Dijkstra's algorithm

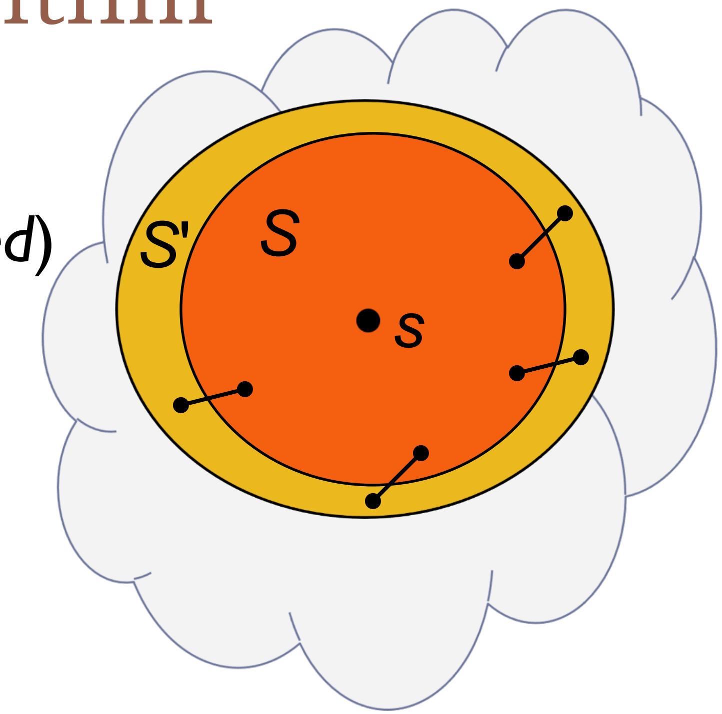
Assumption: $w(p, q) \geq 0$ for all edges (p, q)

```
DIJKSTRA(G,w,s)
INIT;
S = ∅ ;  Q = V ;
while Q ≠ ∅  do {
      q  = MIN_d(Q) ;  Q = Q \ {q} ;  S = S∪{q} ;
      for all r successor of q  do
            RELAX(q, r) ;
}
```

- At each iteration, the algorithm extracts a node from Q that is never returned to Q
- **RELAX**(q, r) may change $d[r]$

Properties of Dijkstra's algorithm

- Algorithm maintains three sets:
  - S: *finished* nodes, for which $d[t] = \delta(s, t)$ (*red*)
  - S': nodes of Q with $d[t]<\infty$ (*yellow*)
  - nodes of Q with $d[t]=\infty$ (*white*)



- Algorithm can be seen as expanding a ball centered at s following a *greedy strategy*

Complexity of Dijkstra's algorithm

Complexity of Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

depends on the data structure for *Q*

we need to support operations:

- insert an element to *Q*
- extract an element with minimum *d* value
- modify (decrease) the *d* value of an element (when relaxing)

Complexity of Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

depends on the data structure for *Q*

we need to support operations:

- insert an element to *Q*
- extract an element with minimum *d* value
- modify (decrease) the *d* value of an element (when relaxing)

⇒ (min-)priority queue

Priority Queues

- (max-)Priority Queue is a data structure that supports operations
  - INSERT(S,*x*)
  - MAX(S)
  - EXTRACT-MAX(S)
  - INCREASE-KEY(S,*x*,*k*): increase the key of *x* to *k*
- Priority Queues are used in
  - Dijkstra’s algorithm for shortest paths
  - Prim’s algorithm for minimum spanning tree
  - other greedy algorithms
- Implemented using heaps

Priority Queues: time bounds

- MAX: $O(1)$
- EXTRACT-MAX, INCREASE-KEY, INSERT: $O(\log n)$

Various improvements have been proposed

- *Fibonacci heaps* take $O(1)$ *amortized* time for INSERT and INCREASE-KEY
- if keys are integers bounded by B, van Emde Boas trees support INSERT, DELETE, MAX, MIN, SUCC, PRED in time $O(\log \log B)$

Back to Dijkstra's algorithm

With adjacency matrix

time $O(n^2)$ (where $n = |V|$)

With adjacency lists

Q : priority queue

if implemented by binary heaps:

n building a heap of n elements : $O(n)$

n operations $\mathbf{MIN}_d$: $O(n \log n)$

m operations **RELAX** : $O(m \log n)$ (where $m = |E|$)

total time $O((n+m) \log n)$

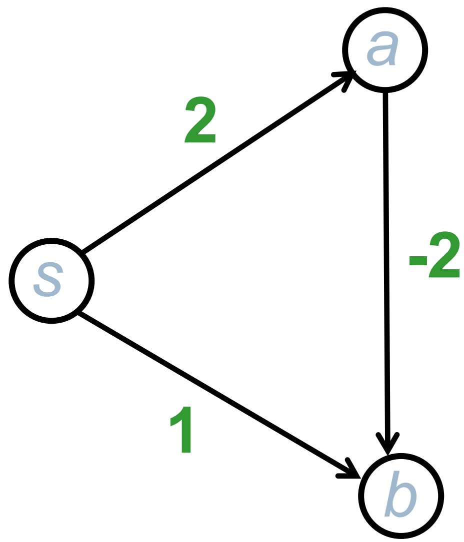
improves over $O(n^2)$ if $m = o(n^2 / \log n)$

time can be improved to $O(n \log n + m)$ using *Fibonacci heaps,*

as decreasing the key takes $O(1)$ amortized

Bellman-Ford algorithm

What about negative weights?



Bellman-Ford algorithm

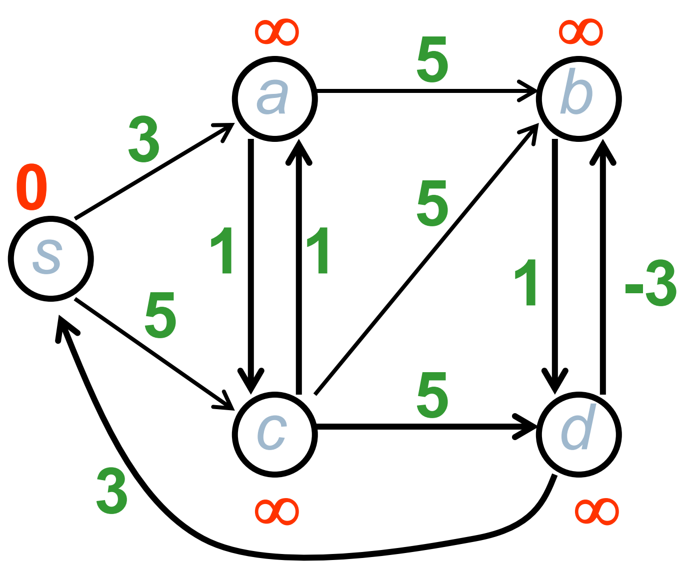
No condition on weights : for all edges (p, q), $w(p, q) \in \mathbf{R}$

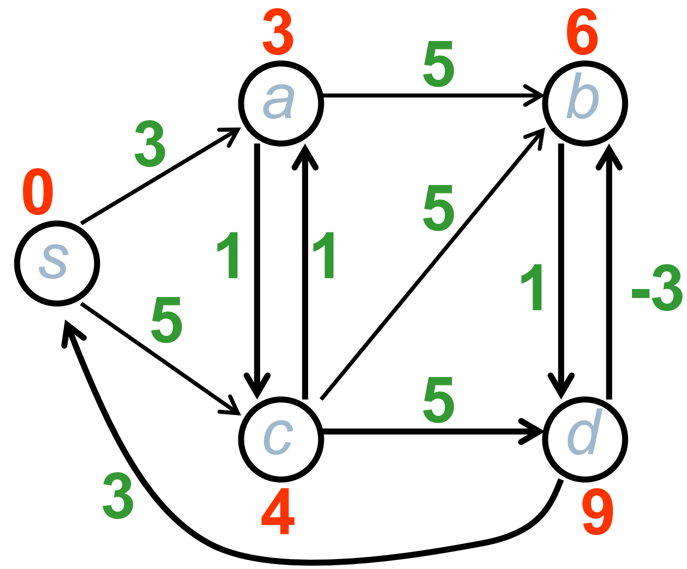
BELLMAN-FORD(G, w, s)

```
INIT;
for i = 1 to |V| - 1 do
    for each (q,r) ∈ E do
        RELAX(q,r) ;
for each (q,r) ∈ E do
    if d[q] + w(q,r) < d[r] then
        return 'negative cost cycle detected'
    else
        return 'minimum costs computed'
```

Time complexity : $O(nm)$

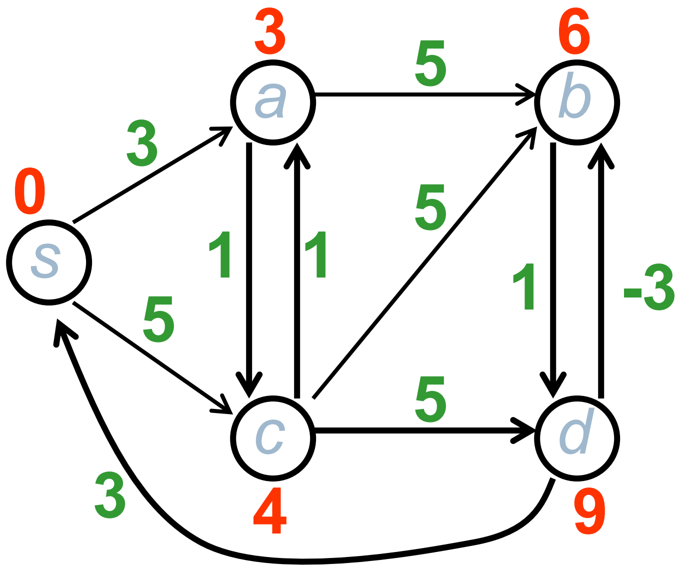
Example 1

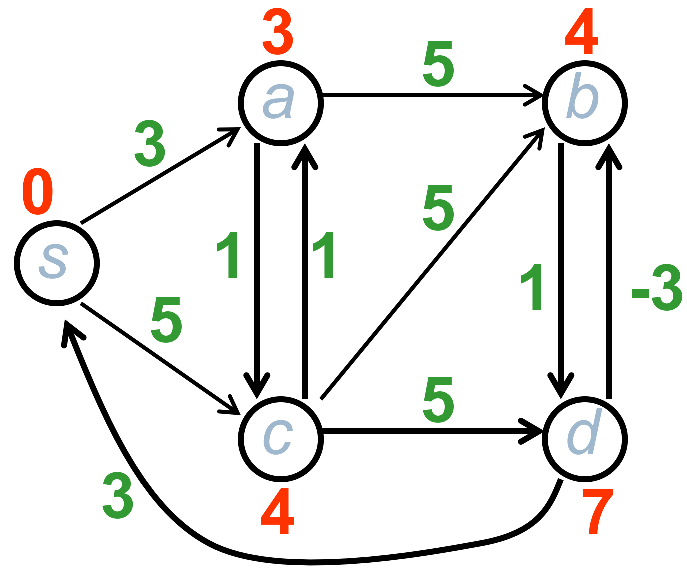




Step 1: relaxing all edges in the following order:
(s,a) (s,c) (a,b) (a,c) (b,d) (c,a) (c,b) (c,d) (d,b) (d,s)

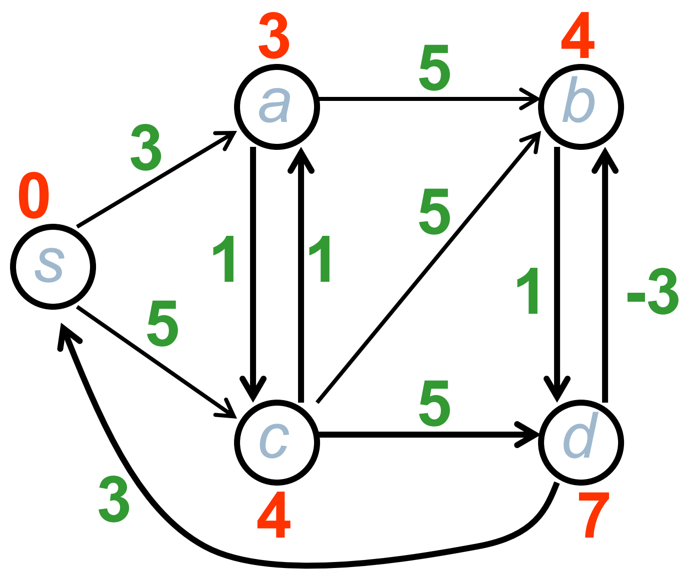
Example 1 (cont)

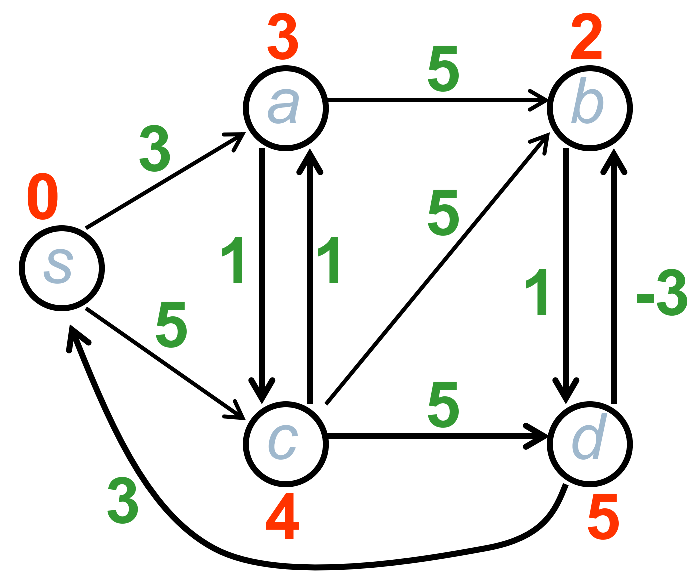




Step 2: relaxing all edges in the following order:
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

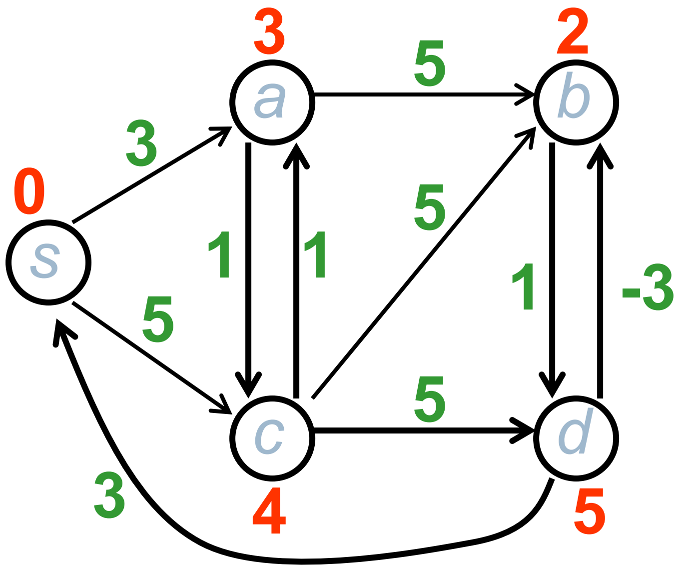
Example 1 (cont)

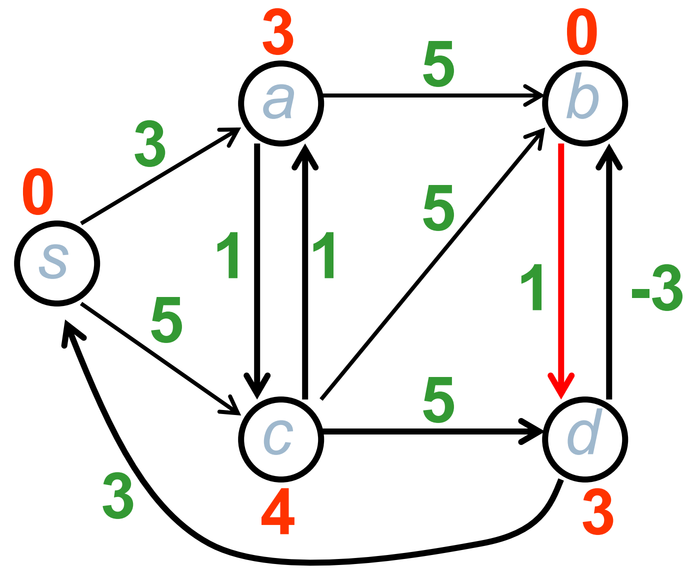




Step 3: relaxing all edges in the following order:
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Example 1 (cont)

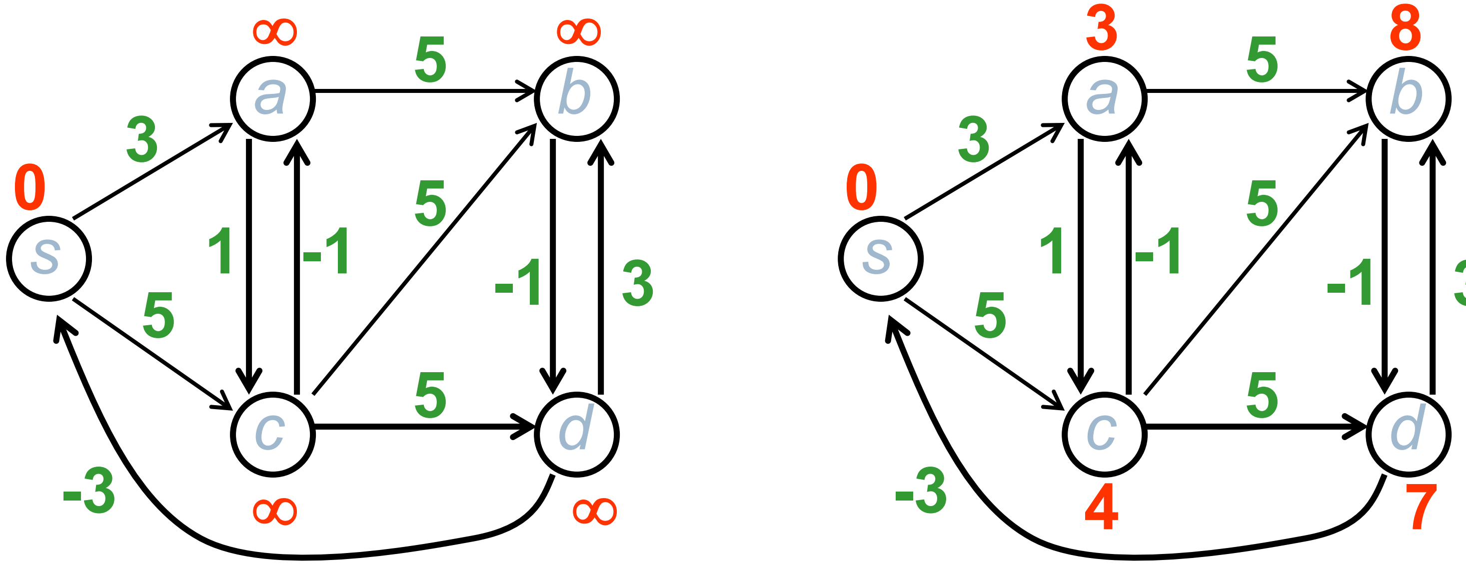




Step 4: relaxing all edges in the following order:
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

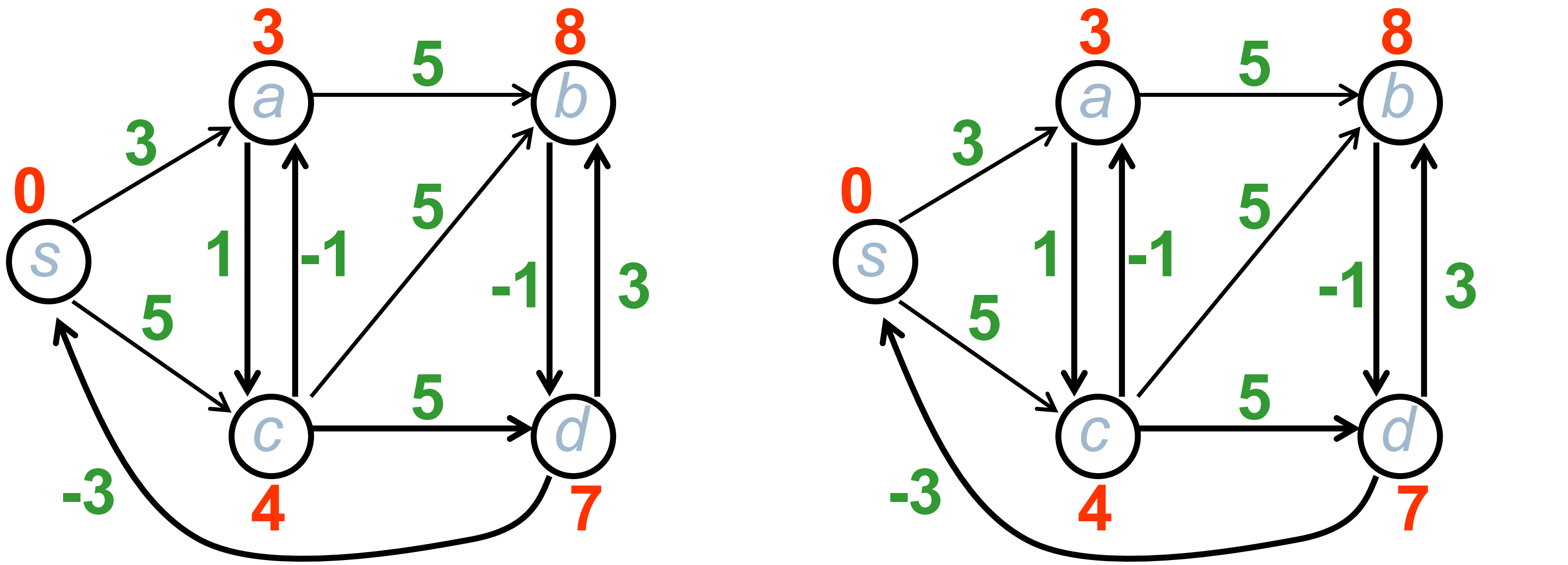
relaxation still possible ⇒ cycle of negative cost

Example 2



Step 1: relaxing all edges in the following order:
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

Example 2 (cont)



Step 2: relaxing all edges in the following order:
(*s*,*a*) (*s*,*c*) (*a*,*b*) (*a*,*c*) (*b*,*d*) (*c*,*a*) (*c*,*b*) (*c*,*d*) (*d*,*b*) (*d*,*s*)

no more possible relaxation ⇒ costs correctly computed

Why Bellman-Ford is correct?

because, if there is no negative-cost cycle, every node has a cycle-free shortest path with at most $|V| - 1$ edges

$((s_0, s_1), (s_1, s_2), \cdots, (s_{k-1}, s_k))$ with $s_0 = s$ and $s_k = t$, $k \le |V| - 1$

At iteration i, we will relax (among other edges) (s_{i-1}, s_i). This guarantees the shortest path value for all nodes. No relaxation will be possible anymore.

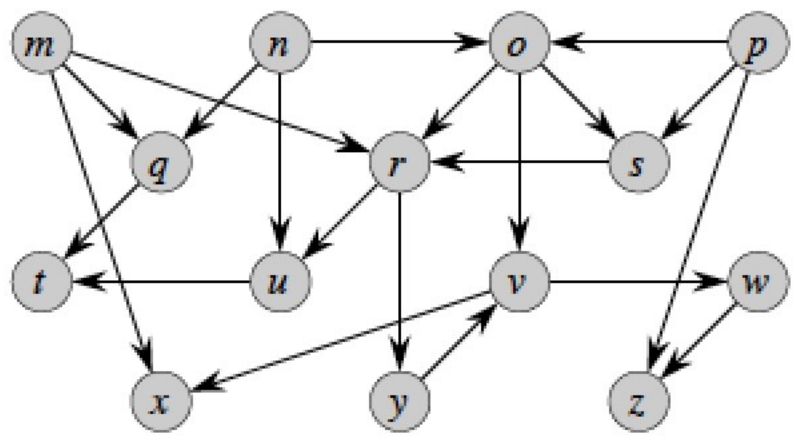
If there exists a negative-cost cycle, one of the edges along the cycle must be possible to relax (prove).

Quiz 1

Shortest paths in Directed Acyclic Graphs

Directed Acyclic Graph (DAG)

- Directed graph without cycles
- ⇒ at least one node with indegree 0, and at least one with outdegree 0

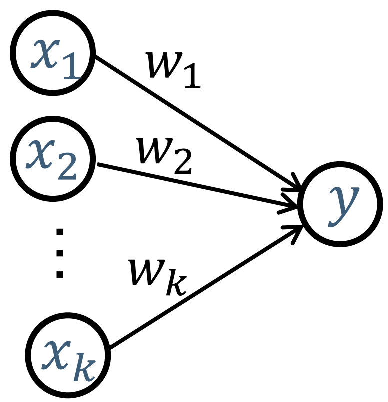


Shortest paths in Directed Acyclic Graphs

- $G = (V, E), w: E \rightarrow \mathbb{R}$ (possibly negative)
- *Problem*: given a node $s \in V$, compute shortest paths from s to all other nodes reachable from s

Shortest paths in Directed Acyclic Graphs

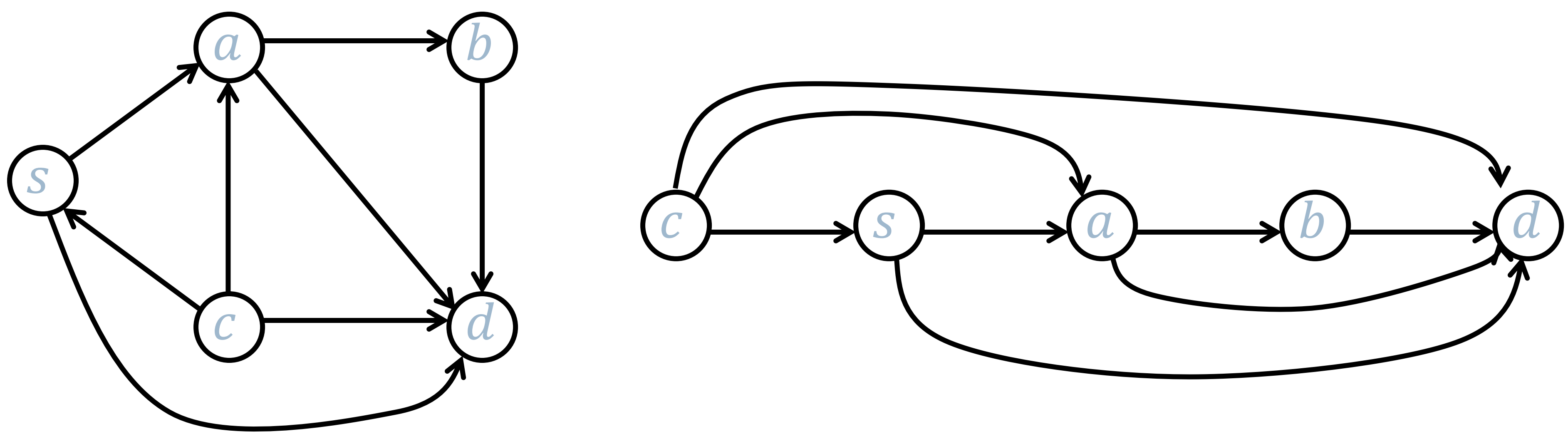
- $G = (V, E), w: E \to \mathbb{R}$ (possibly negative)
- *Problem*: given a node $s \in V$, compute shortest paths from s to all other nodes reachable from s



- *main step*: $d[y] = \min_i \{d[x_i] + w_i\}$

Topological sort

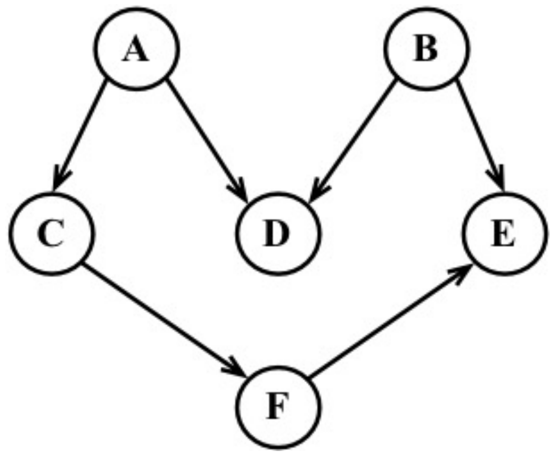
- linearly order vertices such that all edges go from smaller to larger



- Topological sort can be done in time $O(n + m)$ (iterative solution using a queue, solution based on DFS, …)

Topological sort (cont)

- Topological order is not unique



Possible orders:

ABCDFE
BADCFE
ACFBED
...

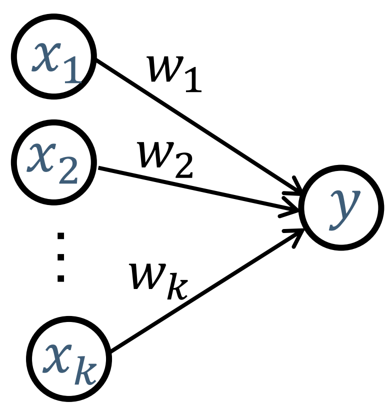
Quiz 2

“Swipe-through” solution

- **`INIT`**;
- starting from s, for all y in topological order

$$d[y] = \min\{d[x_1] + w_1, d[x_2] + w_2, \cdots, d[x_k] + w_k\}$$

$$\pi[y] = x_i \text{ for } i = \operatorname{argmin}\{\cdots\}$$

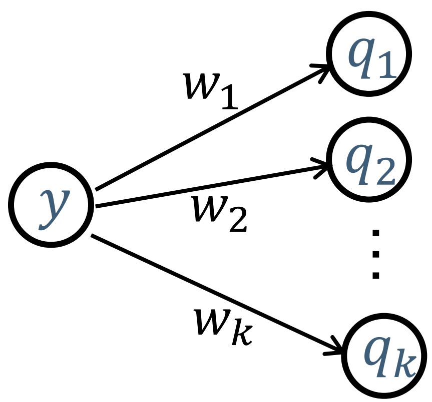


Time: $O(n+m)$

“Dijkstra-style” solution

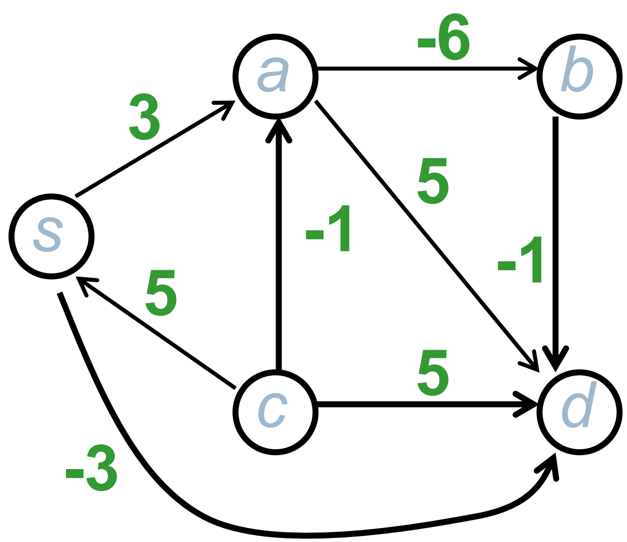
- **`INIT`**;
- starting from s, for all y in topological order

  for each edge (y, q), **`RELAX`**(y, q)



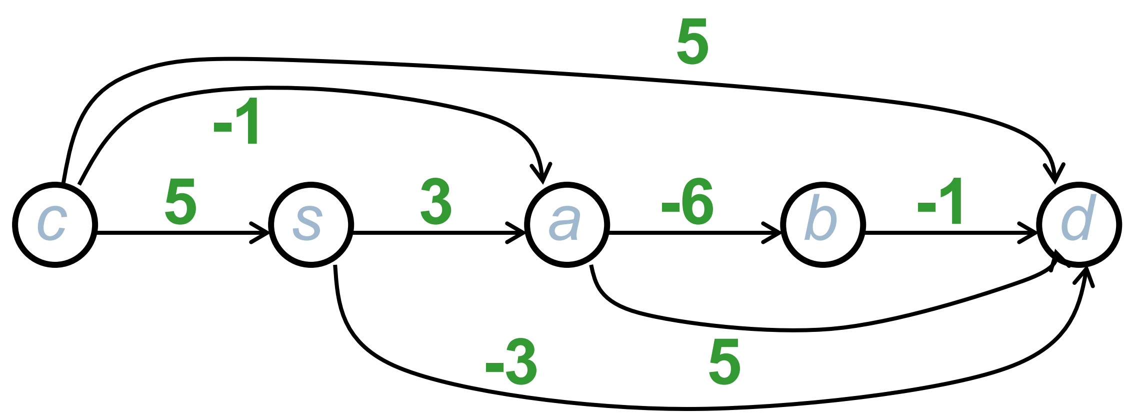
Time: $O(n + m)$

Example

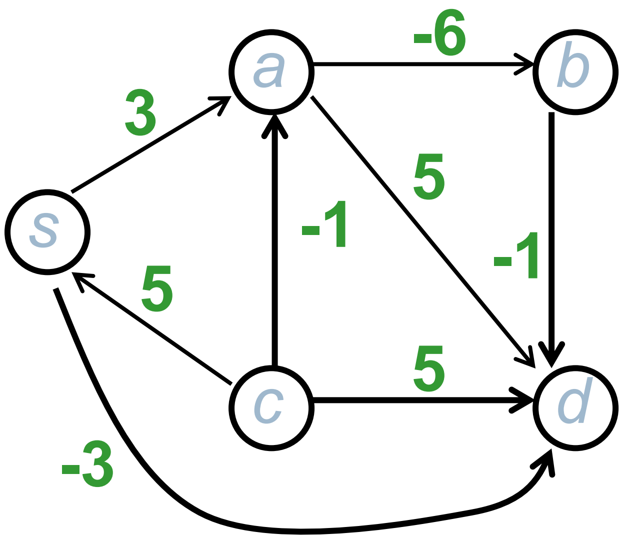


Topological order

c, *s*, *a*, *b*, *d*

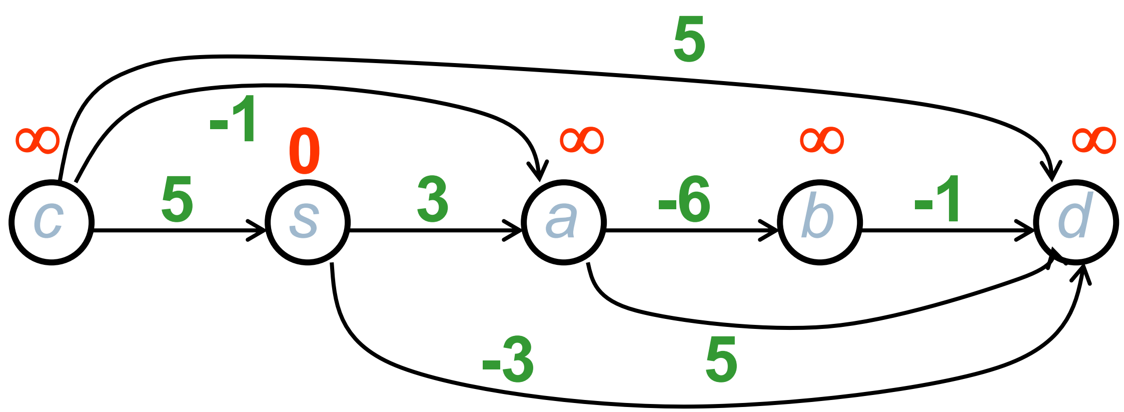


Example

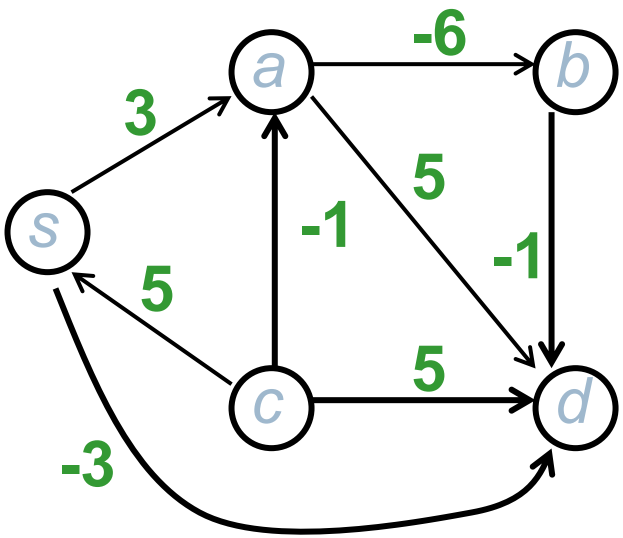


Topological order

c, *s*, *a*, *b*, *d*

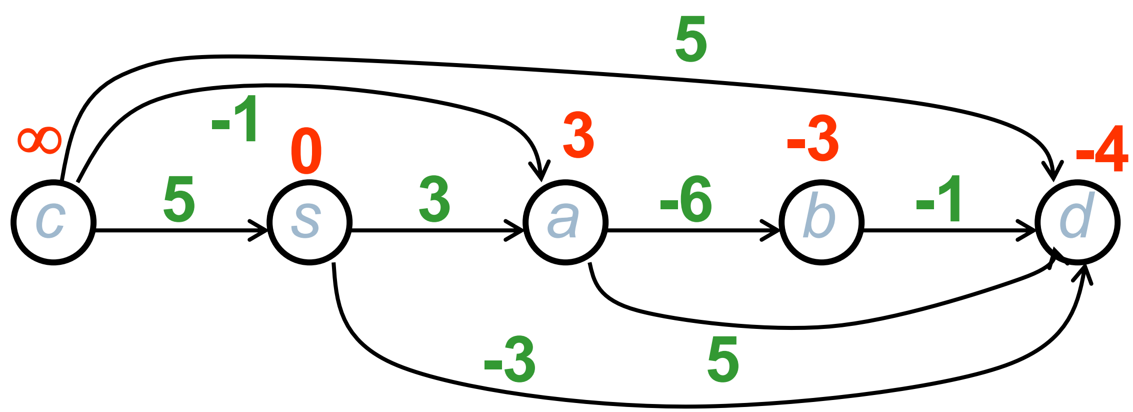


Example

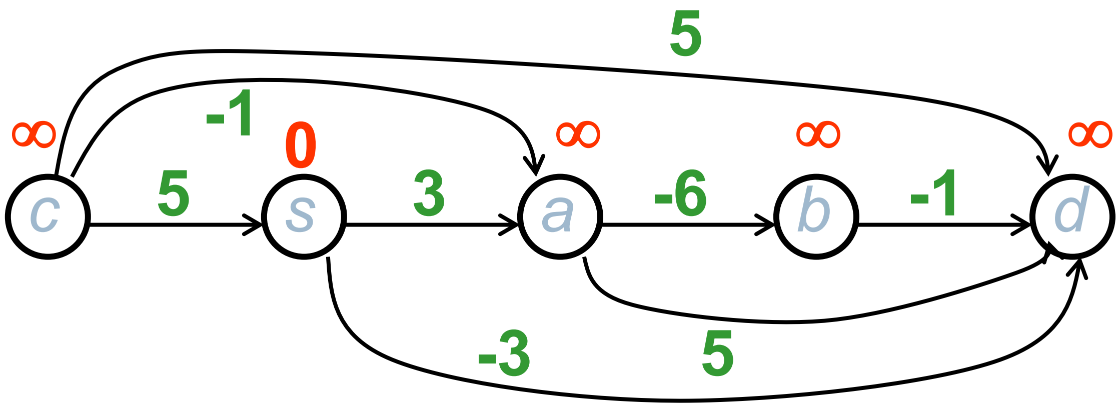


Topological order

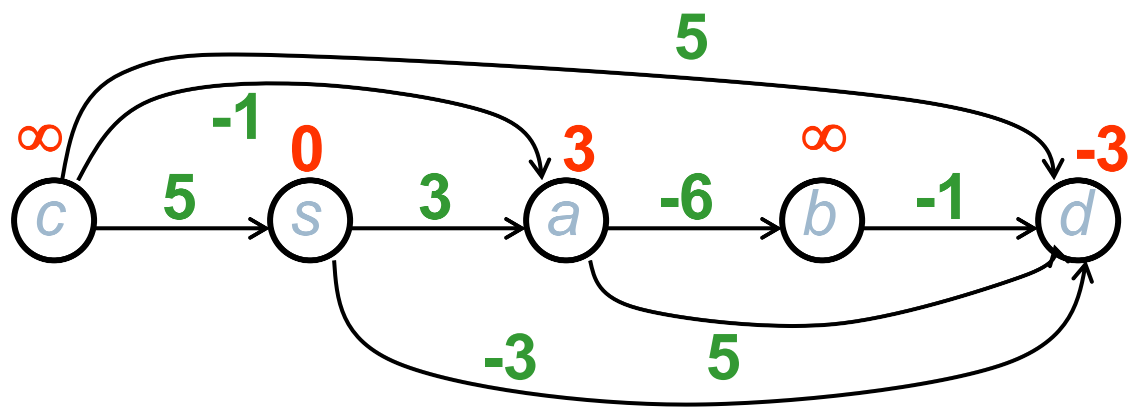
c, *s*, *a*, *b*, *d*



Computing shortest paths (Dijkstra-style)

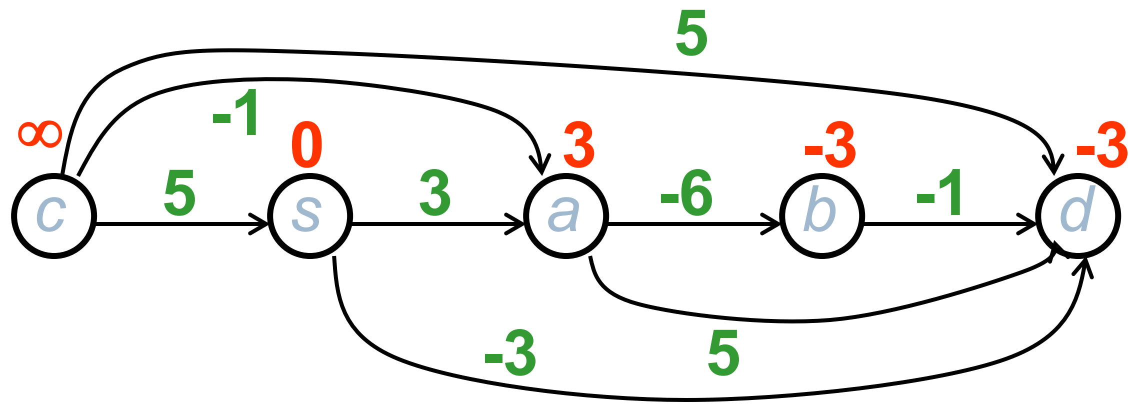


Processing *s*

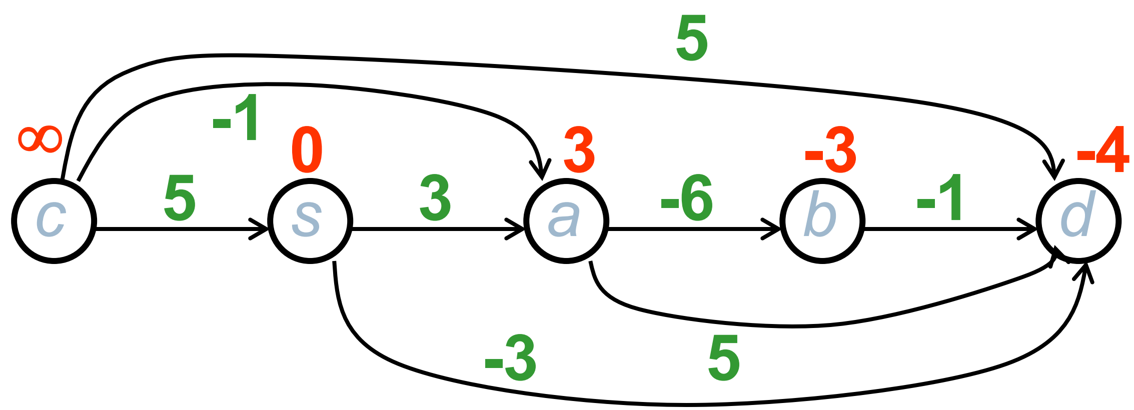


Computing shortest paths (Dijkstra-style)

Processing *a*



Processing *b*



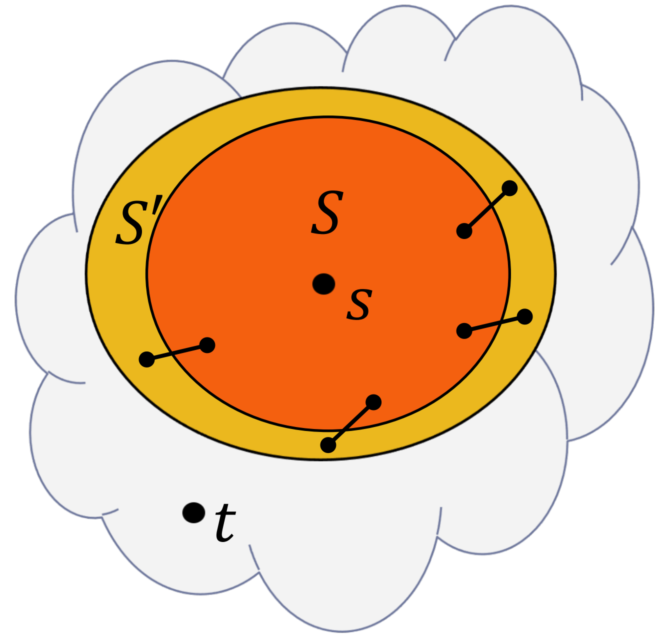
Source-to-destination search

Source-to-destination search

- Assume all edges have non-negative weight. How to search for a shortest path from s to t with Dijkstra's algorithm?

Source-to-destination search

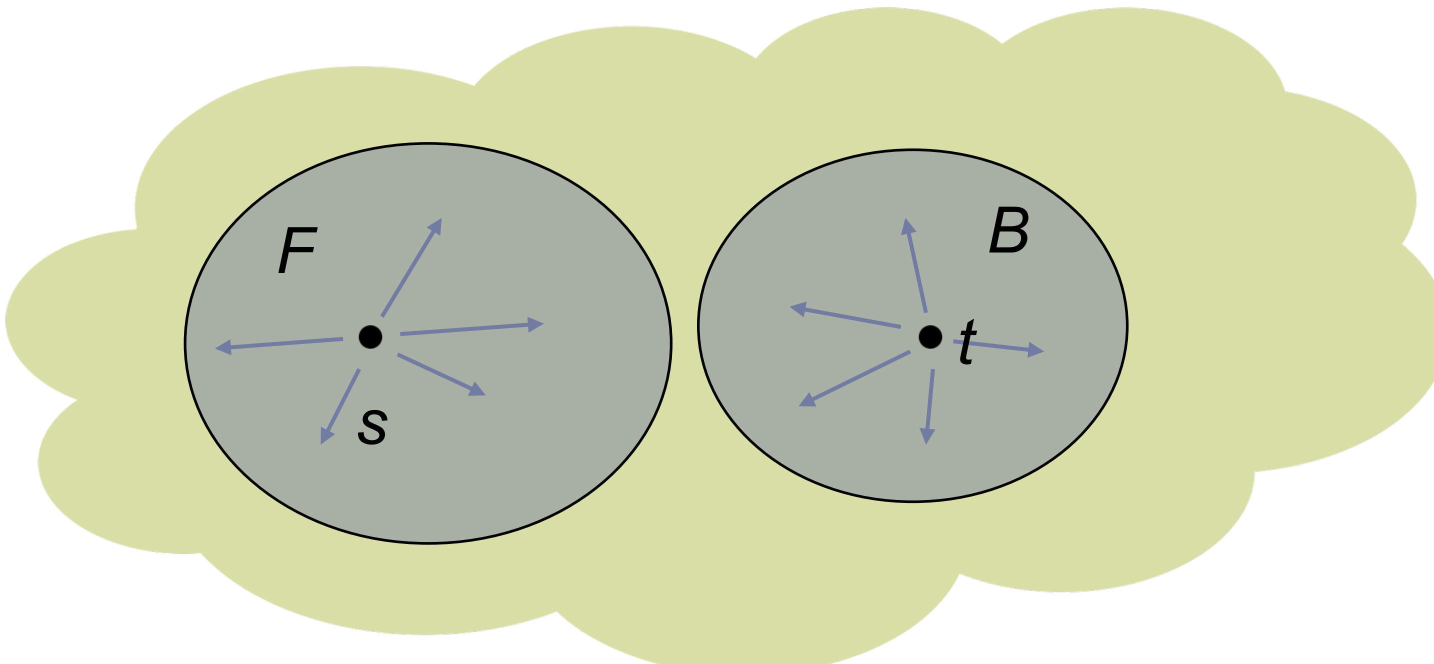
- Assume all edges have non-negative weight. How to search for a shortest path from s to t with Dijkstra's algorithm?



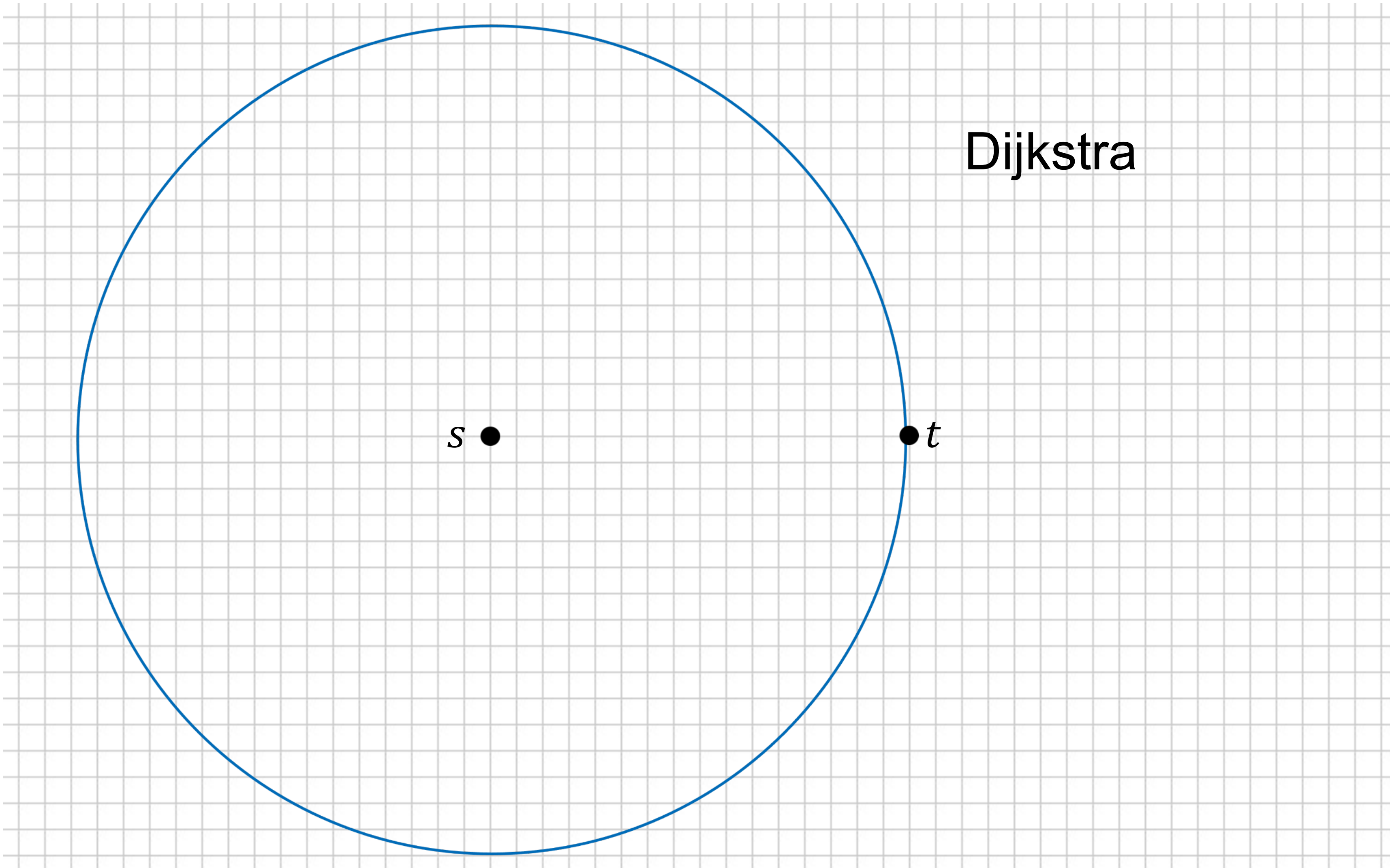
Early exit: Run Dijkstra's algorithm starting from s. Once t is extracted from Q, stop.

Better idea: bidirectional search

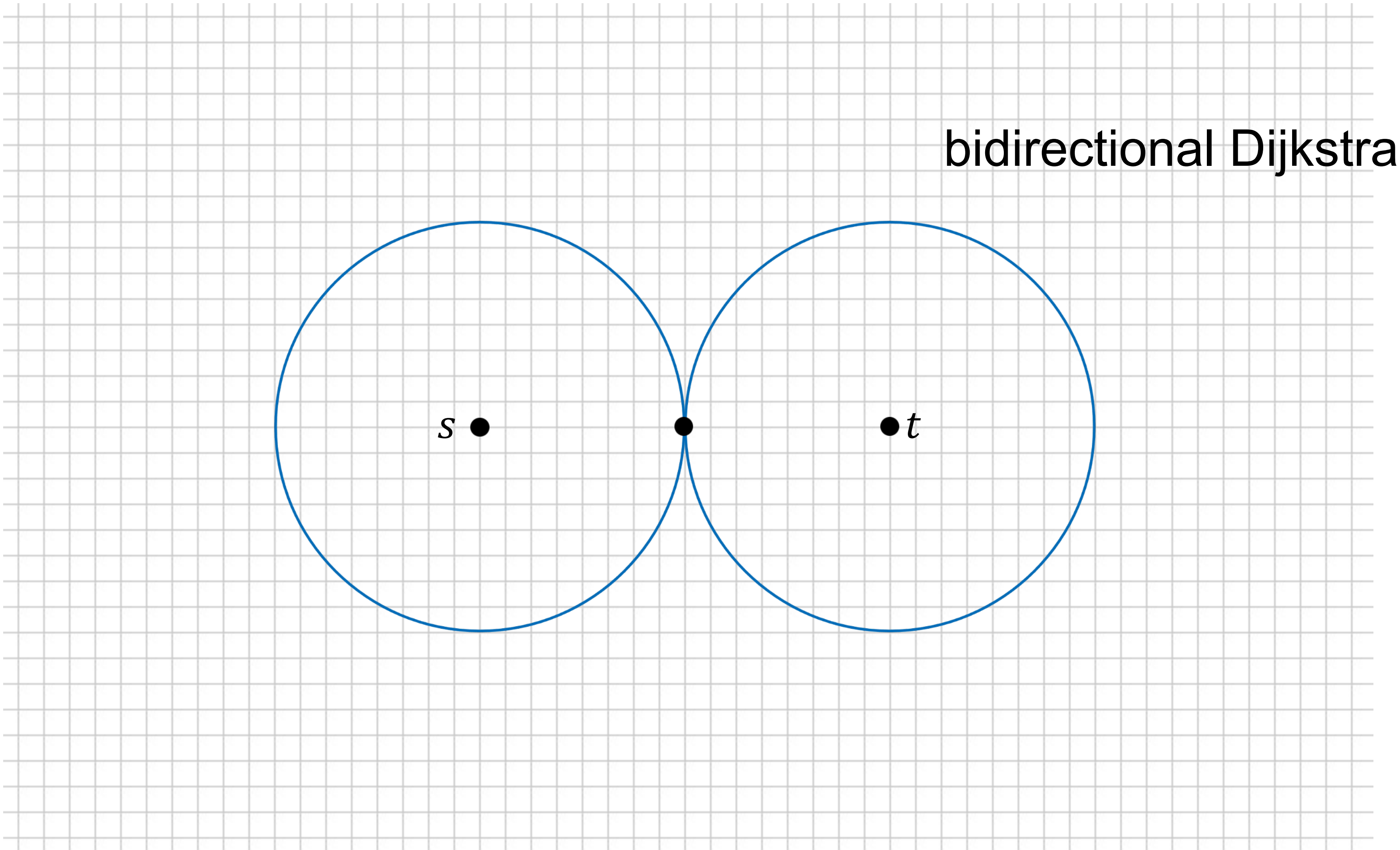
- Bidirectional search (idea): perform Dijkstra on G starting from s and on the reverse graph G^R starting from t. Stop when these searches "meet" (*to be defined*)



Why this is a good idea?

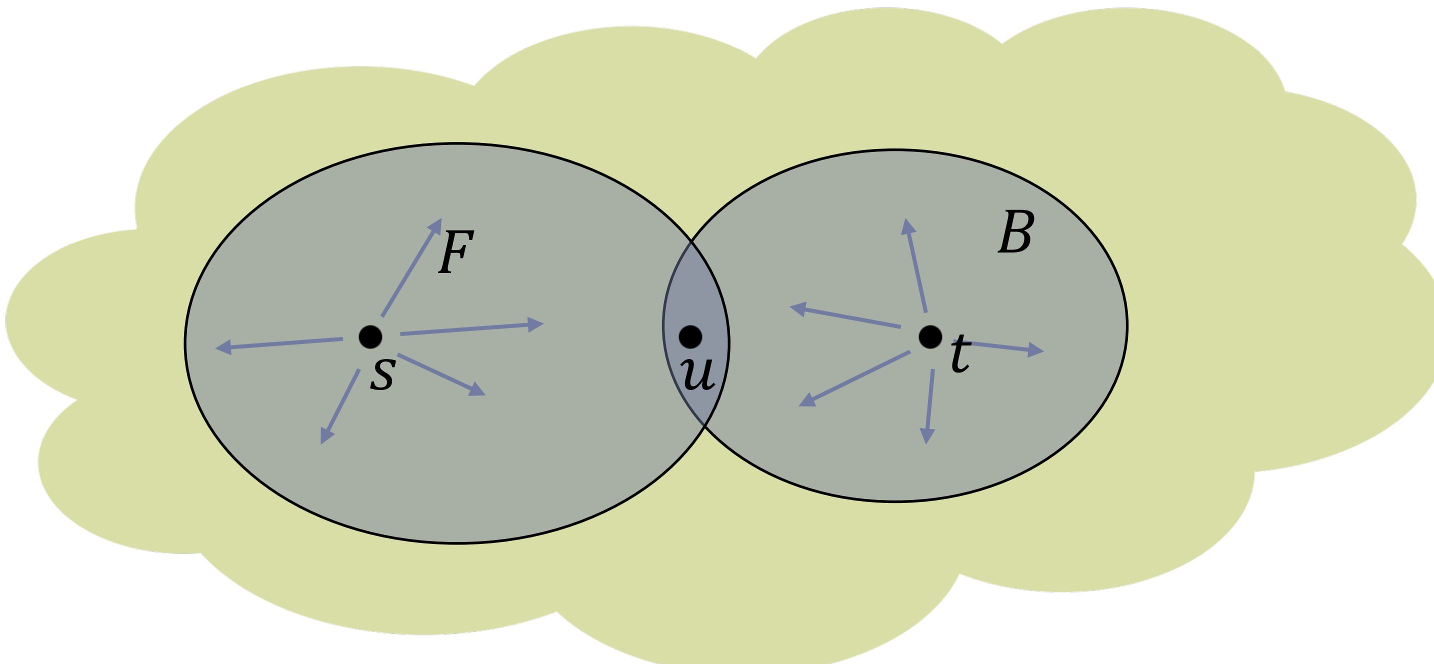


Why this is a good idea?



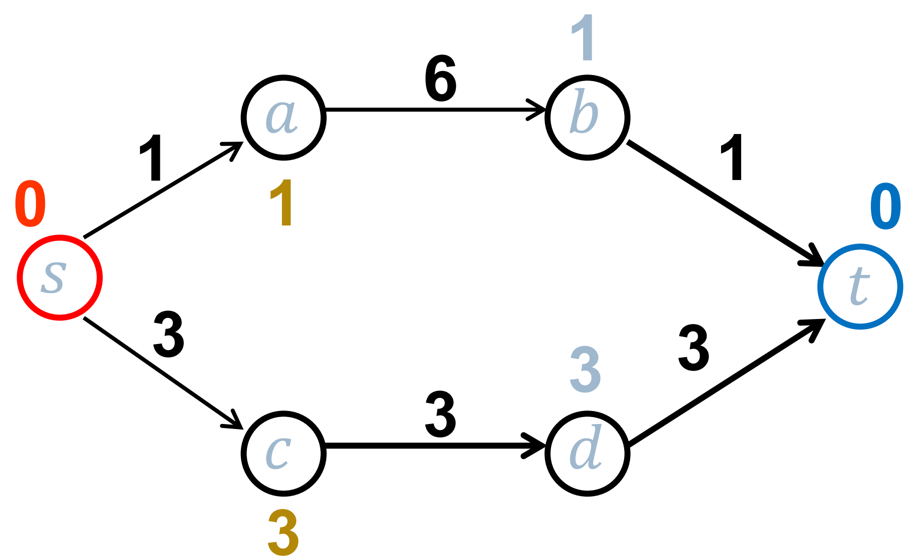
Better idea: bidirectional search

- Bidirectional search (idea): perform Dijkstra on G starting from s and on the reverse graph G^R starting from t. Stop when these searches "meet" (*to be defined*)



- *Catch*: if u is the first occurred node from $F \cap B$, the shortest path from s to t may not pass through u !

Counter-example



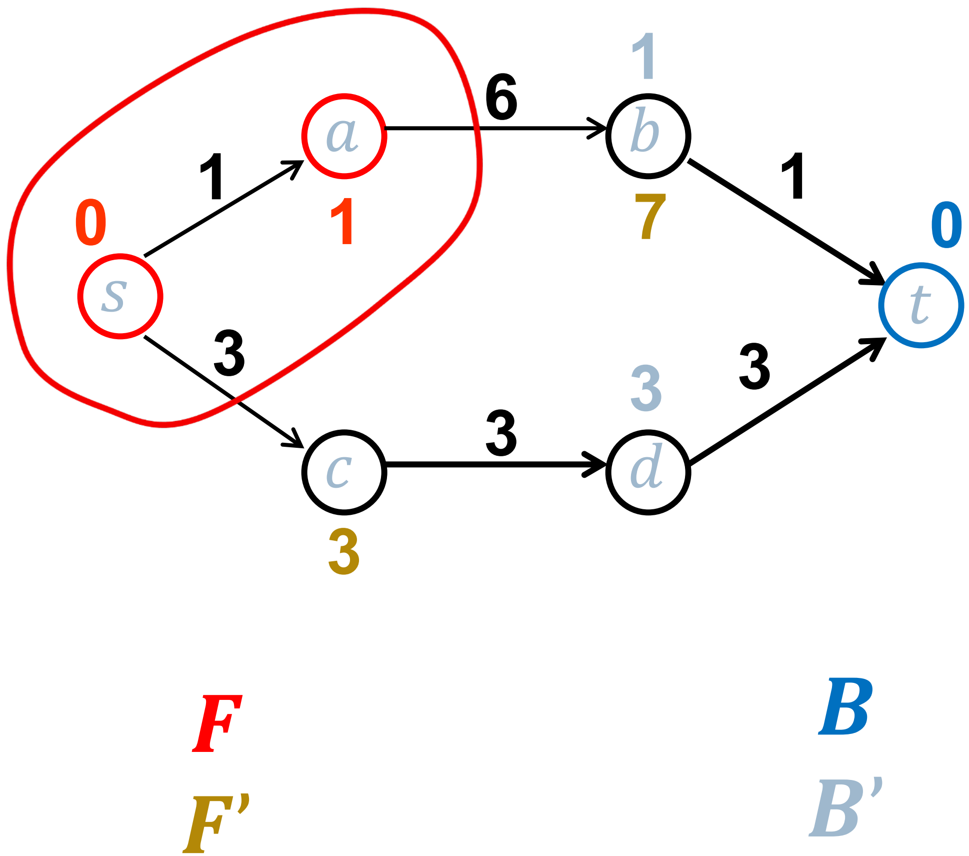
$\boldsymbol{F}$

$\boldsymbol{F'}$

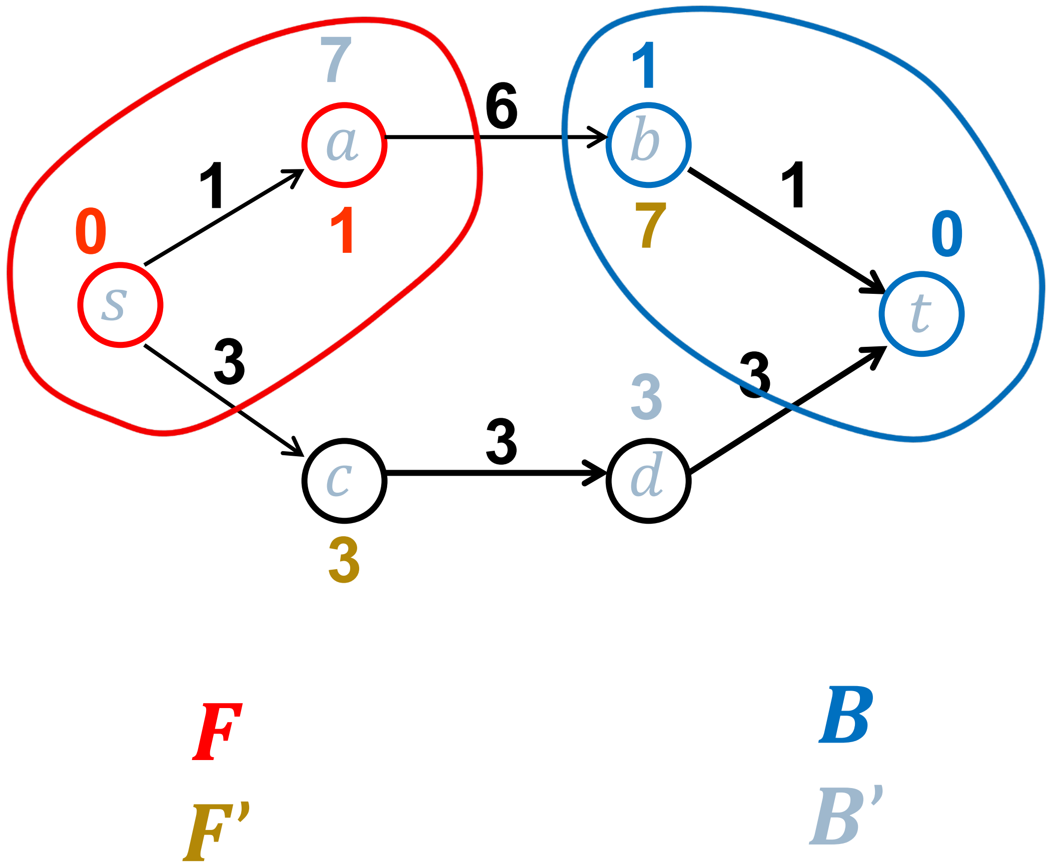
$\boldsymbol{B}$

$\boldsymbol{B'}$

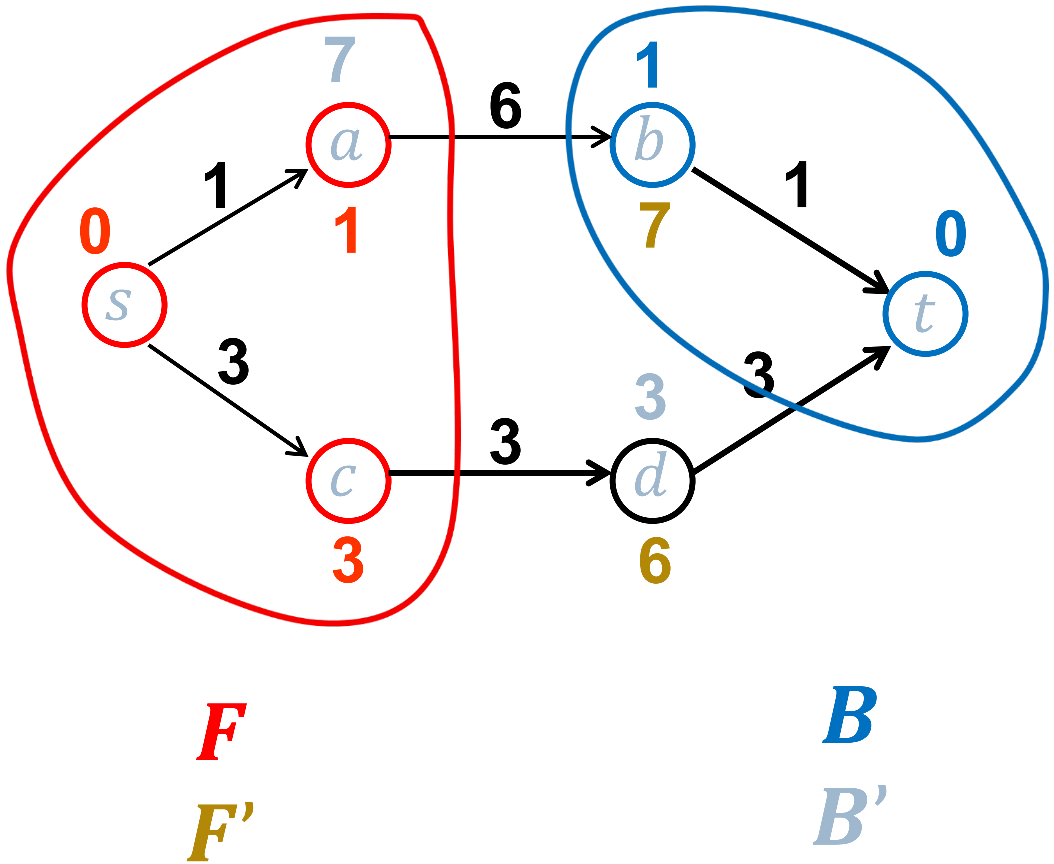
Counter-example



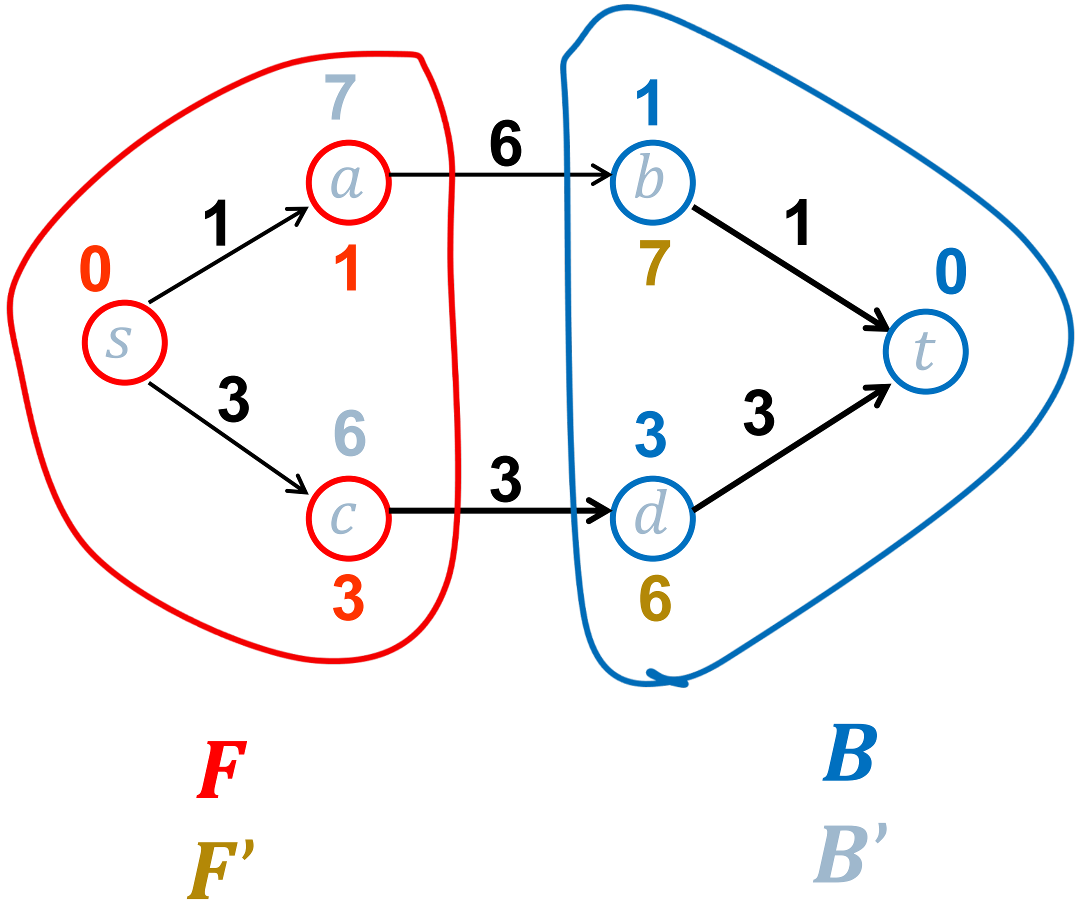
Counter-example



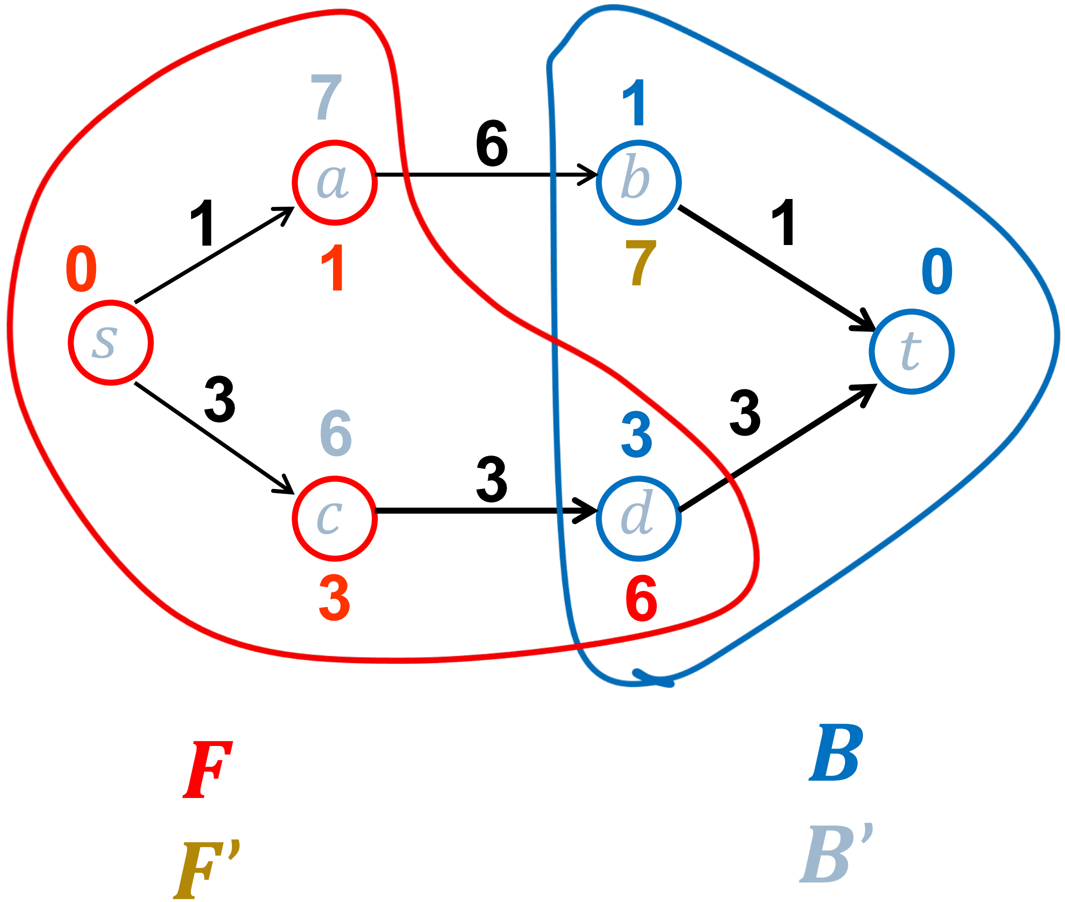
Counter-example



Counter-example



Counter-example

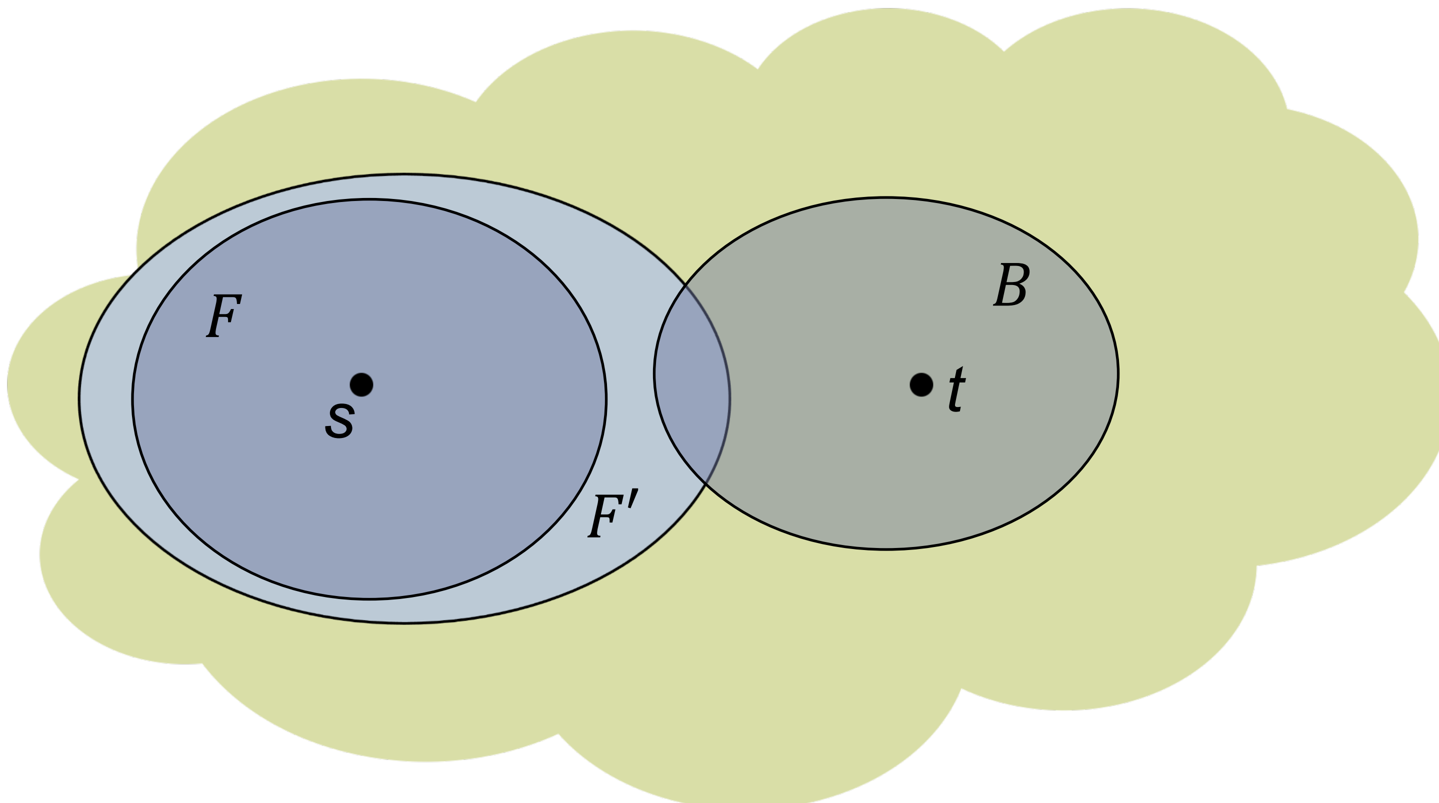


Correct stopping strategy

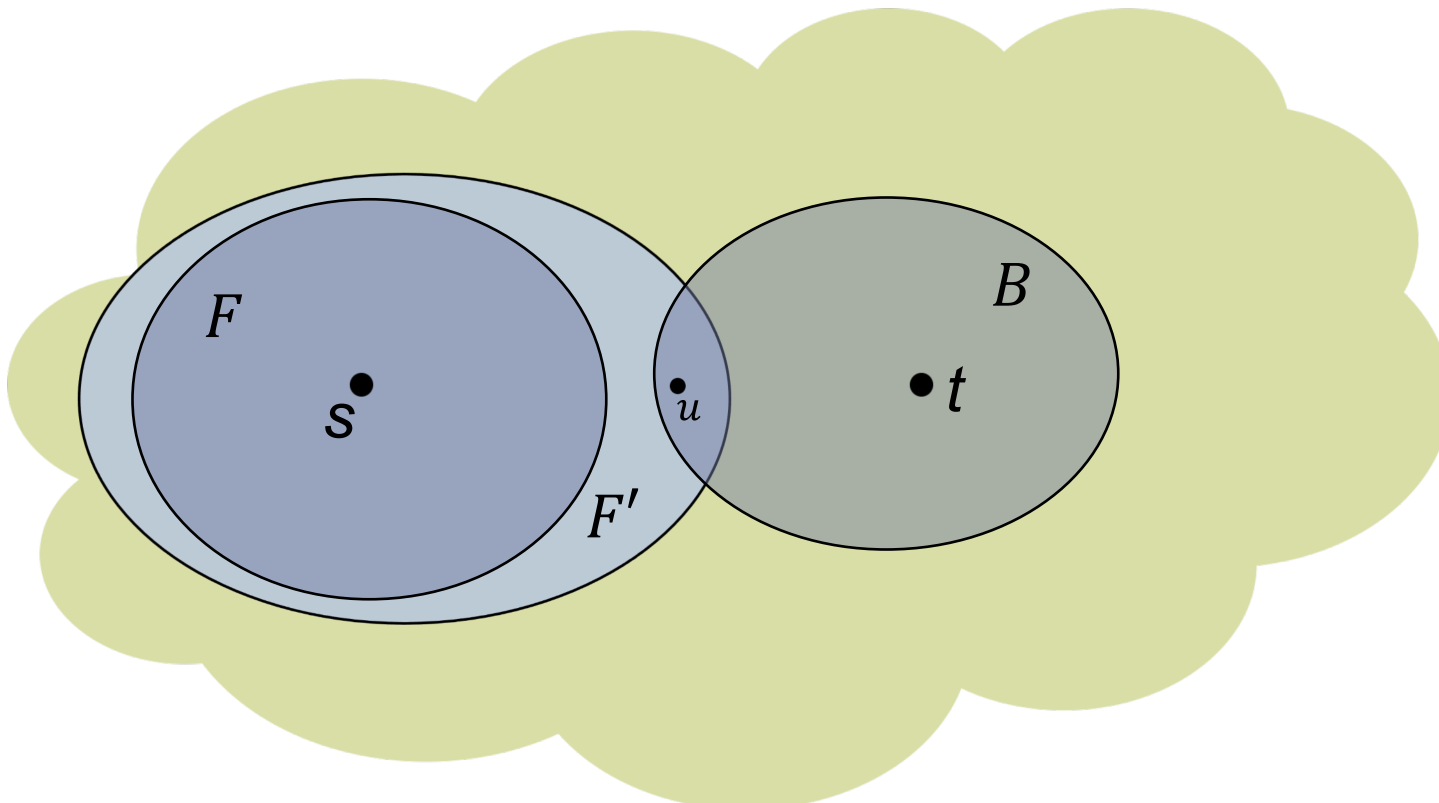
1. assume (by symmetry) that the forward search extracted from F' a node $u \in B$ (i.e. forward and backward searches "met")
2. we should check all other nodes v from $F' \cap B$ and choose the one which minimizes $(d_f[v] + d_b[u])$

Proof: by contradiction

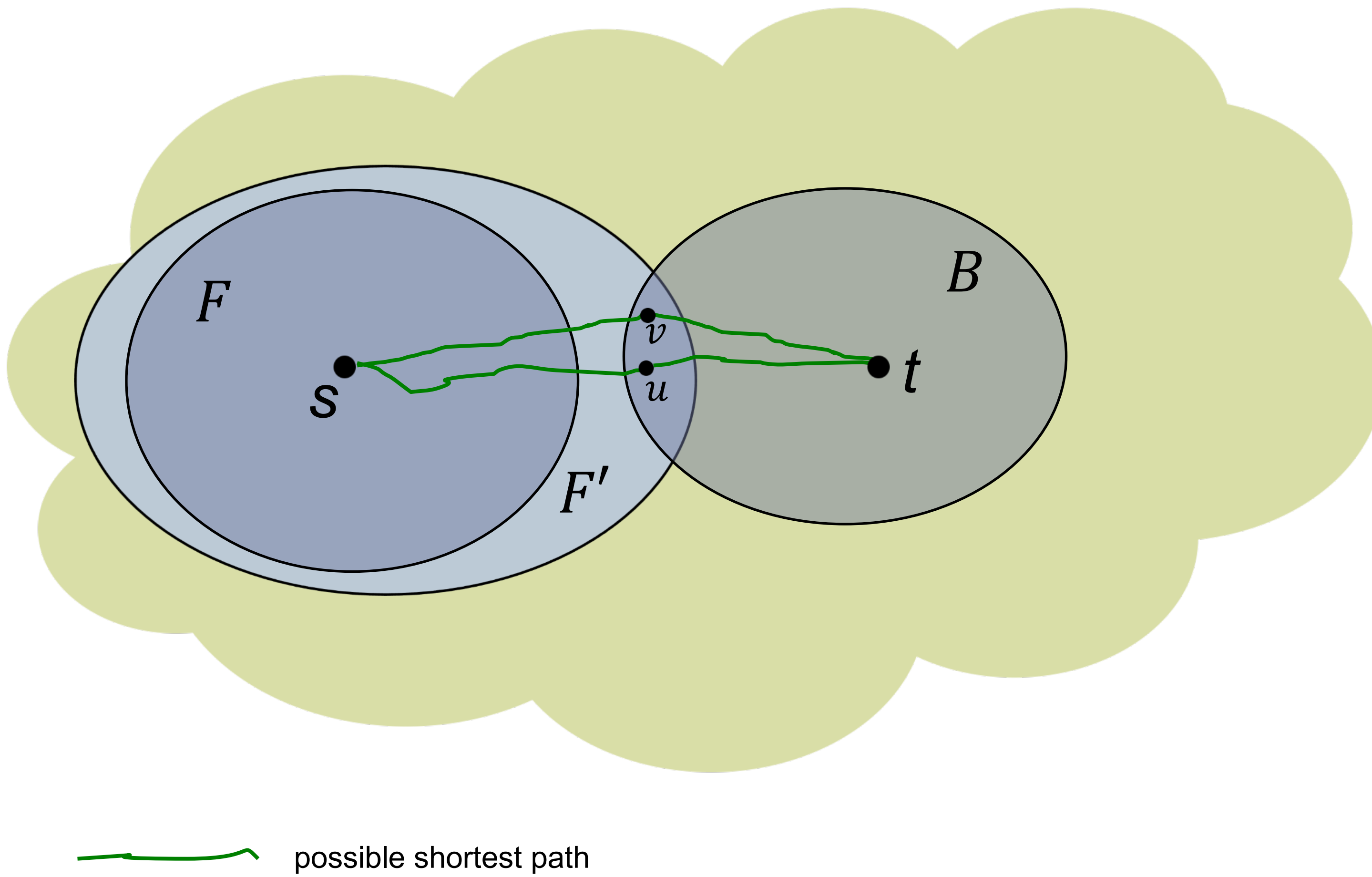
Why this works?



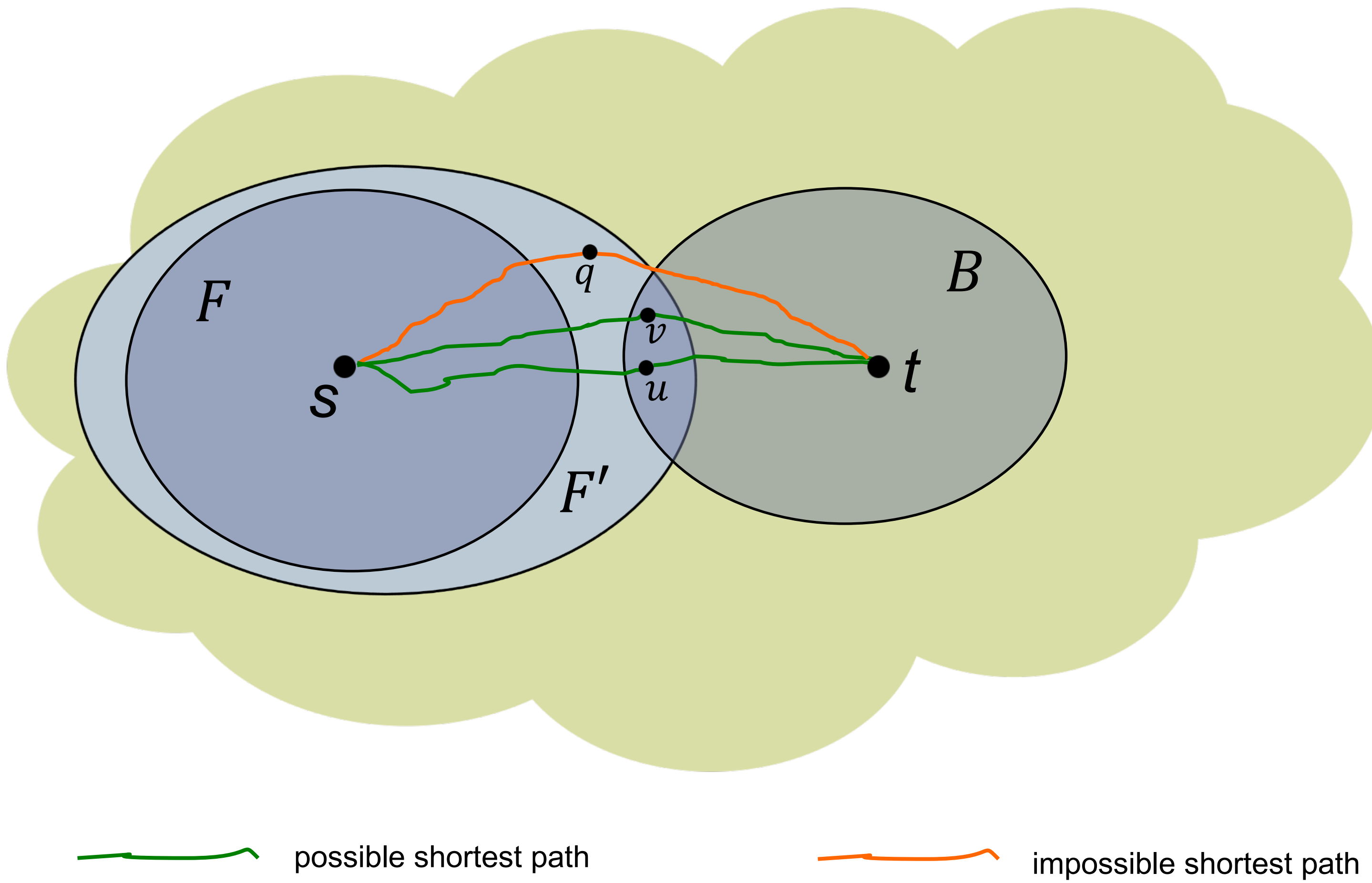
Why this works?



Why this works?



Why this works?



Proof (sketch): u has the smallest d-value in the queue of F, then $\delta(s,u) \leq \delta(s,q)$; since q is not in B, then $\delta(q,t) \geq \delta(u,t) \Rightarrow$ orange path cannot be better than a green one

To sum up

- **Breadth-first search** explores *the whole graph* and finds shortest paths *to all nodes* under assumption that all moves have equal cost. It uses a *queue*.
- **Dijkstra's algorithm** explores *the whole graph* and finds shortest paths *to all nodes* taking into account different move costs. It uses a *priority queue*
- **Bidirectional search** solves point-to-point shortest path problem by running two Dijkstra's

Heuristics for point-to-point search

- **(Greedy) Best-first** search finds a path to *a target node* by exploring the frontier nodes that are estimated to be closer to the target
  - $h(v)$: lower bound of min distance from v to target
  - Strategy: select v minimizing $h(v)$
  - Example: https://www.youtube.com/watch?v=TdHbO3w68fY

Heuristics for point-to-point search

- **(Greedy) Best-first** search finds a path to *a target node* by exploring the frontier nodes that are estimated to be closer to the target
  - $h(v)$: lower bound of min distance from v to target
  - Strategy: select v minimizing $h(v)$
  - Example: https://www.youtube.com/watch?v=TdHbO3w68fY
- **A*** search finds a path to *a target node* by exploring the frontier nodes that have the minimum sum of distance *from* the source and estimated distance *to* the target
  - $f(v)$: computed distance from the source
  - Strategy: select v minimizing $(f(v) + h(v))$
  - Examples: http://www.redblobgames.com/pathfinding/a-star/introduction.html

Does A* compute the optimal solution?

- YES if
  - the branching degree is finite
  - arc costs are strictly positive
  - $h(v)$ is a non-negative *underestimate* of the min distance from v to the target

- Moreover, A* is optimally efficient, that is it expands the minimum number of paths among all algorithms using the same function $h(.)$

Example: 15 puzzle

https://medium.com/@prestonbjensen/solving-the-15-puzzle-e7e60a3d9782

- $\sim 10^{13}$ distinct states, exploring the tree of possible moves leads to $\sim 10^{38}$ states
- possible functions h for best-first search:
  1. number of tiles in incorrect positions
  2. sum of Manhattan distances (absolute horizontal distance + absolute vertical distance) of every tile to its correct location

Example: 15 puzzle

https://medium.com/@prestonbjensen/solving-the-15-puzzle-e7e60a3d9782

- $\sim 10^{13}$ distinct states, exploring the tree of possible moves leads to $\sim 10^{38}$ states
- possible functions h for best-first search:
  1. number of tiles in incorrect positions
  2. sum of Manhattan distances (absolute horizontal distance + absolute vertical distance) of every tile to its correct location
- second is better than first

Solution Length		
	Manhattan	Number Wrong
mean	10.58	18.22
10th percentile	10	10
50th percentile	10	10
90th percentile	10	36

Explored States		
	Manhattan	Number Wrong
mean	27.71	580.1
10th percentile	11	11
50th percentile	11	14
90th percentile	28	1076

Example: 15 puzzle (cont)

https://medium.com/@prestonbjensen/solving-the-15-puzzle-e7e60a3d9782

- A*: $g(v) + h(v)$ where
  - $g(x)$: number of moves to state x
  - $h(v)$: sum of Manhattan distances (as before)
- best-first: $h(v)$ only

- A* is better than best-first

Solution Lengths		
	A*	Pure Heuristic
mean	22	59.66
10th percentile	17	23
50th percentile	23	52
90th percentile	25	111

Explored States		
	A*	Pure Heuristic
mean	755.87	1240.35
10th percentile	71.1	45.8
50th percentile	350.5	664.5
90th percentile	1738.2	3498.1

If you want to know more …

- more on heuristic search: *Pearl, J. Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley, 1984*