
    put s in the bag
    while the bag is not empty
        take u from the bag
        for all edges u→v
            if u→v is tense
                Relax(u→v)
                put v in the bag
```

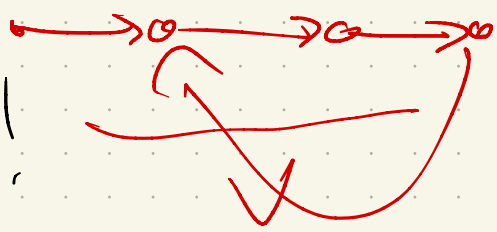
Bellman-Ford:

Relax edges for a while.

Stop when every edge has been relaxed at least once ← any path has length ≤ V-1

If any one is still tense:

you've relaxed ≥2 times!



Runtime: $O(V \cdot E)$

```
BellmanFord(s)
  InitSSSP(s)
  repeat V − 1 times
    for every edge u→v
      if u→v is tense
        Relax(u→v)
  for every edge u→v
    if u→v is tense
      return "Negative cycle!"
```

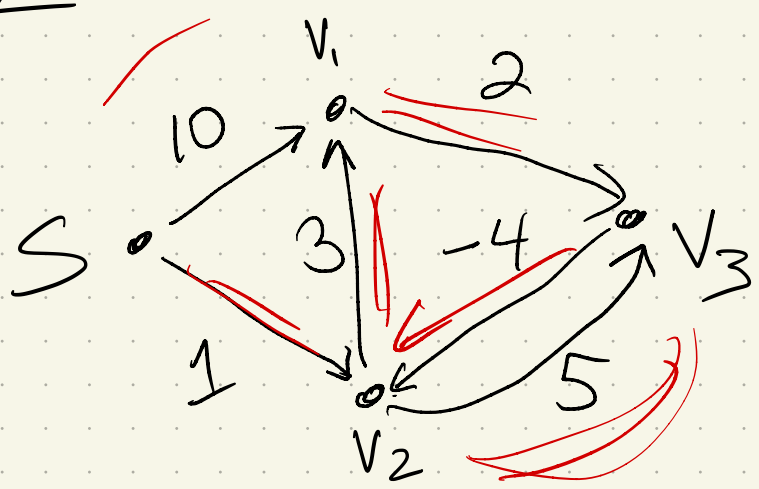
How to prove correctness?

Notation:

Let $dist_{\leq i}(v) :=$ length of shortest s to v path using $\leq i$ edges

↳ any path uses $\leq |V|$

Ex:



	S:	V1:	V2:	V3:
≤1	0	10	1	∞
≤2	0	4	1	6
≤3	0	4	1	6
⋮				

Claim: $\forall v \neq i$, after i iterations of B-F,

$$dist(v) \leq dist_{\leq i}(v)$$

Induction on i:

↖ # iterations

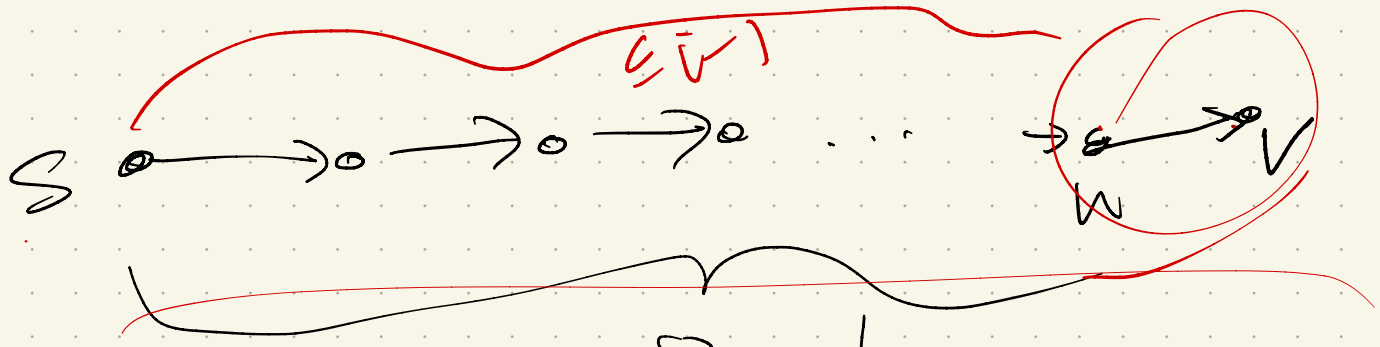
BC: $i = 0$: $s = 0$

all other v, $d(v) = \infty$

IH: After $i-1$ iterations, all tentative guesses are $\leq d_{i-1}(v)$.

IS: Now consider $d_i(v)$:

built from a path →



We know in round $i-1$, $dist(x) \leq d_{\leq i-1}(x)$ $\forall x \in V$.

Consider $u \to v$ in next round:

It was tense:

got shorter

or not:

same $\leftarrow$ ttl holds

Then:

Any simple path uses $\leq V-1$ edges, so $d(x) \leq d_{\leq V-1}(x)$.

The result!

Algorithm 3: The Bellman-Ford Algorithm

Input: A digraph $G = (V, E)$ with edge weights $w_e \in \mathbb{R}$, and source vertex $s \in V$

Output: The shortest-path distances from s to each vertex, or report that a negative-weight cycle exists

```
3.1 dist(s) = 0                          // the source has distance 0
3.2 for v ∈ V do
3.3 |   dist(v) ← ∞
3.4 for |V| iterations do
3.5 |   for edge e = (u, v) ∈ E do
3.6 |   |   dist(v) ← min{ dist(v), dist(u) + weight(e) }
3.7 If any distances changed in the last (n^th) iteration, output "G
    has a negative weight cycle".
```

Runtime:

$O(V \cdot E)$

Shortest paths from one source:

$O(E \log V)$ if no negative edges

$O(V \cdot E)$ if negative

if no weights: BFS $O(V+E)$

Computing MSSPs ← all pairs shortest paths

First attempt: [for each vertex v
Run SSSP(v)

Runtime: $O(V \times (\text{SSSP alg}))$

- no negative edge weights, Dijkstra was $O(E \log V) \Rightarrow O(VE \log V)$
- Bellman-Ford was $O(VE)$

$\Rightarrow O(V^2 E) \leq O(V^4)$

Can we do better? Yes!

Side question:

Since negative edges are bad, why can't we just re-weight?

Idea: increase all edge weights by some amount, so all positive:

+C

Doesn't work.

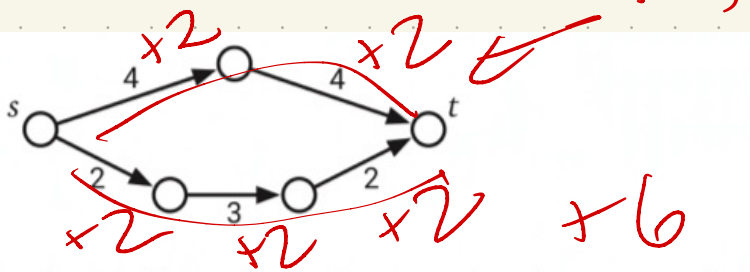


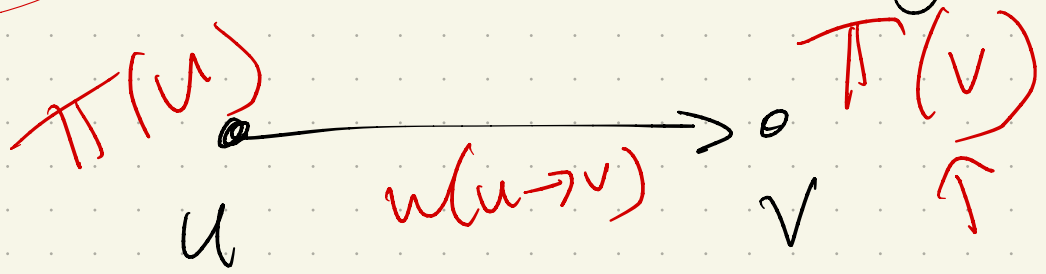
Figure 9.1. Increasing all the edge weights by 2 changes the shortest path from s to t.

Why? # of edges is also additive: +(path length)·C

Another idea (that works):

Suppose each v has a price attached, $\pi(v)$. ← anything

(Still have edge weights.)



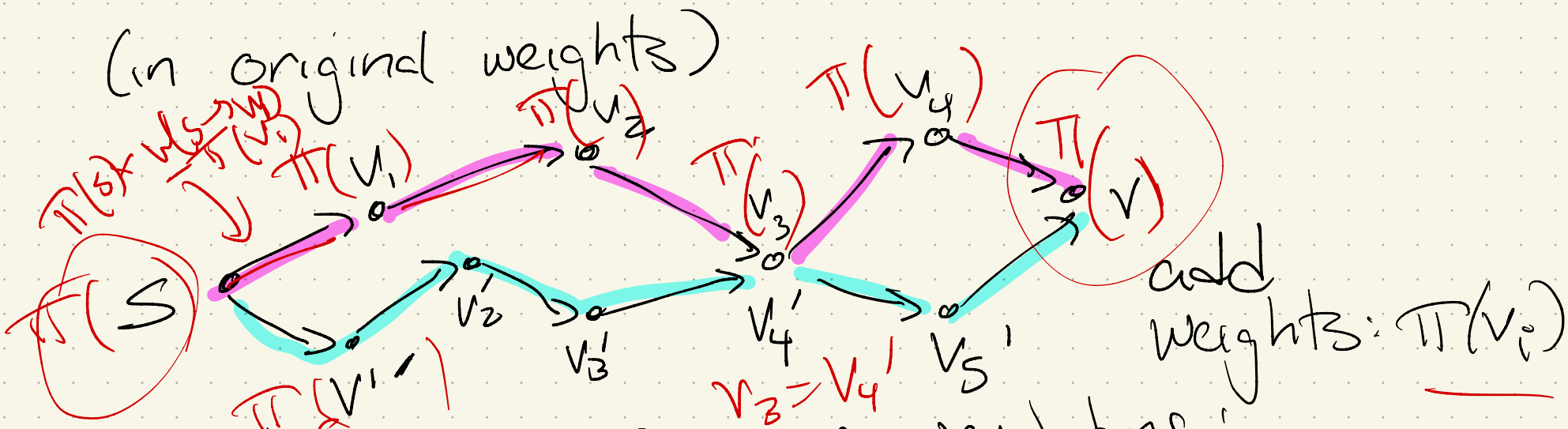
Define a new function w':

$$w'(u \to v) = \pi(u) + w(u \to v) - \pi(v)$$

What happens to paths? → goal is consistent edge

Claim: under w', shortest paths are the same as under w.

Why? Consider 2 $s \rightsquigarrow v$ paths:



purple path after reweighting:

$$\big(\pi(s) + w(s \to v_1) - \pi(v_1)\big) + \big(\pi(v_1) + w(v_1 \to v_2) - \pi(v_2)\big) + \big(\pi(v_2) + w(v_2 \to v_3) - \pi(v_3)\big) + \cdots$$

$$= \pi(s) + \text{length under } w - \pi(v)$$

blue path:

$$\pi(s) + \text{length under } w - \pi(v)$$

Johnson's algorithm for MSSP:

use this new kind of weight function!

- Run Bellman-Ford once from some $s \in V$
  - ↳ no negative cycles or give up
- Reweight (if it works):

Let $\Pi(v) =$ dist from s to v

compute via B-F $= d(s,v)$

$$\text{or } w'(u \to v) = \Pi(u) + w(u \to v) - \Pi(v)$$

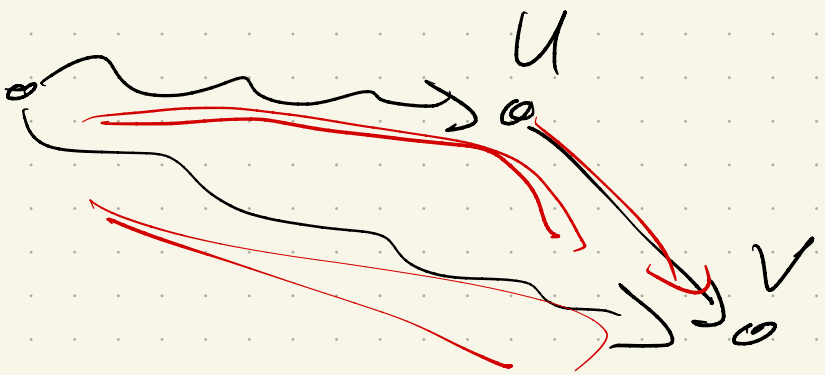
$$= \text{dist}(s,u) + w(u \to v) - \text{dist}(s,v)$$

Claim: Under w', all weights are ≥ 0.

use Dijkstra's

Why?

No tense edges: s



so $d(s,u) + w(u \to v) \geq d(s,v)$

not <

Then $\pi(u) = d(s,u)$

$\pi(v) = d(s,v)$

$\therefore w'(u \to v) = d(s,u) + w(u \to v) - d(s,v)$

≥ 0

So - Algorithm:

Runtime:

```
JOHNSONAPSP(V, E, w):
  ⟨⟨Add an artificial source⟩⟩
  add a new vertex s
  for every vertex v
      add a new edge s→v
      w(s→v) ← 0
  ⟨⟨Compute vertex prices⟩⟩
  dist[s, ·] ← BELLMANFORD(V, E, w, s)
  if BELLMANFORD found a negative cycle
      fail gracefully
  ⟨⟨Reweight the edges⟩⟩
  for every edge u→v ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  ⟨⟨Compute reweighted shortest path distances⟩⟩
  for every vertex u
      dist'[u, ·] ← DIJKSTRA(V, E, w', u)
  ⟨⟨Compute original shortest-path distances⟩⟩
  for every vertex u
      for every vertex v
          dist[u, v] ← dist'[u, v] − dist[s, u] + dist[s, v]
```

Figure 9.2. Johnson's all-pairs shortest paths algorithm

O(VE)

weird

not B-F

$VE \log V$

(Compare to naive: $O(V^2E) = O(V^4)$

Can we get better?

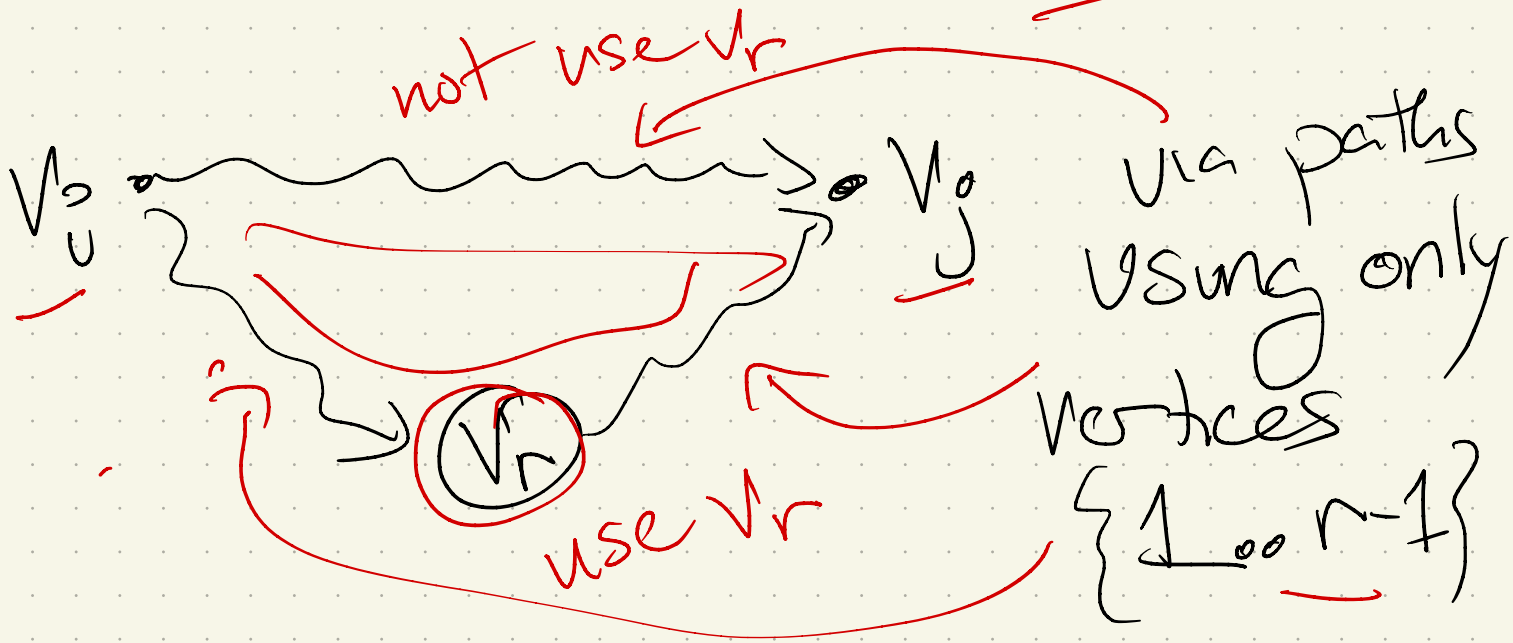
Floyd-Warshall:

order vertices $1, \ldots, V \rightarrow v_1, \ldots v_V$

Let $d(x, y, r)$ = 

best length path via vertices labeled $1 \ldots r$

Then recursion

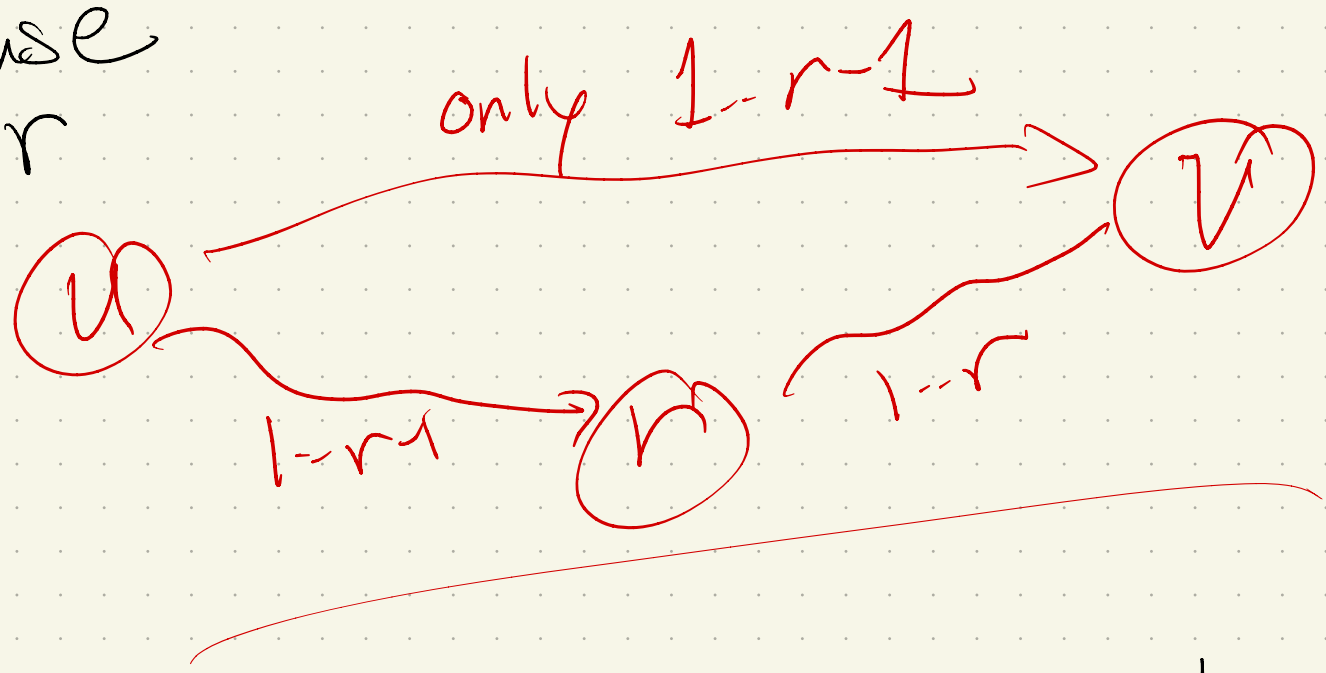


Recursion:

$$dist(u,v,r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min \begin{cases} dist(u,v,r-1) \\ dist(u,r,r-1) + dist(r,v,r-1) \end{cases} & \text{otherwise} \end{cases}$$

use vertex r

don't use vertex r



code ↝

```
KleeneAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r − 1] < dist[u, r, r − 1] + dist[r, v, r − 1]
                  dist[u, v, r] ← dist[u, v, r − 1]
              else
                  dist[u, v, r] ← dist[u, r, r − 1] + dist[r, v, r − 1]
```

$O(V^3)$

Runtime:

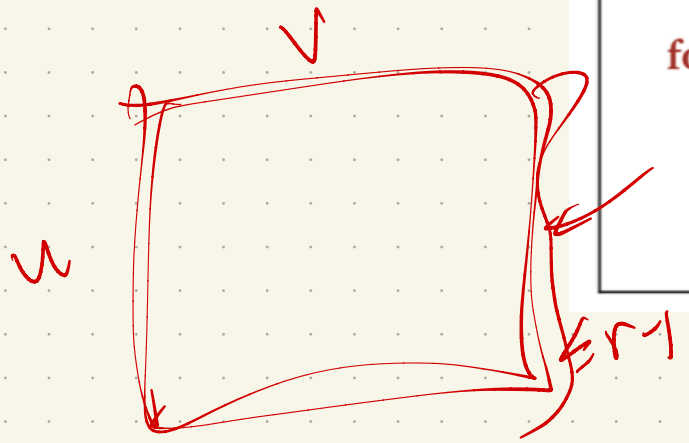
← V matrices

$O(V^3)$

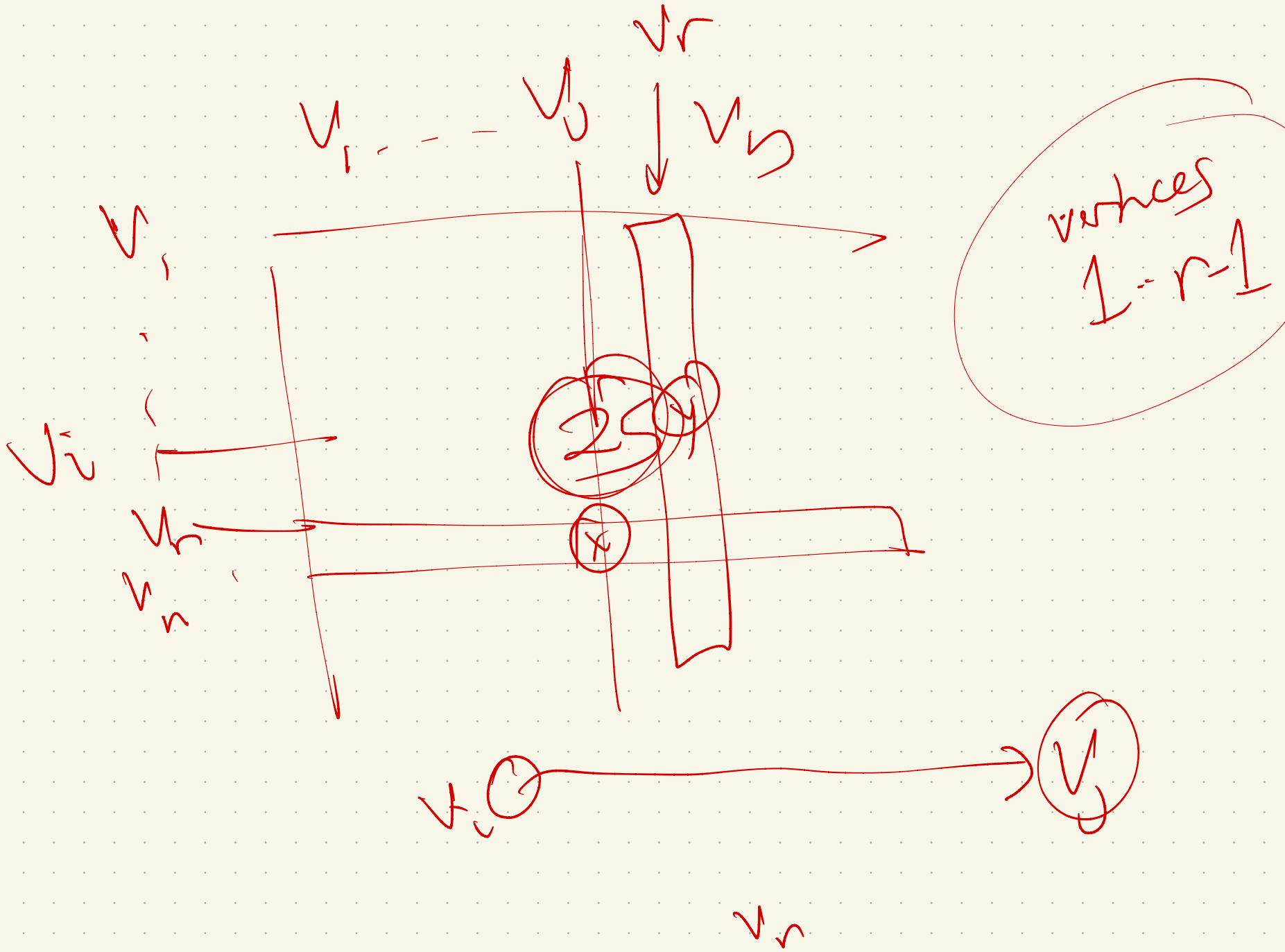
save space →

```
FloydWarshall(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```



use r if it's better


v_r
v_1
v_j
v_b
v_1
v_i
v_r
v_n
250
vertices
1 - r-1
v_i
v_j
v_n