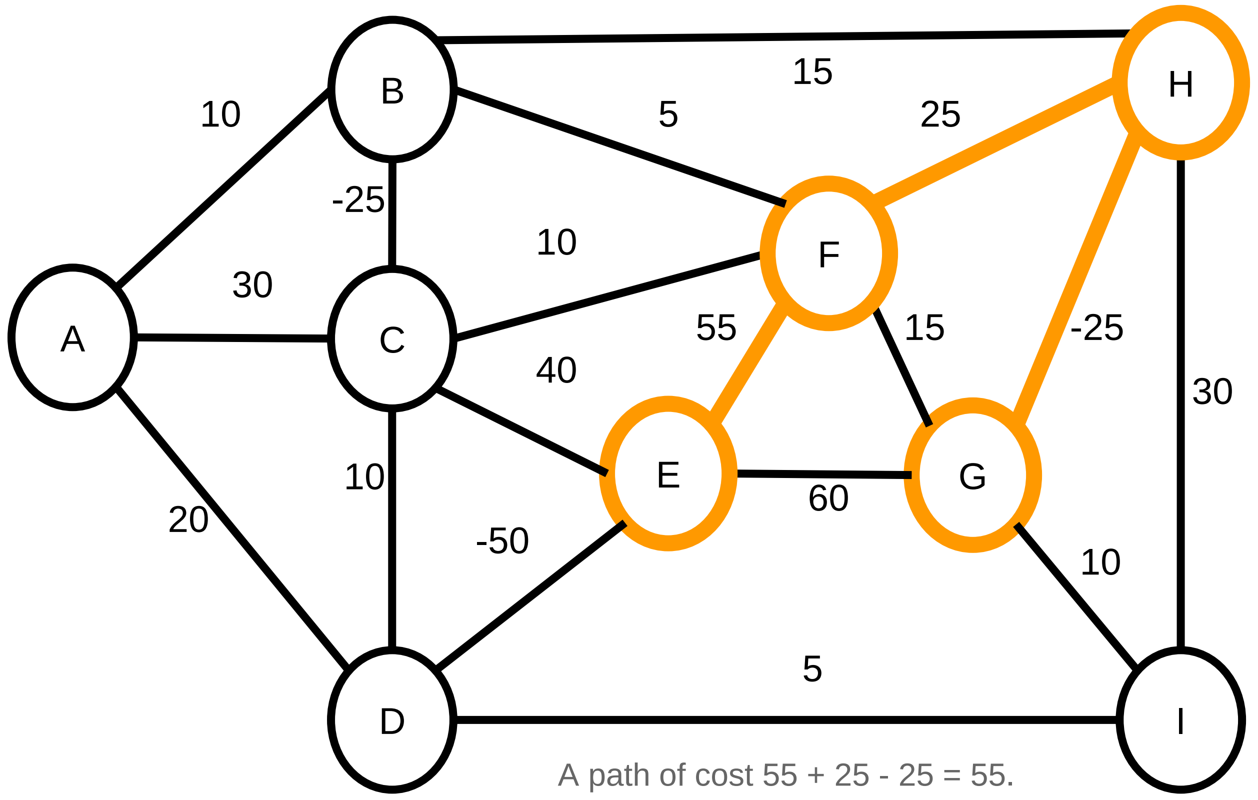
Shortest Paths Revisited

Lecture 07.08 by *Marina Barsky*

Paths with costs

In a weighted graph, the ***cost of a path*** is the sum of the weights (costs) of the edges along the path.



A path of cost 55 + 25 - 25 = 55.

A path from x to y is a *minimum-cost path* if it has the smallest cost among all paths from x to y.

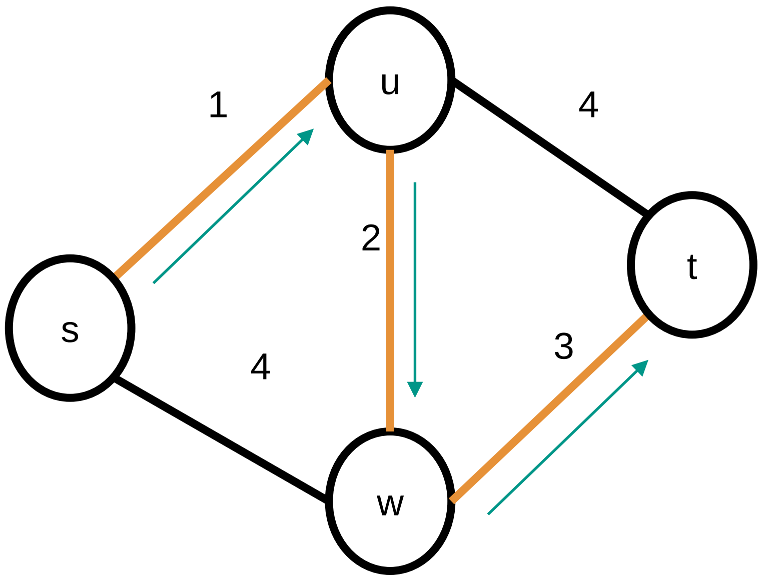
Recap:

Single-source Minimum-Cost Paths

without negative edge weights

Dijkstra Algorithm

Minimum Cost Paths: will simple greedy work?



The straightforward greedy approach: from each node on the path, take the edge with the smallest cost

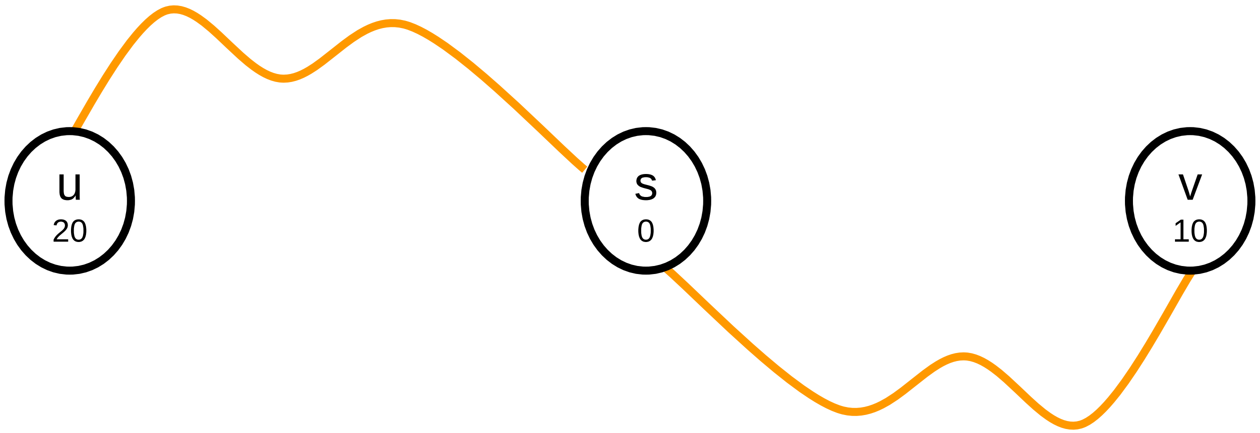
Is 6 the cost of the minimum-cost path from *s* to *t* ?

Simply taking the smallest-weight edge does not work.
Dijkstra algorithm is the combination of greedy and iterative improvement

Storing the Minimum Cost

We store the minimum cost from the start node in a `min_cost` array.

- The minimum cost from the start node to `x` is `min_cost[x]`.
- The start node `s` has `min_cost[s] = 0`.



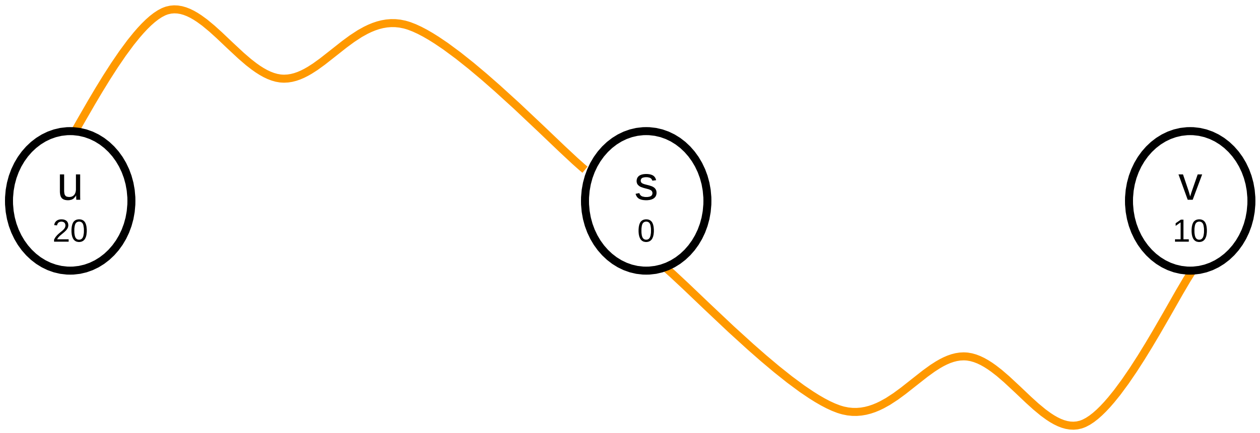
There is a path of cost 20 from s to u.
There is a path of cost 10 from s to v.

Question: Will we ever need to change a `min_cost` value during the algorithm?
Or will the value be set once and then never change?

Storing the Minimum Cost

We store the minimum cost from the start node in a `min_cost` array.

- The minimum cost from the start node to `x` is `min_cost[x]`.
- The start node `s` has `min_cost[s] = 0`.

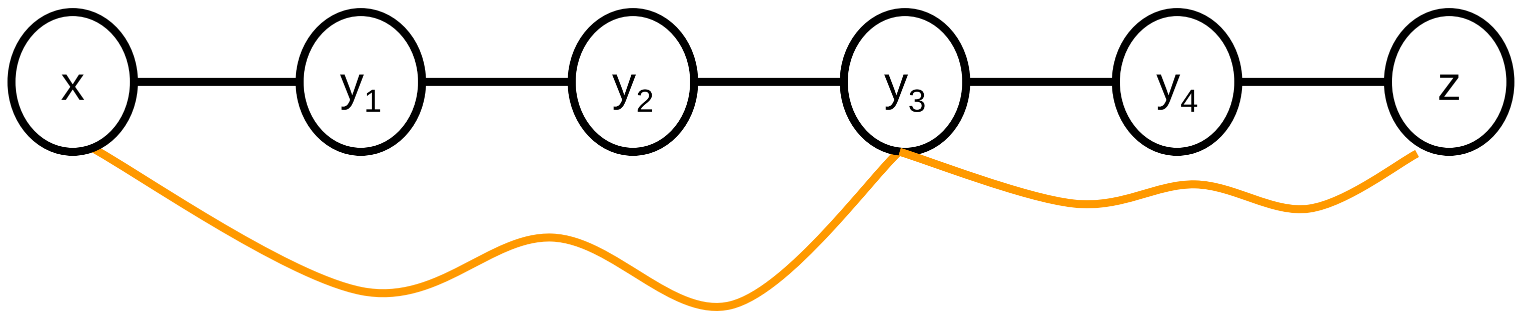


There is a path of cost 20 from s to u.
There is a path of cost 10 from s to v.

Question: Will we ever need to change a `min_cost` value during the algorithm?
Or will the value be set once and then never change? **The value can change**

Property of Minimum-Cost Paths

Suppose that a minimum cost path from x to z goes through the nodes y_1, y_2, …, y_k. Notice that the subpath from x to y_i is also a minimum cost path from x to y_i for all i.



A minimum cost path from x to z. One of its subpaths is a path from x to the intermediate node y_3.

If there is another path from x to y_3 that has lesser cost than this subpath, then the original path from x to z was not minimum cost (could have been improved).

Therefore, if we want to build minimum cost paths, then we never want to extend a path that is not itself minimum-cost.

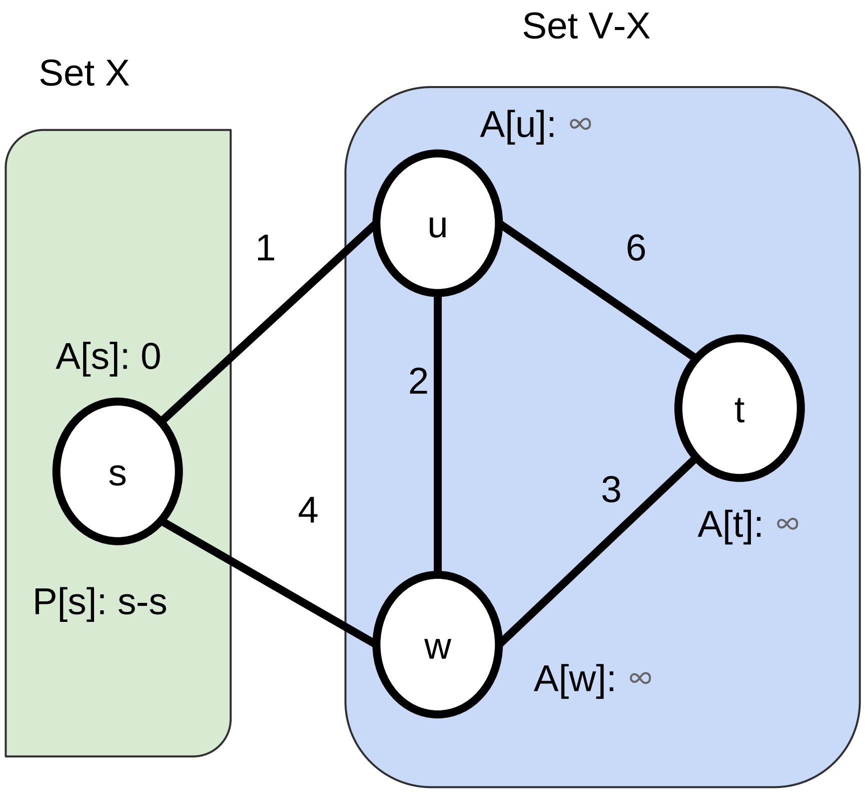
- Don't add node to the solution until we know that we have a minimum cost path to it.
- We need to keep track of the current minimum cost path to each node.
- We will compute the shortest path from 'source' node x to all other nodes y.

Dijkstra Algorithm: intuition

We maintain 2 sets of nodes:
Set *X* for nodes for which we already know the final cost of the min-cost paths from *S* (**Processed**).
Set *V*-*X* of remaining nodes for which the min-cost path is yet to be found (**Unprocessed**).

- We perform *n* iterations of the main loop:
    - At each iteration we choose one node from *V*-*X* and add it to *X* with its corresponding cost (and path if required).
    - The node is chosen according to the minimum *Dijkstra Greedy Score* (***DGS***).
    - We store the current greedy score for vertex v in the *min_cost* array
    - We grow set *X* until all *n* vertices from *V* are added to X.

Dijkstra algorithm: illustration



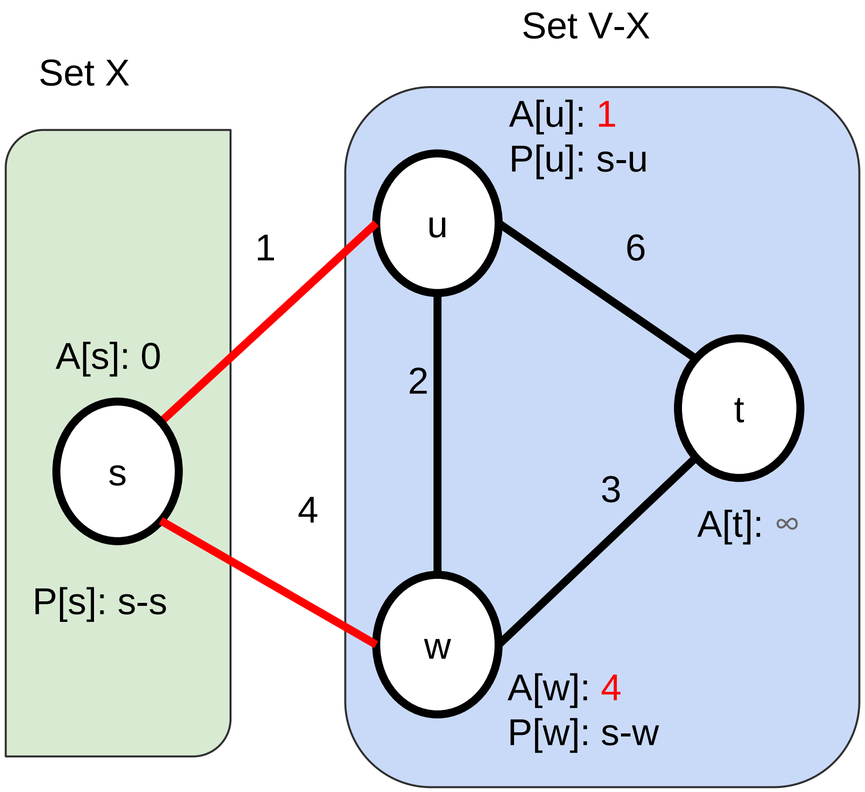
A is a *min_cost* array containing DGS for each of *n* nodes

P is an array containing min-cost paths from *s* to each of *n* nodes

- Originally, only the source vertex S is in X: the cost of the path S-S is 0.
- For paths from S to other nodes the cost is unknown, we mark them as ∞.
- At each step, there will be edges inside X, inside V-X, and the edges between the 2 sets.

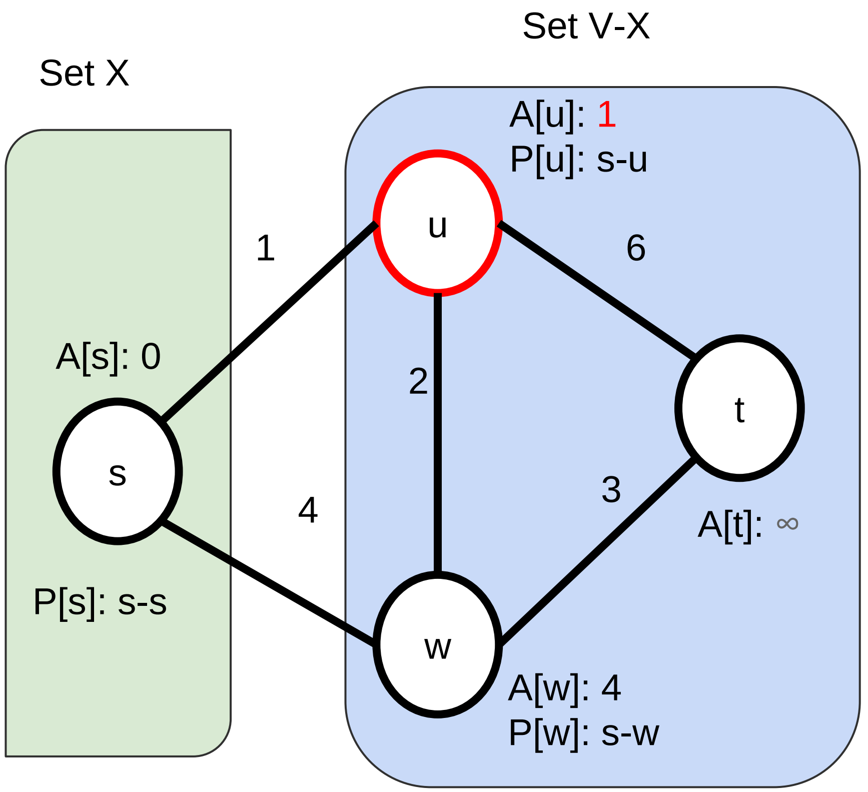
- **We are interested only in edges that "cross the border" - they will allow us to improve the DGS for each remaining node**

Dijkstra algorithm: illustration



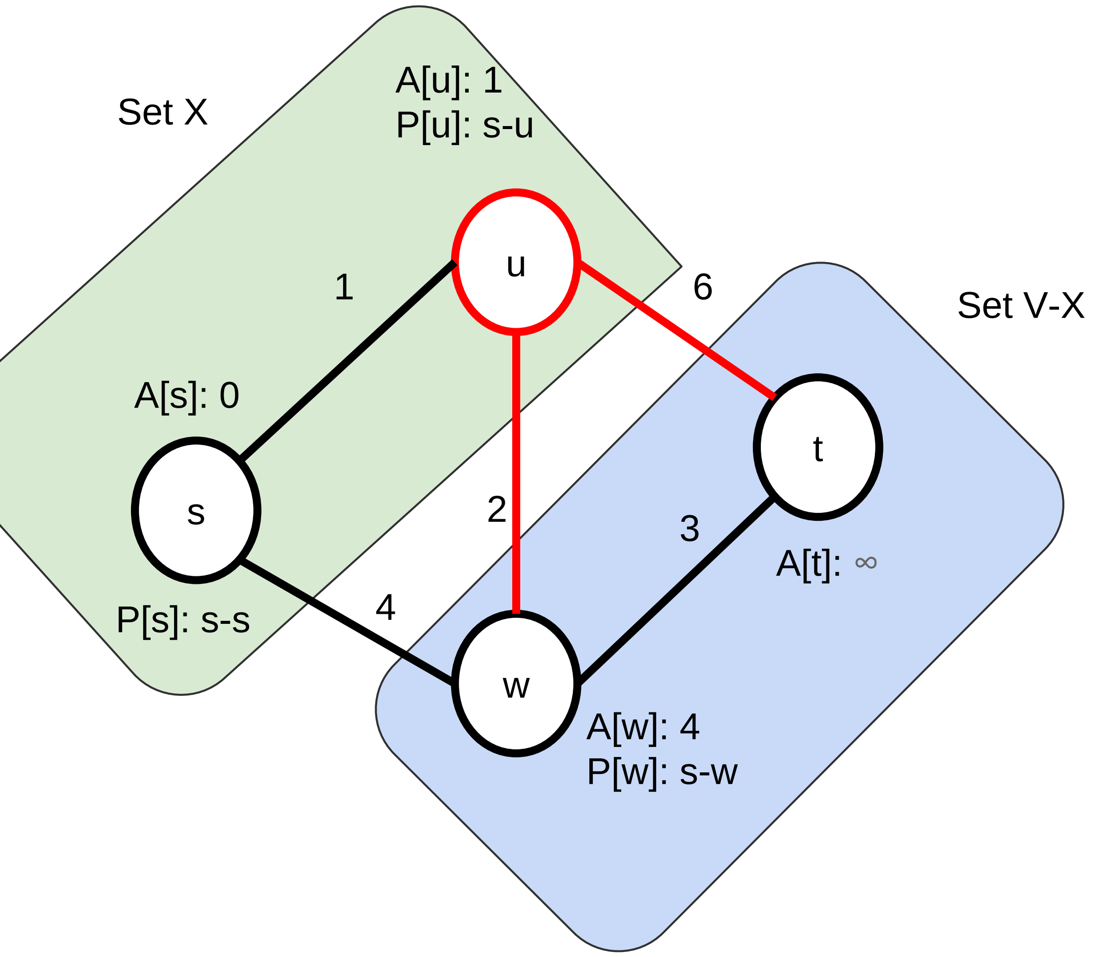
- The goal is to add more nodes to X.
- The only 2 edges extending already known min-cost path are (s,u) and (s,w).
- For both u and w, we update their DGS to the sum of A[s] + cost(s,u) and A[s] + cost(s,w) respectively.
- This will be a new Dijkstra Greedy Score for these nodes.

Dijkstra algorithm: illustration



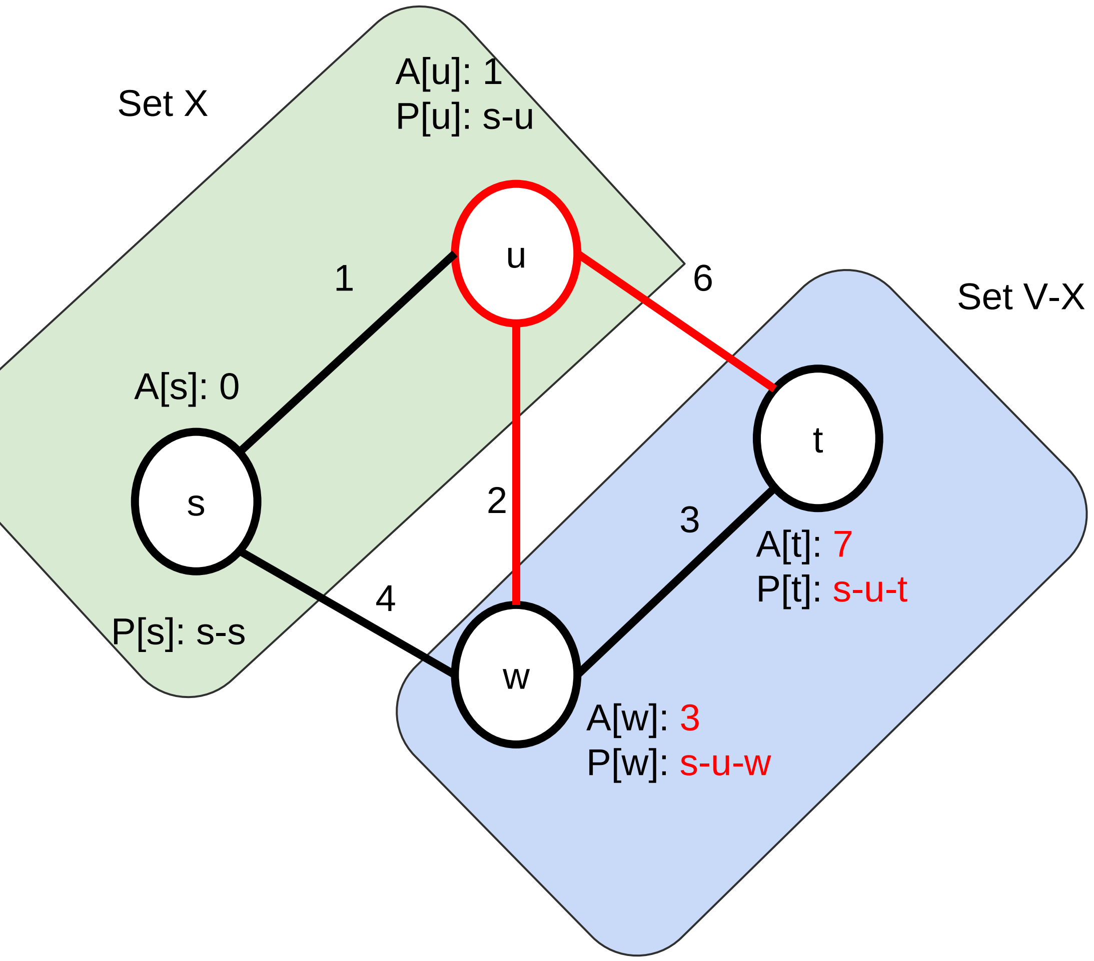
- Next, we select the vertex with the minimum DGS - vertex u - and add it to X

Dijkstra algorithm: illustration



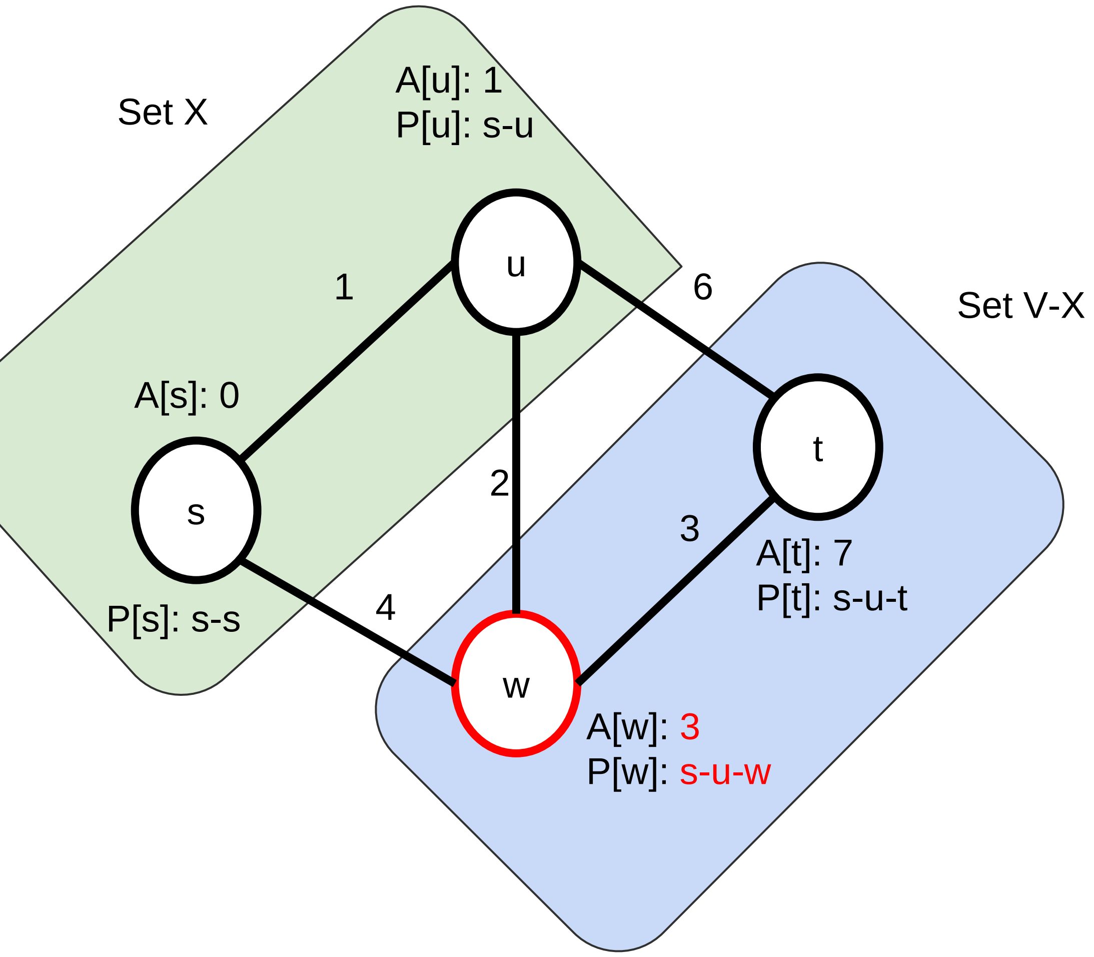
- Now we have a new node u in X, and we know that s~>u is the next smallest min-cost path from s
- There are 2 new edges out of u which cross the border between X and V-X
- They may help improve the DGS of remaining nodes

Dijkstra algorithm: illustration



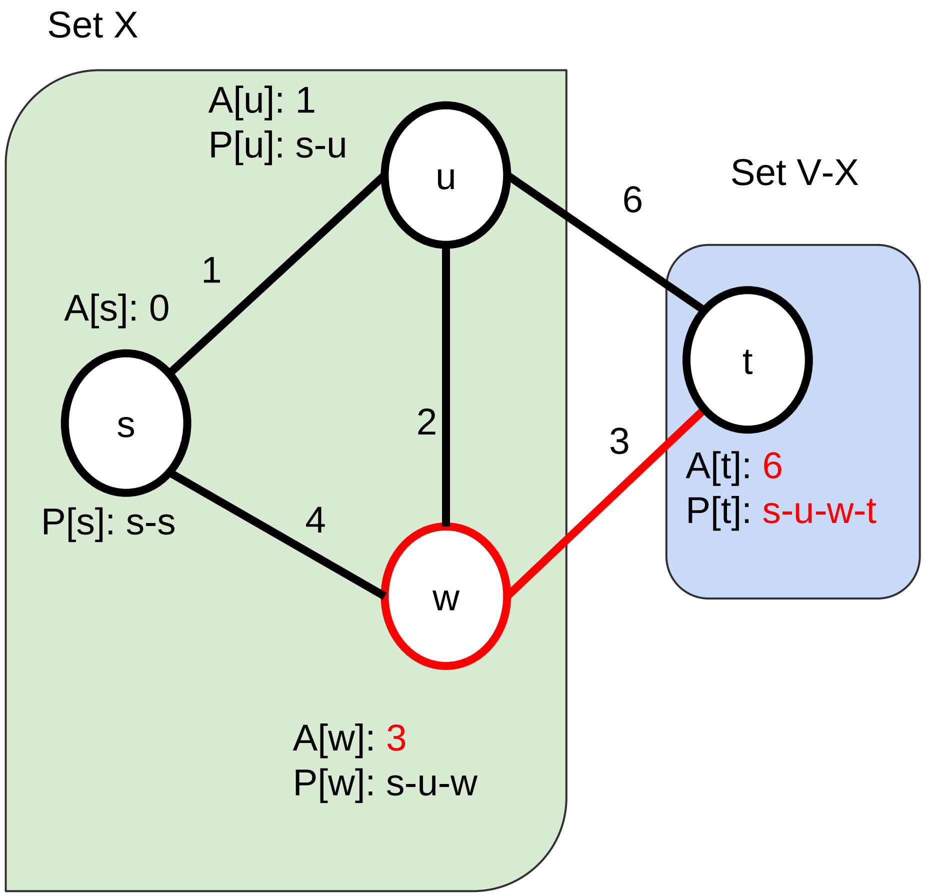
- We check if we can update the DGS using A[u] + cost(u,t) and A[u]+cost(u,w)

Dijkstra algorithm: illustration



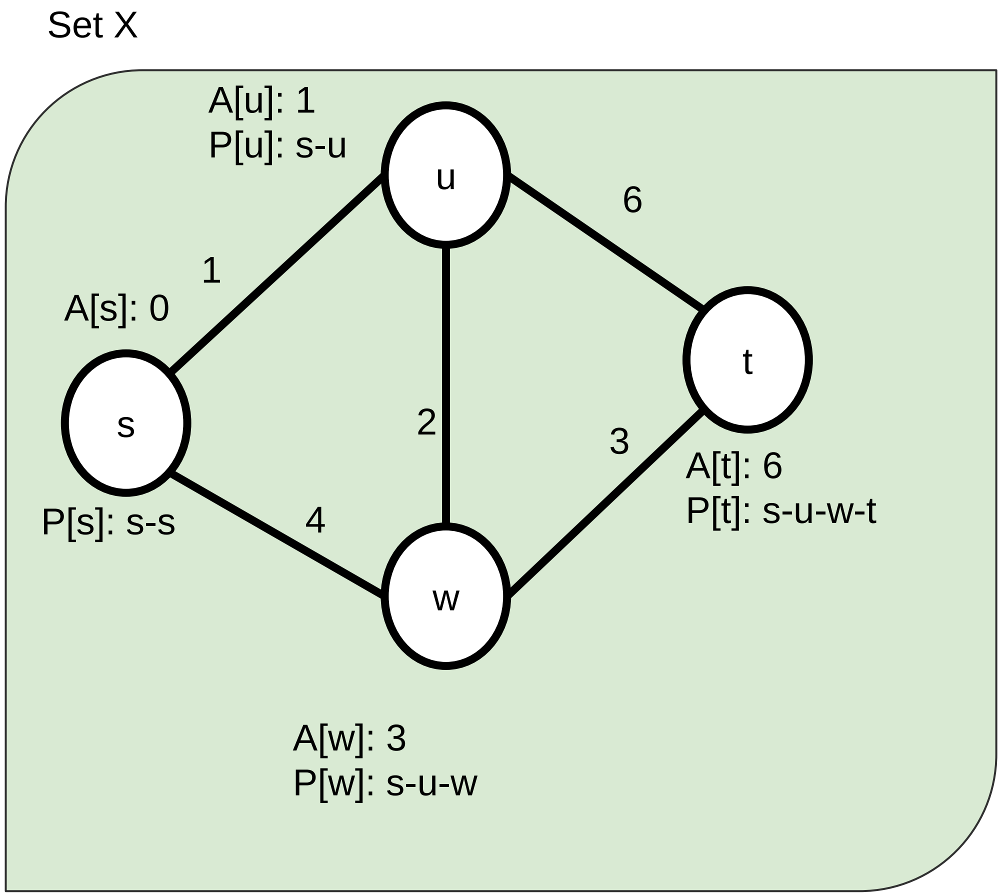
- Next we select the node with the smallest DGS and add it to X

Dijkstra algorithm: illustration



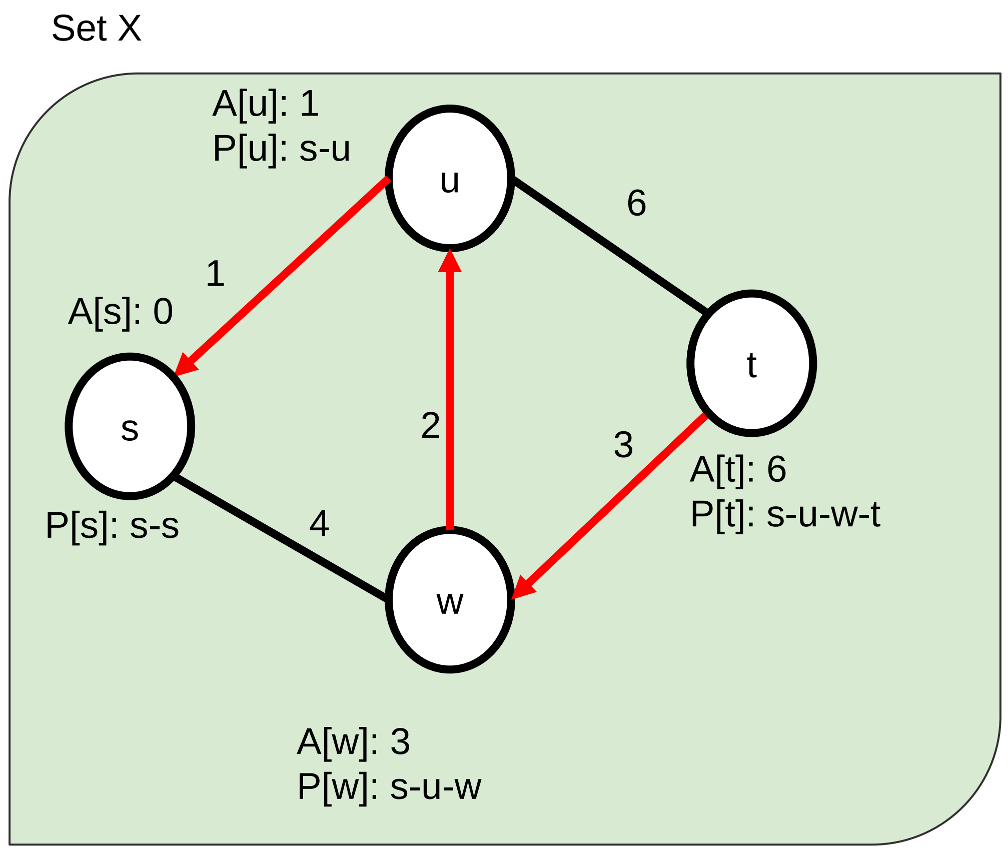
- The only new edge that can update DGS for t is (w,t).
- We check if the new score going through w is better, and update the final score of t to A[w] + cost(w,t)

Dijkstra algorithm: illustration



- The last vertex is added to X
- At this point all min-cost paths from s to each other vertex have been computed

Dijkstra algorithm: the paths



- Do we really need to store the paths themselves?

- No, instead of storing the min path for each node, we could just record the parent node when we update DGS, and we will be able to recover the shortest path from any node to s

Dijkstra Algorithm: correctness

Intuitively: the algorithm is correct because we transfer the node v into set X by extending the shortest paths from the nodes for which we already know that the paths from s are optimal.

Proof by induction (sketch):

- Base case: A[s] = 0
- Inductive hypothesis:
  for all v ∈ X, A[v] is the cost of the shortest path s~>v
- In each iteration:
  We pick the vertex w ∉ X with the lowest DGS among all vertices ∉ X.
  The path from s to w extends some shortest path s~>v for some v ∈ X.
  We updated the DGS(w) with the lowest possible cost of extending any such path
- Then any alternative path from s to w which we did not explore yet must go through some vertex z in V-X. But for any z, DGS(z) ≥ DGS(w), so any such path will have the cost at least A[w] (not shorter).
- Hence, if we assume that each path from s to v ∈ X was a shortest path, the extension of one of such paths will be a shortest path too.

A full formal correctness proof of Dijkstra's algorithm can be found here

Pseudocode

```
Algorithm Dijkstra(G, start)

    unprocessed: = empty set
    min_cost:= empty dictionary
    for each u in vertices of G
        min_cost[u]: = ∞
        unprocessed.add(u)

    min_cost[start]: = 0
    processed: = empty set
    processed.add(start)

    while unprocessed is not empty
        v: = remove v with min_cost from unprocessed
        processed.add(v)
        for each edge (v,u)
            if u in unprocessed:
                min_cost[u]: = min(min_cost[u], min_cost[v] + $w_{v,u}$)
```

Naive Dijkstra Algorithm

Running time of Dijkstra's Algorithm

```
Algorithm Dijkstra(G, start)

    unprocessed: = empty set
    min_cost:= empty dictionary

    for each u in vertices of G
        min_cost[u]: = ∞
        unprocessed.add(u)

    min_cost[start]: = 0
    processed: = empty set

    while unprocessed is not empty
        v: = remove v with min_cost from unprocessed
        processed.add(v)
        for each edge (v,u)
            if u in unprocessed:
                min_cost[u]: = min(min_cost[u], min_cost[v] + $w_{v,u}$)
```

Loop is executed O(n) times

Search for min in set of size O(n)

Each node may have degree O(n) - but total O(m) edges to process

The running time: n*n + m = $O(n^2)$

Recap: Min-Priority Queue

A *min-priority queue* is an ADT for fast retrieval of min element.
Implementations: binary heap, balanced BST, *Fibonacci heap* (retirieval in time O(1) but large constants).
For Dijkstra Priority Queue ADT should be enhanced with the *`update*`* operation.

	Priority queue
enqueue	O(log n)-time
dequeue	O(log n)-time
update	O(log n)-time

*The *`update`* operation decreases the associated value of a given item.
In other words, it increases its priority.
We can keep pointers to each queue node to locate it quickly.
However if the priority of the heap node changed, we need then rebalance the heap

Dijkstra's Algorithm with Priority Queue

The `min_cost` priority queue (`min_pq`) stores tuples `(node, DGS)` prioritized by `DGS`.

```
Algorithm Dijkstra Improved(G, start)

    min_pq:= empty priority queue
    for each u in vertices of G
        min_pq.enqueue((u,∞))

    processed: = empty set
    min_pq.update((start, 0))

    while min_pq is not empty
        (cost_v, v): = min_pq.dequeue()
        processed.add(v, cost_v)

        for each edge (v,u):
            if u in min_pq:      # we have pointer to each node in the extended priority queue
                cost_u:= min_pq.get(u).cost
                if cost_v + $w_{v,u}$ < cost_u:
                   min_pq.update(u, cost_v + $w_{v,u}$)
```

Dijkstra's with Priority Queue: running time

```
Algorithm Dijkstra Improved(G, start)

    min_pq:= empty priority queue
    for each u in vertices of G
        min_pq.enqueue((u,∞))

    processed: = empty set
    min_pq.update((start, 0))

    while min_pq is not empty
        (cost_v, v): = min_pq.dequeue()
        processed.add(v, cost_v)

        for each edge (v,u):
            if u in min_pq:
                cost_u:= min_pq.get(u).cost
                if cost_v + $w_{v,u}$ < cost_u:
                    min_pq.update(u, cost_v + $w_{v,u}$)
```

Loop is executed O(n) times

Each dequeue in time log n

In sum O(m) edges to process

Quickly finds `u` in `min_pq`, and updates only if new DGS is better: rebalance in time O(log n)

Running time O(n log n) + O(m log n) = **O(m log n)**

Dijkstra Algorithm: running time

Running time with Priority Queue: $O(m \log n)$

- If $m = O(n)$ [sparse graphs], then running time $O(n \log n)$
- If $m = O(n^2)$ [dense graphs], then running time is $O(n^2 \log n)$

Dijkstra Algorithm: non-negative weights

- The algorithm combines ideas from both greedy and iterative improvement techniques
- It iteratively improves DGS of each node until no more improvement is possible and at this point the node is transferred into a processed set X
- However if edges are allowed to have a negative cost, then some of them could potentially improve the DGS of already processed nodes (including the source node s!)
- Then we would never have the processed set to start with
- Therefore this algorithm is not applicable for graphs with negative edge weights

Single-Source shortest paths with positive **and negative** edge costs

Bellman-Ford Algorithm

Dynamic Programming

Negative edge costs

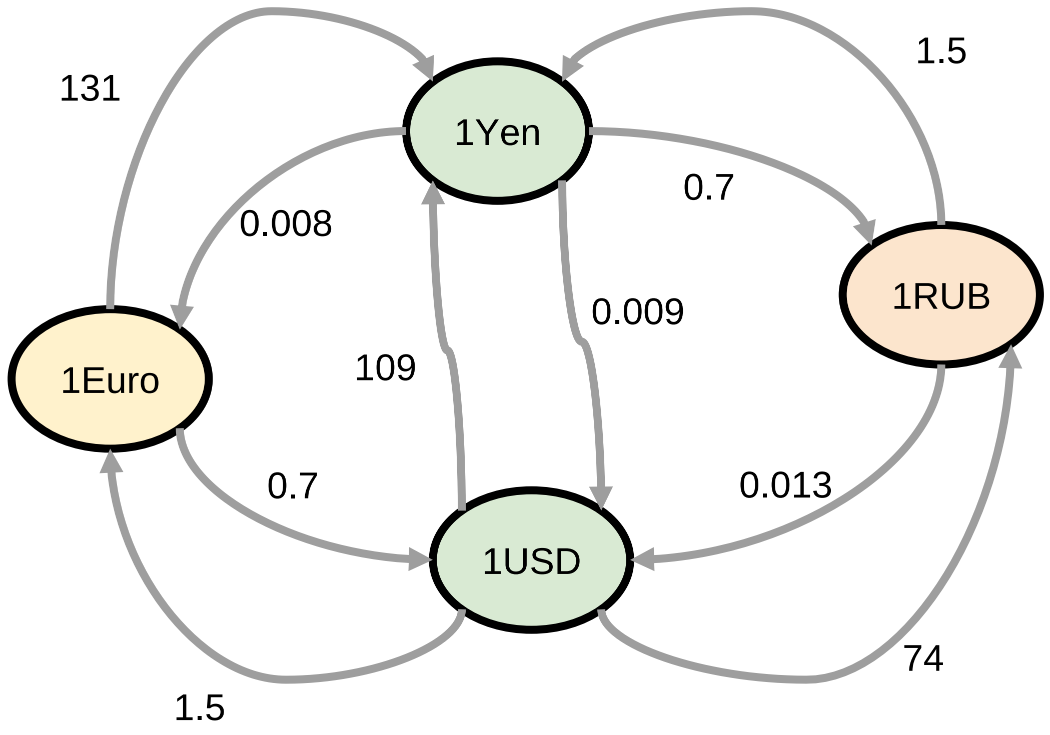
It is probably hard to imagine the cases in physical world when the costs of edges are negative: think of a network of roads

However graphs model many different problems

In *decision problems modeled with graphs* we can easily get negative costs (penalties) and positive costs (rewards)

The problem then is to find the shortest (min-cost) path that minimizes overall penalties – to make the best possible sequence of decisions

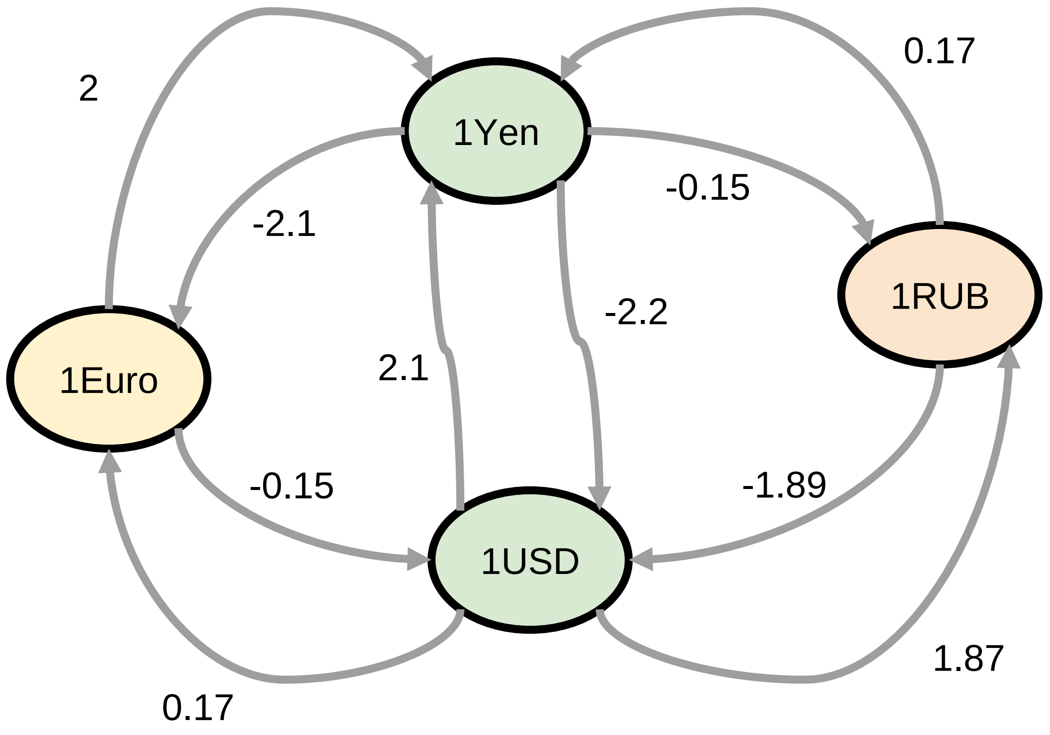
Example of a graph with negative edge weights



Prices for buying and selling currencies.
These are conversion rates

Goal: find the best way to convert from RUB to EURO
Note that we need to multiply here

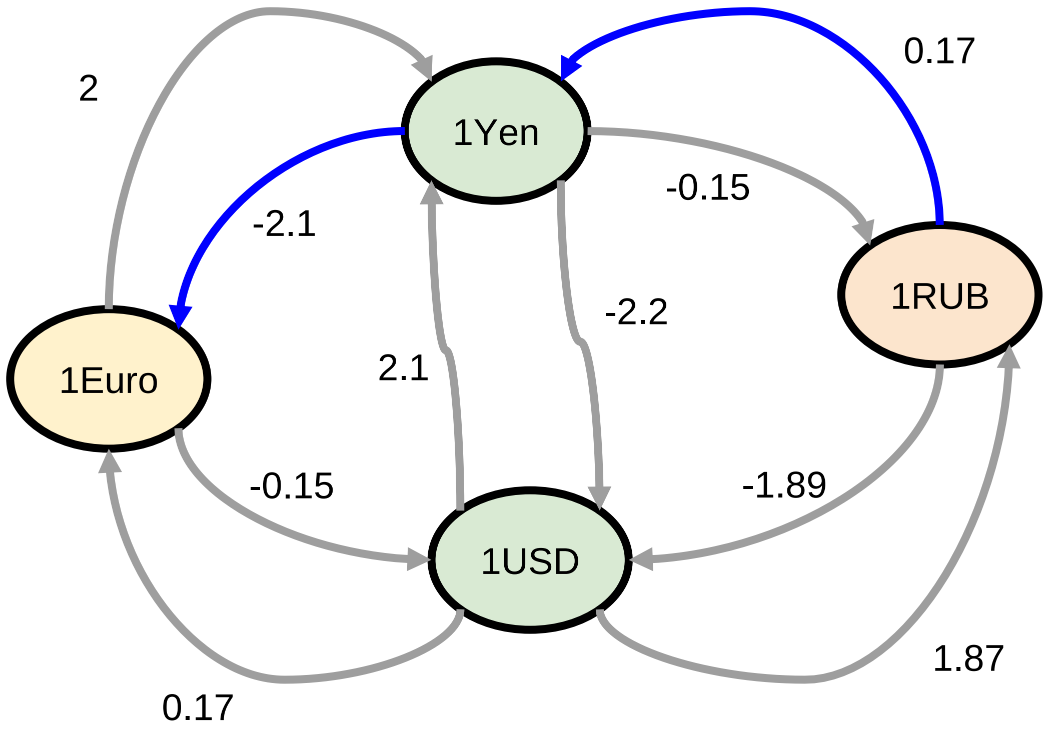
Example of a graph with negative edge weights



To reduce the problem to the shortest path problem:
Represent weights as logs of conversion rates

Now the product will become a sum, and we can compute the shortest path.
However some weights are negative!

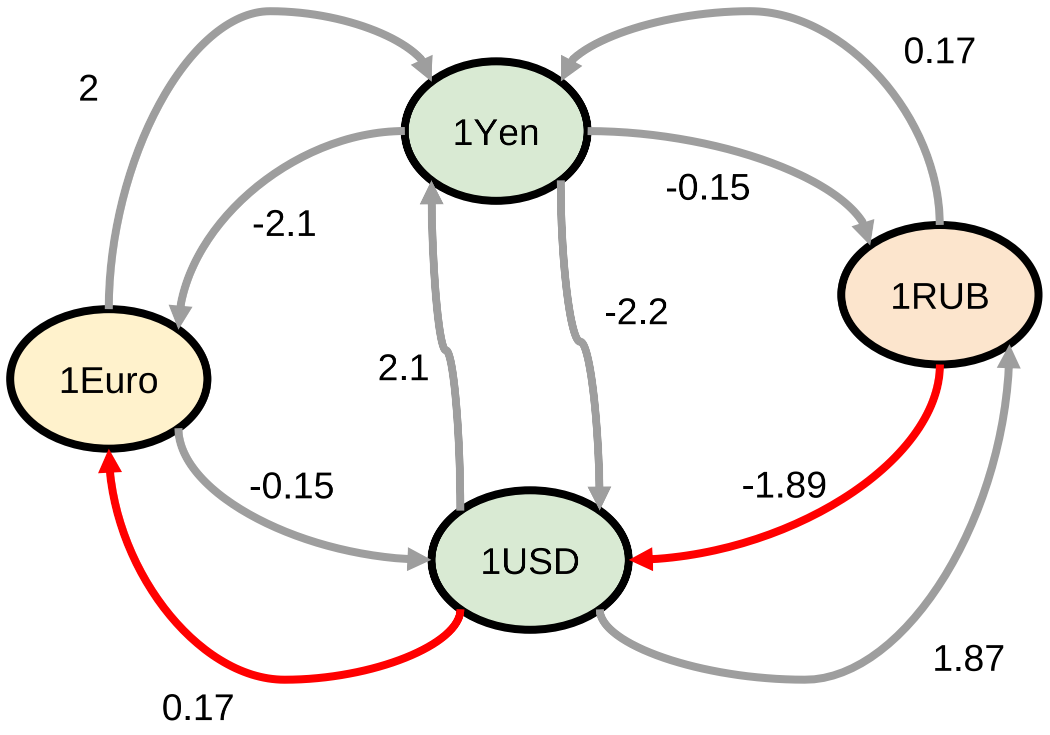
Example of a graph with negative edge weights



What is the best path from RUB to EUR?

0.17 - 2.1 = -1.93

Example of a graph with negative edge weights

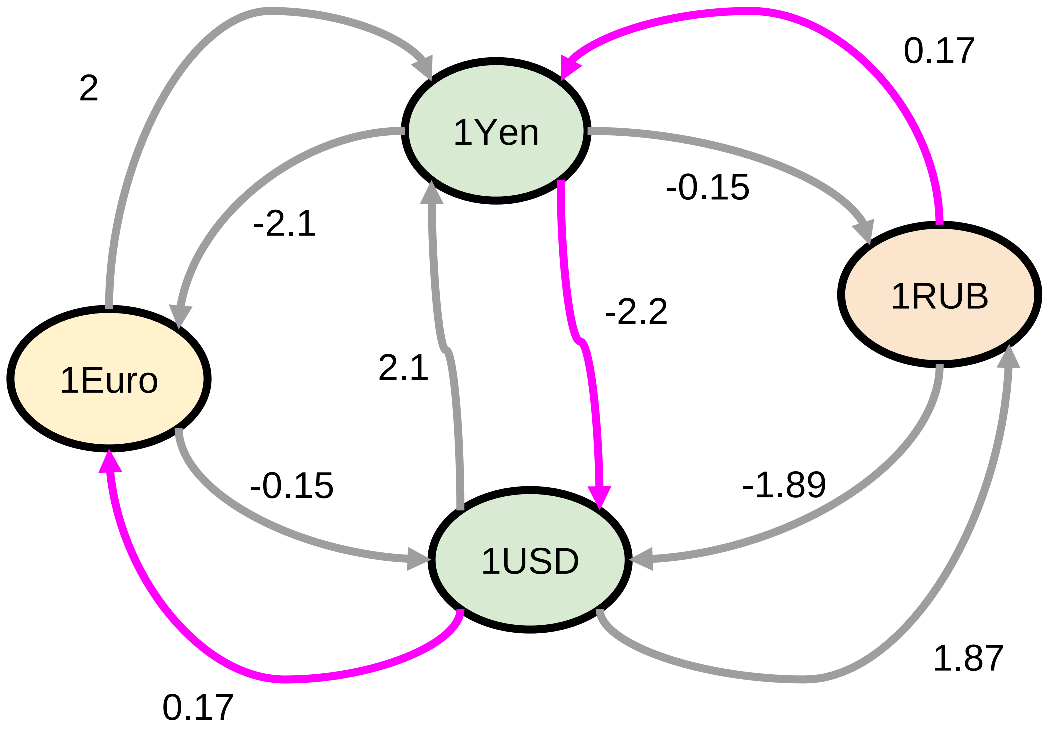


What is the best path from RUB to EUR?

0.17 - 2.1 = -1.93

-1.89 + 0.17 = -1.72

Example of a graph with negative edge weights



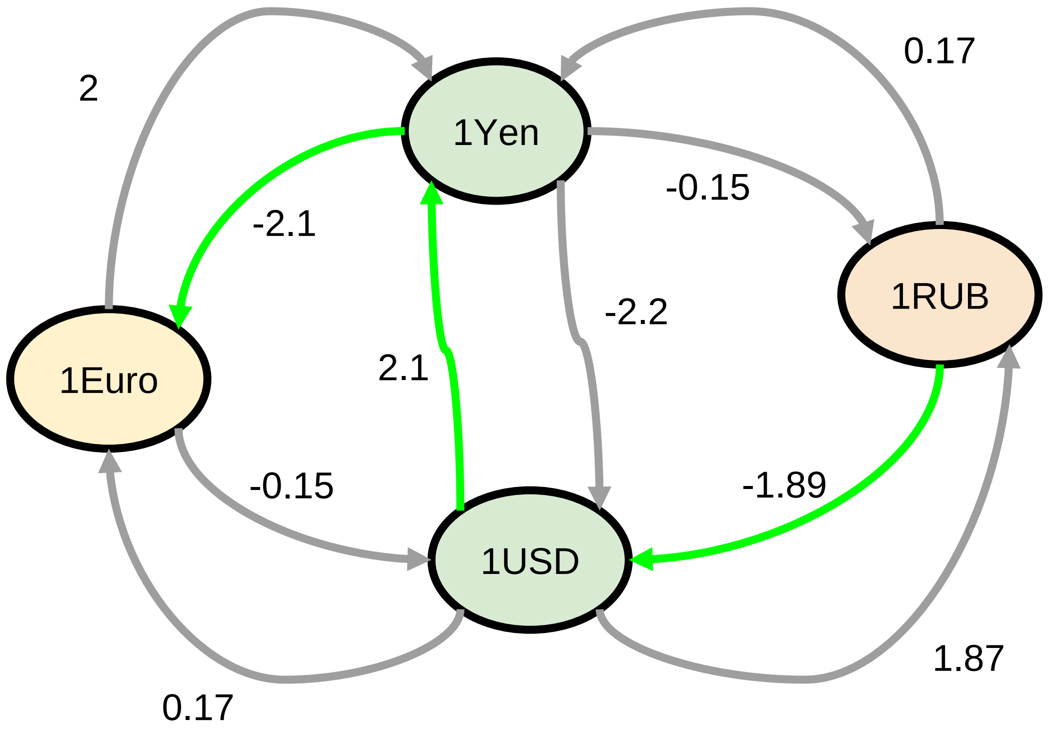
What is the best path from RUB to EUR?

0.17 - 2.1 = -1.93

-1.89 + 0.17 = -1.72

0.17 - 2.2 + 0.17 = -1.86

Example of a graph with negative edge weights



What is the best path from RUB to EUR?

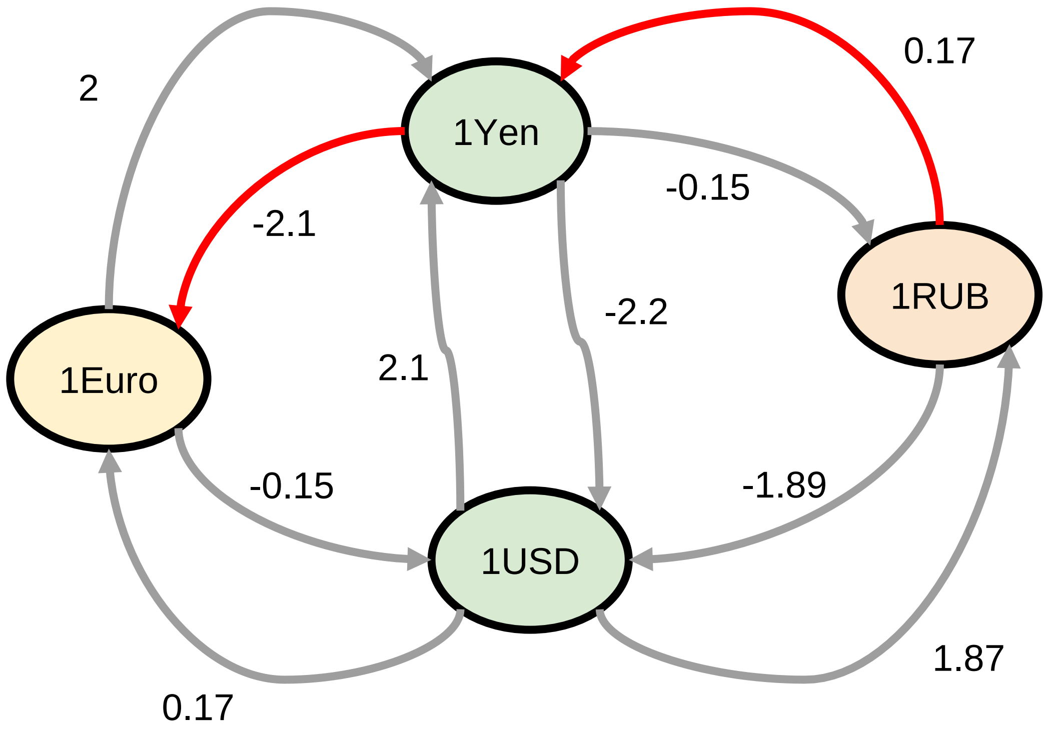
0.17 - 2.1 = -1.93

-1.89 + 0.17 = -1.72

0.17 - 2.2 + 0.17 = -1.86

-1.89 + 2.1 - 2.1 = -1.89

Example of a graph with negative edge weights



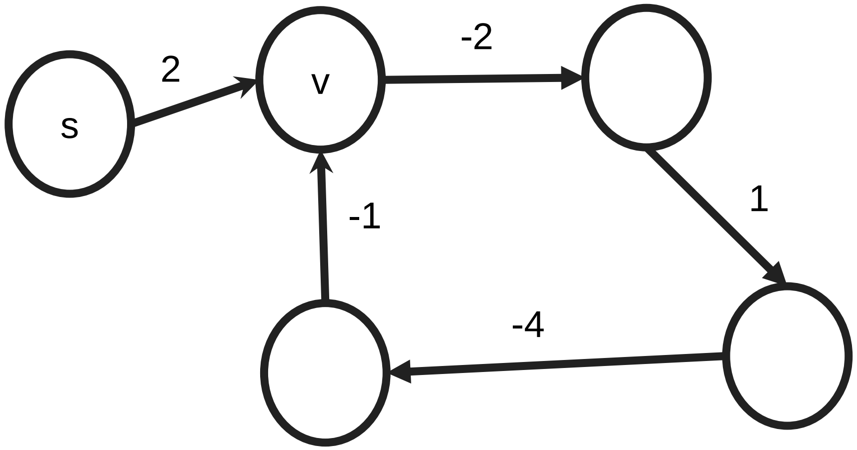
The min-cost path:
0.17 - 2.1 = -1.93
-1.89 + 0.17 = -1.72
0.17 - 2.2 + 0.17 = -1.86
-1.89 + 2.1 - 2.1 = -1.89

Luckily we have only 4 nodes:
Dijkstra does not work here!

Use Bellman-Ford

Negative edge costs: problem!

- If we allow some weights be negative, we facing the problem of a **negative cycle:** a cycle with the total cost < 0
- All shortest-path algorithms based on iterative improvement will fail here, because the cost of a path can be improved indefinitely!
- We may think of limiting the search to paths that avoid traversing cycles, but that leads to even bigger problem:
  - If we do not allow paths to use cycles, we are asking for something which is called *a simple path*: a path that repeats no vertex.
  - This is nothing else but a Hamiltonian Path – and no efficient algorithm is known for computing it

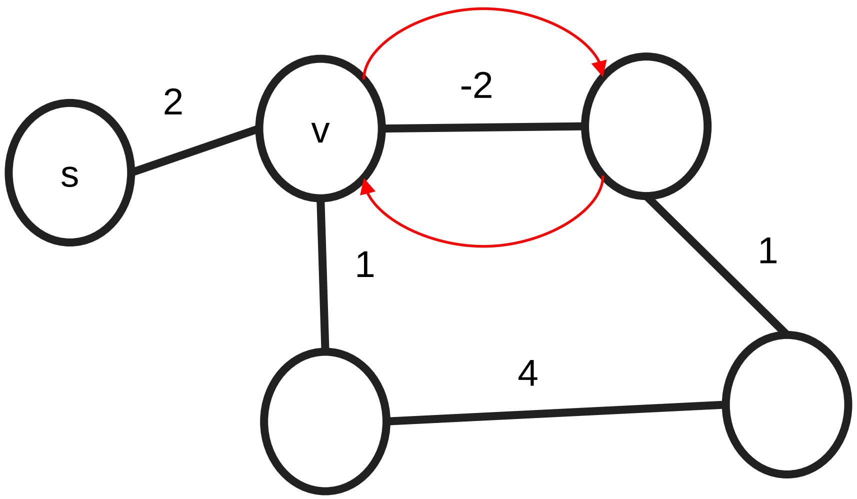


The cost of path s~>v can be improved indefinitely!

Negative-weight cycles

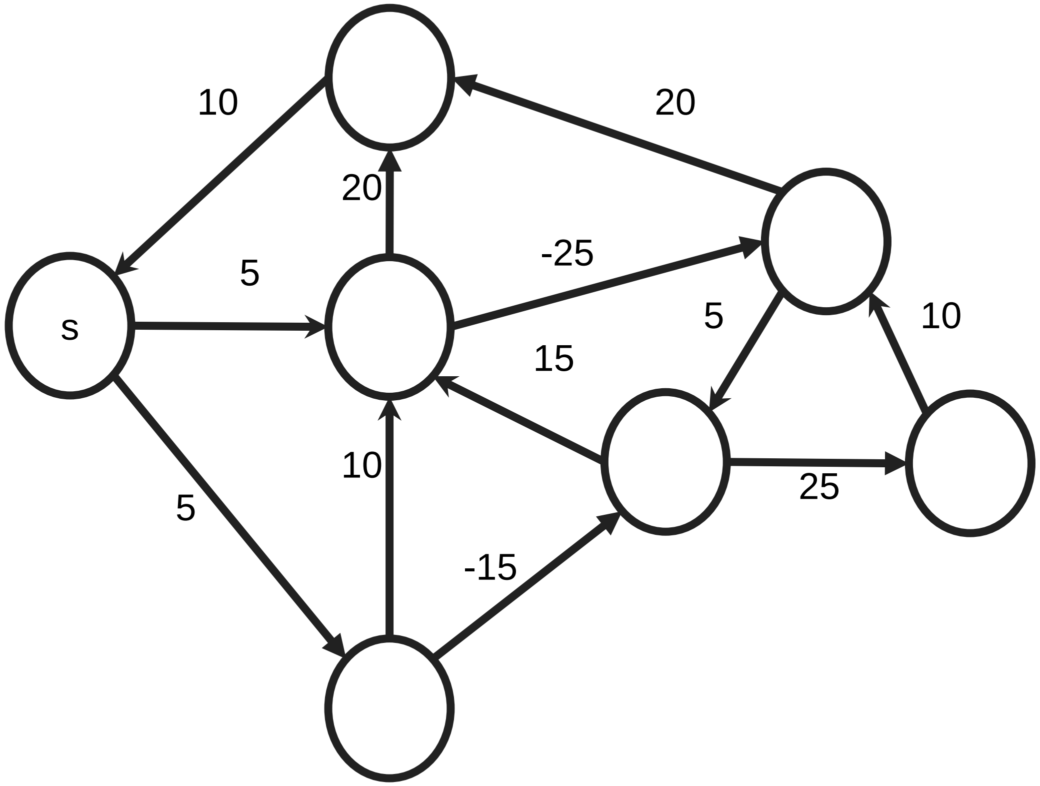
- If the graph contains a negative cycle, then all the shortest paths produced by any of the shortest paths algorithms are unreliable (may be not the shortest)
- Thus we either believe that our input graph does not contain negative-weight cycles, or we ask the algorithm to at least inform us if such cycle is present

- For the same reason, while working with negative-edge weights we cannot really work with undirected graphs: each negative-cost edge can be considered as a negative-weight cycle of 2 nodes



We cannot work with undirected graphs with negative edge costs

Quiz: how many edges in any shortest path?

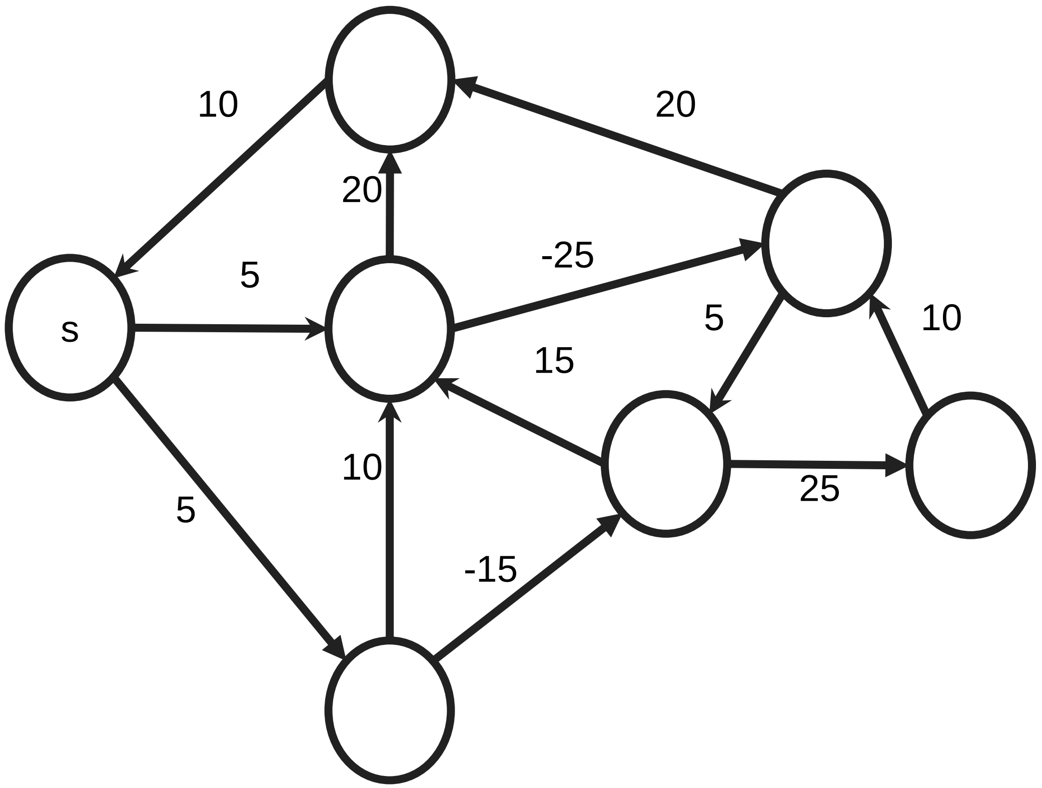


Given directed graph G=(V,E) without negative cost cycles, what is the maximum number of edges in a shortest path u~>v?

- Total number of edges:

A. At most n
B. At most n-1
C. At most n+1
D. At most n^2

Quiz: how many edges in any shortest path?



Given directed graph G=(V,E) without negative cost cycles, what is the maximum number of edges in a shortest path u~>v?

- Total number of edges:

A. At most n
B. At most n-1
C. At most n+1
D. At most n^2

A shortest path from s to v would contain in total no more that n vertices and n-1 edges, because the paths would not contain cycles: the only cycles that could improve the path cost are negative-weight cycles, and they are not allowed

General Single-Source Shortest Paths problem

Input: directed graph G=(V,E), array C of edge costs [possibly negative], source vertex s
Output: if G has no negative-weight cycles, then for every vertex v ∈ V, shortest path s~>v

Recap: when to use Dynamic Programming

- **There is a “natural” ordering of subproblems from smallest to largest such that you can obtain the solution for a subproblem by only looking at smaller subproblems.**

- It is easy to decide which subproblem is smaller when the input is a sequence: array (knapsack items) or strings (edit distance)
- It is much harder to imagine a “natural” ordering of subproblems on graphs: they have no order on vertices or edges

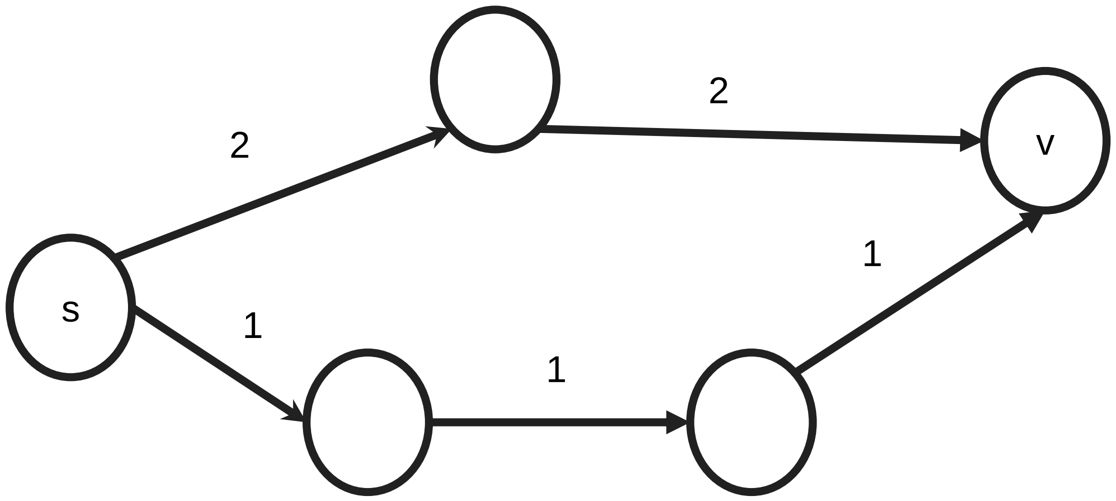
- **If we do not have a “natural” ordering we need to impose an artificial ordering: this is the main step in designing DP algorithms on graphs**

Order of subproblems

- We will exploit the sequential nature of a path: if a path is optimal, then every subpath must also be optimal

- **Issue:** not clear how to define smaller and larger subproblems
- **Key idea:** artificially restrict the number of edges in the path

- Subproblems are ordered by the number of edges allowed



Example of subproblems:

The shortest path s~>v with edge budget = 2 has cost 4

The shortest path s~>v with edge budget = 3 has cost 3

First subproblem is smaller than the second and is solved first

Optimal subproblems

Let $P(v,k-1)$ be the cost of shortest path from the source vertex s to v using at most k-1 edges

We increase the edge budget by allowing one more edge and want to compute $P(v, k)$

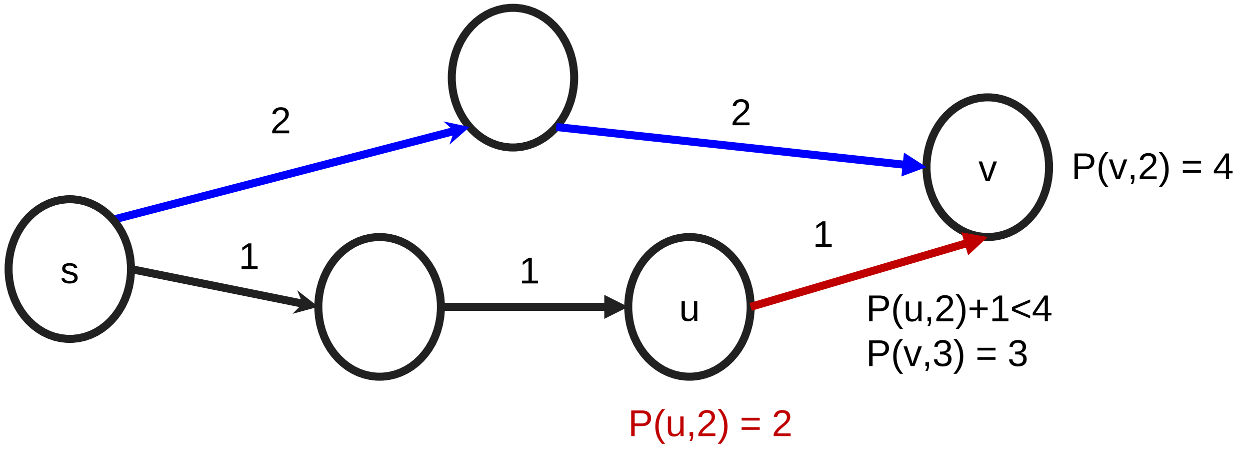
What are possible choices?

- For each incoming edge (u,v) we extend all (already computed) paths $P(u, k-1)$ by edge (u,v)
- If adding any of these edges to paths $P(u, k-1)$ does not result in a shorter path: then

  $P(v, k) = P(v, k-1)$ [we keep previous shortest path]
- Otherwise we get a shorter path using one of the incoming (u,v) edges:

  $P(v,k) = P(u, k-1) + c_{uv}$

For each vertex v we need to consider at most 1 + in-degree(v) candidate paths with the edge budget <= k



Recurrence relation

- Let P(v,k) be the cost of the shortest path s~>v with the total budget k of allowed edges [path s~>v contains ≤ k edges]

Base case: k=0 [0 edges allowed]

$$P(v,0) = \begin{cases} 0 & \text{if } v=s \\ \infty & \text{if } v \neq s \end{cases}$$

Recurrence relation

- Let P(v,k) be the cost of the shortest path s~>v with the total budget k of allowed edges [path s~>v contains ≤ k edges]

Base case: k=0 [0 edges allowed]

$$P(v,0) = \begin{cases} 0 & \text{if } v=s \\ \infty & \text{if } v \neq s \end{cases}$$

Recurrence: $0 < k \leq n-1$

Max number of edges n-1

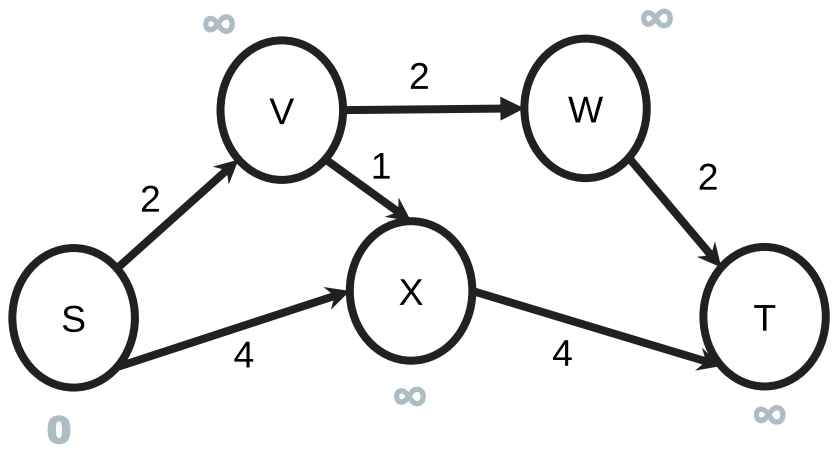
$$P(v,k) = \min \begin{cases} P(v, k-1) \\ \underset{\text{over all edges}(u,v)}{\min} (P(u, k-1) + c_{uv}) \end{cases}$$

Pseudocode

```
Algorithm BellmanFord(digraph G=(V, E), edge costs C)

A: = nxn 2D array indexed by k and v

# base case
A[0, s] := 0
for each v ∈ V:
    A[0, v] := ∞

# DP table
for k from 1 to n-1:
    for each v ∈ V:
        A[k,v]:= A[k-1][v]
        for each edge (u, v): # check all incoming edges of v
            if  A[k-1][u] + C[u,v] < A[k,v]:
                A[k,v]: = A[k-1][u] + C[u,v]

return A[n-1]      # the last row contains final shortest paths from s
```

Bellman-Ford: illustration

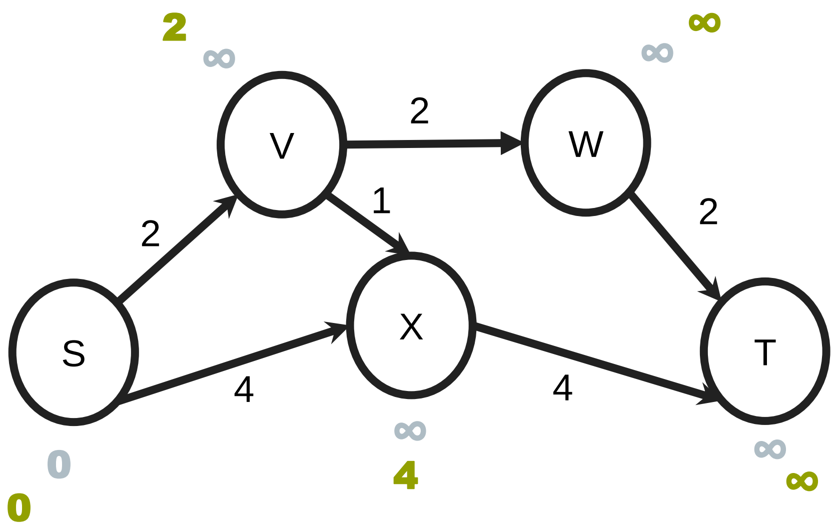
- k=0 [zero edges allowed]



k	S	T	V	W	X
0	0	∞	∞	∞	∞
1					
2					
3					
4					

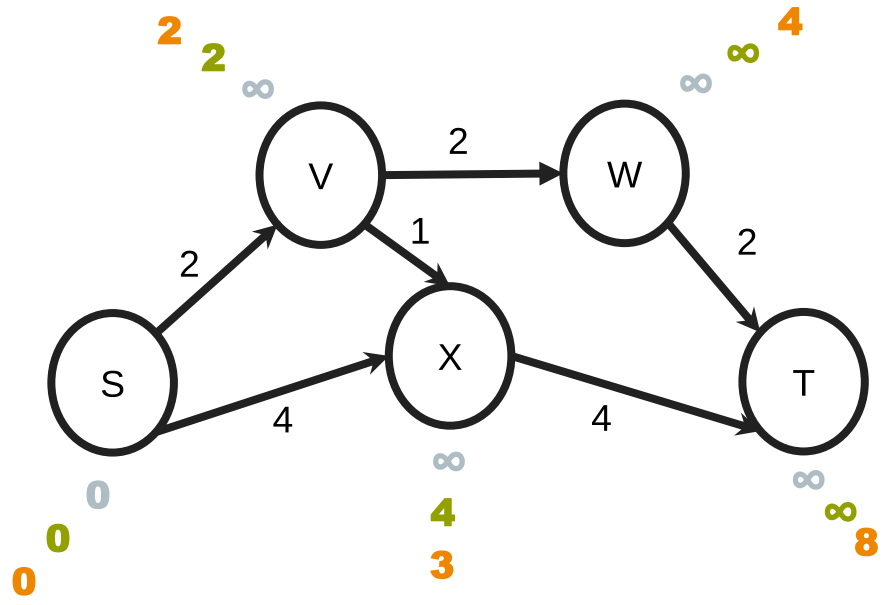
Bellman-Ford: illustration

- k=1 [shortest paths with 1 edge]



k	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2					
3					
4					

Bellman-Ford: illustration

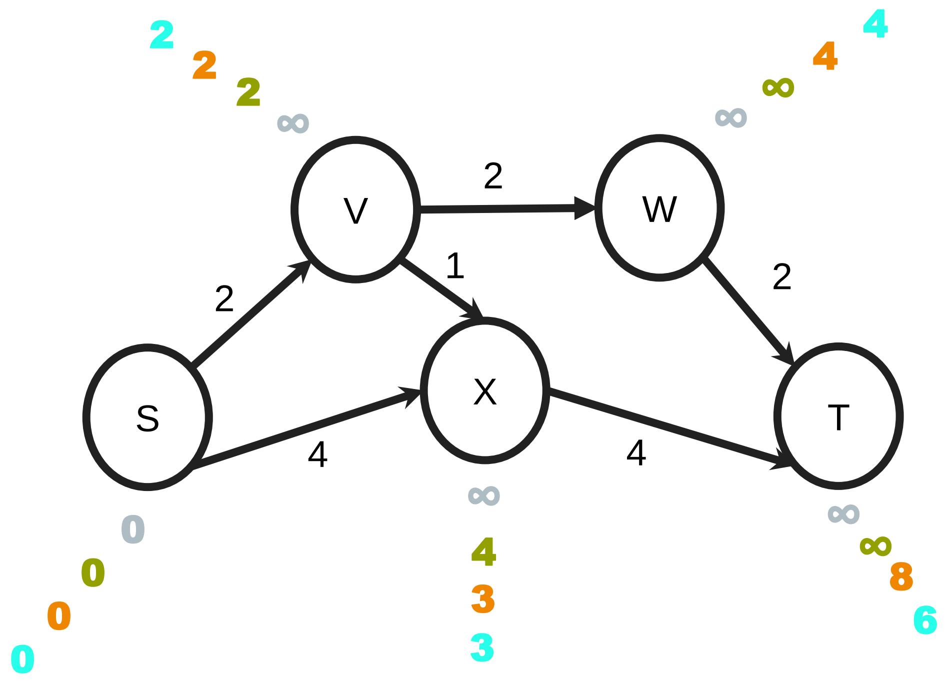


- k=2

k	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3					
4					

Bellman-Ford: illustration

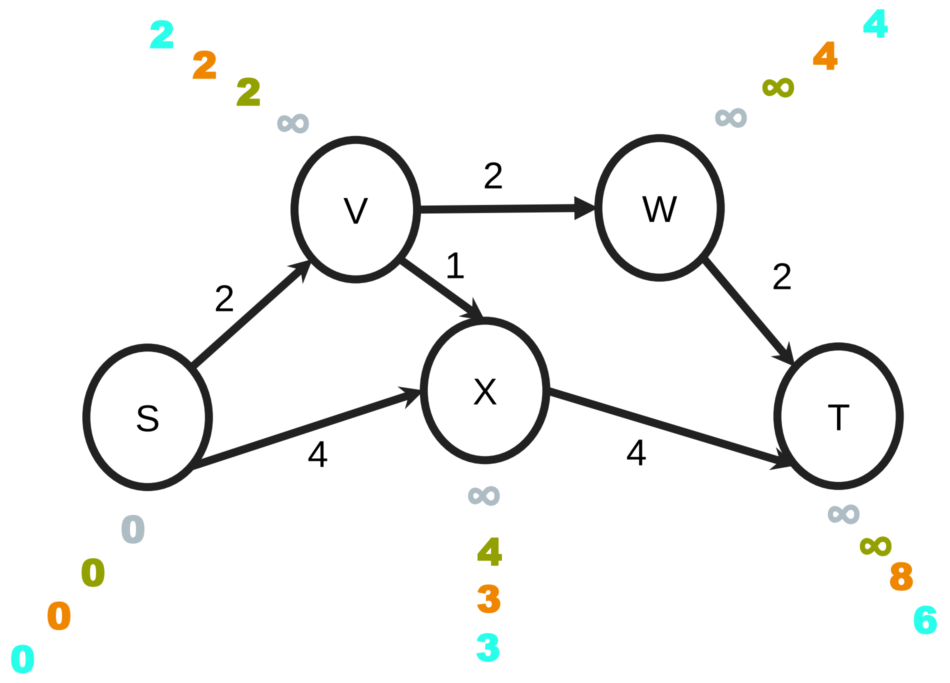
- k=3



i	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3	0	6	2	4	3
4					

Bellman-Ford: illustration

- k=4



i	S	T	V	W	X
0	0	∞	∞	∞	∞
1	0	∞	2	∞	4
2	0	8	2	4	3
3	0	6	2	4	3
4	0	6	2	4	3

Running Time

Algorithm BellmanFord(digraph G=(V, E), edge costs C)

```
A: = nxn 2D array indexed by k and v

# base case
A[0, s] := 0
for each v ∈ V:
        A[0, v] := ∞

# DP table
for k from 1 to n-1:
        for each v ∈ V:
                A[k,v]:= A[k-1][v]
                for each edge (u, v): # check all incoming edges of v
                        if  A[k-1][u] + C[u,v] < A[k,v]:
                                A[k,v]: = A[k-1][u] + C[u,v]

return A[n-1]         # the last row contains final shortest paths from s
```

Loop is executed n times

At each iteration – total O(m) edges are checked for all the subproblems at iteration k
Sum(in-degree(V)) = O(m)

Running time: O(nm)

Bellman-Ford algorithm: notes

- **Early stopping:**
  - We can run less than n-1 iterations
  - If there is no improvements between iteration k-1 and iteration k, then the algorithm computed all shortest paths

- **Detecting negative-weight cycles:**
  - If algorithm continues until iteration n-1, then we run one more iteration
  - If we have improvements in iteration n, then G contains a negative-cost cycle
  - Conclusion: all the shortest paths are unreliable

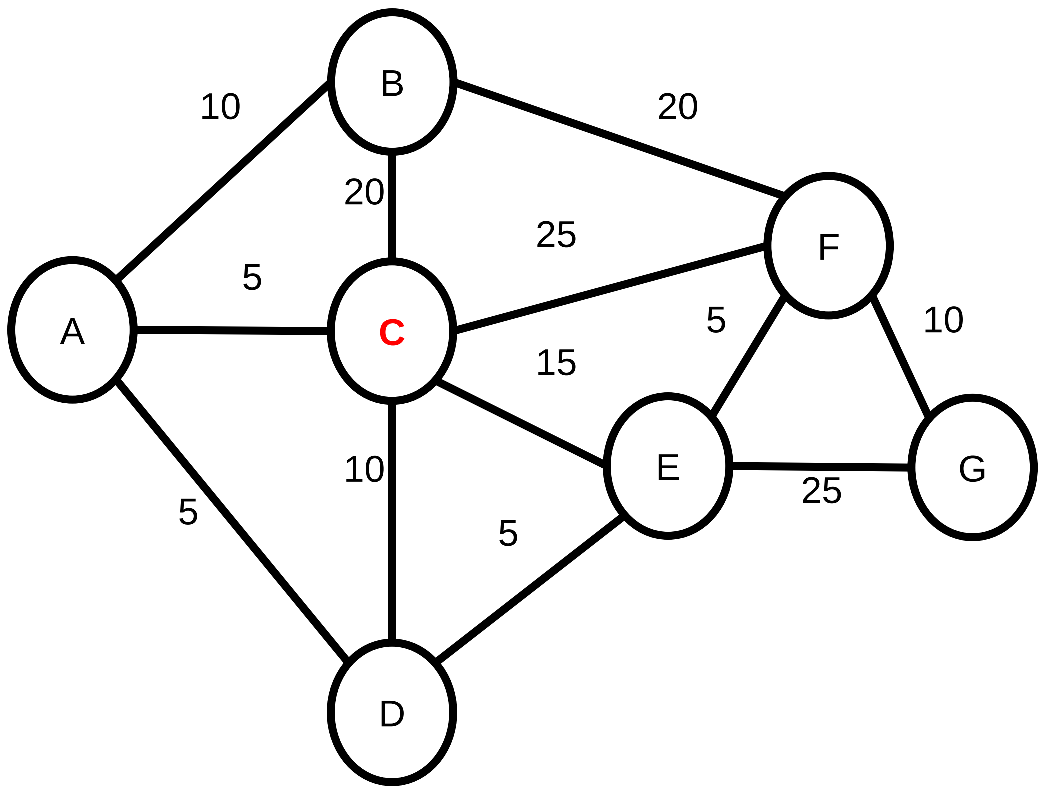
- **Space improvement:**
  - We can reconstruct the shortest paths by a regular traceback: but this requires to store all n^2 cells of the DP table
  - However due to sequential nature of a path and the fact that each subpath of the optimal path is by itself optimal – we just need to store the predecessor node for each destination vertex v: when the path gets improved, we store the source node u which caused this improvement
  - Because the subpath s~>u is by itself optimal, we can continue recovering the path by looking at predecessor of u etc., until we reach node s.

Appendix: full step-by-step example of Dijkstra’s algorithm

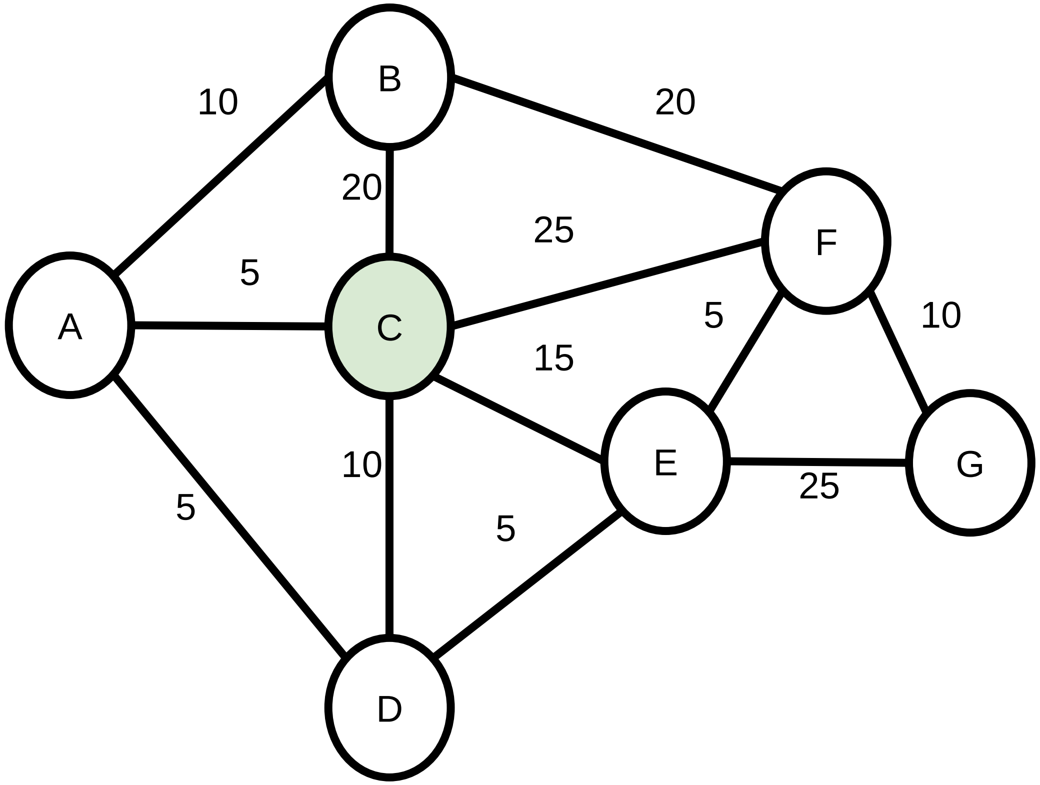
For those of you who forgot how the algorithm works

Dijkstra's Algorithm: full example

Find all minimum cost paths from the source node C.



Dijkstra's Algorithm: full example



Known shortest paths from C

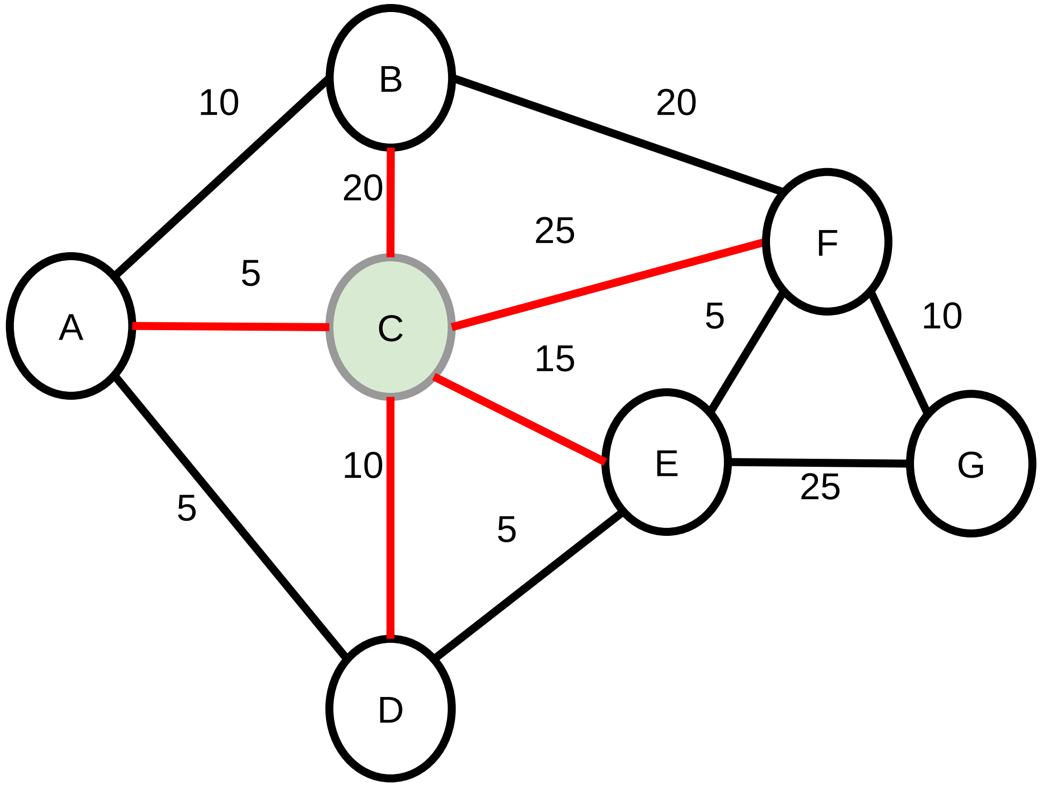
To v_i	Shortest path
C	C-C: 0

Remaining nodes with their Dijkstra Greedy Score

	DGS
A	∞
B	∞
D	∞
E	∞
F	∞
G	∞

We start by assigning Dijkstra Greedy Score (DGS) to each node as ∞
The only known min-cost path is C-C of length 0.
We know that it cannot be improved so we add it to the Processed nodes (green)

Dijkstra's Algorithm: full example



Update DGS for all nodes adjacent to C.
Improve their DGS using edges that cross Processed and Unprocessed sets.

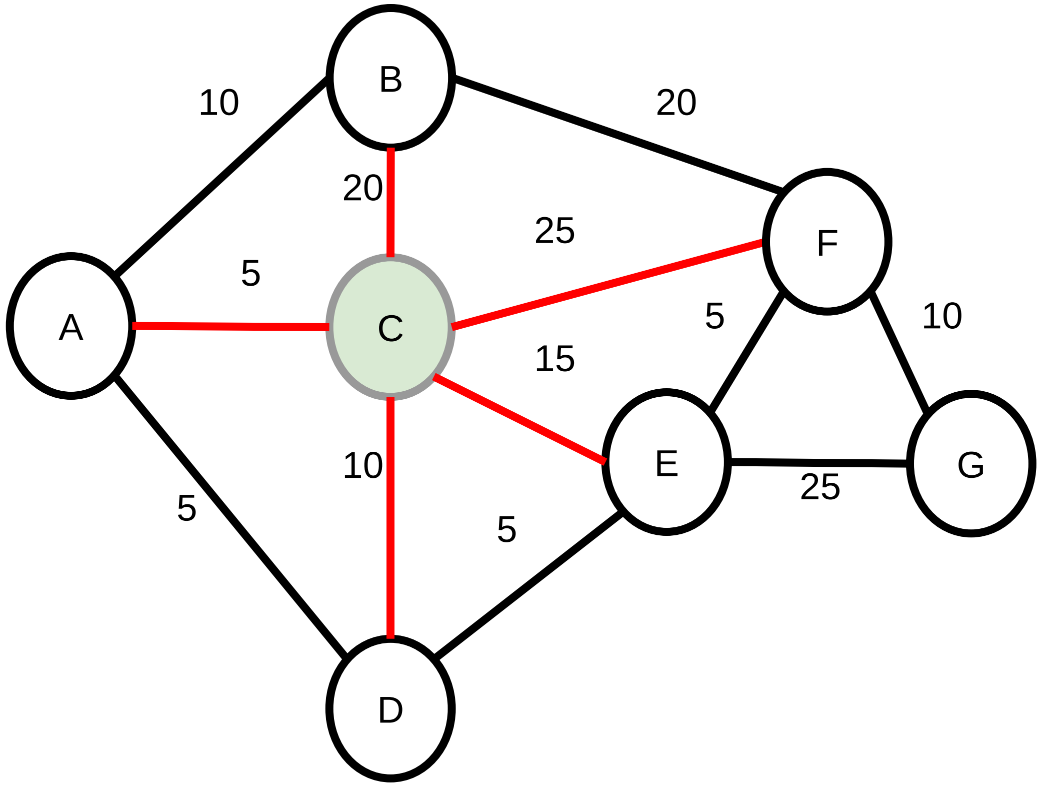
Known shortest paths from C

To v_i	Shortest path
C	C-C: 0

Remaining nodes with their Dijkstra Greedy Score

	DGS
A	5
B	20
D	10
E	15
F	25
G	∞

Dijkstra's Algorithm: full example



Select the node with min DGS and add it to known min-cost paths.

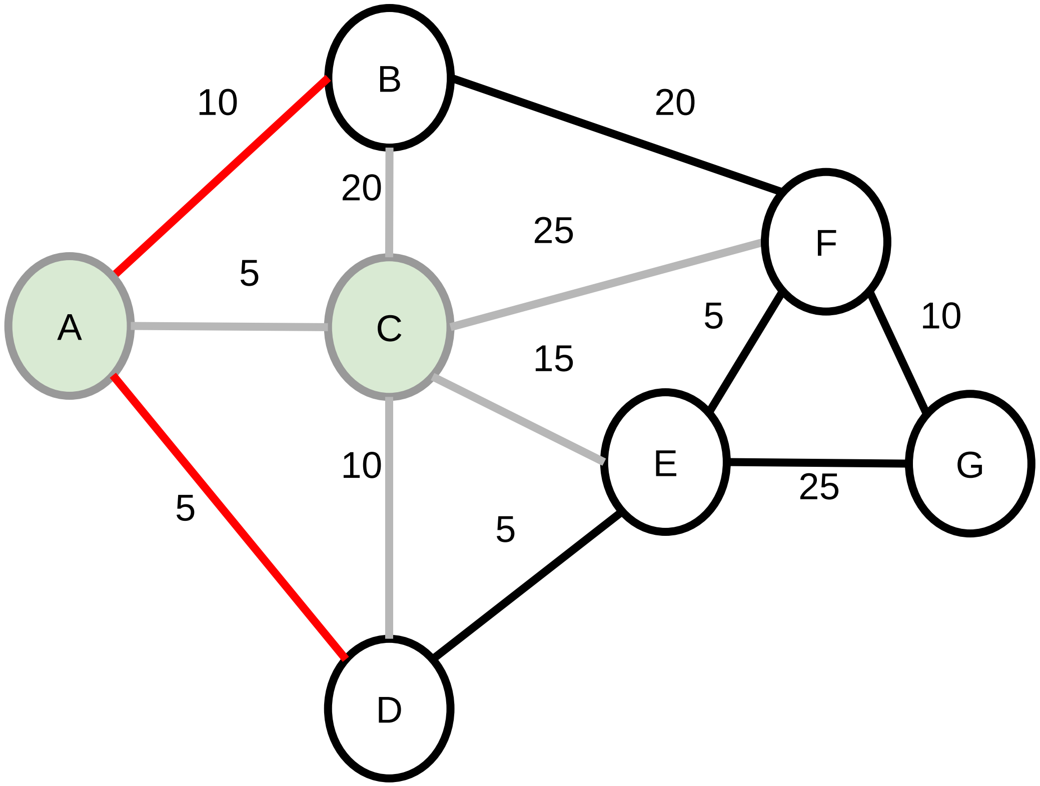
Known shortest paths from C

To v_i	Shortest path
C	C-C: 0

Remaining nodes with their Dijkstra Greedy Score

	DGS
$A_{c\text{-}a}$	5
$B_{c\text{-}b}$	20
$D_{c\text{-}d}$	10
$E_{c\text{-}e}$	15
$F_{c\text{-}f}$	25
G	∞

Dijkstra's Algorithm: full example



Known shortest paths from C

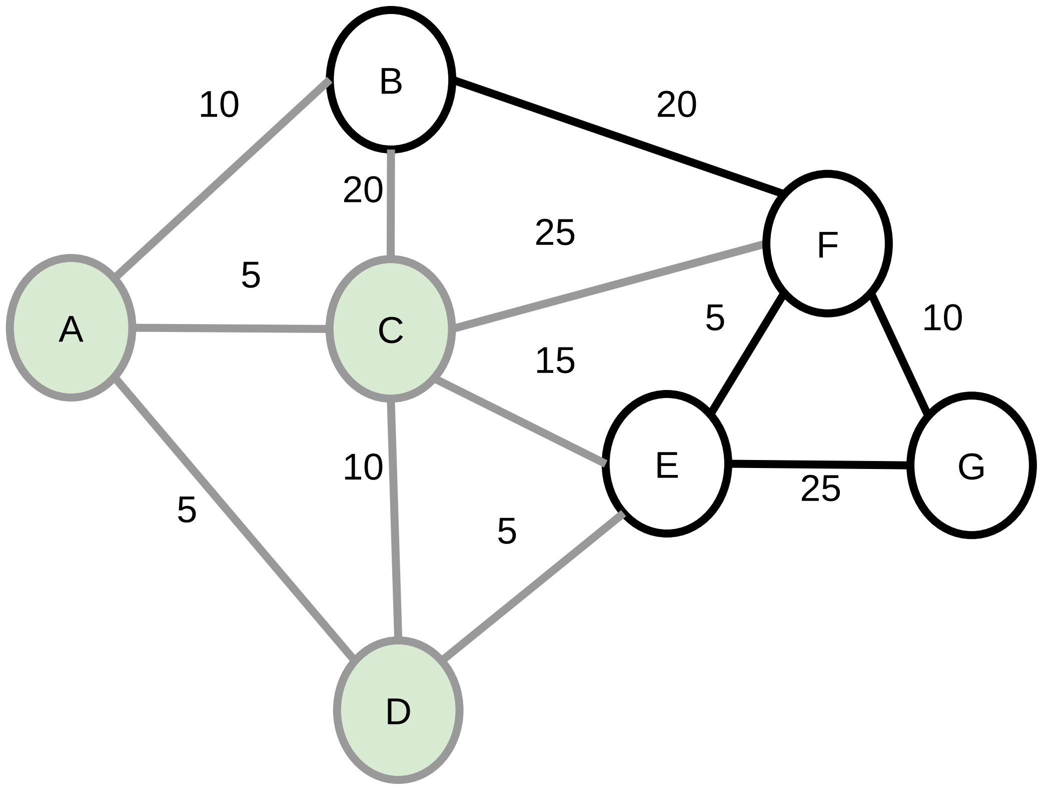
To v_i	Shortest path
C	C-C: 0
A	C-A: 5

Remaining nodes with their Dijkstra Greedy Score

	DGS
$B_{c\text{-}a\text{-}b}$	~~20~~ 15
$D_{c\text{-}d}$	10
$E_{c\text{-}e}$	15
$F_{c\text{-}f}$	25
G	∞

Update DGS for every node *v* adjacent to A: cost of path(C-A) + cost of edge(A,*v*)
Select the node with min DGS and add it to Processed

Dijkstra's Algorithm: full example



Known shortest paths from C

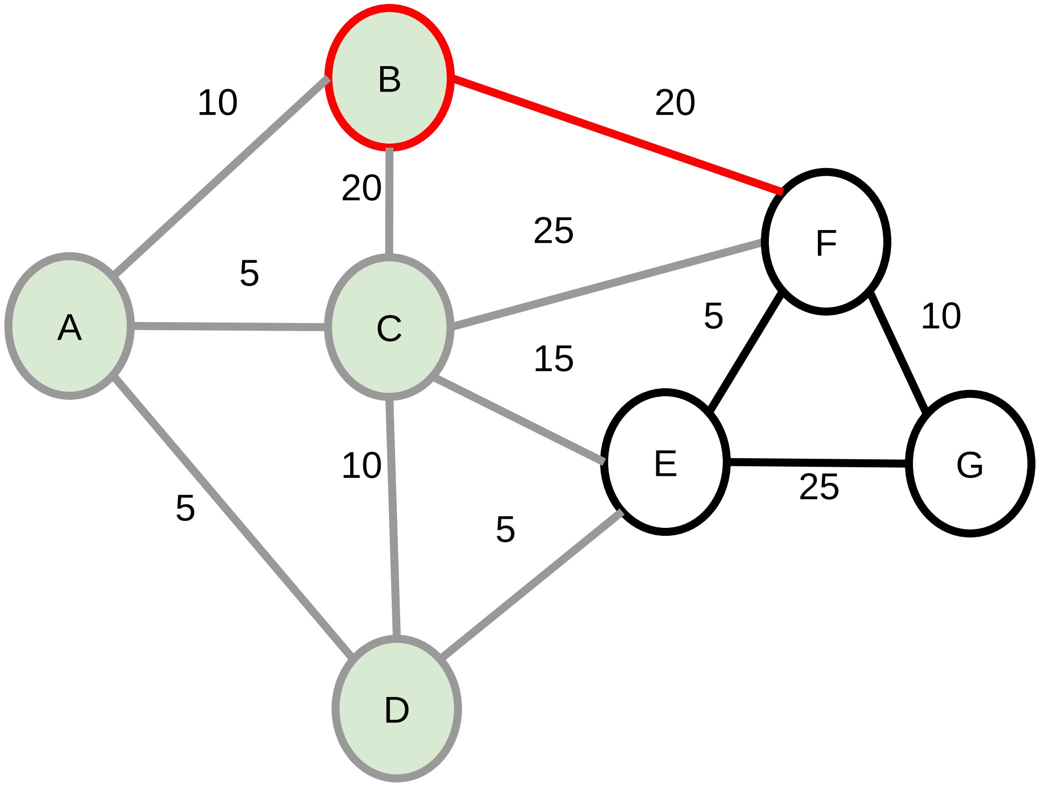
To v_i	Shortest path
C	C-C: 0
A	C-A: 5
D	C-D:10

Remaining nodes with their Dijkstra Greedy Score

	DGS
$B_{c\text{-}a\text{-}b}$	15
$E_{c\text{-}e}$	15
$F_{c\text{-}f}$	25
G	∞

Select the node with min DGS and add it to known min-cost paths.

Dijkstra's Algorithm: full example



Known shortest paths from C

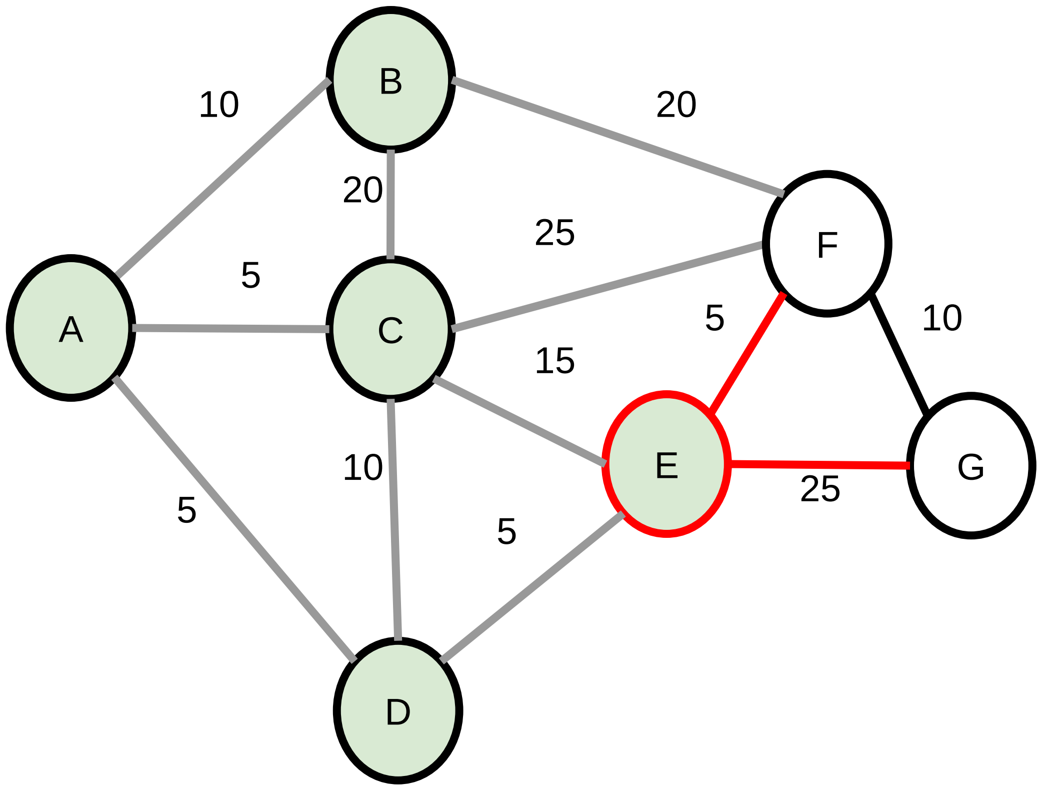
To v_i	Shortest path
C	C-C: 0
A	C-A: 5
D	C-D:10
B	C-A-B:15

Remaining nodes with their Dijkstra Greedy Score

	DGS
E_{c-e}	15
F_{c-f}	25
G	∞

Update DGS for all unprocessed nodes v adjacent to B: cost of min path(C-B) + cost (B,v)
Select the node with min DGS and add it to known min-cost paths

Dijkstra's Algorithm: full example



Update DGS for all unprocessed nodes v adjacent to E

Select the node with min DGS and add it to known min-cost paths

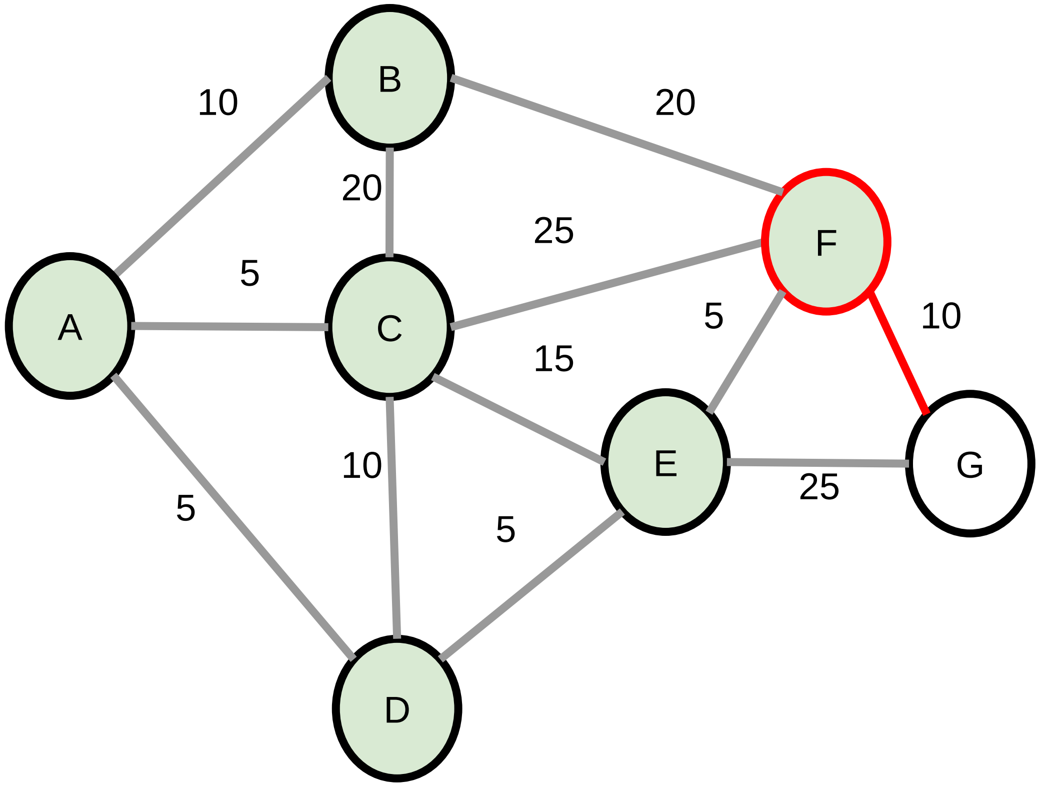
Known shortest paths from C

T o v_i	Shortest path
C	C-C: 0
A	C-A: 5
D	C-D:10
B	C-A-B:15
E	C-E: 15

Remaining nodes with their Dijkstra Greedy Score

	DGS
$F_{c\text{-}e\text{-}f}$	~~25~~ 20
$G_{c\text{-}e\text{-}g}$	40

Dijkstra's Algorithm: full example



Update DGS for all unprocessed nodes v adjacent to F: len(C-F) + len(F,v).
This is the last node - mark it as processed.

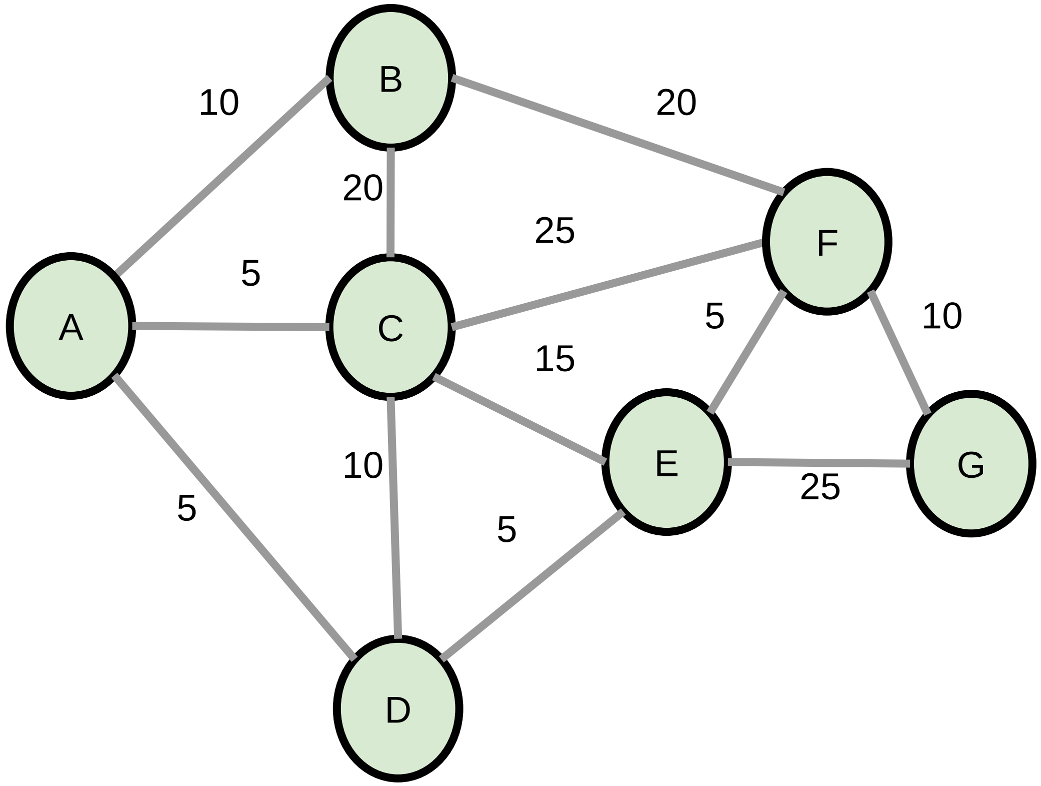
Known shortest paths from C

To v_i	Shortest path
C	C-C: 0
A	C-A: 5
D	C-A-D:10
B	C-A-B:15
E	C-E: 15
F	C-E-F: 20

Remaining nodes with their Dijkstra Greedy Score

	DGS
$G_{c\text{-}e\text{-}f\text{-}g}$	~~40~~ 30

Dijkstra's Algorithm: full example

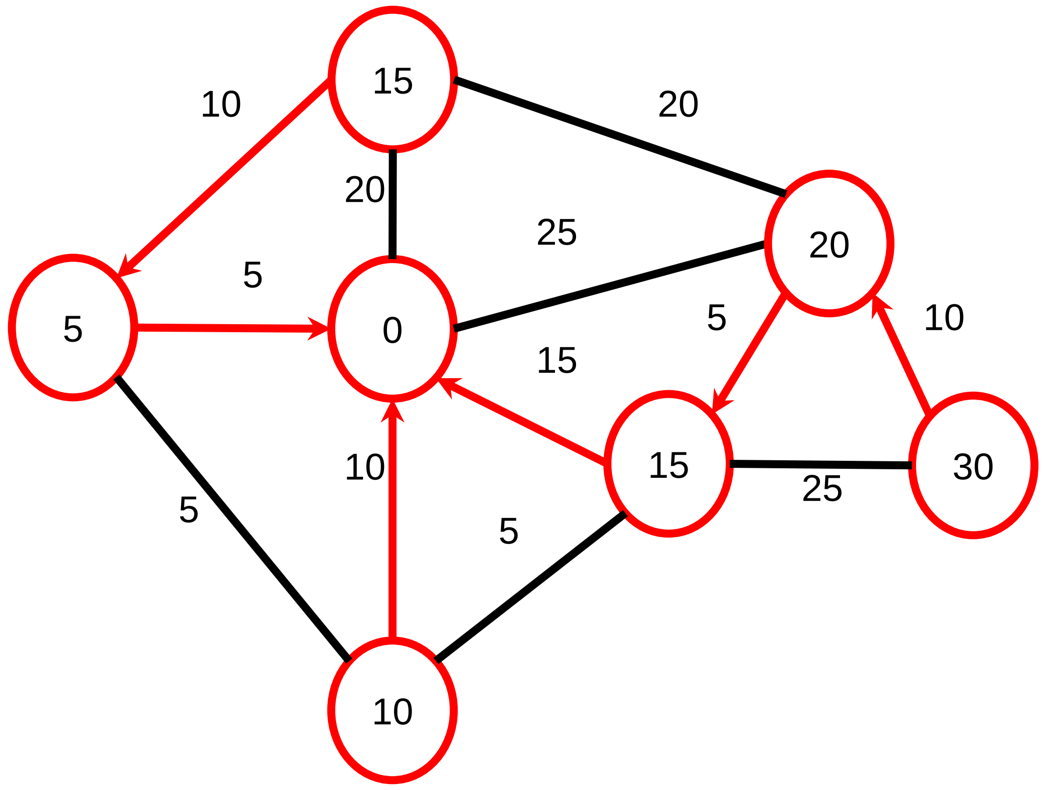


All min-cost paths from C to any other node have been computed.

All shortest paths from C

To v_i	Shortest path
C	C-C: 0
A	C-A: 5
D	C-D:10
B	C-A-B:15
E	C-E: 15
F	C-E-F: 20
C	C-E-F-G:30

Traceback



Of course, instead of storing the min path for each node, we could just store the cost of the path and the link to the parent node when we update DGS, and we will be able to find the shortest path from any node to C