Dijkstra’s Algorithm

(Algorithms and Data Structures)

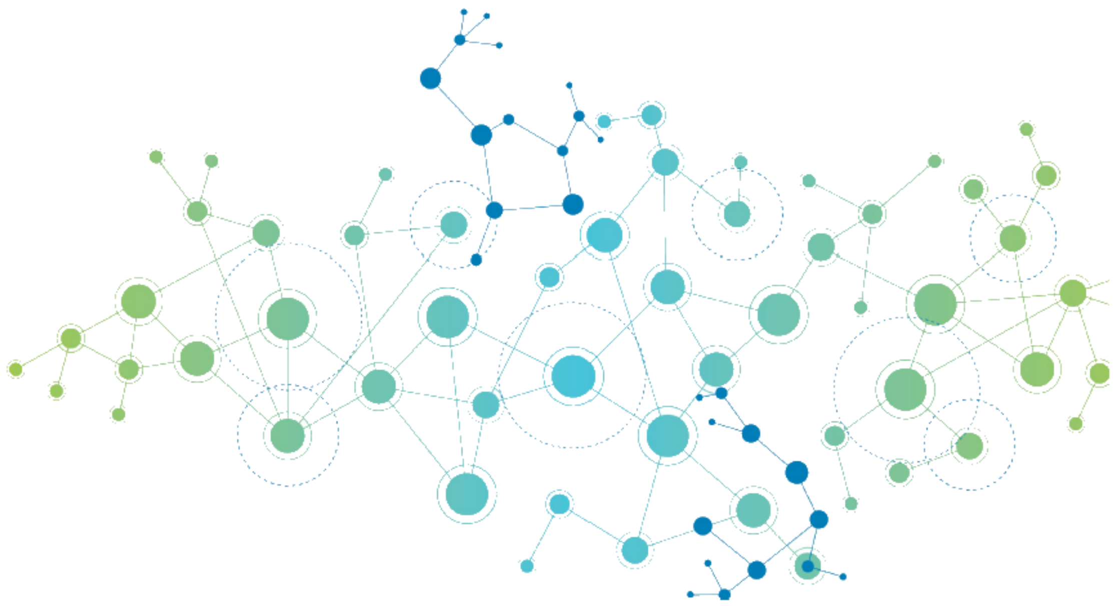
Dijkstra’s Algorithm

*“In graph theory the **shortest path problem** is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized”*

Dijkstra's Algorithm

Finding a path between two vertices in a **G(V,E)** graph such that the sum of the weights of its edges is minimized.

ALGORITHMS



1.) Dijkstra's algorithm
2.) Bellman-Ford algorithm
3.) A* search
4.) Floyd-Warshall algorithm

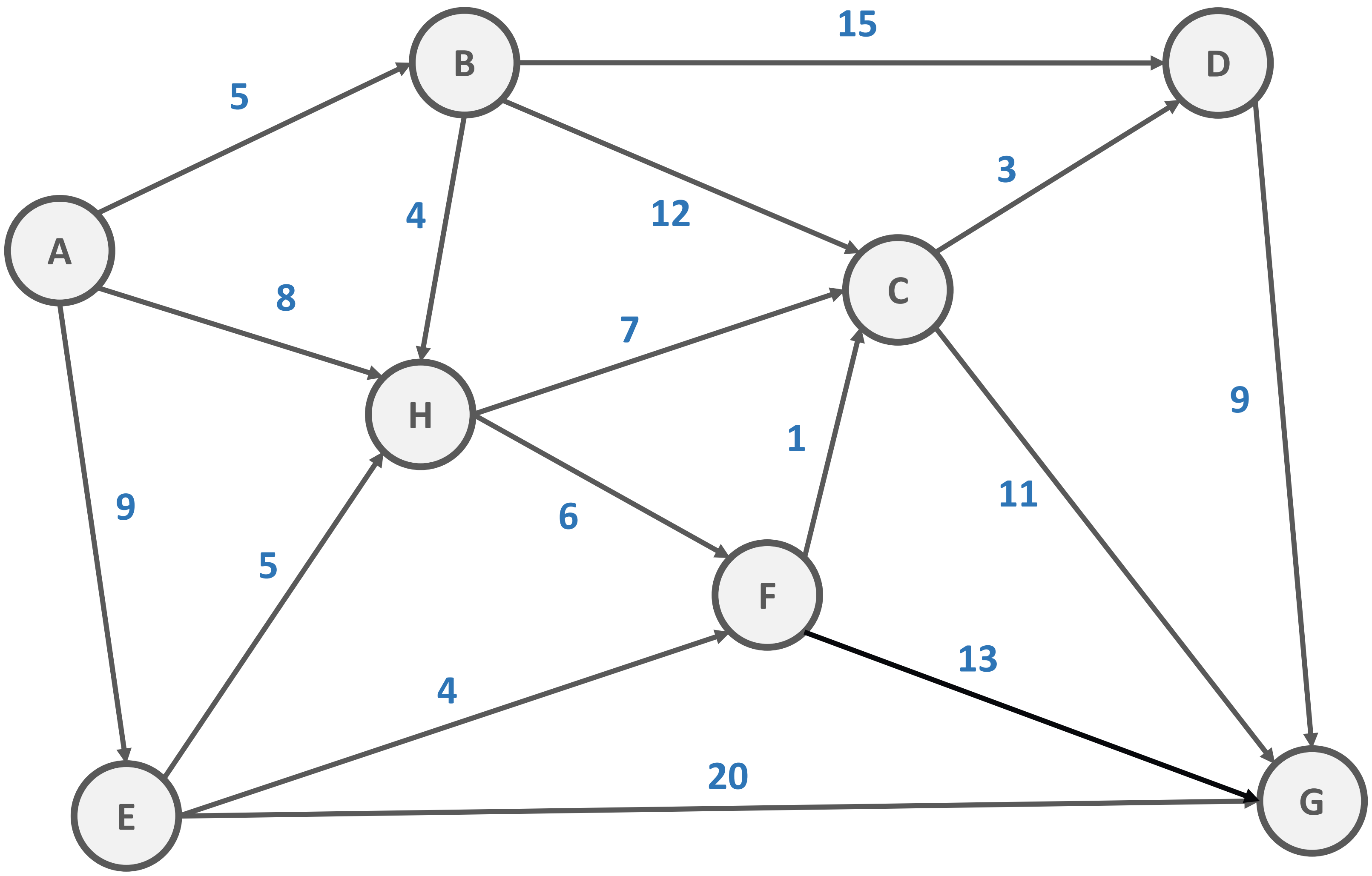
Dijkstra’s Algorithm

- it was constructed by computer scientist **Edsger Dijkstra** in **1956**
- Dijkstra’s method can handle positive edge weights - Bellman-Ford algorithm can have negative weights as well
- it can find the shortest path in a **G(V,E)** graph from **v** to **u** but it is able to construct a shortest path tree as well
- the **shortest path tree** defines the shortest paths from a source to all the other nodes
- it is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights

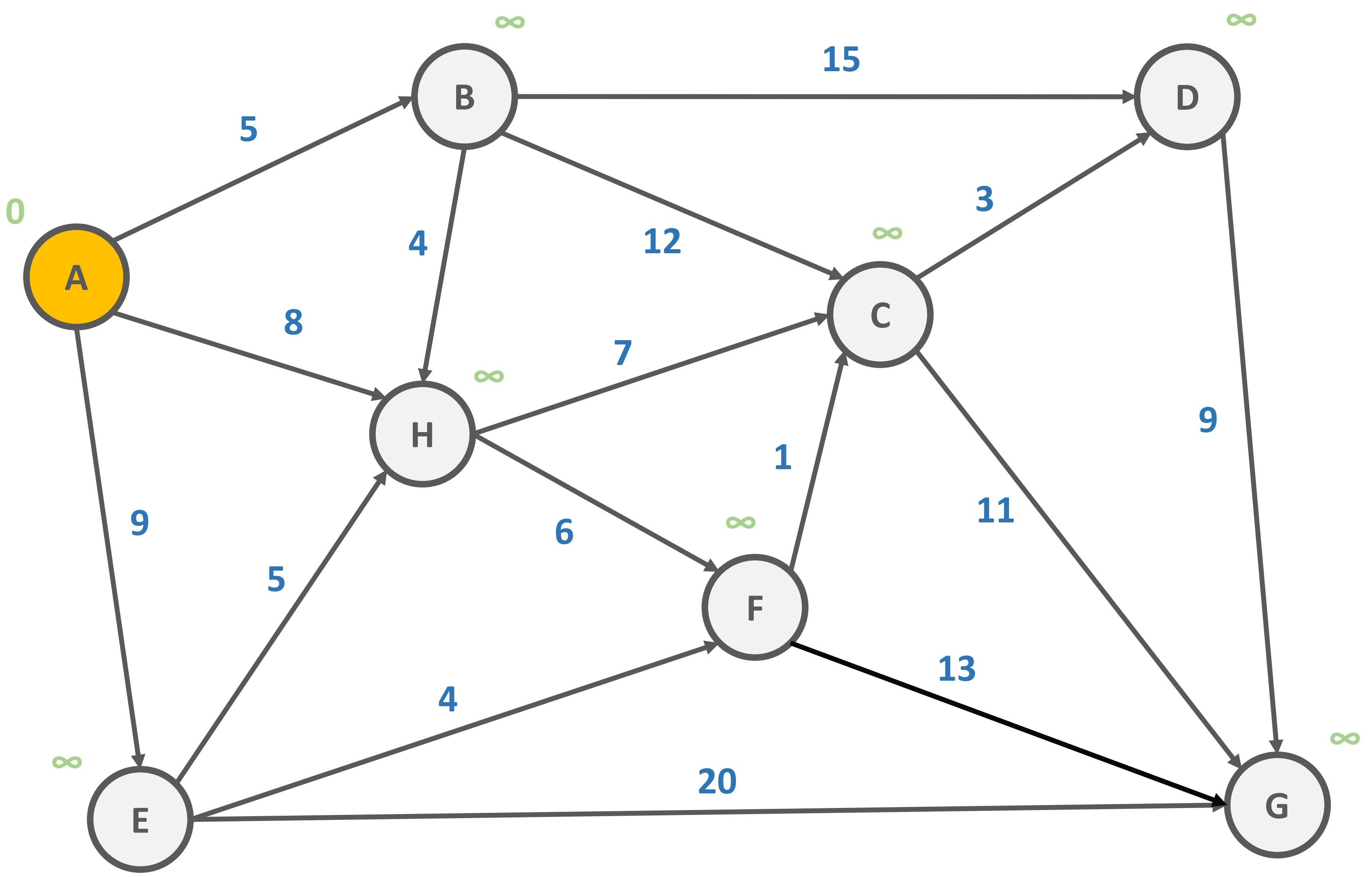
Dijkstra's Algorithm

- Dijkstra's algorithm has **O(VlogV + E)** running time complexity
- it is a **greedy** approach – it tries to find the global optimum with the help of local optimum
- on every iteration we want to find the minimum distance to the next vertex possible
- the appropriate data structure is a **priority queue** (heap)

Dijkstra’s Algorithm

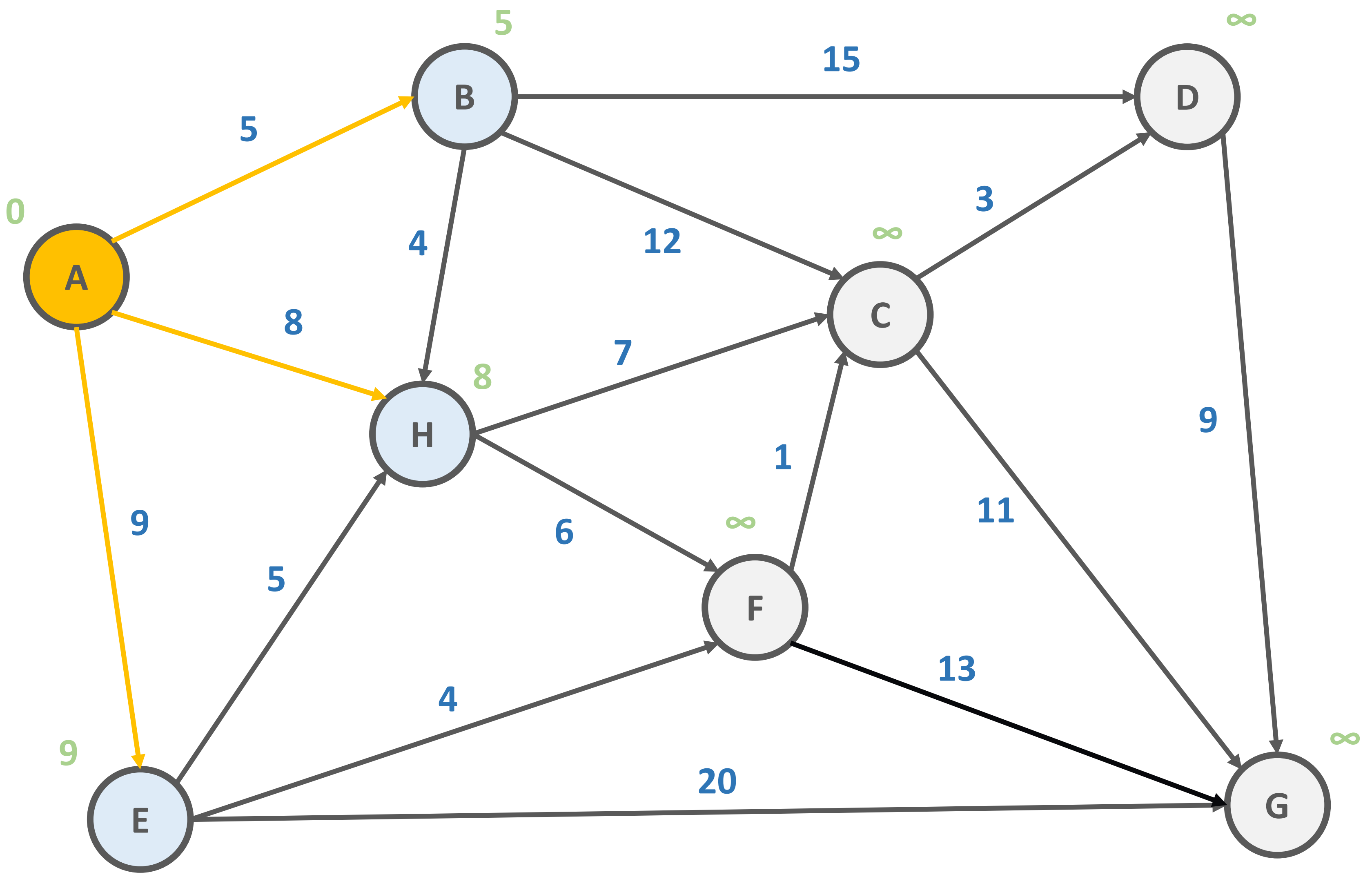


Dijkstra's Algorithm



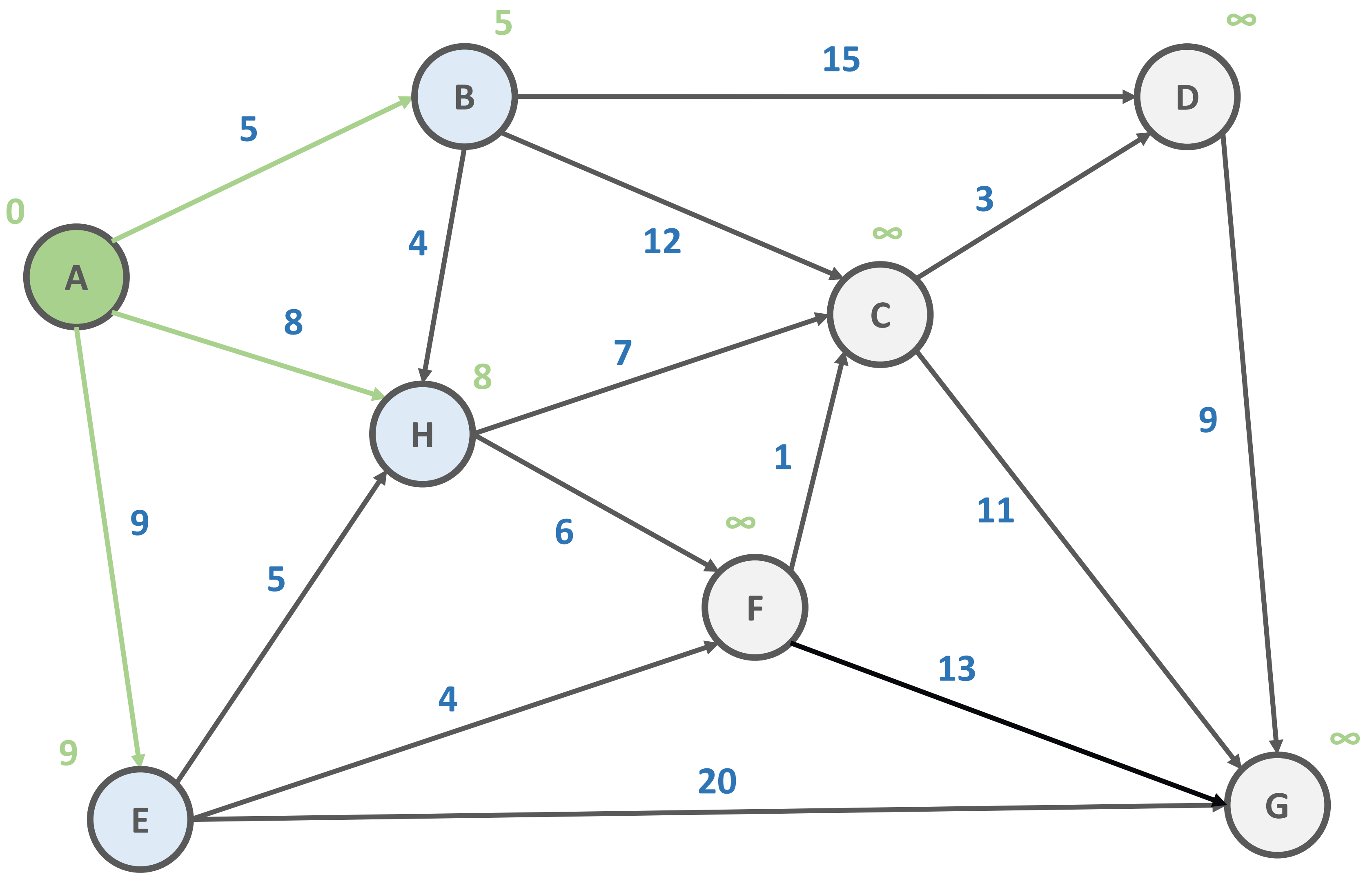
Dijkstra's Algorithm

HEAP: [B-5 H-8 E-9]



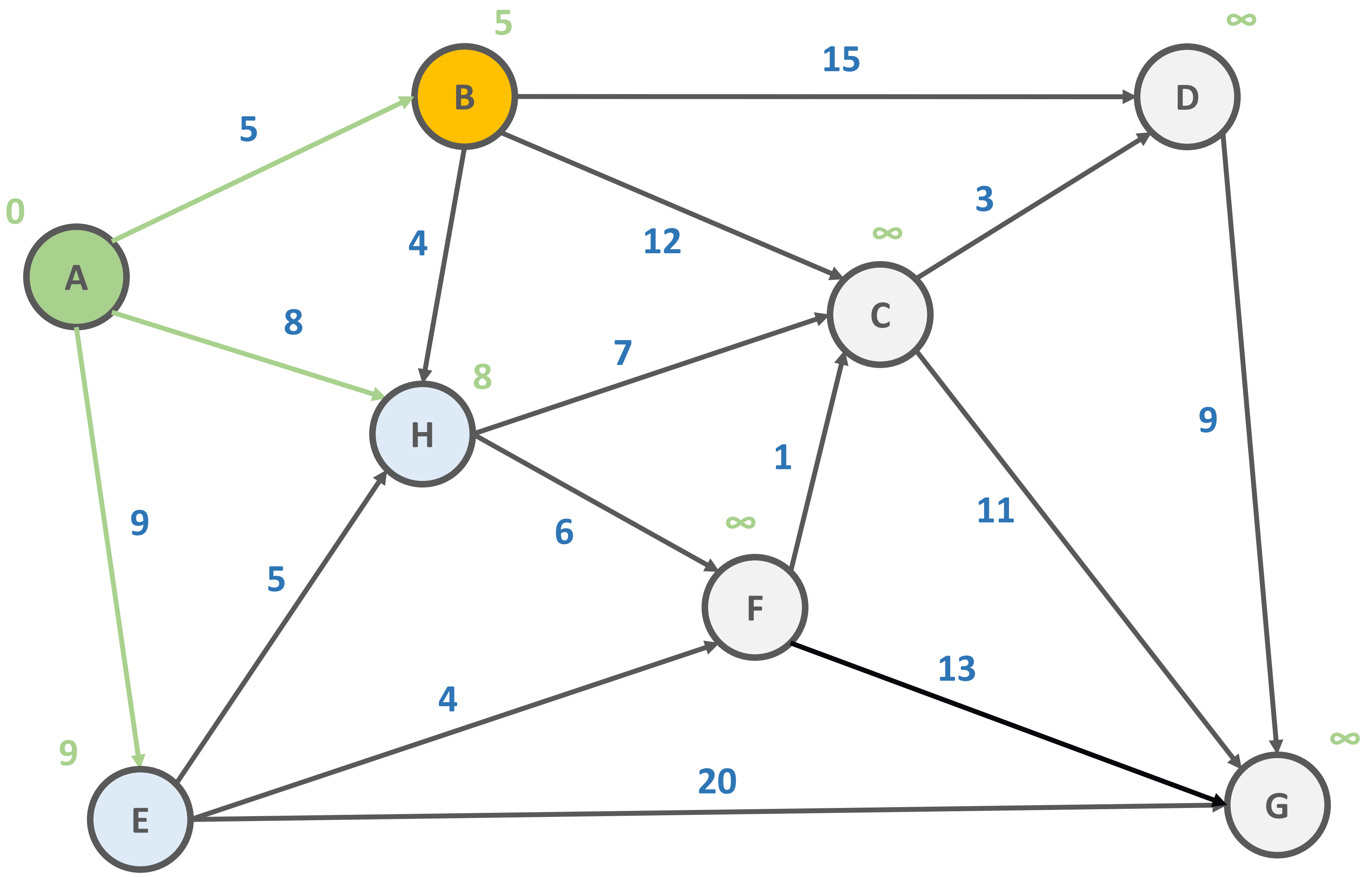
Dijkstra's Algorithm

HEAP: [B-5 H-8 E-9]



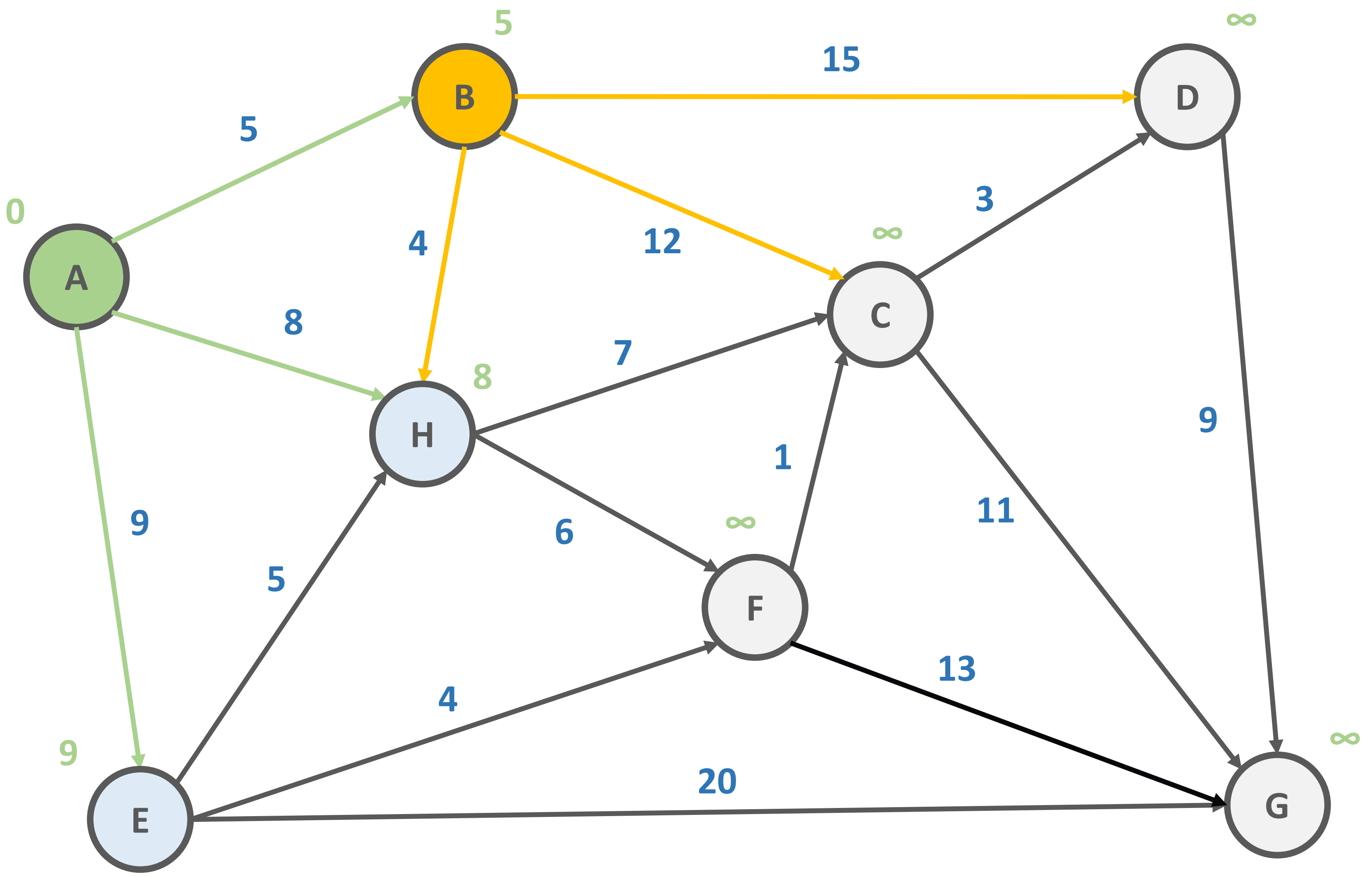
Dijkstra's Algorithm

HEAP: [B-5 H-8 E-9]



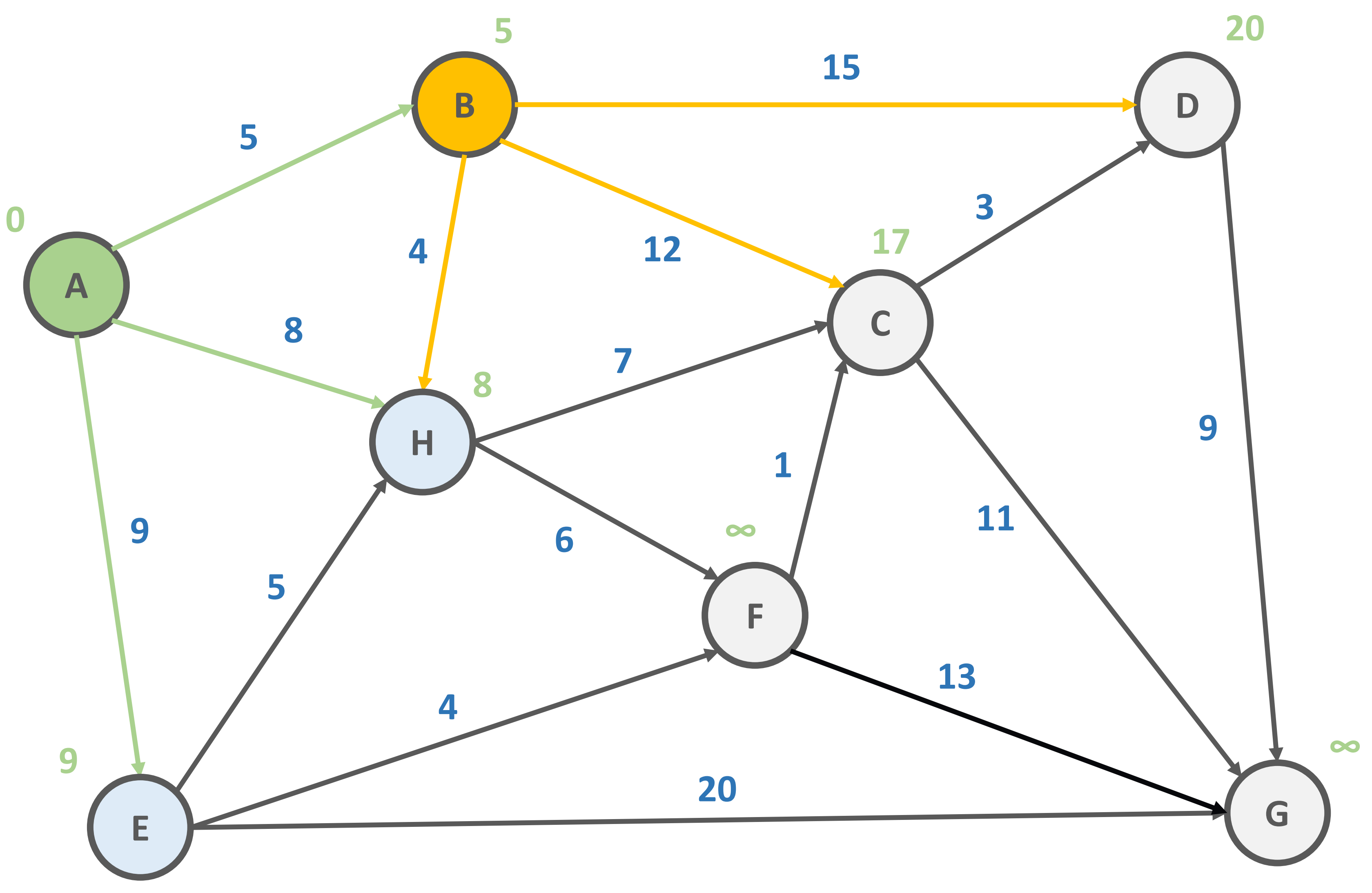
Dijkstra's Algorithm

HEAP: [H-8 E-9]



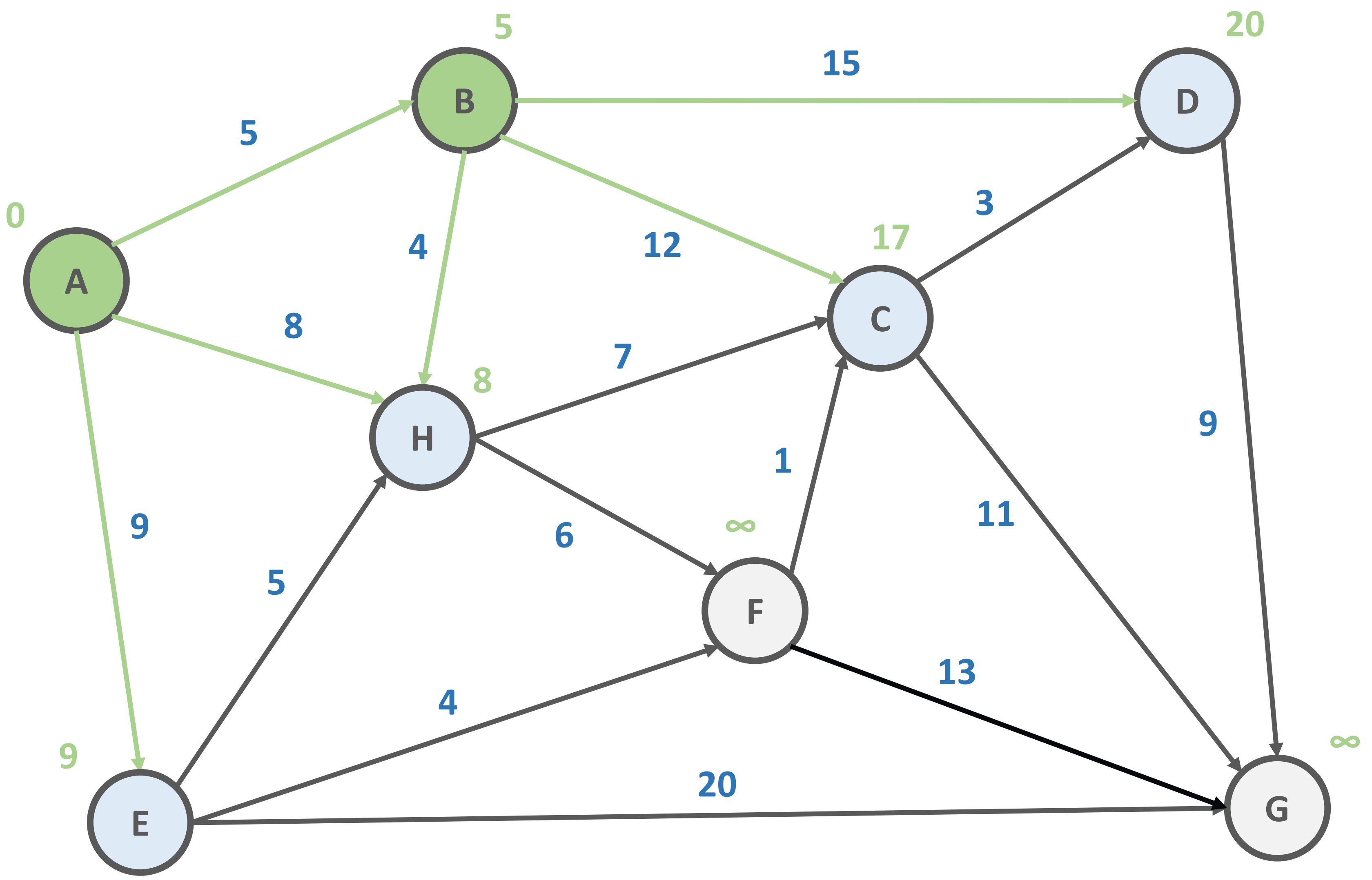
Dijkstra’s Algorithm

HEAP: [H-8 E-9]



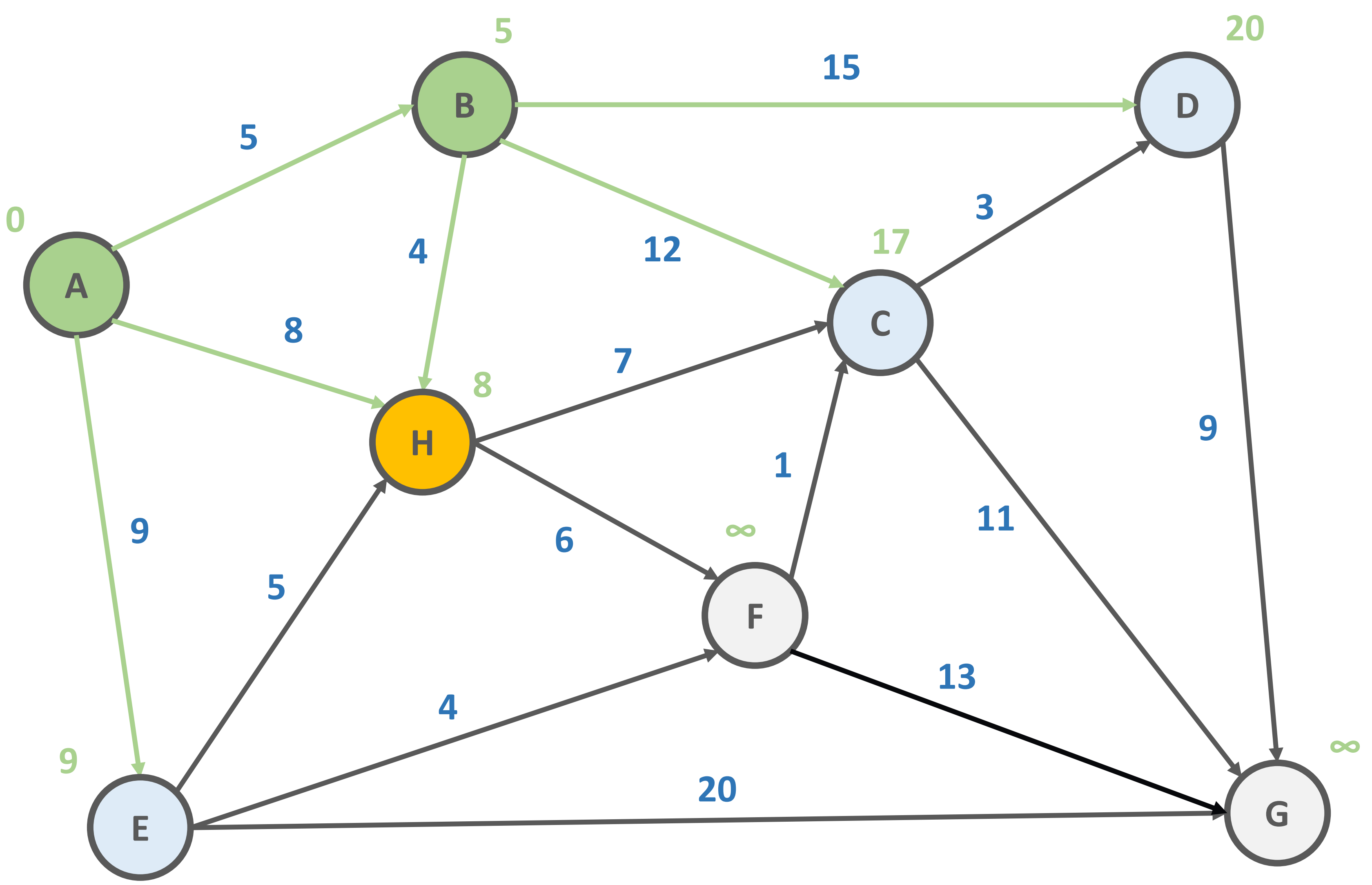
Dijkstra's Algorithm

HEAP: [H-8 E-9 D-20 C-17]



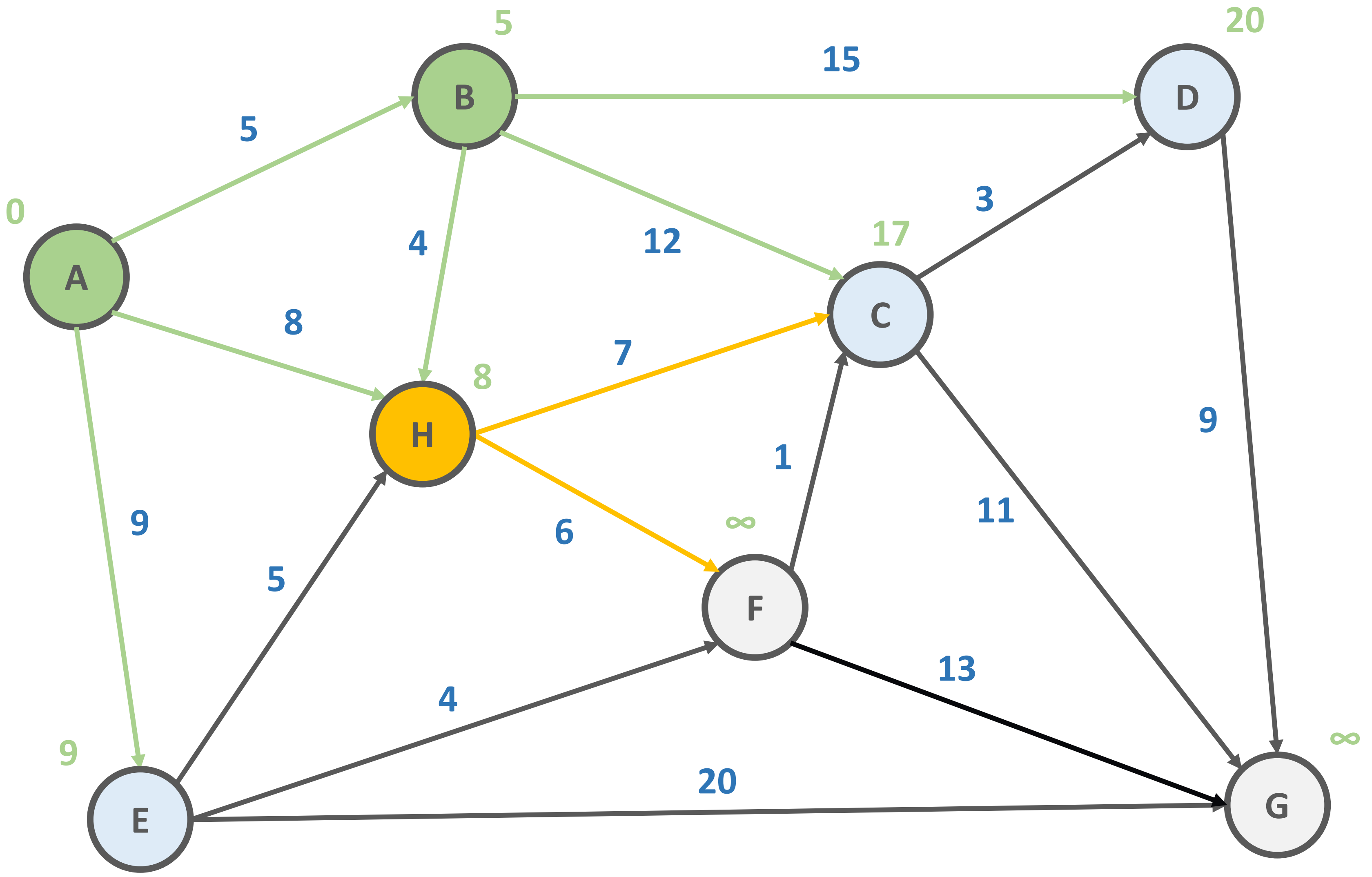
Dijkstra's Algorithm

HEAP: [H-8 E-9 D-20 C-17]



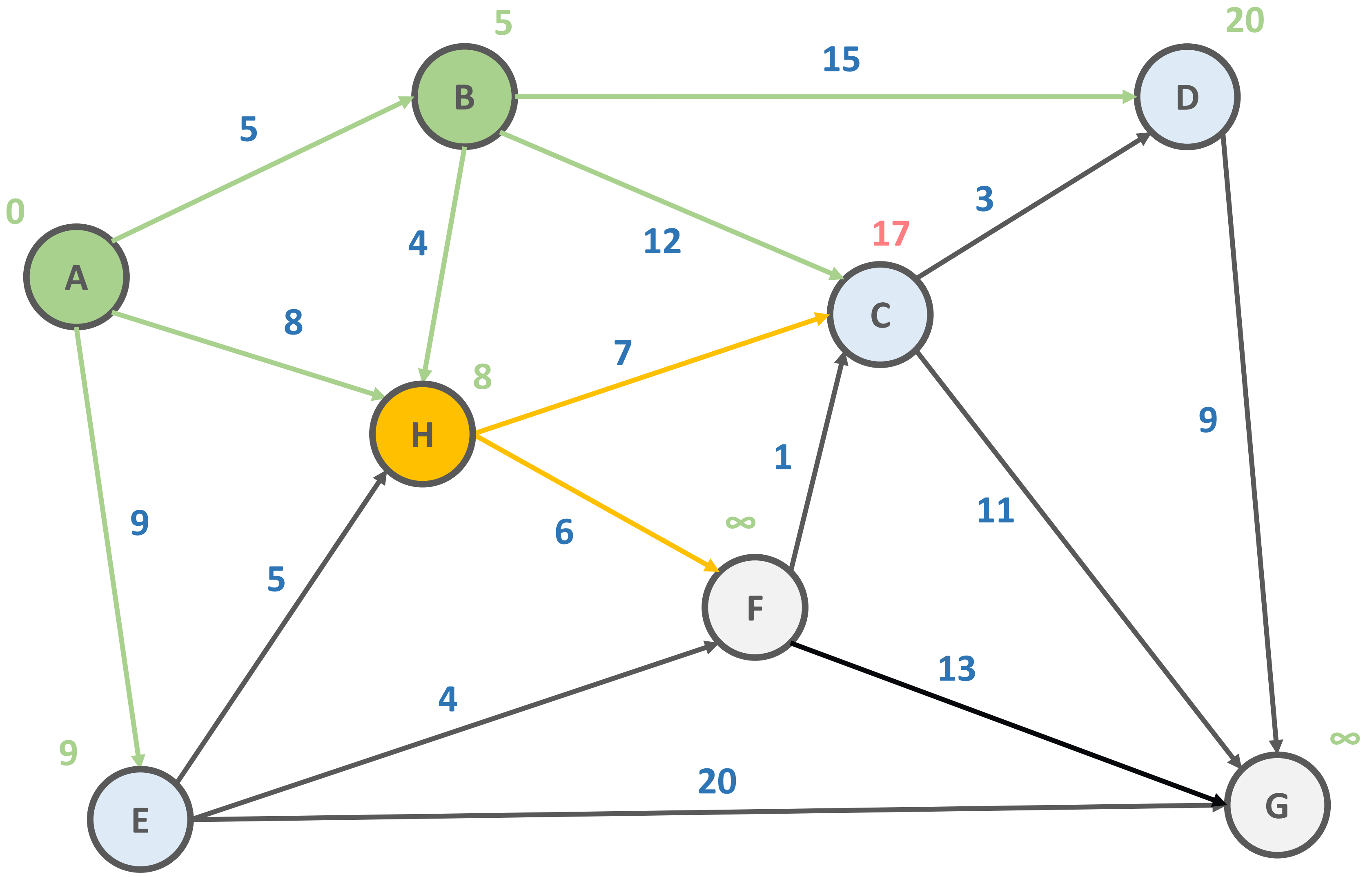
Dijkstra’s Algorithm

HEAP: [E-9 D-20 C-17]



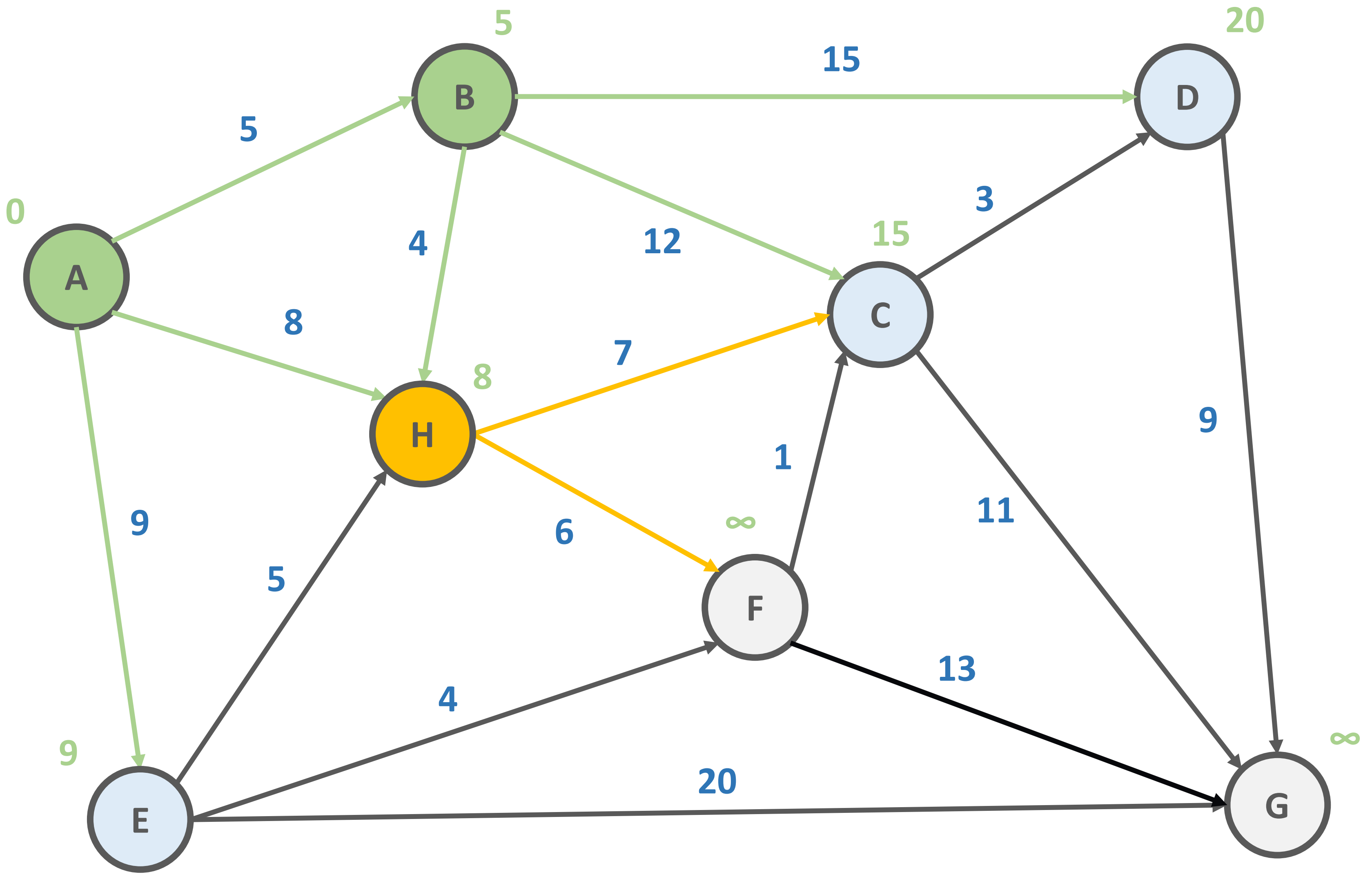
Dijkstra's Algorithm

HEAP: [E-9 D-20 C-17]



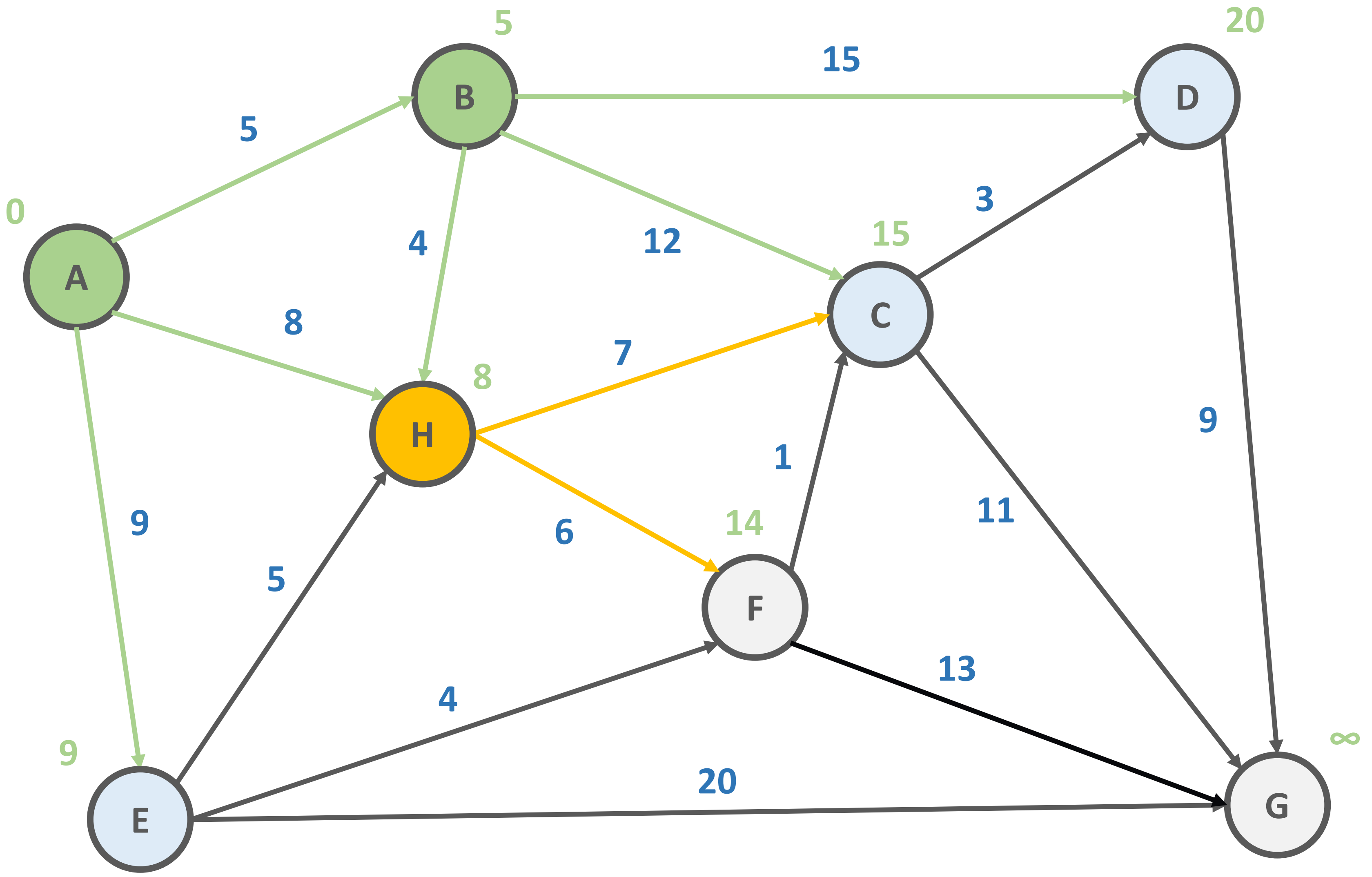
Dijkstra's Algorithm

HEAP: [E-9 D-20 C-15]



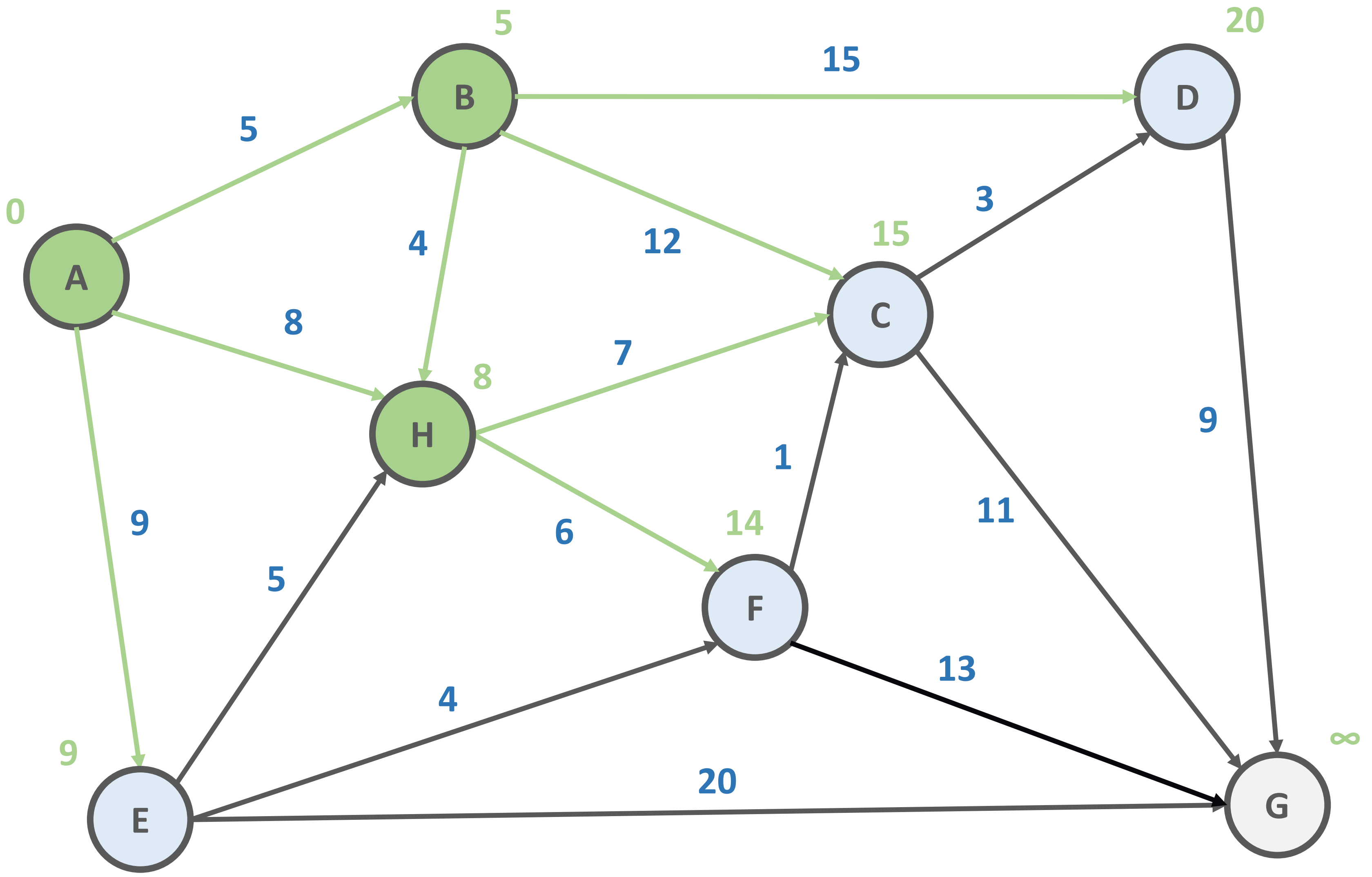
Dijkstra’s Algorithm

HEAP: [E-9 D-20 C-15]



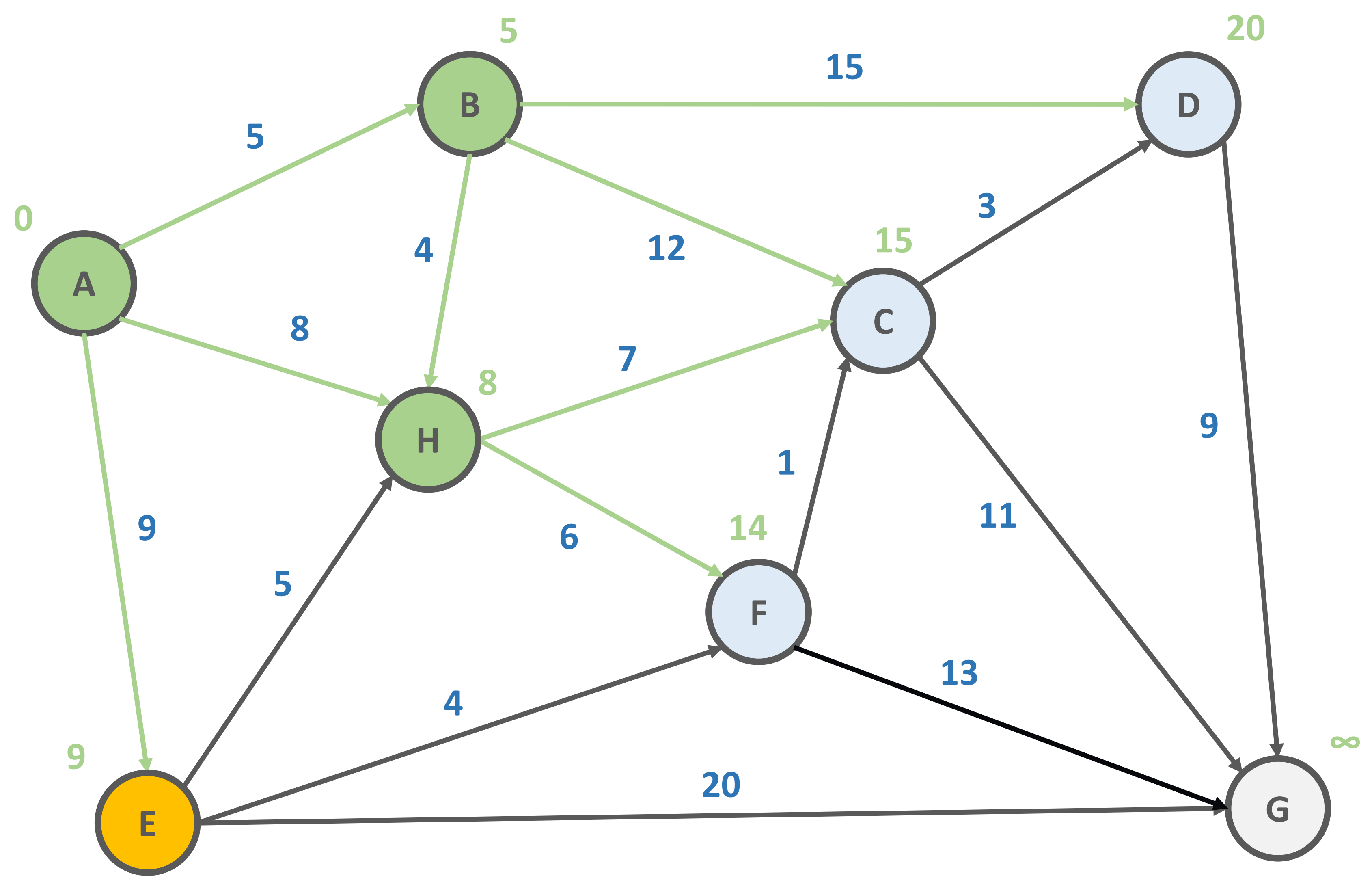
Dijkstra’s Algorithm

HEAP: [E-9 D-20 C-15 F-14]



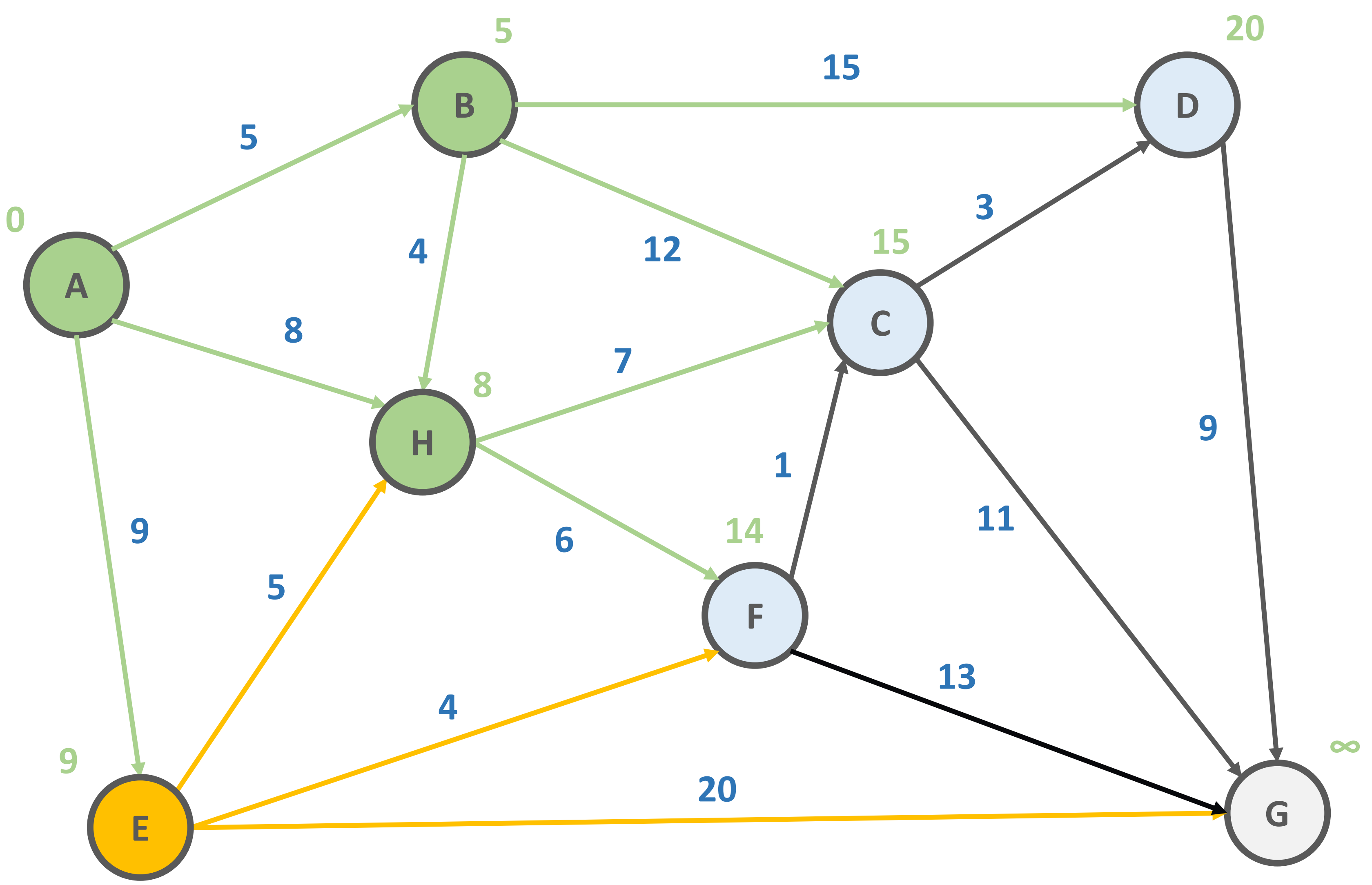
Dijkstra's Algorithm

HEAP: [E-9 D-20 C-15 F-14]



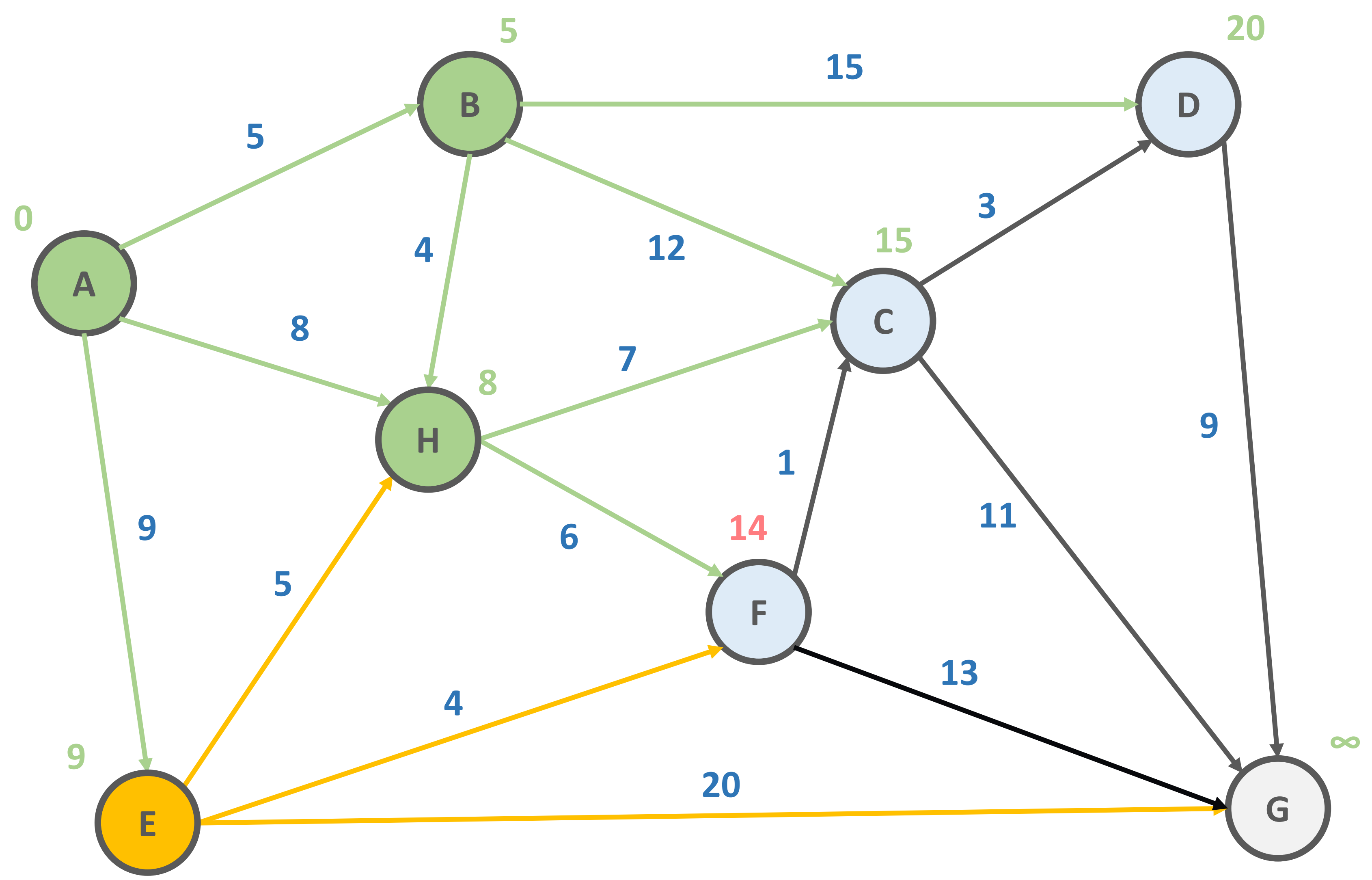
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-14]



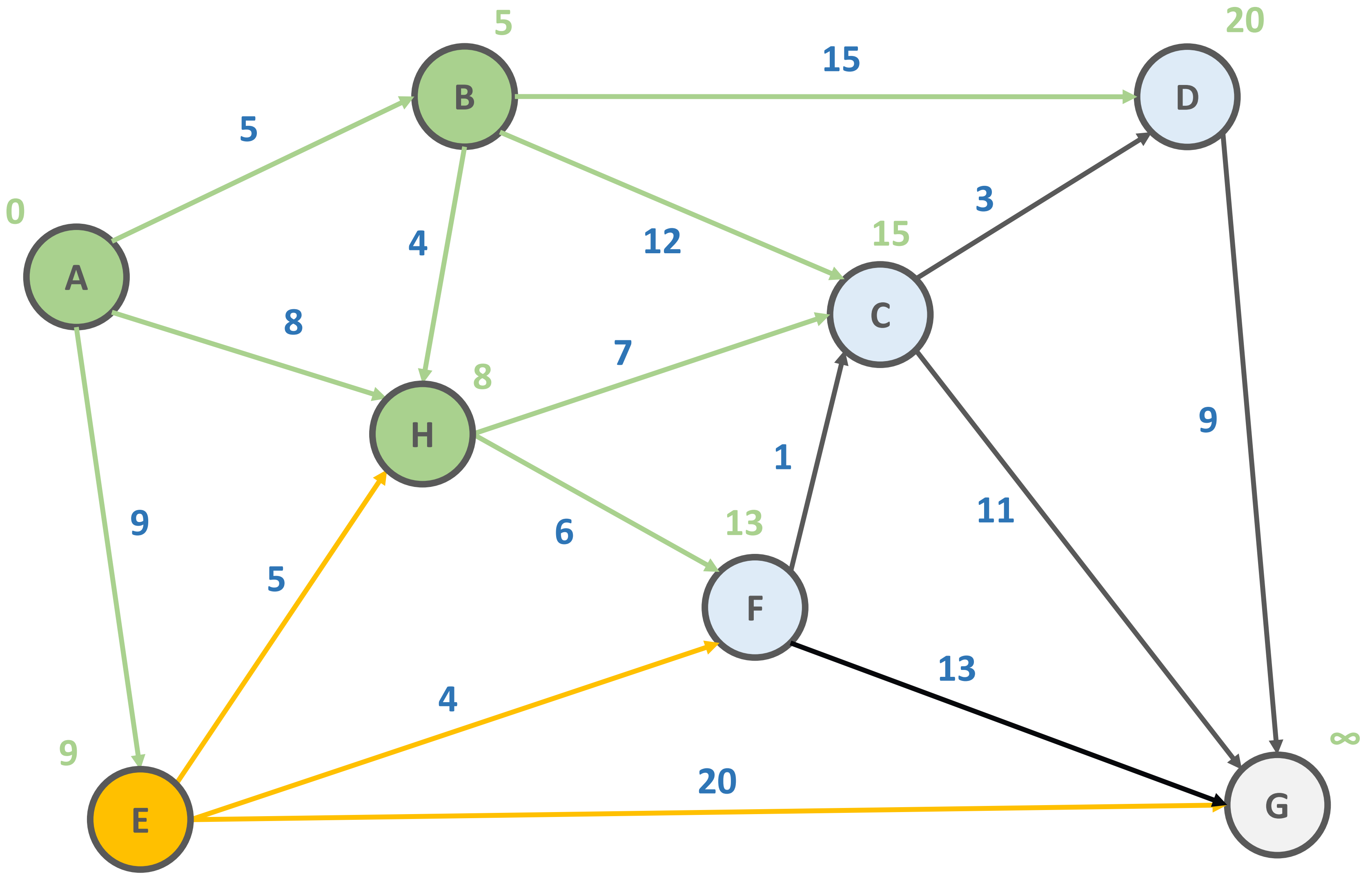
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-14]



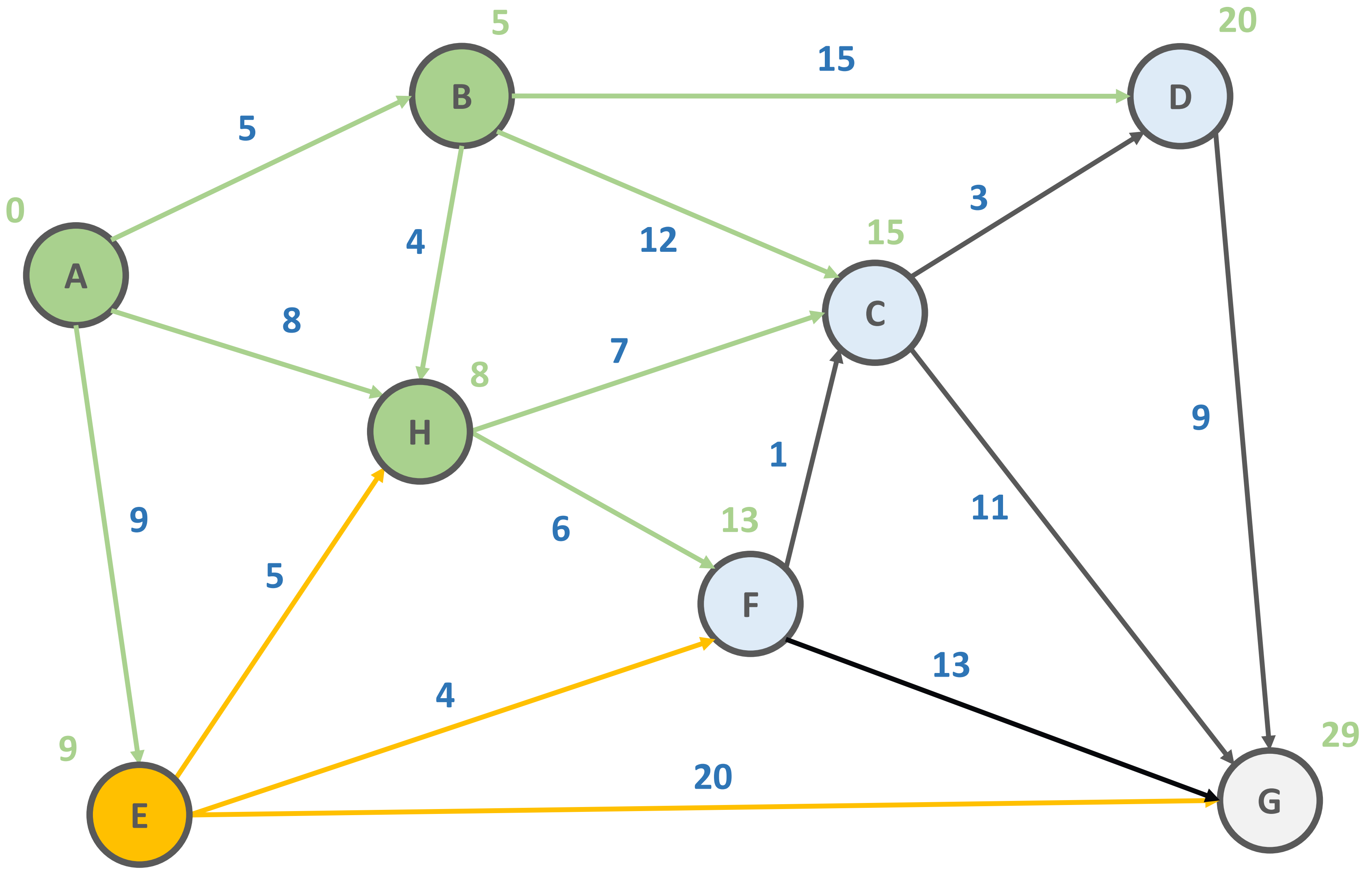
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-13]



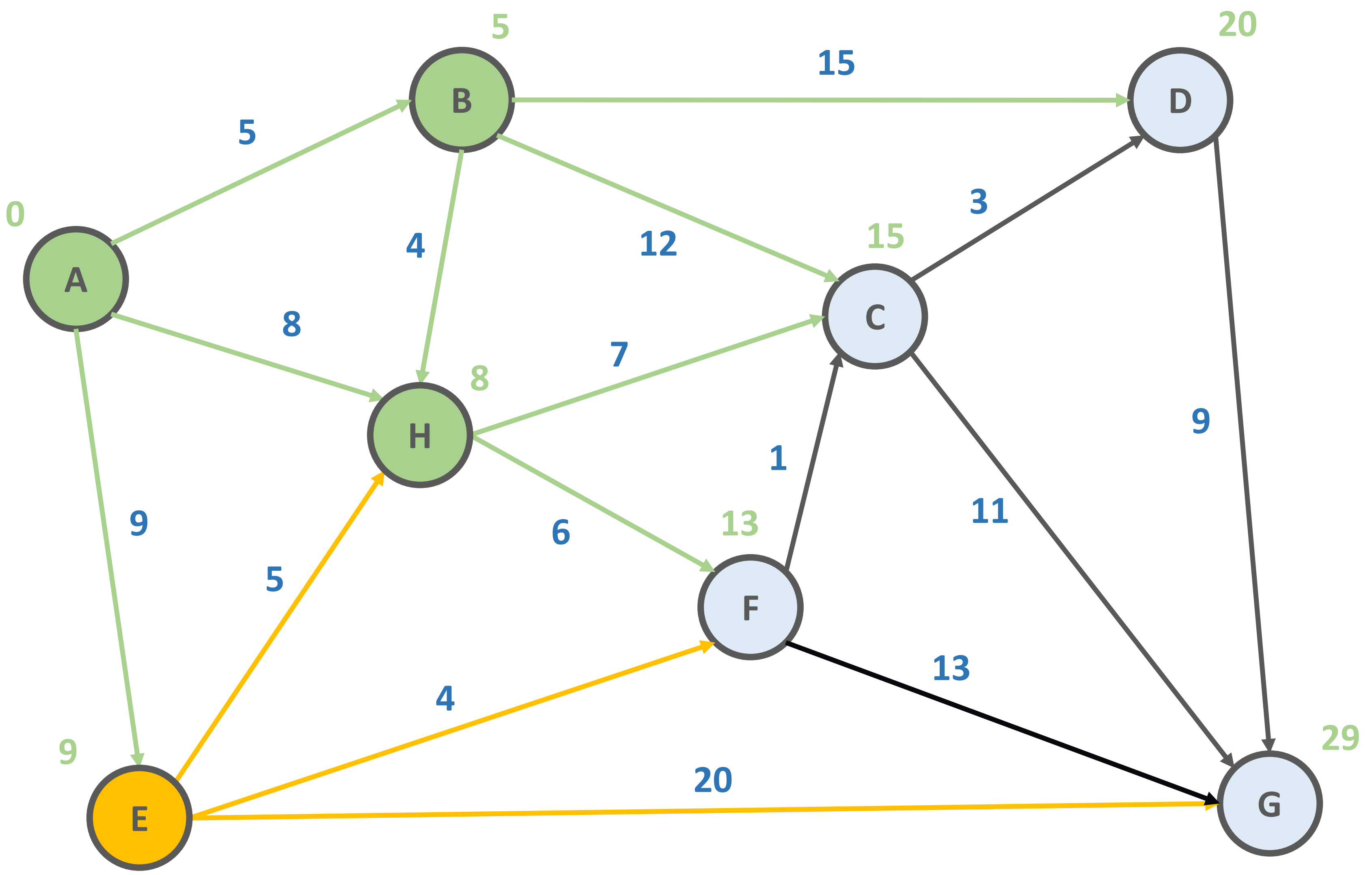
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-13]



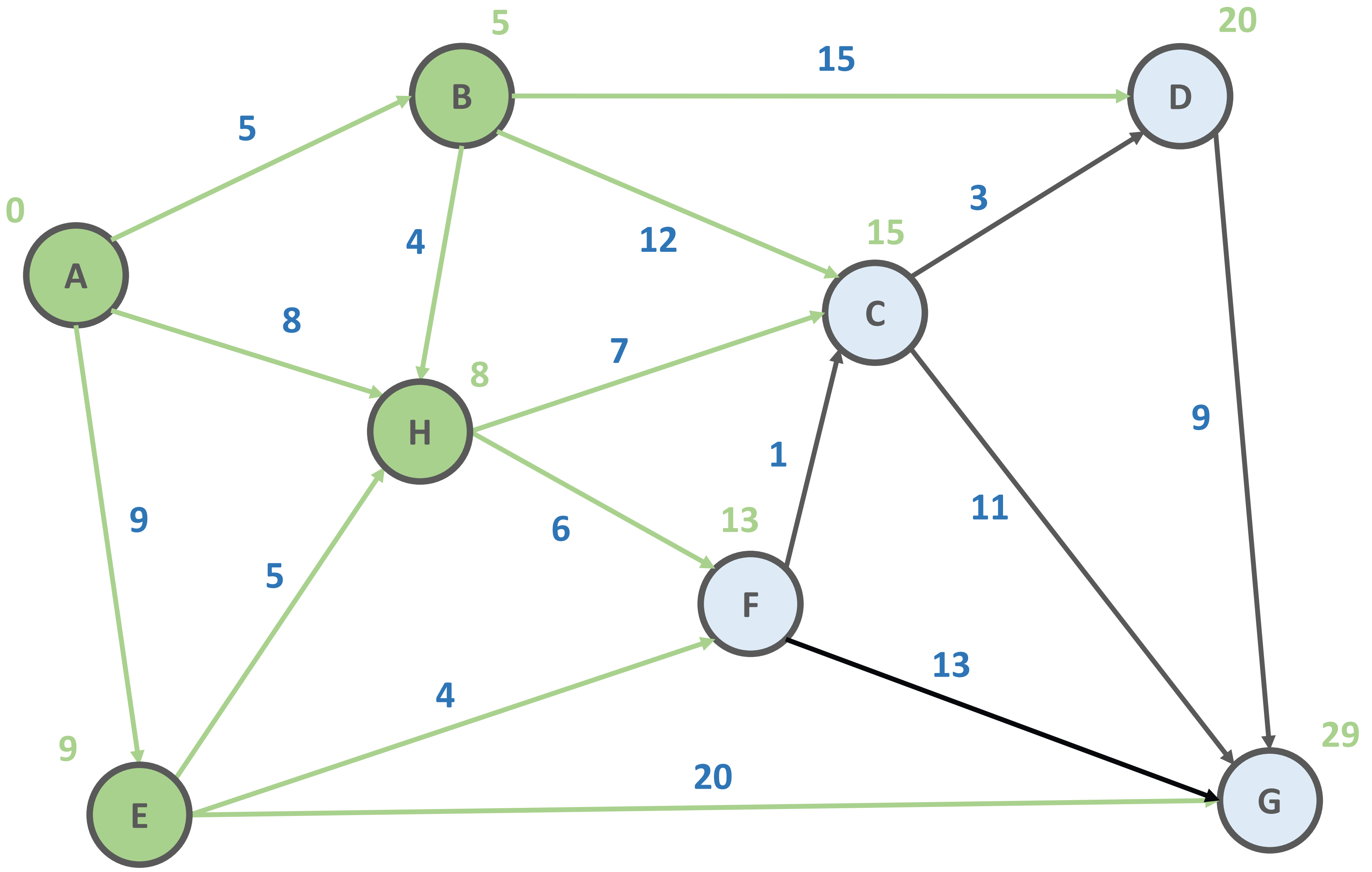
Dijkstra’s Algorithm

HEAP: [D-20 C-15 F-13 G-29]



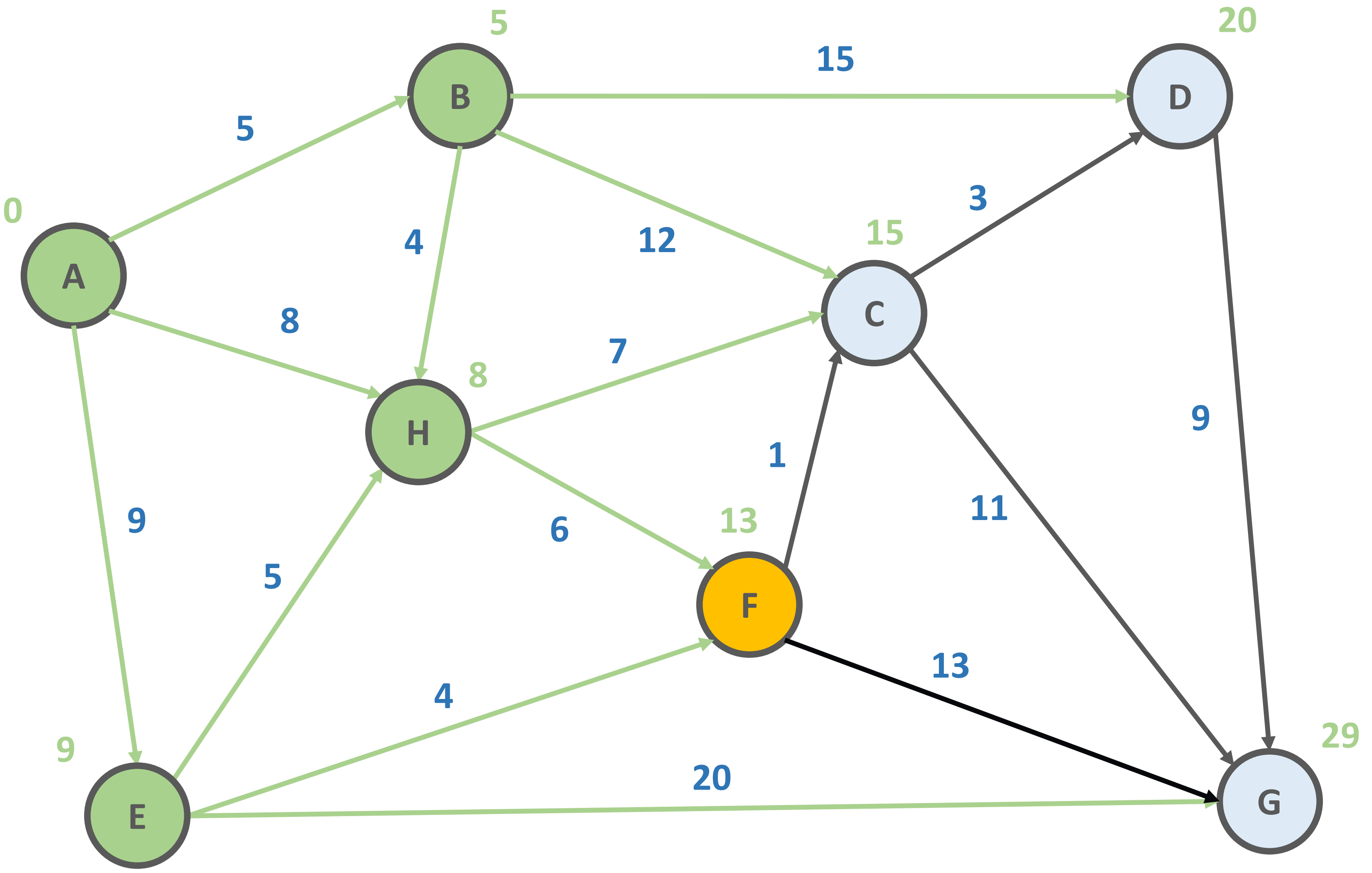
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-13 G-29]



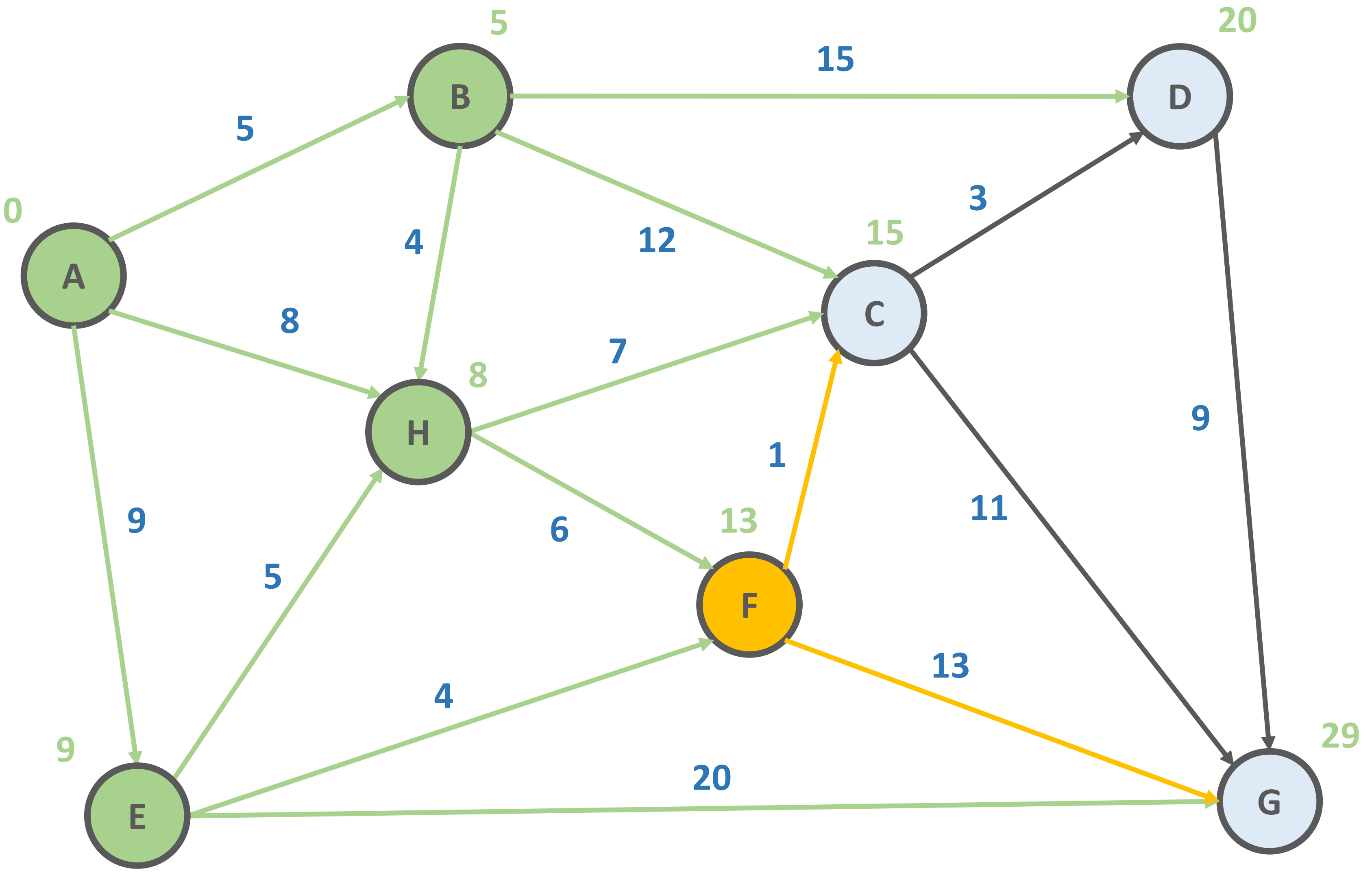
Dijkstra's Algorithm

HEAP: [D-20 C-15 F-13 G-29]



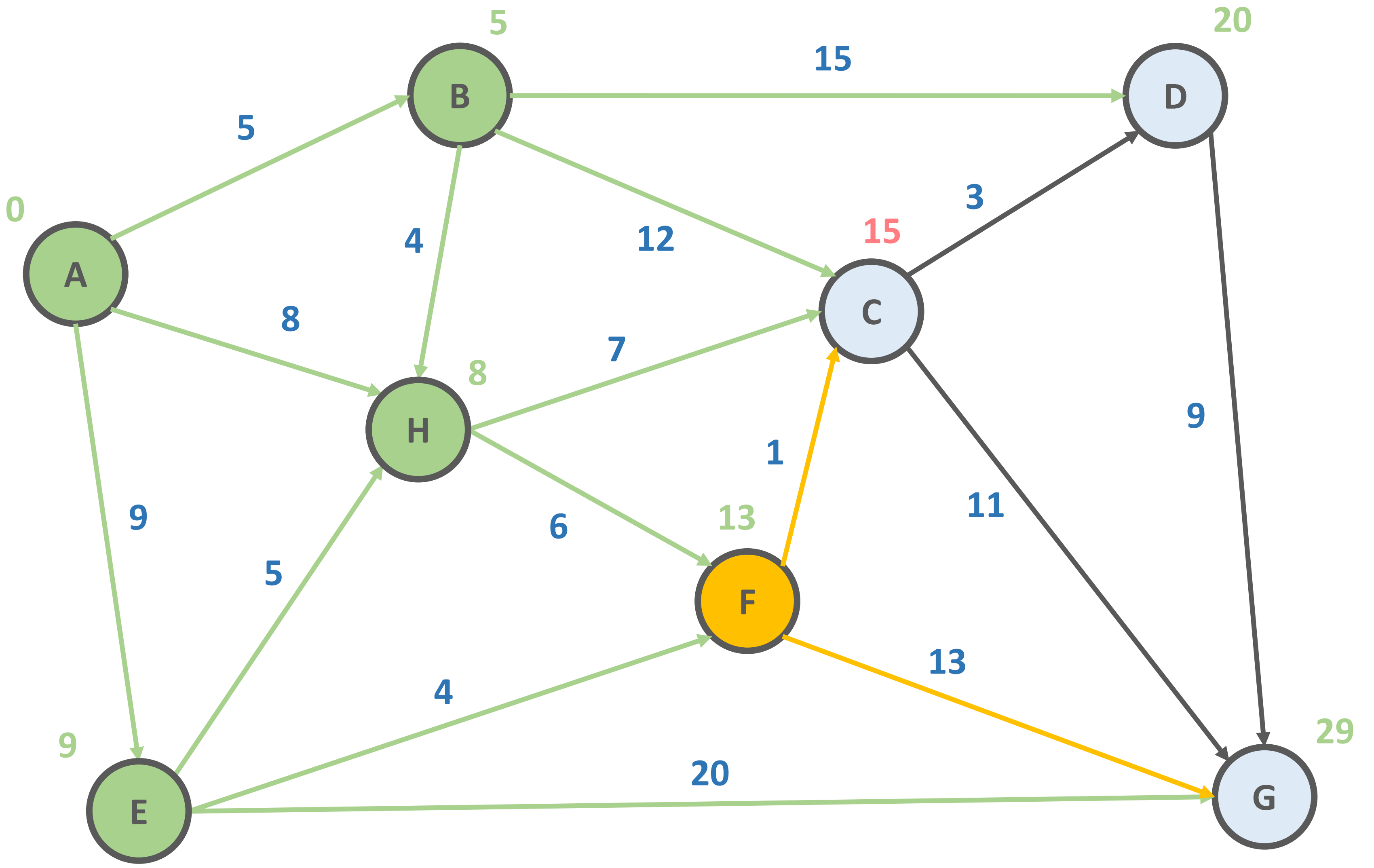
Dijkstra's Algorithm

HEAP: [D-20 C-15 G-29]



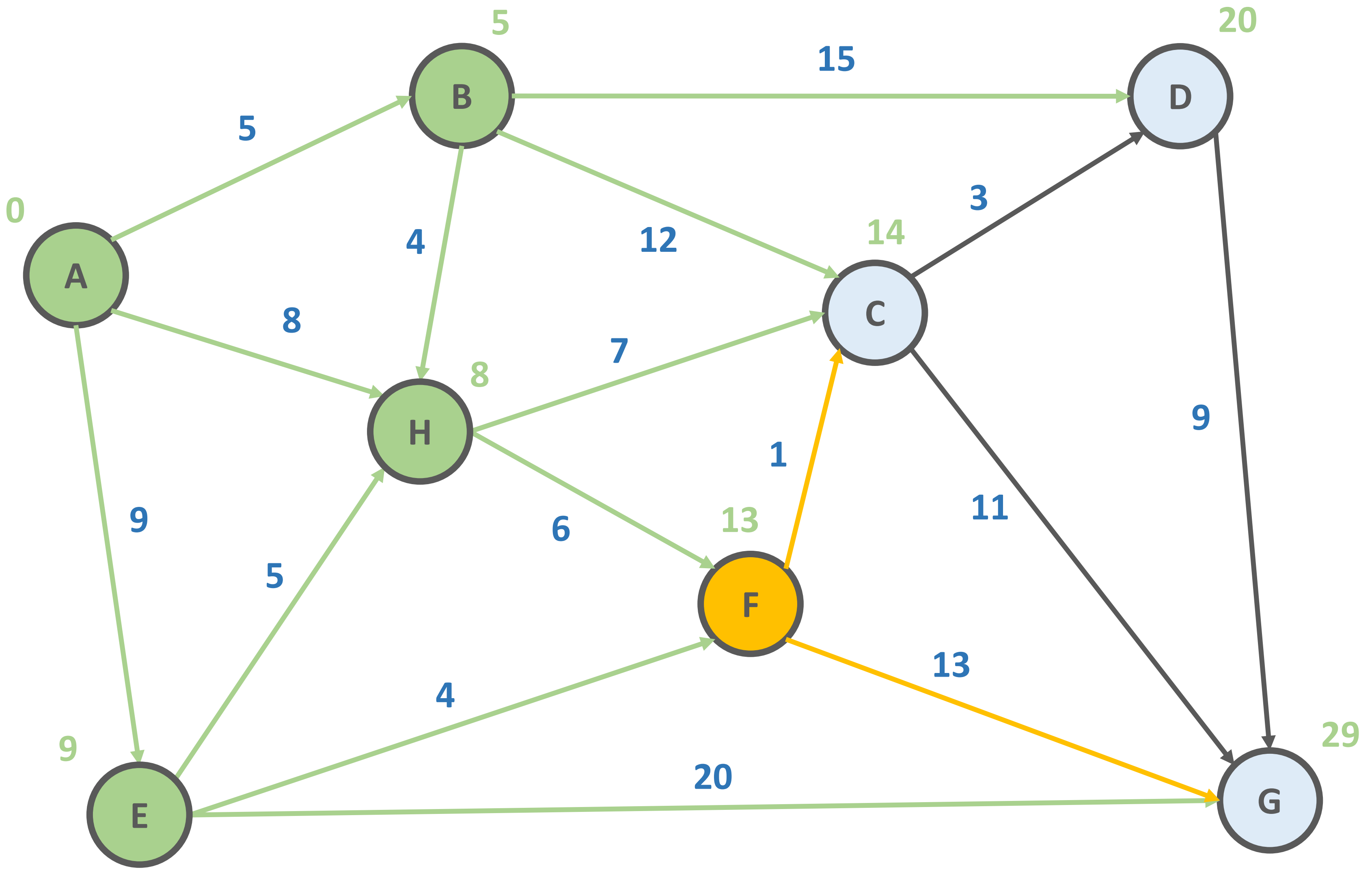
Dijkstra's Algorithm

HEAP: [D-20 C-15 G-29]



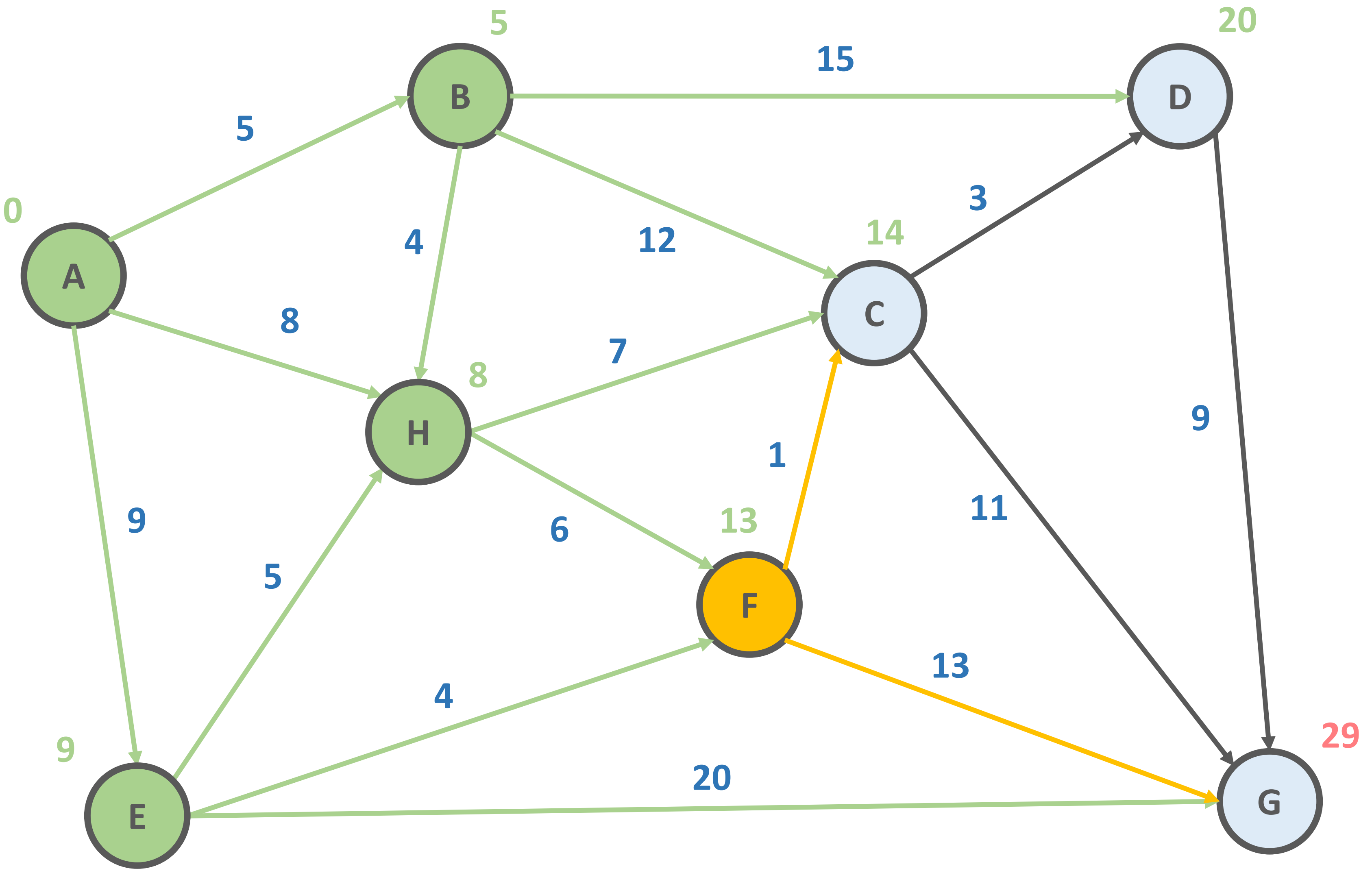
Dijkstra's Algorithm

HEAP: [D-20 C-14 G-29]



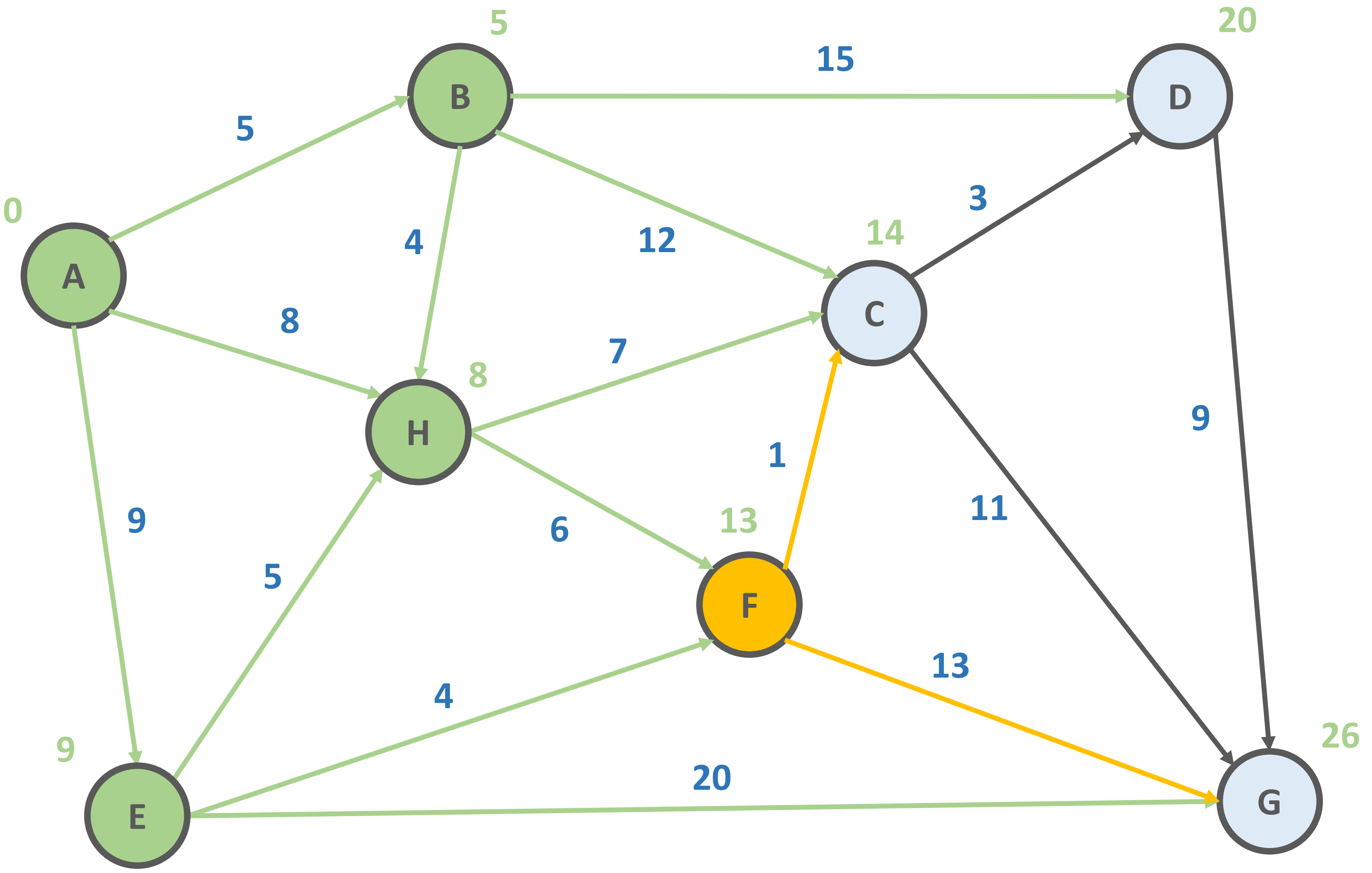
Dijkstra's Algorithm

HEAP: [D-20 C-14 G-29]



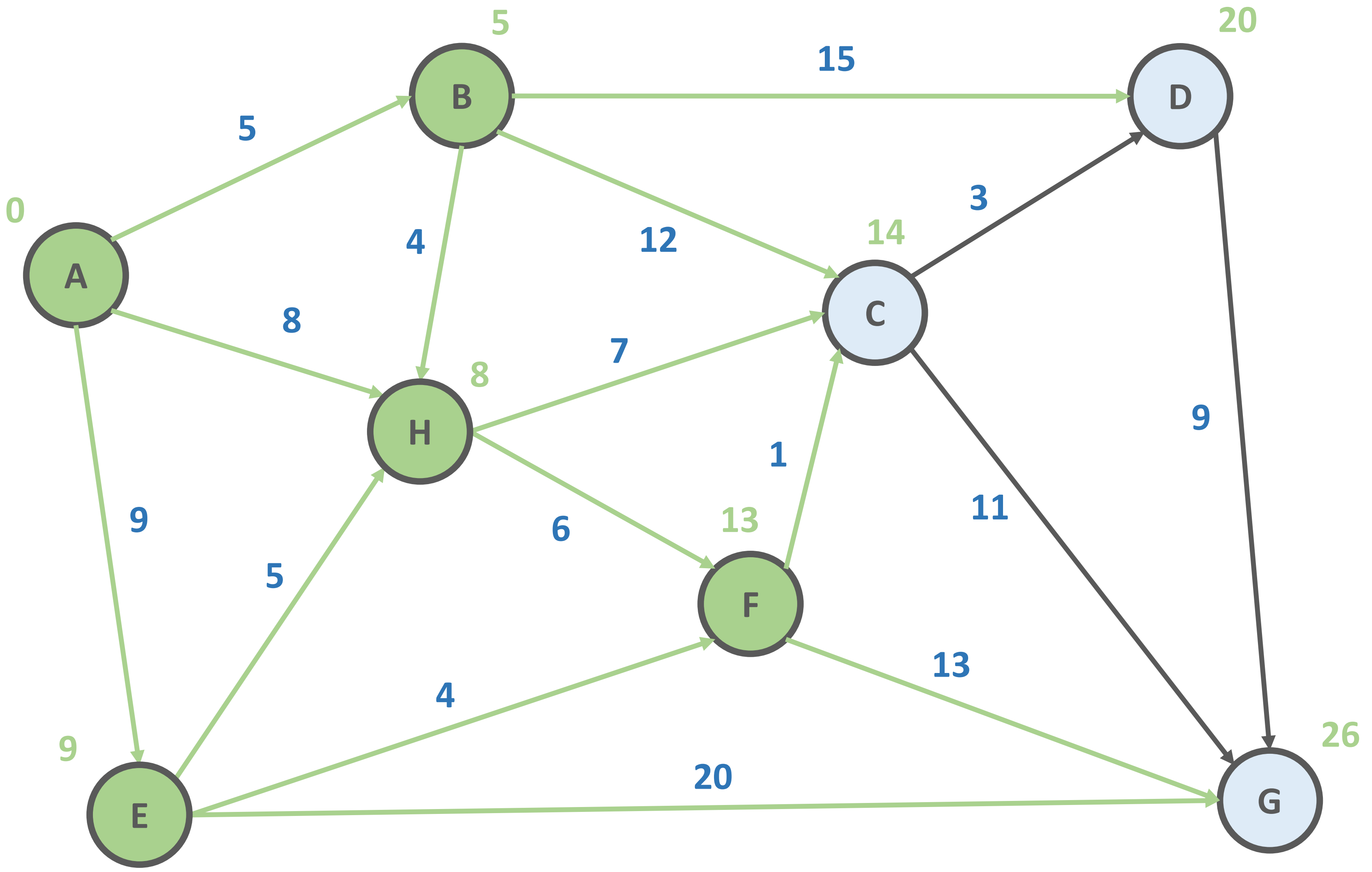
Dijkstra's Algorithm

HEAP: [D-20 C-14 G-26]



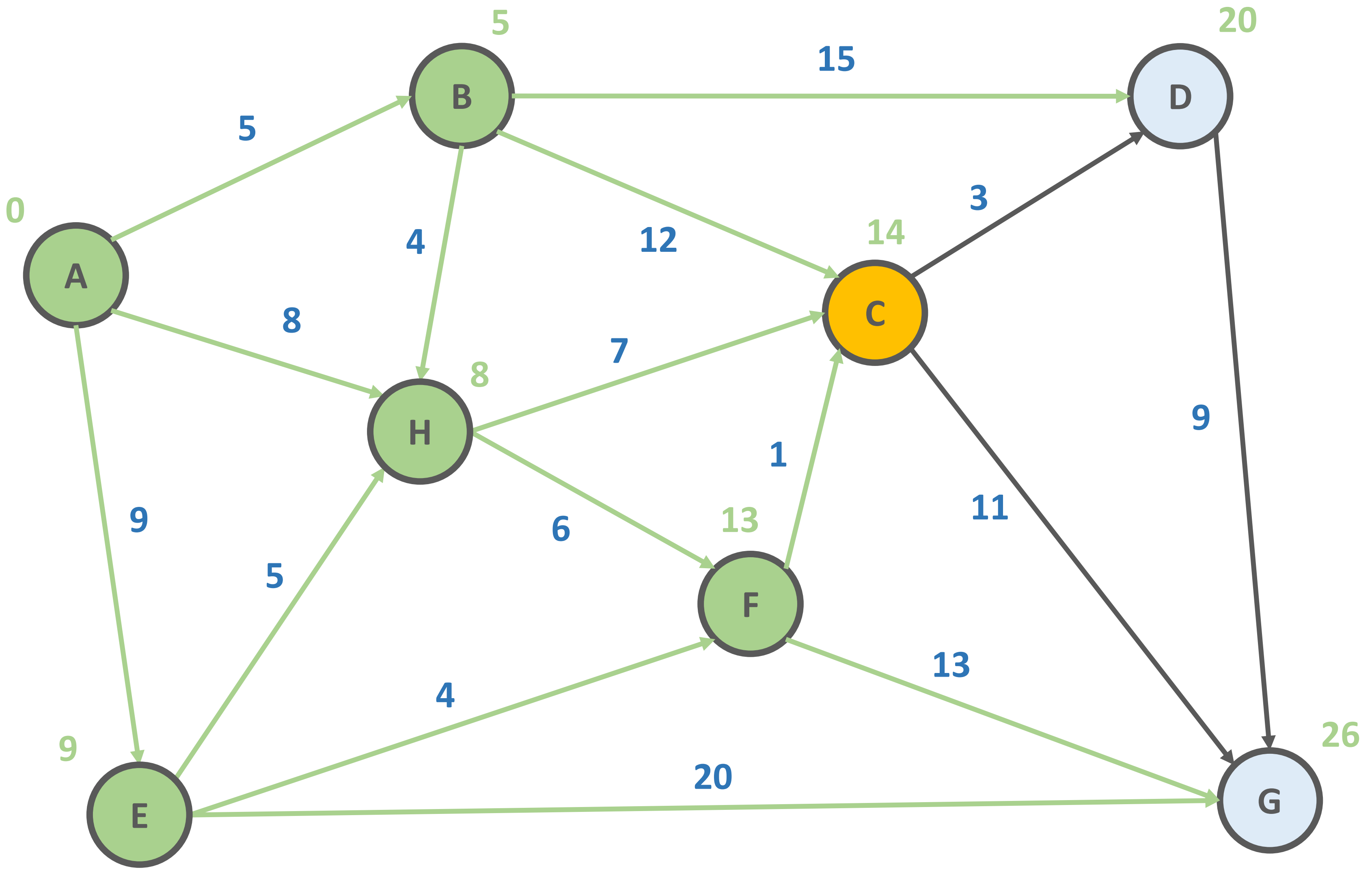
Dijkstra's Algorithm

HEAP: [D-20 C-14 G-26]



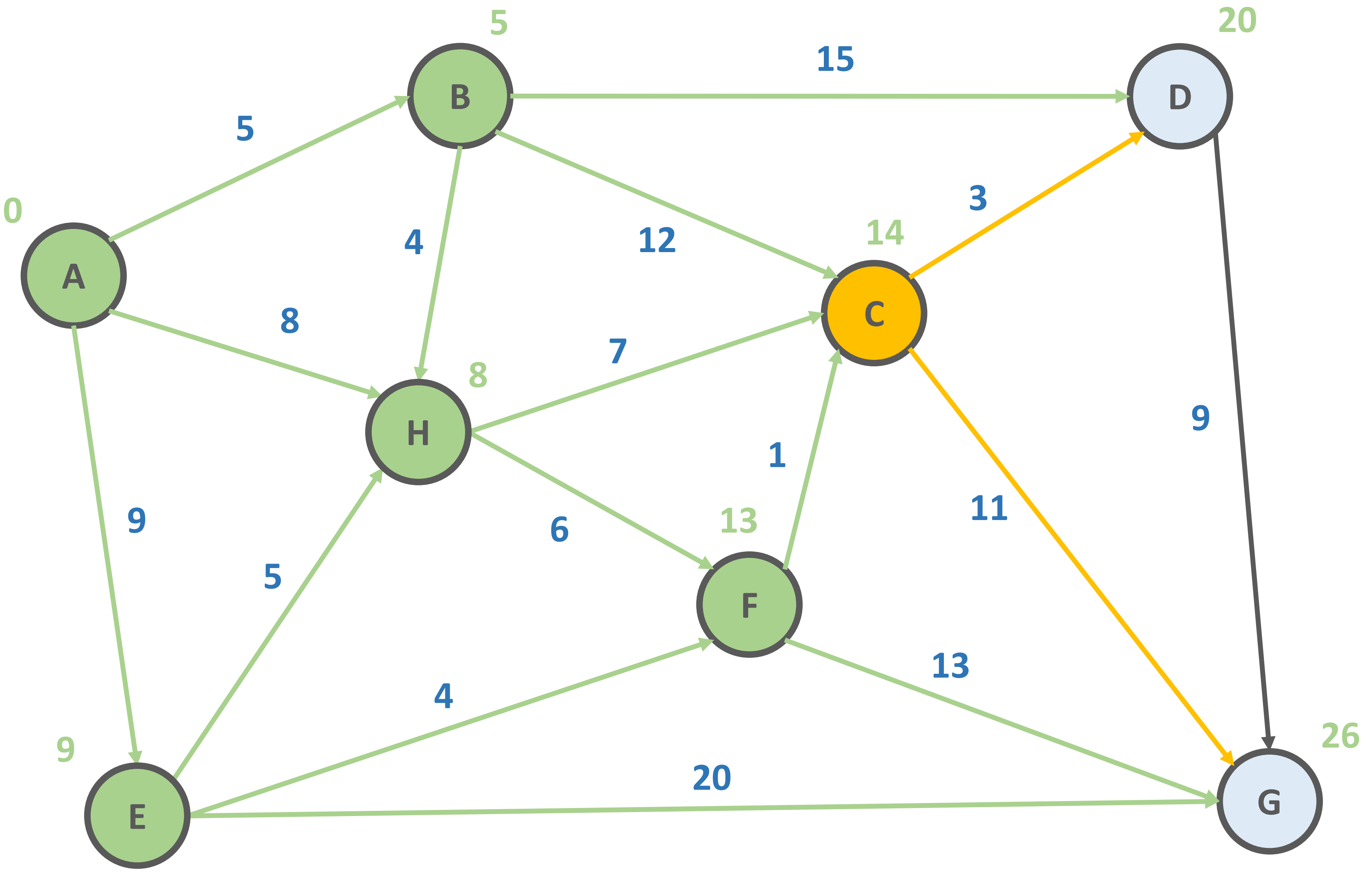
Dijkstra’s Algorithm

HEAP: [D-20 C-14 G-26]



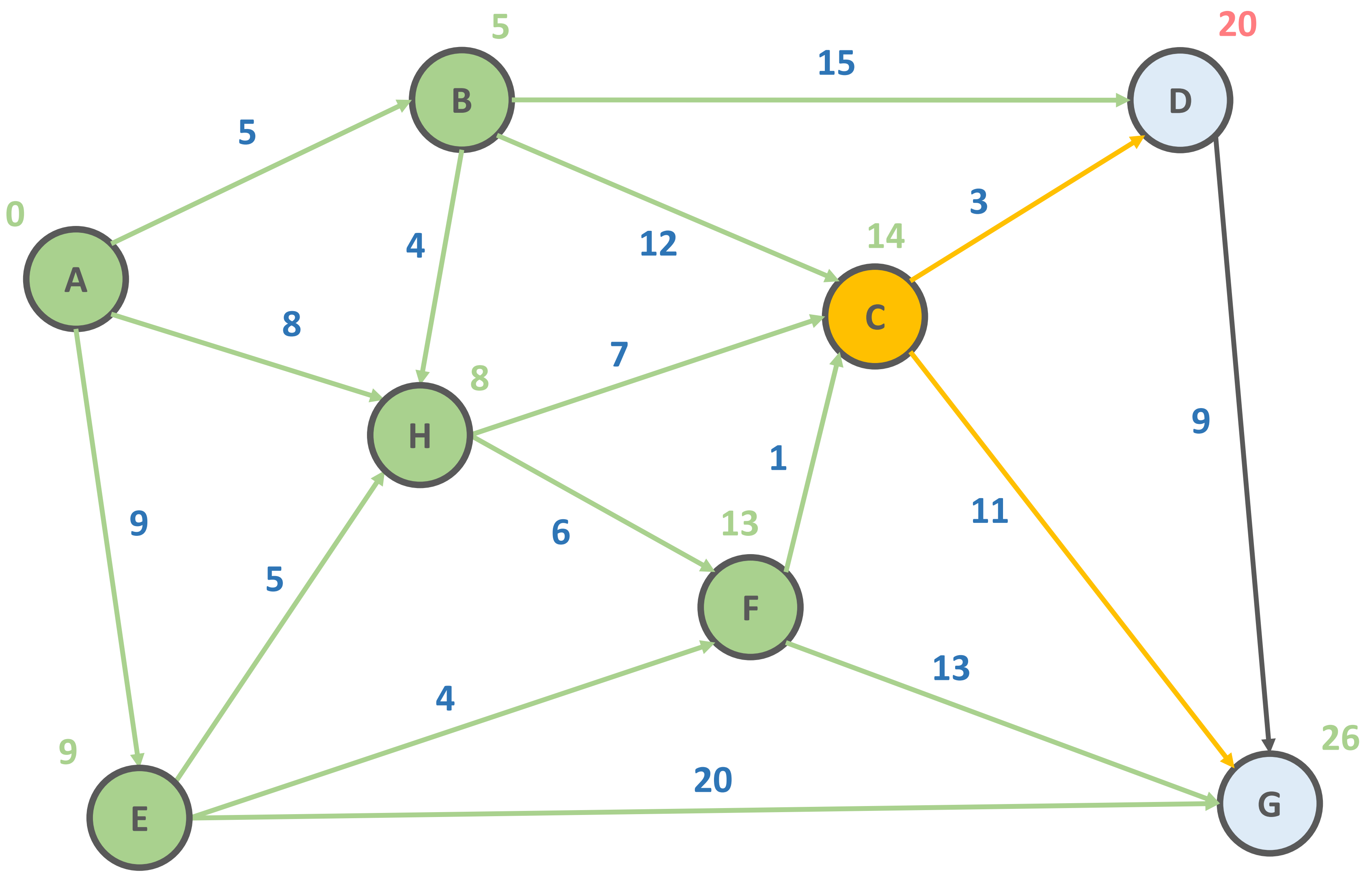
Dijkstra’s Algorithm

HEAP: [D-20 G-26]



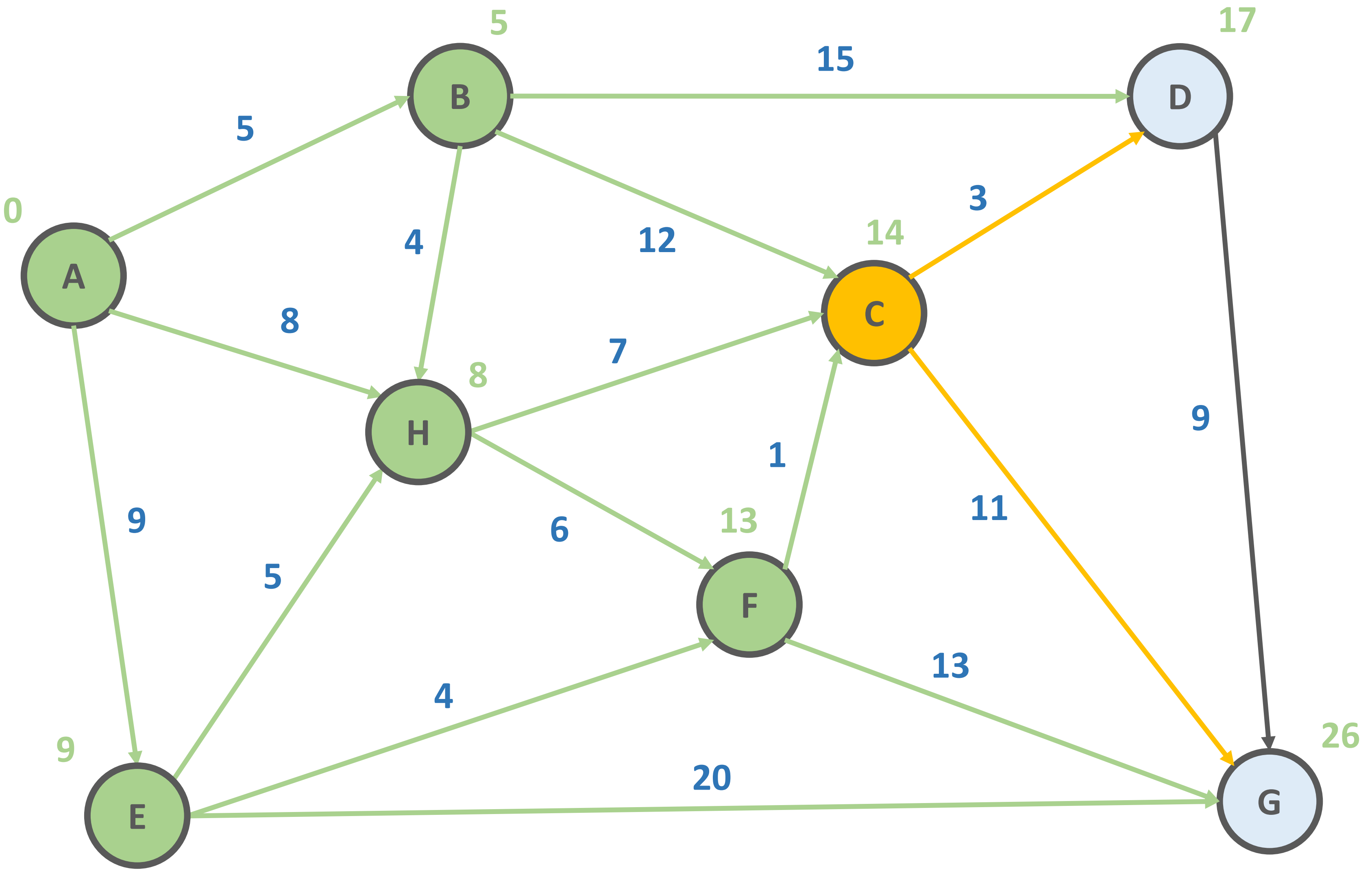
Dijkstra’s Algorithm

HEAP: [D-20 G-26]



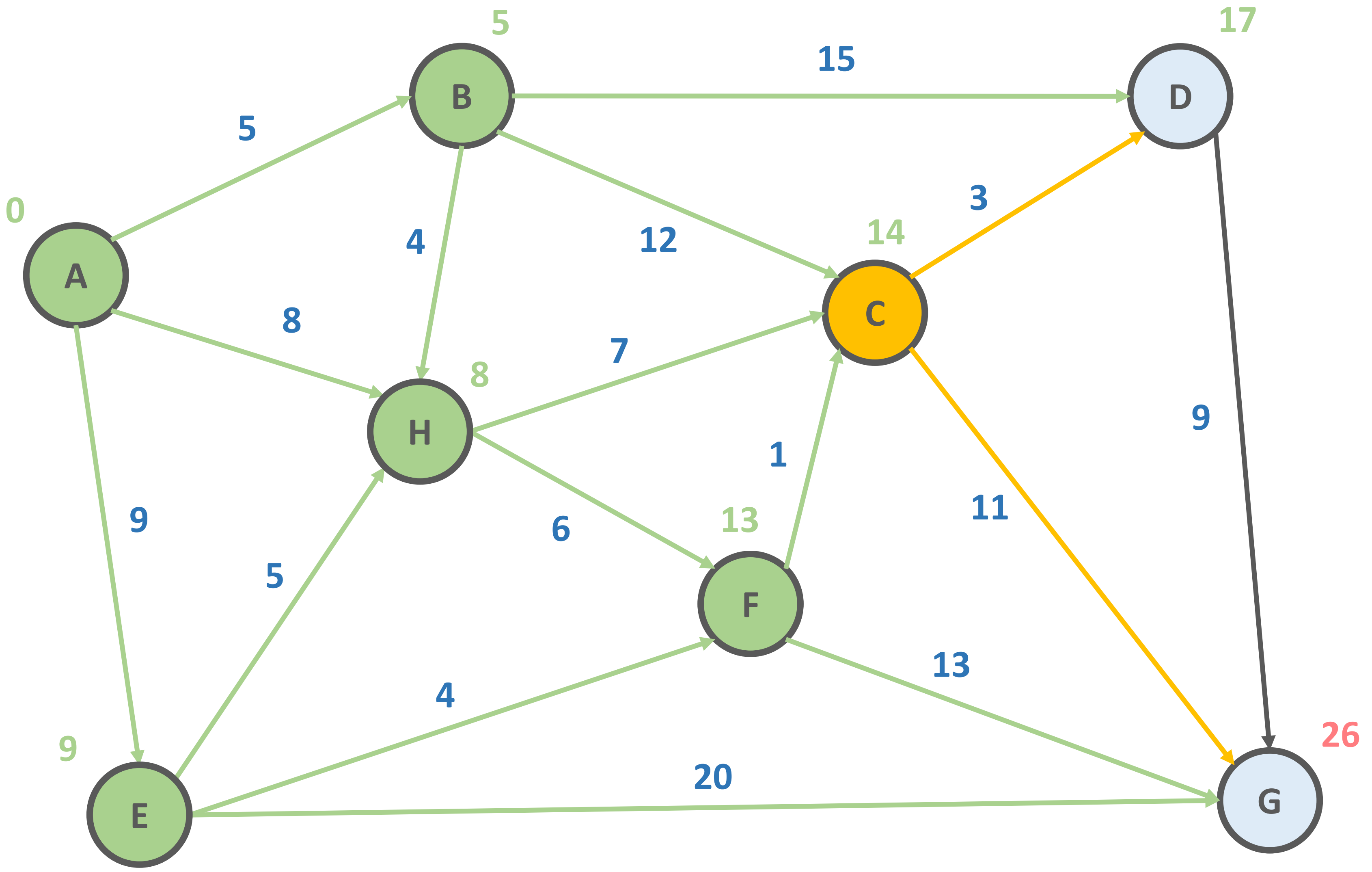
Dijkstra's Algorithm

HEAP: [D-17 G-26]



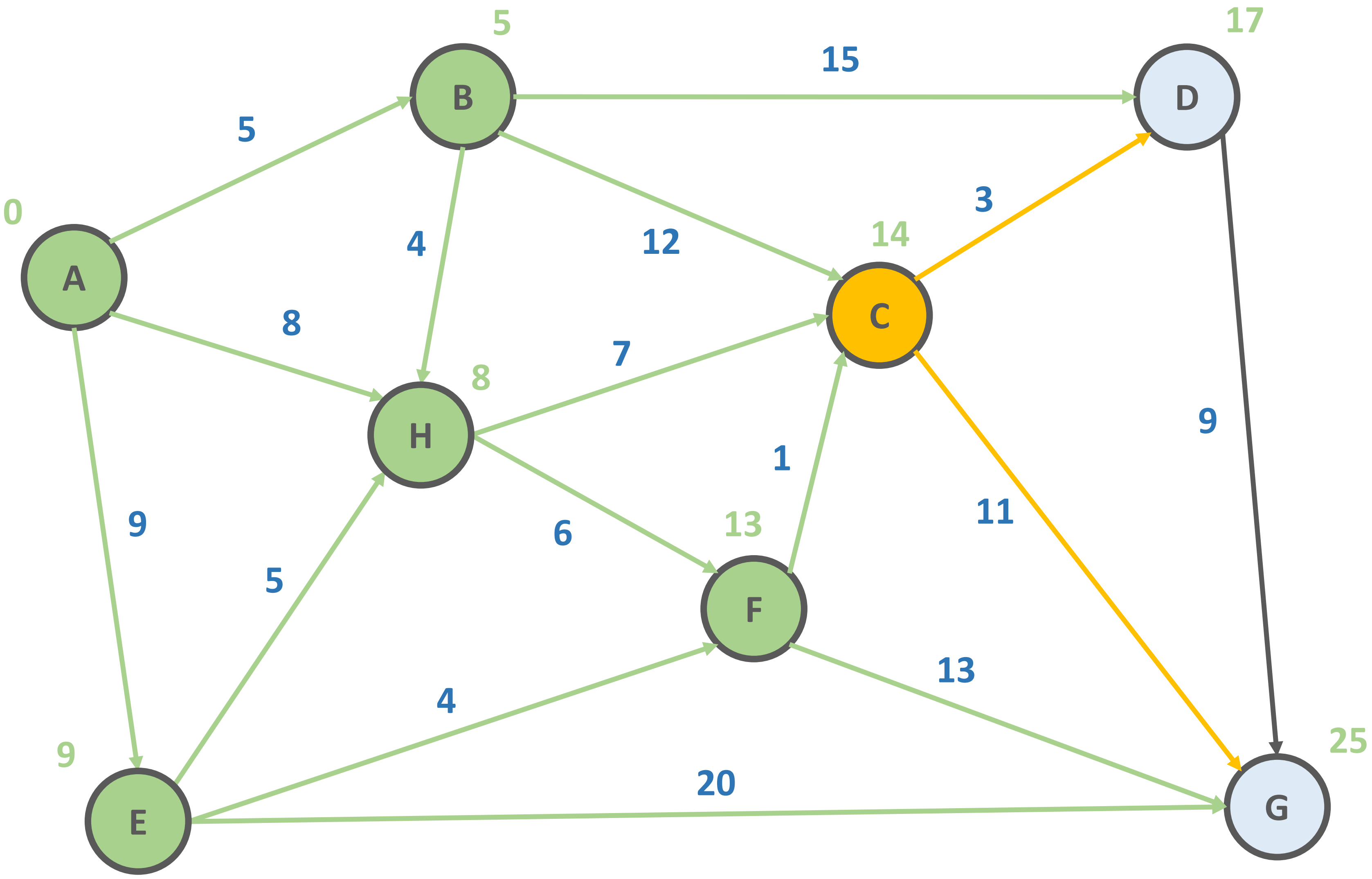
Dijkstra's Algorithm

HEAP: [D-17 G-26]



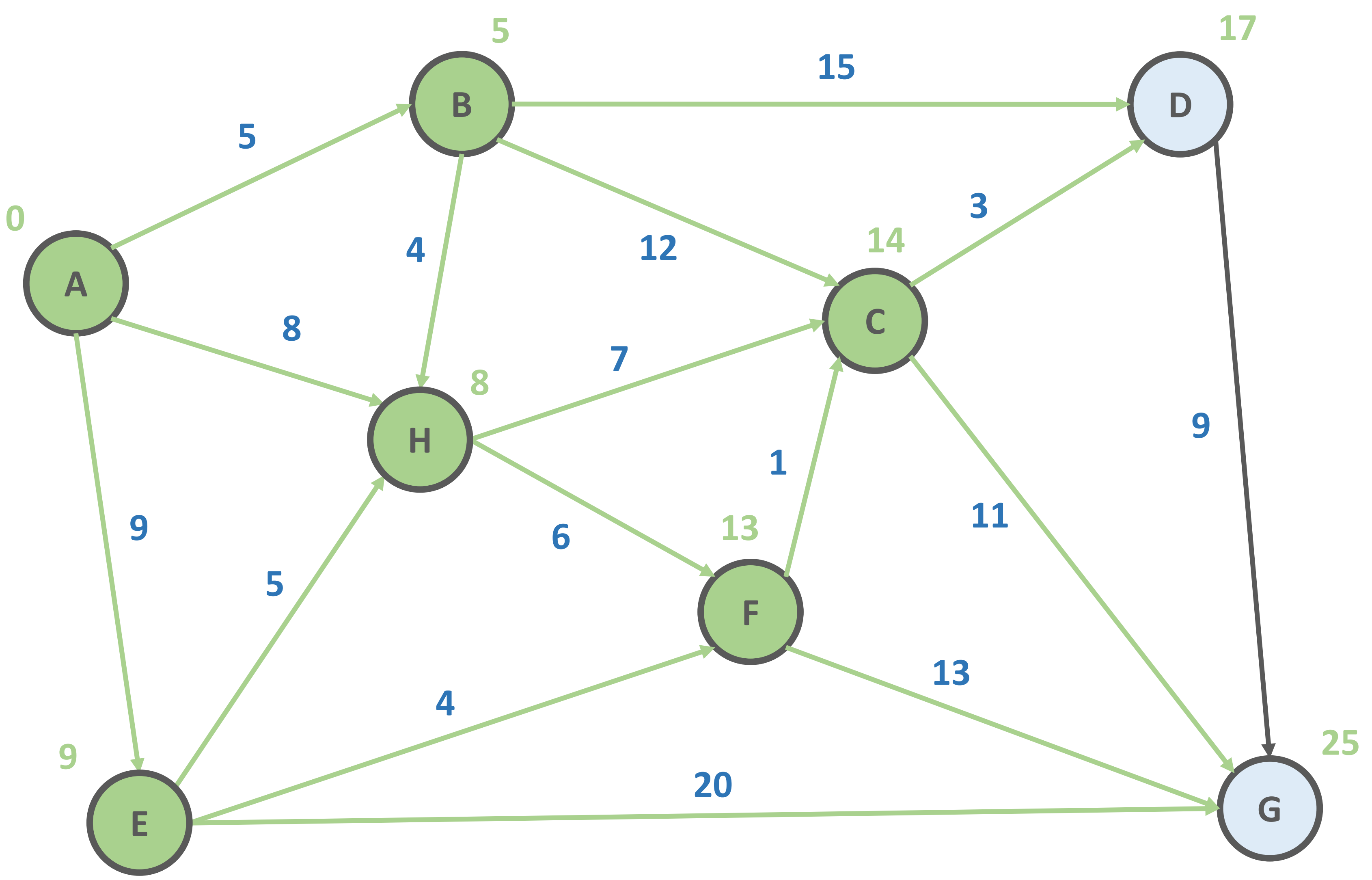
Dijkstra's Algorithm

HEAP: [D-17 G-25]



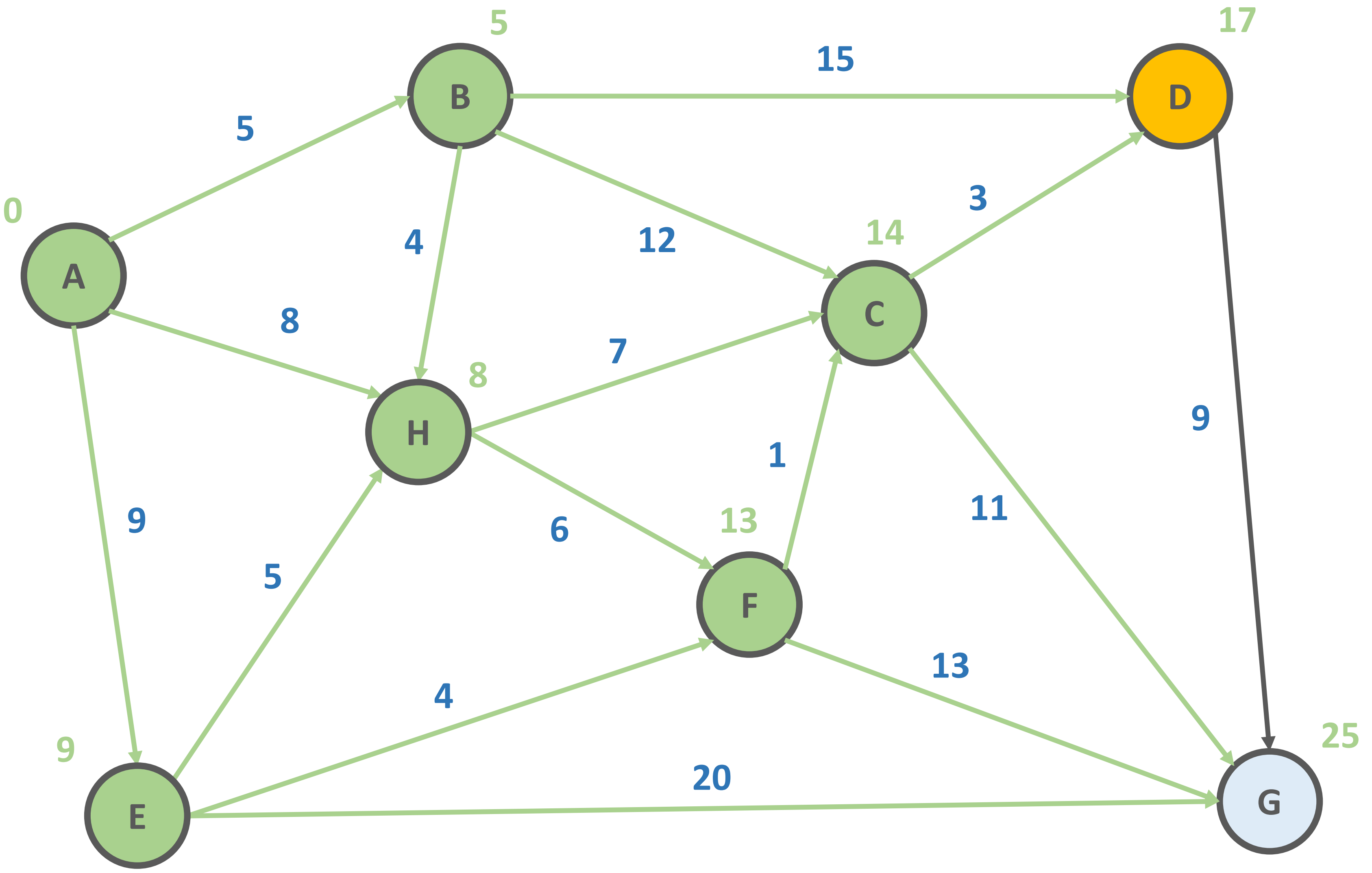
Dijkstra's Algorithm

HEAP: [D-17 G-25]



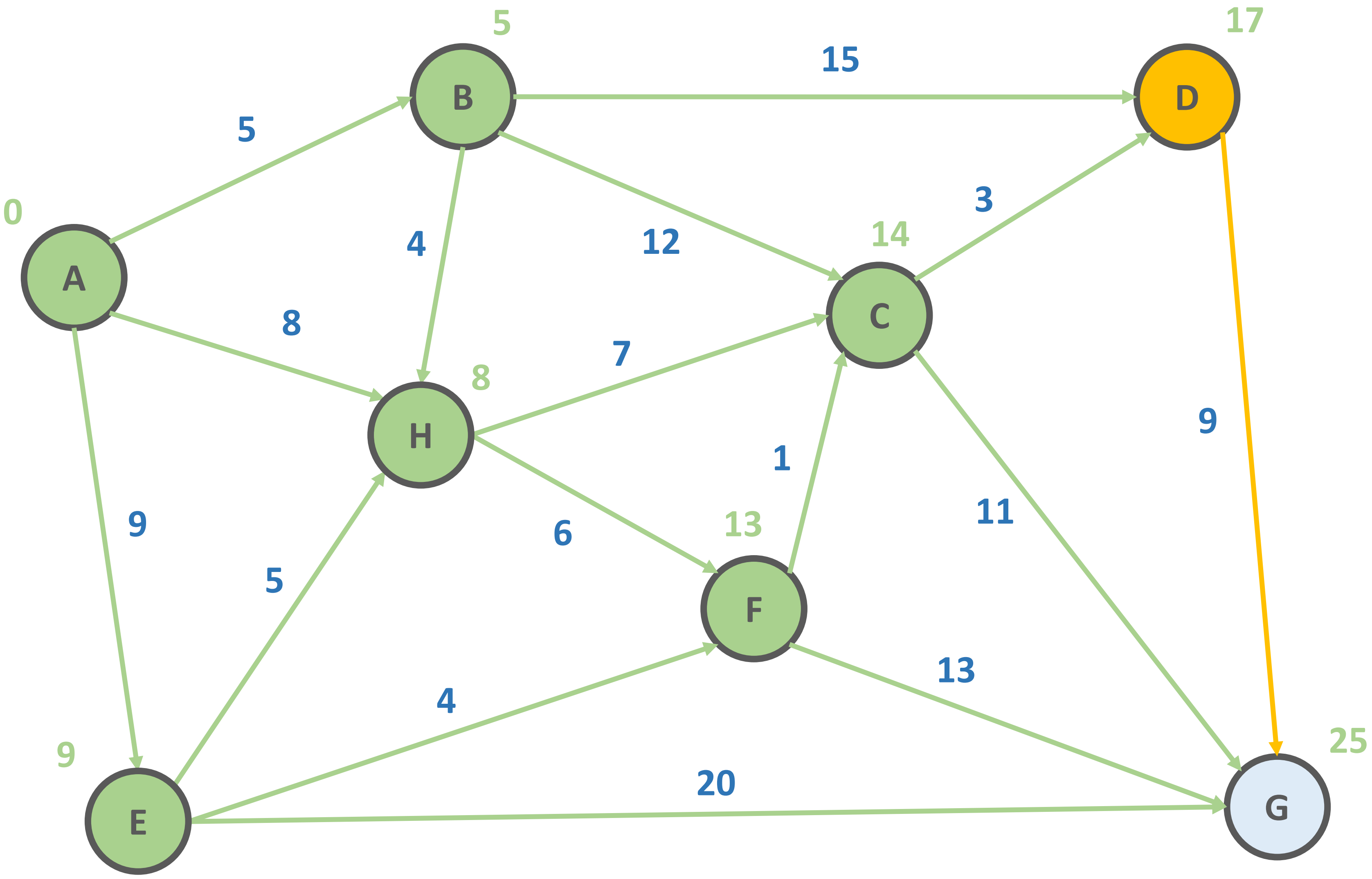
Dijkstra's Algorithm

HEAP: [D-17 G-25]



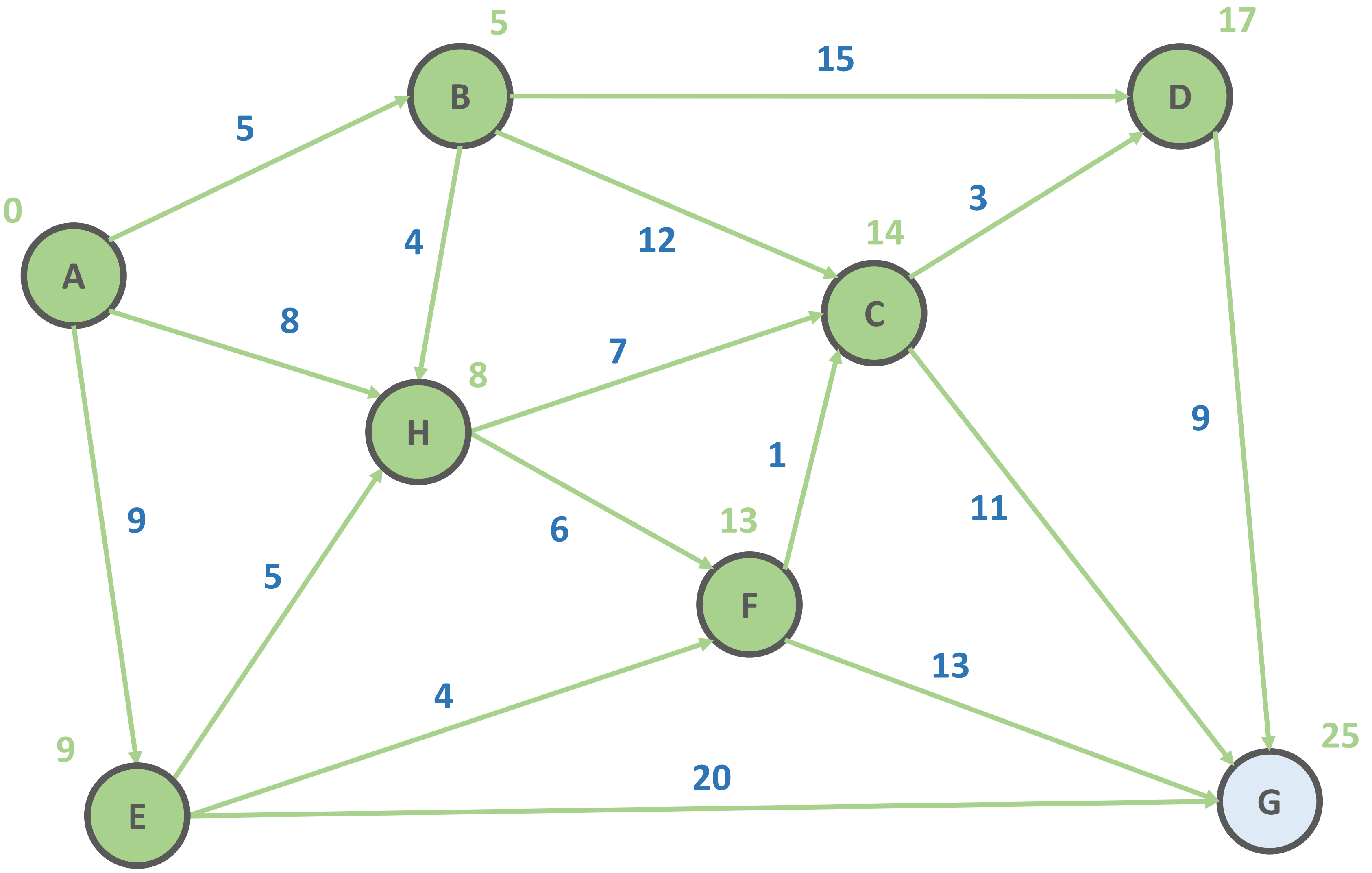
Dijkstra’s Algorithm

HEAP: [G-25]



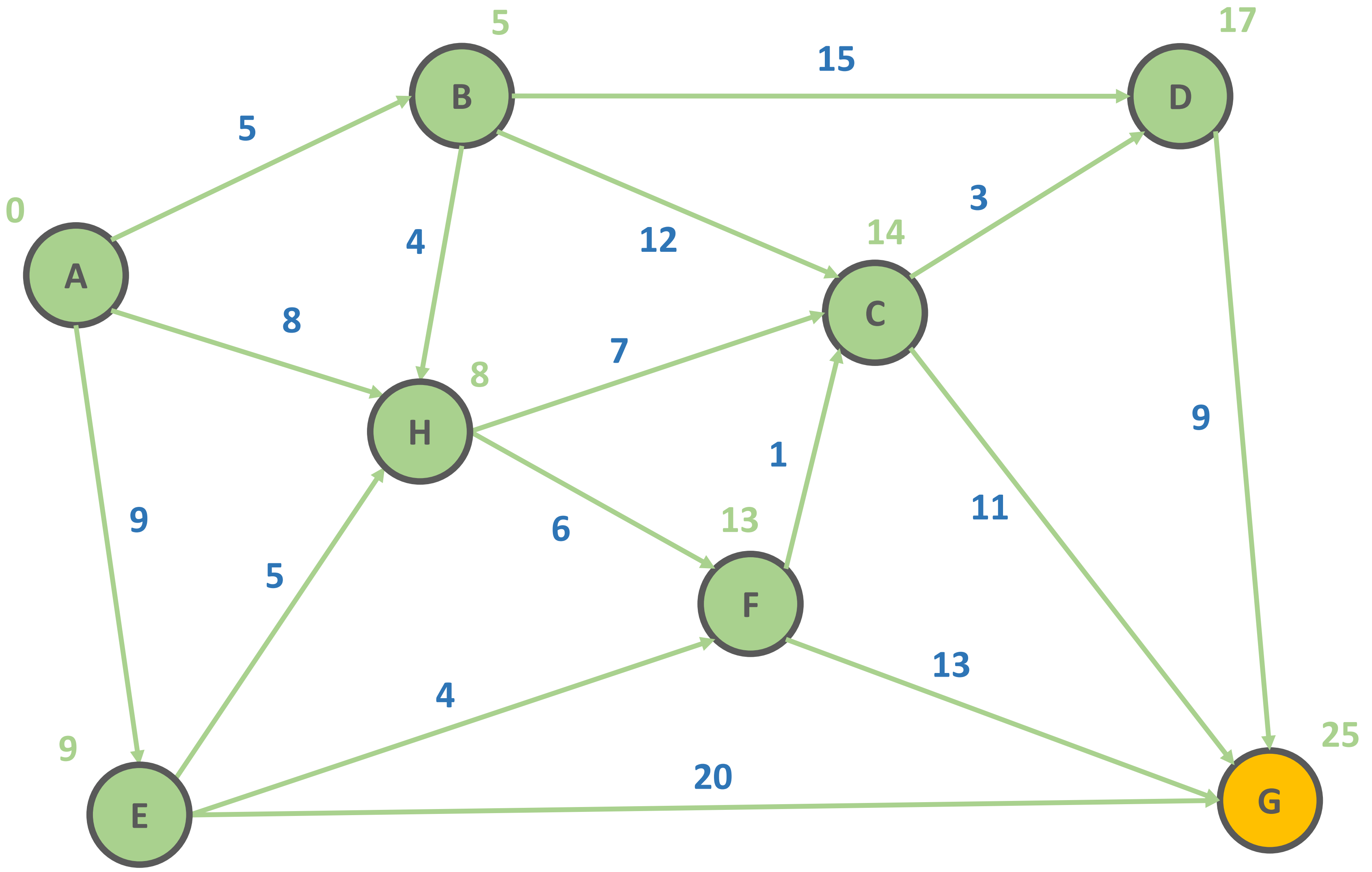
Dijkstra's Algorithm

HEAP: [G-25]



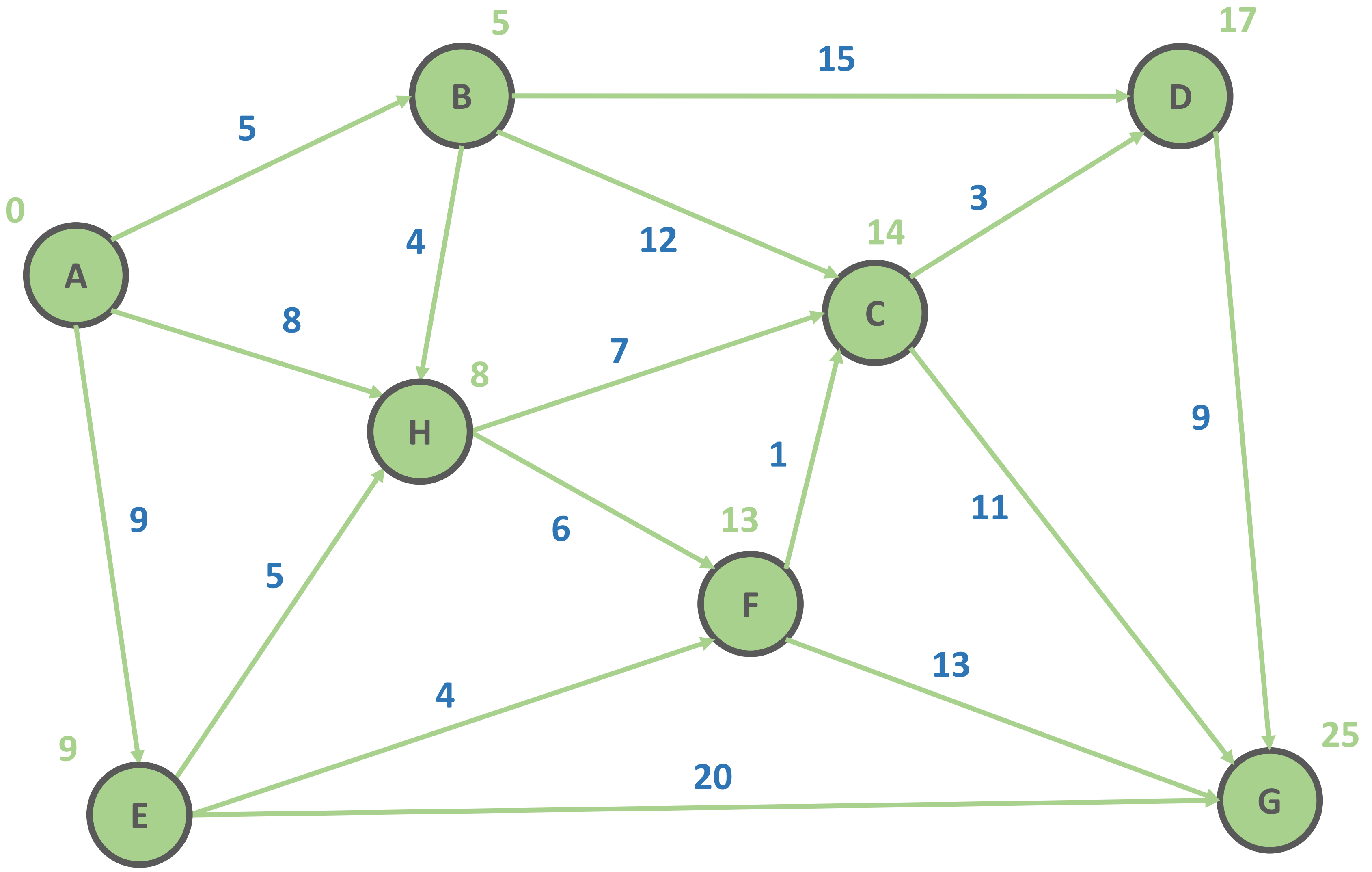
Dijkstra's Algorithm

HEAP: [G-25]



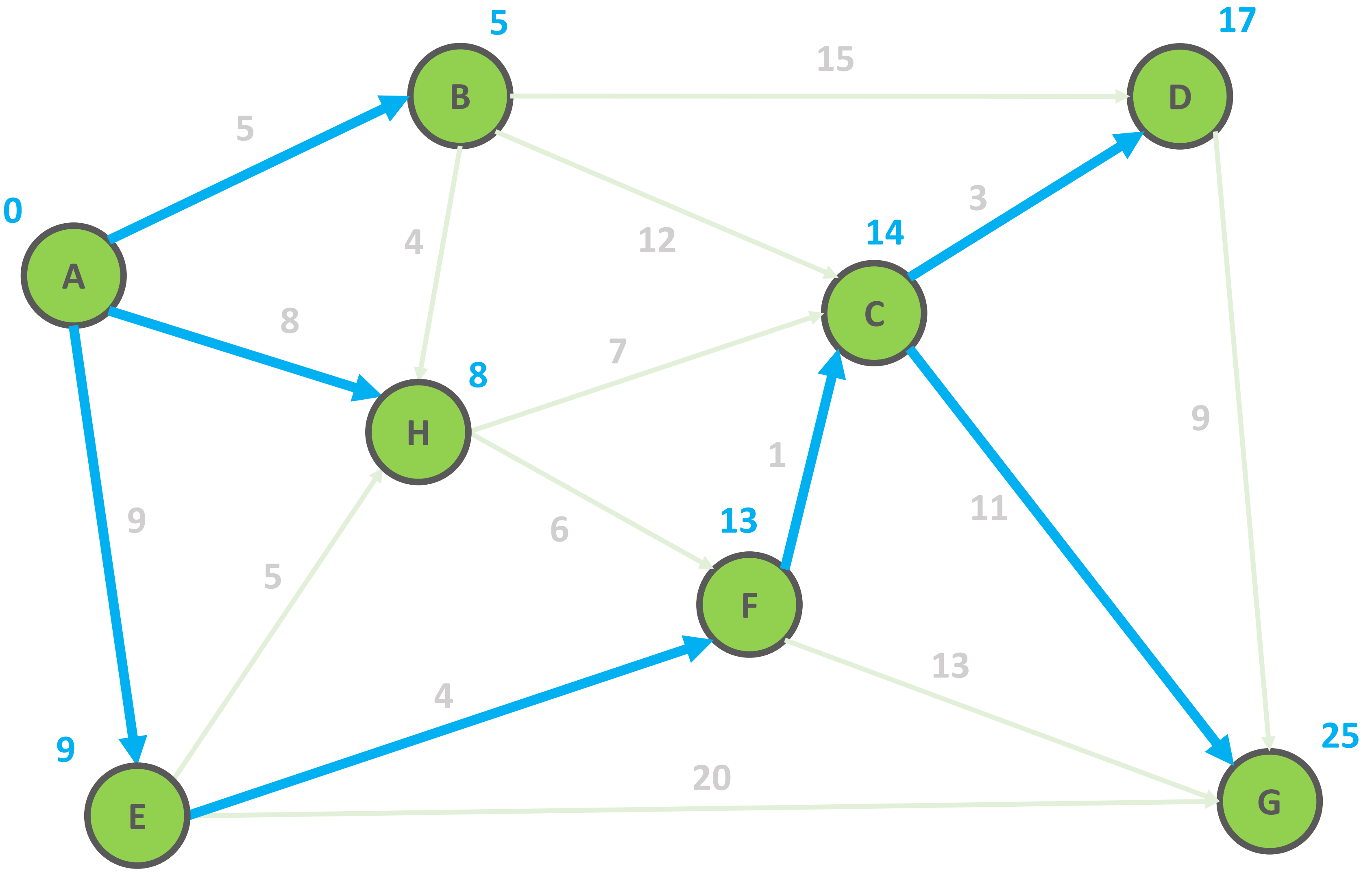
Dijkstra's Algorithm

HEAP: [ ]



Dijkstra’s Algorithm

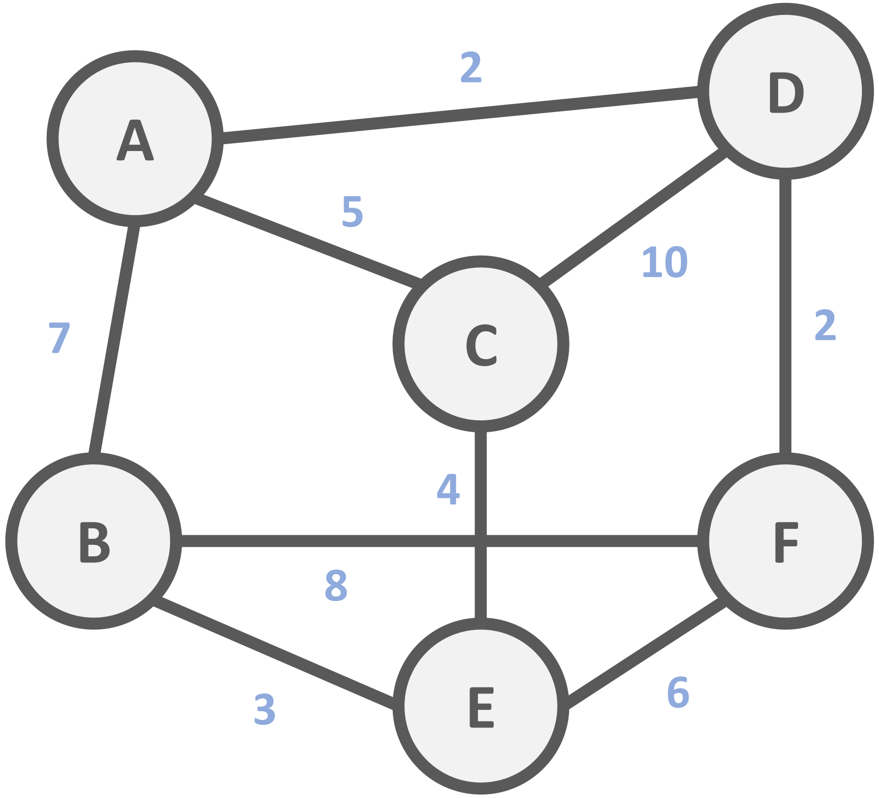
HEAP: [ ]



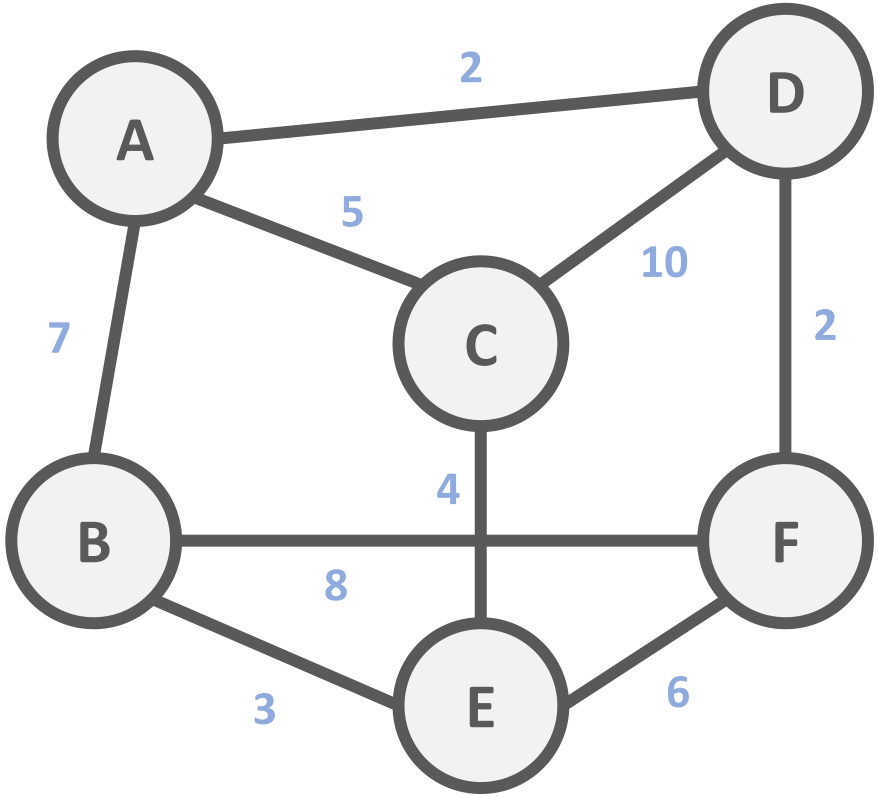
Dijkstra’s Algorithm

(Algorithms and Data Structures)

Dijkstra's Algorithm

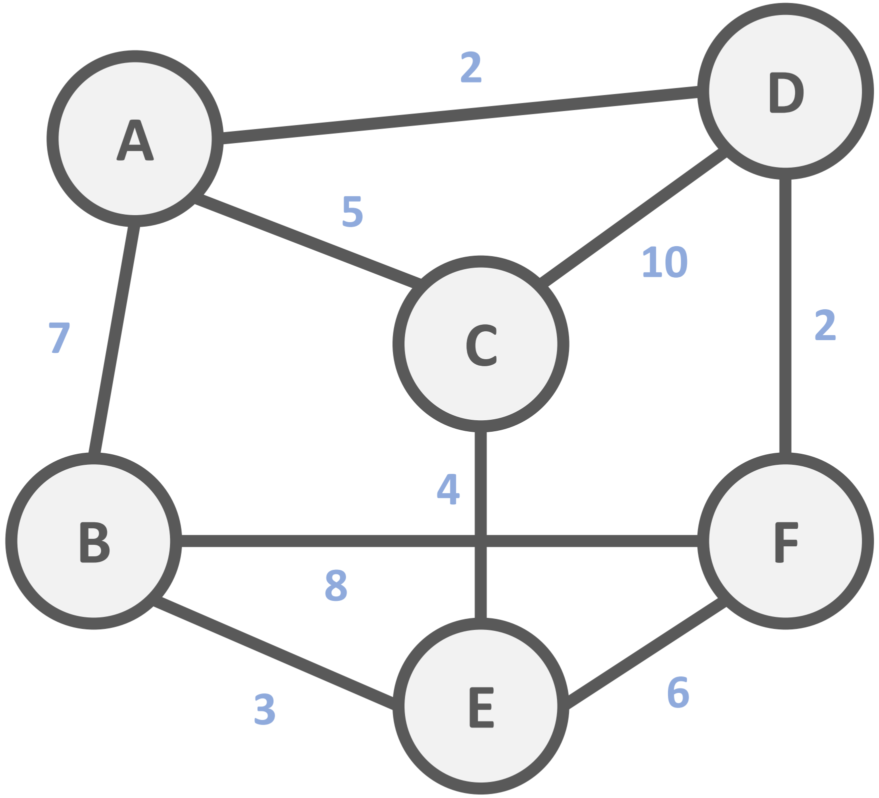


Dijkstra’s Algorithm



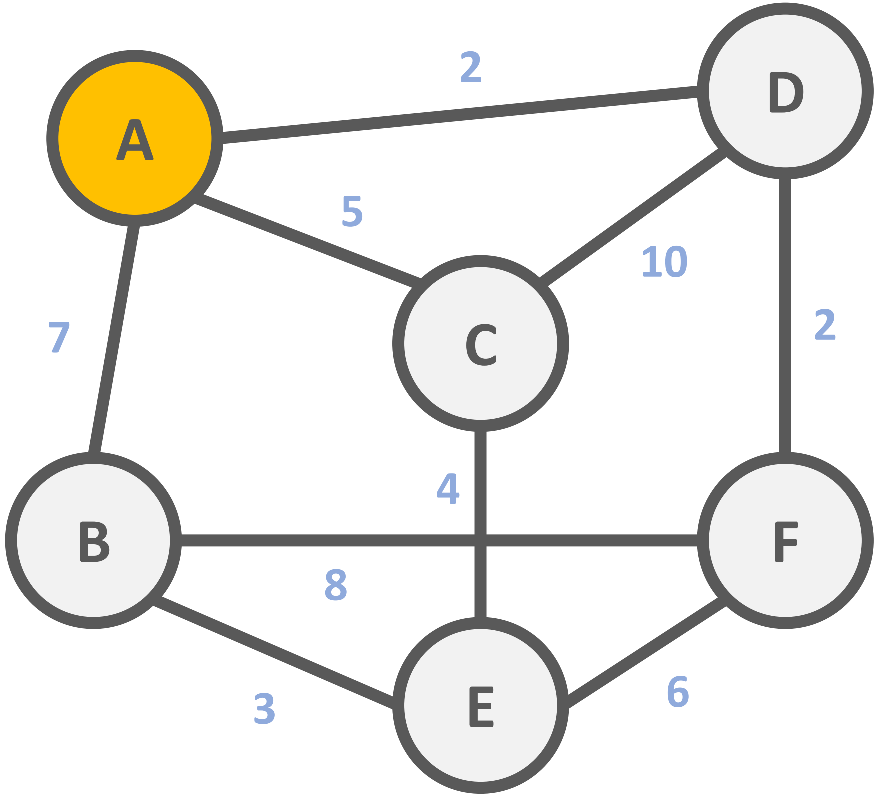
	A	B	C	D	E	F
A	0	7	5	2	0	0
B	7	0	0	0	3	0
C	5	0	0	10	4	0
D	2	0	10	0	0	2
E	0	3	4	0	0	6
F	0	8	0	2	6	0

Dijkstra's Algorithm



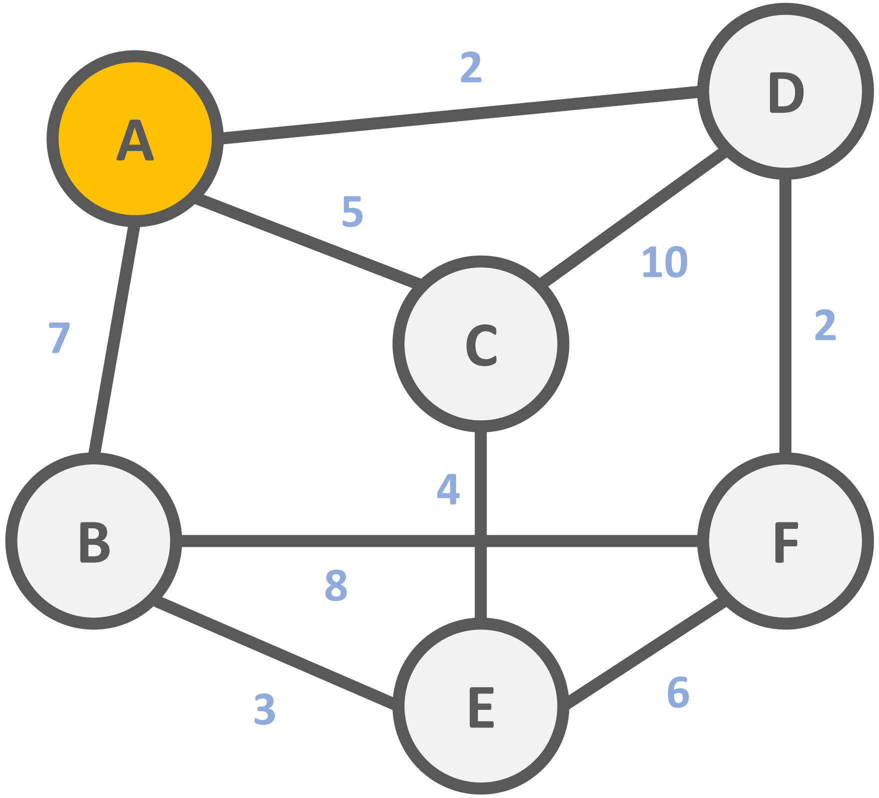
v	A	B	C	D	E	F

Dijkstra's Algorithm



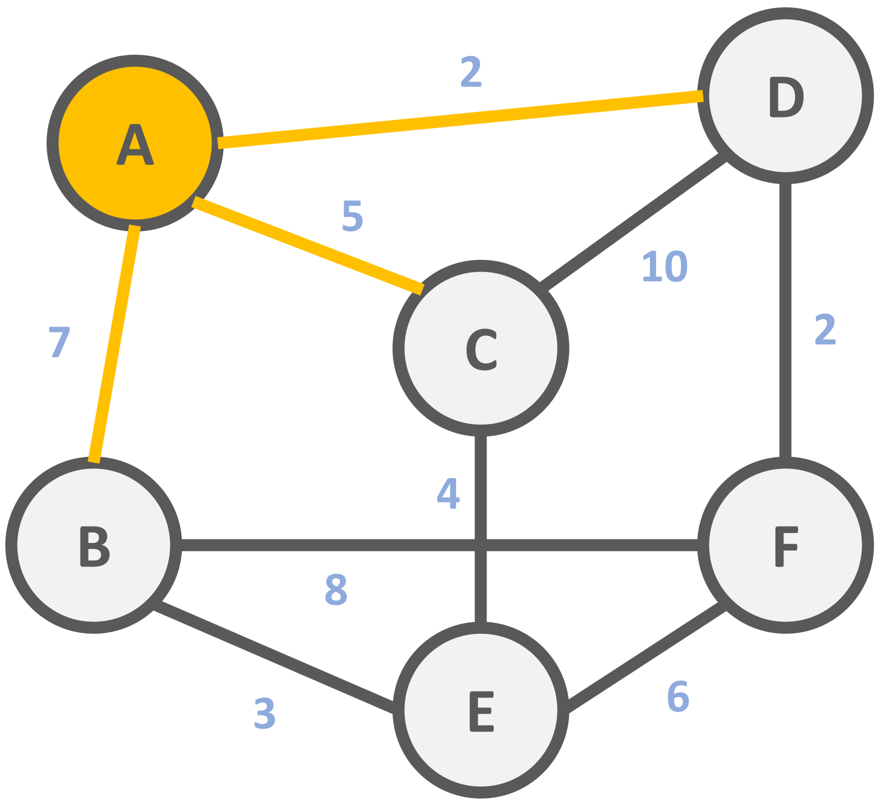
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞

Dijkstra's Algorithm



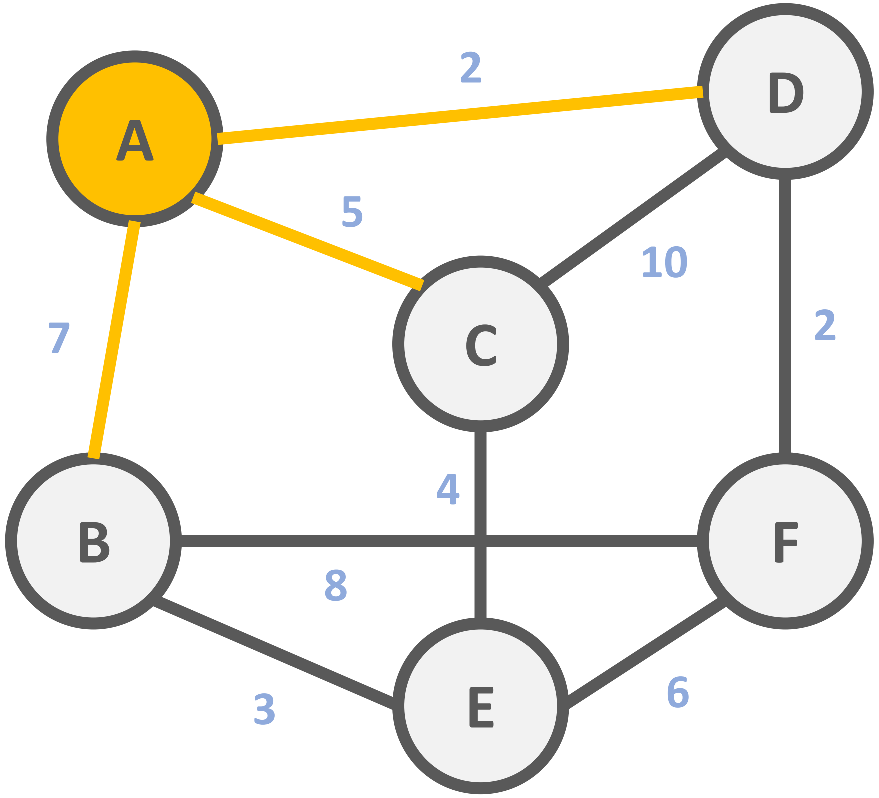
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞

Dijkstra's Algorithm



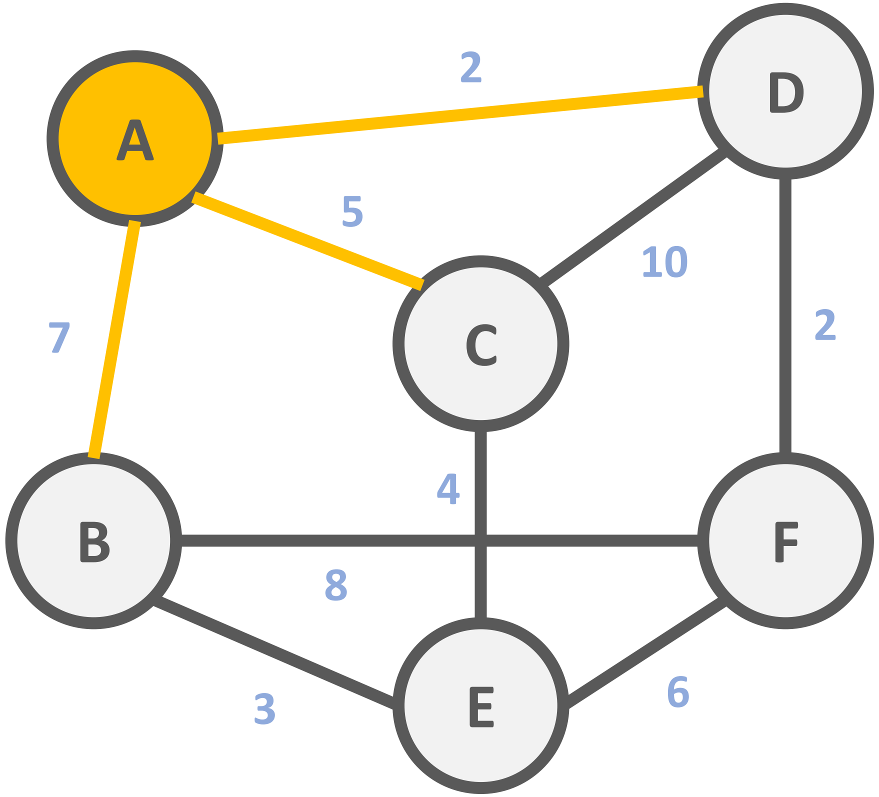
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞

Dijkstra's Algorithm



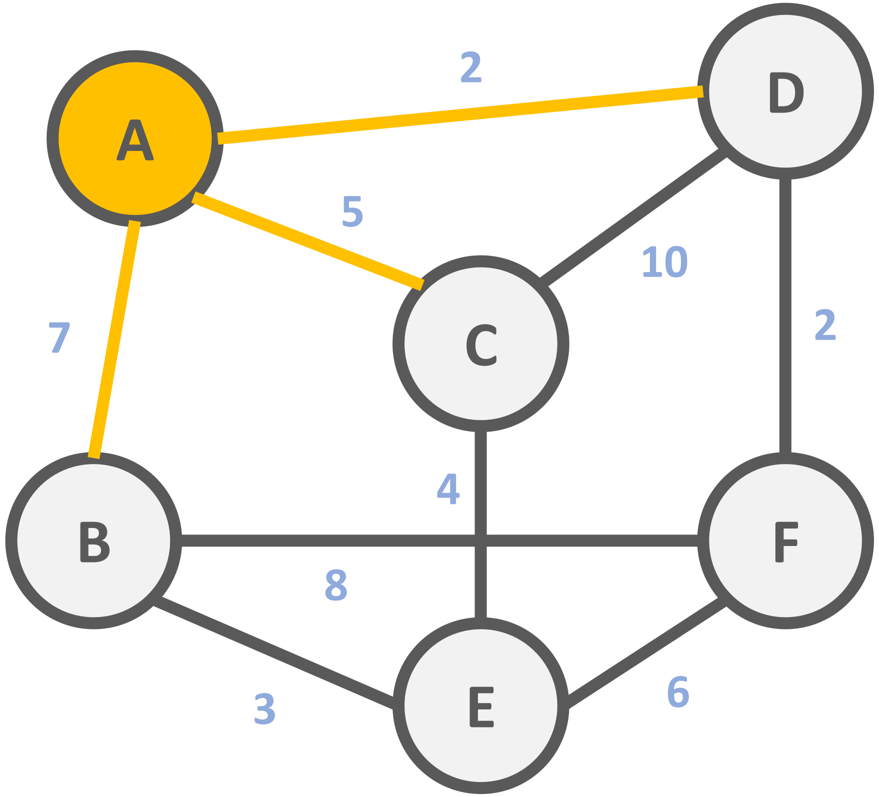
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7				

Dijkstra's Algorithm



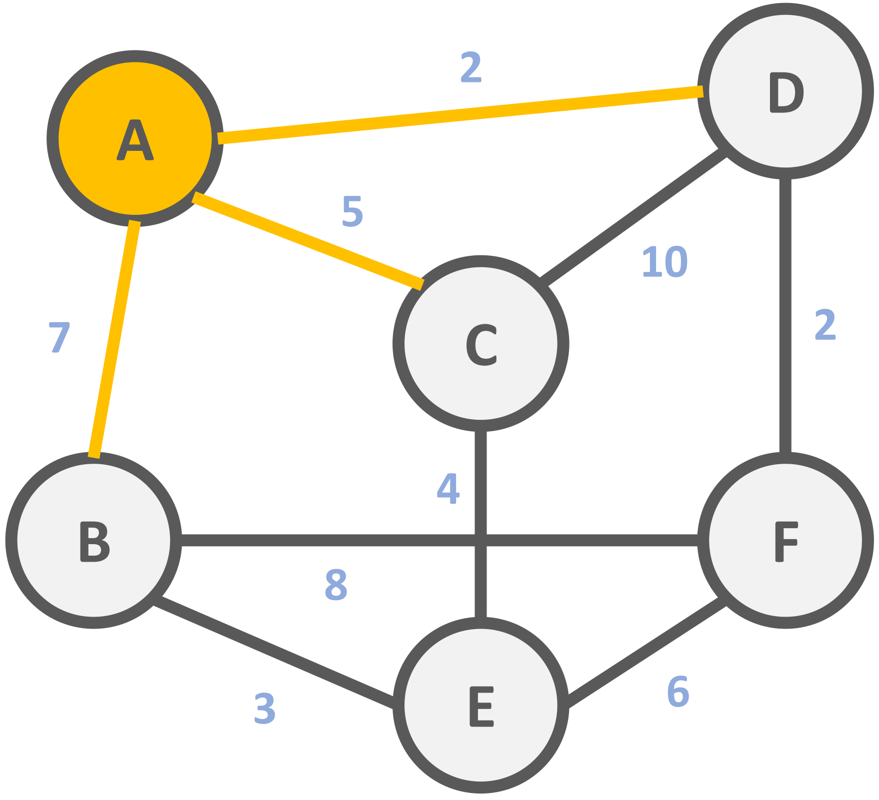
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5			

Dijkstra’s Algorithm



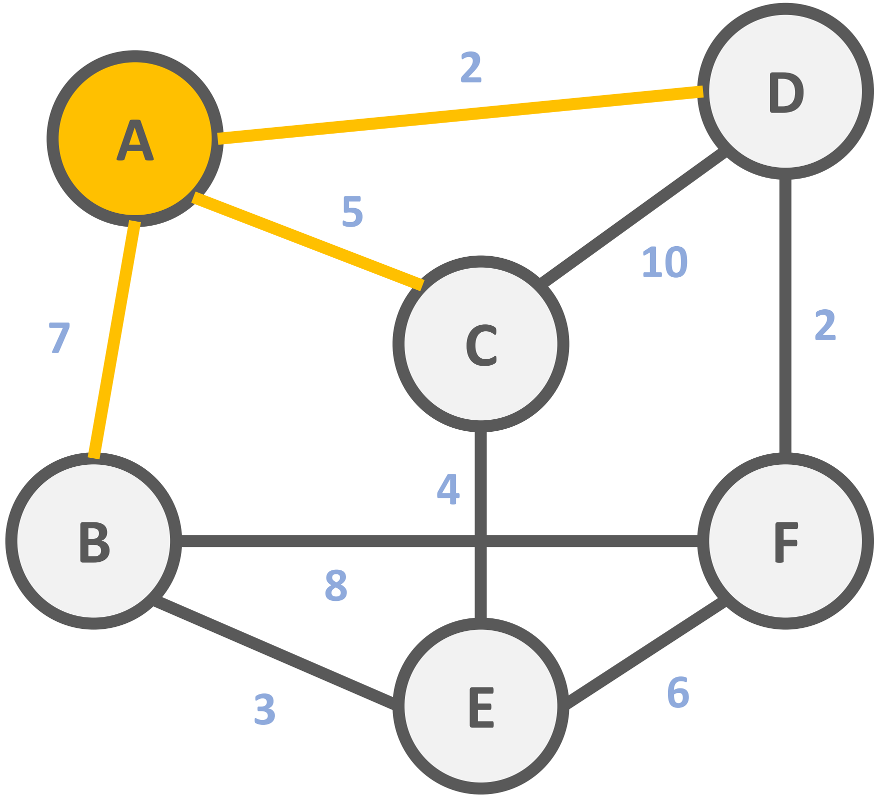
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2		

Dijkstra’s Algorithm



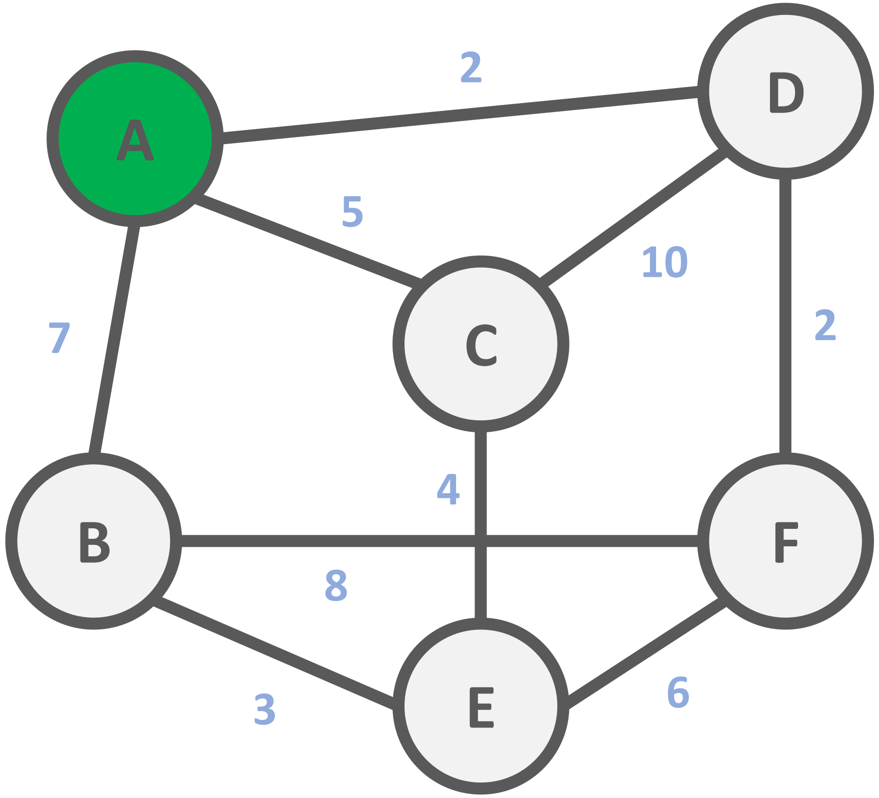
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	

Dijkstra's Algorithm



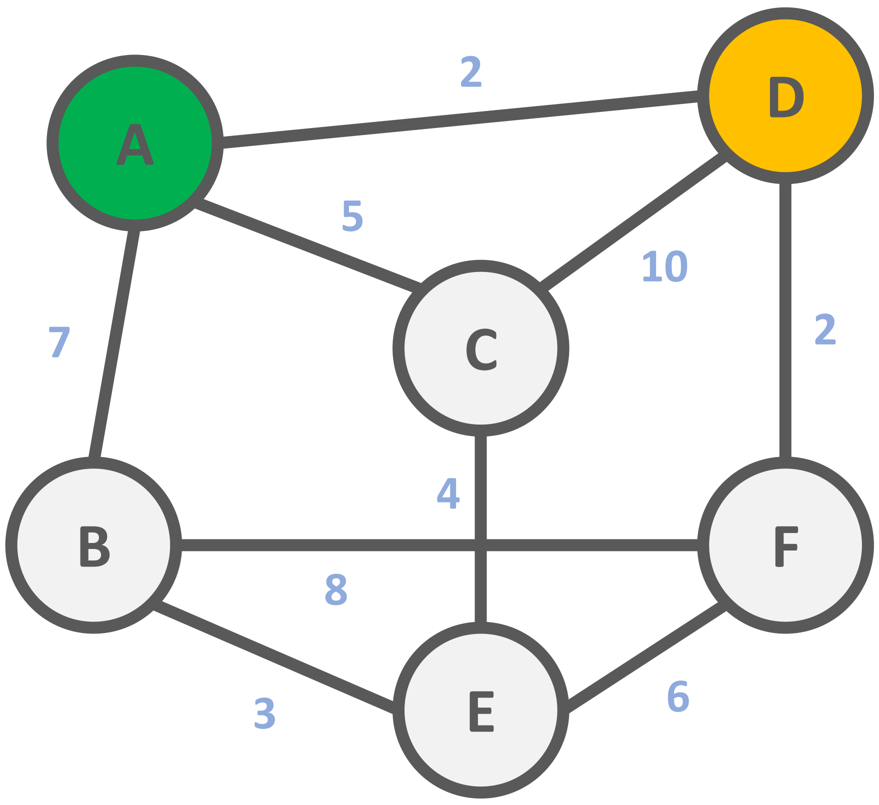
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞

Dijkstra's Algorithm



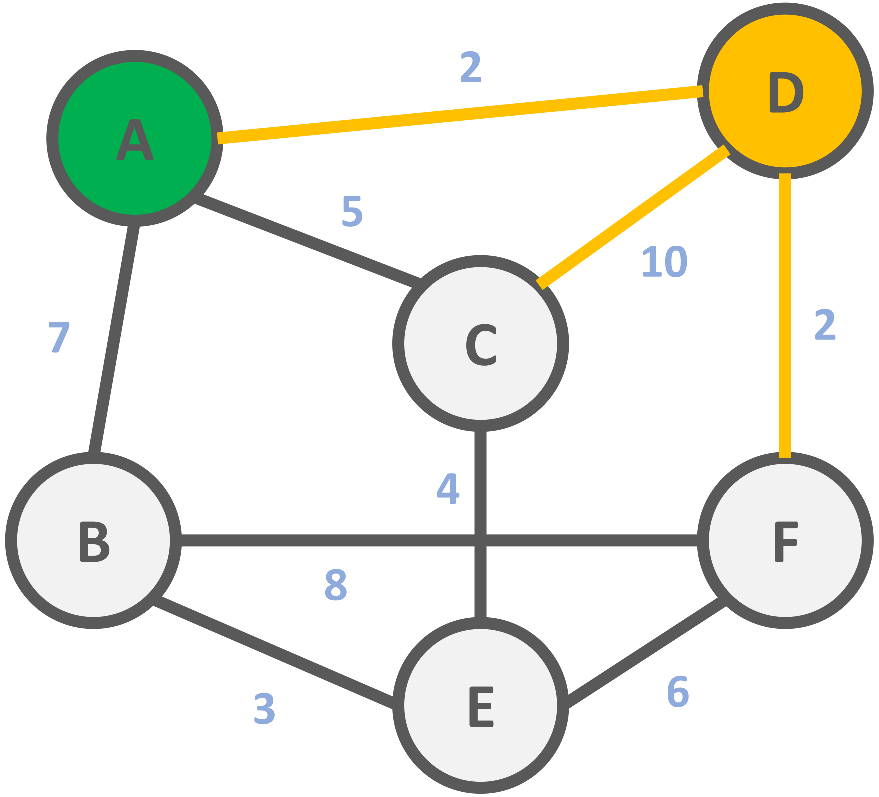
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞

Dijkstra's Algorithm



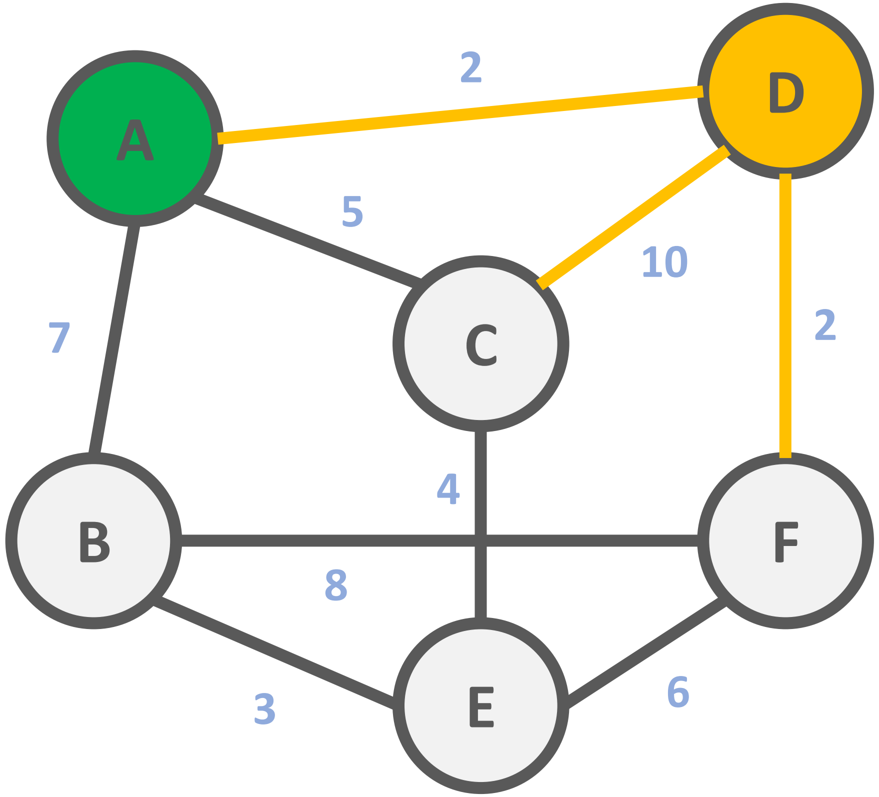
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D						

Dijkstra's Algorithm



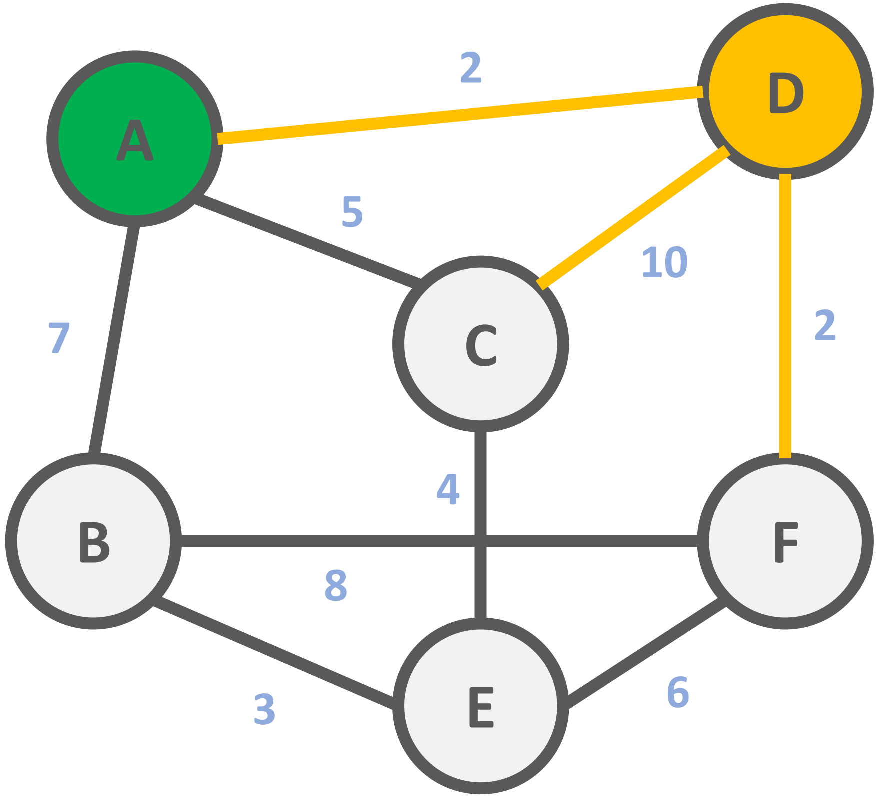
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7				

Dijkstra's Algorithm



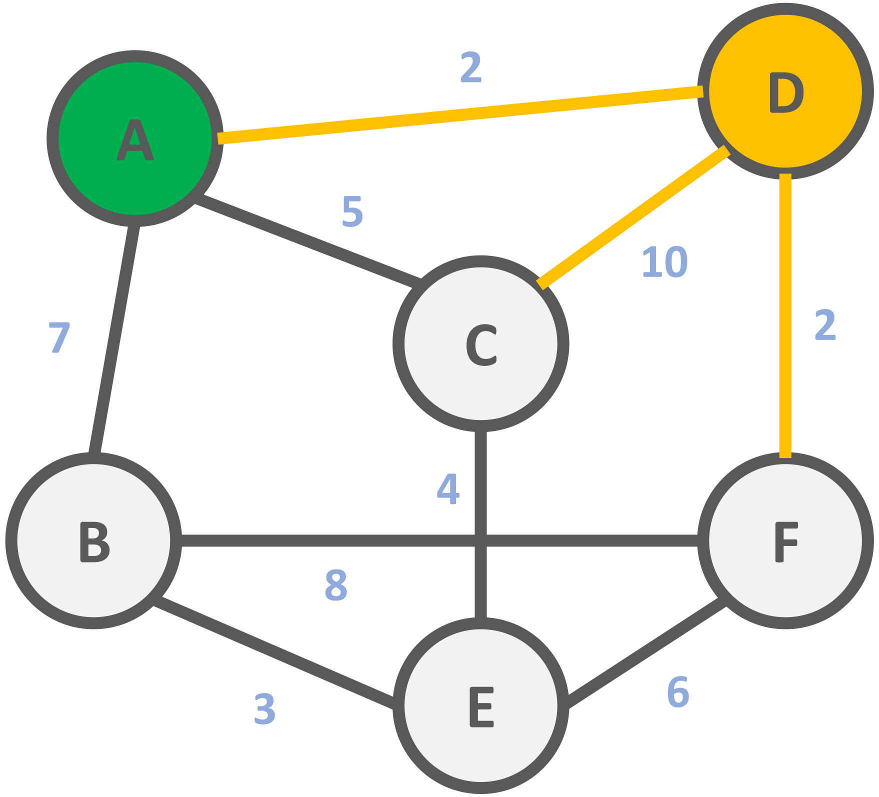
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5			

Dijkstra’s Algorithm



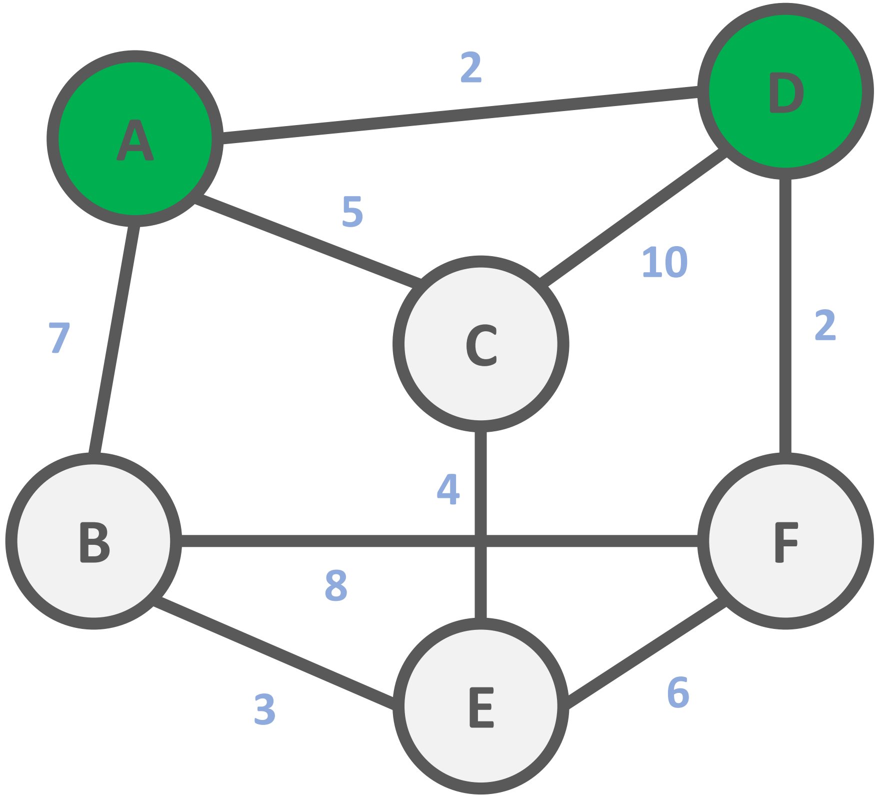
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	

Dijkstra's Algorithm



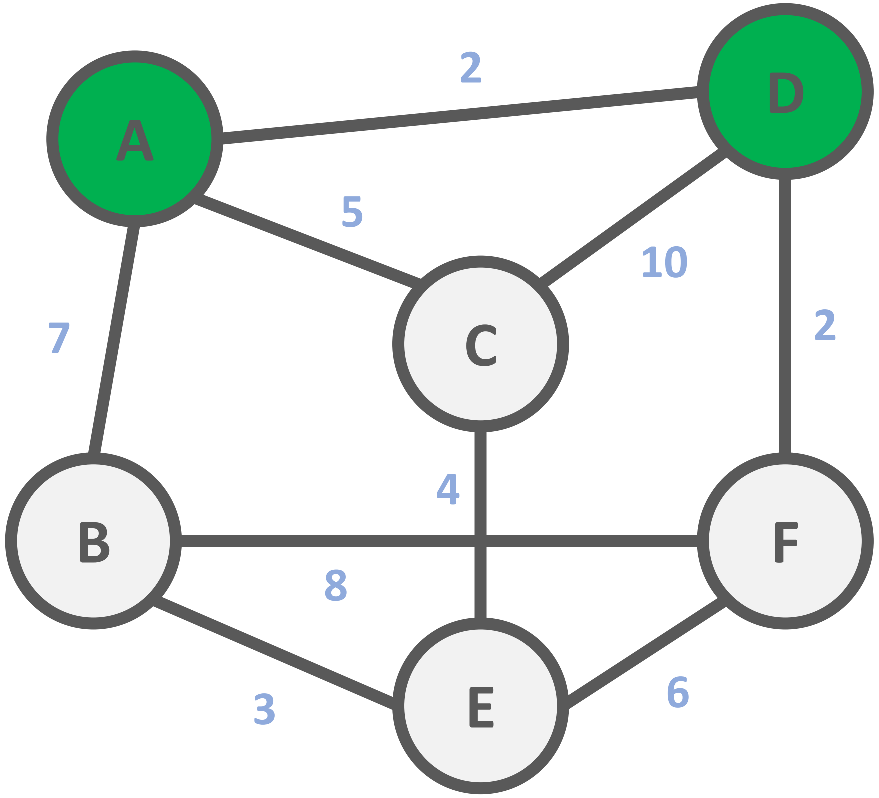
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4

Dijkstra's Algorithm



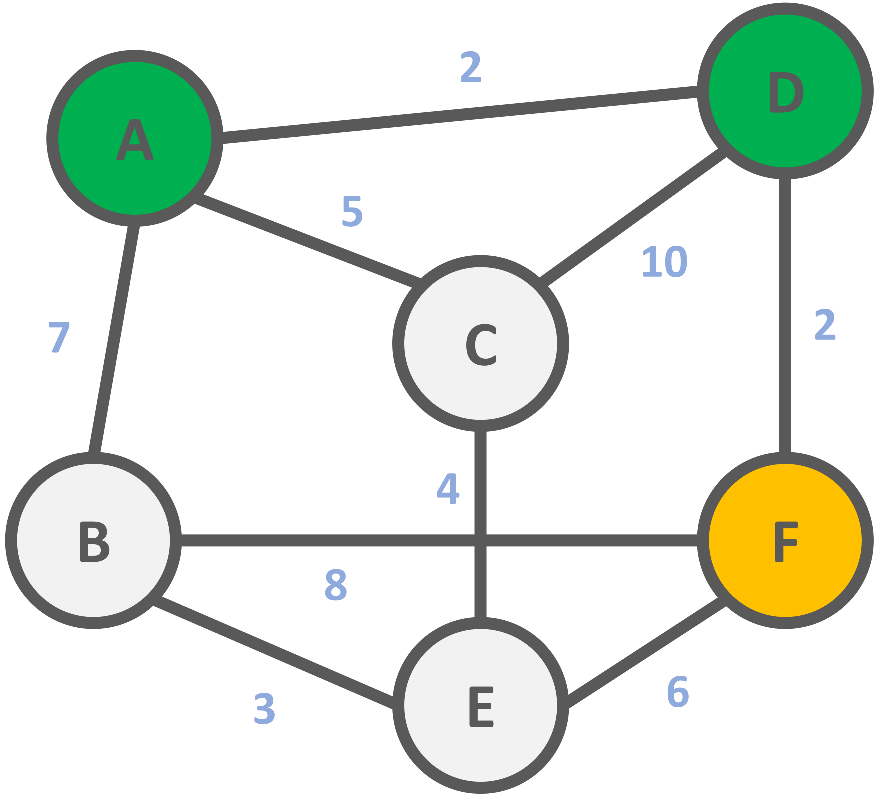
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4

Dijkstra’s Algorithm



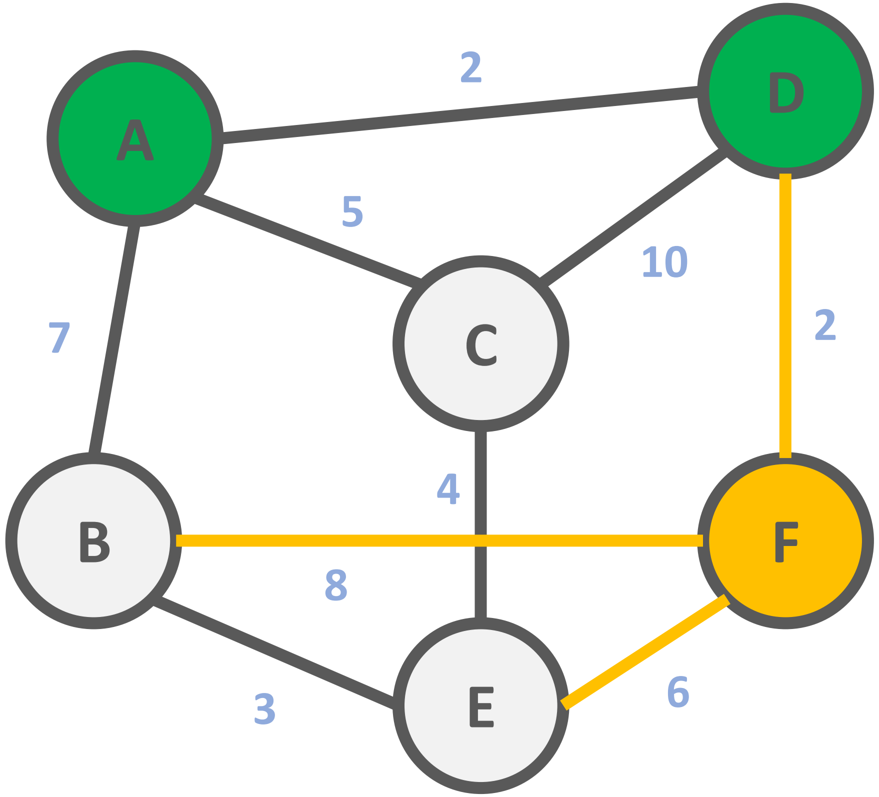
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4

Dijkstra's Algorithm



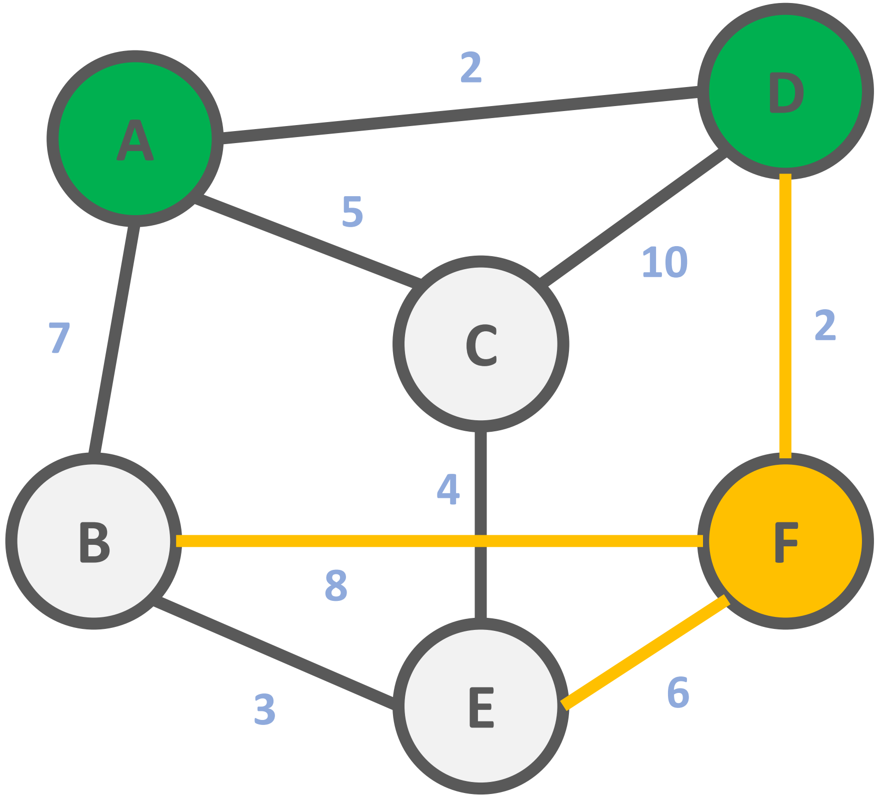
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F						

Dijkstra's Algorithm



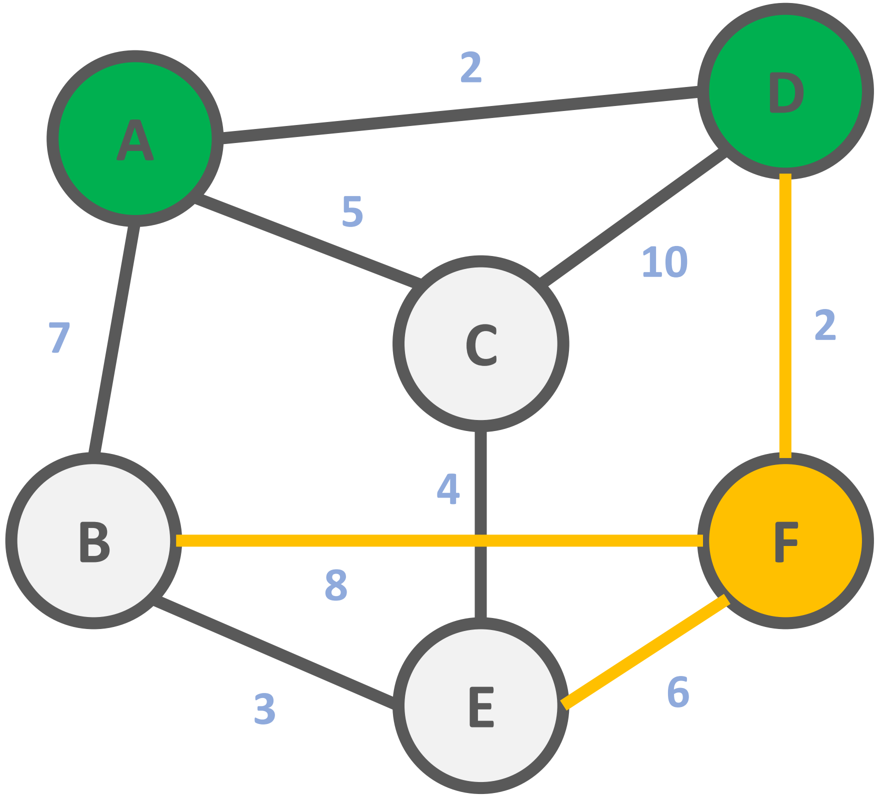
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F						

Dijkstra's Algorithm



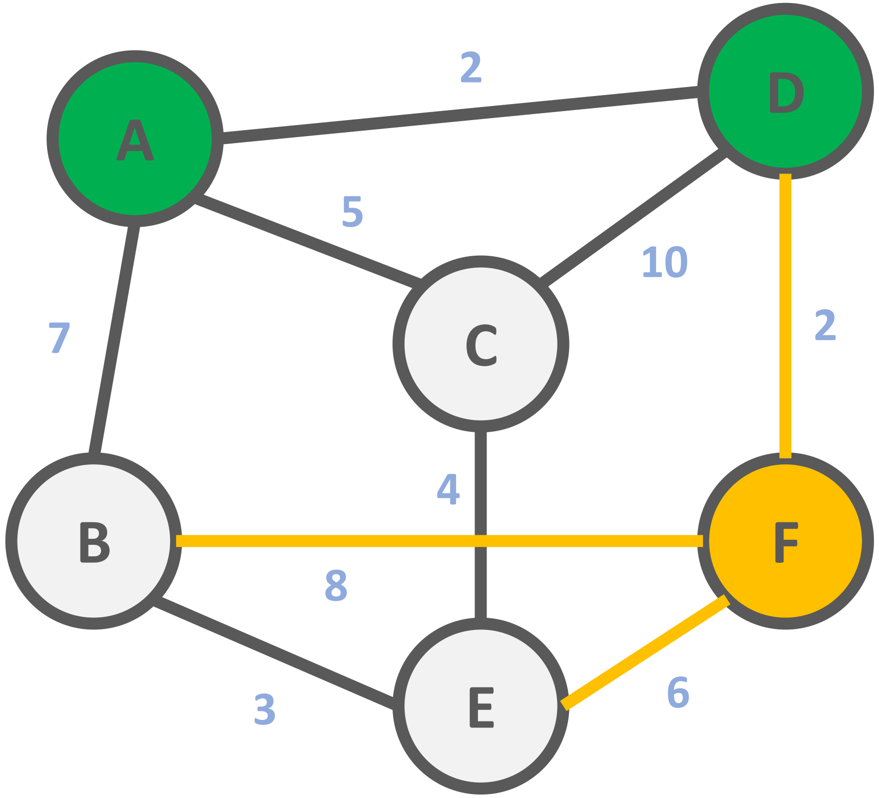
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7				

Dijkstra’s Algorithm



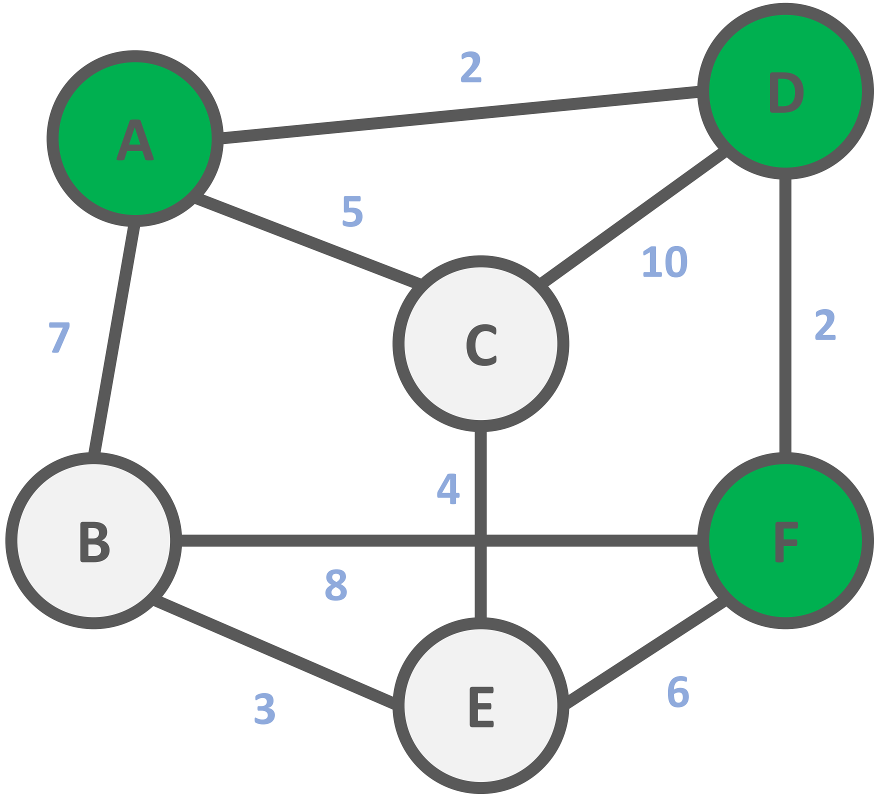
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5			

Dijkstra’s Algorithm



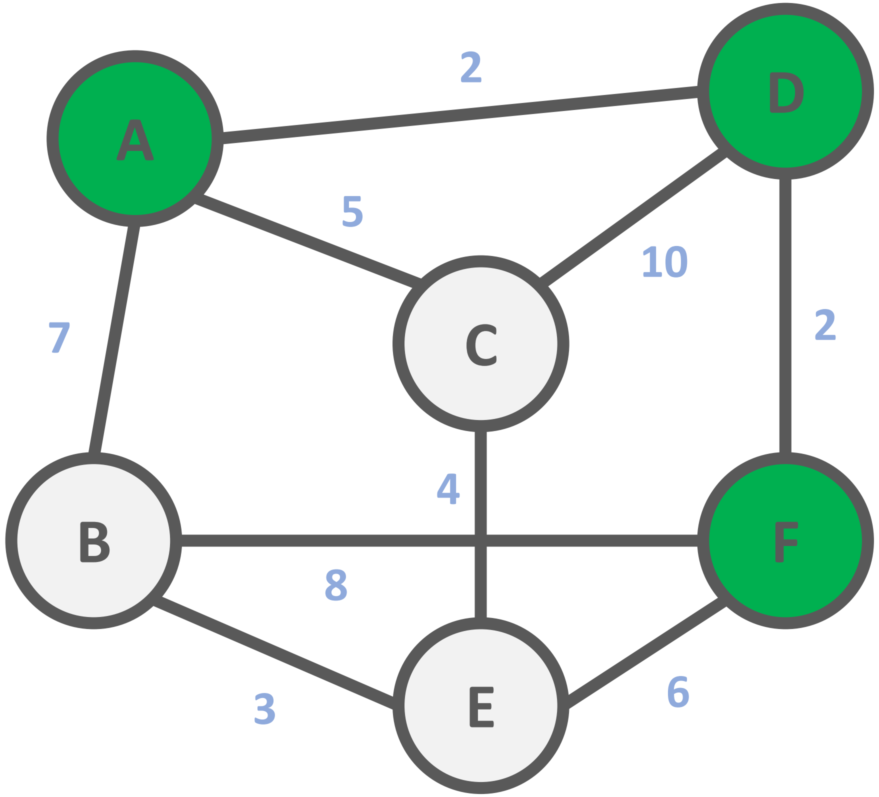
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	

Dijkstra's Algorithm



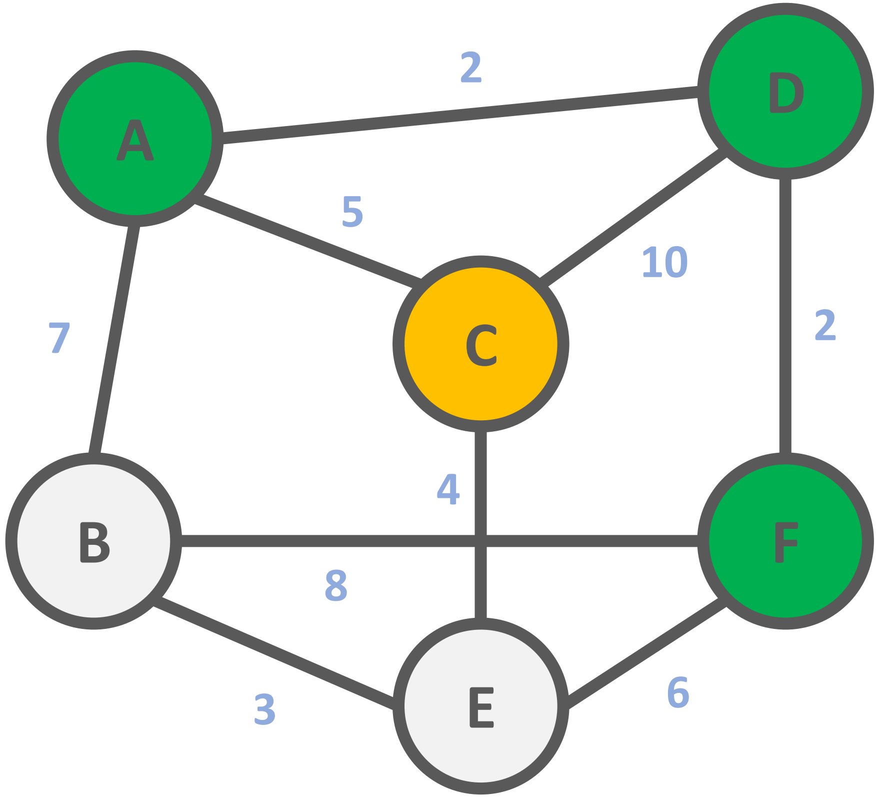
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	

Dijkstra’s Algorithm



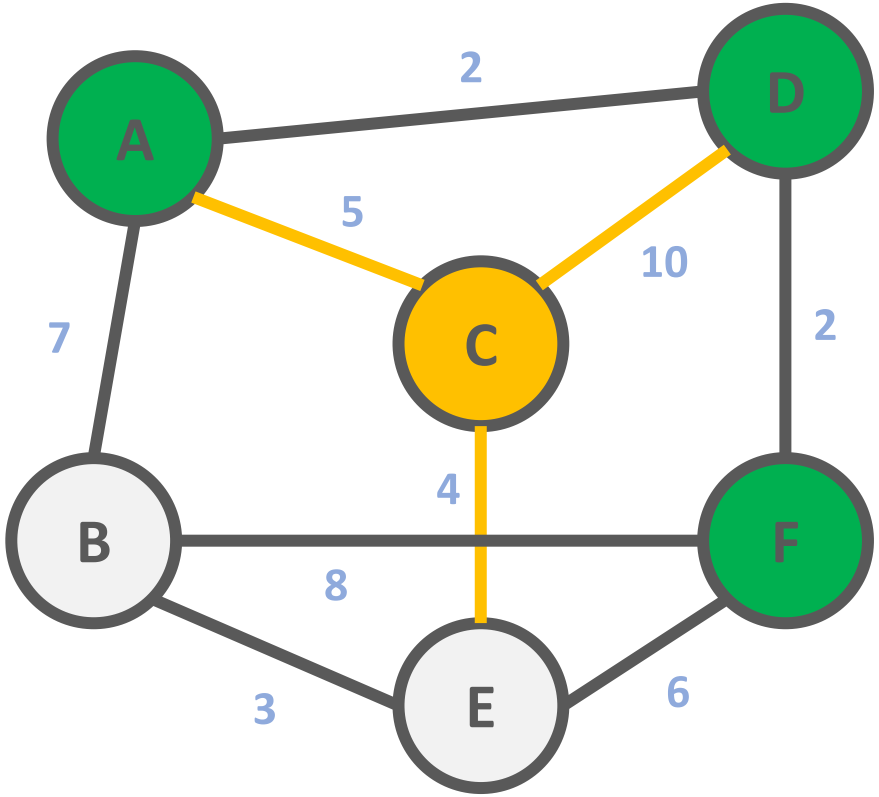
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	

Dijkstra's Algorithm



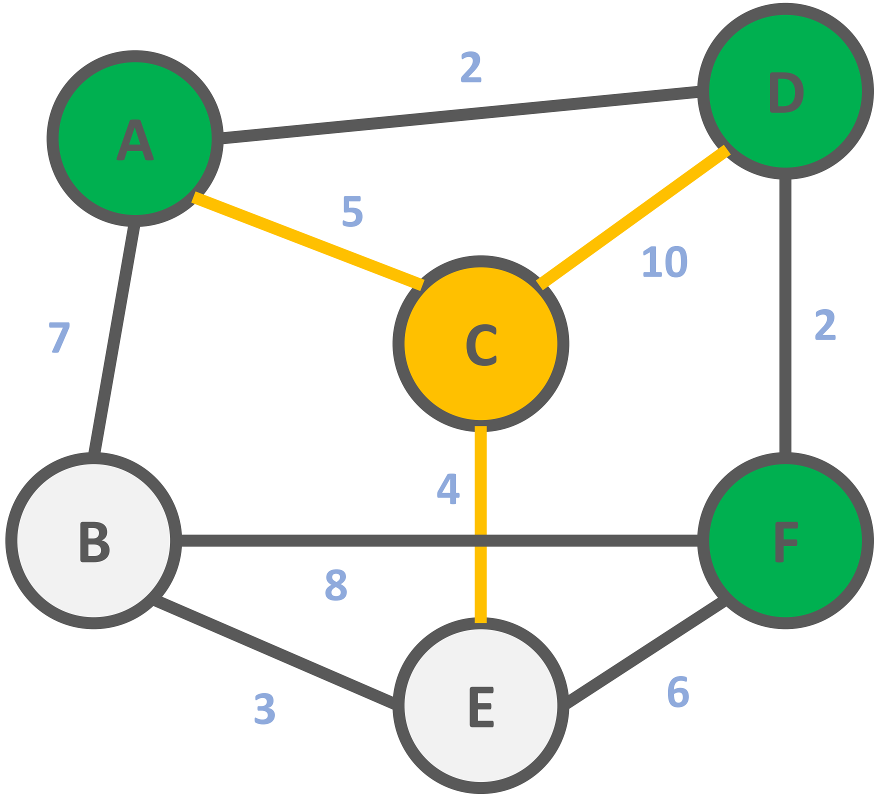
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C						

Dijkstra’s Algorithm



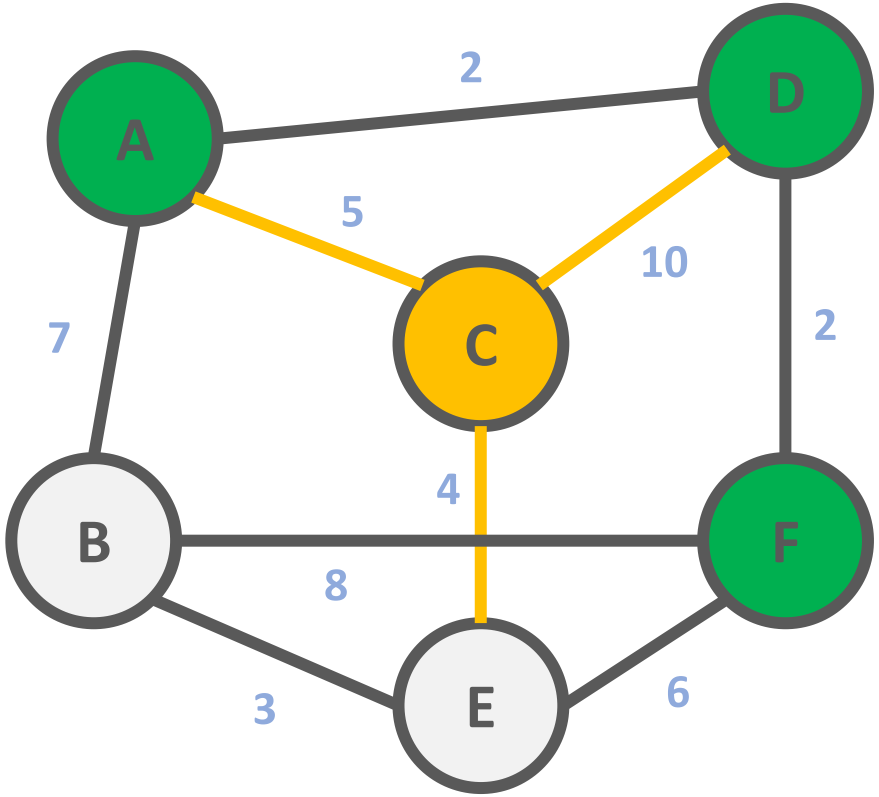
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C						

Dijkstra's Algorithm



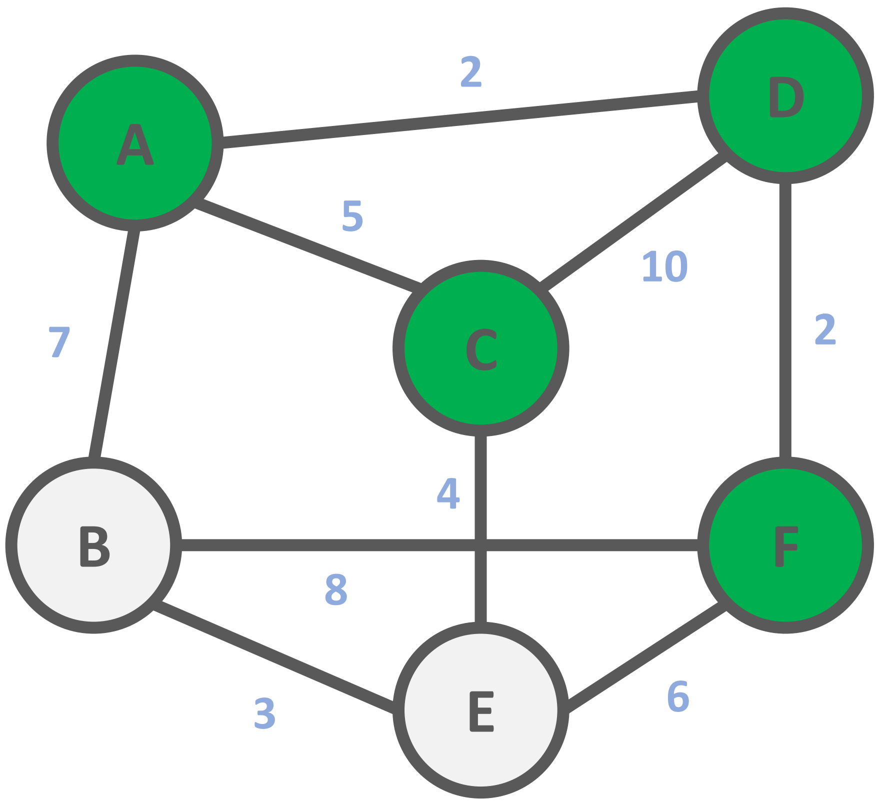
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7				

Dijkstra's Algorithm



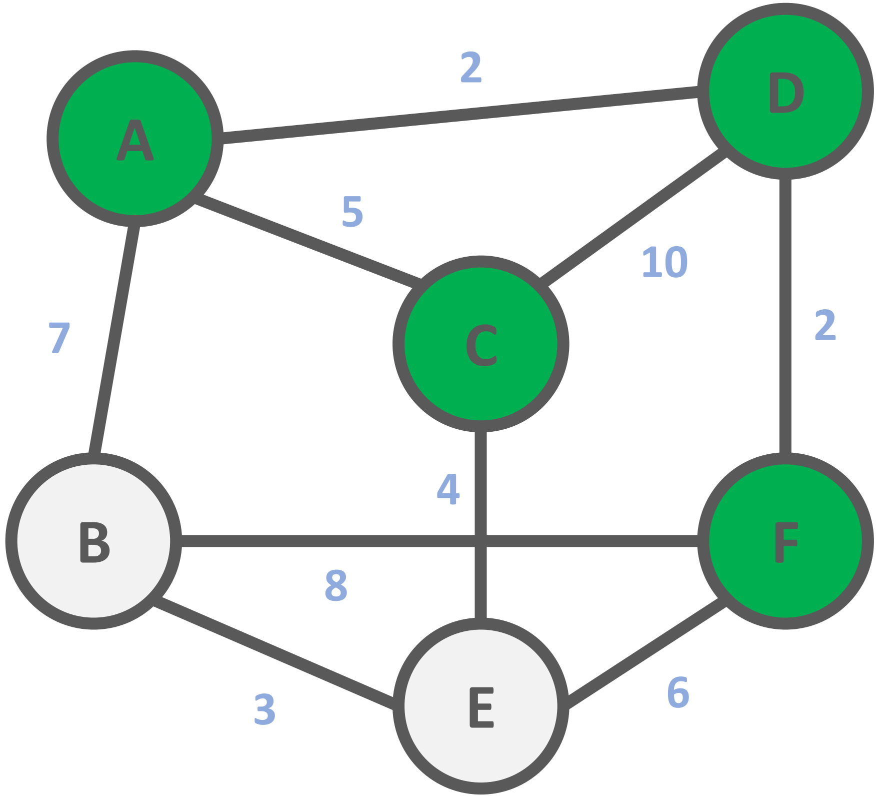
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	

Dijkstra’s Algorithm



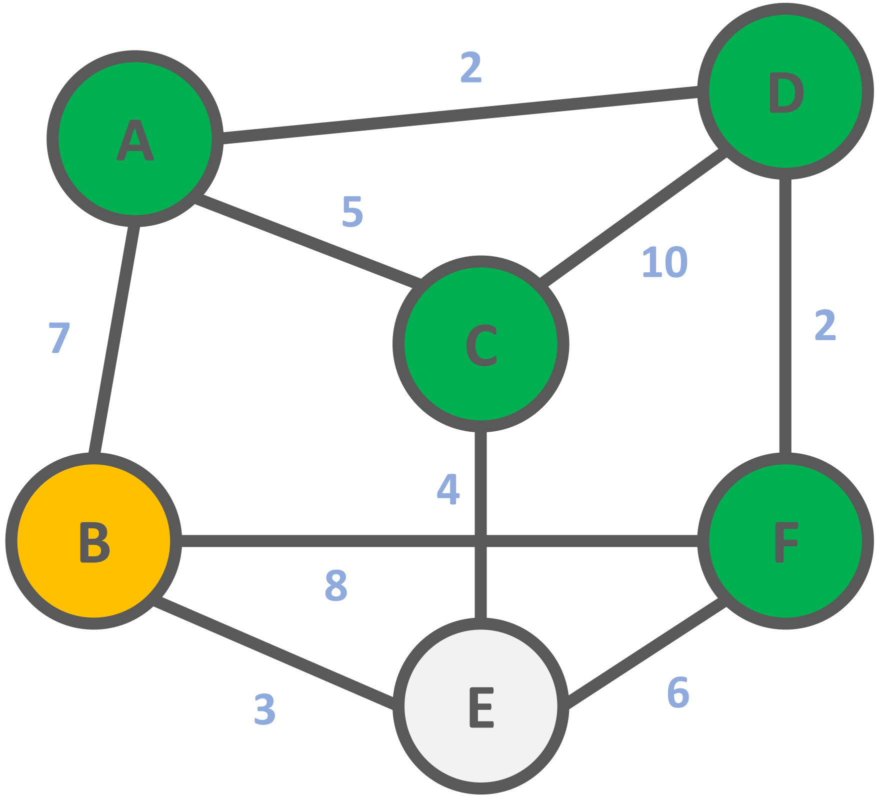
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	

Dijkstra's Algorithm



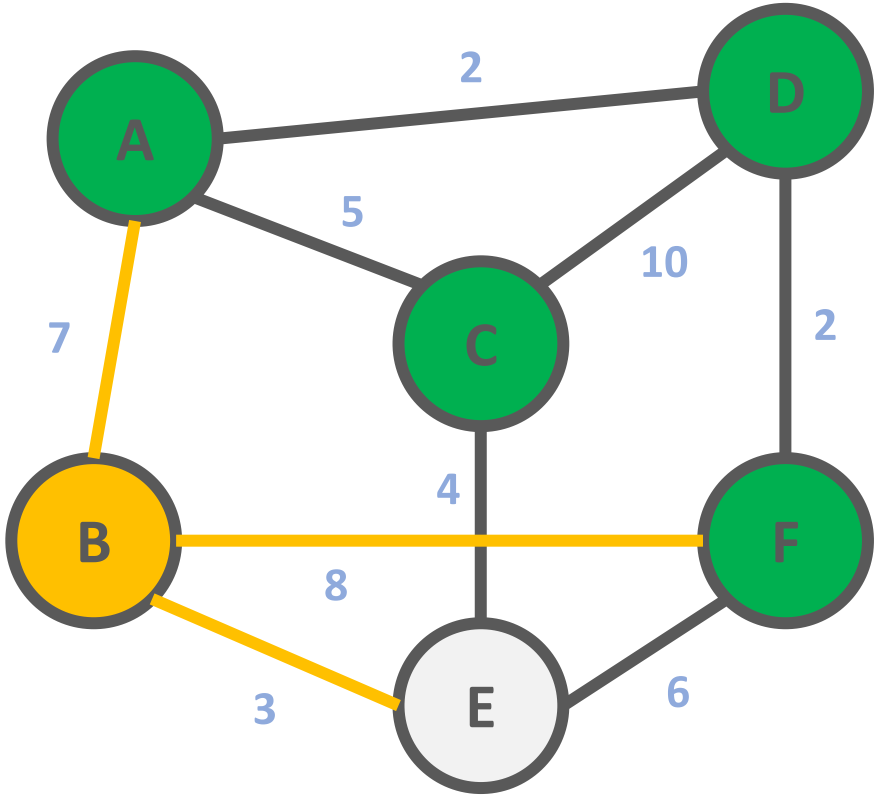
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	

Dijkstra's Algorithm



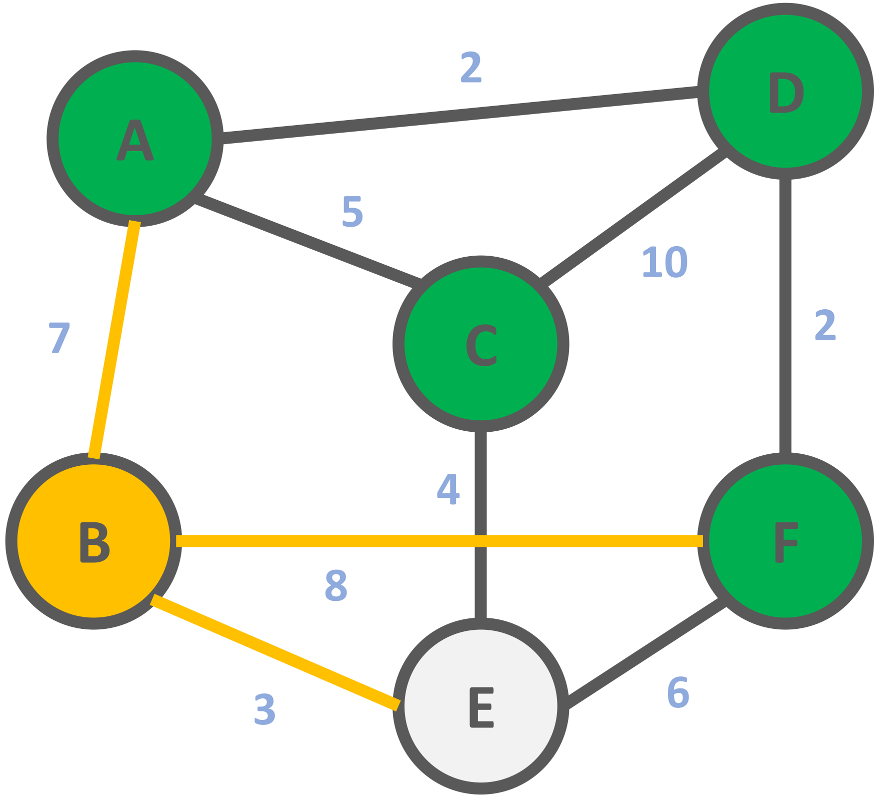
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	

Dijkstra's Algorithm



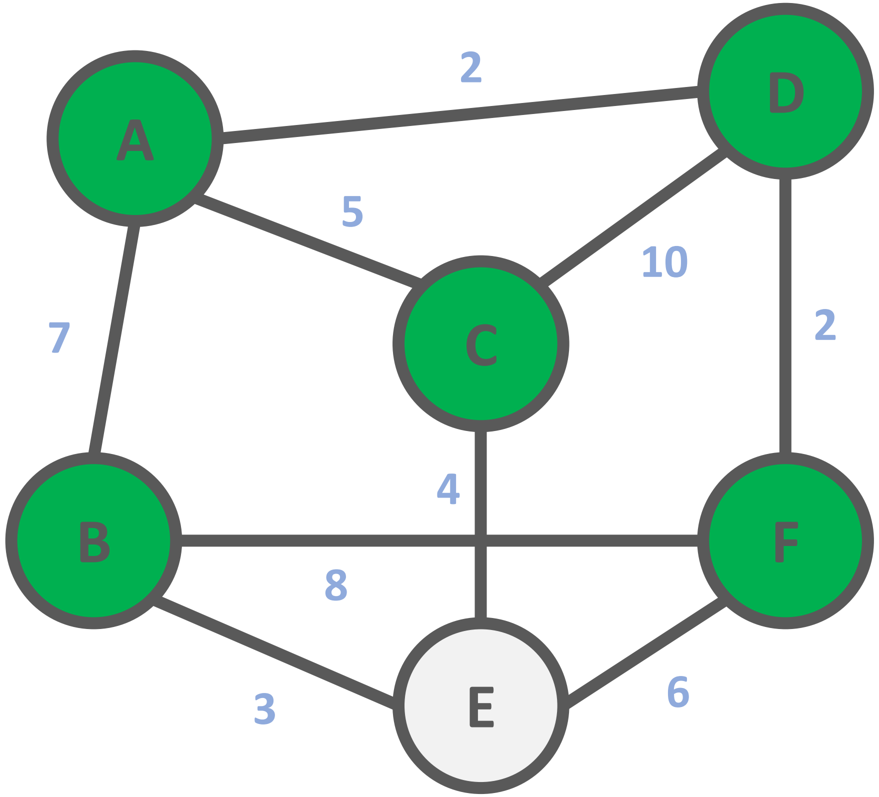
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B						

Dijkstra's Algorithm



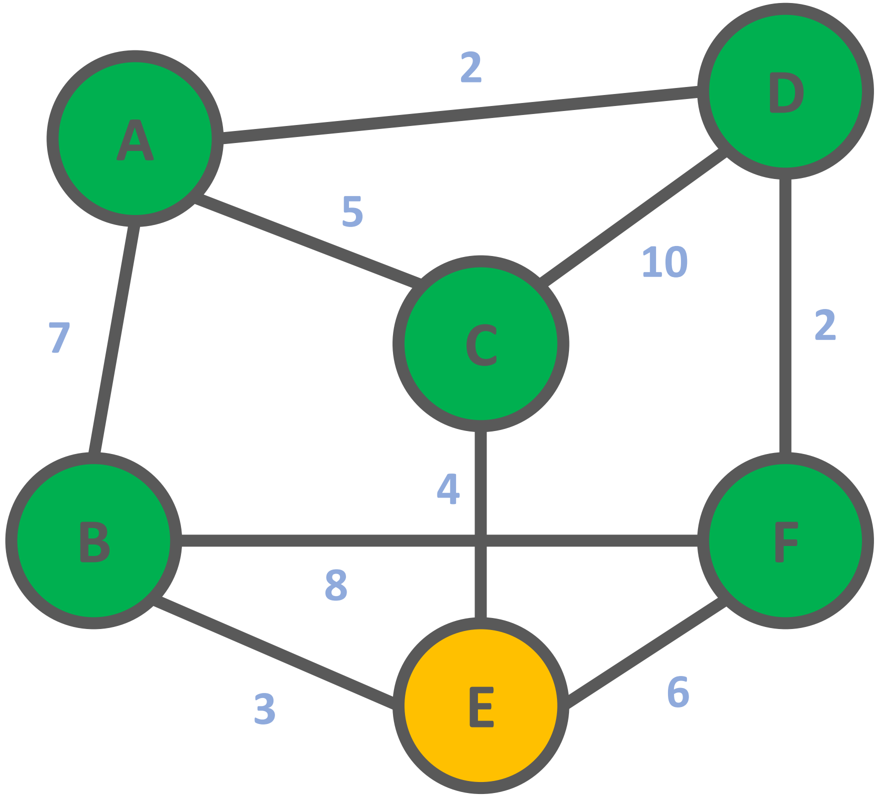
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	

Dijkstra's Algorithm



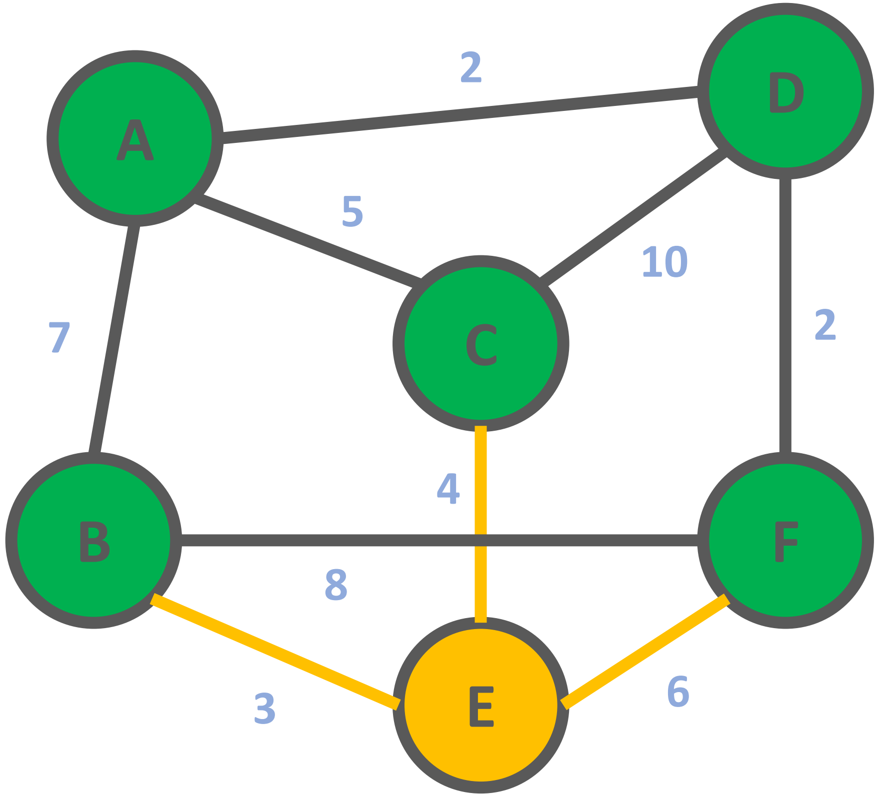
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	

Dijkstra's Algorithm



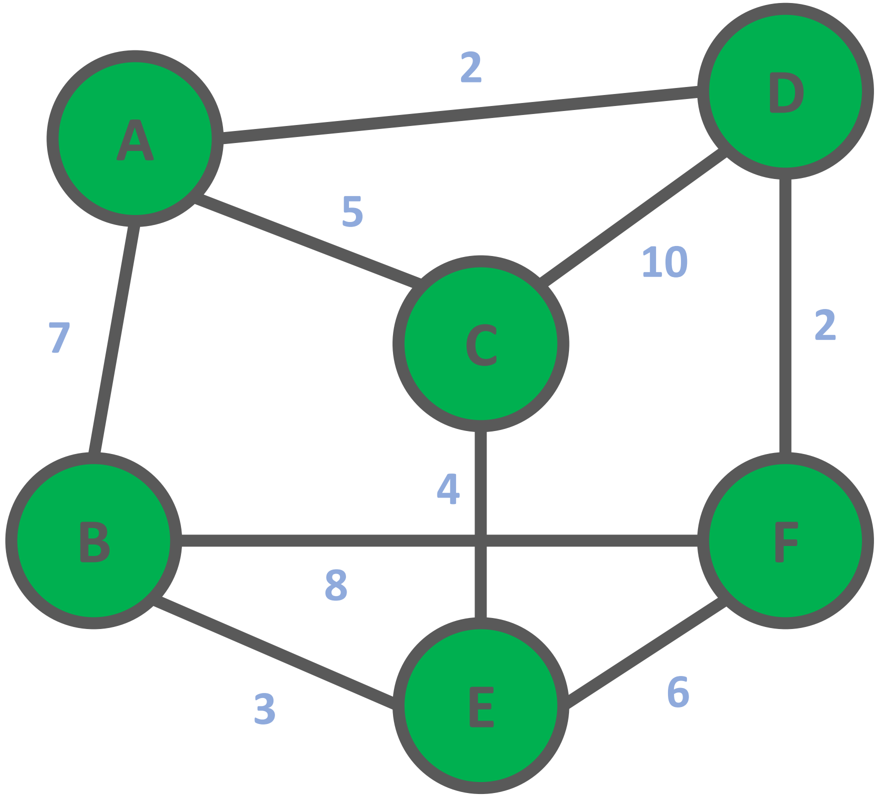
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	

Dijkstra's Algorithm



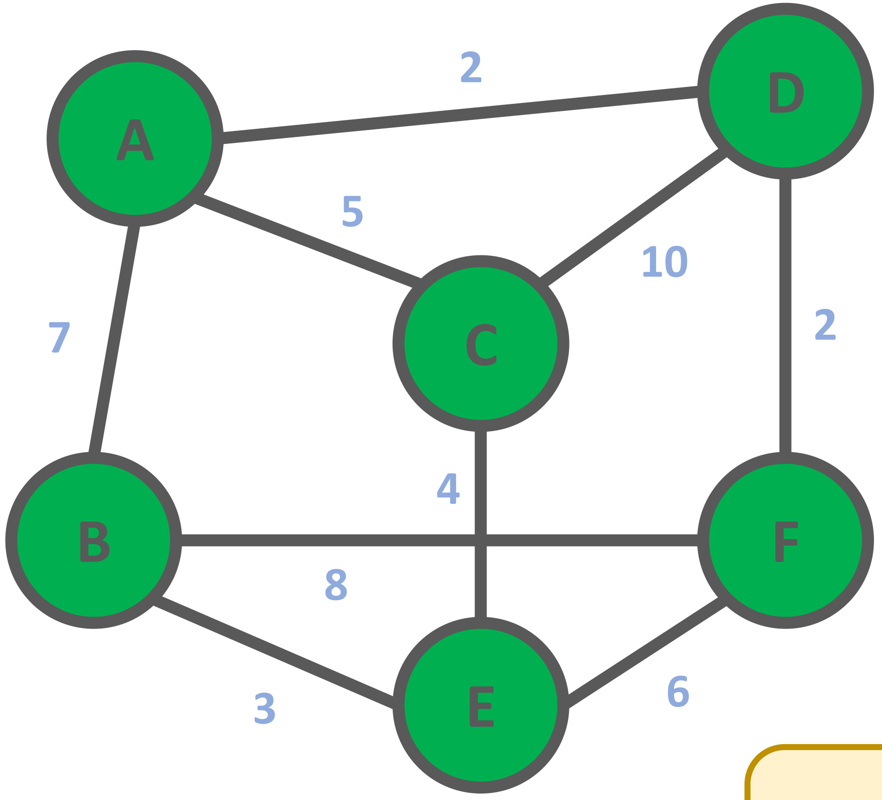
v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	
E						

Dijkstra's Algorithm



v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	
E						

Dijkstra's Algorithm



v	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
A		7	5	2	∞	∞
D		7	5		∞	4
F		7	5		10	
C		7			9	
B					9	
E						

*Dijkstra's algorithm with **adjacency matrix** representation has **O(V²)** quadratic running time*

Shortest Path Algorithms Application

(Algorithms and Data Structures)

Shortest Path Algorithms Application

1.) GPS and navigation

Maybe navigation is the most crucial application of the **shortest path problem** and Dijkstra’s algorithm

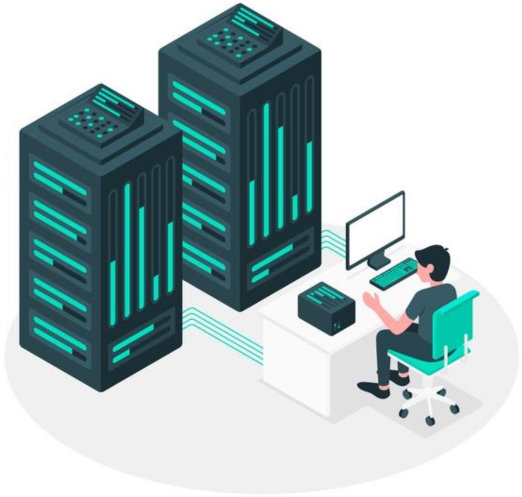
- Google Maps
- Apple Maps
- Waze

Shortest Path Algorithms Application

2.) RIP – routing information protocol

Shortest path approaches are important in **computer networking** as well such as with the routing information protocol in the application layer

- data is splitted into packages and these packages are sent one by one – with the **UDP** protocol
- the packages follow the shortest path

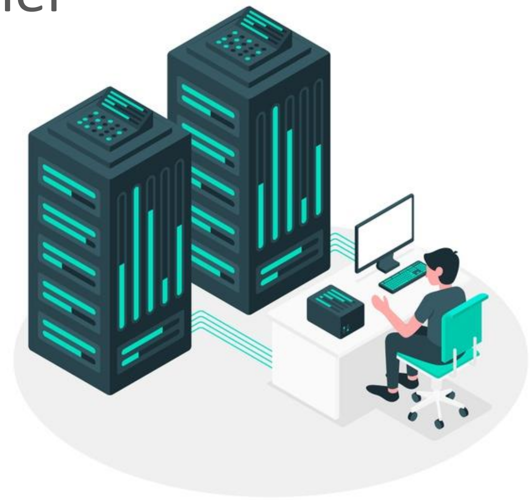


Shortest Path Algorithms Application

2.) RIP – routing information protocol

Shortest path approaches are important in **computer networking** as well such as with the routing information protocol in the application layer

- each node calculates the distances between itself and all other nodes and stores this information as a table
- each node sends its table to all adjacent nodes
- when a node receives distance tables from its neighbors it calculates the shortest routes to all other nodes and updates its own table to reflect any changes



Shortest Path Algorithms Application

3.) Avidan-Shamir method

When we want to **shrink an image** for example in the browser or on a smartphone without distortion

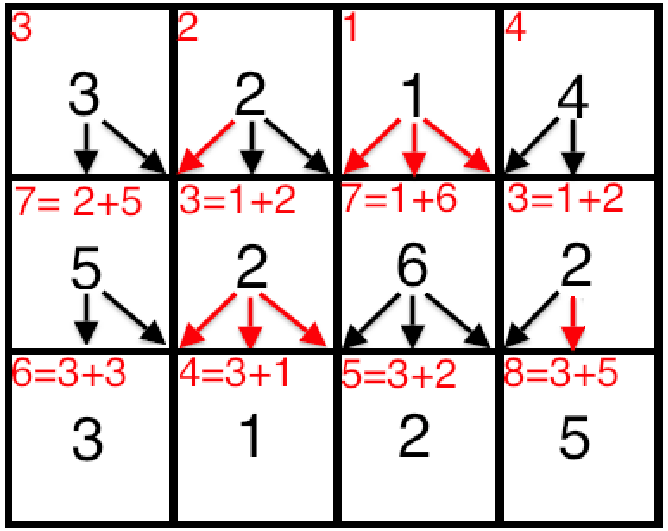
- it is important to make sure the image will not deform
- we have to eliminate the least significant bit strings
- construct a so-called **energy function** – and remove the connected string of pixels containing the least energy
- **Photoshop** and **GIMP** use this method

Shortest Path Algorithms Application

3.) Avidan-Shamir method

When we want to **shrink an image** for example in the browser or on a smartphone without distortion

- we build a huge graph: vertices are the pixels and the edges are pointing from every vertex to its downward **3** neighbours
- the **energy function** determines the edge weights
- we can use topological order shortest path to find the string of pixels to be removed



Critical Path Method (CPM)

(Algorithms and Data Structures)

Longest Path Problem

- we have discussed how to find the shortest path in a **G(V,E)** graph from **s** source vertex to a **d** destionation vertex
- but what if we are looking for the **longest path**?
- it is an **NP-hard** problem with no known polynomial running time algorithm to solve
- but if the **G(V,E)** graph is a **directed acyclic graph** (DAG) then we can solve the problem in linear running time
- SCHEDULING ALGORITHMS RELY HEAVILY ON LONGEST PATHS

Longest Path Problem

- is it possible to tranform longest path problem into a shortest path problem?
- we just have to **negate the edge weights** – multiply them by **-1** and run the standard shortest path algorithms
- because of the negative edge weights we have to use **Bellman-Ford** algorithm for finding the shortest path
- it can solve **the parallel job scheduling** problem
- given a set of **V** jobs with $\mathbf{d_i}$ durations and precedence constraints: schedule the jobs - by finding a start time to each - so as to achive the minimum completion time while respecting the constraints

Critical Path Method (CPM)

- the **critical path method** was first used between **1940** and **1943** in the Manhattan project
- the first time CPM was used for major skyscraper development was in **1966** while constructing the world trade center
- we want an algorithm for scheduling a set of project activities so that the **total running time** will be as **minimal** as possible

Critical Path Method (CPM)

CPM ALGORITHM NEEDS:

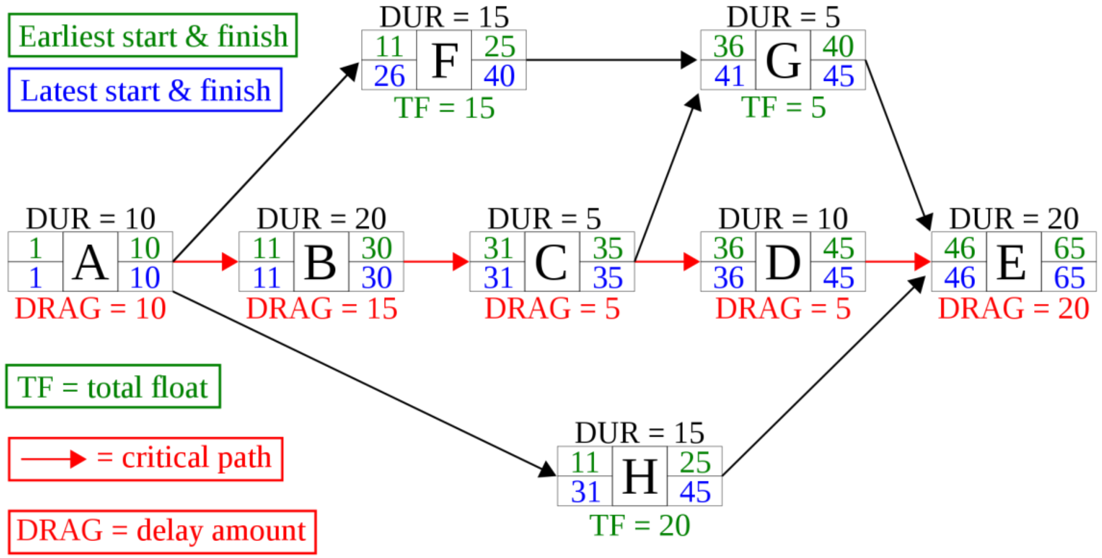
1.) a list of all activities required to complete the project

2.) the time (duration) that each activity will take to complete

3.) the dependencies between the activities

Critical Path Method (CPM)

- we construct an edge weighted **G(V,E)** directed acyclic graph (DAG) because it can be solved in linear running time
- add edges with **0** weight for each precedence constraint
- we have to find the longest path in order to solve the problem
- there are no cycles in such graphs



Bellman-Ford Algorithm

(Algorithms and Data Structures)

Bellman-Ford Algorithm

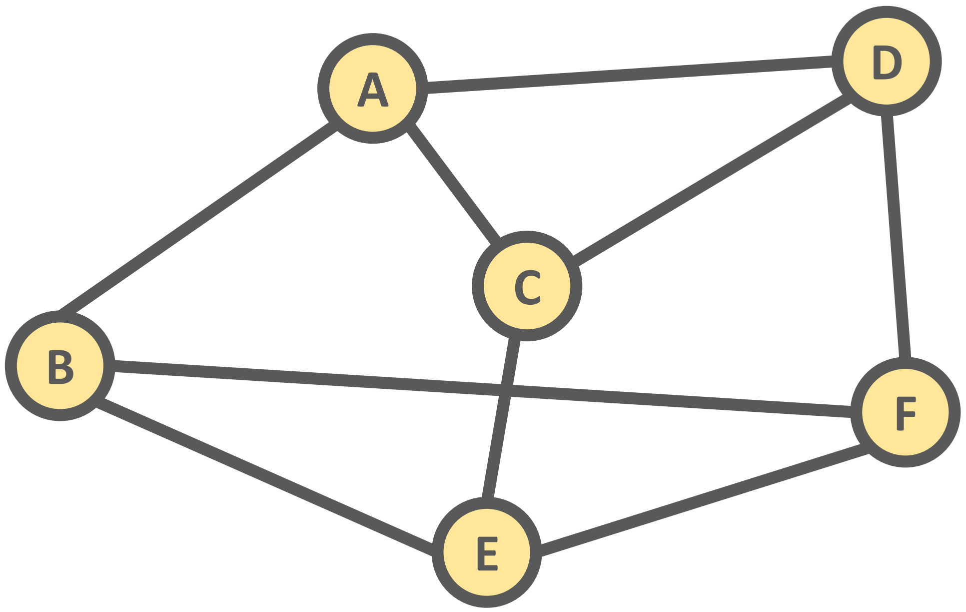
- it was first constructed in **1958** by **Bellman** and **Ford** independently
- slower than Dijkstra's algorithm but it is more robust – it can handle negative edge weights too
- Dijkstra's algorithm chooses the edges greedely in every iteration with the lowest cost
- **Bellman-Ford** relaxes all edges in a **G(V,E)** graph at the same time for **V-1** iterations

Bellman-Ford Algorithm

- **Bellman-Ford** relaxes all edges in a **G(V,E)** graph at the same time for **V-1** iterations
- the running time complexity is **O(V*E)**
- there is a minor problem: because of the negtive edge weights there may be **negative cycles**
- so it does **V-1** iterations and then an extra one to detect cycles: if cost decreases in the **V**th iteration than there is a negative cycle because all the paths are traversen up to the **V-1** iteration

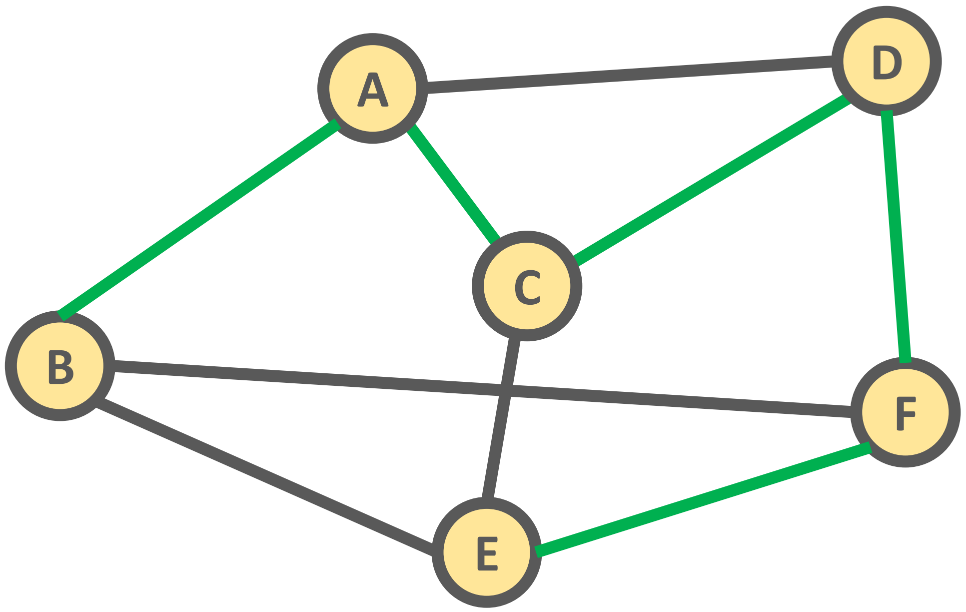
Bellman-Ford Algorithm

- why does Bellman-Ford algorithm make **V-1** iterations?
- because the maximal length of a shortest path between $\mathbf{v_i}$ and $\mathbf{v_j}$ arbitrary nodes in a **G(V,E)** graph is **|V|-1** (without cycles)



Bellman-Ford Algorithm

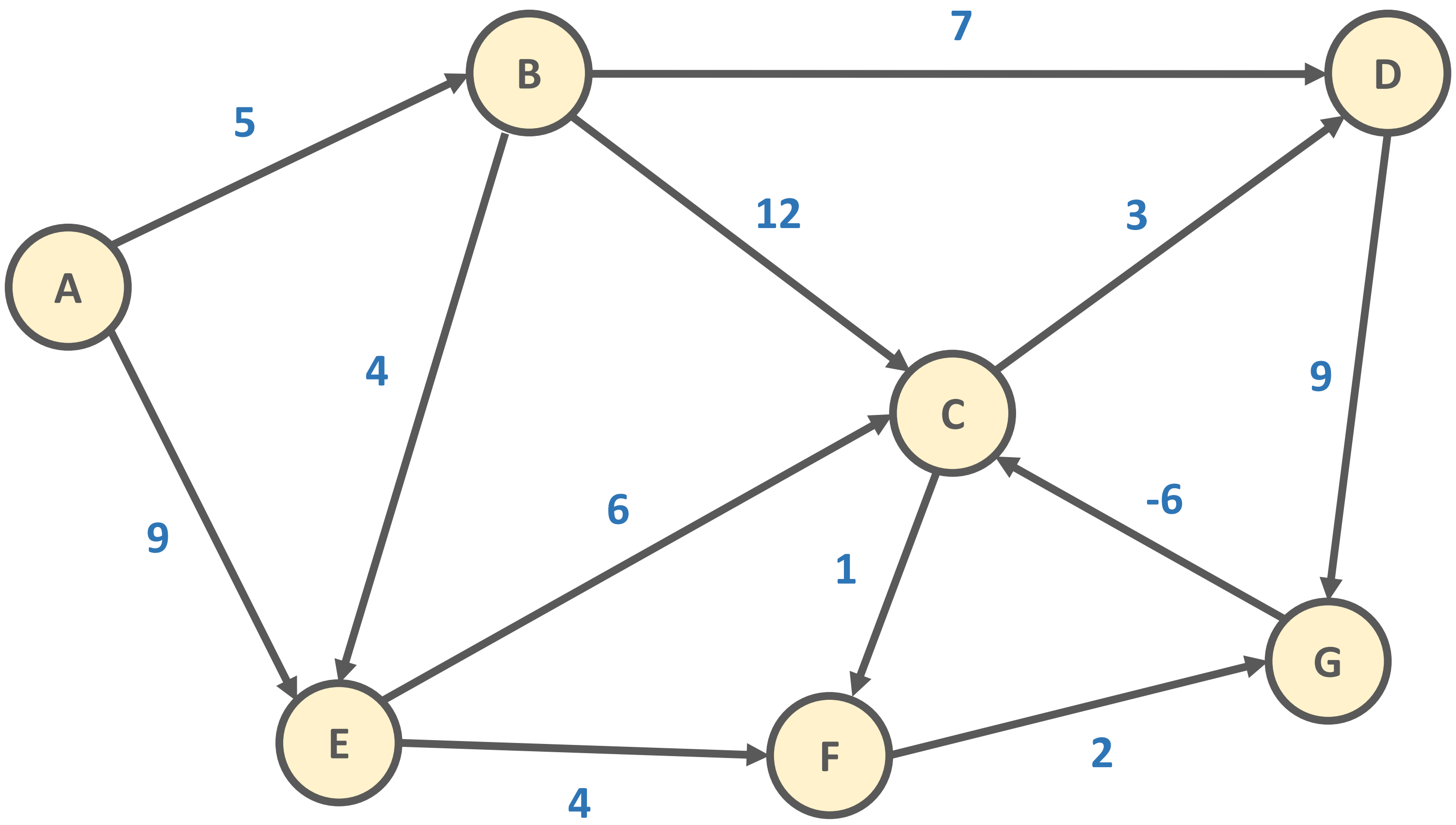
- why does Bellman-Ford algorithm make **V-1** iterations?
- because the maximal length of a shortest path between $\mathbf{v_i}$ and $\mathbf{v_j}$ arbitrary nodes in a **G(V,E)** graph is **|V|-1** (without cycles)



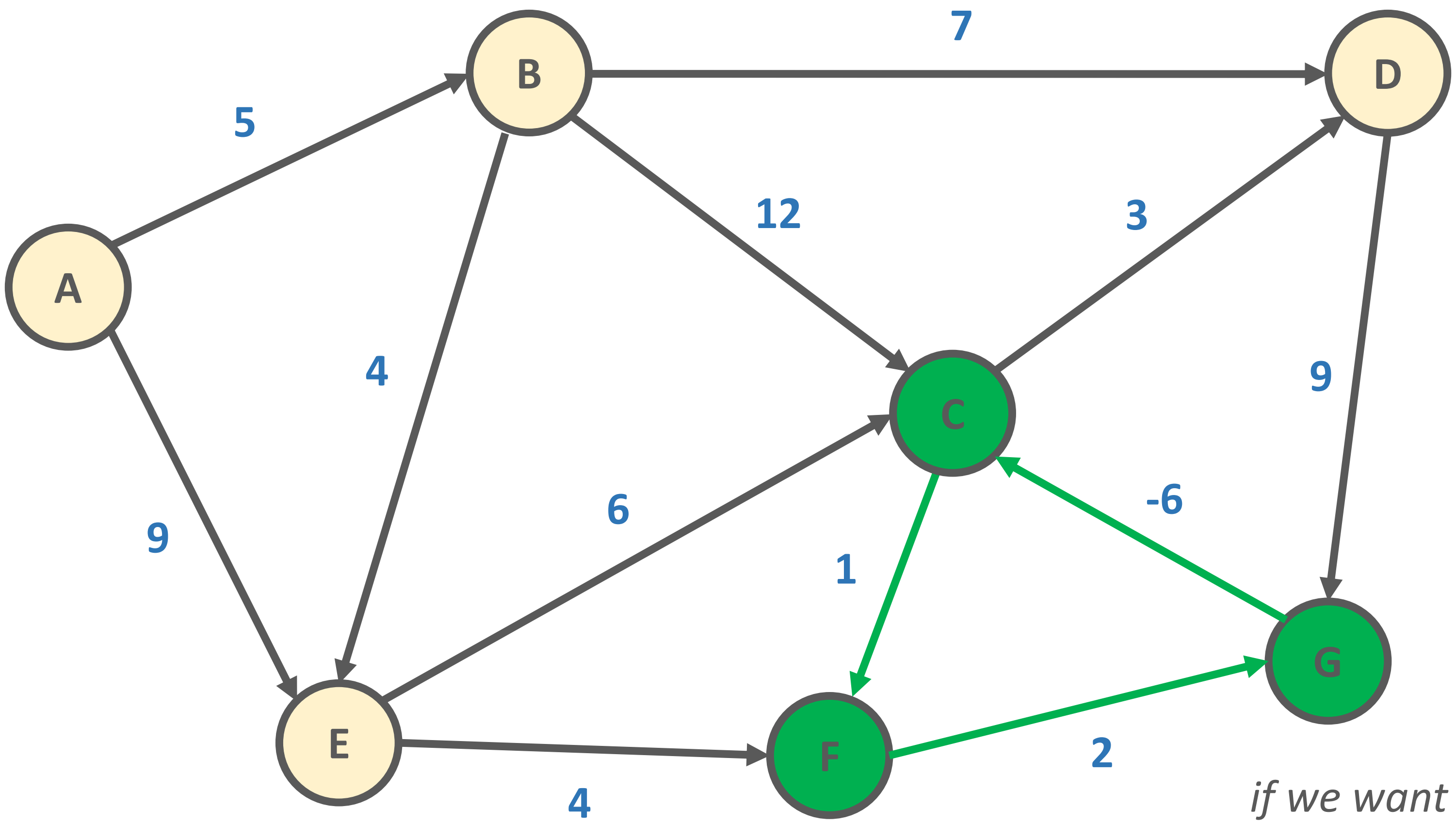
Bellman-Ford Algorithm

- why does Bellman-Ford algorithm make **V-1** iterations**?**
- because the maximal length of a shortest path between $\mathbf{v_i}$ and $\mathbf{v_j}$ arbitrary nodes in a **G(V,E)** graph is **|V|-1**
- so we know that if we make an additional iteration after **V-1** iterations and the there is a change in the shortest path then there is a negative cycle in the **G(V,E)** graph

Bellman-Ford Algorithm



Bellman-Ford Algorithm



*if we want to find the **shortest path** then we **make infinite loops** in the cycle because every loop decreases the total cost*

Bellman-Ford Algorithm

```
for {v1 v2 ... vn} all nodes in G(V,E):

        for each edge (u,v) with weight w in edges

                dist = distance[u] + w

                if dist < distance[v]
                    distance[v] = dist
                    predecessor[v] = u

for each edge (u,v) with weight w in edges
        if distance[u] + w < distance[v]
                error: „Negative cycle detected”
```

Bellman-Ford Algorithm

for {$\mathbf{v_1}$ $\mathbf{v_2}$ ... $\mathbf{v_n}$} all nodes in **G(V,E)**:

*consider all the $\mathbf{v_i}$ nodes in the **G(V,E)** graph in **O(V)** running time*

    for each edge **(u,v)** with weight **w** in edges

        dist = distance[u] + w

        if dist < distance[v]
            distance[v] = dist
            predecessor[v] = u

for each edge **(u,v)** with weight **w** in edges

    if distance[u] + w < distance[v]

        error: „Negative cycle detected”

Bellman-Ford Algorithm

```
for {v1 v2 ... vn} all nodes in G(V,E):

        for each edge (u,v) with weight w in edges

                dist = distance[u] + w

                if dist < distance[v]
                    distance[v] = dist
                    predecessor[v] = u

for each edge (u,v) with weight w in edges
        if distance[u] + w < distance[v]
                error: „Negative cycle detected”
```

*in every iteration we consider all the **(u,v)** edges with **w** edge weight in **O(E)** running time*

Bellman-Ford Algorithm

for {v_1 v_2 ... v_n} all nodes in **G(V,E)**:

for each edge **(u,v)** with weight **w** in edges

dist = distance[u] + w

if dist < distance[v]
distance[v] = dist
predecessor[v] = u

*this is the so-called **RELAXATION***
we calculate the possible shortest paths
*to the given nodes in **O(1)** running time*

for each edge **(u,v)** with weight **w** in edges
if distance[u] + w < distance[v]
error: „Negative cycle detected”

Bellman-Ford Algorithm

for $\{v_1\ v_2\ \ldots\ v_n\}$ all nodes in **G(V,E)**:

```
for {v1 v2 ... vn} all nodes in G(V,E):

        for each edge (u,v) with weight w in edges

                dist = distance[u] + w

                if dist < distance[v]
                        distance[v] = dist
                        predecessor[v] = u

for each edge (u,v) with weight w in edges
        if distance[u] + w < distance[v]
                error: „Negative cycle detected”
```

*after making **V-1** iterations we have found even the longest shortest path so we make an additional loop to check **negative cycles** in **O(E)** running time*

Bellman-Ford Algorithm

```
for {v1 v2 ... vn} all nodes in G(V,E):

        for each edge (u,v) with weight w in edges

                dist = distance[u] + w

                if dist < distance[v]
                    distance[v] = dist
                    predecessor[v] = u

for each edge (u,v) with weight w in edges
        if distance[u] + w < distance[v]
                error: „Negative cycle detected”
```

RUNNGINT TIME ANALYSIS

O(V) * [O(E) * O(1)] + O(E) =
O(V) * O(E) + O(E) =
O(V*E) + O(E) =
O(V*E)

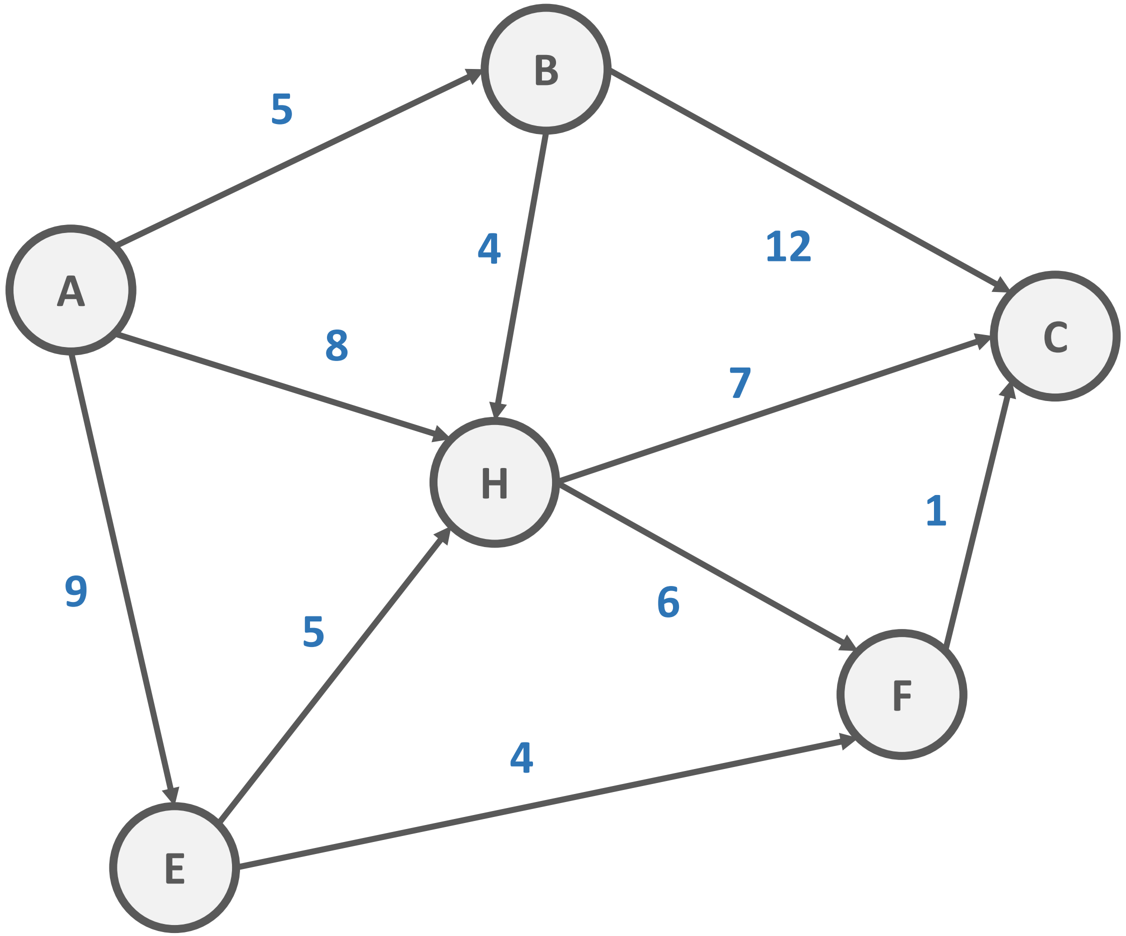
Bellman-Ford Algorithm

- there may be a slight optimization for **Bellman-Ford algorithm**
- it was first introduced by **Yen** back in **1970**
- we can terminate the algorithm if there is no change in the distances between two iterations (in the relaxation phases)
- we use the same technique in **bubble sort**

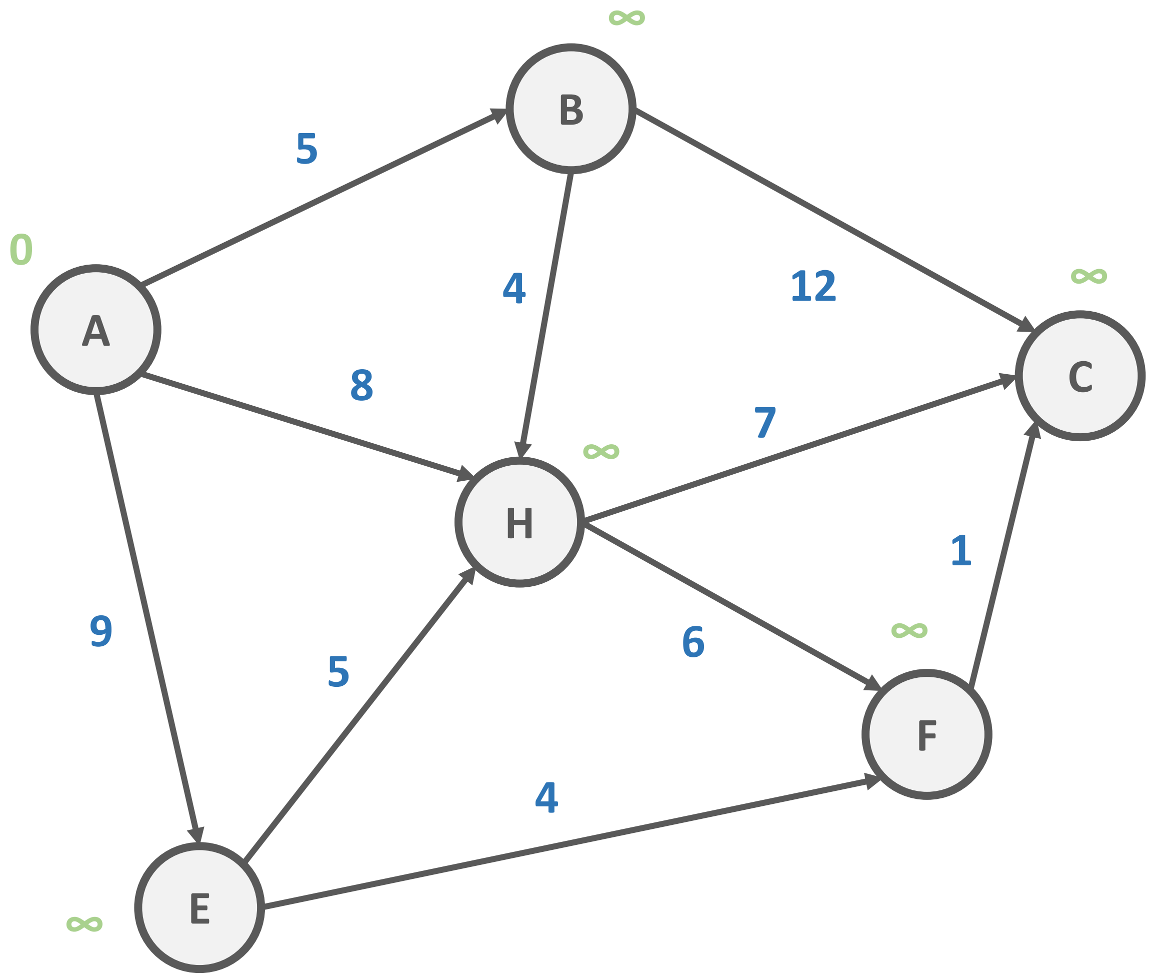
Bellman-Ford Algorithm

(Algorithms and Data Structures)

Bellman-Ford Algorithm



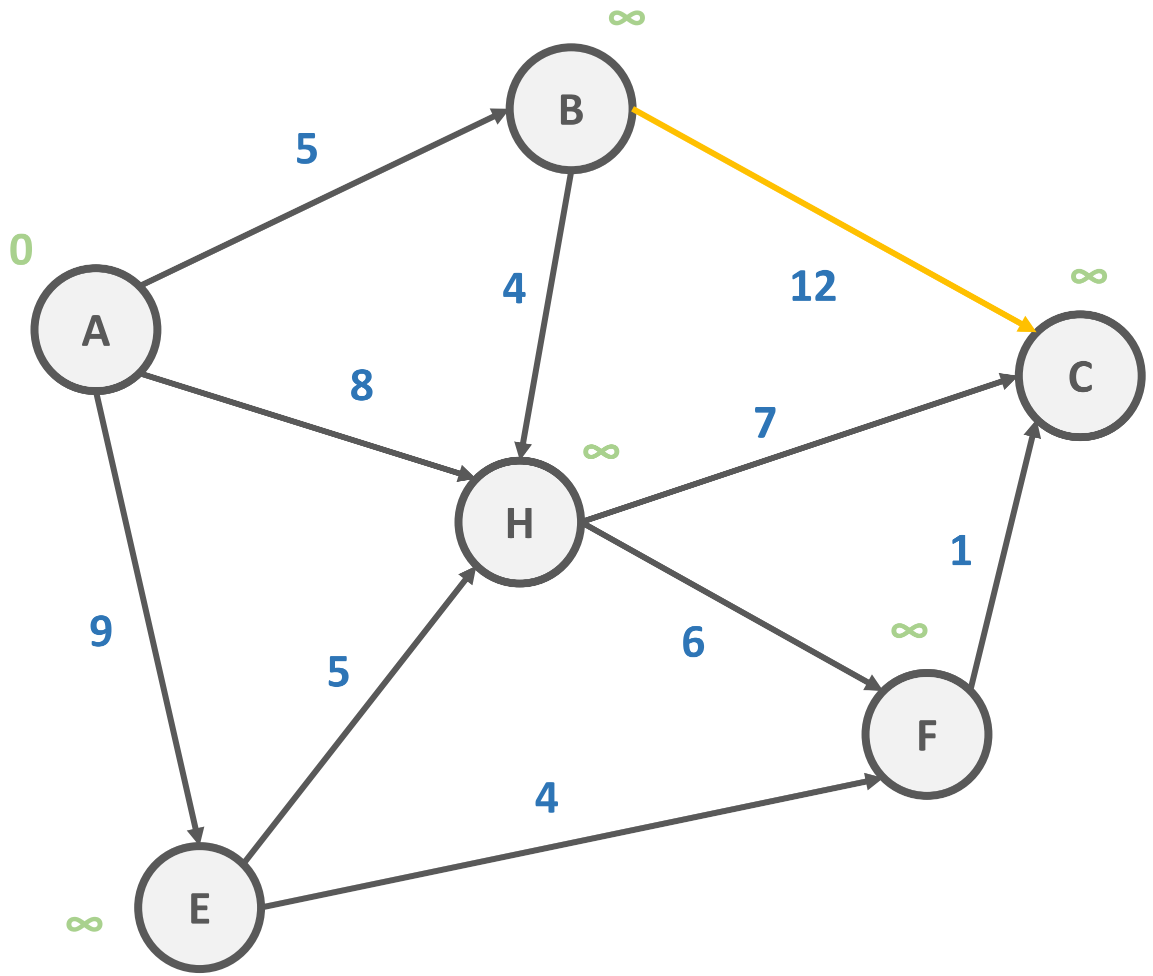
Bellman-Ford Algorithm



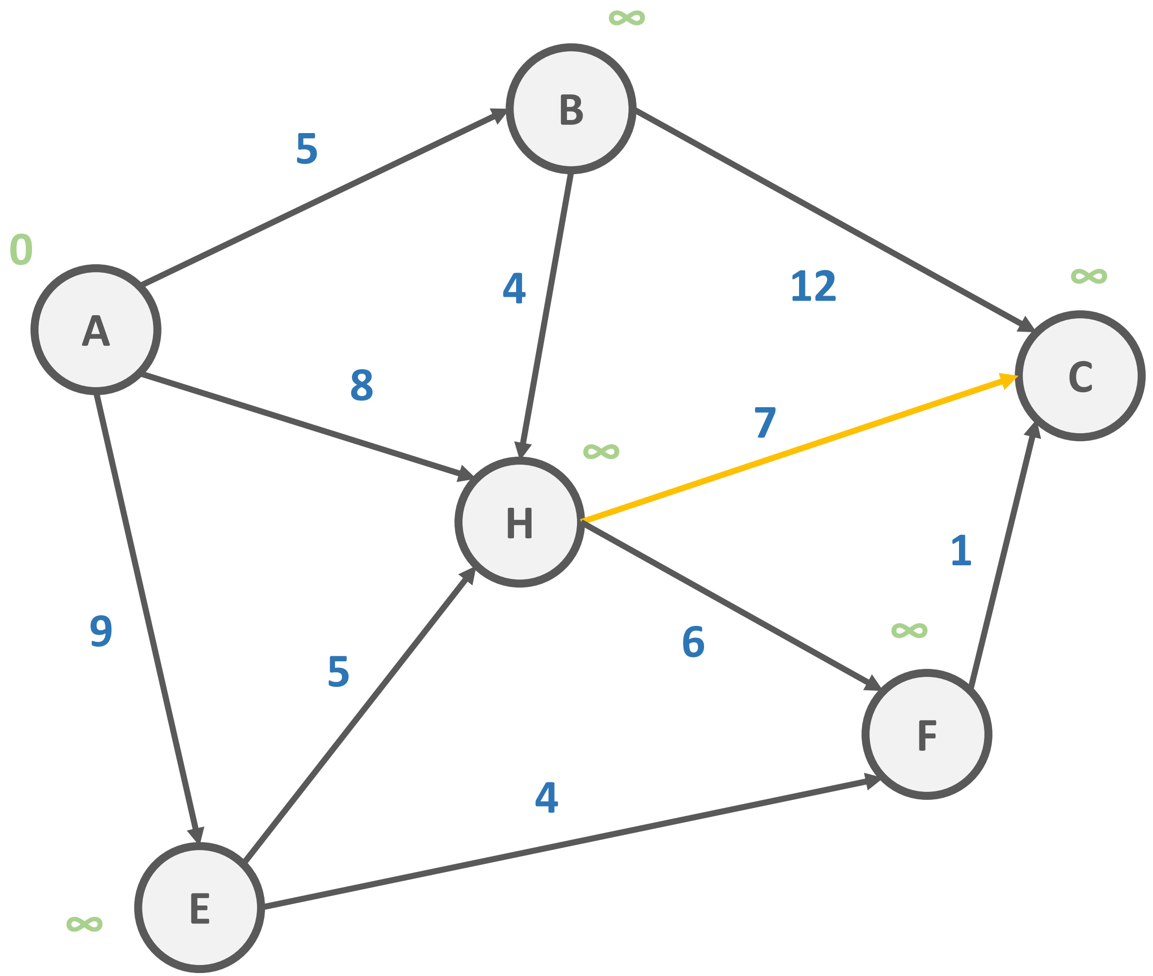
Bellman-Ford Algorithm

ITERATION #1

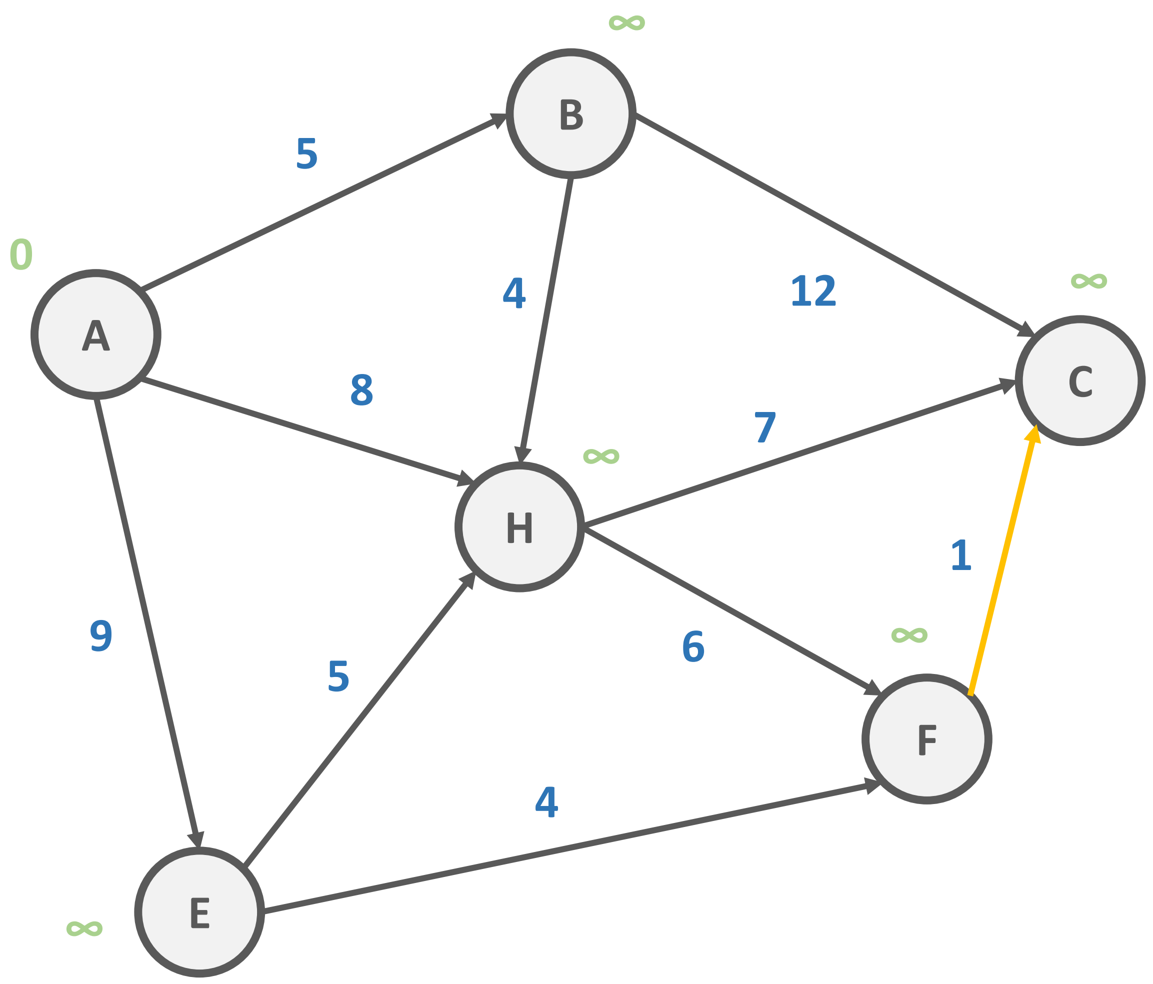
Bellman-Ford Algorithm



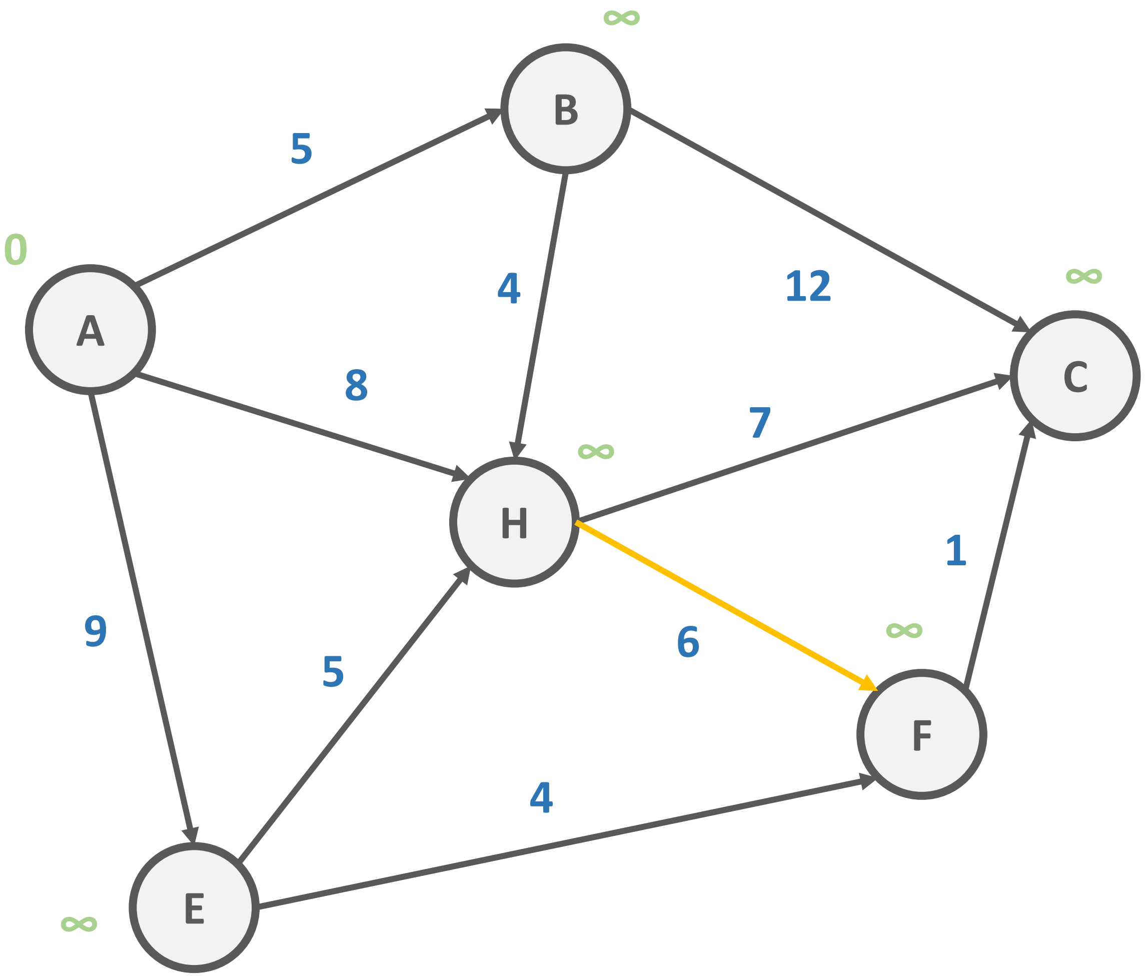
Bellman-Ford Algorithm



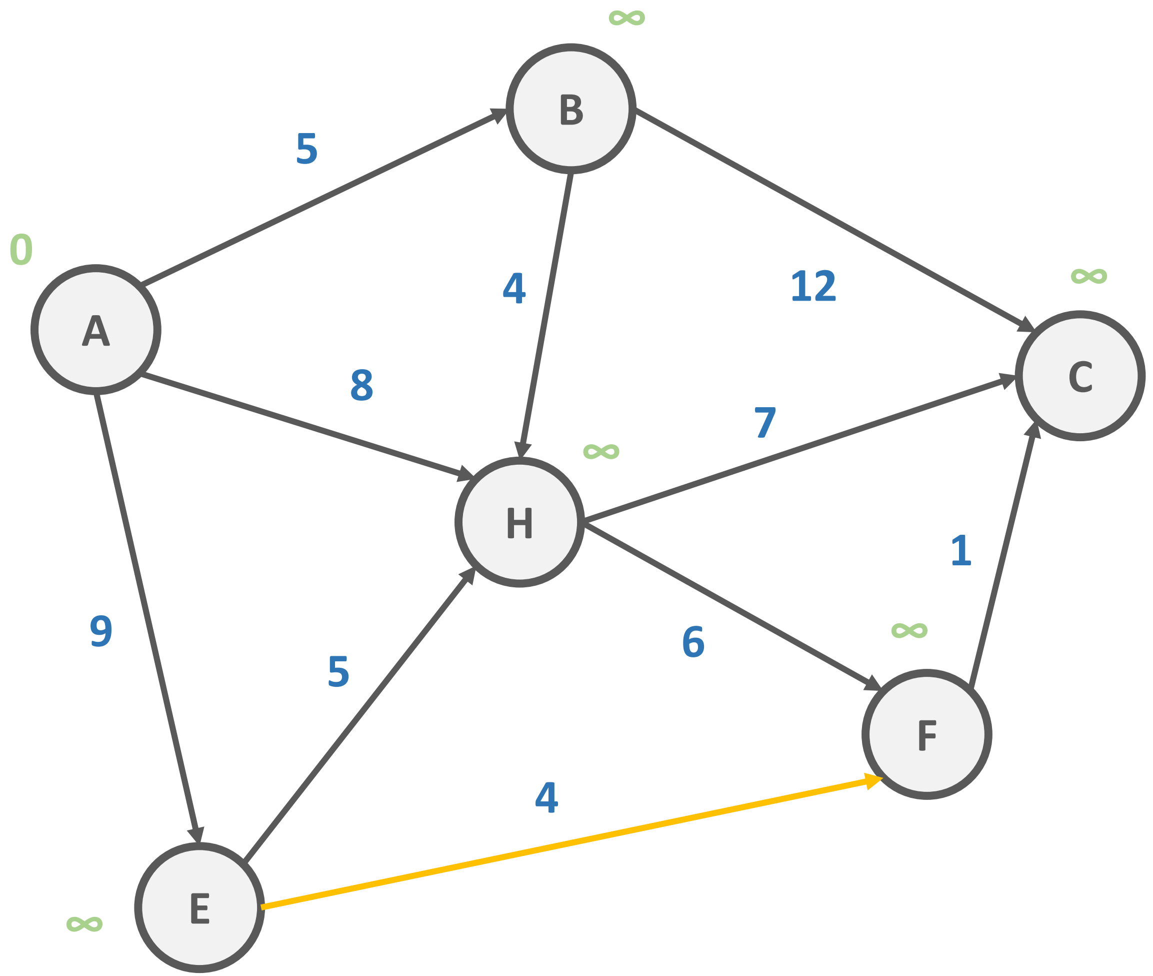
Bellman-Ford Algorithm



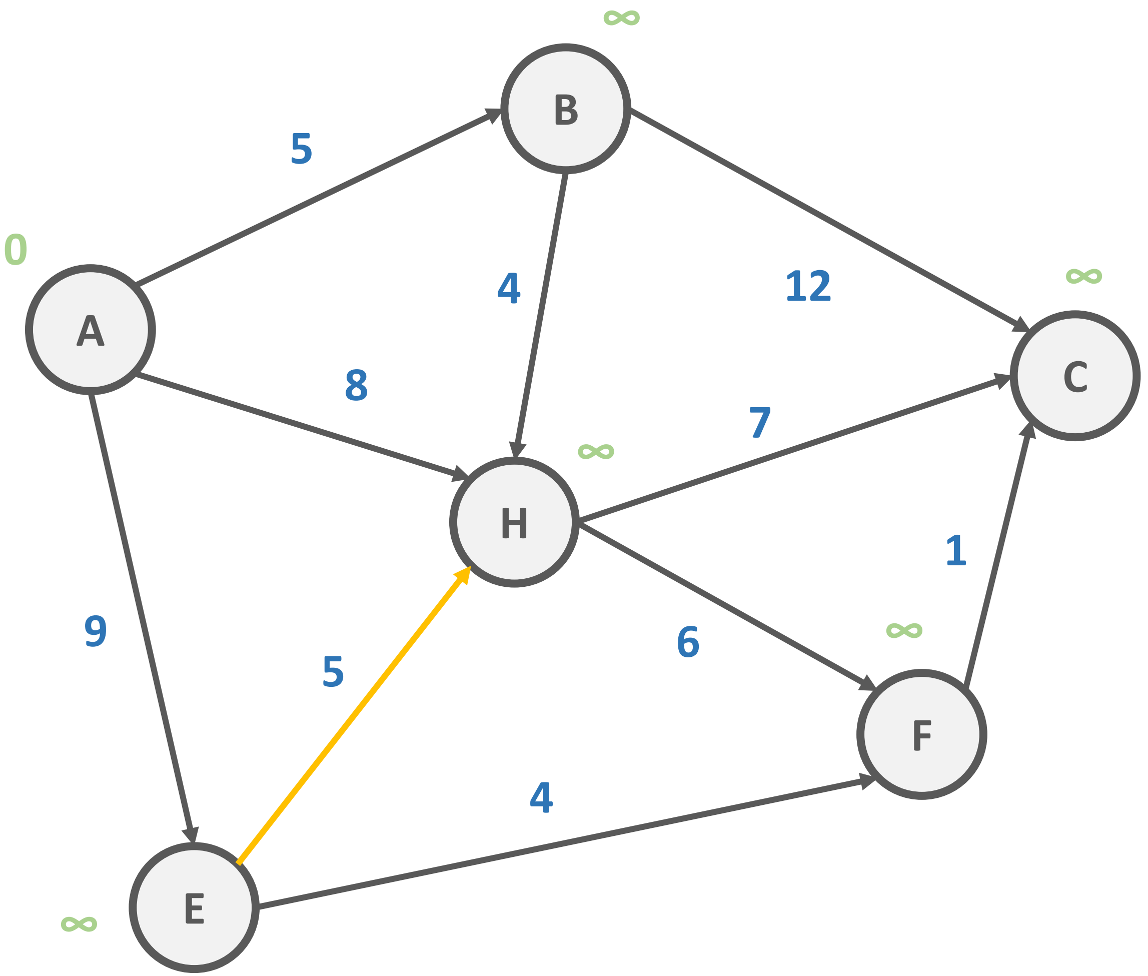
Bellman-Ford Algorithm



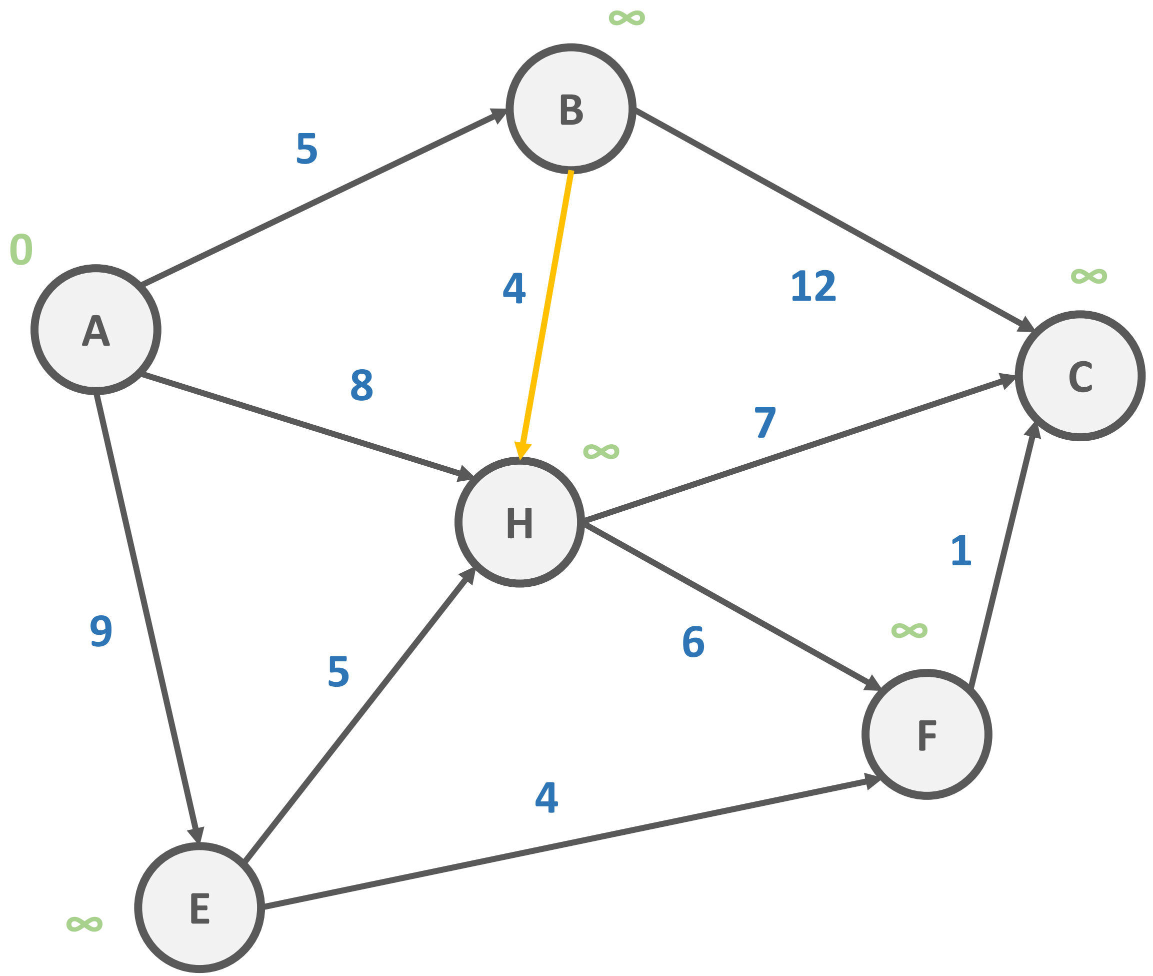
Bellman-Ford Algorithm



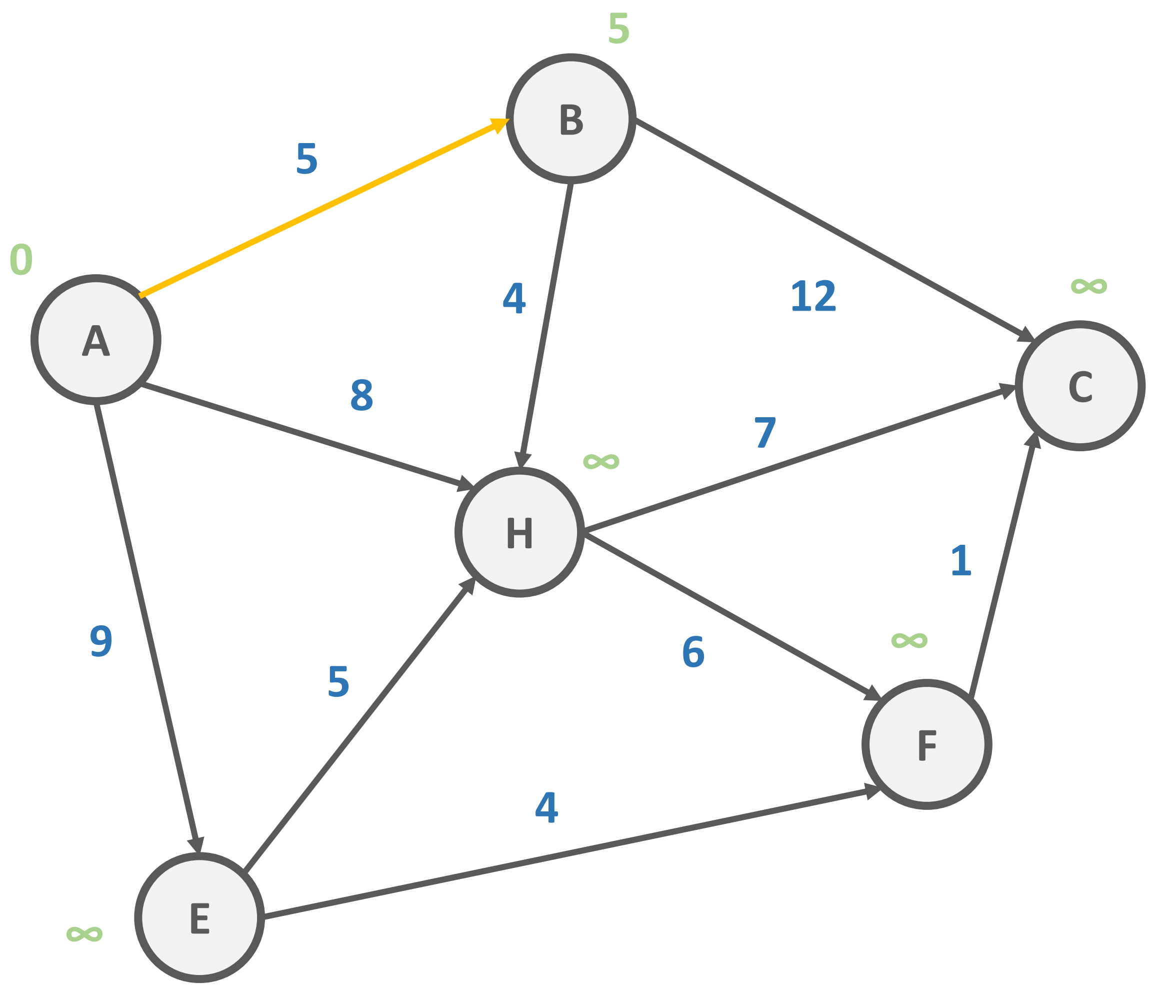
Bellman-Ford Algorithm



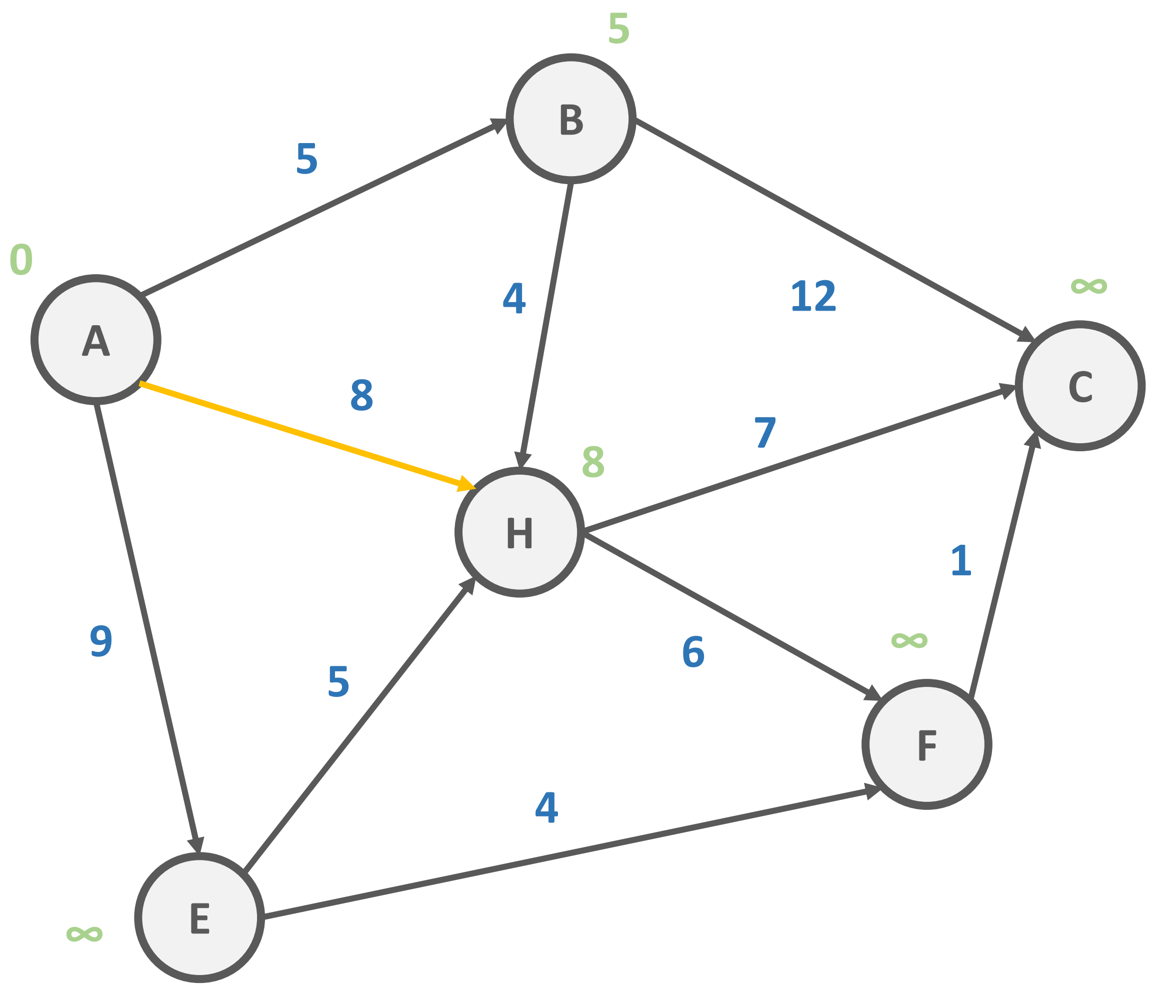
Bellman-Ford Algorithm



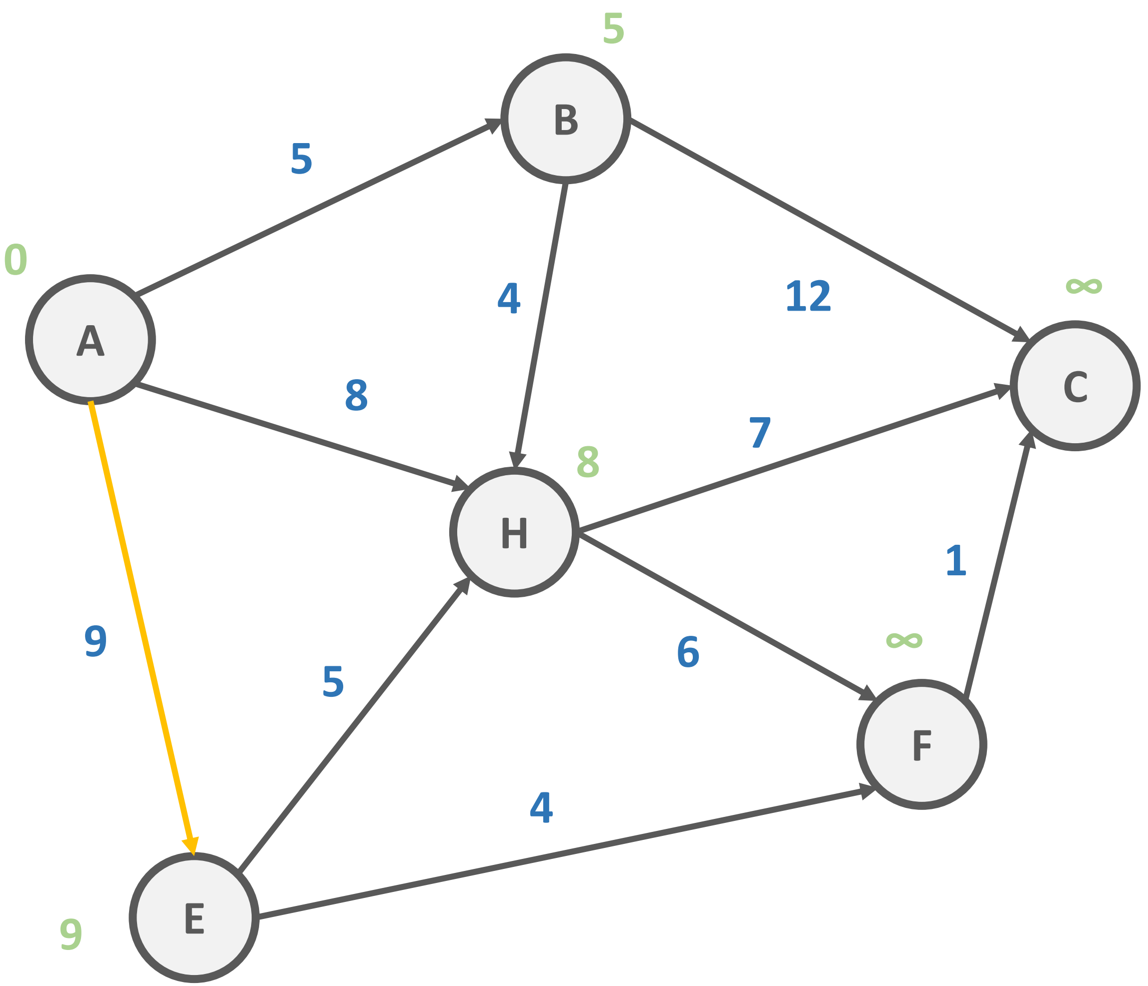
Bellman-Ford Algorithm



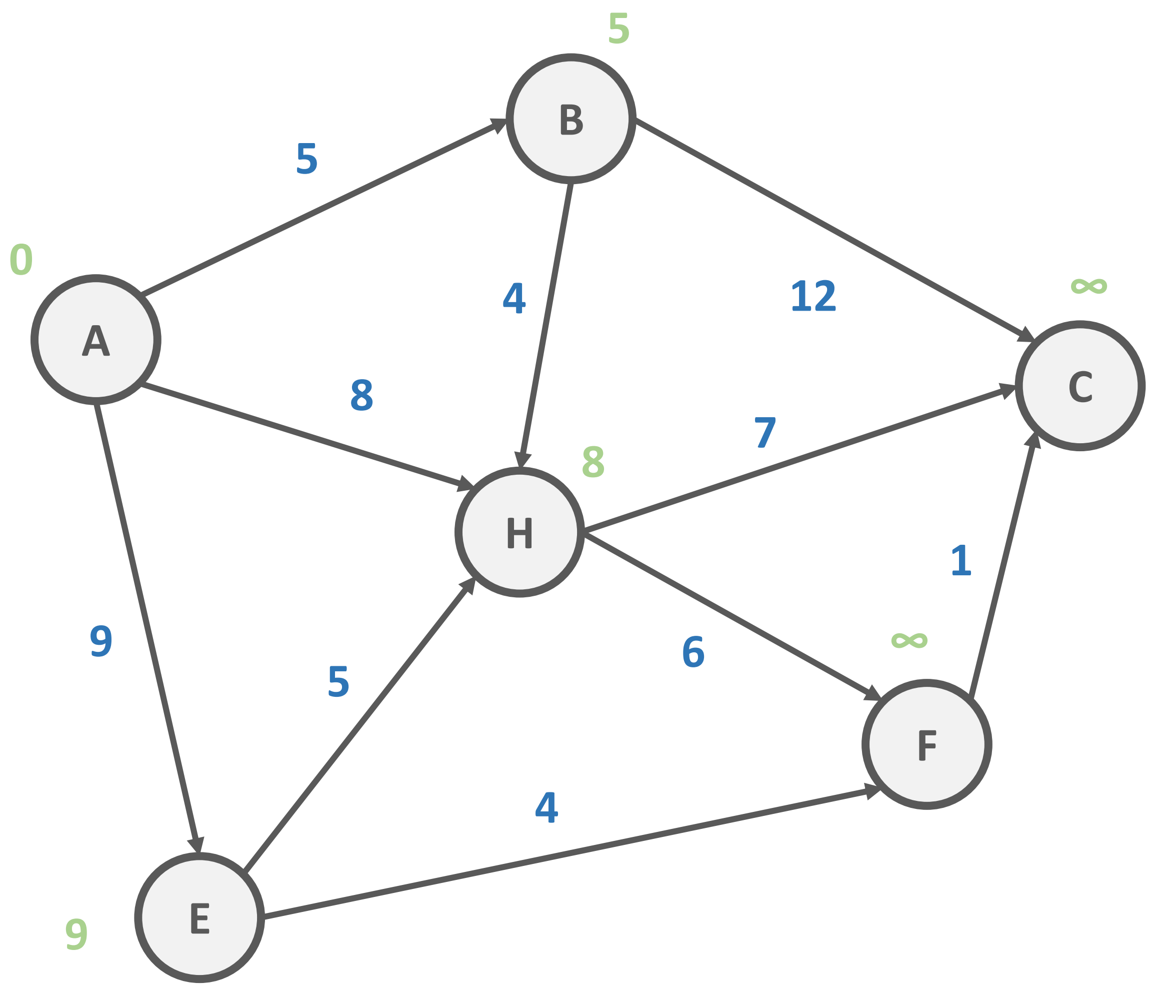
Bellman-Ford Algorithm



Bellman-Ford Algorithm



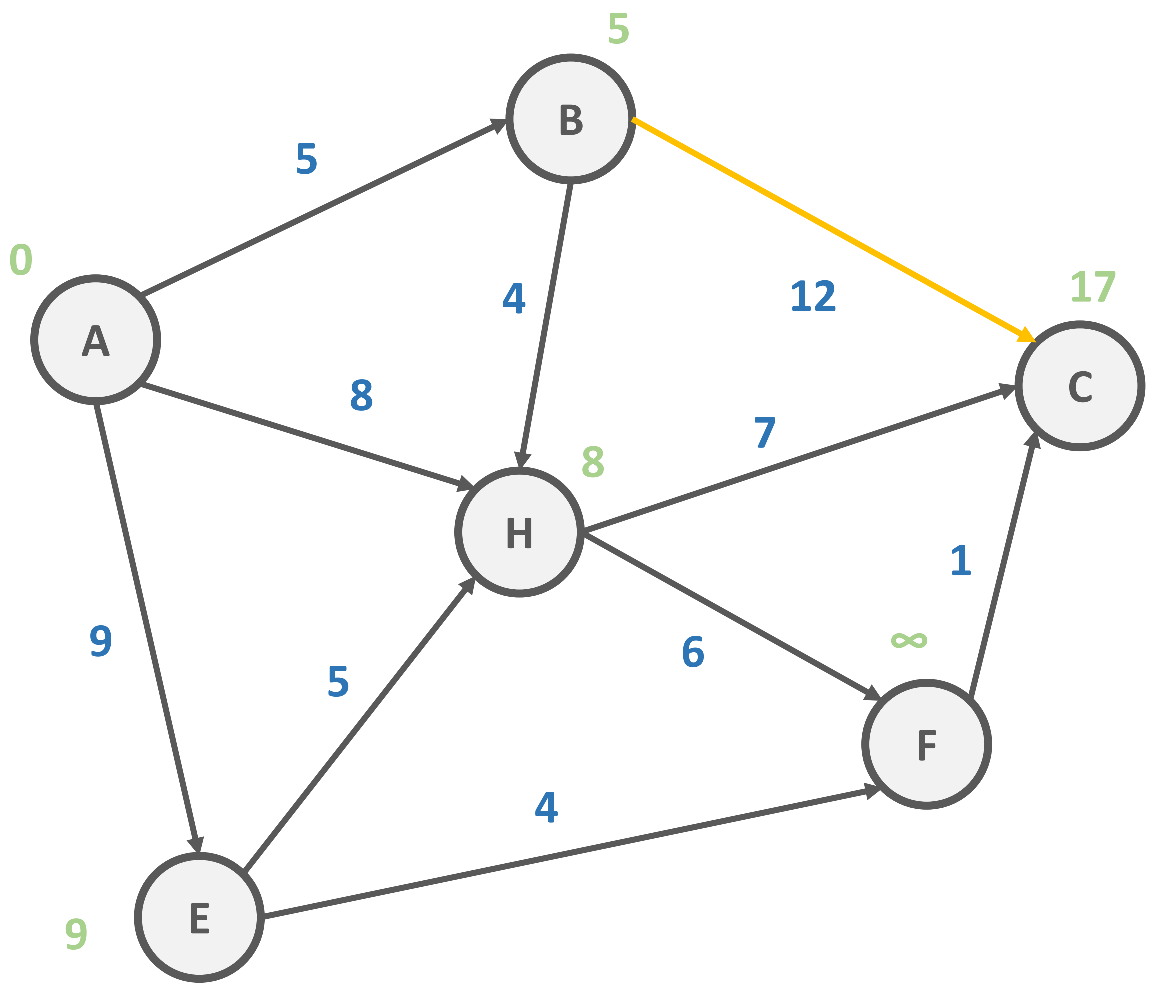
Bellman-Ford Algorithm



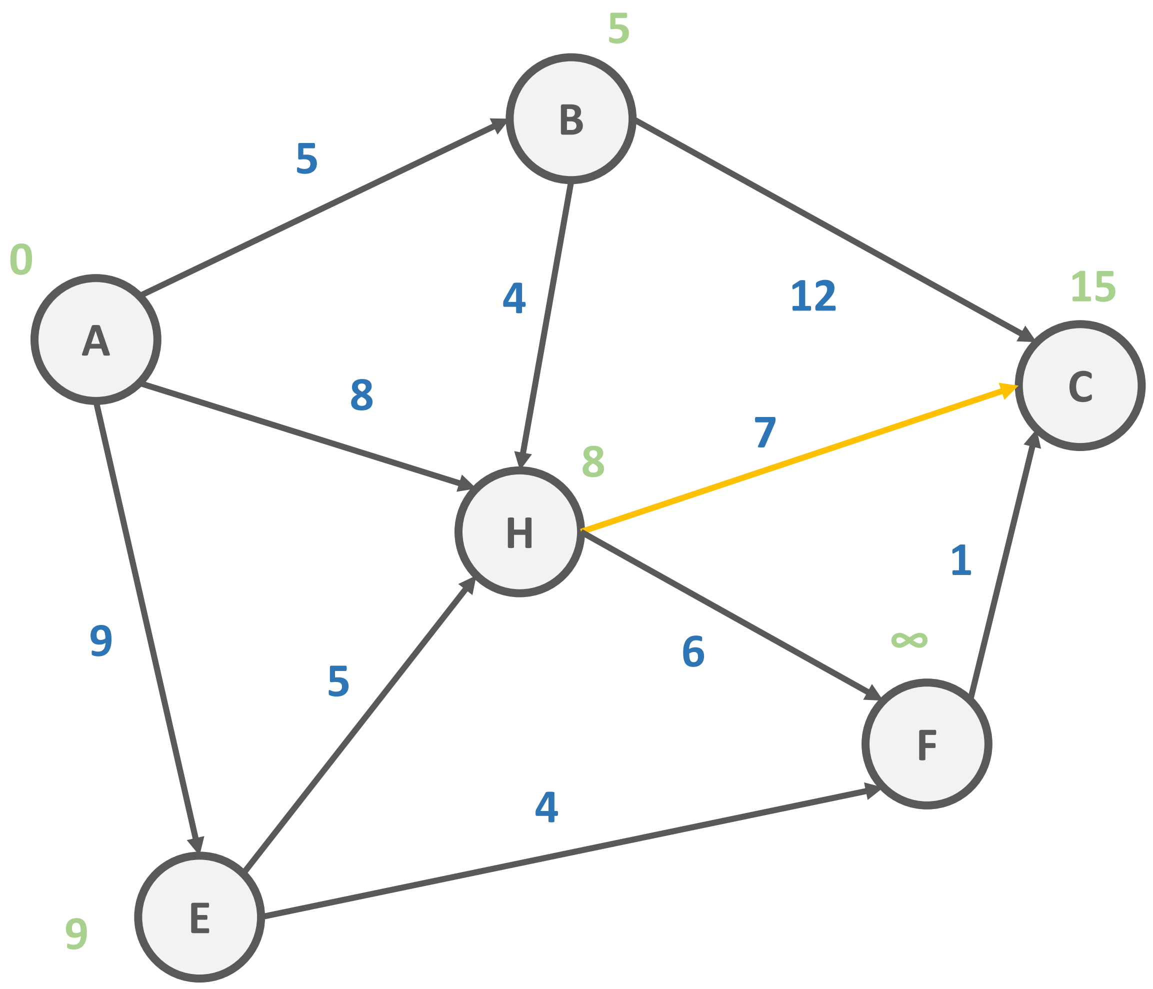
Bellman-Ford Algorithm

ITERATION #2

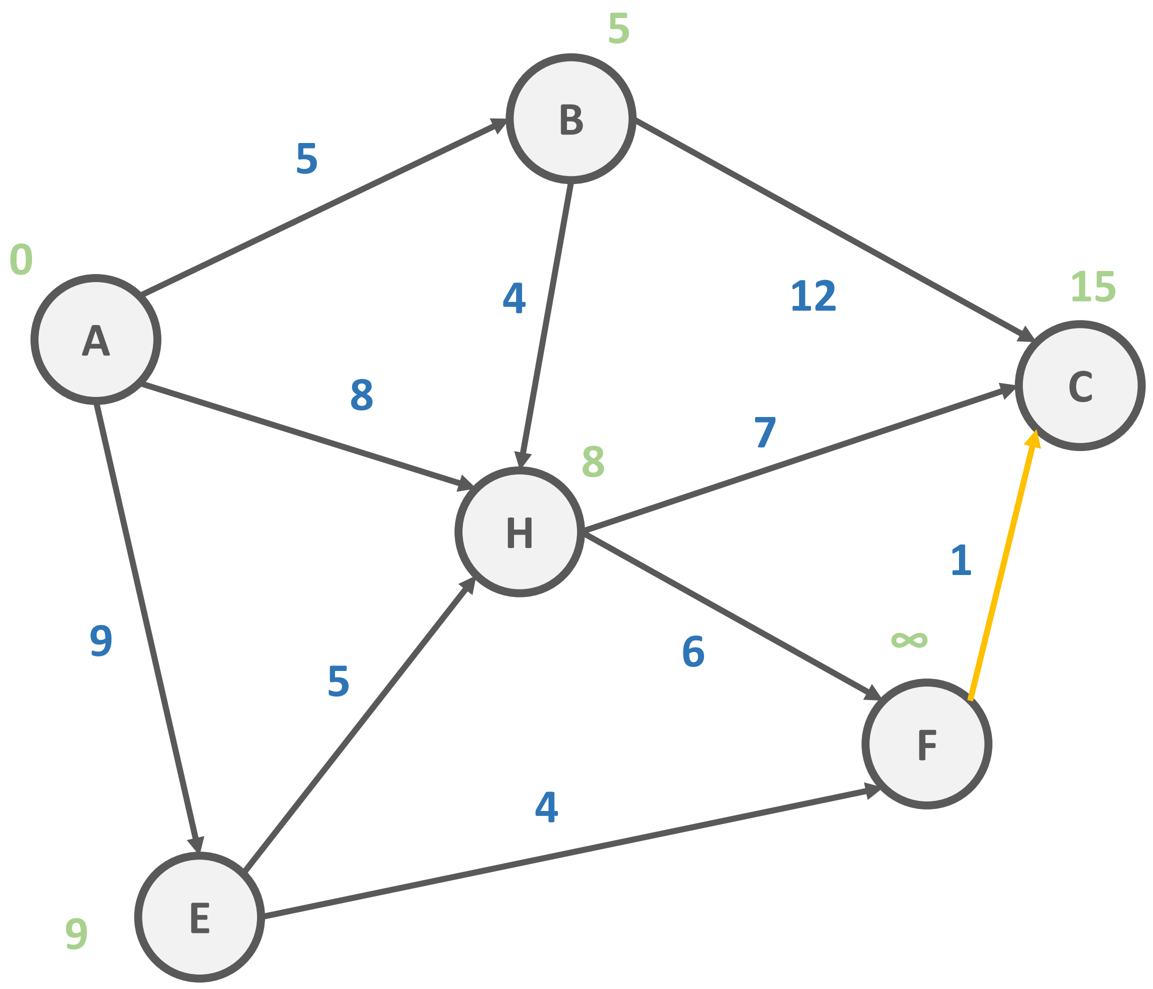
Bellman-Ford Algorithm



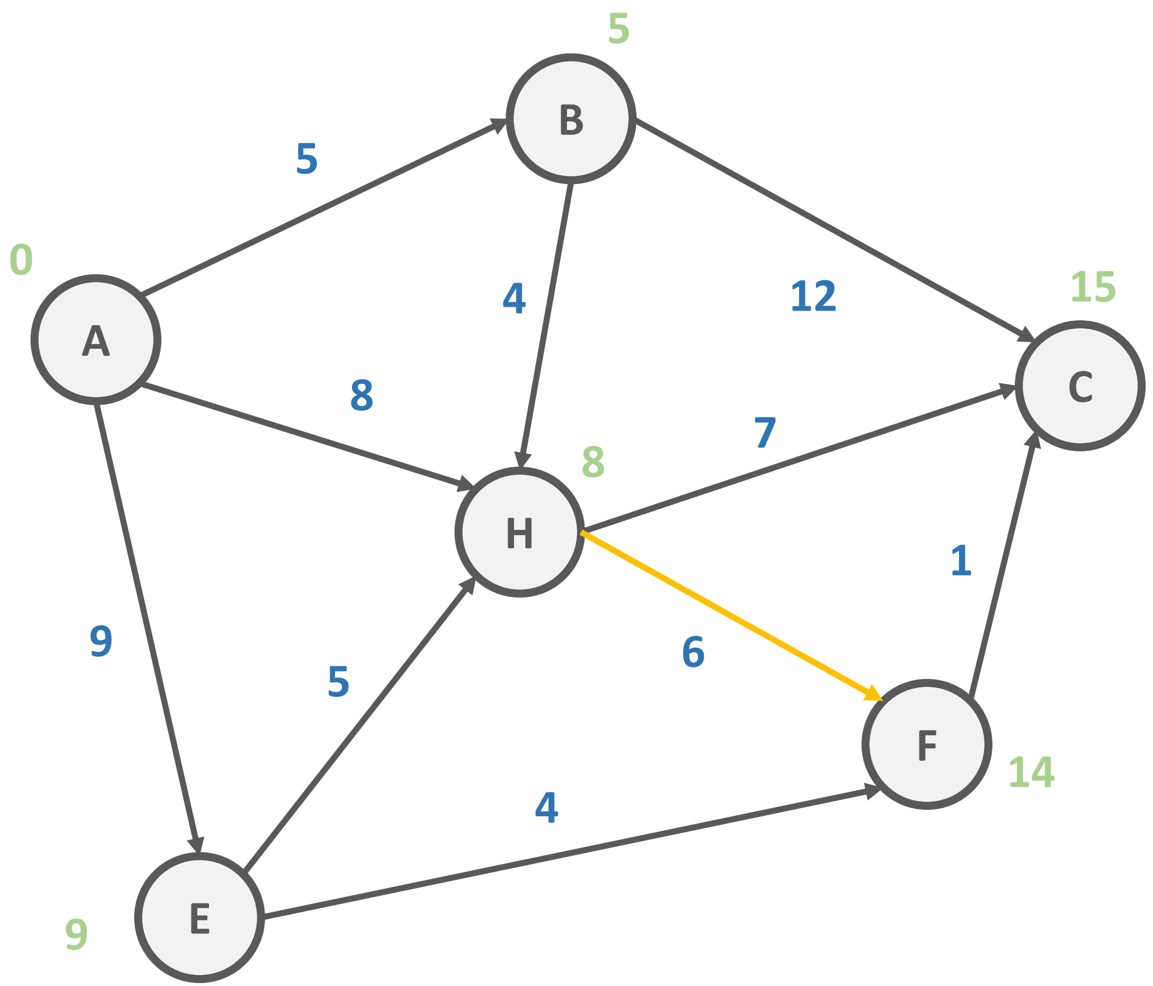
Bellman-Ford Algorithm



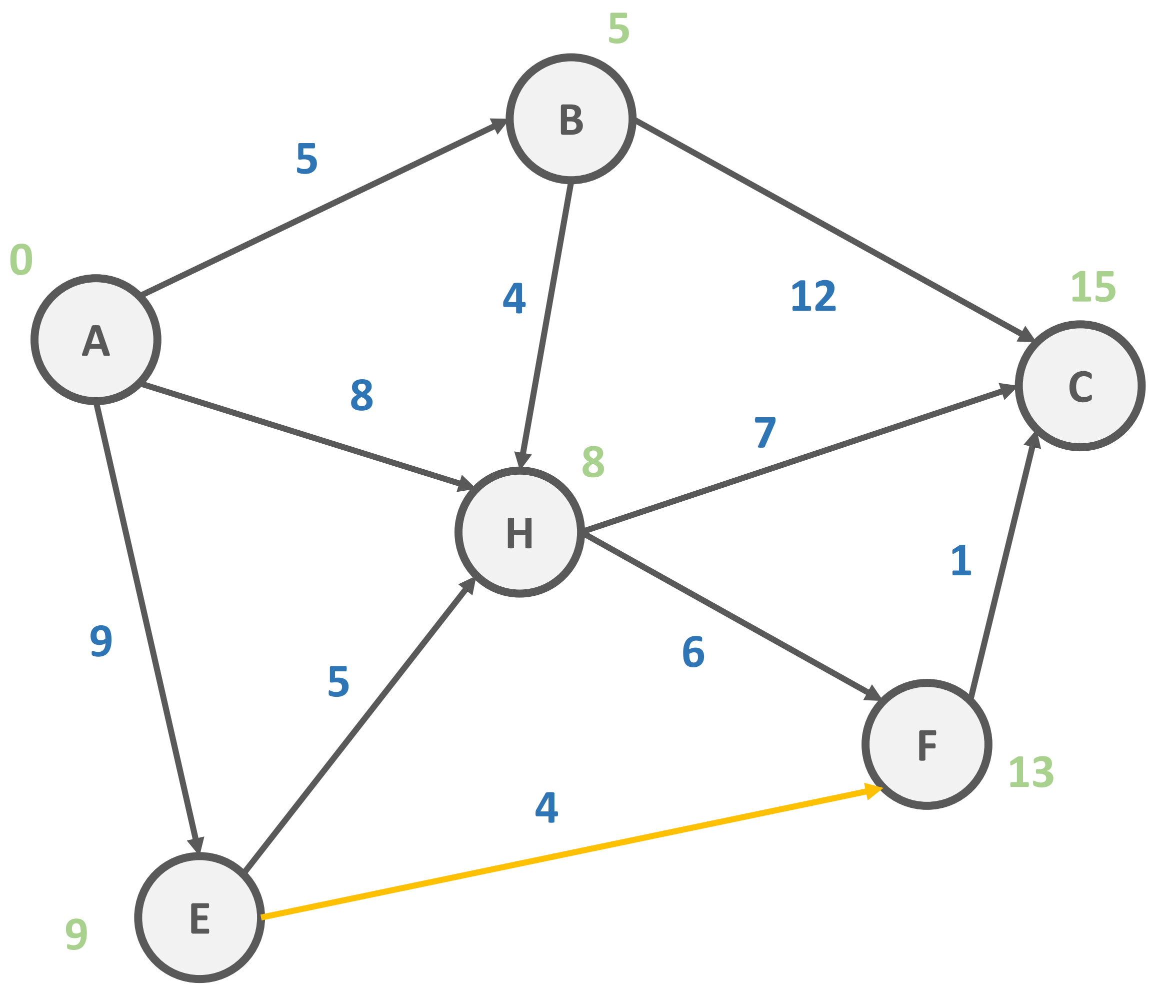
Bellman-Ford Algorithm



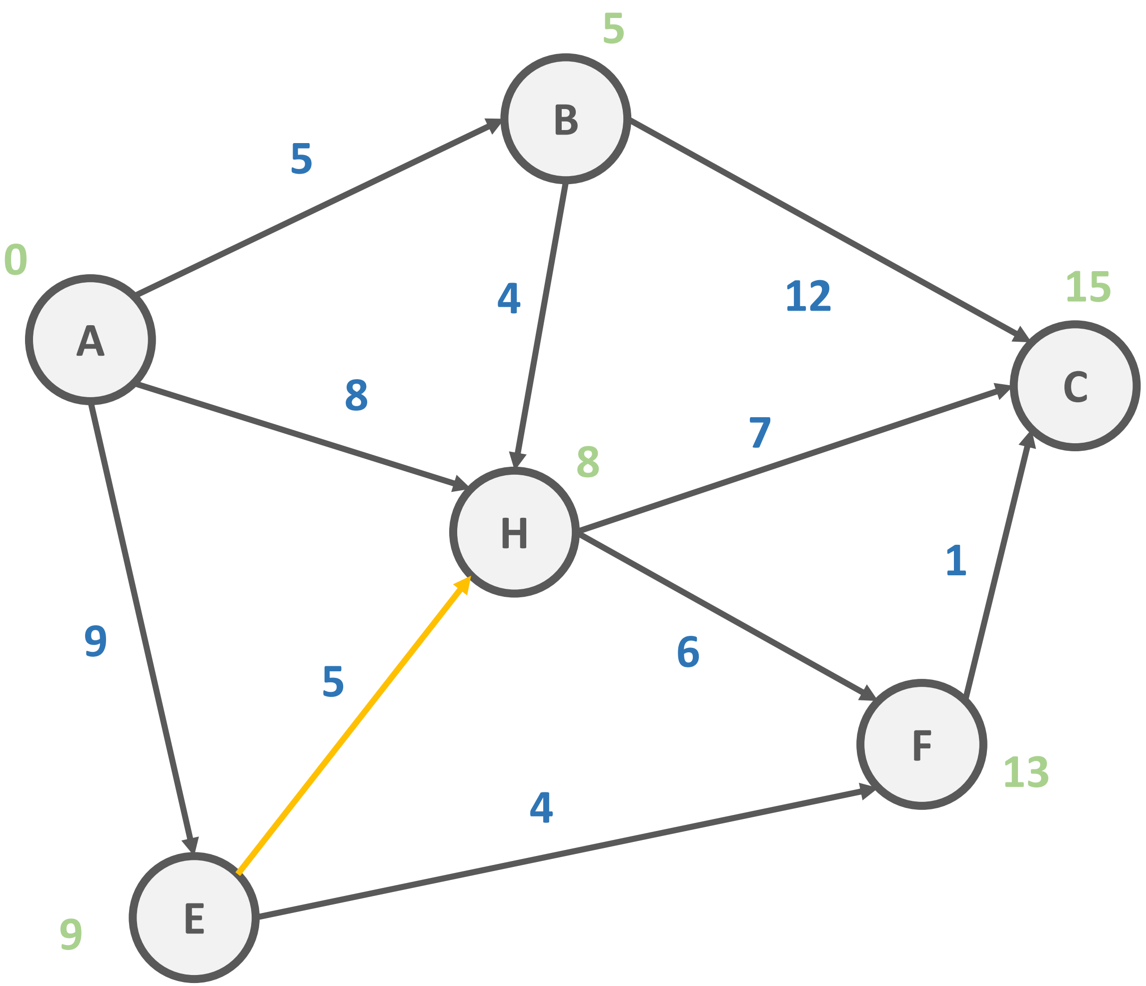
Bellman-Ford Algorithm



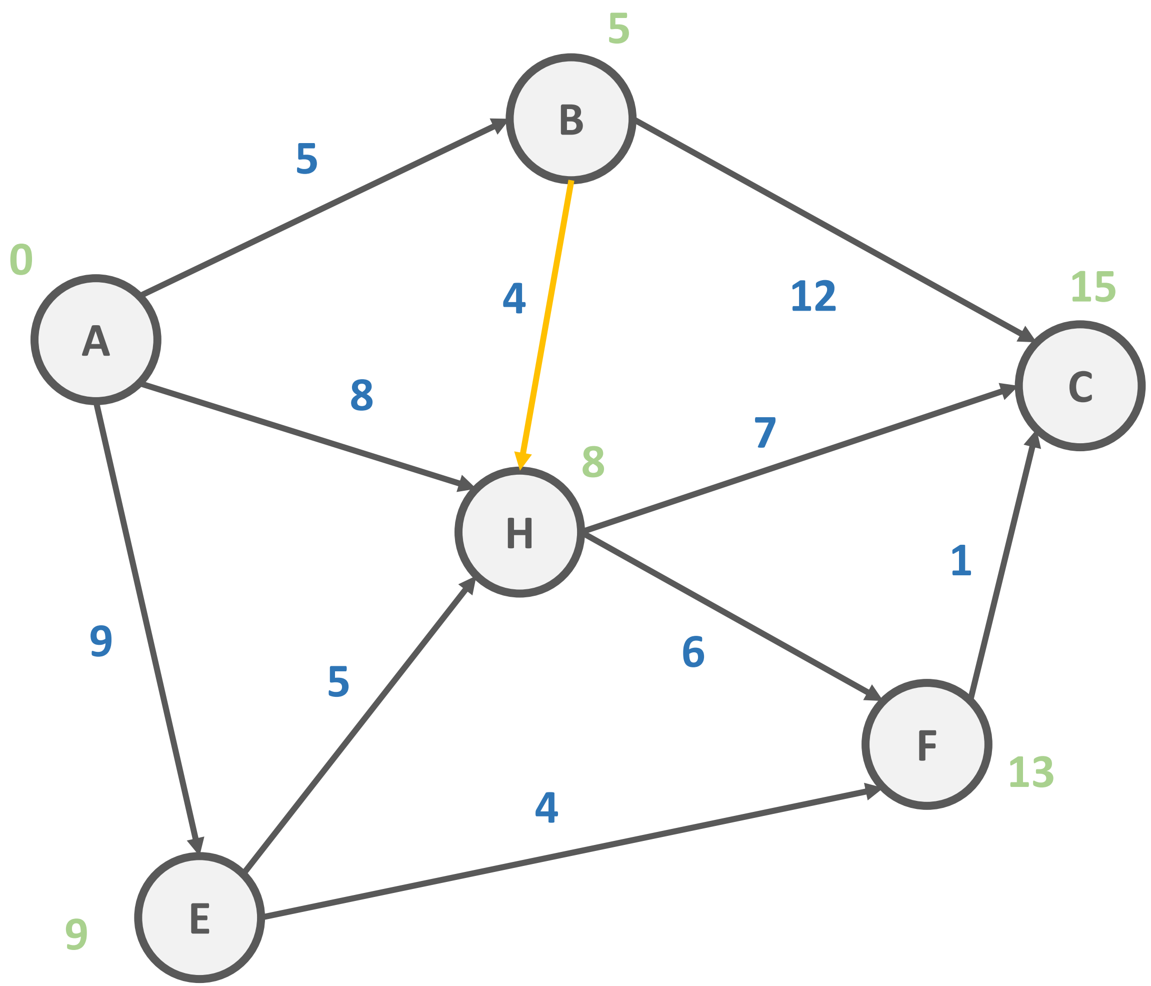
Bellman-Ford Algorithm



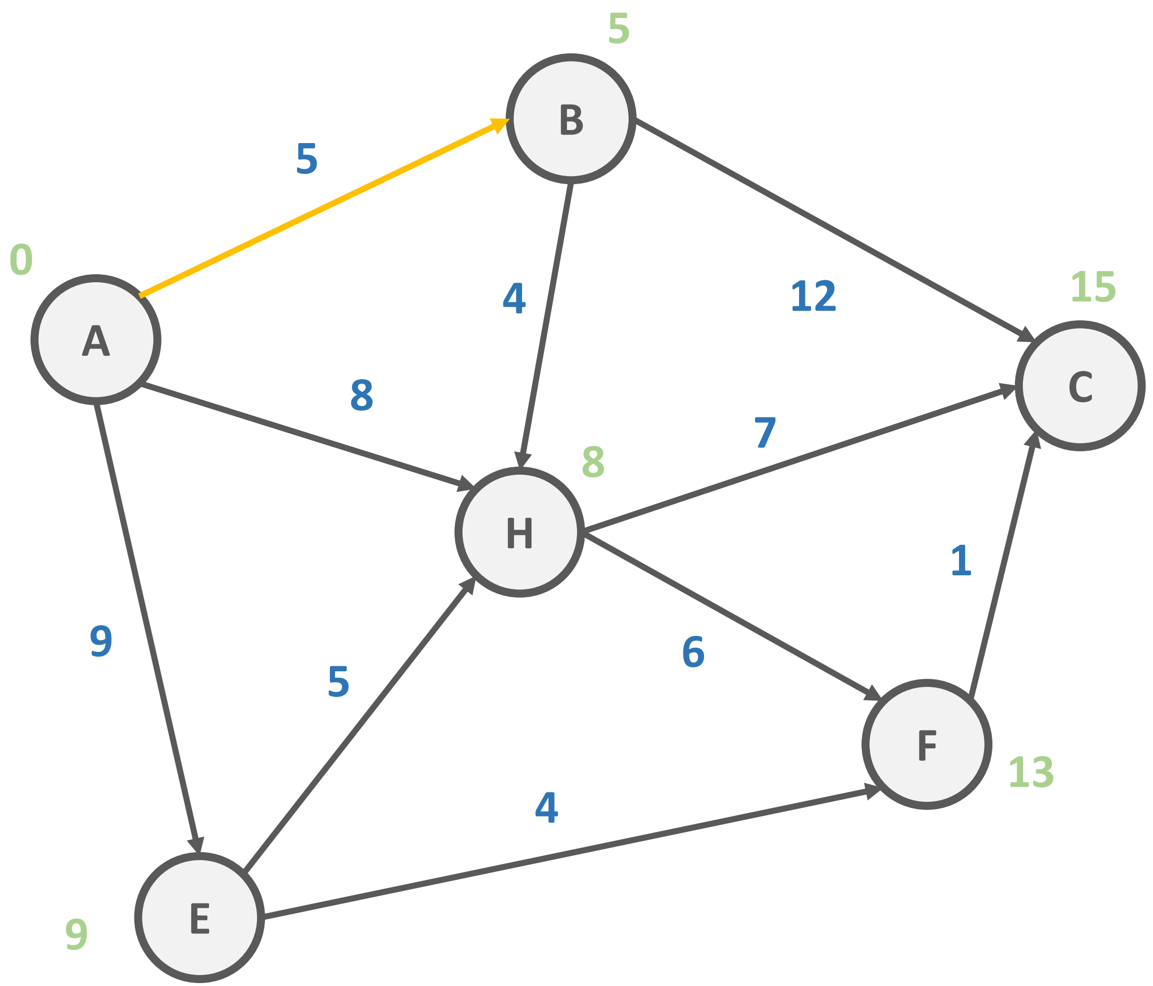
Bellman-Ford Algorithm



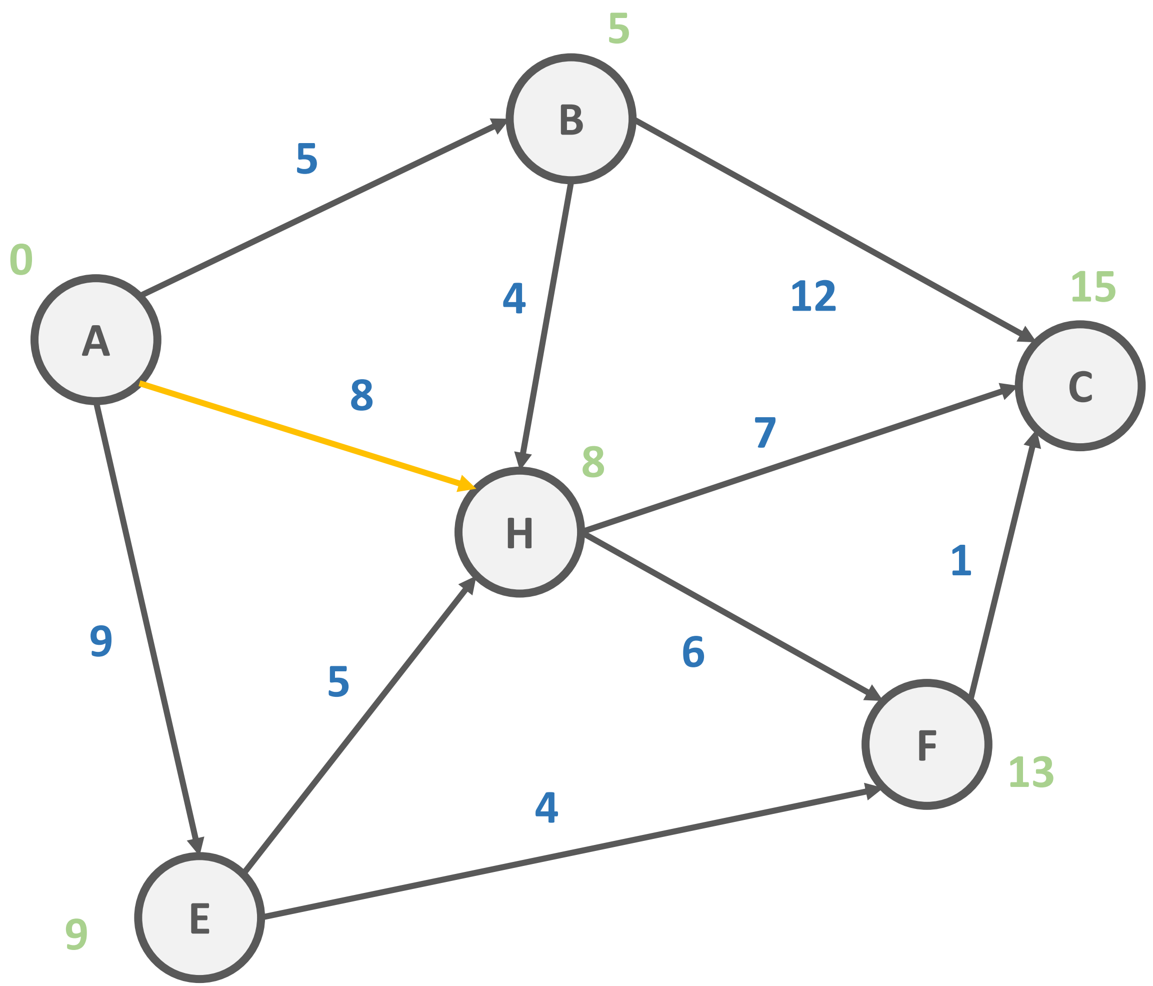
Bellman-Ford Algorithm



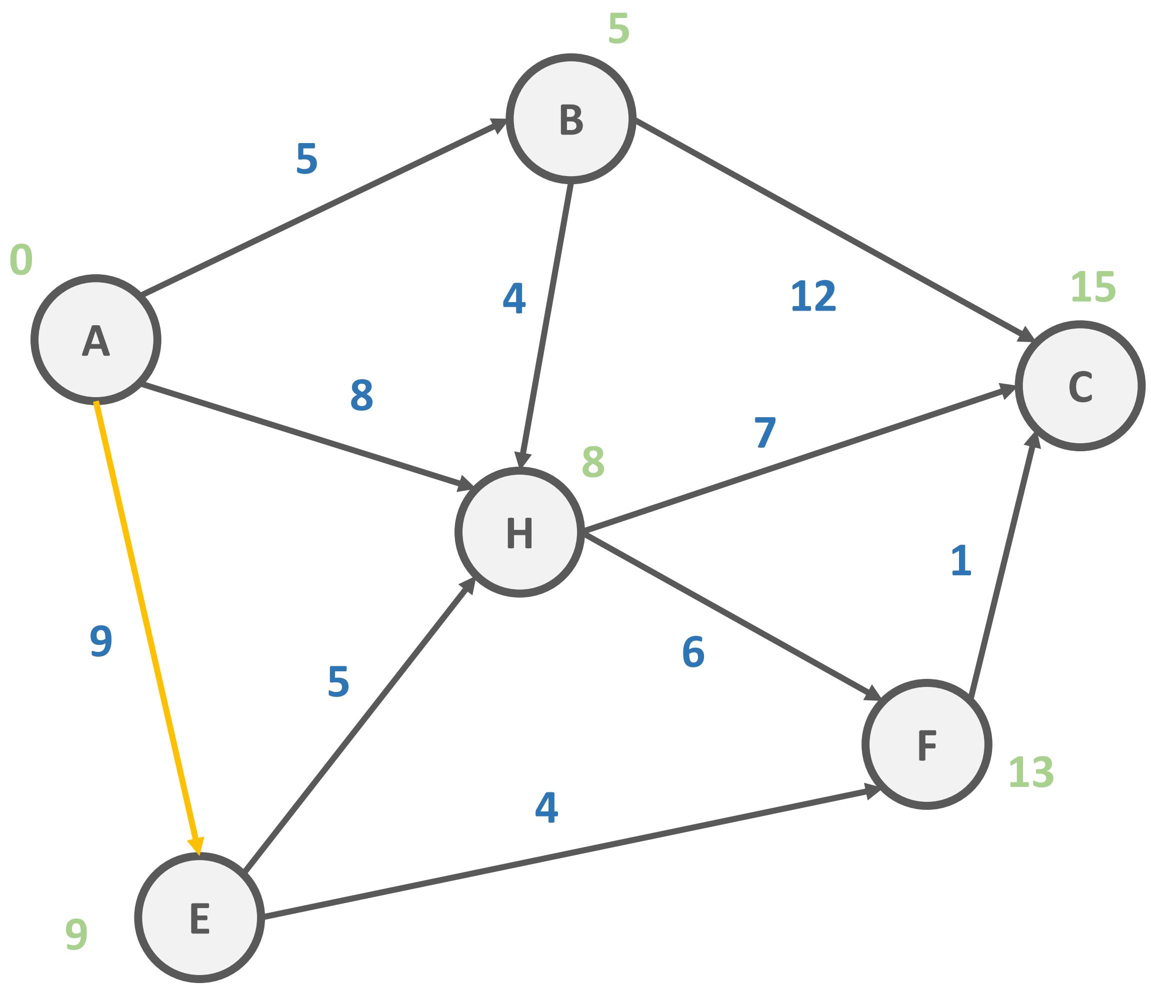
Bellman-Ford Algorithm



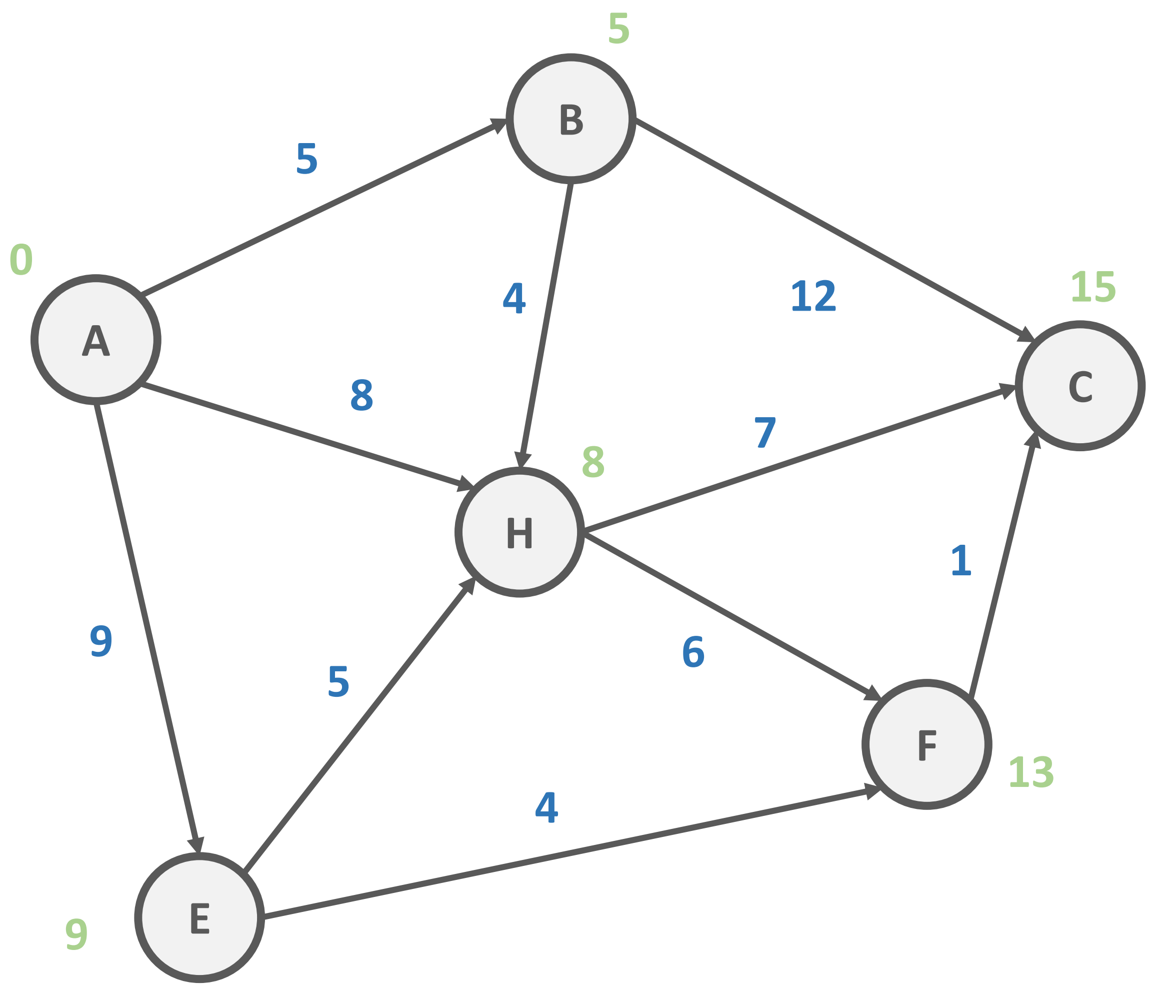
Bellman-Ford Algorithm



Bellman-Ford Algorithm



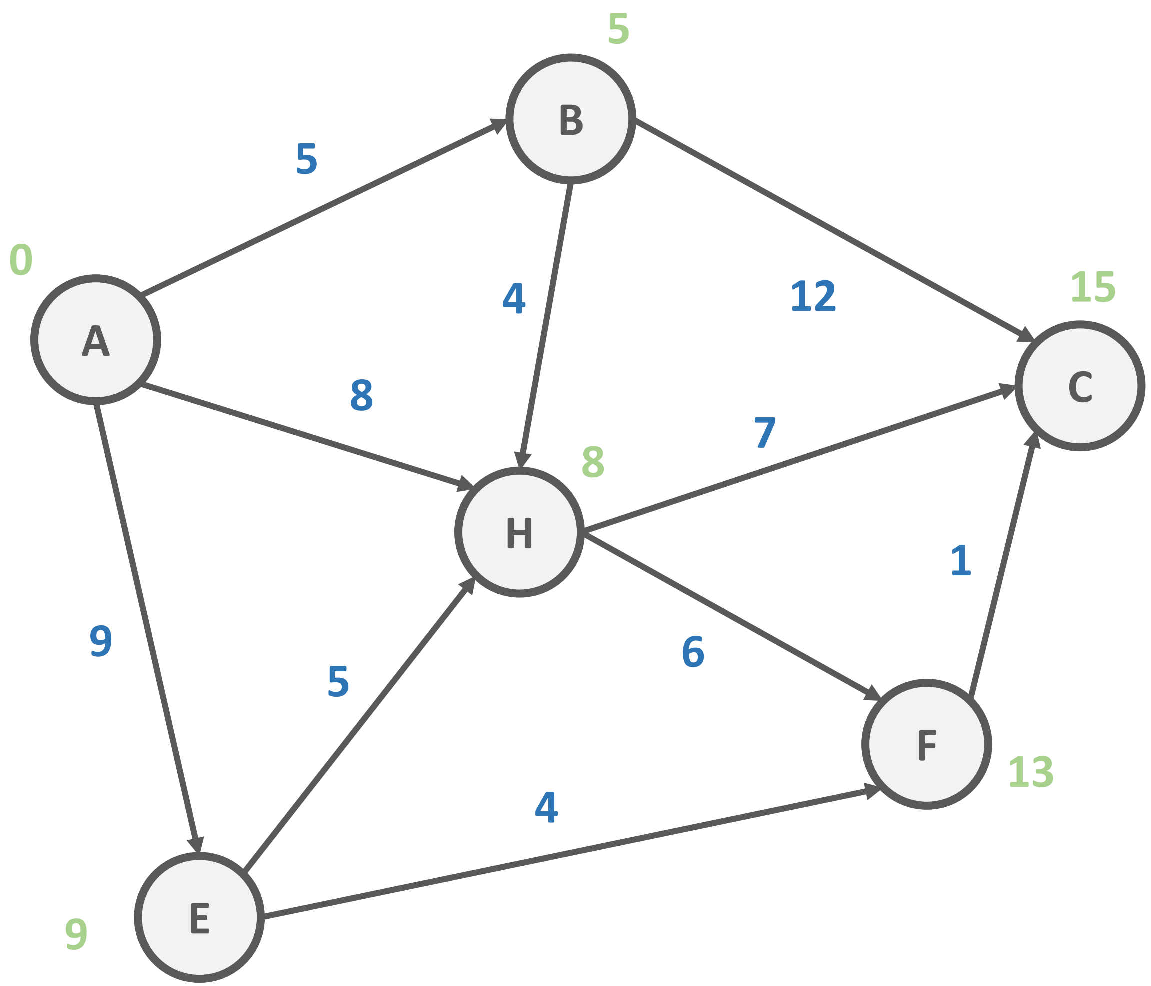
Bellman-Ford Algorithm



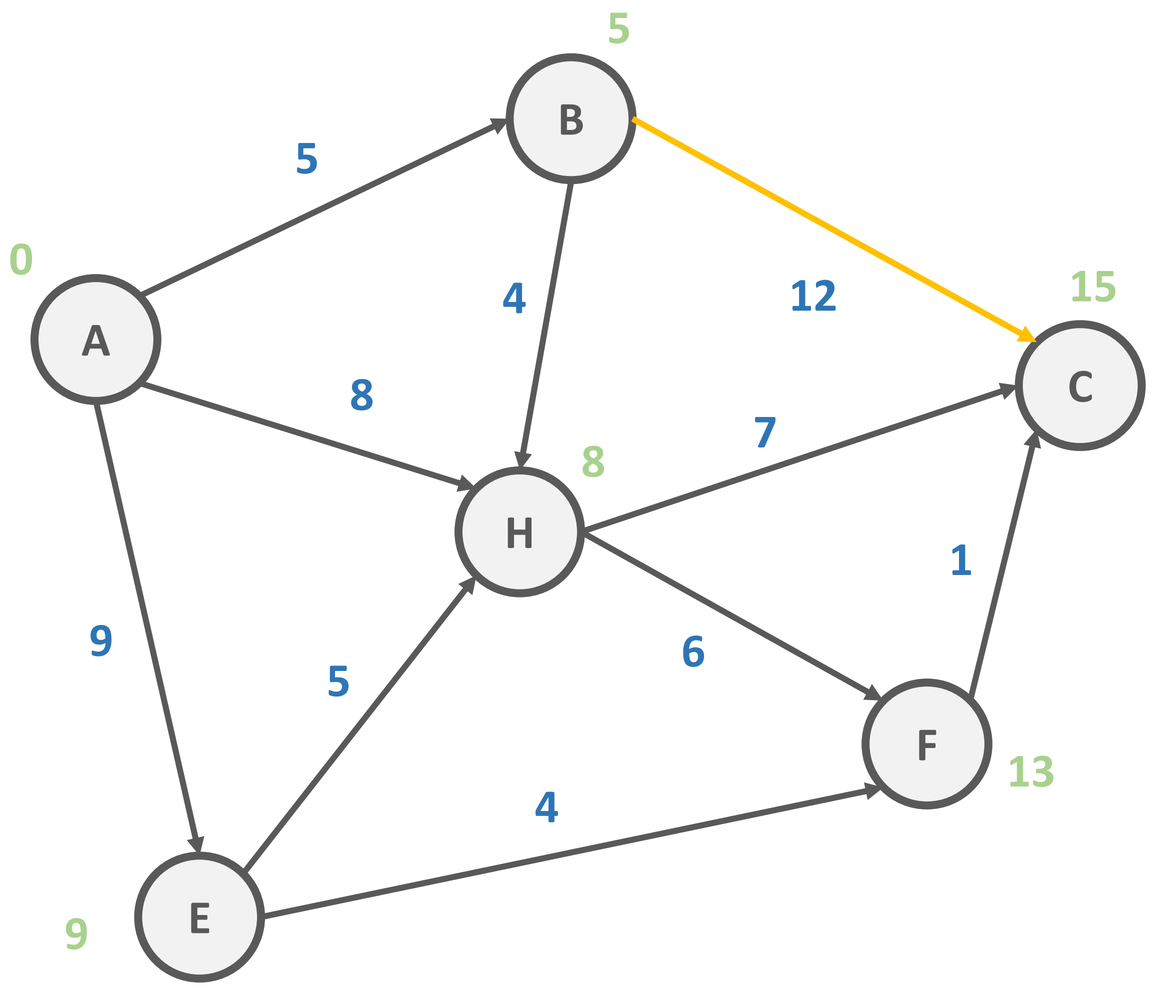
Bellman-Ford Algorithm

ITERATION #3

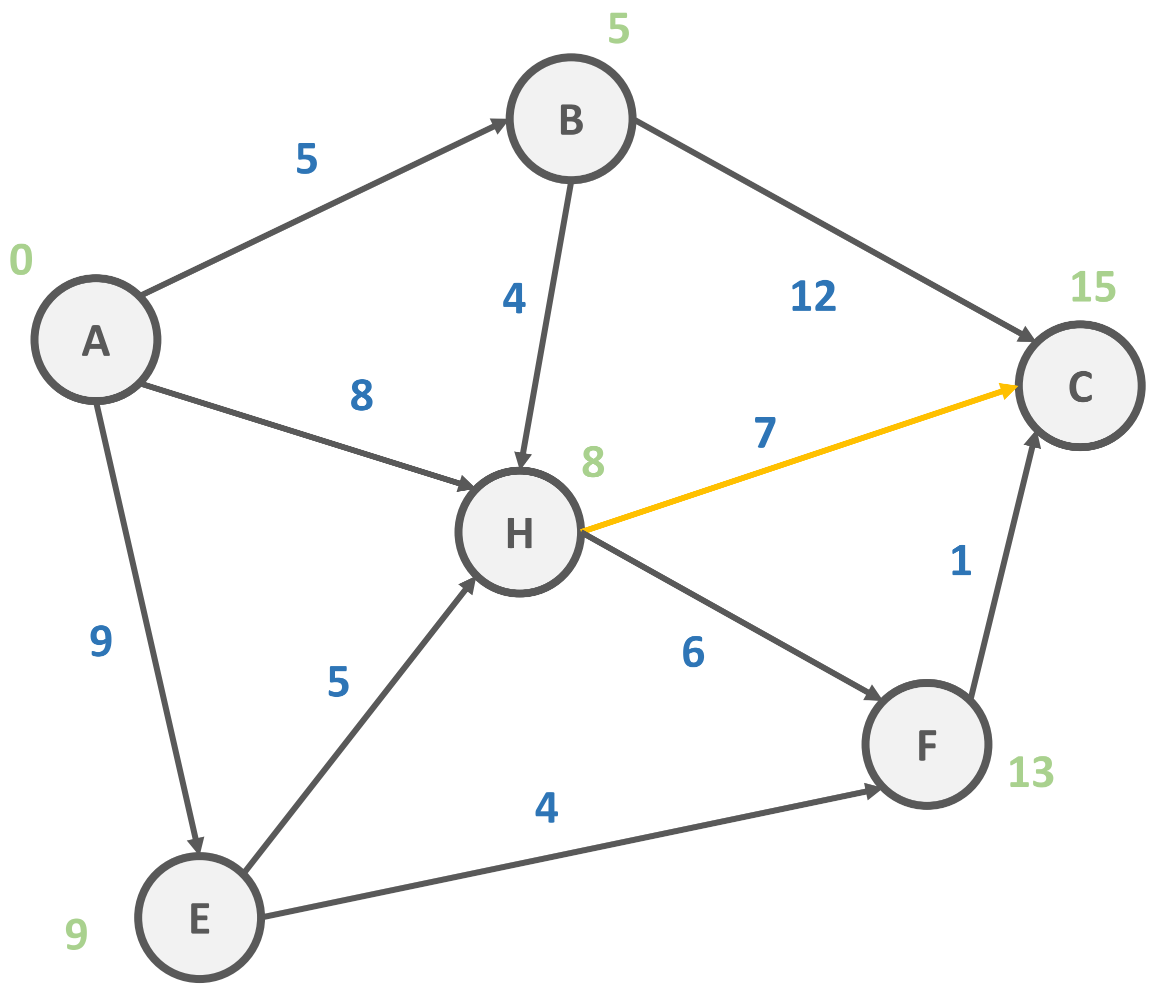
Bellman-Ford Algorithm



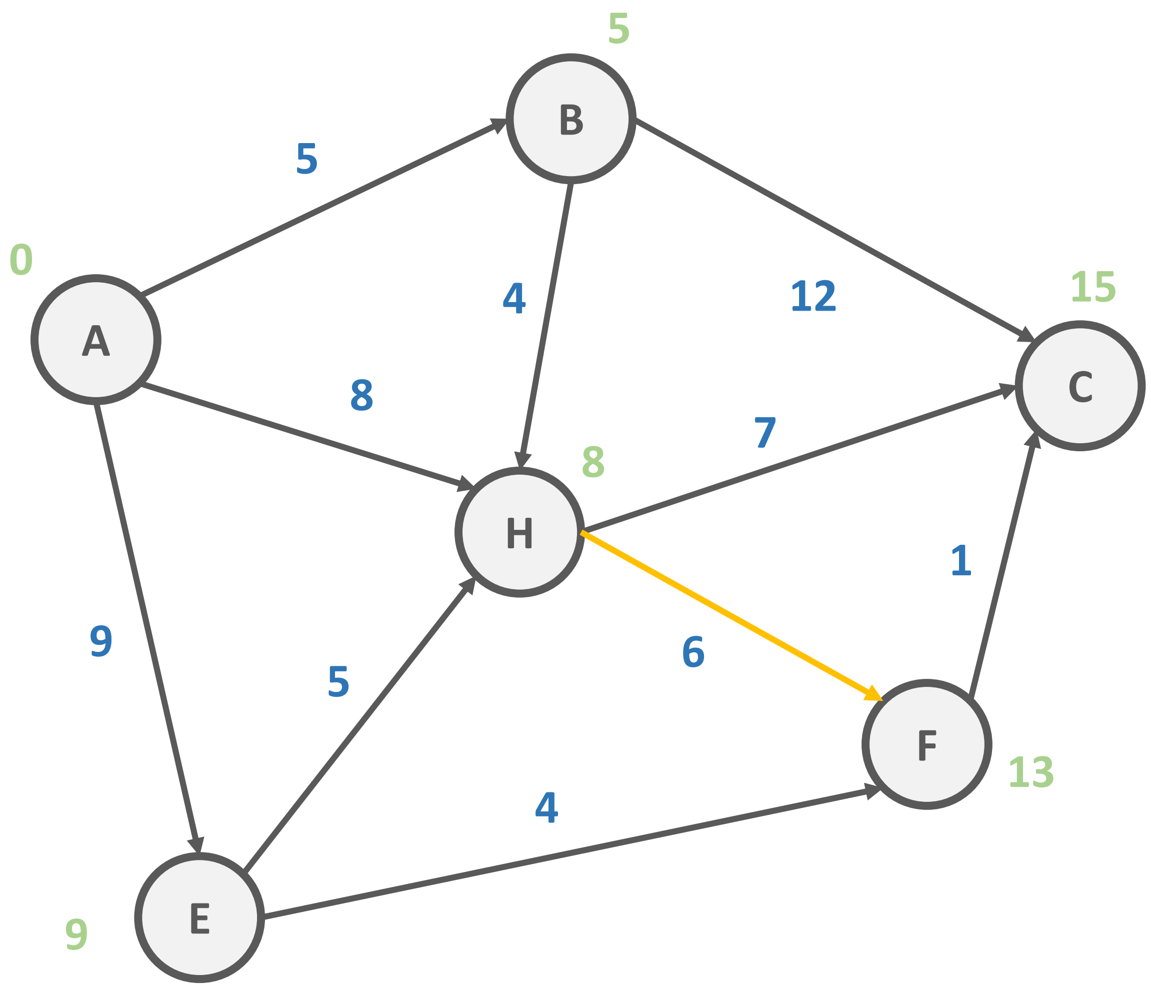
Bellman-Ford Algorithm



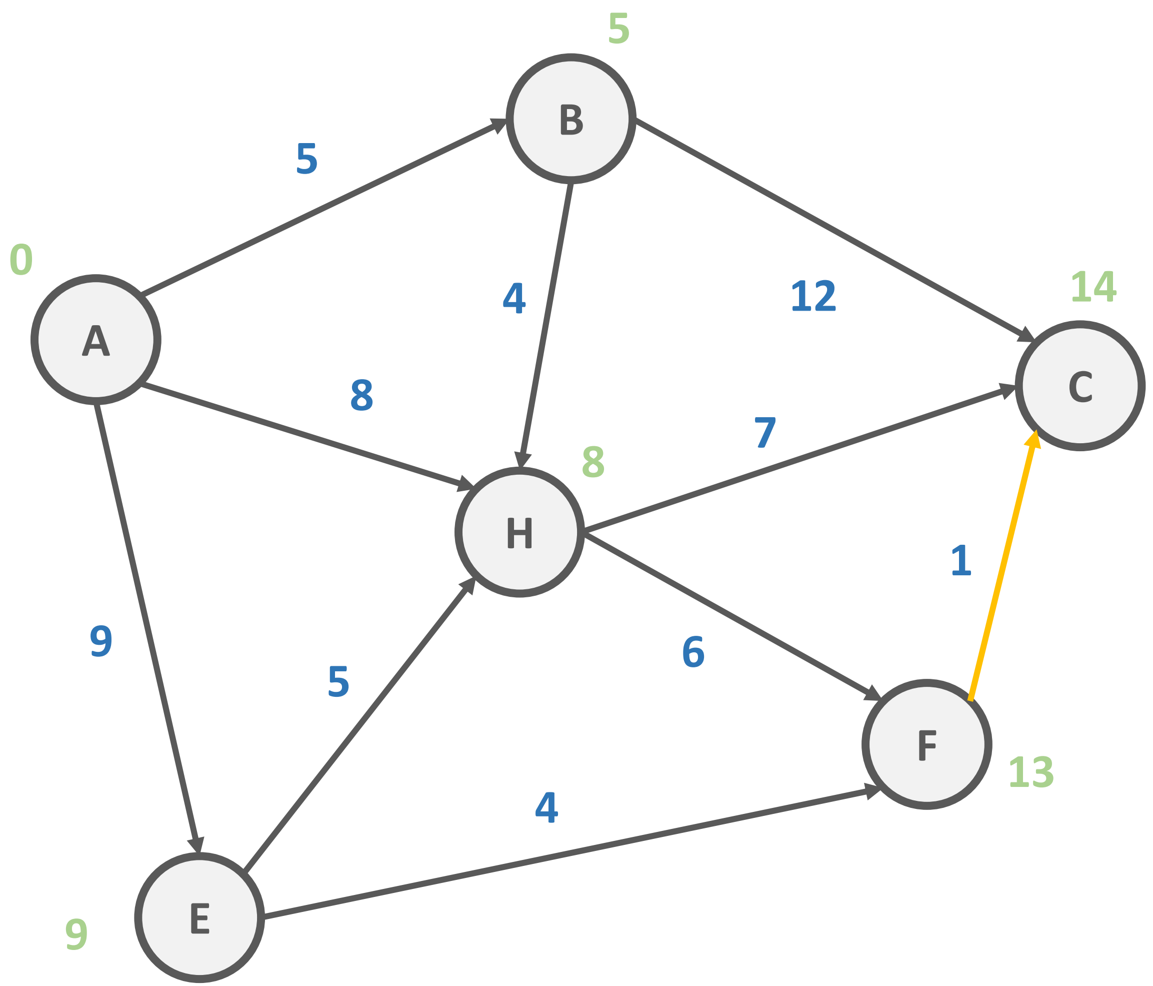
Bellman-Ford Algorithm



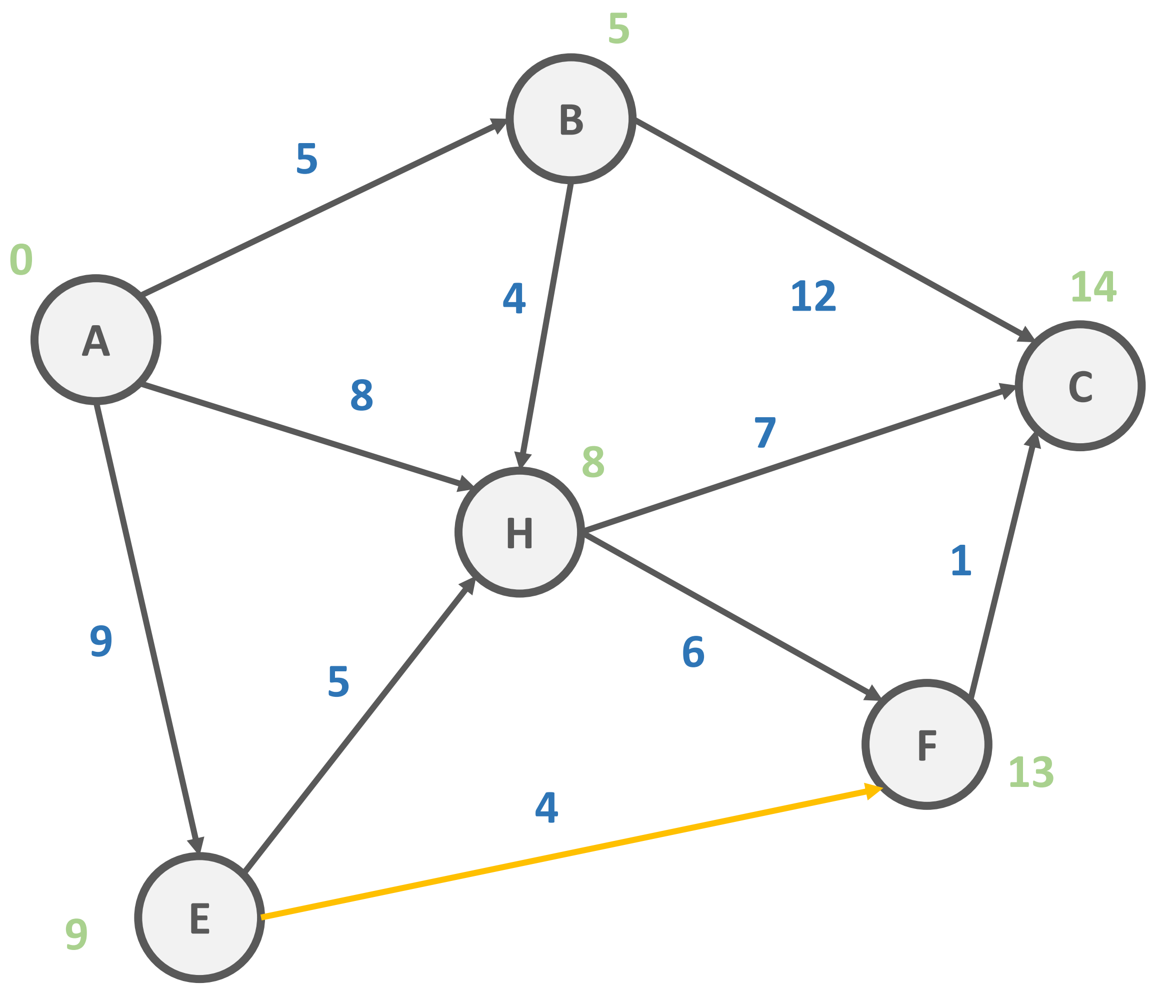
Bellman-Ford Algorithm



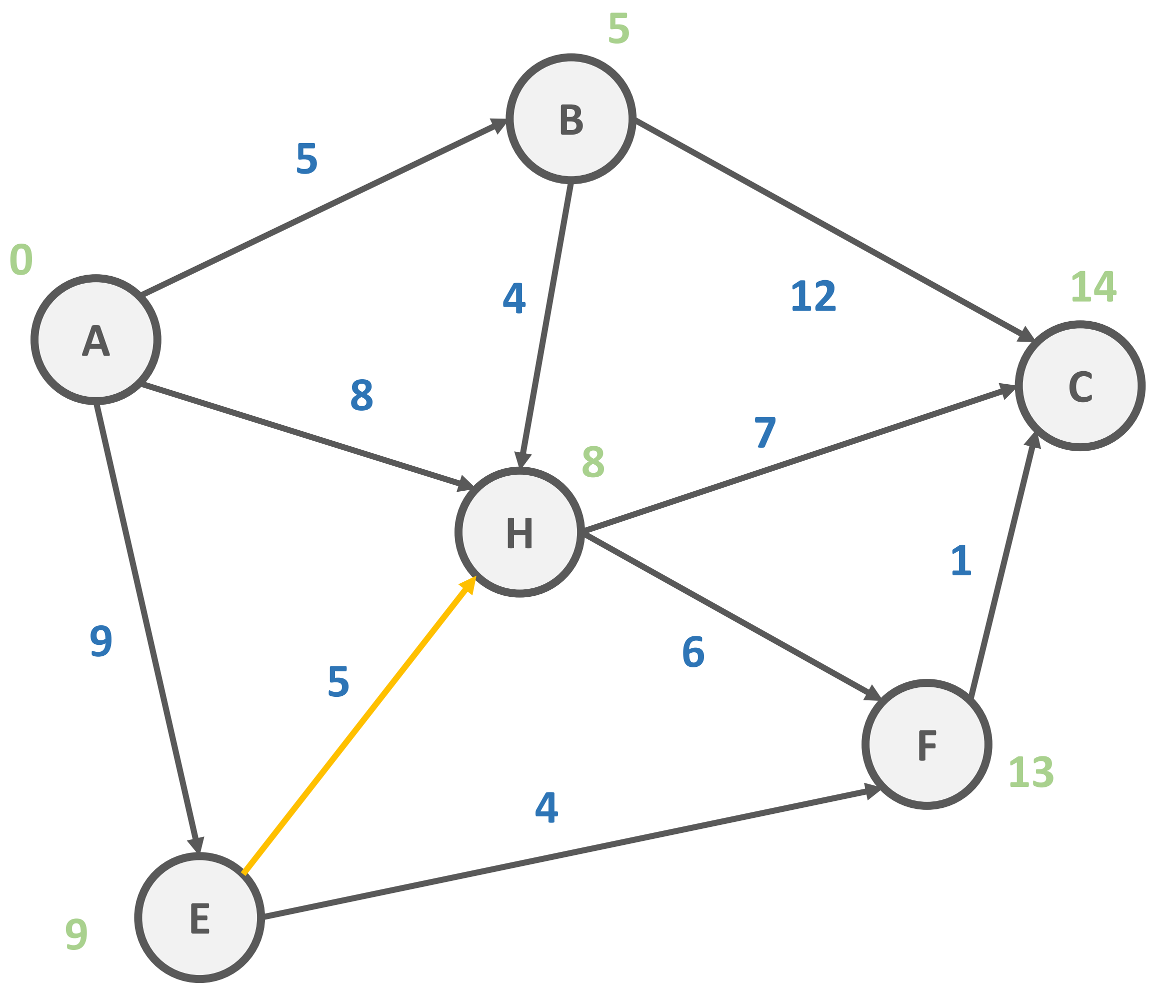
Bellman-Ford Algorithm



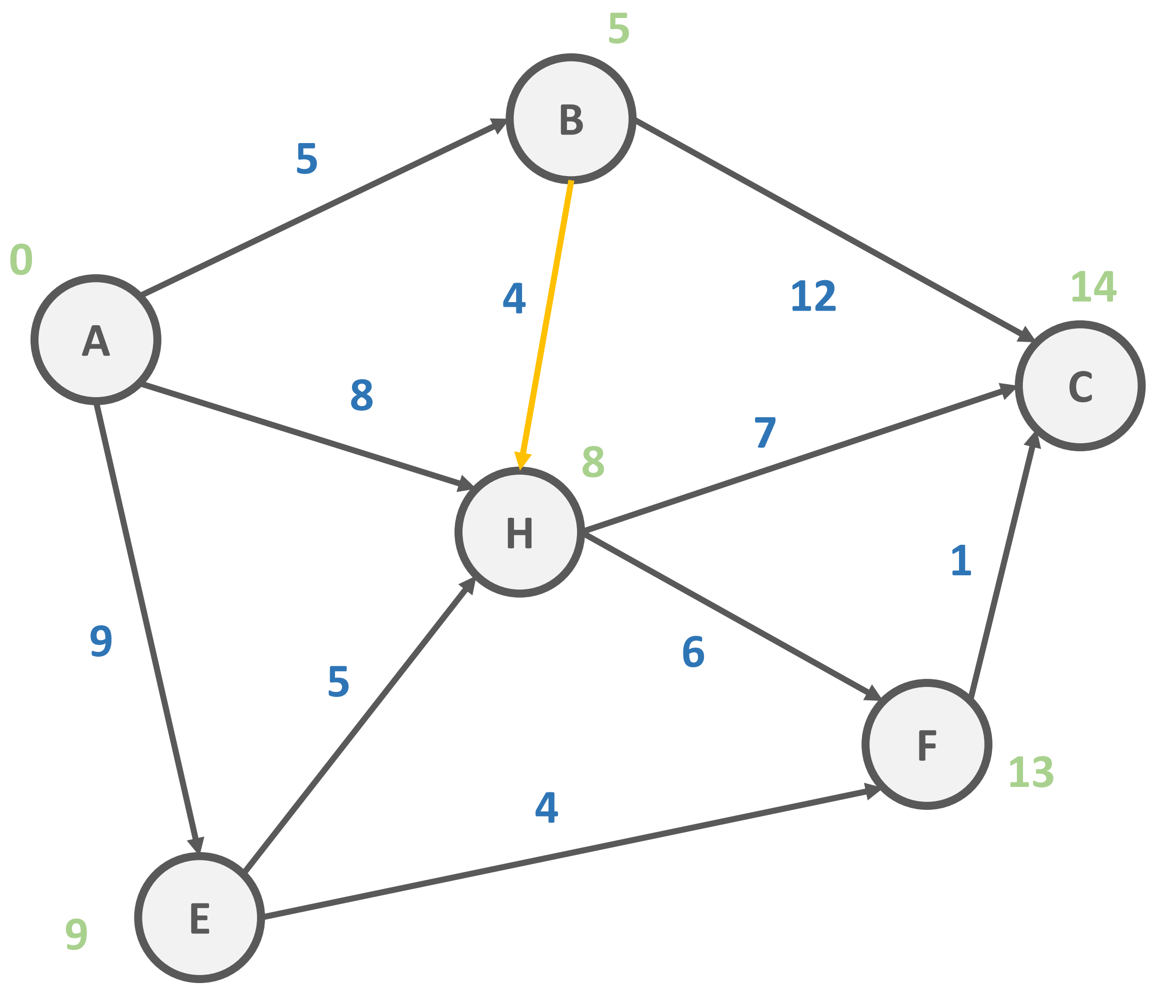
Bellman-Ford Algorithm



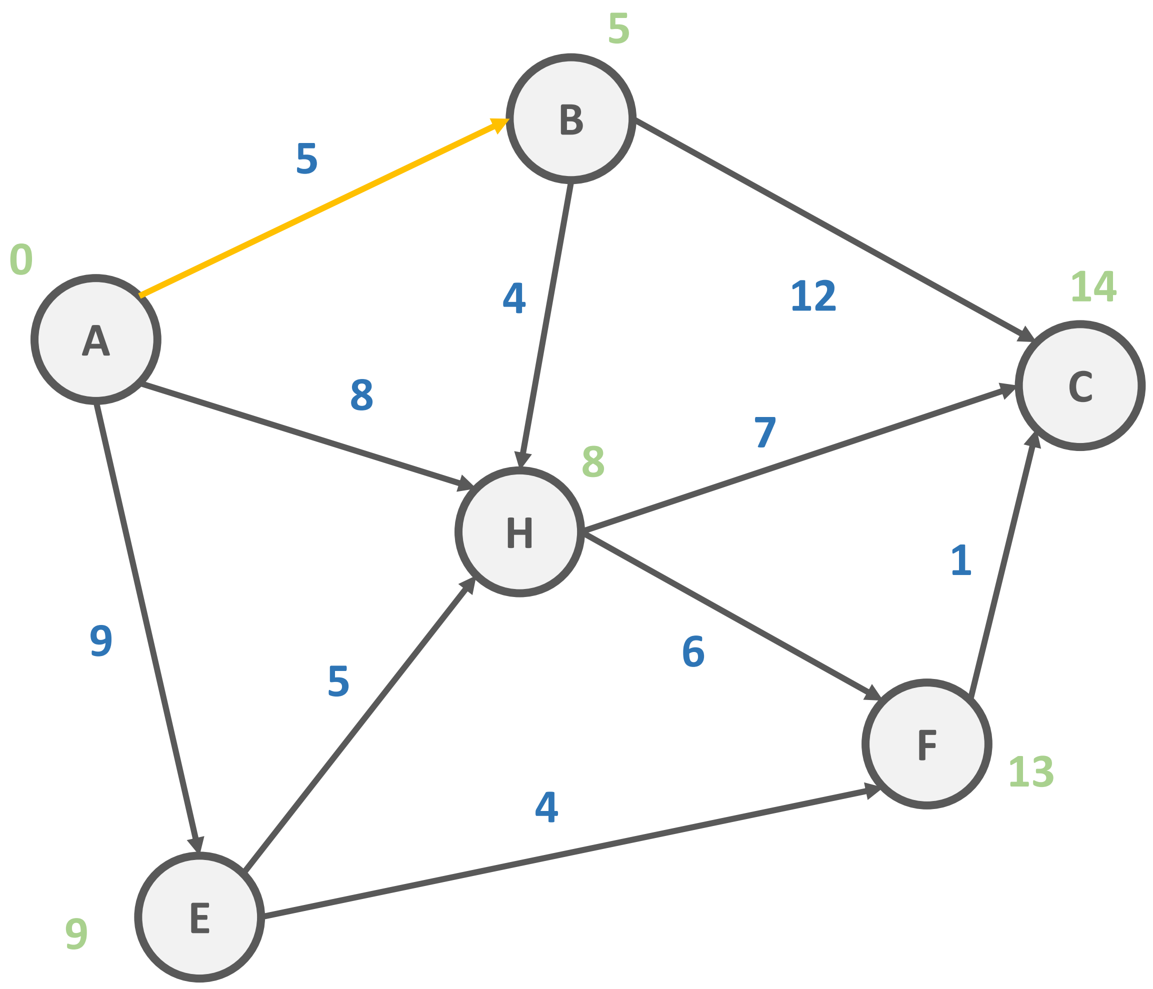
Bellman-Ford Algorithm



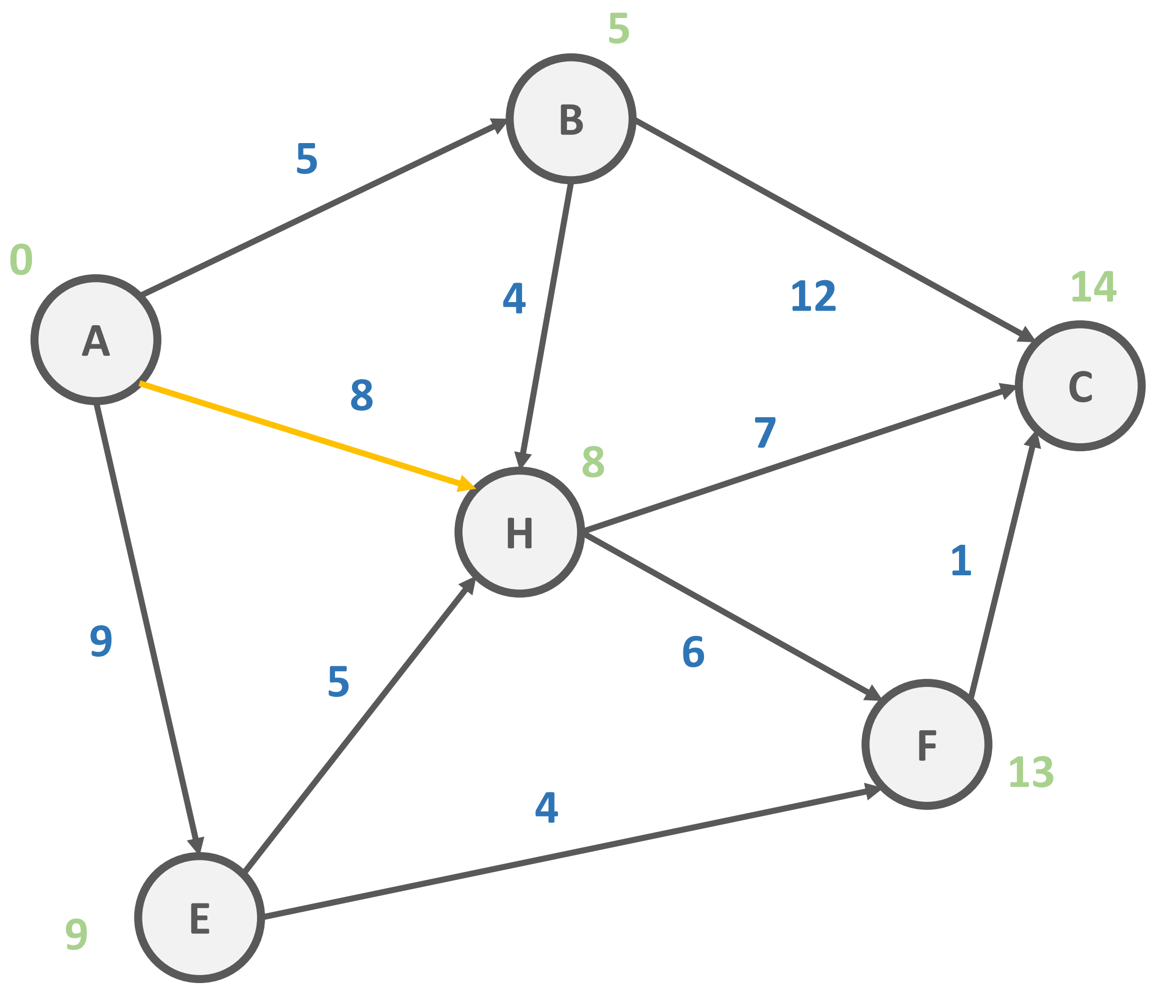
Bellman-Ford Algorithm



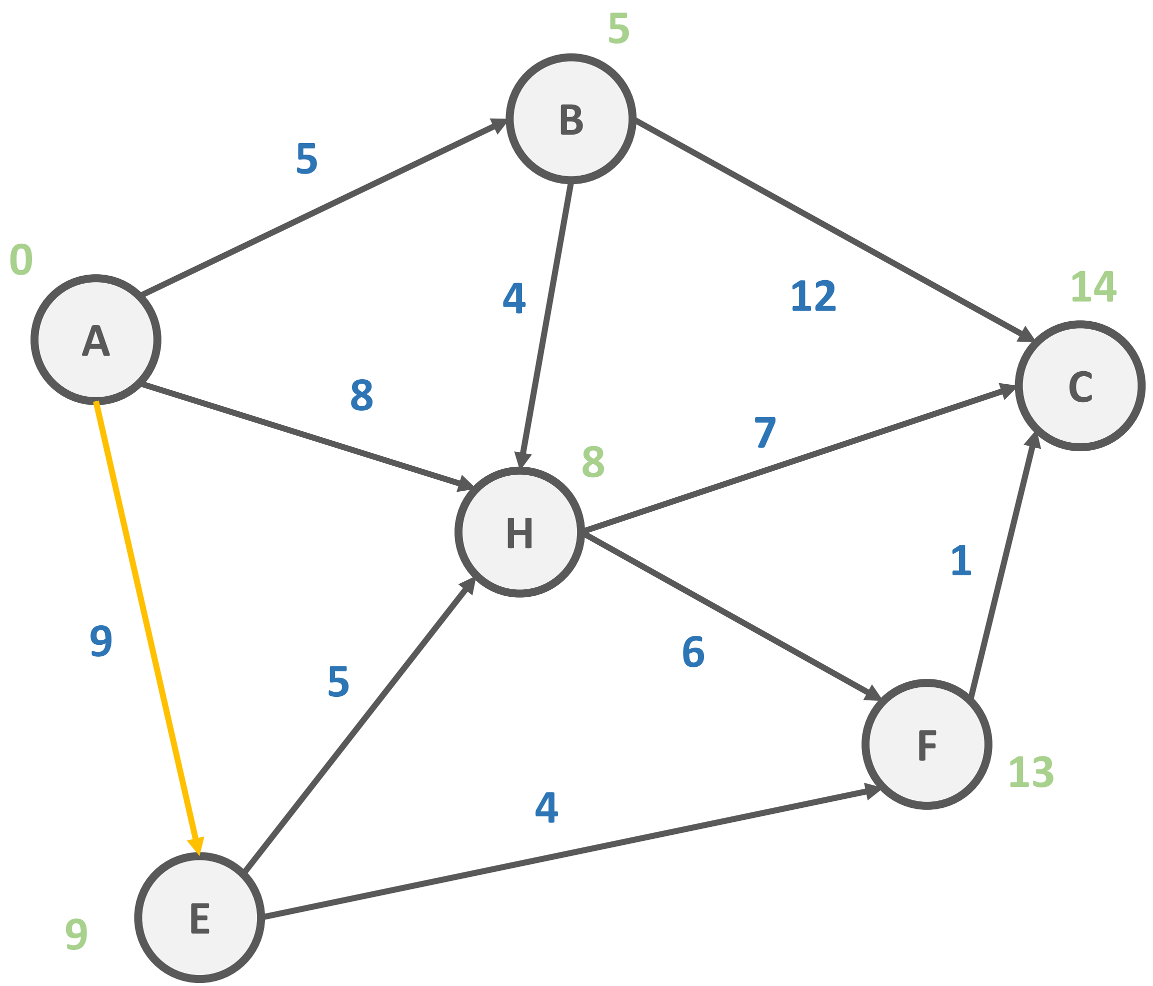
Bellman-Ford Algorithm



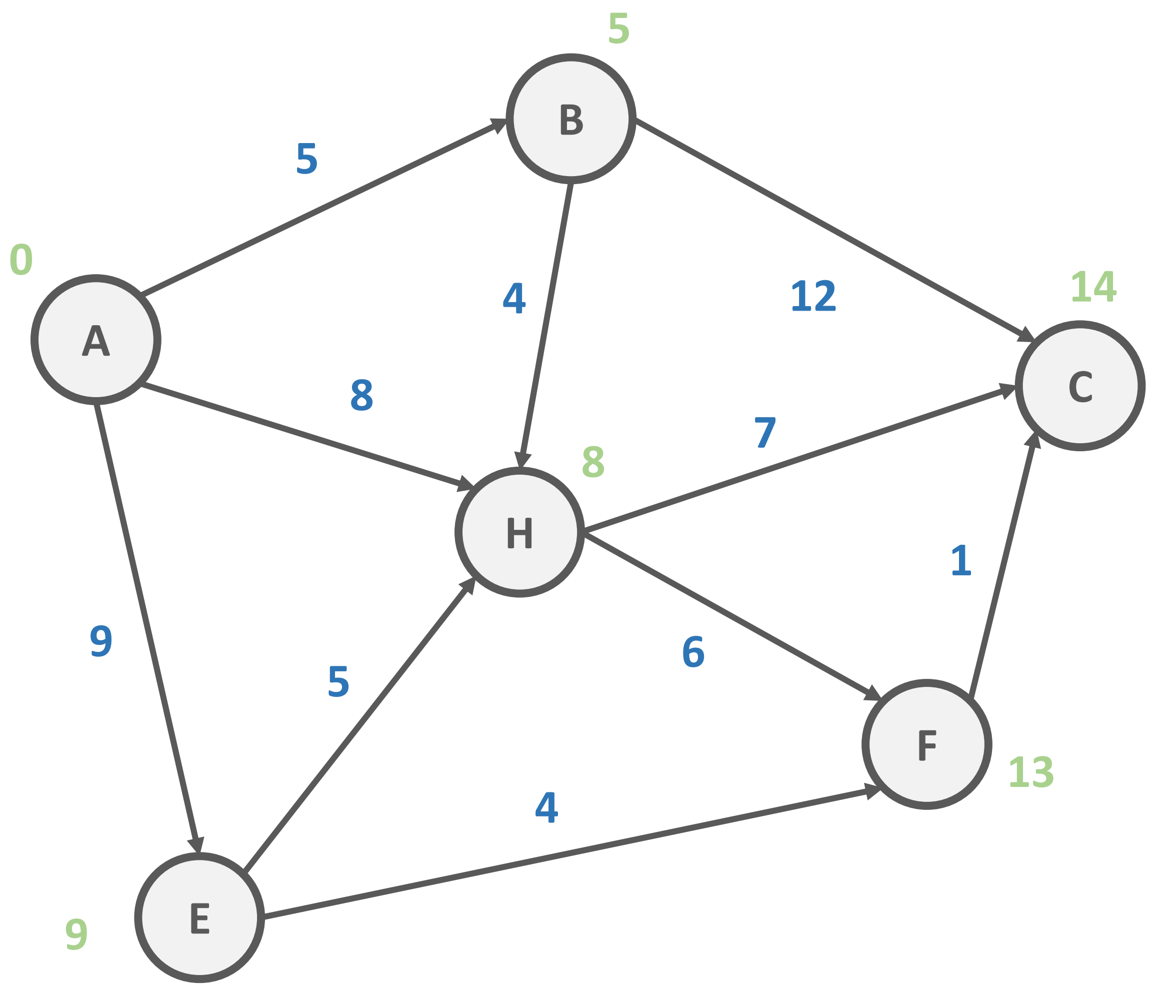
Bellman-Ford Algorithm



Bellman-Ford Algorithm



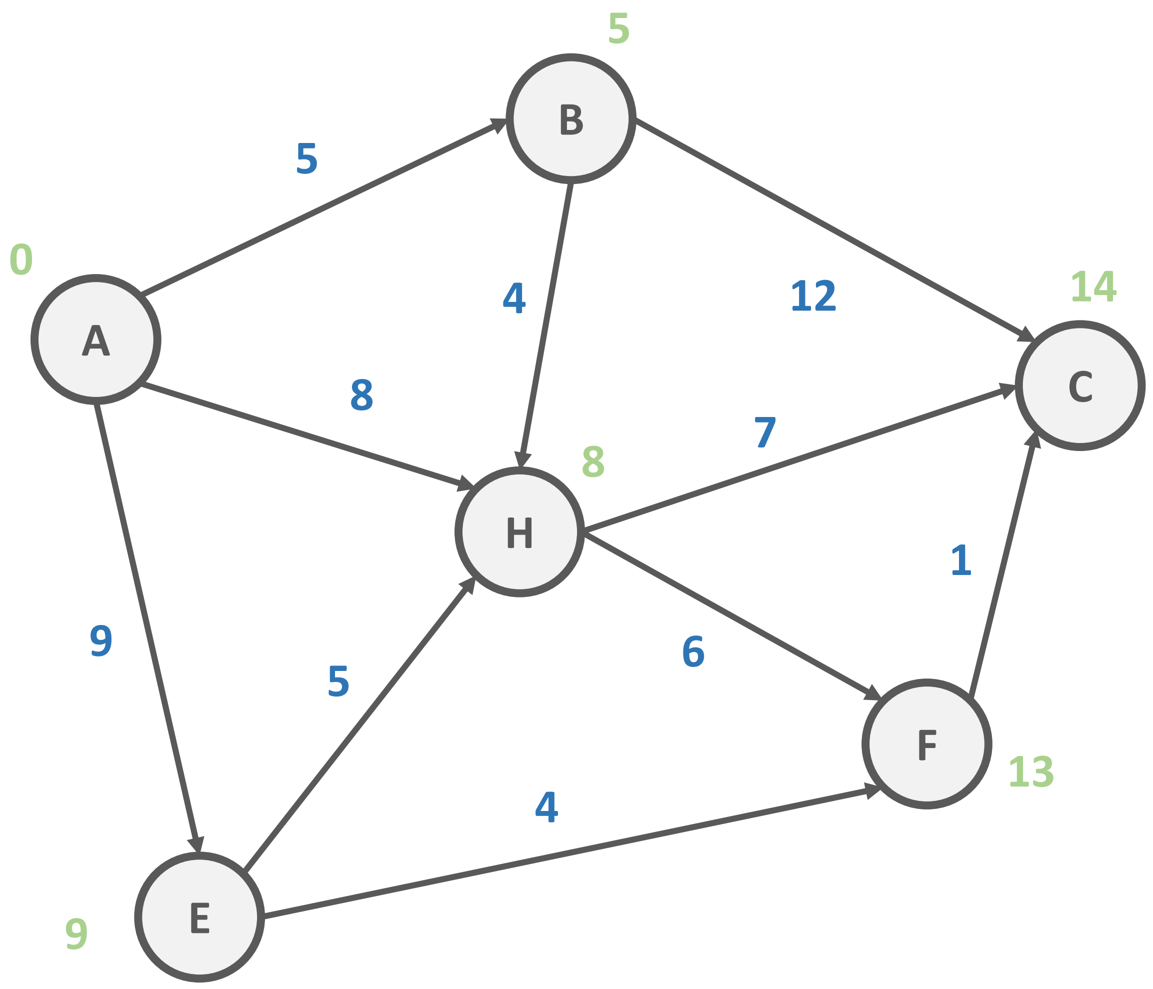
Bellman-Ford Algorithm



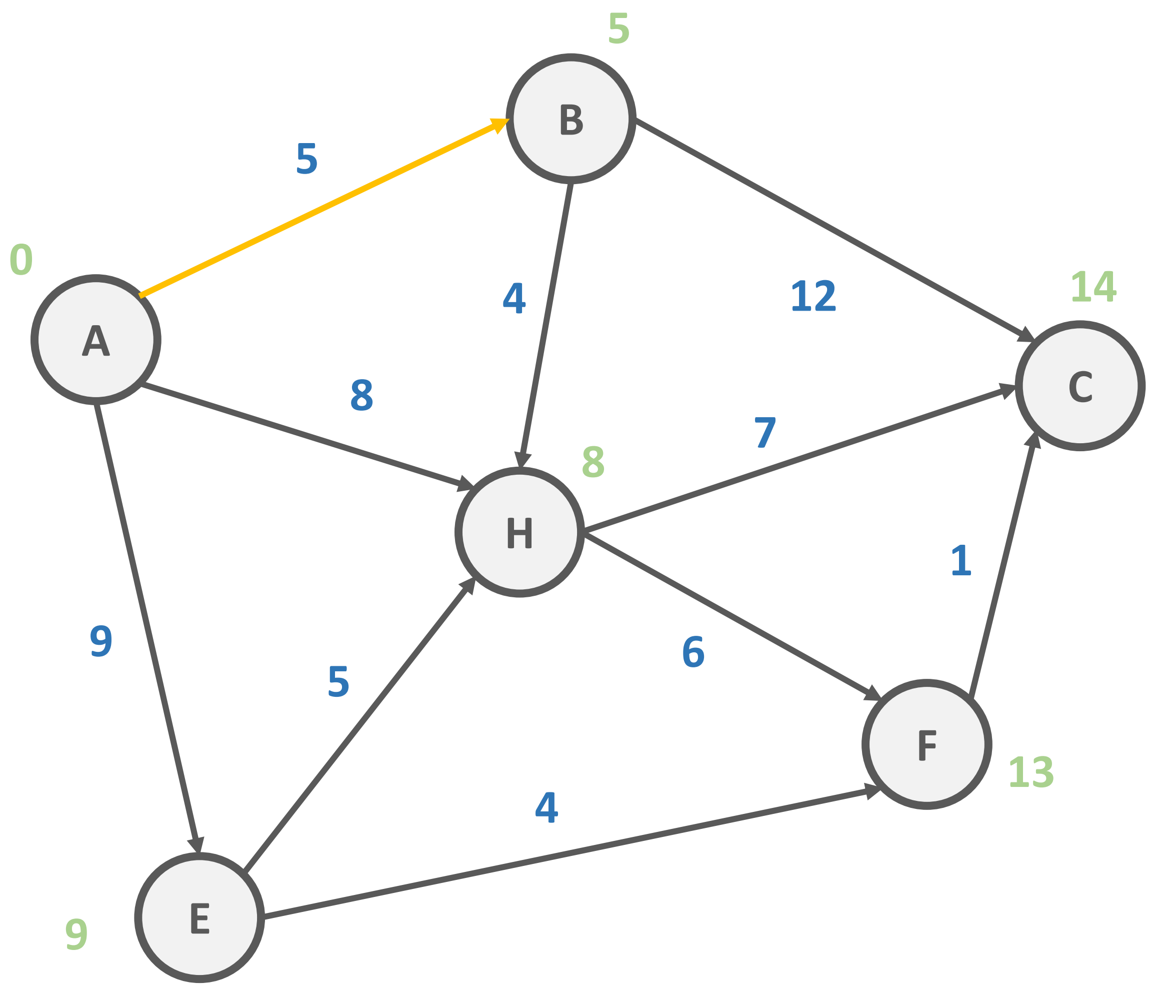
Bellman-Ford Algorithm

ITERATION #4

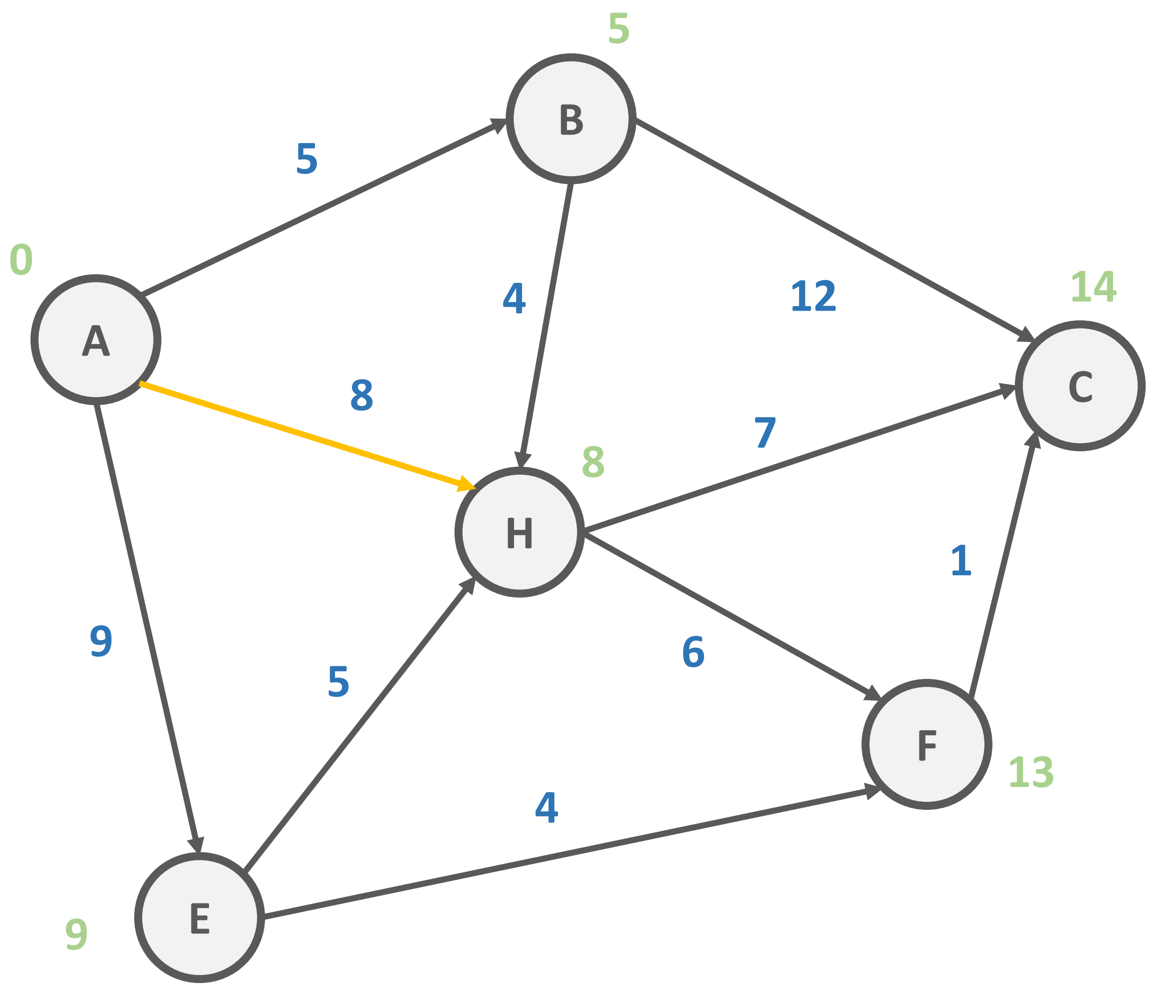
Bellman-Ford Algorithm



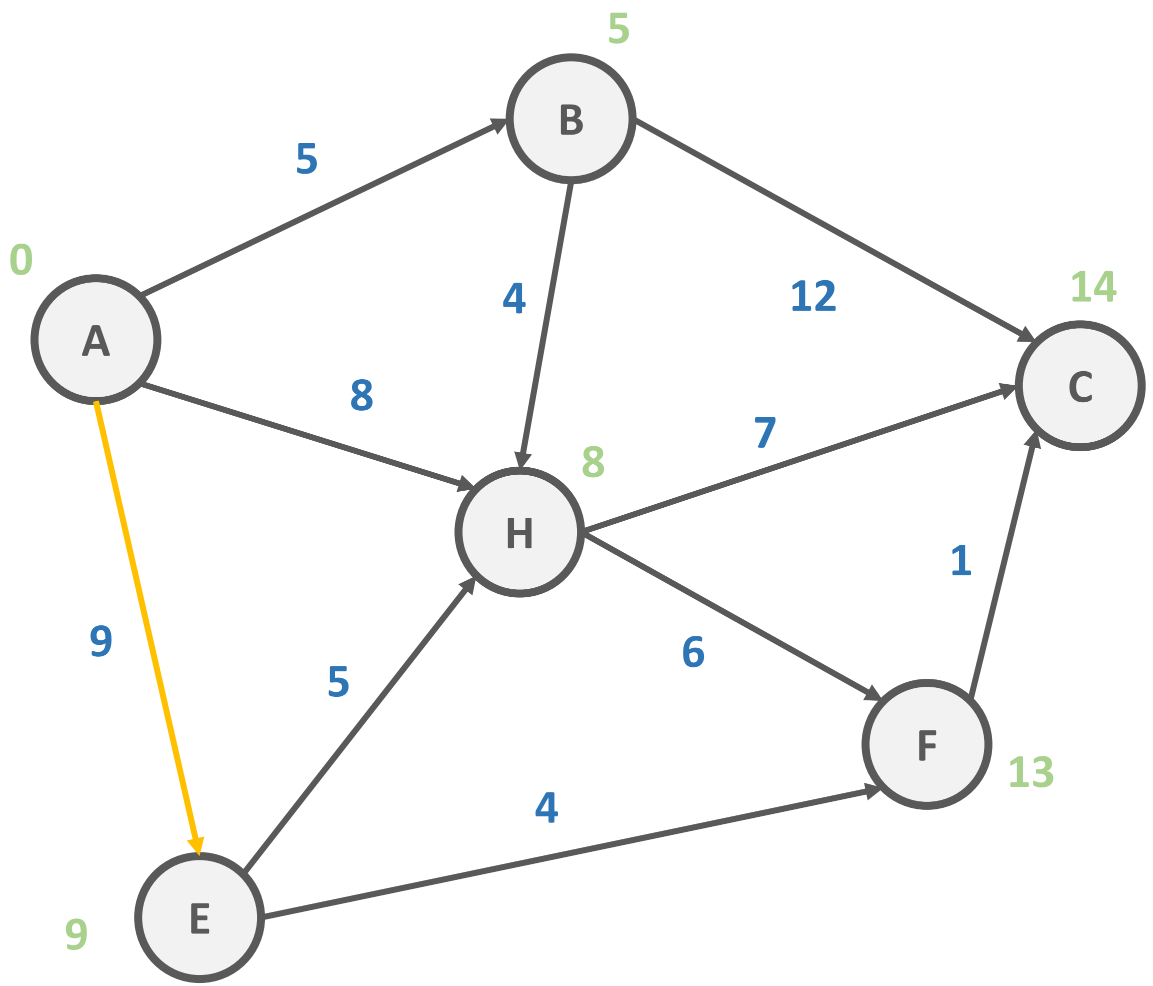
Bellman-Ford Algorithm



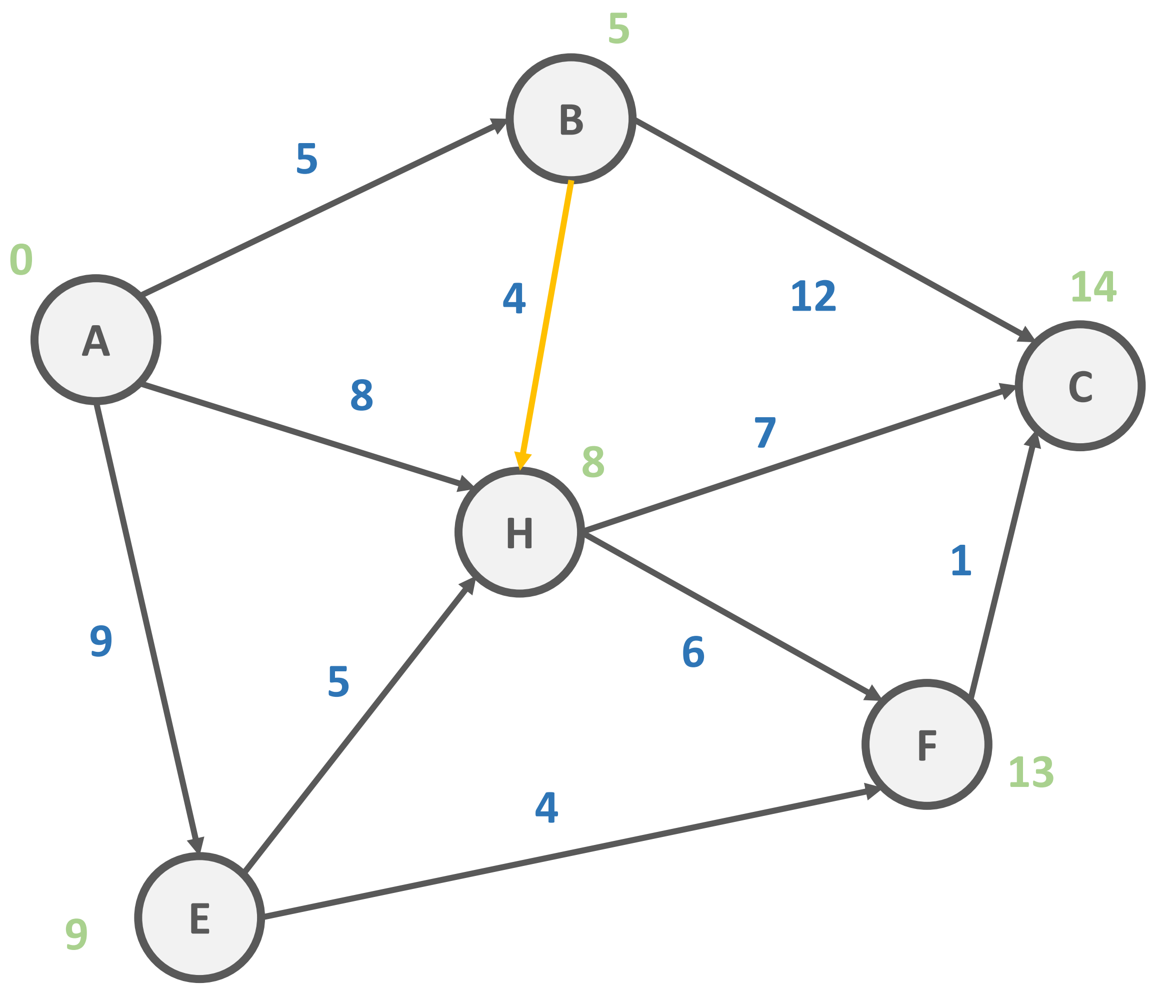
Bellman-Ford Algorithm



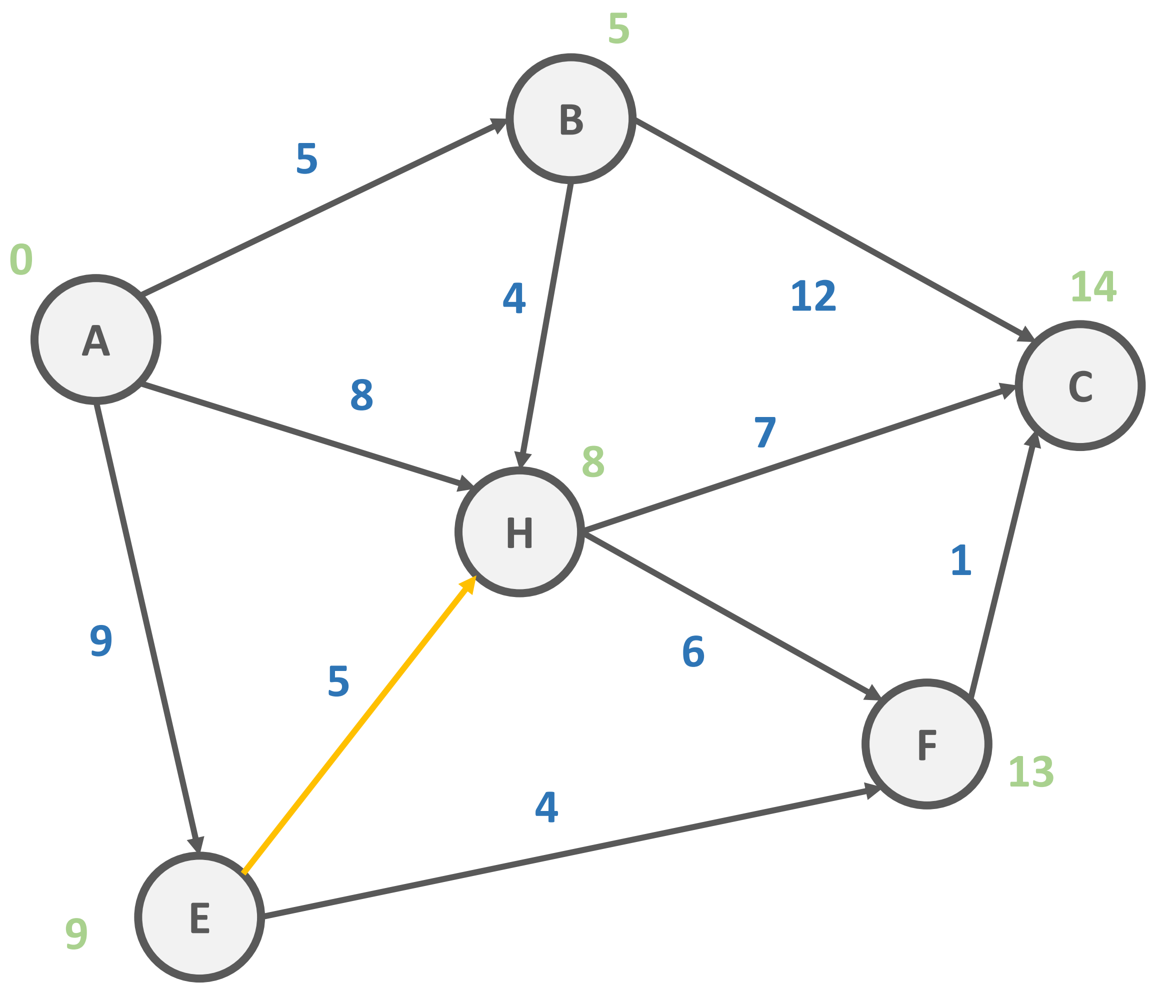
Bellman-Ford Algorithm



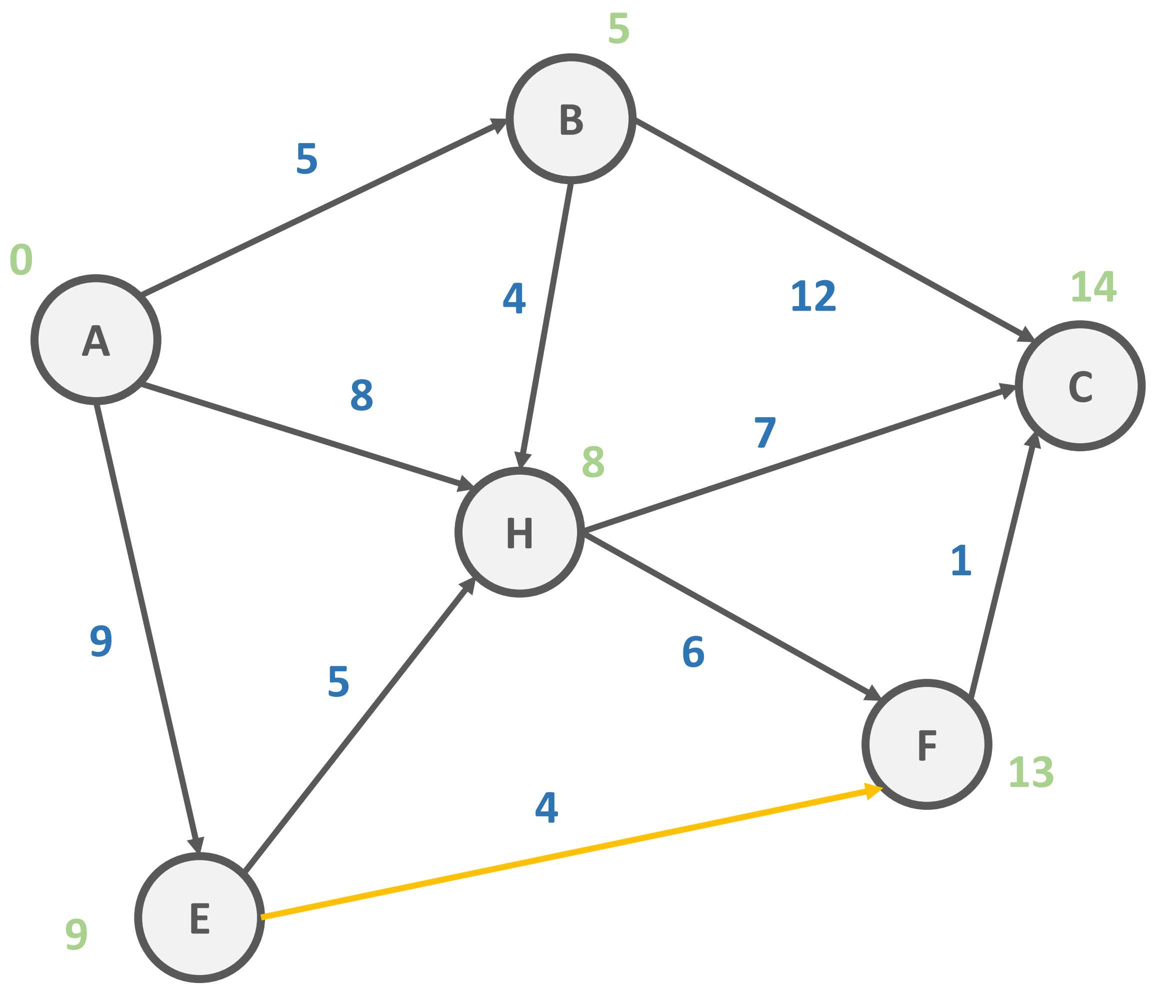
Bellman-Ford Algorithm



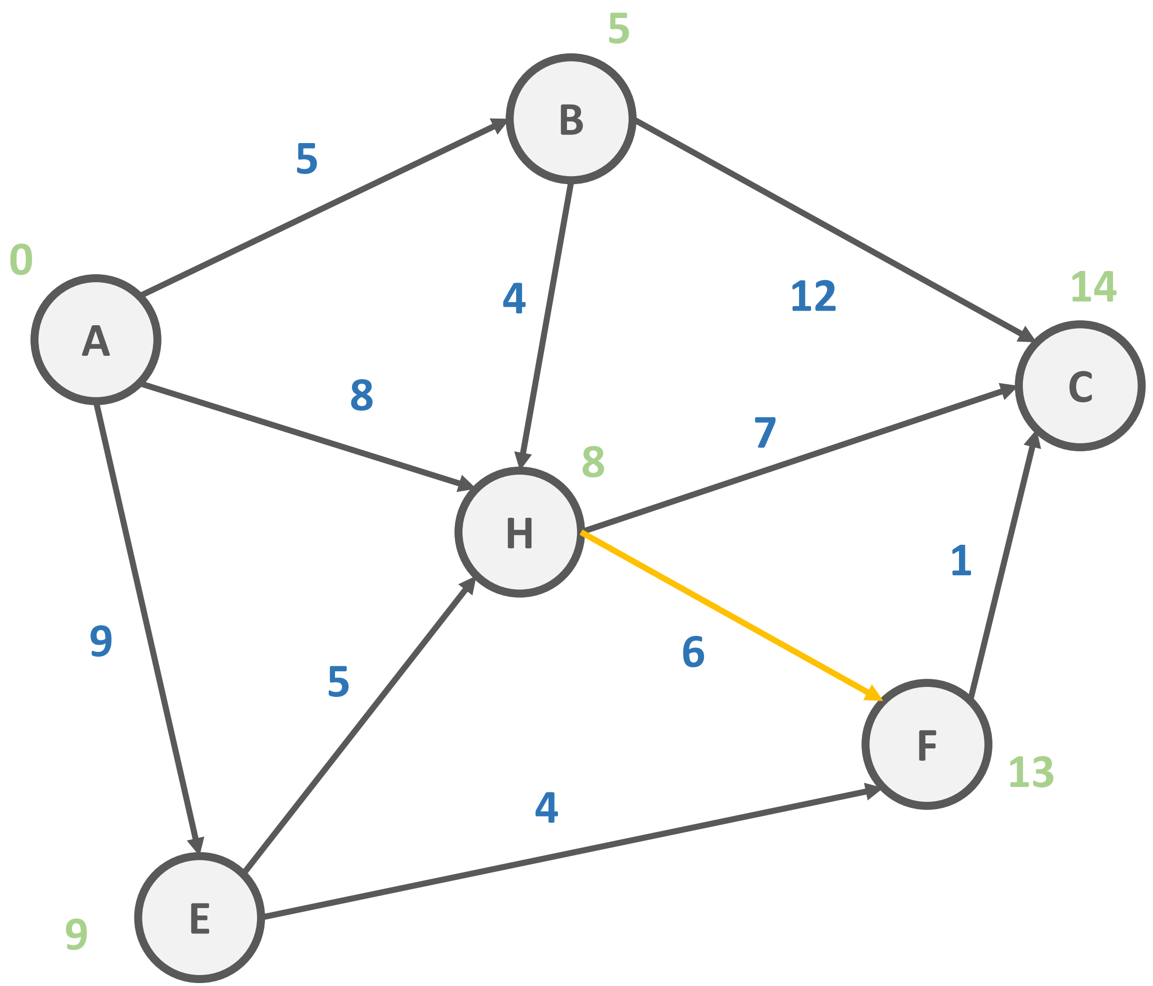
Bellman-Ford Algorithm



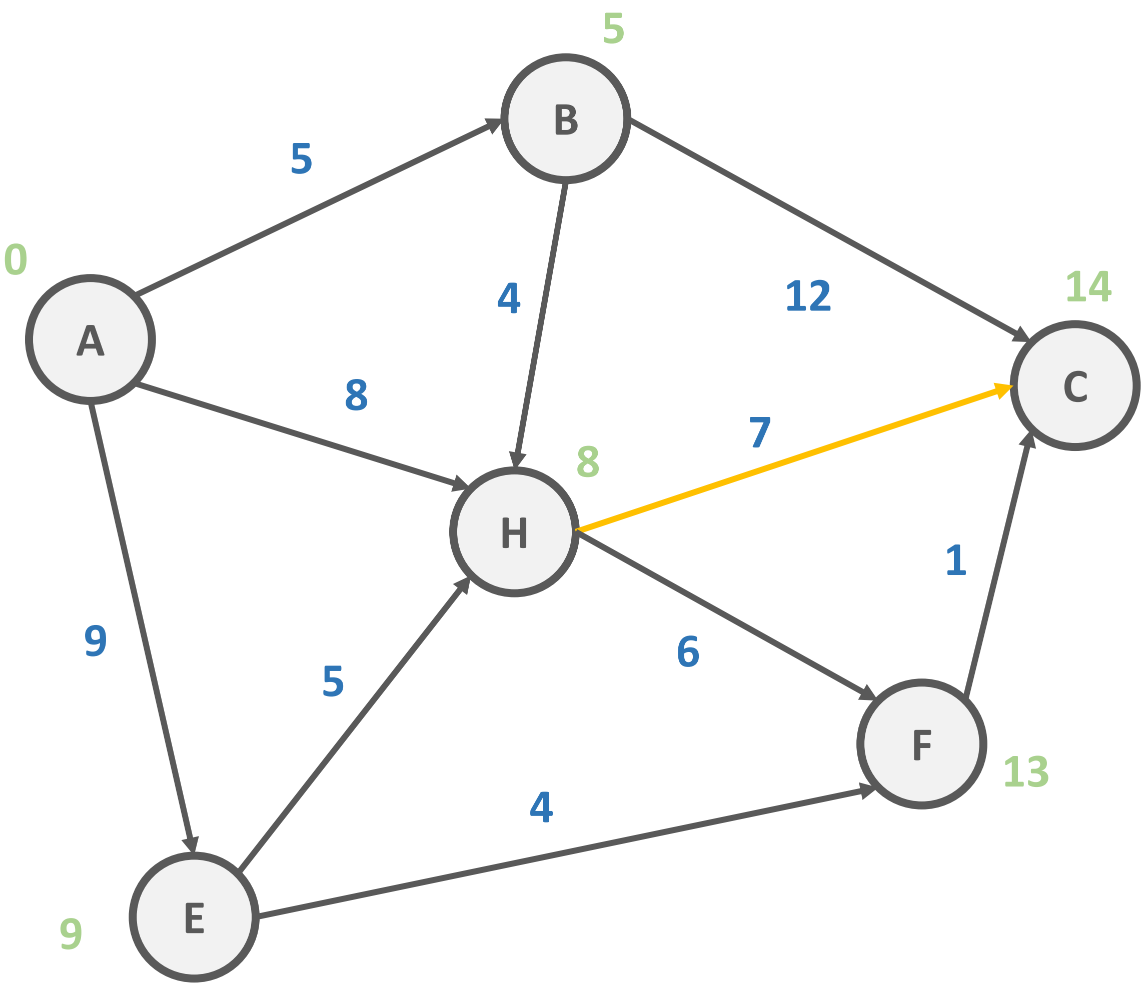
Bellman-Ford Algorithm



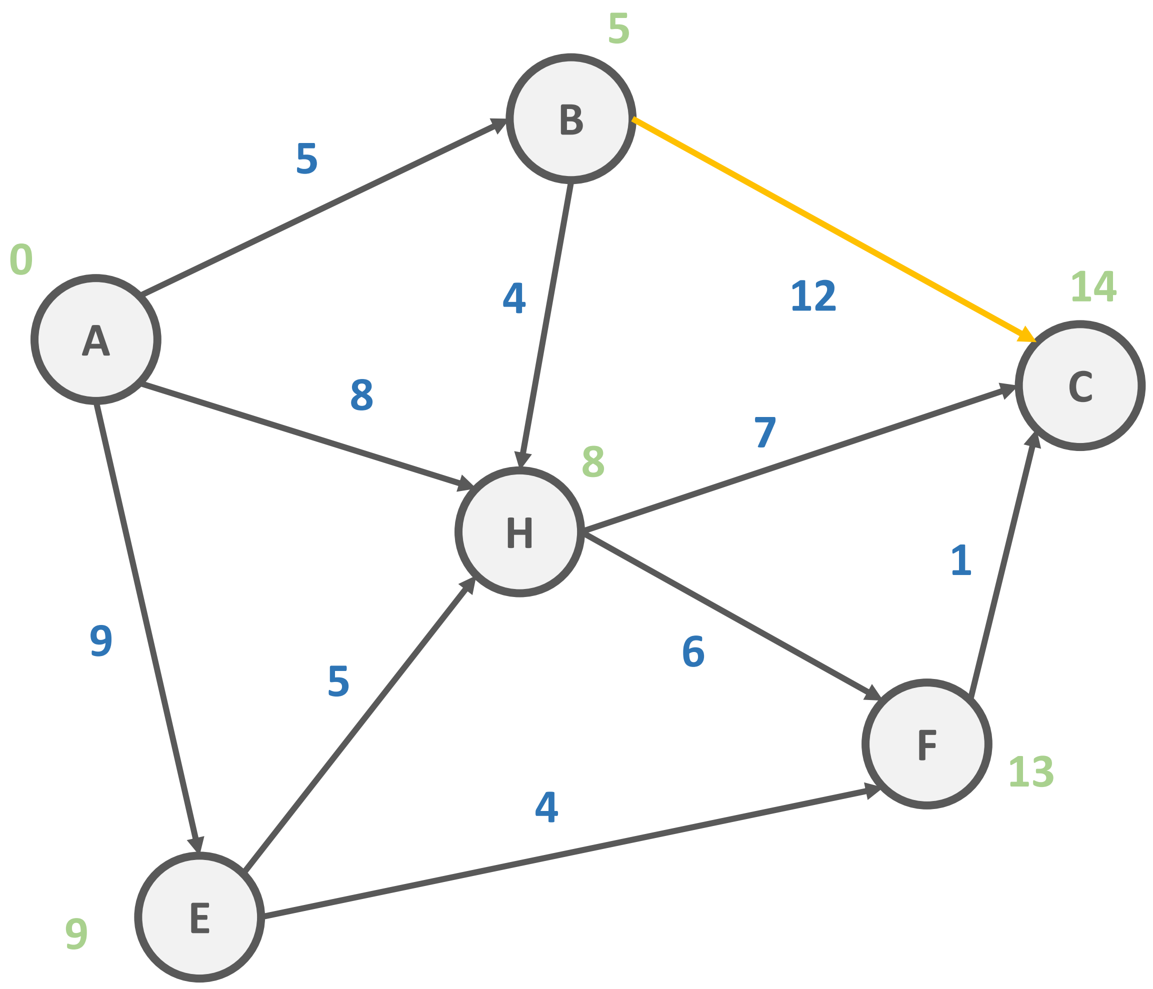
Bellman-Ford Algorithm



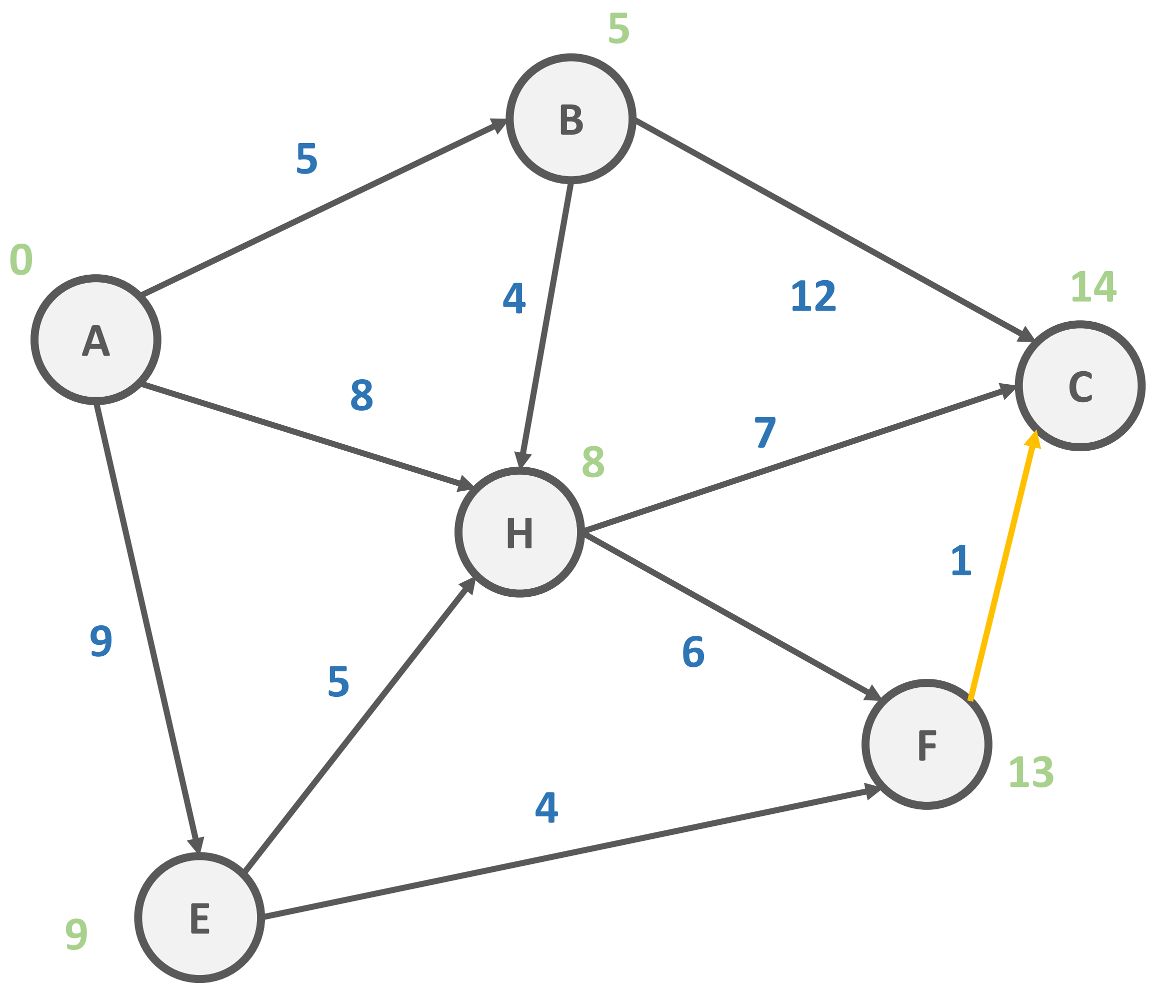
Bellman-Ford Algorithm



Bellman-Ford Algorithm



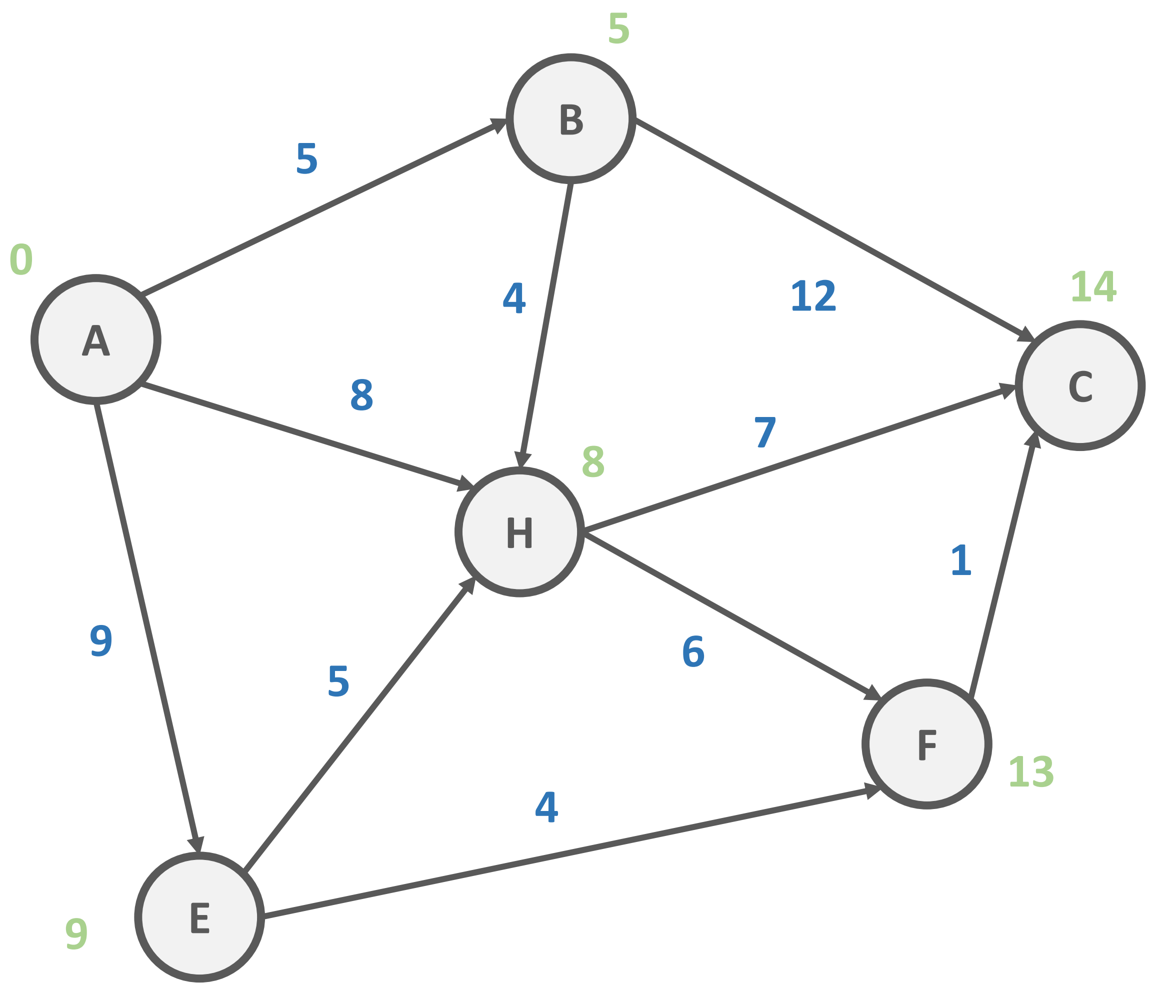
Bellman-Ford Algorithm



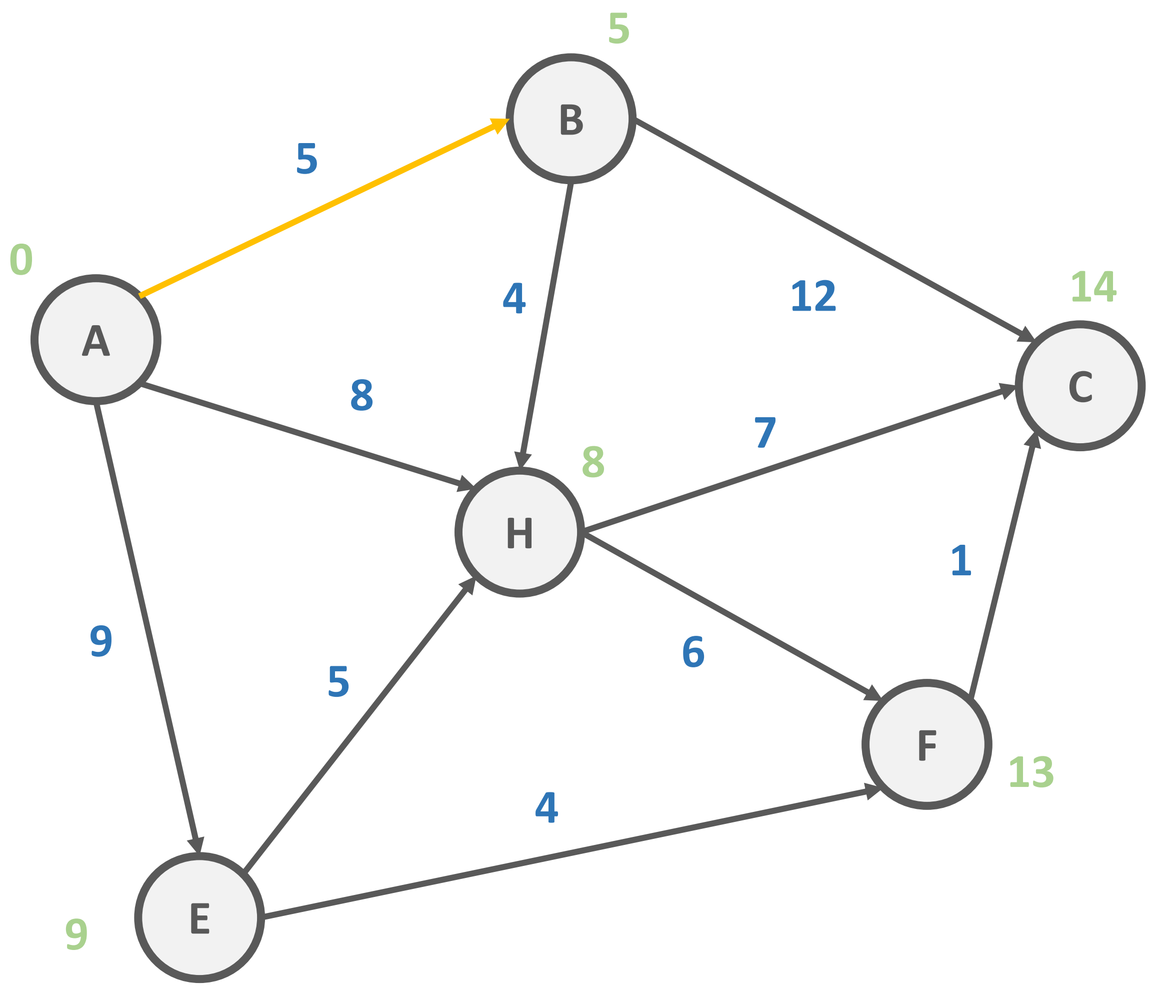
Bellman-Ford Algorithm

ITERATION #5

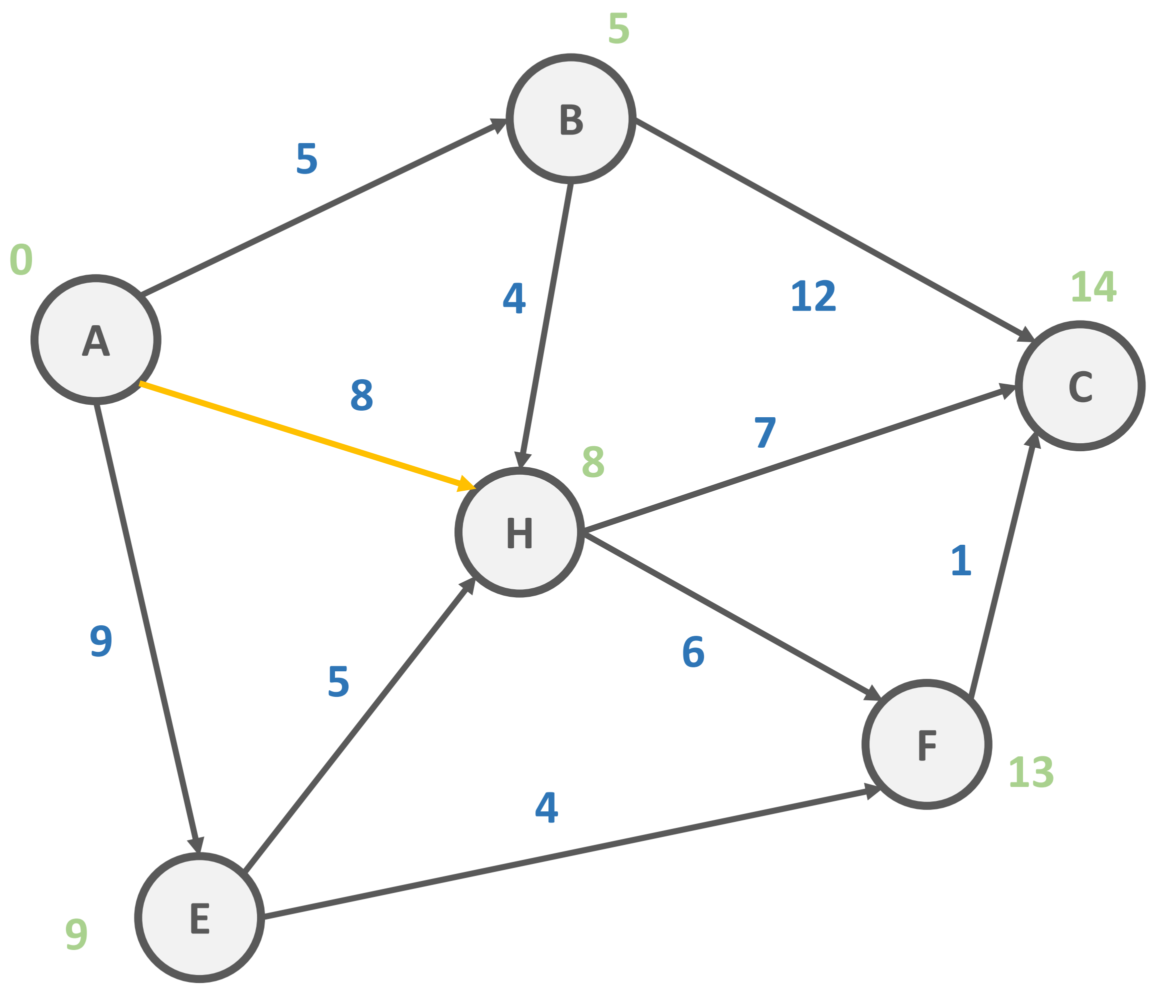
Bellman-Ford Algorithm



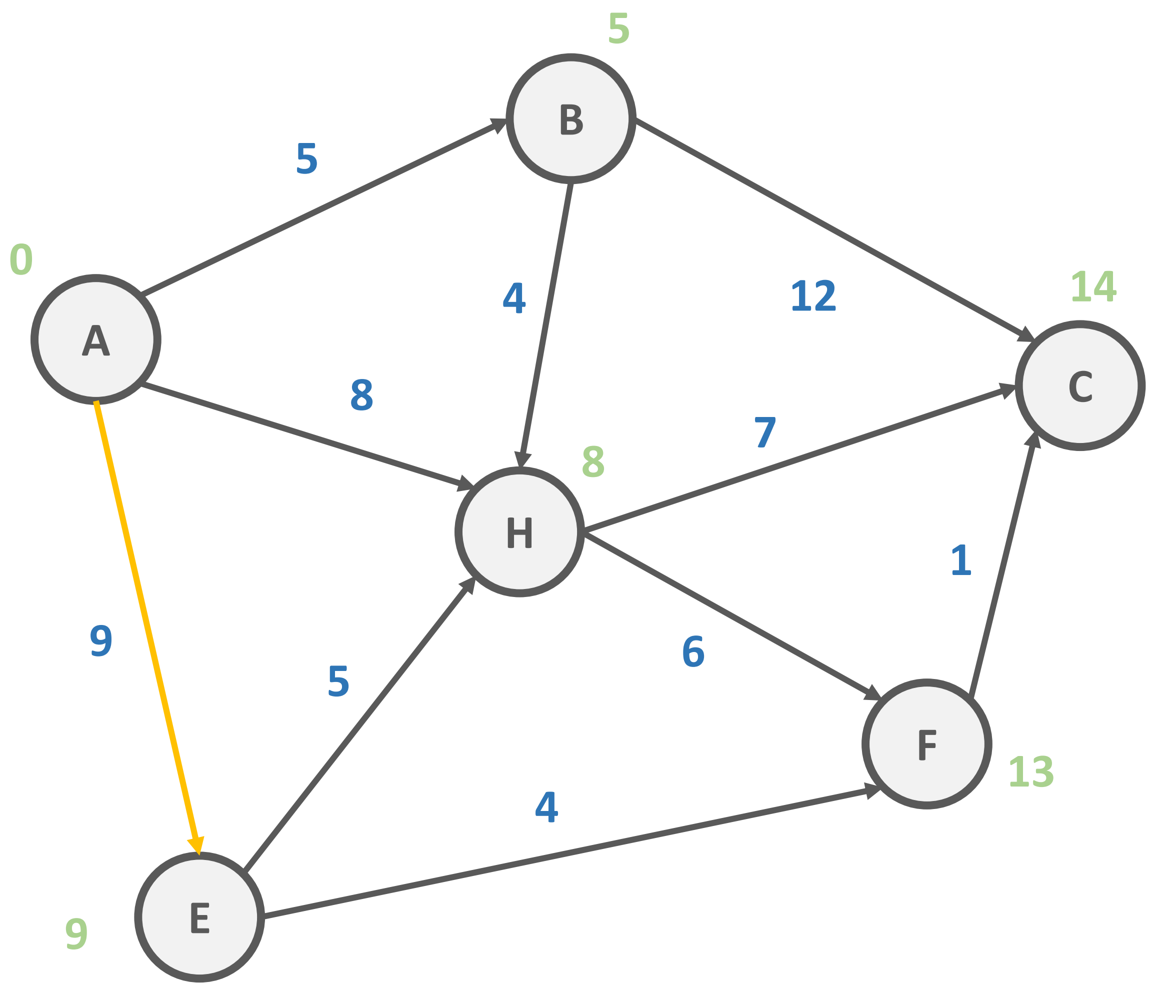
Bellman-Ford Algorithm



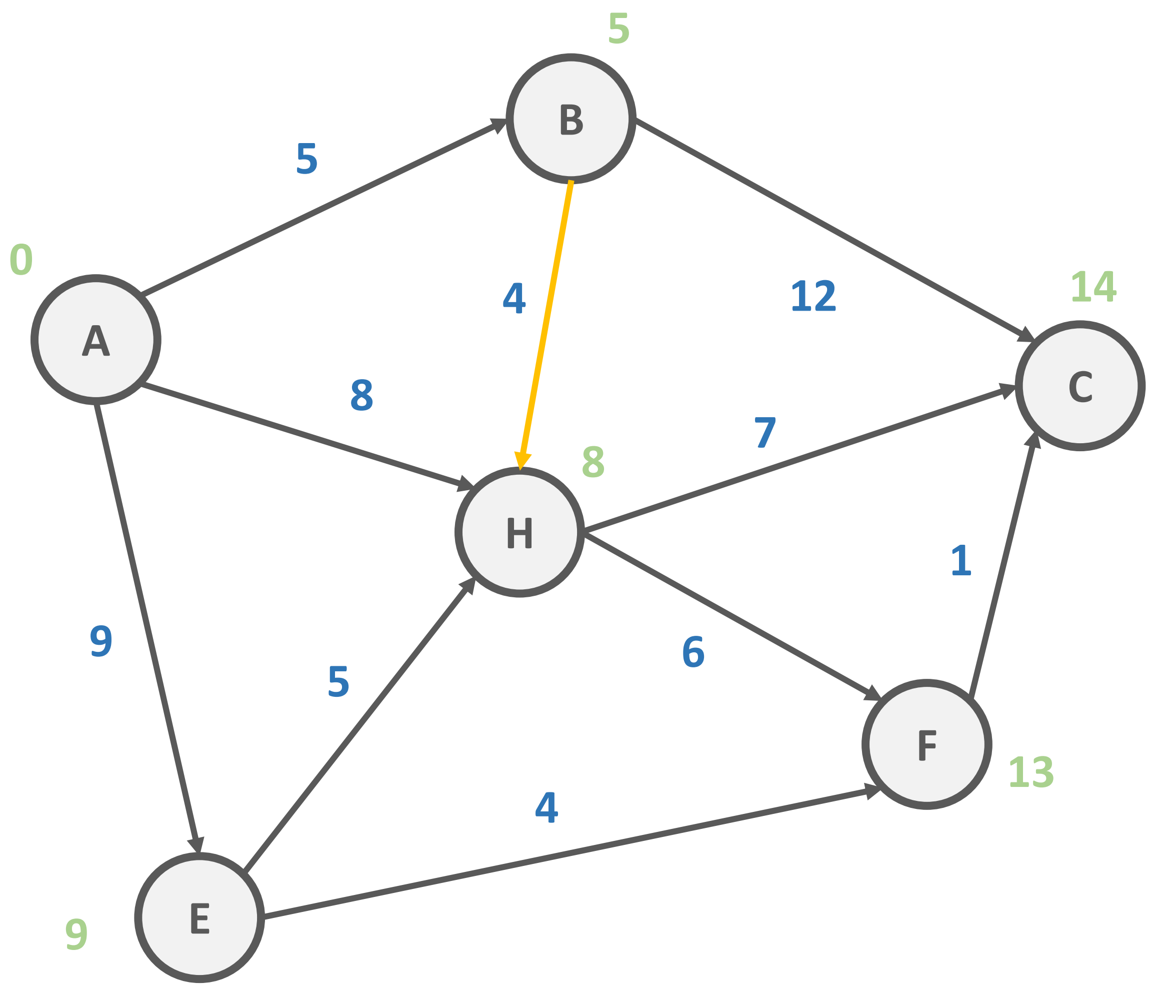
Bellman-Ford Algorithm



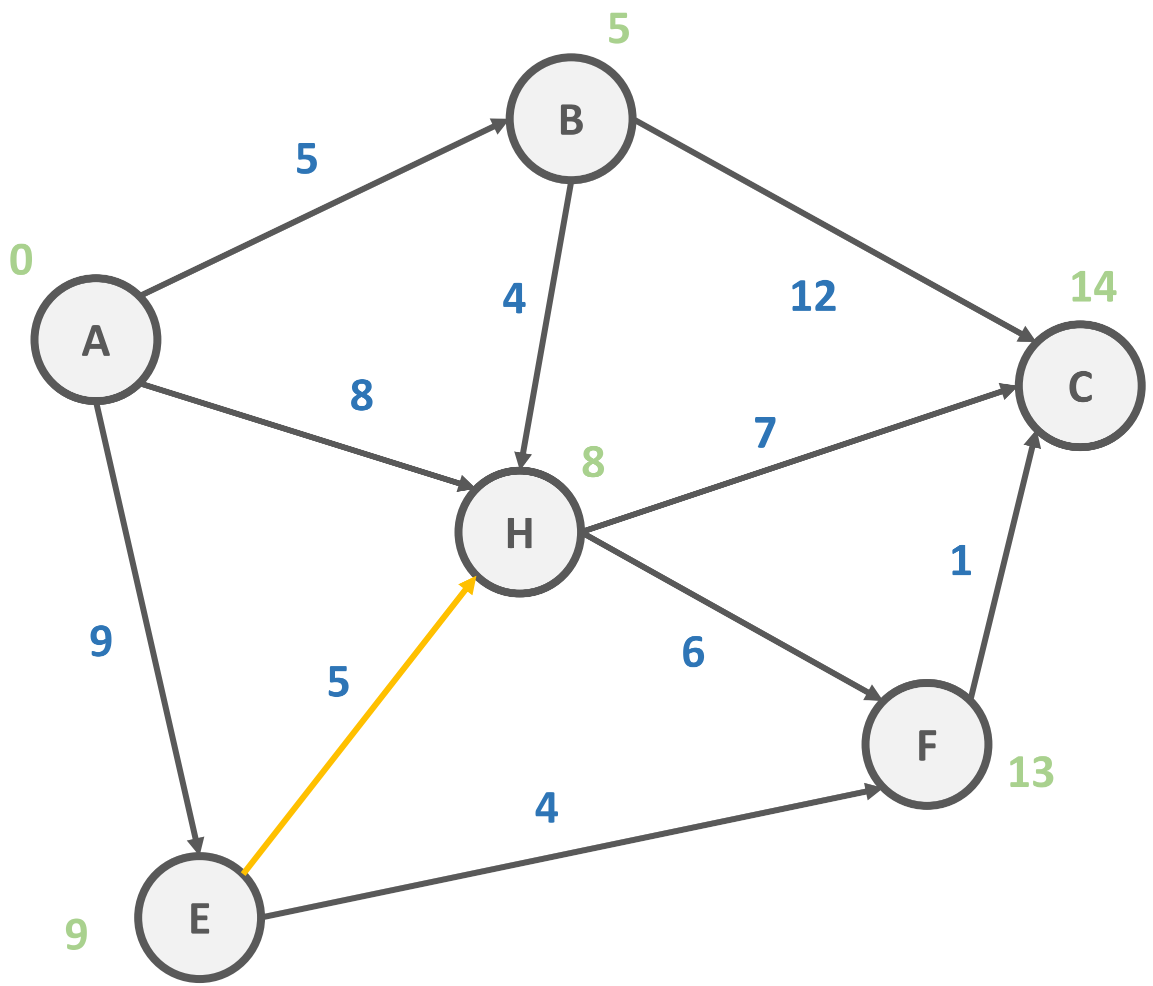
Bellman-Ford Algorithm



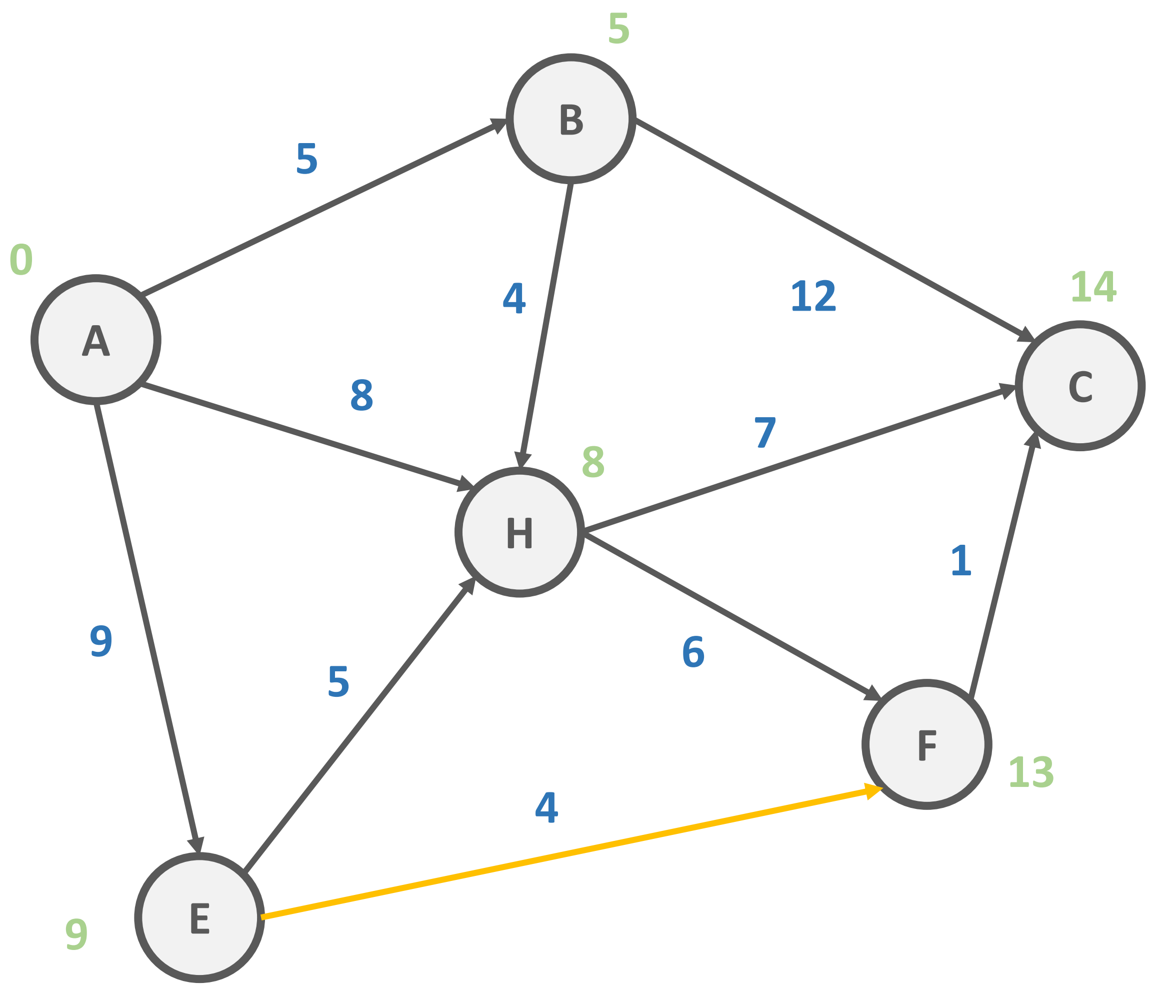
Bellman-Ford Algorithm



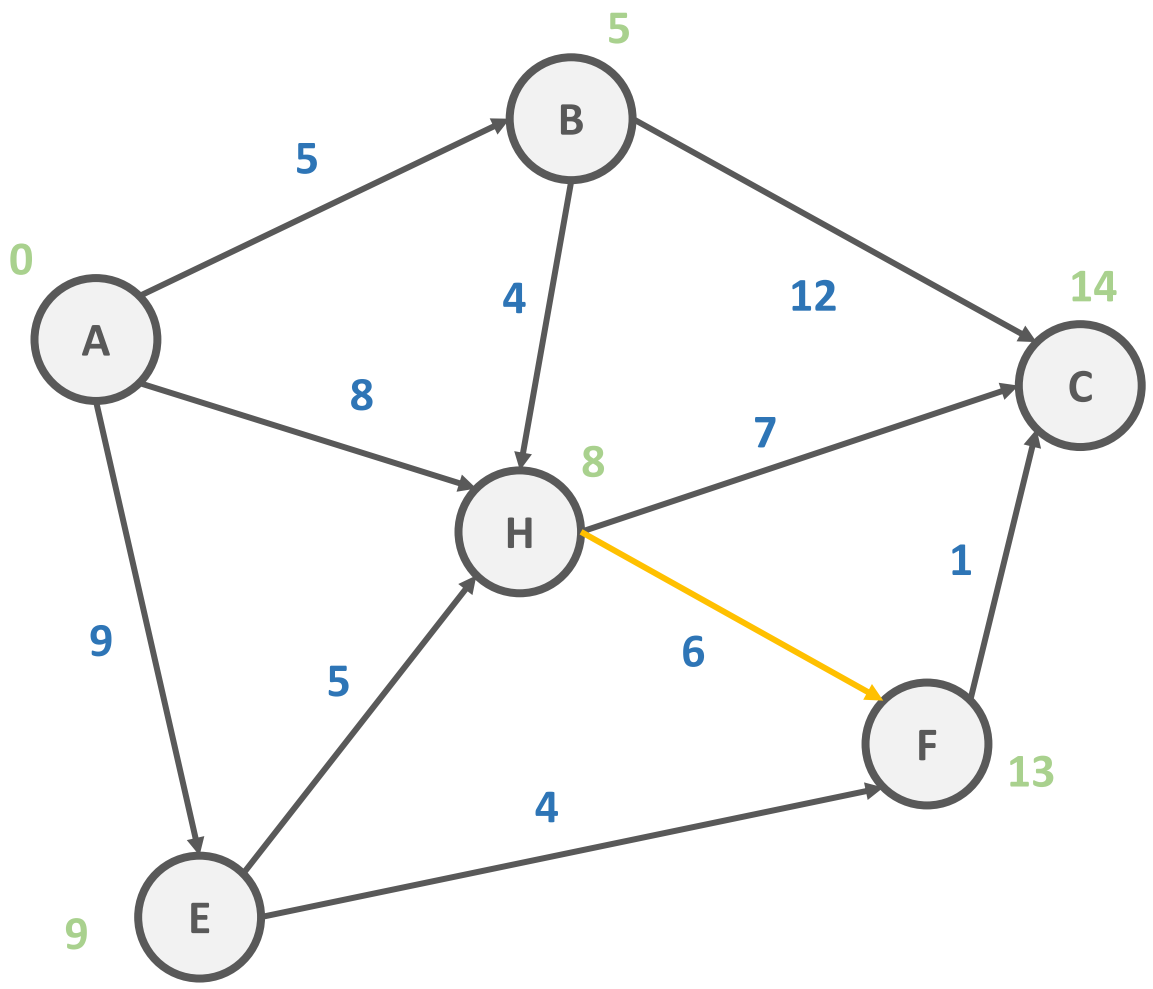
Bellman-Ford Algorithm



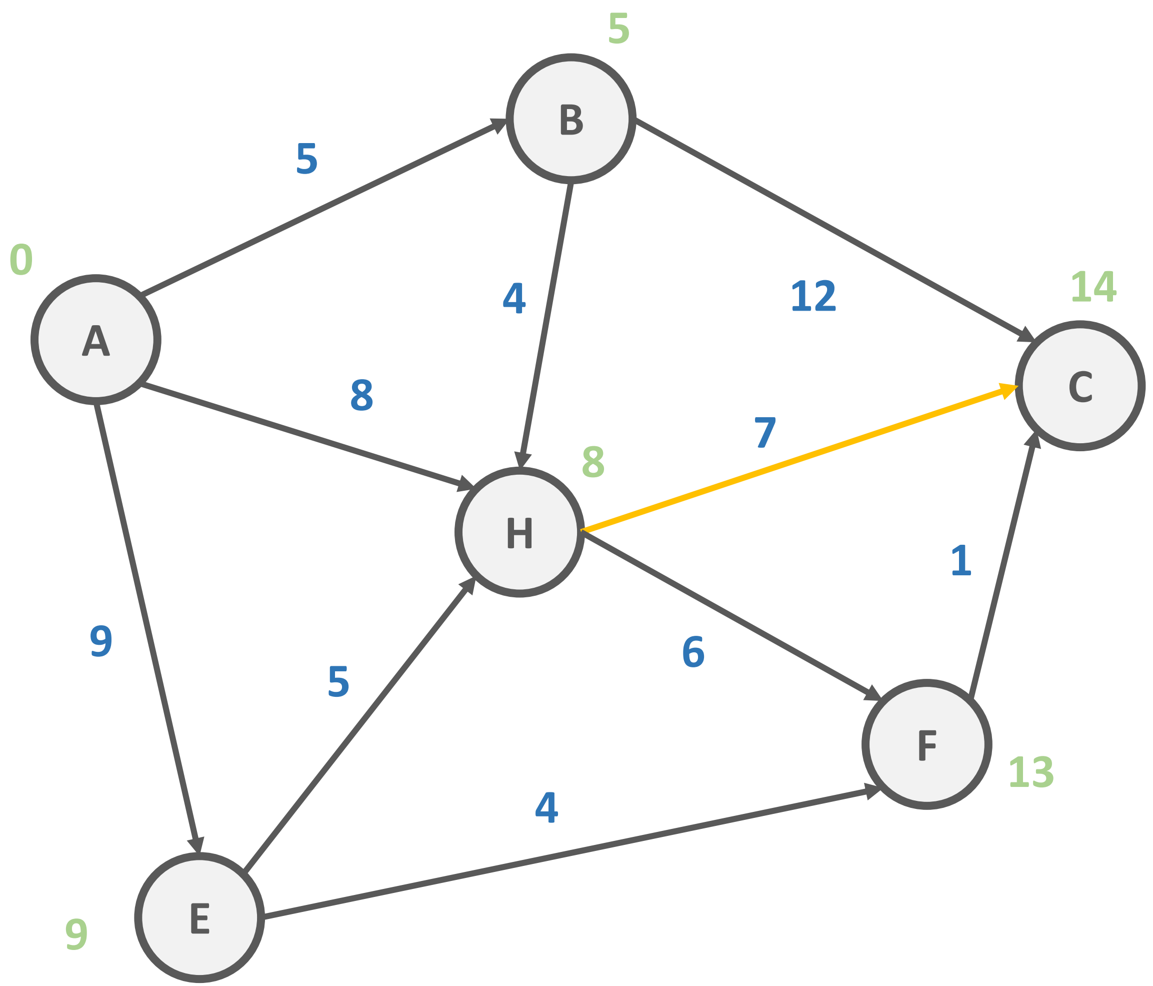
Bellman-Ford Algorithm



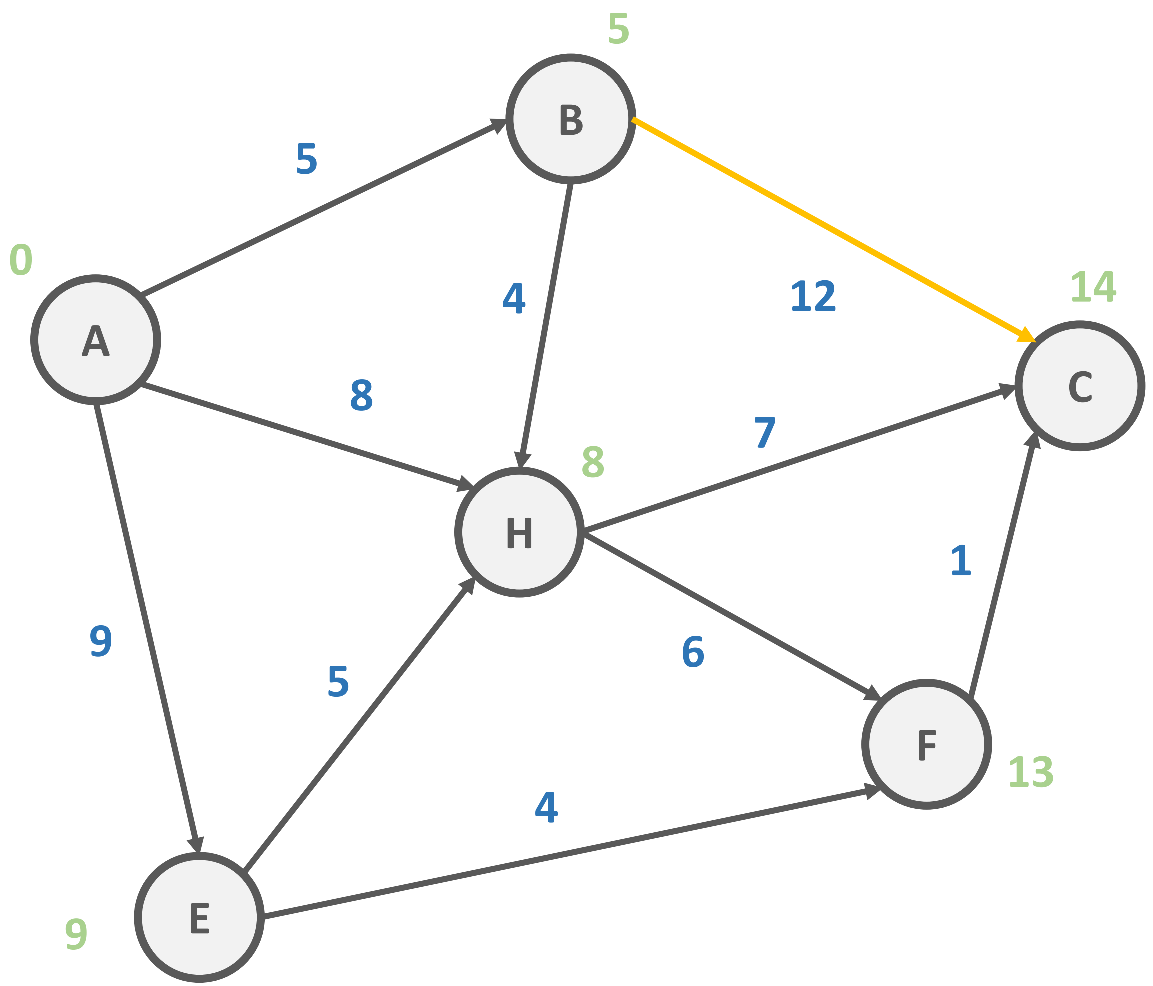
Bellman-Ford Algorithm



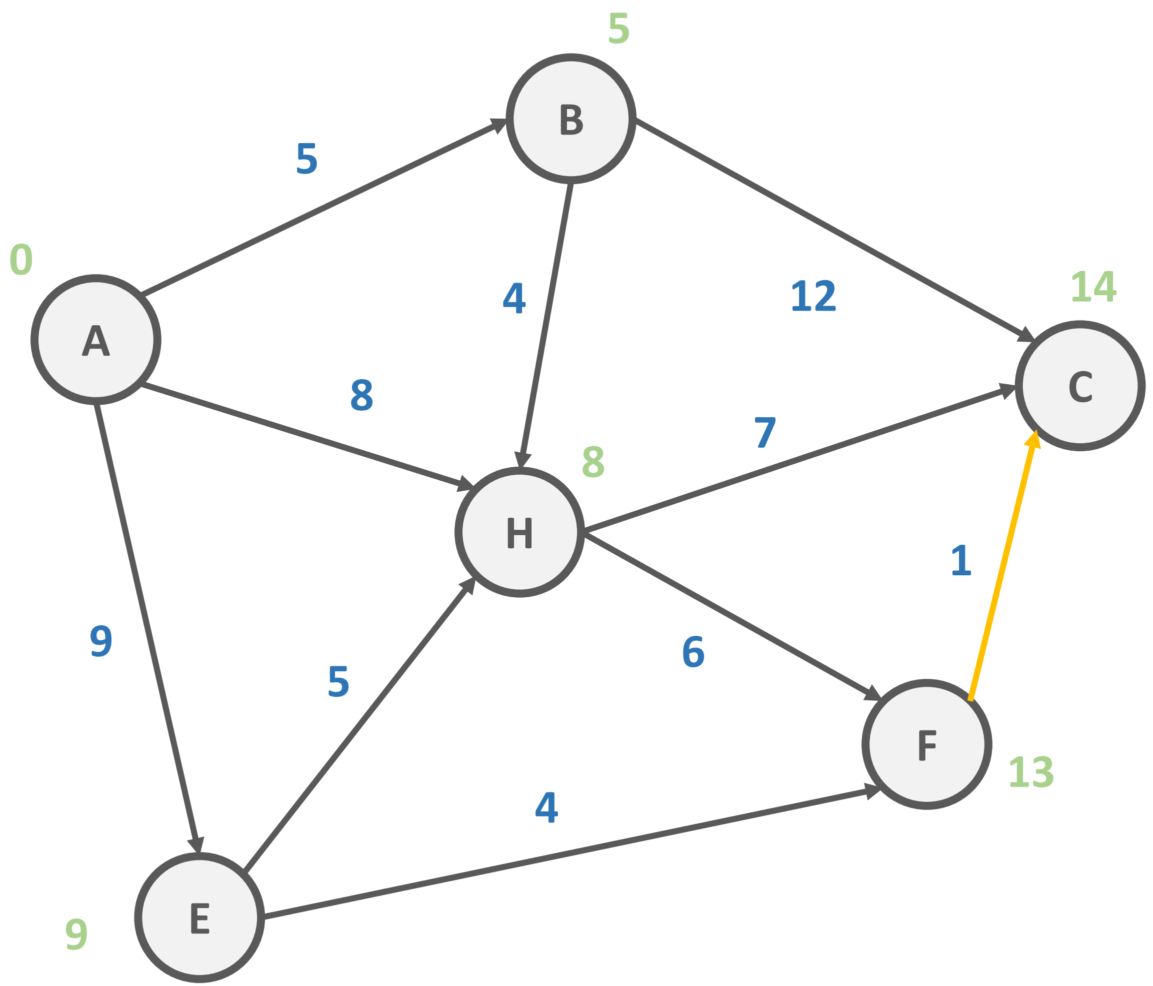
Bellman-Ford Algorithm



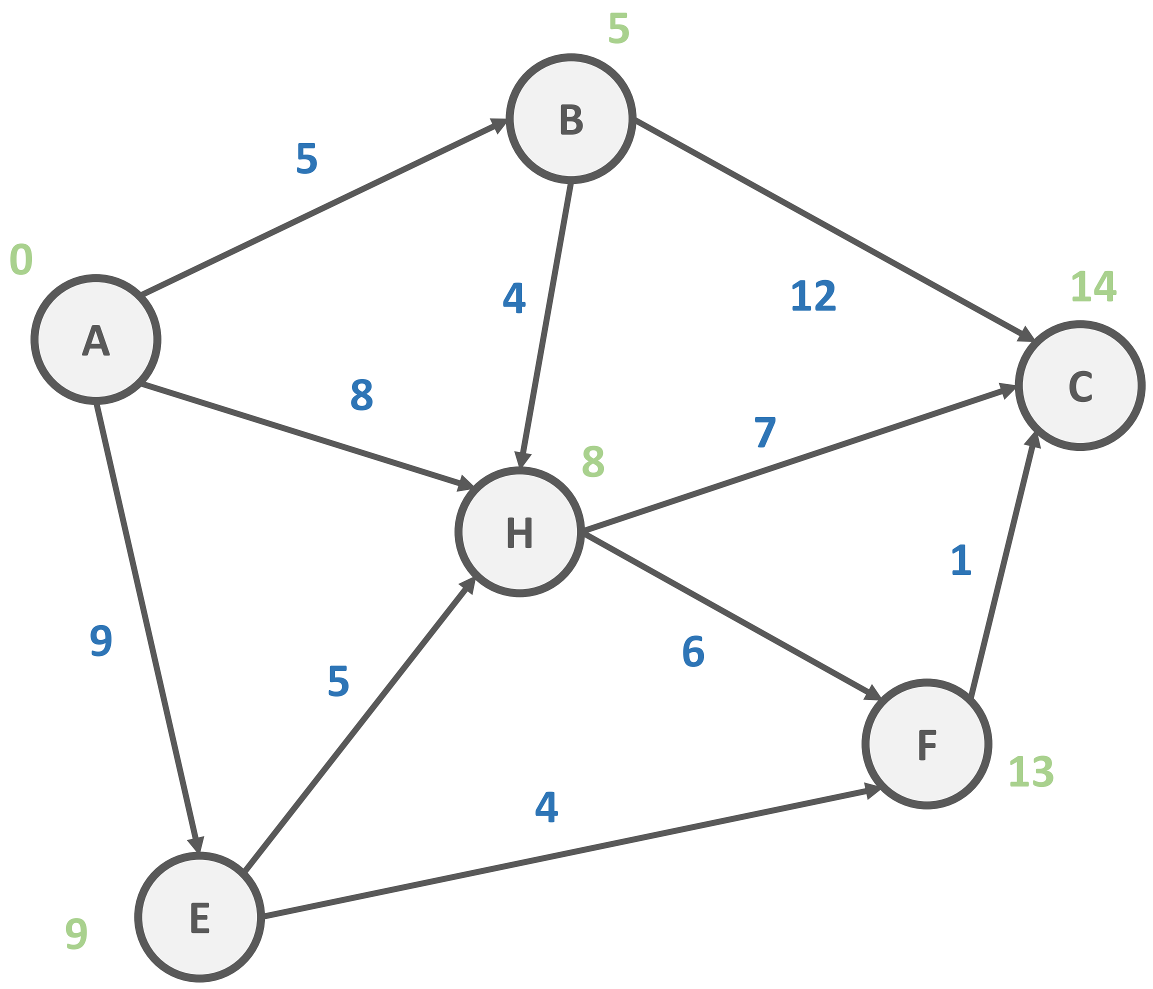
Bellman-Ford Algorithm



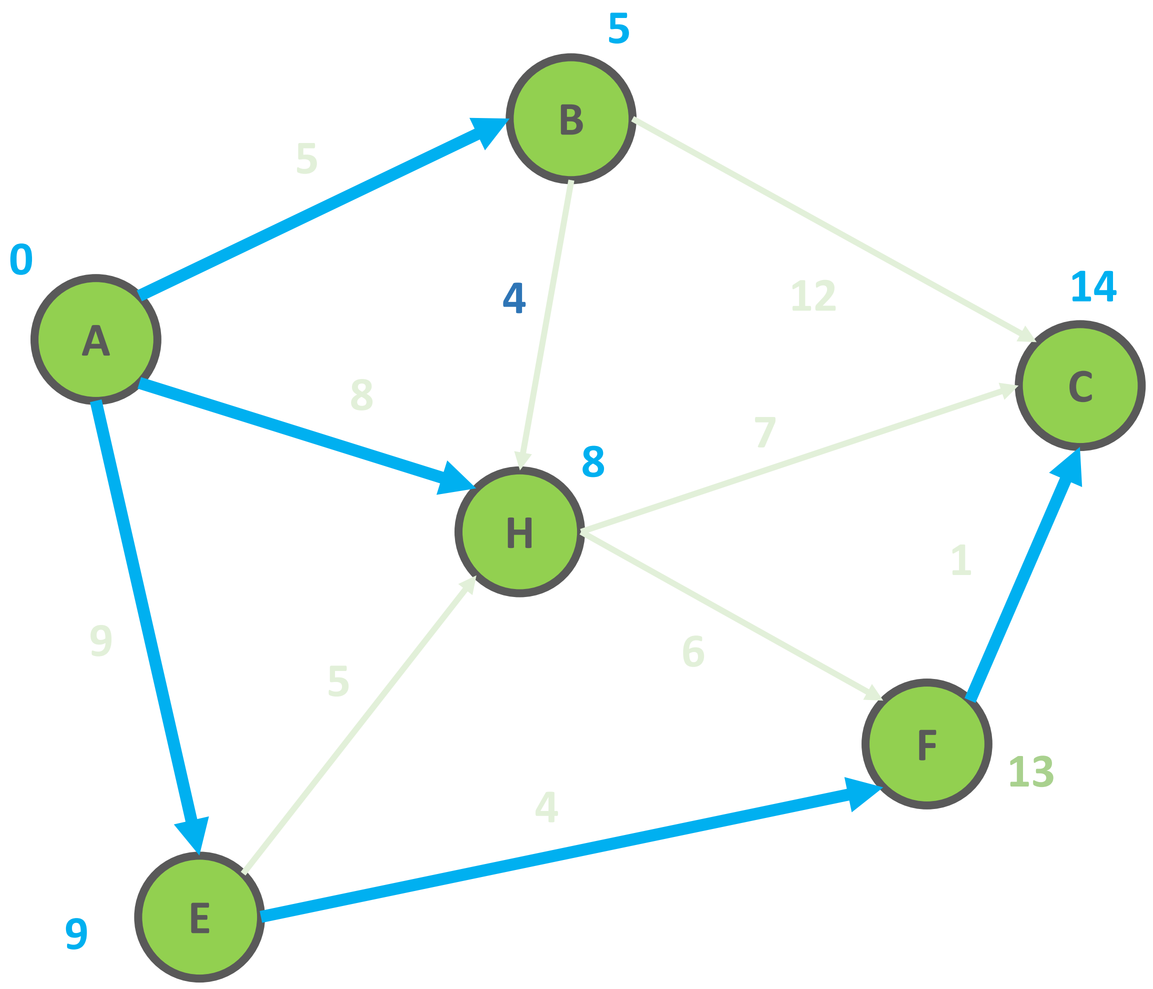
Bellman-Ford Algorithm



Bellman-Ford Algorithm



Bellman-Ford Algorithm



Greedy vs Dynamic Programming

(Algorithms and Data Structures)

Greedy Algorithms and Dynamic Programming

GREEDY ALGORITHM *is an algorithmic paradigm that constructs the final solution by choosing the best option possible in every iteration*

It combines locally optimal solutions to get the global solution (final result)

DYNAMIC PROGRAMMING *is an algorithmic paradigm that avoids recalculating the same problems over and over again*

It uses extra memory (memoization or tabulation) to store the subresults

Dynamic Programming Paradigm

- **dynamic programming** is both an optimization technique and a computer programming method
- it was introduced by **Richard Bellman** in **1953**
- the main idea is that **we can break down complicated problems into smaller subproblems** usually in a recursive manner
- then we find the solutions for these subproblems and finally we combine the subresults to find the final solution

Dynamic Programming Paradigm

We can apply **dynamic programming** approach if the problem has the following features:

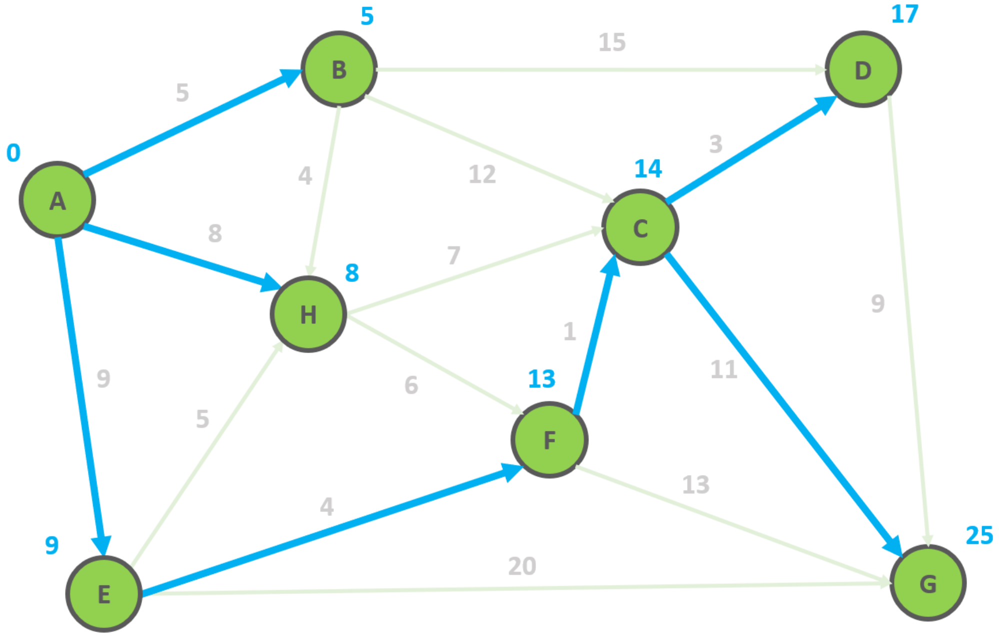
1.) OPTIMAL SUBSTRUCTURE

In computer science a problem is said to have **optimal substructure** if an **optimal** solution can be constructed from **optimal** solutions of its subproblems

2.) OVERLAPPING SUBPROBLEMS

The given subproblems are not independent of each other

Dynamic Programming Paradigm



finding the shortest paths to vertex **G** and to vertex **C** are not independent of each other

$$\delta(A,G) = \delta(A,C) + \delta(C,G)$$

if a vertex **x** lies in the shortest path from **A** source to **G** destination then the shortest path **δ(A,G)** is the combination of shortest path from **A** to **x** and shortest path from **x** to **G**