

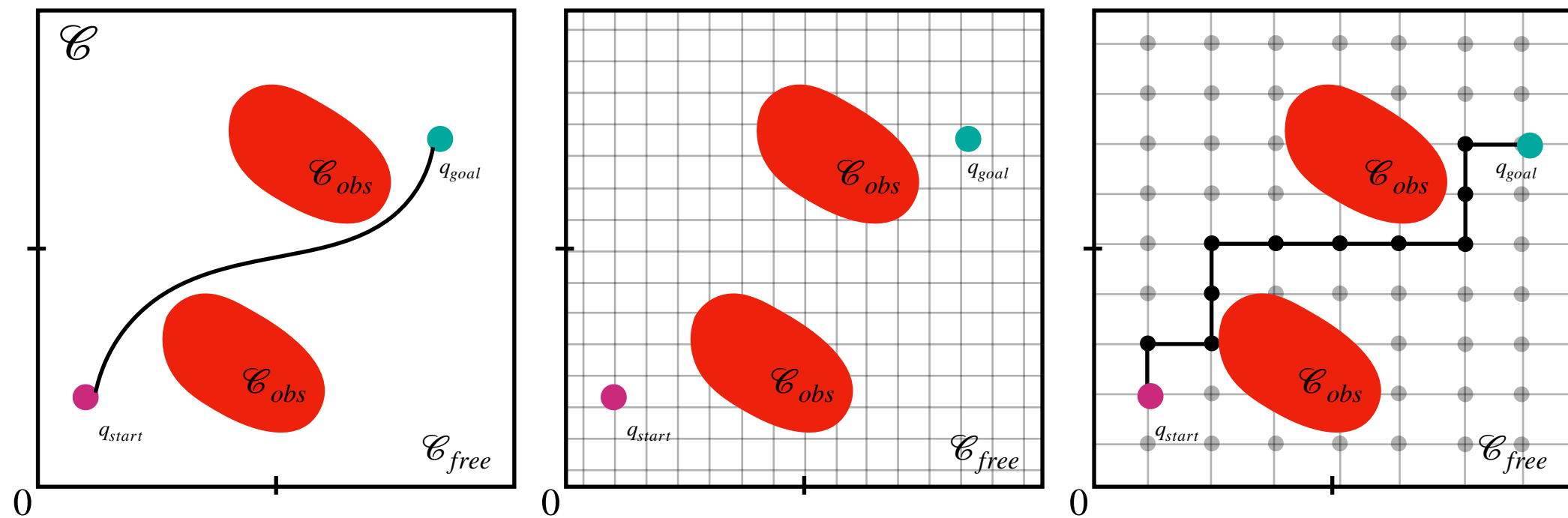


JRL302: Robotics Technologies

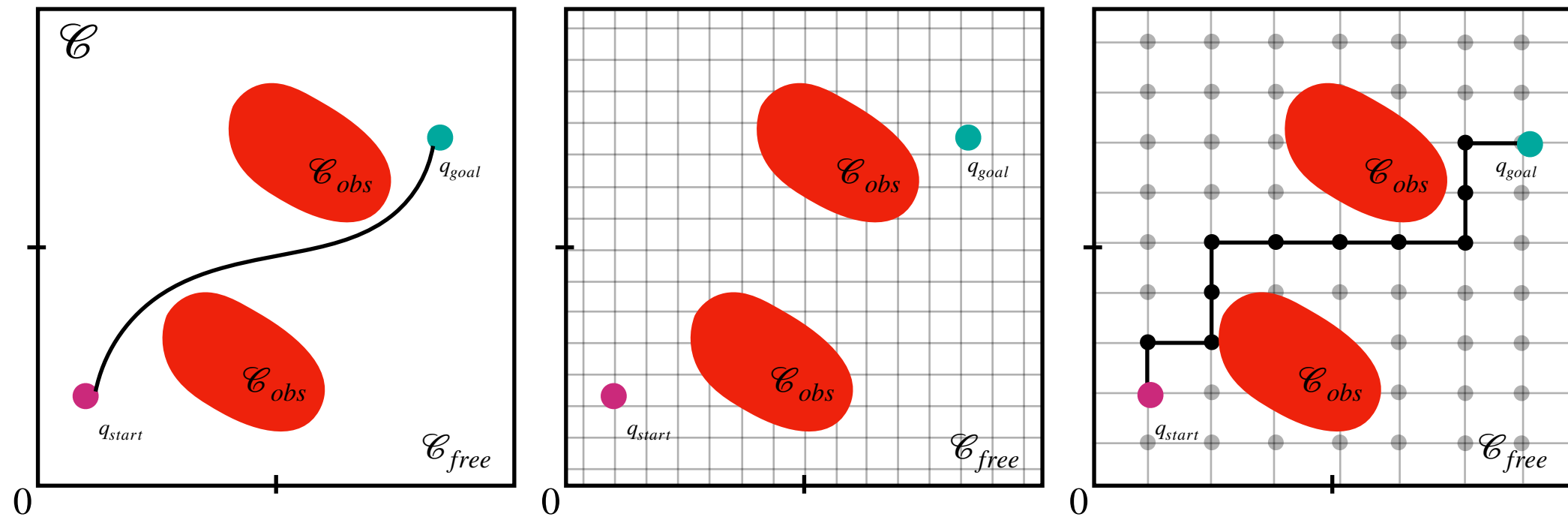
Lecture 2: Discrete and Sampling-Based Motion Planning

Mayank Sewlia
sewlia@iitd.ac.in

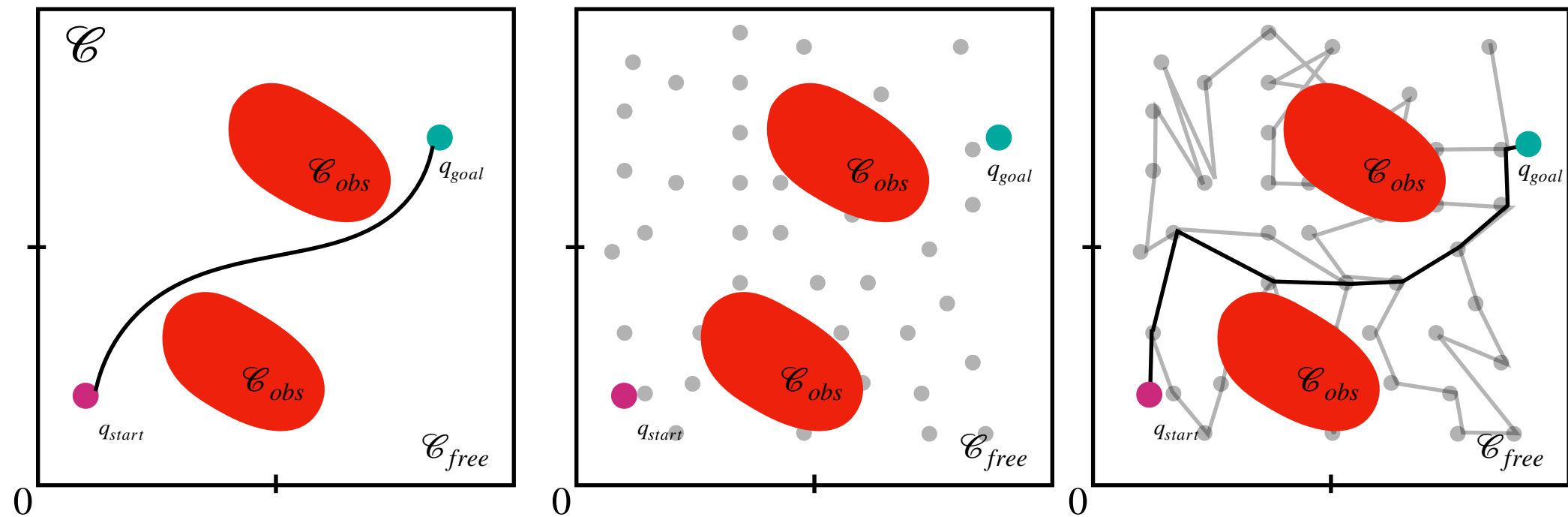
1. Grid Discretisation



1. Grid Discretisation



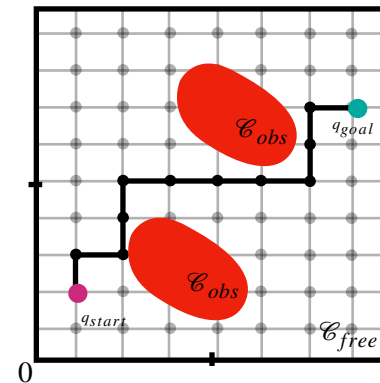
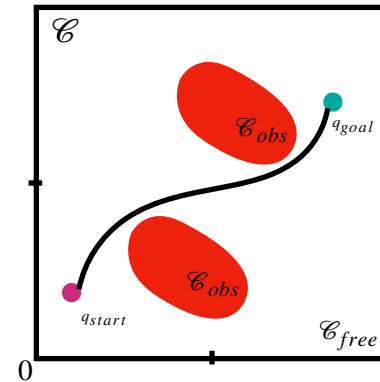
2. Sampling in C-space



1. Discretisation

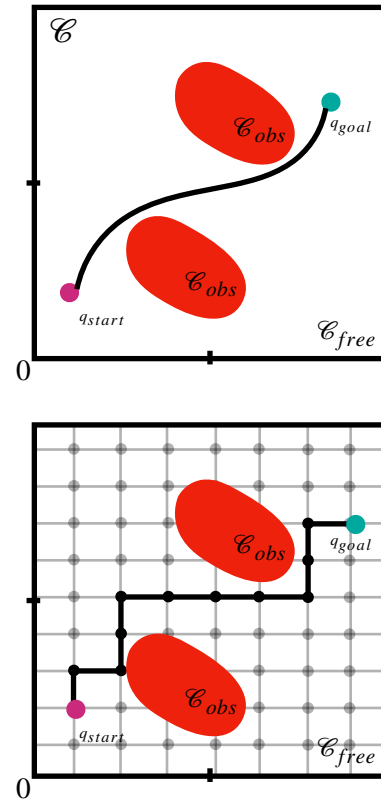
- Given: $C_{\text{free}} \subseteq \mathbb{R}^n$
- Find: $\sigma : [0,1] \rightarrow C_{\text{free}}$
s.t. $\sigma(0) = q_{\text{start}}, \sigma(1) = q_{\text{goal}}$

- Infinite paths in C-space
- Continuous planning -> search over a finite graph



1. Discretisation

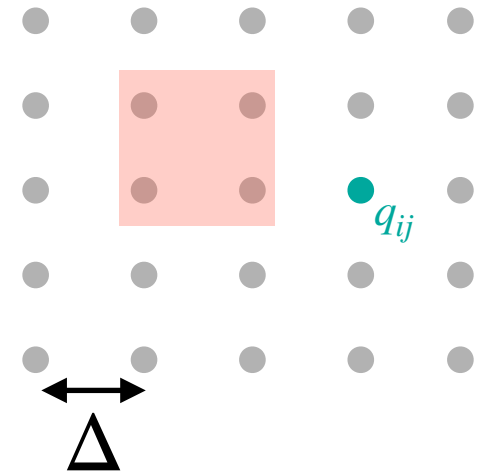
- Given: $C_{\text{free}} \subseteq \mathbb{R}^n$
- Find: $\sigma : [0,1] \rightarrow C_{\text{free}}$
s.t. $\sigma(0) = q_{\text{start}}, \sigma(1) = q_{\text{goal}}$
- Infinite paths in C-space
- Continuous planning $\rightarrow$ search over a finite graph



2. Grid Refinement

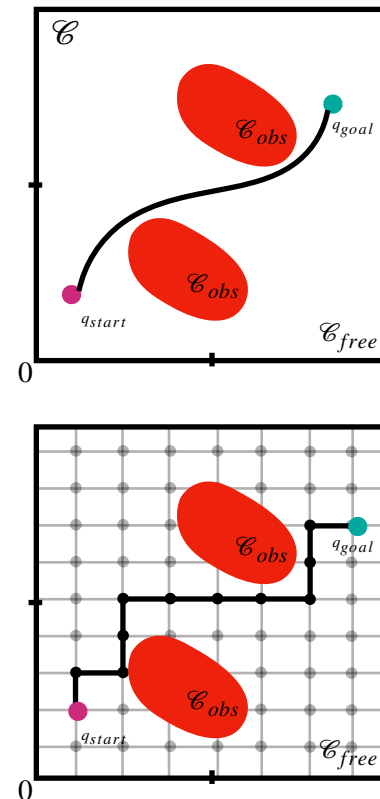
- For 2D: $C = [0, L_x] \times [0, L_y]$
- Pick Δ such that
 $q_{ij} = (i\Delta, j\Delta)$
- The number of configurations
scale as $N = \mathcal{O}\left(\frac{L_x L_y}{\Delta^2}\right)$

Halving Δ quadruples N



1. Discretisation

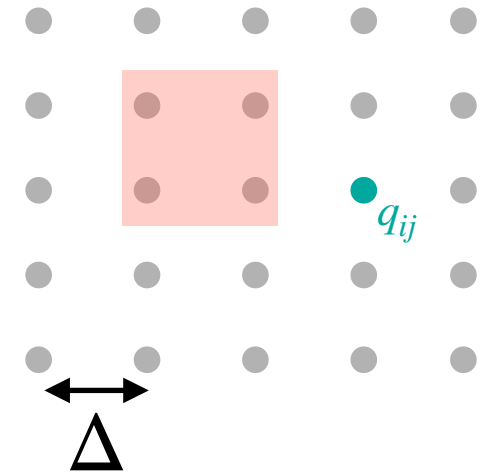
- Given: $C_{\text{free}} \subseteq \mathbb{R}^n$
 - Find: $\sigma : [0,1] \rightarrow C_{\text{free}}$
 - s.t. $\sigma(0) = q_{\text{start}}, \sigma(1) = q_{\text{goal}}$
-
- Infinite paths in C-space
 - Continuous planning -> search over a finite graph



2. Grid Refinement

- For 2D: $C = [0, L_x] \times [0, L_y]$
- Pick Δ such that $q_{ij} = (i\Delta, j\Delta)$
- The number of configurations scale as $N = \mathcal{O}\left(\frac{L_x L_y}{\Delta^2}\right)$

Halving Δ quadruples N



3. Curse of Dimensionality

For n dimensions, with m grid points per dimension,

$$N = m^n$$

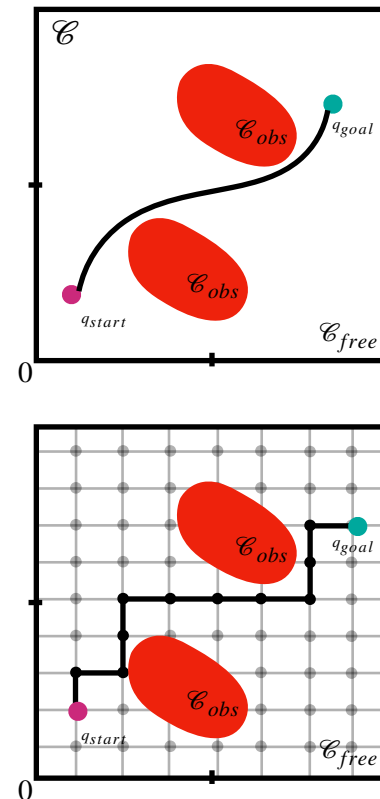
Grows exponentially

Degrees of freedom	Grid points (m=10)
2	100
3	1,000
6	10,00,000
7	1,00,00,000

1. Discretisation

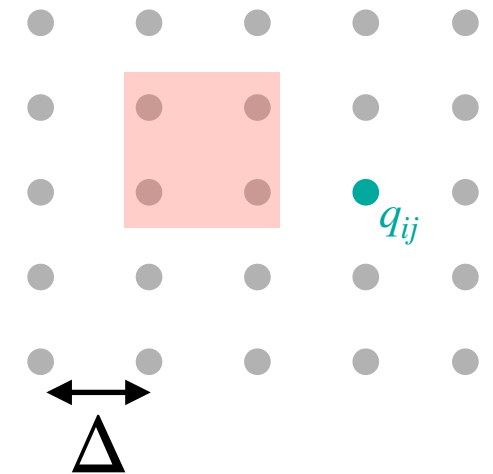
- Given: $C_{\text{free}} \subseteq \mathbb{R}^n$
- Find: $\sigma : [0,1] \rightarrow C_{\text{free}}$
- s.t. $\sigma(0) = q_{\text{start}}, \sigma(1) = q_{\text{goal}}$

- Infinite paths in C-space
- Continuous planning -> search over a finite graph



2. Grid Refinement

- For 2D: $C = [0, L_x] \times [0, L_y]$
- Pick Δ such that $q_{ij} = (i\Delta, j\Delta)$
- The number of configurations scale as $N = \mathcal{O}\left(\frac{L_x L_y}{\Delta^2}\right)$



Halving Δ quadruples N

3. Curse of Dimensionality

For n dimensions, with m grid points per dimension,

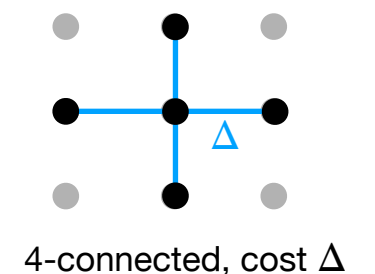
$$N = m^n$$

Grows exponentially

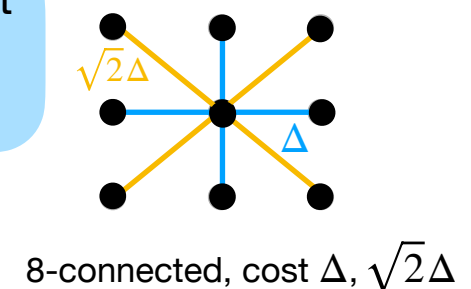
Degrees of freedom	Grid points (m=10)
2	100
3	1,000
6	10,00,000
7	1,00,00,000

4. Grid Traversal

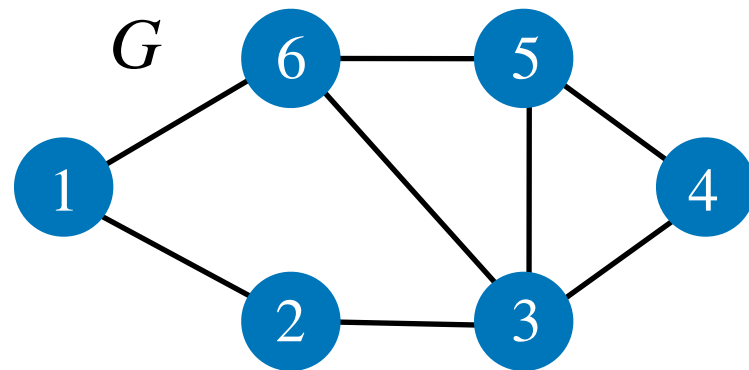
- 4-connected: Axial edge cost Δ
- 8-connected: +Diagonal $\sqrt{2}\Delta$



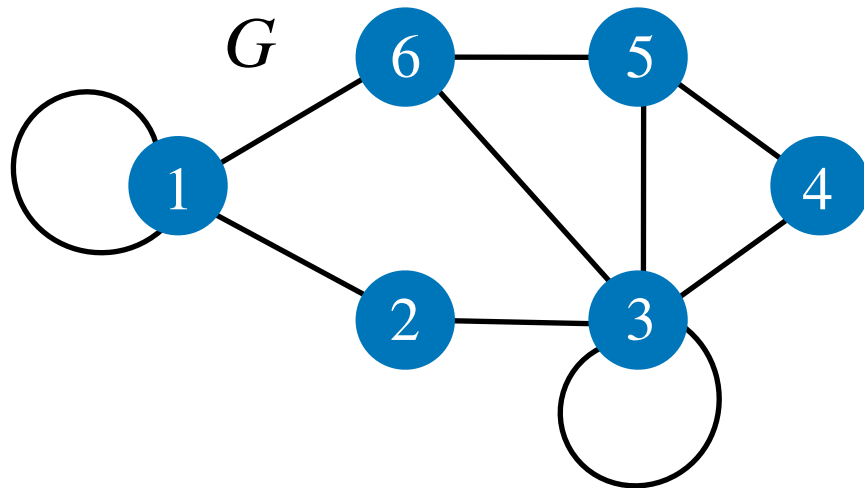
More edges give more path choices, but also more edges and collision checks. Check corner-cutting explicitly.



Graphs

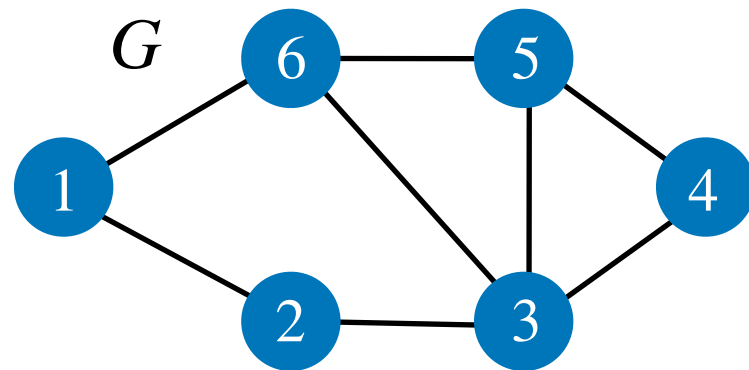


- Graph $G(V, E)$
- A set V of vertices or nodes
 - Connected by a set of E edges or links
 - Elements of E are unordered pairs (u, v) , $u, v \in V$
-
- Vertices are $V = \{1, 2, 3, 4, 5, 6\}$
 - Edges $E = \{(1, 2), (1, 5), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (4, 6)\}$

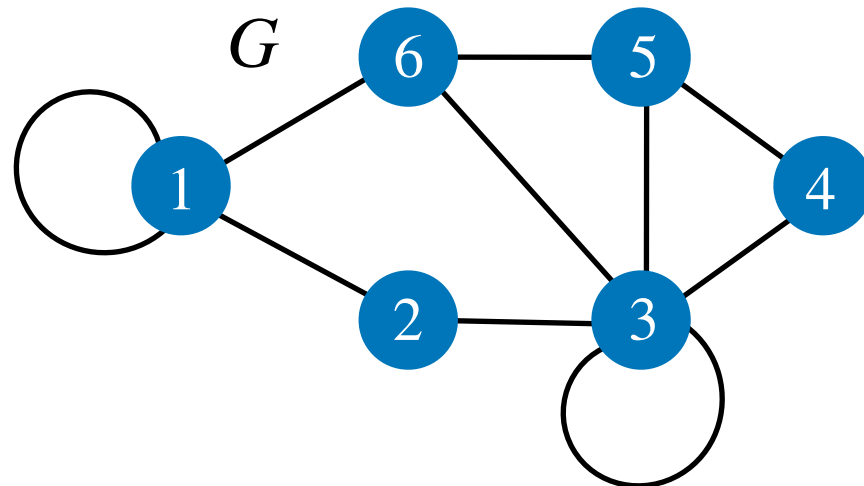


Simple graph = no self loops

Graphs

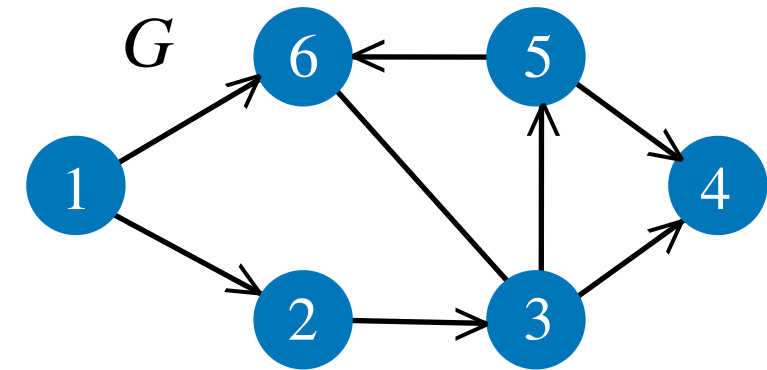


- Graph $G(V, E)$
- A set V of vertices or nodes
 - Connected by a set of E edges or links
 - Elements of E are unordered pairs (u, v) , $u, v \in V$
- Vertices are $V = \{1, 2, 3, 4, 5, 6\}$
 - Edges $E = \{(1, 2), (1, 5), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (4, 6)\}$

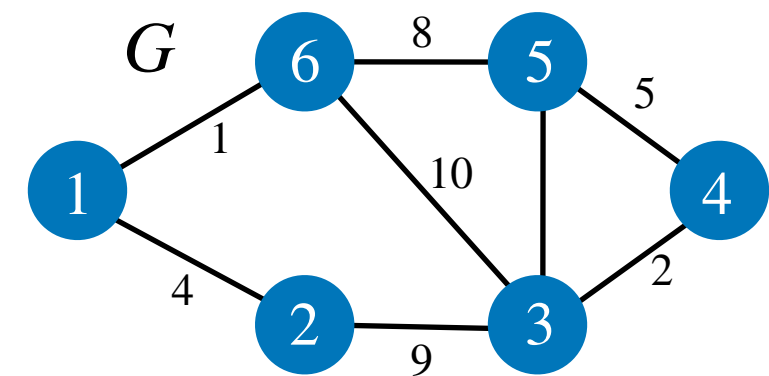


Simple graph = no self loops

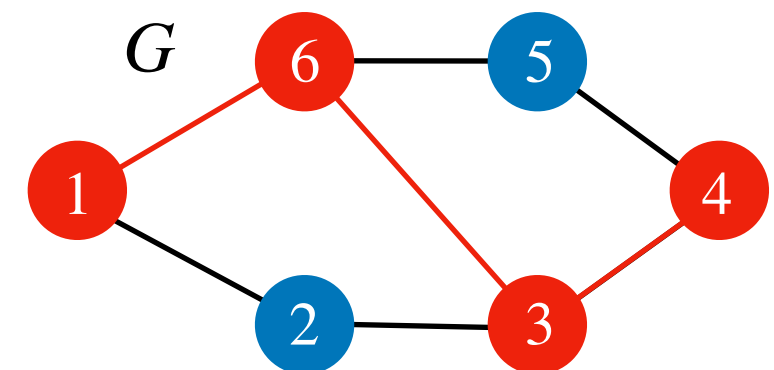
Directed graphs



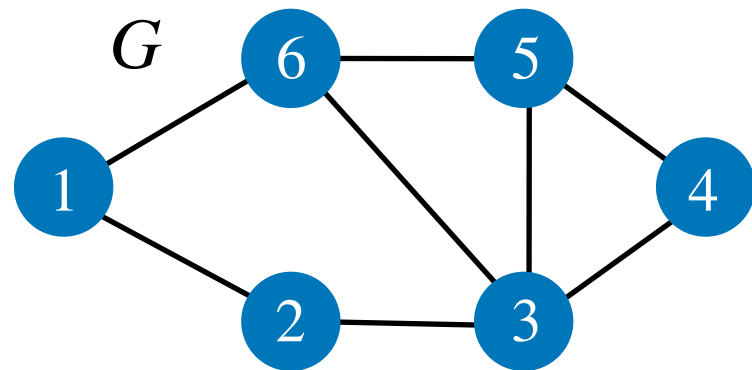
Weighted graphs



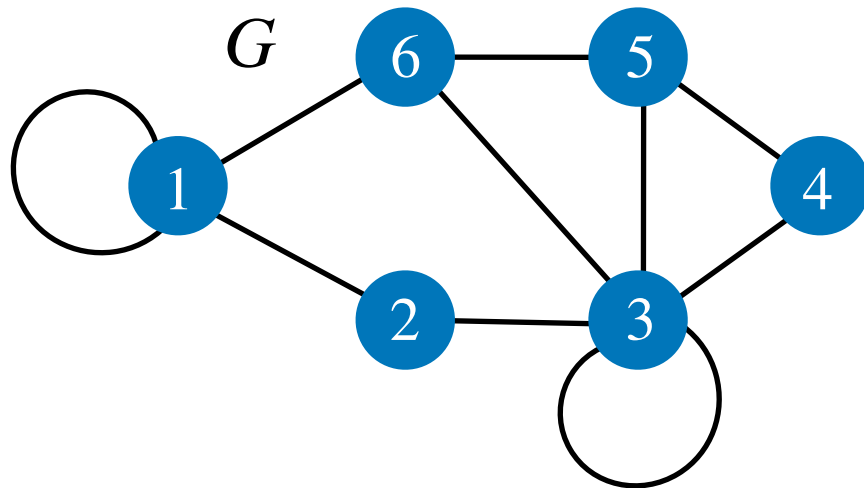
Path between adjacent vertices



Graphs



- Graph $G(V, E)$
- A set V of vertices or nodes
 - Connected by a set of E edges or links
 - Elements of E are unordered pairs (u, v) , $u, v \in V$
 - Vertices are $V = \{1, 2, 3, 4, 5, 6\}$
 - Edges $E = \{(1, 2), (1, 5), (2, 3), (3, 4), (3, 5), (3, 6), (4, 5), (4, 6)\}$

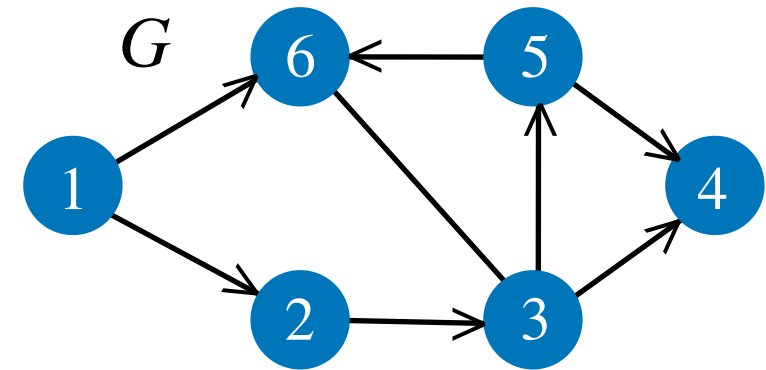


Simple graph = no self loops

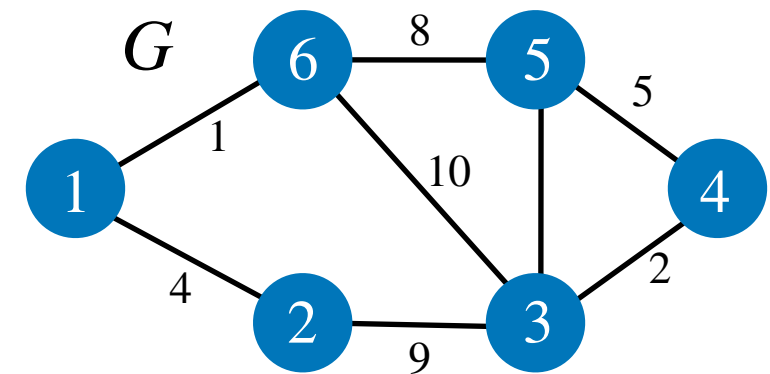
Shortest path problem

$$P = \arg \min_{P:s \rightarrow g} \sum_{(u,v) \in E} w(u, v)$$

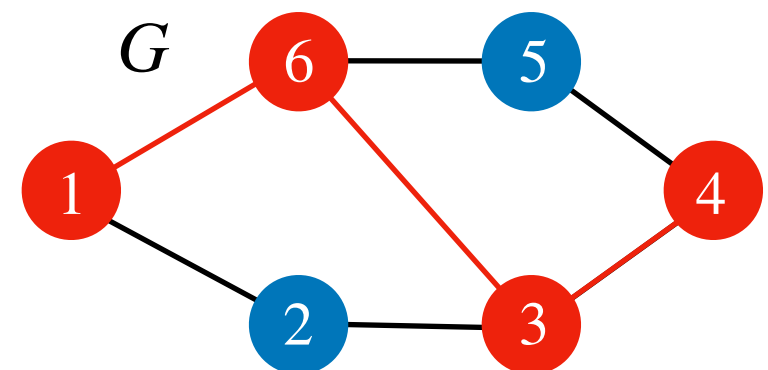
Directed graphs



Weighted graphs



Path between adjacent vertices



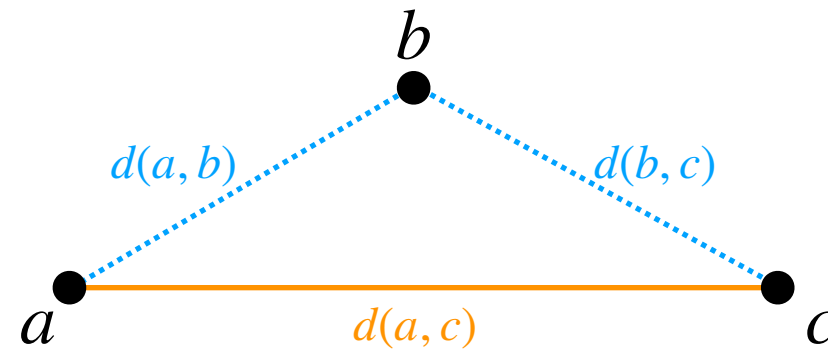
Distance and Volume in C -space

- All sampling-based planners require a function that measures distance between two points in $\mathcal{C}$: *metric spaces*
- Many algorithms also require the notion of volume in $\mathcal{C}$: *measure space*

Metric Spaces

A metric space (C, d) is equipped with a function $d : C \times C \rightarrow \mathbb{R}_{\geq 0}$ satisfying,

- Nonnegativity: $d(a, b) \geq 0$
- Reflexivity: $d(a, b) = 0$ if and only if $a = b$
- Symmetry: $d(a, b) = d(b, a)$
- Triangle inequality: $d(a, b) + d(b, c) \geq d(a, c)$



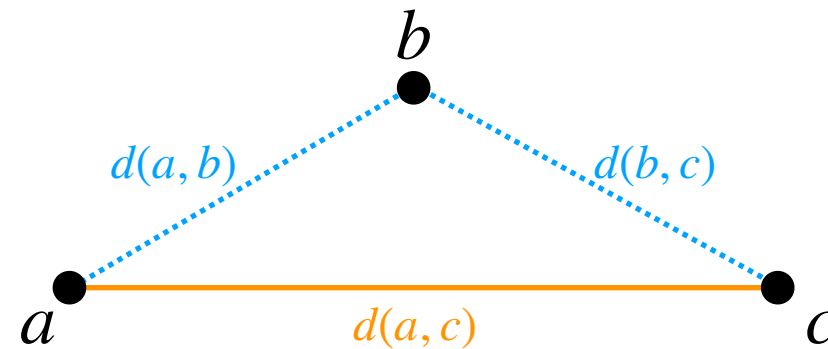
Distance and Volume in C-space

- All sampling-based planners require a function that measures distance between two points in $\mathcal{C}$: *metric spaces*
- Many algorithms also require the notion of volume in $\mathcal{C}$: *measure space*

Metric Spaces

A metric space (C, d) is equipped with a function $d : C \times C \rightarrow \mathbb{R}_{\geq 0}$ satisfying,

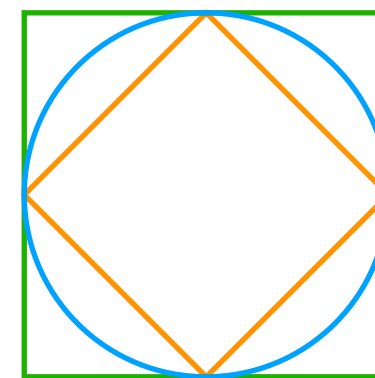
- Nonnegativity: $d(a, b) \geq 0$
- Reflexivity: $d(a, b) = 0$ if and only if $a = b$
- Symmetry: $d(a, b) = d(b, a)$
- Triangle inequality: $d(a, b) + d(b, c) \geq d(a, c)$



L_p Metrics in $\mathbb{R}^n$

For $q, q' \in \mathbb{R}^n$, define $d_p(q, q') = \left(\sum_{i=1}^n |q_i - q'_i|^p \right)^{\frac{1}{p}}$

- $L_1 : d_1 = \sum_i |q_i - q'_i|$
- $L_2 : d_2 = \sqrt{\sum_i (q_i - q'_i)^2}$
- $L_\infty : d_\infty = \max_i |q_i - q'_i|$



L_1 : Diamond

L_2 : Circle

L_∞ : Square

Unit balls in $\mathbb{R}^2$

Distance from $(0,0)$ to $(3,4)$ are ___ in L_1, L_2, L_∞ respectively

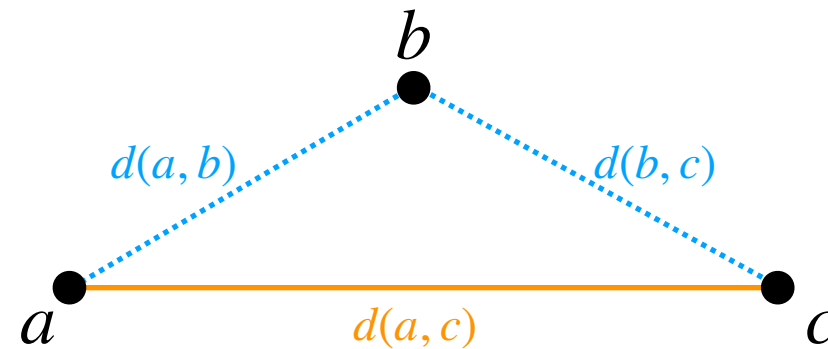
Distance and Volume in C-space

- All sampling-based planners require a function that measures distance between two points in $\mathcal{C}$: *metric spaces*
- Many algorithms also require the notion of volume in $\mathcal{C}$: *measure space*

Metric Spaces

A metric space (C, d) is equipped with a function $d : C \times C \rightarrow \mathbb{R}_{\geq 0}$ satisfying,

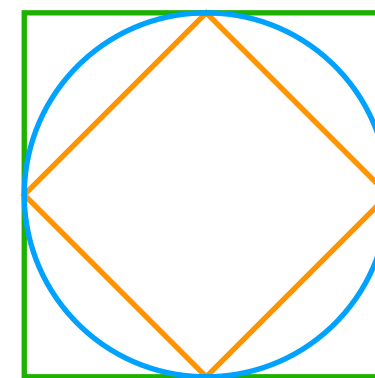
- Nonnegativity: $d(a, b) \geq 0$
- Reflexivity: $d(a, b) = 0$ if and only if $a = b$
- Symmetry: $d(a, b) = d(b, a)$
- Triangle inequality: $d(a, b) + d(b, c) \geq d(a, c)$



L_p Metrics in $\mathbb{R}^n$

For $q, q' \in \mathbb{R}^n$, define $d_p(q, q') = \left(\sum_{i=1}^n |q_i - q'_i|^p \right)^{\frac{1}{p}}$

- $L_1 : d_1 = \sum_i |q_i - q'_i|$
- $L_2 : d_2 = \sqrt{\sum_i (q_i - q'_i)^2}$
- $L_\infty : d_\infty = \max_i |q_i - q'_i|$



L_1 : Diamond

L_2 : Circle

L_∞ : Square

Unit balls in $\mathbb{R}^2$

Distance from $(0,0)$ to $(3,4)$ are 7,5,4 in L_1, L_2, L_∞ respectively

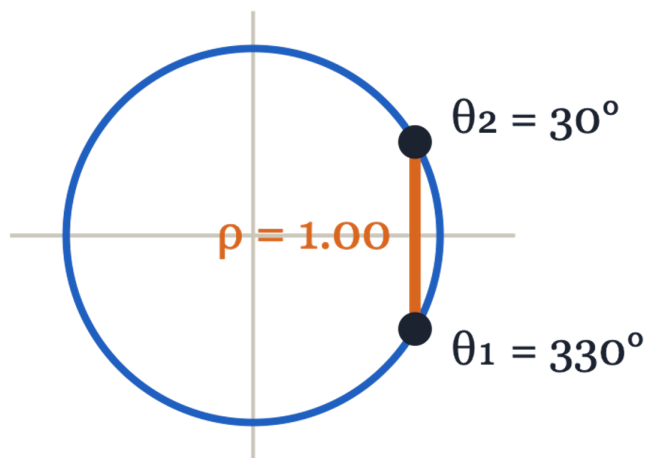
Ex 5.1: $SO(2)$ metric

If $SO(2)$ represented by unit complex numbers:

$$\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}.$$

Distance is,

$$d(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$$



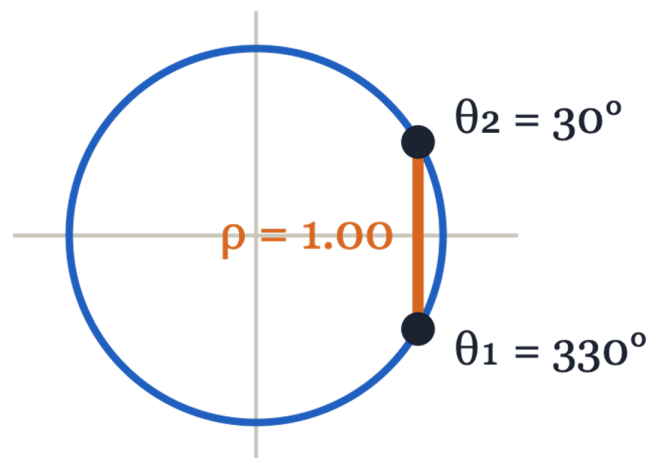
$$d(a_1, b_1, a_2, b_2) = 1.0$$

- $\theta_1 = 330^\circ \rightarrow (a_1, b_1) = (\cos 330, \sin 330) = (0.866, -0.5)$
- $\theta_2 = 30^\circ \rightarrow (a_2, b_2) = (\cos 30, \sin 30) = (0.866, 0.5)$

Ex 5.1: $SO(2)$ metric

If $SO(2)$ represented by unit complex numbers:
 $\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}$.
 Distance is,

$$d(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$$

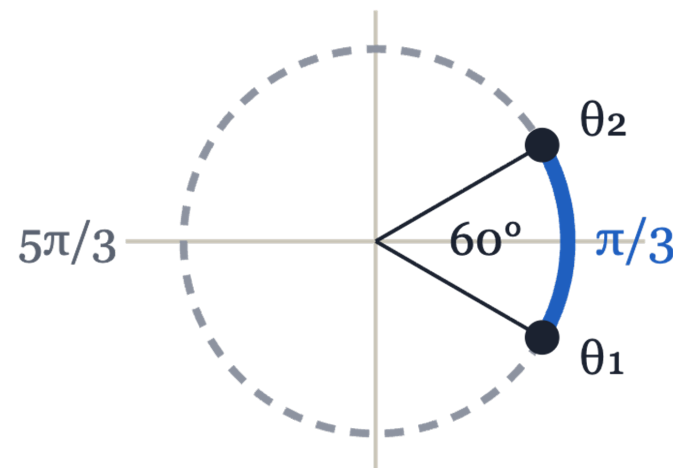


$$d(a_1, b_1, a_2, b_2) = 1.0$$

Ex 5.2: $SO(2)$ metric

Alternatively, directly comparing angles θ_1 and θ_2 gives the direct distance between two rotations.

$$d(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\}$$



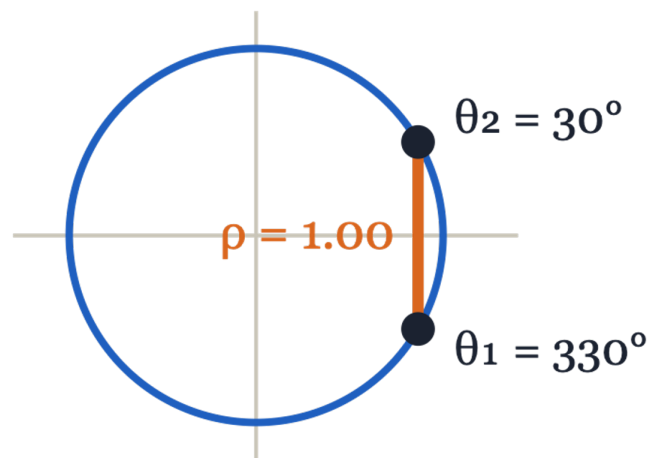
$$d(a_1, b_1, a_2, b_2) = 1.047$$

- $\theta_1 = 330^\circ \rightarrow (a_1, b_1) = (\cos 330, \sin 330) = (0.866, -0.5)$
- $\theta_2 = 30^\circ \rightarrow (a_2, b_2) = (\cos 30, \sin 30) = (0.866, 0.5)$

Ex 5.1: $SO(2)$ metric

If $SO(2)$ represented by unit complex numbers:
 $\{(a, b) \in \mathbb{R}^2 \mid a^2 + b^2 = 1\}$.
 Distance is,

$$d(a_1, b_1, a_2, b_2) = \sqrt{(a_1 - a_2)^2 + (b_1 - b_2)^2}$$

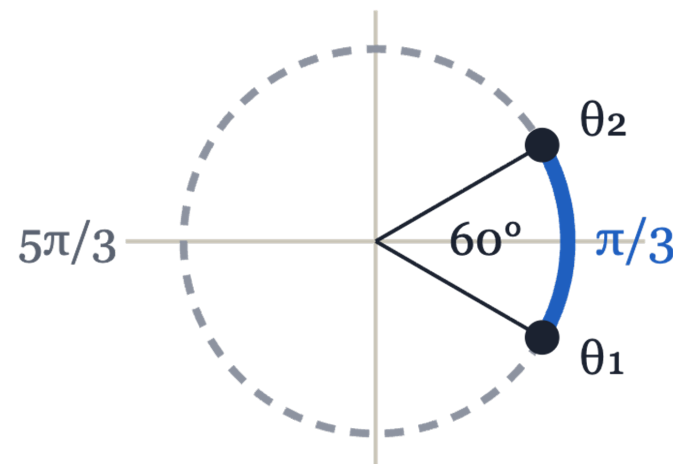


$$d(a_1, b_1, a_2, b_2) = 1.0$$

Ex 5.2: $SO(2)$ metric

Alternatively, directly comparing angles θ_1 and θ_2 gives the direct distance between two rotations.

$$d(\theta_1, \theta_2) = \min\{|\theta_1 - \theta_2|, 2\pi - |\theta_1 - \theta_2|\}$$



$$d(a_1, b_1, a_2, b_2) = 1.047$$

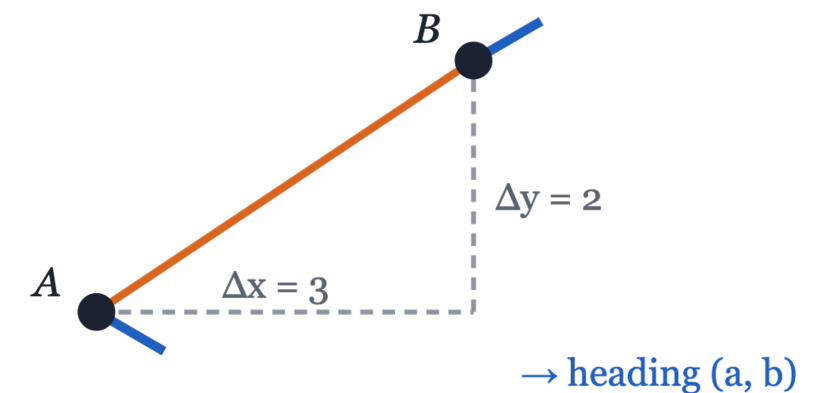
Ex 5.3: $SE(2)$ metric

With $SO(2)$ as (a, b) , a pose is a point (x, y, a, b)

$$d = \sqrt{(\Delta x)^2 + (\Delta y)^2 + \Delta a^2 + \Delta b^2}$$

pose1 : (1, 1, 0.866, -0.5)

pose2 : (4, 3, 0.866, 0.5)



$$d = 3.74$$

- $\theta_1 = 330^\circ \rightarrow (a_1, b_1) = (\cos 330, \sin 330) = (0.866, -0.5)$
- $\theta_2 = 30^\circ \rightarrow (a_2, b_2) = (\cos 30, \sin 30) = (0.866, 0.5)$

Dijkstra's Algorithm

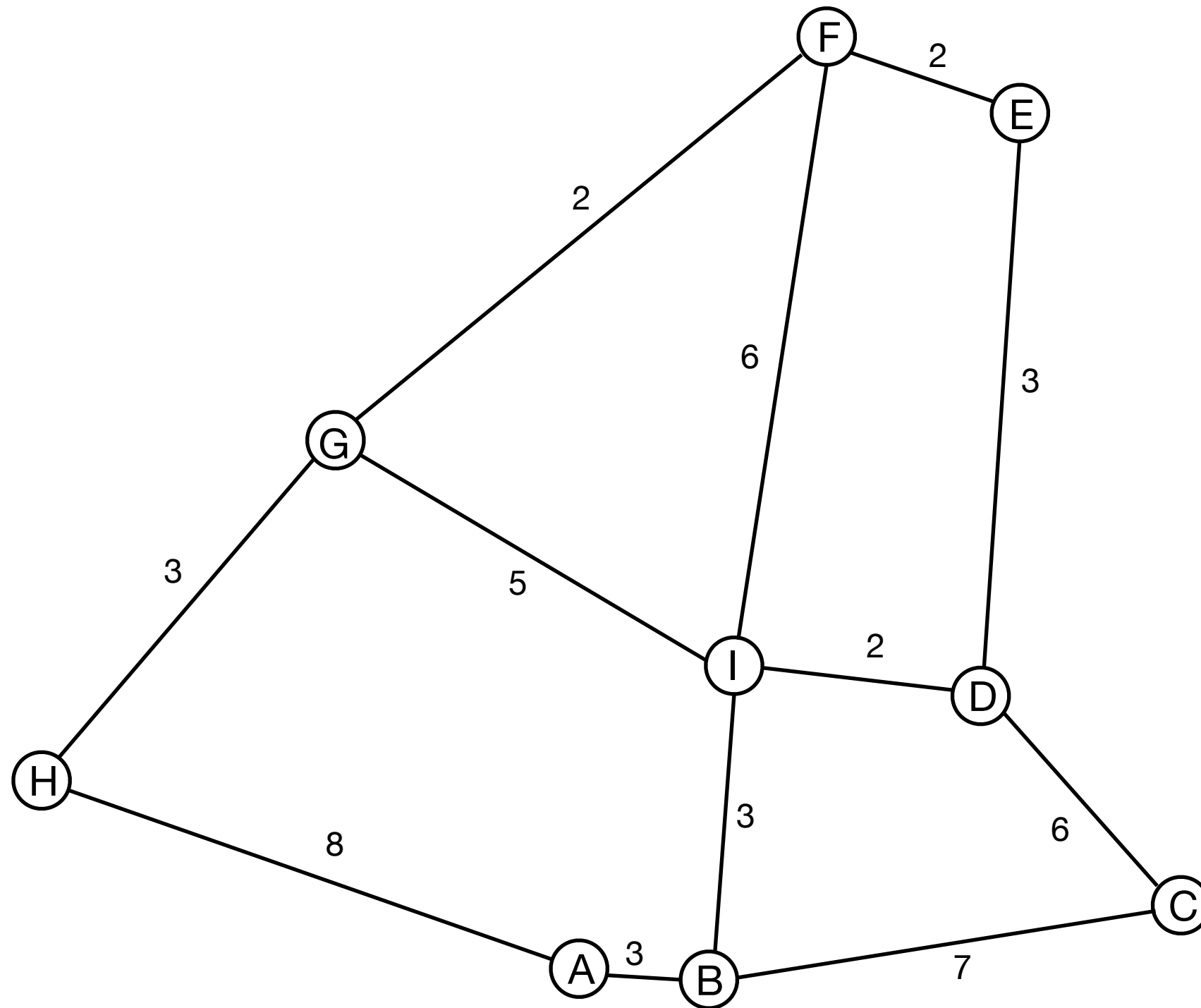
Goal: Find the lowest-cost path from some start vertex (source) to every other vertex in the graph

Assumptions: All edges have non-negative weights

Motivation:

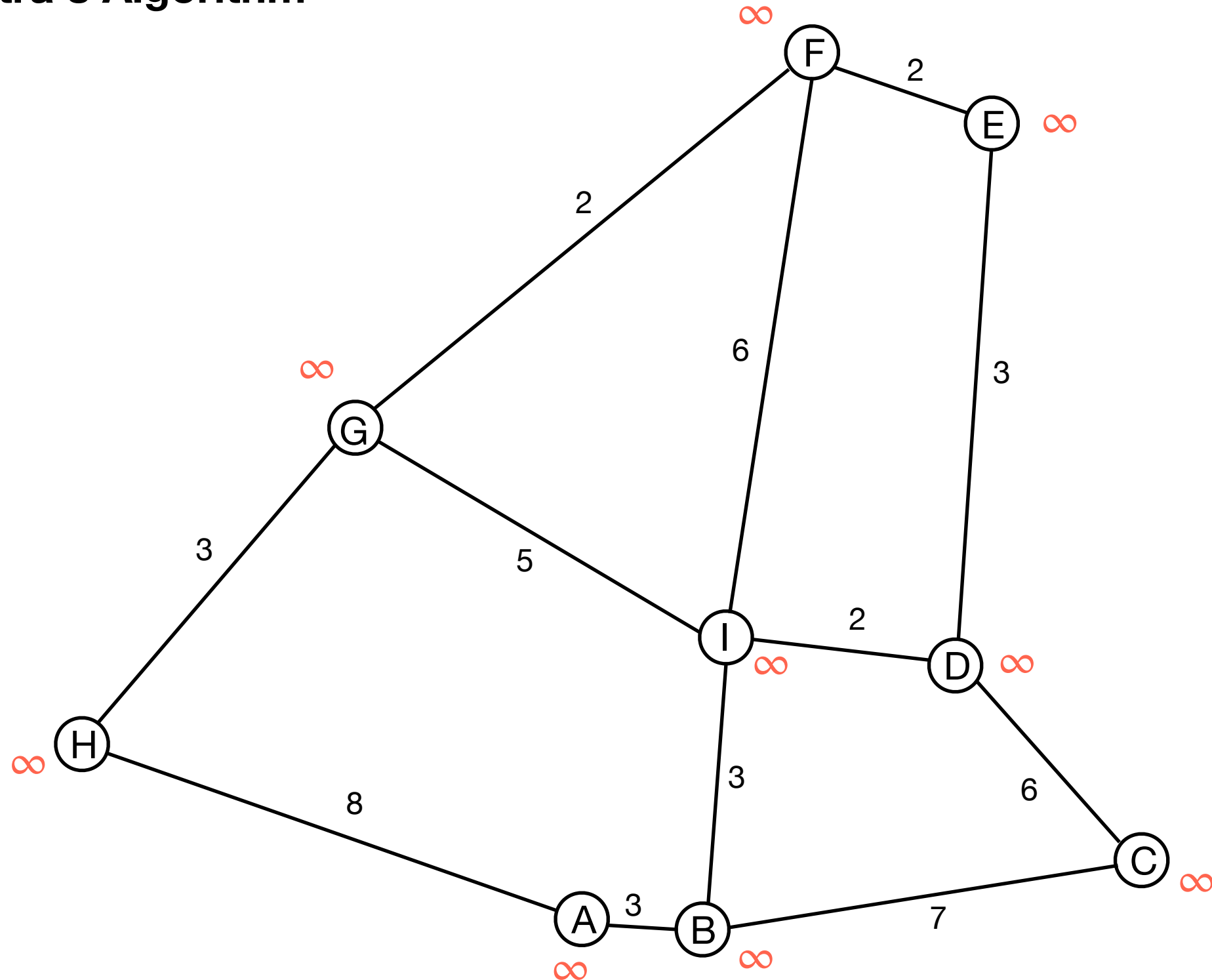
- Calculates the fastest driving routes by modelling roads as graph edges weighted dynamically by distance, speed limits, and live traffic congestion.
- Determines the most efficient pathways for internet data packets by creating a shortest-path tree based on real-time link bandwidth and latency.
- Discovers optimal, collision-free trajectories for autonomous vehicles and robotic arms by searching through a grid-mapped configuration space.
- Optimises delivery and shipping lanes by finding the lowest-cost paths between global distribution hubs to reduce fuel and transit times.
- Enables non-player characters to navigate dynamic game maps intelligently by computing the shortest path around walls and hazardous terrain.

Dijkstra's Algorithm



1. Initialise a value at each vertex to ∞ . Call these values $d[i]$

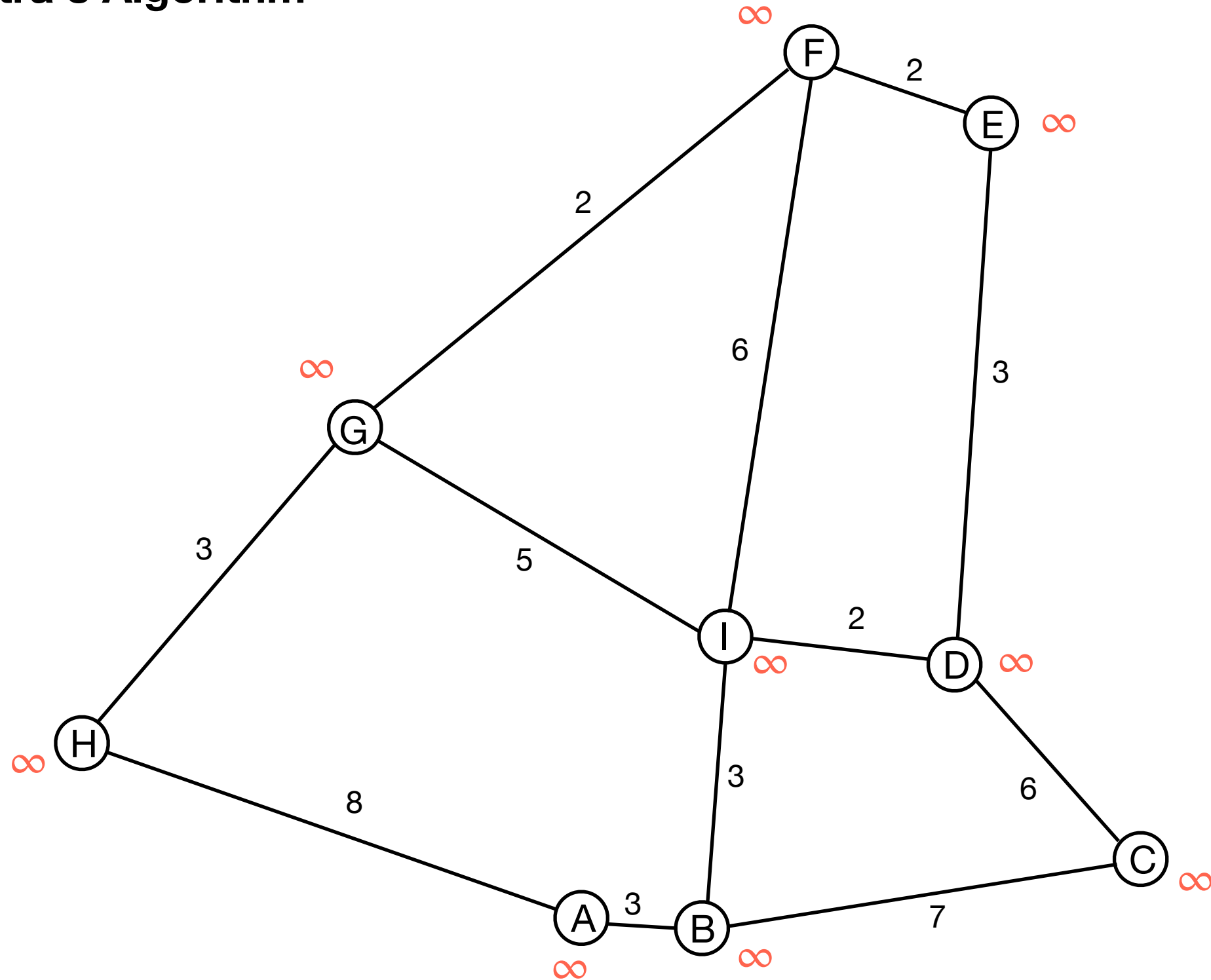
Dijkstra's Algorithm



1. Initialise a value at each vertex to ∞ . Call these values $d[i]$

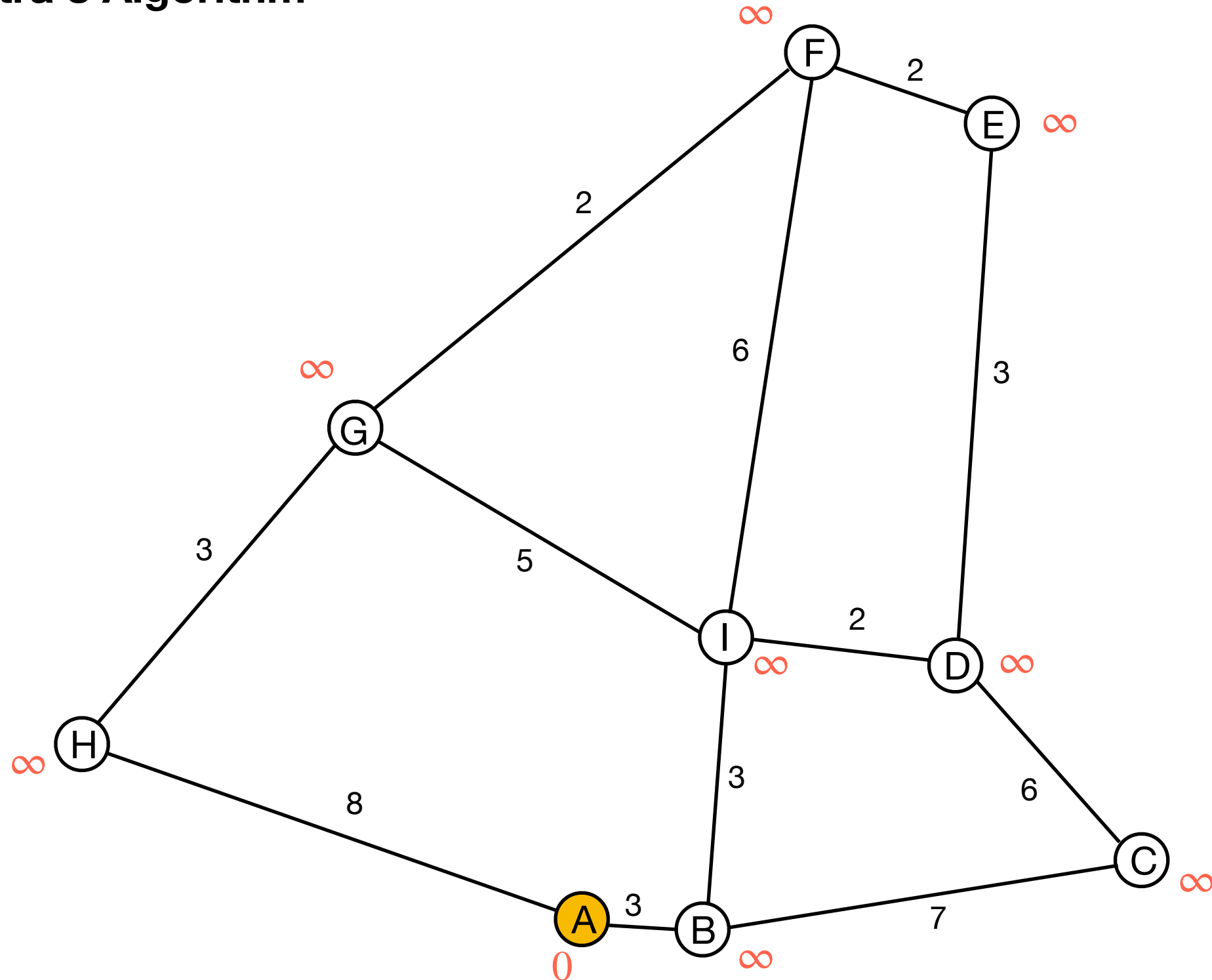
These ∞ values represent the cost of reaching each vertex from our source, using only intermediary vertices whose shortest paths we have already found.

Dijkstra's Algorithm



2. Initialise the value at our source vertex to zero and mark the source vertex as visited

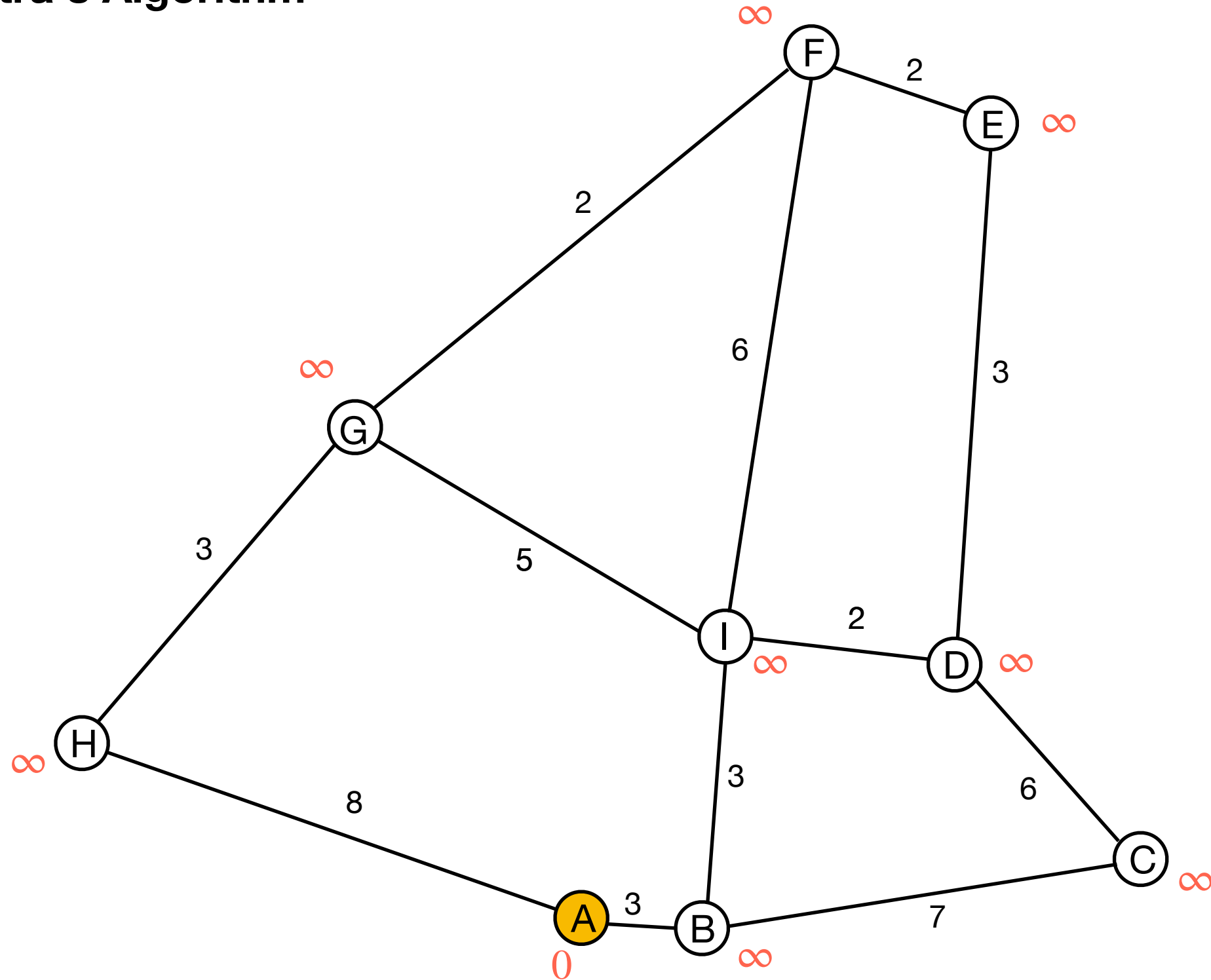
Dijkstra's Algorithm



2. Initialise the value at our source vertex to zero and mark the source vertex as visited

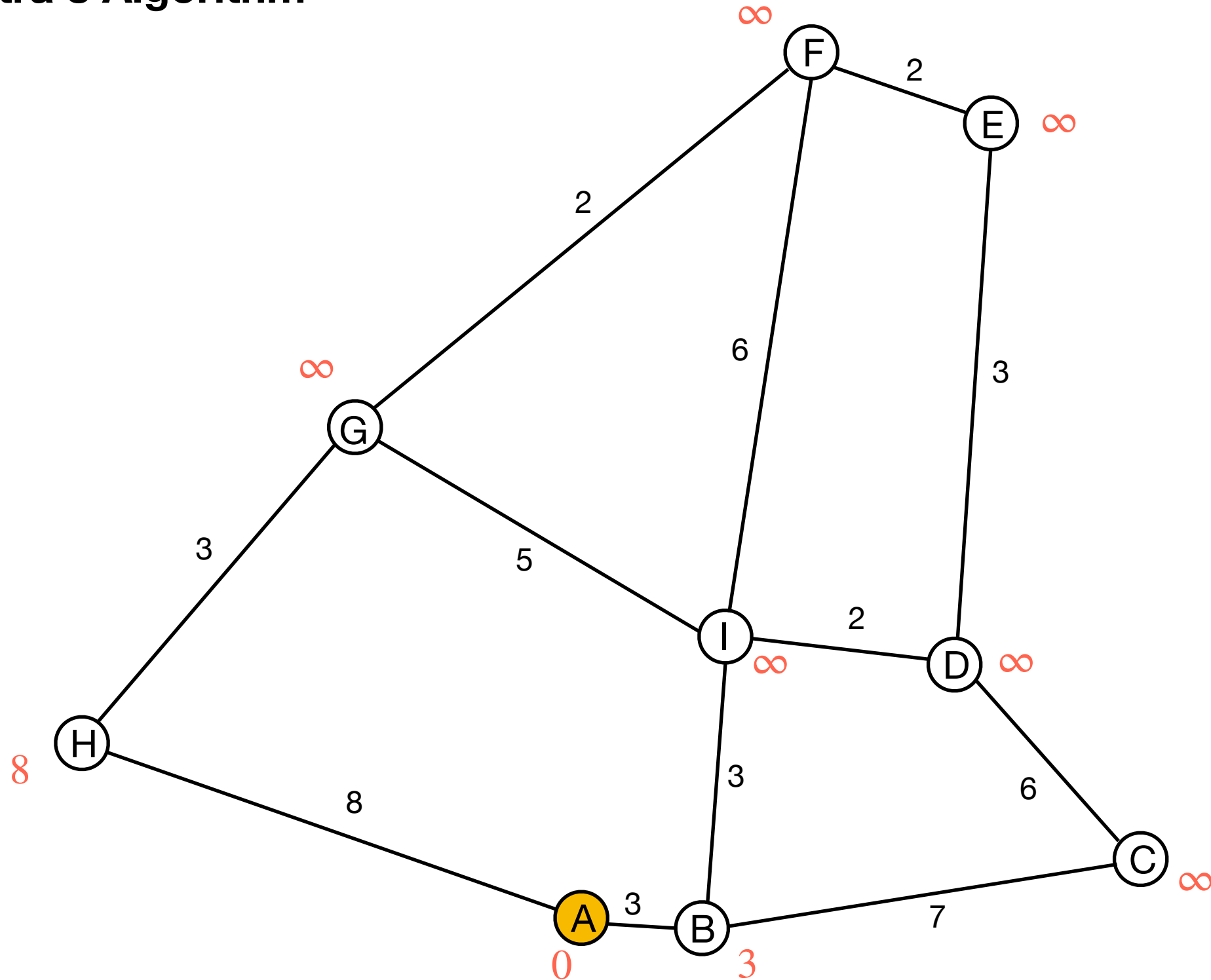
Clearly we can get from vertex A to vertex A at zero cost

Dijkstra's Algorithm



3. Update the cost of getting from the source vertex to every other vertex: $\min \{ \text{weight}[\text{source}, i], d[i] \}$

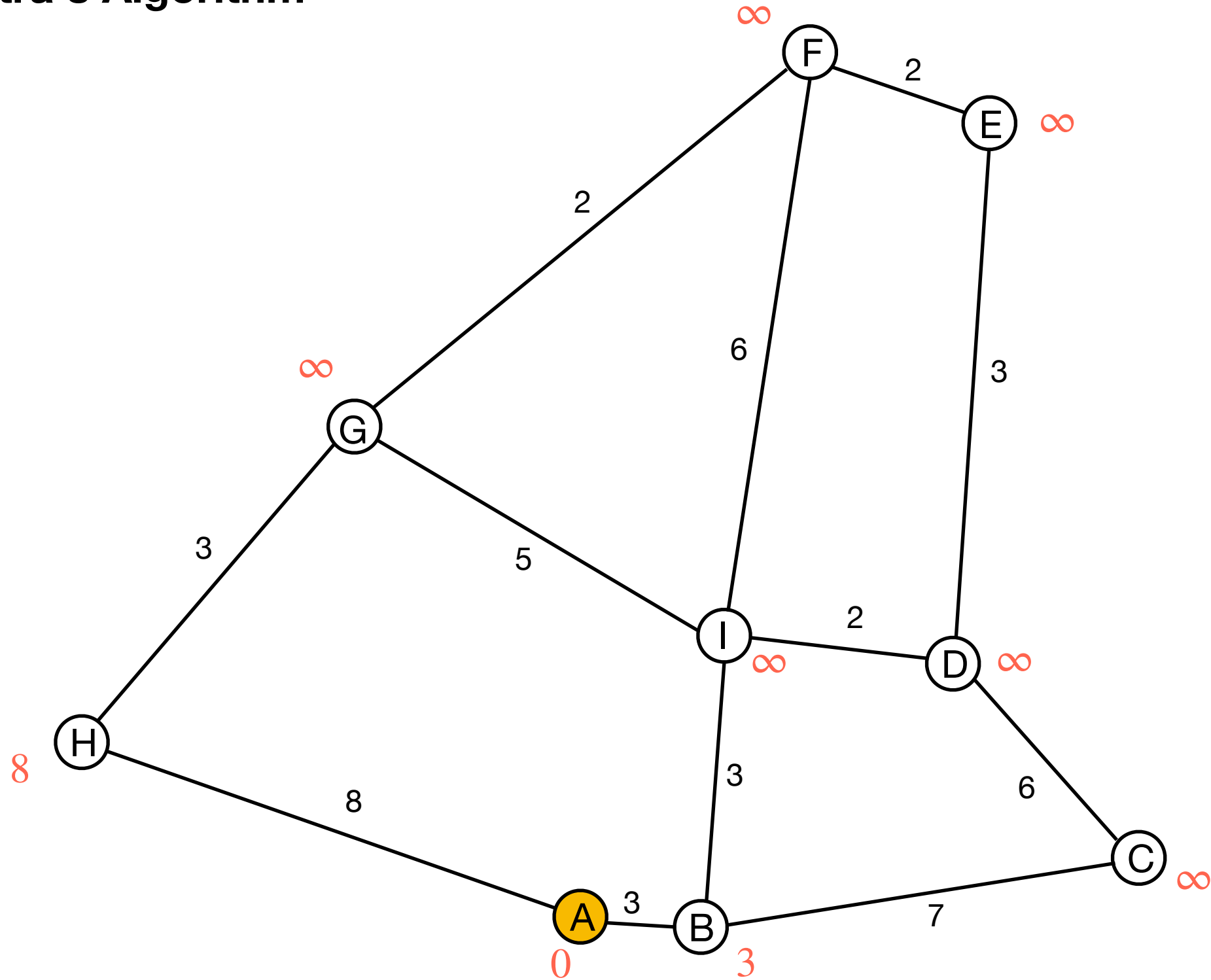
Dijkstra's Algorithm



3. Update the cost of getting from the source vertex to every other vertex: $\min \{ \text{weight}[\text{source}, i], d[i] \}$

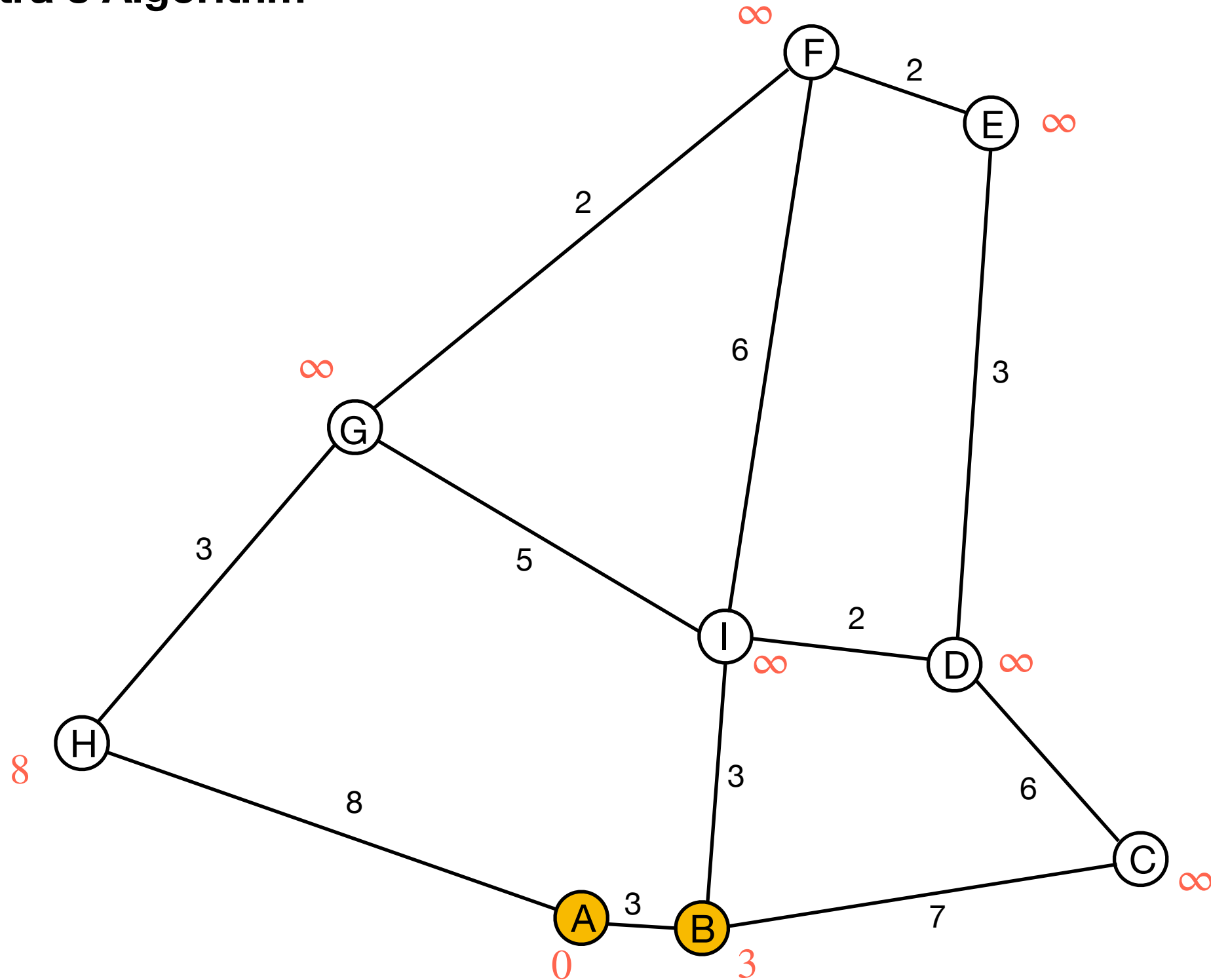
These $d[i]$ values now represent the lowest-cost path from A using no intermediate vertices

Dijkstra's Algorithm



4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it

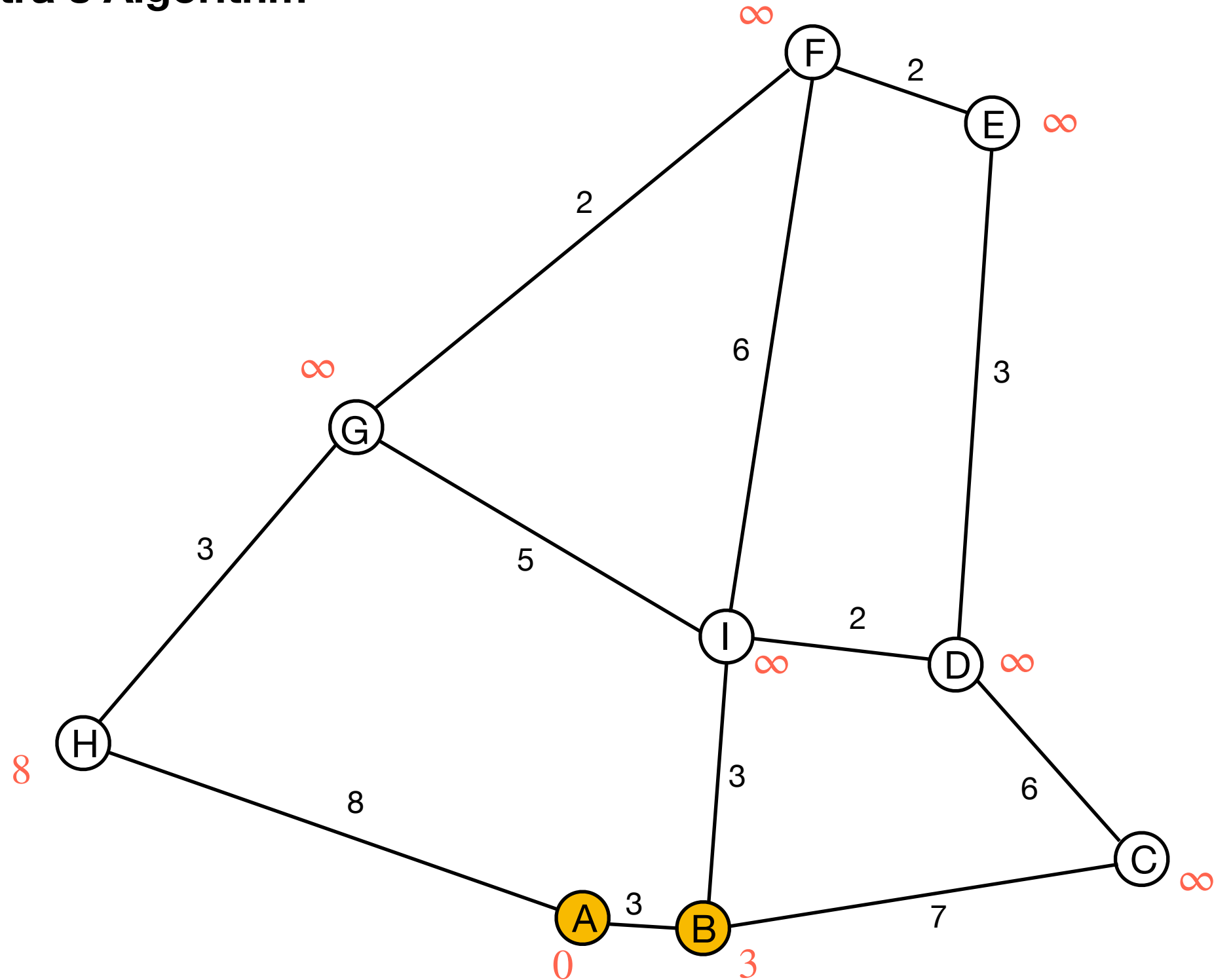
Dijkstra's Algorithm



4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it

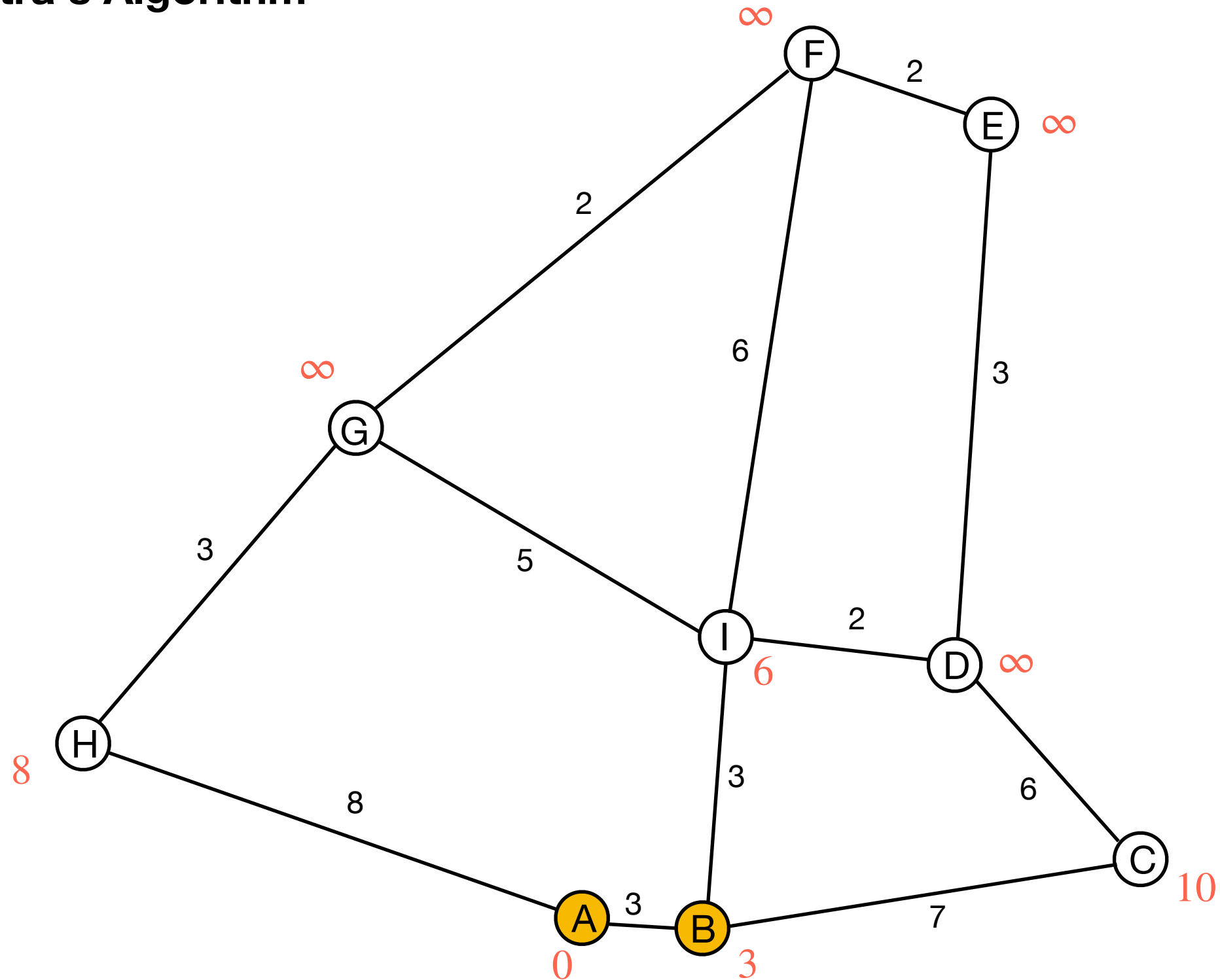
Clearly we have found the shortest path from vertex A to vertex B, since any other path must go through edges of greater(or equal) weight

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$

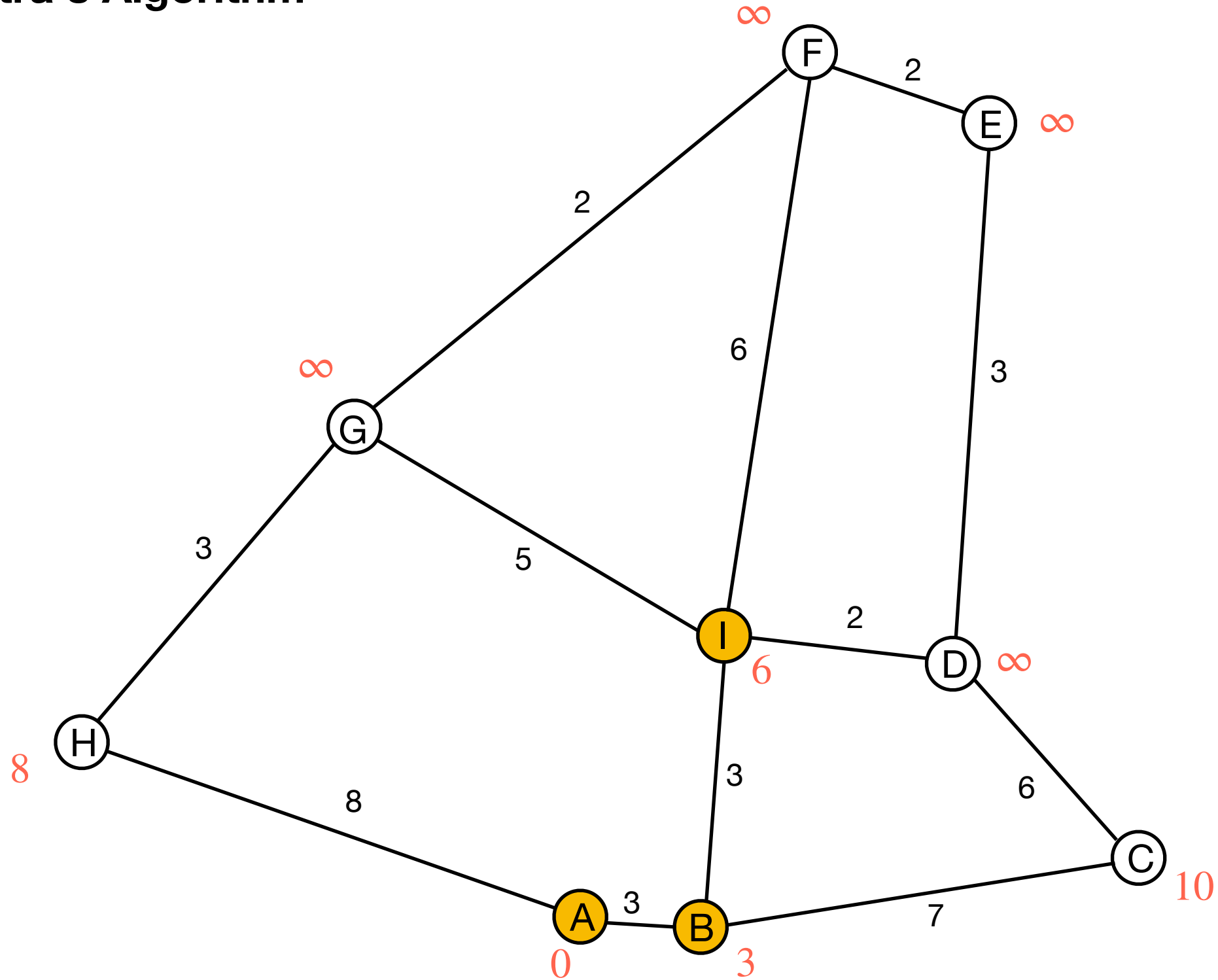
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$

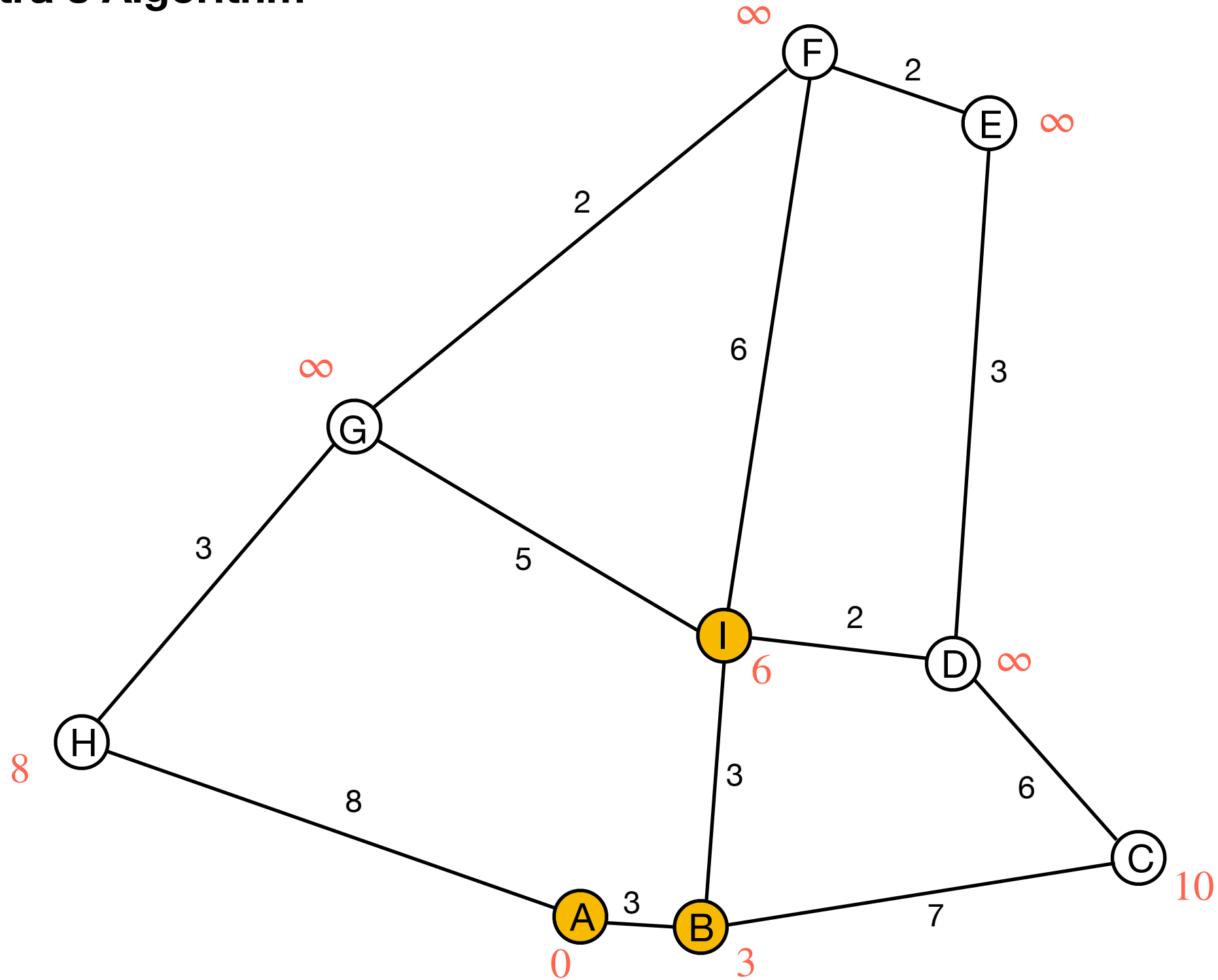
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B to be used as an intermediary vertex along the path

Dijkstra's Algorithm



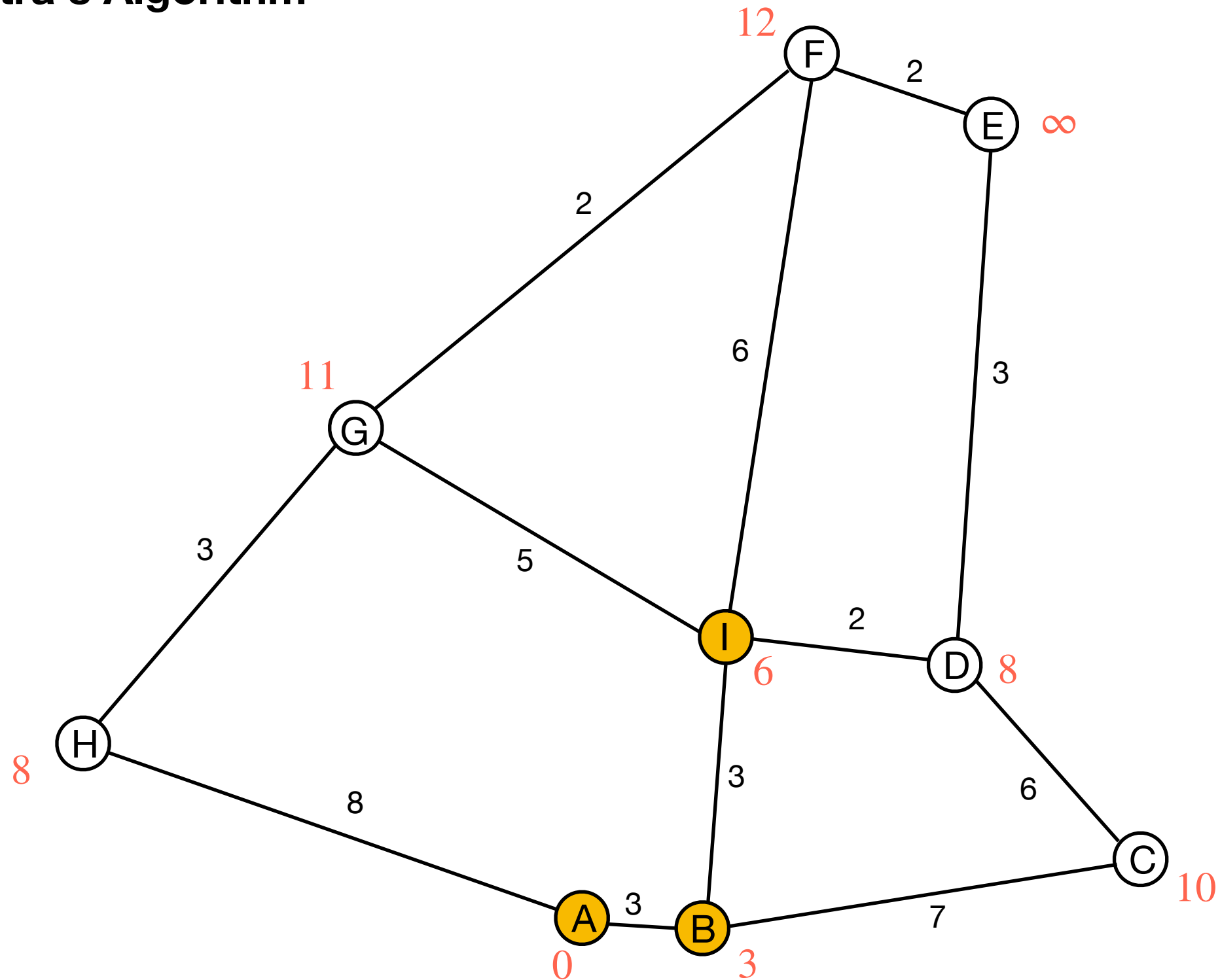
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

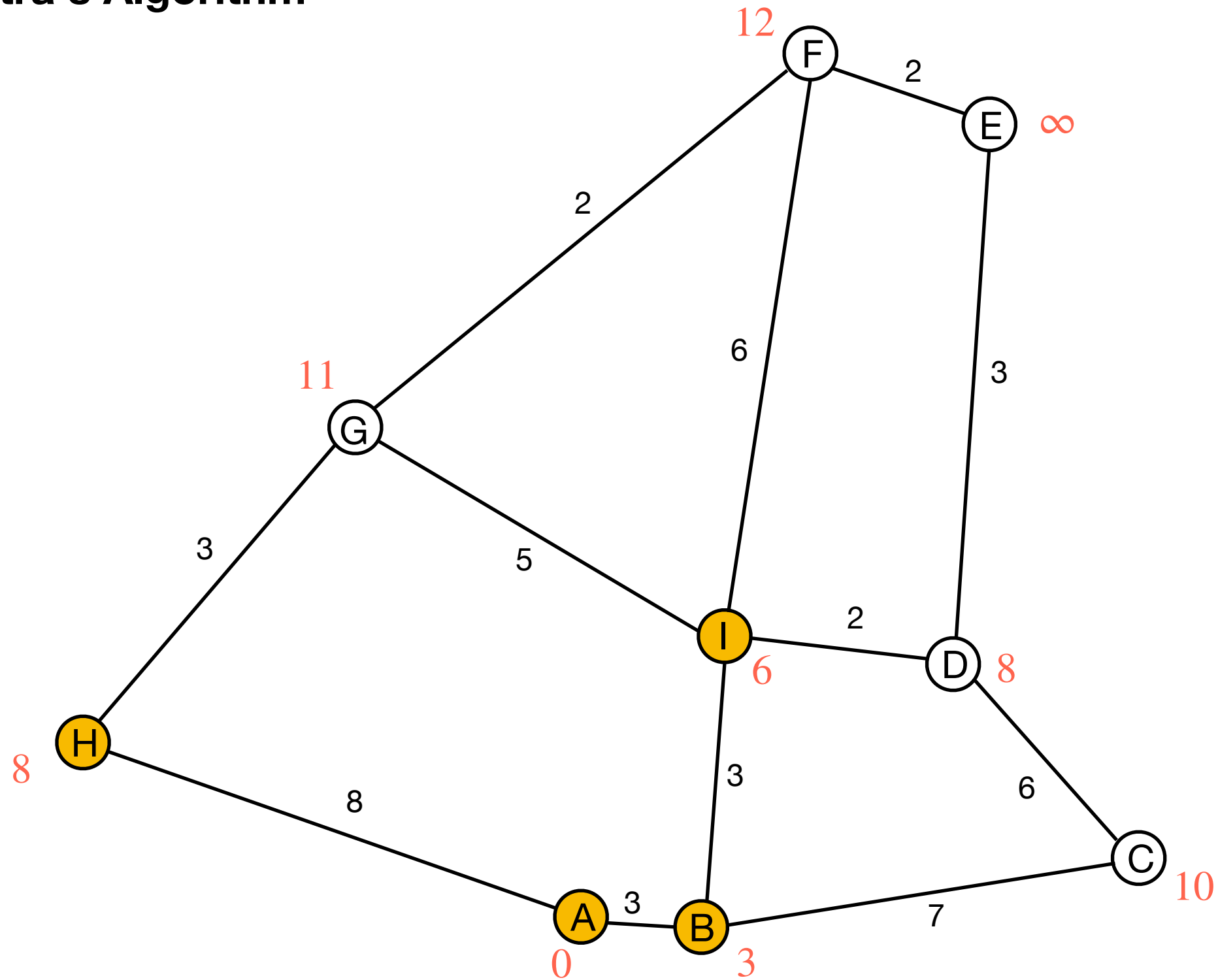
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

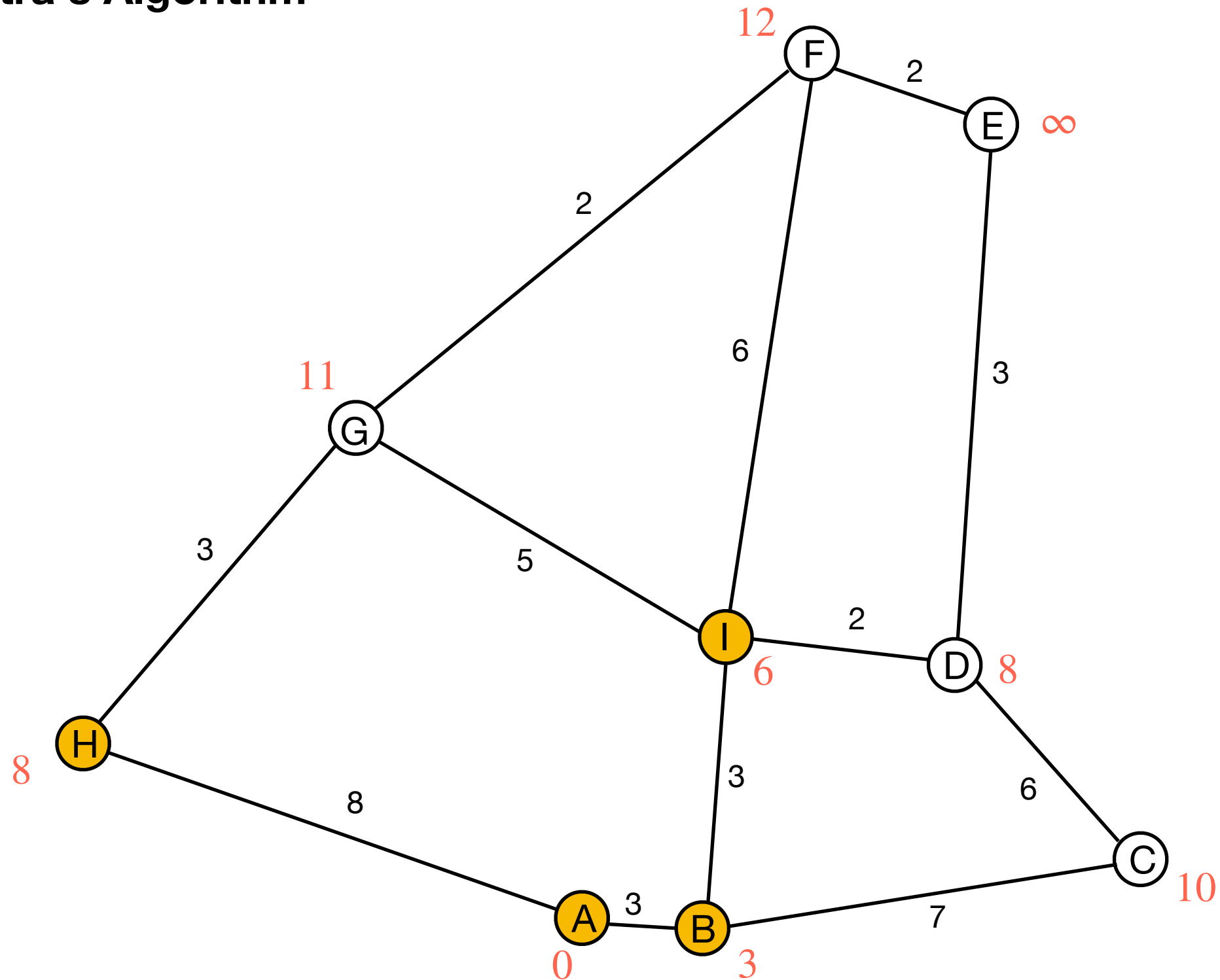
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B and I to be used as an intermediary vertex along the path

Dijkstra's Algorithm



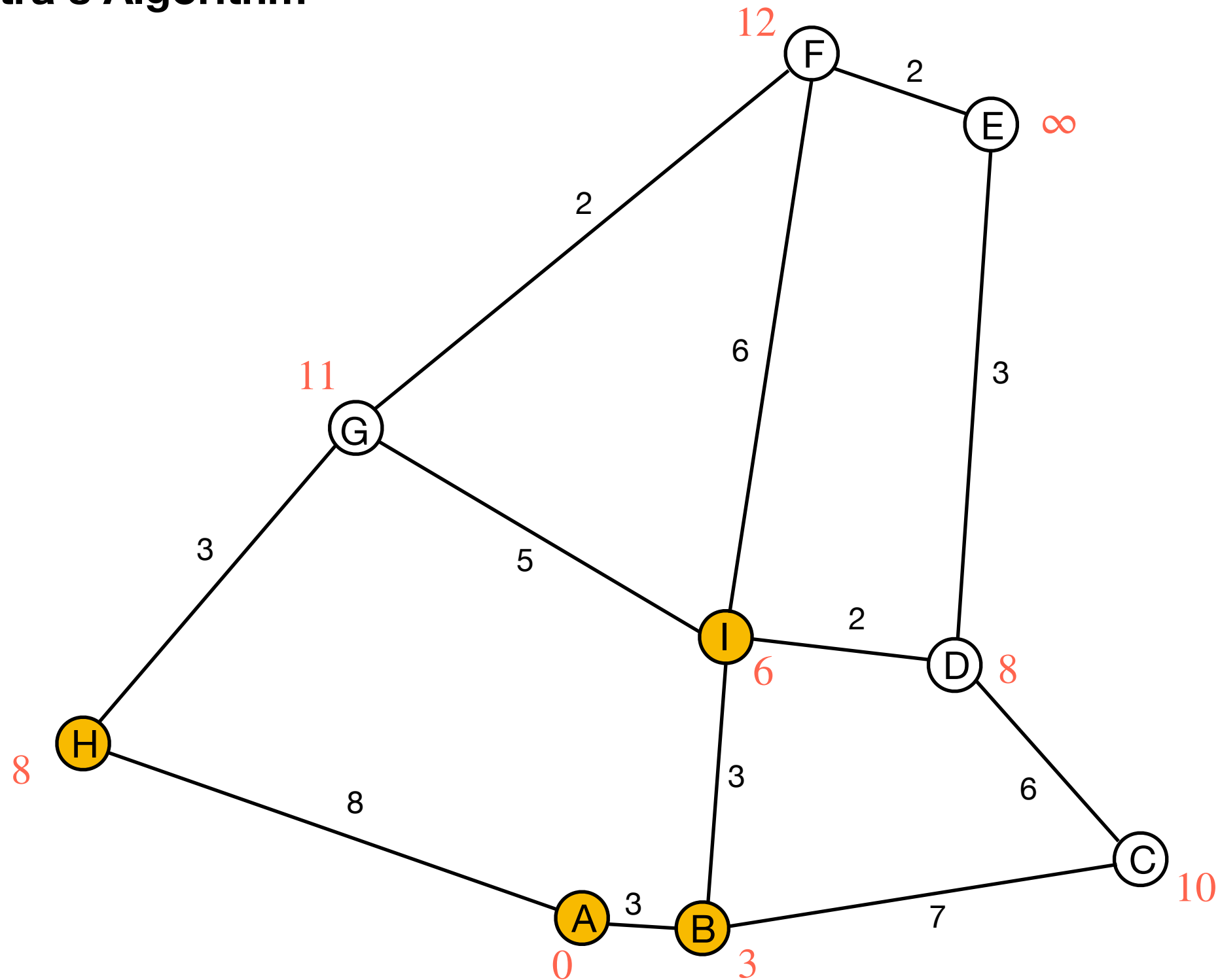
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

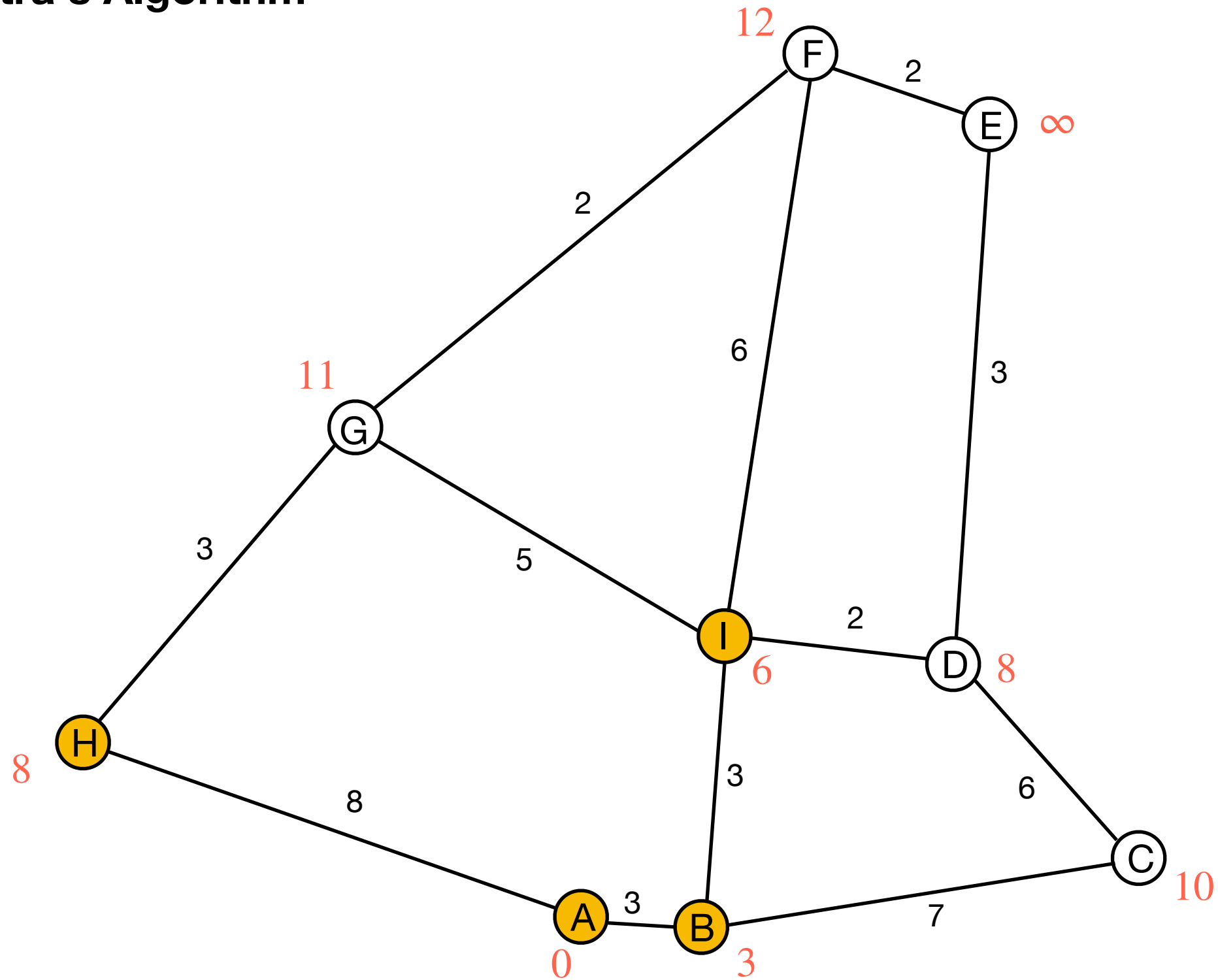
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

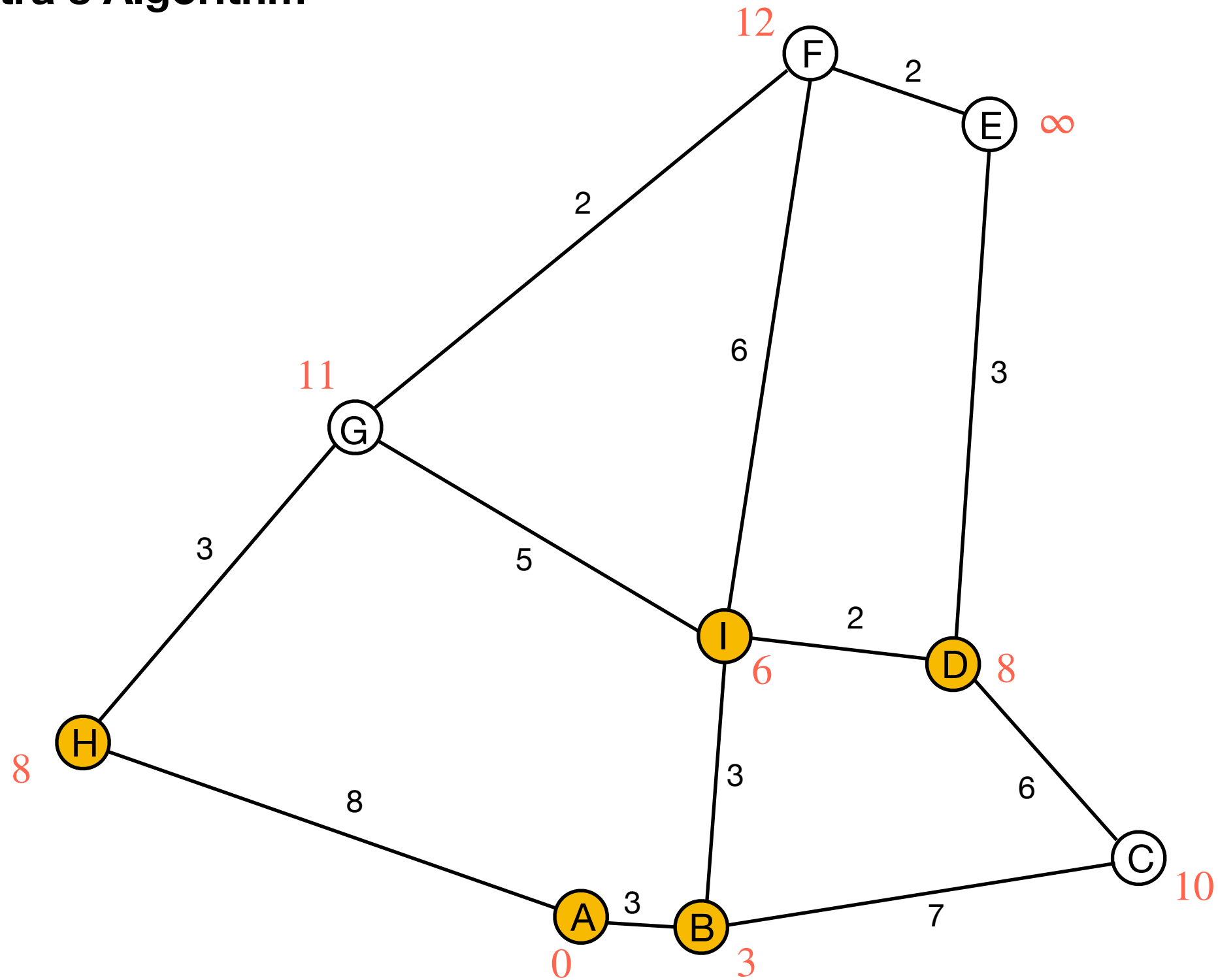
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B, I and H to be used as an intermediary vertex along the path

Dijkstra's Algorithm



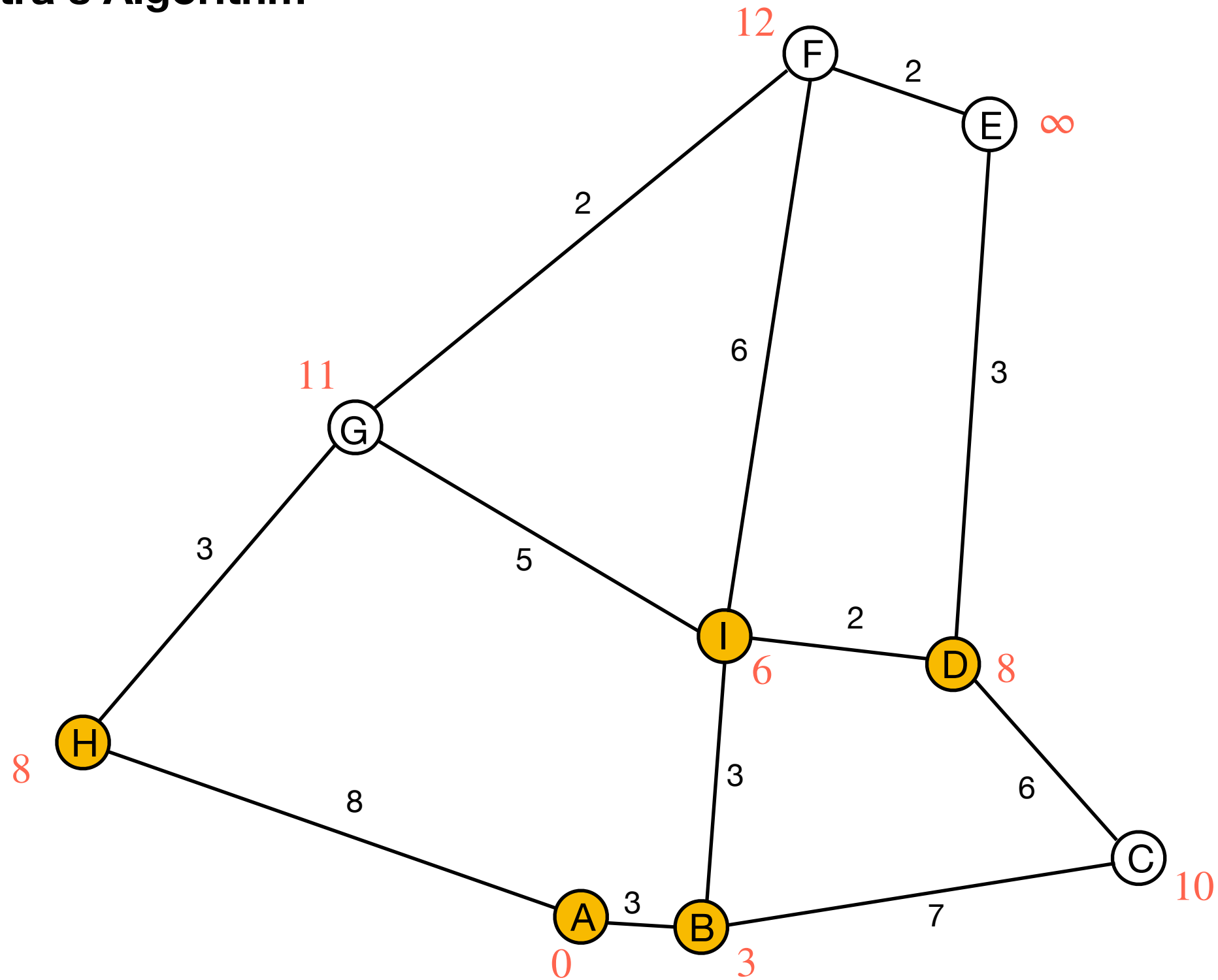
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



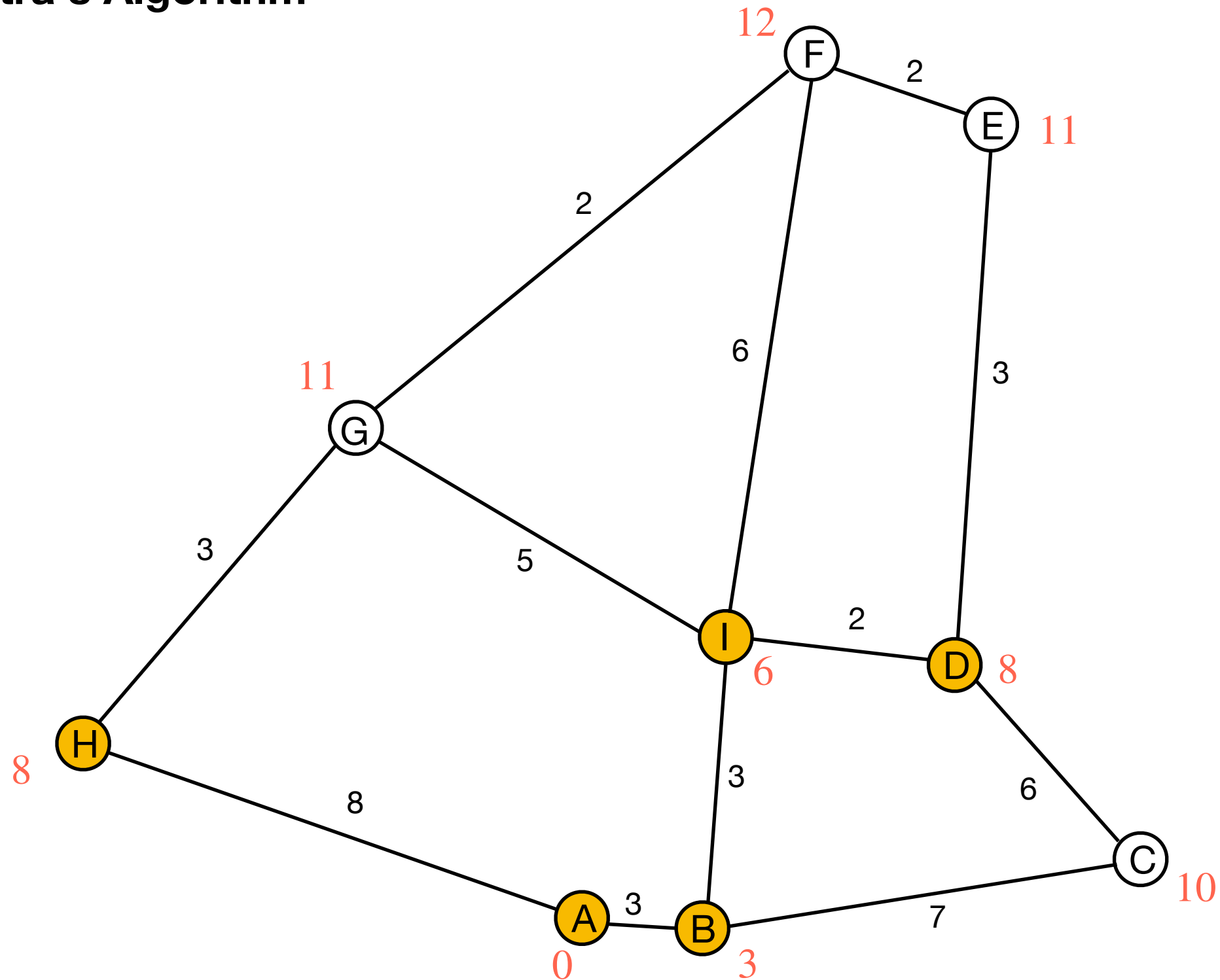
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

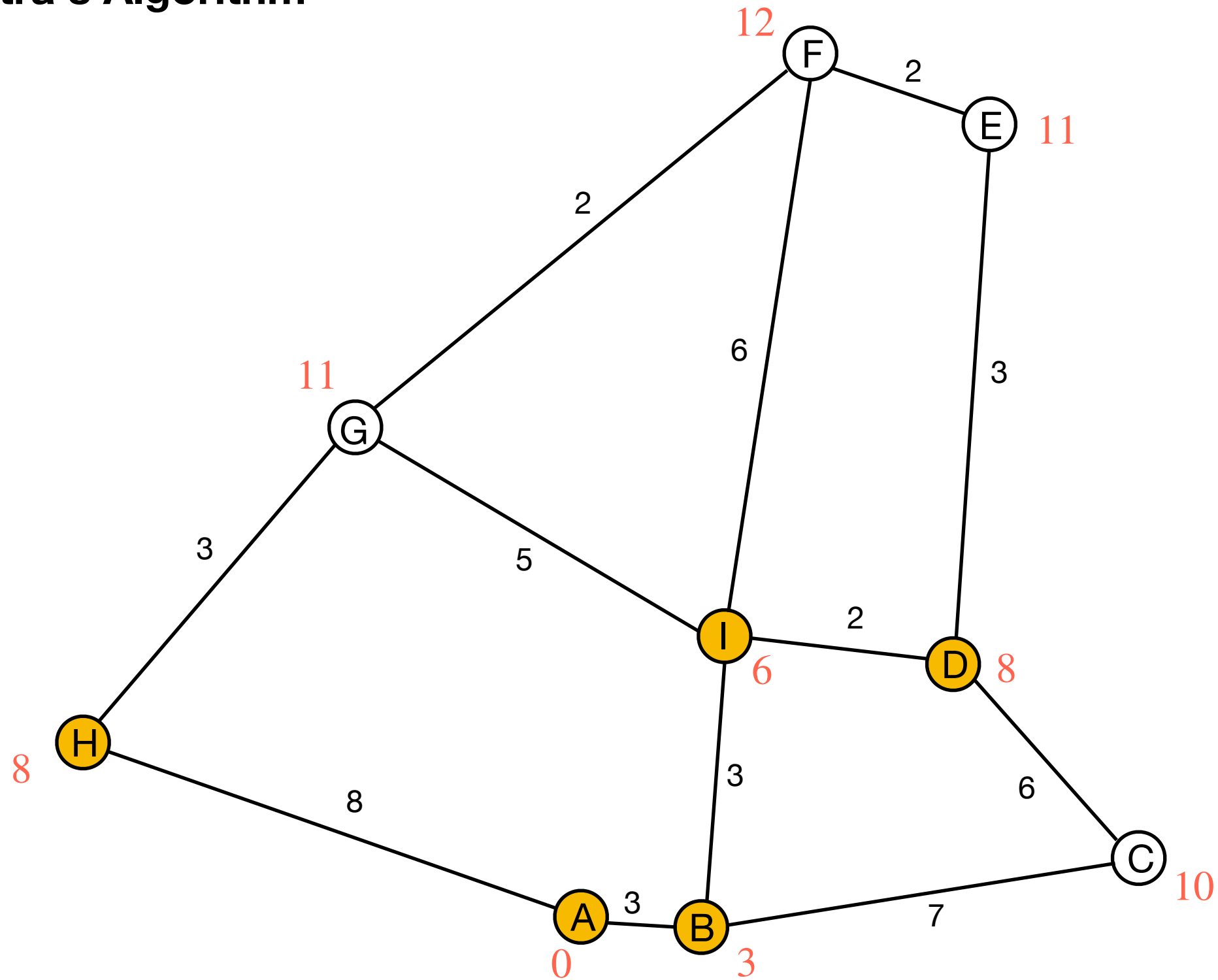
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

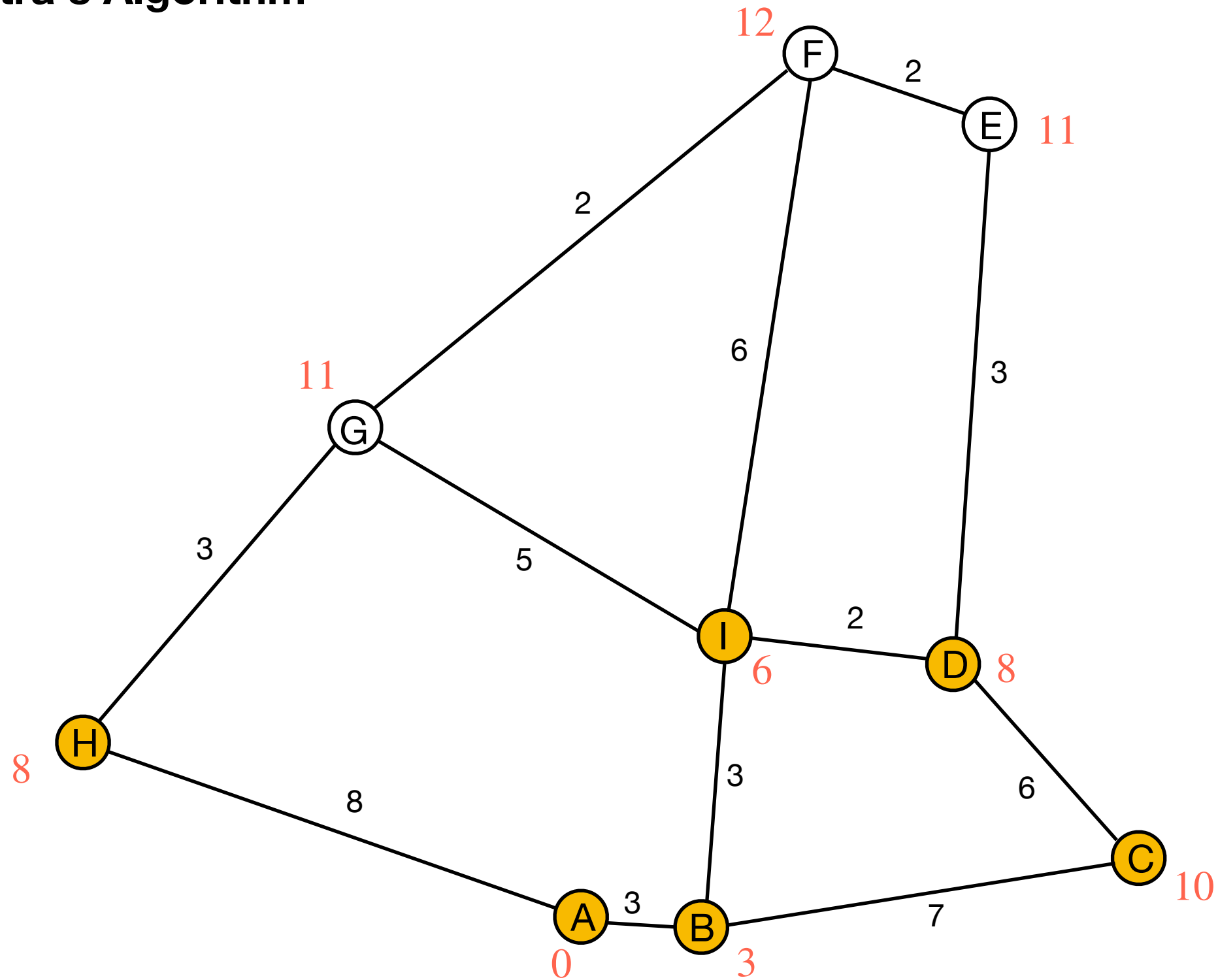
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B, I, H and D to be used as an intermediary vertex along the path

Dijkstra's Algorithm



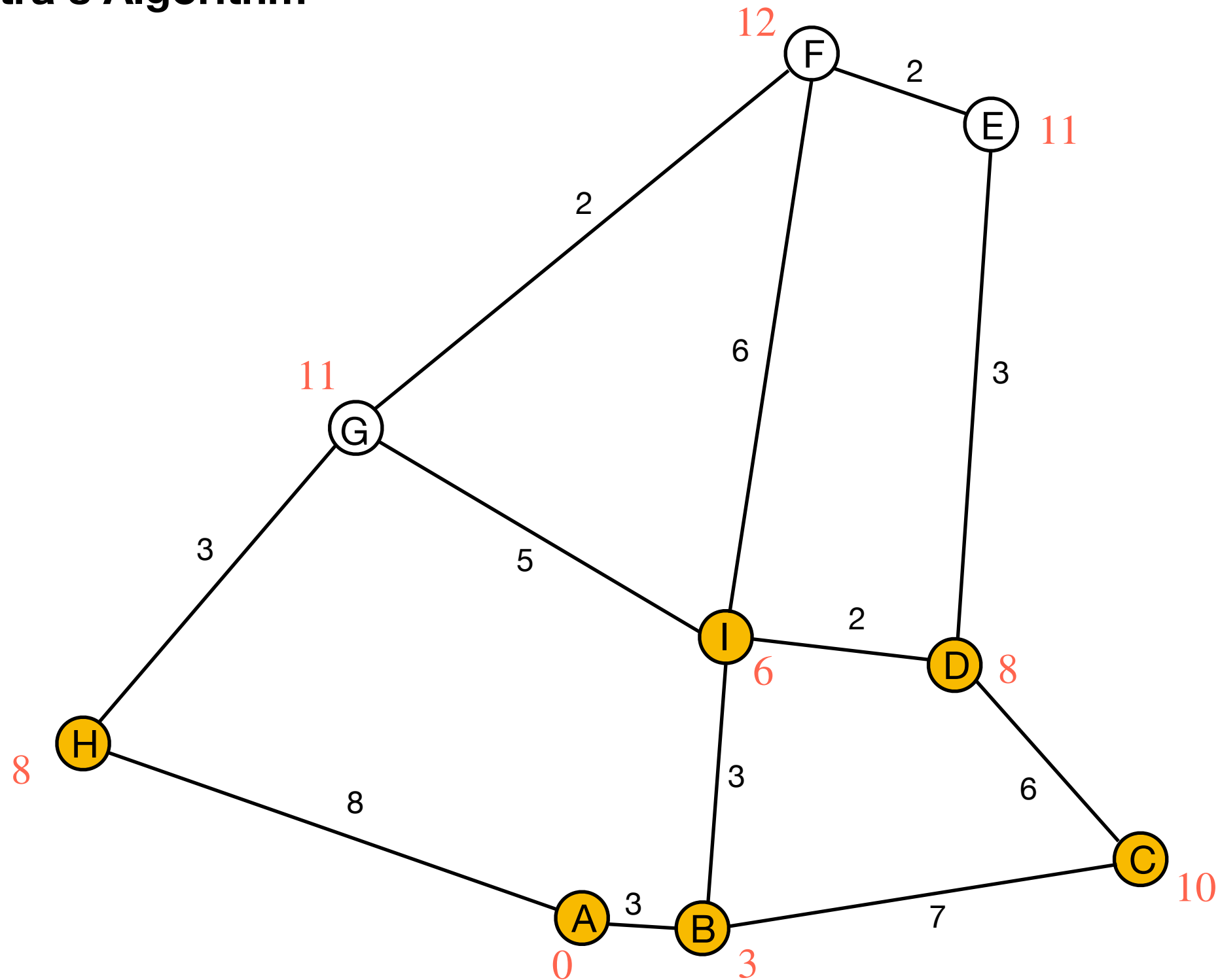
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



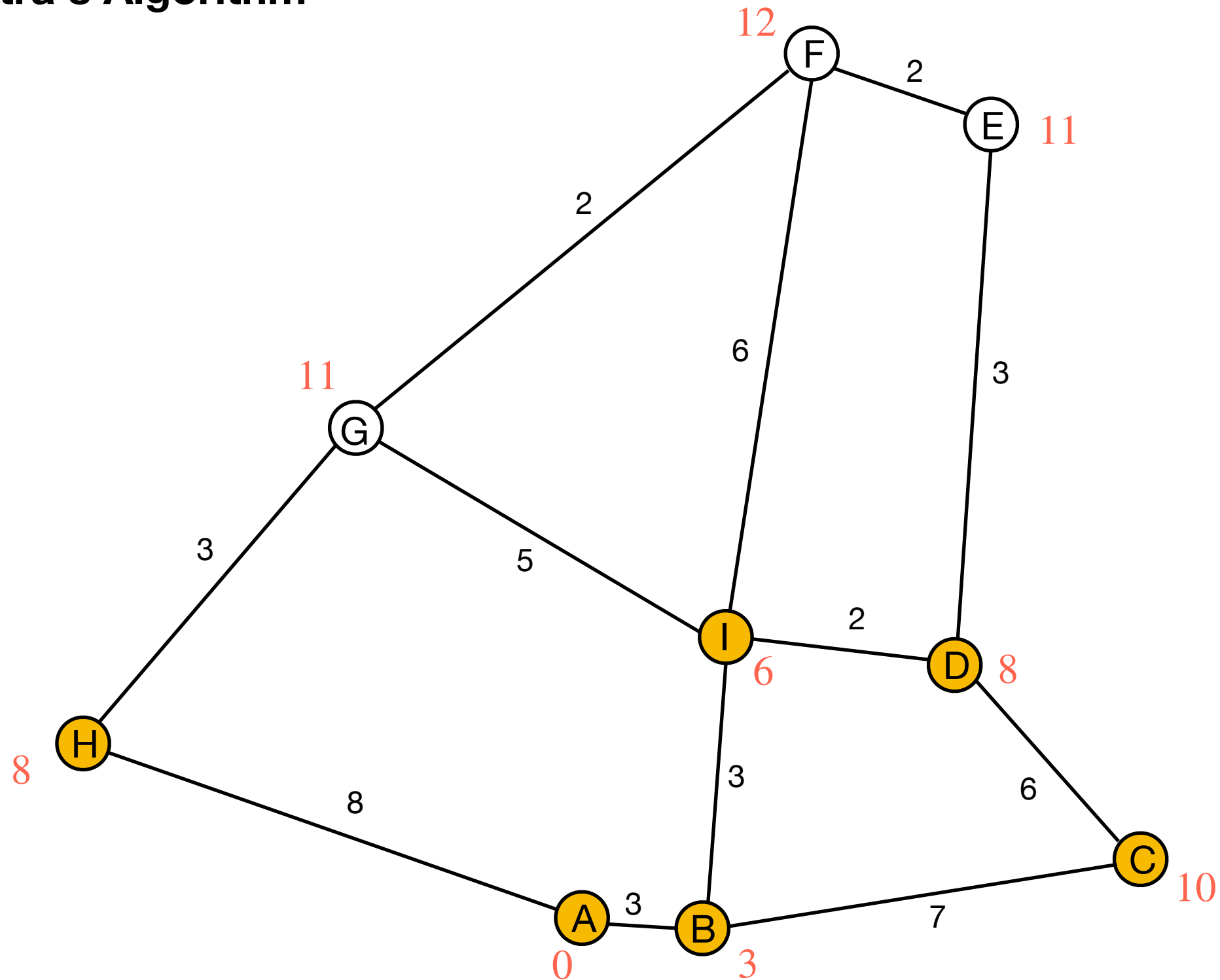
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

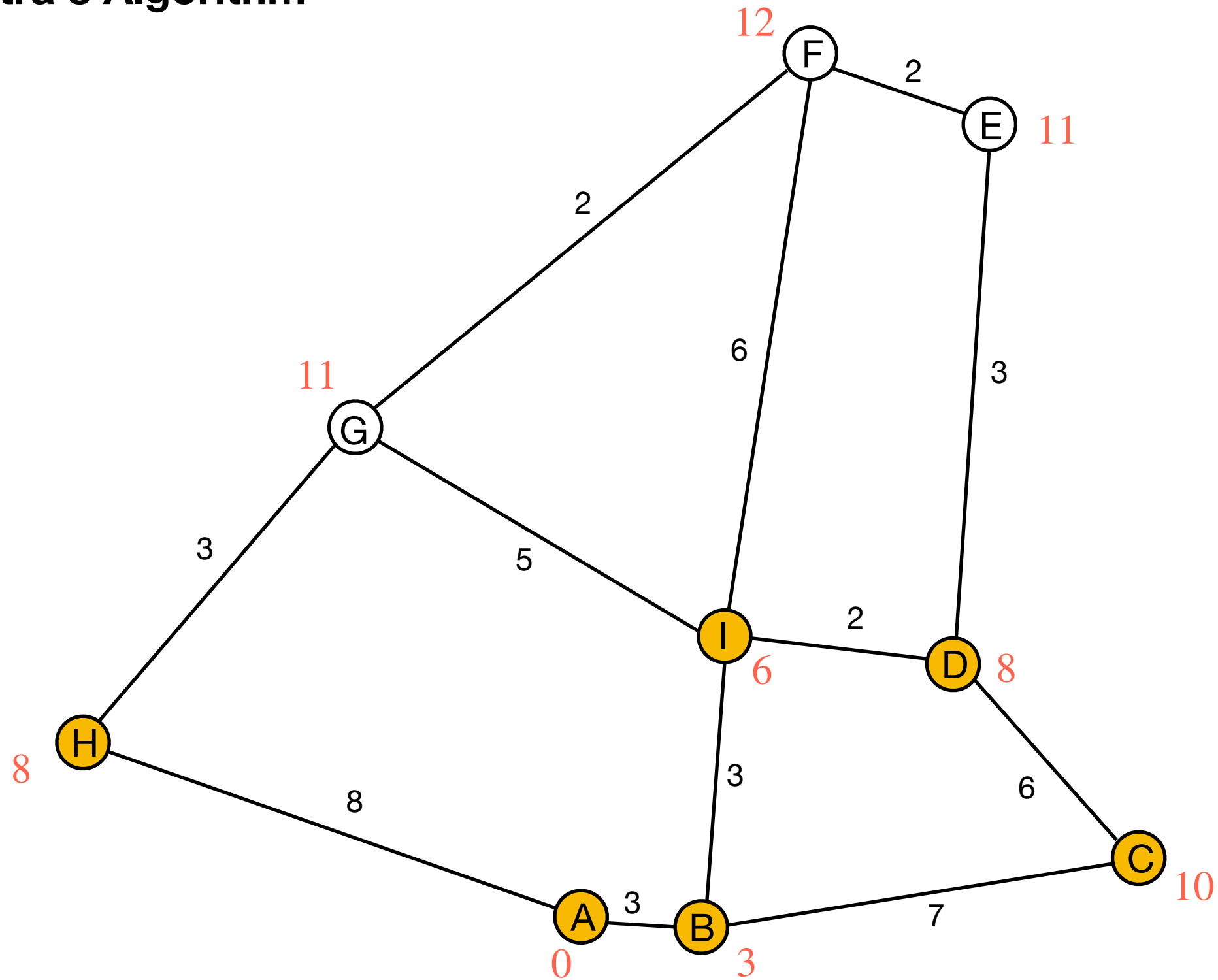
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

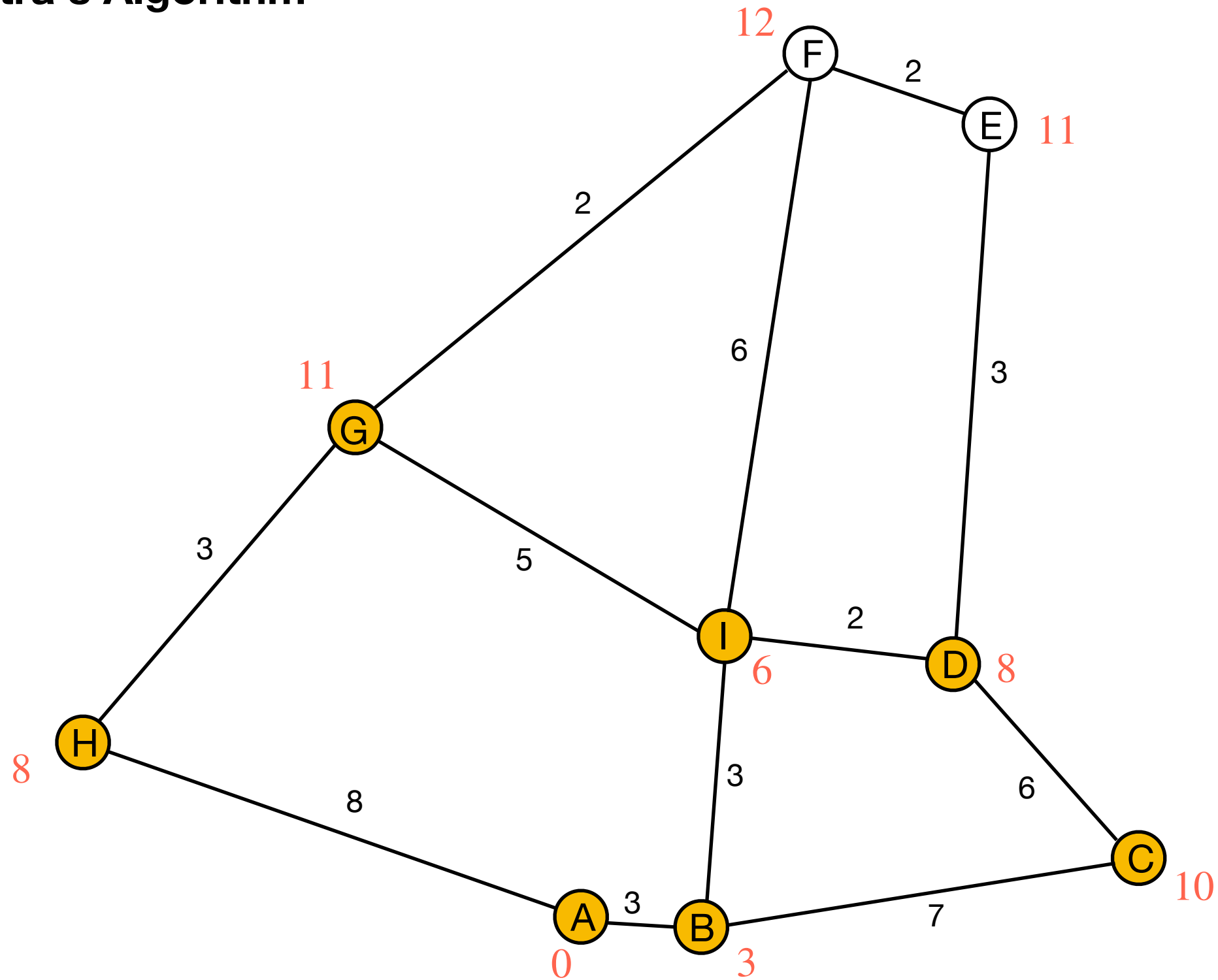
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B, I, H, D and C to be used as an intermediary vertex along the path

Dijkstra's Algorithm



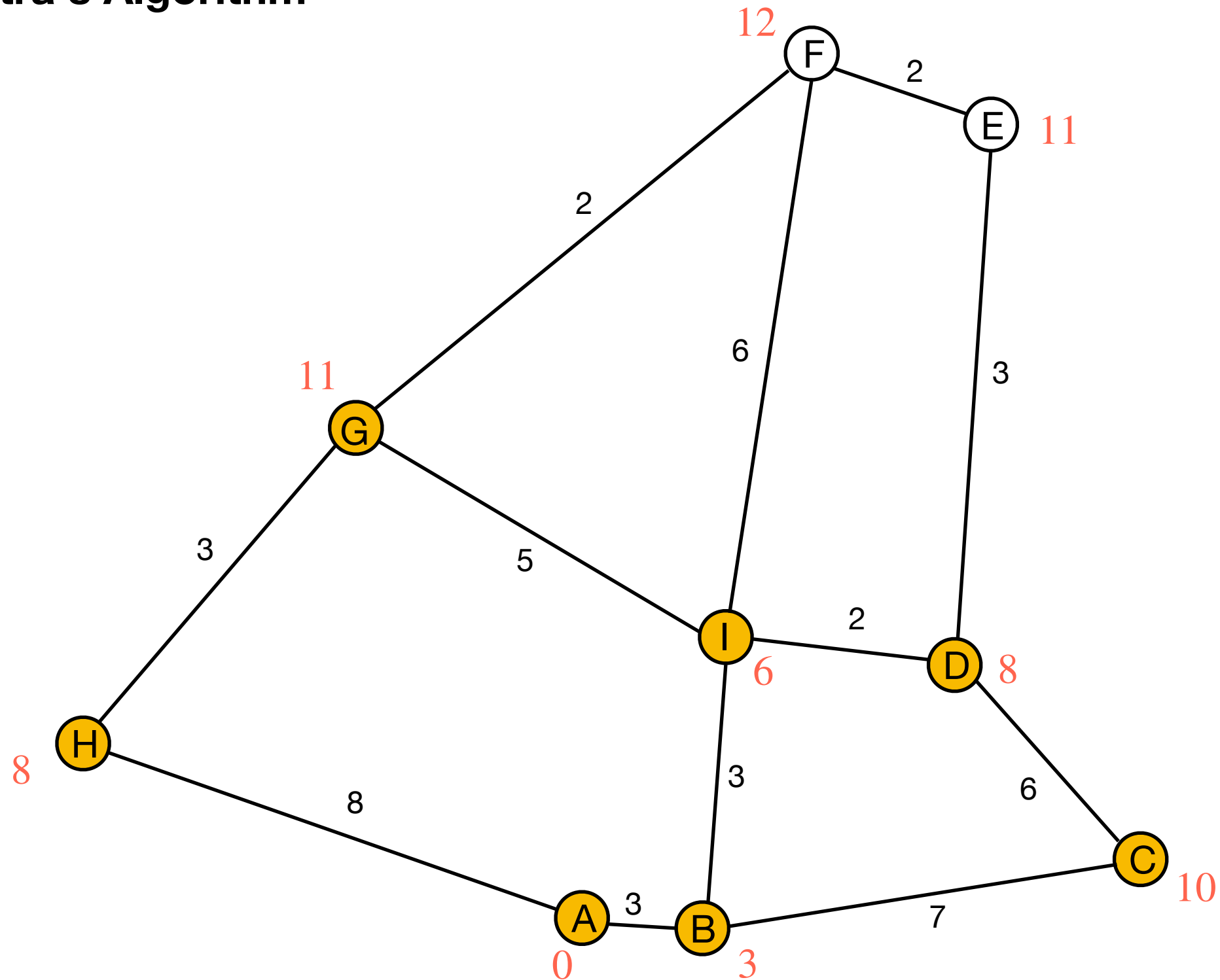
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



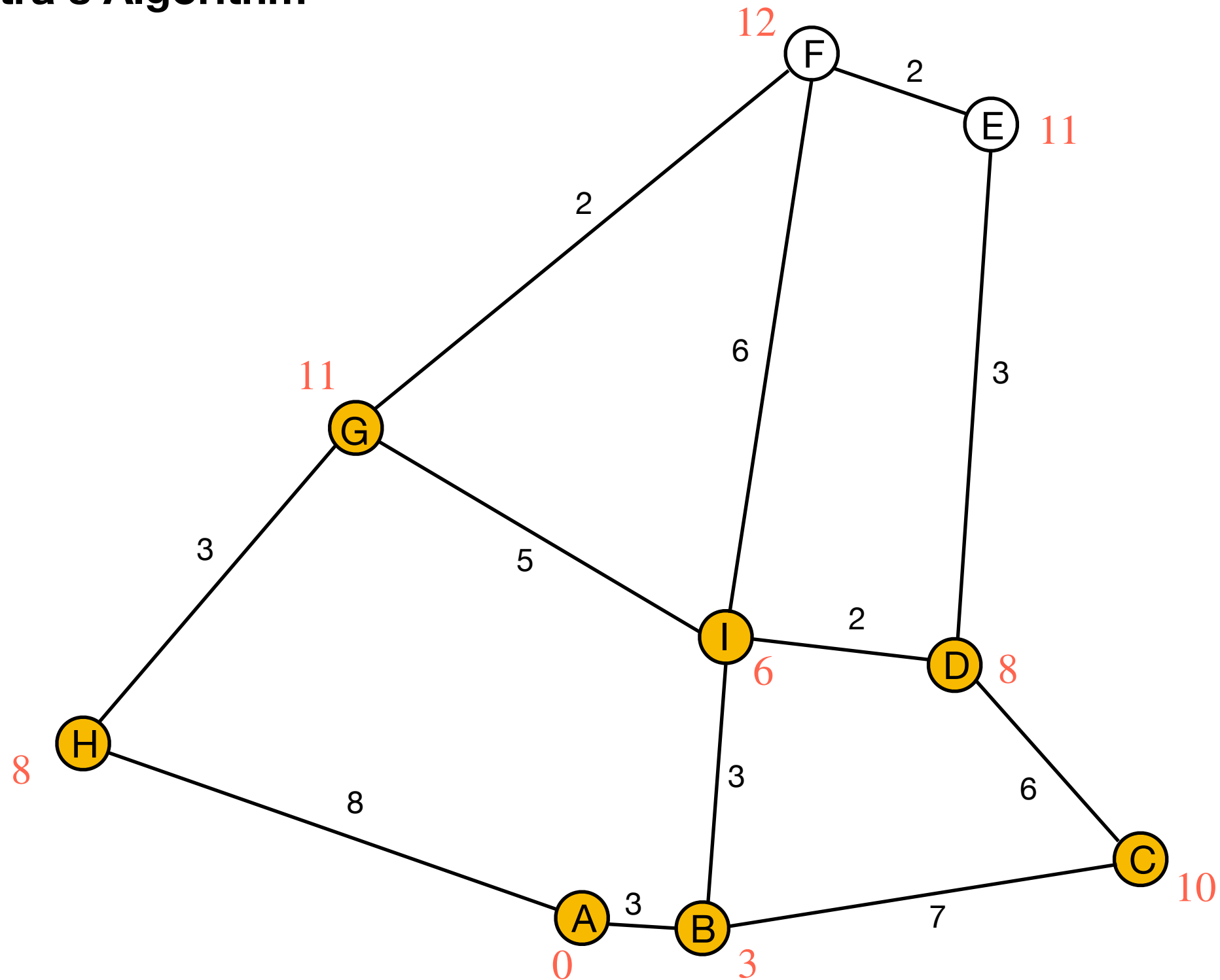
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

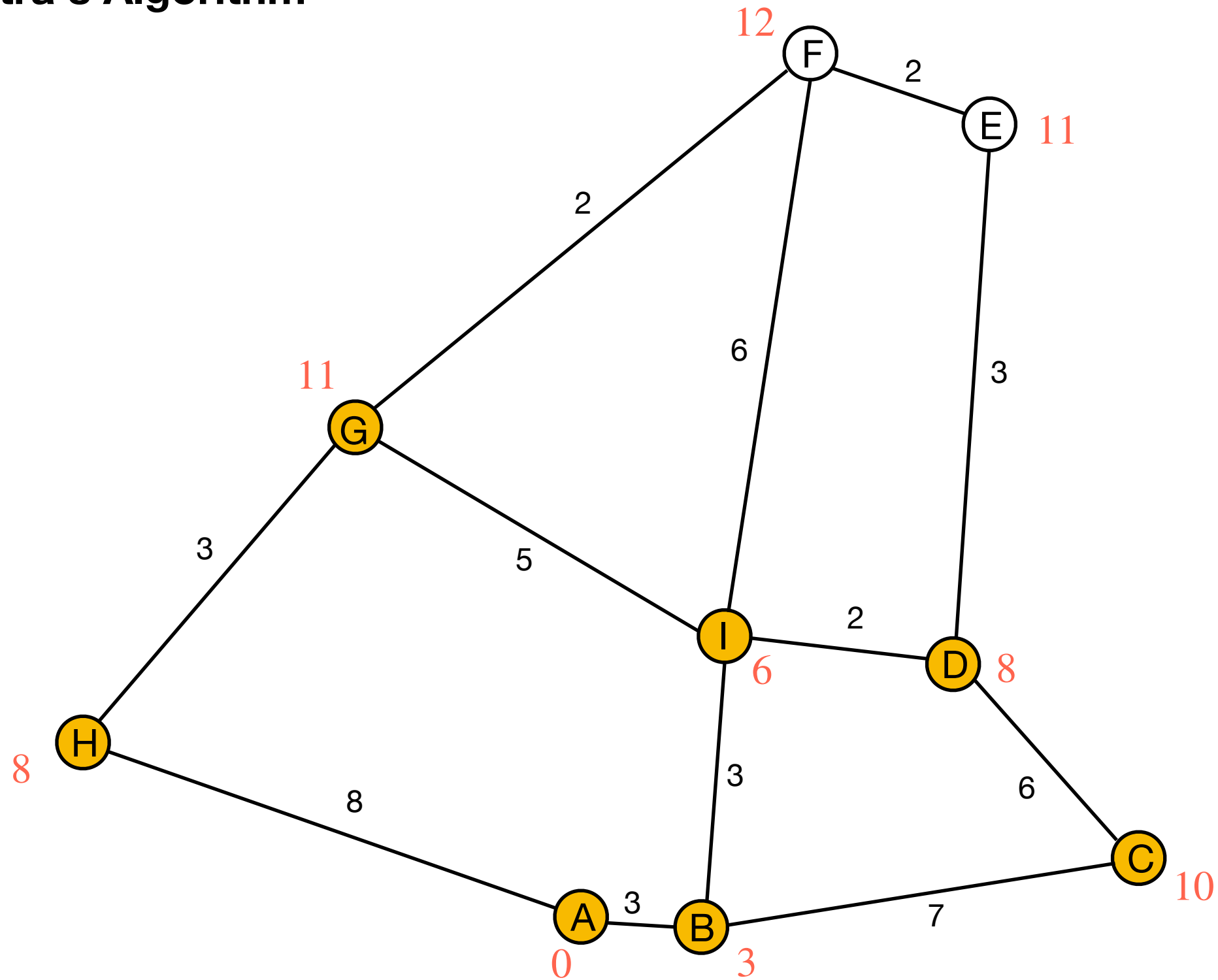
Dijkstra's Algorithm



5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

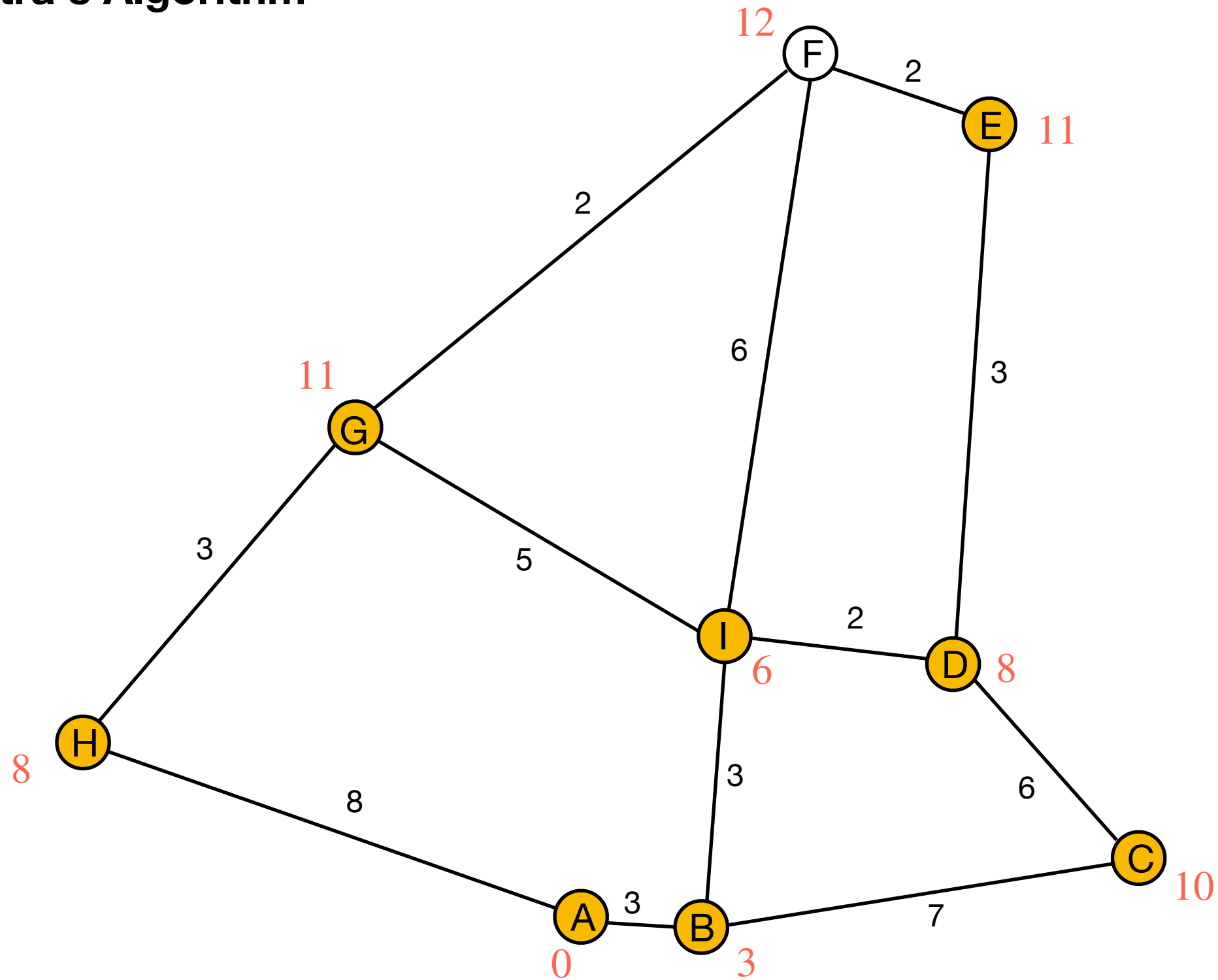
The $d[i]$ values now indicate the lowest cost path from vertex A if we allow vertex B, I, H, D, C and G to be used as an intermediary vertex along the path

Dijkstra's Algorithm



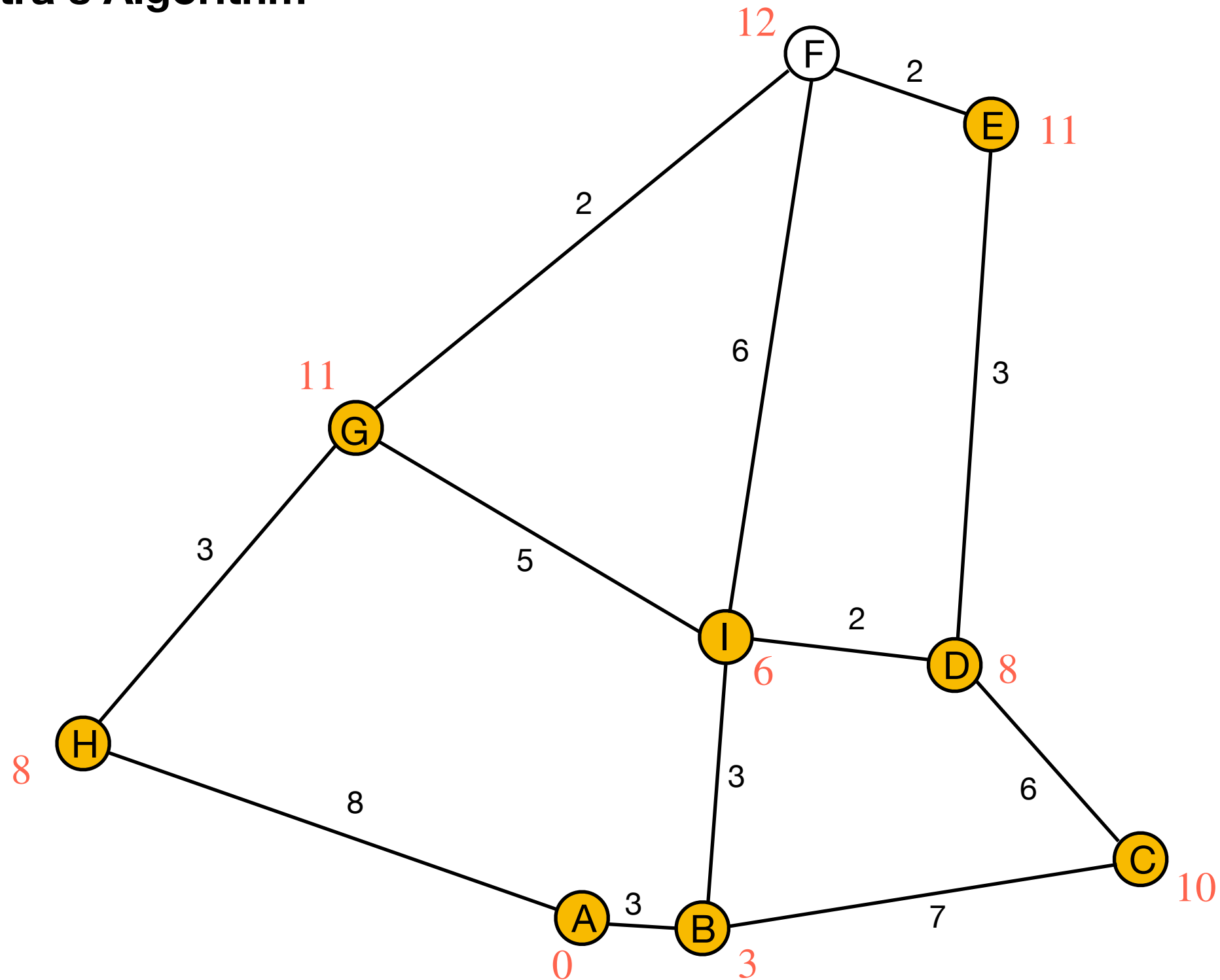
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



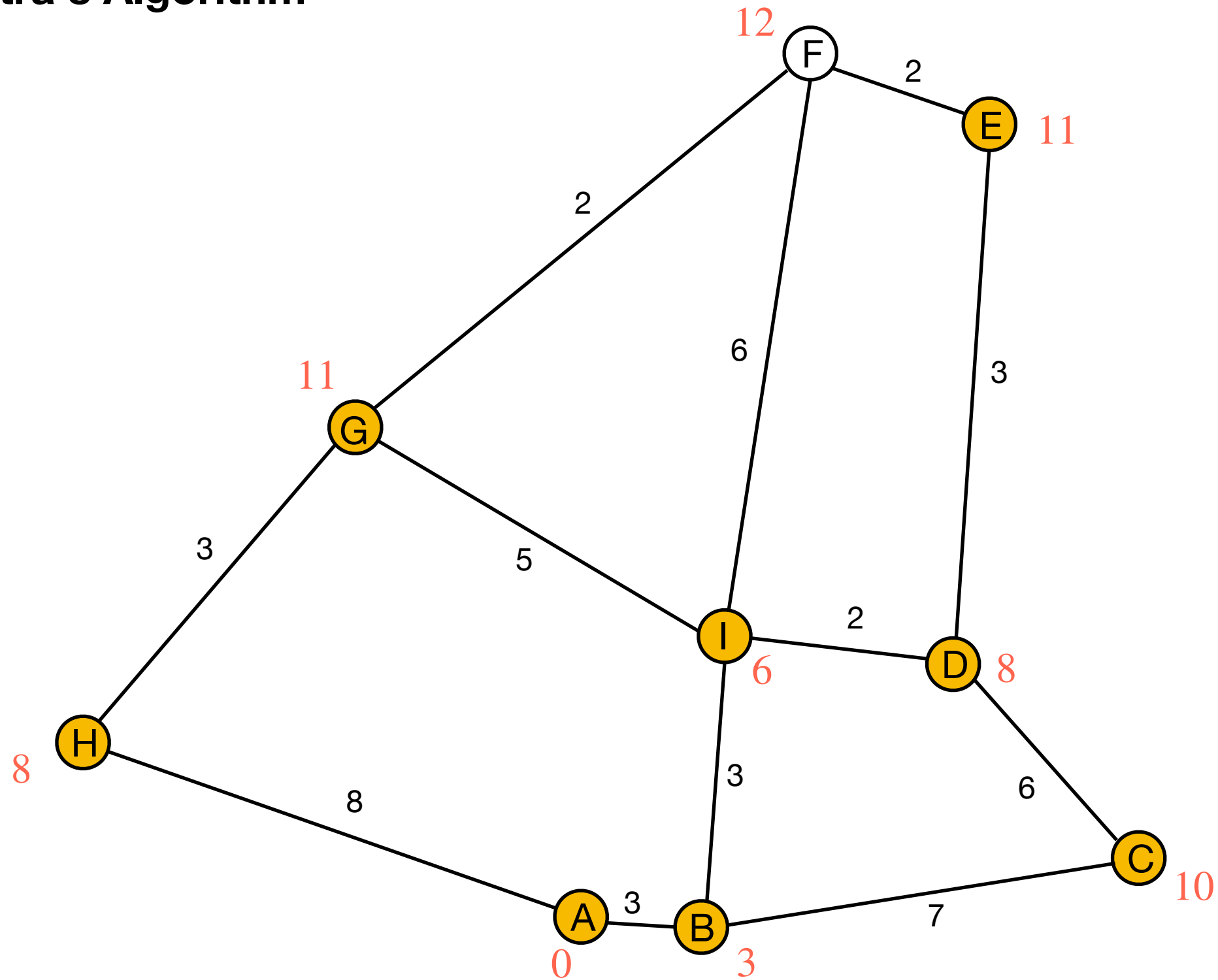
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



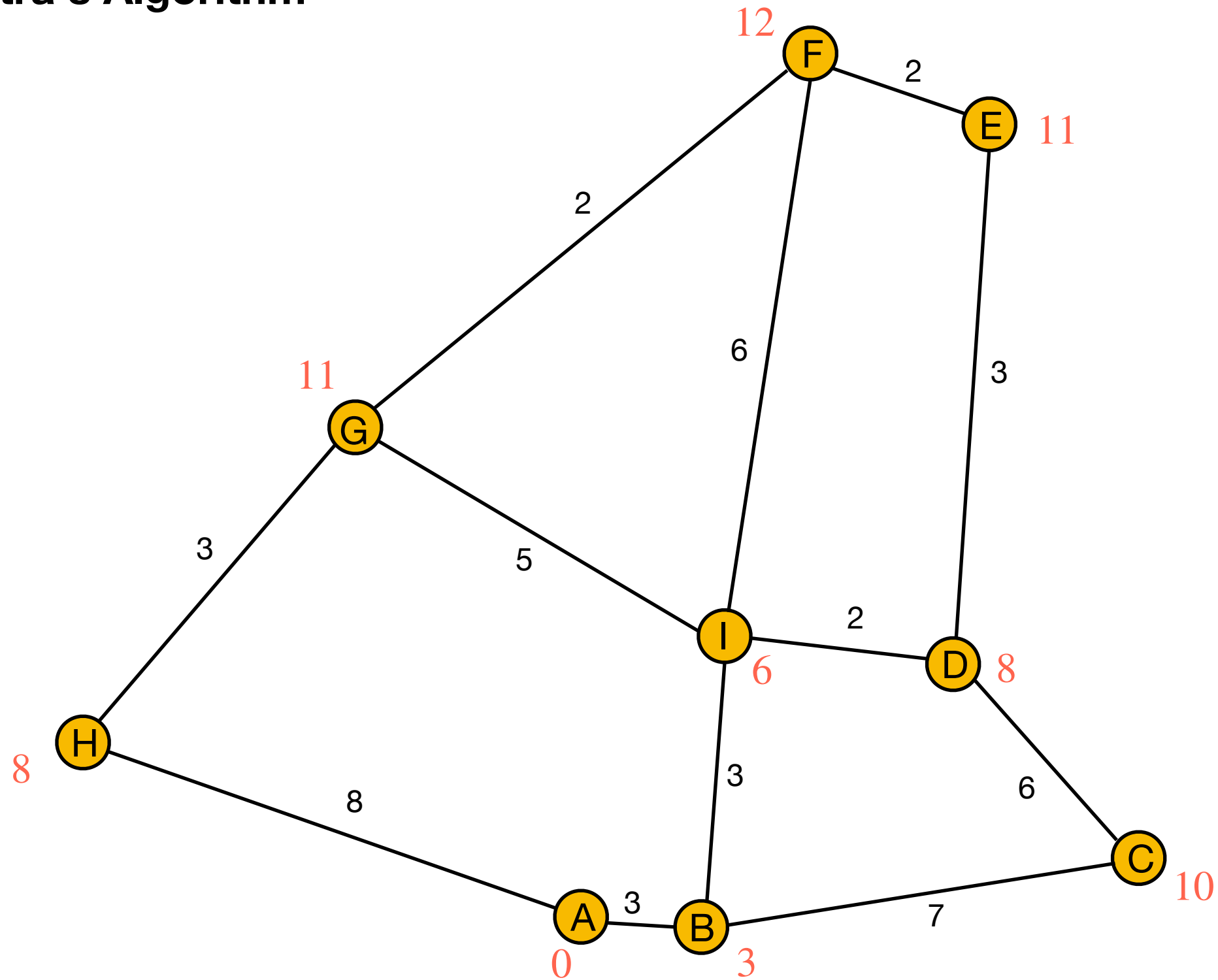
5. From that vertex, i , update the $d[j]$ values for all adjacent vertices j : $\min \left\{ d[i] + \text{weight}[i, j], d[j] \right\}$ (again)

Dijkstra's Algorithm



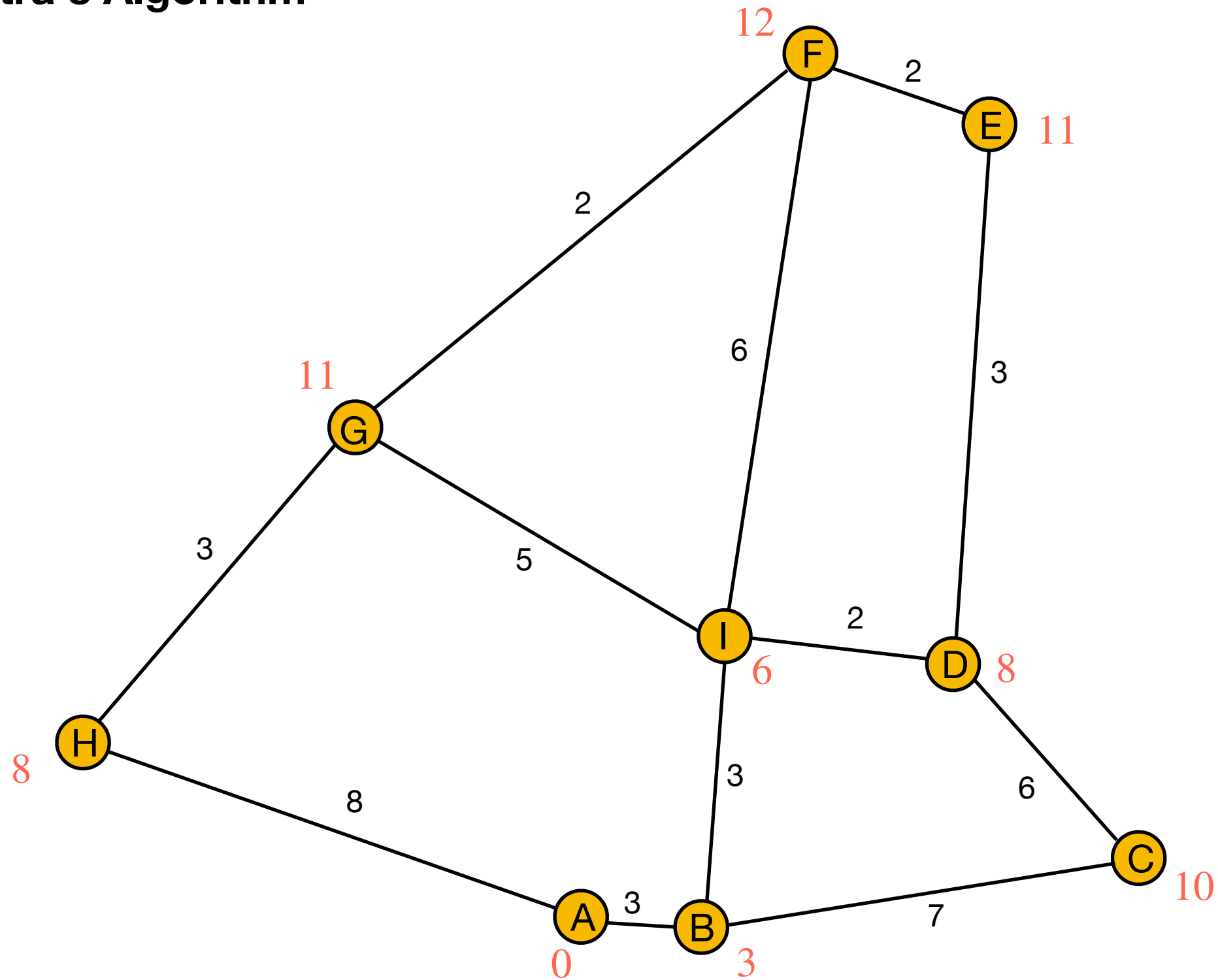
4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

Dijkstra's Algorithm



4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

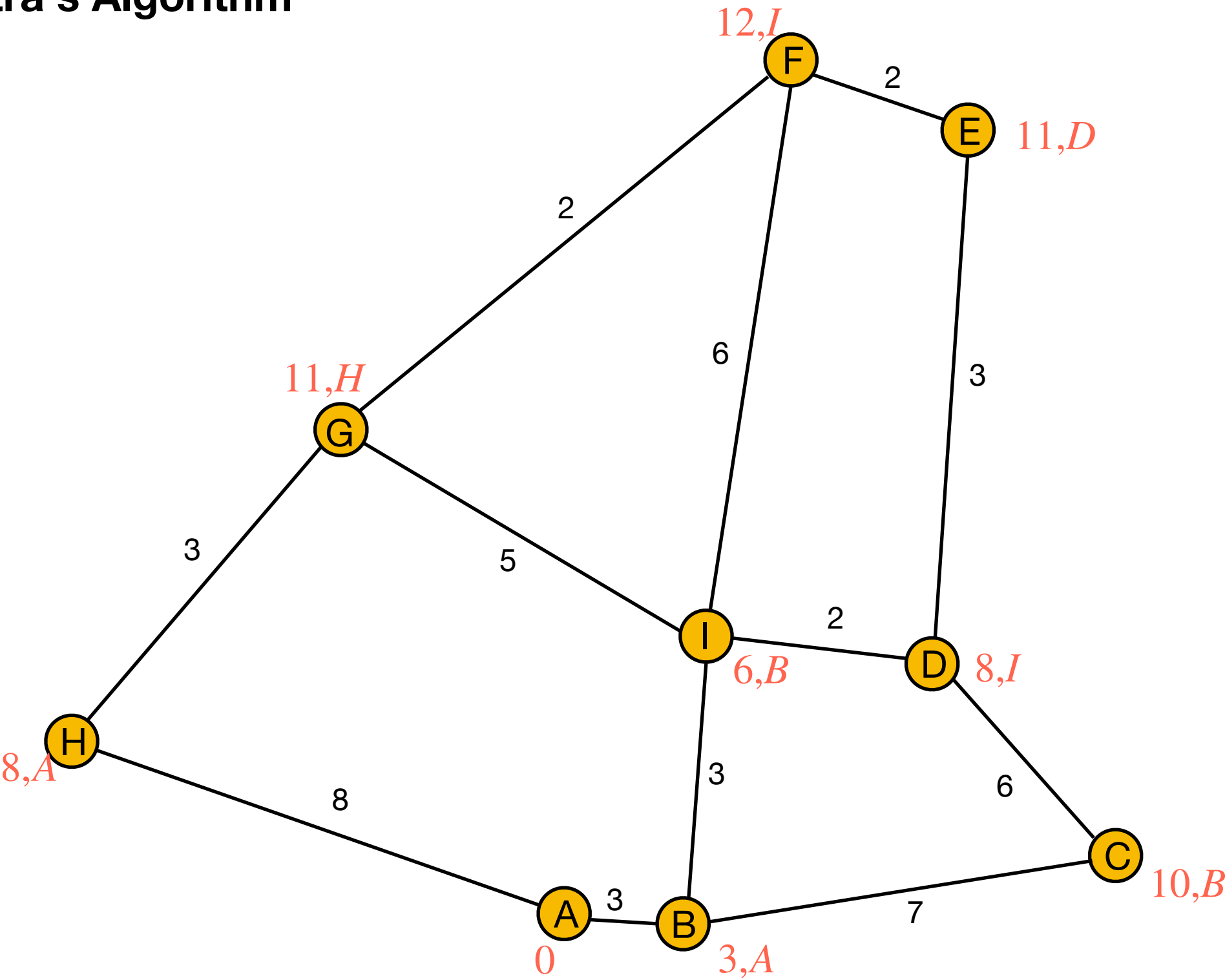
Dijkstra's Algorithm



6. All vertices are visited, so terminate.

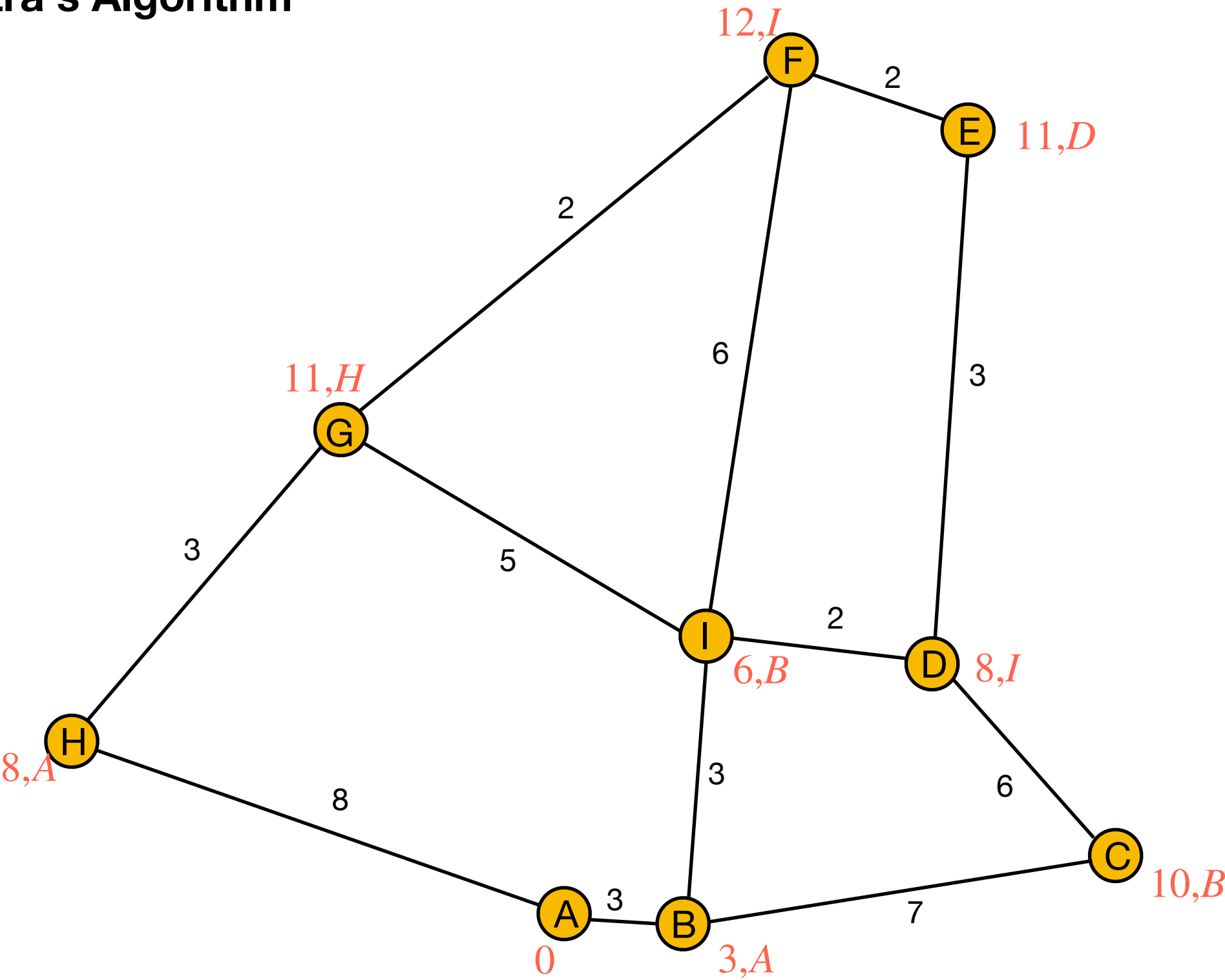
We now know the lowest cost to get from the vertex A to each other vertex in the graph!

Dijkstra's Algorithm



Keep track of the vertex we take every time we update $d[i]$ value

Dijkstra's Algorithm



Keep track of the vertex we take every time we update $d[i]$ value
Follow the vertices backwards to the source: $F \rightarrow I \rightarrow B \rightarrow A$

Dijkstra's Algorithm

The map illustrates the shortest path from node A to node H using Dijkstra's Algorithm. The nodes and their associated shortest path (distance, path) are:

- A: 0
- B: 3, A
- C: 10, C
- D: 8, I
- E: 11, D
- F: 12, I
- G: 11, H
- H: 8, A
- I: 6, B
- J: 11, H

The path from A to H is highlighted in red, passing through nodes B, C, D, E, F, G, and H. The map also shows various landmarks, roads, and the Yamuna River.

Dijkstra's Algorithm

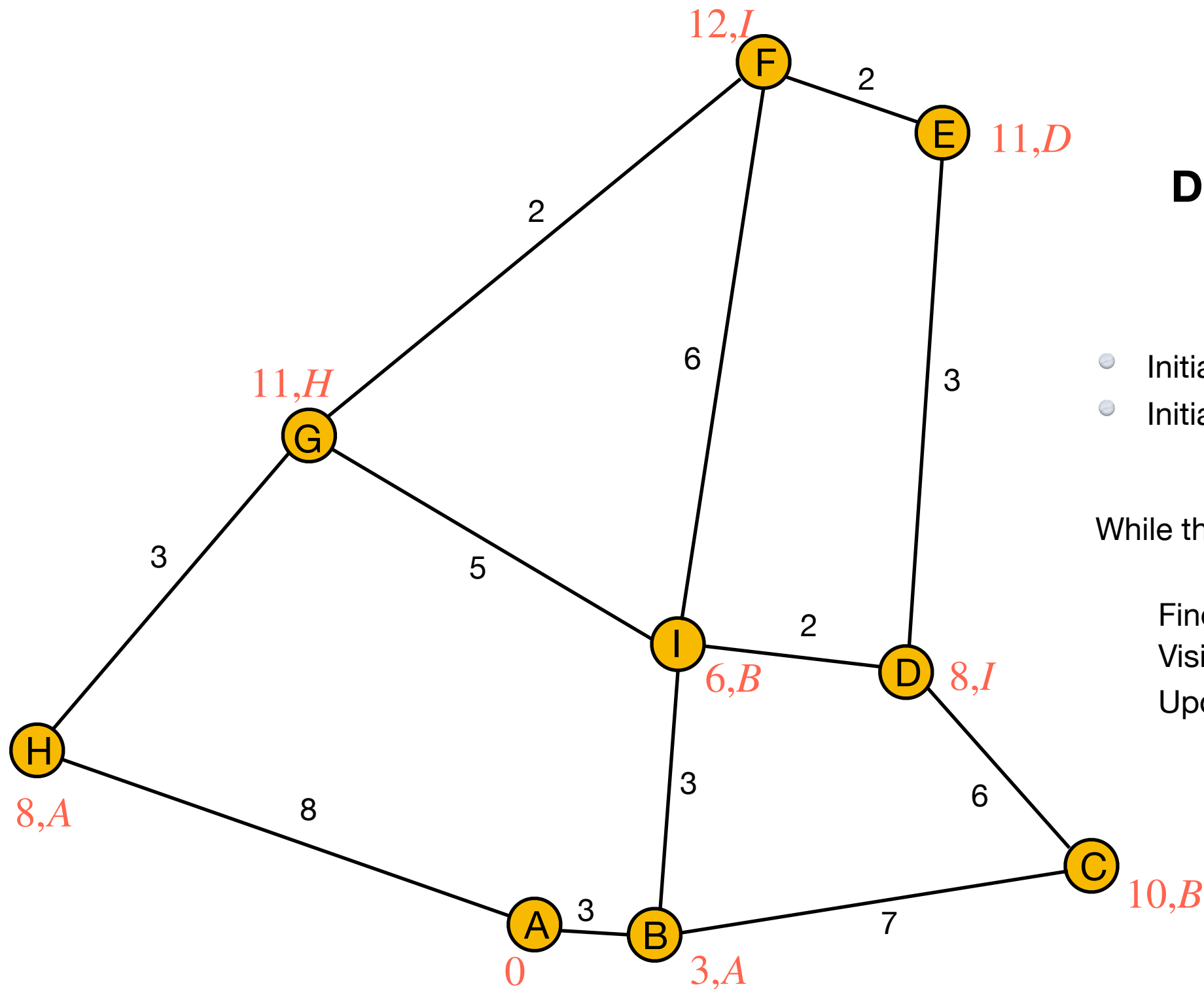
- Initialise $d[i]$ to ∞
- Initialise $d[\text{start}]$ to 0

While there are unvisited vertices:

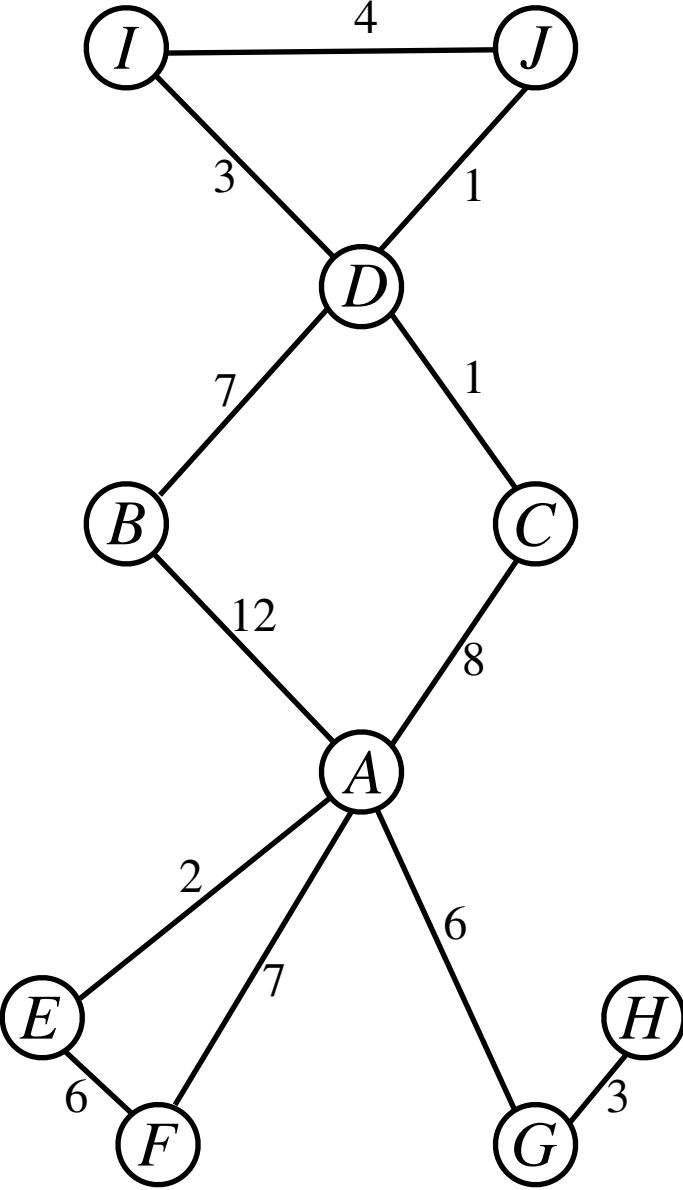
Find the minimum $d[i]$ vertex

Visit the vertex

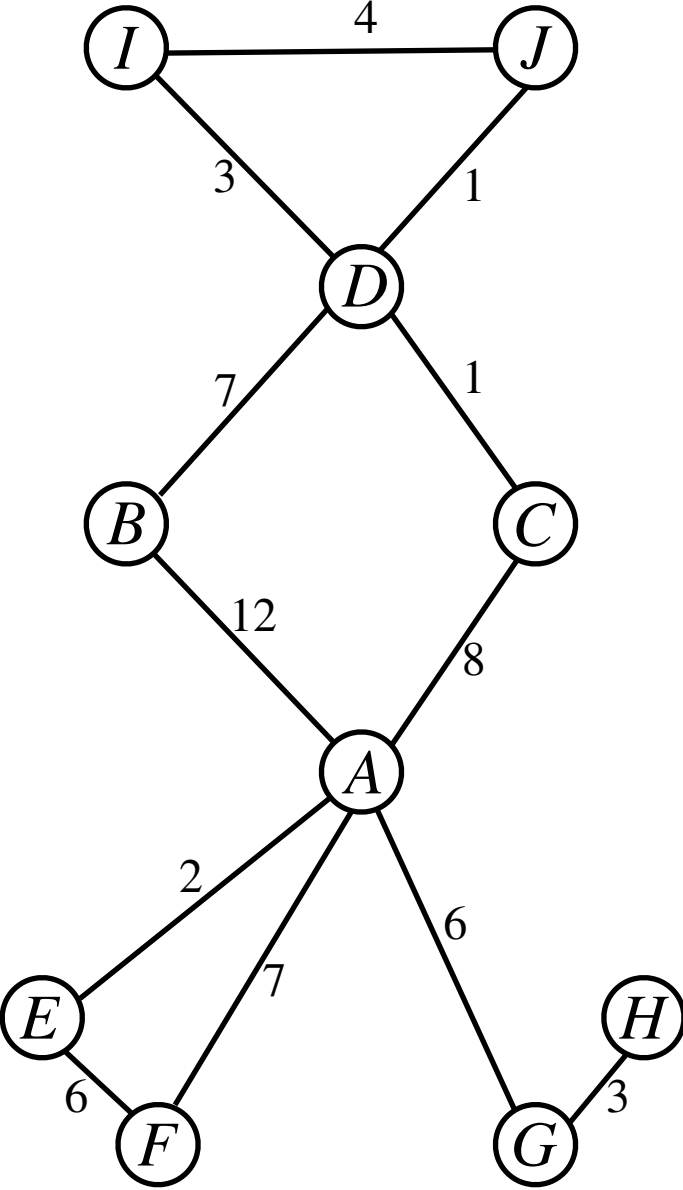
Update $d[i]$ for its unvisited neighbours



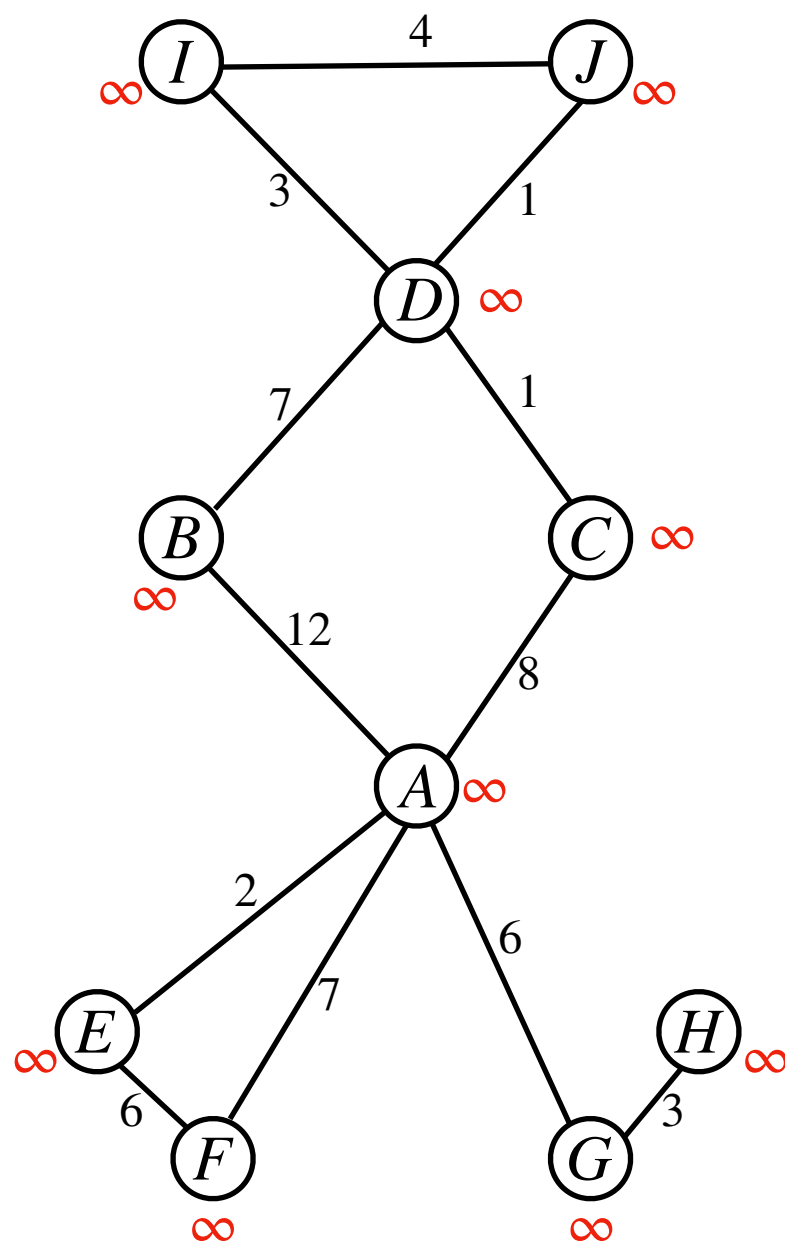
Shortest Path Problem



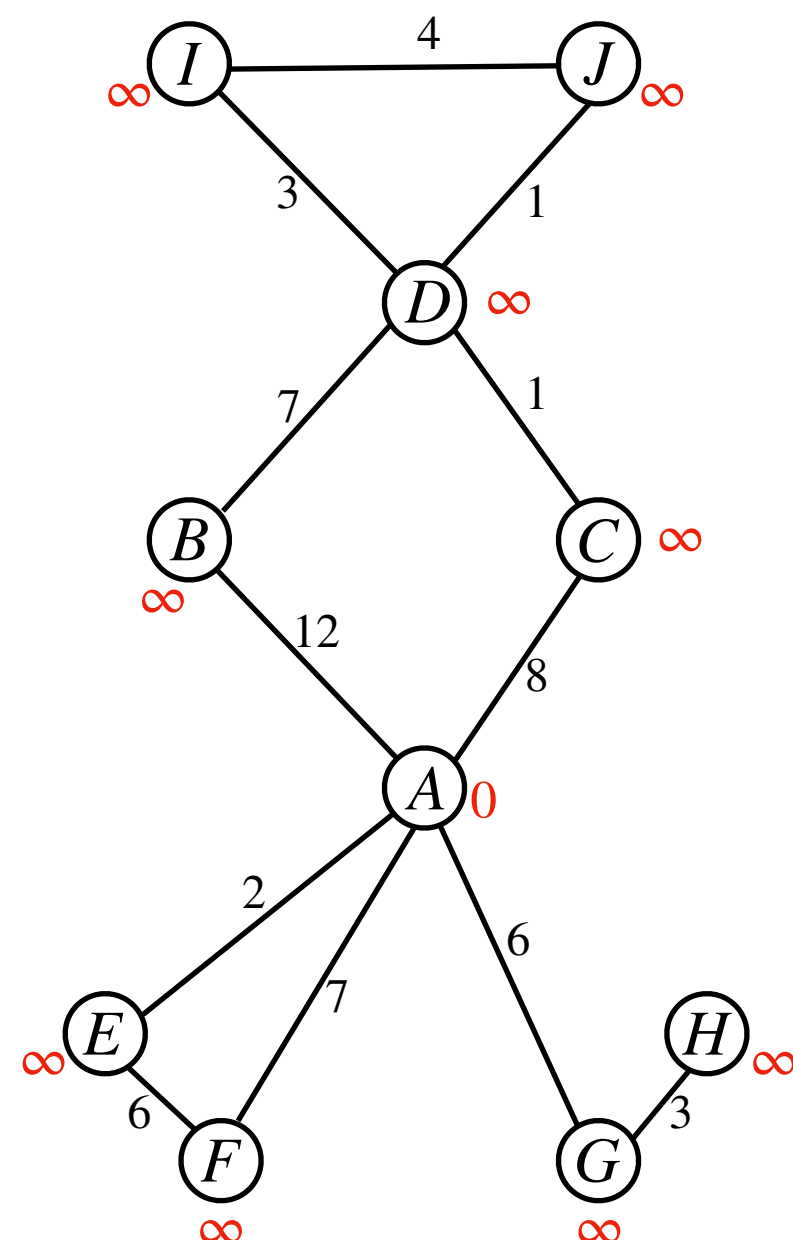
Shortest Path Problem



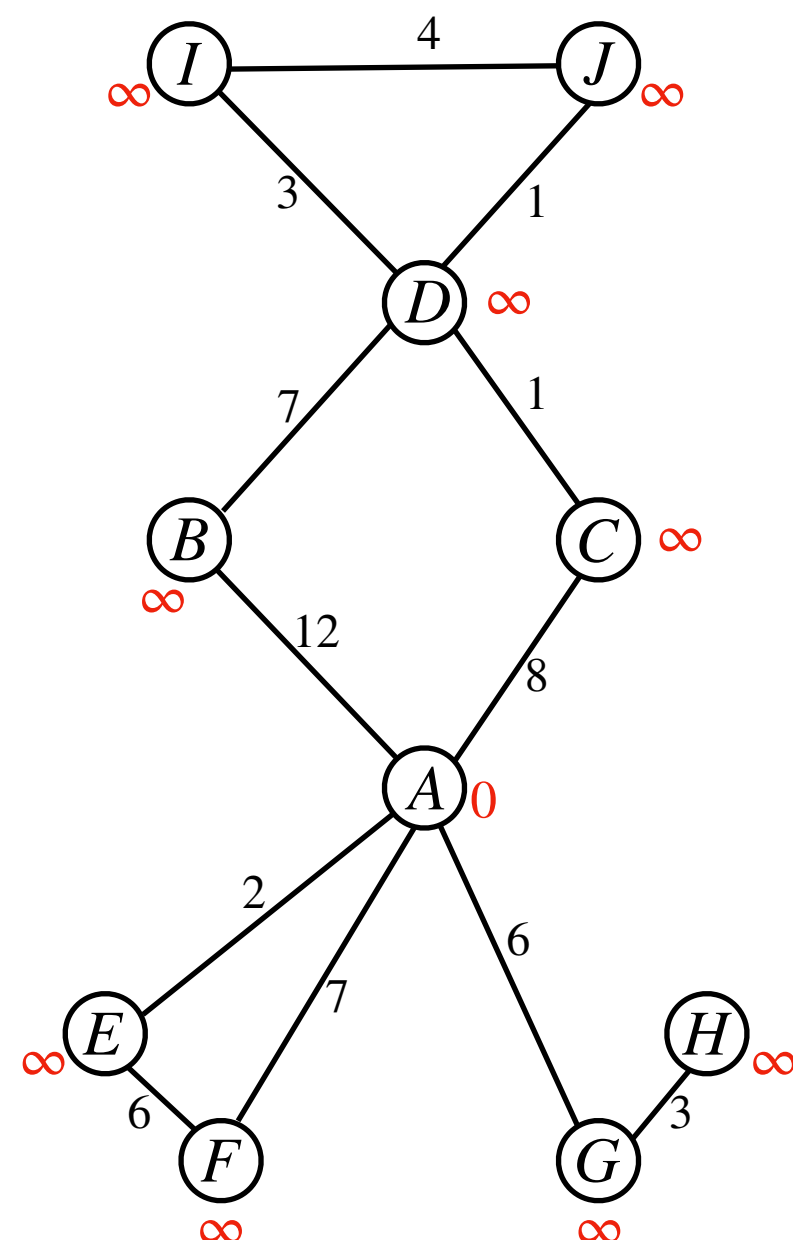
Shortest Path Problem



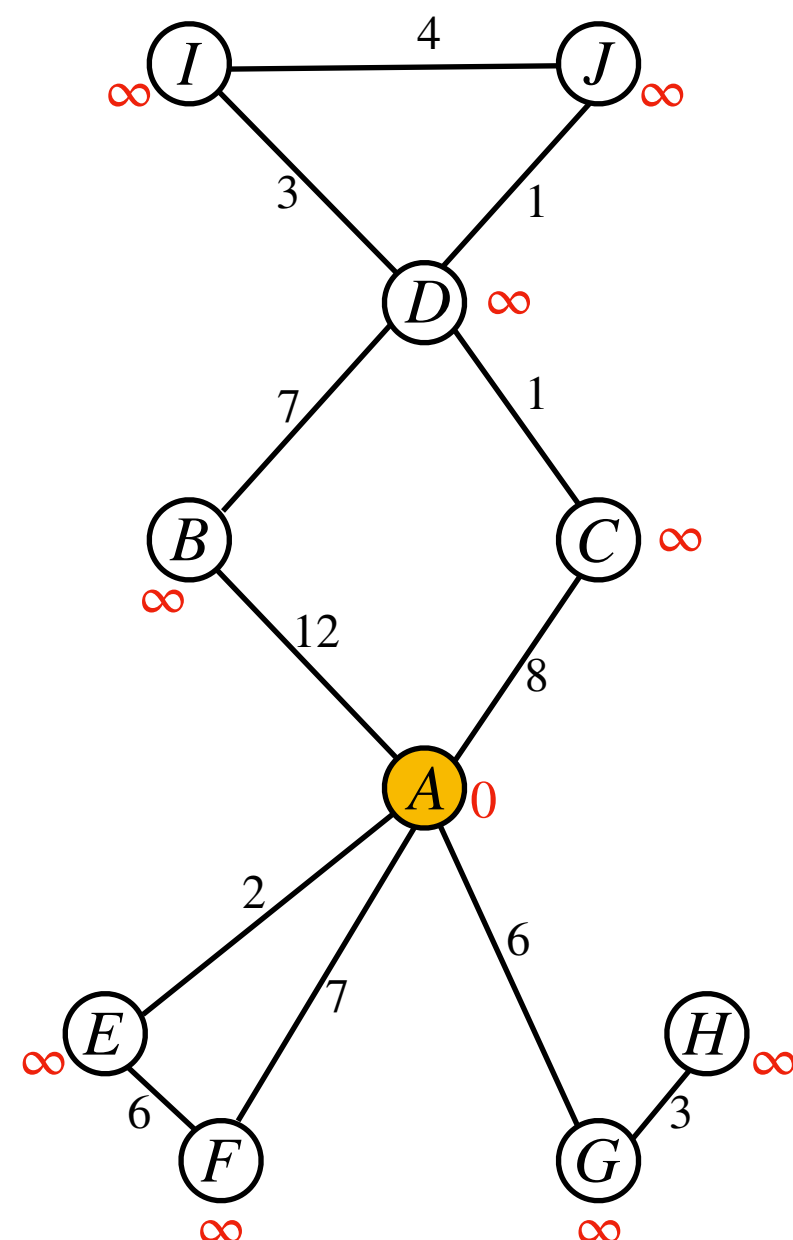
Shortest Path Problem



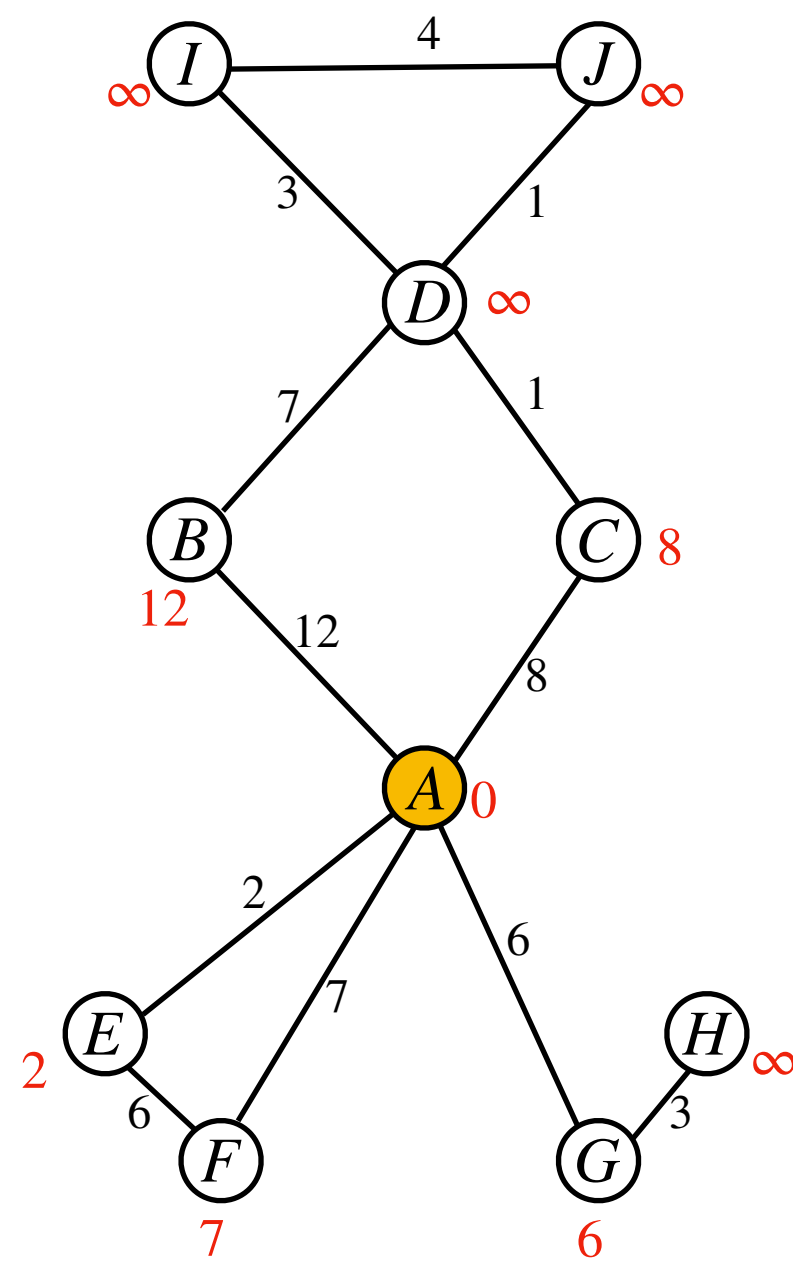
Shortest Path Problem



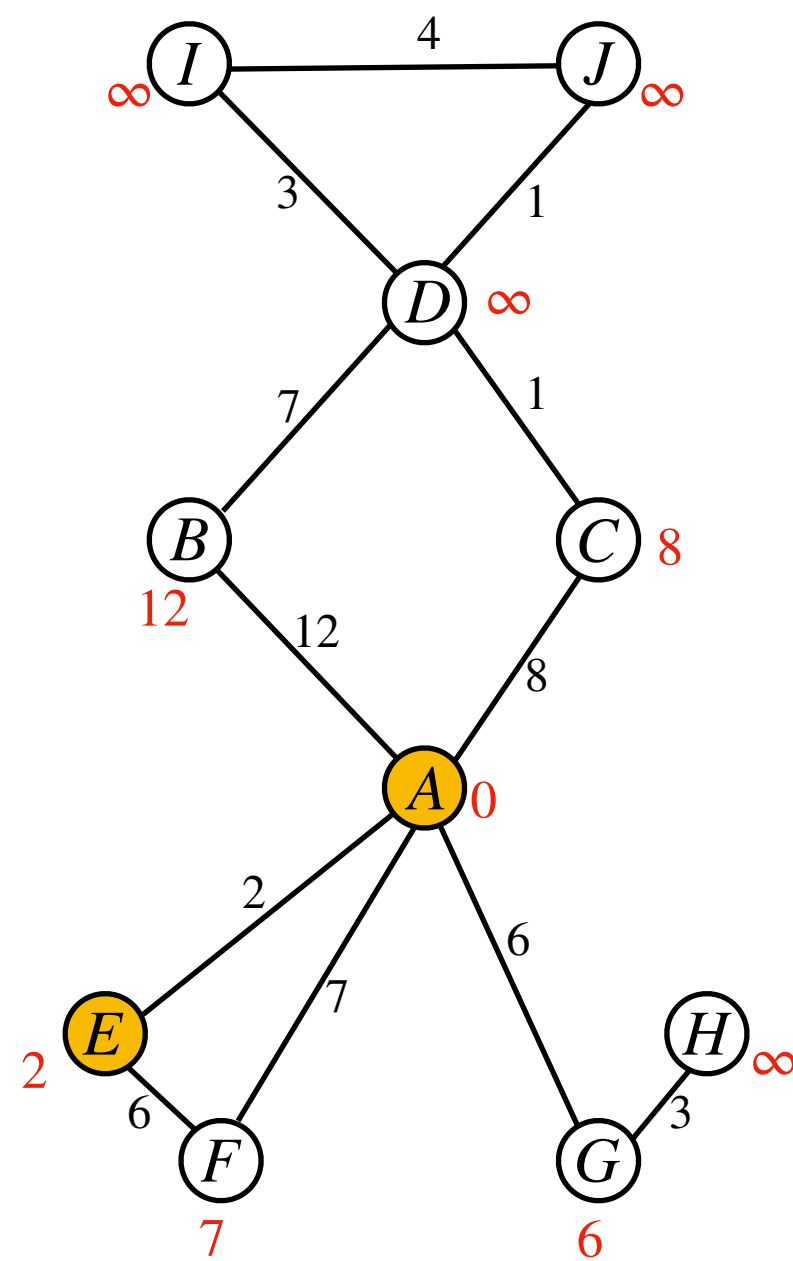
Shortest Path Problem



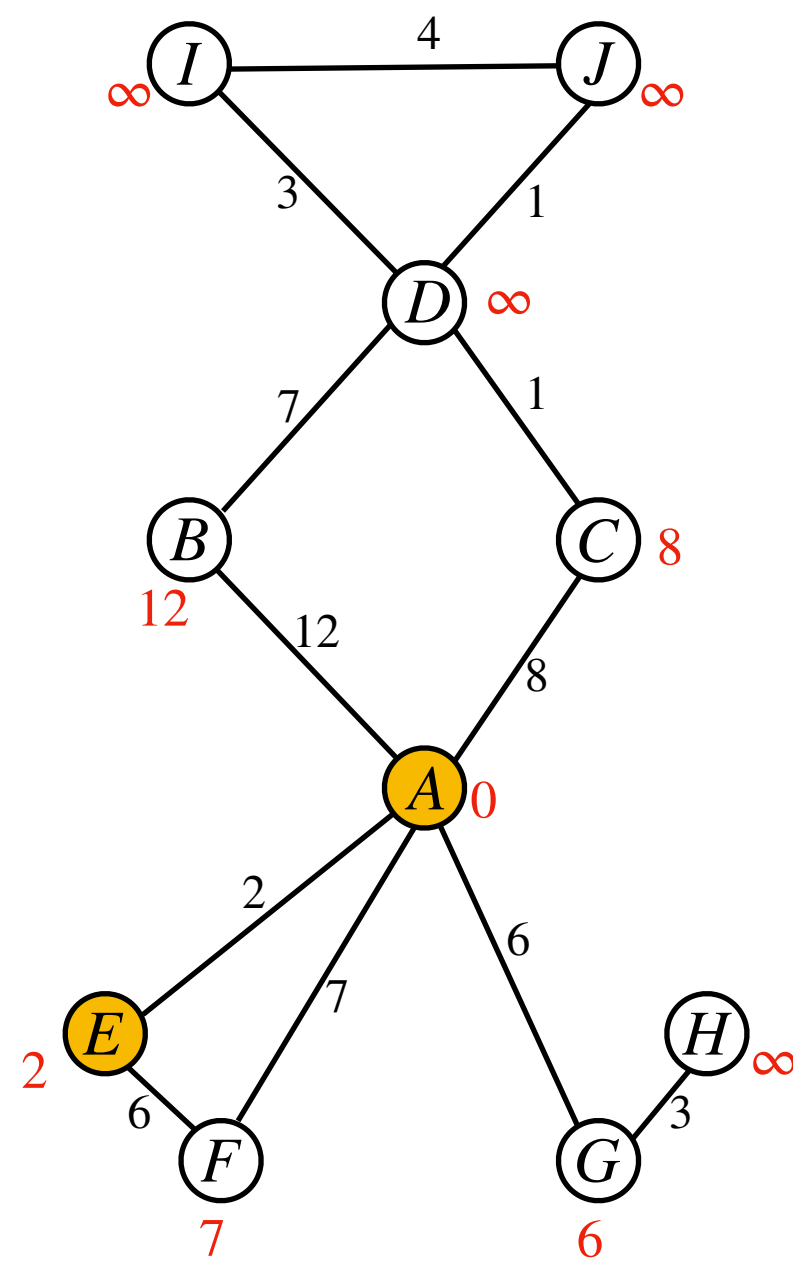
Shortest Path Problem



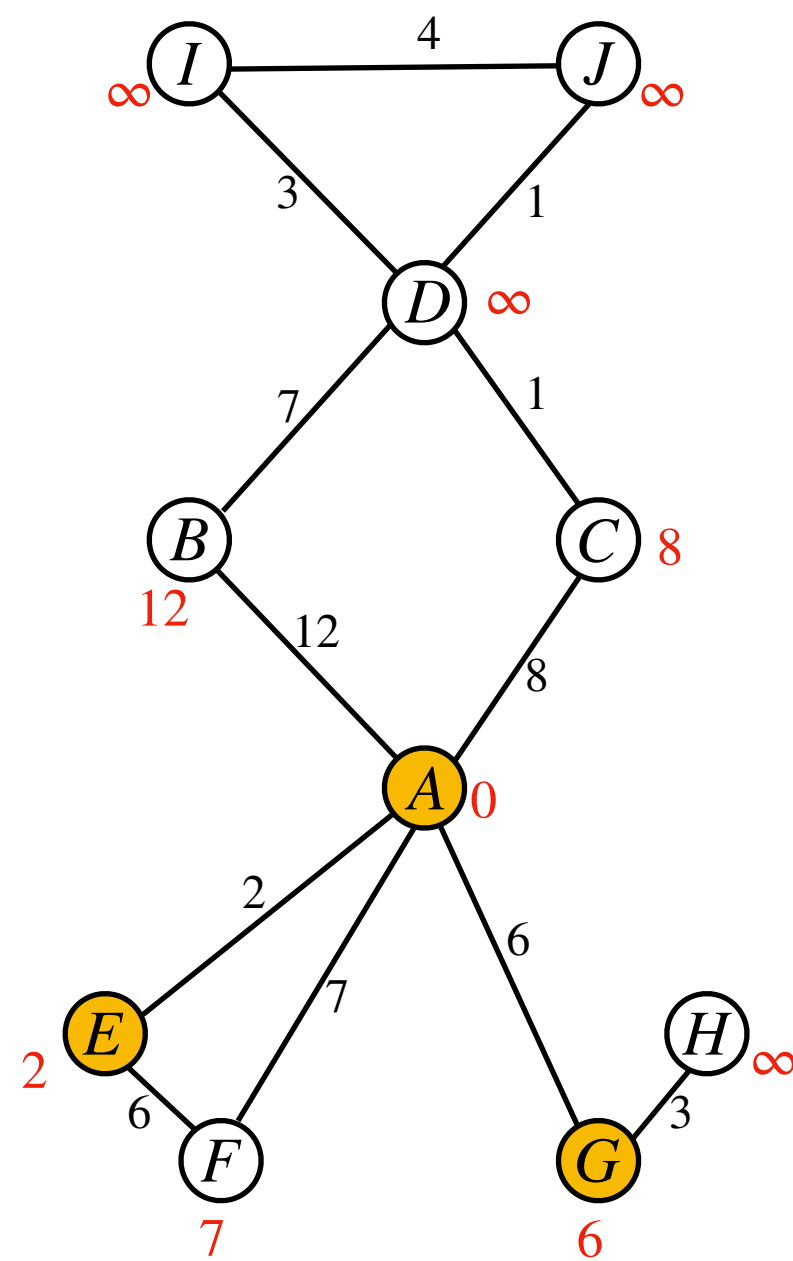
Shortest Path Problem



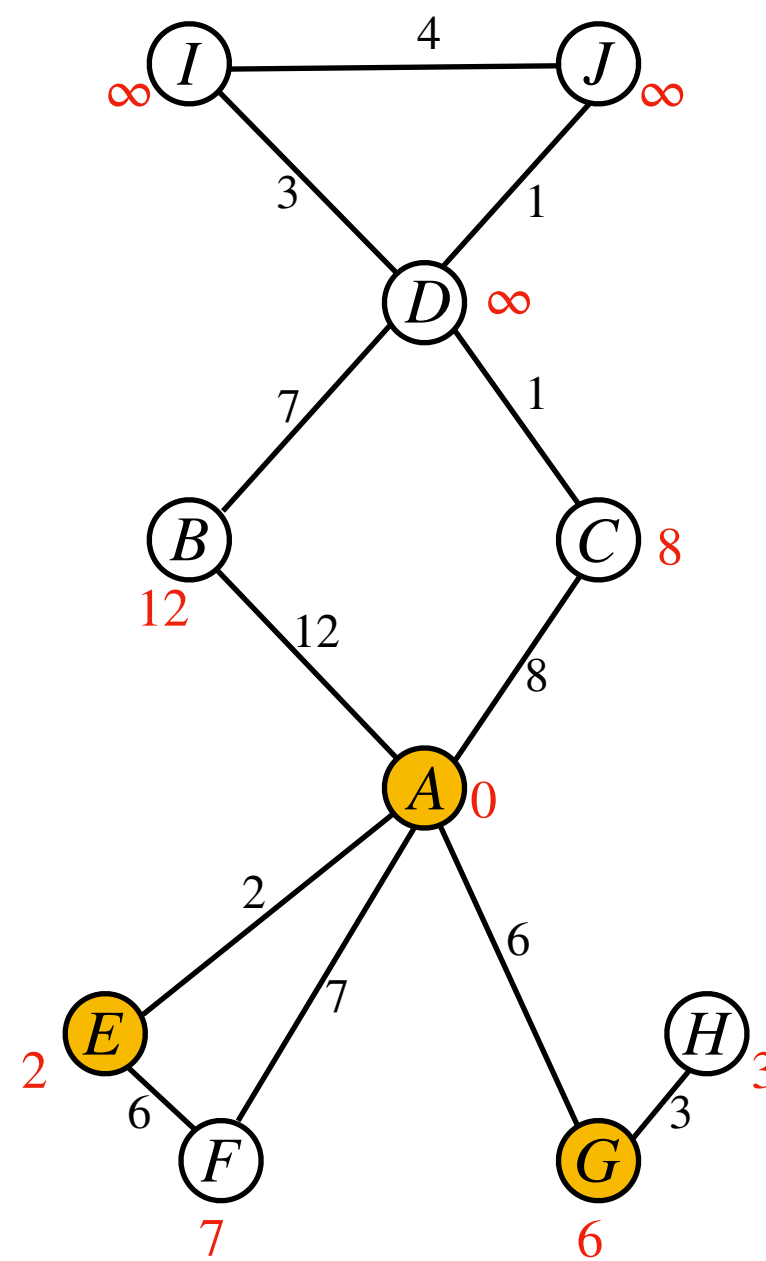
Shortest Path Problem



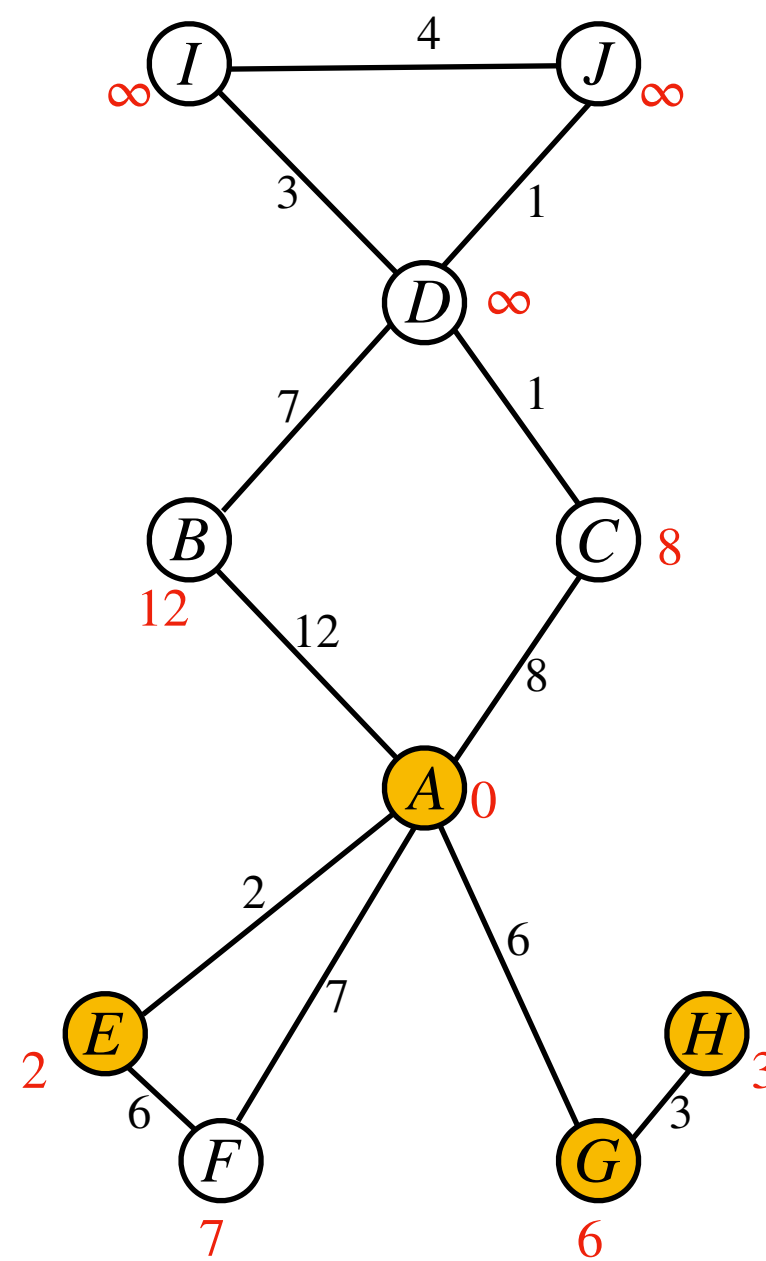
Shortest Path Problem



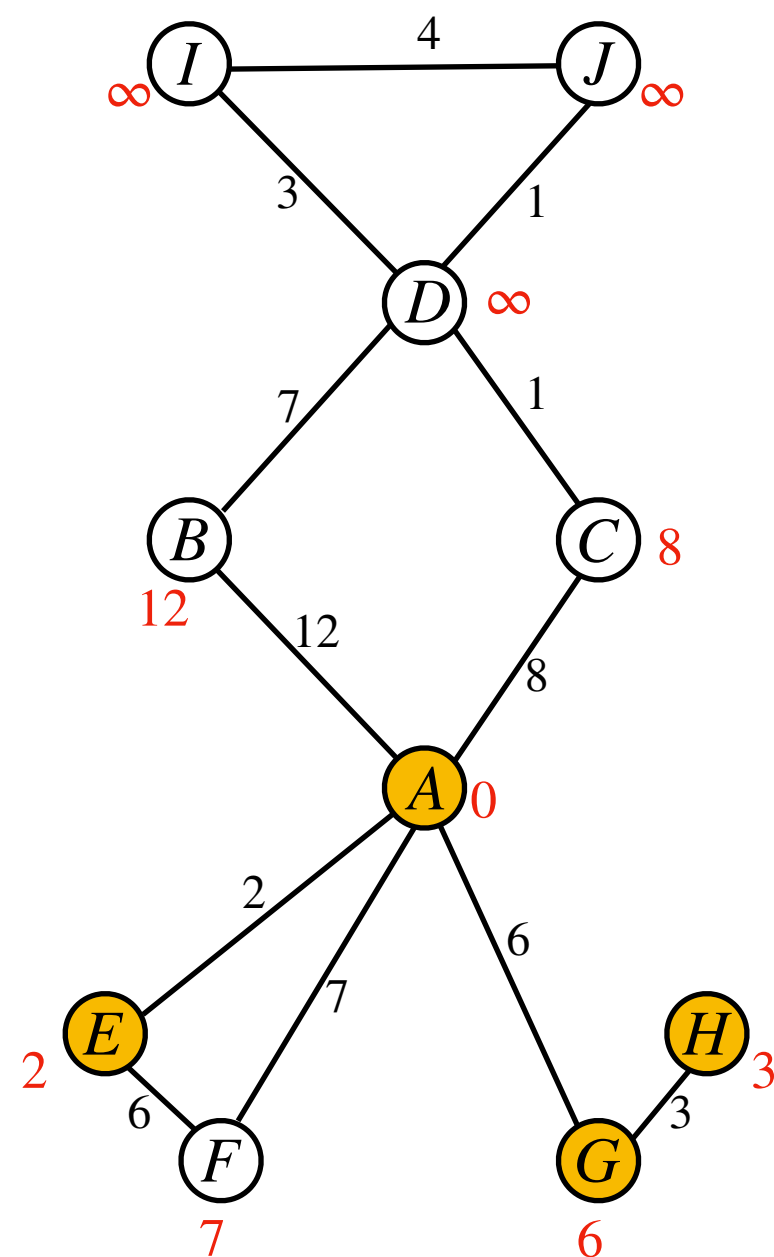
Shortest Path Problem



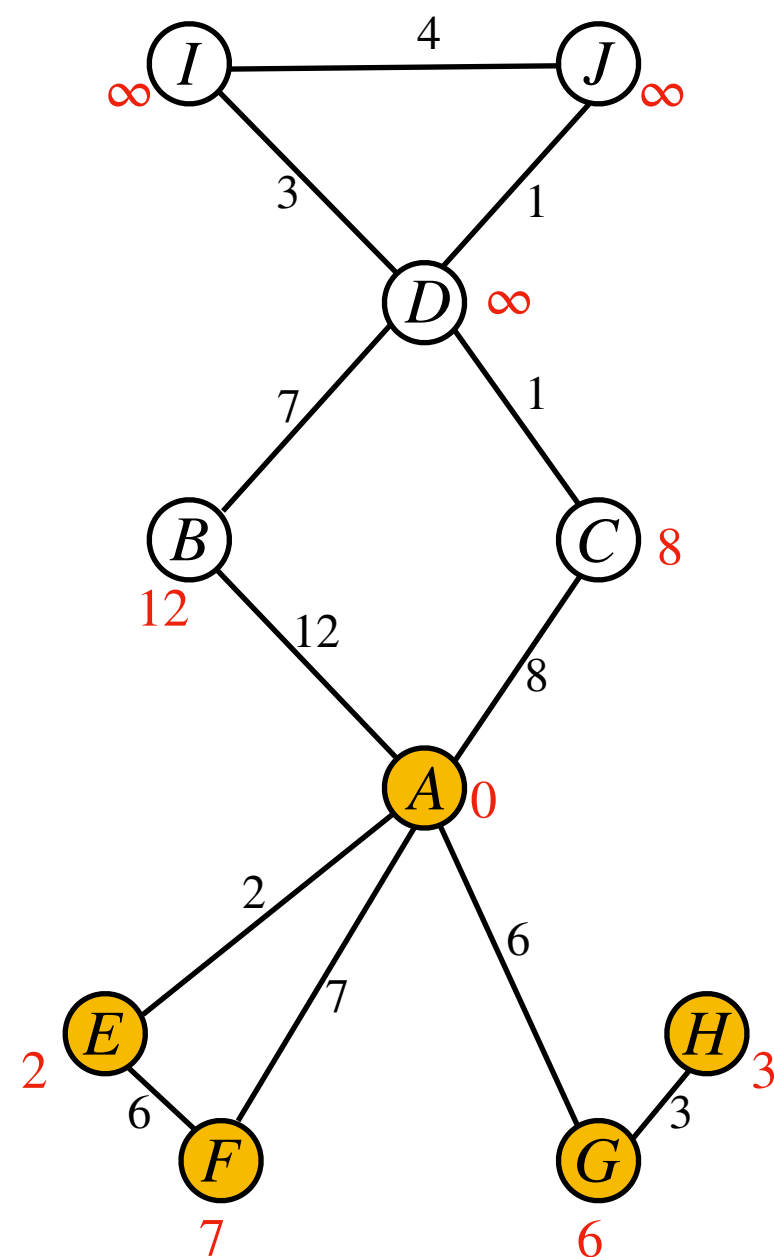
Shortest Path Problem



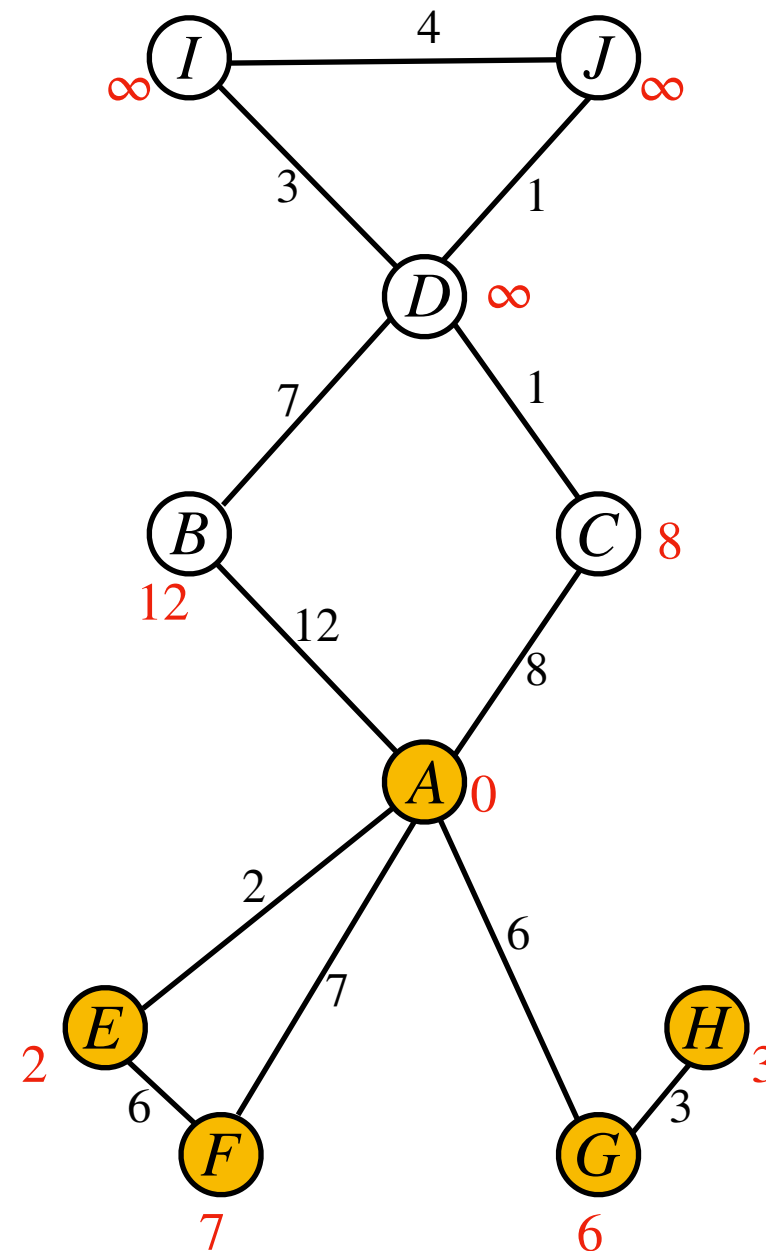
Shortest Path Problem



Shortest Path Problem

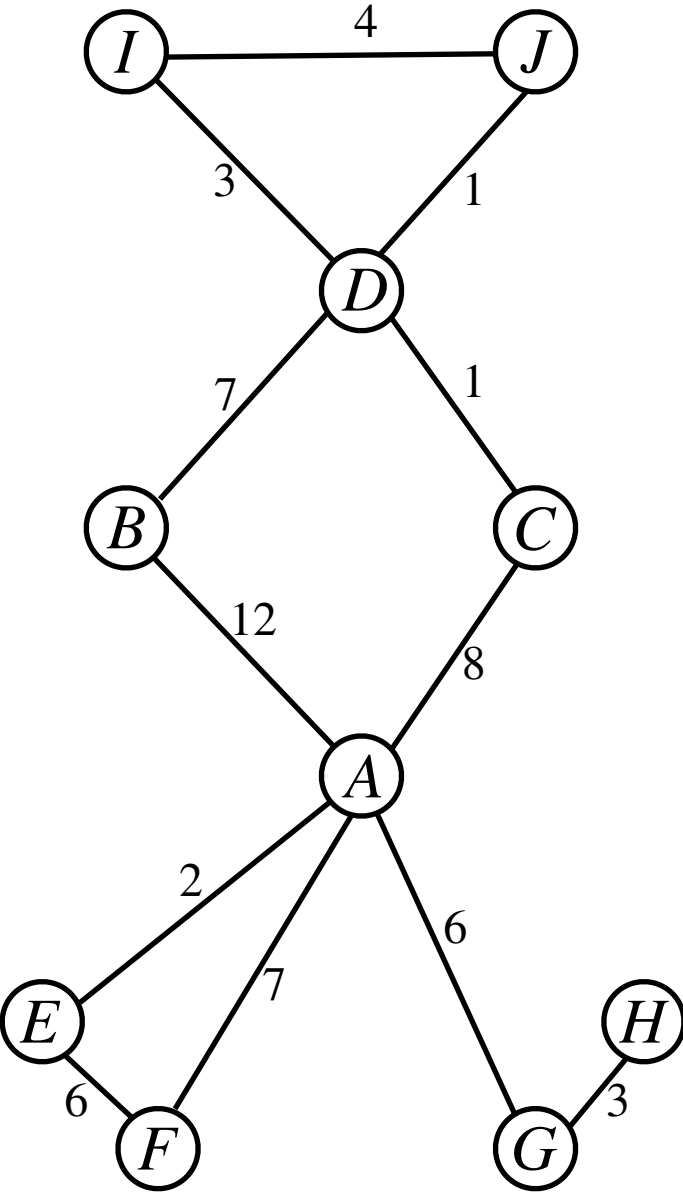


Shortest Path Problem

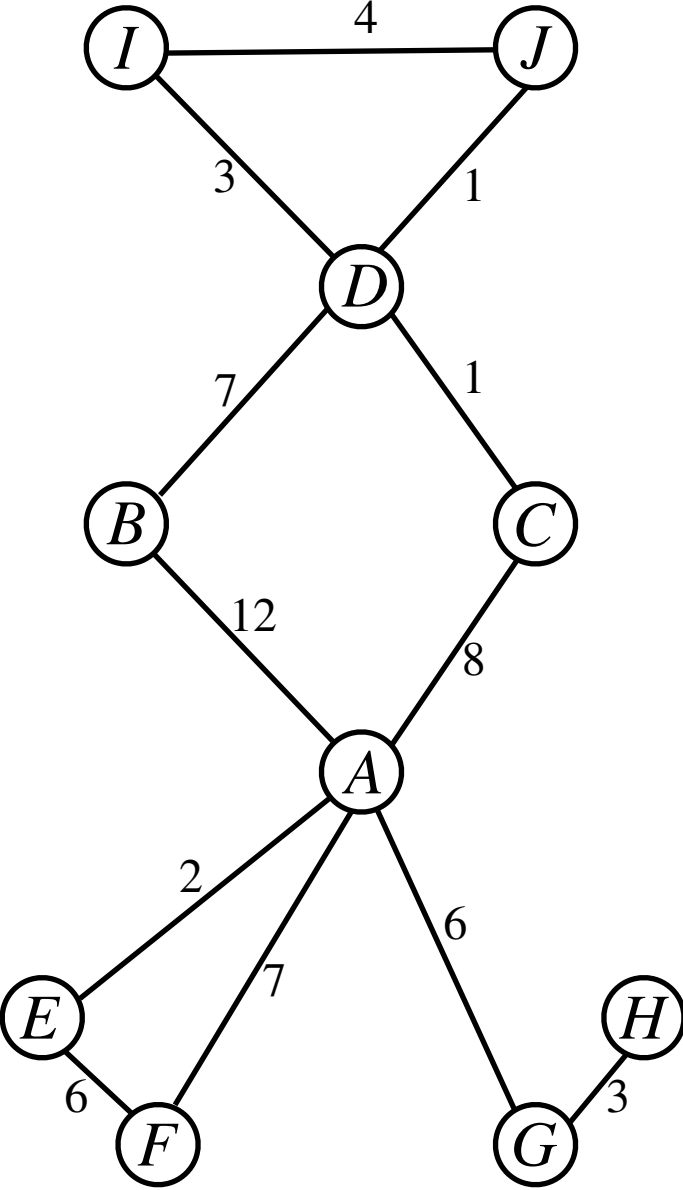


Why visit vertices that take you farther from the goal?

A* Algorithm

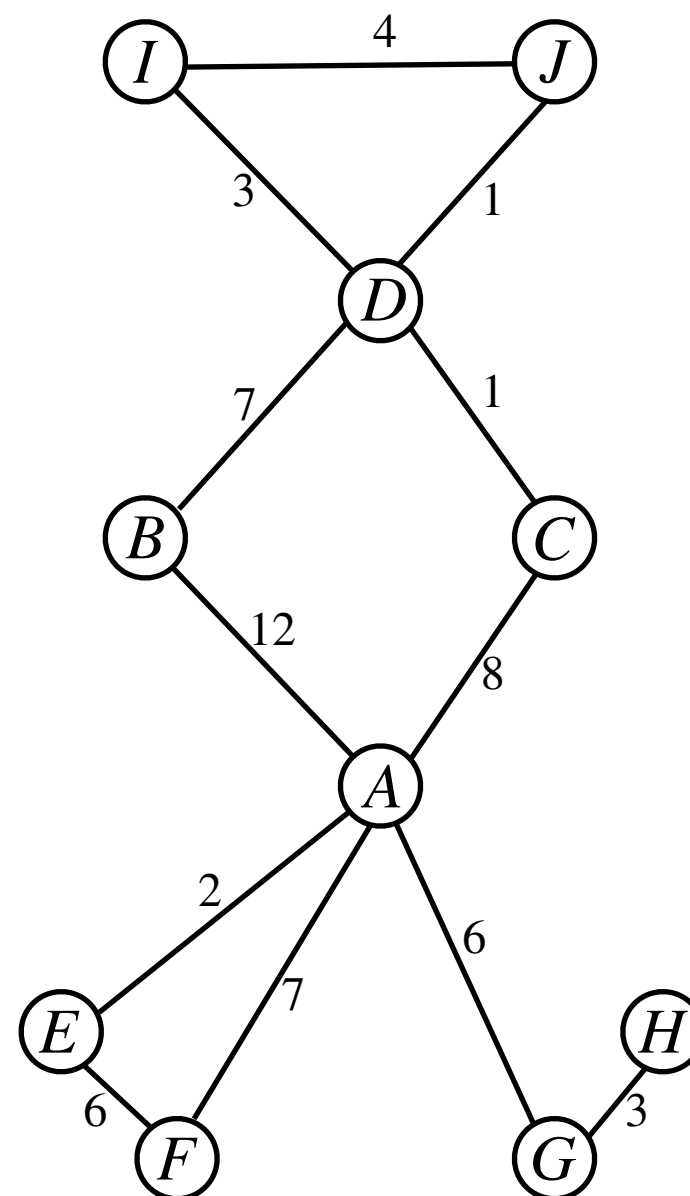


A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

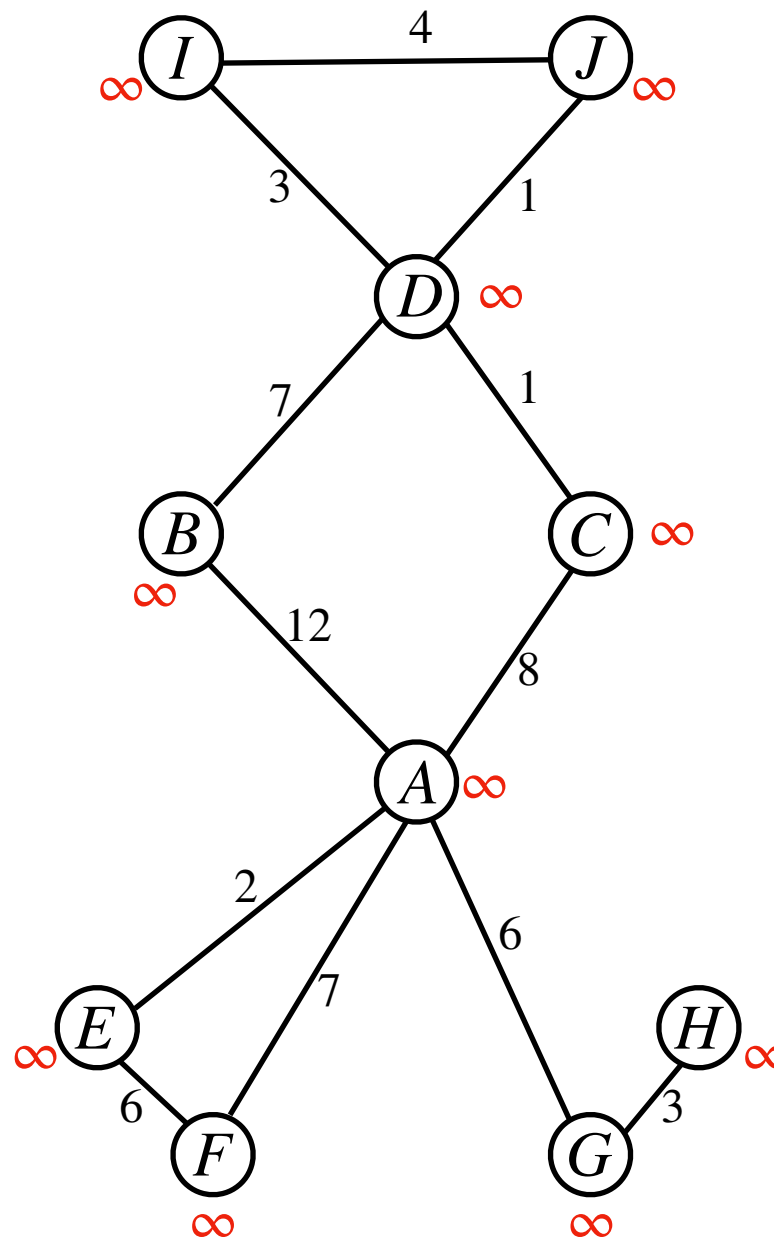
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

1. Initialise a value at each vertex to ∞ . Call these values $d[i]$

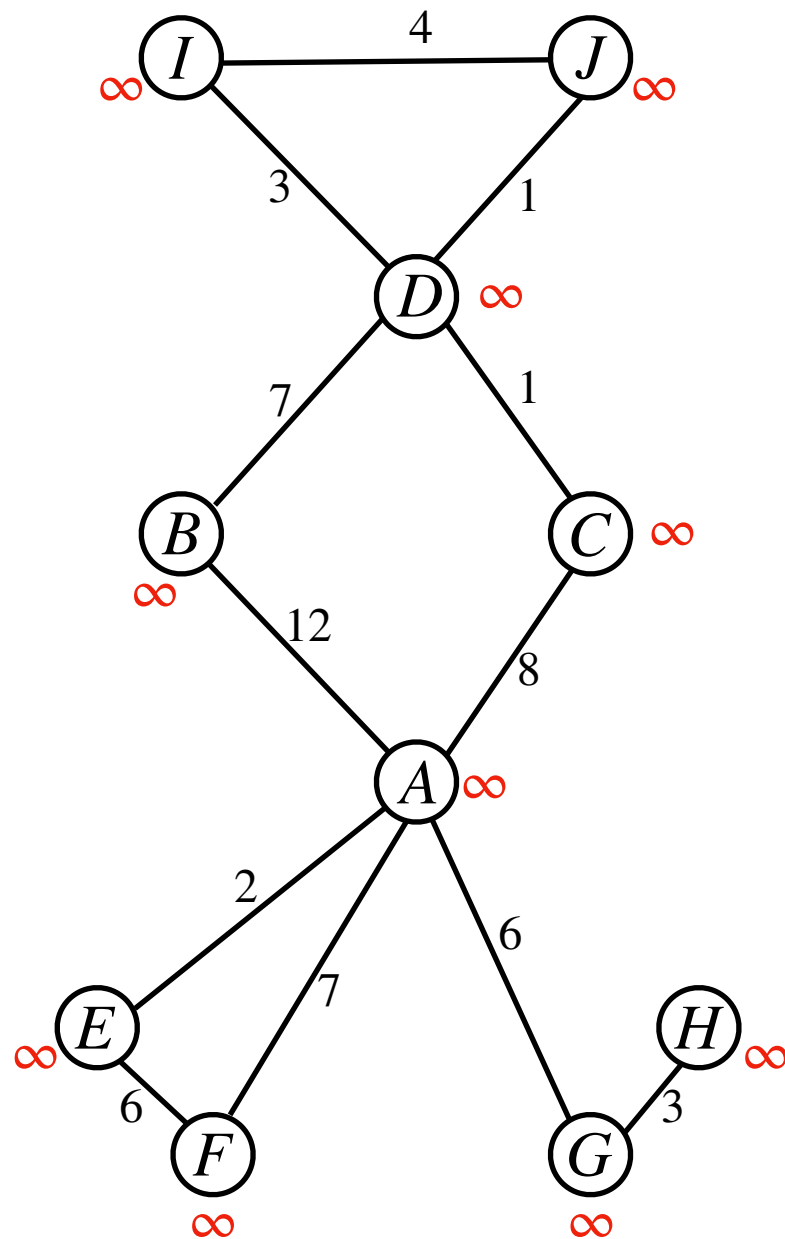
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

1. Initialise a value at each vertex to ∞ . Call these values $d[i]$

A* Algorithm

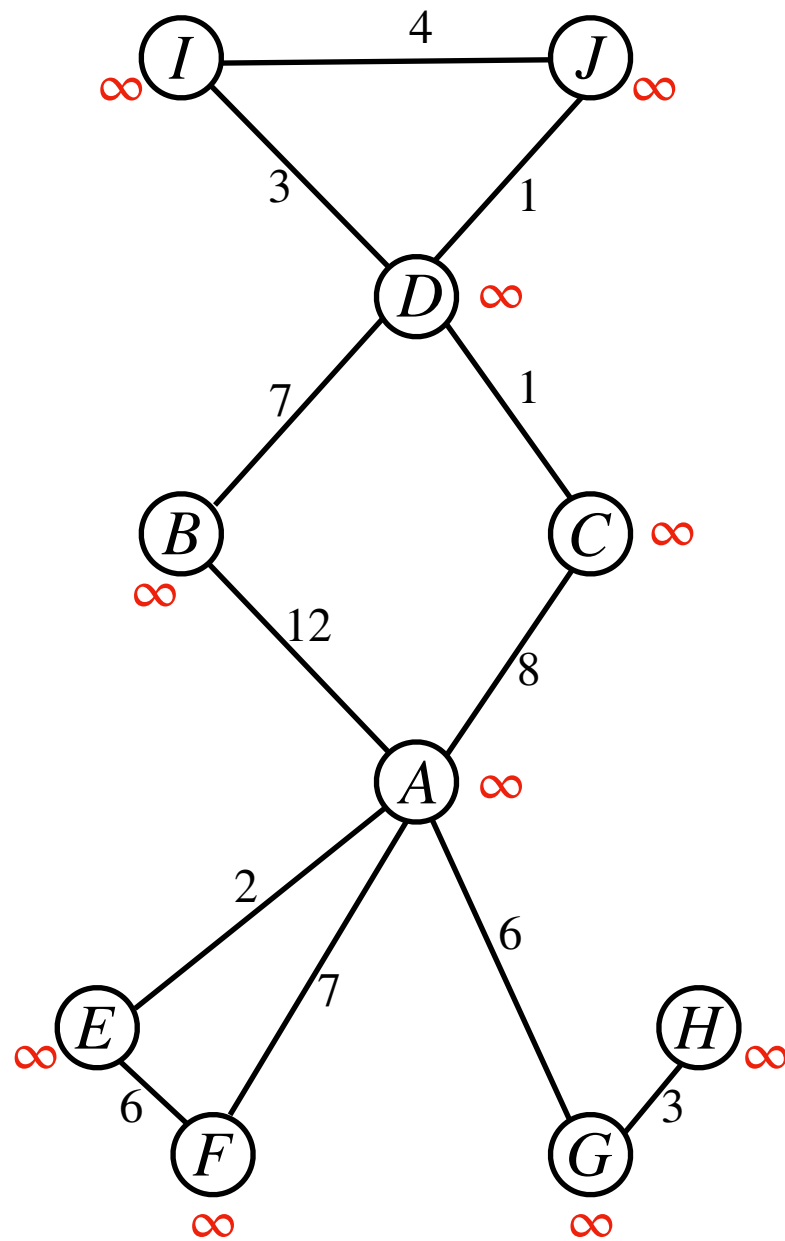


Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

1. Initialise a value at each vertex to ∞ . Call these values $d[i]$

These ∞ values represent the cost of reaching each vertex from our source, using only intermediary vertices whose shortest paths we have already found.

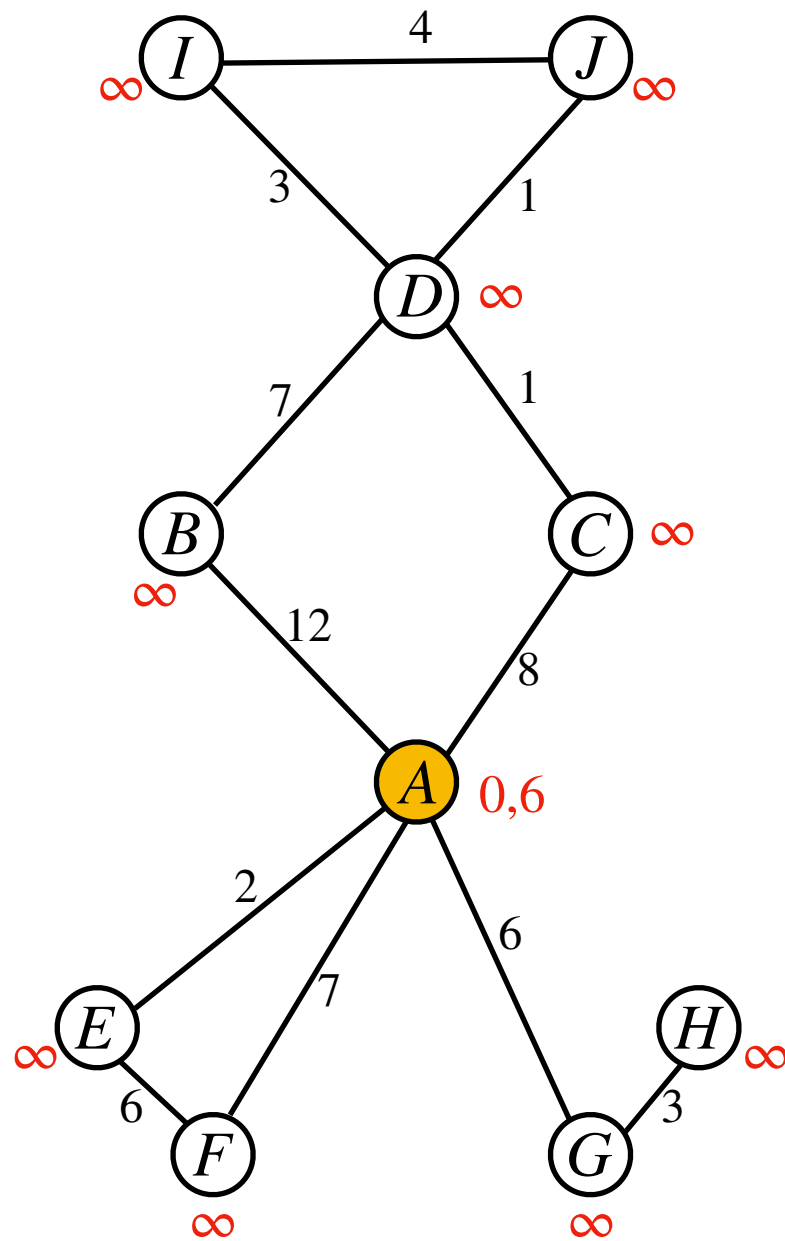
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

2. Initialise the value at our source vertex to $g[\text{start}] = 0$ and $d[\text{start}] = g[\text{start}] + h[\text{start}]$

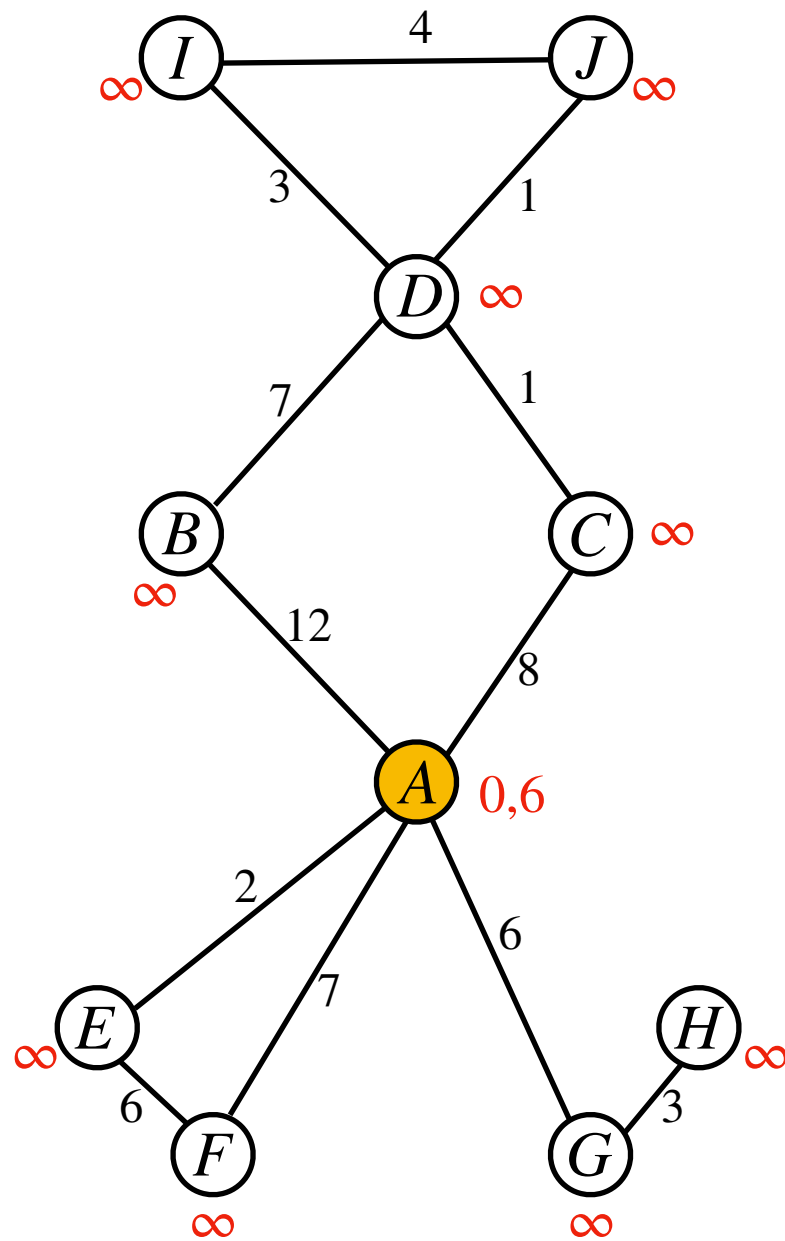
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

2. Initialise the value at our source vertex to $g[\text{start}] = 0$ and $d[\text{start}] = g[\text{start}] + h[\text{start}]$

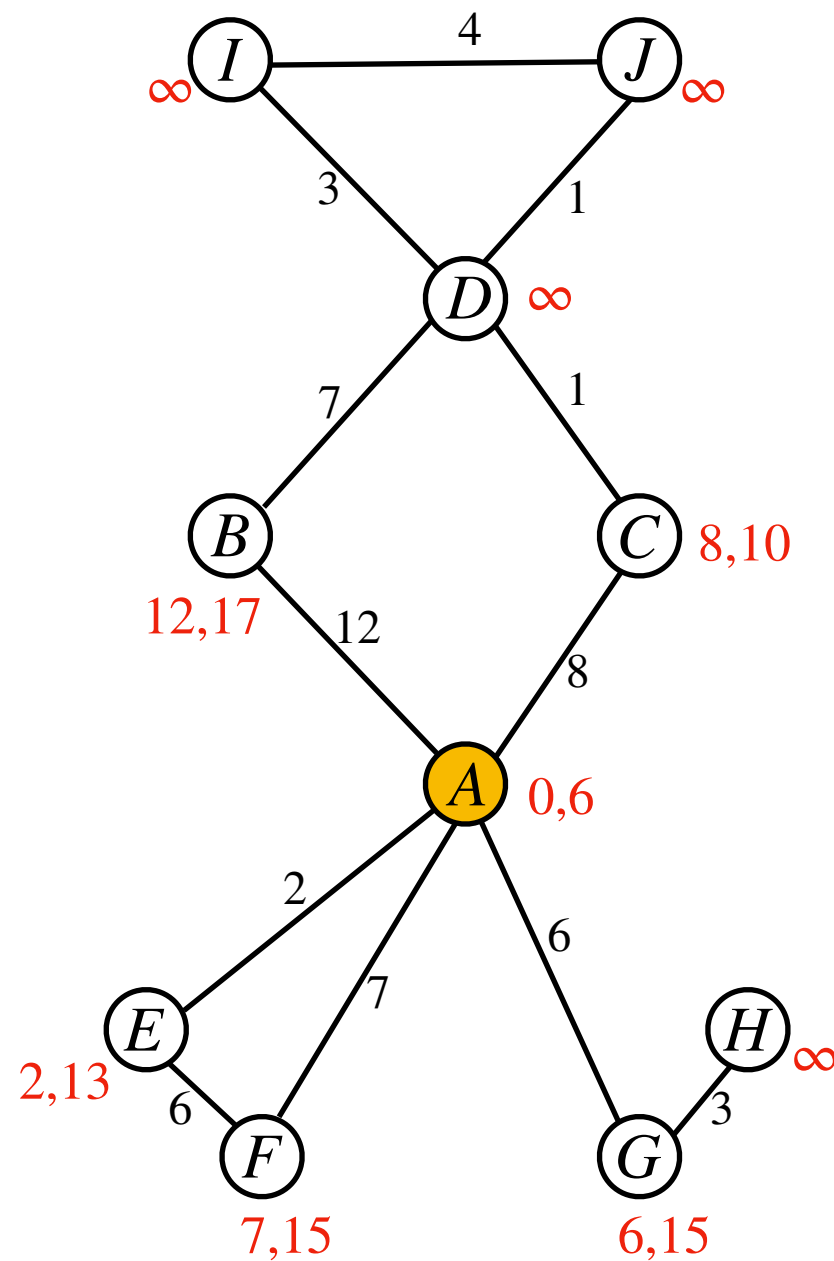
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

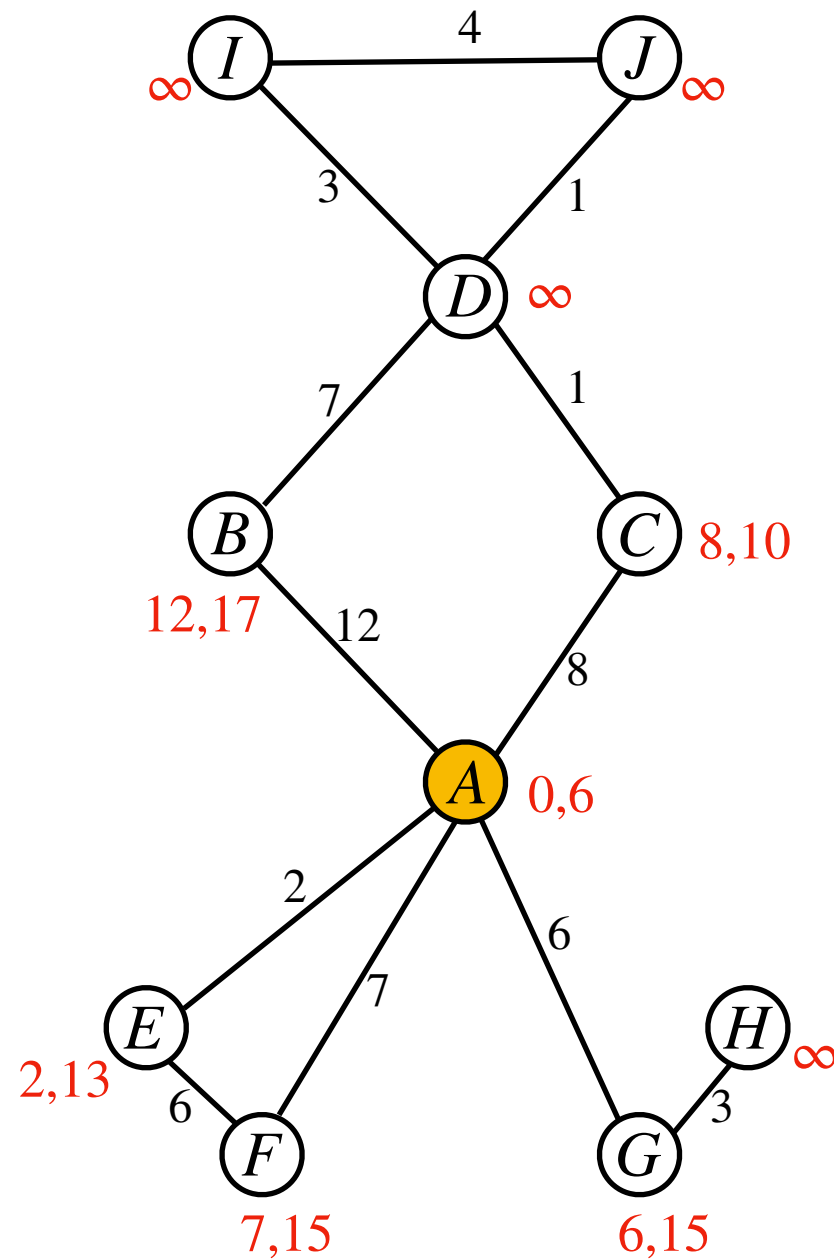
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

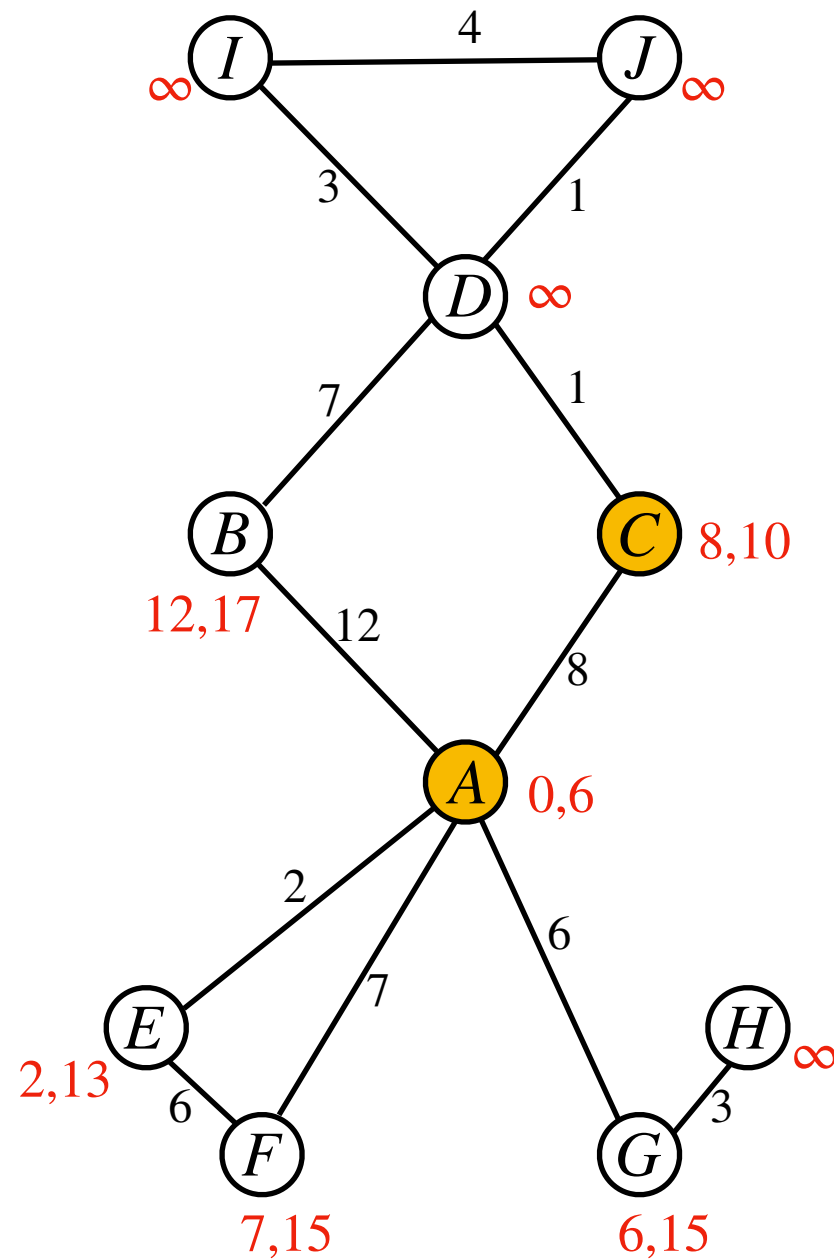
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

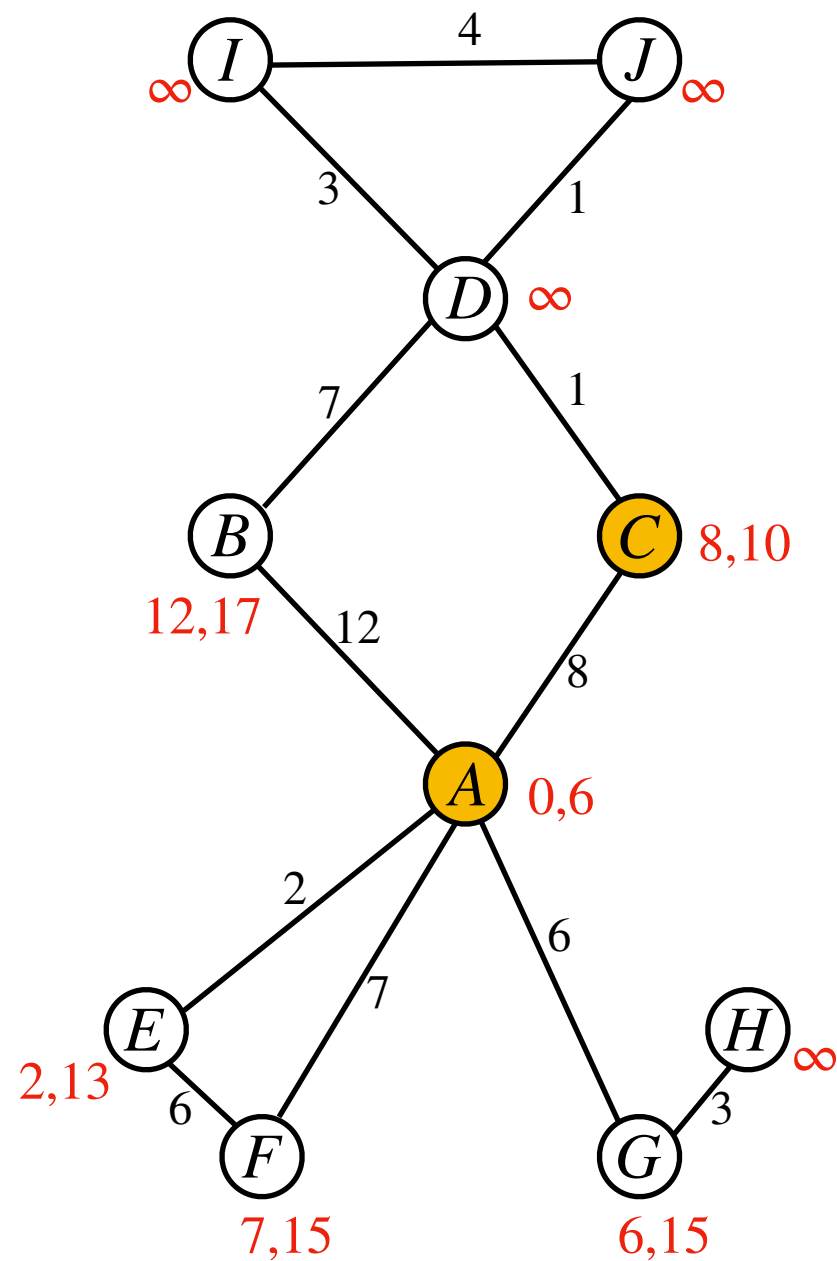
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

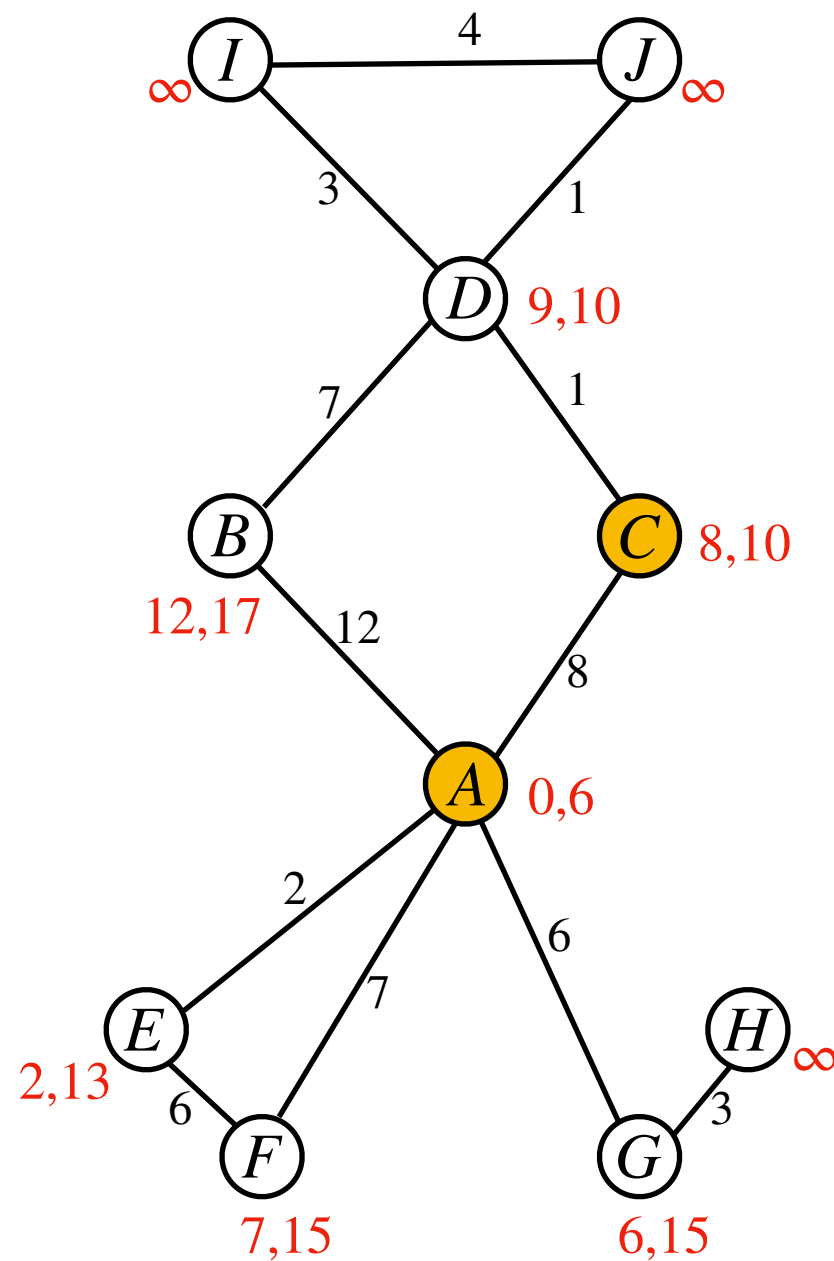
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

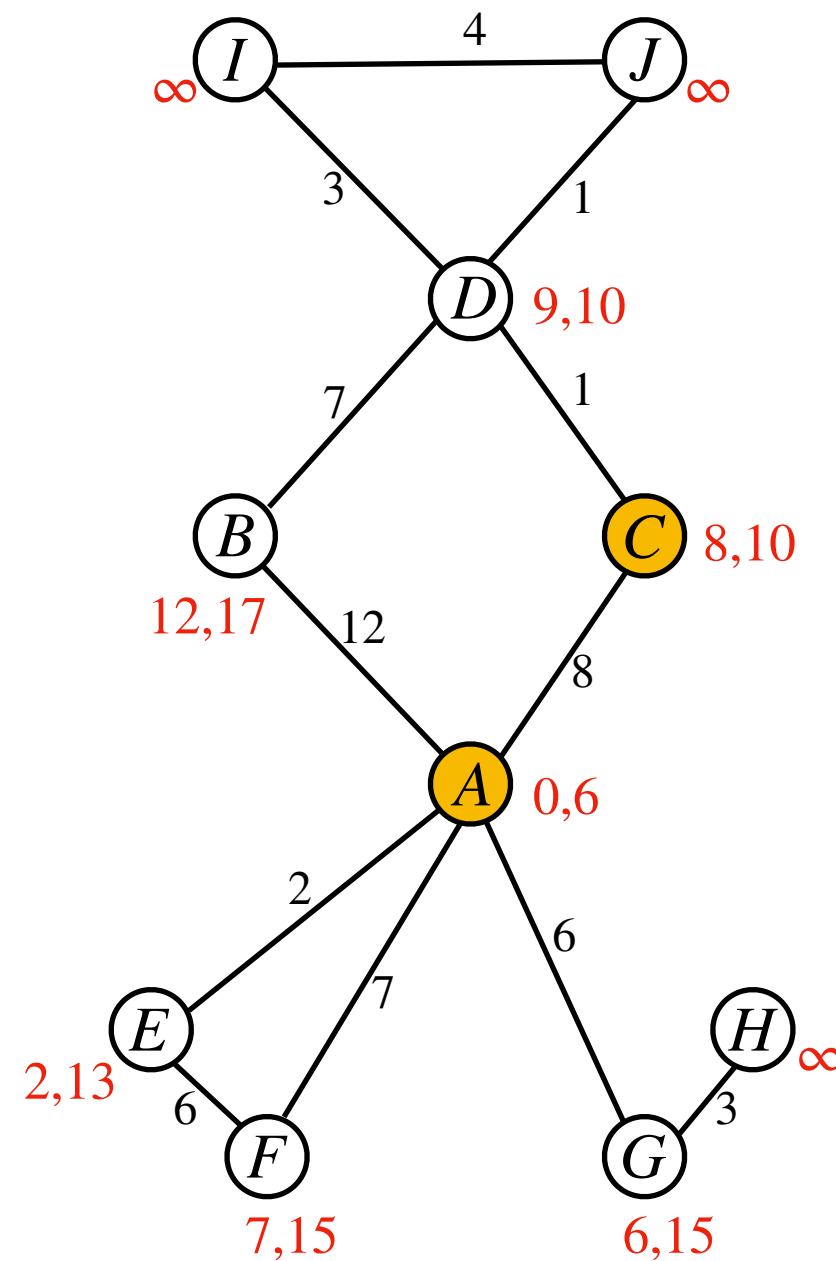
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

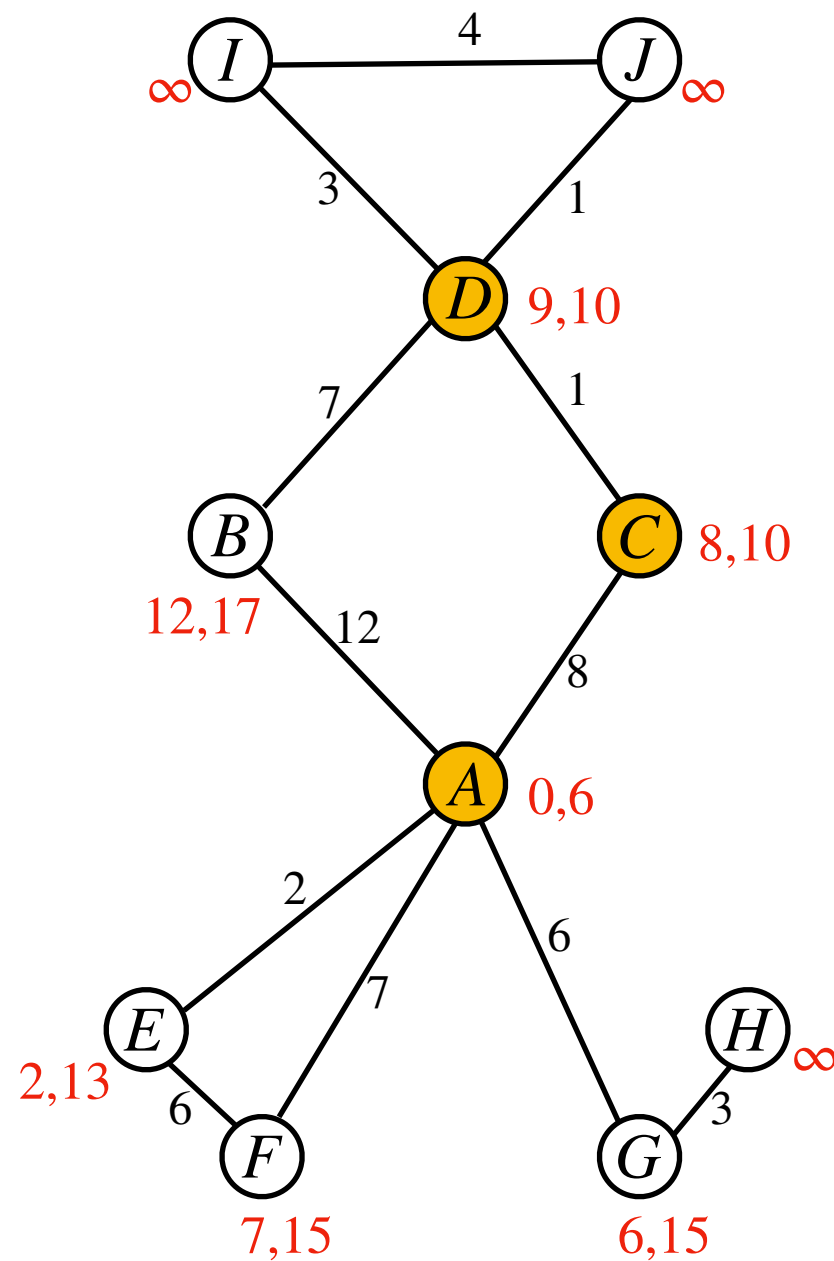
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

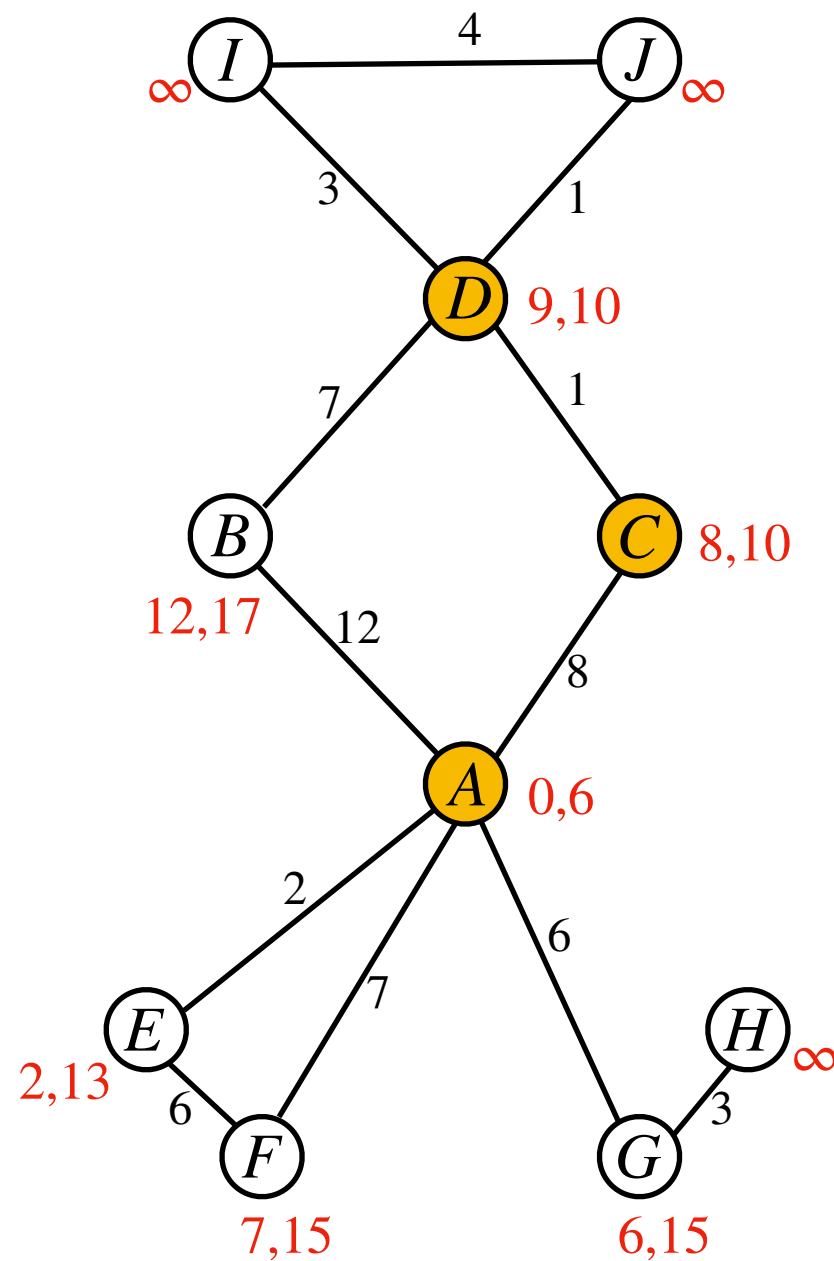
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

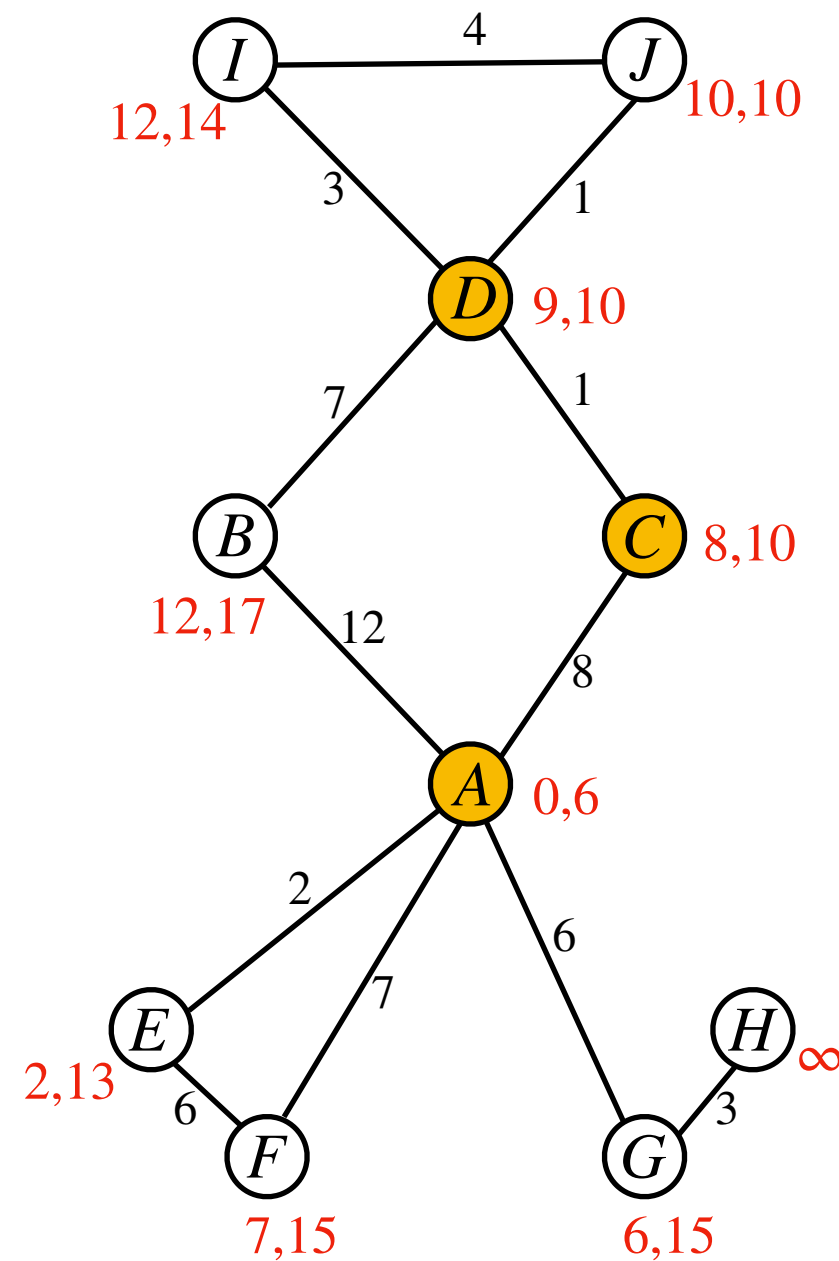
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

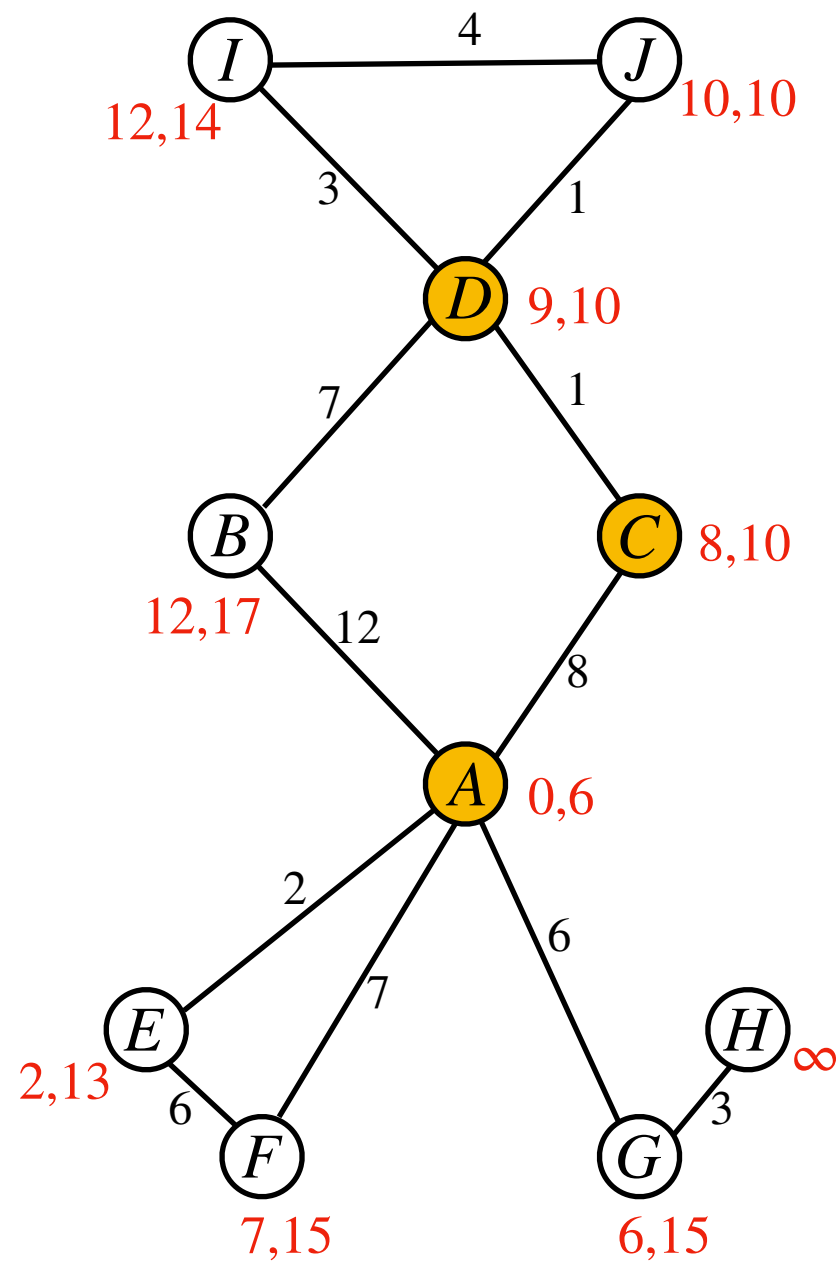
A* Algorithm



Vertex	Distance to J h[i]
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

3. Update the cost of getting from the source vertex to every other vertex: $\min \left\{ d[j], d[i] - h[i] + w[i, j] + h[j] \right\}$

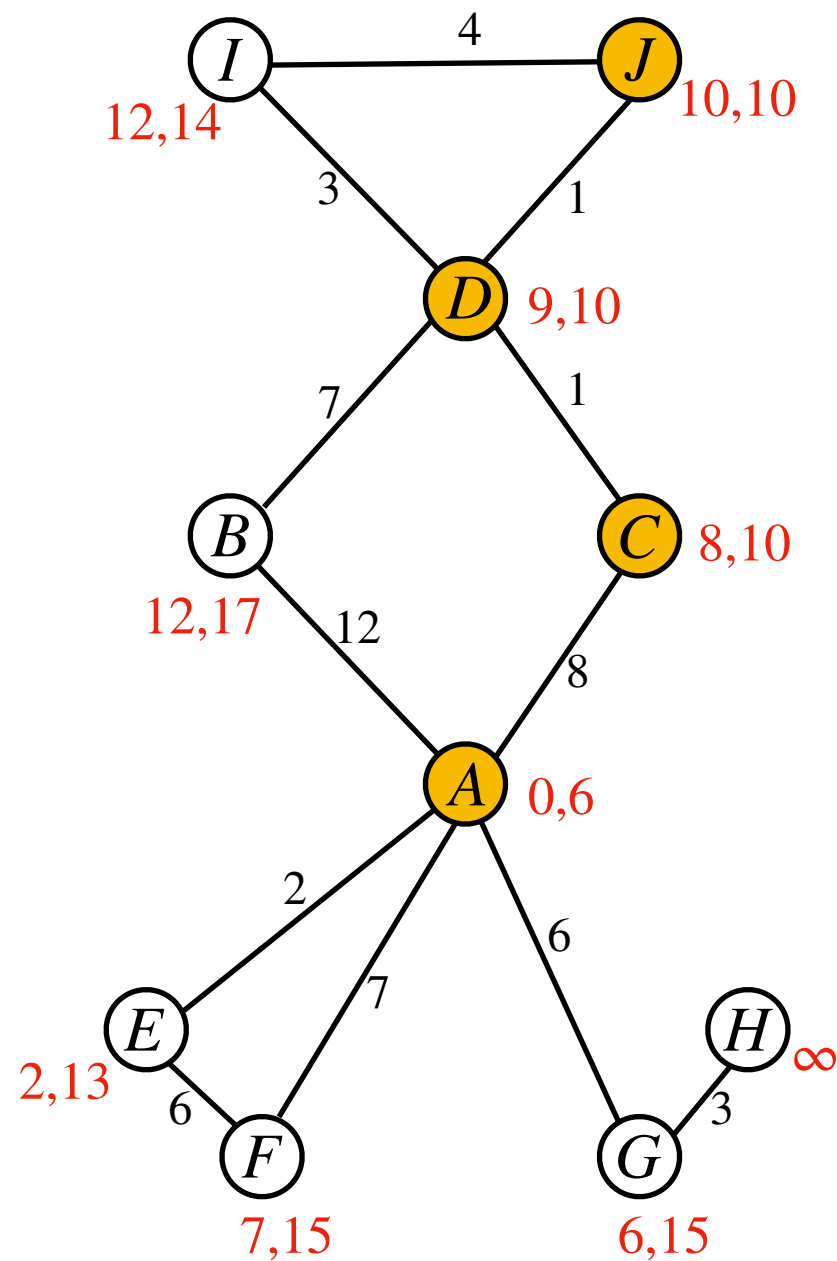
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

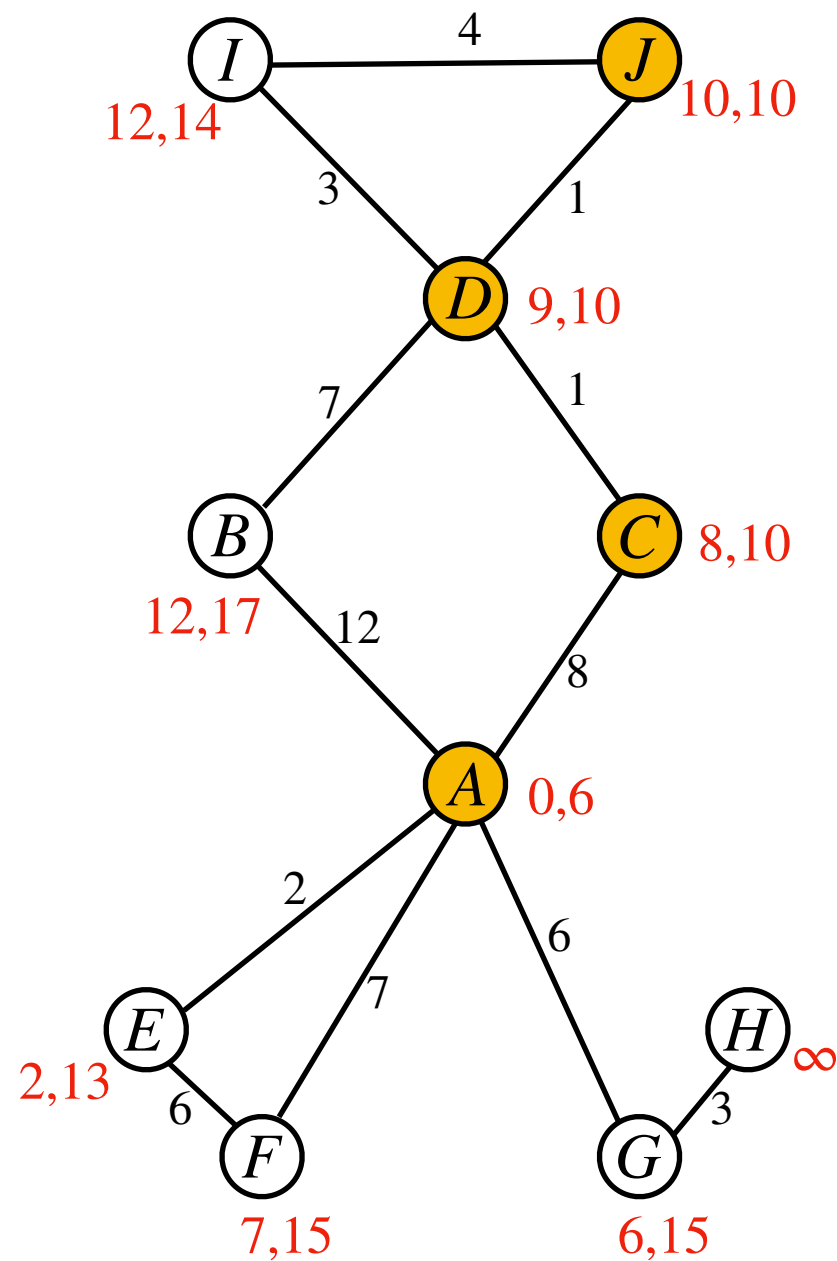
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

4. Choose the unvisited vertex with the smallest $d[i]$ value and visit it (again)

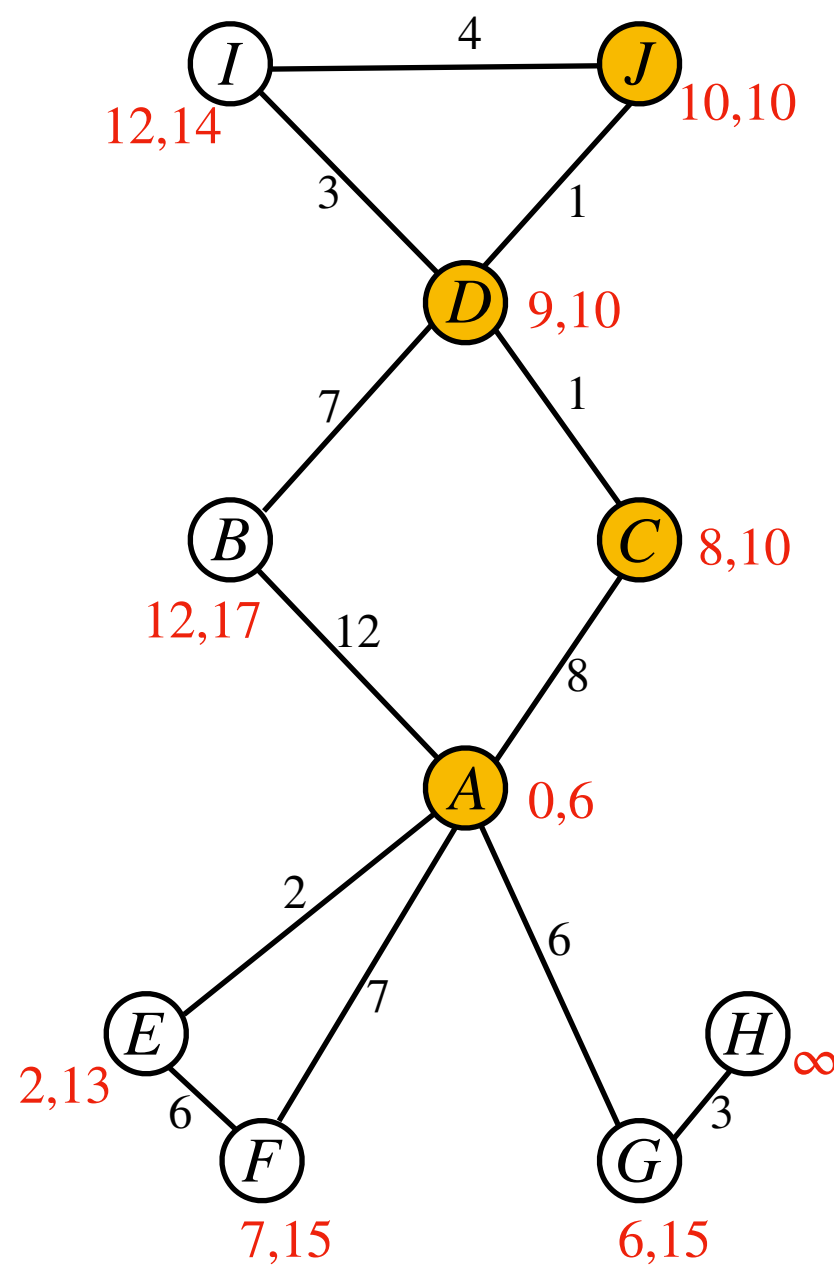
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

6. Terminate as J is the goal

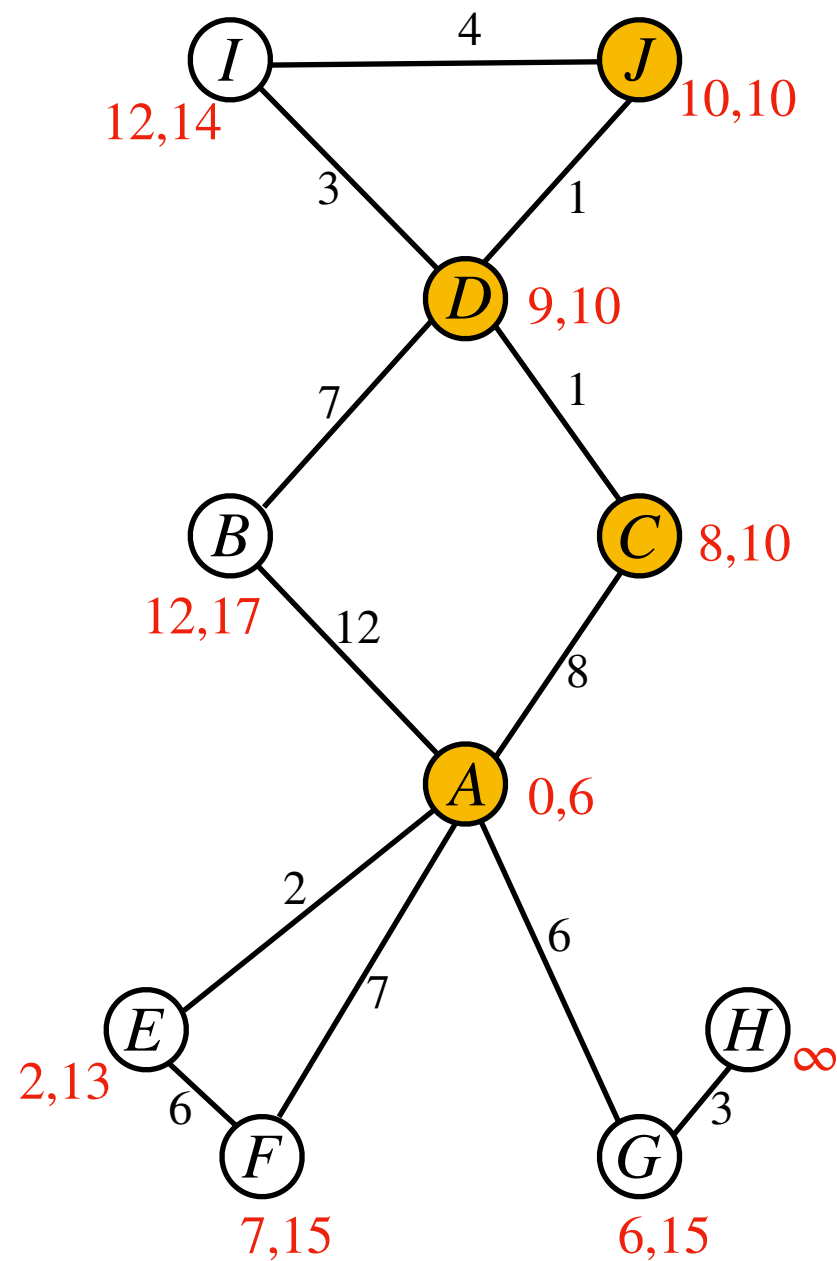
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

6. Terminate as J is the goal

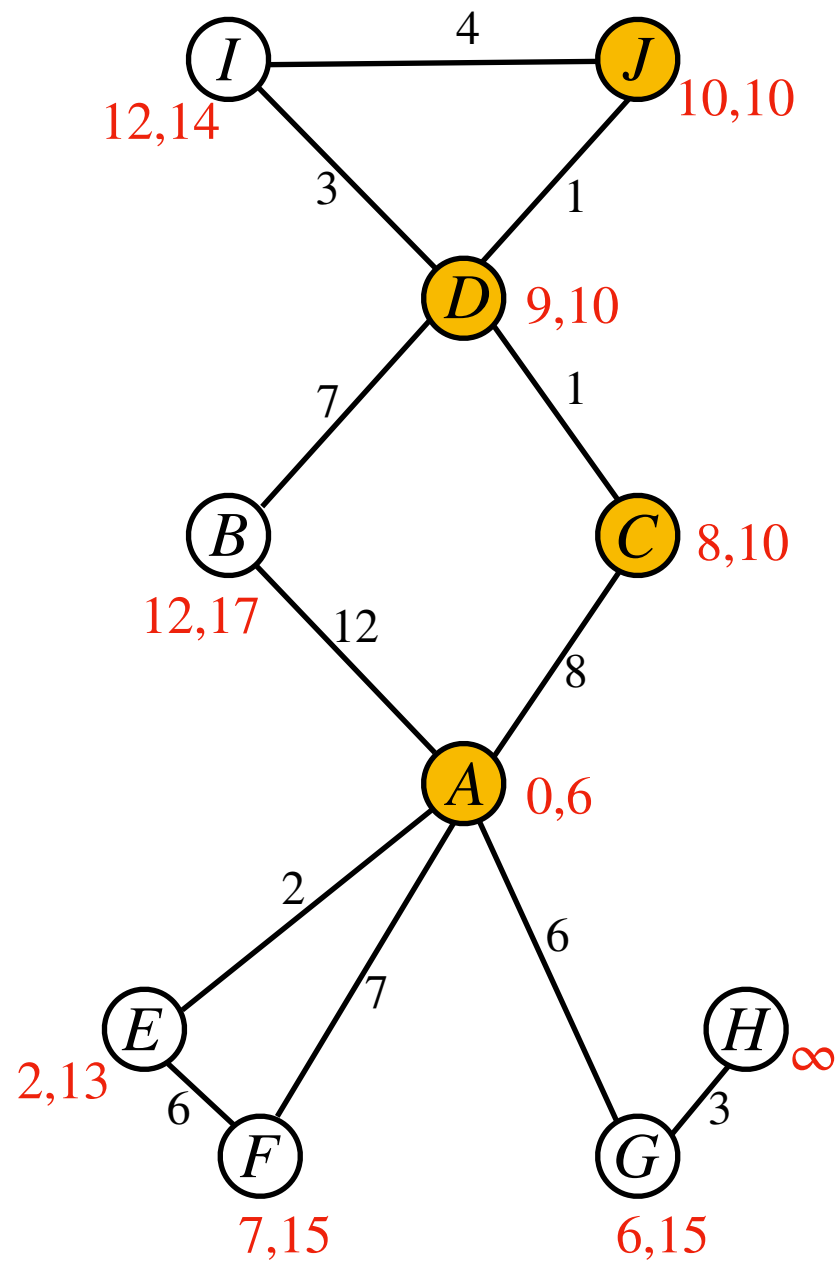
A* Algorithm



Vertex	Distance to J $h[i]$
A	6
B	5
C	2
D	1
E	11
F	8
G	9
H	10
I	2
J	0

6. Terminate as J is the goal

Follow the vertices backwards to the source: $J \rightarrow D \rightarrow C \rightarrow A$



A* Algorithm

- Initialise $d[i]$ to ∞
- Calculate heuristic $h[i]$ to goal from each vertex
- Initialise $d[\text{start}]$ to $0 + h[\text{start}]$

While there are unvisited vertices:

Find the minimum $d[i]$ vertex

Visit the vertex

Update $d[i]$ for its unvisited neighbours

A* Algorithm

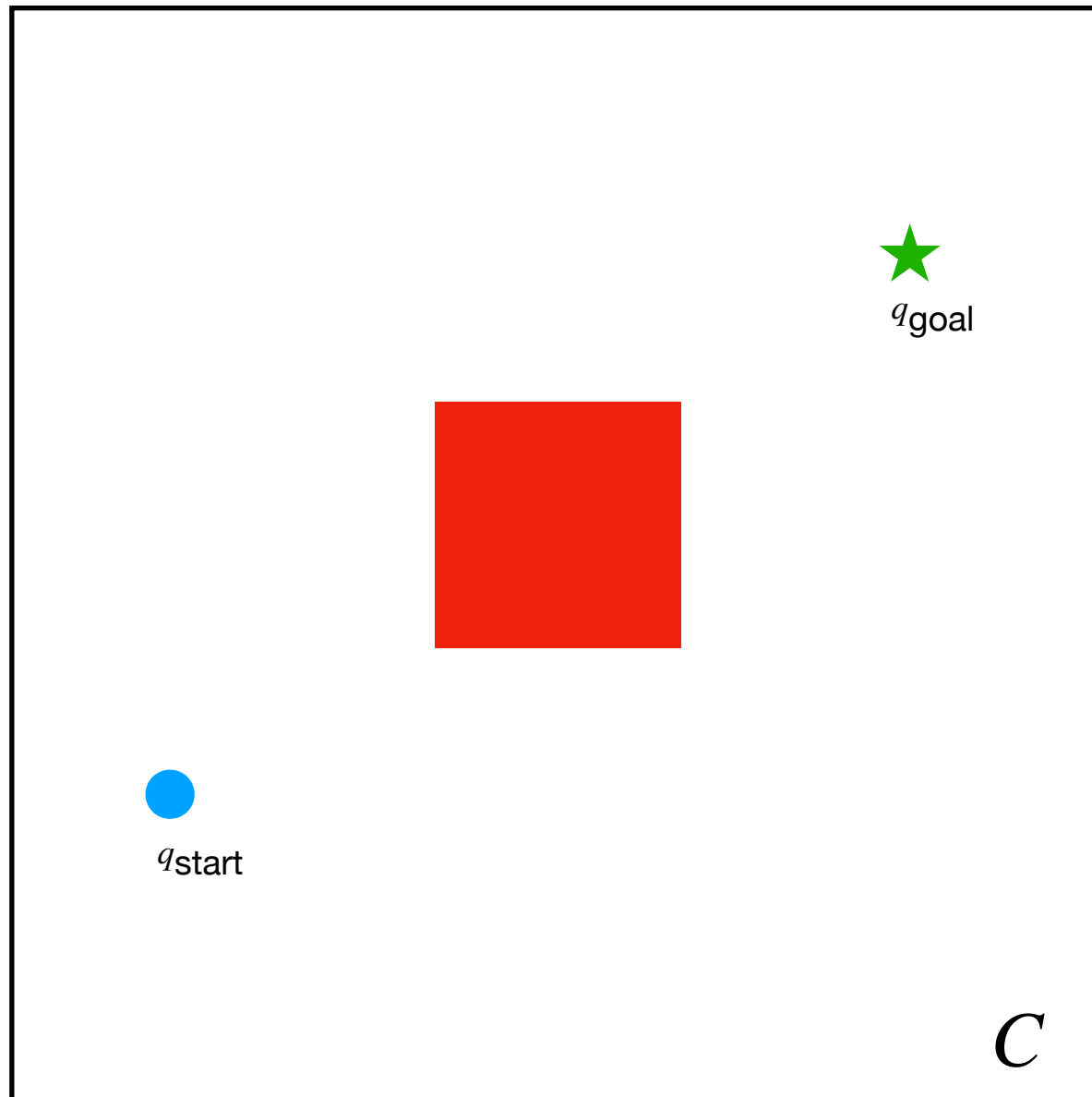
The heuristic function $h[i]$ tells A* an estimate of the minimum cost from any vertex i to the goal. It's important to choose a good heuristic function!

- At one extreme, if $h[i]$ is 0, then only $g[i]$ plays a role, and A* turns into Dijkstra's Algorithm, which is guaranteed to find a shortest path.
- If $h[i]$ is always lower than (or equal to) the cost of moving from i to the goal, then A* is guaranteed to find a shortest path. The lower $h[i]$ is, the more node A* expands, making it slower.
- If $h[i]$ is exactly equal to the cost of moving from i to the goal, then A* will only follow the best path and never expand anything else, making it very fast.
- If $h[i]$ is sometimes greater than the cost of moving from i to the goal, then A* is not guaranteed to find a shortest path, but it can run faster.
- At the other extreme, if $h[i]$ is very high relative to $g[i]$, then only $h[i]$ plays a role, and A* turns into Greedy Best-First-Search.

Resources

- Chapter 2.1, 2.2 and 5.1 from Planning Algorithms by Lavalle
- <https://www.redblobgames.com/pathfinding/a-star/introduction.html>
- <https://qiao.github.io/PathFinding.js/visual/>

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

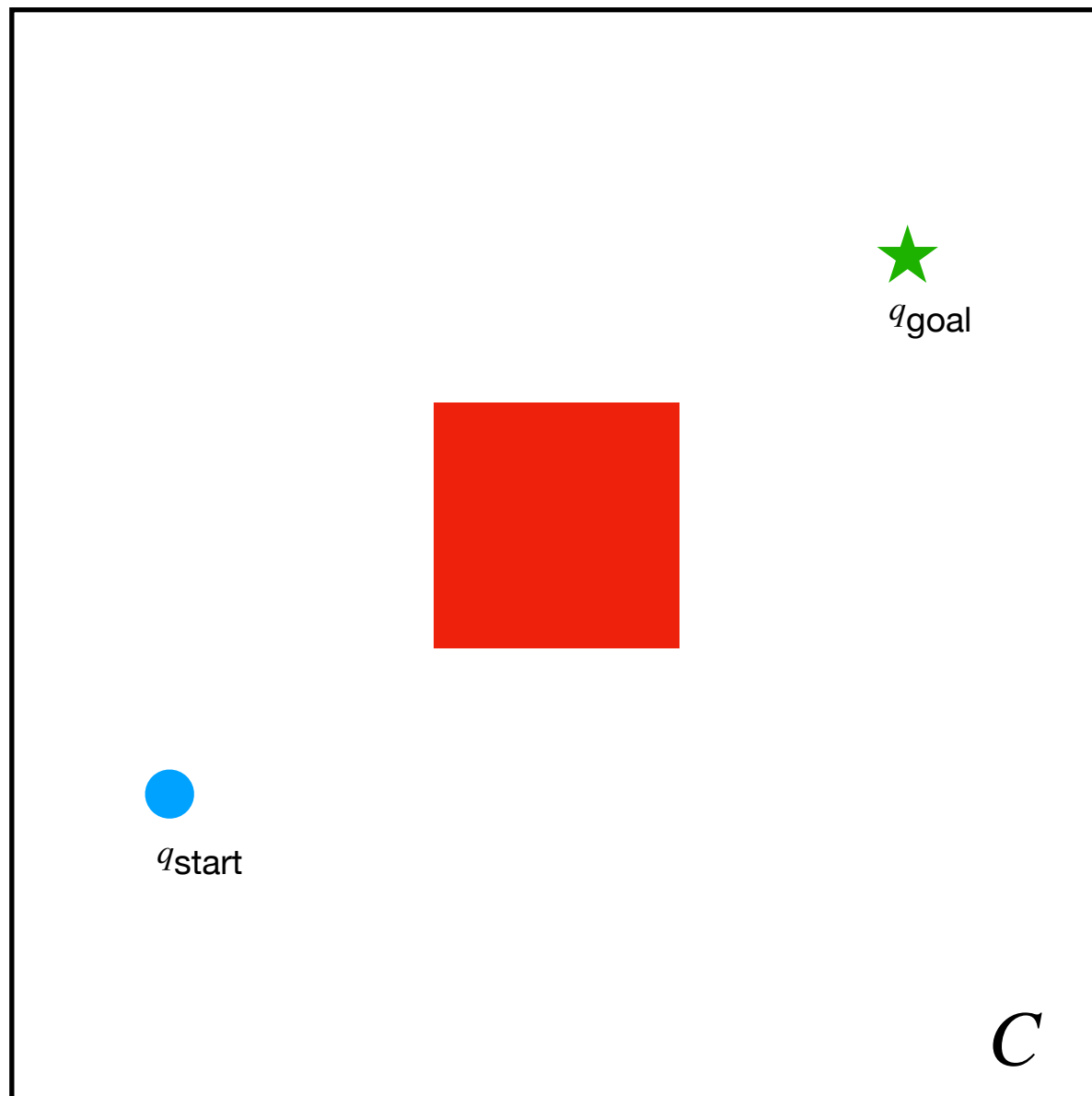
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

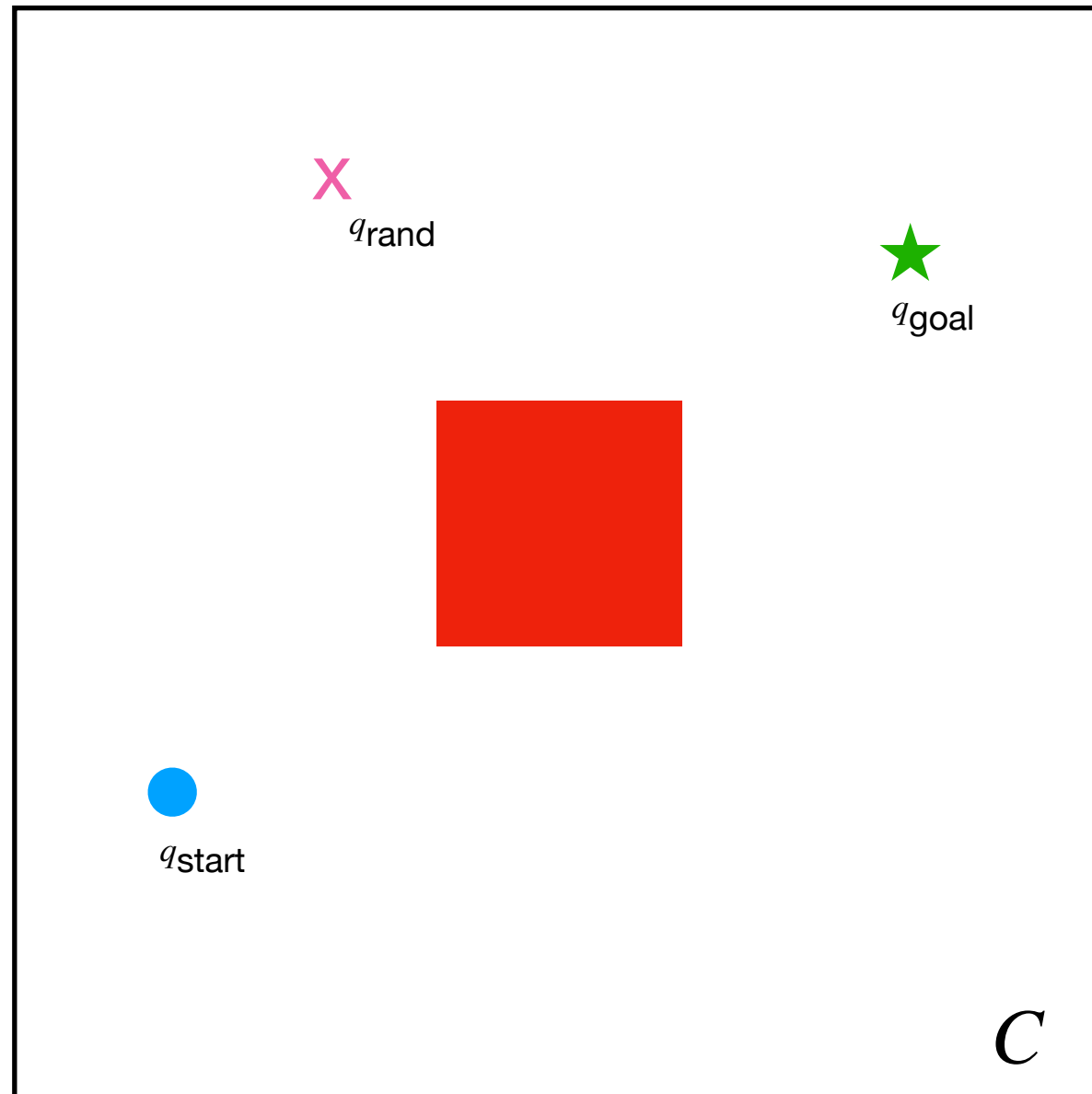
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

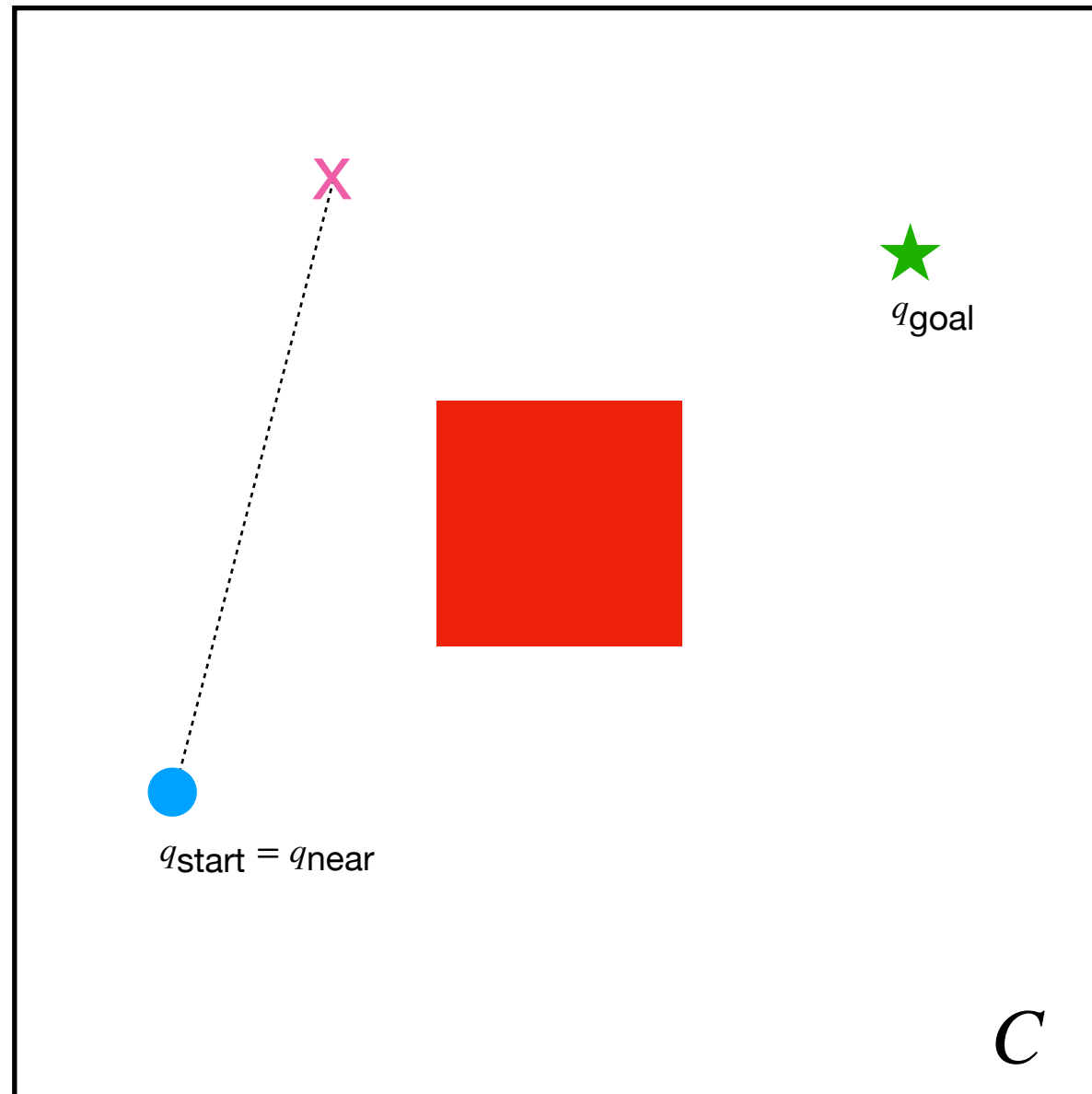
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

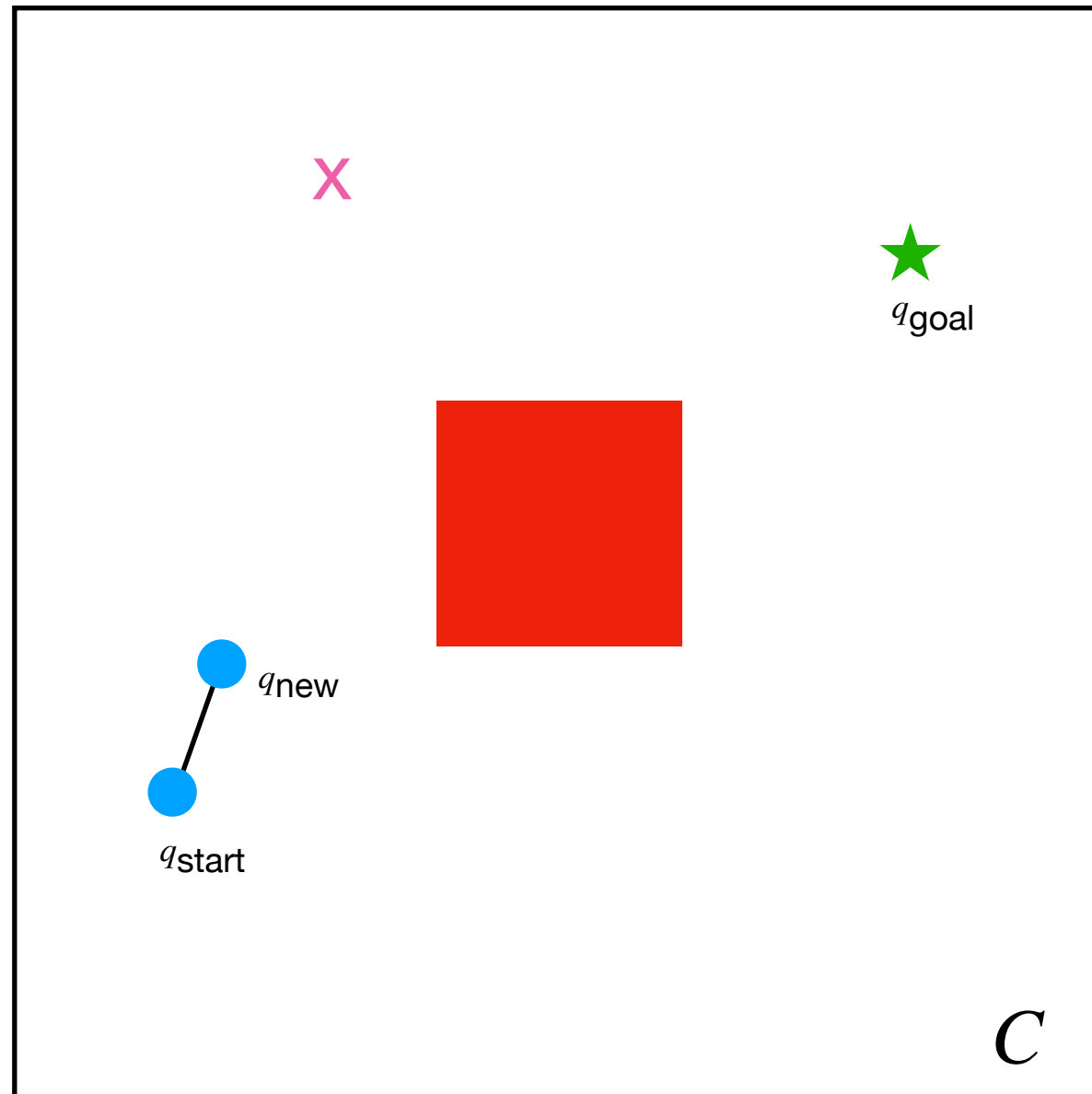
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

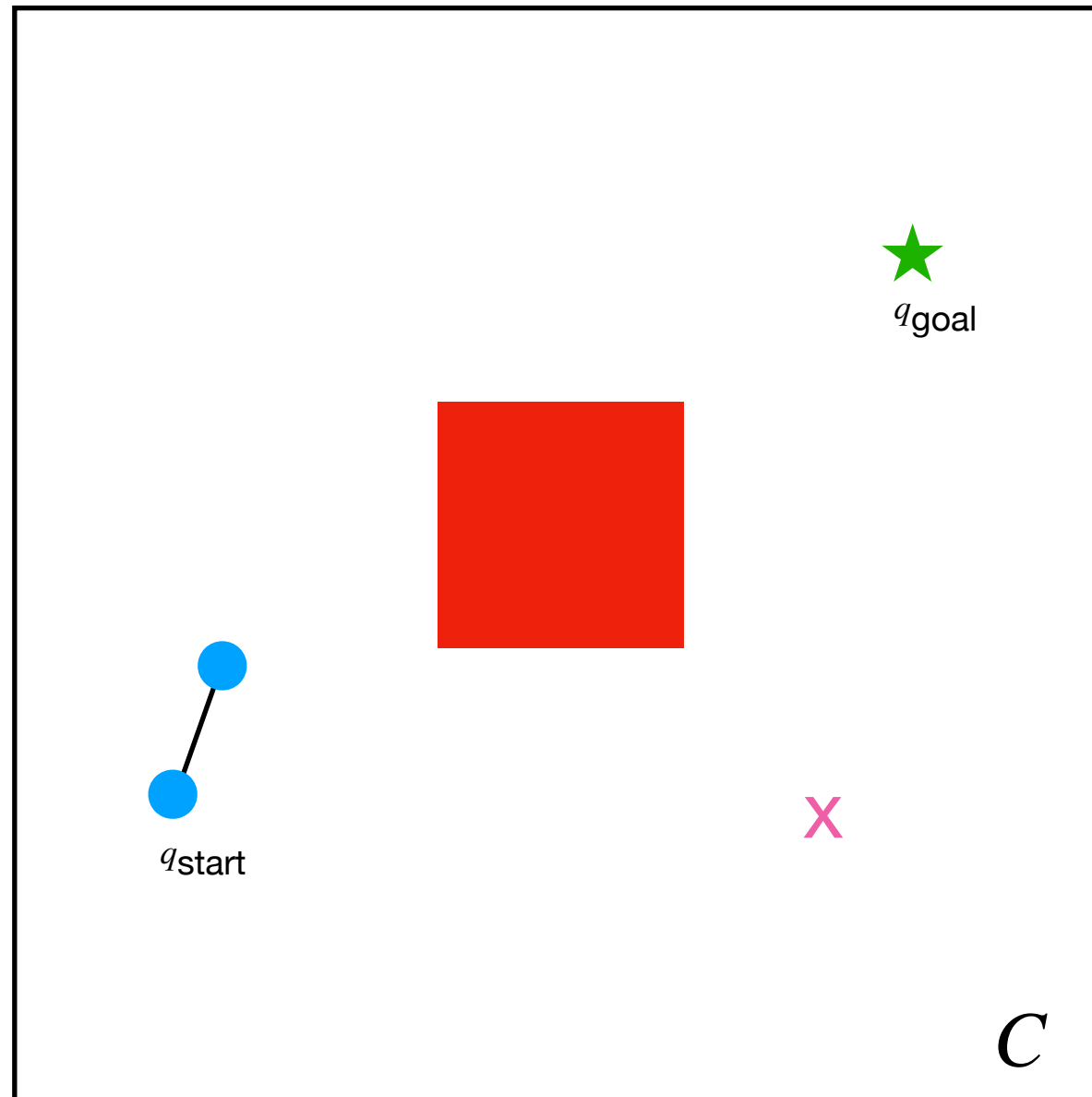
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

 for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

 if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

 if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

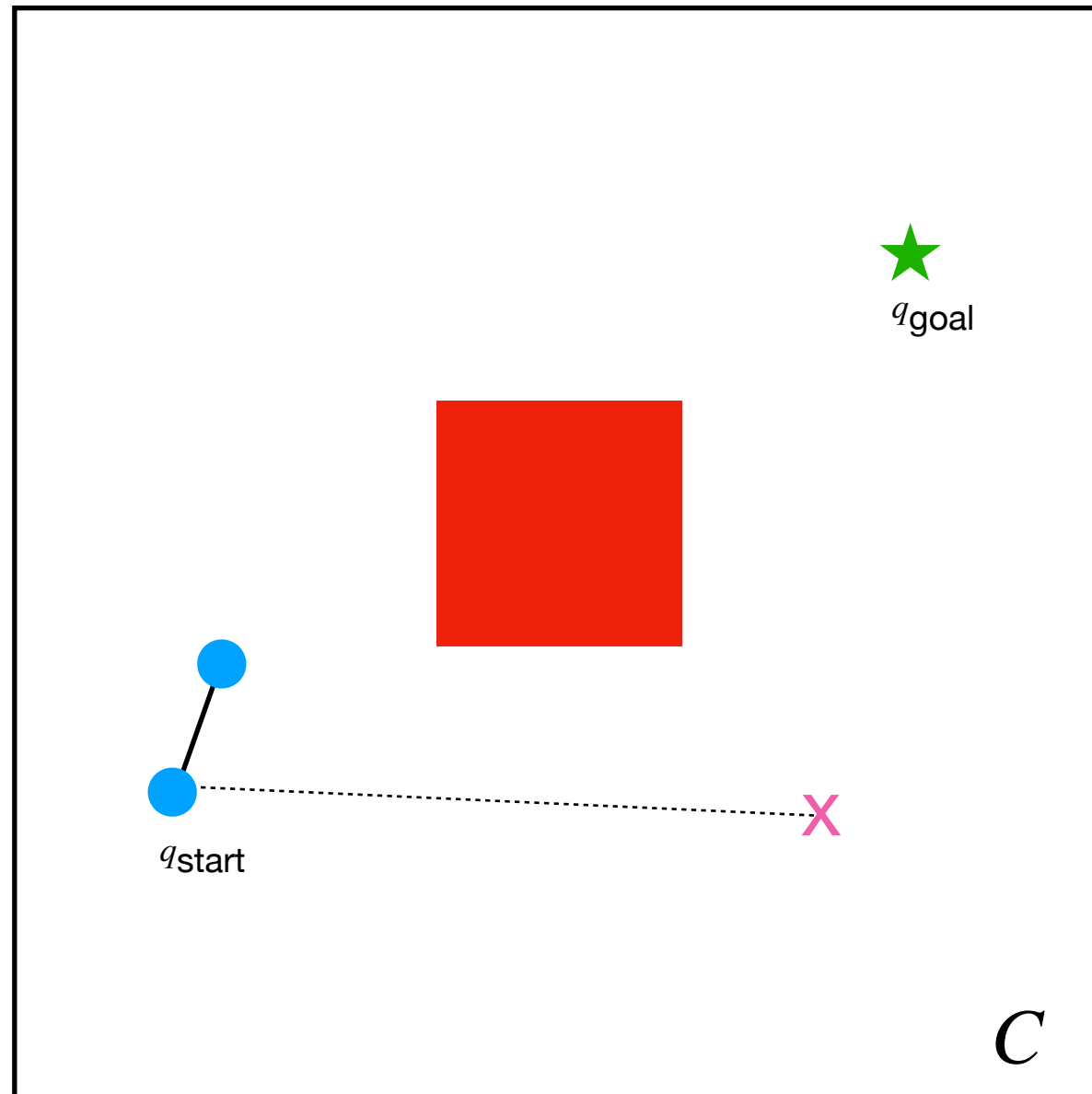
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

 return $\text{Path}(T, q_{start}, q_{goal})$

 return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

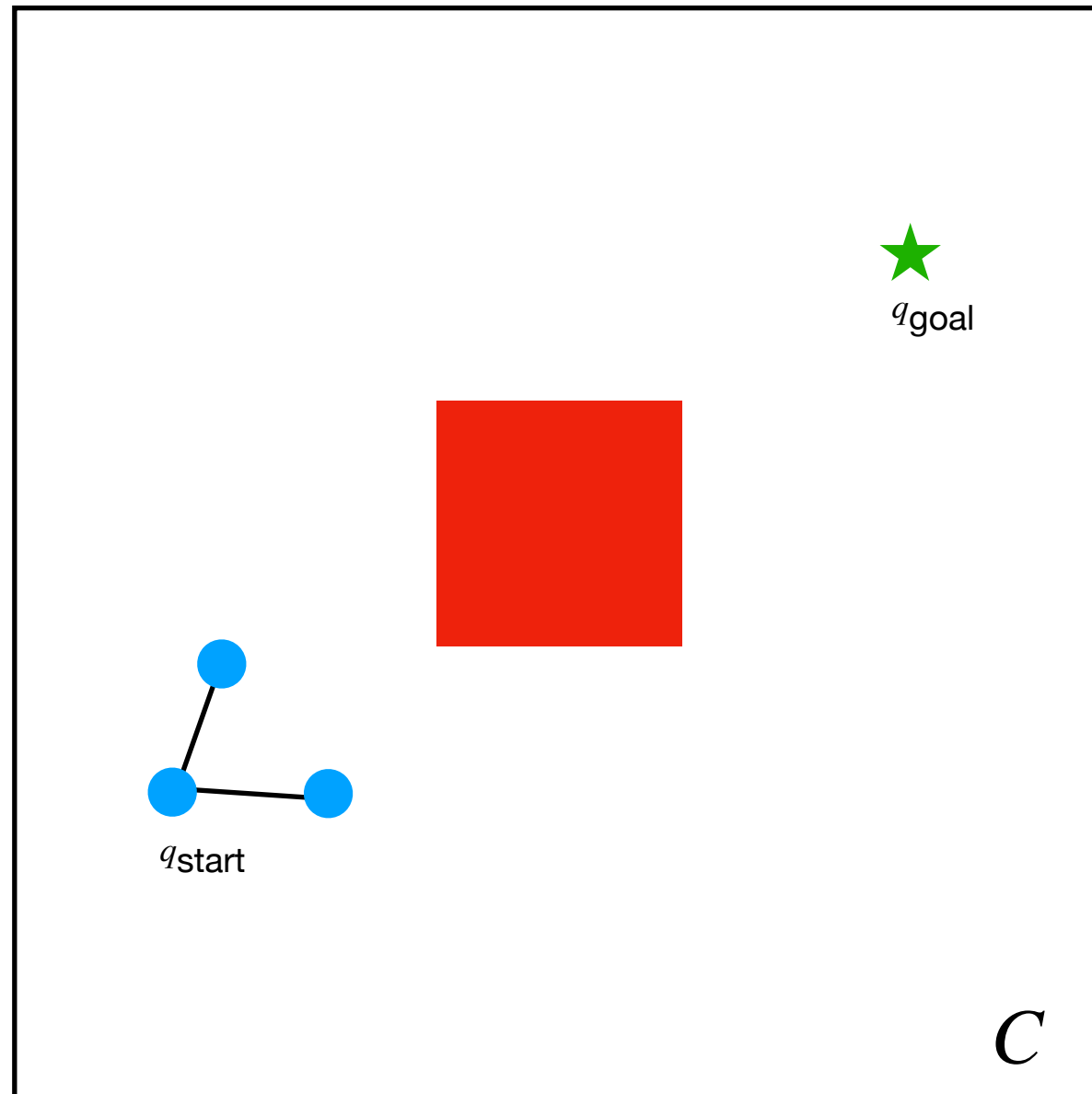
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

Rapidly Exploring Random Trees (RRT)



RRT(q_{start} , q_{goal} , N , δ):

$T.init(q_{start})$

for $i = 1$ to N :

$q_{rand} = \text{SampleRandom}(C_{free})$

$q_{near} = \text{NearestVertex}(T, q_{rand})$

$q_{new} = \text{Steer}(q_{near}, q_{rand}, \delta)$

if $\text{CollisionFree}(q_{near}, q_{new})$:

$T.addVertex(q_{new})$

$T.addEdge(q_{near}, q_{new})$

if $\text{Distance}(q_{new}, q_{goal}) \leq \delta$:

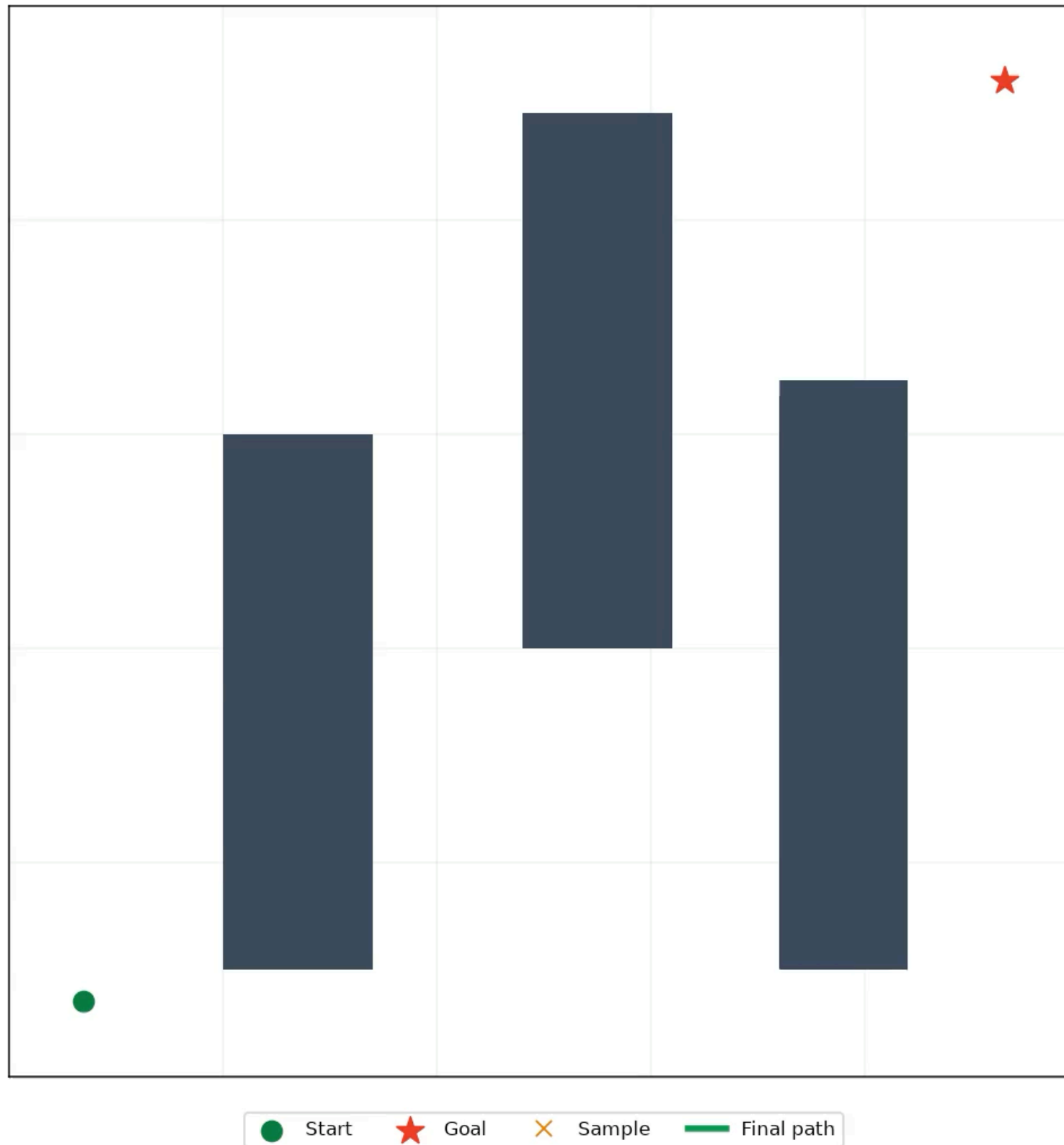
$T.addVertex(q_{goal})$

$T.addEdge(q_{new}, q_{goal})$

return $\text{Path}(T, q_{start}, q_{goal})$

return Failure

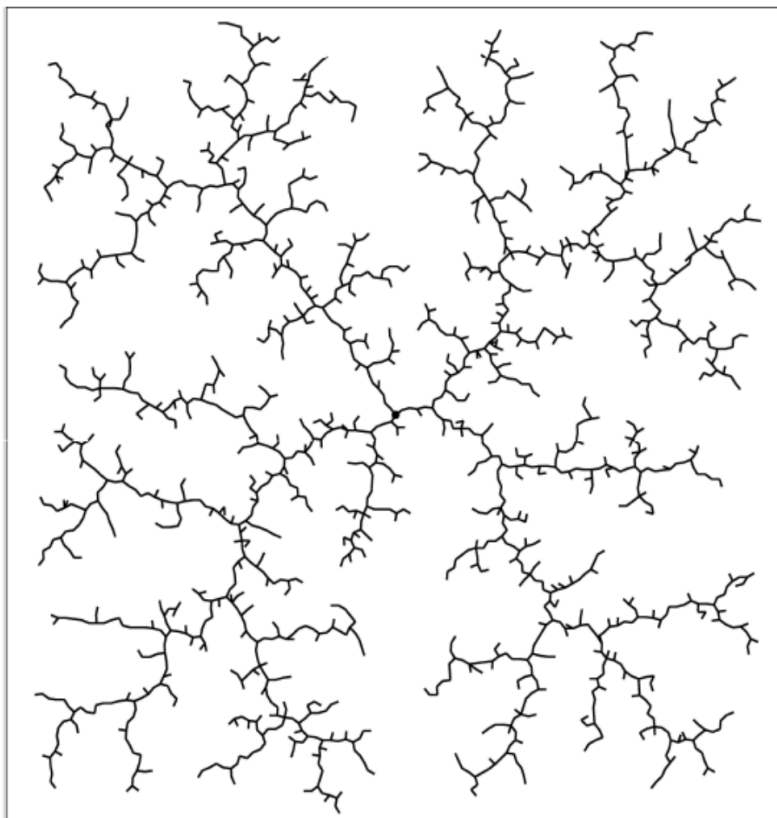
Rapidly Exploring Random Trees (RRT)



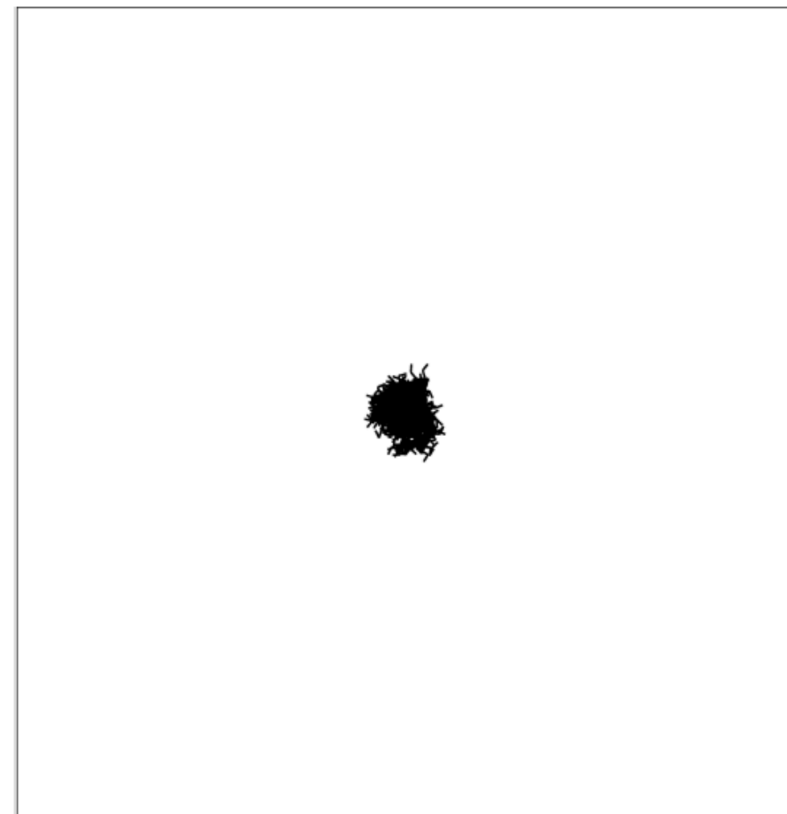
Rapidly Exploring Random Trees (RRT)

- Very efficient in practice
- Highly customisable with lots of variants (RRT*, RRTConnect, BidirectionalRRT)
- Different distributions across different dimensions
- Possibility to add goal bias
- Steer can be used with different distance metrics
- The axis-aligned motions of a grid planner and the randomised motions of sampling planners may lead to jerky motion of a robot
- One approach is to let the planner handle the work of searching globally for a solution, then post-process the resulting motion to make it smoother

RRT



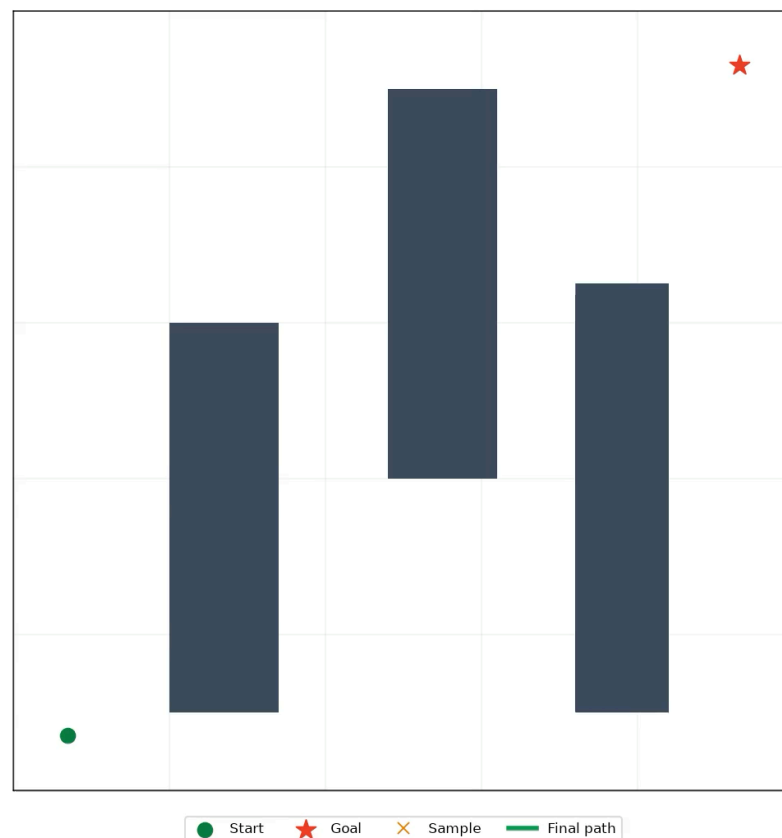
Naïve random tree



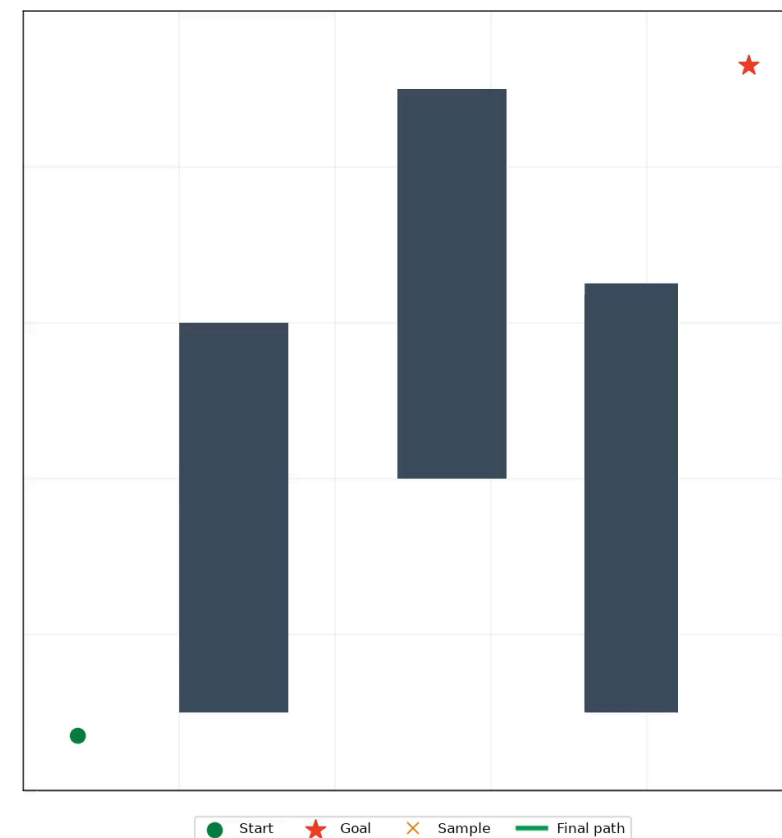
Rapidly Exploring Random Trees (RRT)

- Very efficient in practice
- Highly customisable with lots of variants (RRT*, RRTConnect, BidirectionalRRT)
- Different distributions across different dimensions
- Possibility to add goal bias
- Steer can be used with different distance metrics
- The axis-aligned motions of a grid planner and the randomised motions of sampling planners may lead to jerky motion of a robot
- One approach is to let the planner handle the work of searching globally for a solution, then post-process the resulting motion to make it smoother

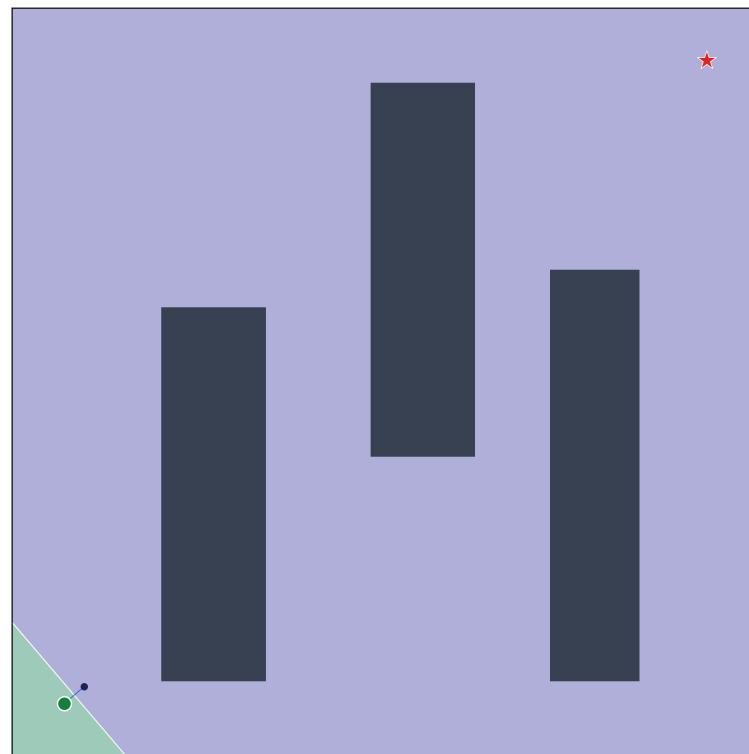
RRT



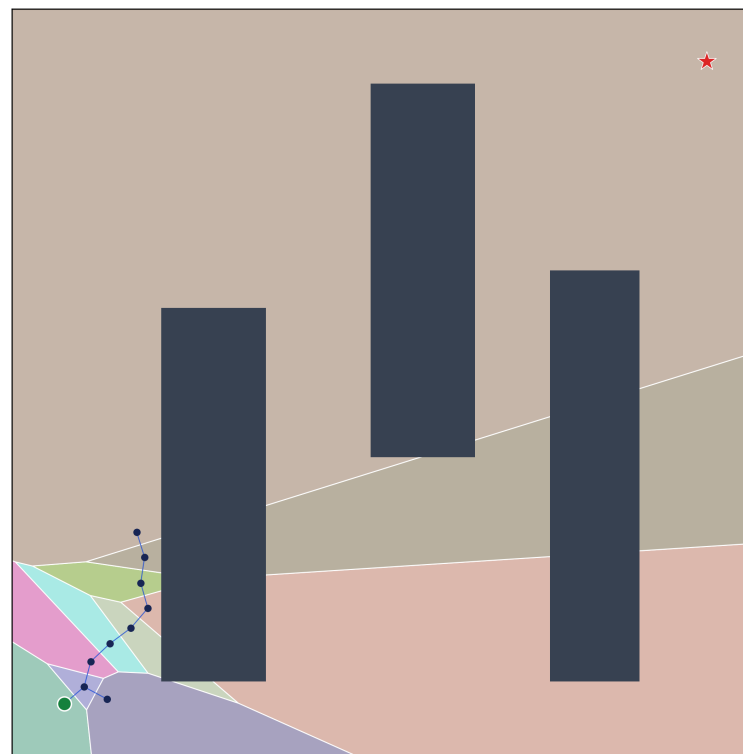
Naïve random tree



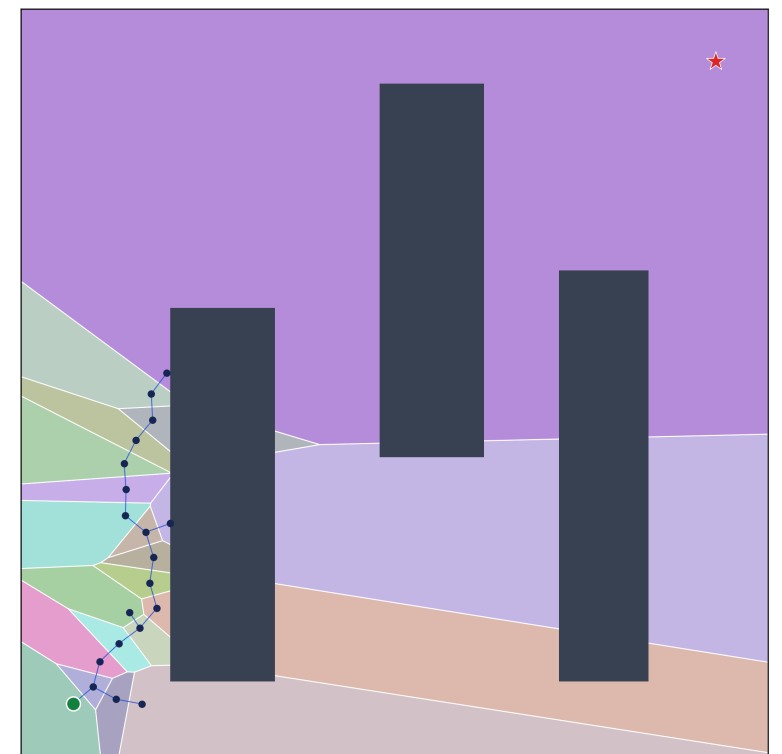
RRT Bias towards larger Voronoi regions



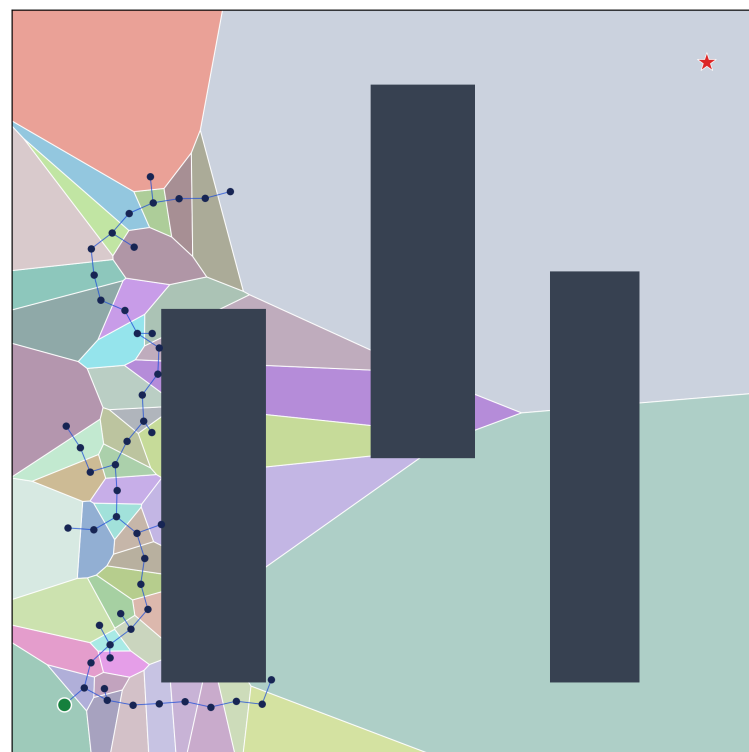
● Start ★ Goal ● Vertices — RRT tree



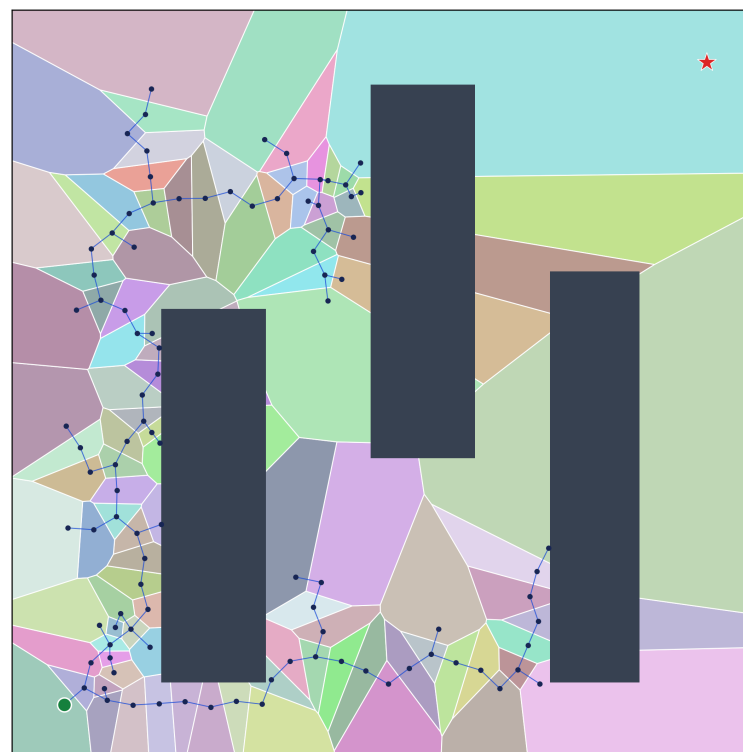
● Start ★ Goal ● Vertices — RRT tree



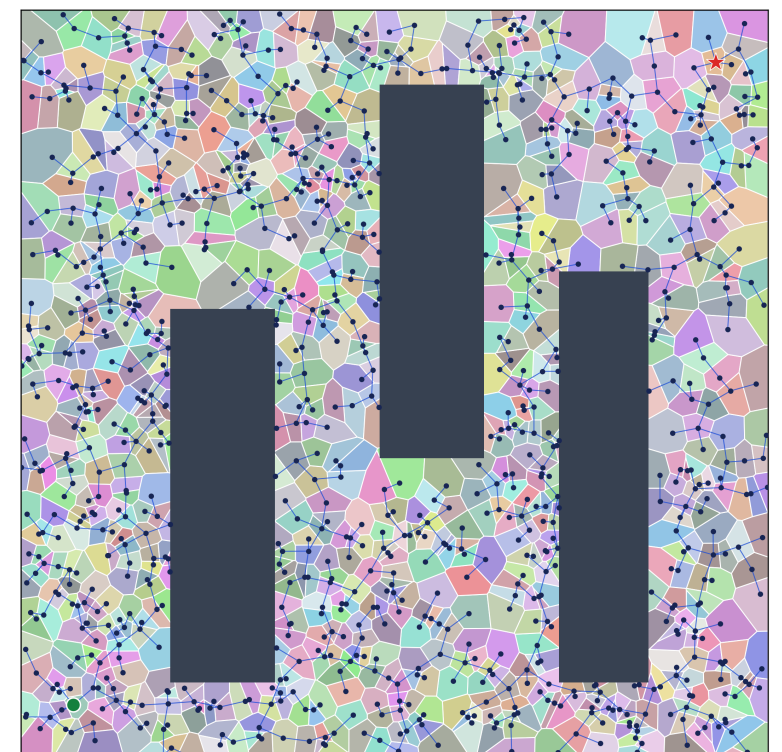
● Start ★ Goal ● Vertices — RRT tree



● Start ★ Goal ● Vertices — RRT tree

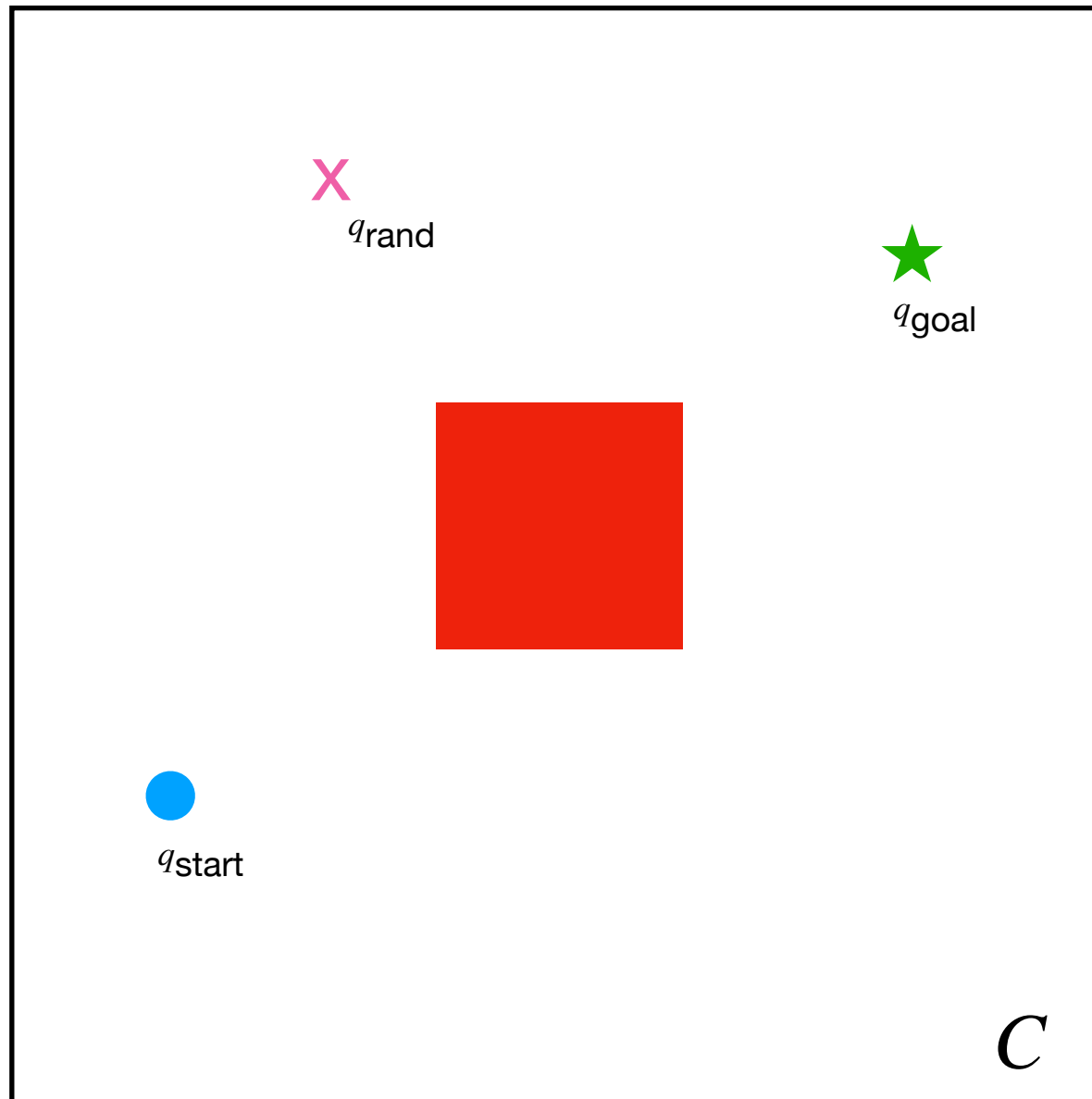


● Start ★ Goal ● Vertices — RRT tree



● Start ★ Goal ● Vertices — RRT tree

Kinodynamic RRT



KinodynamicRRT(x_{start} , X_{goal} , N , dt):

$T.\text{init}(x_{\text{start}})$

for $i = 1$ to N :

$x_{\text{rand}} = \text{SampleRandom}(X_{\text{free}})$

$x_{\text{near}} = \text{NearestVertex}(T, x_{\text{rand}})$

$u_{\text{rand}} = \text{SampleControl}(U)$

$x_{\text{new}} = \text{PropagateDynamics}(x_{\text{near}}, u_{\text{rand}}, dt)$

if $\text{CollisionFreeTrajectory}(x_{\text{near}}, u_{\text{rand}}, dt)$:

$T.\text{addVertex}(x_{\text{new}})$

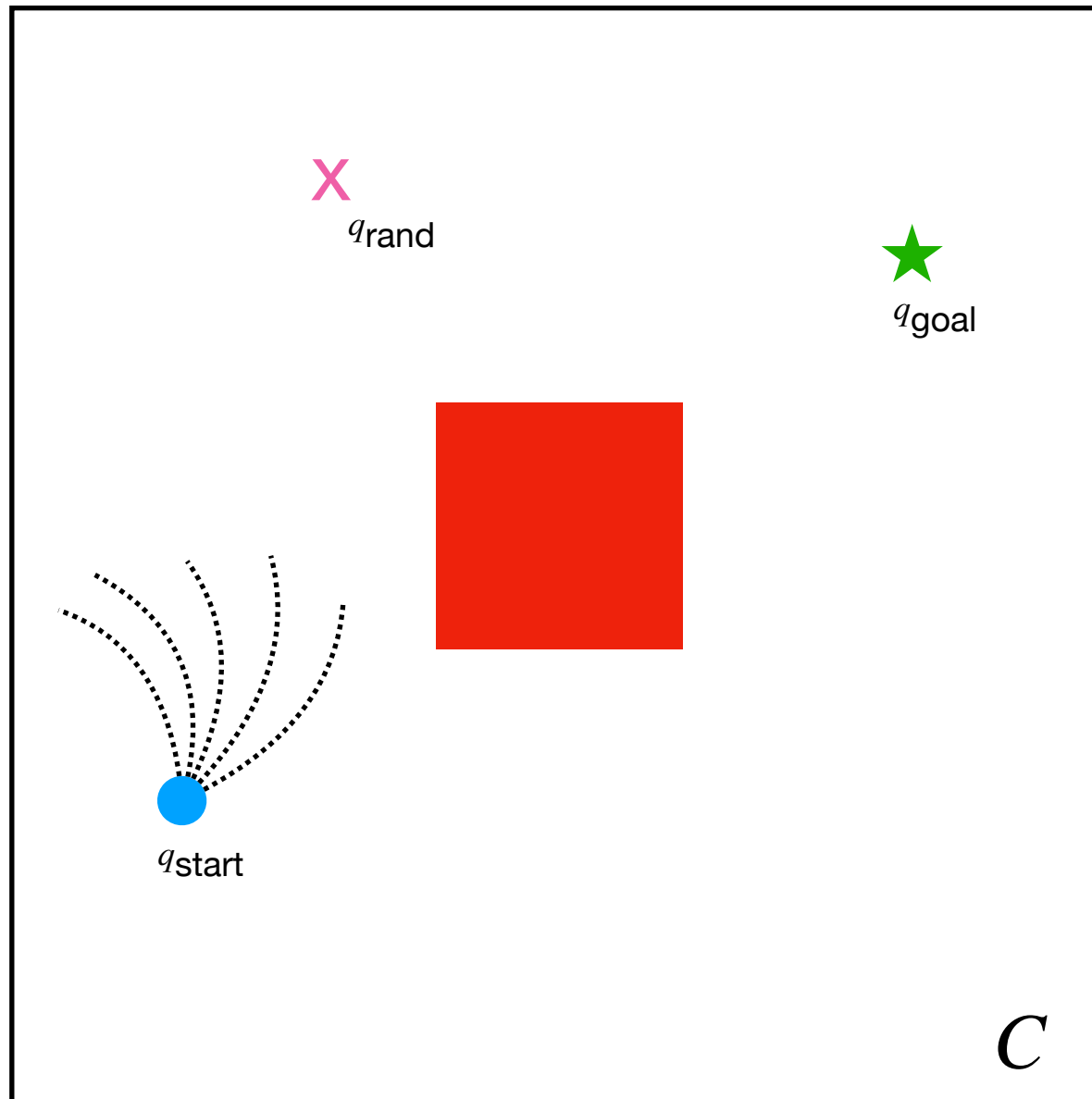
$T.\text{addEdge}(x_{\text{near}}, x_{\text{new}}, u_{\text{rand}}, dt)$

if $x_{\text{new}} \in X_{\text{goal}}$:

return $\text{Trajectory}(T, x_{\text{start}}, x_{\text{new}})$

return Failure

Kinodynamic RRT



KinodynamicRRT(x_{start} , X_{goal} , N , dt):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

$u_{rand} = \text{SampleControl}(U)$

$x_{new} = \text{PropagateDynamics}(x_{near}, u_{rand}, dt)$

if $\text{CollisionFreeTrajectory}(x_{near}, u_{rand}, dt)$:

$T.addVertex(x_{new})$

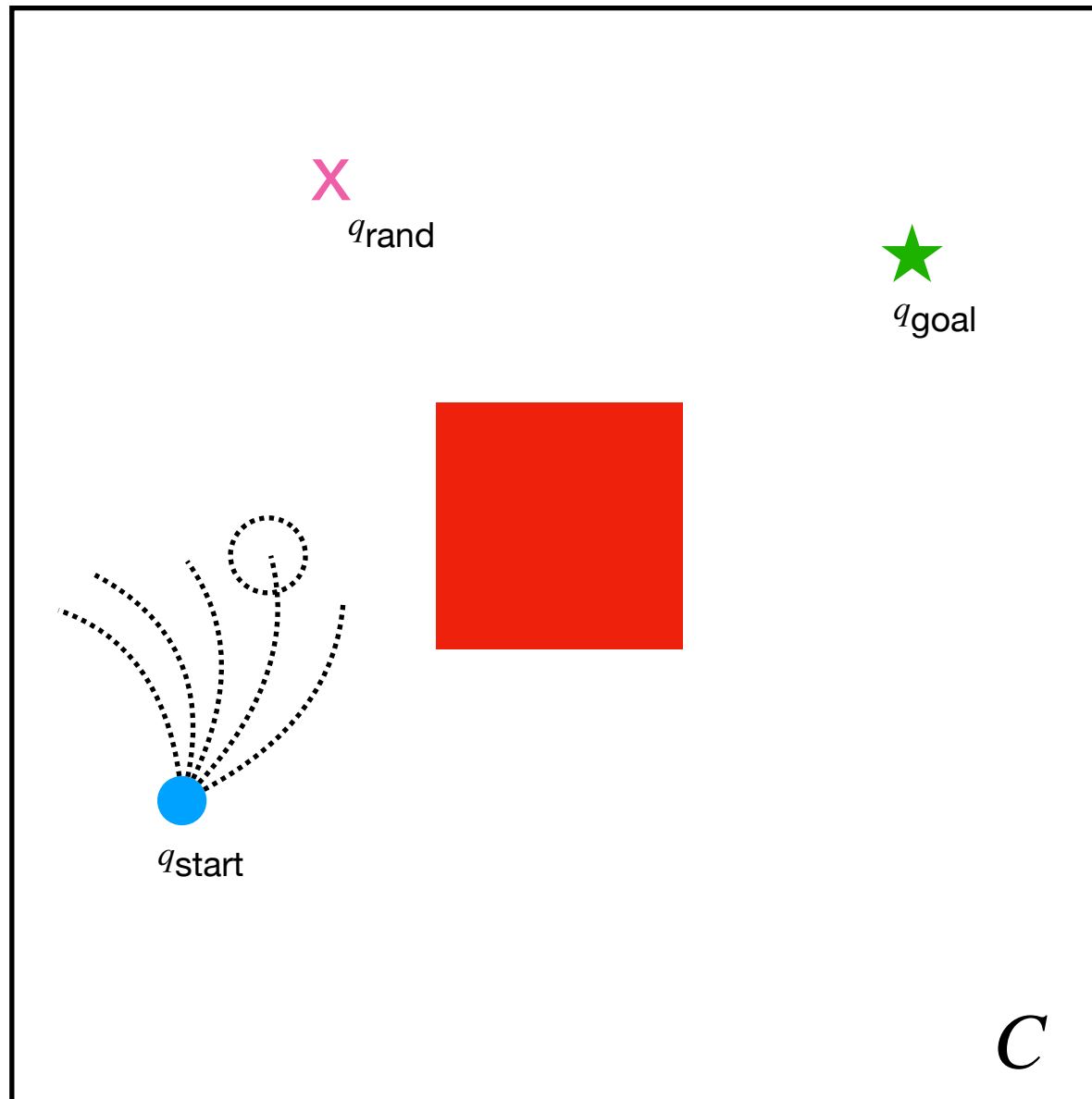
$T.addEdge(x_{near}, x_{new}, u_{rand}, dt)$

if $x_{new} \in X_{goal}$:

return $\text{Trajectory}(T, x_{start}, x_{new})$

return Failure

Kinodynamic RRT



KinodynamicRRT(x_{start} , X_{goal} , N , dt):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

$u_{rand} = \text{SampleControl}(U)$

$x_{new} = \text{PropagateDynamics}(x_{near}, u_{rand}, dt)$

if $\text{CollisionFreeTrajectory}(x_{near}, u_{rand}, dt)$:

$T.addVertex(x_{new})$

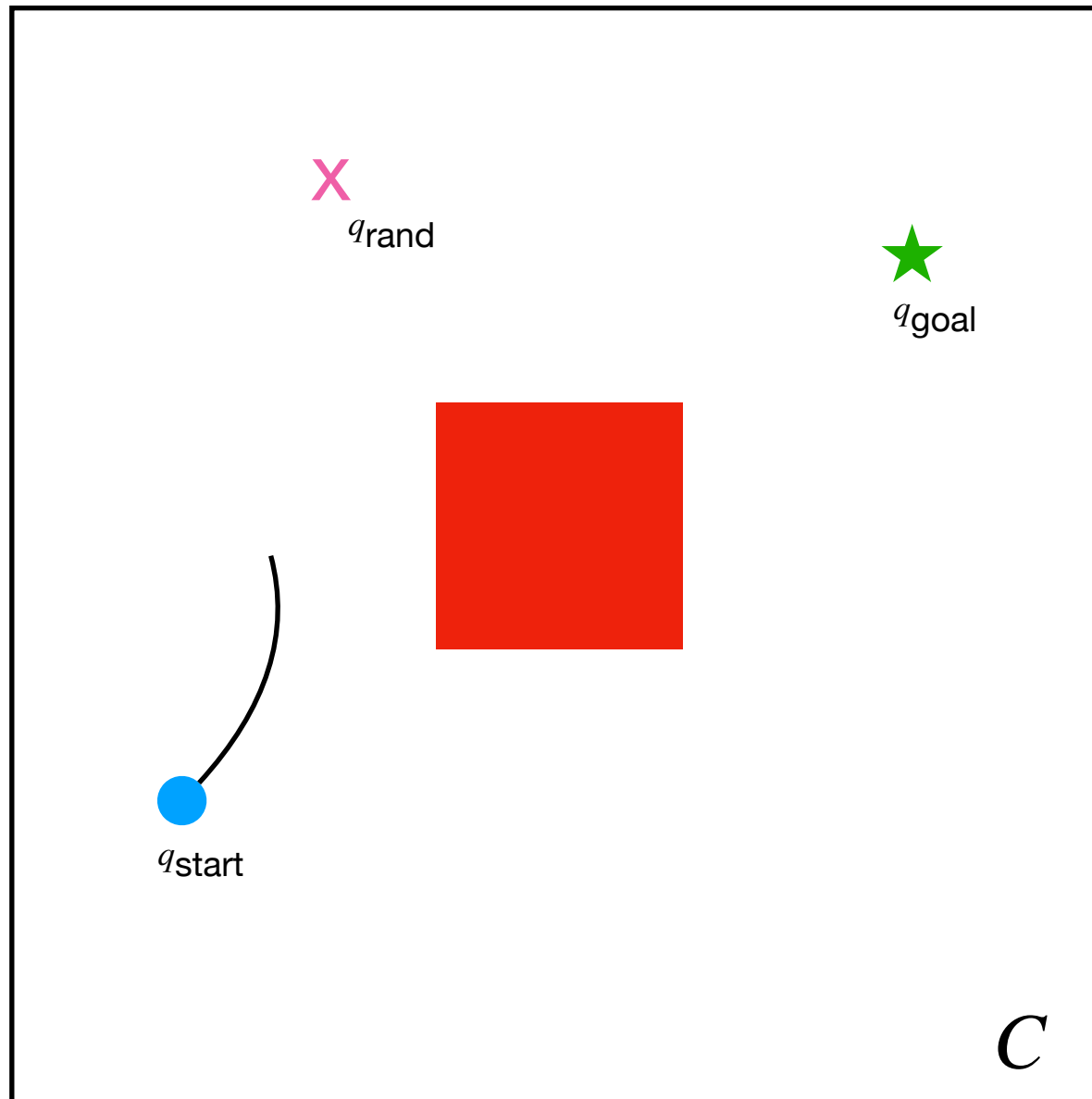
$T.addEdge(x_{near}, x_{new}, u_{rand}, dt)$

if $x_{new} \in X_{goal}$:

return $\text{Trajectory}(T, x_{start}, x_{new})$

return Failure

Kinodynamic RRT



KinodynamicRRT(x_{start} , X_{goal} , N , dt):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

$u_{rand} = \text{SampleControl}(U)$

$x_{new} = \text{PropagateDynamics}(x_{near}, u_{rand}, dt)$

if $\text{CollisionFreeTrajectory}(x_{near}, u_{rand}, dt)$:

$T.addVertex(x_{new})$

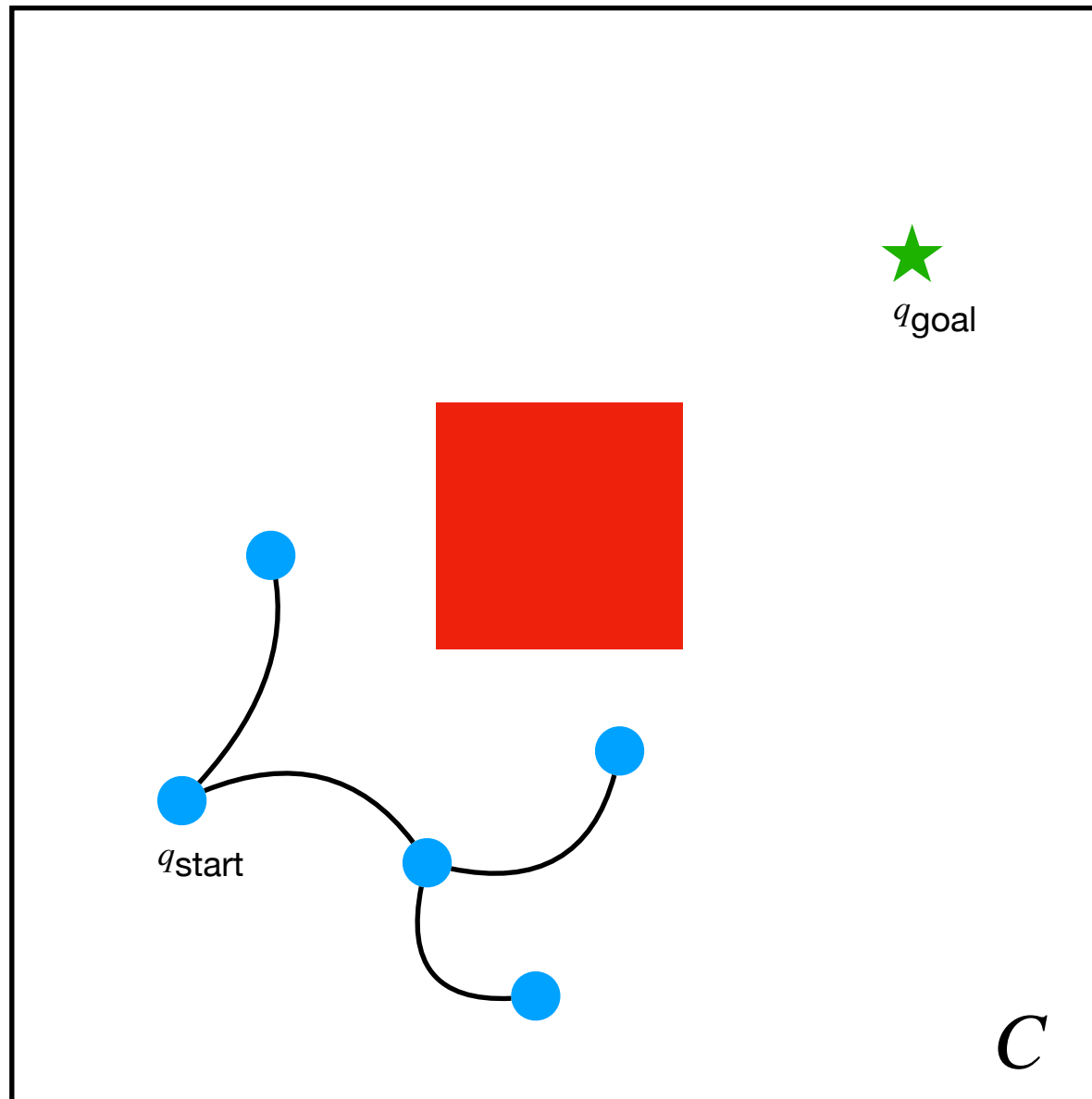
$T.addEdge(x_{near}, x_{new}, u_{rand}, dt)$

if $x_{new} \in X_{goal}$:

return $\text{Trajectory}(T, x_{start}, x_{new})$

return Failure

Kinodynamic RRT



KinodynamicRRT(x_{start} , X_{goal} , N , dt):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

$u_{rand} = \text{SampleControl}(U)$

$x_{new} = \text{PropagateDynamics}(x_{near}, u_{rand}, dt)$

if $\text{CollisionFreeTrajectory}(x_{near}, u_{rand}, dt)$:

$T.addVertex(x_{new})$

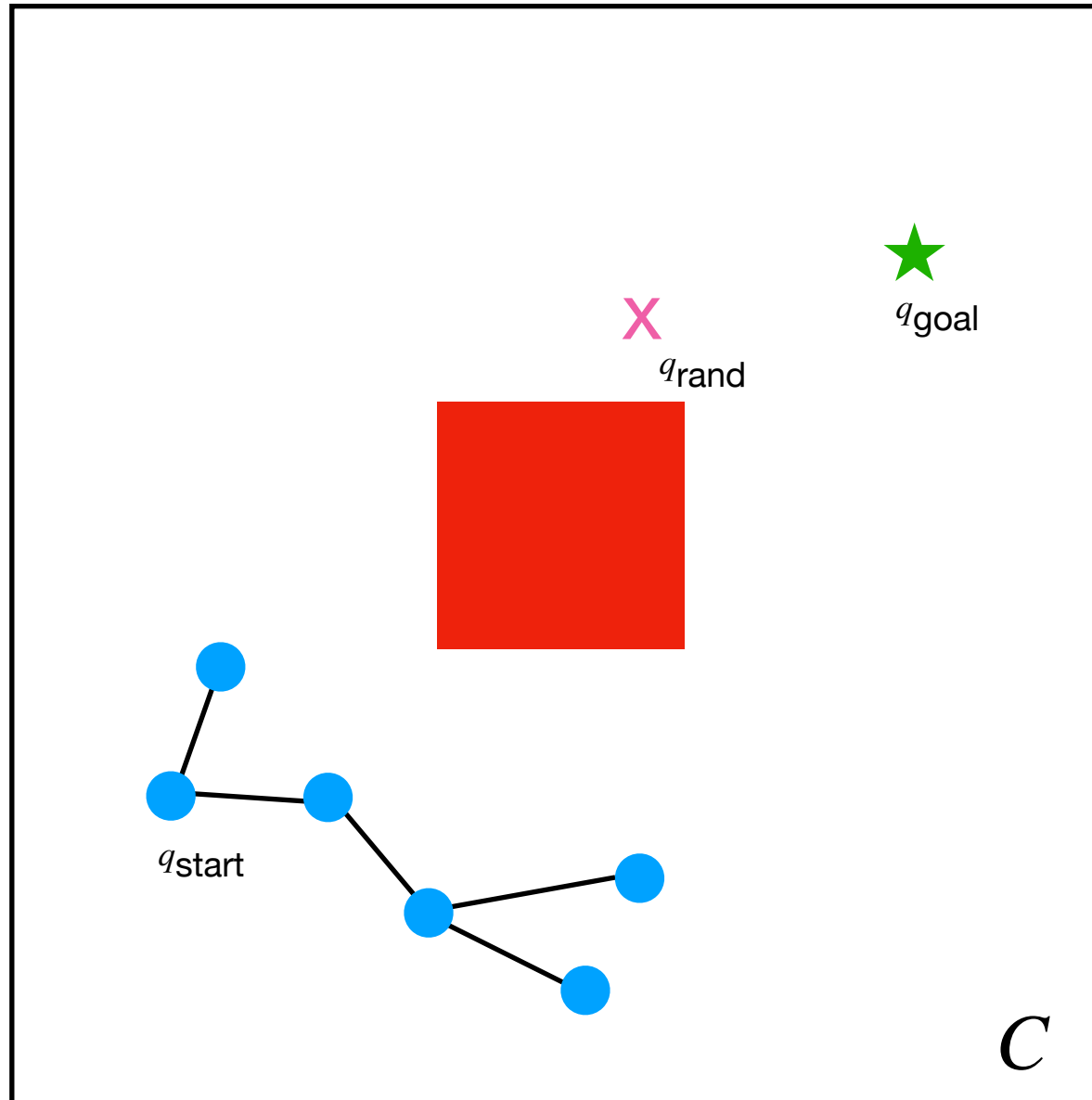
$T.addEdge(x_{near}, x_{new}, u_{rand}, dt)$

if $x_{new} \in X_{goal}$:

return $\text{Trajectory}(T, x_{start}, x_{new})$

return Failure

Rapidly Exploring Random Trees* (RRT*)



KinodynamicRRTStar(x_{start} , X_{goal} , N , r):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

if $\text{CollisionFreeTrajectory}(x_{near}, x_{rand})$:

$X_{near} = \text{NearVertices}(T, x_{new}, r)$

$x_{parent} = x_{near}$

for x in X_{near} :

if $\text{FeasibleConnection}(x, x_{new})$:

if $\text{Cost}(x) + \text{EdgeCost}(x, x_{new})$
 $< \text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x_{new})$:

$x_{parent} = x$

$T.addVertex(x_{new})$

$T.addEdge(x_{parent}, x_{new})$

for x in X_{near} :

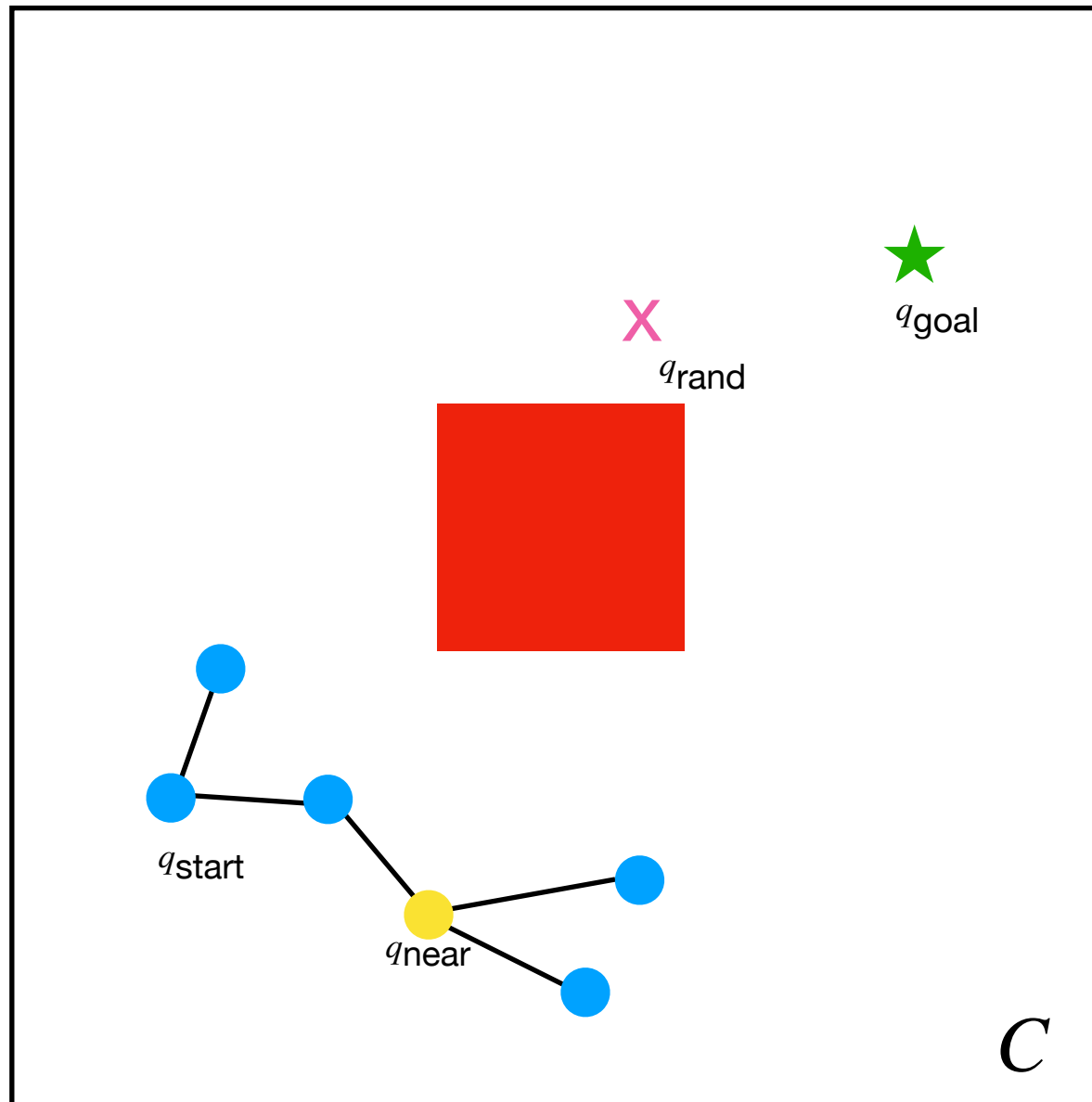
if $\text{FeasibleConnection}(x_{new}, x)$:

if $\text{Cost}(x_{new}) + \text{EdgeCost}(x_{new}, x) < \text{Cost}(x)$:

$T.rewire(x, x_{new})$

return $\text{BestTrajectory}(T, X_{goal})$

Rapidly Exploring Random Trees* (RRT*)



KinodynamicRRTStar(x_{start} , X_{goal} , N , r):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

if $\text{CollisionFreeTrajectory}(x_{near}, x_{rand})$:

$X_{near} = \text{NearVertices}(T, x_{rand}, r)$

$x_{parent} = x_{near}$

for x in X_{near} :

if $\text{FeasibleConnection}(x, x_{parent})$:

if $\text{Cost}(x) + \text{EdgeCost}(x, x_{parent})$
 $< \text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x)$:

$x_{parent} = x$

$T.addVertex(x_{parent})$

$T.addEdge(x_{parent}, x)$

for x in X_{near} :

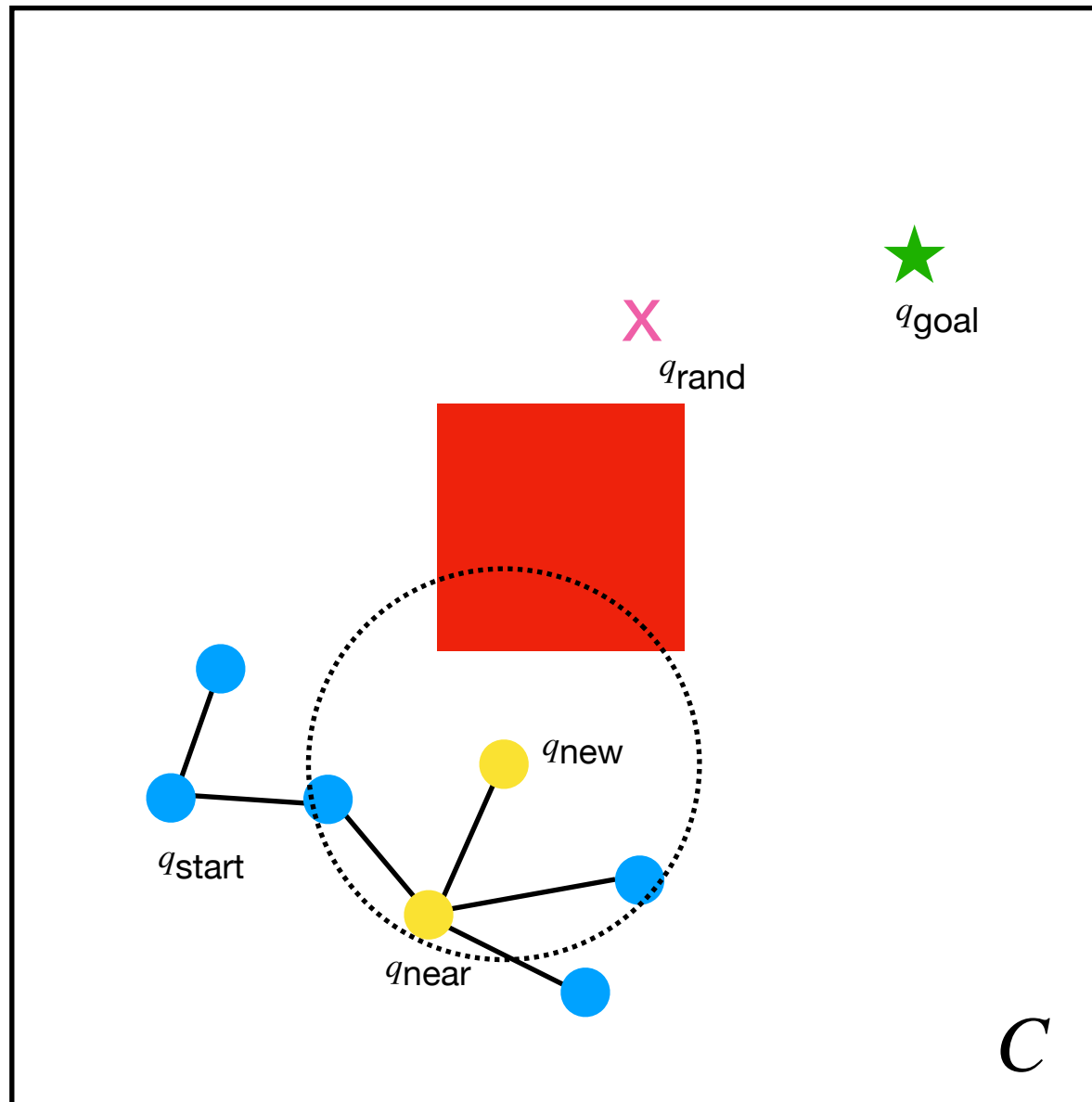
if $\text{FeasibleConnection}(x_{parent}, x)$:

if $\text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x) < \text{Cost}(x)$:

$T.rewire(x, x_{parent})$

return $\text{BestTrajectory}(T, X_{goal})$

Rapidly Exploring Random Trees* (RRT*)



KinodynamicRRTStar(x_{start} , X_{goal} , N , r):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

if $\text{CollisionFreeTrajectory}(x_{near}, x_{rand})$:

$X_{near} = \text{NearVertices}(T, x_{new}, r)$

$x_{parent} = x_{near}$

for x in X_{near} :

if $\text{FeasibleConnection}(x, x_{new})$:

if $\text{Cost}(x) + \text{EdgeCost}(x, x_{new})$
 $< \text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x_{new})$:

$x_{parent} = x$

$T.addVertex(x_{new})$

$T.addEdge(x_{parent}, x_{new})$

for x in X_{near} :

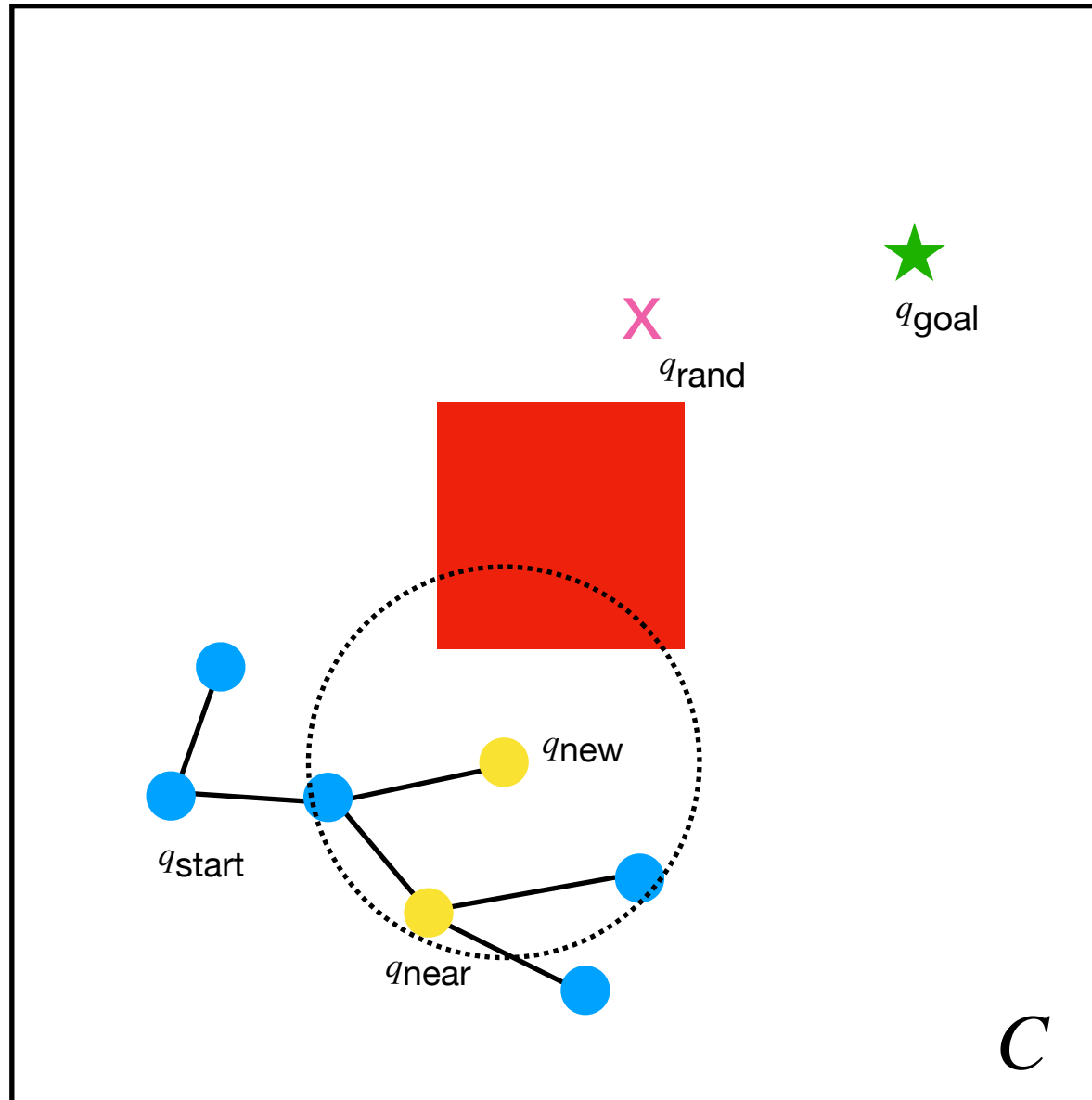
if $\text{FeasibleConnection}(x_{new}, x)$:

if $\text{Cost}(x_{new}) + \text{EdgeCost}(x_{new}, x) < \text{Cost}(x)$:

$T.rewire(x, x_{new})$

return $\text{BestTrajectory}(T, X_{goal})$

Rapidly Exploring Random Trees* (RRT*)



KinodynamicRRTStar(x_{start} , X_{goal} , N , r):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

if $\text{CollisionFreeTrajectory}(x_{near}, x_{rand})$:

$X_{near} = \text{NearVertices}(T, x_{new}, r)$

$x_{parent} = x_{near}$

for x in X_{near} :

if $\text{FeasibleConnection}(x, x_{new})$:

if $\text{Cost}(x) + \text{EdgeCost}(x, x_{new})$
 $< \text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x_{new})$:

$x_{parent} = x$

$T.addVertex(x_{new})$

$T.addEdge(x_{parent}, x_{new})$

for x in X_{near} :

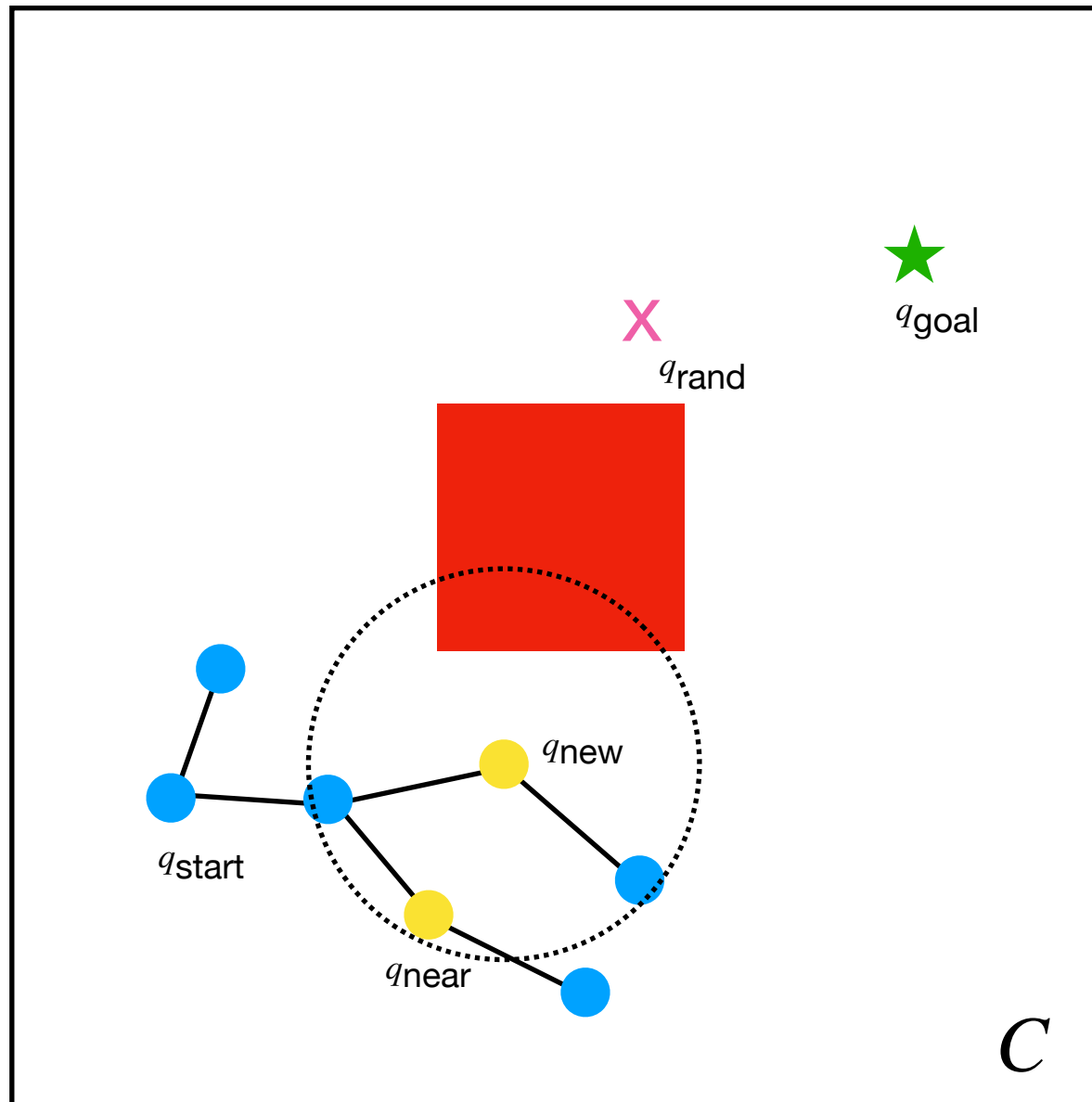
if $\text{FeasibleConnection}(x_{new}, x)$:

if $\text{Cost}(x_{new}) + \text{EdgeCost}(x_{new}, x) < \text{Cost}(x)$:

$T.rewire(x, x_{new})$

return $\text{BestTrajectory}(T, X_{goal})$

Rapidly Exploring Random Trees* (RRT*)



KinodynamicRRTStar(x_{start} , X_{goal} , N , r):

$T.init(x_{start})$

for $i = 1$ to N :

$x_{rand} = \text{SampleRandom}(X_{free})$

$x_{near} = \text{NearestVertex}(T, x_{rand})$

if $\text{CollisionFreeTrajectory}(x_{near}, x_{rand})$:

$X_{near} = \text{NearVertices}(T, x_{new}, r)$

$x_{parent} = x_{near}$

for x in X_{near} :

if $\text{FeasibleConnection}(x, x_{new})$:

if $\text{Cost}(x) + \text{EdgeCost}(x, x_{new})$
 $< \text{Cost}(x_{parent}) + \text{EdgeCost}(x_{parent}, x_{new})$:

$x_{parent} = x$

$T.addVertex(x_{new})$

$T.addEdge(x_{parent}, x_{new})$

for x in X_{near} :

if $\text{FeasibleConnection}(x_{new}, x)$:

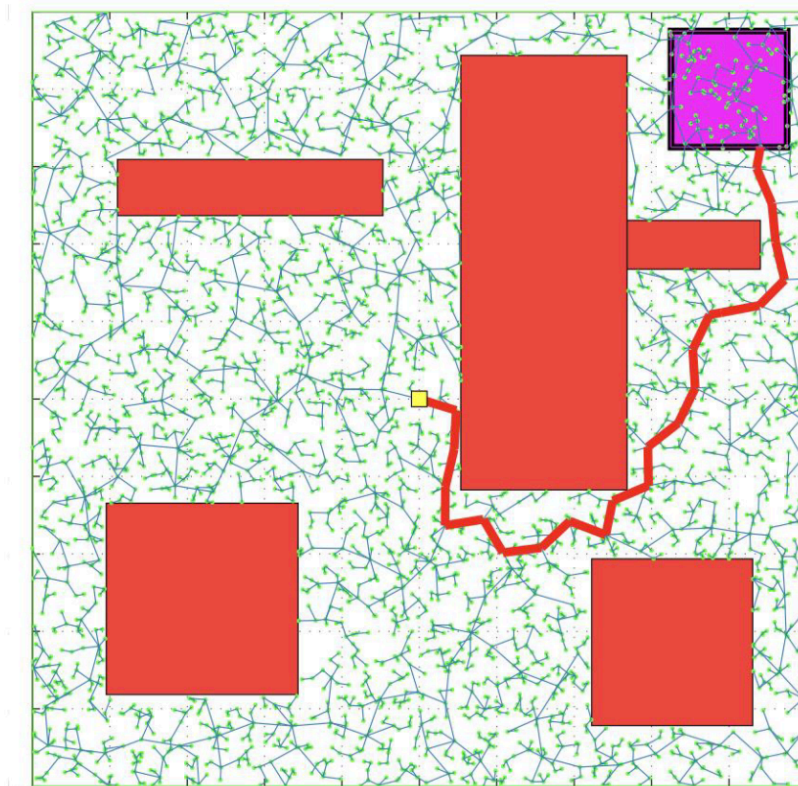
if $\text{Cost}(x_{new}) + \text{EdgeCost}(x_{new}, x) < \text{Cost}(x)$:

$T.rewire(x, x_{new})$

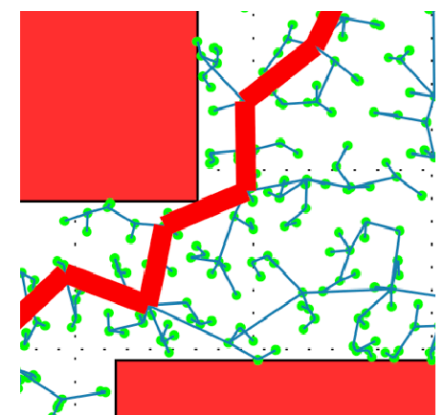
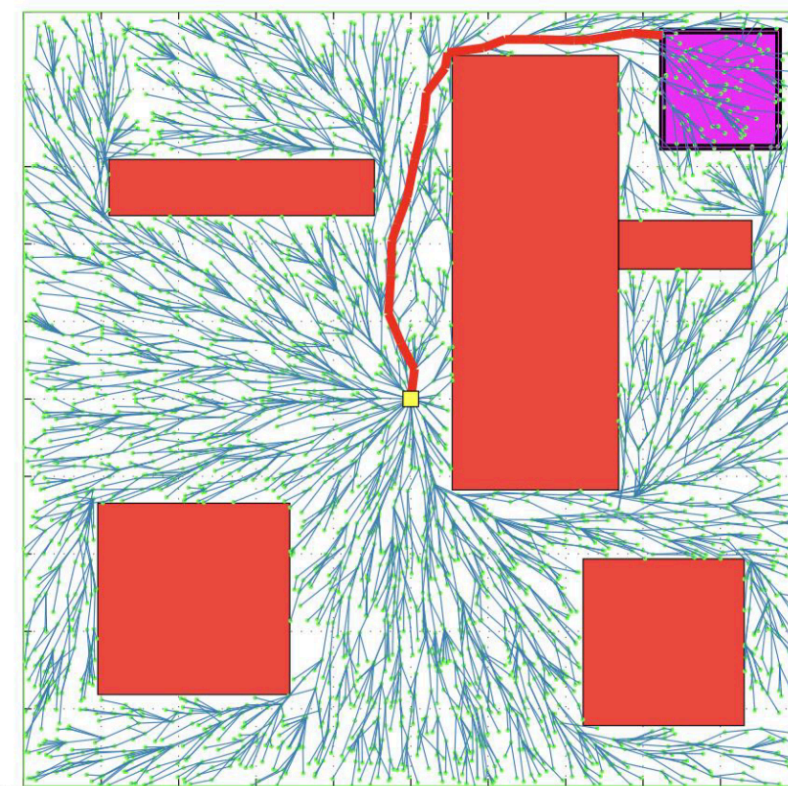
return $\text{BestTrajectory}(T, X_{goal})$

Rapidly Exploring Random Trees* (RRT*)

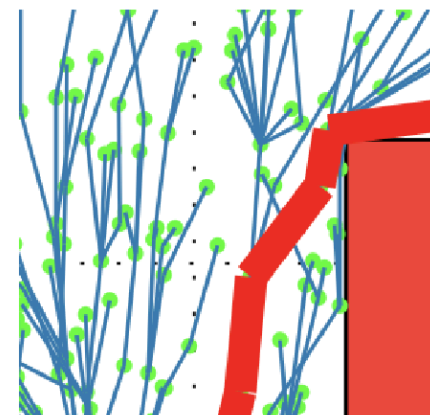
RRT after 5000 nodes



RRT* after 5000 nodes



Zoomed in view



Zoomed in view

Summary

Discrete Grid Planners

- Dijkstra's Algorithm: Finds the shortest path in a weighted graph by expanding vertices in order of their accumulated cost. Guarantees an optimal path for non-negative edge weights.
- A* Algorithm: Uses a heuristic to guide the search towards the goal, often reducing unnecessary exploration. Guarantees an optimal path with an admissible heuristic and appropriate search implementation.

Continuous Sampling Planners

- RRT: Incrementally constructs a tree by randomly sampling configurations in continuous C-space. Efficiently explores high-dimensional spaces but does not guarantee an optimal path.
- RRT*: Extends RRT by selecting better parents and rewiring nearby vertices to reduce path cost. Under suitable assumptions, the solution converges towards the optimum as the number of samples increases.