Graphs

Dijkstra's Algorithm

all edges are non-negative

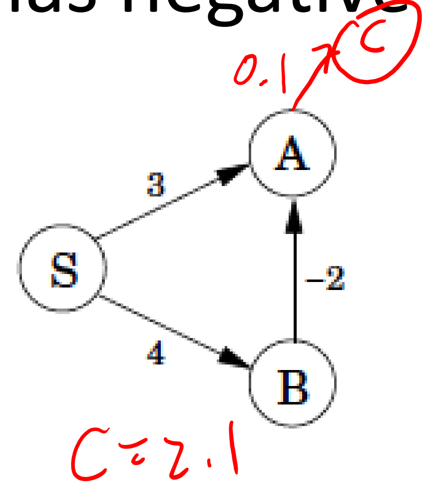
$O((|V|+|E|)\log|V|)$

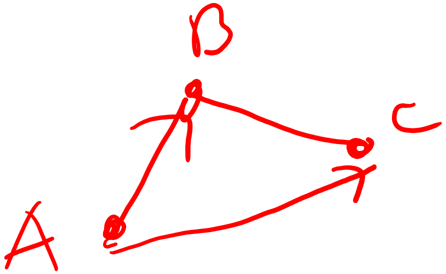
```
for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)   (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):  and v ∈ H:
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

What About Negative Edges?

- What if the graph has negative edges?





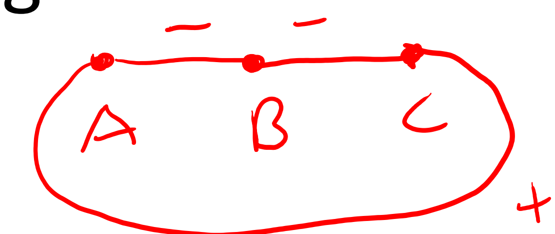
dist: S A B
0 ~~∞~~ ~~3~~ 2 ~~∞~~ 4

H: ~~S~~ ~~A~~ ~~B~~
~~0~~ ~~∞~~ 3 ~~∞~~ 4

- What does this even mean? Any examples of a real-world graph with negative edges?

social: friends / enemies
+ −

~~che~~ molecular transformation

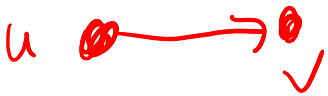


In-Class Exercise

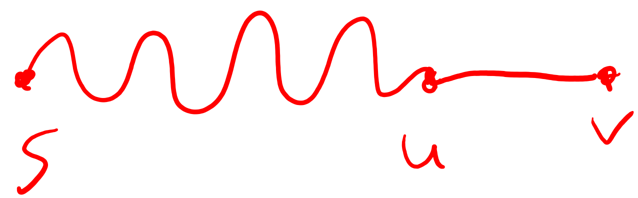
- Does Dijkstra’s Algorithm work when there are negative edges?
- If yes, explain, if no, give counterexample

Dijkstra's Algorithm with Negative Edges

- The heart of Dijkstra's Algorithm can be phrased as an update procedure update((u,v))



$$\underline{\texttt{procedure update}}(\,(u,v)\in E\,)$$
$$\texttt{dist}(v)=\min\{\texttt{dist}(v),\texttt{dist}(u)+l(u,v)\}$$

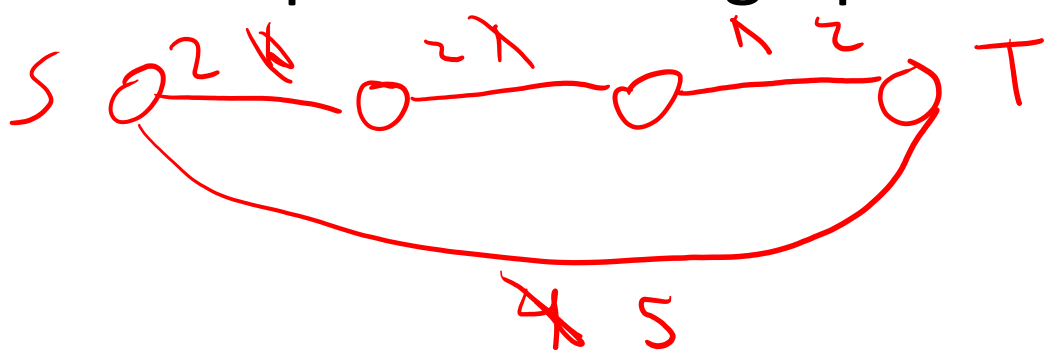


Negative edges

- Suppose you have a graph with negative edges (but no negative cycles).

$c > (-1)$(smallest negative)

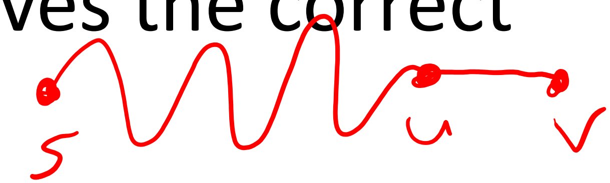
- I propose the following algorithm for finding shortest paths: Add some amount *c* to each edge so that all edges have non-negative costs, and then find shortest paths in this graph.
- Will it work?



Update with Negative Edges

update((u,v))

- The Update operation will never underestimate the length of the shortest path: it is safe
- If we are trying to get the distance from *s* to *v*, then the Update operation gives the correct result if the following hold:



  - *u* is the node right before *v* in the actual shortest path from *s* to *v*
  - We have the correct shortest path from *u* to *v*

Bellman-Ford Algorithm

- Consider the actual shortest path from node *s* to node *t*

- What is the maximum length of this path?

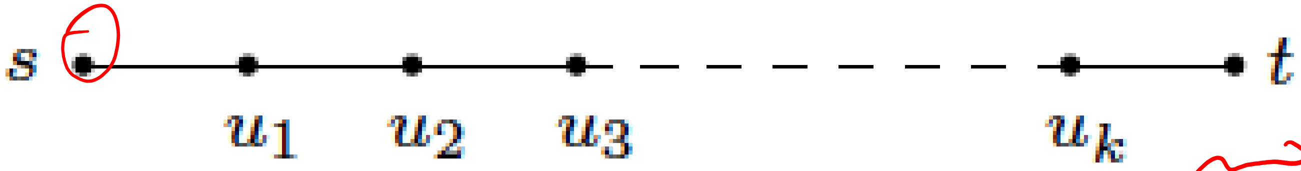
in worst case, goes through every node

N−1 edges in path

Bellman-Ford Algorithm

actual s.p. → t

update(u_2,u_3)

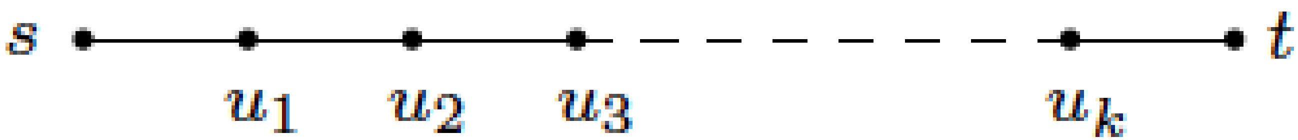


update(s,u_1)
update(u_1,u_2)
update(u_2,u_3)
⋮
update(u_k,t)

- If we correctly get the distance from *s* to u_1, then we can get the distance from *s* to u_2
- If we correctly get the distance from *s* to u_2, then we can get the distance from *s* to u_3
- Etc.
- We have to do the updates in the right order! (though not necessarily in a row)

consecutively

Bellman-Ford Algorithm

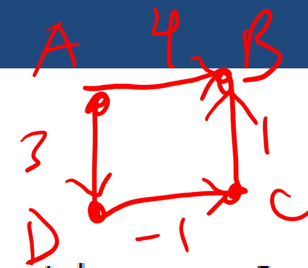


- So if the path is at most |V| - 1 edges long, then let's just update *all* the edges |V| - 1 times!
- This will guarantee that we do the updates in the correct order!

Bellman-Ford Algorithm

A → B

A → D → C → B



procedure shortest-paths(G, l, s)
Input: Directed graph $G = (V, E)$;
edge lengths $\{l_e : e \in E\}$ with no negative cycles;
vertex $s \in V$
Output: For all vertices u reachable from s, dist(u) is set
to the distance from s to u.

for all $u \in V$:
    dist(u) $= \infty$
    prev(u) $=$ nil

$O(N)$

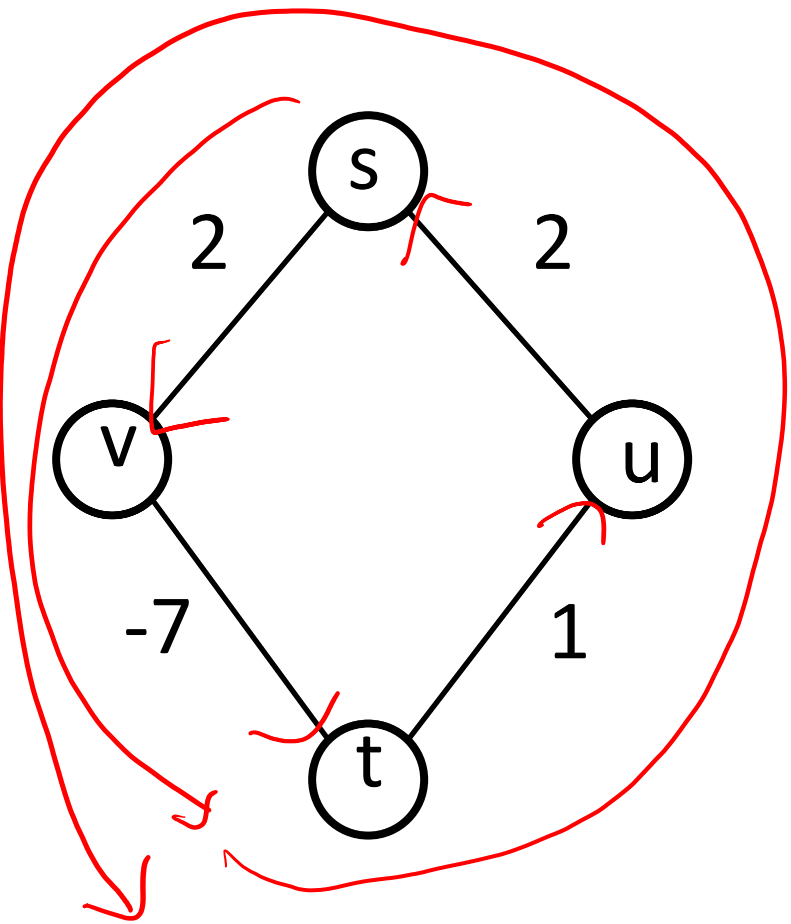
dist(s) $= 0$
repeat $|V| - 1$ times:
    for all $e \in E$:
        update(e)

$O(N) \cdot O(M)$

$= O(N \cdot M)$

round 1	R1	R2	R3
for all edges (x,y) update(x,y)	ud(A,B)	ud(A,B)	ud(A,B)
	ud(A,D)*	ud(A,D)	ud(A,D)
	ud(D,C)	ud(D,C)*	ud(D,C)
	ud(C,B)	ud(C,B)	ud(C,B)*

Graphs with Negative Cycles

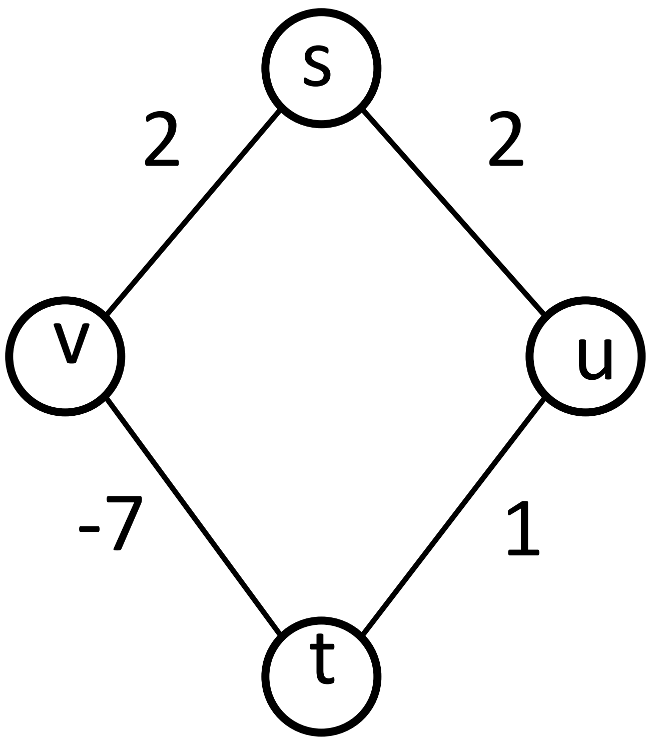


What is the shortest path from *s* to *t*?

$2 + -7 + 1 + 2 + 2 + -7$

$= -7$

Graphs with Negative Cycles



It doesn't make sense to talk about shortest paths in a graph with negative cycles!

Where did we go wrong in Bellman-Ford?

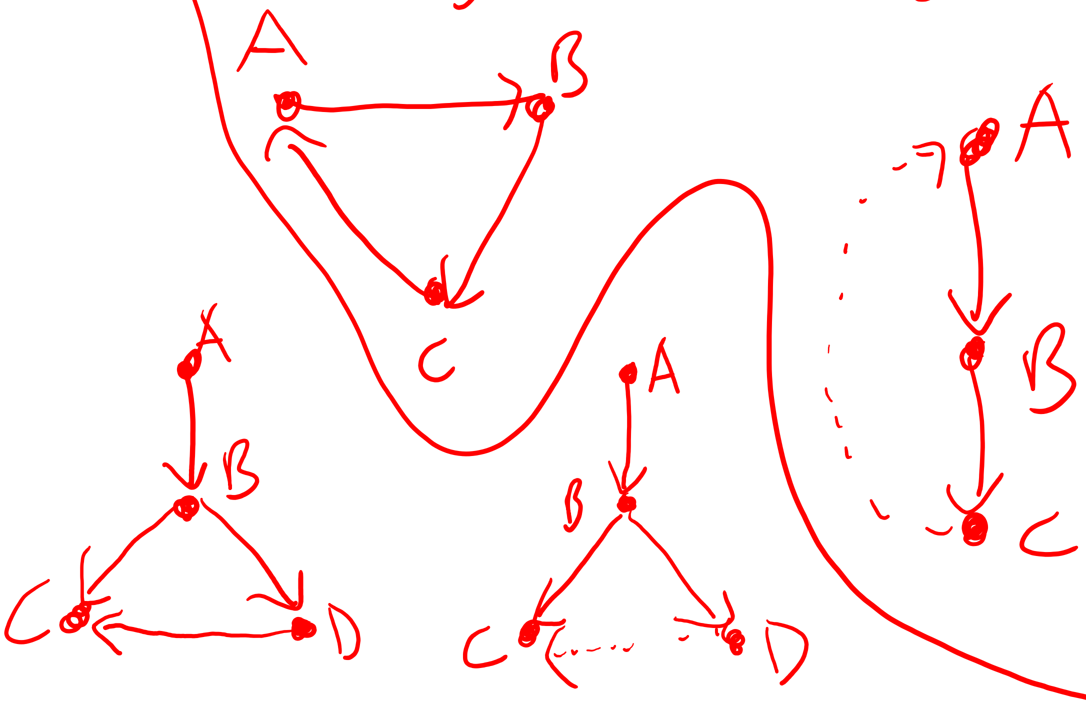
Graphs with Negative Edges

- Fortunately, it’s easy to tell if there is a negative cycle
- Just do an extra round of update- if any distance values change, you have a negative cycle!

Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph without cycles
- How can you tell whether a directed graph has a cycle?

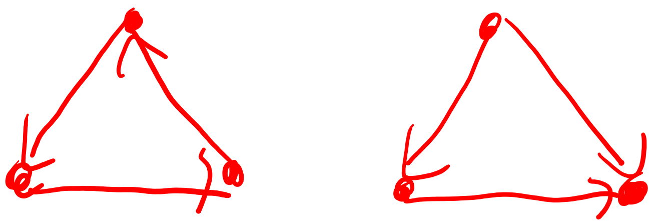
DFS → back edge



back edge: edge leading from a descendant to ancestor

Directed Acyclic Graphs (DAGs)

- A DAG is a directed graph without cycles
- How can you tell whether a directed graph has a cycle?
- Look for the presence of back edges!



Directed Acyclic Graphs (DAGs)

Claim: A directed graph has a cycle if and only ~~iff~~ if its DFS reveals a back edge

Proof: First, prove that if DFS reveals a back edge, the graph has a cycle. If there is a back edge (v, u), that edge plus the tree edges leading from $u \rightarrow v$, form a cycle.

Directed Acyclic Graphs (DAGs)

Next, we show that if the graph has a cycle, DFS will reveal a back edge.

Suppose there is a cycle. Let u denote the first node from that cycle to be visited by DFS. Let v denote the node right before u in that cycle. When u is visited, none of the ~~other~~ remaining cycle has been visited, so DFS will explore all of those nodes from u. When it reaches v, v will thus be a descendant of u, so (v, u) will be a back edge. □

Directed Acyclic Graphs (DAGs)

- What good are DAGs?
- Often used to model situations with constraints
- Every day…
  - First you get out of bed 1
  - Then you could eat breakfast, brush your teeth, get dressed
    4 2 3
  - All three of those have to be done before you leave the house 5
  - Etc.