[1]Dijkstra-like algorithm: Relaxation of the Dijkstra choice rule (1976)
[2]Hybrid Bellman-Ford-Dijkstra Algorithm (2010)

[1,2]Yefim Dinitz [2]Rotem Itzhak

Ben-Gurion University

Outline

1. Introduction
2. Dijkstra-like Algorithm
3. Bellman-Ford-Dijkstra Algorithm

The single source cheapest paths problem

Problem Definition

Given digraph with edge costs $G = (V, E, c)$ and source $s \in V$, we would like to find the cheapest paths from s to all $v \in V$.

Motivation

- GPS: Finding shortest paths from current location.
- For determining if locations are reachable.

Reminder: Dijkstra Algorithm (1959)

- Works for the non-negative edge costs case.
- A search type algorithm: it makes just a single pass on all vertices and edges reachable from s, in a greedy order.
- Notations:
  1. $d(v)$: denotes the weight of the minimal path known so far.
  2. $\pi(v)$: a back-pointer, points to v's father in the minimal path known so far.
  3. $\delta(v)$: the weight of the cheapest path from s to v. $\delta(v) = \infty$ if v is not reachable from s.
- **Running time:** $O(|E| + |V| \log |V|)$, when using Fibonacci heap.

Reminder: Dijkstra Algorithm (1959)

Algorithm 1 Initialization()

1: $d(v) \leftarrow \infty$, for all $v \in V$
2: $\pi(v) \leftarrow null$, for all $v \in V$
3: d(s) $\leftarrow$ 0

Algorithm 2 Relax(u, v)

1: **if** $d(u) + c(u, v) < d(v)$ **then**
2: d(v) $\leftarrow$ d(u) + c(u, v)
3: π(v) $\leftarrow$ u
4: **end if**

Reminder: Dijkstra Algorithm (1959)

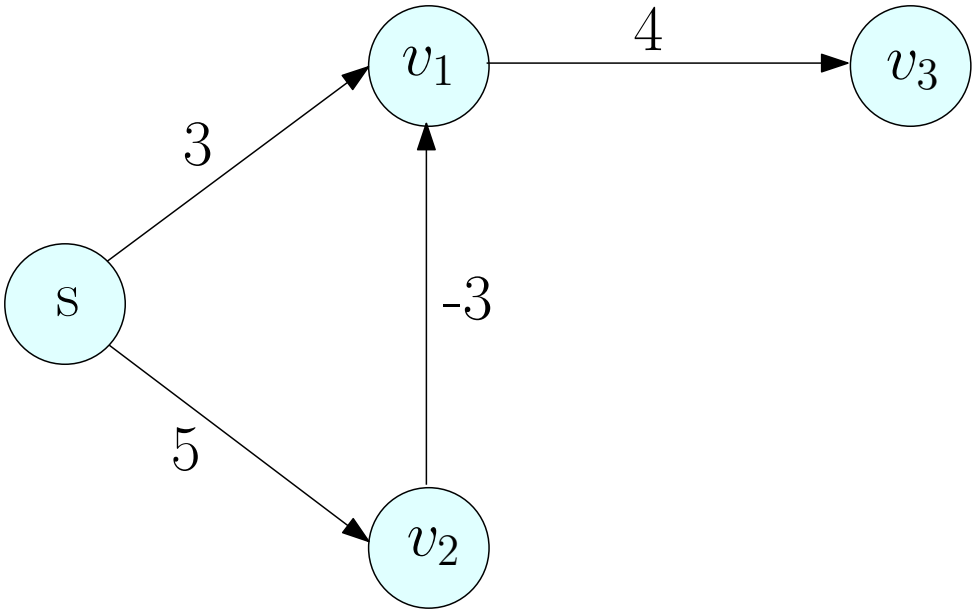
Algorithm 3 Dijkstra-scan()

```
S ← ∅
while there is a vertex in V \ S with d < ∞ do
    find vertex u ∈ V  S with the minimal value of d
    S ← S ∪ {u}
    for each edge (u, v) ∈ E do
        Relax(u, v)
    end for
end while
```

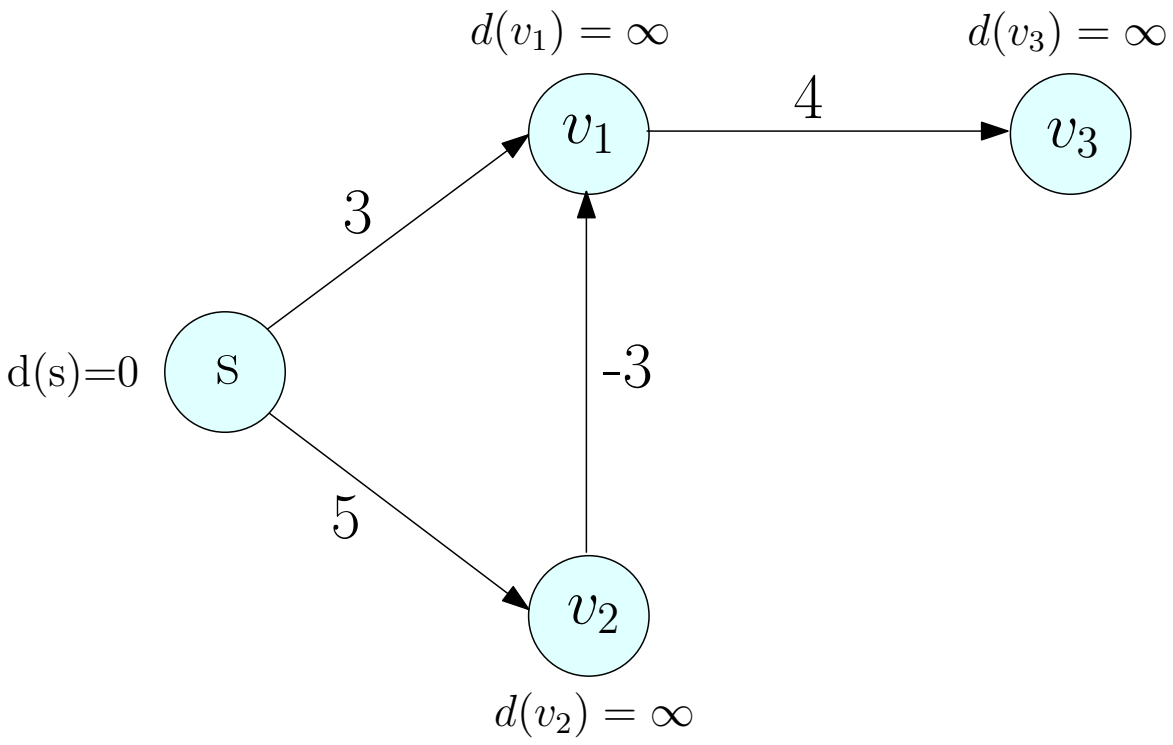
Algorithm 4 Dijkstra(G, s)

```
Initialization()
Dijkstra-scan()
return (d, π)
```

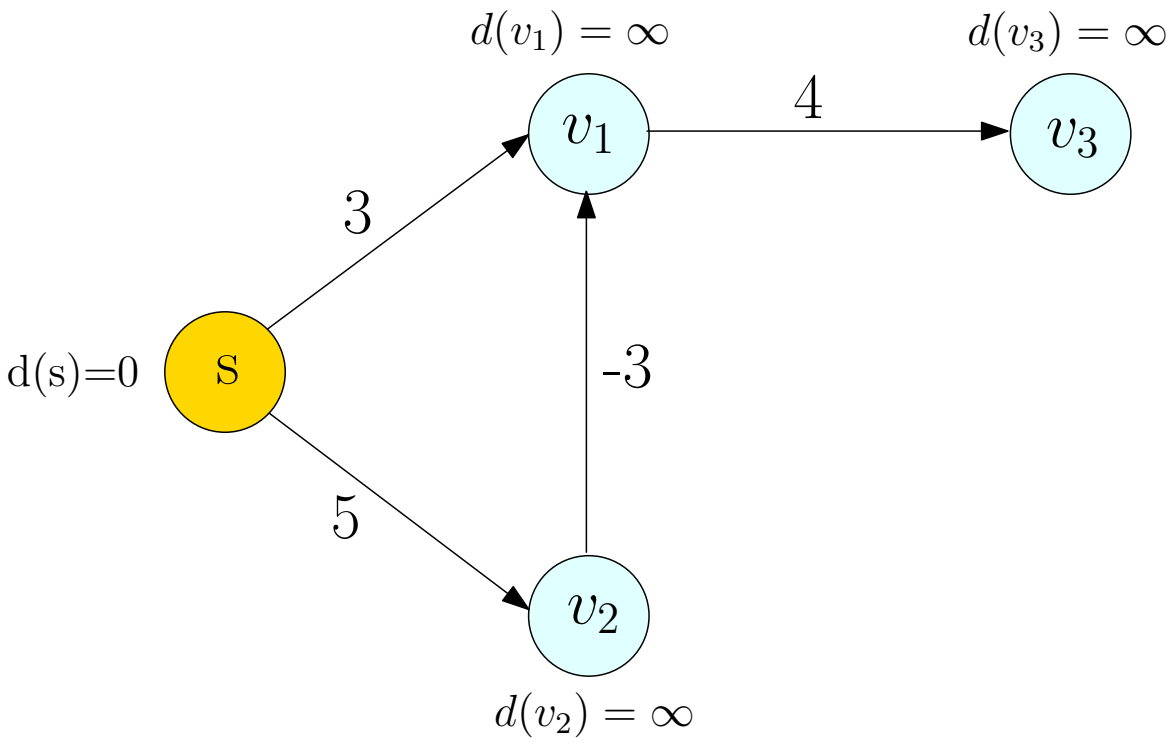
Reminder: Dijkstra Algorithm (1959)



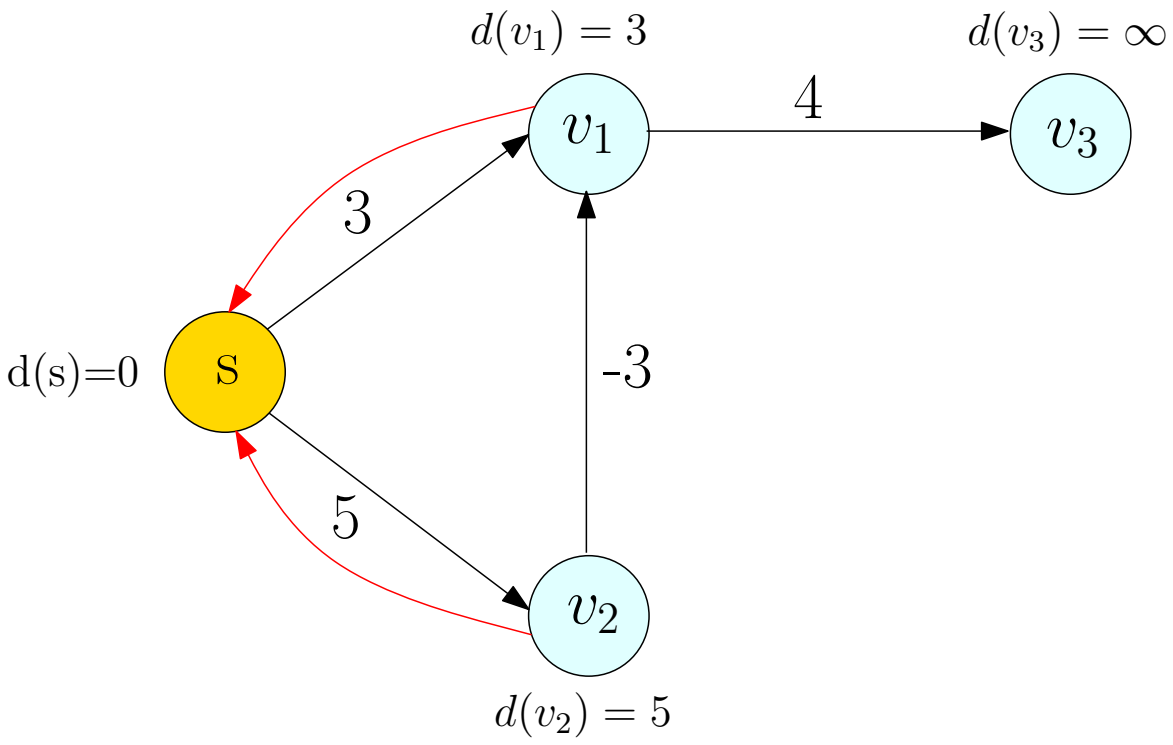
Reminder: Dijkstra Algorithm (1959)



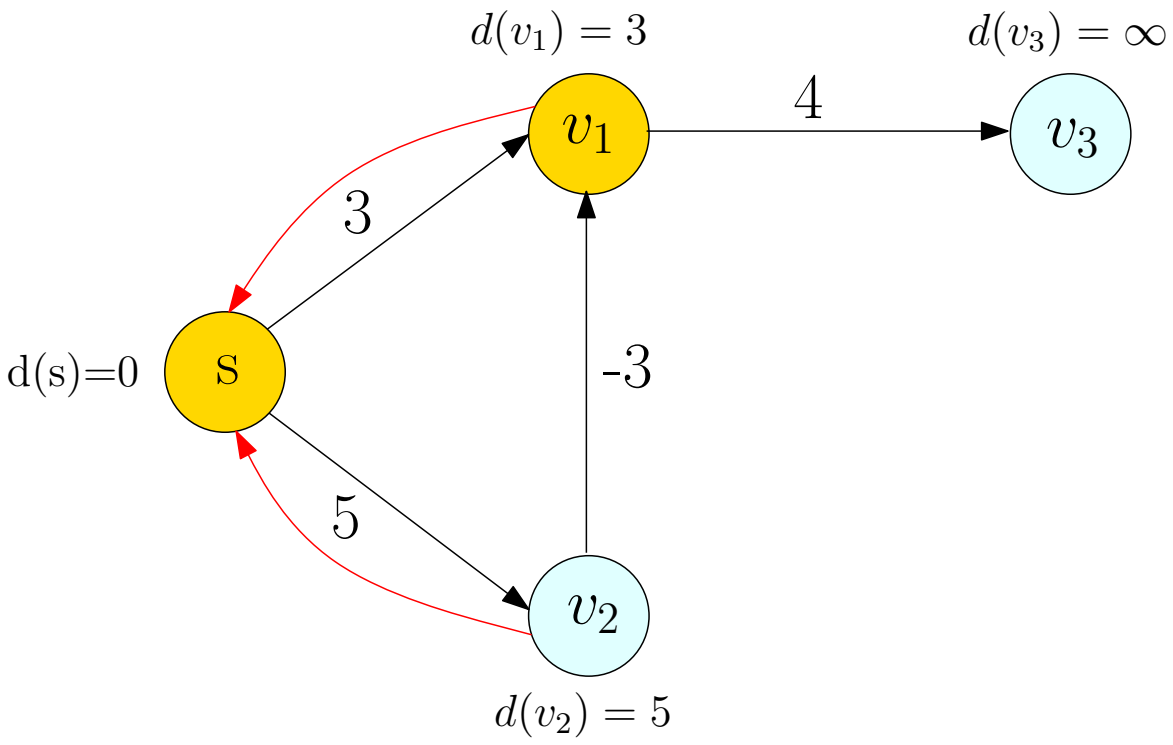
Reminder: Dijkstra Algorithm (1959)



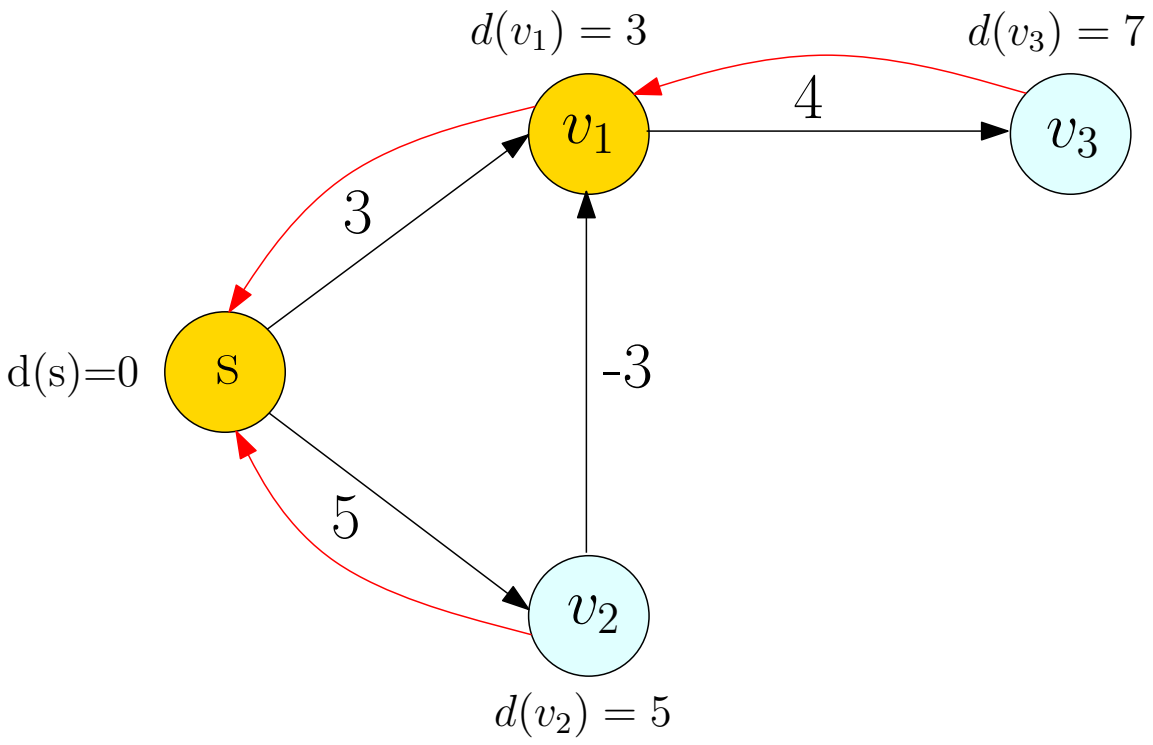
Reminder: Dijkstra Algorithm (1959)



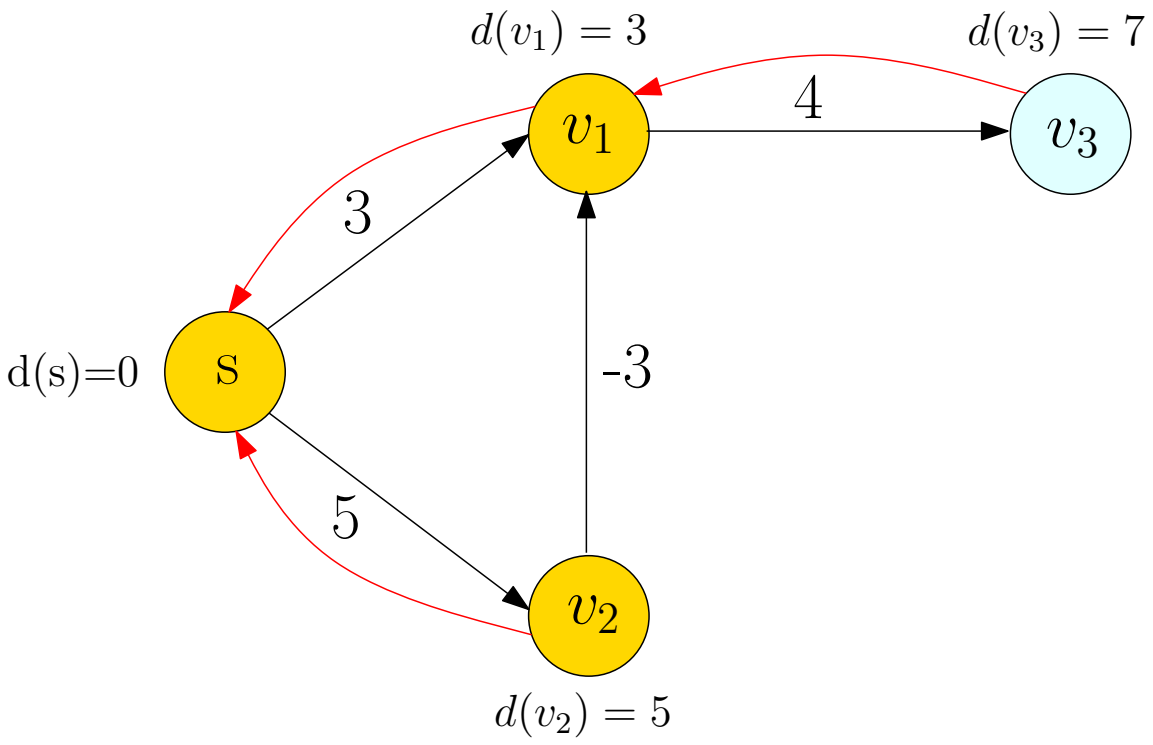
Reminder: Dijkstra Algorithm (1959)



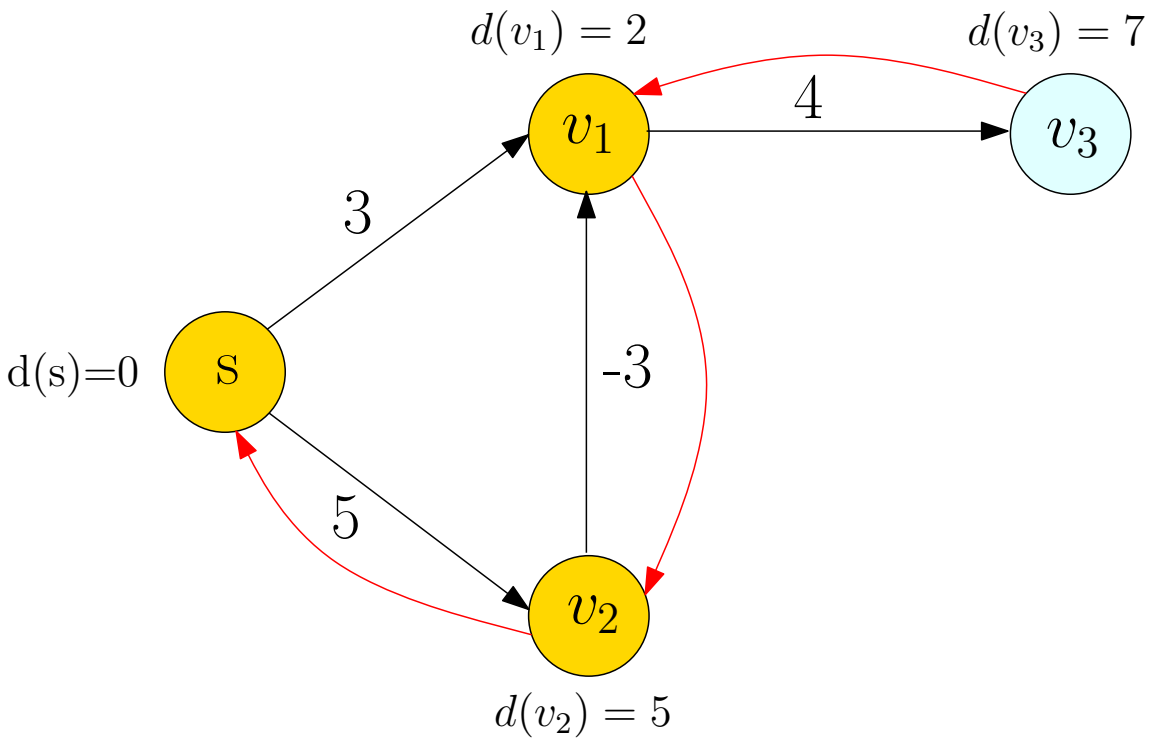
Reminder: Dijkstra Algorithm (1959)



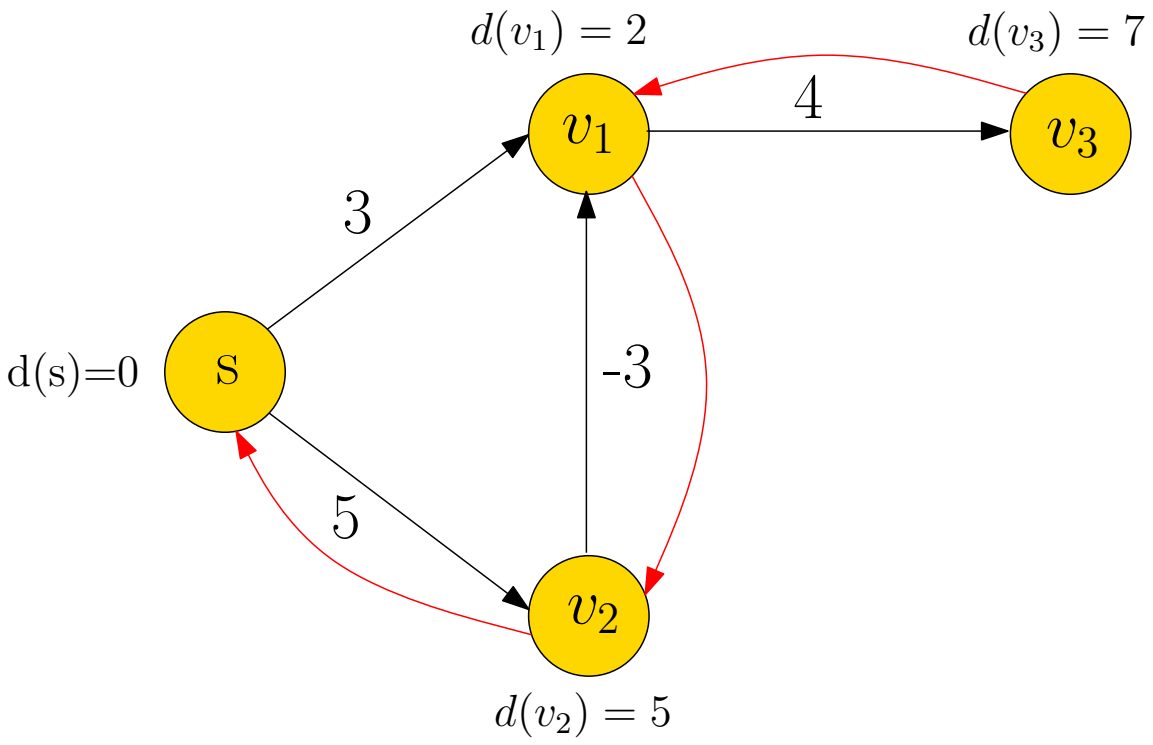
Reminder: Dijkstra Algorithm (1959)



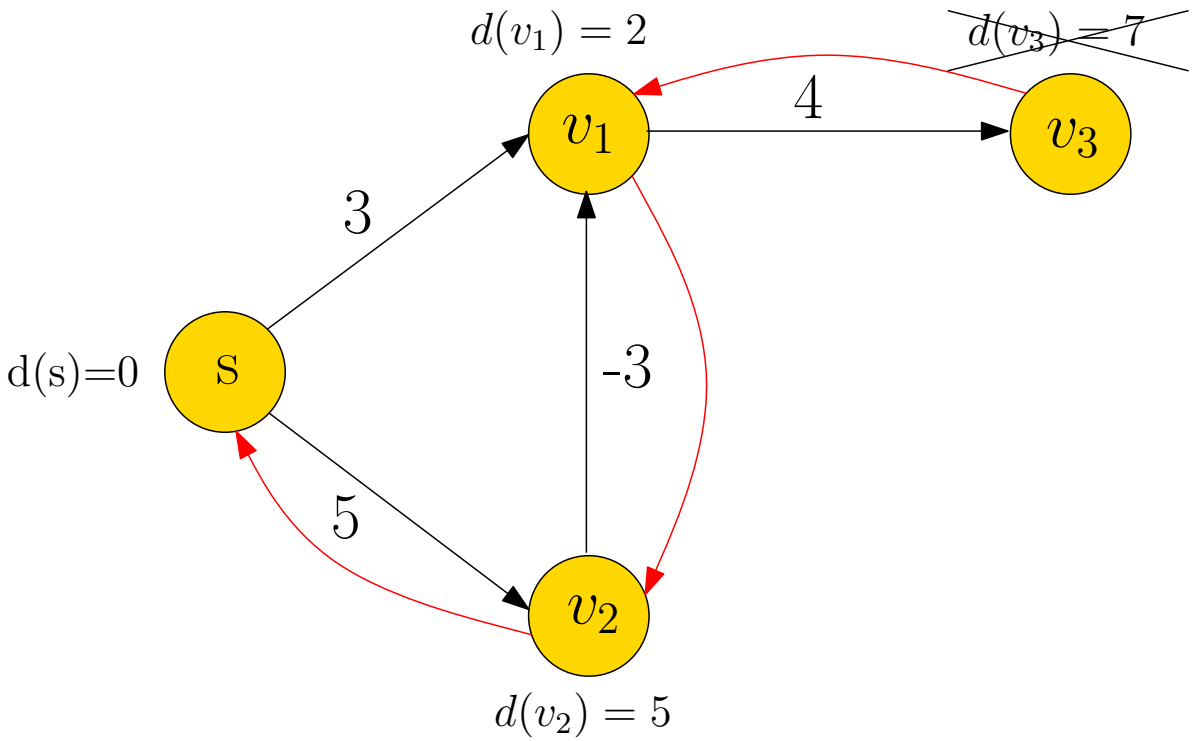
Reminder: Dijkstra Algorithm (1959)



Reminder: Dijkstra Algorithm (1959)



Reminder: Dijkstra Algorithm (1959)



Outline

1 Introduction

2 Dijkstra-like Algorithm

3 Bellman-Ford-Dijkstra Algorithm

Dijkstra-like Algorithm (1976)

We assume that all the costs are positive and denote the minimum cost by C_{min}.
Let u be the next vertex chosen by Dijkstra. Hence, $d(u)$ is minimal, denote it by m.
Recall that $d(v) = min\{d(u) + c(u, v)\}$, among all $u \in S$.

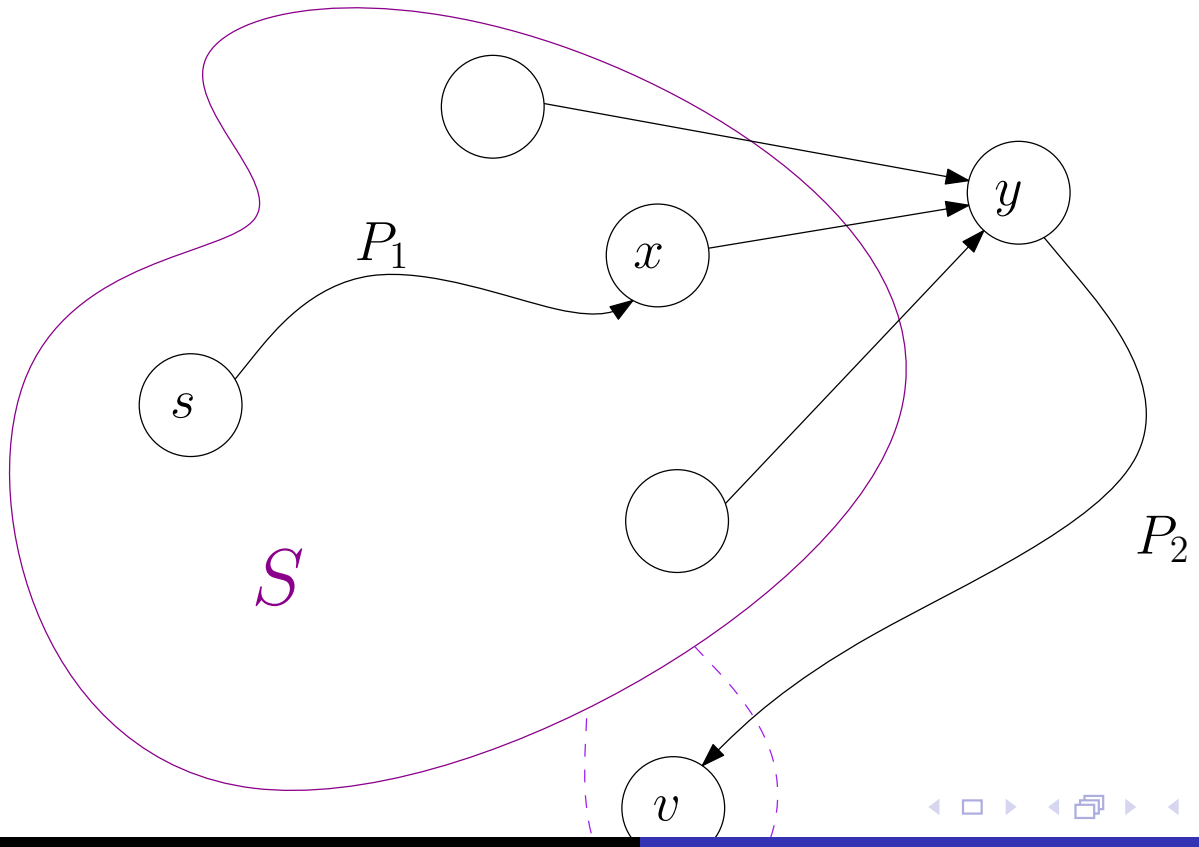
Lemma

If $d(v) \leq m + C_{min}$, then $d(v) = \delta(v)$.

Dijkstra-like Algorithm (1976)

Proof

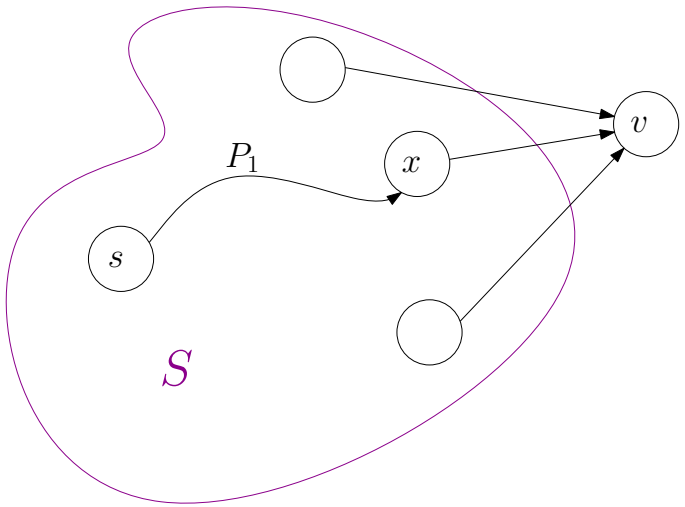
We will show that for each path P, from s to v, holds $d(v) \leq w(P)$.
$P = P_1||(x, y)||P_2$.



Dijkstra-like Algorithm (1976)

Case 1: $y = v$

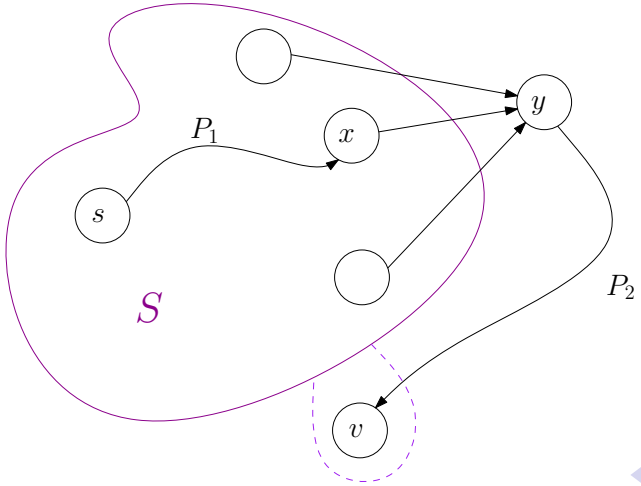
$$\begin{aligned} w(P) &= w(P_1) + w(x, v) \\ &\geq \delta(x) + w(x, v) \\ &= d(x) + w(x, v) \\ &\geq d(v) \end{aligned}$$



Dijkstra-like Algorithm (1976)

Case 2: $y \neq v$

$$
\begin{aligned}
w(P) &= w(P_1) + w(x,y) + w(P_2) \\
&\geq \delta(x) + w(x,y) + w(P_2) \\
&= d(x) + w(x,y) + w(P_2) \\
&\geq d(y) + w(P_2) \geq m + w(P_2) \\
&\geq m + C_{min} \geq d(v)
\end{aligned}
$$



Dijkstra-like Algorithm (1976)

Conclusion

Instead of choosing the vertex with the minimal d value, as in Dijkstra algorithm. We can choose any vertex v, such that $d(v) \leq m + C_{min}$.

At the same paper, Yefim Dinitz is suggesting an implementation of generalized Dijkstra (wich exploits the above conclusion), that works in $O(|E| + \frac{\delta_{max}}{C_{min}})$ time.

Dijkstra-like Algorithm (1976)

Motivation

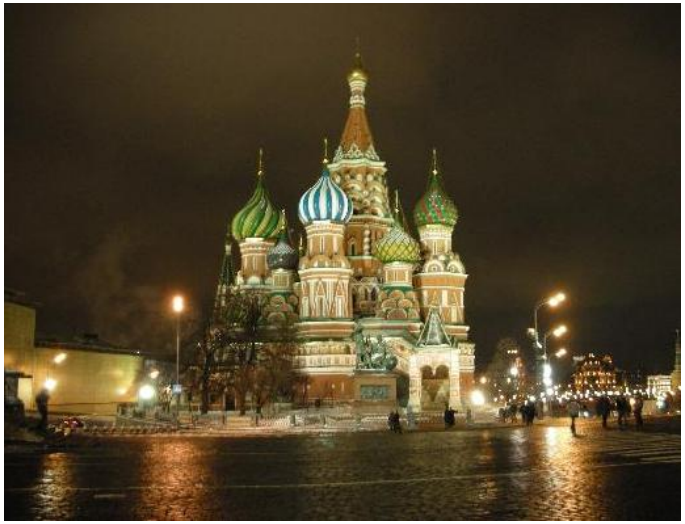
Moscow Map:

$|E| \approx 10,000$. $|V| \approx 4,000$.

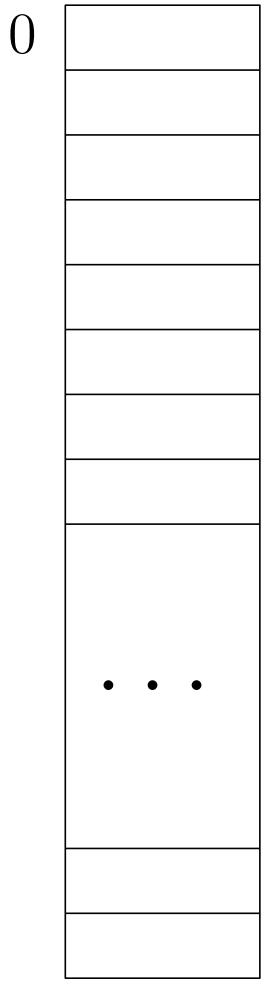
$\Rightarrow |E| \leq 3|V| \Rightarrow |E| = O(|V|)$.

The implementation of fibonacci heap is not recommended due to its big constant and complexity.

Linear and practical algorithm was required.

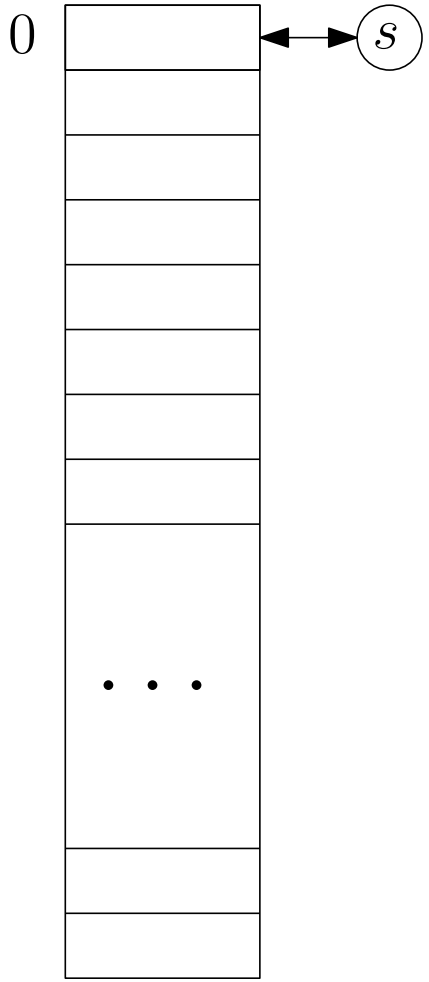


Dijkstra-like Algorithm (1976)



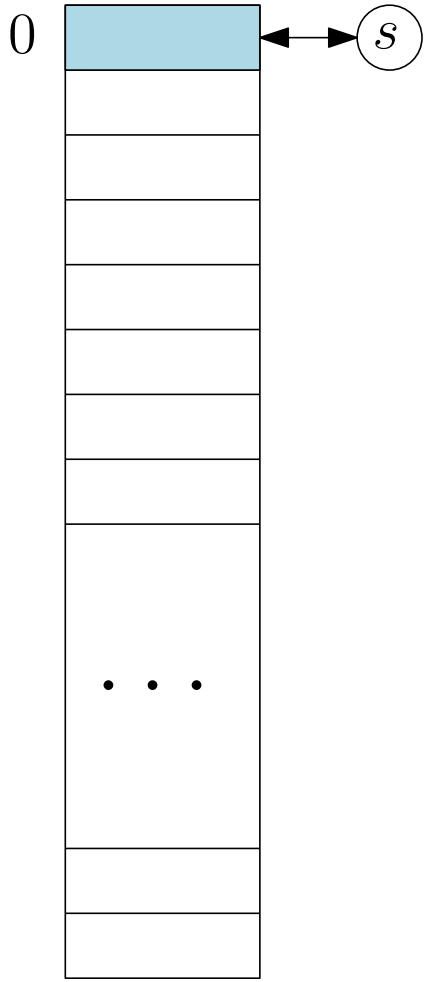
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



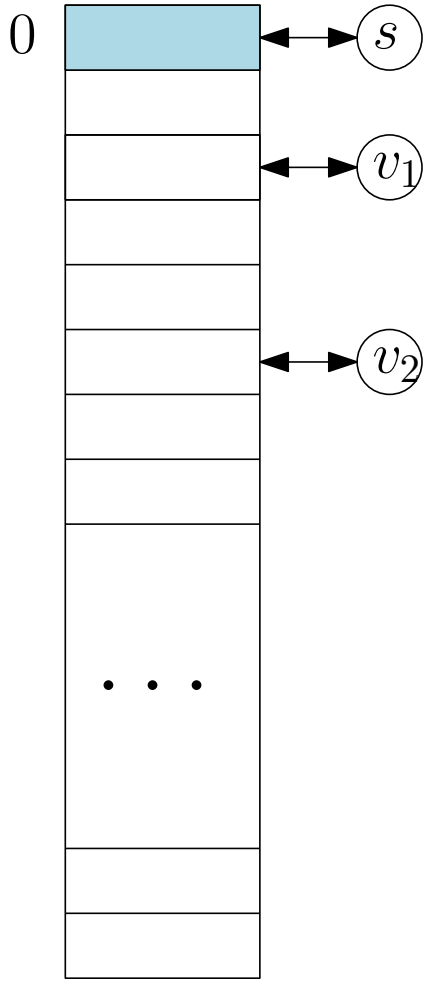
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



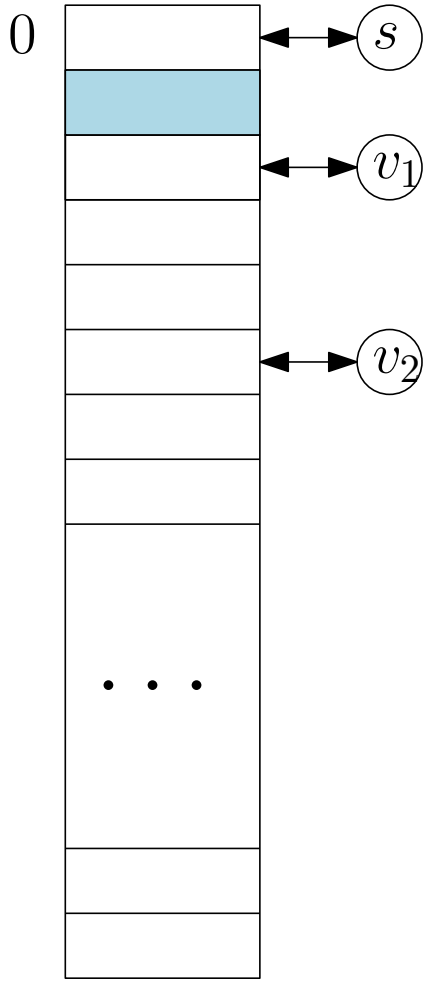
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



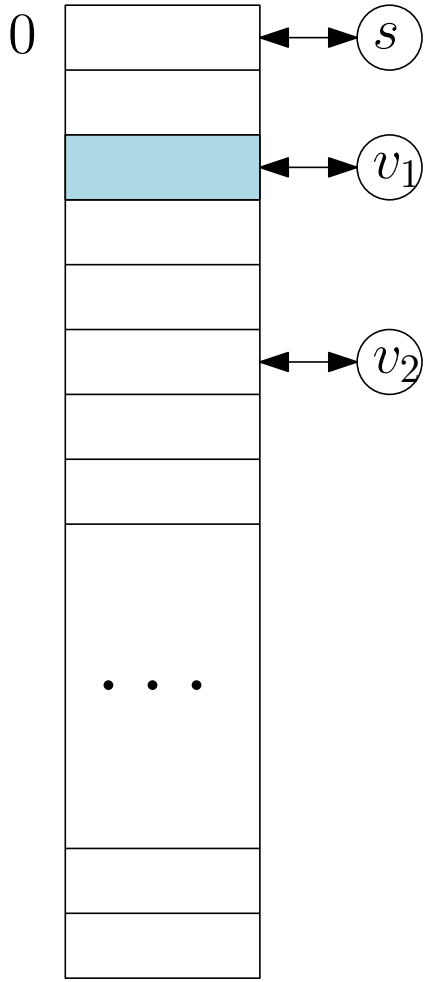
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



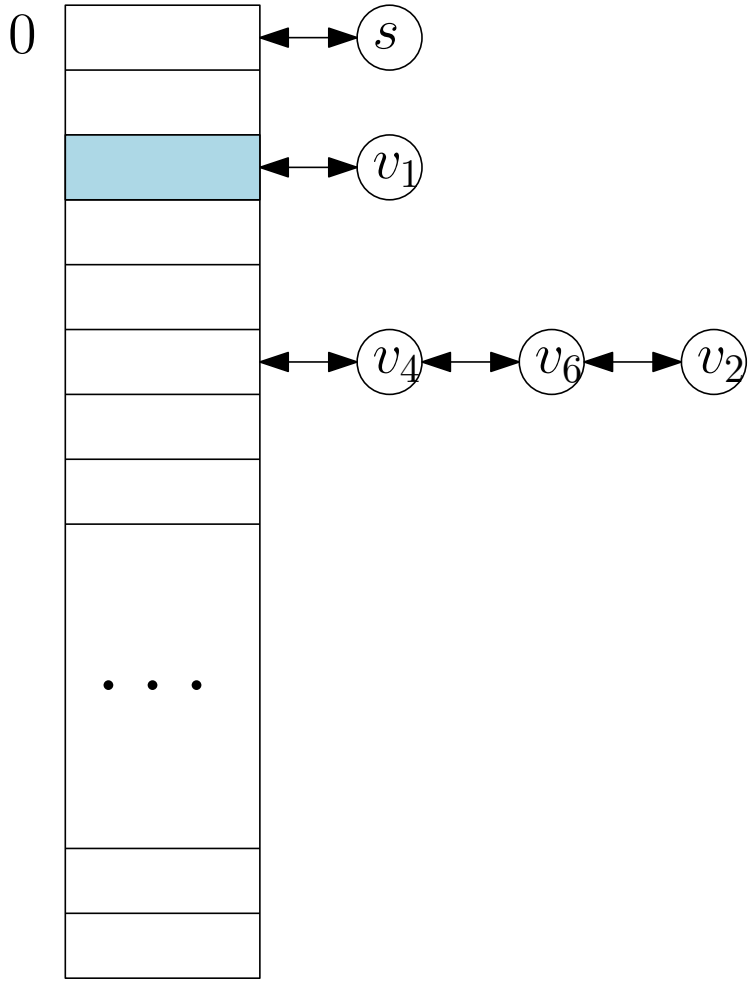
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



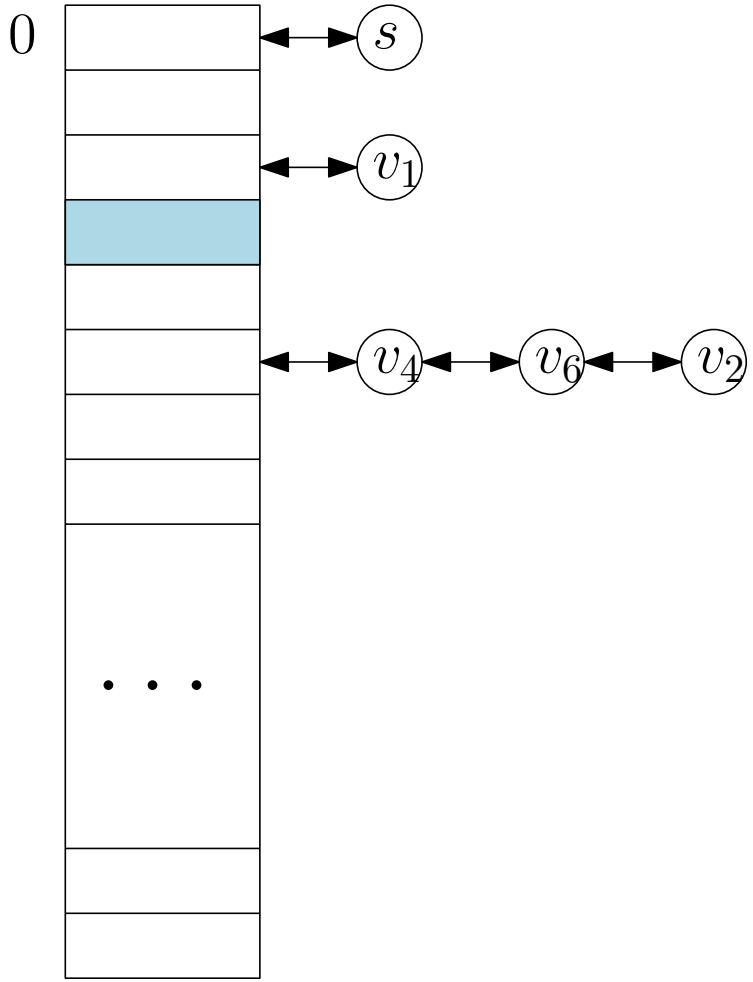
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



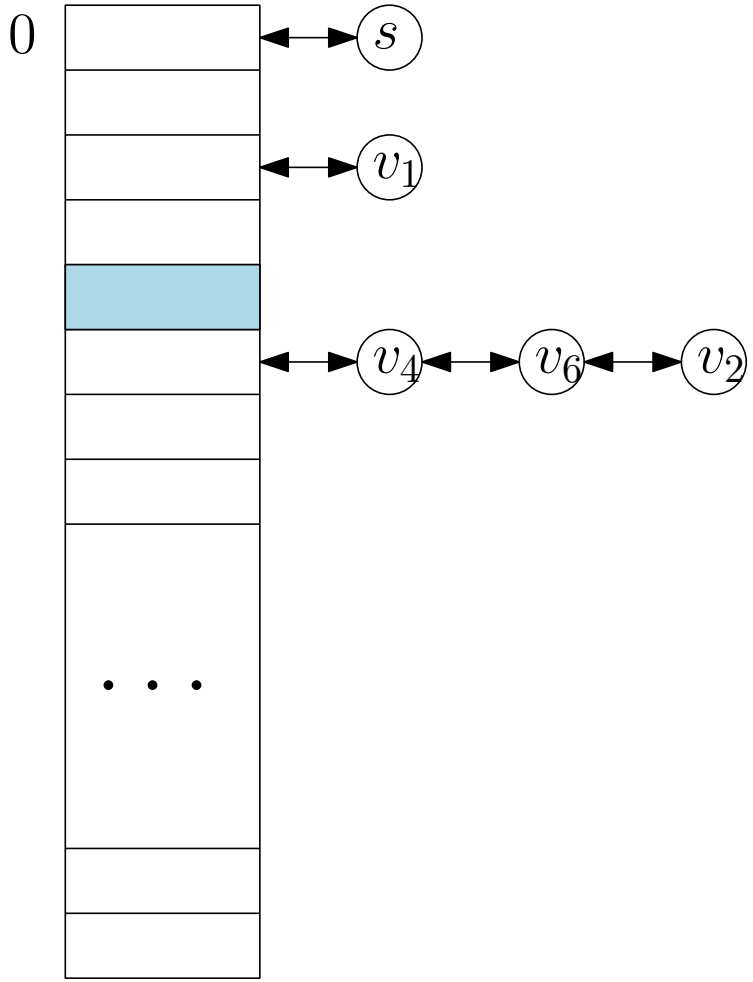
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



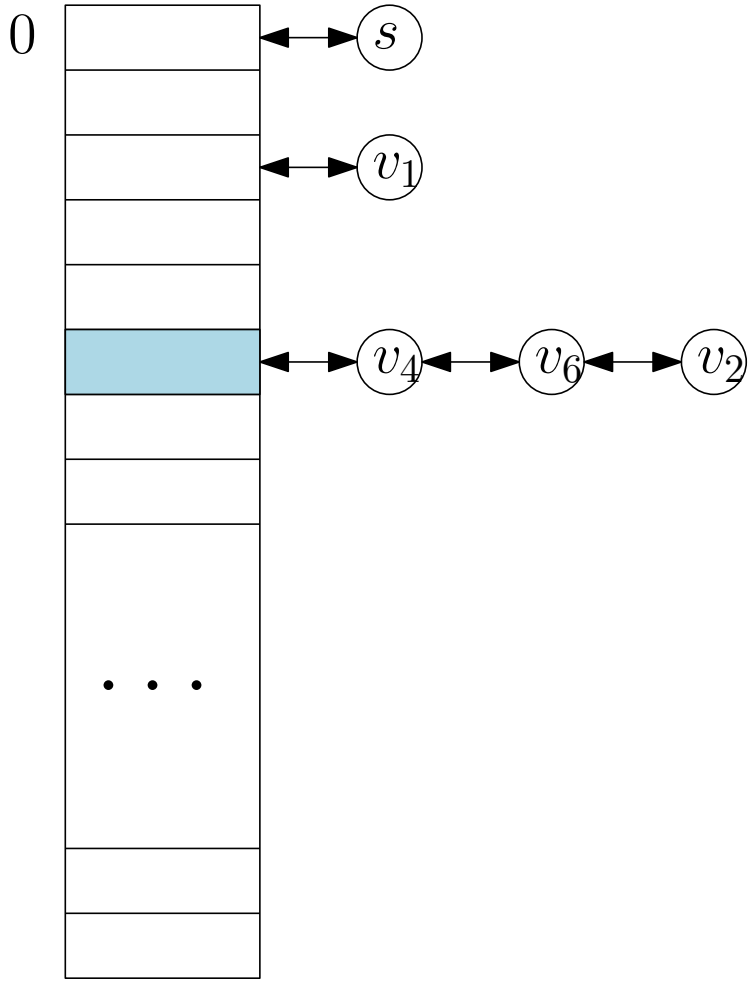
$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)



$N = \frac{D+C_{max}}{C_{min}}$.
where $D \geq \delta_{max}$

Dijkstra-like Algorithm (1976)

Algorithm 5 Dijkstra-like(G, s)

Initialization: create an array of $N = \frac{D+C_{max}}{C_{min}}$ cells (all are initialized by *null*). Insert s to the first cell.
for each cell c_i **do**
  while the cell's linked-list is noy empty **do**
    $u \leftarrow$ extract the first vertex from the linked-list
    for each edge (u, v) **do**
      Relax(u,v)
      update $d(v)$: add it at the head of cell $\left\lfloor \frac{d(v)}{C_{min}} \right\rfloor$ (if it changes cell)
    end for
  end while
end for
return (d, π)

Dijkstra-like Algorithm (1976)

Implementation: Updating $d(v)$:

- First Approach: Removing $d(v)$ from its current LL (connecting its right and left neighbors), and inserting it to its new LL.
  Disadvantage: The element removal takes expansive time.
- Second Approach: Adding on/off-bits to LL elements. Setting the old element's bit off, and creating a new element in its new location.
  Disadvantages:
  - Extra bit processing and checks.
  - Storage: $O(|E|)$.

R.A. Wagner (1976)

At the same time, R.A. Wagner suggested a new implementation of Dijkstra for integer wieght function.
In his algorithm, the number of buckets is $N = \delta_{max}$. Thus its runtime is $O(|E| + \delta_{max})$. Because of the integer weight function, each $d(v)$ is located at the cell representing its accurate value.
Disadvantage: The accuracy of the graph may grow, and so the algorithm runtime. Such an algorithm (which is sensitive to the input) is not scalable.
On the other hand, Dinitz algorithm is scalable for such an input change, because both the denominator and the numerator grow.

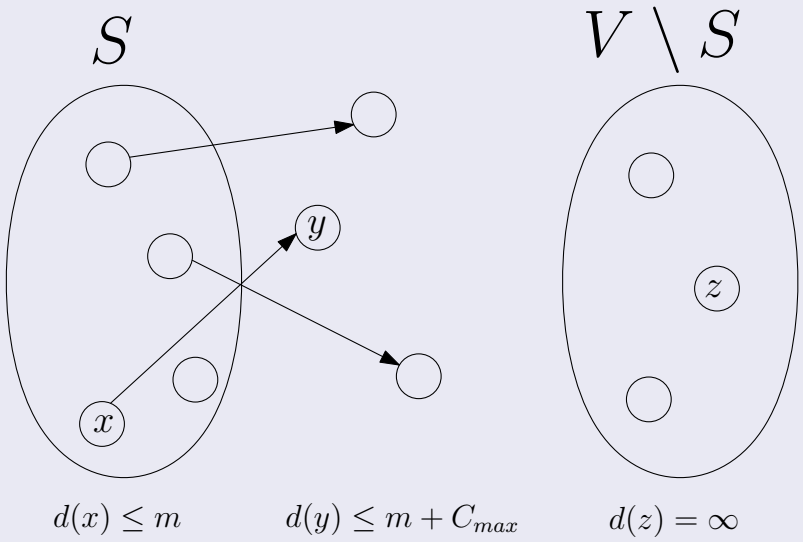
R.A. Wagner (1976)

At the same paper, Wagner also introduced an interesting observation:

Observation

The storage complexity of the algorithm is C_{max}.

Correctness



Thorup (1999)

In 1999, M. Thorup published a paper, presenting an algorithm for undirected single source shortest paths with positive integer weights. In this paper, he shows a way to decide the sizes of the buckets (the buckets are in different sizes). As a result, Thorup obtains a linear time algorithm.

Outline

1 Introduction

2 Dijkstra-like Algorithm

3 Bellman-Ford-Dijkstra Algorithm

Reminder: Bellman-Ford Algorithm (1958)

- Works for digraphs with negative costs of some edges.
- Works in rounds, each being a simple loop of relaxations on the graph edges, in any order.
- Finds the cheapest paths cost in at most $r + 1$ rounds, where for any vertex reachable from s, there exists a cheapest path from s to it containing at most r edges (i.e., r is the depth of the cheapest path tree).
- Detects a negative cycle reachable from s in $|V|$ rounds, if exists. Otherwise, r is at most $|V| - 1$.
- **Running time:** $O(|E||V|)$.

Reminder: Bellman-Ford Algorithm (1958)

Algorithm 6 Plain-scan()

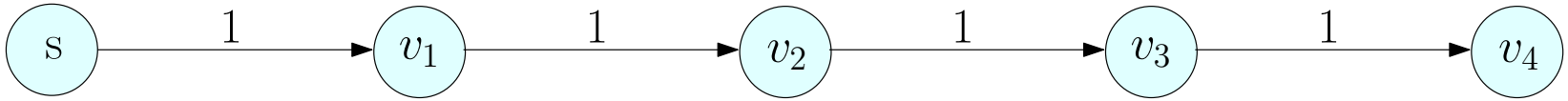
```
for each edge (u, v) ∈ E do
    Relax(u, v)
end for
```

Reminder: Bellman-Ford Algorithm (1958)

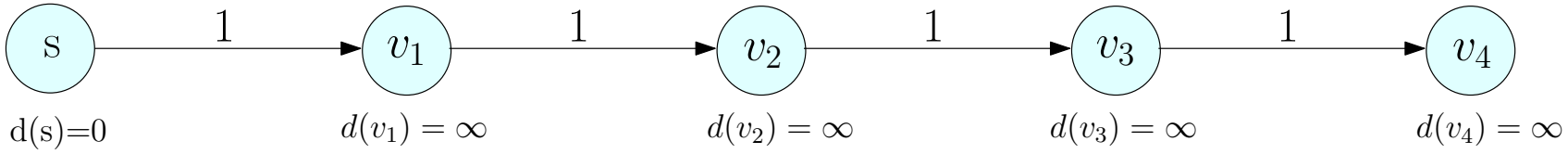
Algorithm 7 Bellman-Ford(G, s)

```
Initialization()
i ← 0
repeat
    i++
    Plain-scan()
until ((there was no change of d at Plain-scan) or (i = |V|))
if i < |v| then
    return (d, π)
else
    return "There exists a negative cycle reachable from s."
end if
```

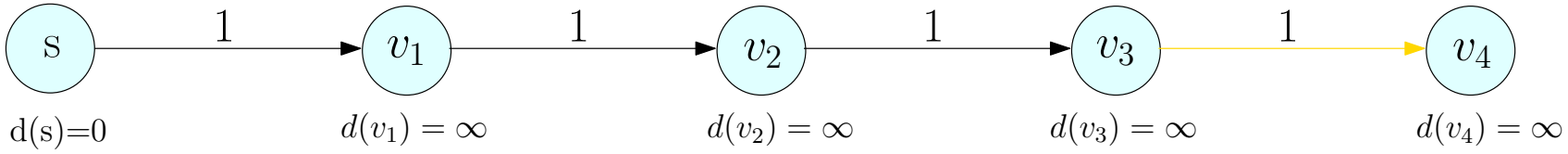
Reminder: Bellman-Ford Algorithm (1958)



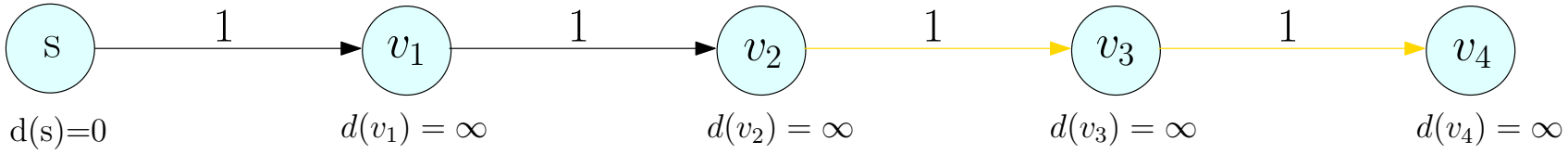
Reminder: Bellman-Ford Algorithm (1958)



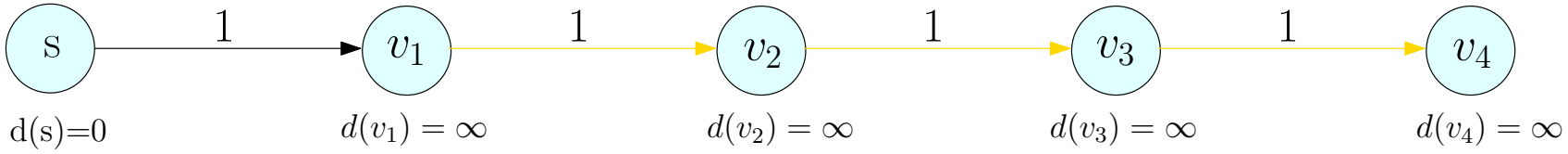
Reminder: Bellman-Ford Algorithm (1958)



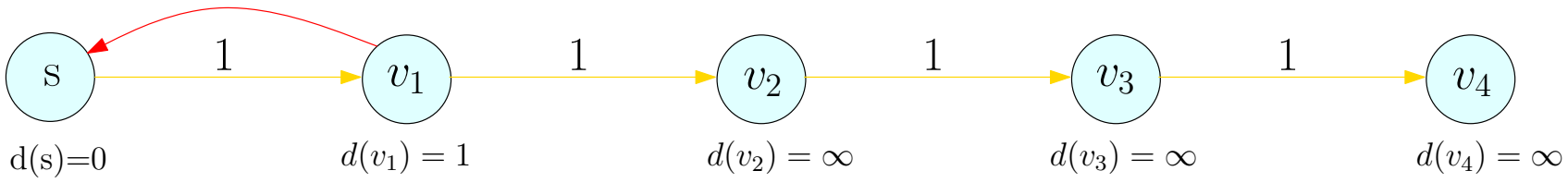
Reminder: Bellman-Ford Algorithm (1958)



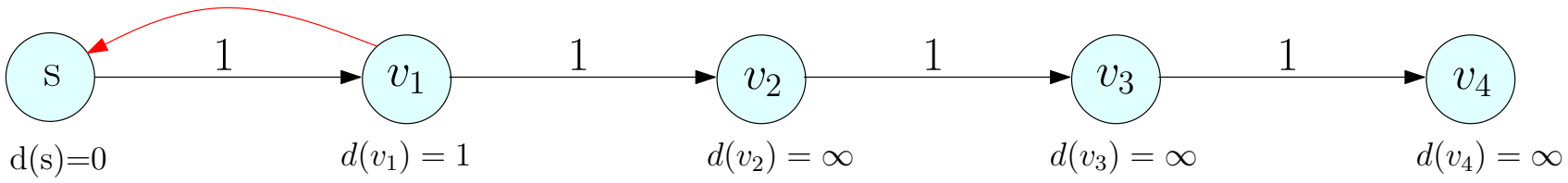
Reminder: Bellman-Ford Algorithm (1958)



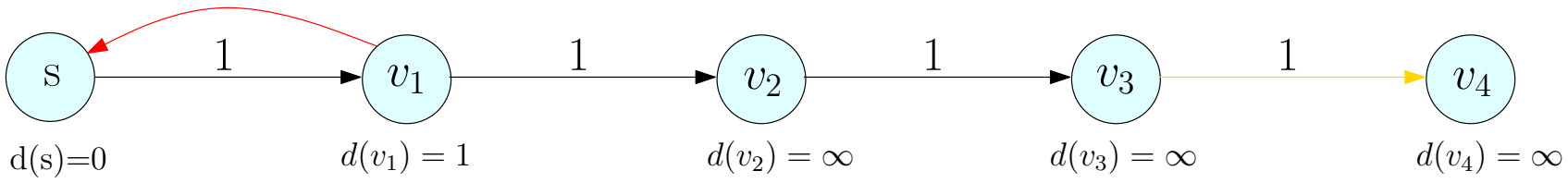
Reminder: Bellman-Ford Algorithm (1958)



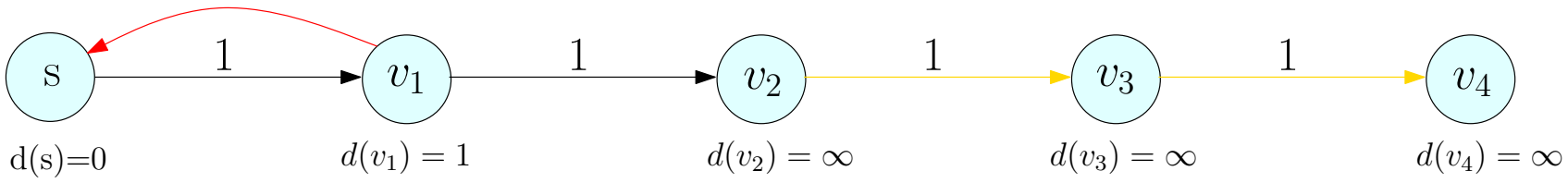
Reminder: Bellman-Ford Algorithm (1958)



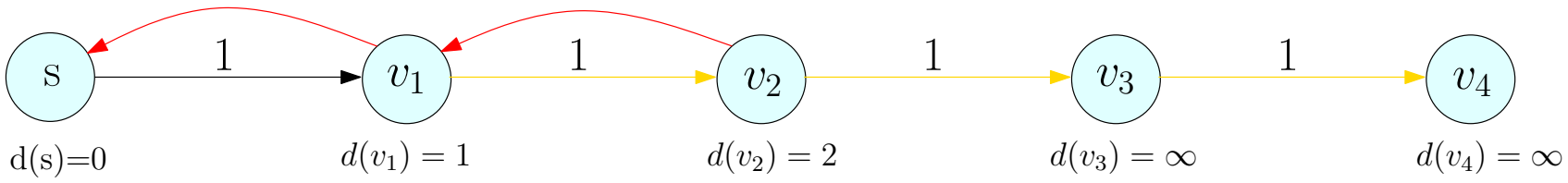
Reminder: Bellman-Ford Algorithm (1958)



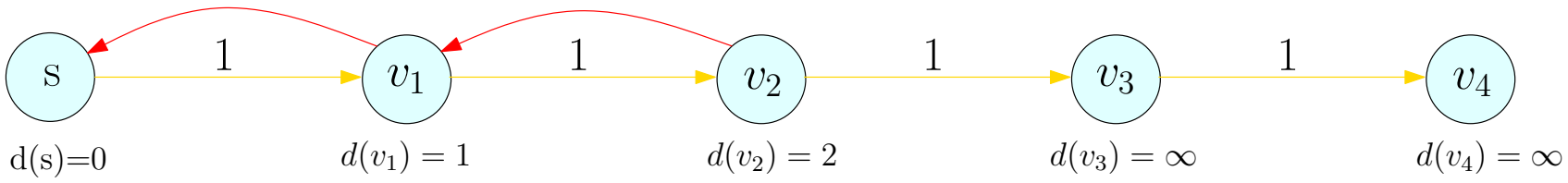
Reminder: Bellman-Ford Algorithm (1958)



Reminder: Bellman-Ford Algorithm (1958)



Reminder: Bellman-Ford Algorithm (1958)



Bellman-Ford-Dijkstra Algorithm(2010)

- Solves the single source cheapest paths problem in a digraph with general edge costs.
- A hybrid of Bellman-Ford and Dijkstra algorithms.
- Improves the running time bound of Bellman-Ford for graphs with a sparse distribution of negative cost edges.
- Iterates Dijkstra several times without reinitializing the values of d at vertices. Dijkstra is just a smart loop of relaxations.

Possible use:

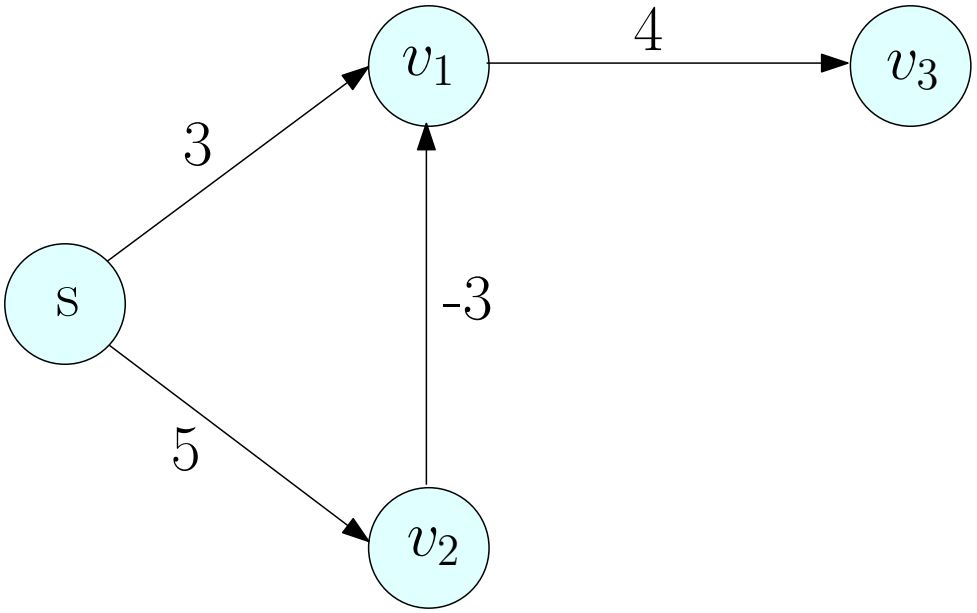
- GPS: Finding a shortest route in a road map while choosing objects of interest.
- May be used as instructive supplement for courses.

Bellman-Ford-Dijkstra Algorithm(2010)

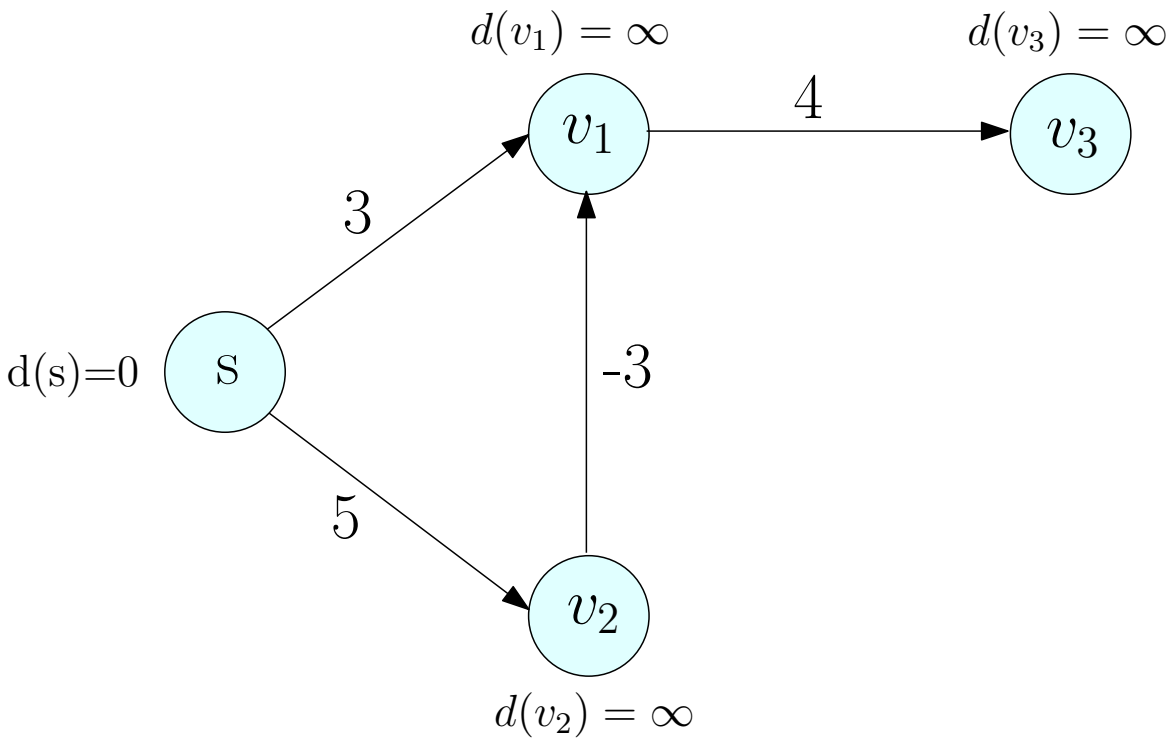
Algorithm 8 Bellman-Ford-Dijkstra(G, s)

```
Initialization()
i ← 0
repeat
    i++
    Dijkstra-scan()
until ((there was no change of d at Dijkstra-scan) or (i = |V| − 1))
if i < |v| − 1 then
    return (d, π)
else
    return "There exists a negative cycle reachable from s."
end if
```

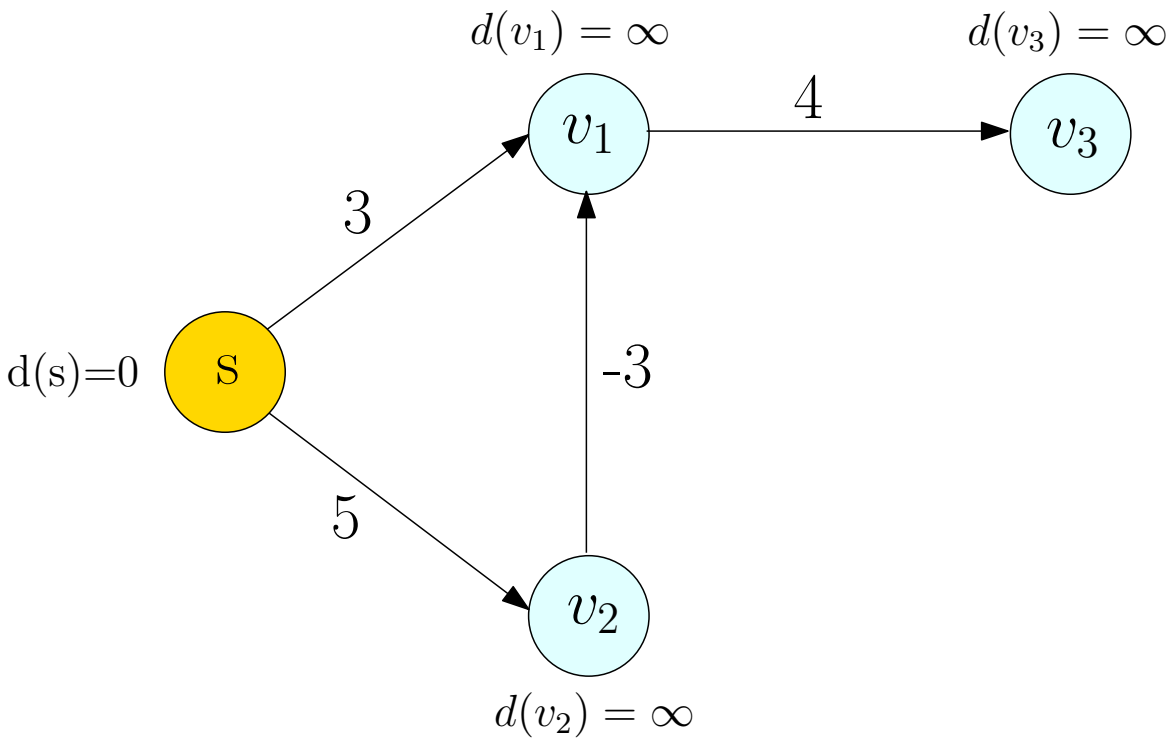
Bellman-Ford-Dijkstra Algorithm(2010)



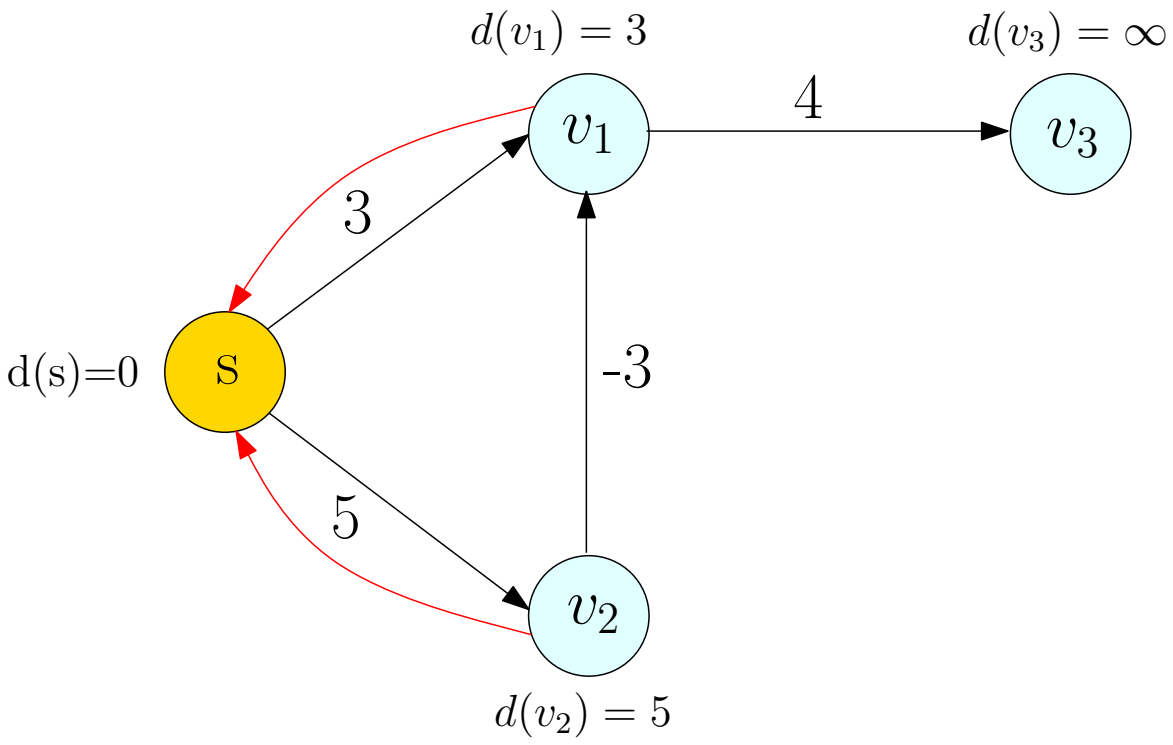
Bellman-Ford-Dijkstra Algorithm(2010)



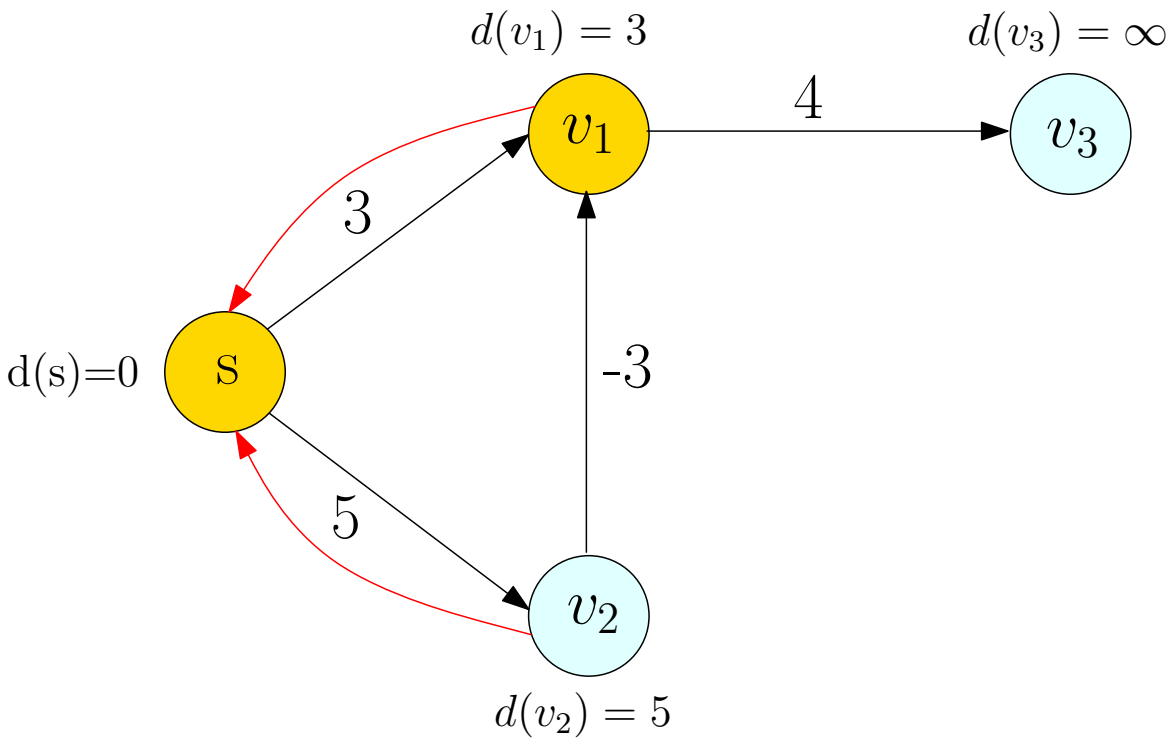
Bellman-Ford-Dijkstra Algorithm(2010)



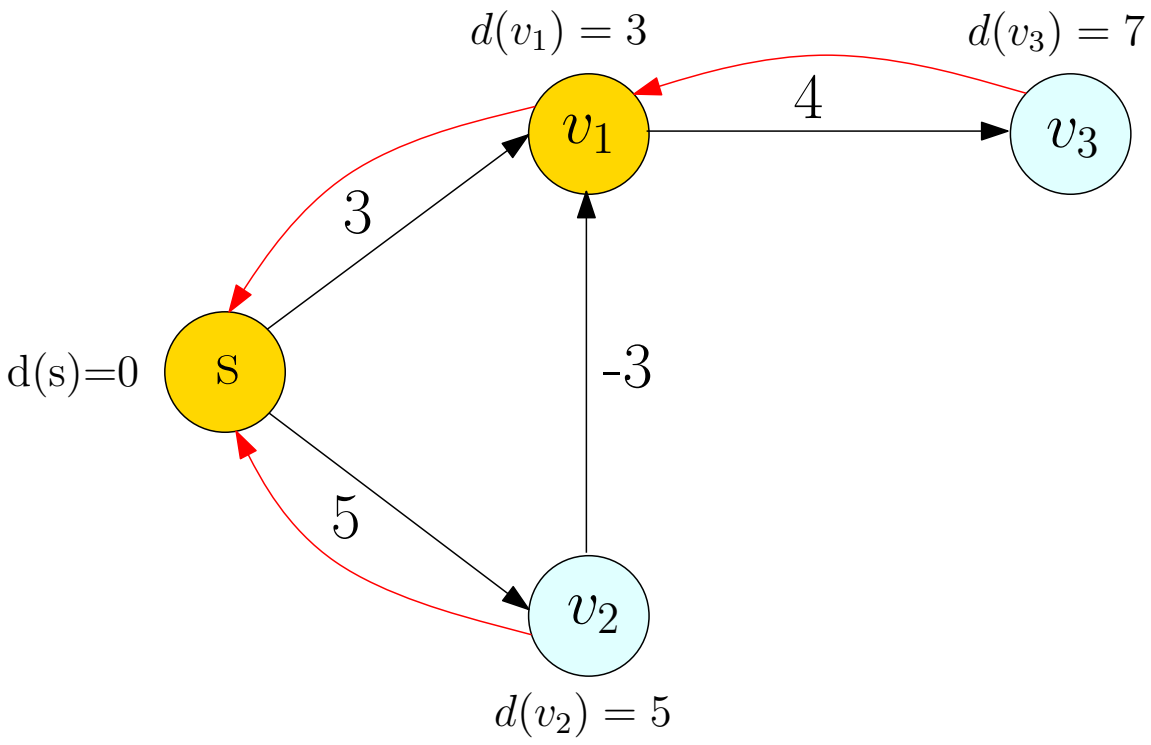
Bellman-Ford-Dijkstra Algorithm(2010)



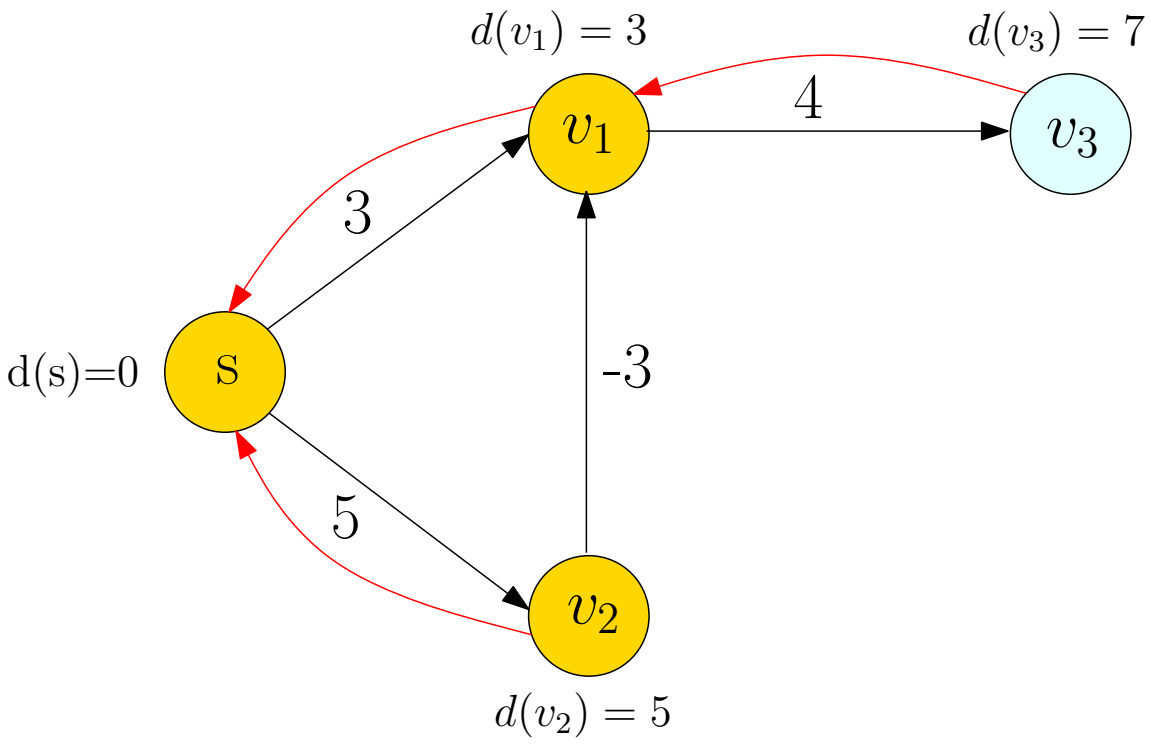
Bellman-Ford-Dijkstra Algorithm(2010)



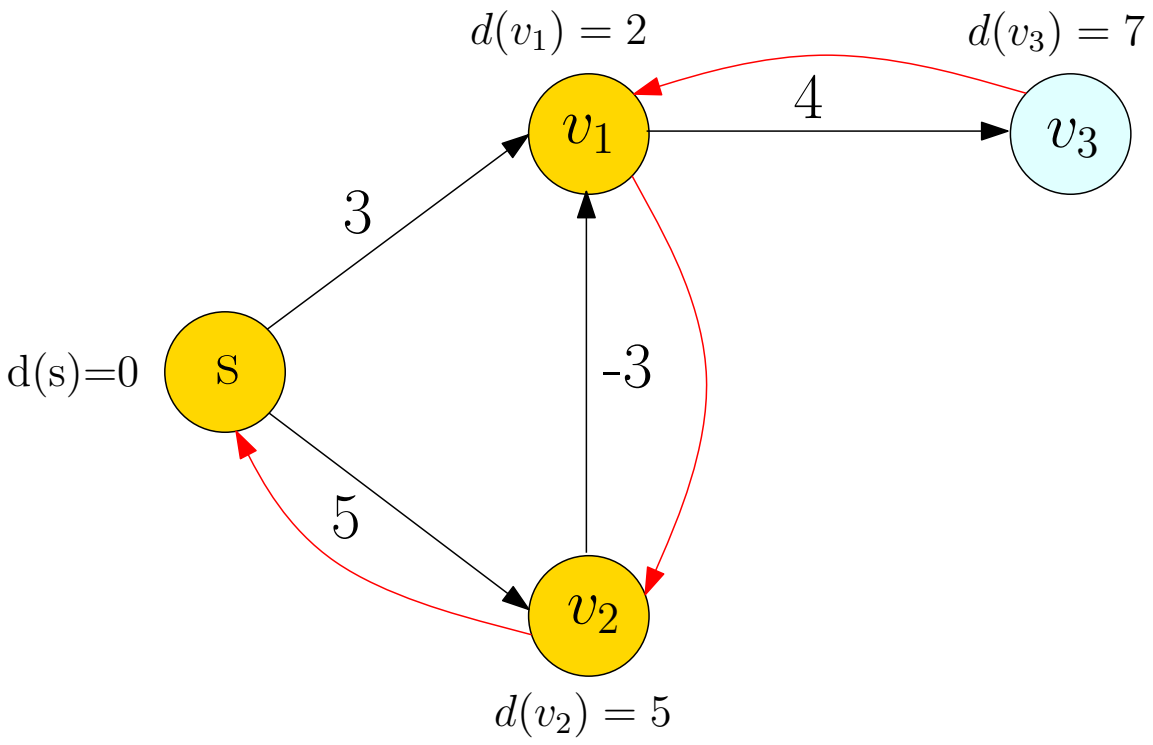
Bellman-Ford-Dijkstra Algorithm(2010)



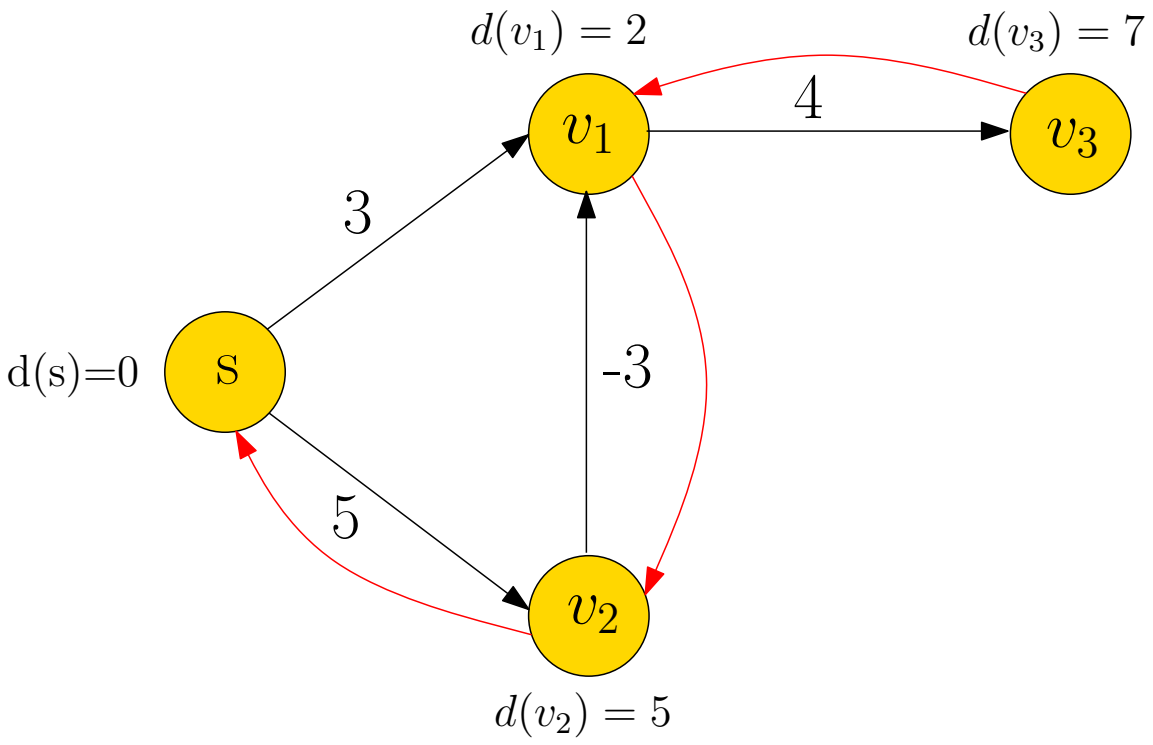
Bellman-Ford-Dijkstra Algorithm(2010)



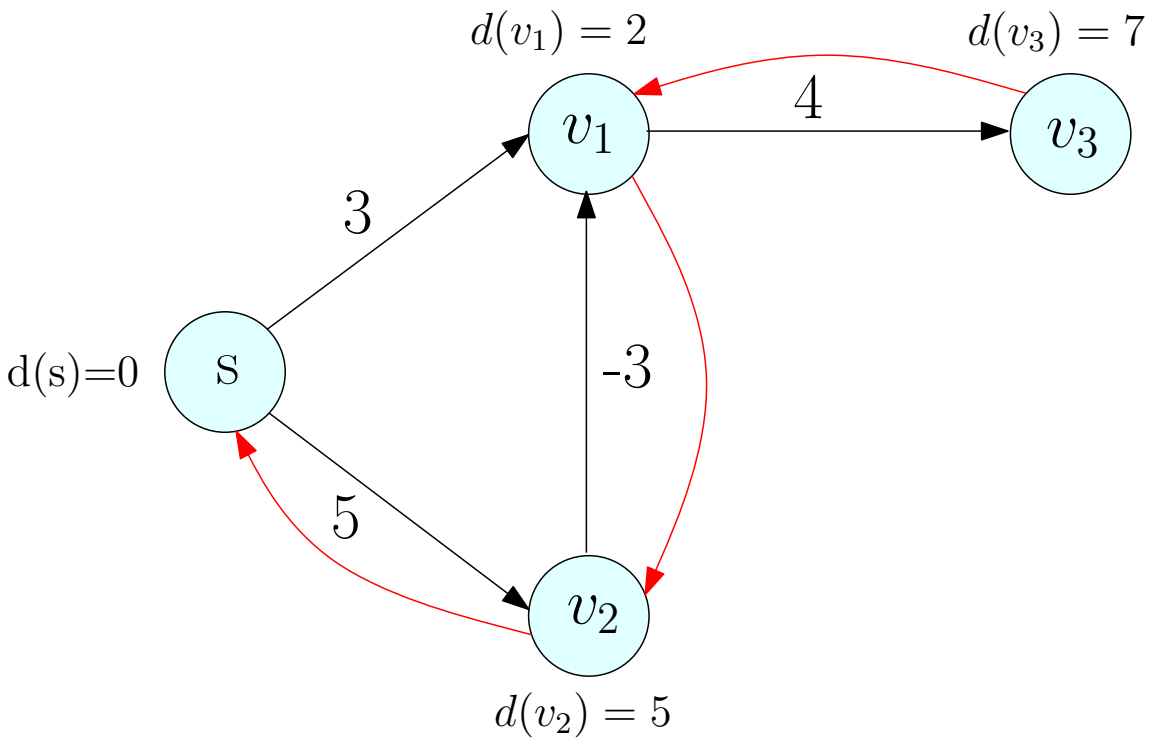
Bellman-Ford-Dijkstra Algorithm(2010)



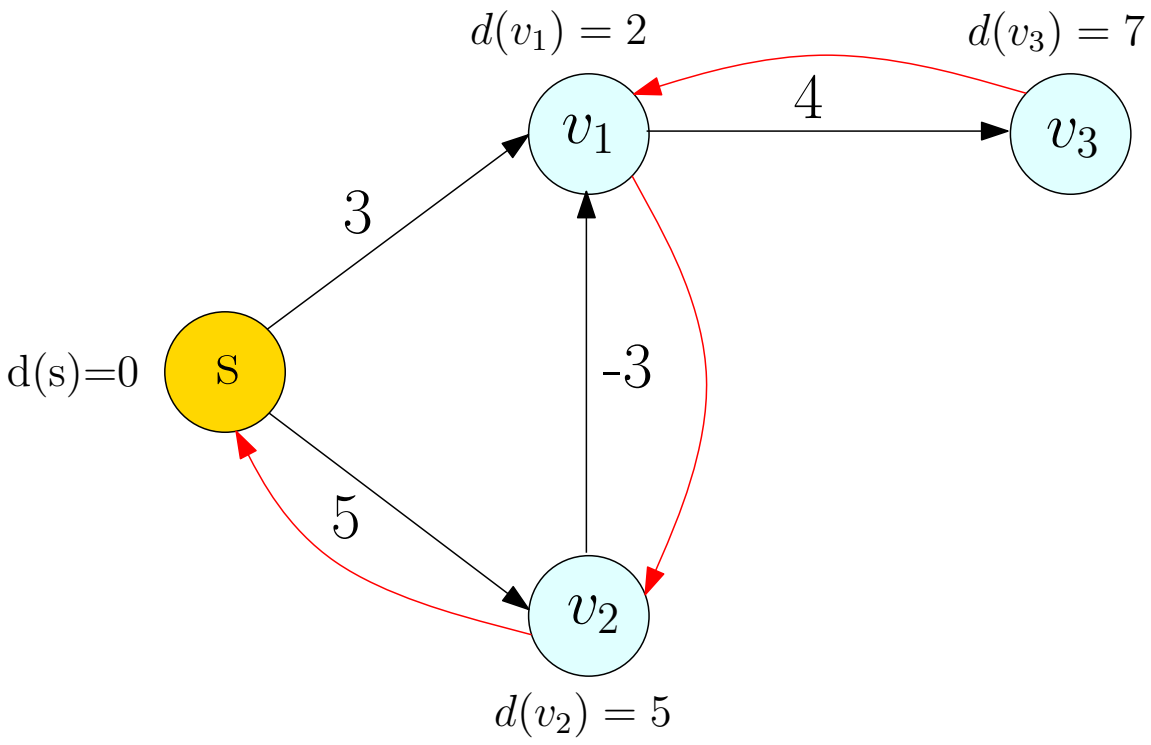
Bellman-Ford-Dijkstra Algorithm(2010)



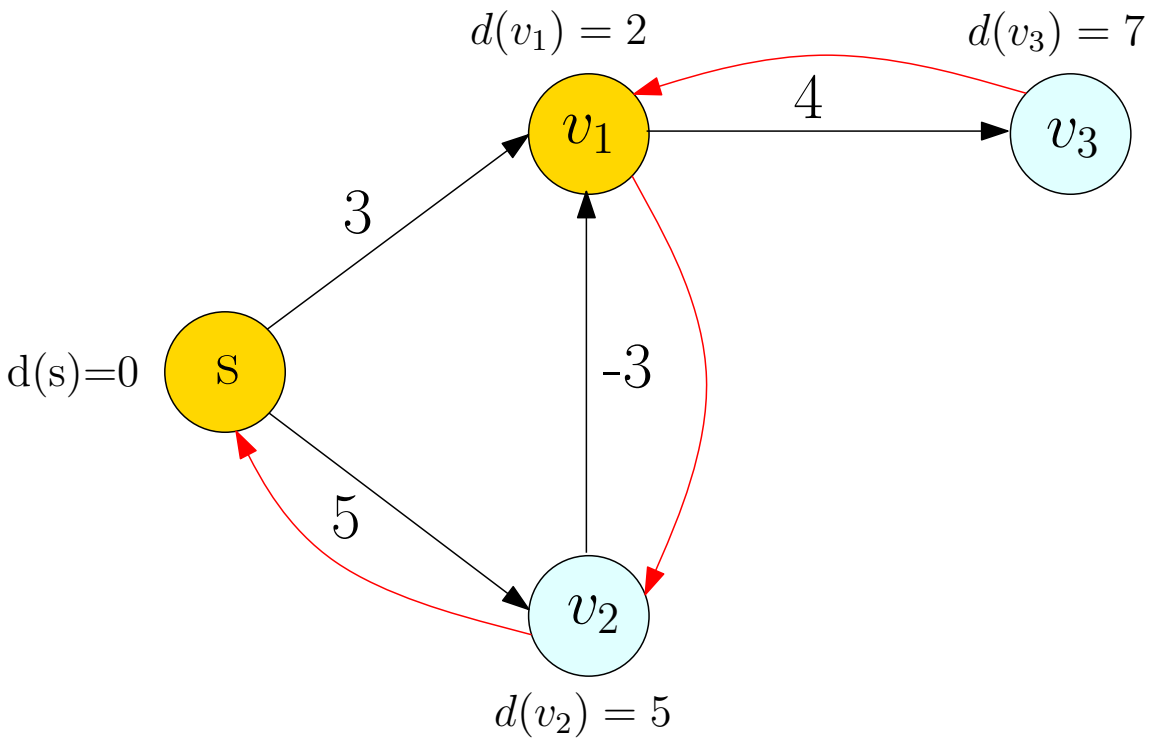
Bellman-Ford-Dijkstra Algorithm(2010)



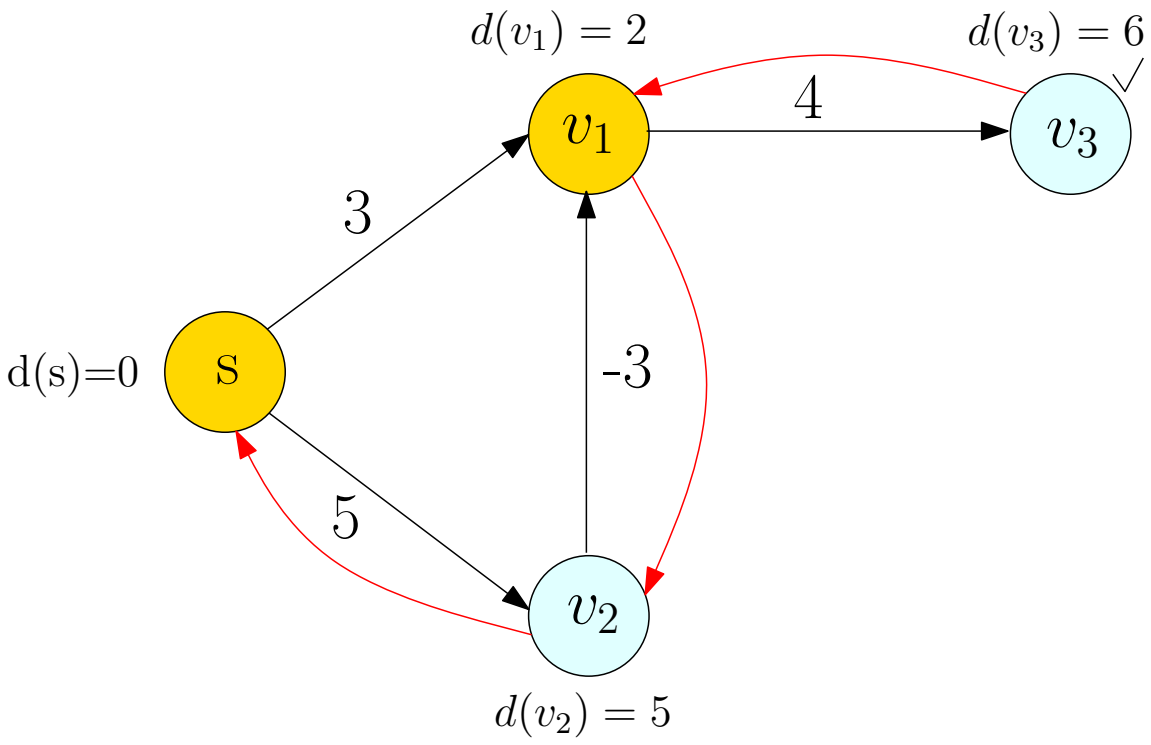
Bellman-Ford-Dijkstra Algorithm(2010)



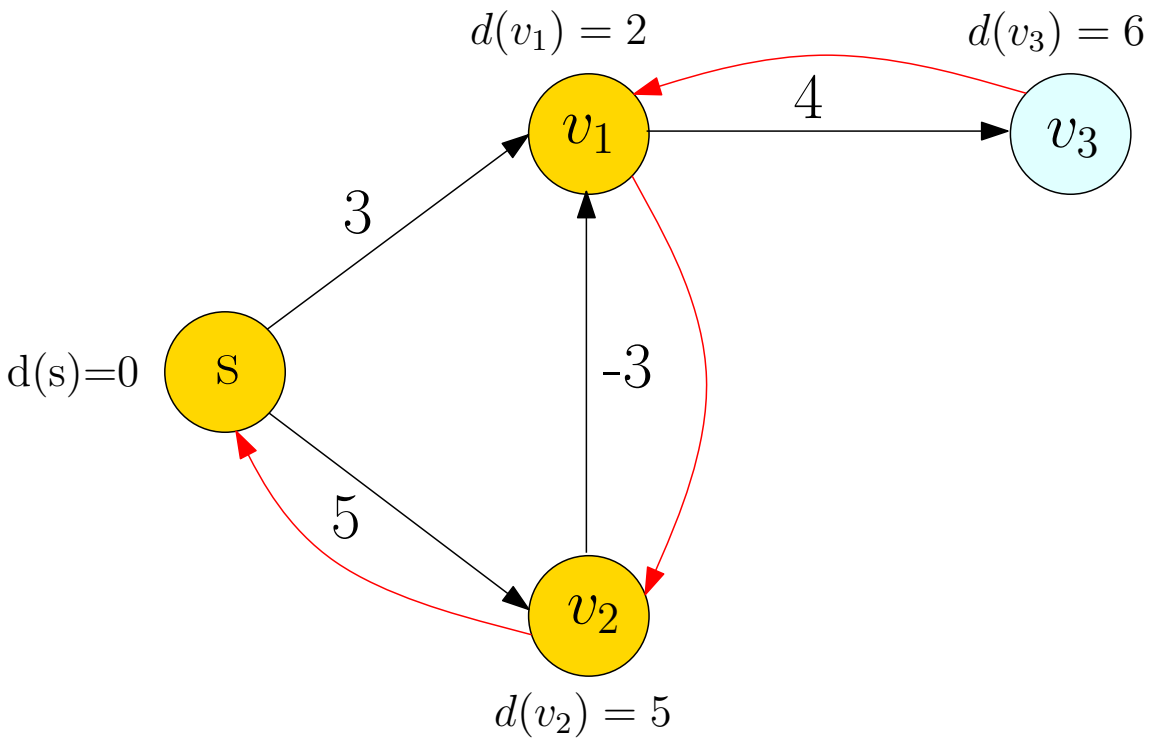
Bellman-Ford-Dijkstra Algorithm(2010)



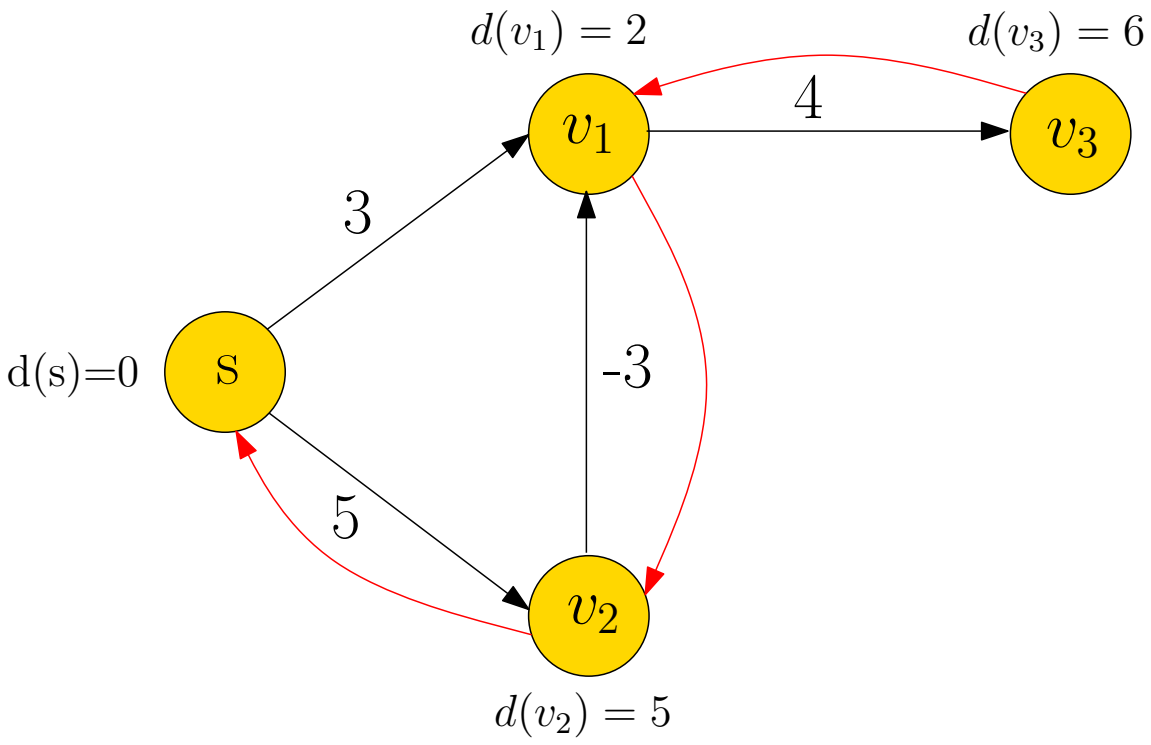
Bellman-Ford-Dijkstra Algorithm(2010)



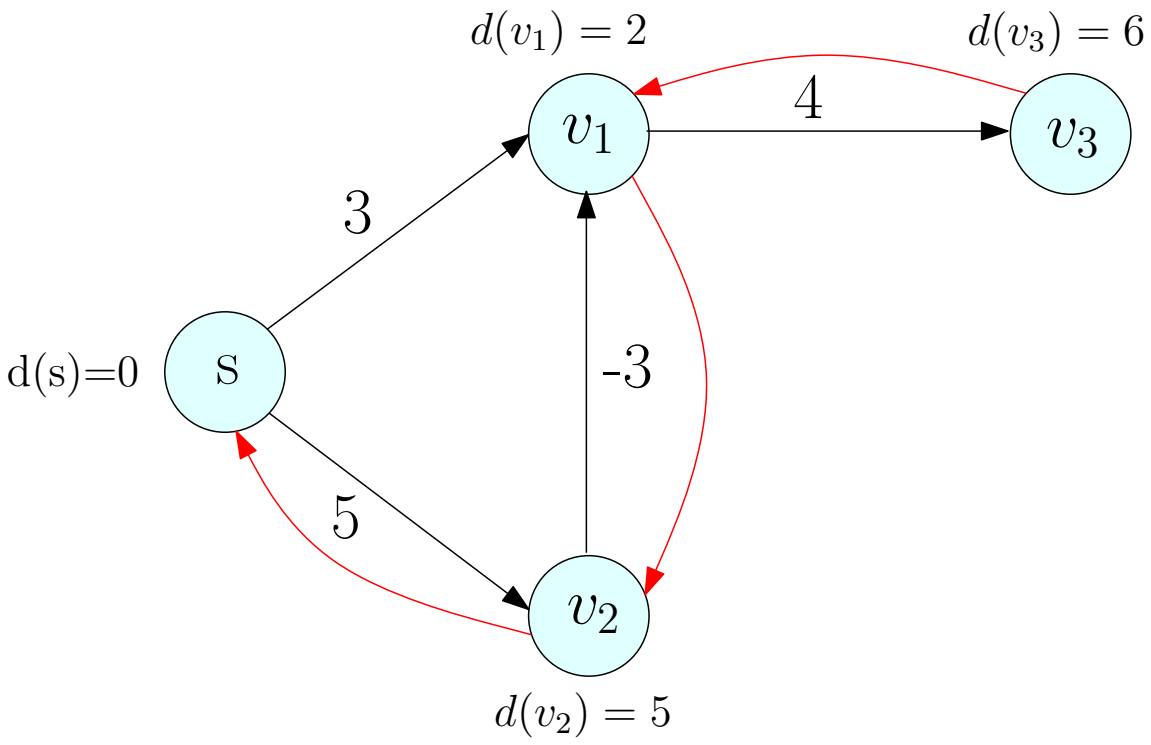
Bellman-Ford-Dijkstra Algorithm(2010)



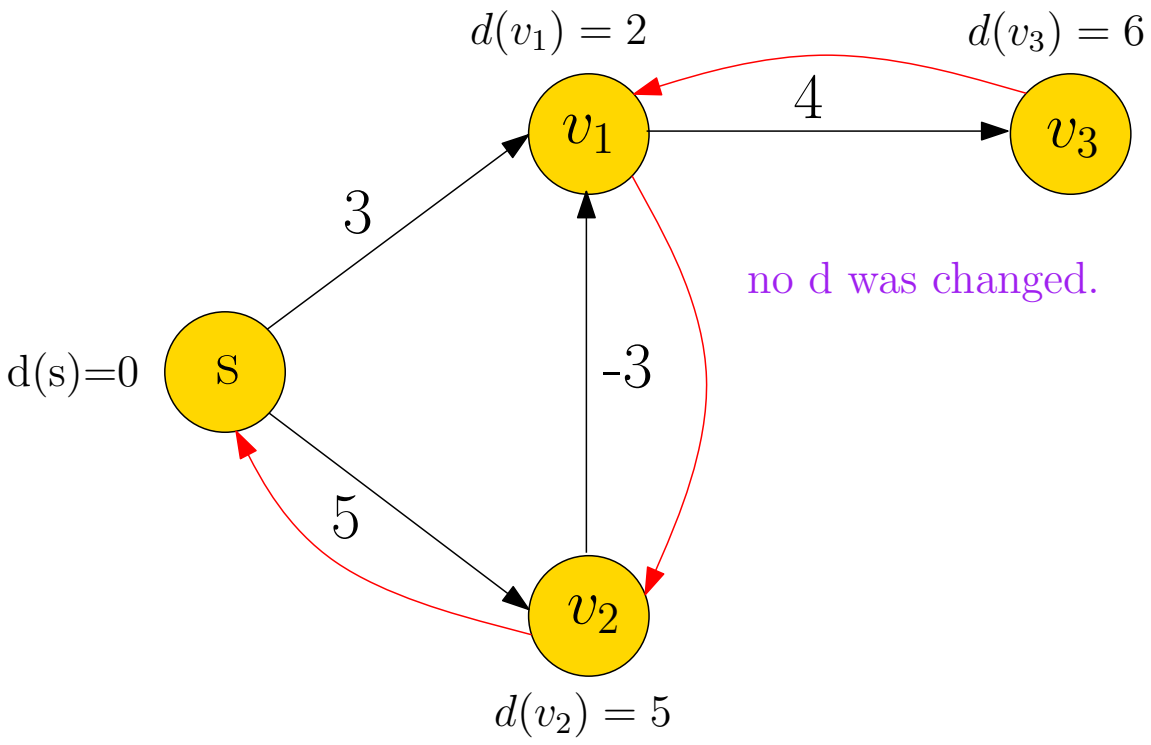
Bellman-Ford-Dijkstra Algorithm(2010)



Bellman-Ford-Dijkstra Algorithm(2010)



Bellman-Ford-Dijkstra Algorithm(2010)



BFD: Analysis

Recall known properties of Relax-based algorithms:

Fact

Consider an arbitrary (properly initialized) sequence of Relax executions.

1. At any moment and for any vertex v, holds $d(v) \geq \delta(v)$.
2. Values of d may only decrease. Therefore, after $d(v)$ reaches $\delta(v)$, neither $d(v)$ nor $\pi(v)$ change.
3. If there is a negative cycle reachable from s, then at any moment there exists an edge (u, v) with $d(u) + c(u, v) < d(v)$.

BFD: Analysis

Notations:

- For a path P, **neg(P)**- the number of negative edges in P, not including its first and last edges.
- For a vertex v reachable from s, **neg(v)**- the minimal value of $neg(P)$ over all cheapest paths from s to v.
- **neg-optimal**- the path providing that minimum for v. $neg(s) = -1$, and $neg(v) = \infty$ if v is not reachable from s.
- **neg(G, s)**- the maximal finite value of $neg(v)$.

Observation

If there exists a cheapest path from s to v, then there exists a simple such path.

Conclusion

Neither $neg(v)$ nor $neg(G, s)$ can exceed $(|V| - 1) - 2 = |V| - 3$.

BFD: Analysis

Proposition

If there exists a cheapest path from s to v, $d(v)$ equals $\delta(v)$ after $neg(v) + 1$ BFD rounds.

Theorem

If there is no negative cycle reachable from s in G, BFD correctly computes the cheapest path value for all $v \in V$ in at most $neg(G, s) + 2$ rounds.
Otherwise, it reports on the existence of such a cycle.

BFD: Analysis

Proof

- By the proposition, if there is no negative cycle reachable from s, after $neg(G, s) + 1$ BFD rounds, d values equal δ values at all vertices.
- By Fact 2, at the round numbered at most $(neg(G, s) + 2)$ there is no change of d values, and BFD stops.
- Since $neg(G, s) \leq |V| - 3$, this should happen not later than after round $|V| - 1$.

BFD: Analysis

Proof

If there exists a negative cycle reachable from s then:

- by Fact 3, even at round $|V| - 1$ there would exist an edge (u, v) with $d(u) + c(u, v) < d(v)$.
- Since Dijkstra scan executes Relax on all edges reachable from s, there would be a change of d at round $|V| - 1$.
- BFD will then stop and report accordingly.

BFD: Analysis

Proposition

If there exists a cheapest path from s to v, $d(v)$ equals $\delta(v)$ after $neg(v) + 1$ BFD rounds.

Proof

We prove by induction on $neg(v)$.

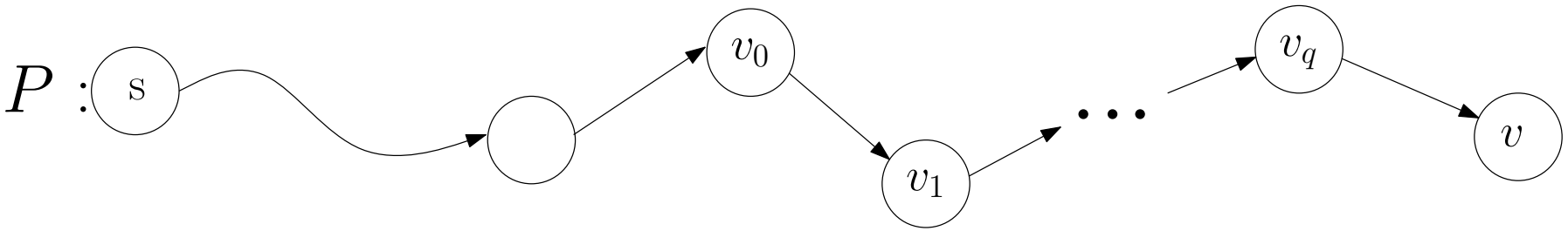
The basis case: for $neg(v) = -1$, i.e., $v = s$ the statement is correct.

The induction assumption: the statement is correct for all v' such that $neg(v') < k$, $k \geq 0$.

The induction step: let v be a vertex such that $neg(v) = k$.

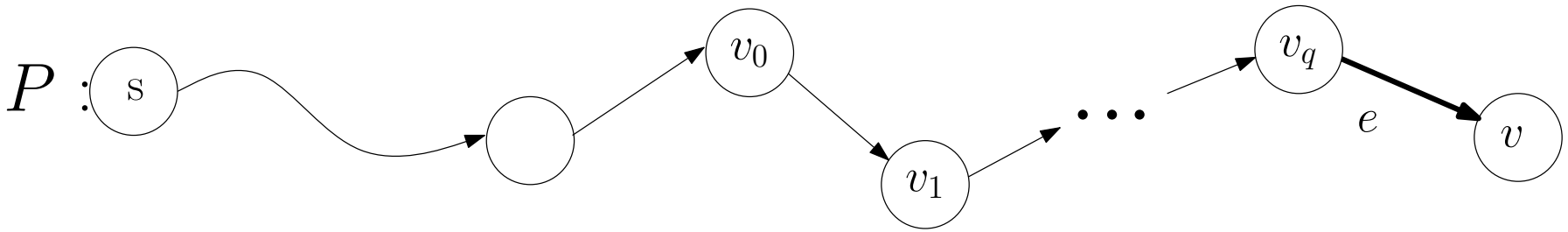
BFD: Analysis

P is a neg-optimal path to v.



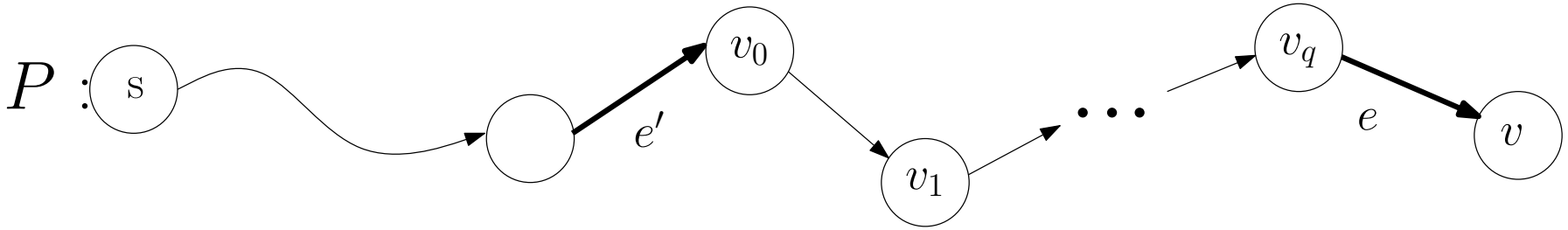
BFD: Analysis

e is the last edge along P.



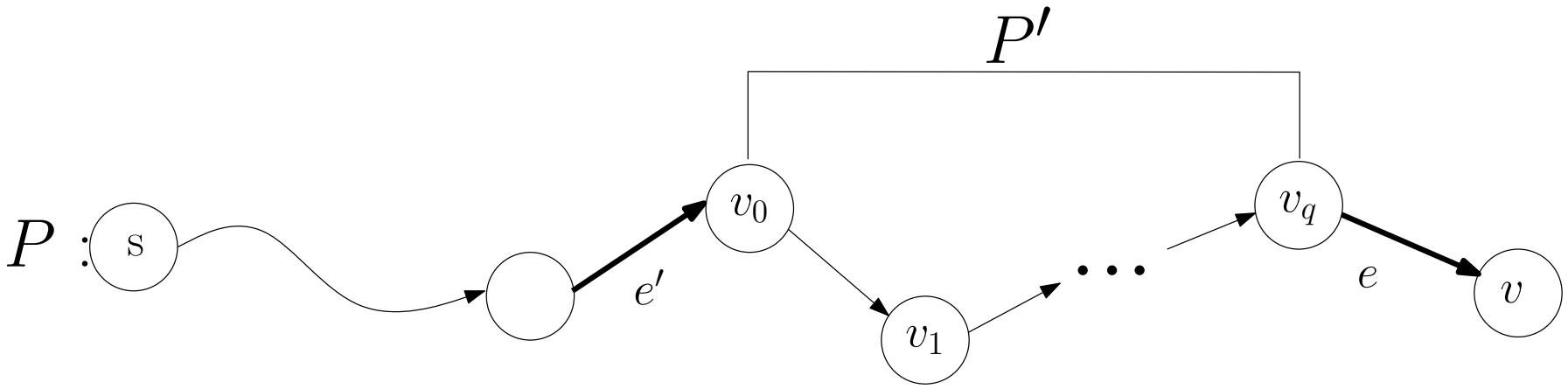
BFD: Analysis

e' is the last negative edge along P, which is neither its first nor its last edge, if it exists (if $k \geq 1$).



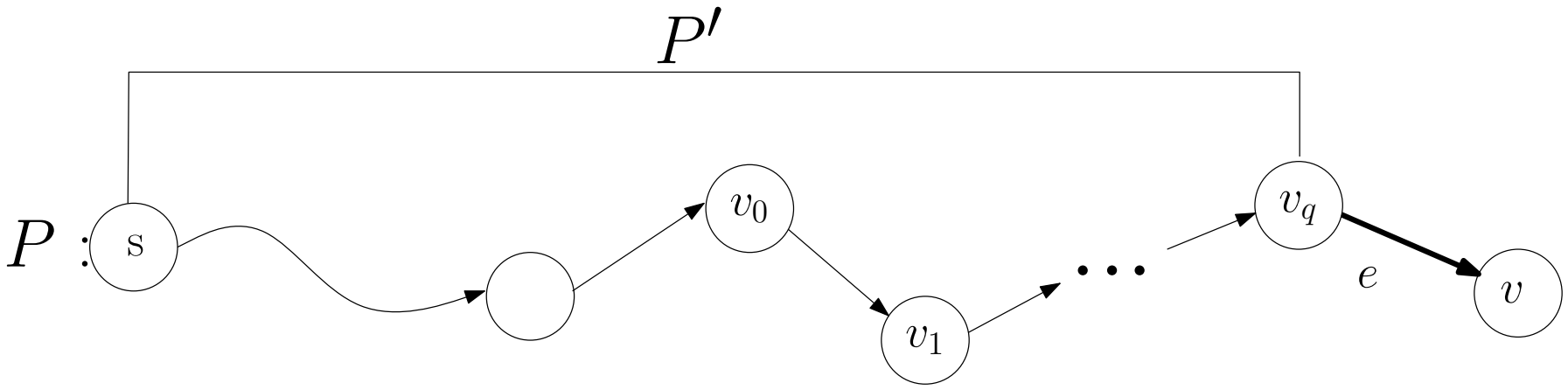
BFD: Analysis

If e' exists, P' is the part of P between e' and e.



BFD: Analysis

Otherwise, P' is P without e.



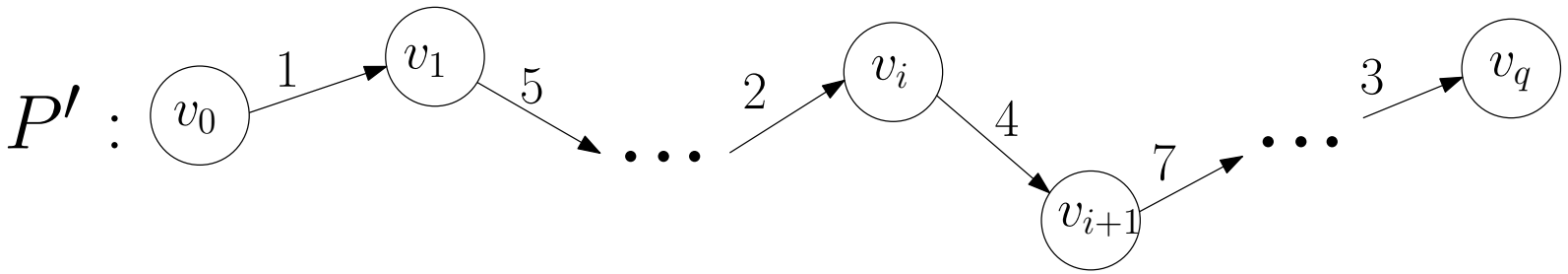
BFD: Analysis

Proof

- Let $P' = (v_0, v_1, ..., v_q)$ then $neg(v_0) = k - 1$, by definition. Hence, by the induction assumption $d(v_0) = \delta(v_0)$ after k rounds.
- Let k' be the first round where v_0 is scanned with $d(v_0) = \delta(v_0)$ $(k' \leq k + 1)$.
- We will prove that $d(v) = \delta(v)$ at the end of that round.

BFD: Analysis

At first, we prove for the case, when all edges of P', except its first edge if $v_0 = s$, have a positive cost.



BFD: Analysis

Auxiliary Statement

At the k'th round, the vertices $v_i \in P'$ are scanned with $d(v_i) = \delta(v_i)$, in the increasing order of i.

Proof

We prove by induction on the indexes.

The basis case: If $k = 0$ then $v_0 = s$. s is always scanned first in the first Dijkstra scan with $d(v_0) = \delta(v_0)$, by initializing.

if $k \geq 1$ then

1. All edges in P' have a positive cost.
2. Any prefix of P is an optimal path to its end-vertex.

BFD: Analysis

Proof

- Therefore, function δ strictly grows along P'.
- For any $v_i, v_j, i < j$, always holds $\delta(v_i) < \delta(v_j) \leq d(v_j)$.

During the k'th round $d(v_0) = \delta(v_0)$ is always the unique minimal value of d on P'. Hence, by the Dijkstra rule, v_0 enters S first among the vertices on P'.

The induction assumption: v_i, $i < q$, is scanned with $d(v_i) = \delta(v_i)$ before v_{i+1} at the k'th round.

The induction step: During the scan of v_i at the k'th round $d(v_{i+1})$ gets its final value $\delta(v_{i+1})$ via the relaxation on edge (v_i, v_{i+1}), if needed.

After that, $d(v_{i+1})$ is always minimal among the values of d on $v_{i+1}, ..., v_q$ by the same reasons used before. Therefore, v_{i+1} is scanned next on P' with $d(v_{i+1}) = \delta(v_{i+1})$.

BFD: Analysis

A reminder:

Proposition

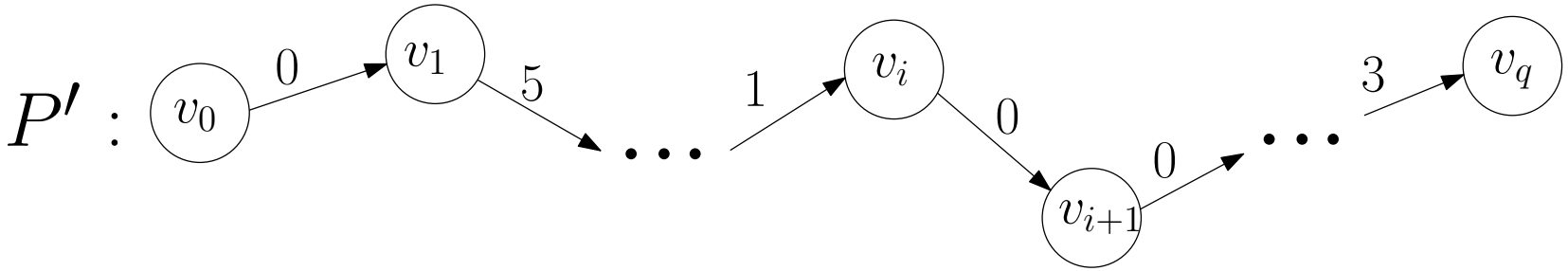
If there exists a cheapest path from s to v, $d(v)$ equals $\delta(v)$ after $neg(v) + 1$ BFD rounds.

Proof

- $d(v_0) = \delta(v_0)$ after k rounds.
- k' is the first round where v_0 is scanned with $d(v_0) = \delta(v_0)$ ($k' \leq k + 1$).
- We will prove that $d(v) = \delta(v)$ at the end of that round.
  By the auxiliary statement, v_q is scanned with $d(v_q) = \delta(v_q)$ at the k'th round.
  After the relaxation on edge (v_q, v), holds $d(v) \leq \delta(v)$.
  Hence, $d(v) = \delta(v)$ after the k'th round as required.

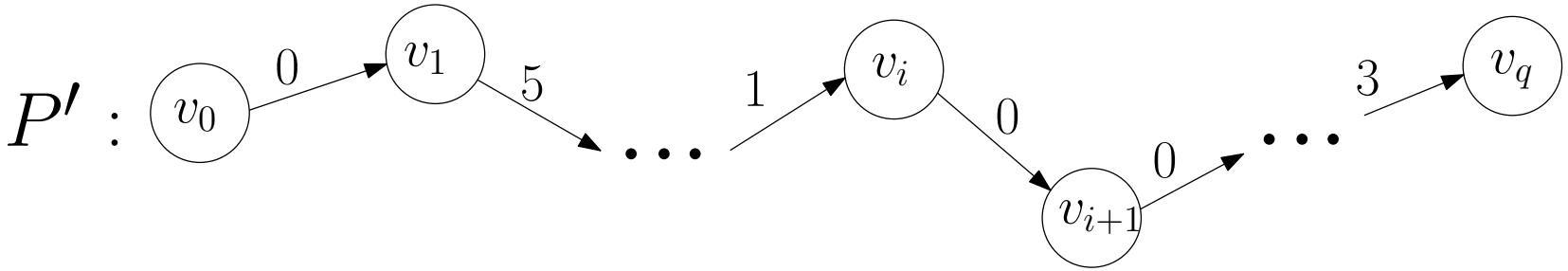
BFD: Analysis

Now, let edge costs on P' be general.



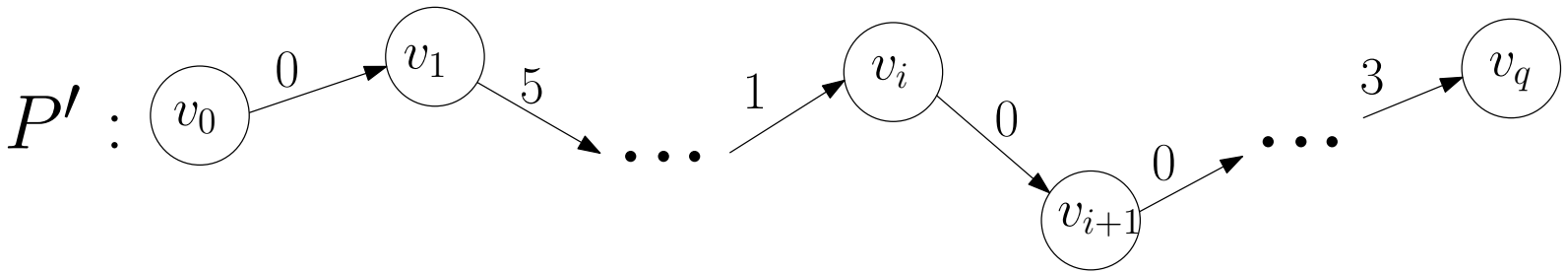
BFD: Analysis

There maybe sub-paths of P' consisting of edges of zero cost only.



BFD: Analysis

δ is a constant at each of them, and those constants strictly grow along P'.



$$\delta(v_0) = \delta(v_1) < \delta(v_2) \ . \ . \ . < \delta(v_i) = \delta(v_{i+1}) < \delta(v_q)$$

BFD: Analysis

Claim

The vertices of such a sub-path may enter S in *any* order between them, with $d(v_i) = \delta(v_i)$ at the moment of there scanning.

This closes the gap between the restricted and the general cases.

BFD: Running Time

An implementation based on Fibonacci heap provides the following bound for BFD:

Theorem

There exists an implementation of BFD running in time $O(neg(G, s) \cdot (|E| + |V| \log |V|))$ for graphs with no negative cycle reachable from s.

BFD: Running Time

Similiarly to BF, BFD detects negative cycles in $|V| - 1$ rounds. This bound can be reduced for some specific cases:

Observation

For any a priory known bound B for $neg(G, s)$, the stopping condition of BFD may be $< i = B + 2 >$.

BFD: Running Time

For example, the total number of negative edges in G is such a bound.
Let $\#neg(G)$ denote the **minimum** of the number of vertices with **outgoing negative edges** (excluding s) and the number of vertices with **incoming negative edges**. It holds $neg(G, s) \leq \#neg(G)$.

Corollary

The stopping condition $< i = |V| - 1 >$ of BFD may be replaced by $< i = \#neg(G) + 2 >$.

BFD: Running Time

- Many implementations of Dijkstra are known. They provide either better worst case bounds for particular graph classes, or better constant factors in running time bounds for the general case.
- Some implementations of Dijkstra might rely on non-negativity of edge costs and hence might work improperly if there are negative edges in G.
- In addition, there exist implementations of Dijkstra-like algorithms, working with special time bounds for specific graph classes, as shown before.

BFD: Running Time

Denote the set of negative edges in G by E^-. BFD-r is a robust version of BFD.
The change in BFD-r is as follows:

- The call to Dijkstra-scan, in the do $< \ldots >$ until loop, is replaced by two consequent calls:
  Dijkstra-scan($E \setminus E^-$) and Plain-scan(E^-).

Theorem

The BFD-r version of BFD is correct, keeping the round number bounds, whichever implementation of Dijkstra on a graph with non-negative edge costs is used in Dijkstra-scan.

BFD: Running Time

Denote the set of edges with positive costs in G by E^+, and the minimal positive edge cost by c^+_{min}.
BFD-r+ is another version of BFD in which:

- The call to Dijkstra-scan is replaced by two consequent calls: Dijkstra-scan(E^+) and Plain-scan($E \setminus E^+$).
- Accordingly, we change the measure of the BFD complexity: Let $non_pos(G, s)$ be defined similarly to $neg(G, s)$, but with respect to edges with non-positive costs.

BFD: Running Time

Proposition

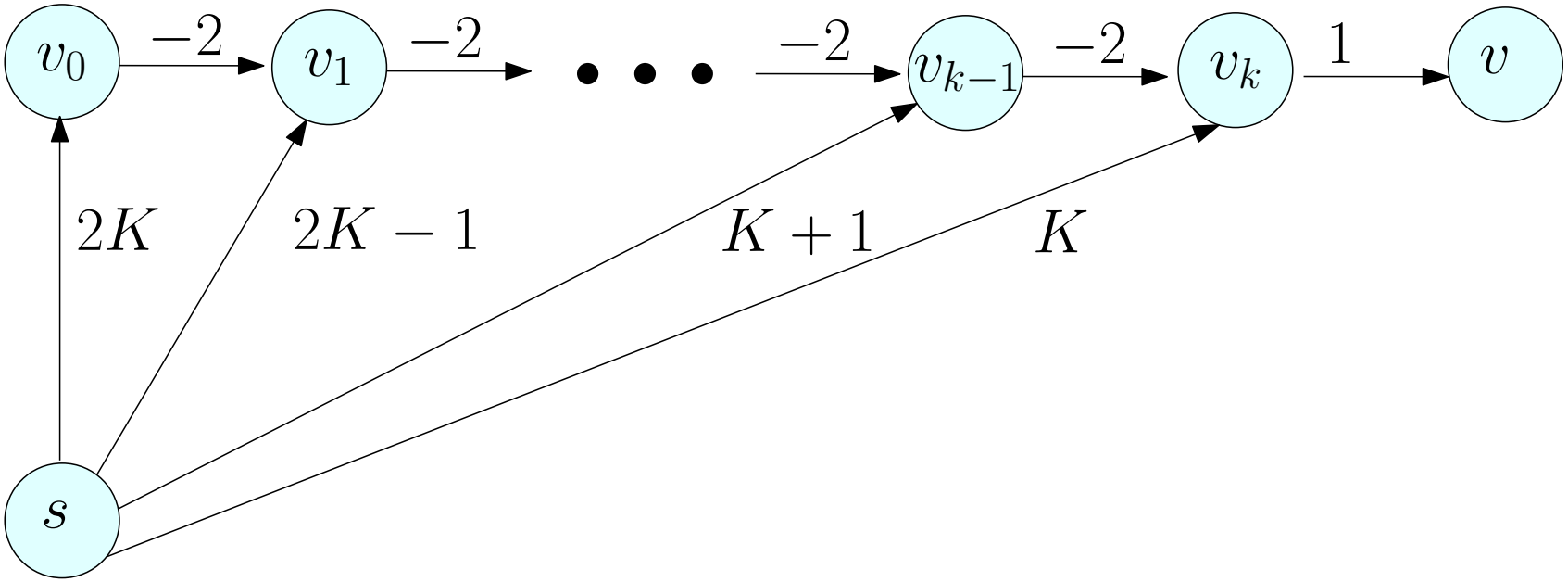
If there exists a cheapest path from s to v, after $non_pos(v) + 1$ rounds of BFD-r+ holds $d(v) = \delta(v)$.

Theorem

The BFD-r+ version of BFD is correct when the Dijkstra like algorithm and its implementation are used in Dijkstra scan, while the round number bound changes to $non_pos(G, s) + 2$.

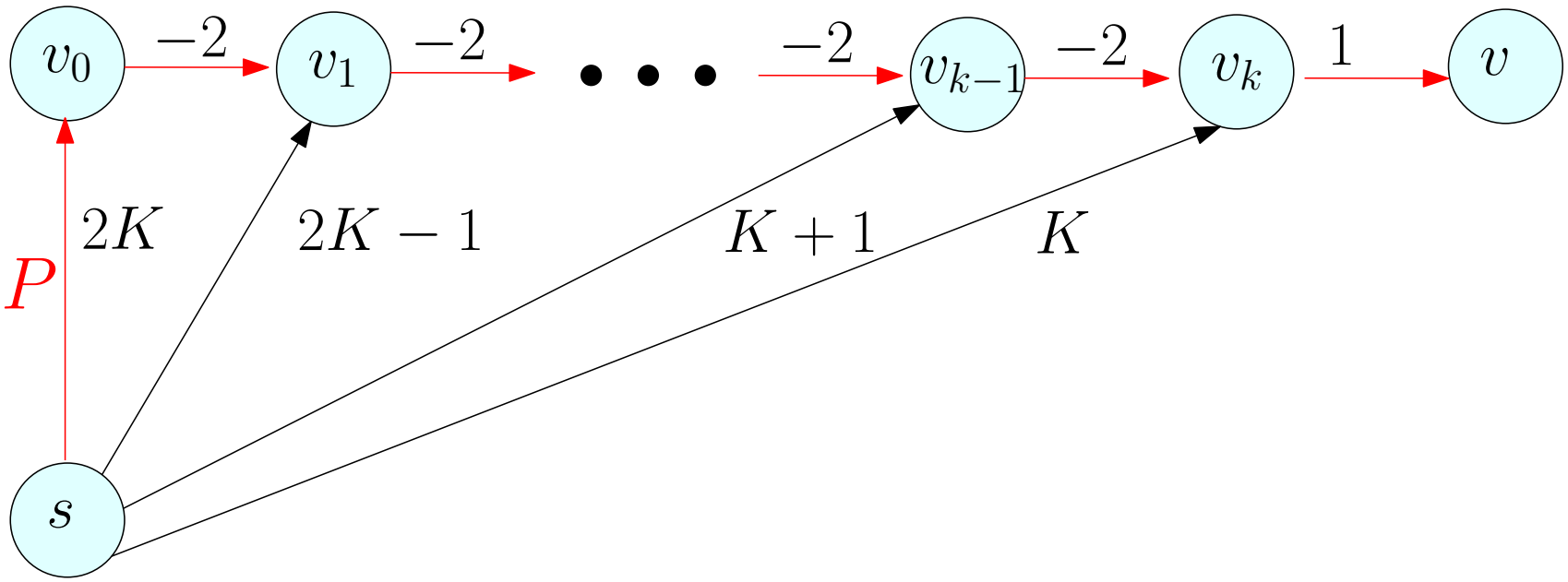
BFD: Tightness of Bounds and a Speeding Up Idea

Consider the graph G:



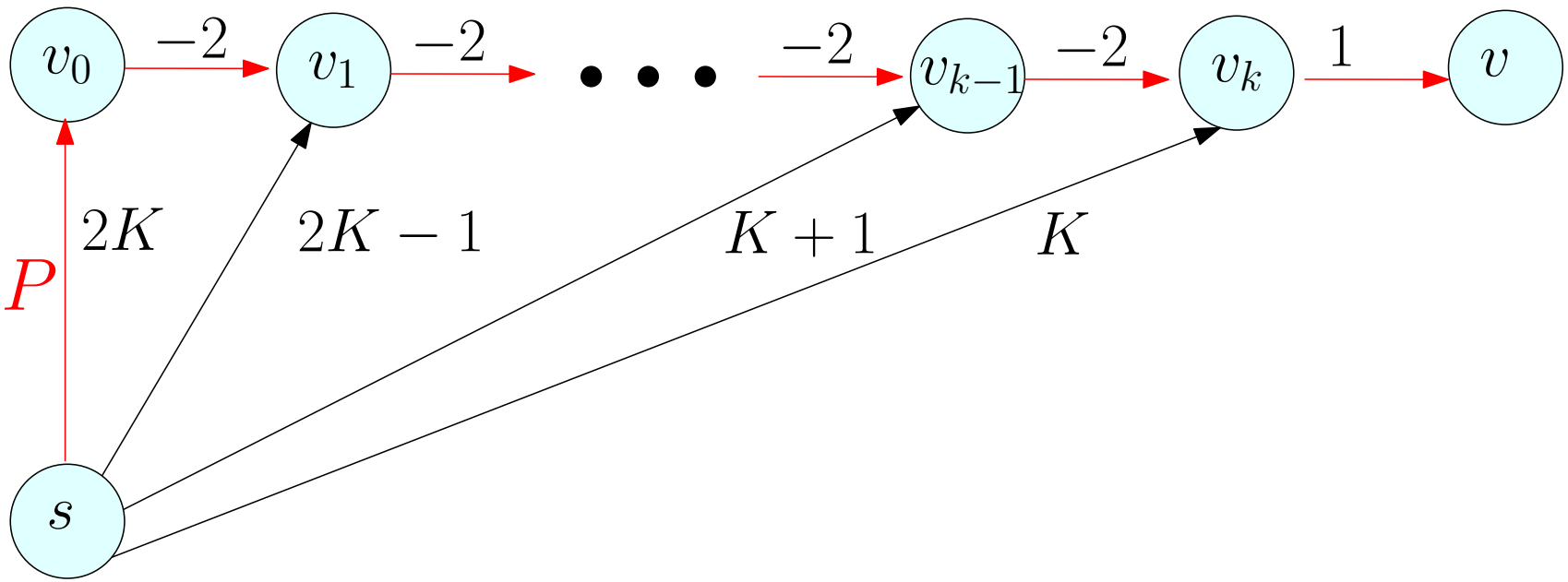
BFD: Tightness of Bounds and a Speeding Up Idea

The cheapest path from s to v is $P = (s, v_0, ..., v_k, v)$, its cost is 1.



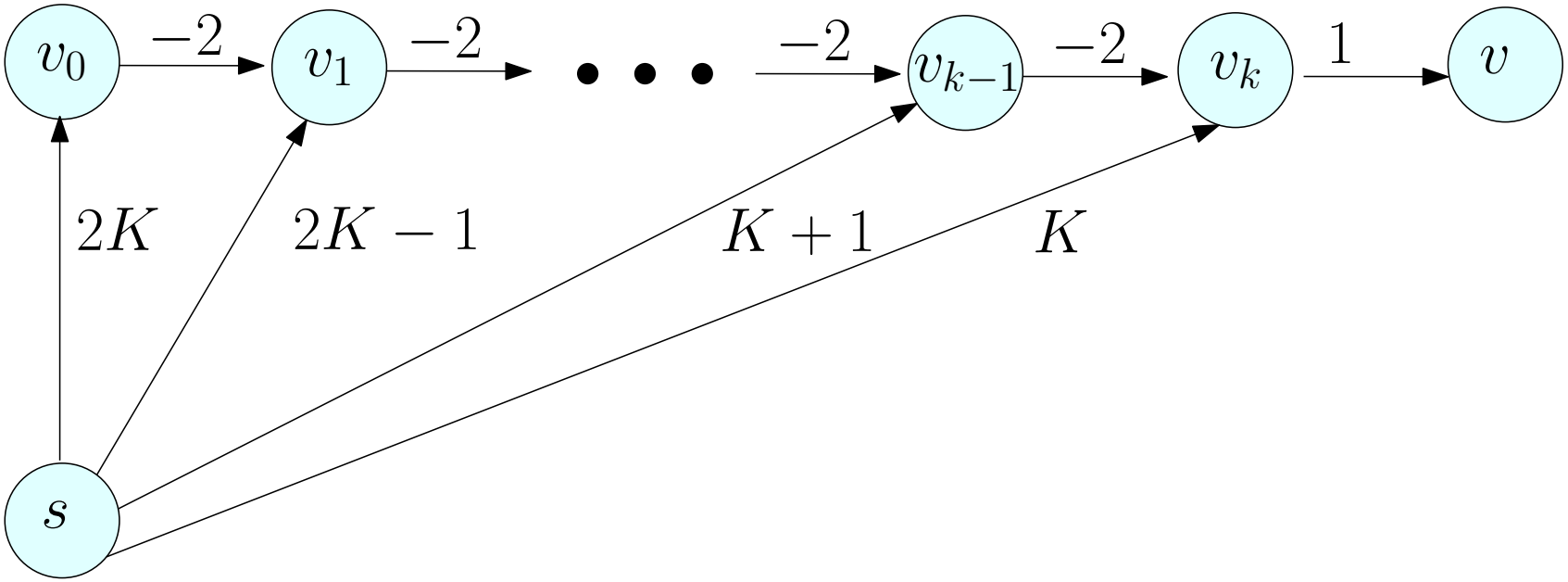
BFD: Tightness of Bounds and a Speeding Up Idea

P contains all k negtive egdes in G as inner edges.
Hence, $neg(G, s) = k = |v| - 3$.

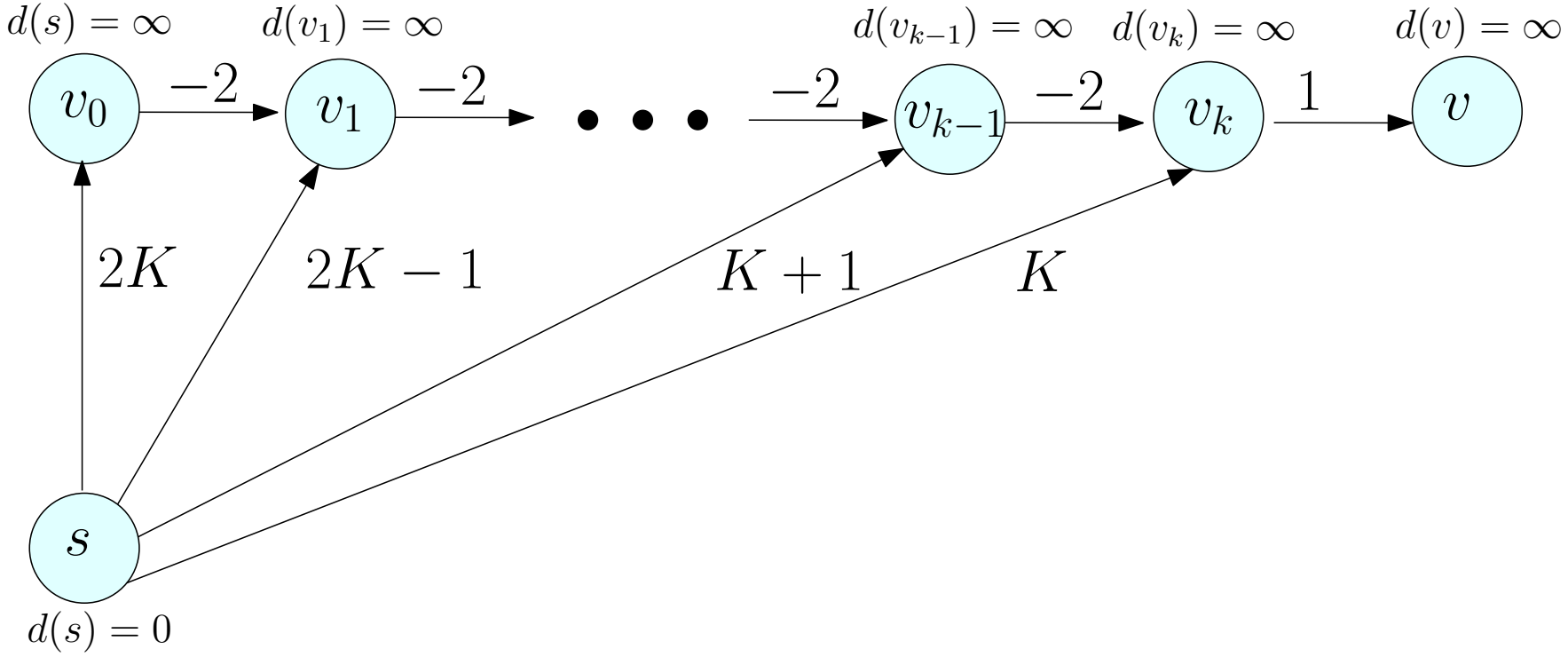


BFD: Tightness of Bounds and a Speeding Up Idea

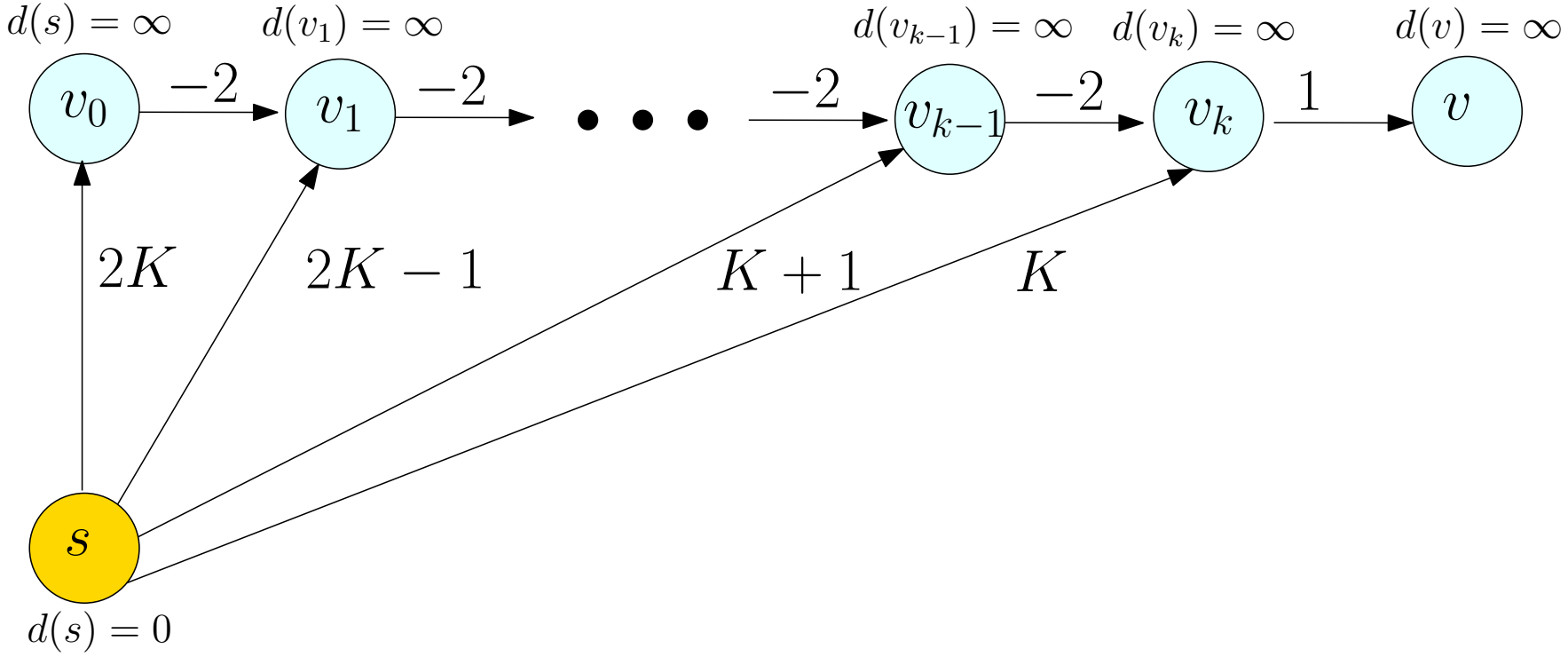
Consider the first Dijkstra scan:



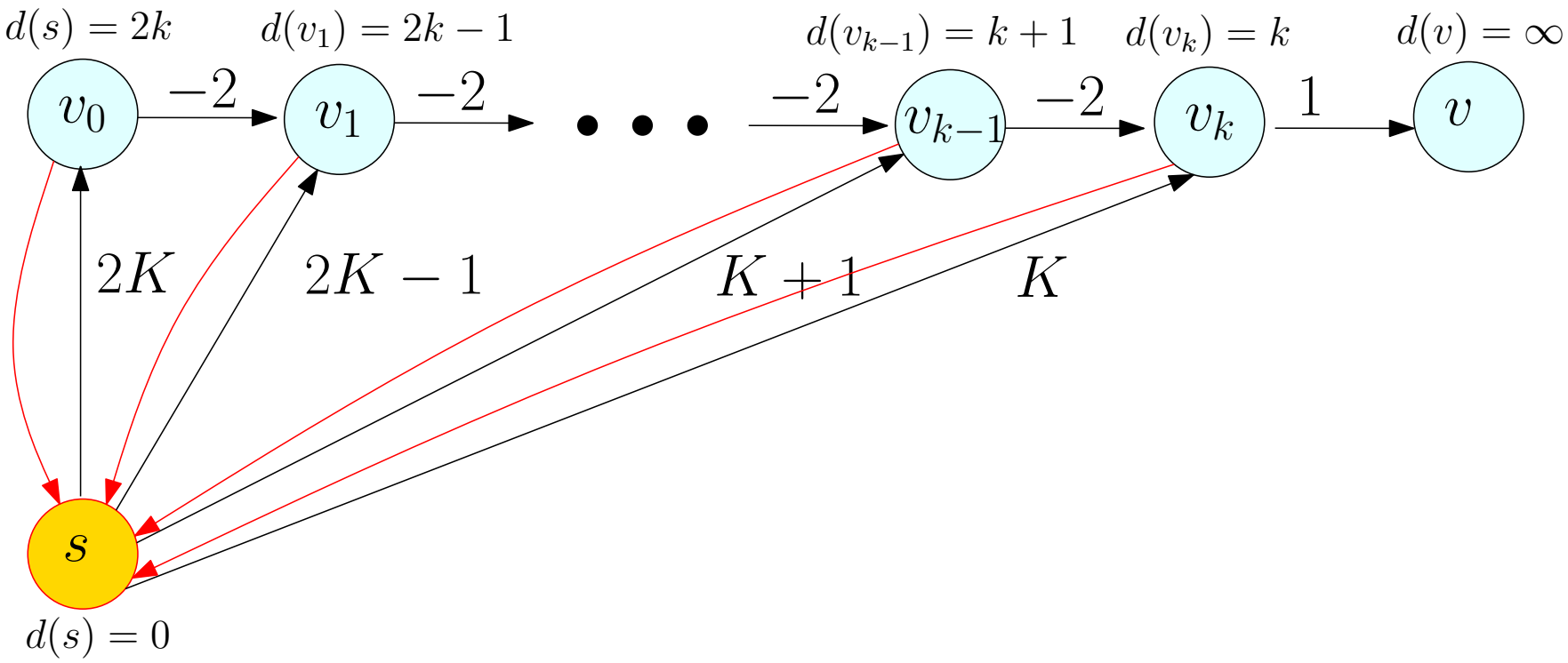
BFD: Tightness of Bounds and a Speeding Up Idea



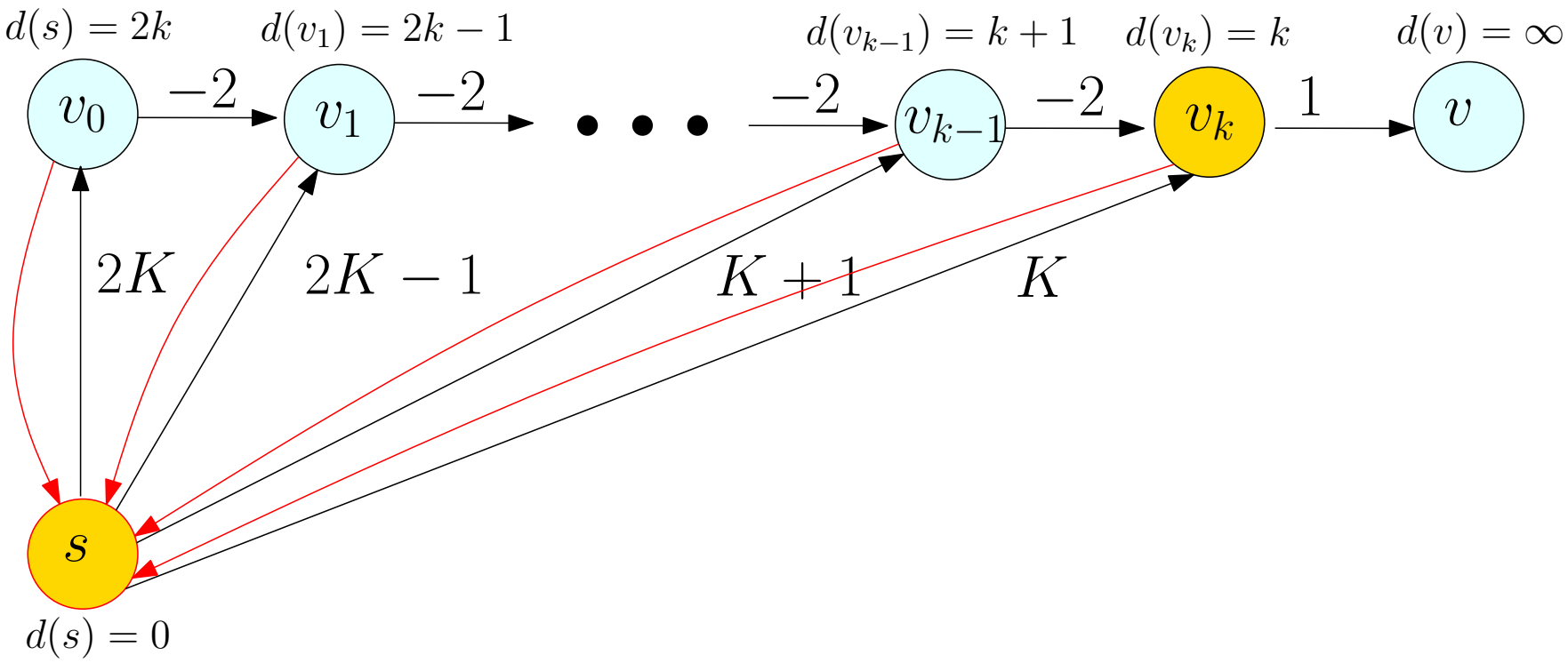
BFD: Tightness of Bounds and a Speeding Up Idea



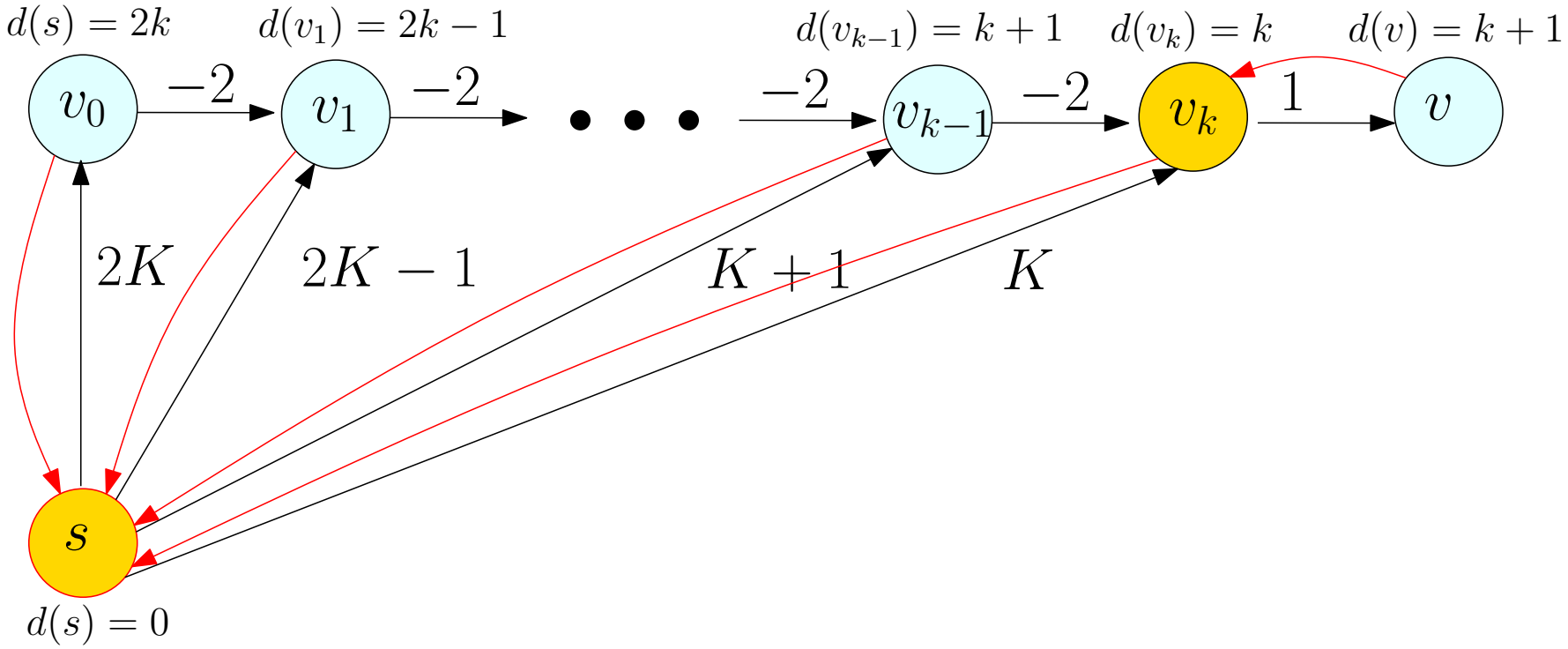
BFD: Tightness of Bounds and a Speeding Up Idea



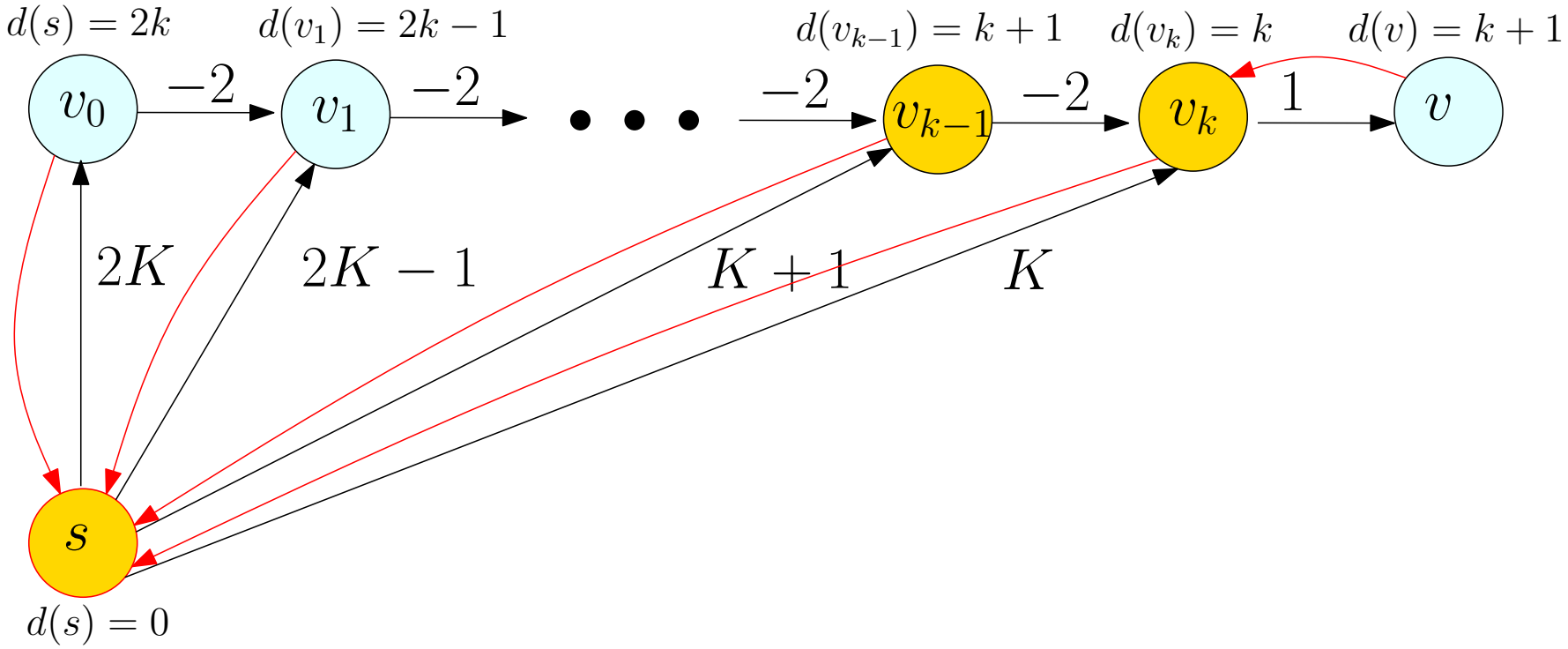
BFD: Tightness of Bounds and a Speeding Up Idea



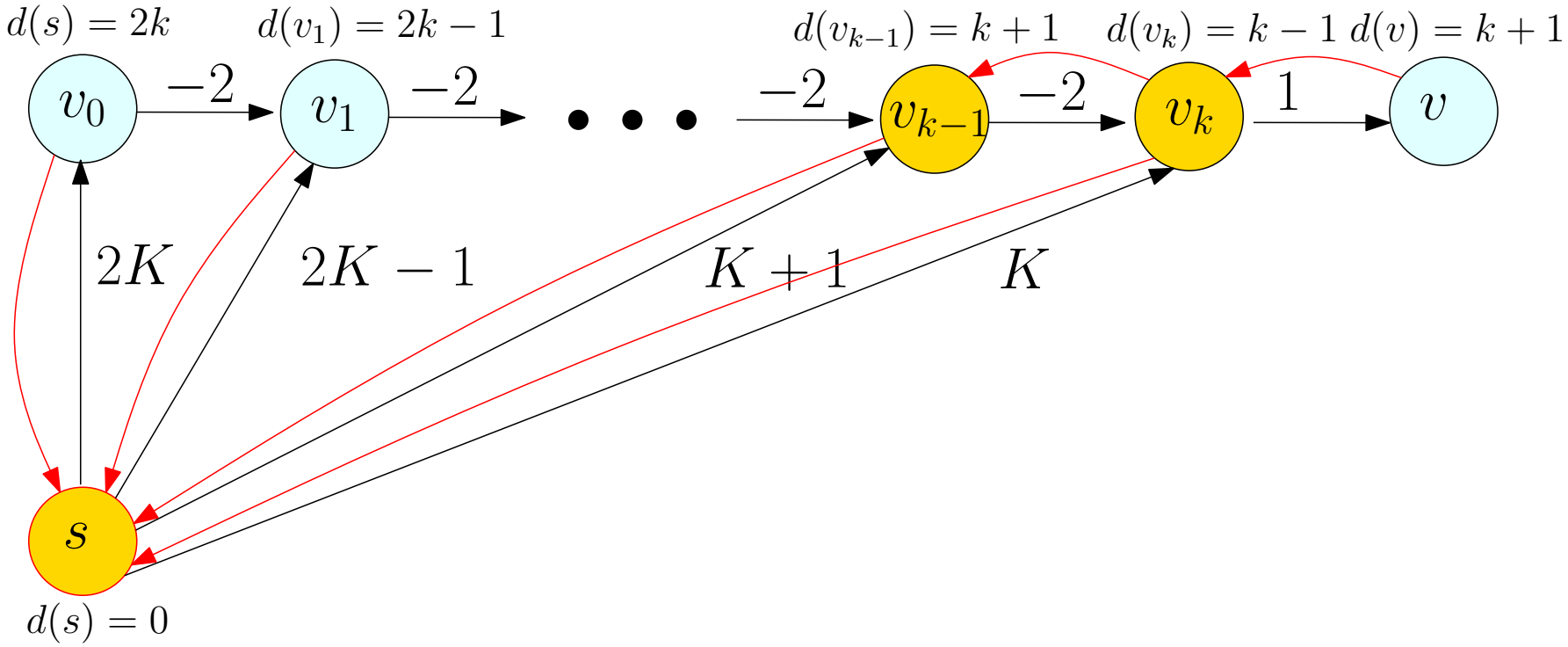
BFD: Tightness of Bounds and a Speeding Up Idea



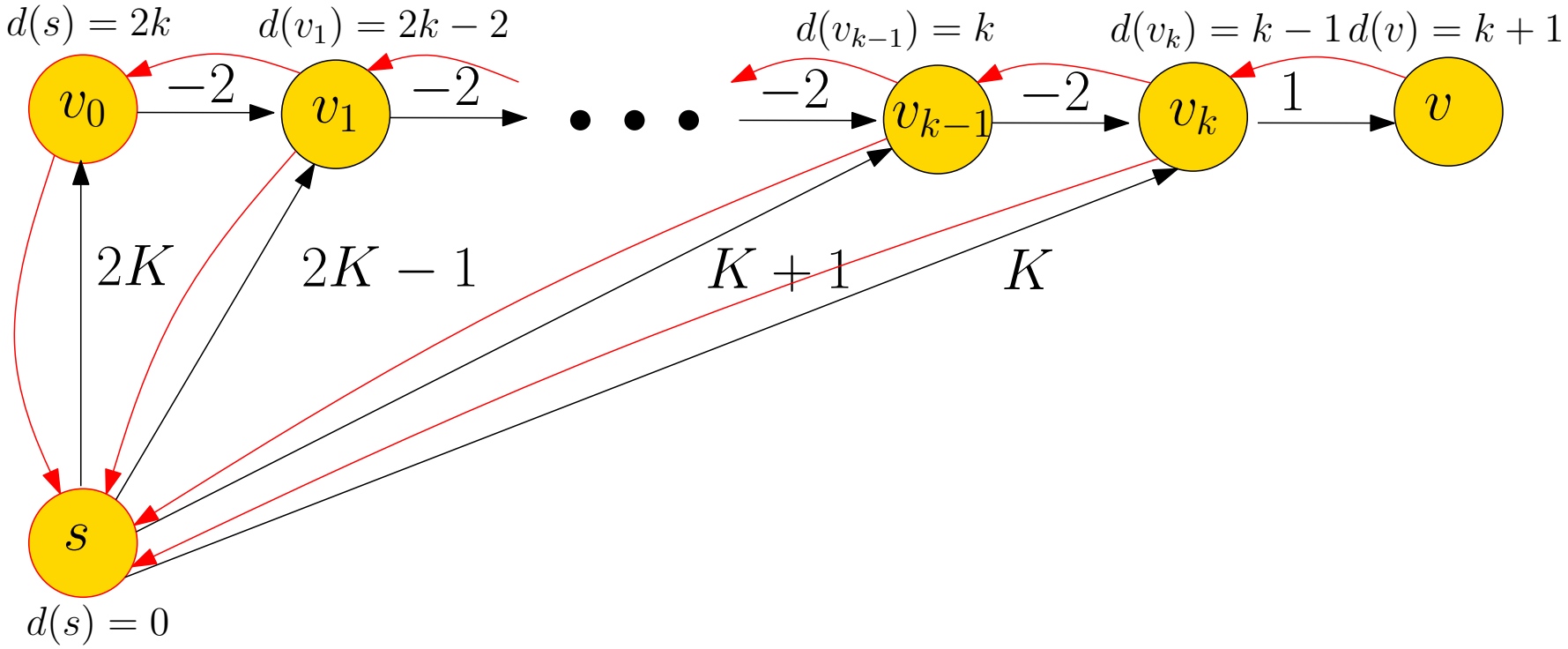
BFD: Tightness of Bounds and a Speeding Up Idea



BFD: Tightness of Bounds and a Speeding Up Idea

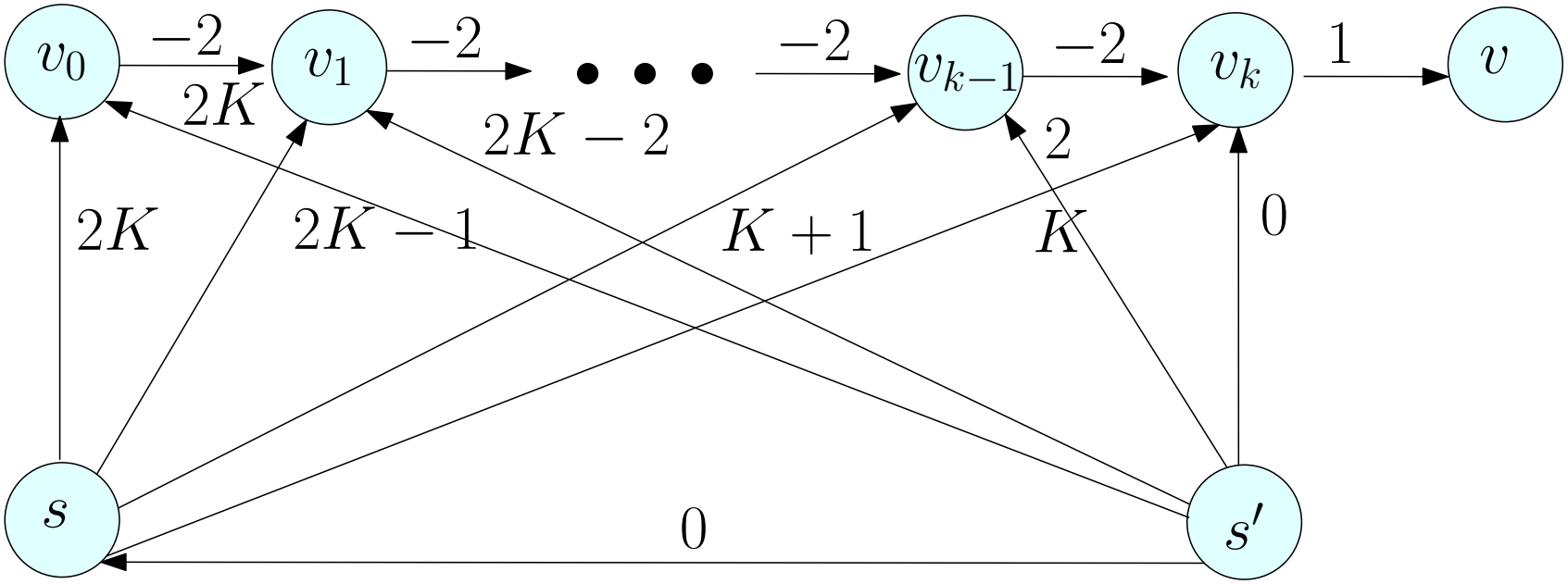


BFD: Tightness of Bounds and a Speeding Up Idea



BFD: Tightness of Bounds and a Speeding Up Idea

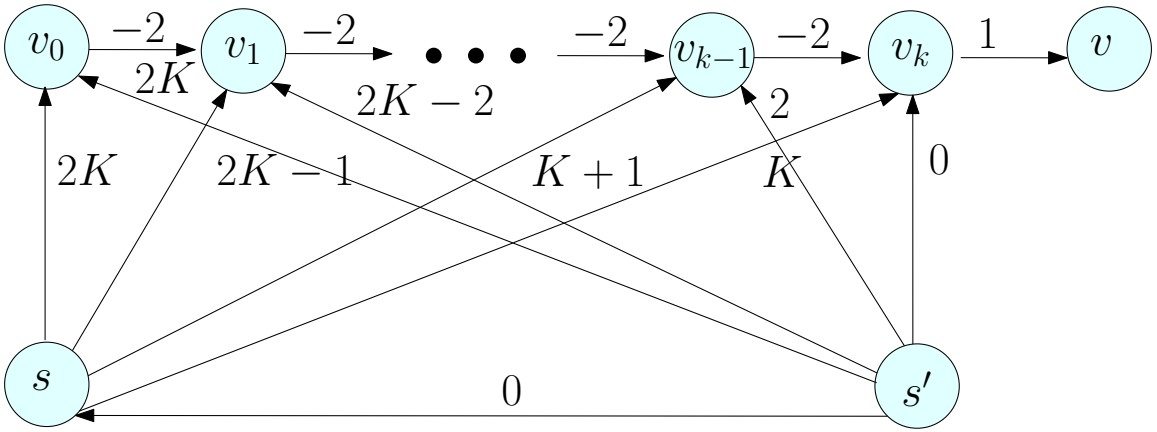
Consider the graph G:



BFD: Tightness of Bounds and a Speeding Up Idea

Notice that:

- $neg(G, s)$ remains equal to k.
- BFD running from s works exactly as above.
- Each edge (s', v_i) is an optimal path from s' to v_i. Hence, $neg(G, s') = 0$.
- Running BFD from s' would require just two rounds.
- All vertices reachable from s are reachable from s' in the new graph.



BFD: Tightness of Bounds and a Speeding Up Idea

The proposed algorithm:

1. Run BFD from s'.
2. Run **Johnson's reweighting technique**- obtaining a graph with non-negative costs of all edges reachable from s.
3. Run Dijkstra from s.

Running time: only three Dijkstra/Dijkstra scan executions are required, instead of $k + 2$ executions.

The End

Thank You!