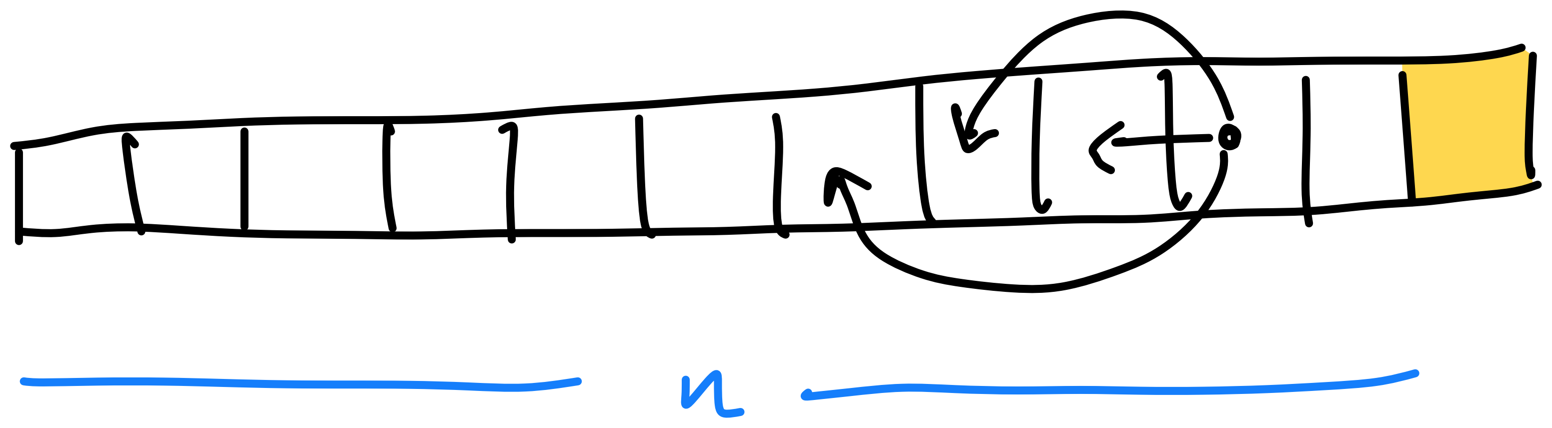
Lecture 15

Dynamic programming IV: The Bellman-Ford algorithm

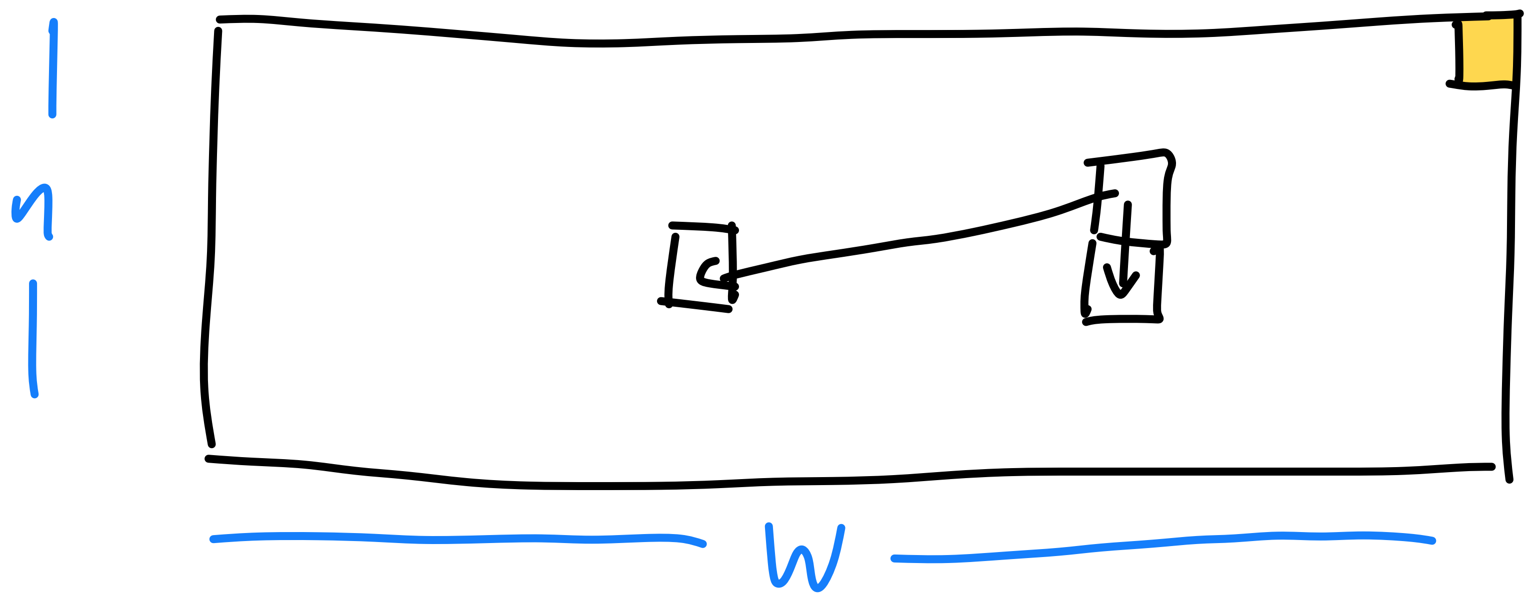
Chinmay Nirkhe | CSE 421 Winter 2026

Dynamic programming patterns

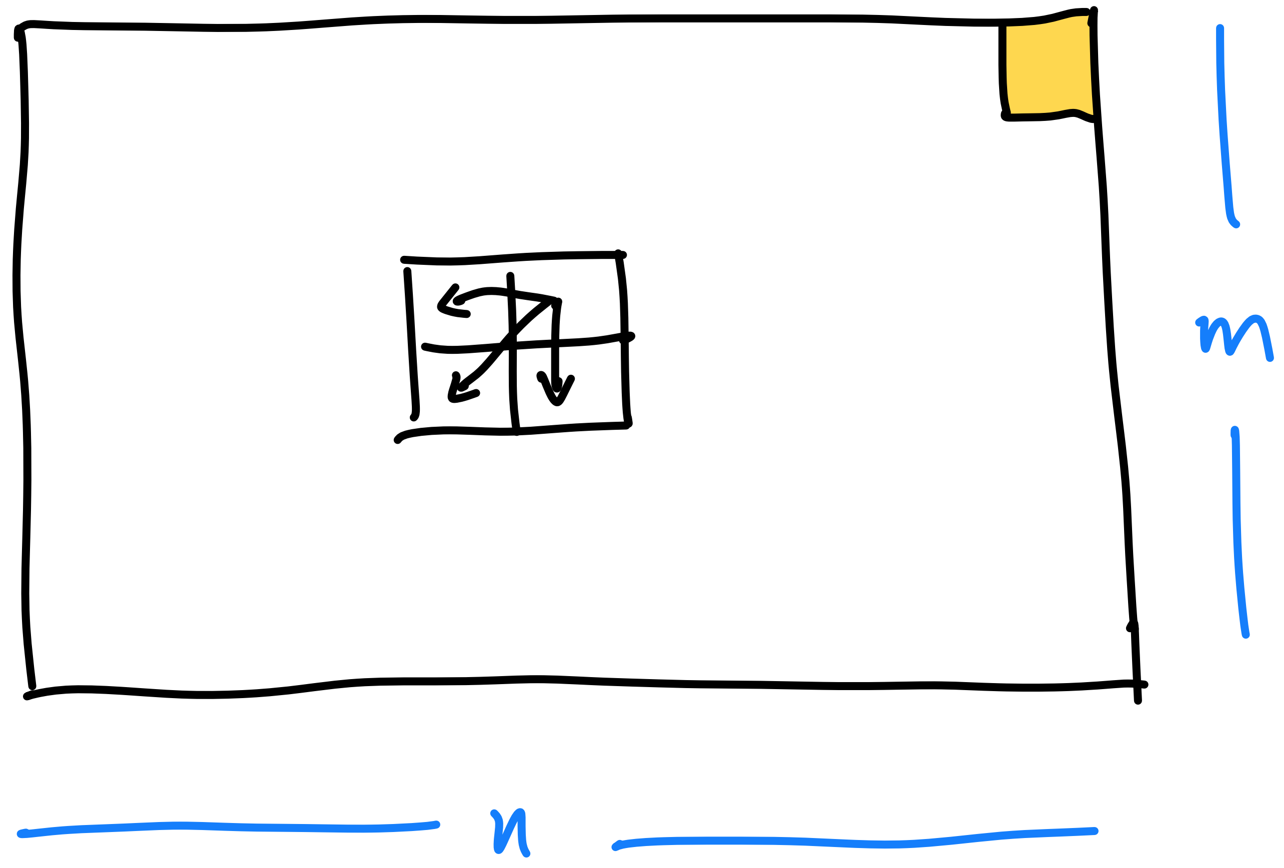
Tribonacci



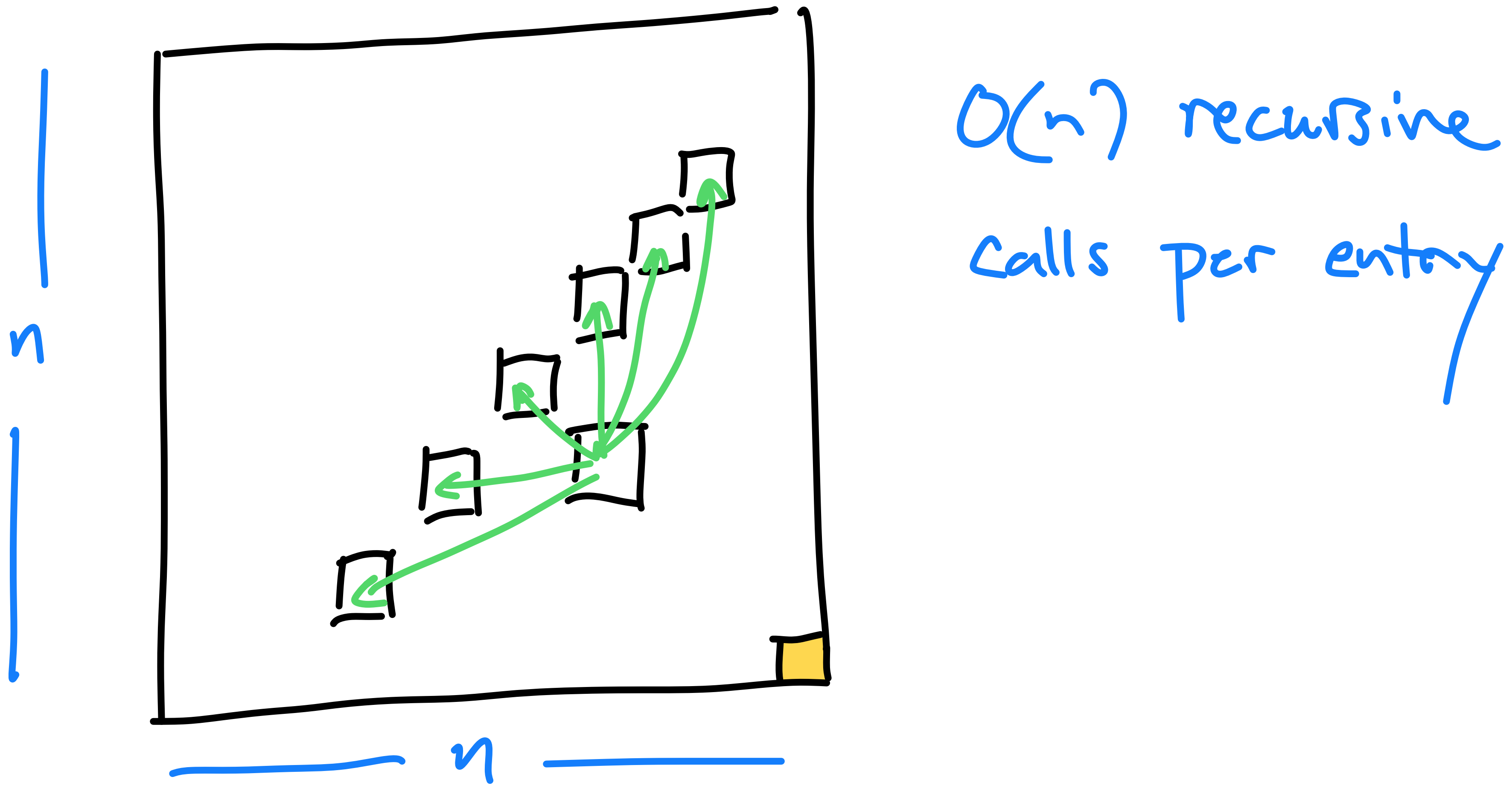
Knapsack



Edit distance



RNA second sequence



Top-down vs bottom-up DP algorithms

- So far we have seen that the recursive subproblems in DP algorithms are always “smaller”. Examples
  - Knapsack: $f(n, W')$ depends on $f(n-1, W'')$ for $W'' \leq W'$
  - RNA SS: $f(i, j)$ depends on $f(i', j')$ where $|j' - i'| < |j - i|$
- Yields a “bottom-up” ordering for filling the memoization table
- Instead we could fill up the table “top-down”

Top-down vs bottom-up DP algorithms

- In a “top-down” DP algorithm $f(x)$
  - Conclude that $f(x)$ can be defined recursively based on $f(y_1), f(y_2), \ldots f(y_k)$
  - For each y_j, check if $f(y_j)$ has been previously calculated
    - If yes, use the value of $f(y_j)$
    - If not, recursive compute $f(y_j)$
- Overall, runtime is asymptotically the same! Each square of the memo is only computed **at most** once.

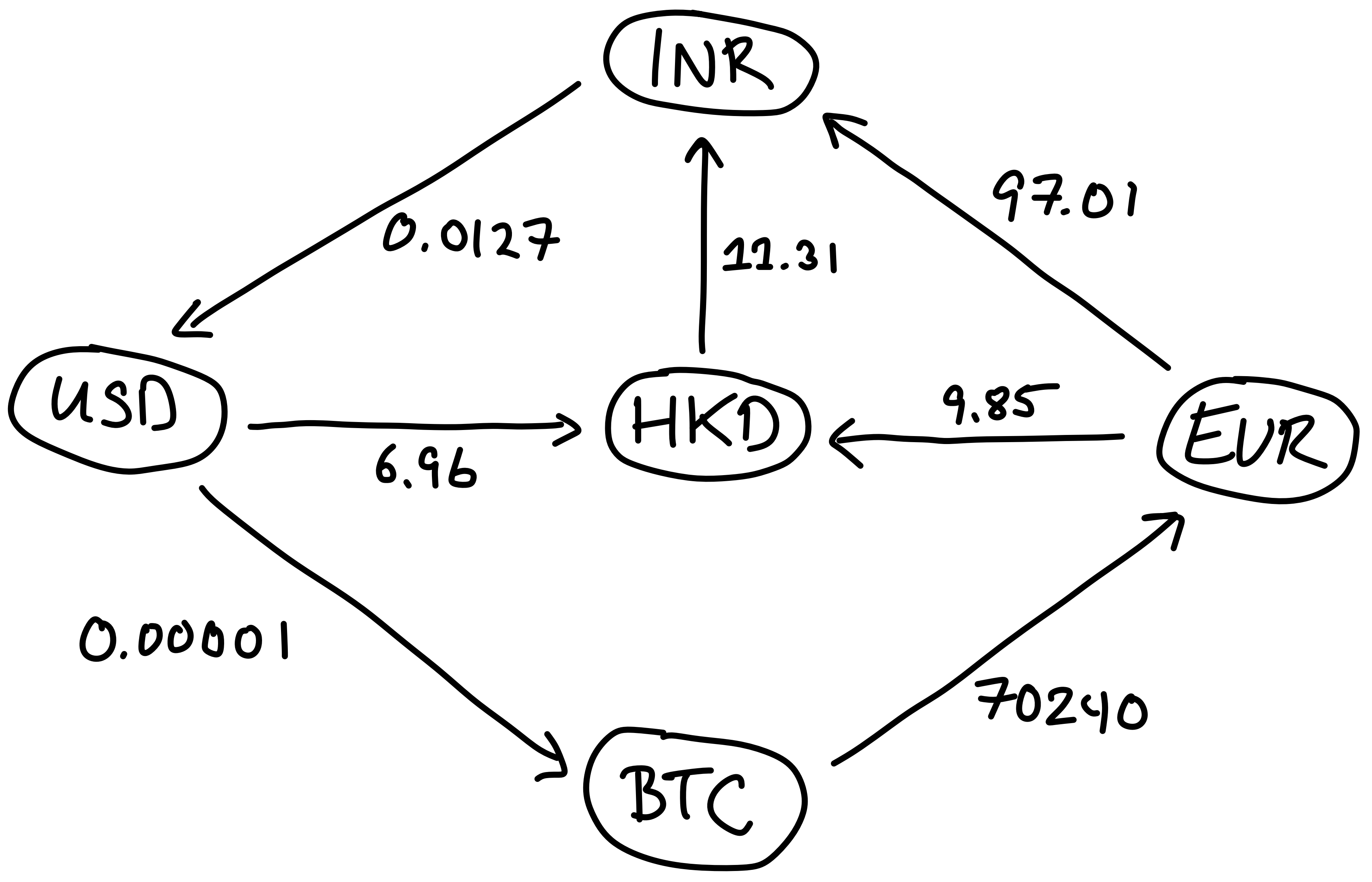
Top-down vs bottom-up DP tradeoffs

- In top-down approaches, not all squares may get calculated
  - Can yield constant factor savings in terms of runtime
- However, the recursion stack usually scales poorly in top-down approaches
  - For example, in Tribonacci, recursion stack would be $\Omega(n)$ in depth
  - Recursion stack is often in computer's memory while data being manipulated is expressed on the hard drive
  - Can yield memory overflow errors if not carefully programmed
- Top-down is better when the order of filling out squares isn't well defined
  - Occurs in graph DP algorithms like Bellman-Ford which we see soon
  - In such cases, a more sophisticated analysis is needed to argue that recursive defs. are not cyclical

Graph dynamic programming

Currency exchange

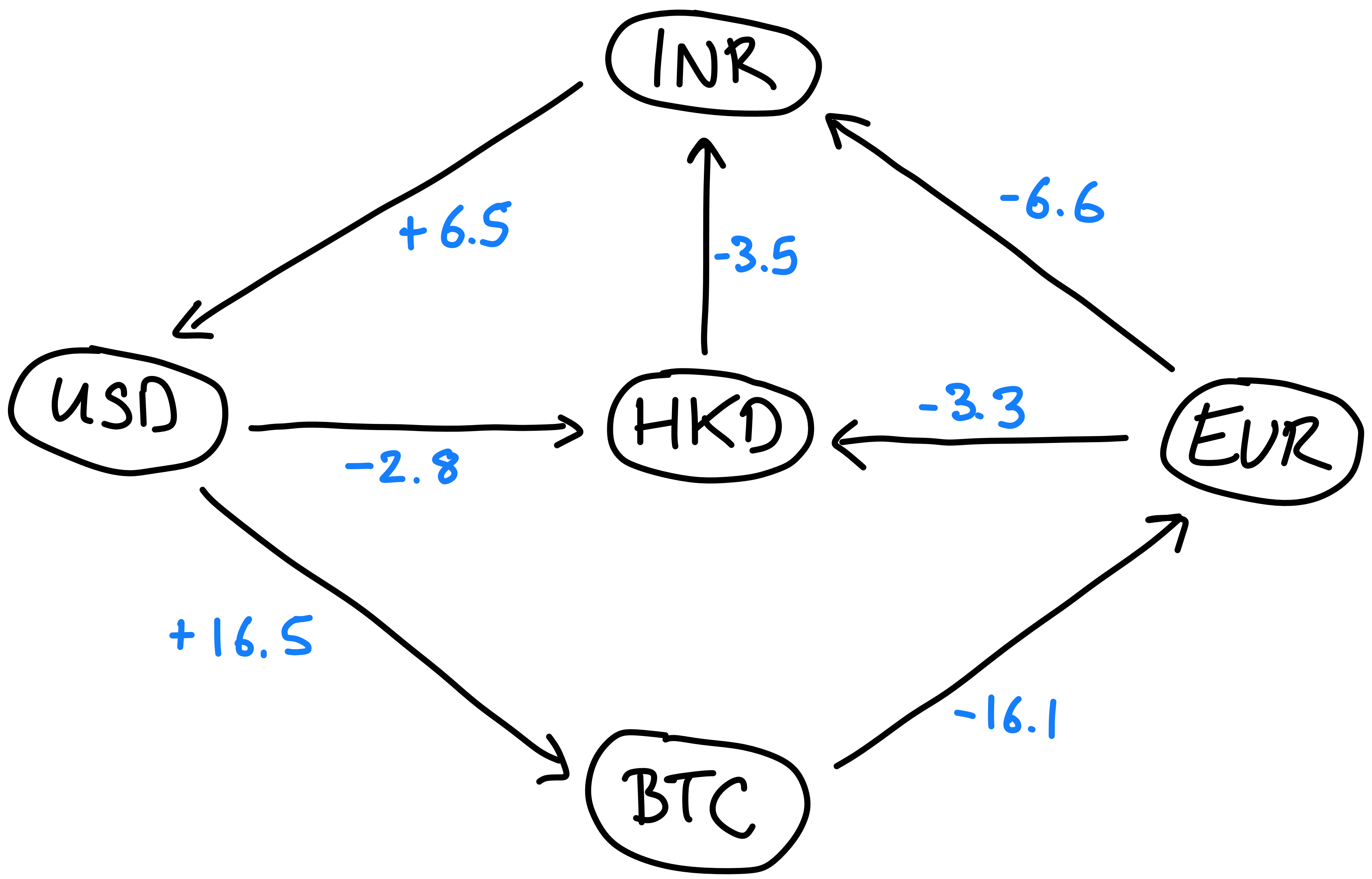
- USD to BTC: 0.00001
- BTC to EUR: 70,240
- INR to USD: 0.0127
- EUR to INR: 97.01
- EUR to HKD: 9.85
- HKD to INR: 11.31
- USD to HKD: 6.96



Currency exchange

- USD to BTC: 0.00001
- BTC to EUR: 70,240
- INR to USD: 0.0127
- EUR to INR: 97.01
- EUR to HKD: 9.85
- HKD to INR: 11.31
- USD to HKD: 6.96

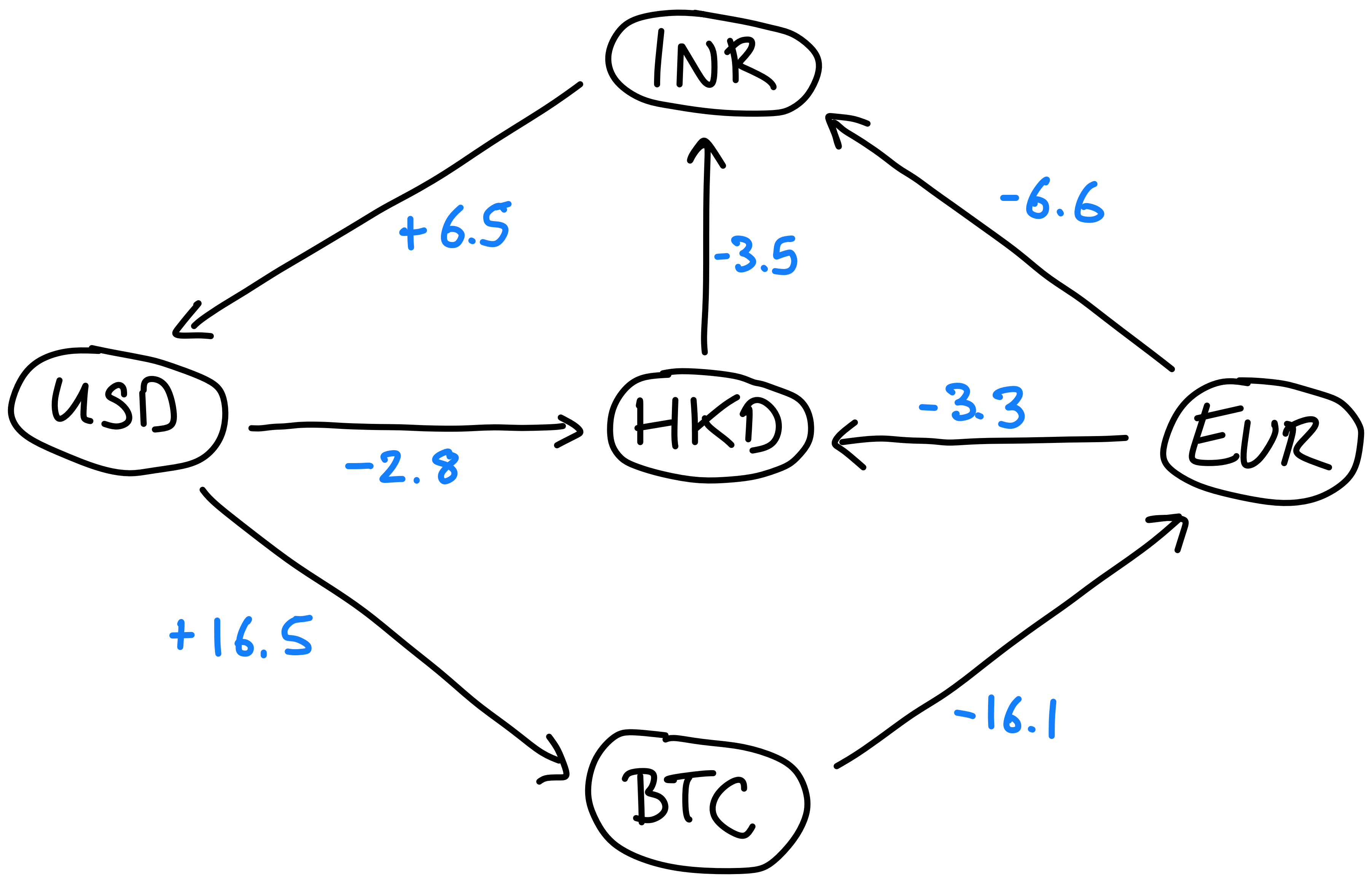
Set edge weight to $\log_2(1/r) = -\log_2(r)$



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

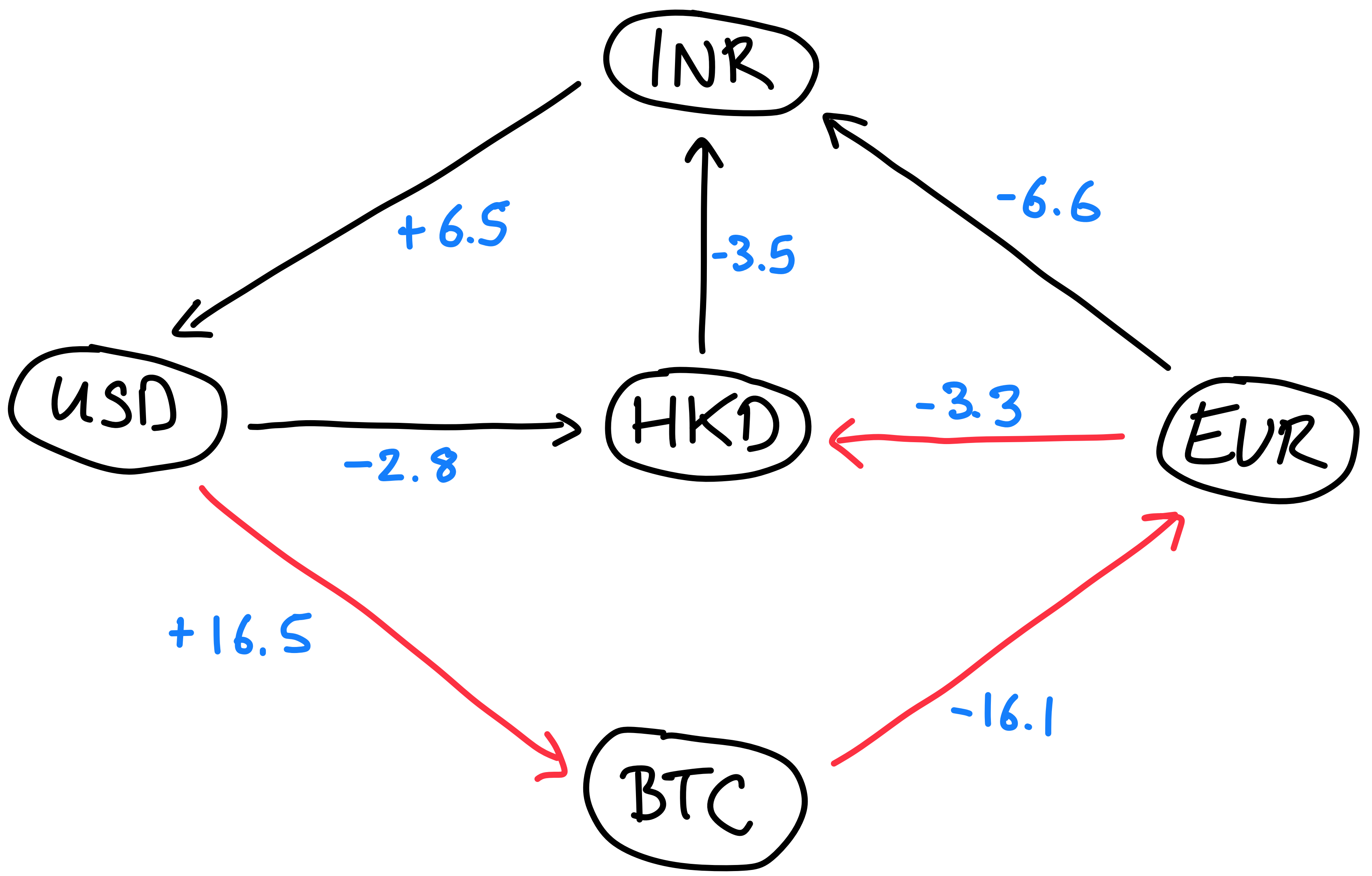
- A path $p : u \rightsquigarrow v$ of net weight w implies a currency conversion from 1 unit of u to 2^{-w} units of v
- Finding a path of least weight from u to v yields the best seq. of currency exchanges
- Direct conversion of USD to HKD yields $2^{2.8}$ HKD per USD



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

- A path $p : u \rightsquigarrow v$ of net weight w implies a currency conversion from 1 unit of u to 2^{-w} units of v
- Finding a path of least weight from u to v yields the best seq. of currency exchanges
- Direct conversion of USD to HKD yields $2^{2.8}$ HKD per USD
- USD→BTC→EUR→HKD yields $2^{-(16.5-16.1-3.3)} = 2^{2.9}$ HKD per USD



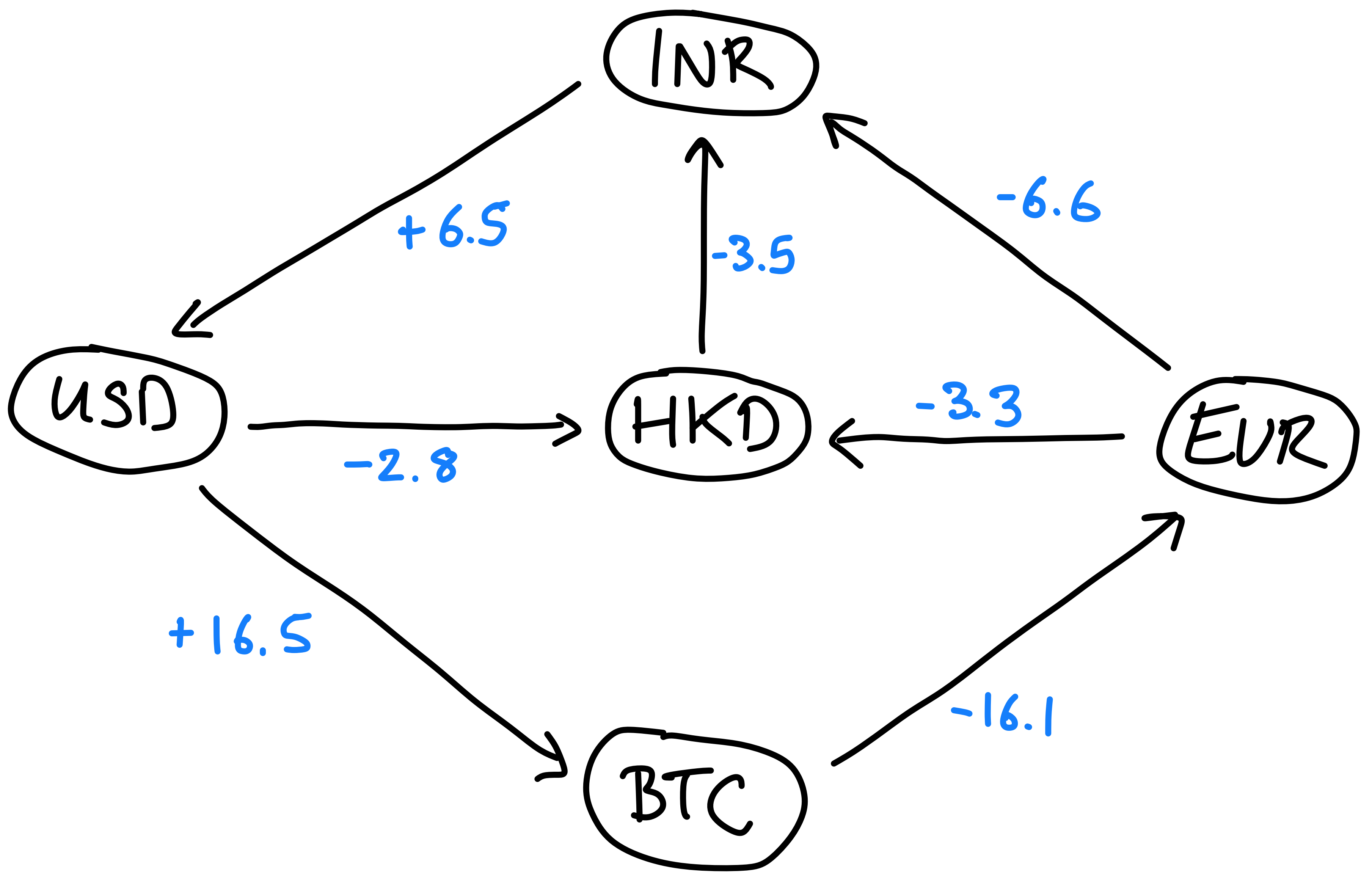
Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

- What happens if HKD to INR rate changes from

  INR = $2^{3.5} \approx 11.3$ HKD to

  INR = $2^{4.0} = 16$ HKD?



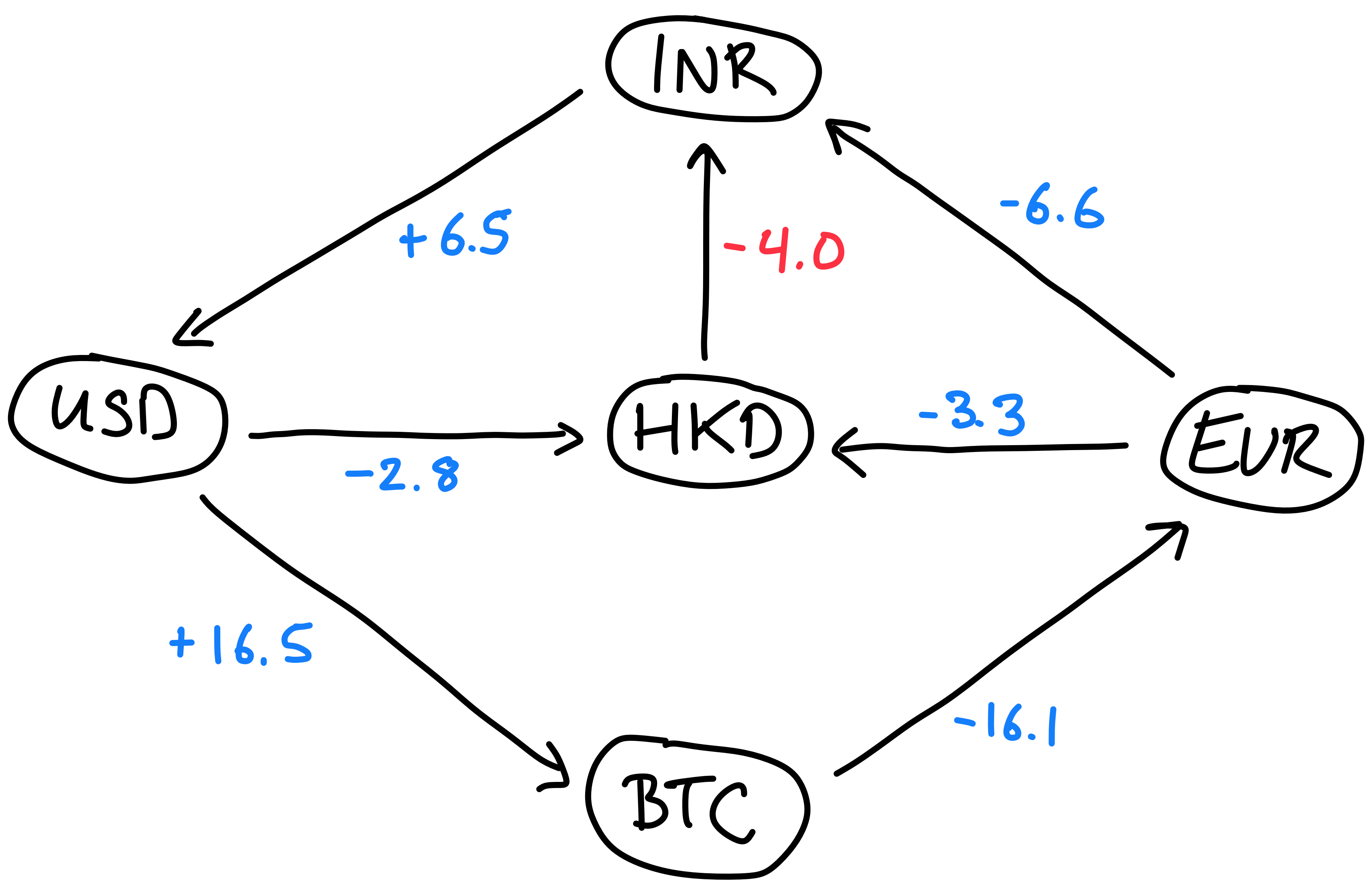
Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

- What happens if HKD to INR rate changes from

  INR = $2^{3.5} \approx 11.3$ HKD to

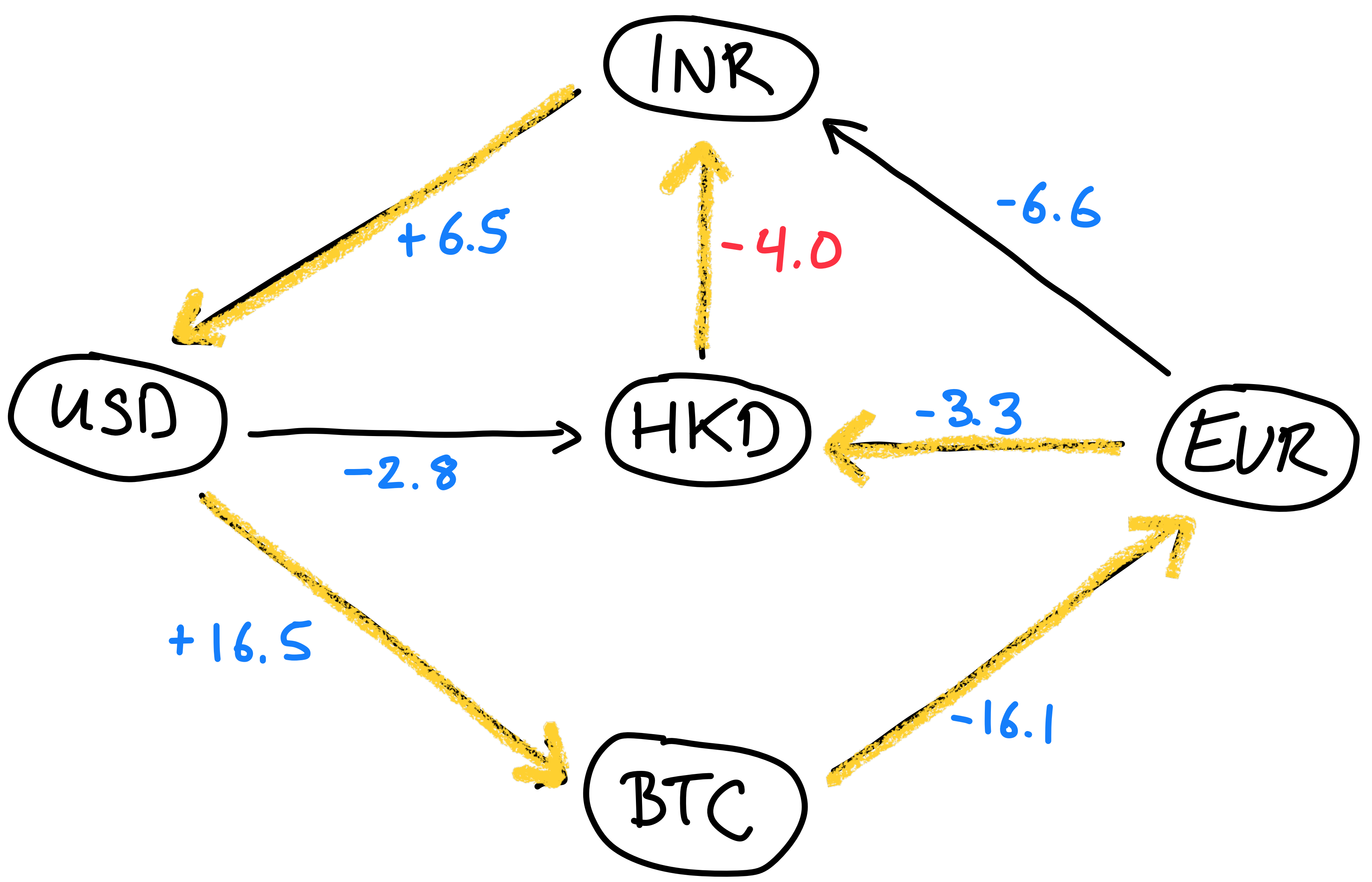
  INR = $2^{4.0} = 16$ HKD?



Currency exchange

Set edge weight to $\log_2(1/r) = -\log_2(r)$

- Consider the highlighted path from USD to USD:
- Converts 1 USD to $2^{0.8} > 1$ USD
- Constitutes a **negative cycle** in the graph
- In the currency exchange problem, negative cycles represent **arbitrage**
- Since there is a negative cycle, any currency can be converted into any other for arbitrarily cheap as the graph is strongly connected



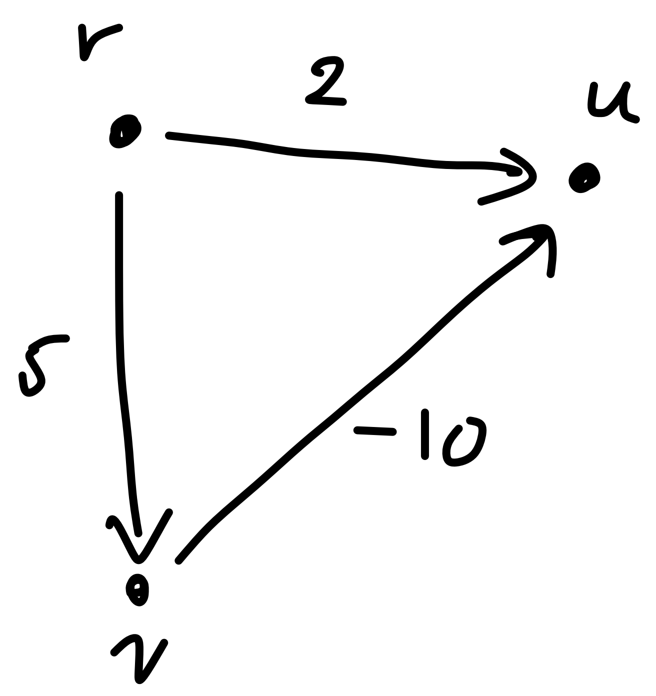
Negative weights shortest paths

- **Input:** A directed graph $G = (V, E)$ with weights $w : E \to \mathbb{R}$ and a vertex r
- **Output:** For every vertex v, the distance of the **lightest** directed path $r \rightsquigarrow v$ where a path's weight is the sum of its weights

- Why not just run Dijkstra's?
- Dijkstra's will incorrectly calculate distances when negative weights are involved

Negative weights shortest paths

- **Dijkstra's property**: Once a vertex v is visited, the distance $d(r, v)$ never needs updating again
  - This does not hold with negative weights
  - Need a slower but more careful algorithm that accounts for negative weights
- In this example,
  - Dijkstra's would set distance of u as 2 with path $r \to v$ in its first step
  - However, need to update the distance of u to -5 after v is visited.



Negative weights shortest paths
Applications

- Trade routes: each vertex is a commodity and edge $x \to y$ of weight w means 1 unit of x can be exchanged for 2^{-w} units of y
    - Multiplicative gains can be converted to linear gains by taking logarithms
    - Negative weights imply multiplicative losses
- Chemical networks: cost represent the excess energy required or **released** when a transformation is made
- Subsidies offered by governments for certain trades being performed
    - Example, US Govt. subsides flights from Portland, Oreg. to Pendleton, Oreg. to incentive airlines to fly to this market. (Annually, about $4 million for just this route)
    - How can an airline design its route network to maximize revenue in light of subsidies?

The Bellman-Ford algorithm

- Dijkstra's is a **greedy** algorithm and suffices to calculate shortest/lightest paths when all weights are non-negative
  - Distances will never need to be recalculated once set
- Bellman-Ford is a **dynamic programming** algorithm for computing shortest path in directed graphs
  - Will run slower than Dijkstra's: $O(mn)$ time versus $O(n + m)\log n)$ time
  - Will involve "resetting" distances as the algorithm goes along
  - Bellman-Ford will detect **negative cycles** as shortest paths are undefined if there are negative cycles

Failed attempt #1

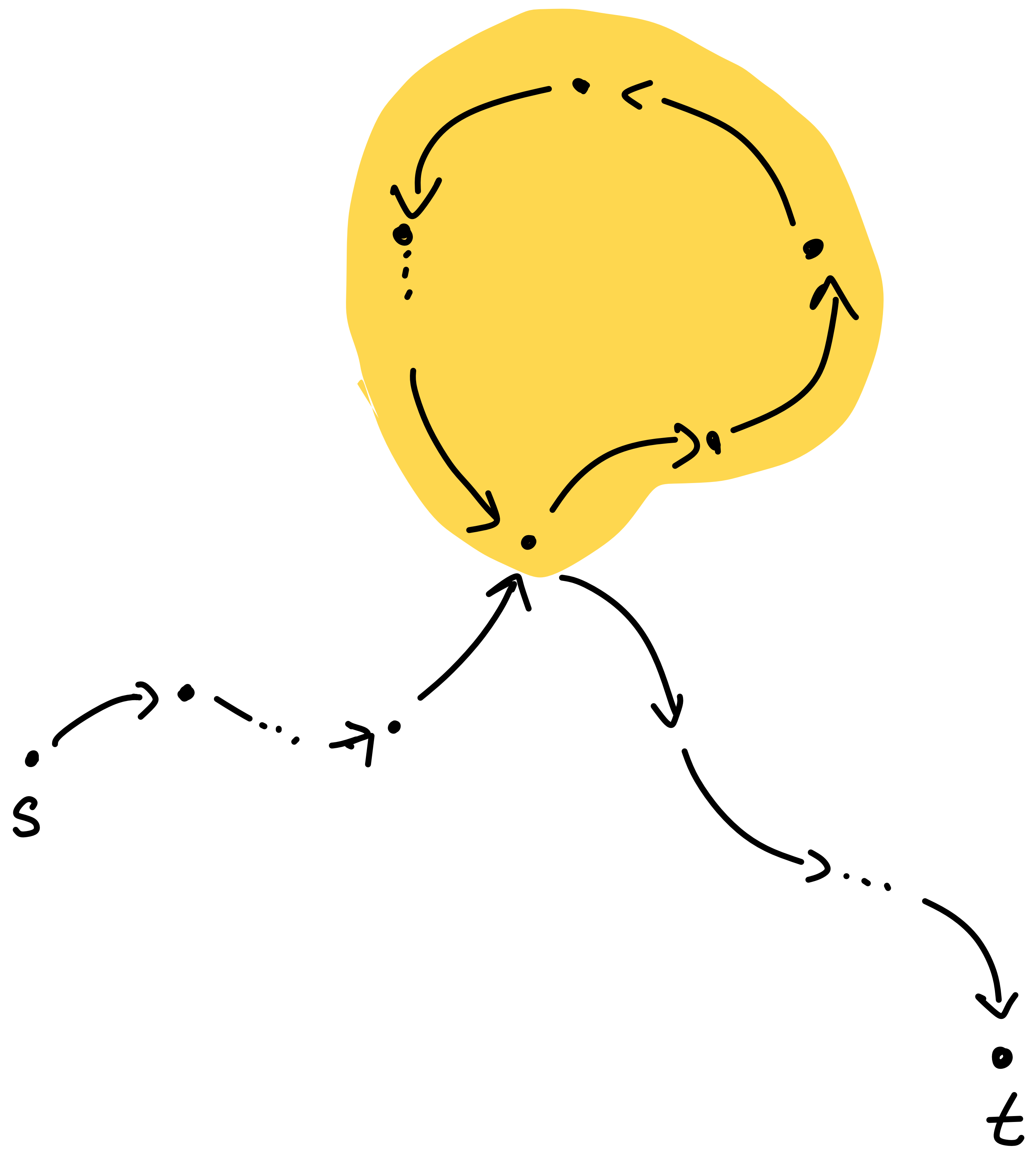
- If a graph has negative weights, let $w_{\min} = \min_{e \in E} w(e)$
- What if we adjusted every edge weight to $w'(e) = w(e) - w_{\min} \geq 0$?
- Can we just run standard Dijkstra's on the adjusted graph?
- **No**. Path weights adjust variably.
  - $w'(p) = w(p) - w_{\min} \cdot |\#$ of edges in $p|$
- Why can we run MST algorithms with negative weights?

Negative weight shortest path

- **Input:** Directed graph $G = (V, E)$ and weights $w : E \to \mathbb{R}$ and a vertex t
- **Output:** For all vertices s, the weight of the shortest path $d(s, t)$
- Note, we are considering shortest paths with respect to the endpoint t
- Its easy enough to convert it to an algorithm for shortest paths with respect to the source

Negative weight shortest path

- **Input:** Directed graph $G = (V, E)$ and weights $w : E \to \mathbb{R}$ and a vertex t
- **Output:** For all vertices s, the weight of the shortest path $d(s, t)$
- **Observation:** If a path $s \rightsquigarrow t$ contains a negative weight cycle, then a shortest path doesn't exist.
- **Observation:** If G has no negative cycles then the shortest path $s \rightsquigarrow t$ is of length $\leq n - 1$.
- **Proof:** A path of length $\geq n$ exists, it has a repeated vertex (i.e. a cycle). That cycle has weight ≥ 0, so removing it only decreases weight. Repeat till path is of length $\leq n - 1$.



Dynamic programming algorithm

- **Definition.** For $i \in \{0, \dots, n-1\}, s \in V$, let $d(i, s)$ be the length of the *shortest path* $s \rightsquigarrow t$ consisting of *at most* i edges
  - Case 1: The shortest path uses $\leq i - 1$ edges. Then

    $$d(i, s) = d(i-1, s)$$

  - Case 2: The shortest path uses exactly i edges. Let u be the first vertex on the path. Then

    $$d(i, s) = w(s, u) + d(i-1, u)$$

Dynamic programming algorithm

- **Definition.** For $i \in \{0, \ldots, n-1\}$, $s \in V$, let $d(i, s)$ be the length of the *shortest path* $s \rightsquigarrow t$ consisting of *at most* i edges
- **DP recursive definition**:

$$d(i,s) = \begin{cases} 0 & \text{if } i = 0 \text{ and } s = t \\ \infty & \text{if } i = 0 \text{ and } s \neq t \\ \min \left\{ d(i-1,s),\ \min_{u:s \to u} w(s,u) + d(i-1,u) \right\} & \text{otherwise} \end{cases}$$

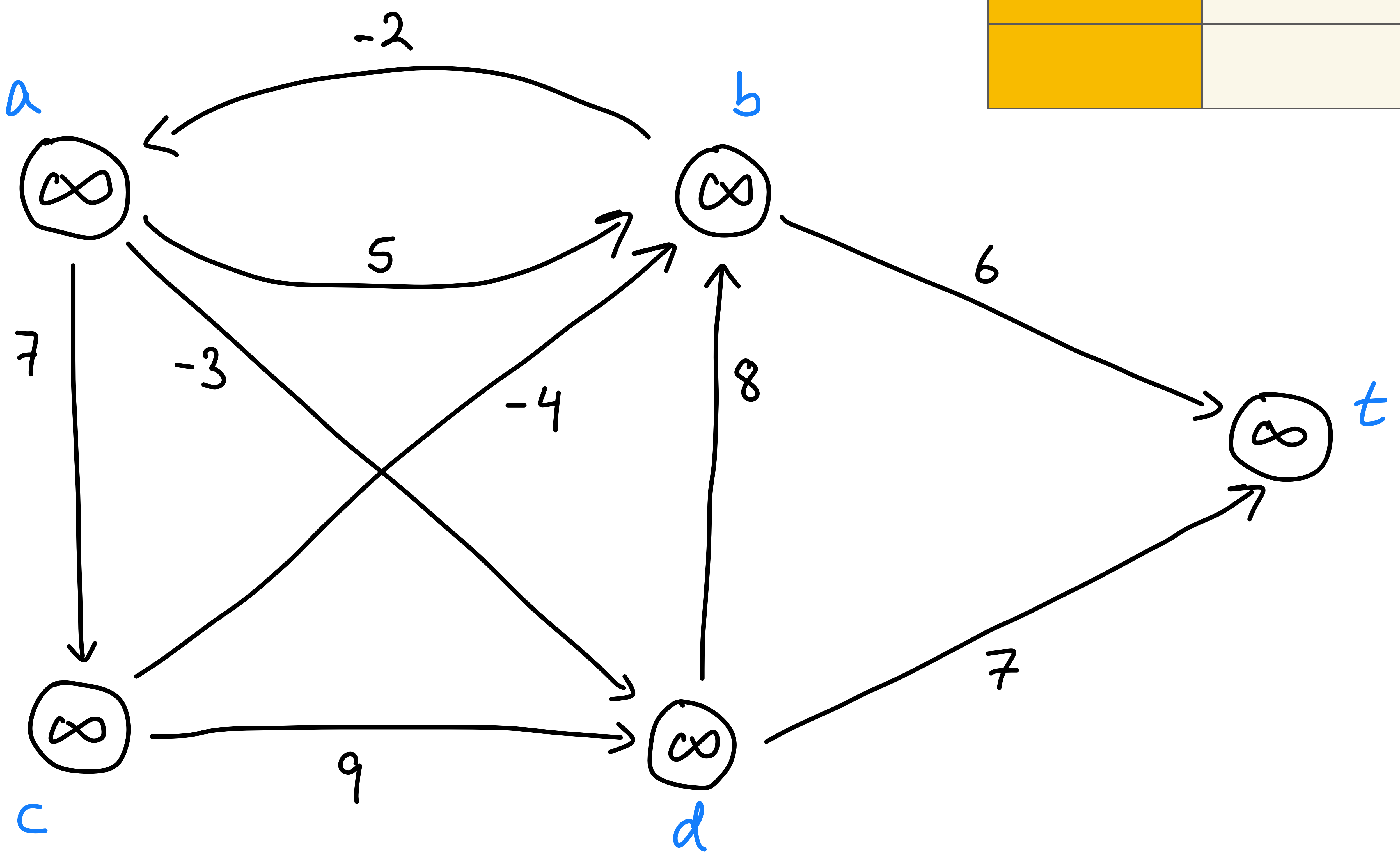
Dynamic programming implementation

(Assuming no negative cycles)

- **Table generation**:
  - Generate table d of size $(n-1) \times n$ and table next of size n
  - Set $d(0,s) \leftarrow \infty$ for $s \neq t$ and $d(0,t) \leftarrow 0$
  - For $i \leftarrow 1$ to n
    - Set $d(i,s) \leftarrow d(i-1,s)$.
    - For each edge $(s \rightarrow u) \in E$
      - If $w(s,u) + d(i-1,u) < d(i-1,s)$,
        - Set $d(i,s) \leftarrow w(s,u) + d(i-1,u)$ and $\mathrm{next}(s) \leftarrow u$
- **Path recovery**: Follow $\mathrm{next}(\,\cdot\,)$ from s until it reaches t.

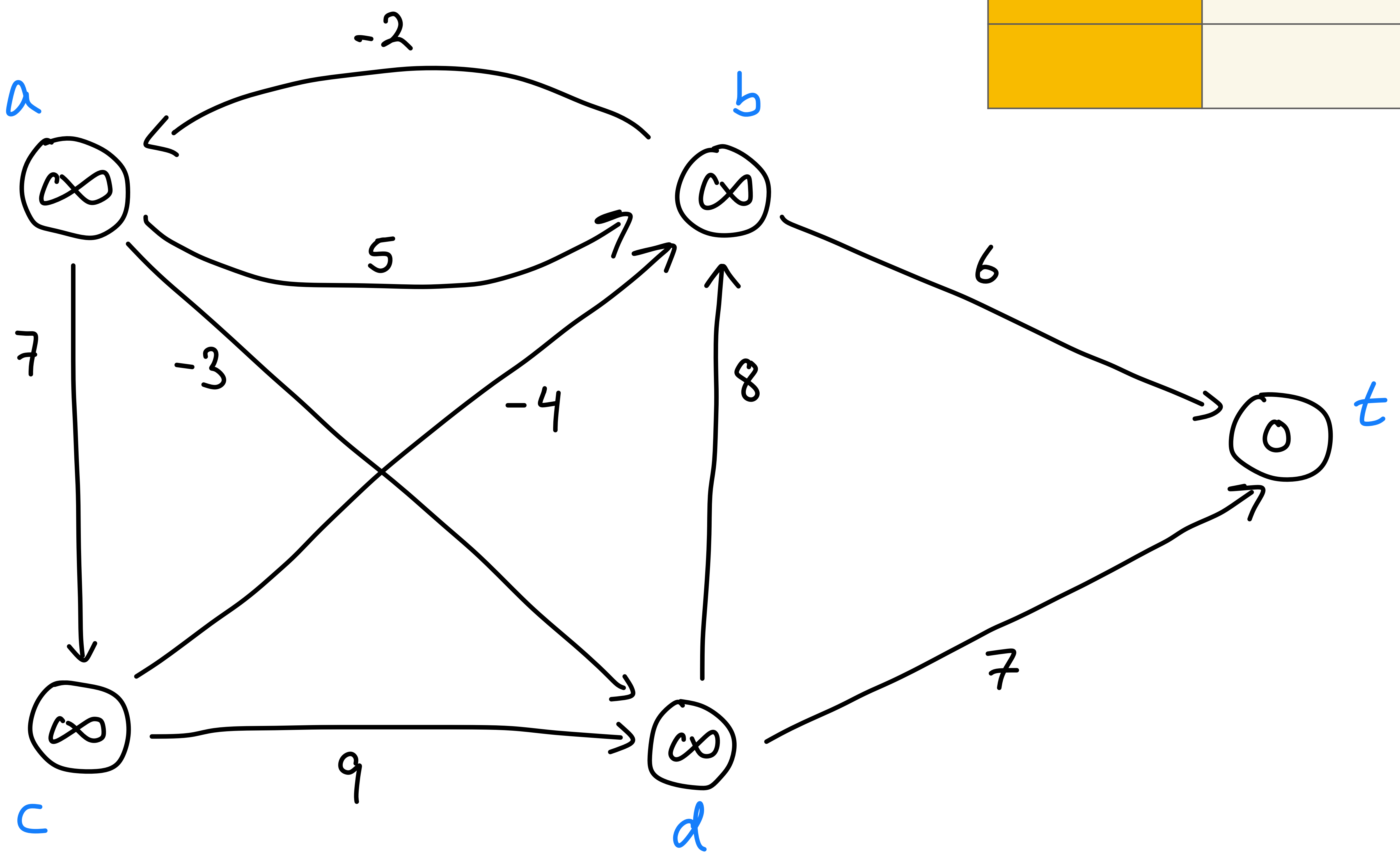
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0



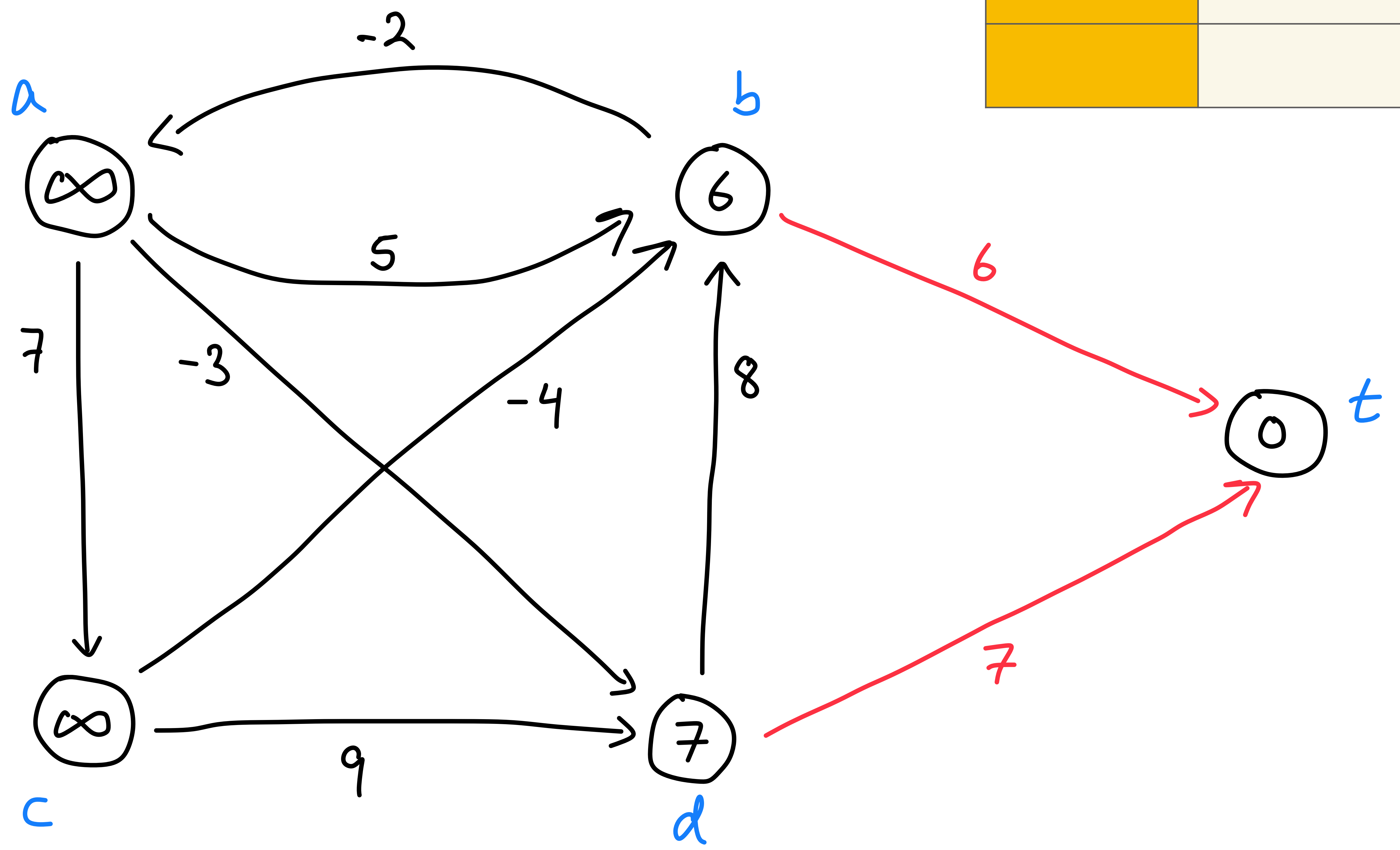
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	inf	inf	inf	0



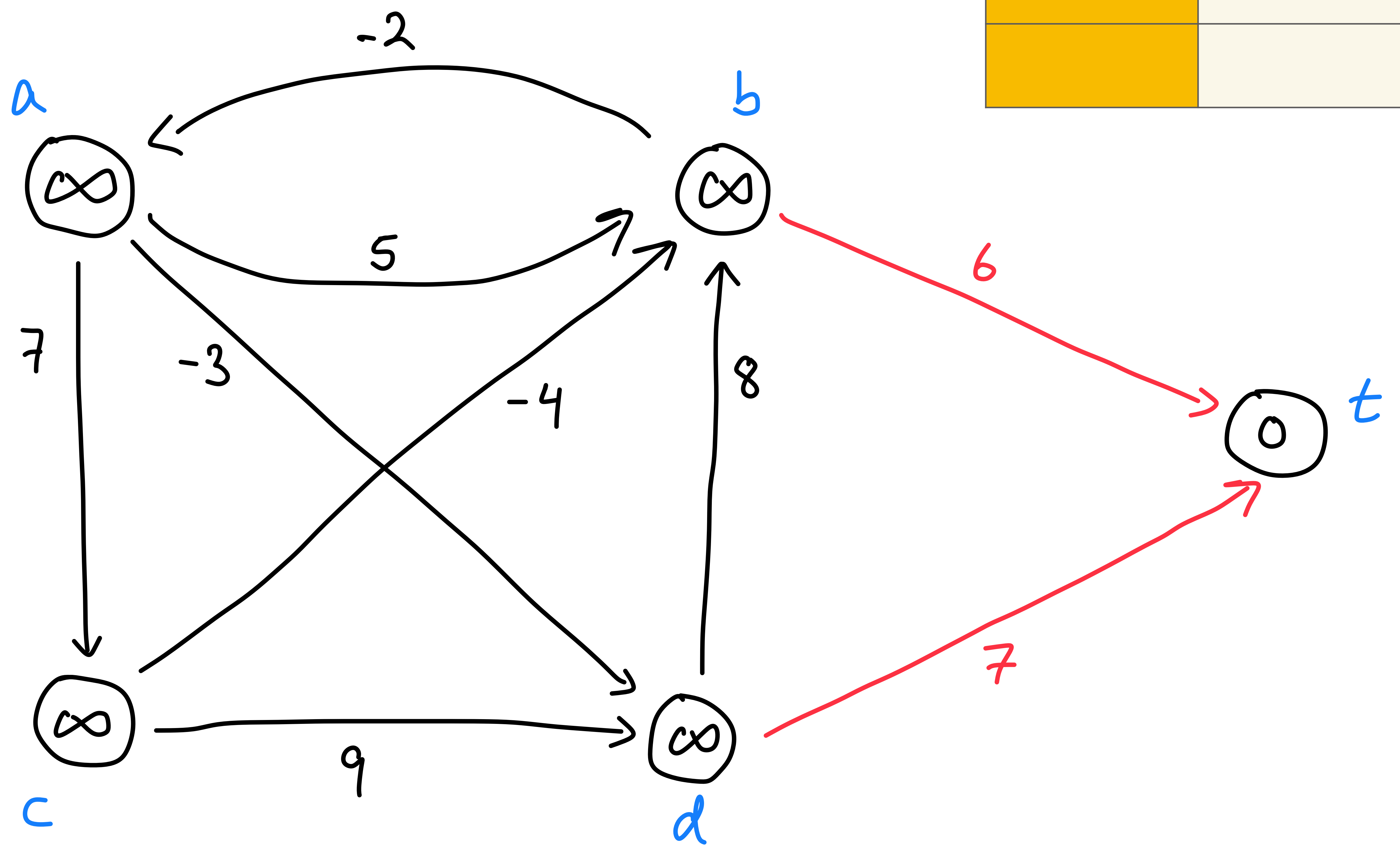
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0



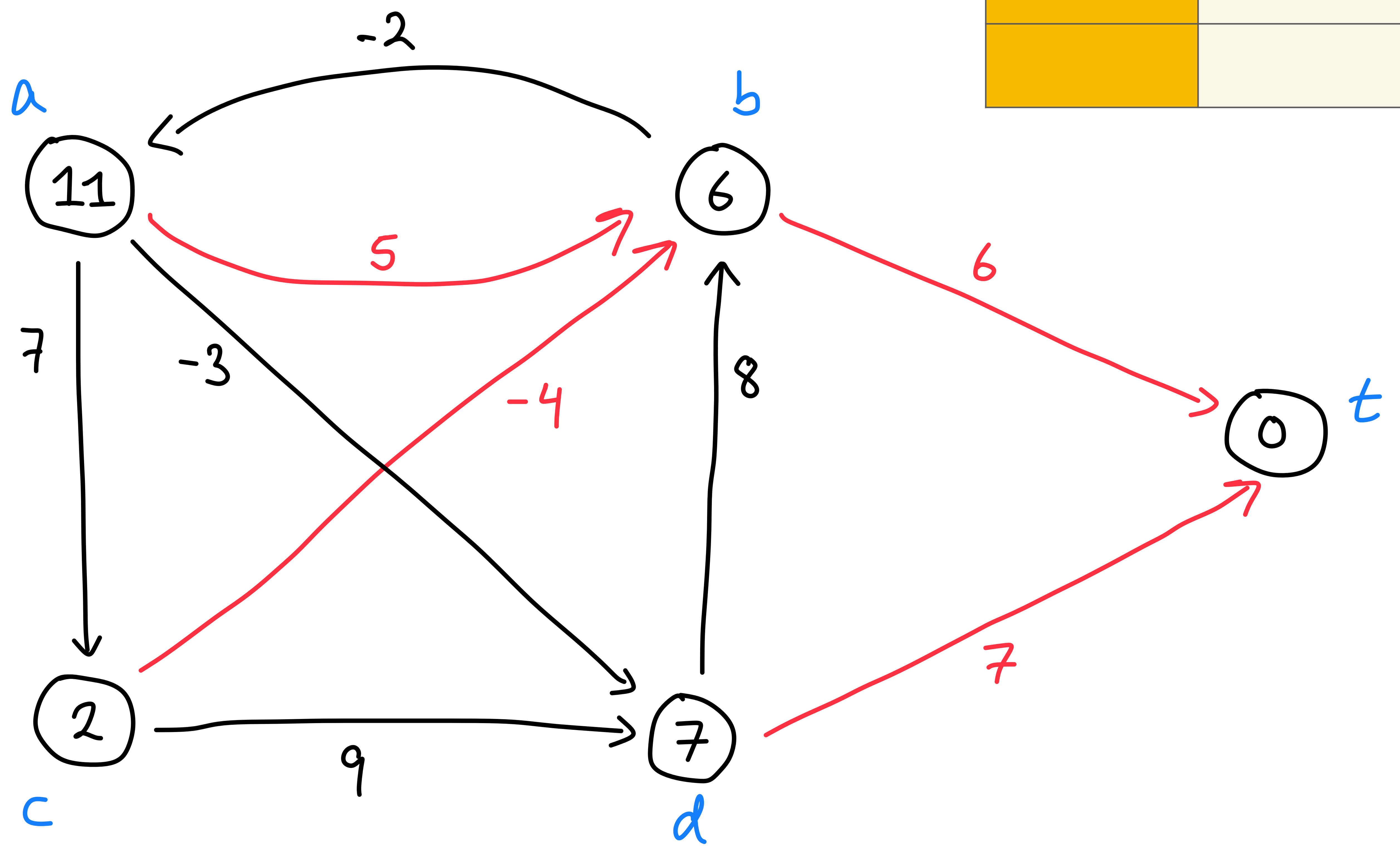
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	inf	6	inf	7	0



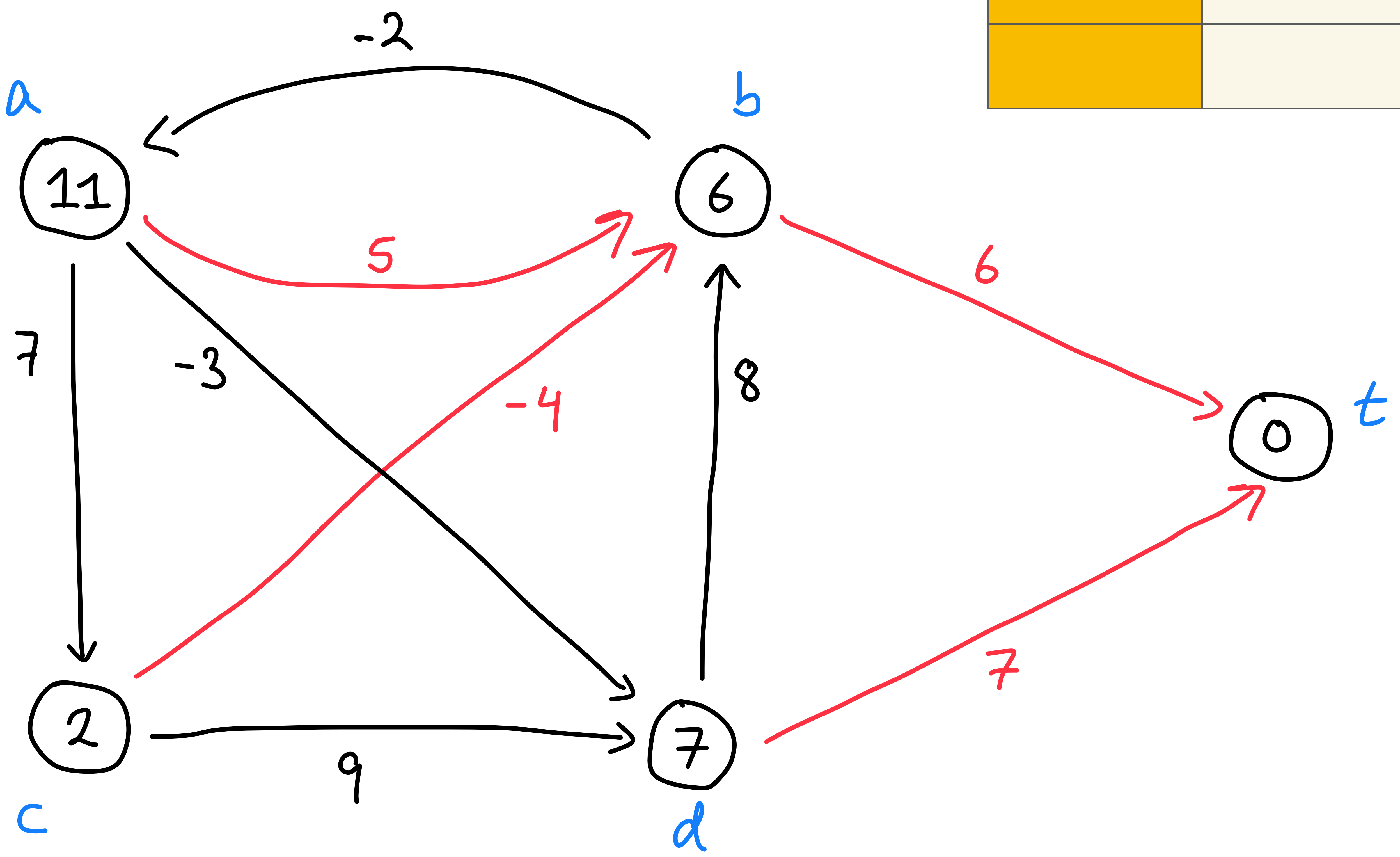
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0



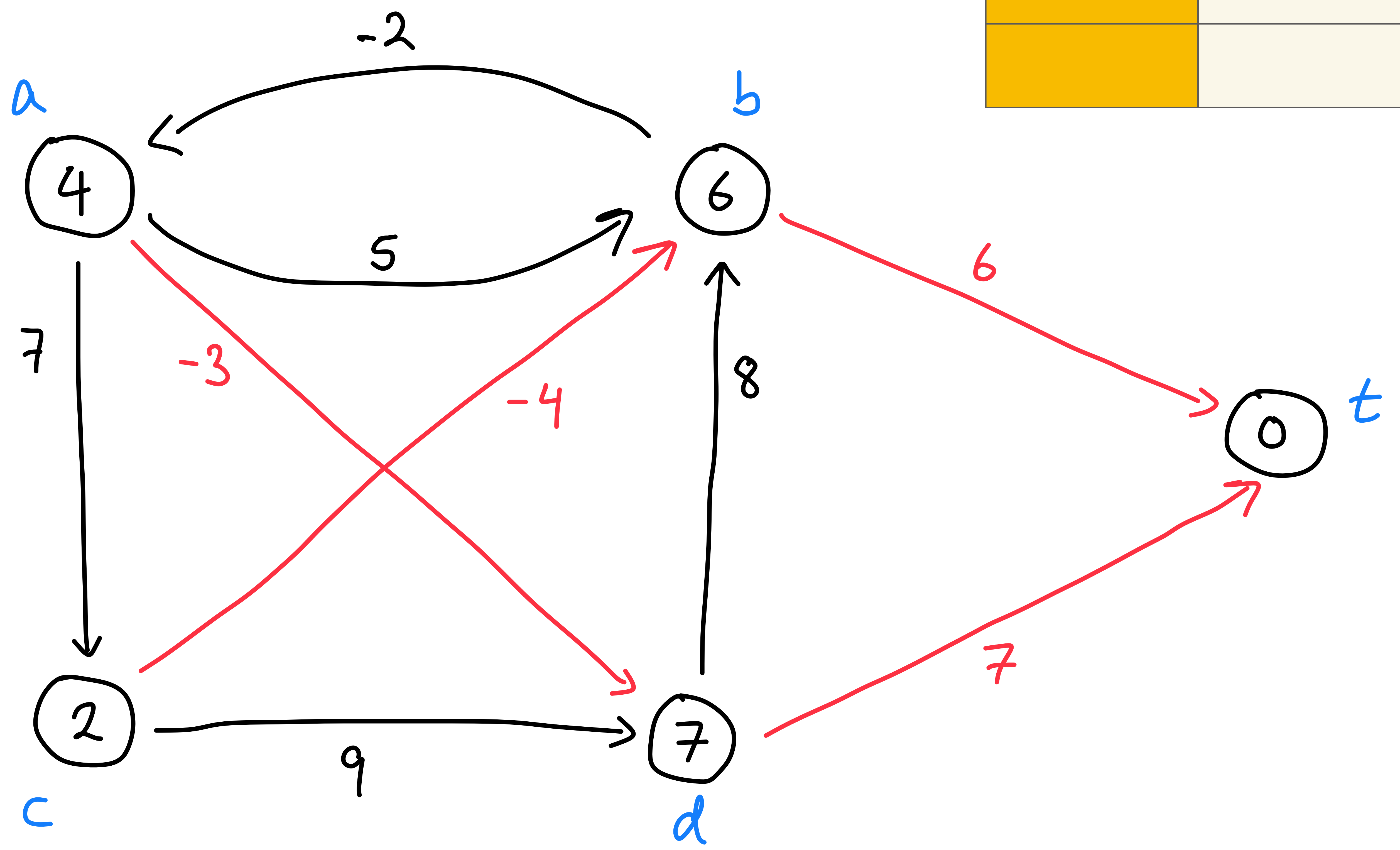
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	11	6	2	7	0



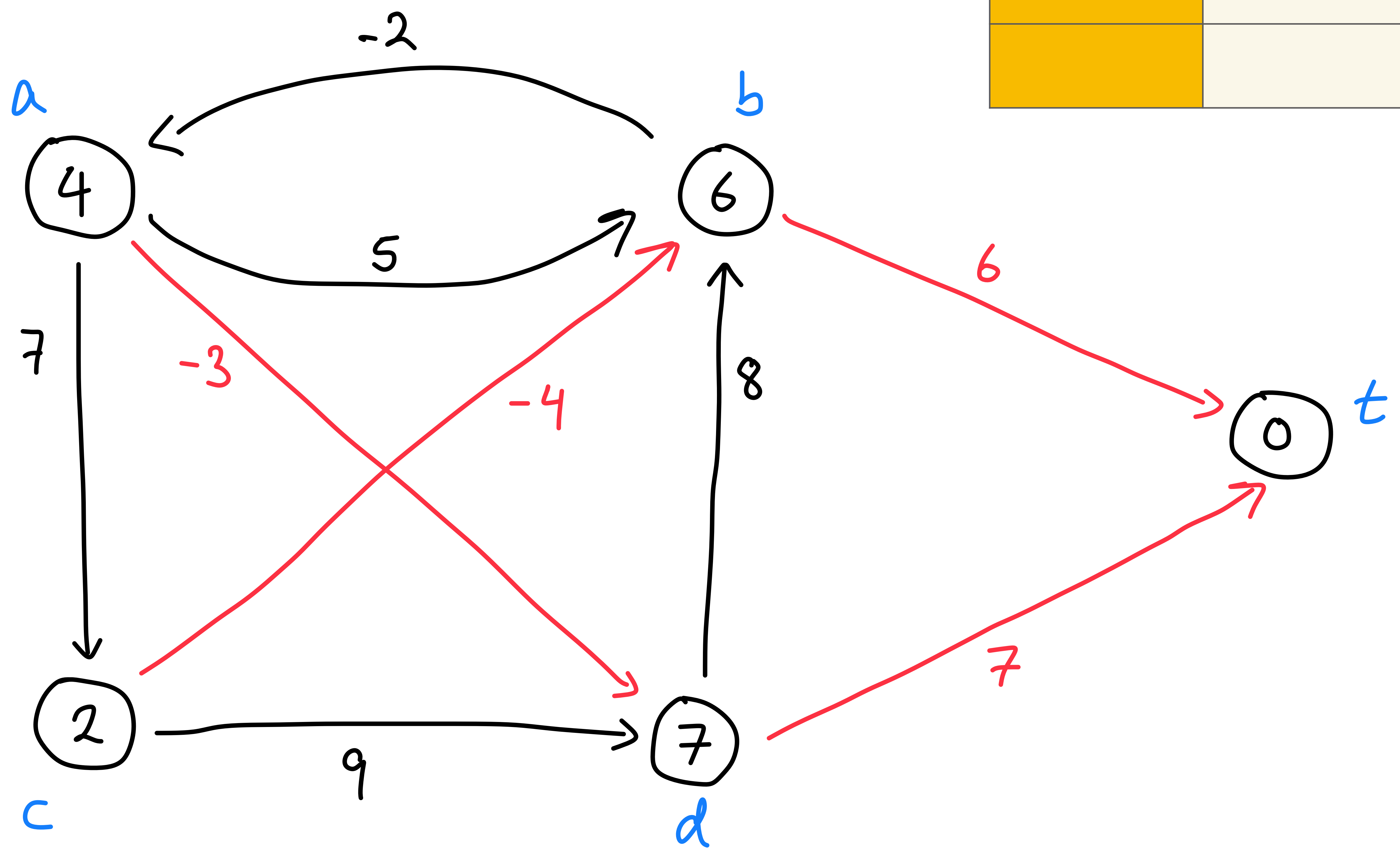
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0



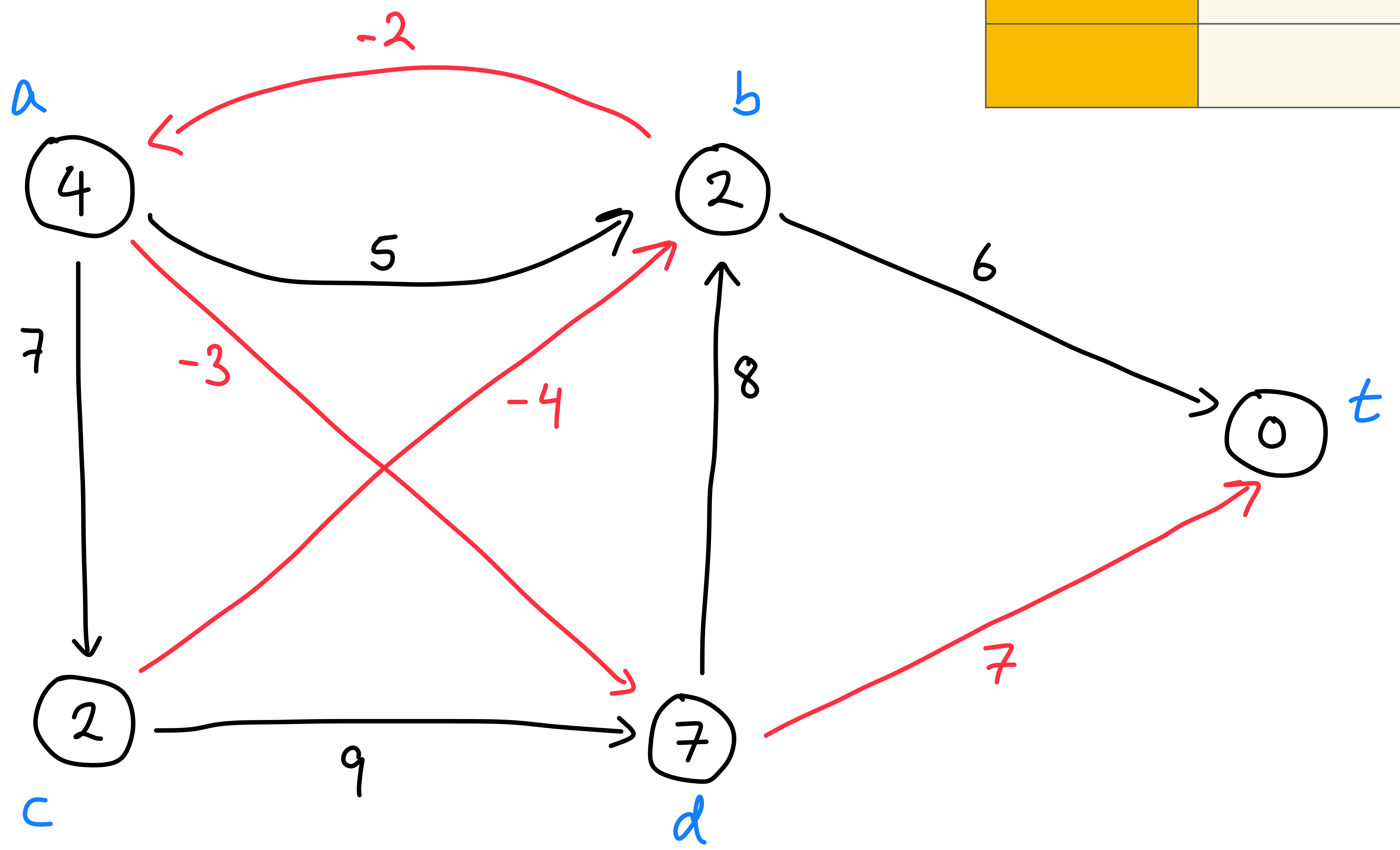
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0
4	4	6	2	7	0



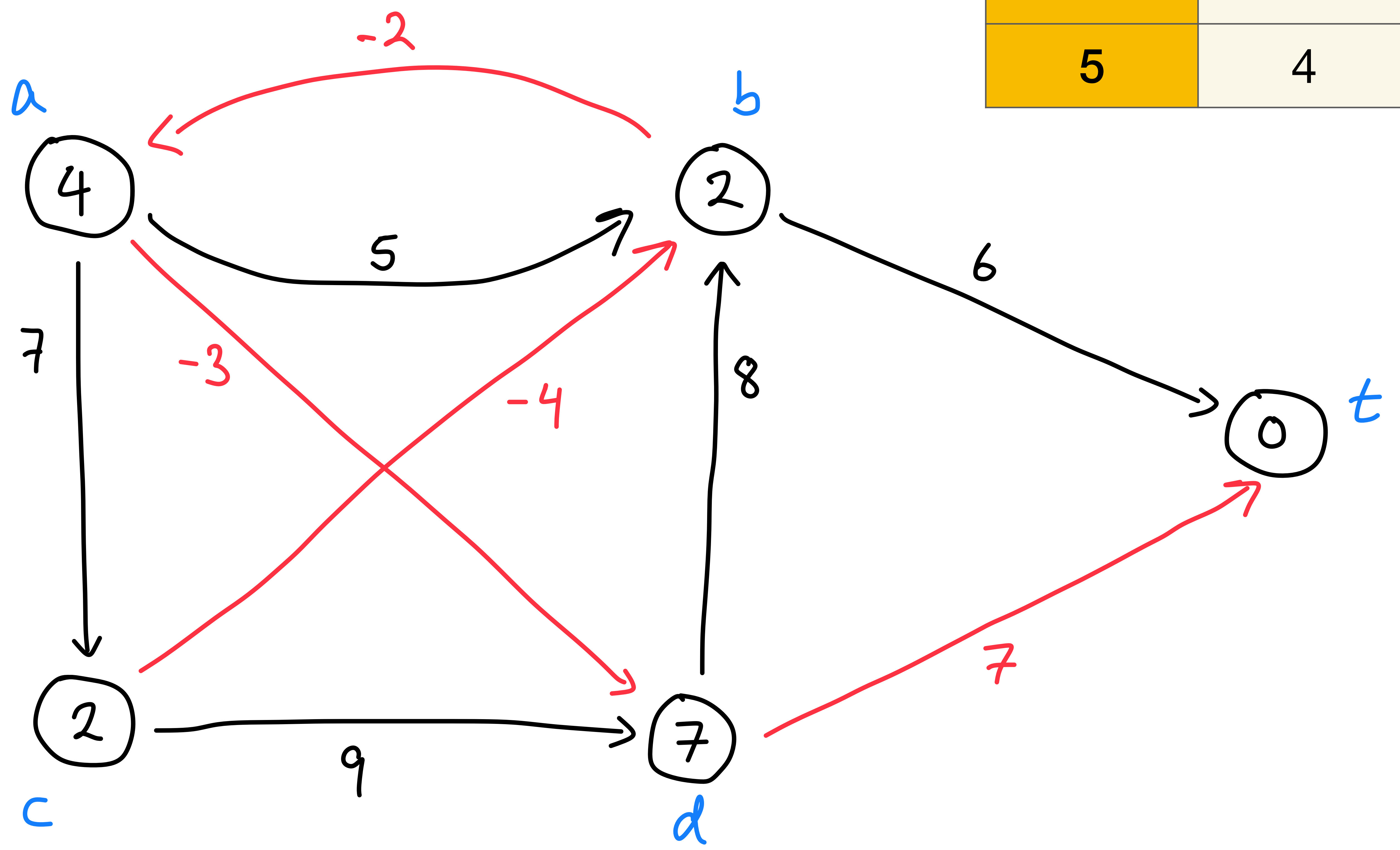
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0
4	4	2	2	7	0



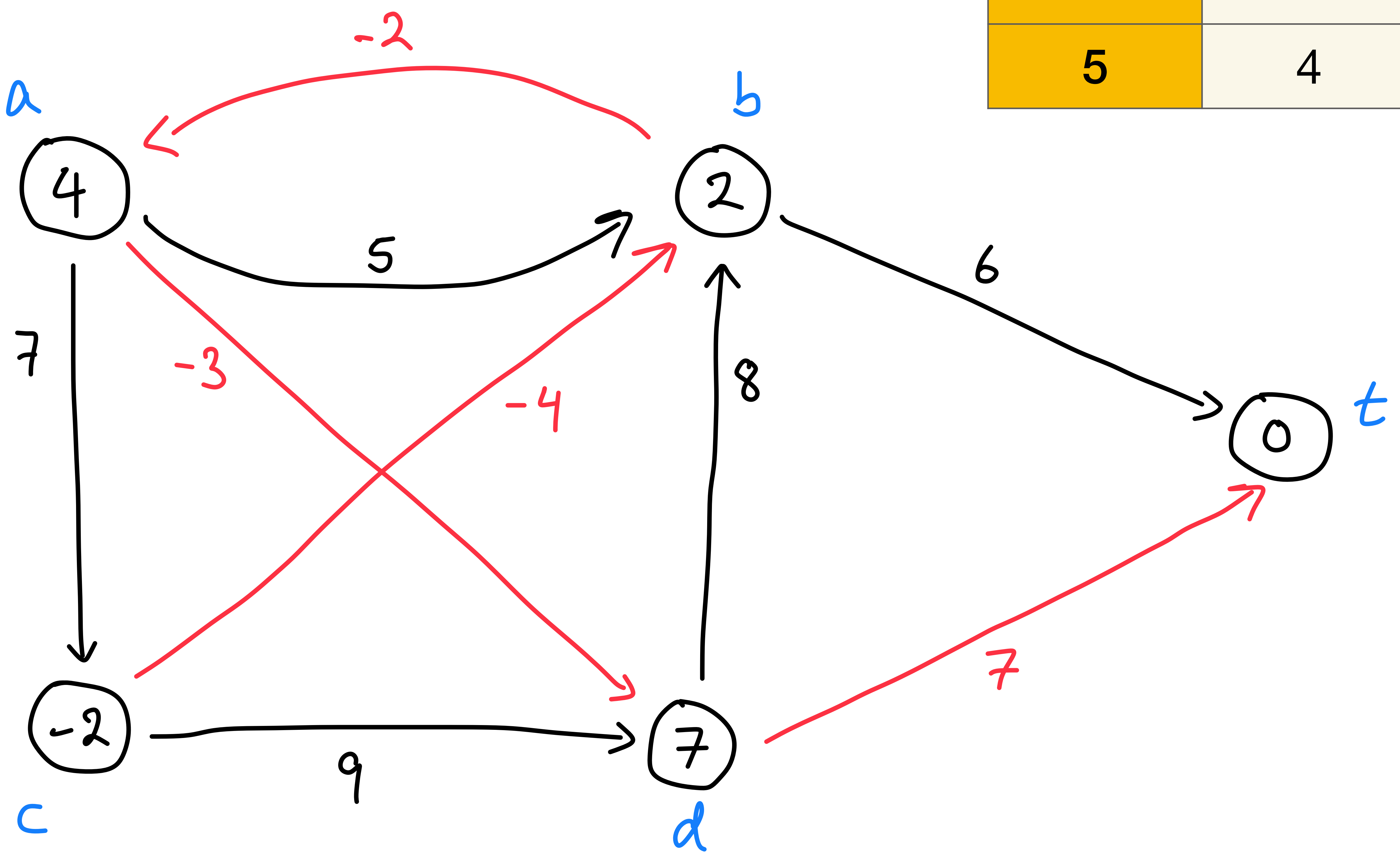
Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0
4	4	6	2	7	0
5	4	6	2	7	0



Bellman-Ford example

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0
4	4	6	2	7	0
5	4	-2	2	7	0



Space saving techniques

- The end result is a DAG mapping paths from every vertex s to the sink t
- The entries of $\text{next}(\cdot)$ list the edges in the path
- $d(i, s)$ only depends on entries $d(i-1, \cdot)$. Rows $i-2, \ldots, 1$ can be discarded.
- Computation should only keep track of the current and previous row.

	a	b	c	d	t
0	inf	inf	inf	inf	0
1	inf	6	inf	7	0
2	11	6	2	7	0
3	4	6	2	7	0
4	4	6	2	7	0
5	4	-2	2	7	0

Better “in-place” DP implementation

(Assuming no negative cycles)

- **Table generation:**
  - Generate table d of size n and table next of size n
  - Set $d(s) \leftarrow \infty$ for $s \neq t$ and $d(t) \leftarrow 0$
  - For $i \leftarrow 1$ to n and edge $(s \rightarrow u) \in E$
    - If $w(s, u) + d(u) < d(s)$,
      - Set $d(s) \leftarrow w(s, u) + d(u)$ and $\mathrm{next}(s) \leftarrow u$
- **Path recovery:** Follow $\mathrm{next}(\,\cdot\,)$ from s until it reaches t.

Even more trimming

- If $d(u)$ doesn't decrease in round i, then we don't need to consider any edges $s \to u$ in round $i+1$ as the best paths through u have already been considered
- Keep a list Q of vertices updated in the previous round and only update edge $s \to u$ if u was in Q

Even better DP implementation

(Assuming no negative cycles)

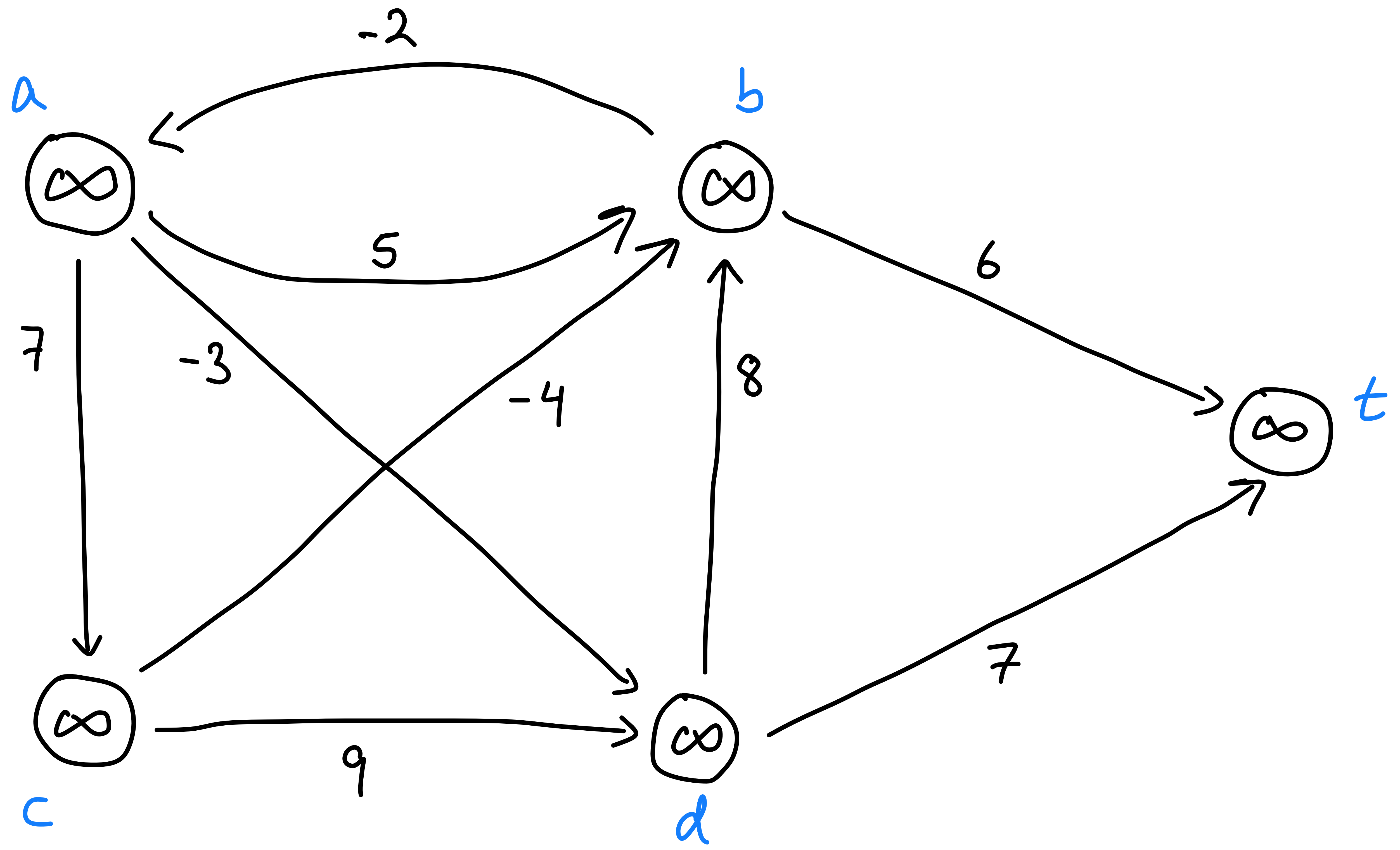
- Compute the reverse adjacency list: For every $u \in V$, $\text{pre}(u) = \{s : s \to u\}$.
- Generate tables d, next of size n with $d(s) \leftarrow \infty \ \forall \ s \neq t$ and $d(t) \leftarrow 0$
- Initialize counter $i \leftarrow 0$ and generate a queue $Q \leftarrow \{t, \perp\}$.
- While $i < n$
    - Pop u off the queue Q.
    - If $u = \perp$, increment $i \leftarrow i + 1$ and push $\perp$ to Q.
    - Else, for each $s \in \text{pre}(u)$,
        - If $w(s, u) + d(u) < d(s)$, set $d(s) \leftarrow w(s, u) + d(u)$ and $\text{next}(s) \leftarrow u$
        - Push s into queue Q.

everytime $\perp$ is seen in queue, we've done one iteration of BF. We need to do n-1.

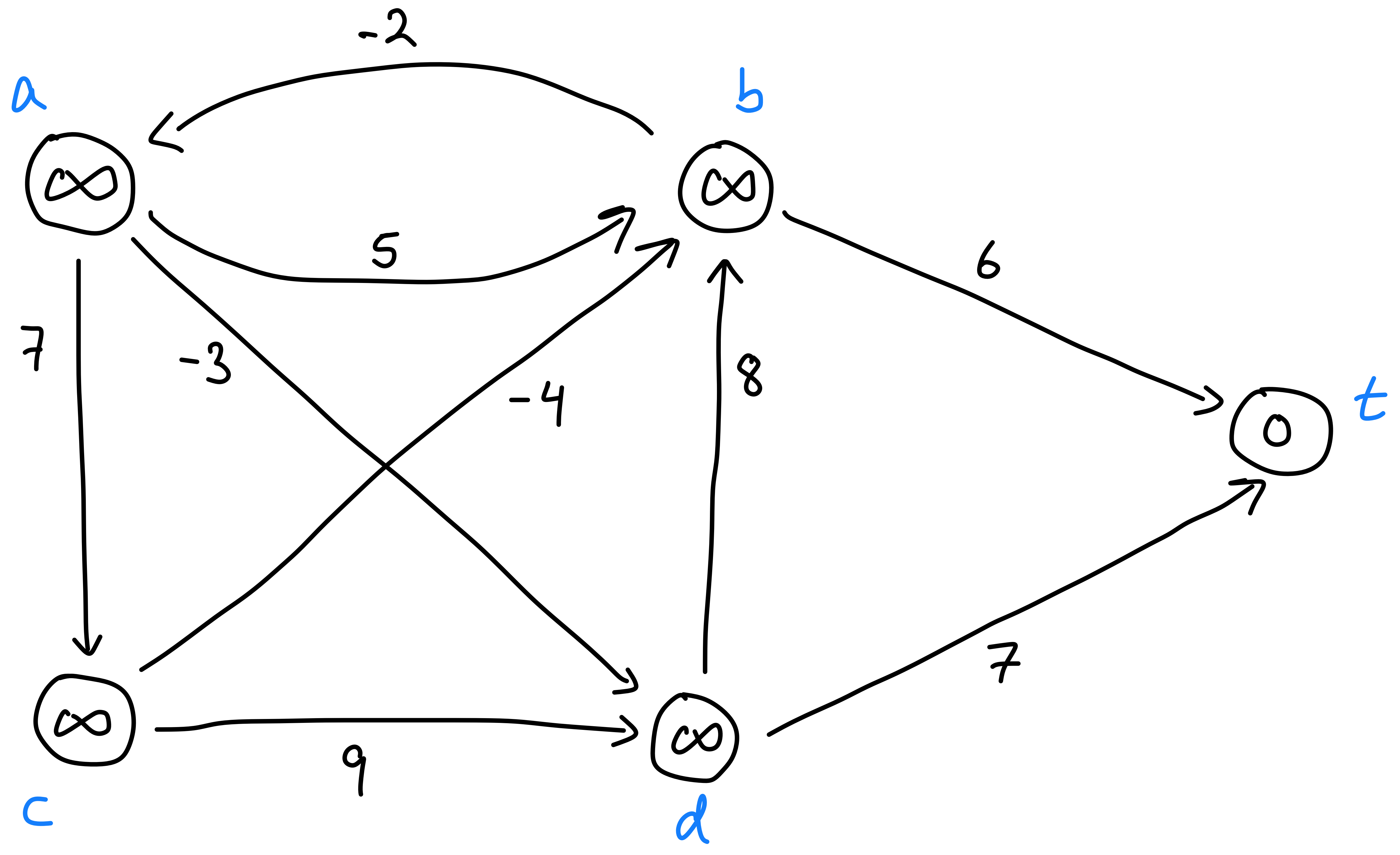
Bellman-Ford properties

- **Theorem**: Throughout the algorithm, $d(s)$ is the length of some path and that path has weight less than the lightest path of $\leq i$ edges after i rounds of updates
- **Impact:** Space decreases to $O(n+m)$ but runtime is still $O(nm)$ in the worst case. In practice, the runtime is much faster!

Bellman-Ford example



Bellman-Ford example

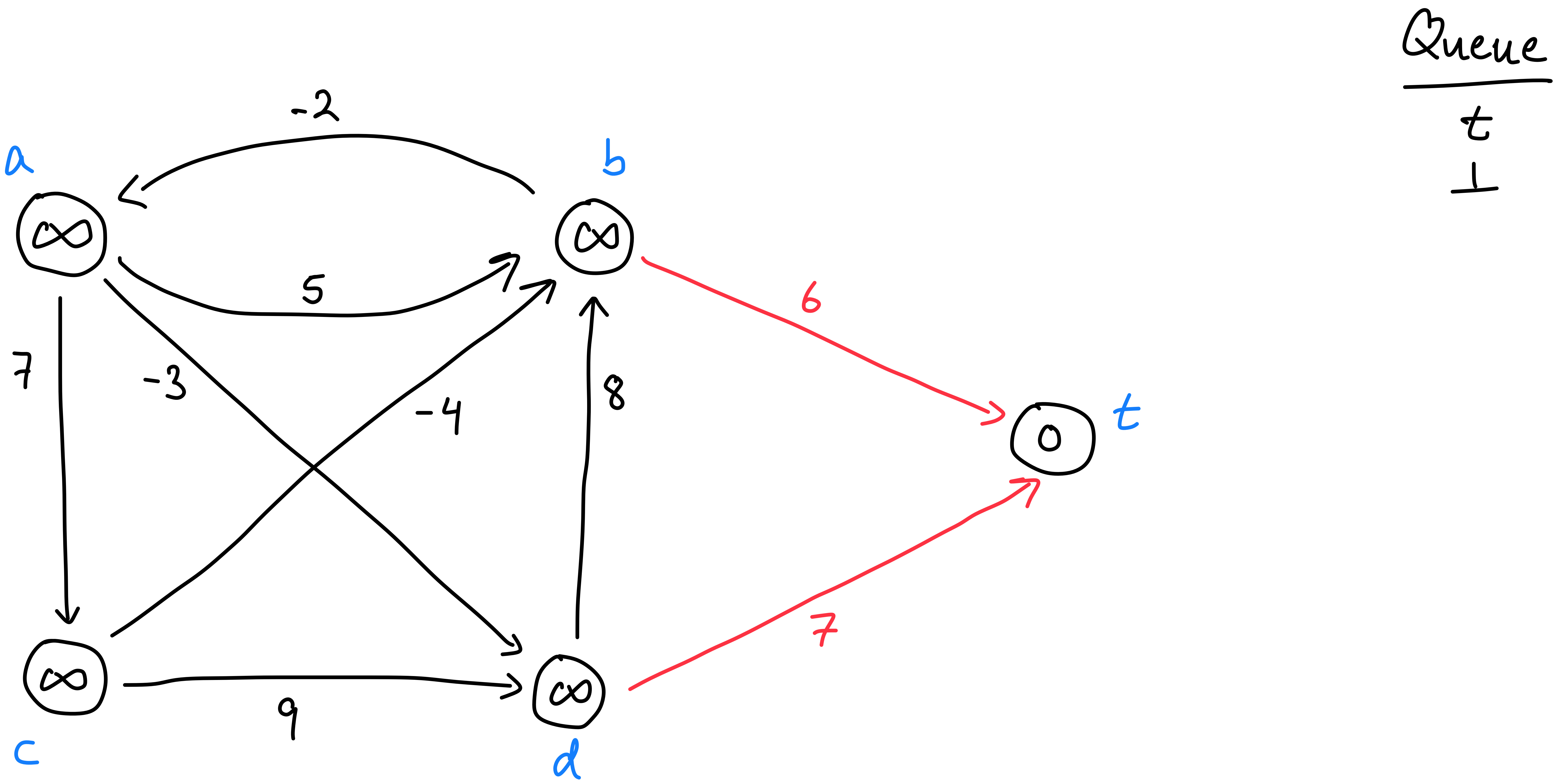


Queue

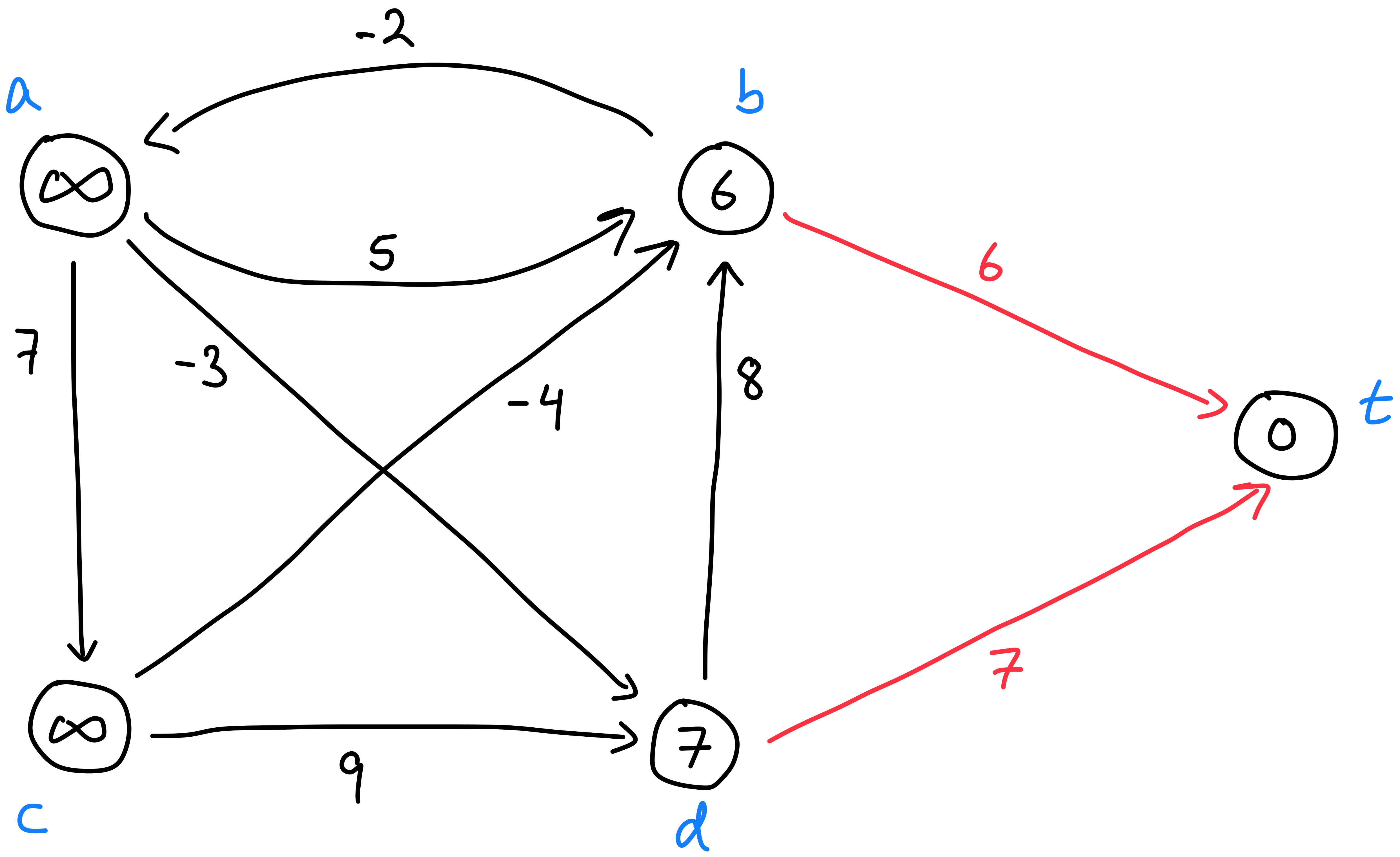
t

⊥

Bellman-Ford example



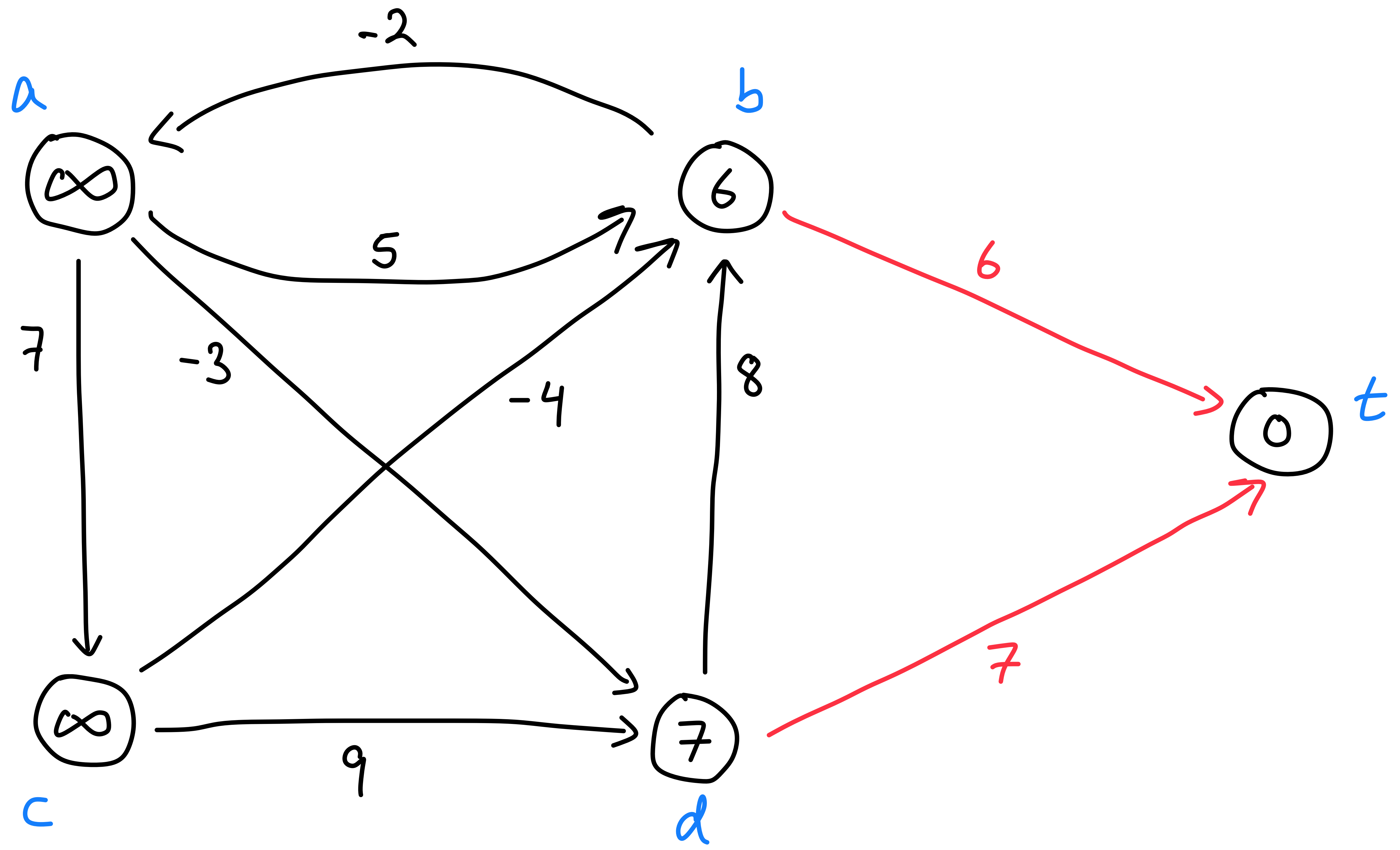
Bellman-Ford example

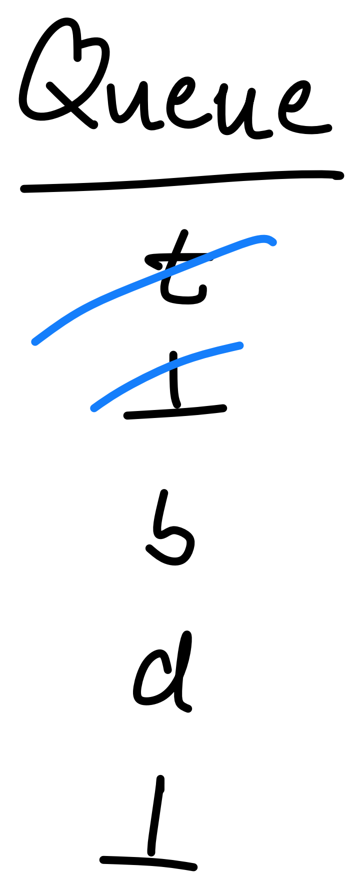


Queue

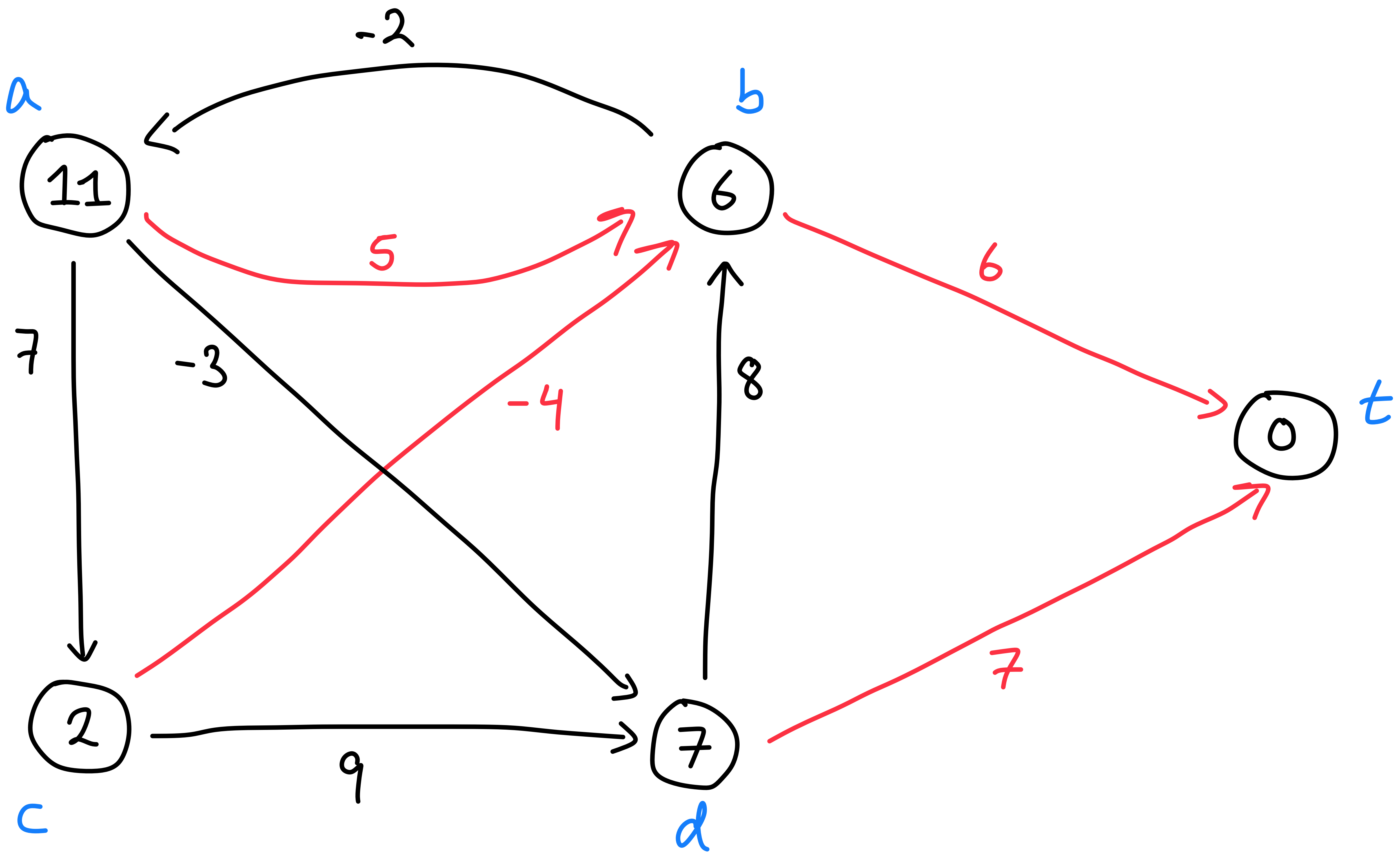
~~t~~
⊥
b
d

Bellman-Ford example





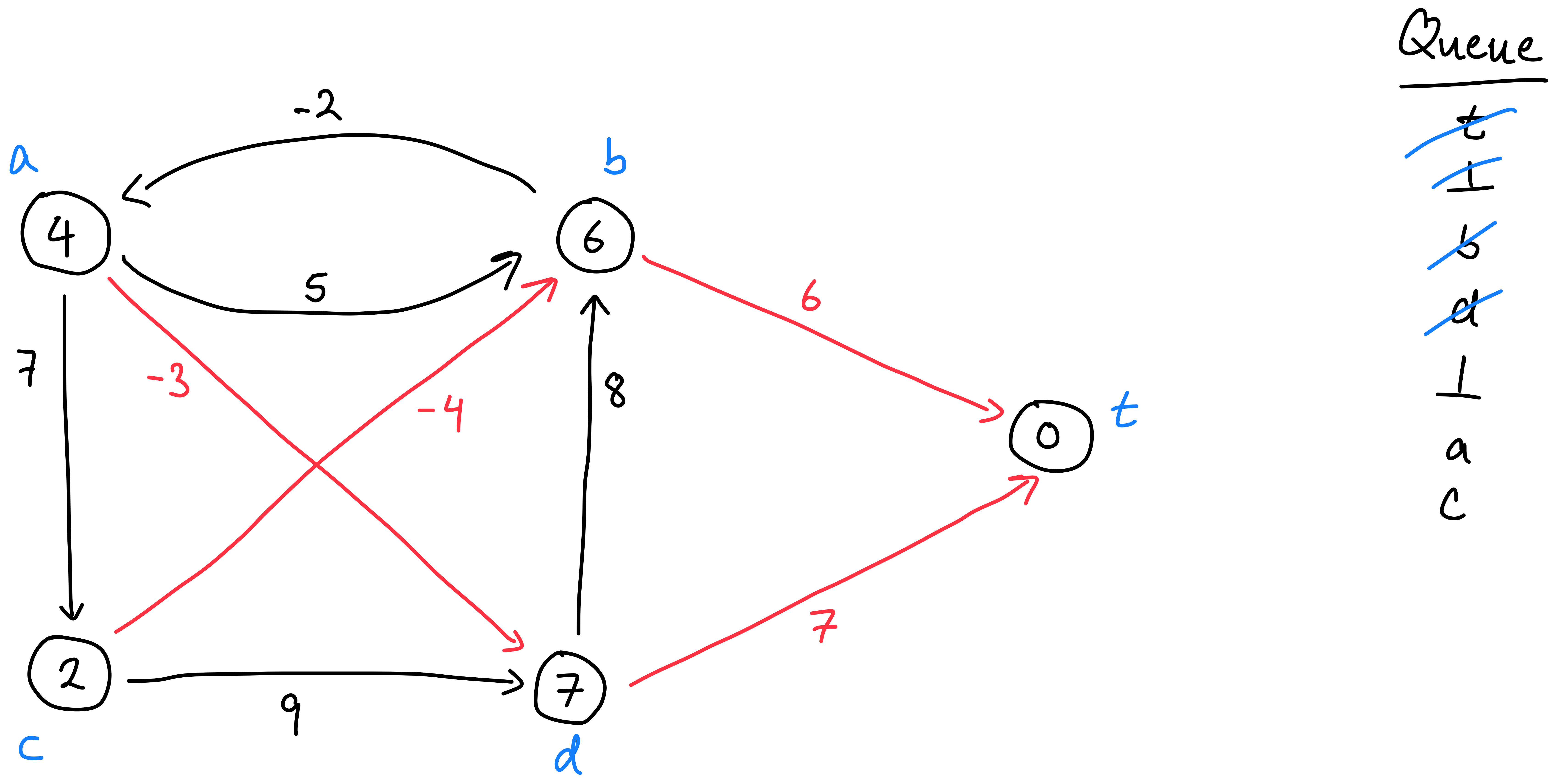
Bellman-Ford example



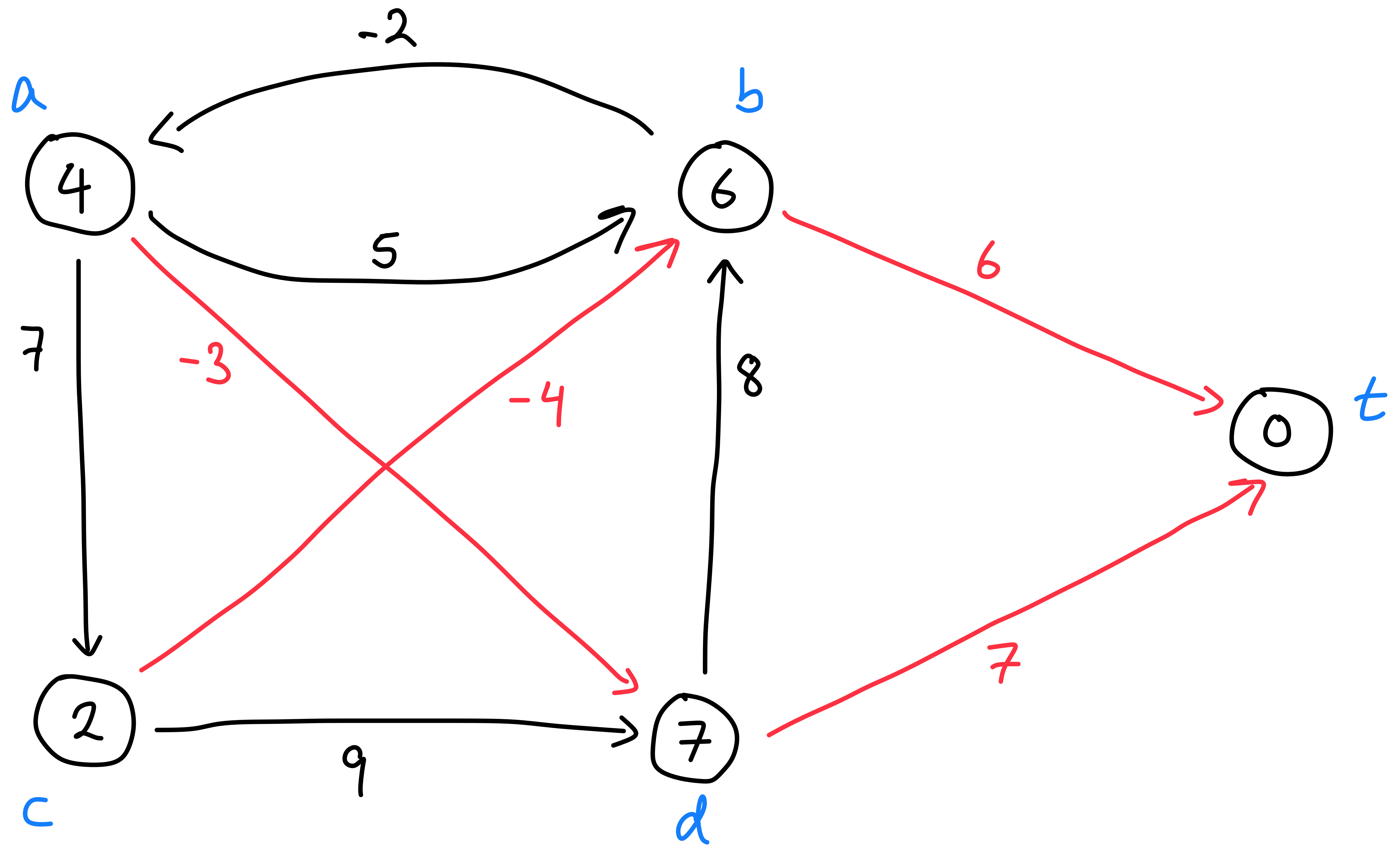
Queue

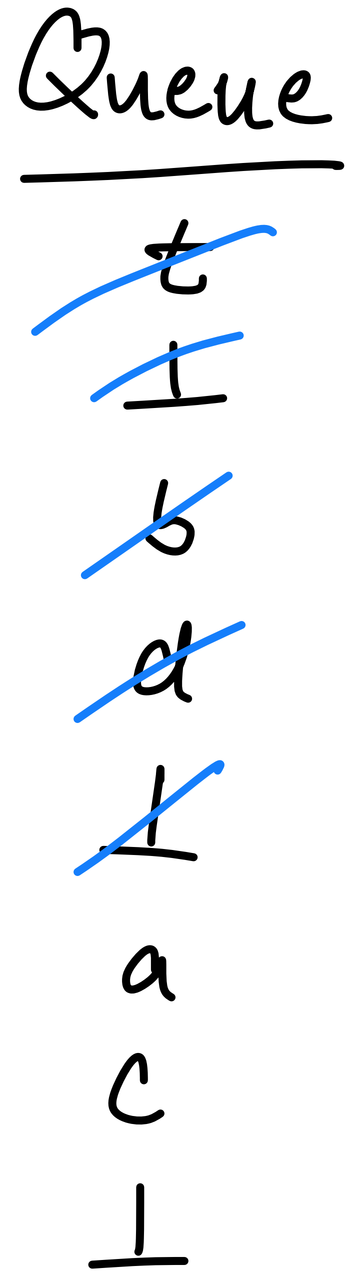
~~t~~
~~⊥~~
~~b~~
d
⊥
a
c

Bellman-Ford example

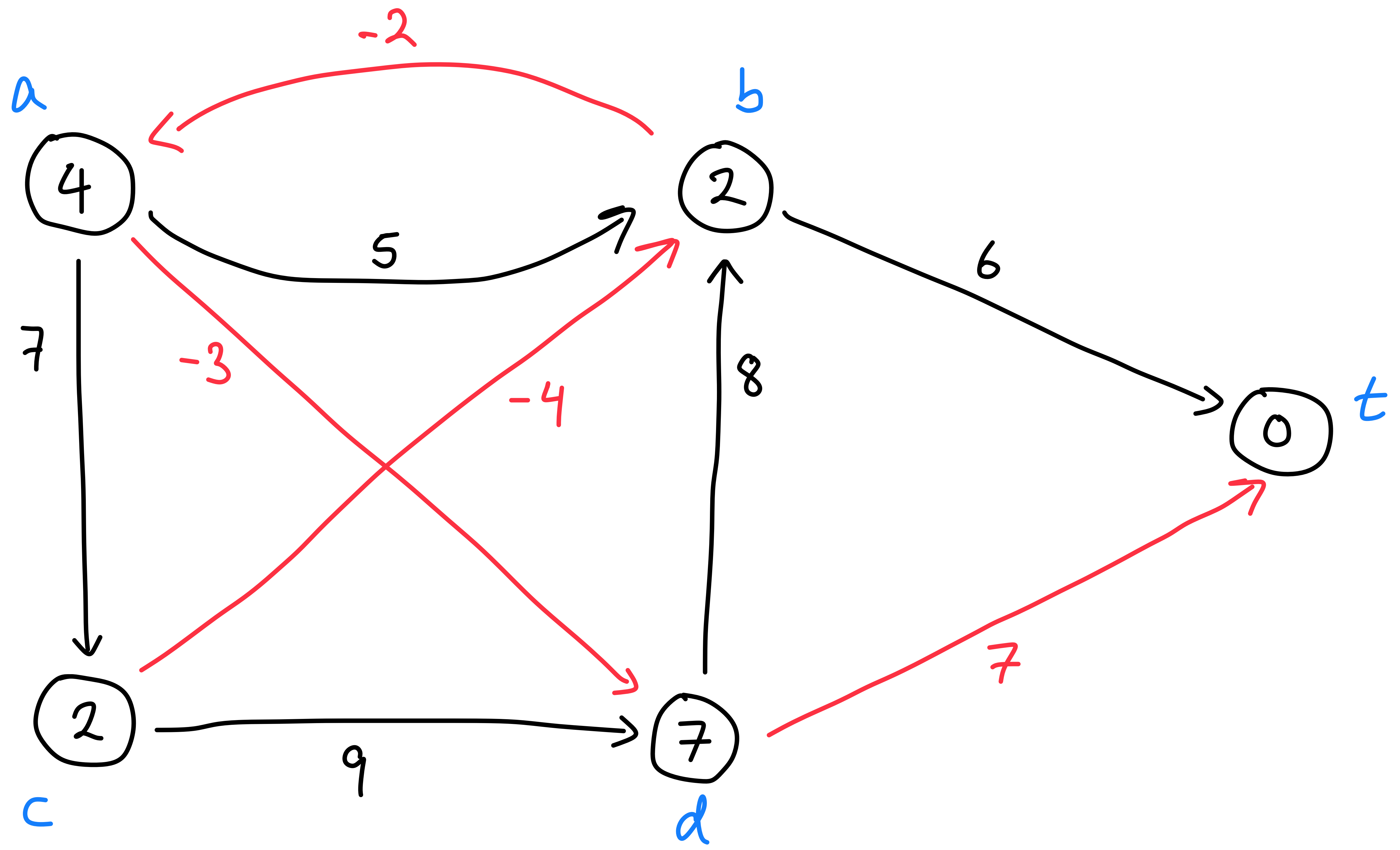


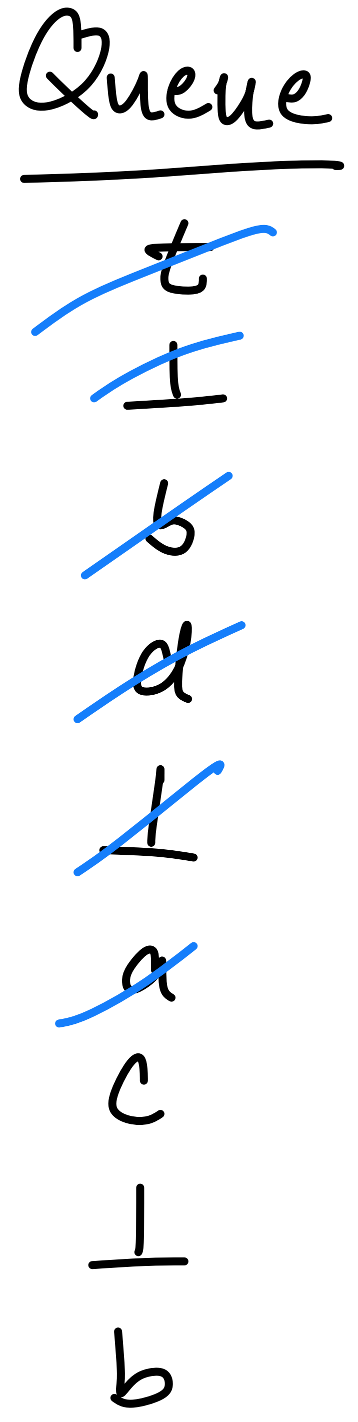
Bellman-Ford example



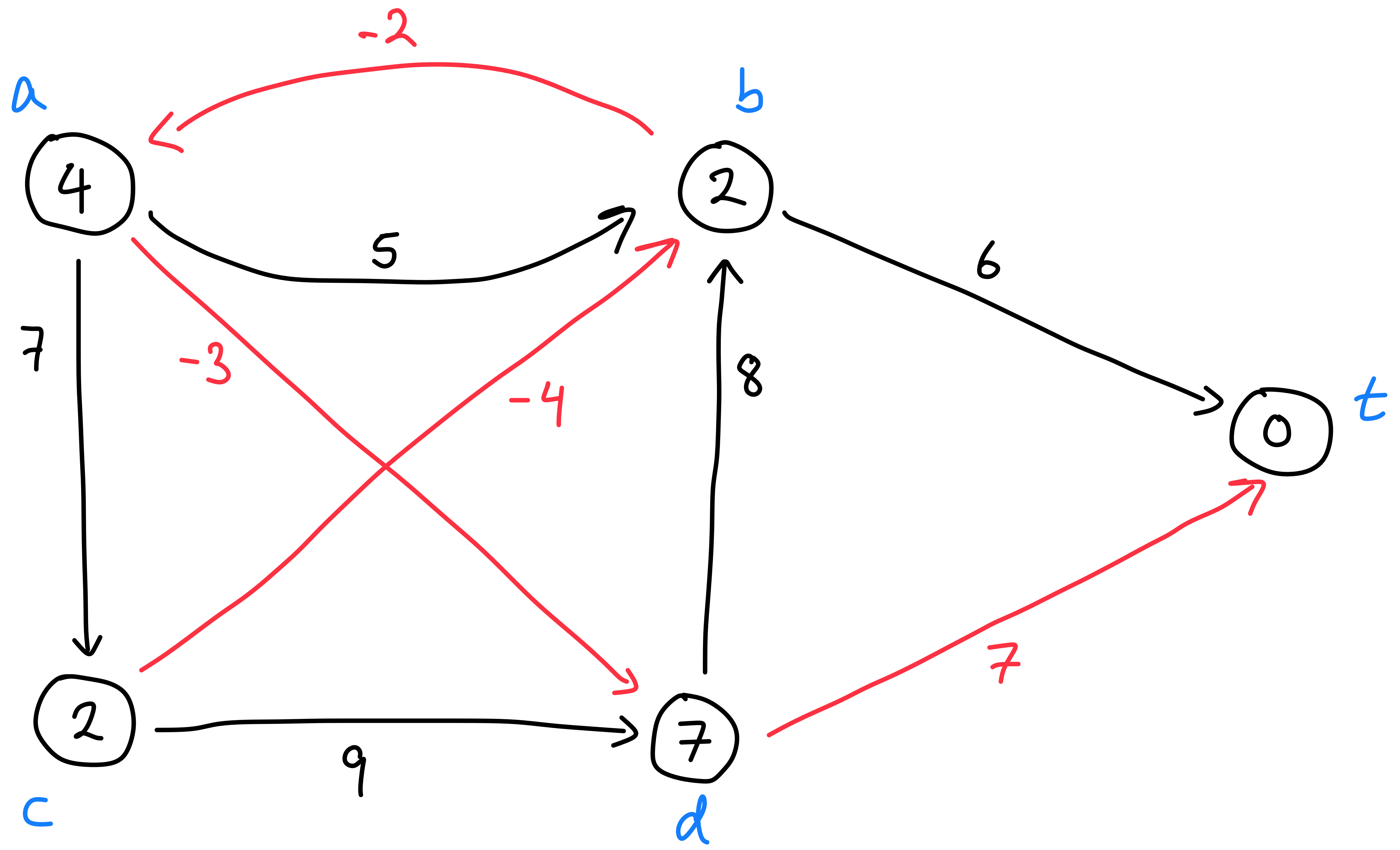


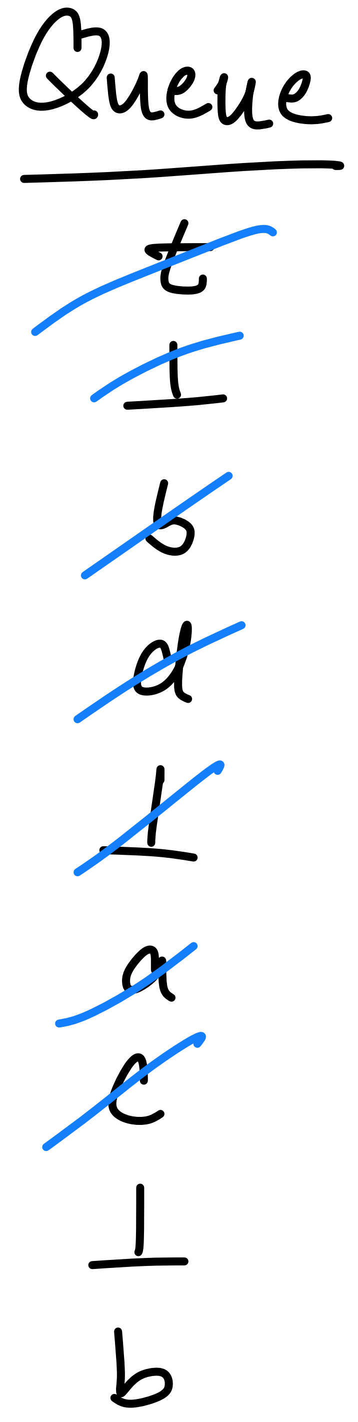
Bellman-Ford example



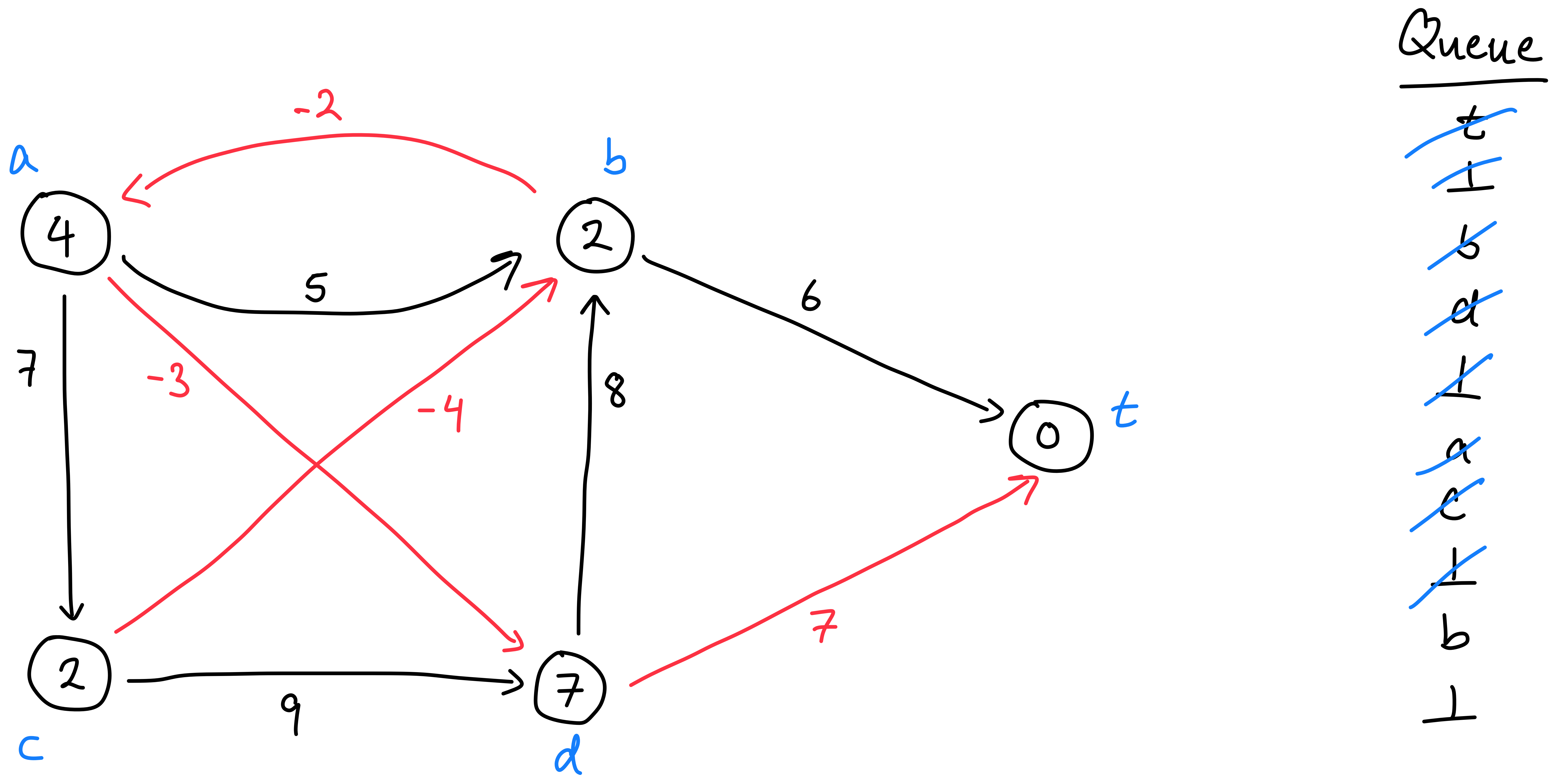


Bellman-Ford example

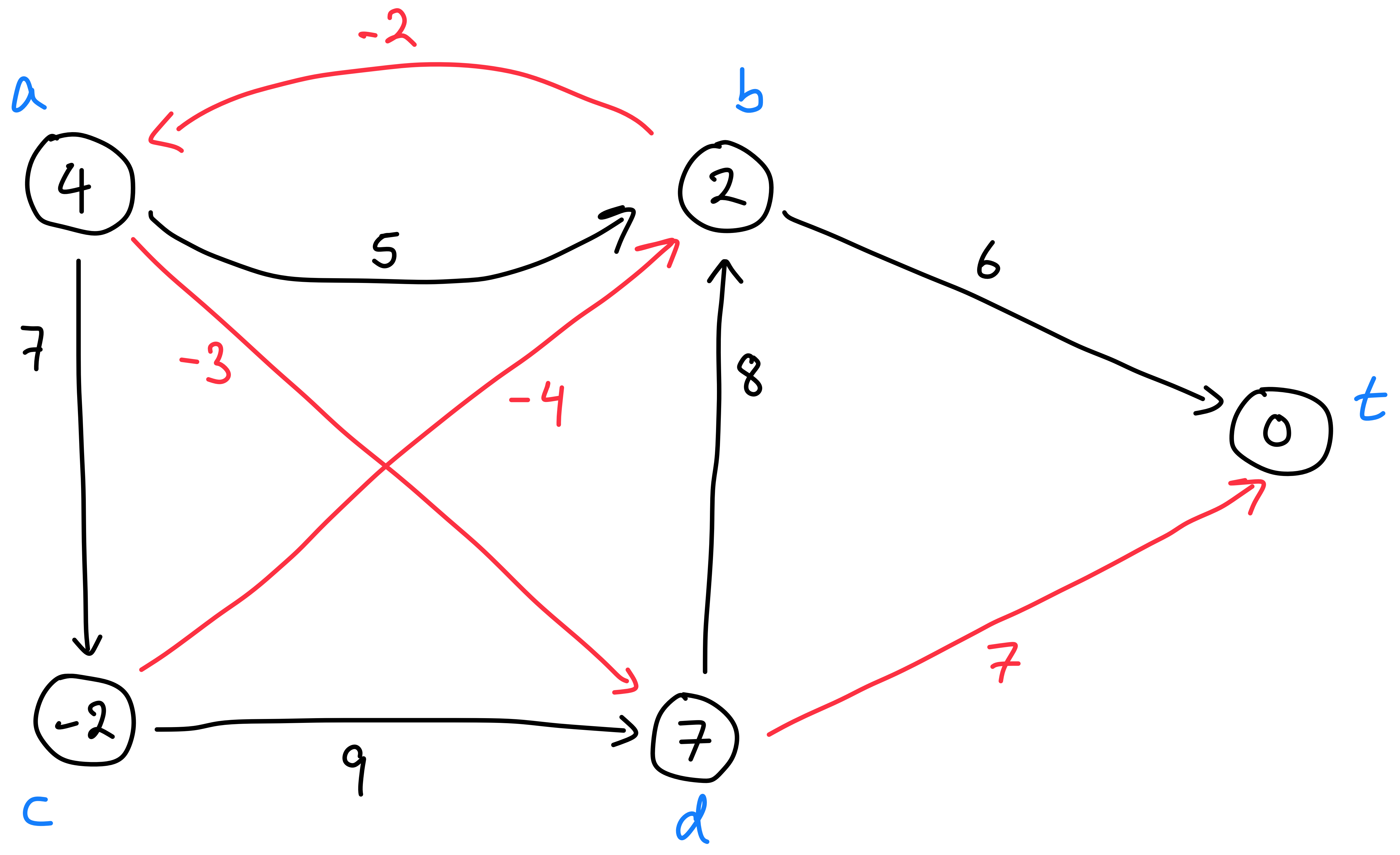


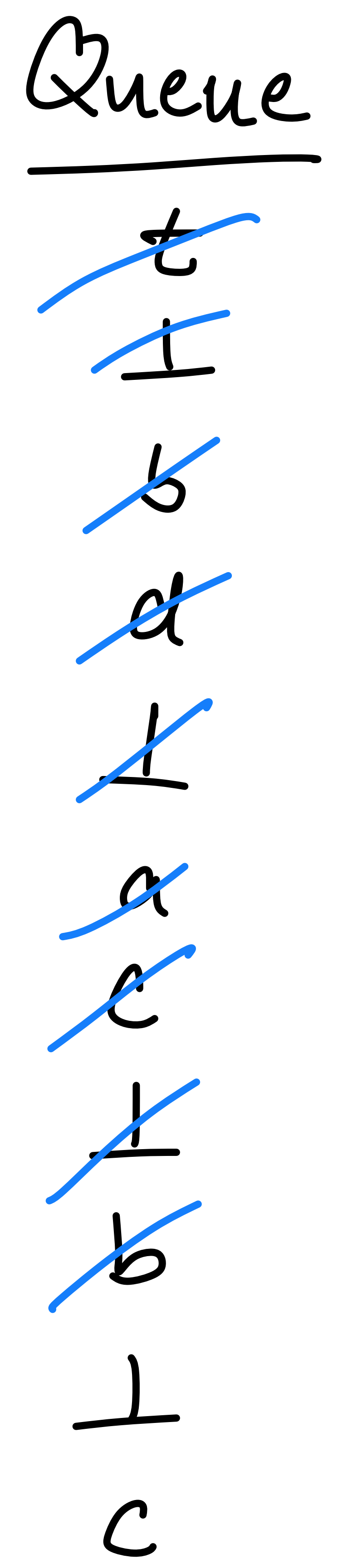


Bellman-Ford example

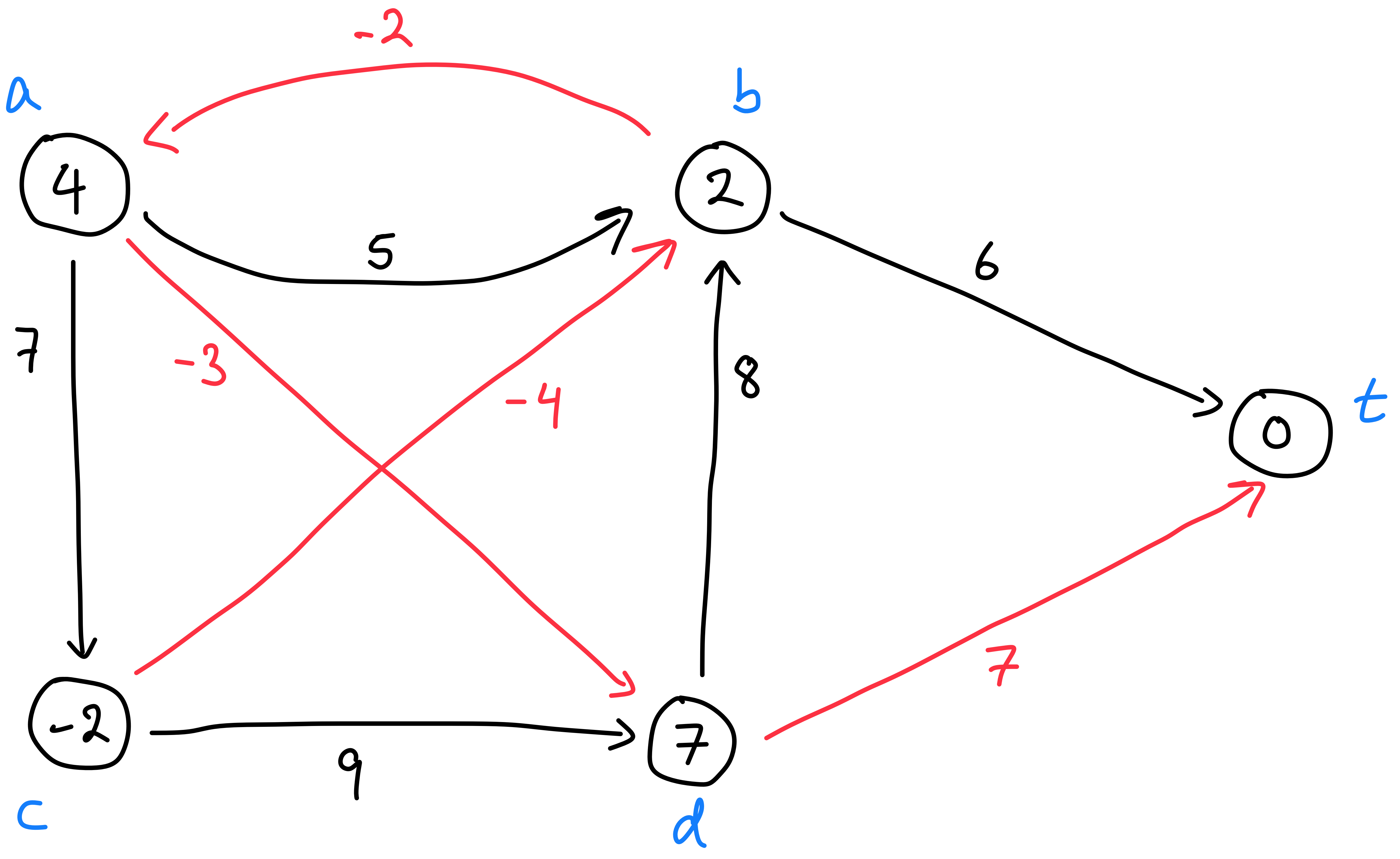


Bellman-Ford example





Bellman-Ford example



Queue

- ~~t~~
- ~~⊥~~ ①
- ~~b~~
- ~~d~~
- ~~⊥~~ ②
- ~~a~~
- ~~c~~
- ~~⊥~~ ③
- ~~b~~
- ~~⊥~~ ④
- c
- ⊥

n-1 iterations.

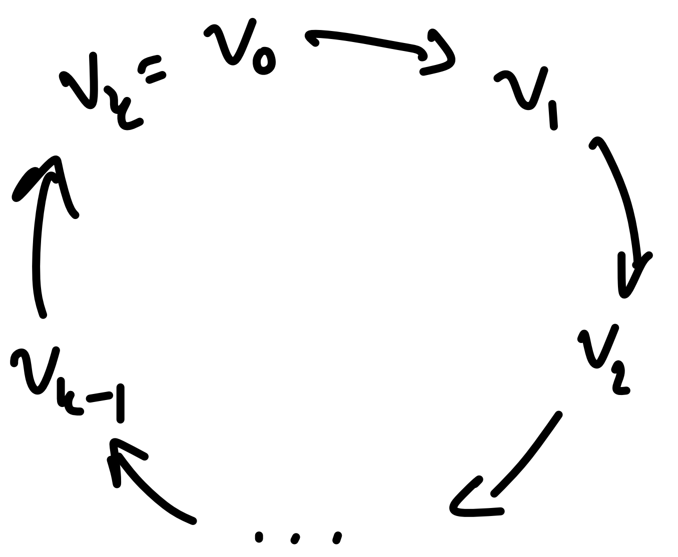
Alg concludes.

Detecting negative cycles

- **Lemma:** If every vertex s can reach t, and G has a negative cycle, then there is some edge $u \to v$ so that $d(n-1,u) > d(n-1,v) + w(u,v)$. If G has no negative cycles, then output of Bellman-Ford is correct on final iteration.

- **Proof:** By contradiction.

Let G have a negative cycle.



$$\sum_{i=0}^{k-1} w(v_i, v_{i+1}) < 0. \quad (1)$$

Assume (for ⊥) that ∀ edges $u \to v$, $d(n-1,u) \le d(n-1,v) + w(u,v)$.

Adding up these equations for the cycle,

$$\sum_{i=0}^{k-1} d(n-1, v_i) \le \sum_{i=0}^{k-1} d(n-1, v_{i+1}) + \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

same term

$$\Rightarrow \quad 0 \le \sum_{i=0}^{k-1} w(v_i, v_{i+1}) \quad (2)$$

(1) and (2) are inconsistent, proving the contradiction.

Detecting negative cycles

- **Lemma:** If every vertex s can reach t, and G has a negative cycle, then there is some edge $u \to v$ so that $d(n-1,u) > d(n-1,v) + w(u,v)$. If G has no negative cycles, then output of Bellman-Ford is correct on final iteration.

- **Proof:** The previous slide proves the first part of the statement.

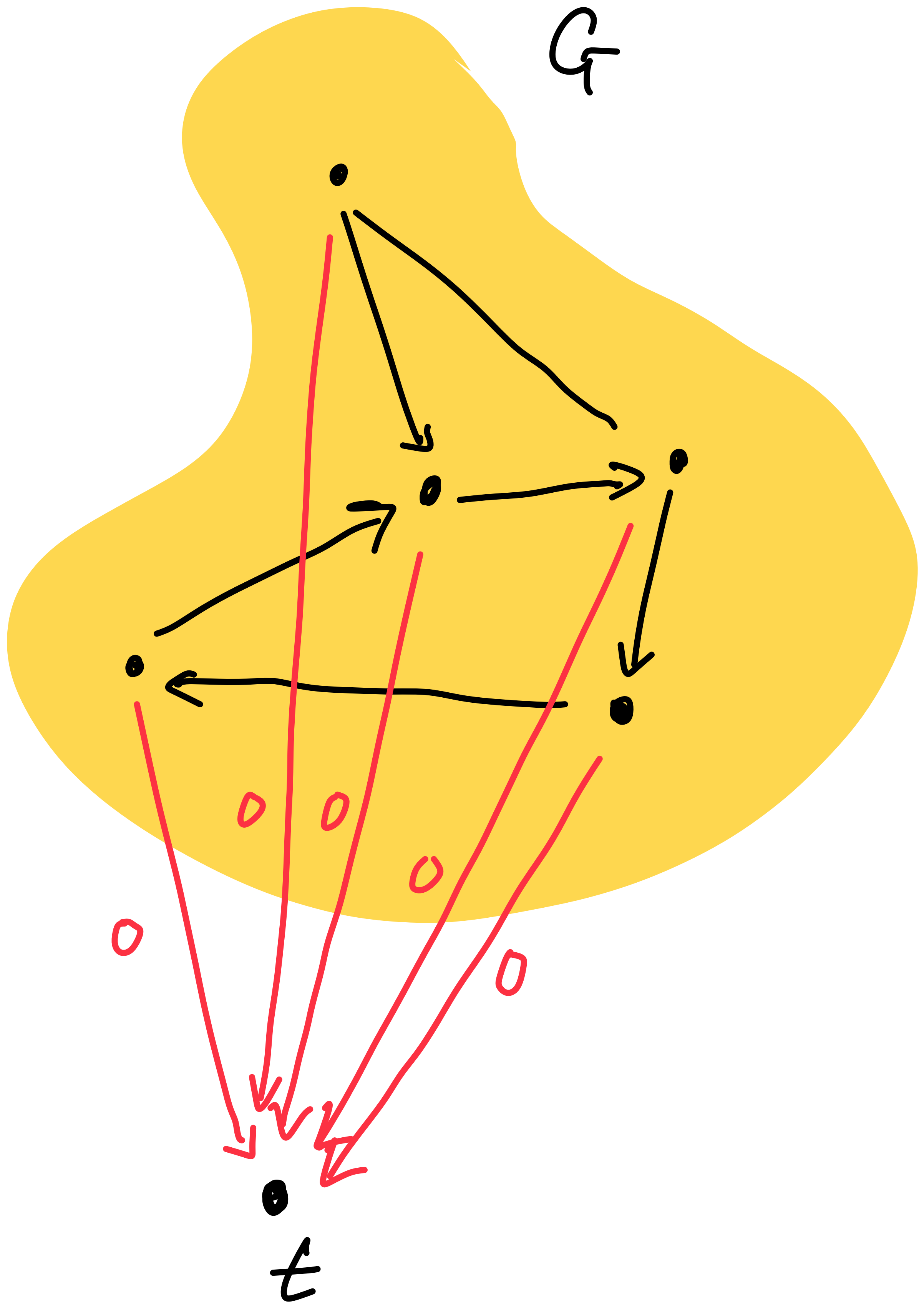
    - If there are no negative cycles, the shortest path $s \rightsquigarrow t$ consists of unique vertices and has length $\leq n-1$.

    - We previously proved that $d(i,s)$ was optimal length of path $s \rightsquigarrow t$ of length $\leq i$.

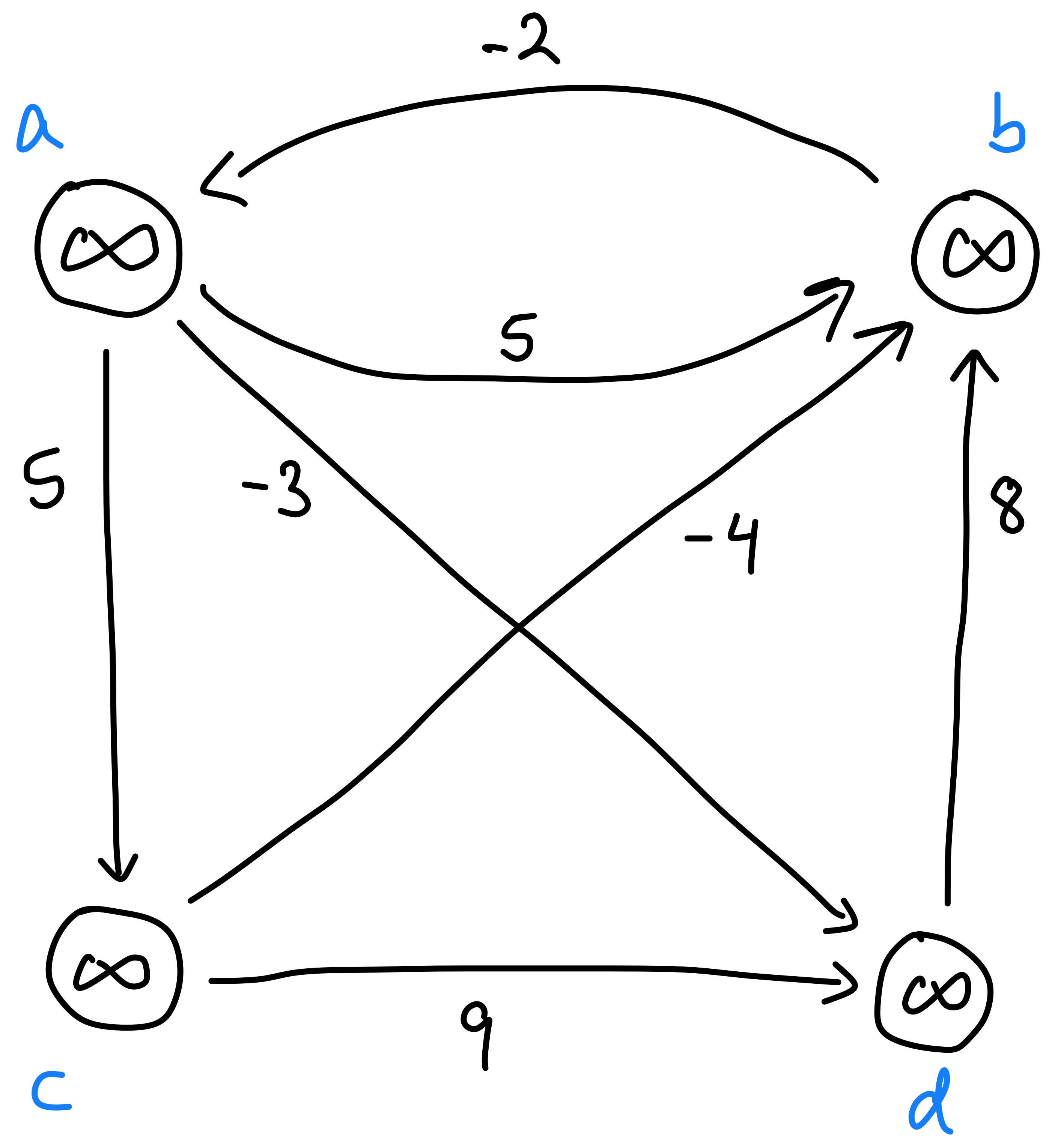
    - Together, concludes proof.

Negative cycle detection

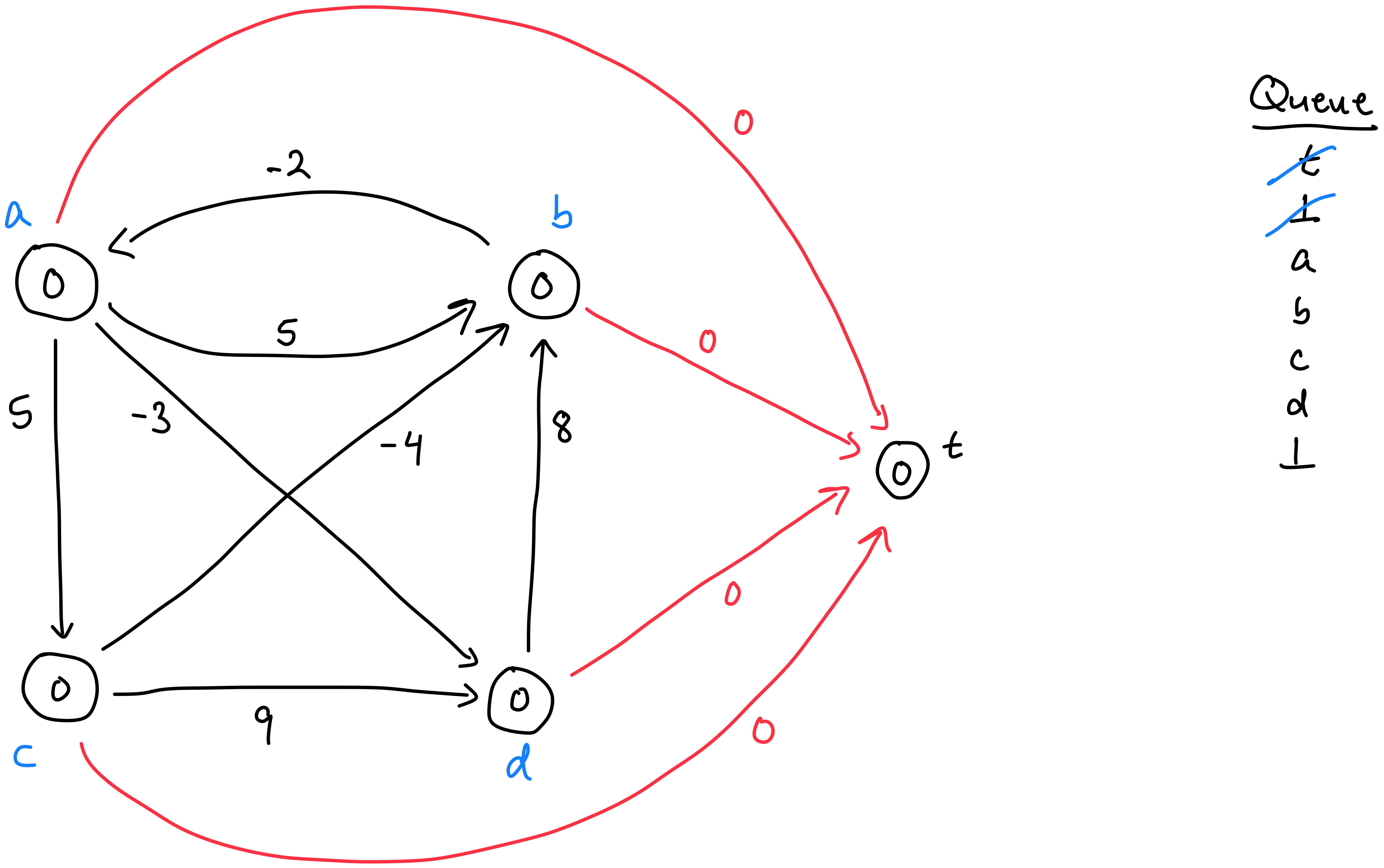
- **Negative cycle detection algorithm:**
  - Run Bellman-Ford assuming there are no negative cycles
  - For each edge $u \to v$, verify that $d(u) \leq d(v) + w(u, v)$. Else, report "negative cycle detected".
- This will only detective negative cycles amongst vertices that have paths to t. Will not detect negative cycles in the entire graph for a poorly connected choice of t.
- Solution: Add a new "sink" t to the graph and add edge $v \to t$ of weight 0 for all vertices. Run detection algorithm w.r.t this sink.



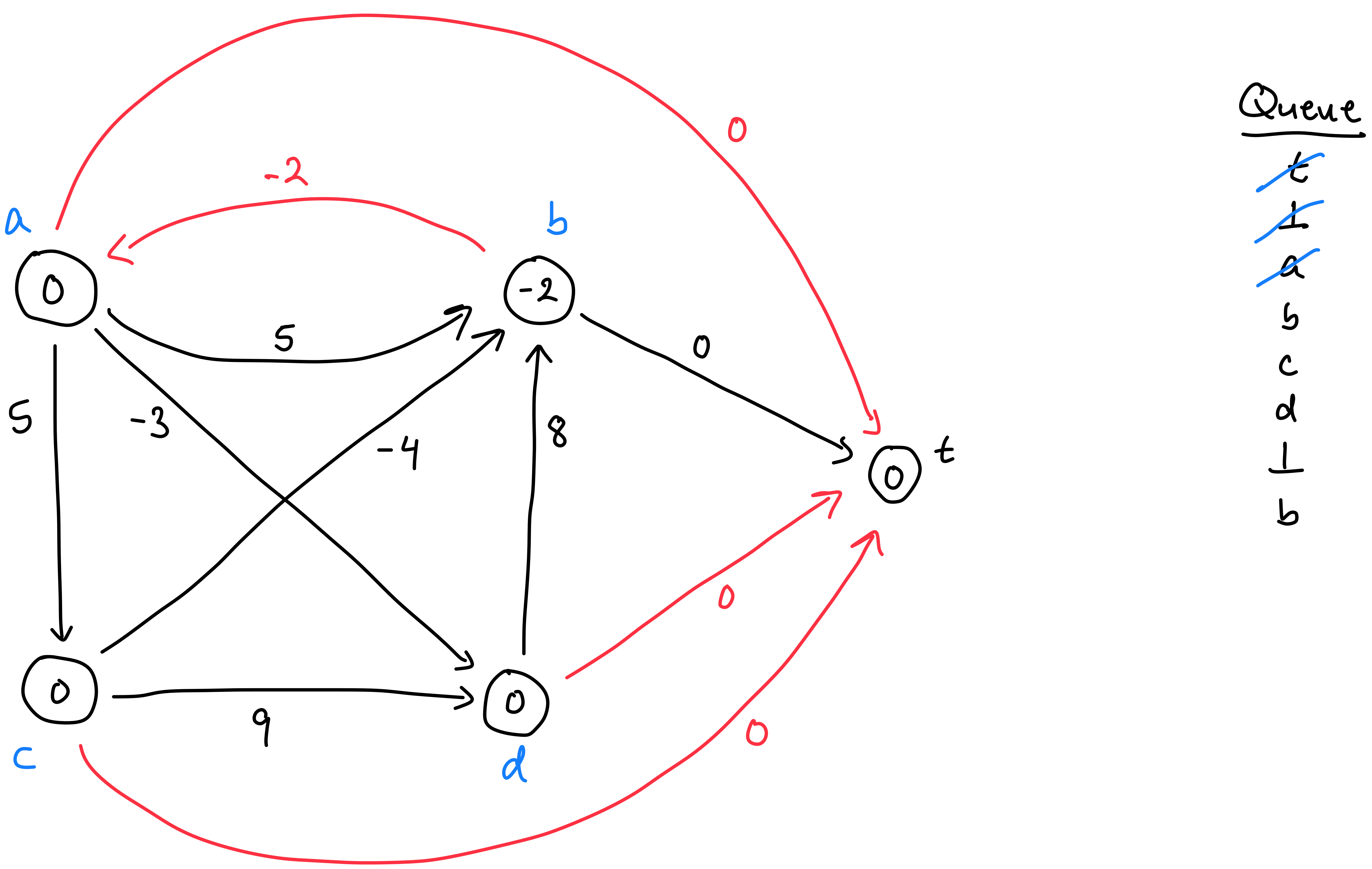
Bellman-Ford with negative cycles example



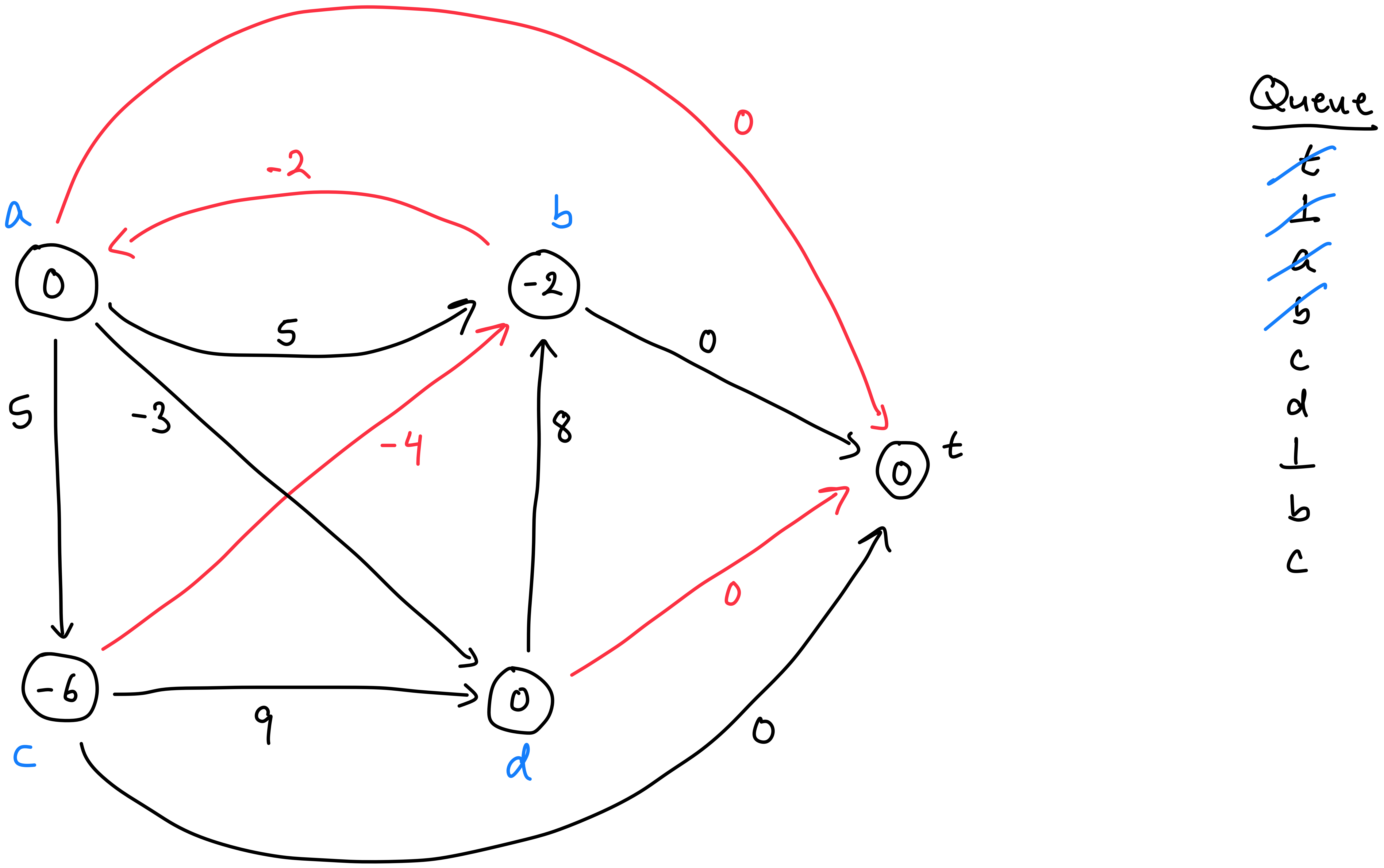
Bellman-Ford with negative cycles example



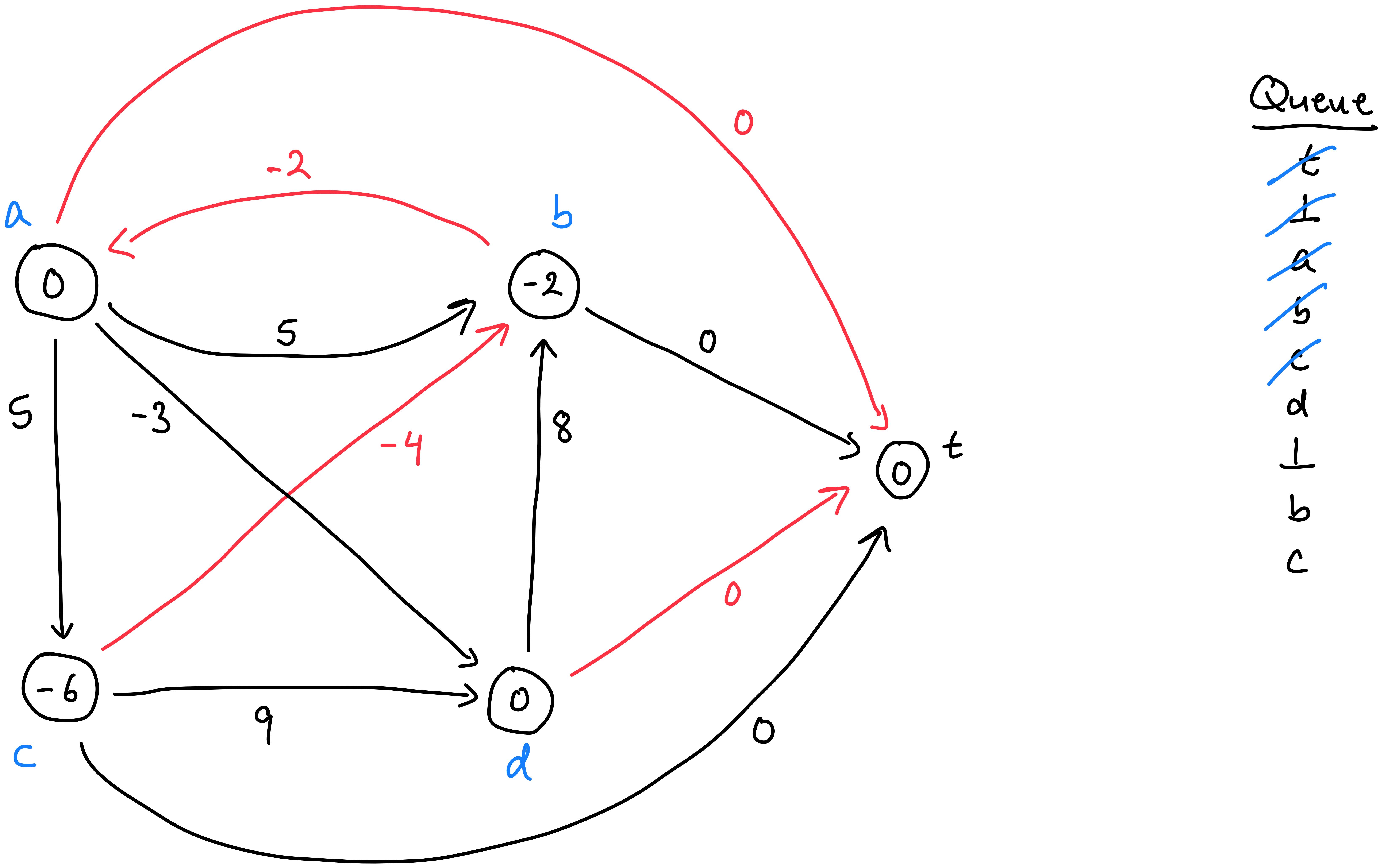
Bellman-Ford with negative cycles example



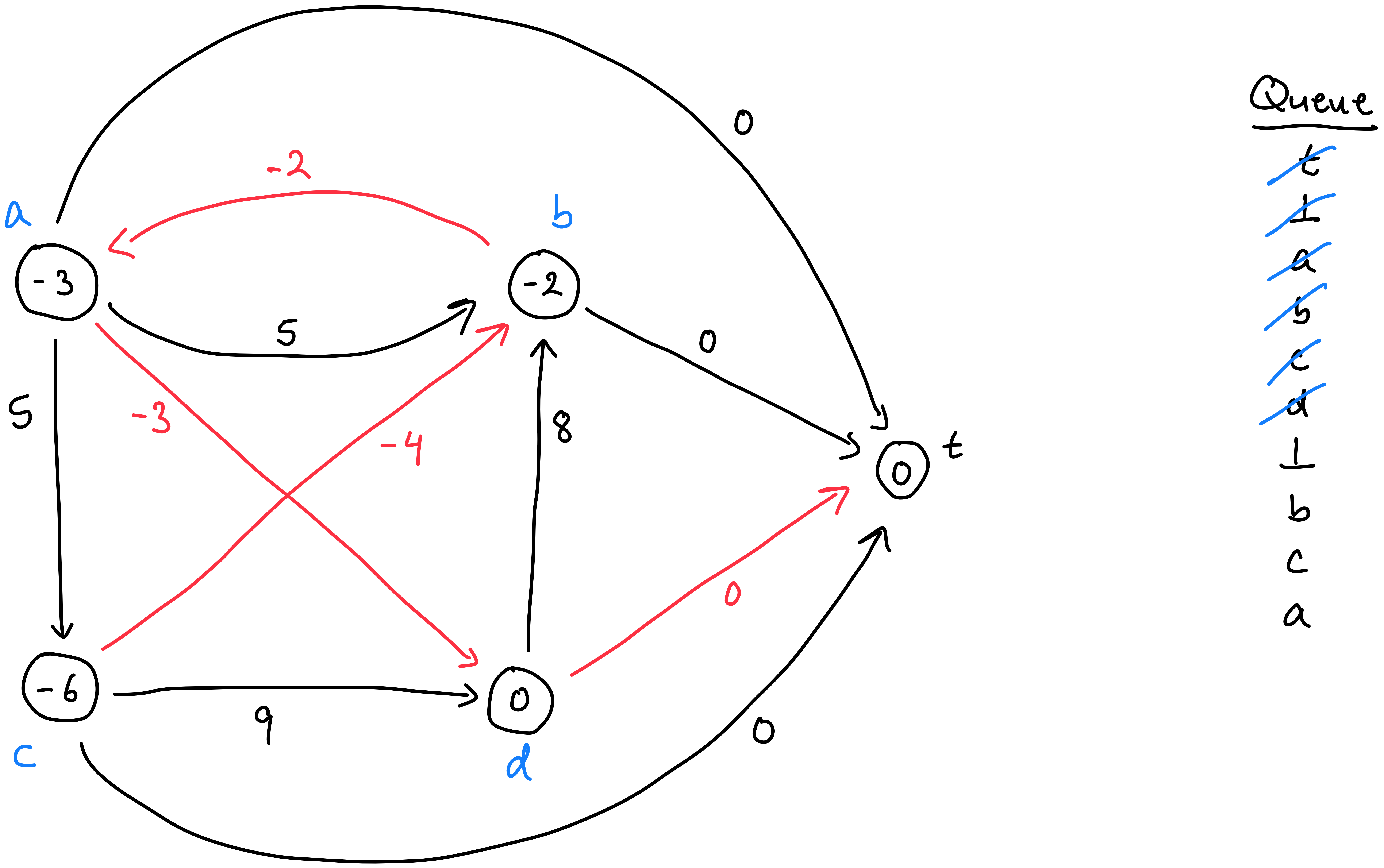
Bellman-Ford with negative cycles example



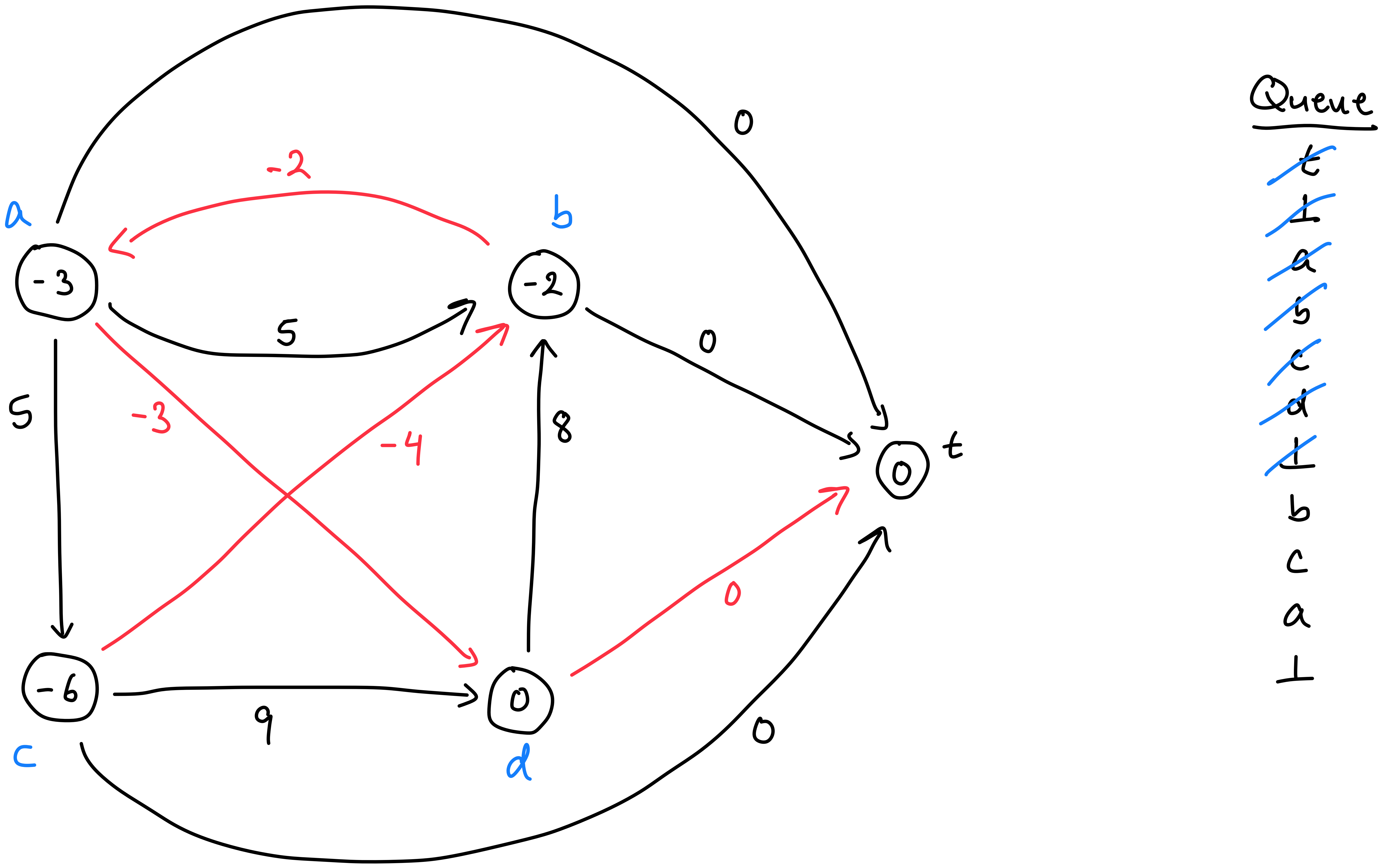
Bellman-Ford with negative cycles example



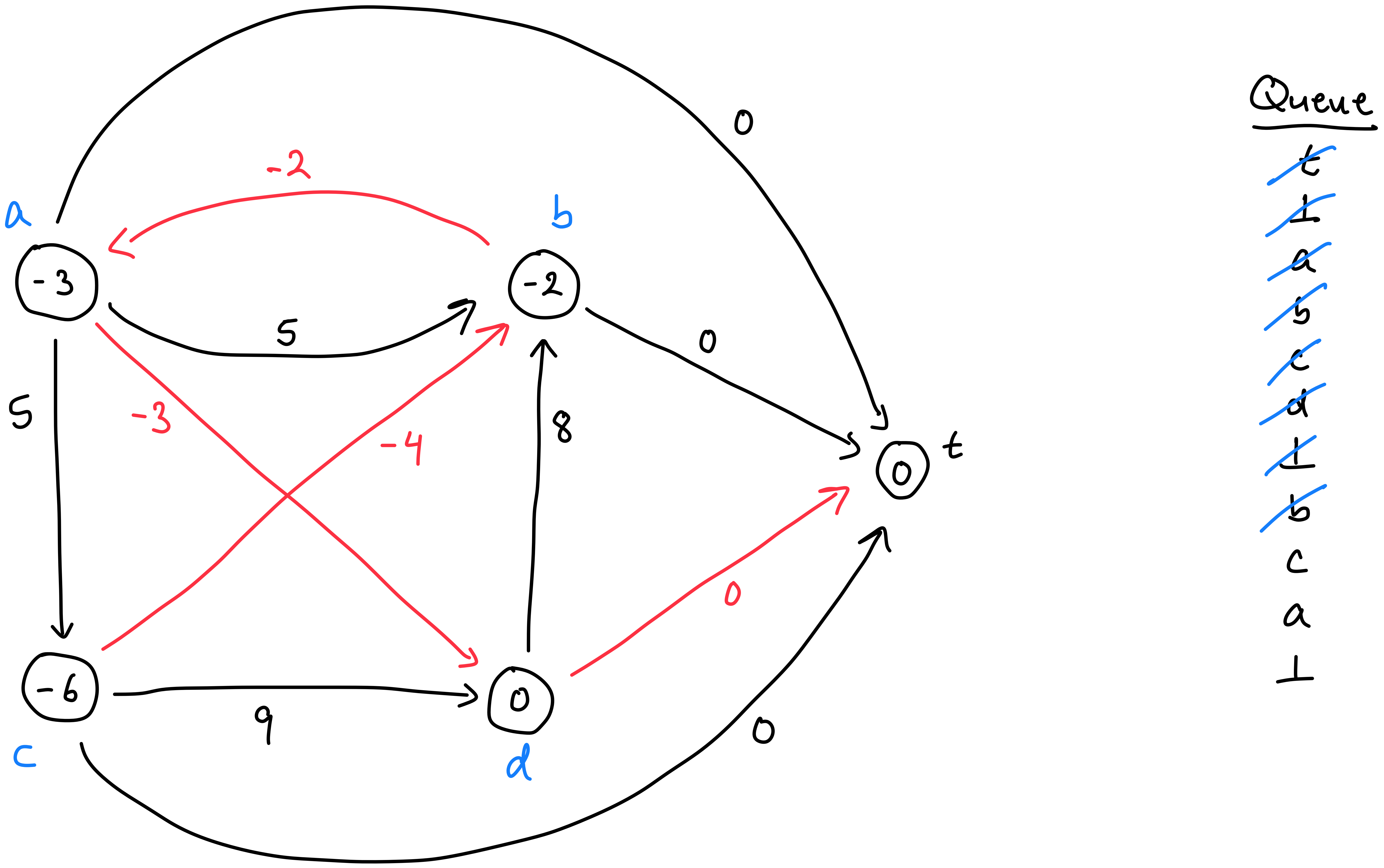
Bellman-Ford with negative cycles example



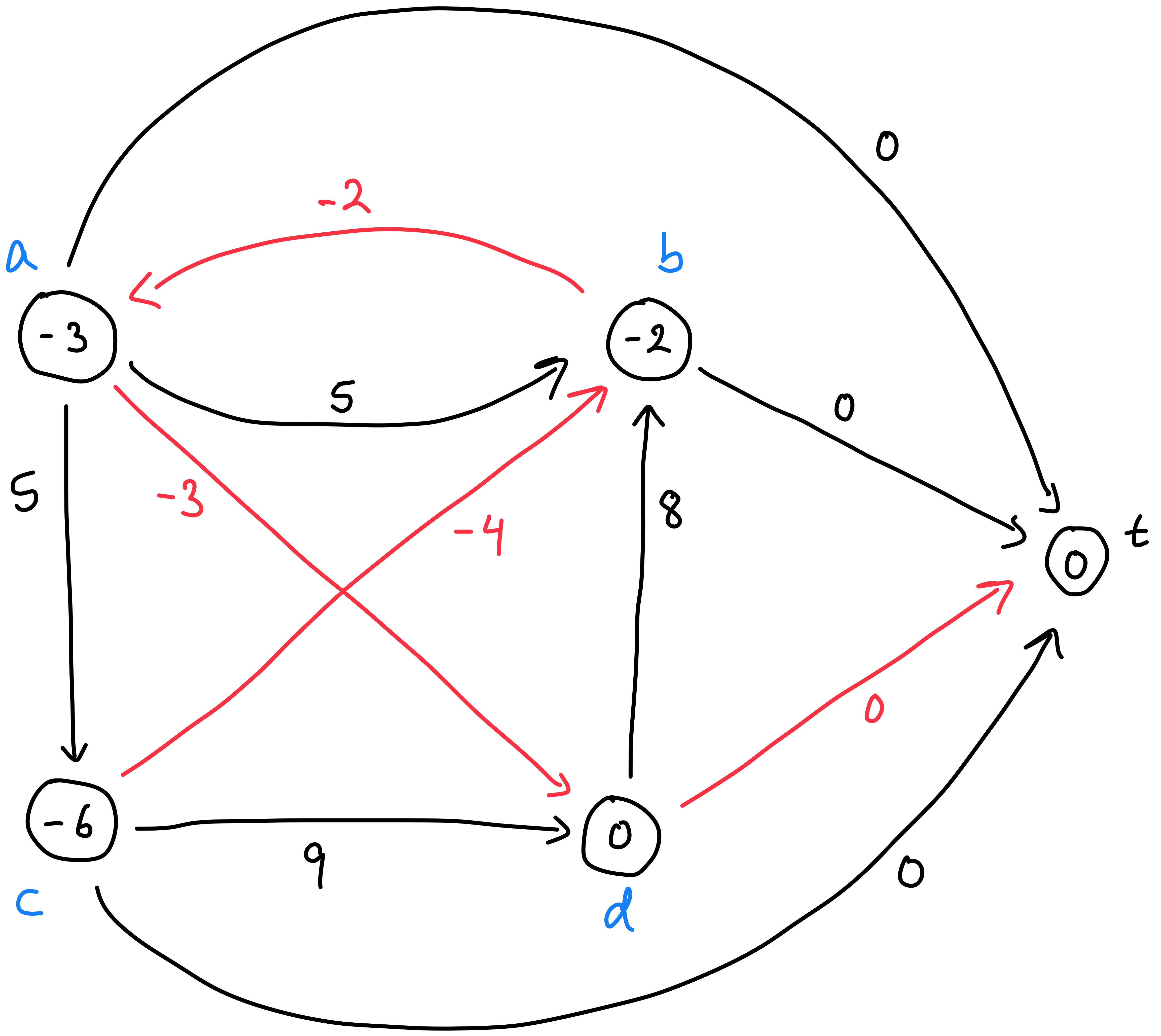
Bellman-Ford with negative cycles example



Bellman-Ford with negative cycles example



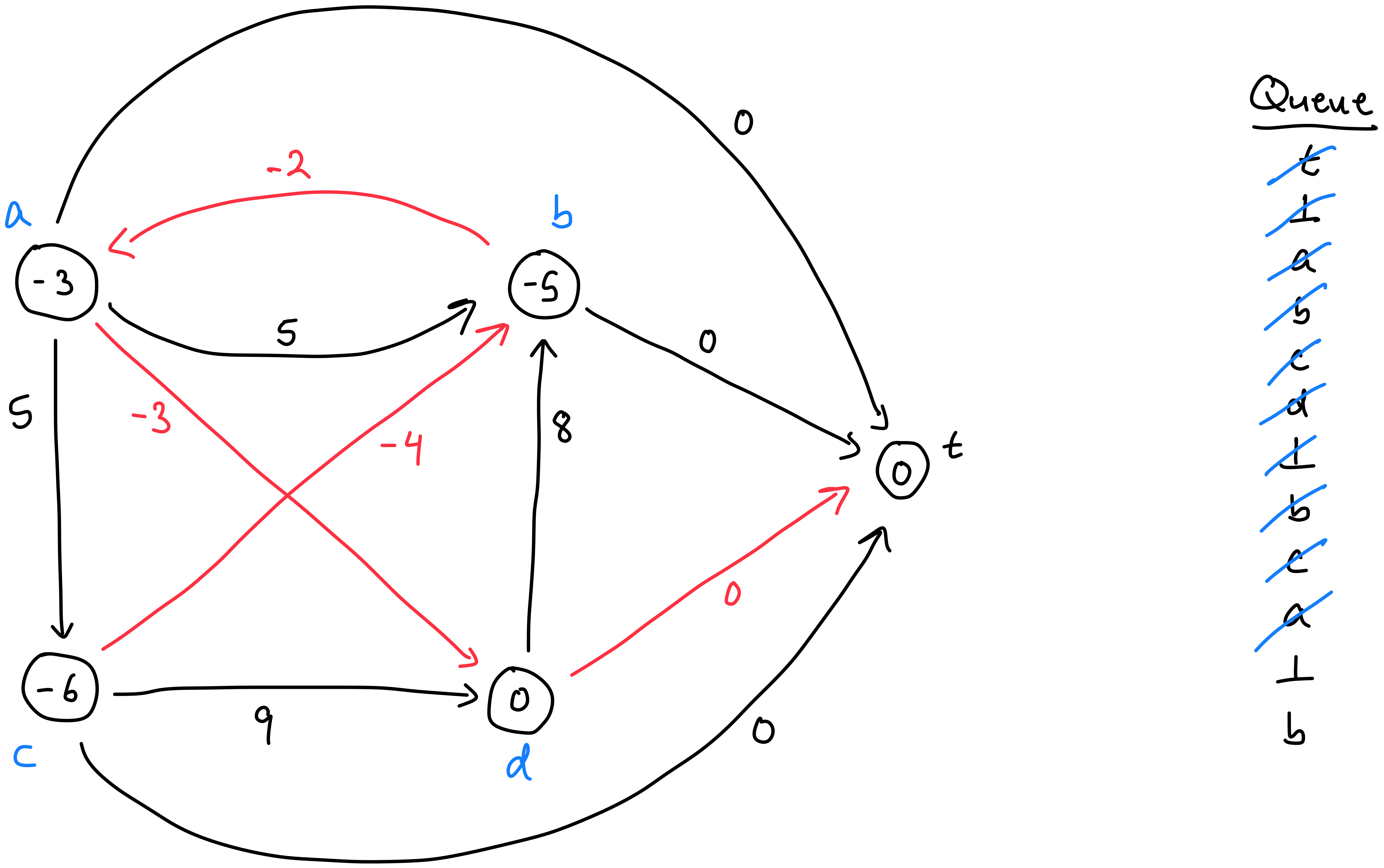
Bellman-Ford with negative cycles example



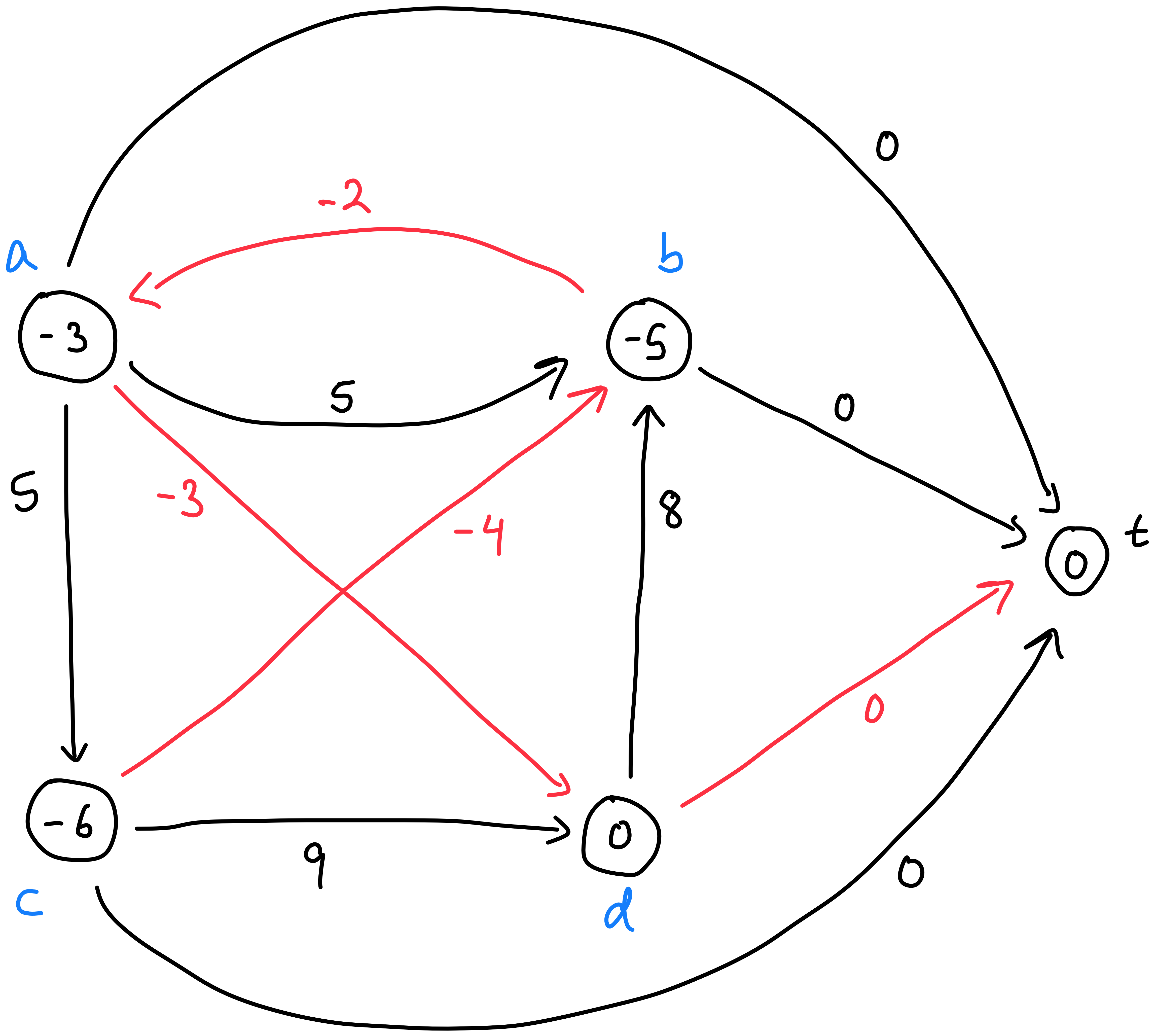
Queue

~~t~~
~~⊥~~
~~a~~
~~b~~
~~c~~
~~d~~
~~⊥~~
~~b~~
~~c~~
a
⊥

Bellman-Ford with negative cycles example



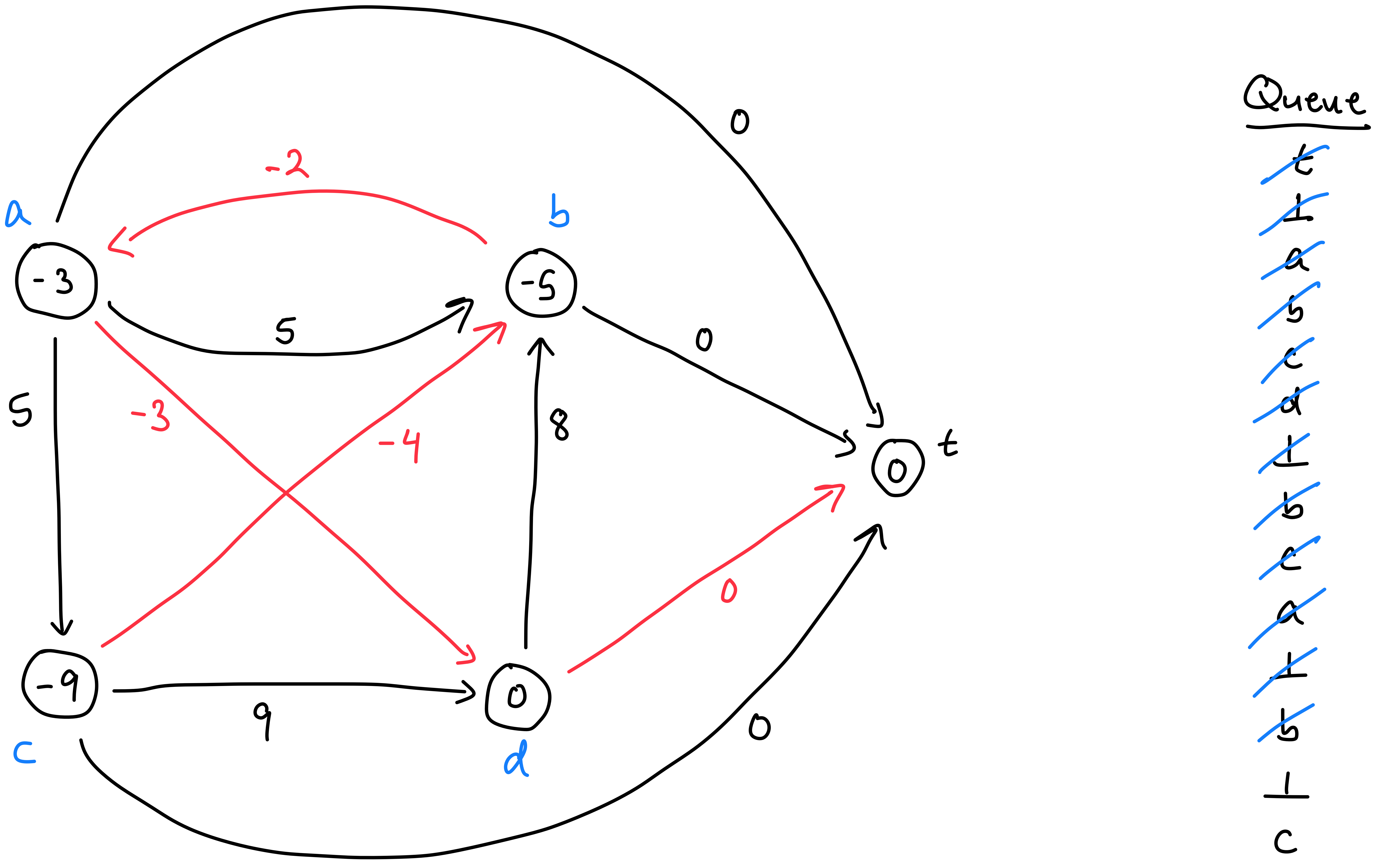
Bellman-Ford with negative cycles example



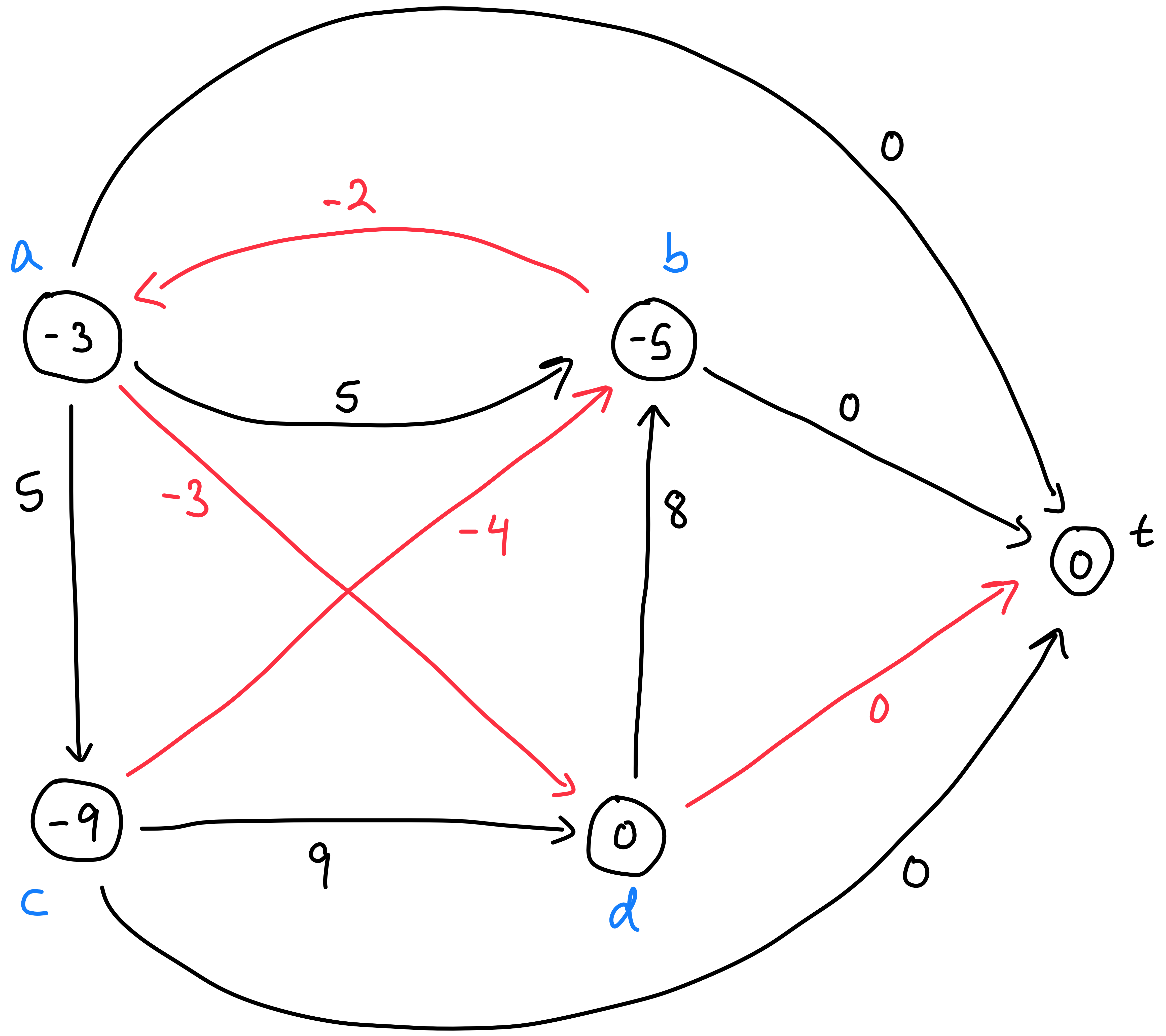
Queue

t
⊥
a
b
c
d
⊥
b
c
a
⊥
b
⊥

Bellman-Ford with negative cycles example



Bellman-Ford with negative cycles example



Queue

~~t~~
~~t~~ ①
~~a~~
~~b~~
~~c~~
~~d~~
~~t~~ ②
~~b~~
~~c~~
~~a~~
~~t~~ ③
~~b~~
~~t~~ ④
c

4 iterations completed.

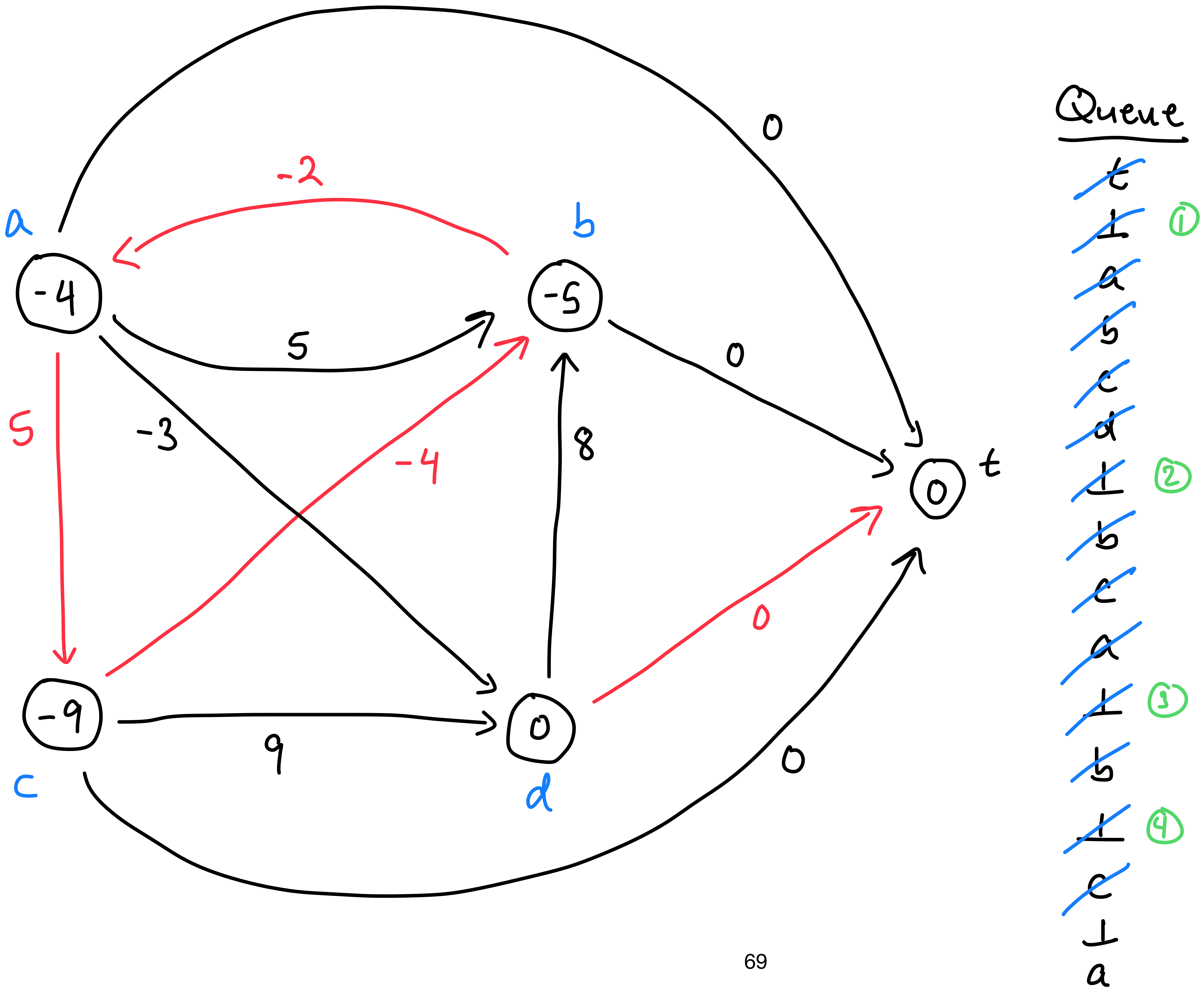
Now checking edges, we notice that

$$d(a) > d(c) + w(a,c)$$

$$-3 > -9 + 5$$

So a negative cycle exists (a → c → b → a).

Bellman-Ford with negative cycles example



Observe what would happen if we updated once more.

Shortest paths with negative weights on a DAG

- No cycles by definition
- Under topological sort, edges only go from low to high numbered vertices
- One pass through the vertices in reverse topological order suffices
- **Runtime:** $O(n + m)$