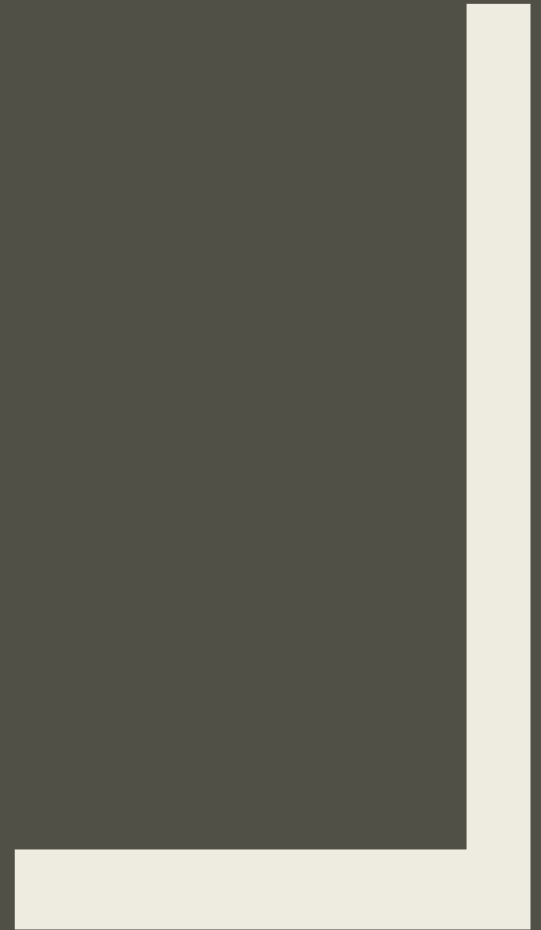


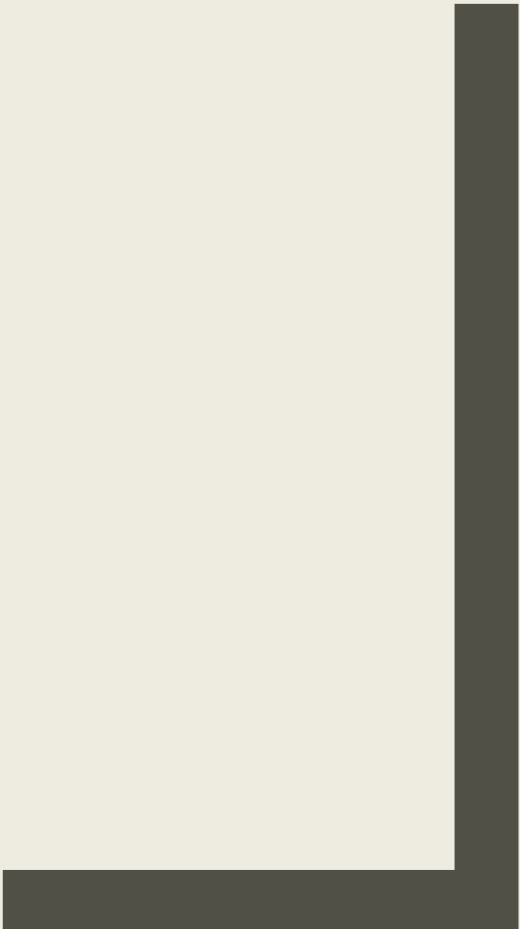
GRAPHS

Mahsa Ziraksima
ACIBADEM UNIVERSITY
Fall 2025





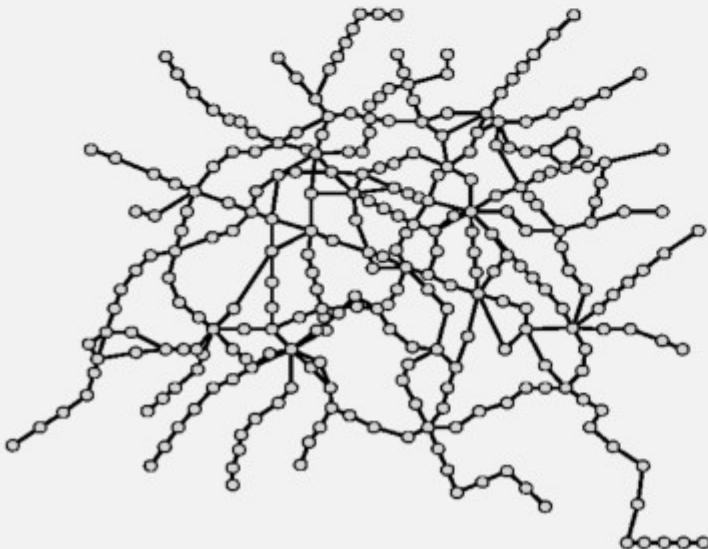
SUMMARY

- Performance
 - Dijkstra's Algorithm
- 

GRAPH

■ Why study graph algorithms?

- *Thousands of practical applications.*
- *Hundreds of graph algorithms known.*
- *Interesting and broadly useful abstraction.*



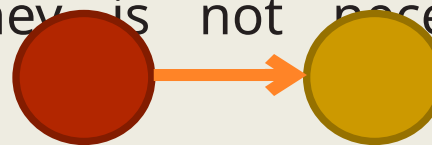
GRAPH

- In real world, there are sets of objects in which some pairs of the objects are in some sense "related". If we represent objects by nodes and connect the related ones with an edge:

- if the vertices represent people at a party, and there is an edge between two people if they shake hands, then this graph is undirected because any person A can shake hands with a person B only if B also shakes hands with A.

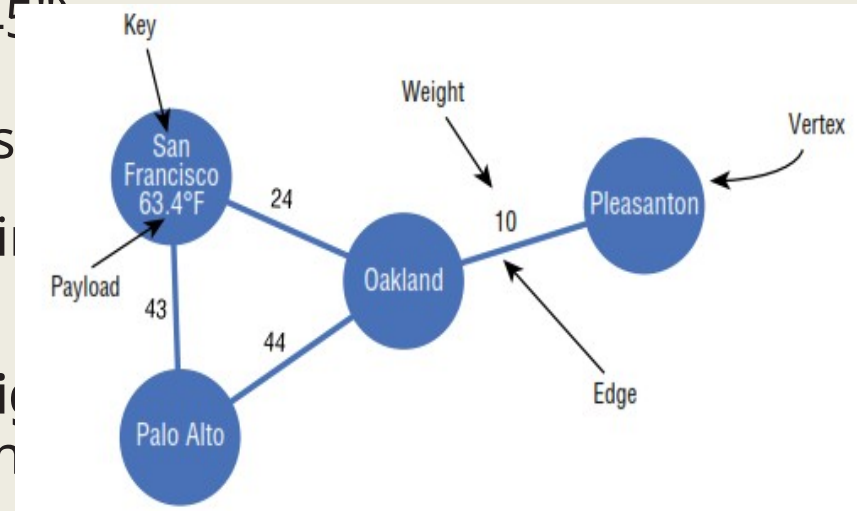


- in contrast, if any edge from a person A to a person B corresponds to A lends money to B, then this graph is directed, because lending money is not necessarily reciprocated.



GRAPH

- It is an abstract data type in which a piece of data connects to one or more other pieces of data.
- Each piece of data in a graph is called a **vertex** or a **node**.
- A vertex has a name called a **key**.
- A vertex can have additional data called its **payload**.
- If a vertex represents a person in a social network, the key might be their user ID (e.g., "12345") while the payload could include their name, age, and email address.
- The connection between vertices in a graph is called an **edge**.
- A graph's edges can contain a **weight**, which is the cost to travel between

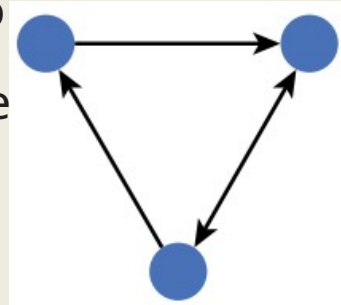


GRAPH

- There are several types of graphs:

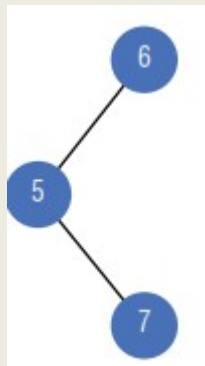
- Directed or ordered graphs***

- each edge has a direction associated with it, and you can move between two vertices only in that direction
 - edges can be one-way connected or two-way connected



- Undirected or unordered graphs***

- edges are bidirectional, which means you can travel in either direction between two connected vertices.

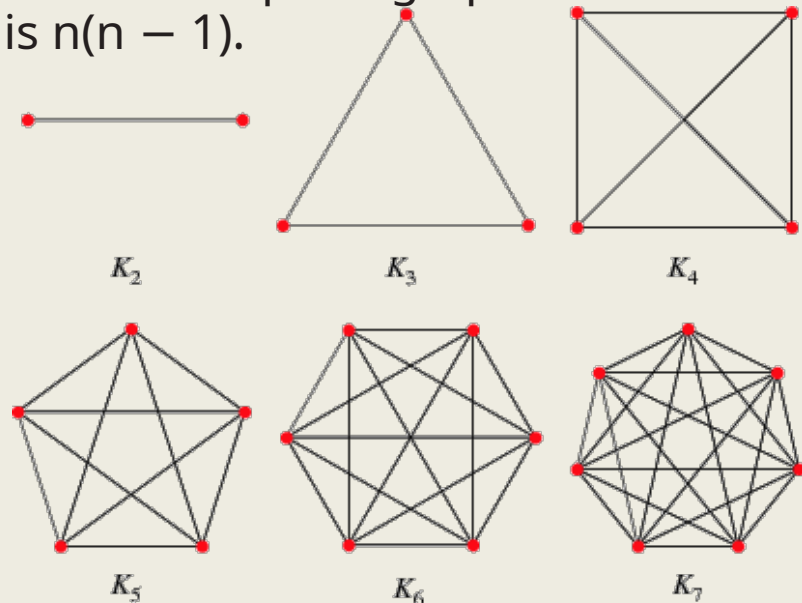


GRAPH

■ There are several types of graphs:

–**Complete** graphs

- every vertex is connected to every other vertex
- Total number of edges in undirected complete graph is $n(n - 1)/2$ and in directed one is $n(n - 1)$.

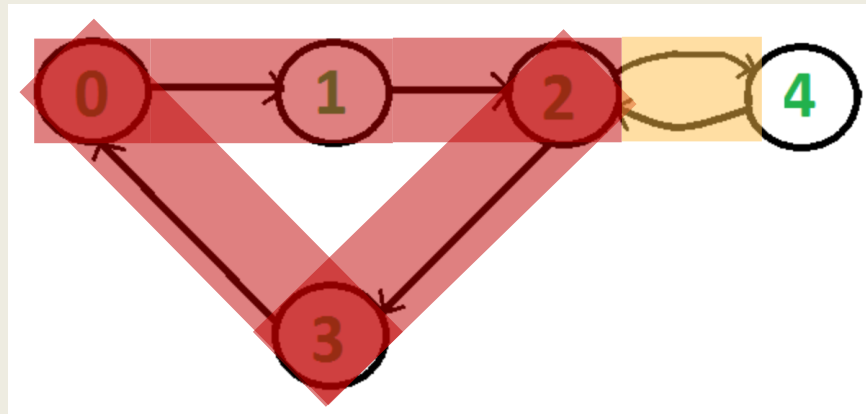


–**Incomplete** graph

- some but not all vertices are connected.

GRAPH

- A graph **path** is a sequence of vertices connected by edges.



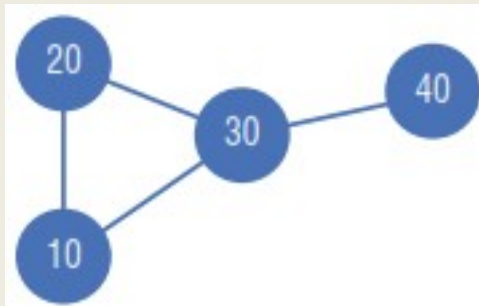
- Two vertices are **connected** if there is a path between them.
- A **cycle** is a path in a graph starting and ending at the same vertex.
- An **acyclic** graph is a graph that does not contain a cycle.

GRAPH

- A tree is a restricted form of a graph.

Trees have direction (the parent- child relationship) and do not contain cycles, which makes them directed acyclic graphs with a restriction: a child can have only one parent.

Graph



Edge list

```
[10, 20]  
[10, 30]  
[20, 10]  
[20, 30]  
[30, 10]  
[30, 20]  
[30, 40]  
[40, 30]
```

Adjacency matrix

	10	20	30	40
10	0	1	1	0
20	1	0	1	0
30	1	1	0	1
40	0	0	1	0

adjacency list

```
10: [20, 30],  
20: [10, 30],  
30: [10, 20, 40],  
40: [30]
```

```
class Graph:
    def __init__(self):
        self.vertex_dict = {}

    def add_vertex(self, key):
        new_vertex = Vertex(key)
        self.vertex_dict[key] = new_vertex

    def get_vertex(self, key):
        if key in self.vertex_dict:
            return self.vertex_dict[key]
        return None

    def add_edge(self, f, t, weight=0):
        if f not in self.vertex_dict:
            self.add_vertex(f)
        if t not in self.vertex_dict:
            self.add_vertex(t)
        self.vertex_dict[f].add_adj(self.vertex_dict[t], weight)
```

```
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B", 1)
graph.add_edge("B", "C", 10)
vertex_a = graph.get_vertex("A")
vertex_b = graph.get_vertex("B")
print(graph.get_vertex("4"))
```

None

GRAPH

- In practice, generally adjacency-lists representation is preferred.

–Algorithms based on iterating over vertices adjacent to v .

–Real-world graphs tend to be sparse.

representation	space	add edge	edge between v and w ?	iterate over vertices adjacent to v ?
list of edges	E	1	E	E
adjacency matrix	V^2	1	1	V
adjacency lists	$E + V$	1	$degree(v)$	$degree(v)$

GRAPH

■ Some graph-processing problems

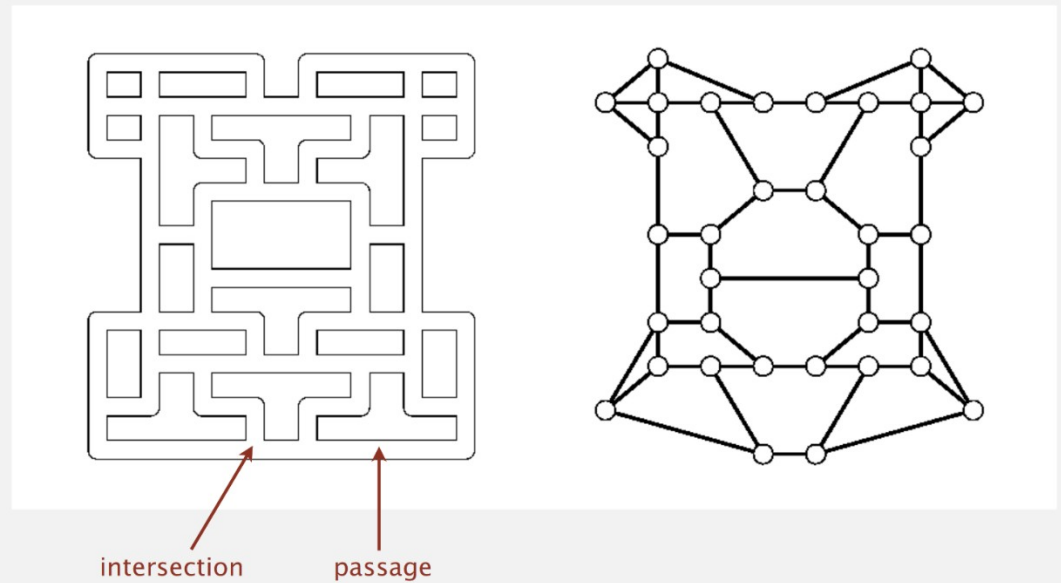
problem	description
s-t path	<i>Is there a path between s and t ?</i>
shortest s-t path	<i>What is the shortest path between s and t ?</i>
cycle	<i>Is there a cycle in the graph ?</i>
Euler cycle	<i>Is there a cycle that uses each edge exactly once ?</i>
Hamilton cycle	<i>Is there a cycle that uses each vertex exactly once ?</i>
connectivity	<i>Is there a way to connect all of the vertices ?</i>
biconnectivity	<i>Is there a vertex whose removal disconnects the graph ?</i>
planarity	<i>Can the graph be drawn in the plane with no crossing edges ?</i>
graph isomorphism	<i>Do two adjacency lists represent the same graph ?</i>

GRAPH

■ Maze exploration

Maze graph.

- Vertex = intersection.
- Edge = passage.



Goal. Explore every intersection in the maze.

GRAPH

■ Depth-first search:

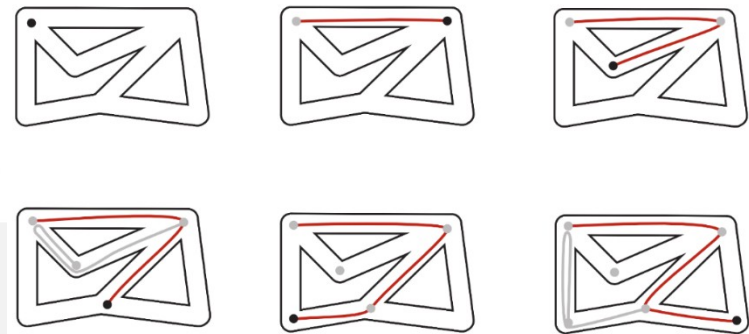
Goal. Systematically traverse a graph.

Idea. Mimic maze exploration.  function-call stack acts as ball of string

DFS (to visit a vertex v)

Mark v as visited.

Recursively visit all unmarked
vertices w adjacent to v .



Typical applications.

- Find all vertices connected to a given source vertex.
- Find a path between two vertices.

GRAPH

■ Depth-first search:

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



<http://algs4.cs.princeton.edu>

4.1 DEPTH-FIRST SEARCH DEMO

GRAPH

■ Depth-first search:

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices adjacent to v .



<http://algs4.cs.princeton.edu>

4.2 DIRECTED DFS DEMO

GRAPH

■ Breadth-first search:

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



<http://algs4.cs.princeton.edu>

4.1 BREADTH-FIRST SEARCH DEMO

GRAPH

■ Breadth-first search:

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent to v and mark them.



<http://algs4.cs.princeton.edu>

4.2 DIRECTED BFS DEMO

GRAPH

- BFS and DFS enables efficient solutions of many (but not all) graph problems.

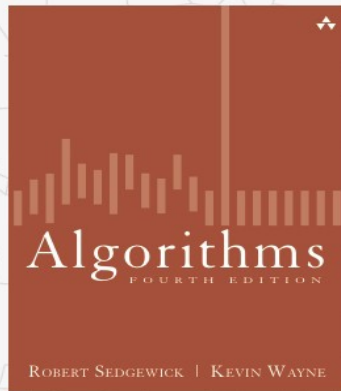
problem	BFS	DFS	time
path between s and t	✓	✓	$E + V$
shortest path between s and t	✓		$E + V$
connected components	✓	✓	$E + V$
biconnected components		✓	$E + V$
cycle	✓	✓	$E + V$
Euler cycle		✓	$E + V$
bipartiteness	✓	✓	$E + V$
planarity		✓	$E + V$

GRAPH

- Challenge: given a set of tasks to be completed with precedence constraints, in which order should we schedule them?

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

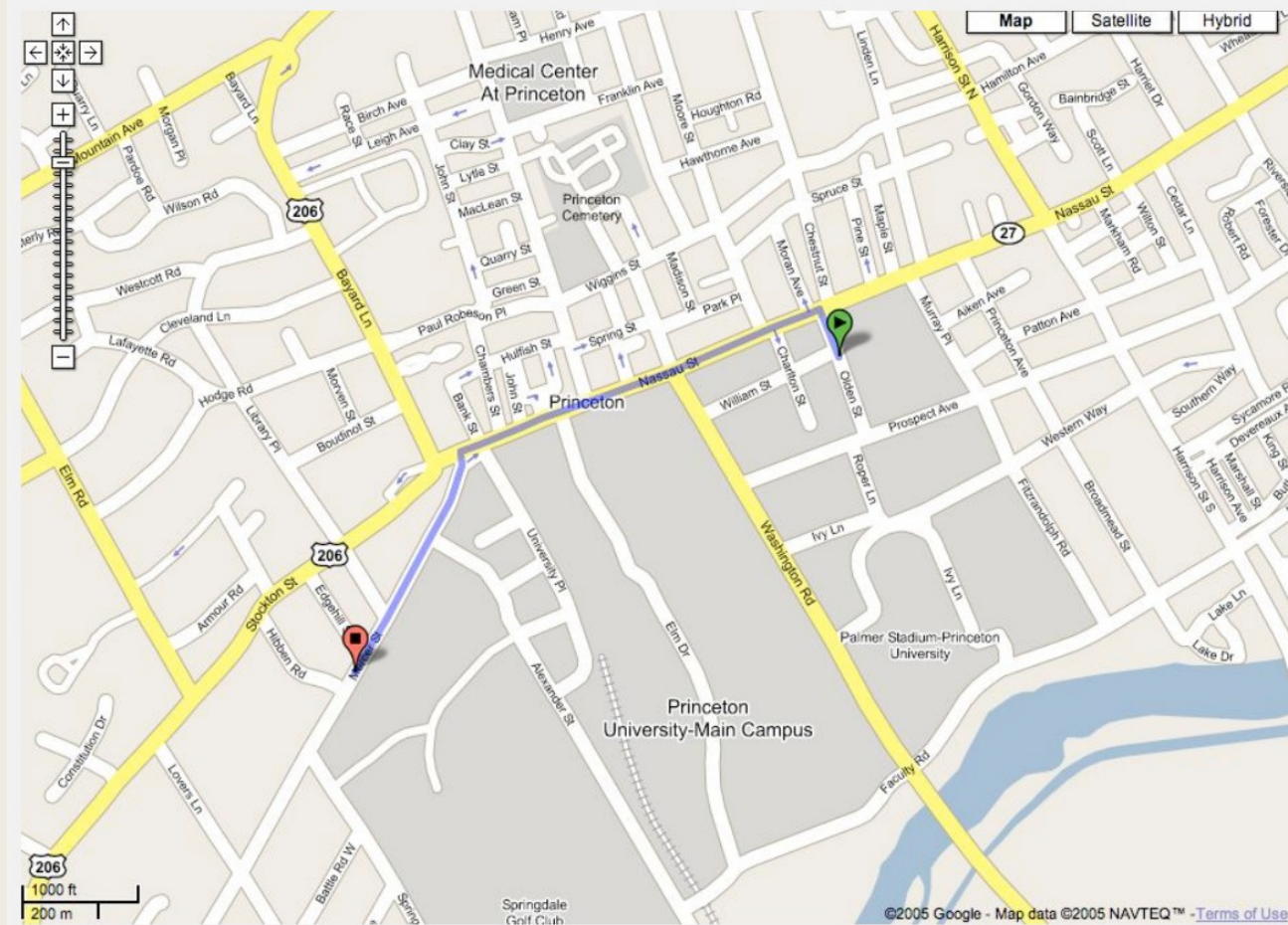


<http://algs4.cs.princeton.edu>

4.2 TOPOLOGICAL SORT DEMO

SHORTEST PATHS

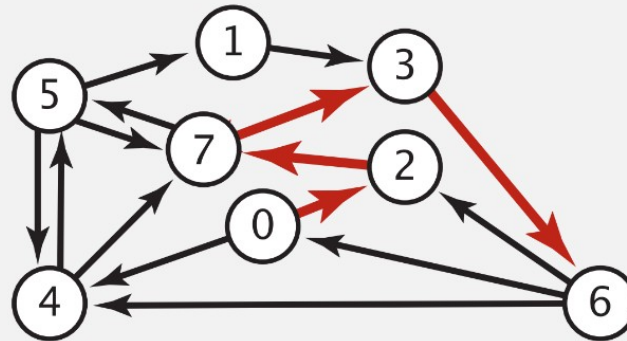
■ Google maps:



SHORTEST PATHS

edge-weighted digraph

4→5	0.35
5→4	0.35
4→7	0.37
5→7	0.28
7→5	0.28
5→1	0.32
0→4	0.38
0→2	0.26
7→3	0.39
1→3	0.29
2→7	0.34
6→2	0.40
3→6	0.52
6→0	0.58
6→4	0.93

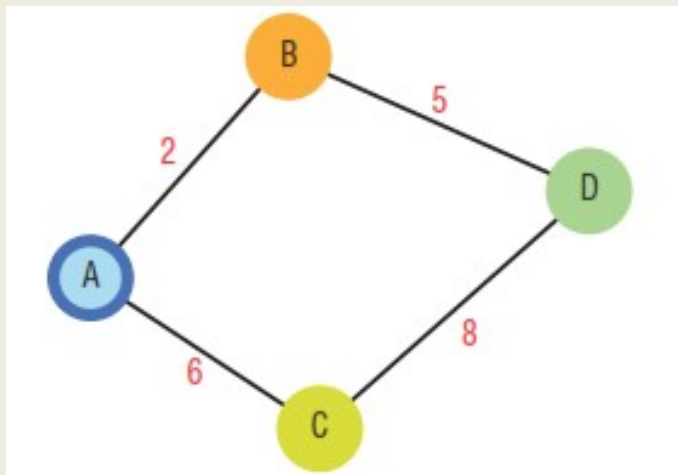


shortest path from 0 to 6

0→2	0.26
2→7	0.34
7→3	0.39
3→6	0.52

DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.

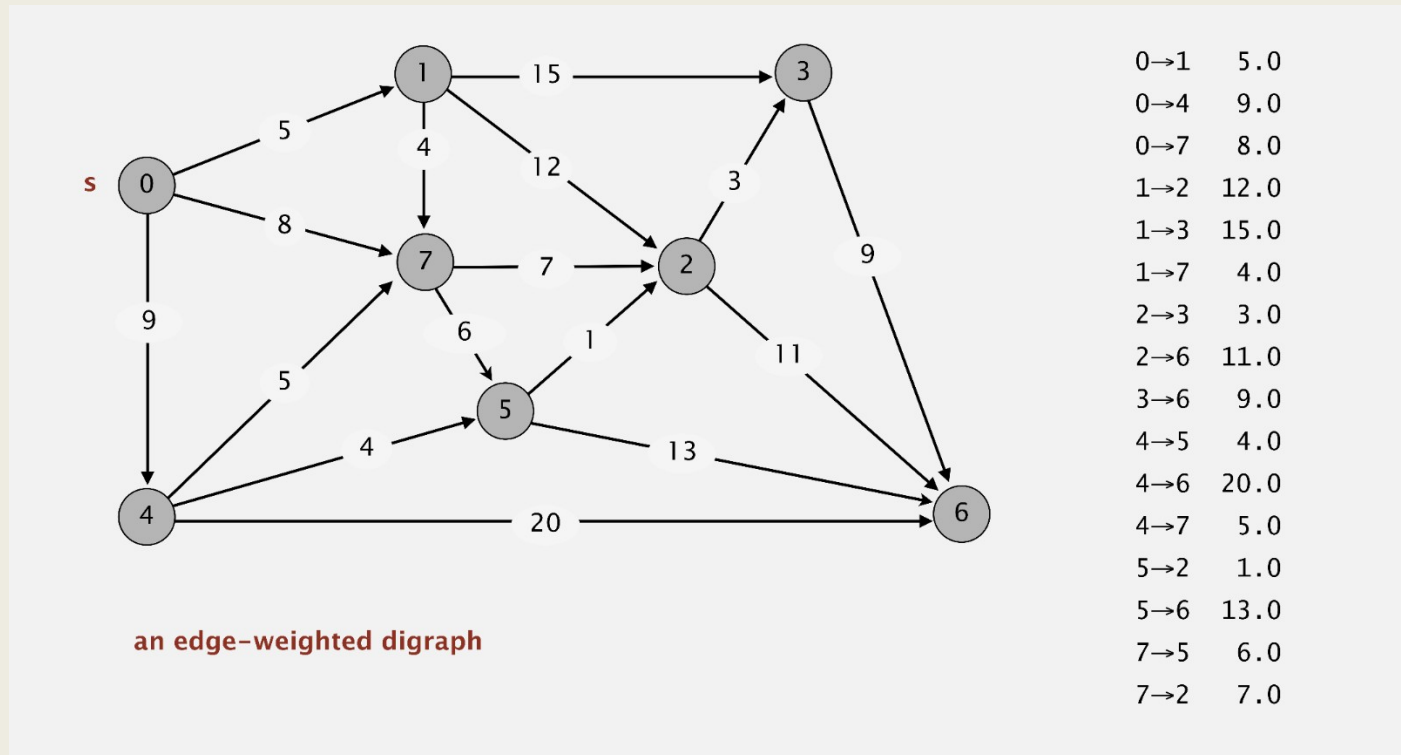


"A"

```
{  
  "A": 0,  
  "B": 2,  
  "C": 6,  
  "D": 7,  
}
```

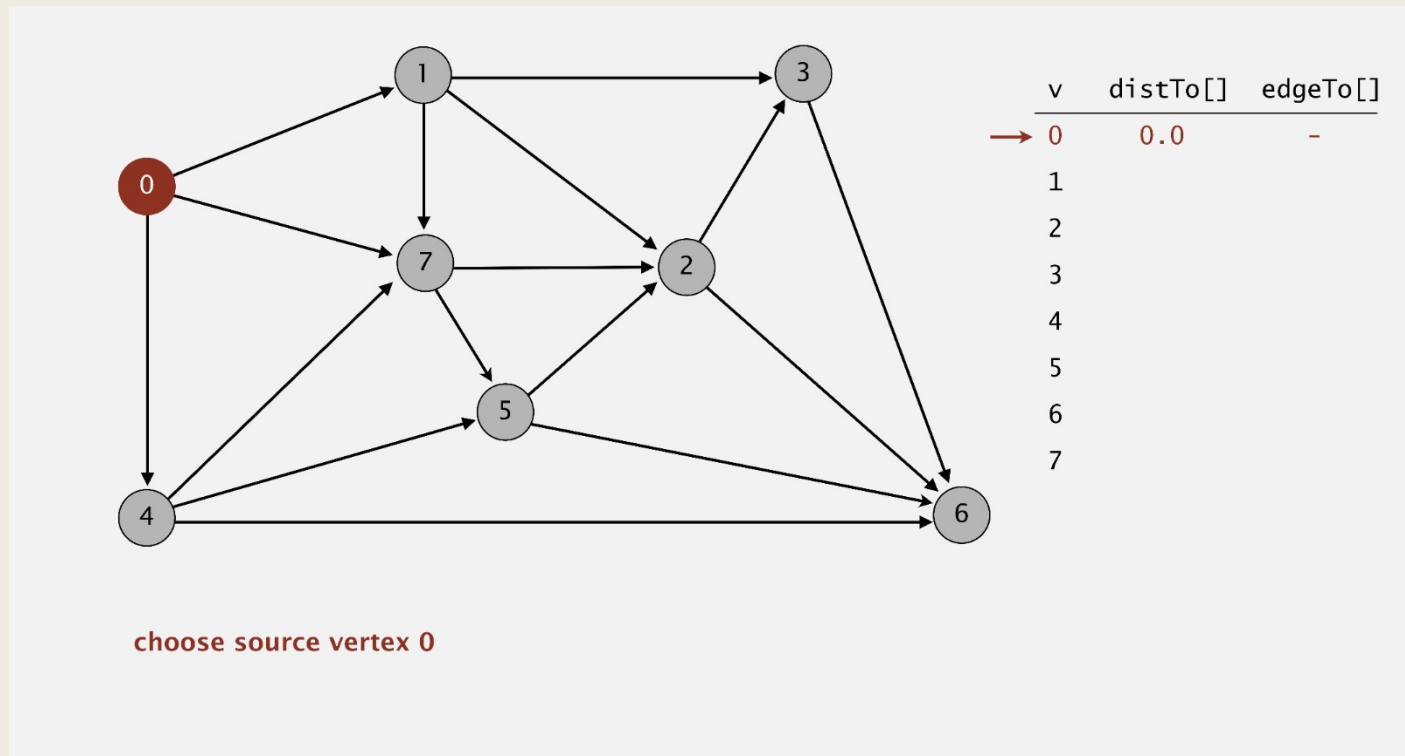
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



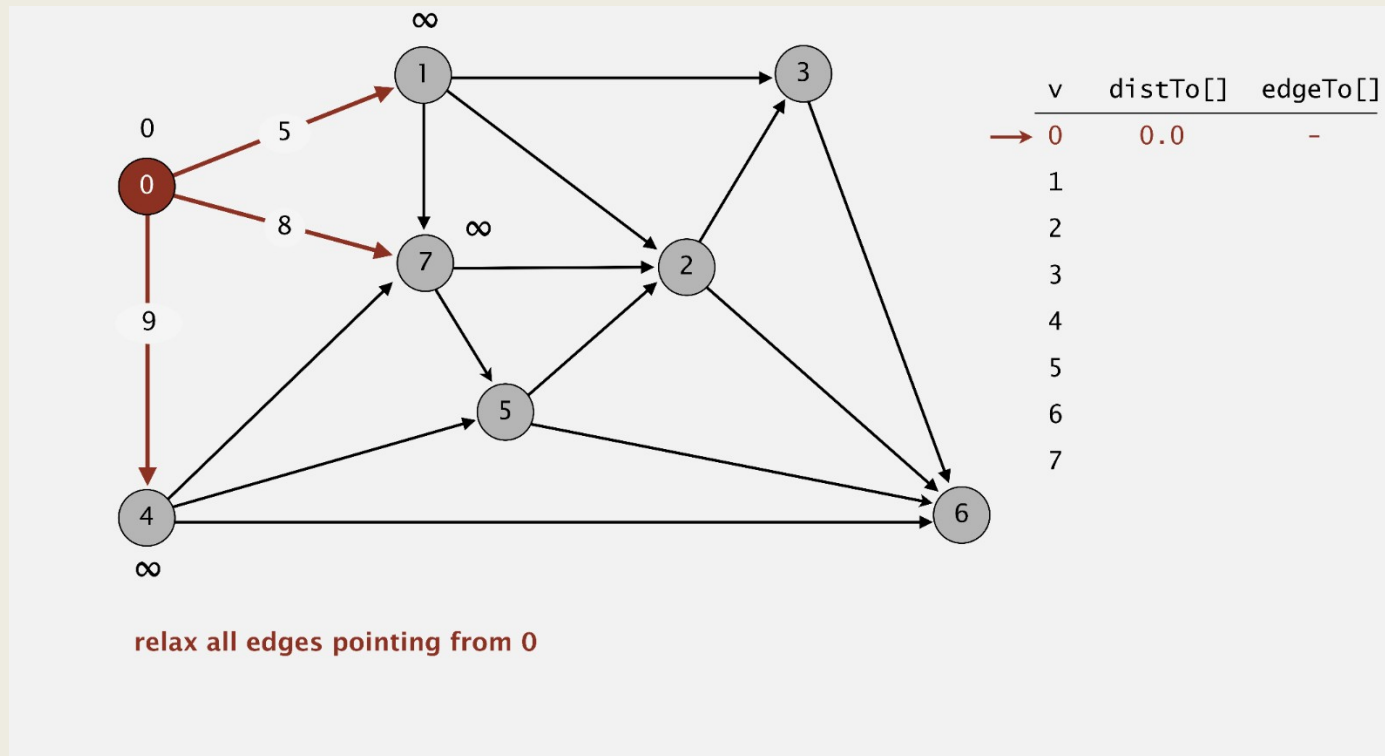
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



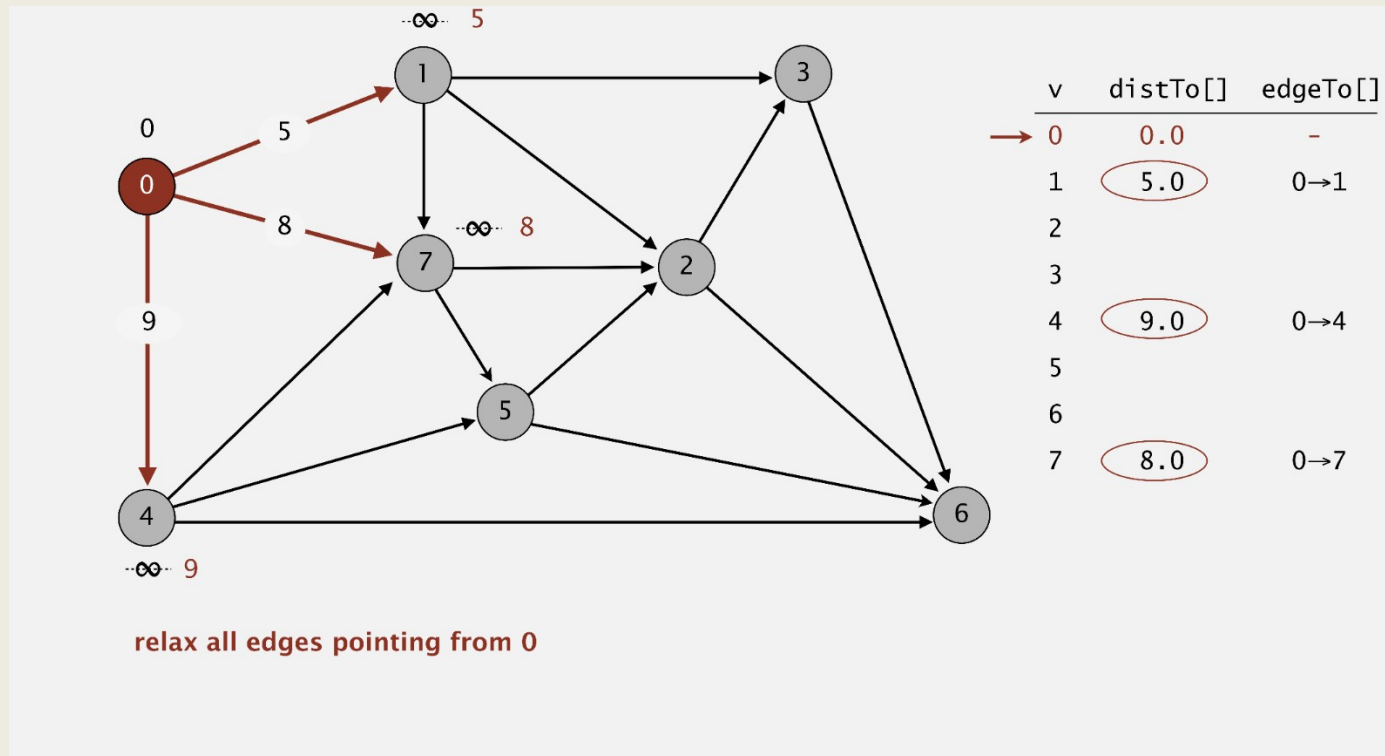
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



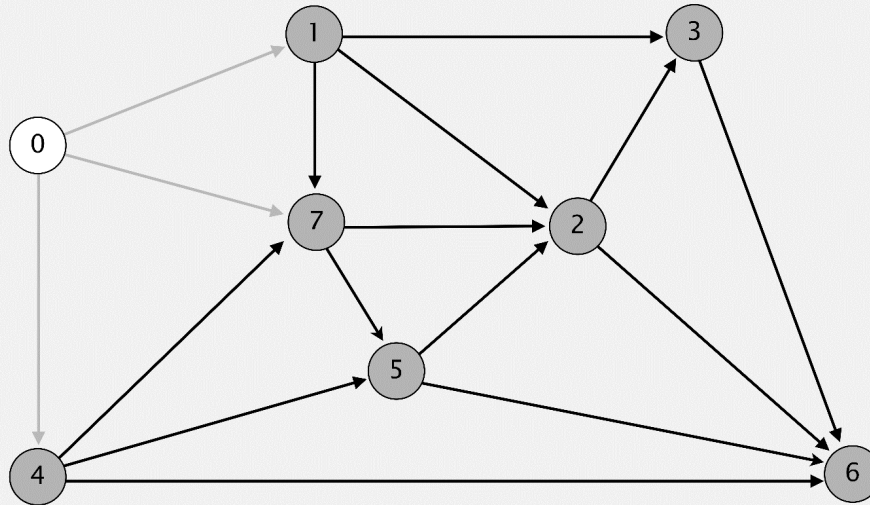
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



DIJKSTRA'S ALGORITHM

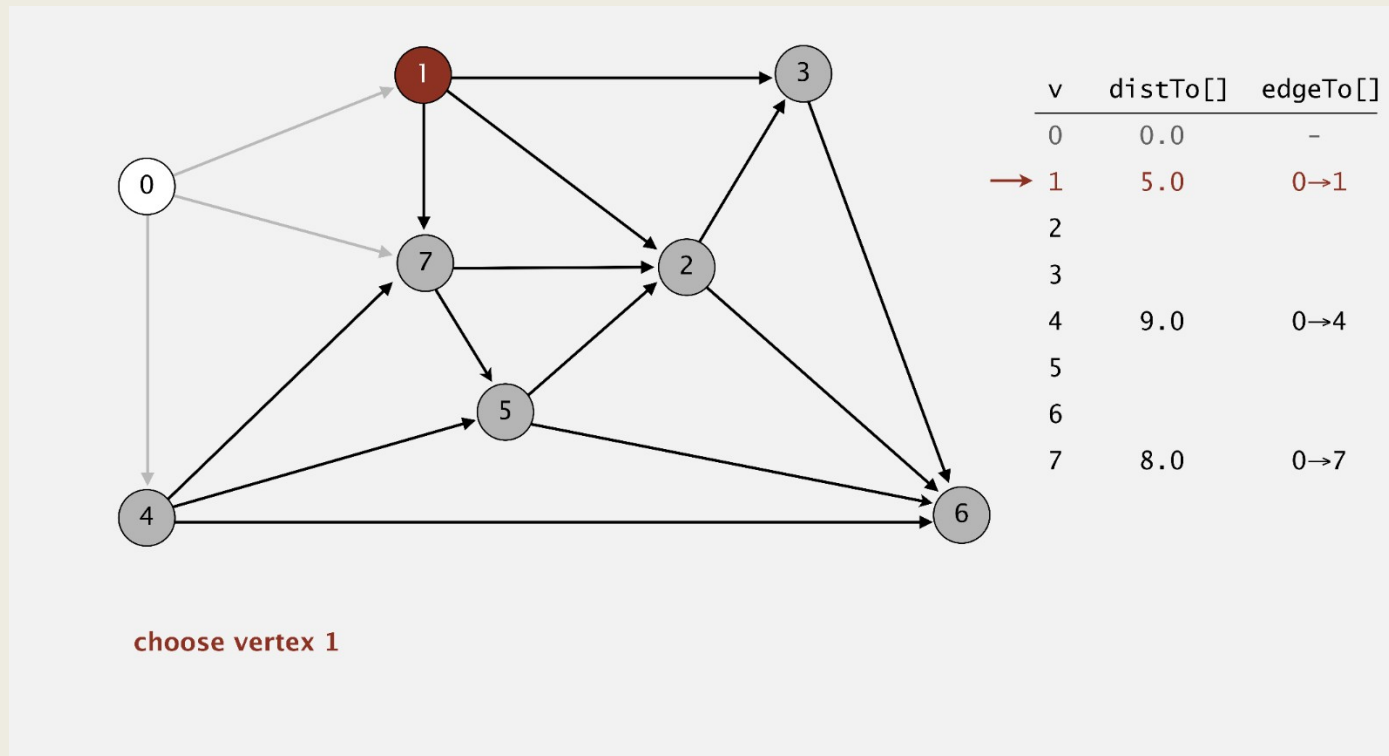
- Challenge: find the shortest path between two vertices.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2		
3		
4	9.0	0→4
5		
6		
7	8.0	0→7

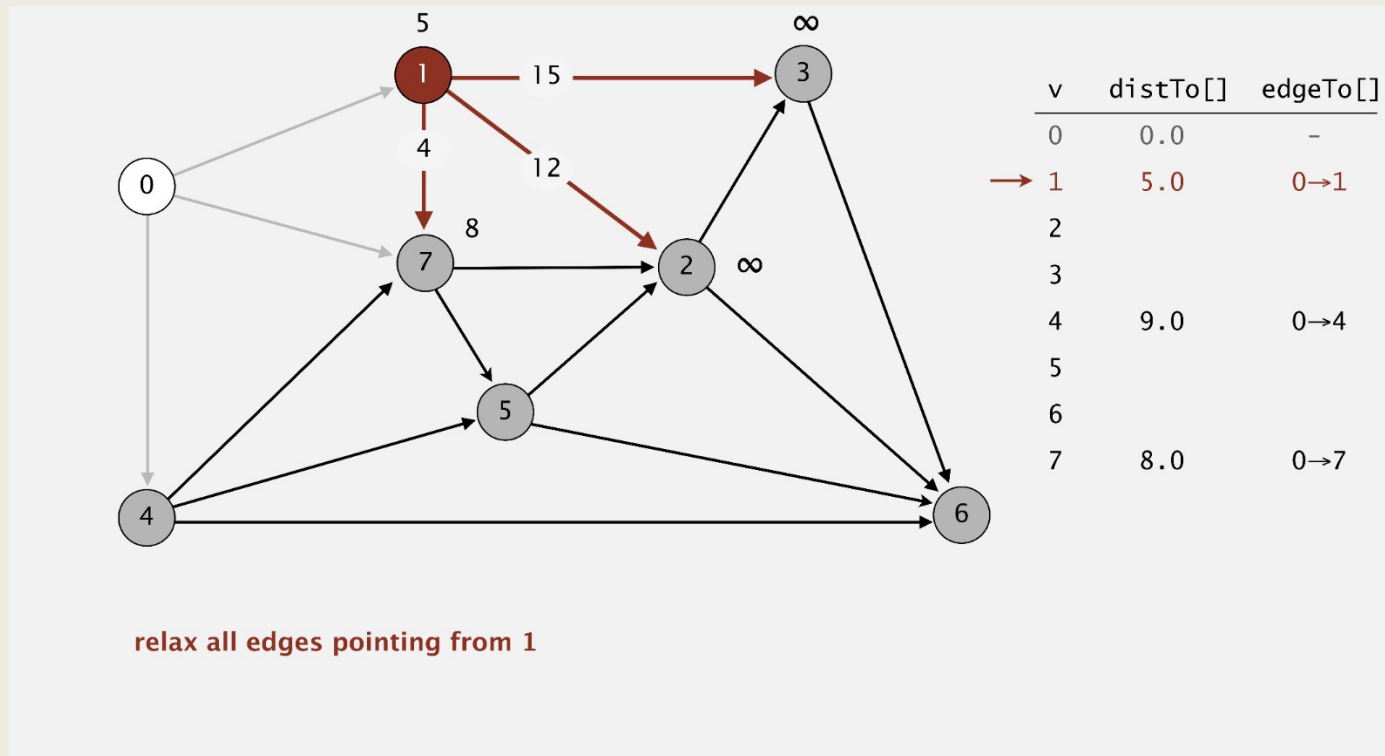
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



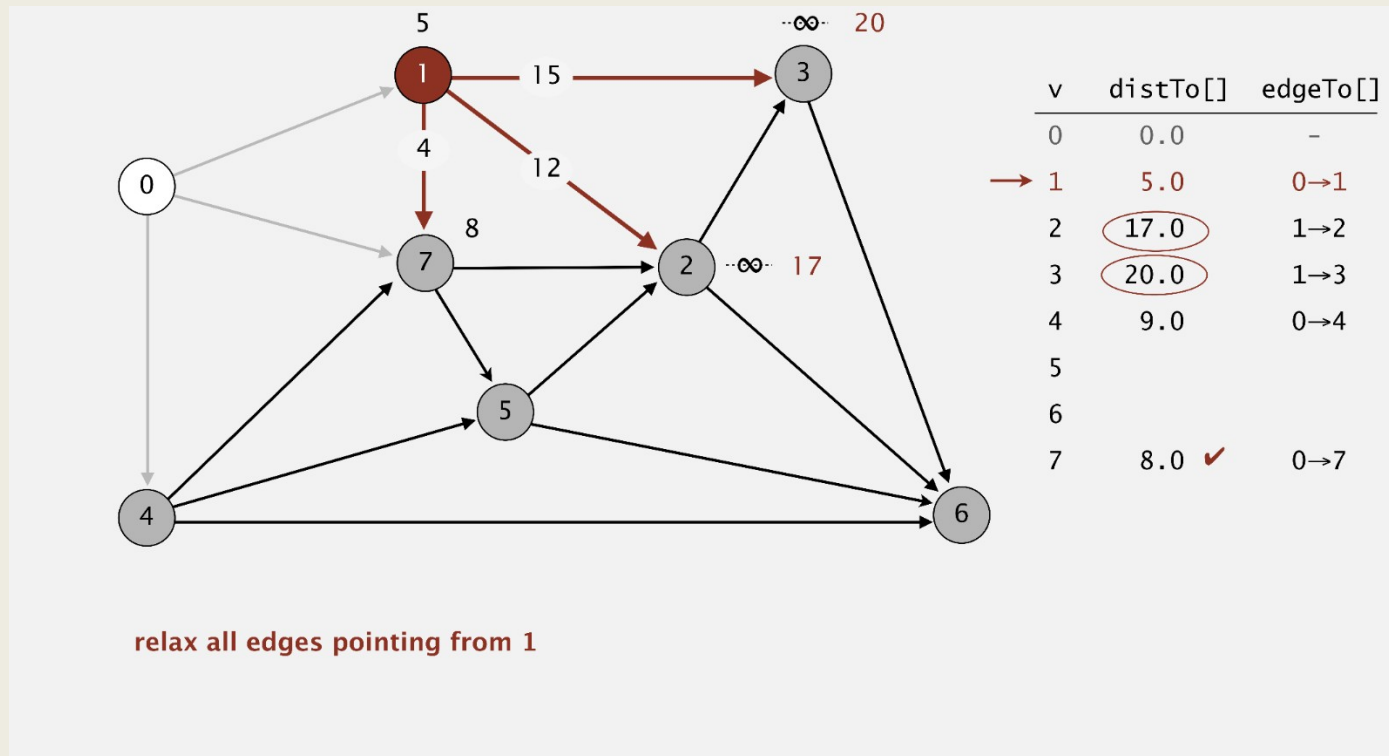
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



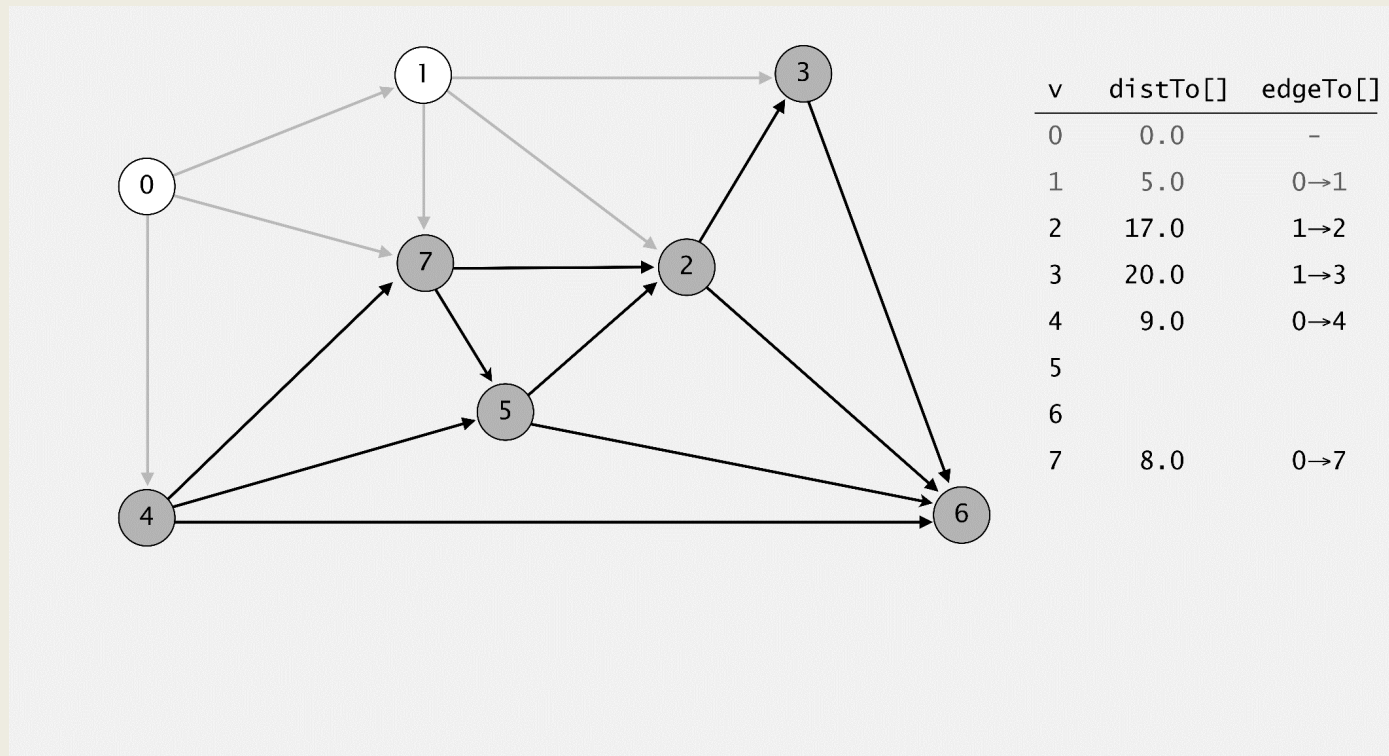
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



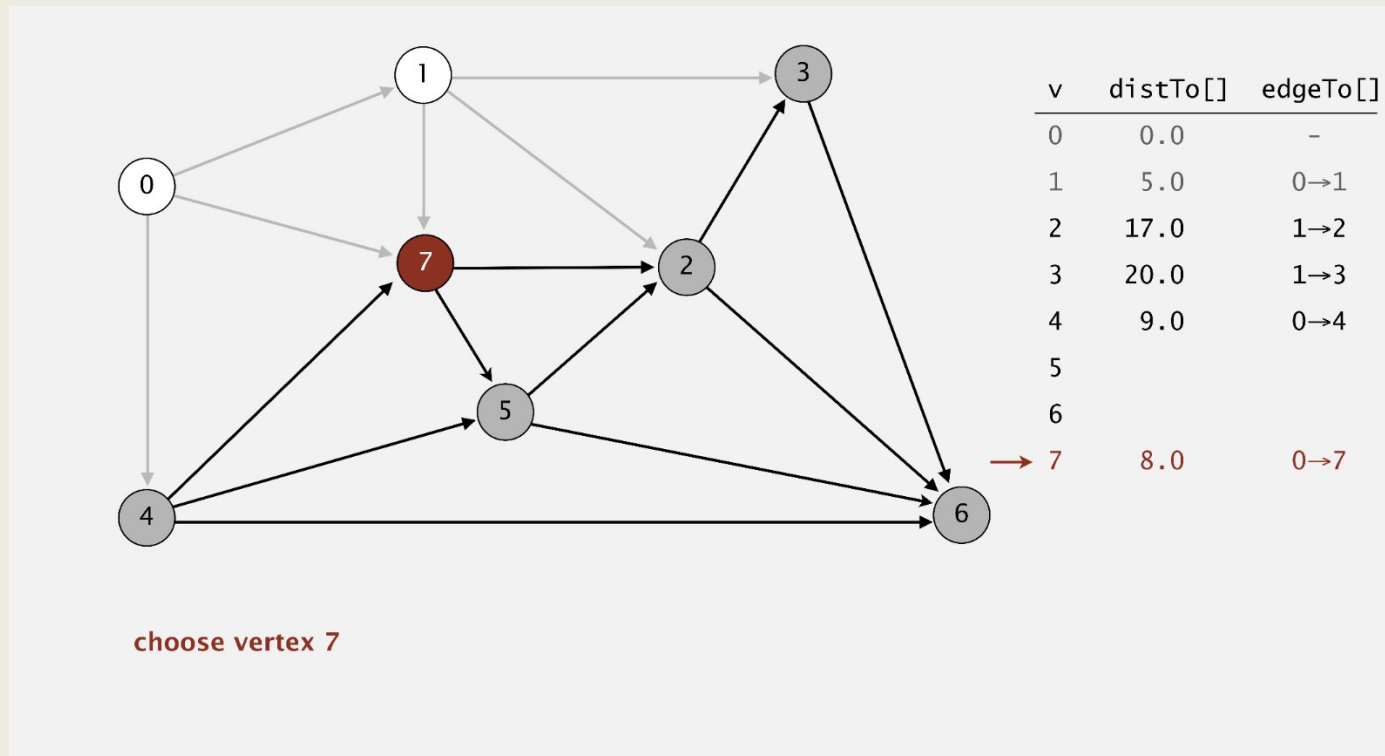
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



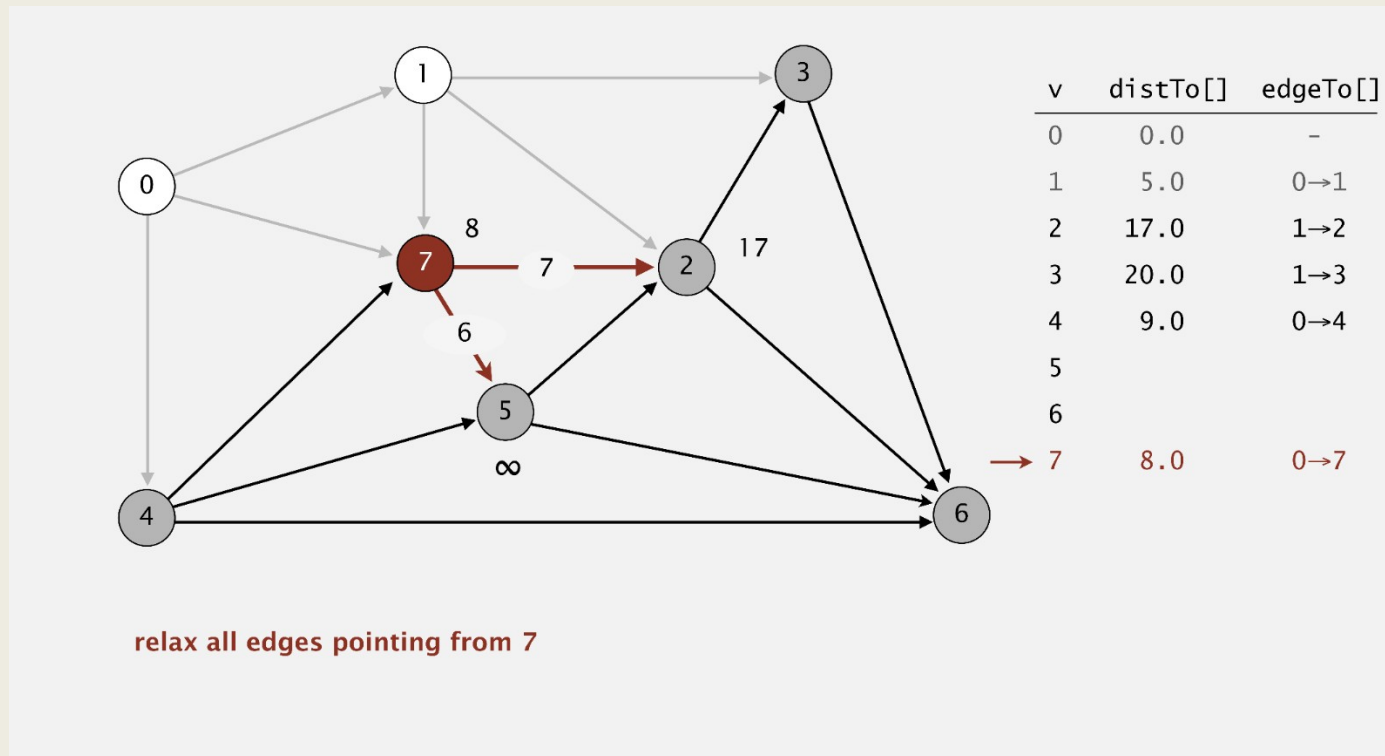
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



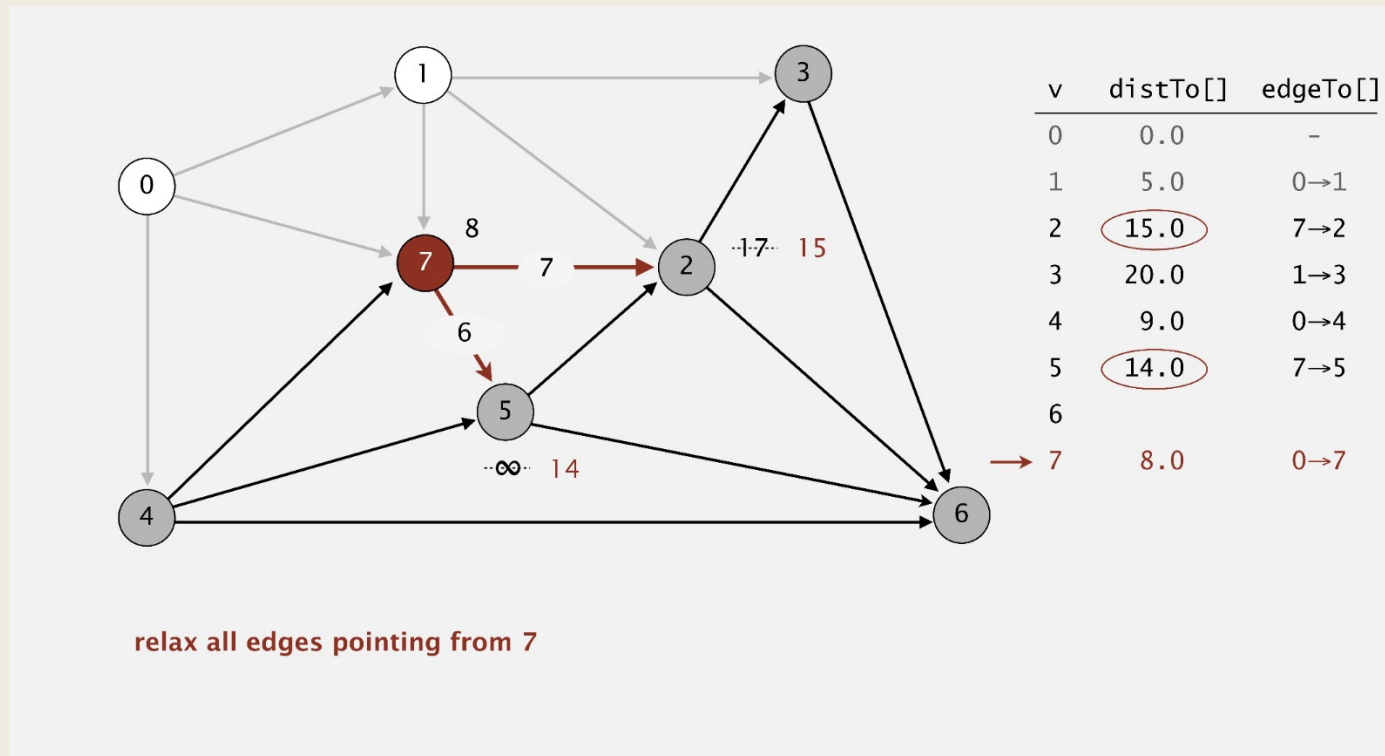
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



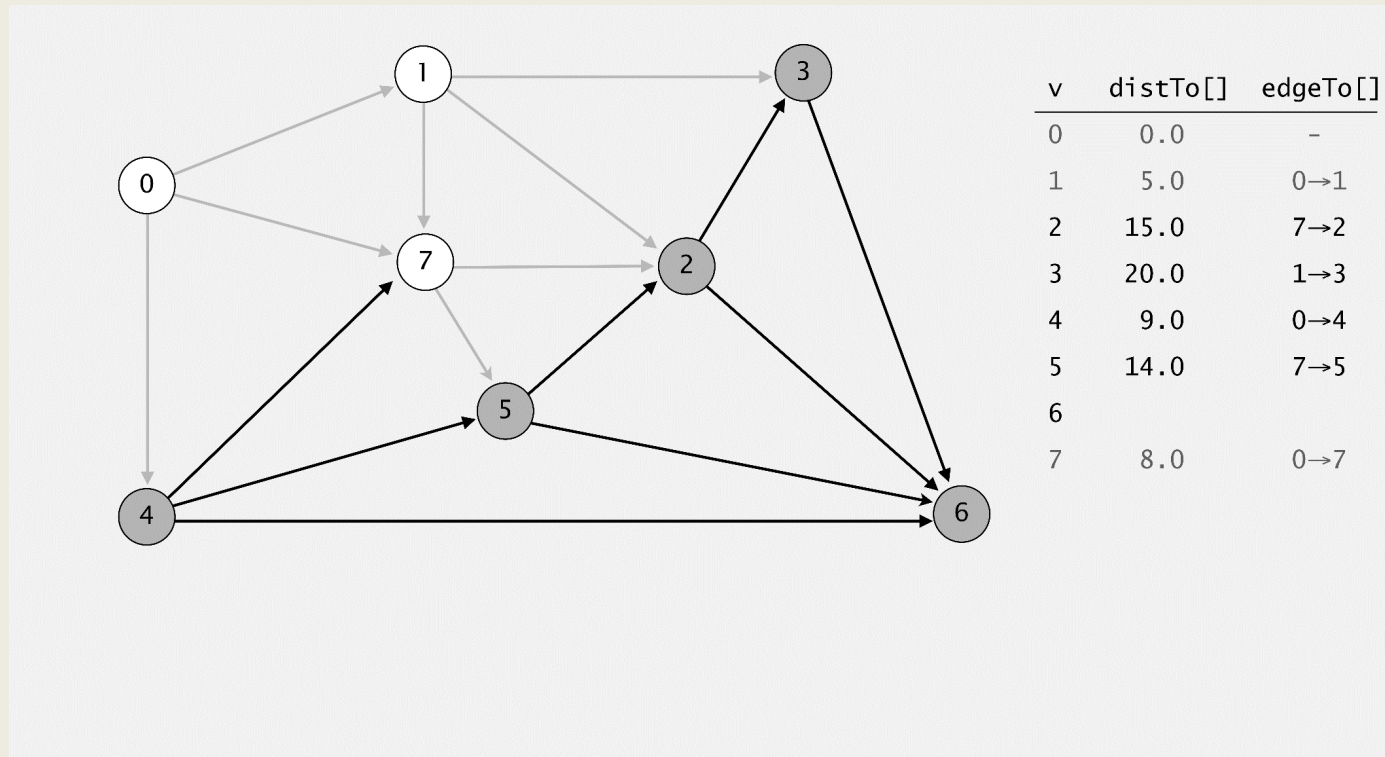
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



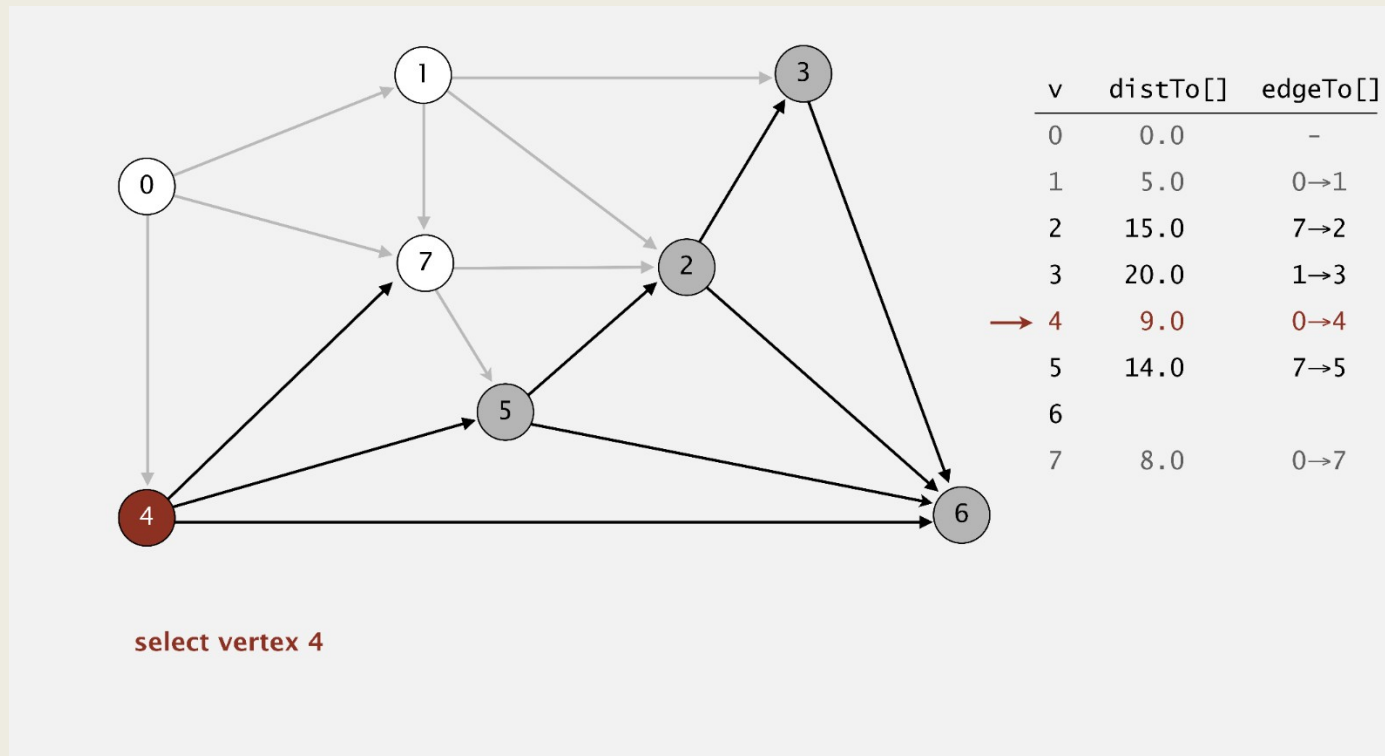
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



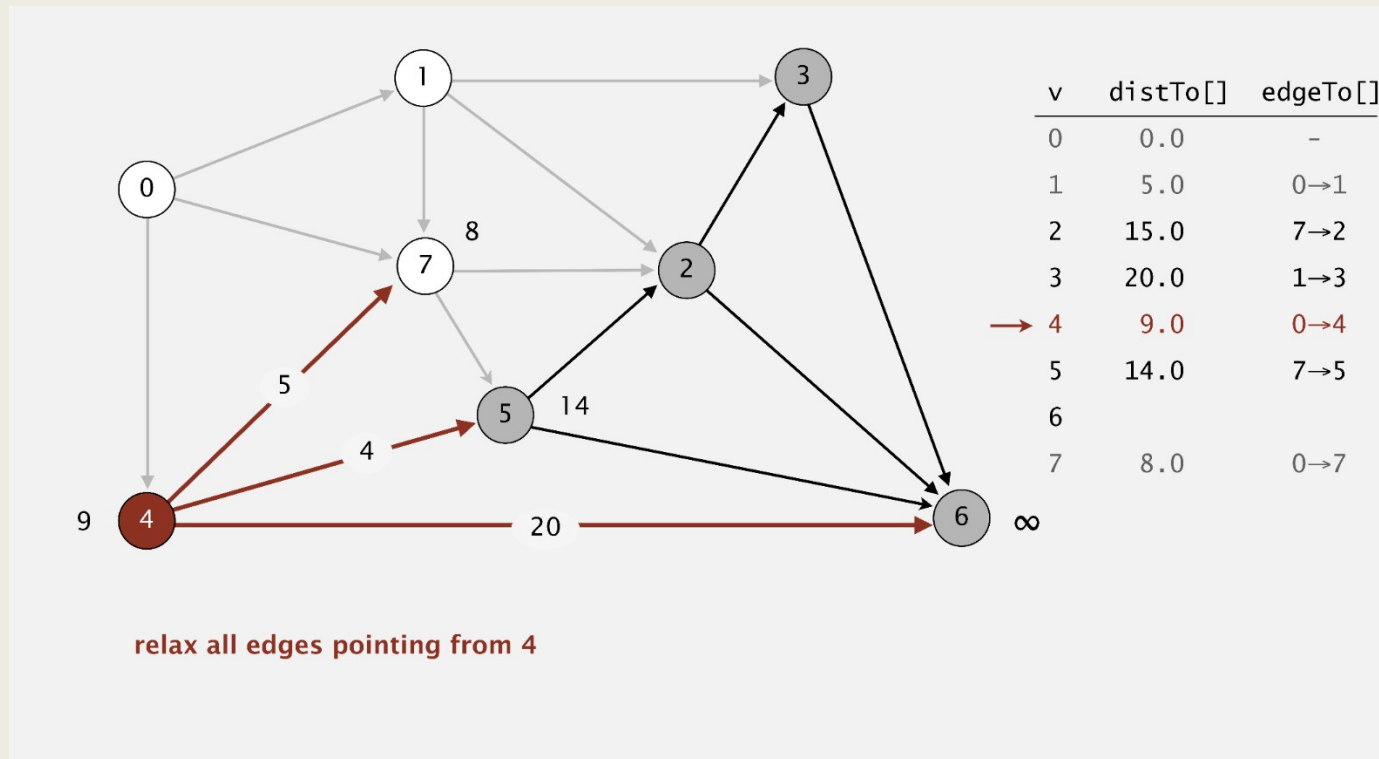
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



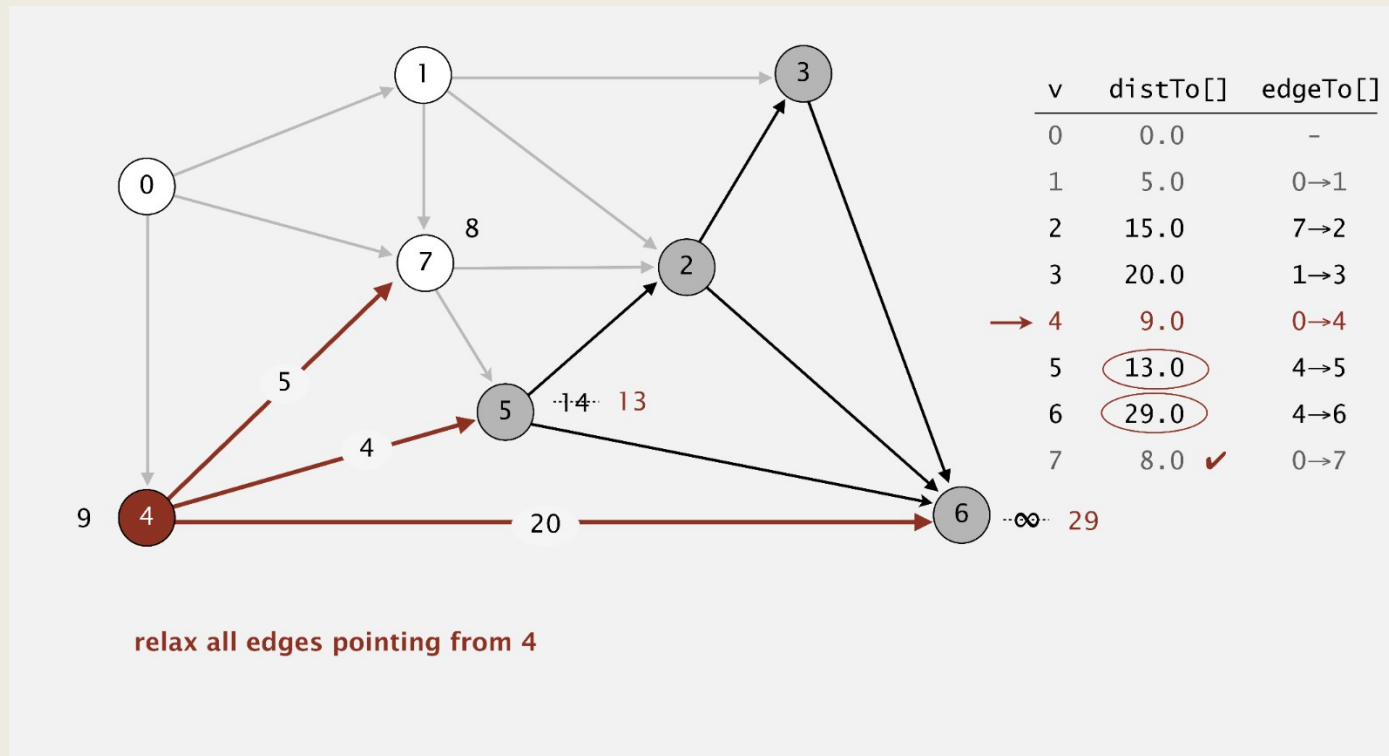
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



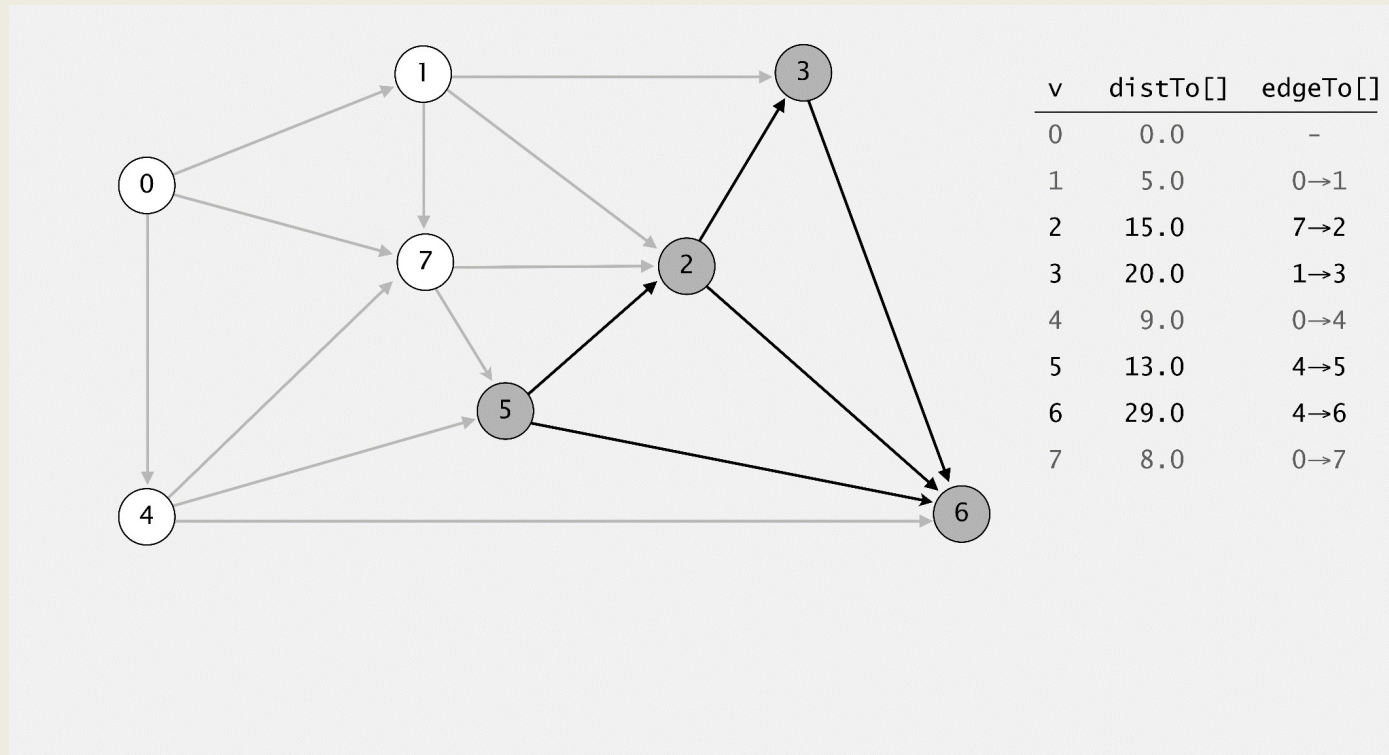
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



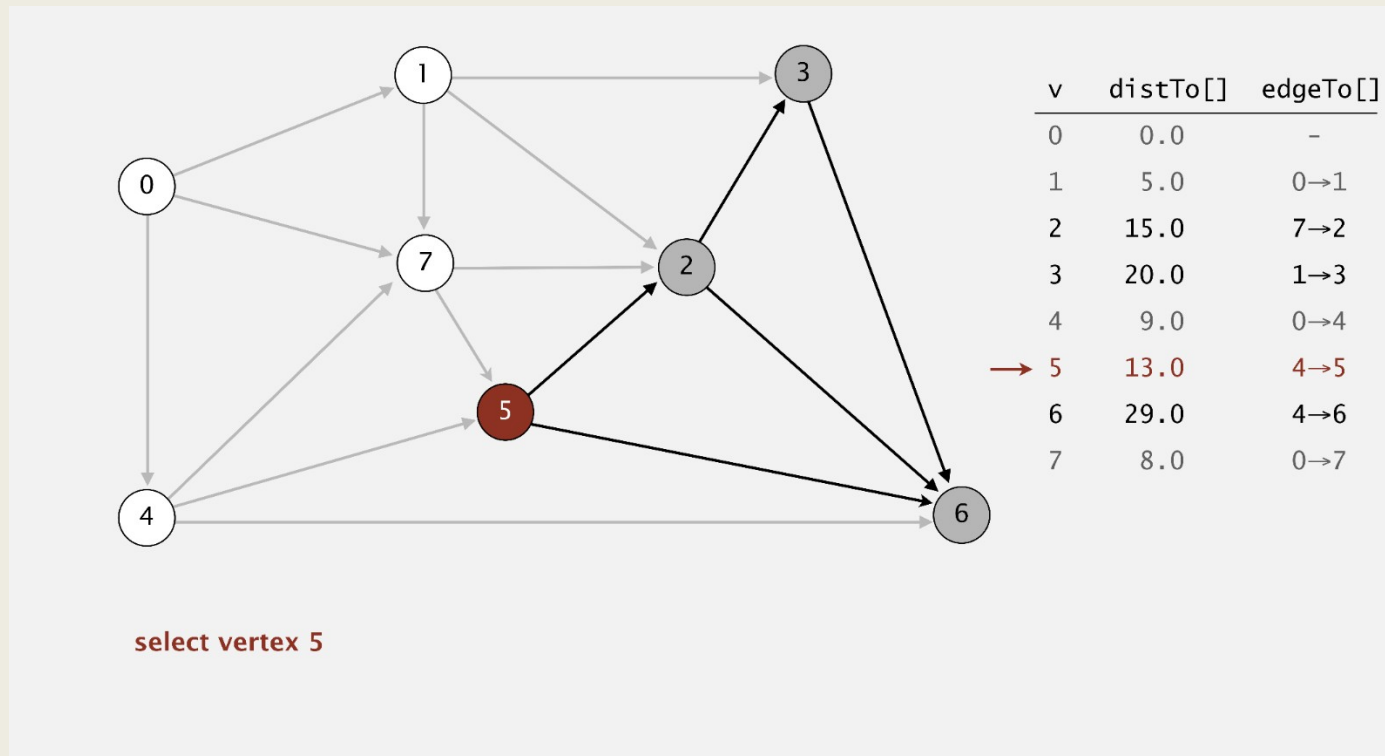
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



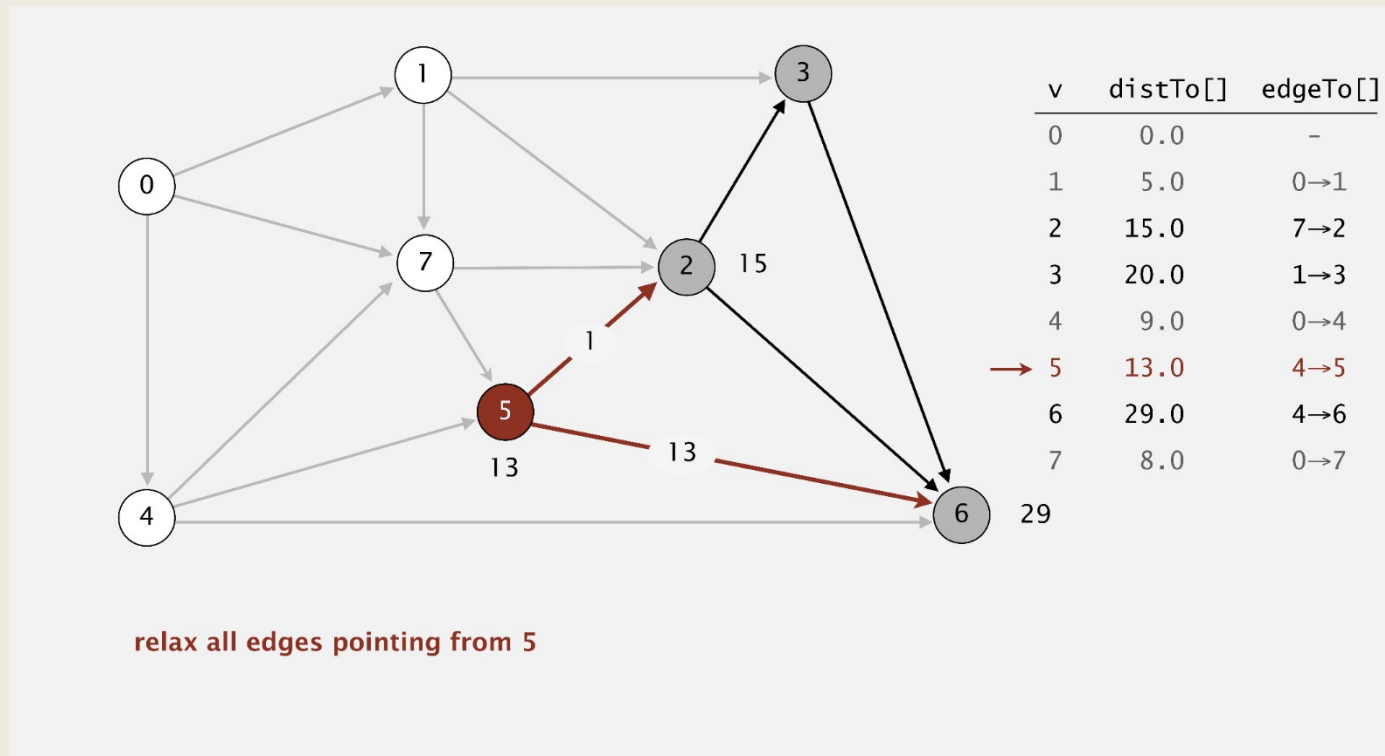
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



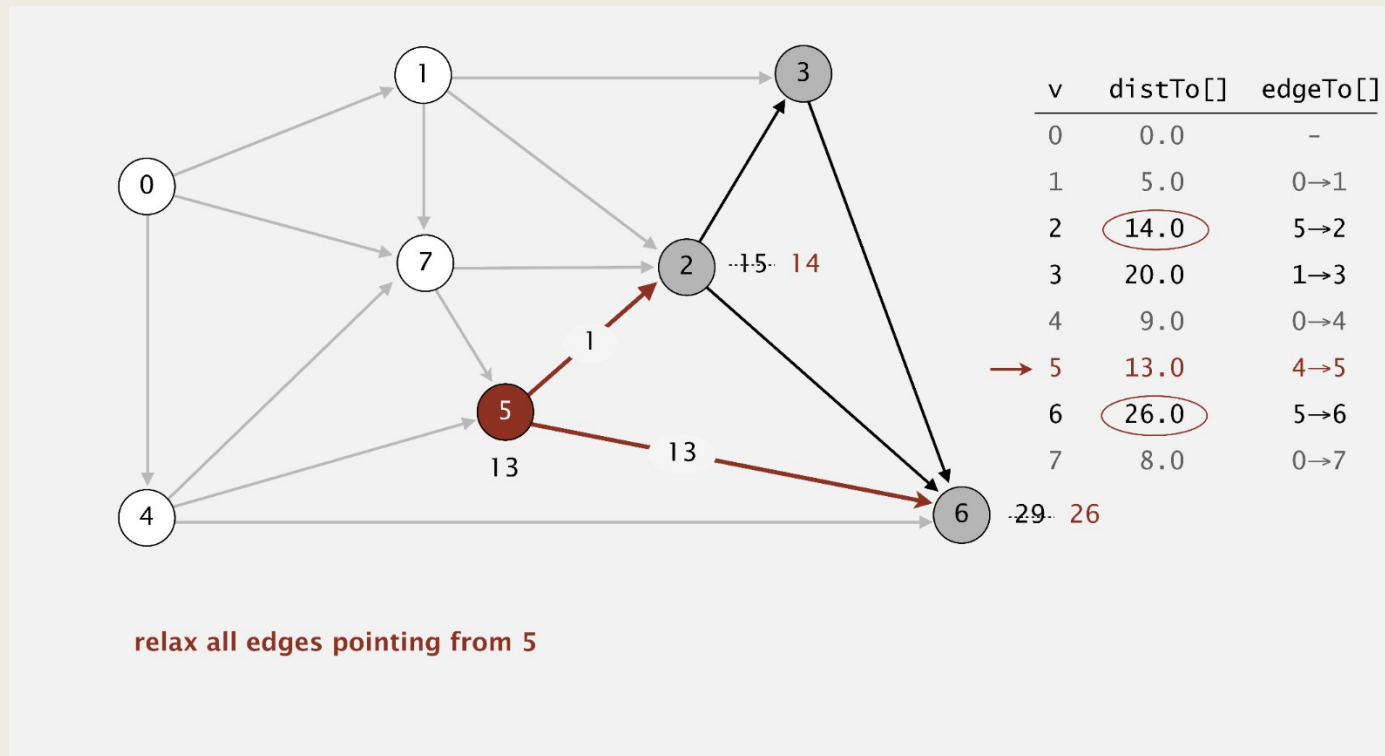
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



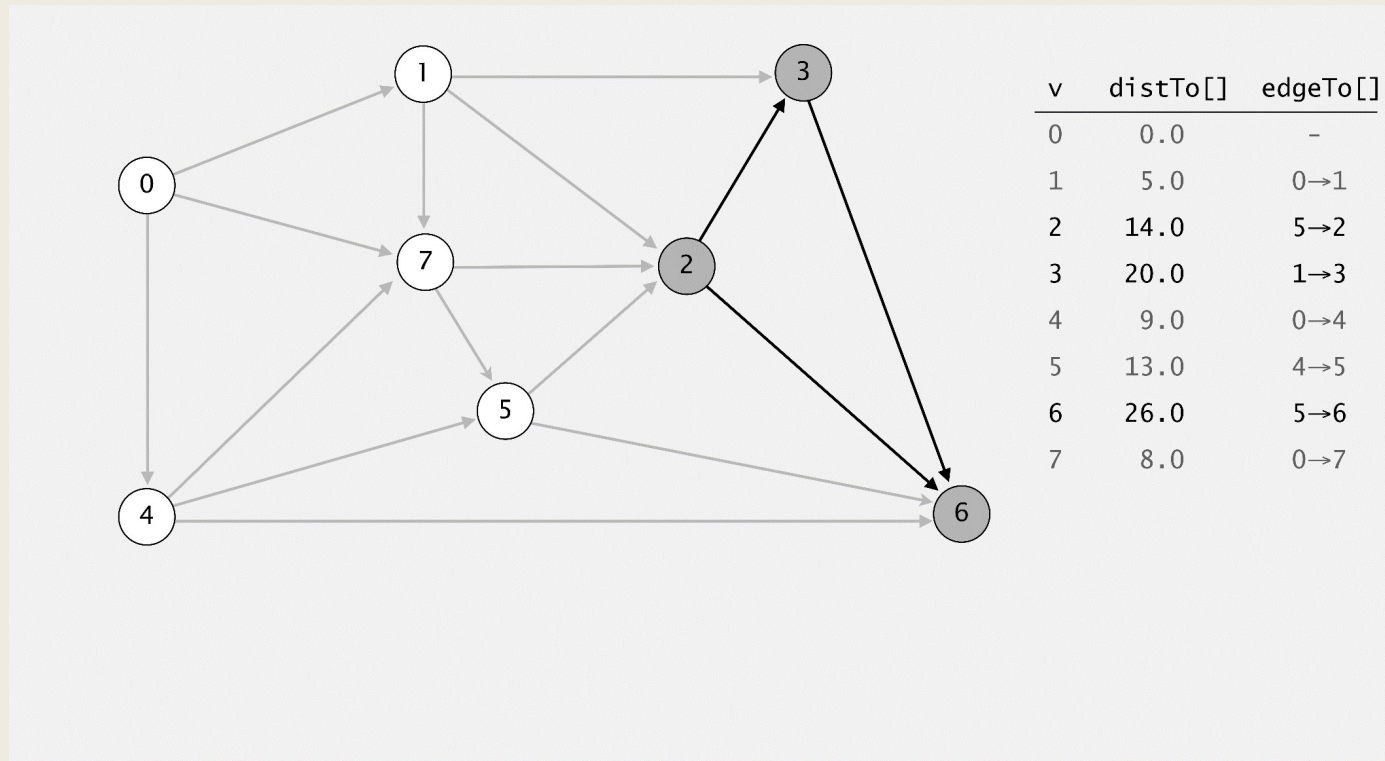
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



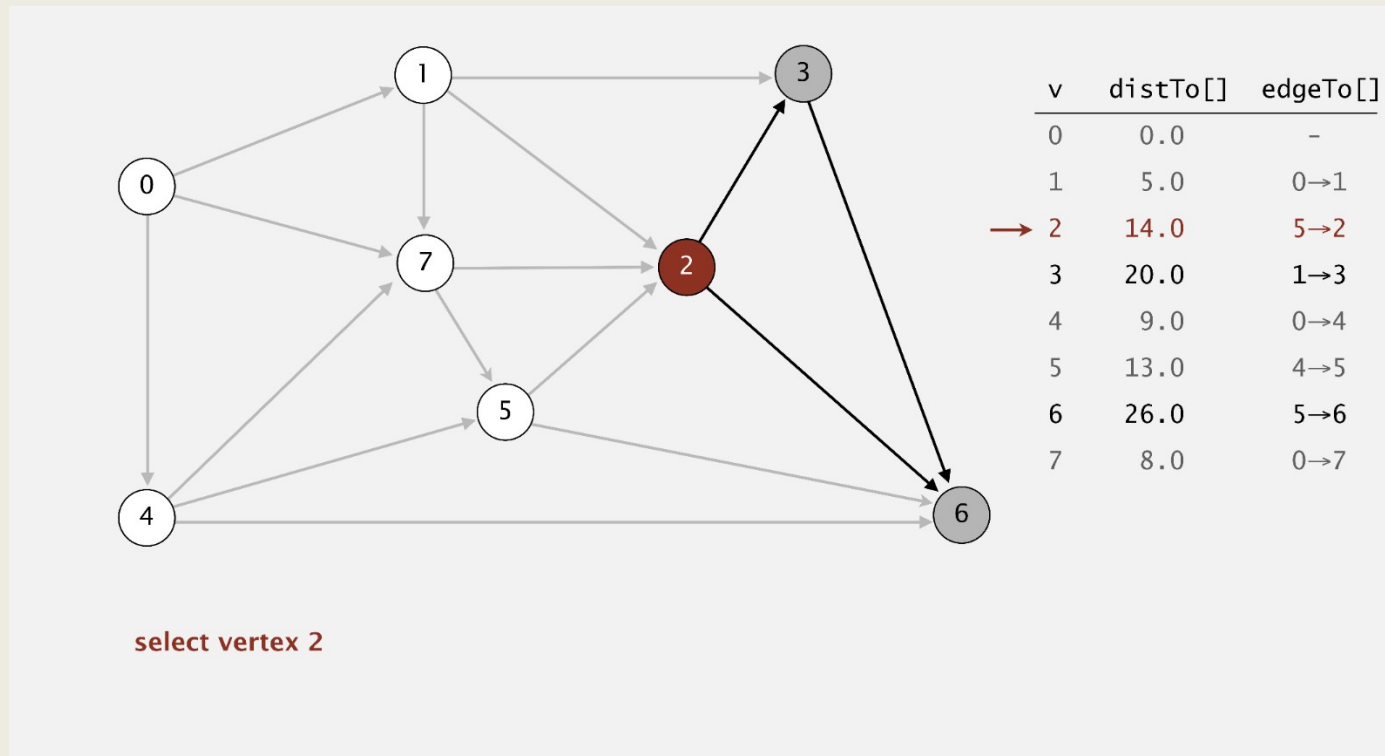
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



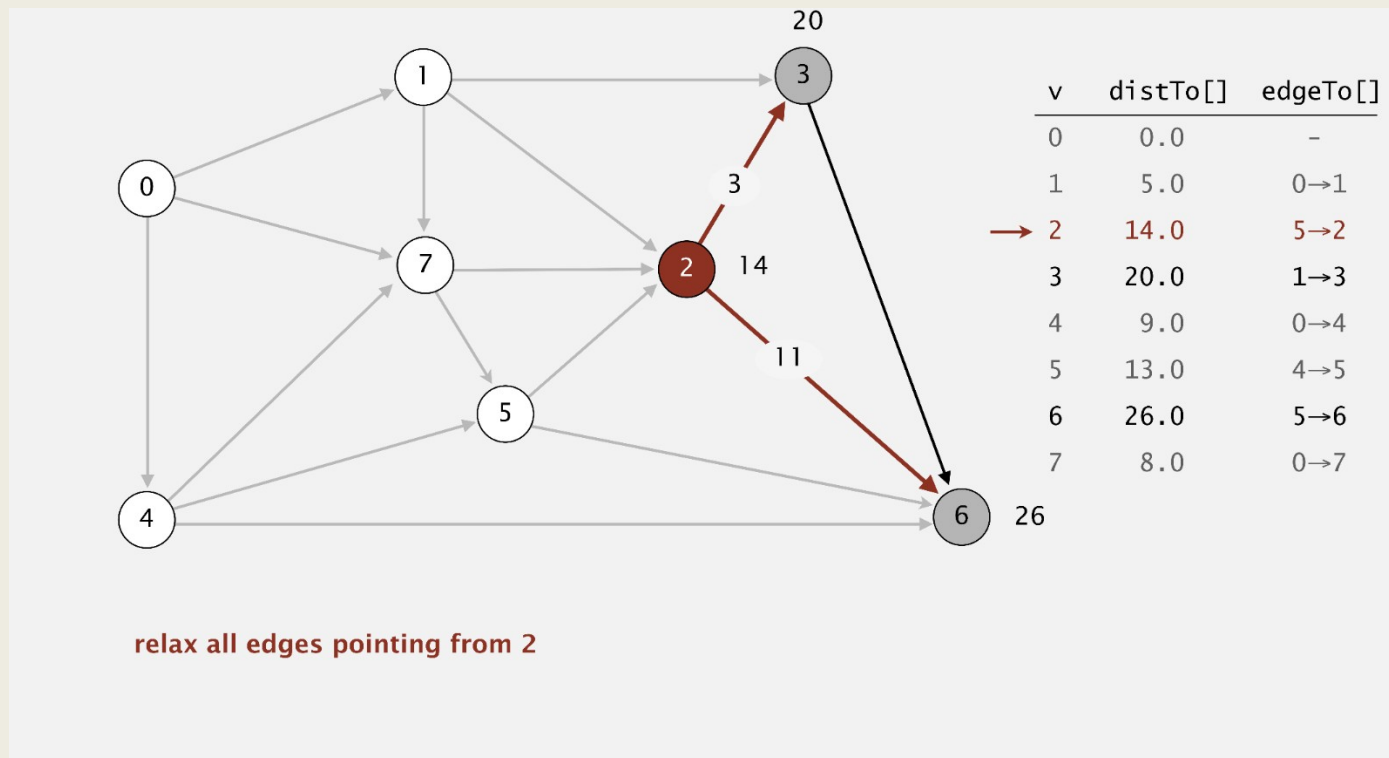
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



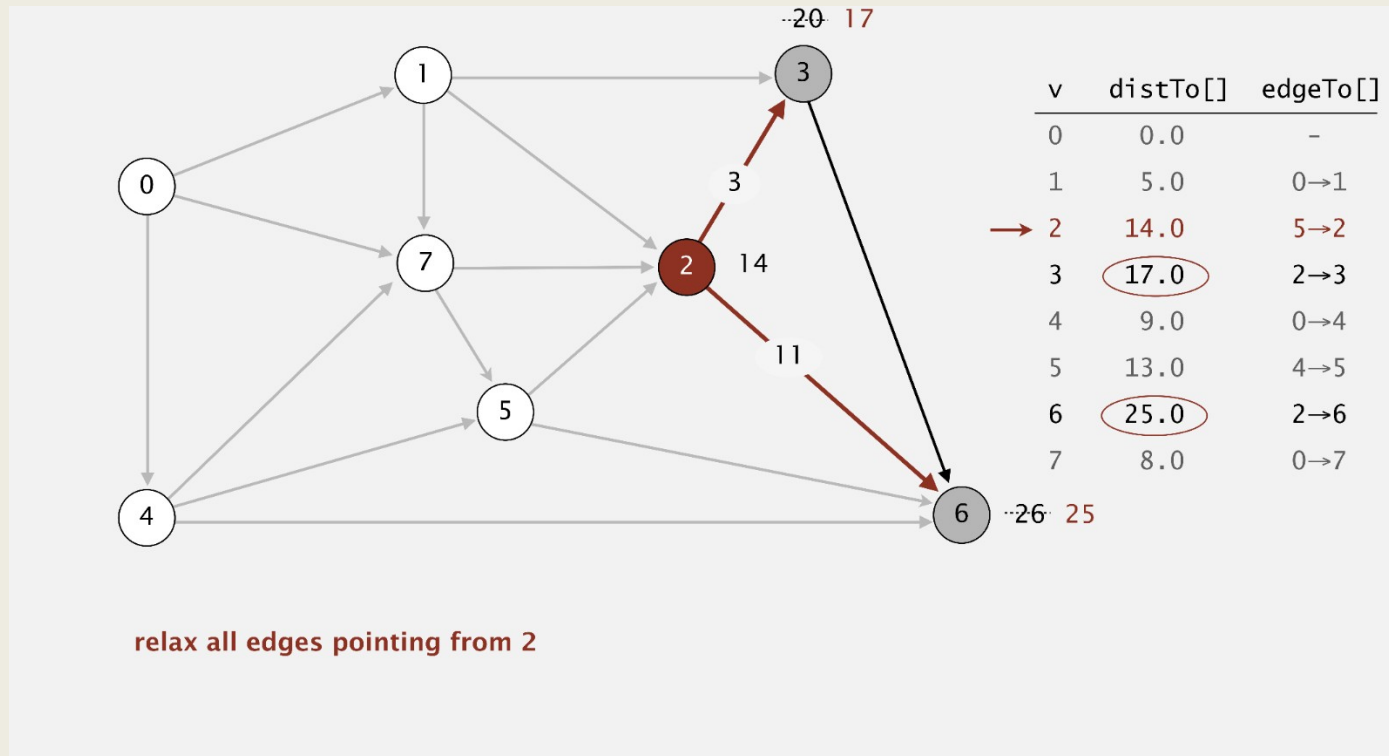
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



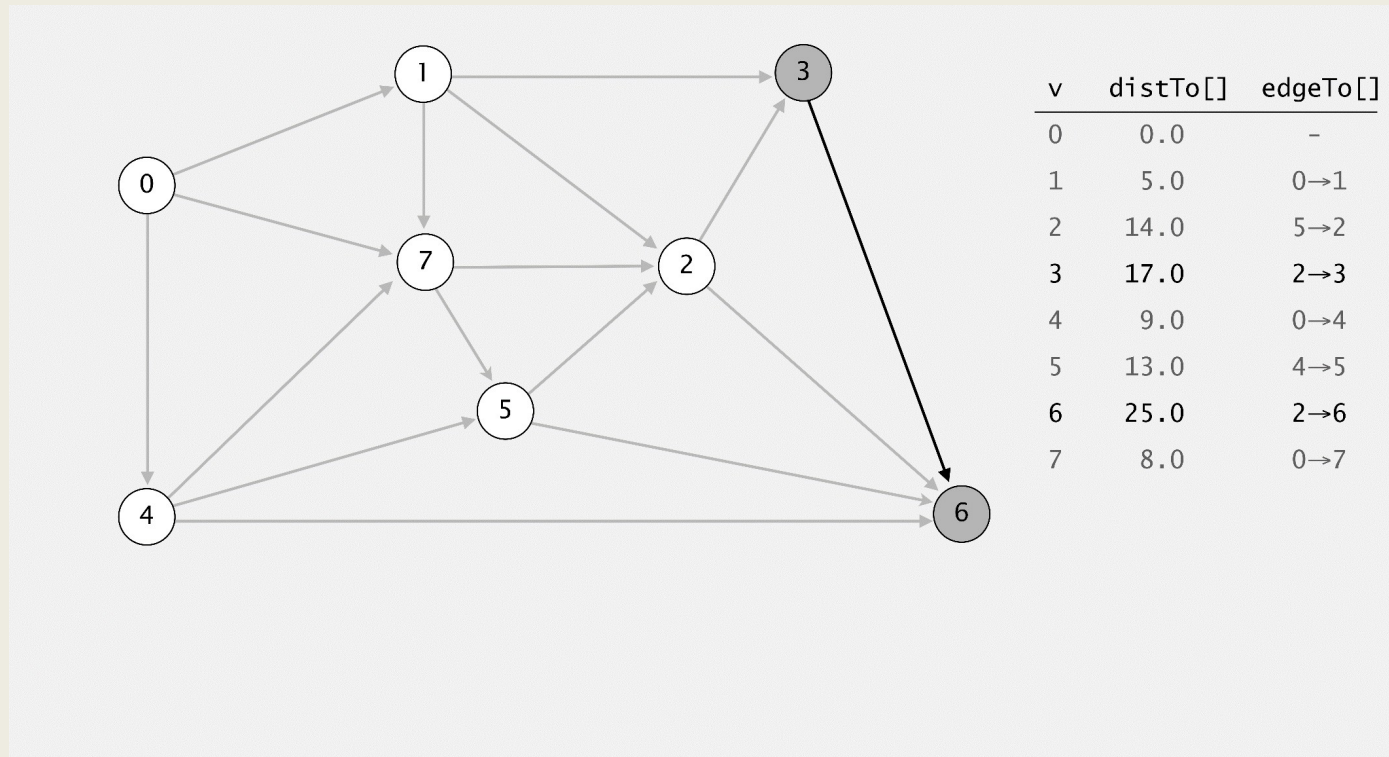
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



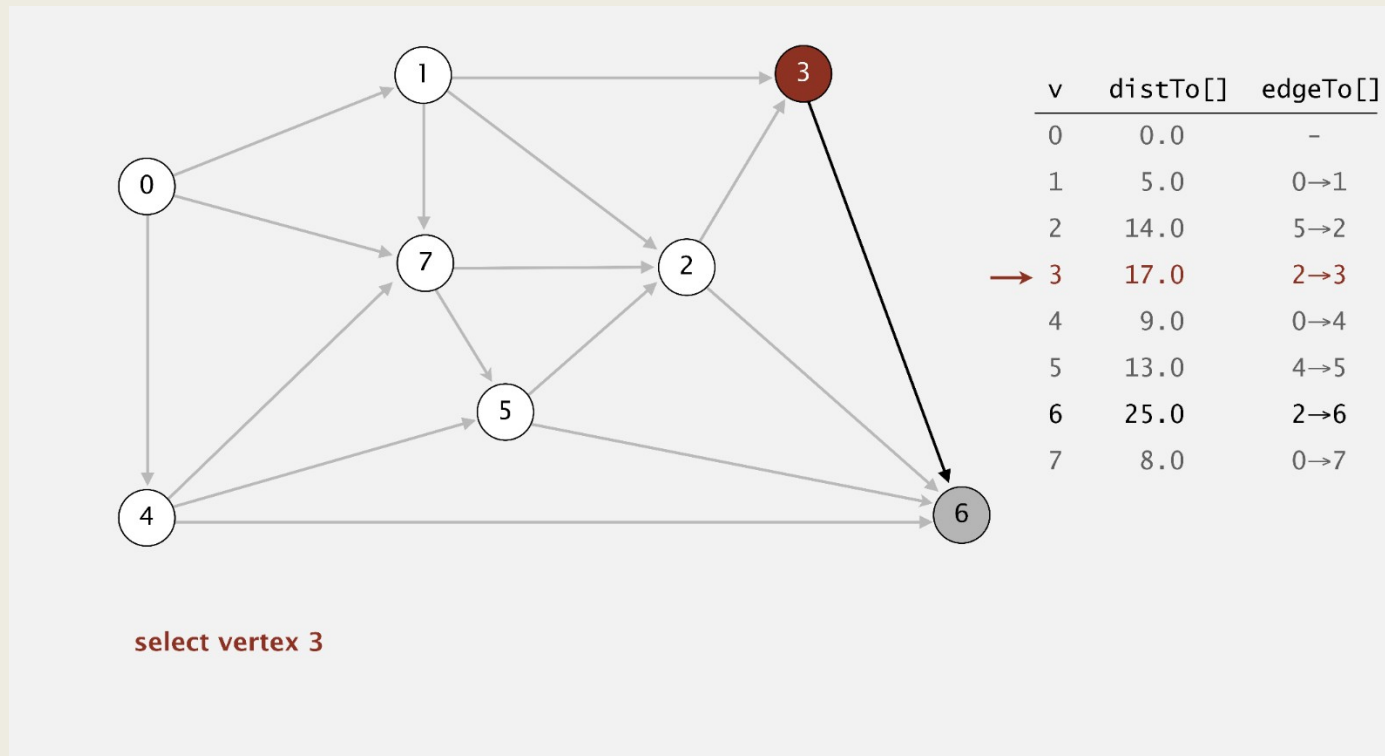
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



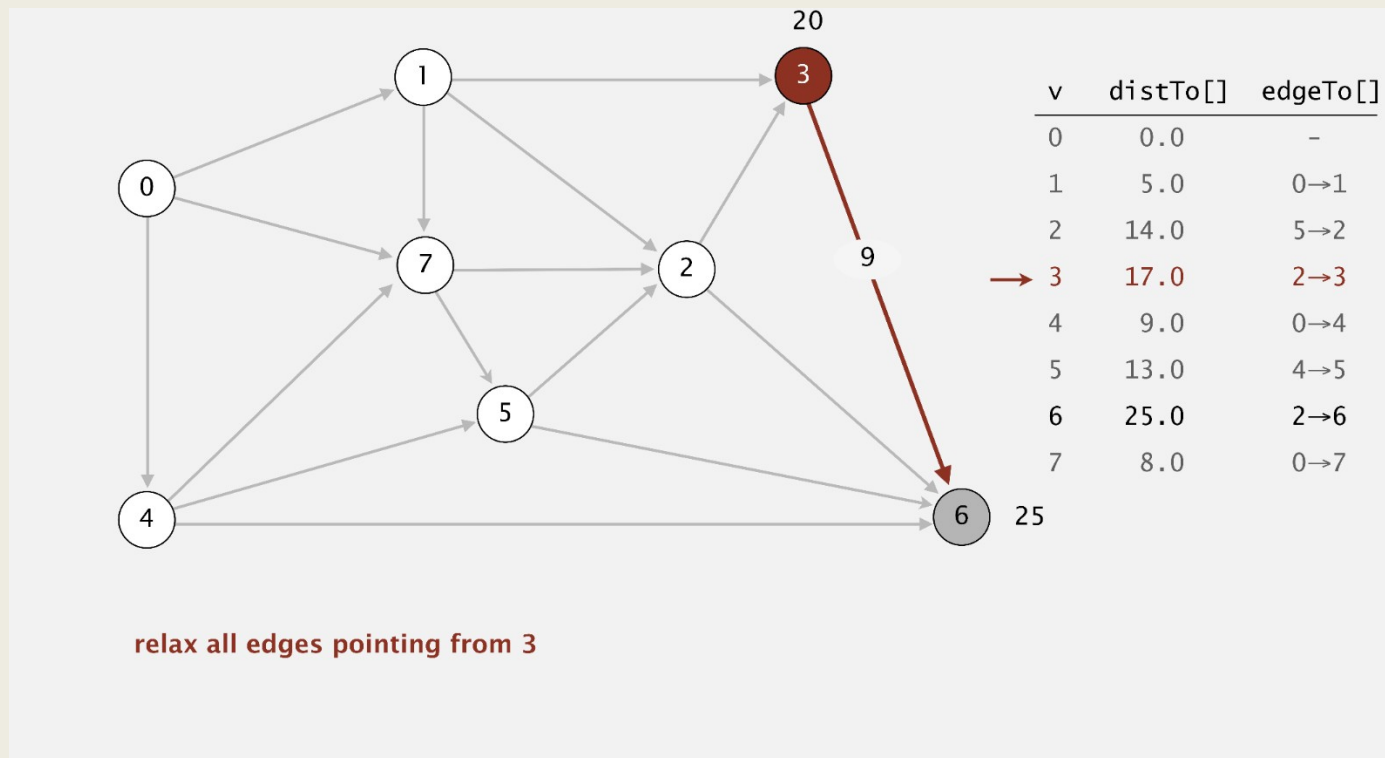
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



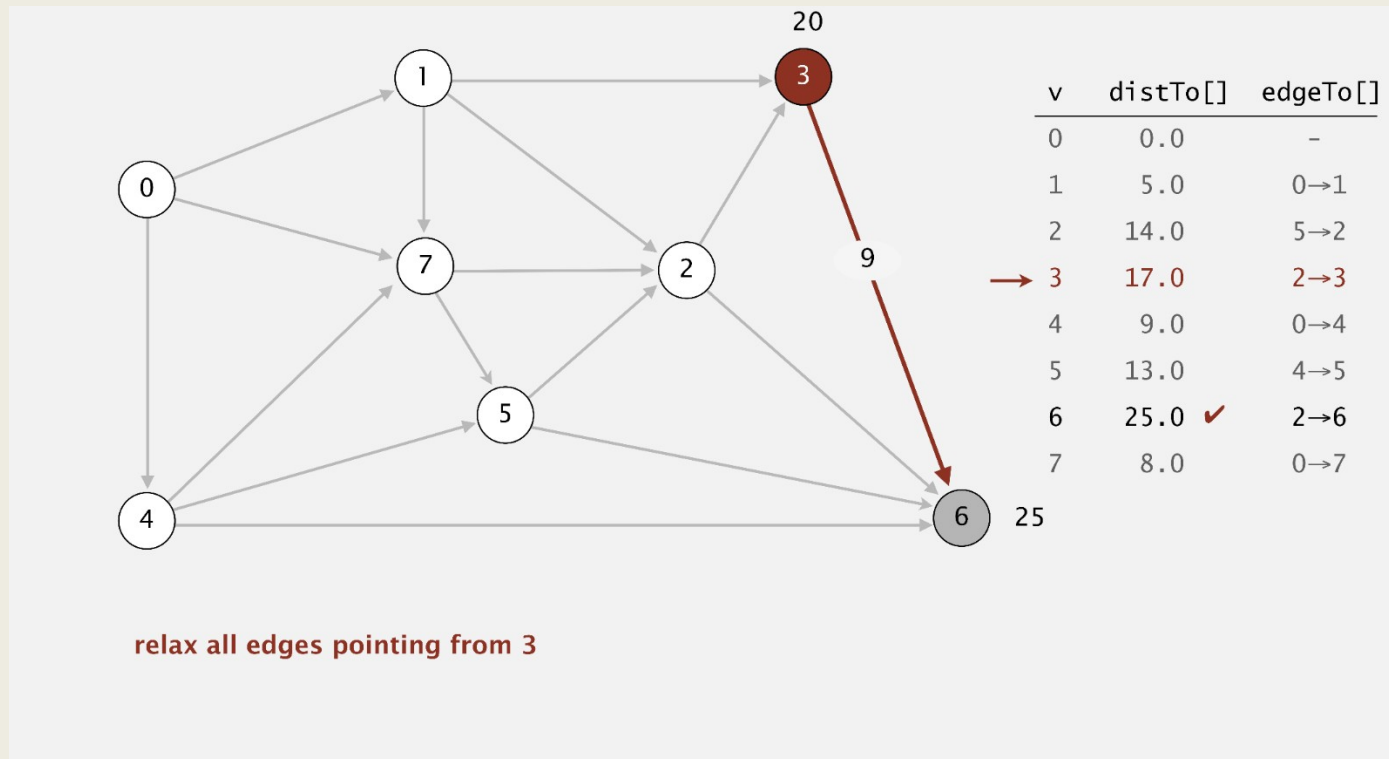
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



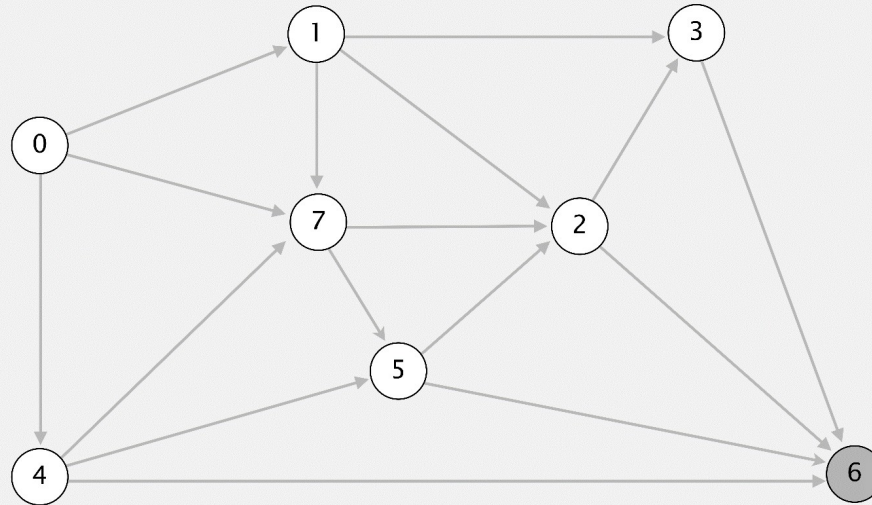
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



DIJKSTRA'S ALGORITHM

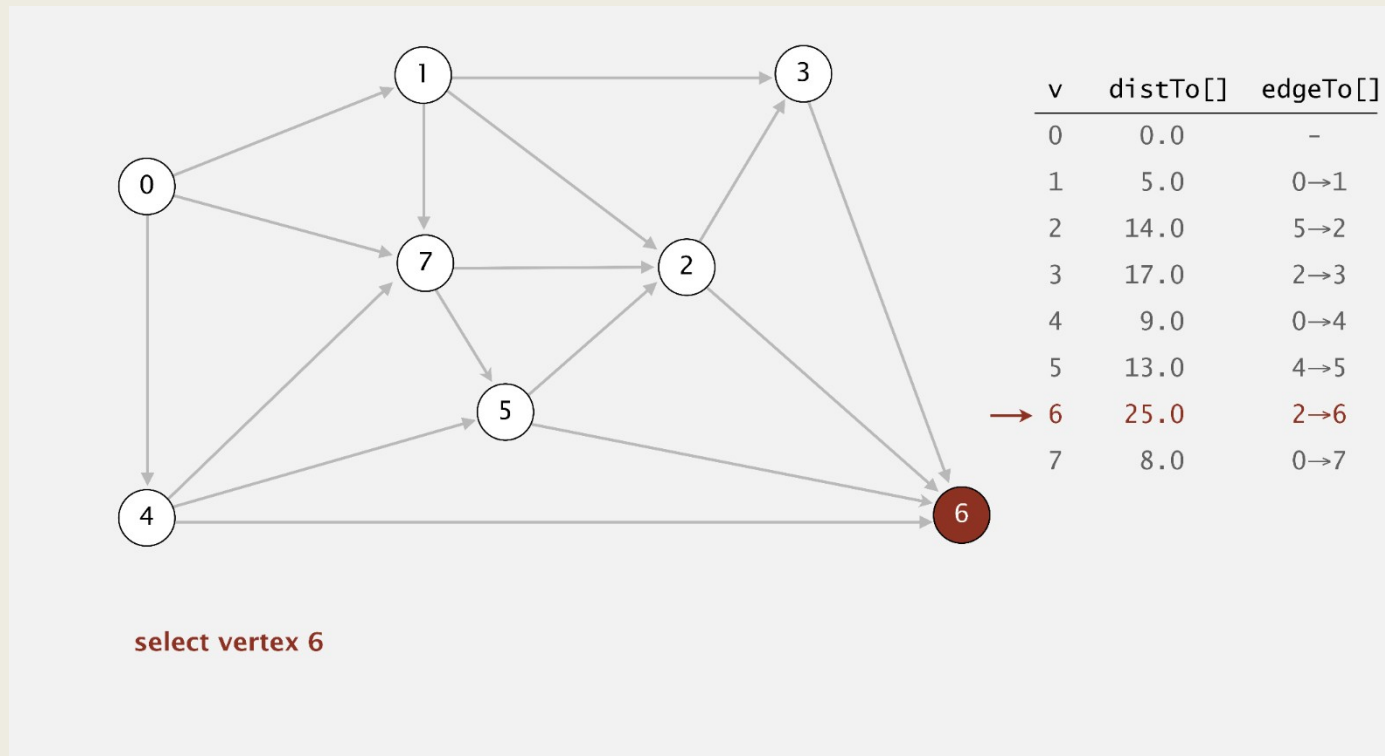
- Challenge: find the shortest path between two vertices.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

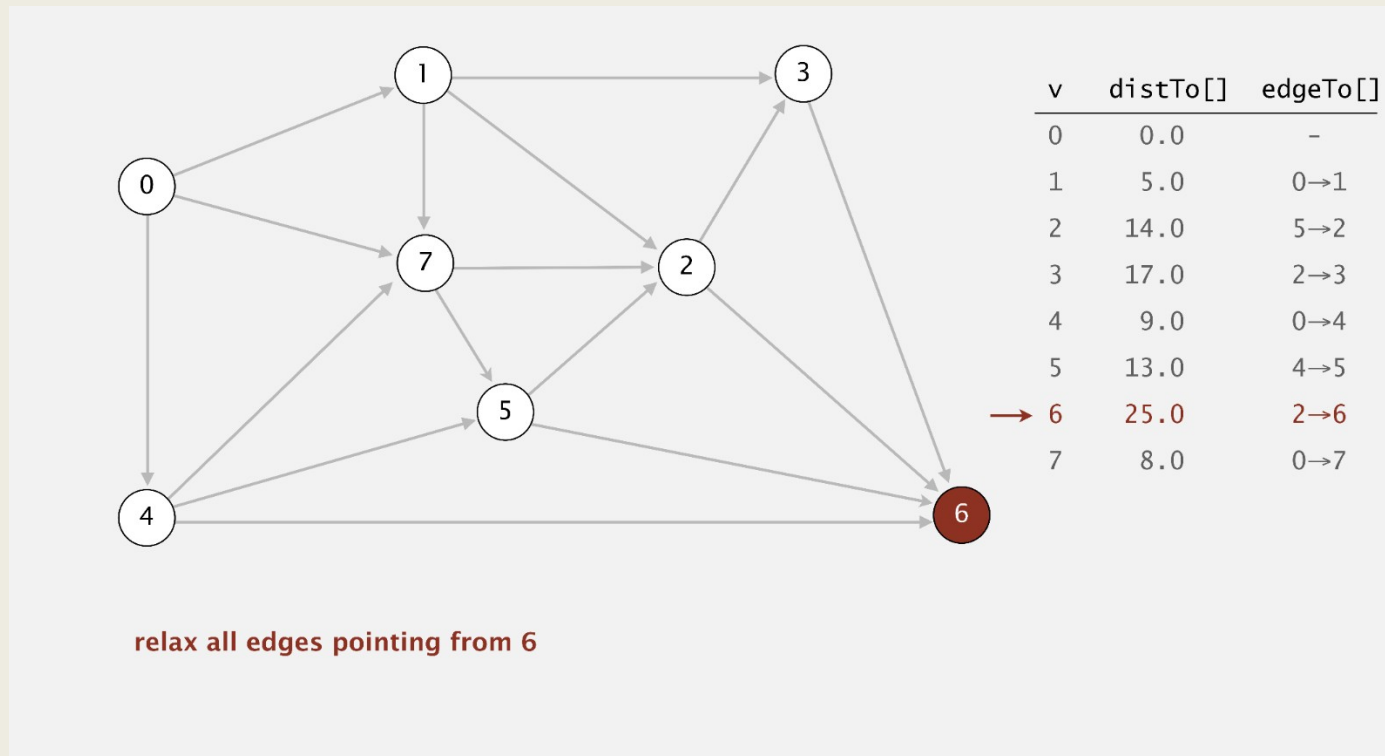
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



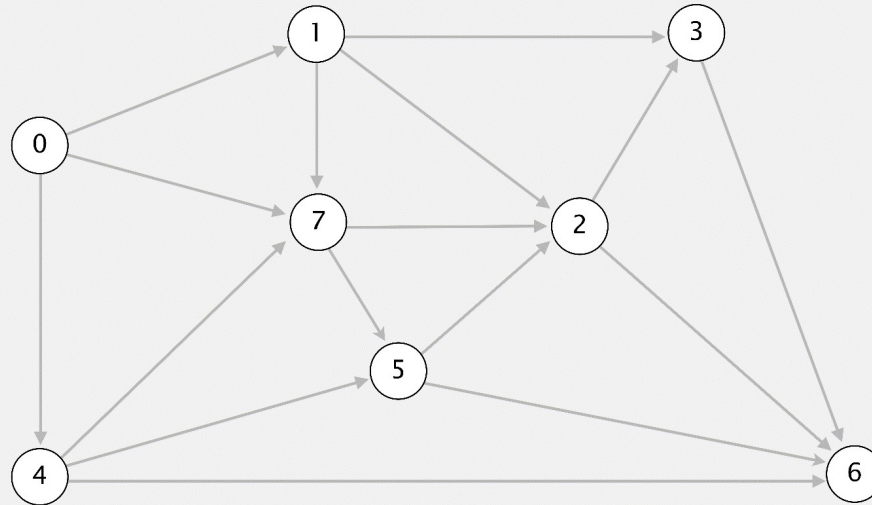
DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



DIJKSTRA'S ALGORITHM

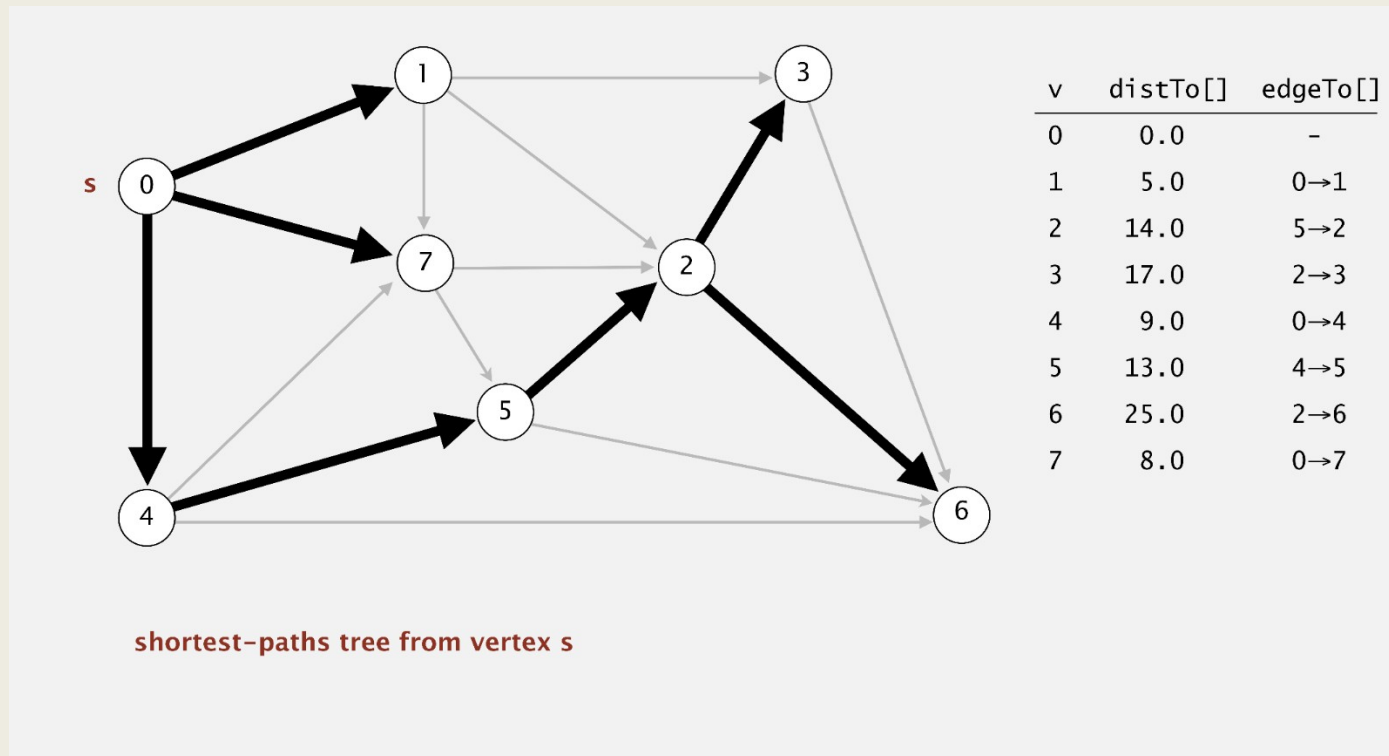
- Challenge: find the shortest path between two vertices.



v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

DIJKSTRA'S ALGORITHM

- Challenge: find the shortest path between two vertices.



```
def dijkstra(graph, starting_vertex):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[starting_vertex] = 0
    pq = [(0, starting_vertex)]

    while len(pq) > 0:
        current_distance, current_vertex = heapq.heappop(pq)
        if current_distance > distances[current_vertex]:
            continue

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances
```

```
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 6},
    'C': {'D': 3},
    'D': {}
}
```

```
dijkstra(graph, 'A')
print(dijkstra(graph, 'A'))
```

```
{'A': 0, 'B': 1, 'C': 3, 'D': 6}
```

