Graphs

Single Source Shortest Paths

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Shortest Paths

- Given a directed and weighted graph $G = (V, E)$
  - With a positive real-value weight function $w: E \to R$
  - With a weight $w(p)$ over a path $p = \{v_0, v_1, \cdots, v_k\}$ equal to $w(p) = \sum_{i=0}^{k} w(v_{i-1}, v_i)$
- We define the shortest path weight $\delta(u, v)$ from u to v as

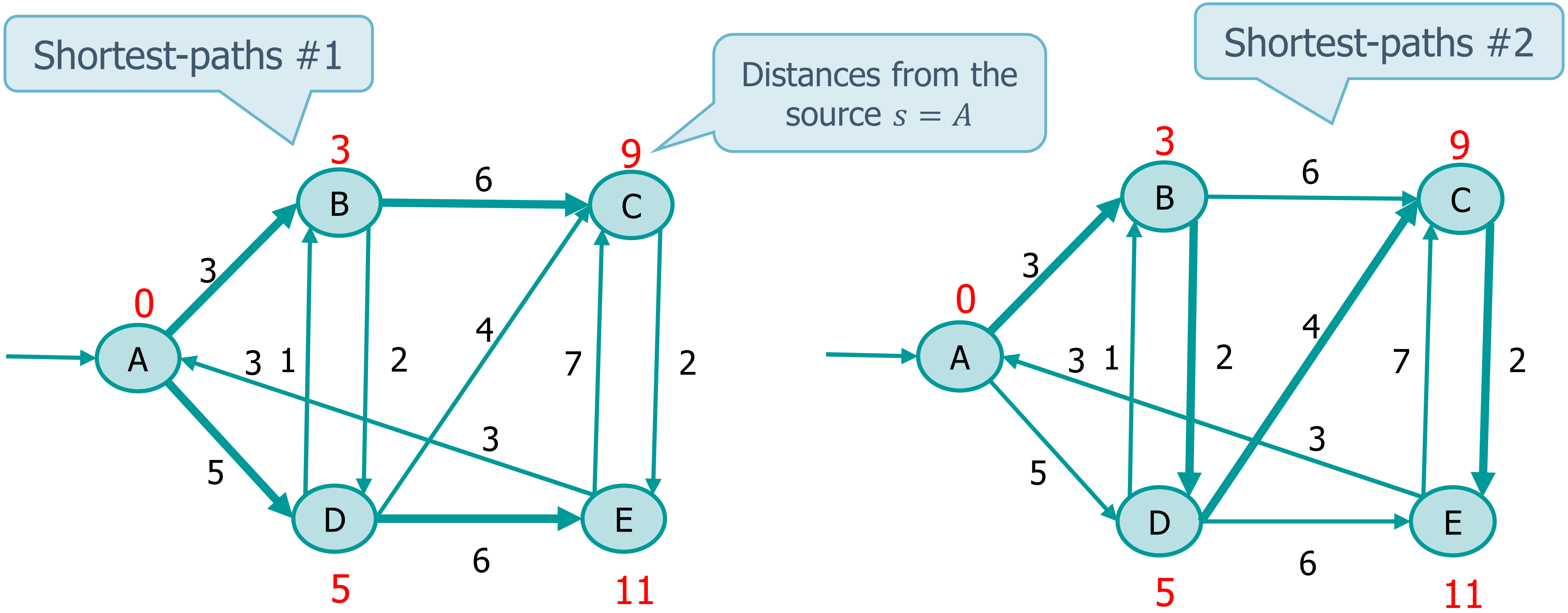
$$\delta(u, v) = \begin{cases} min\{w(p)\} & if\ \exists u \rightarrow_p v \\ \infty & otherwise \end{cases}$$

- A shortest path from u to v is any path p with weigth

$$w(p) = \delta(u, v)$$

Problem definition

- Shortest paths are considered by several different algorithms
  - Single-source shortest-paths (SSSP)
    - Find the minimum path (and its weight) from a source s to all other vertices v



Problem definition

- Application
  - Given a road map
  - How is it possible to determine the shortest route from A to B?
  - Observation
    - A brute force approach would imply checking an enourmous number of paths
    - Enumerate all routes, add the distance on each route, disallowing routes with cycles
    - Select the shortest route

Problem definition

- Single-destination shortest-paths
  - Find the shortest path to a given destination d
    - Use the SSSP working on the reverse graph
- Single-pair shortest-paths
  - Find a shortest path from v to u
    - Solved when the SSSP is solved
    - All alternative solutions have the same worst-case asymptotic running time
- All-pairs shortest-path
  - Find a shortest-path for every vertex pair (v, u)
    - Can be solved running SSSP from each vertex
    - Can be solved faster

Algorithms

- We focus on the SSSP problem
  - For **unweighted** graphs
    - A simple **BFS** (Breadth-First Search) solves the problem
  - For graphs with negative weight edges
    - We can use the **Bellman-Ford**'s algorithm
  - For graphs with non-negative weight edges
    - We can use the more efficient **Dijkstra**'s algorithm
  - For complex graphs in travel-routing systems
    - We can adopt A* an extensiono fthe Dijkstra's method

Algorithms

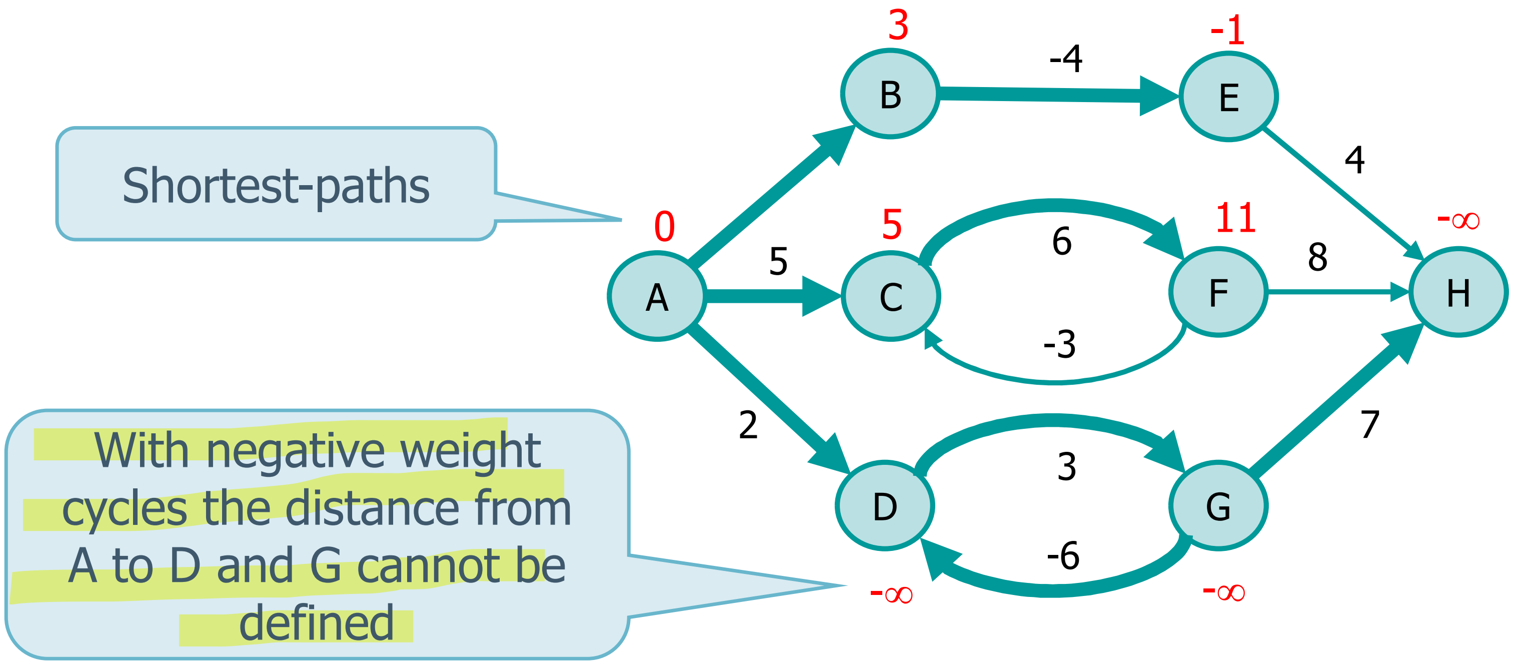
- Bellman-Ford’s algorithms
  - If there are negative edges but no cycles with negative weight, it finds an optimal solution
  - If there are negative weight cycles, it detects them
  - Published by Shimbel (1955), Ford (1956), Bellman (1958) it is modestly scalable and it converges slowly
- Dijsktra’s algorithm
  - With negative edges, it is unable to finds an optimal solution
  - It is more efficient than Bellman-Ford’ algorithm

Algorithms

- A*
  - Published for the first time by Hart, Nilsson and Raphal (1968)
  - It can be seen as an extension of Dijkstra's algorithm
  - Achieve better performance by using heuristics to guide the search

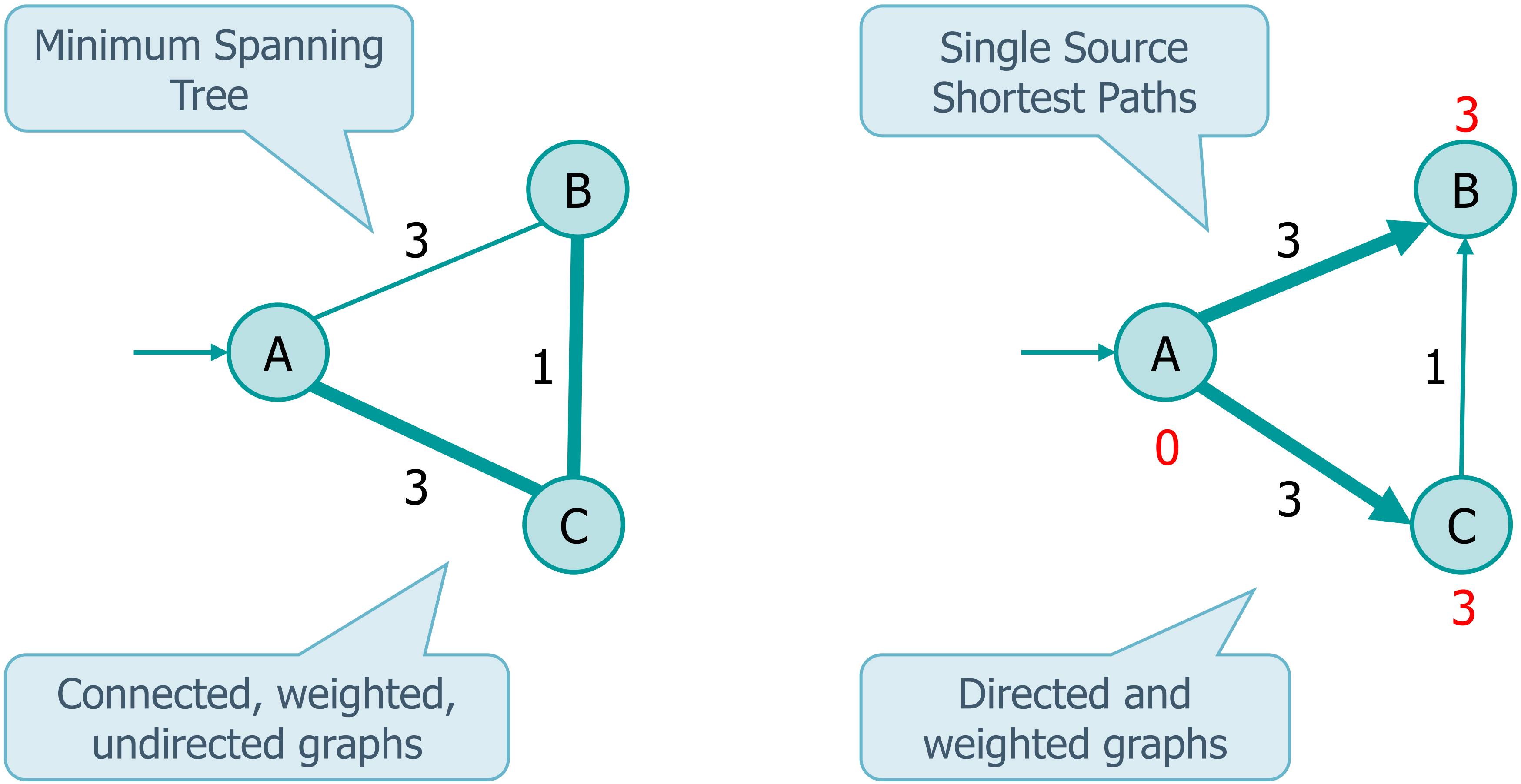
Observations

- With negative weight cycles the SSSP problem cannot be correctly defined



Observations

- SSSPs and MSTs are different

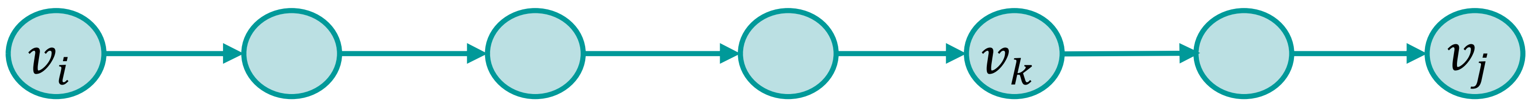


Observations

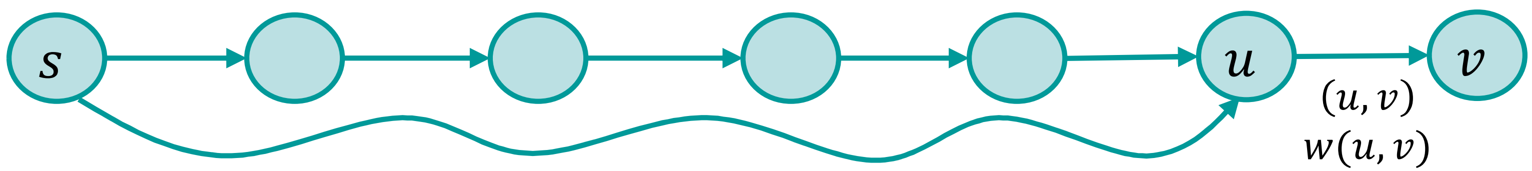
- If we wish to collect the vertices on the shortest path, not only the weight of the path, we can use representations similar to the one used to store a BFS tree
  - Array of predecessors
    - We maintain the predecessor for each vertex
  - Predecessor's sub-graph
    - We create a graph using the predecessor array
  - Shortest-Paths Tree
    - We store the tree rooted at the source node to represent all shortedst path

Theoretical Background

- Lemma
  - Sub-paths of shortest paths are shortest paths



  - It the path $v_i \rightarrow_p v_j$ is mimimum also $v_i \rightarrow_p v_k$ and $v_k \rightarrow_p v_j$ are minimum
- Corollary
  - A shortest path $s \rightarrow_p v$ be decomposed into a shortest path $s \rightarrow_p u$ and an edge (u, v)



Relaxation

- SSSP algorithms are based on **relaxation**
  - For each vertex we mantain an estimate $v.dist$ (superior limit) of the weight of the path from s to v

(Single) source

```
initialize_single_source (G, s)
  for each v ∈ V
    v.dist = ∞
    v.pred = NULL
  s.dist = 0
```

$v.pred$ = predecessor

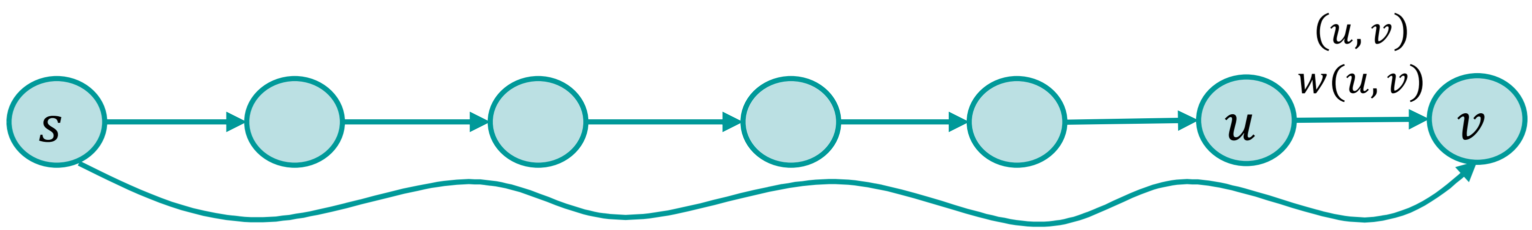
$v.dist$
= shortest path estimate = upper bound on the weight of a shortest path from s to v

Relaxation

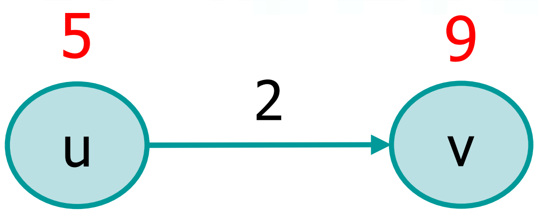
- Relaxation
  - Update $v.dist$ and $v.pred$ by testing whether it is possibile to improve the shortest path to v found so far by going through the edge (u, v), where $w(u, v)$ is the weigth of the edge

Relaxation does not increase $v.dist$

```
relax (u, v, w) {
  if ( v.dist > (u.dist + w(u, v)) ) {
    v.dist = u.dist + w (u, v)
    v.pred = u
  }
}
```



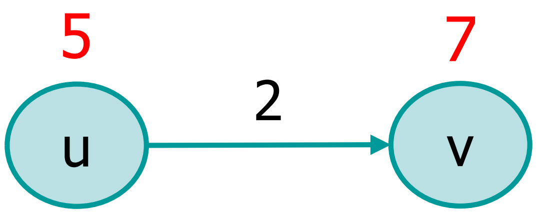
Example



$$u.dist = 5$$
$$w(u,v) = 2$$
$$v.dist = 9$$
$$v.dist > u.dist + w(u,v)$$

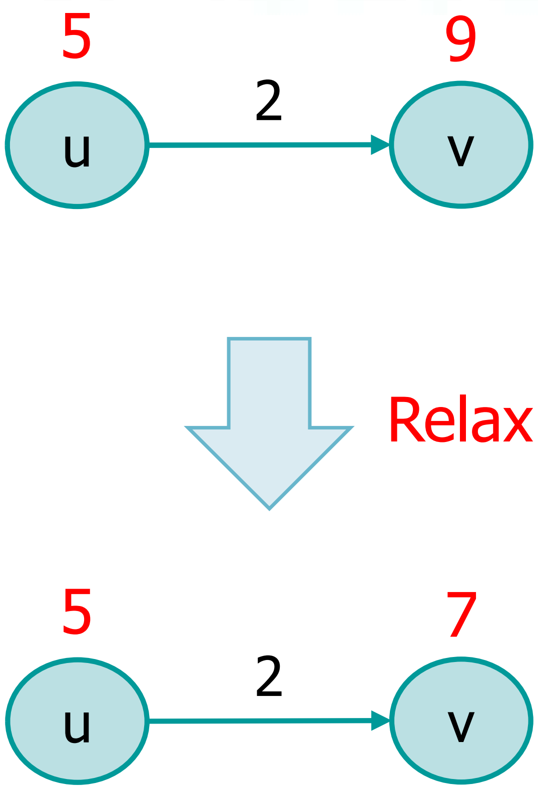
Relax

The shortest path from s to v is the shortest path to u plus $w(u,v)$



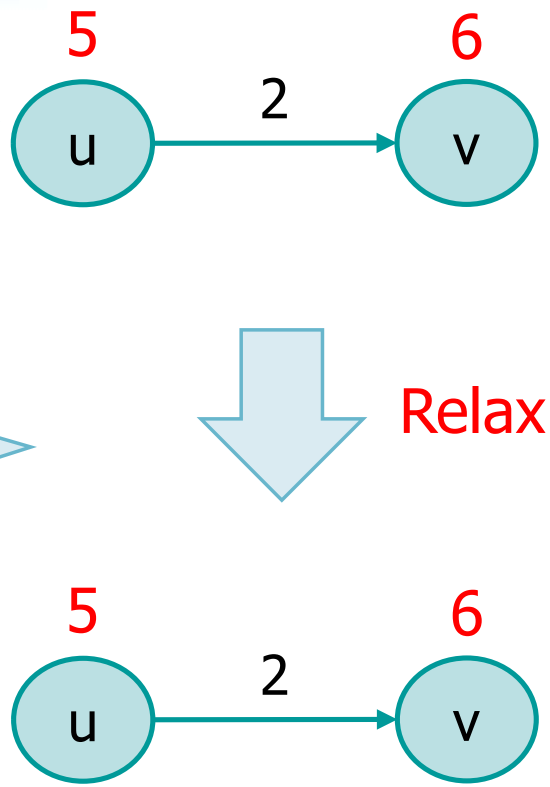
$$u.dist = 5$$
$$w(u,v) = 2$$
$$v.dist = u.dist + w(u,v) = 5 + 2 = 7$$
$$v.pred = u$$

Example



$$u.dist = 5$$
$$w(u,v) = 2$$
$$v.dist = 6$$
$$v.dist < u.dist + w(u,v)$$

The shortest path from s to v is shorter than the path to u plus $w(u,v)$



Relaxation ha so effect

$$u.dist = 5$$
$$w(u,v) = 2$$
$v.dist = 6$ and $v.pred = unchangec$

Dijkstra's Algorithm

- It works on graphs with no negative weigths
- It is a greedy strategy
  - It applies relaxation once for all edges
- Algorithm
  - S = set of vertices whose shortest path from s has already been computed
  - $V-S$ = priority queue Q of vertices till to estimate
  - While Q is not empty
    - Extract u from $V-S$ ($u.dist$ is minimum)
    - Insert u in S
    - Relax all outgoing edges from u

Pseudo-code

Pseudo-code

```
sssp_Dijkstra (G, w, s)
  initialize_single_source (G, s)
  S = ∅
  Q = V
  while Q ≠ ∅
    u = extract_min (Q)
    S = S U {u}
    for each vertex v ∈ adjacency list of u
      relax (u, v, w)
```

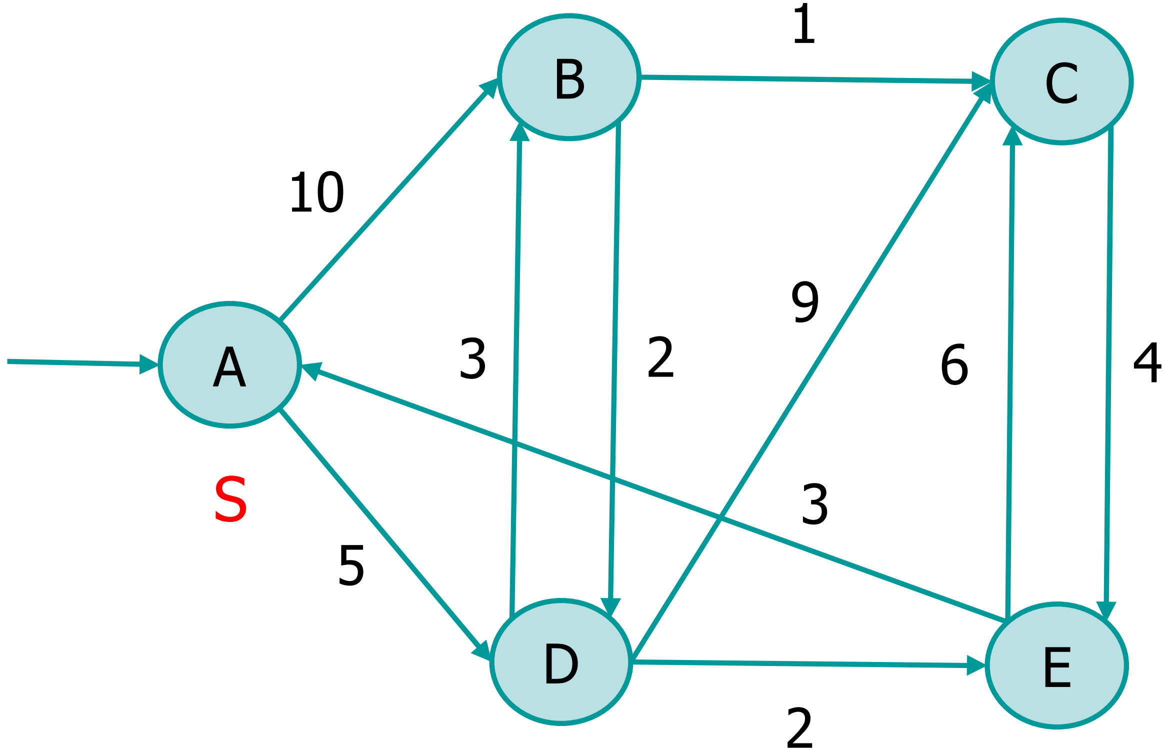
For all vertices starting from s

Extract vertex with minimum distance

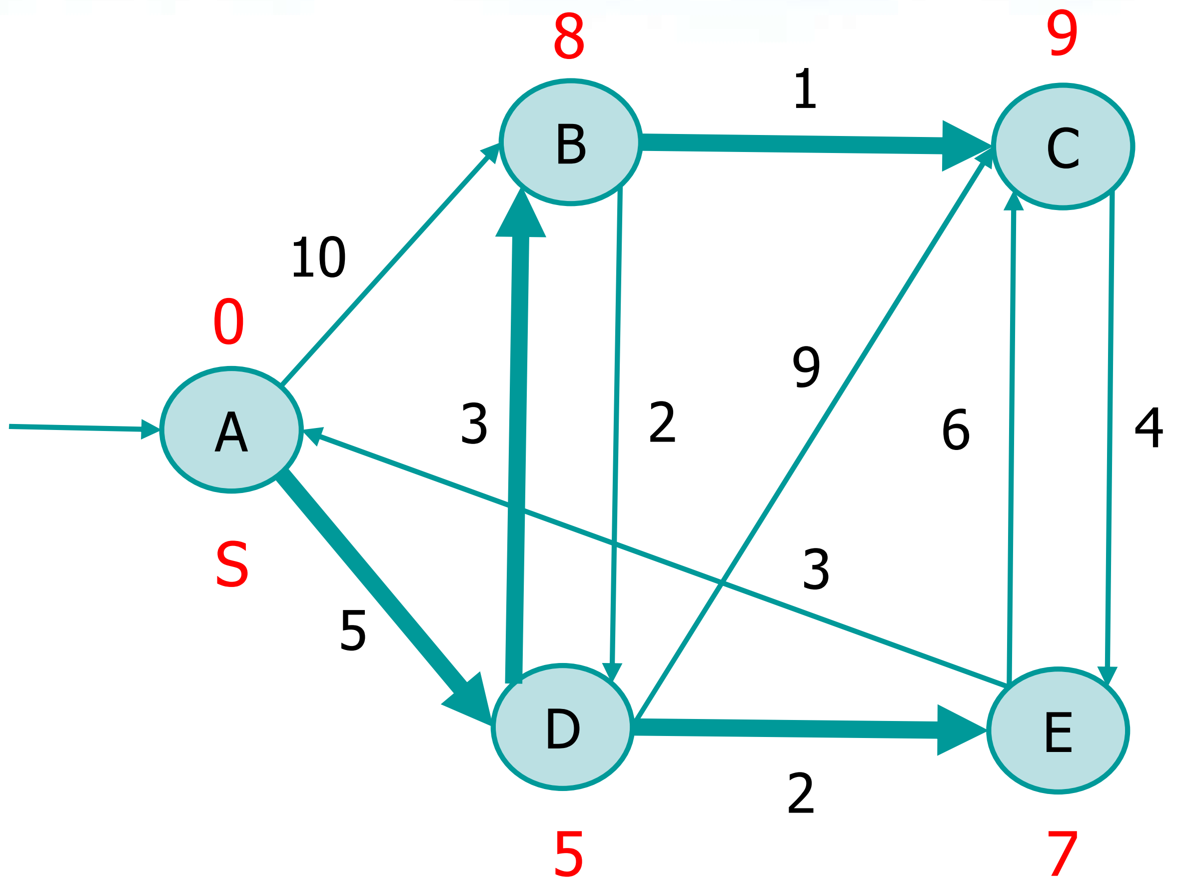
Insert it in S

Relax all adjancecy vertices

Example 1

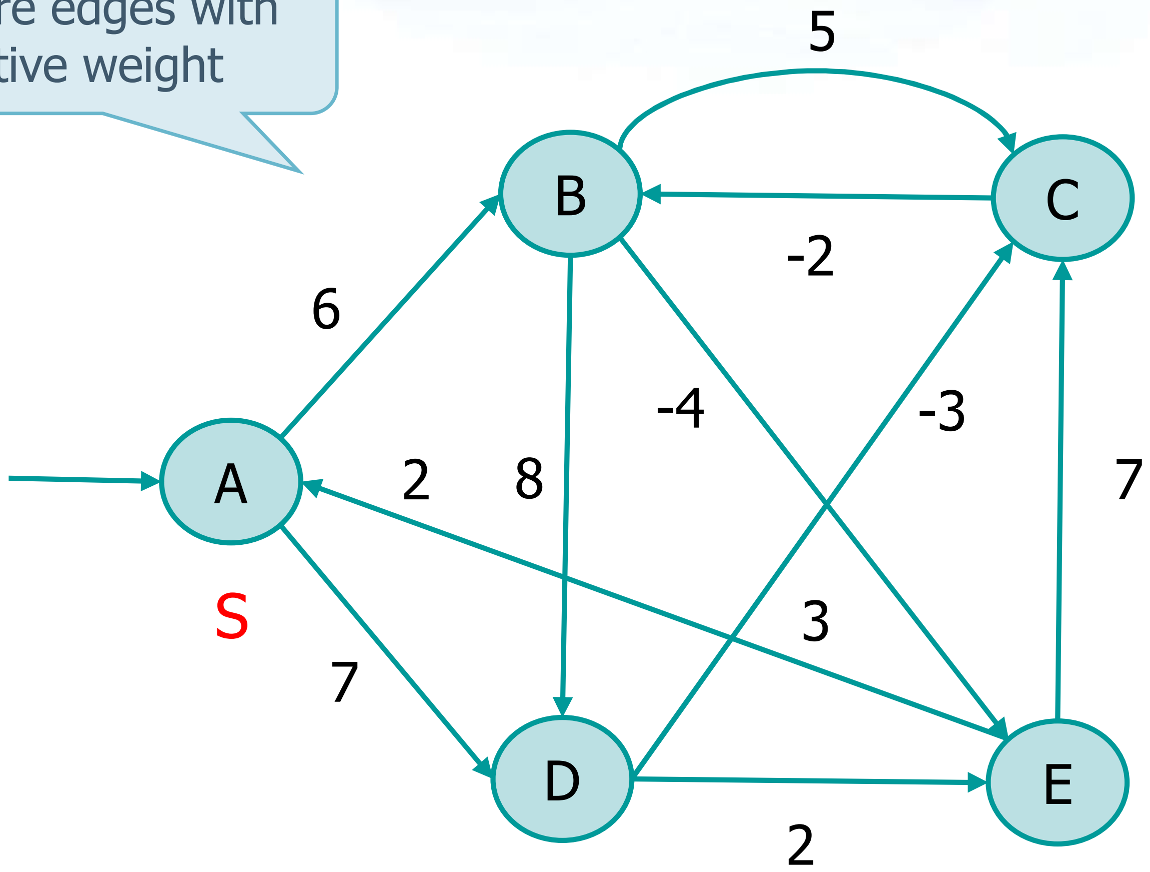


Example 1



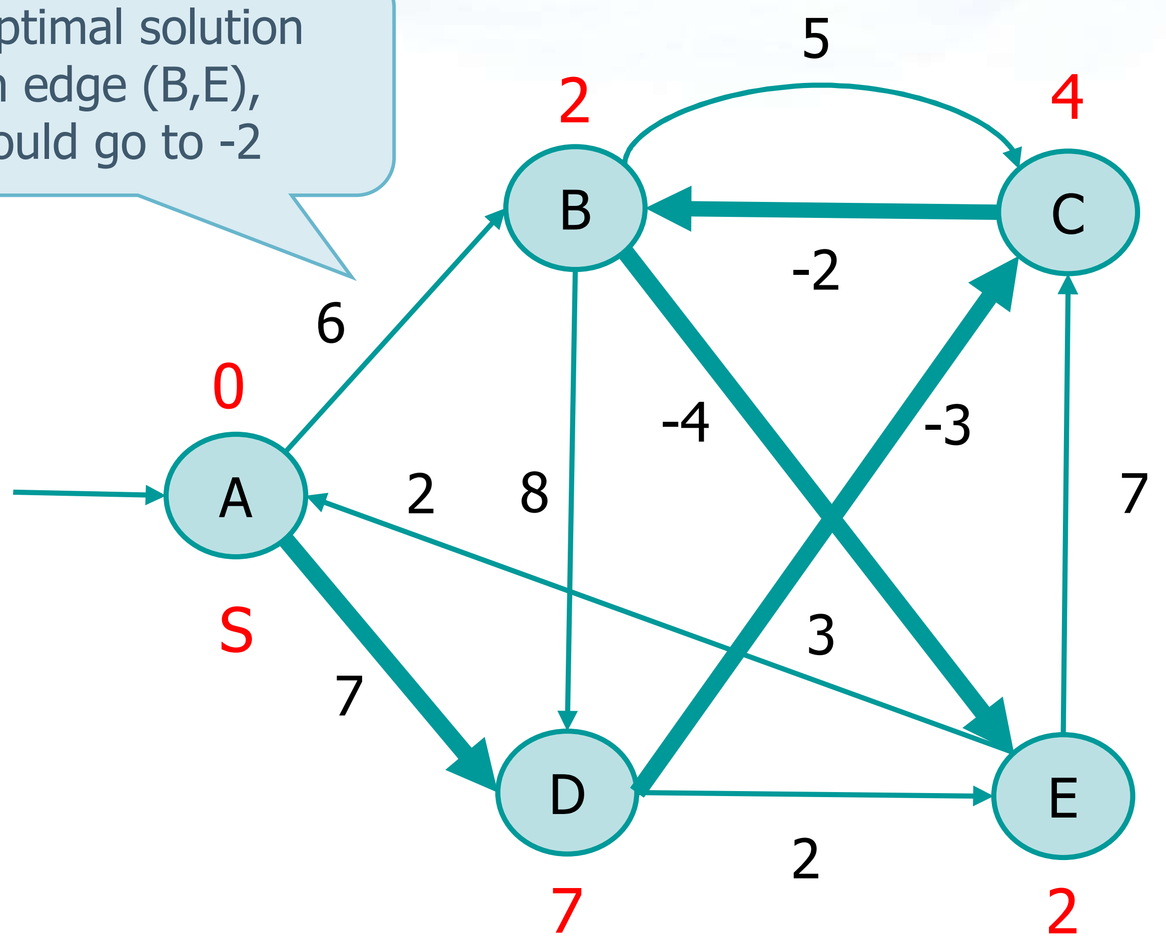
Example 2: Negative edges

There are edges with negative weight



Example 2: Negative edges

Non optimal solution
With edge (B,E),
E would go to -2



Implementation

Graph ADT
(same used for Kruskal's algorithm)

```
struct graph_s {
  vertex_t *g;
  int nv;
};
struct edge_s {
  int weight;
  int dst;
};
struct vertex_s {
  int id;
  int ne;
  int color;
  int dist;
  int scc;
  int disc_time;
  int endp_time;
  int pred;
  edge_t *edges;
};
```

Array of vertices of array of edges

Implementation

Client
(code extract)

```
g = graph_load (argv[1]);

fprintf (stdout, "Initial vertex? ");
scanf("%d", &i);

sssp_dijkstra (g, i);

fprintf (stdout, "Weights starting from vertex %d\n", i);
for (i=0; i<g->nv; i++) {
  if (g->g[i].dist != INT_MAX) {
    fprintf (stdout, "Node %d: %d (%d)\n",
      i, g->g[i].dist, g->g[i].pred);
  }
}

graph_dispose (g);
```

Implementation

```
void sssp_dijkstra (graph_t *g, int i) {
  int j, k;
  g->g[i].dist = 0;
  while (i >= 0) {
    g->g[i].color = GREY;
    for (k=0; k<g->g[i].ne; k++) {
      j = g->g[i].edges[k].dst;
      if (g->g[j].color == WHITE) {
        if (g->g[i].dist+g->g[i].edges[k].weight < g->g[j].dist) {
          g->g[j].dist = g->g[i].dist + g->g[i].edges[k].weight;
          g->g[j].pred = i;
        }
      }
    }
    g->g[i].color = BLACK;
    i = graph_min (g);
  }
}
```

For each outgoing vertex

Relax the connected nodes

Move to next vertex

Implementation

Simplification:
Instead of a priority queue there is an array with linear searches of the maximum

```
int graph_min (graph_t *g) {
  int i, pos=-1, min=INT_MAX;

  for (i=0; i<g->nv; i++) {
    if (g->g[i].color==WHITE && g->g[i].dist<min) {
      min = g->g[i].dist;
      pos = i;
    }
  }

  return pos;
}
```

Complexity

Pseudo-code

1:45

```
sssp_Dijkstra (G, w, s)
  initialize_single_source (G, s)
  S = Ø
  Q = V
  while Q ≠ Ø
    u = extract_min (Q)
    S = S U {u}
    for each vertex v ∈ adjacency list of u
      relax (u, v, w)
```

$O(|V|)$

Executed $|V|$ times

$O(log_2|V|) \rightarrow O(|V| \cdot log_2|V|)$

Overall $O(|E|)$

$O(log_2|V|) \rightarrow O(|E| \cdot log_2|V|)$ due to PQ change

Overall running time complexity $T(n) = O((|V| + |E|) \cdot log_2|V|)$

Complexity

- In general

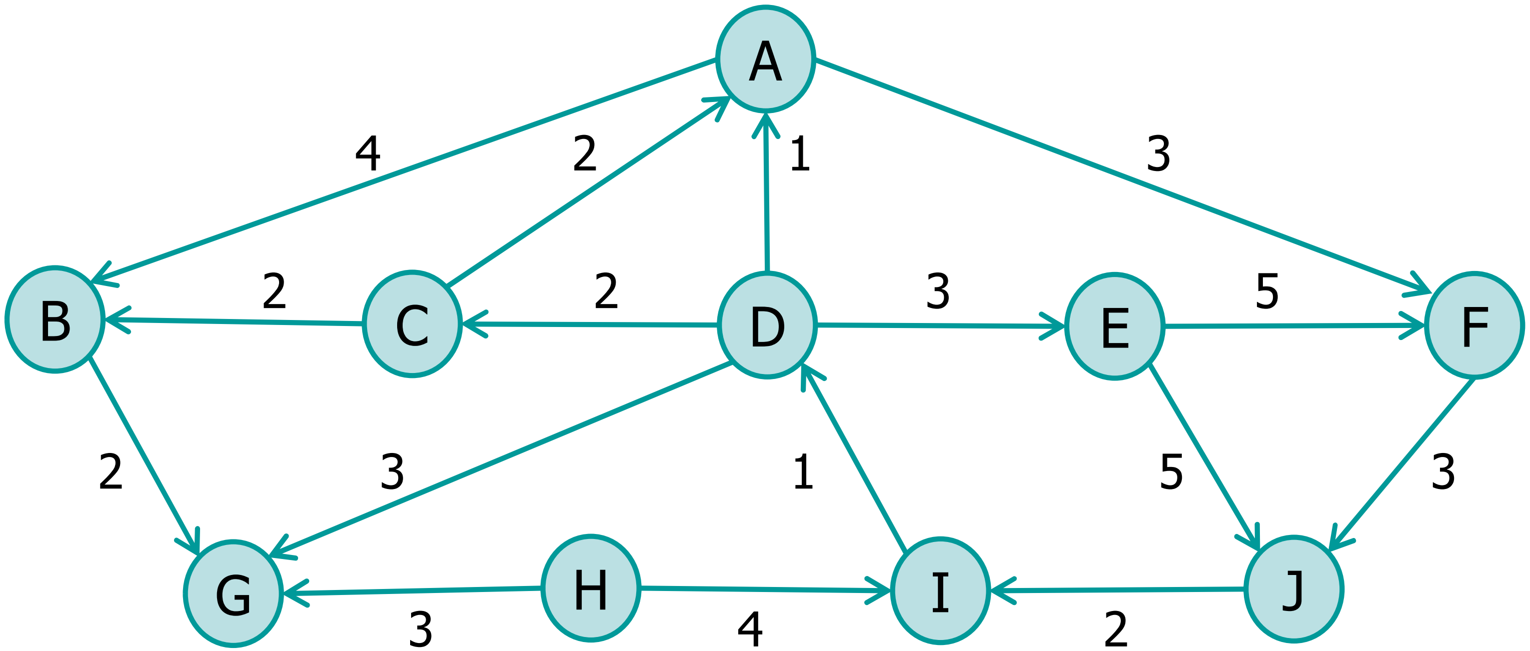
$$T(n) = O((|V| + |E|) \cdot log_2|V|)$$

  - That, if all vertice are reachable from the source, can be reduced to

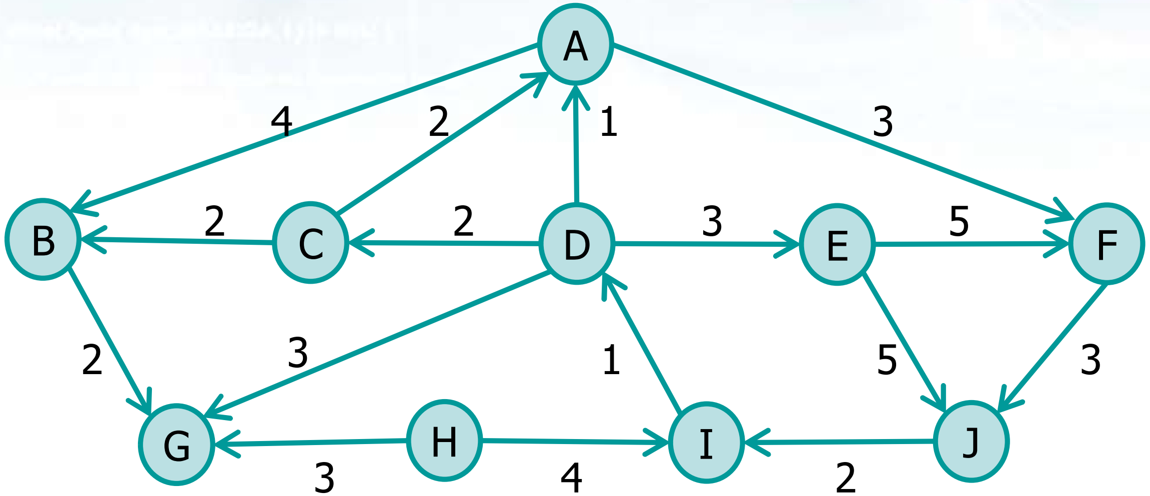
$$T(n) = O(|E| \cdot log_2|V|)$$

Exercise

- Given the following graph apply Dijkstra's algorithm starting from vertex A



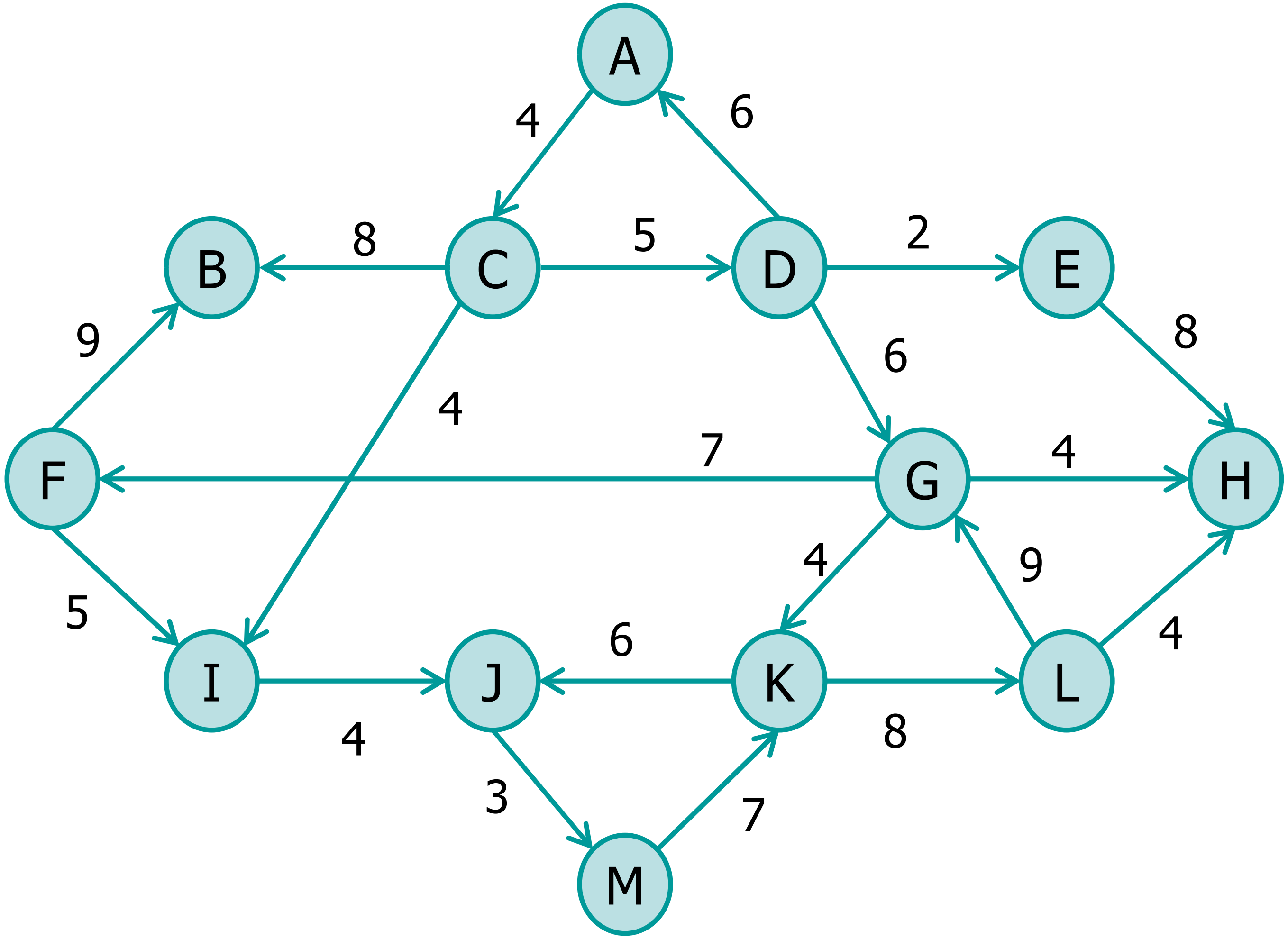
Solution



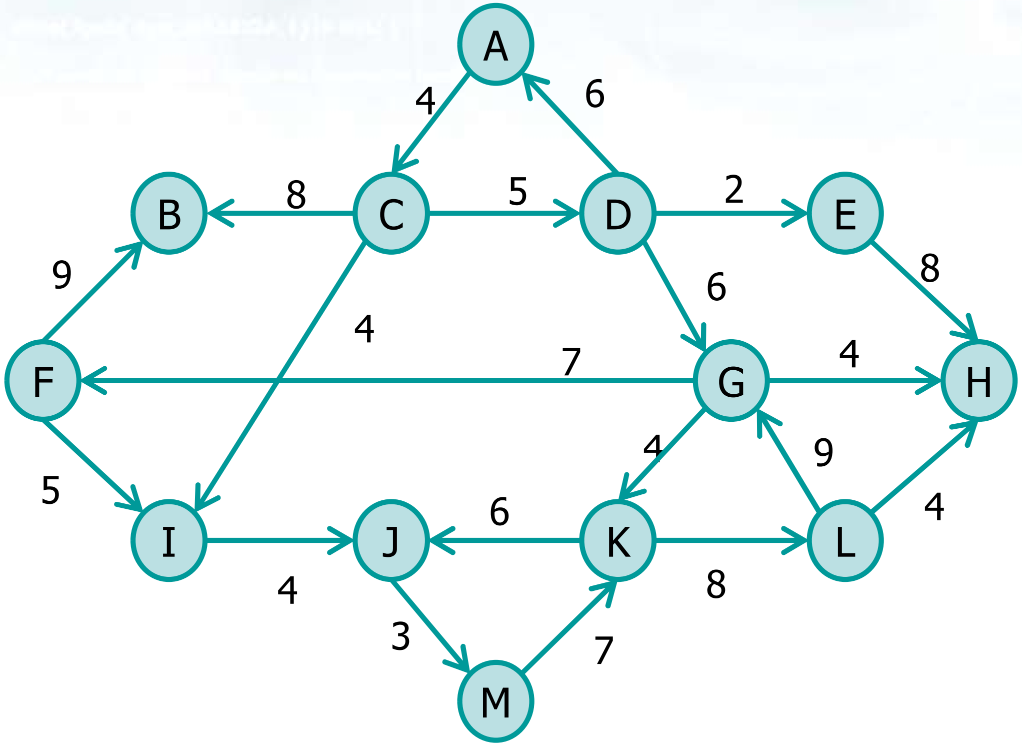

```
Shortest paths from vertex [ 0] A
Node [ 0] A:   0 (pred=-1)
Node [ 5] F:   3 (pred= 0)
Node [ 1] B:   4 (pred= 0)
Node [ 6] G:   6 (pred= 1)
Node [ 9] J:   6 (pred= 5)
Node [ 8] I:   8 (pred= 9)
Node [ 3] D:   9 (pred= 8)
Node [ 2] C:  11 (pred= 3)
Node [ 4] E:  12 (pred= 3)
Node [ 7] H: infty
```

Exercise

- Given the following graph apply Dijkstra's algorithm starting from vertex A



Solution




```
Shortest paths from vertex [ 0] A
Node [ 0] A:  0 (pred=-1)
Node [ 2] C:  4 (pred= 0)
Node [ 8] I:  8 (pred= 2)
Node [ 3] D:  9 (pred= 2)
Node [ 4] E: 11 (pred= 3)
Node [ 1] B: 12 (pred= 2)
Node [ 9] J: 12 (pred= 8)
Node [ 6] G: 15 (pred= 3)
Node [12] M: 15 (pred= 9)
Node [ 7] H: 19 (pred= 4)
Node [10] K: 19 (pred= 6)
Node [ 5] F: 22 (pred= 6)
Node [11] L: 27 (pred=10)
```