Graphs

Single Source Shortest Paths

Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Problem definition

- Example
  - Given a road map on which the distance between each pair of adjacent intersactions is marked
  - How is it possible to determine the shortest route?
  - One possibility is to
    - Enumerate all routes, add the distance on each route, disallowing routes with cycles
    - Select the shortes routes
  - This implies examining an enourmous number of possibilities
- A better solution implies solving the so called Single-Source Shortest Path problem

Shortest Paths

- Given a graph G = (V, E)
  - Directed
  - Weighted
    - With a positive real-value weight function w: E→R
  - With a weight w(p) over a path
    - $p = <v_0, v_1, \ldots, v_k>$

  is equal to
    - $w(p) = \Sigma_{i=1}^{k} w(v_{i-1}, v_i)$

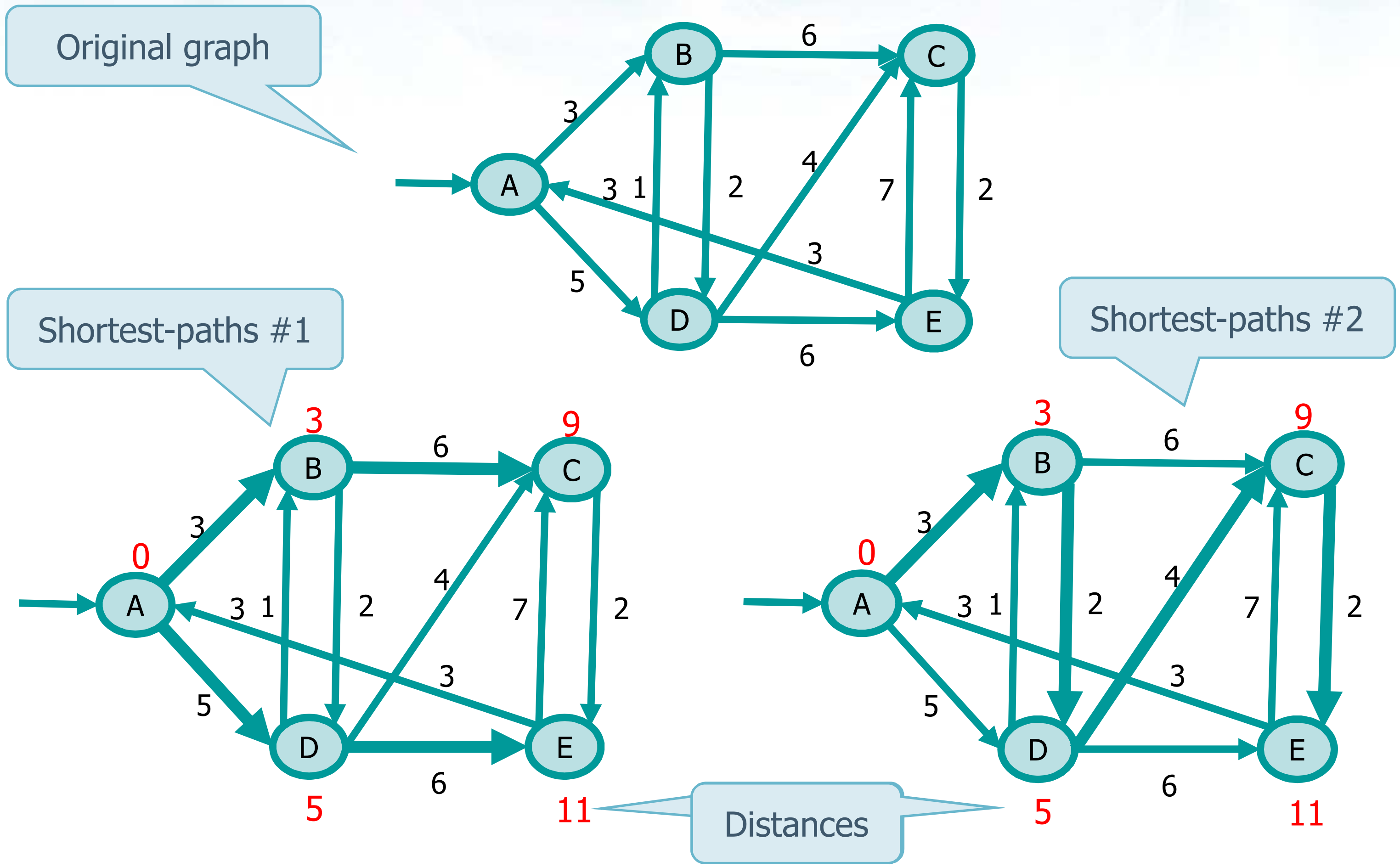
Shortest Paths

- We define the shortest path weight $\delta(u,v)$ from u to v as
  - $$\delta(u,v) = \begin{cases} \min\{w(p)\} & \text{if } \exists\, u \rightarrow_p v \\ \infty & \text{otherwise} \end{cases}$$
- A shortest path from u to v is any path p with weigth
  - $w(p) = \delta(u,v)$

Variants

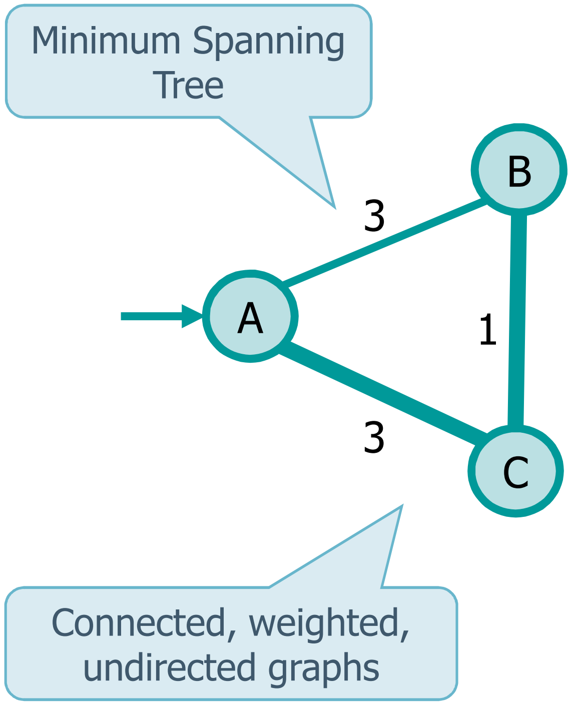
- Shortest path problems
  - Single-source shortest-paths
    - Minimum path and its weight from s to all other vertices v
      - **Dijkstra**'s algorithm
      - **Bellman-Ford**'s algorithm
  - Notice that with **unweighted** graphs a simple **BFS** (Breadth-First Search) solves the problem

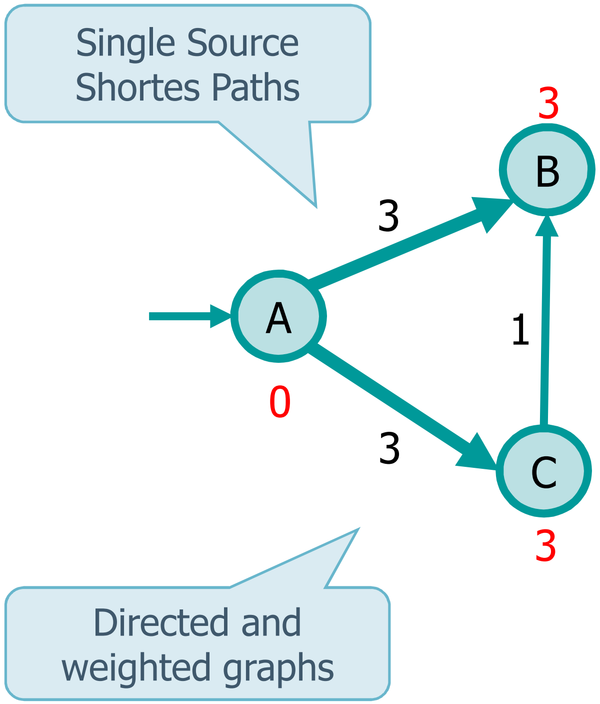
Example


Original graph
A
B
C
D
E
3
6
5
1
2
4
7
2
3
3
6
Shortest-paths #1
0
3
9
5
11
Shortest-paths #2
0
3
9
5
11
Distances

Observation

- SSSPs and MSTs are different





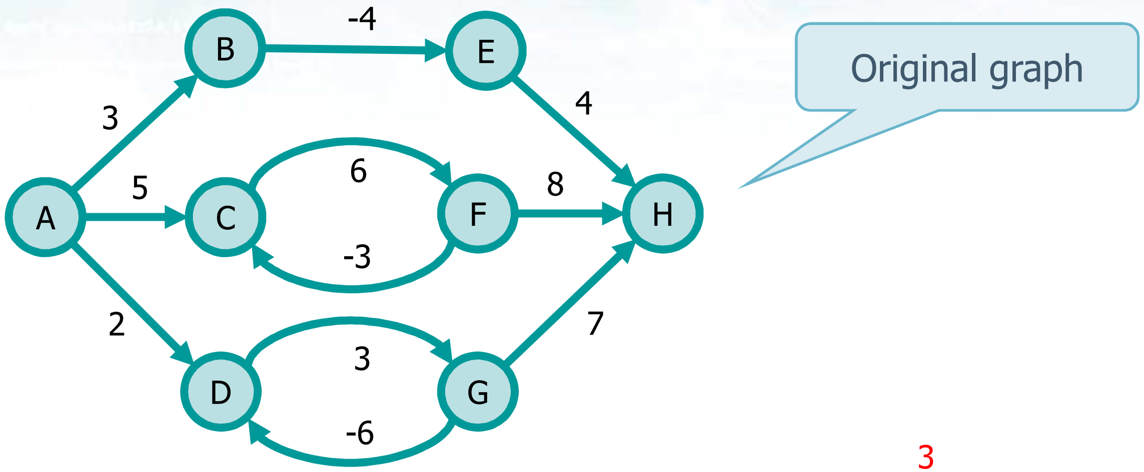
Variants

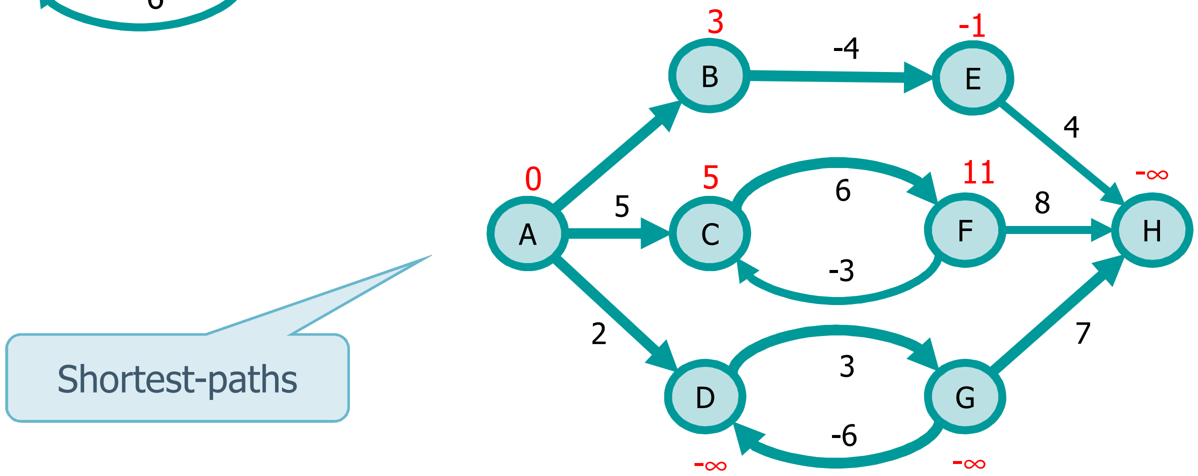
- Single-destination shortest-paths
  - Find the shortest path to a given destination
  - Use the reverse graph
- Single-pair shortest-paths
  - Find a shortest path from v_1 to v_2 given vertices v_1 to v_2
  - Soved when the SSSP is solved
  - All alternative solutions have the same worst-case asymptotic running time
- All-pairs shortest-path
  - Find a shortest-path for every vertex pair
  - Can be solved running SSSP from each vertex
  - Can be solved faster

Negative Weight Edges

- If there are edges with negative weight but there are no cycles with negative weight
  - Dijkstra's algorithm
    - Optimum solution not guaranted
  - Bellman-Ford's algorithm
    - Optimum solution guaranted
- It there are cycle with negative weight
  - The problem is not defined (there is no solution)
  - Dijkstra's algorithm
    - Meaningless result
  - Bellman-Ford's algorithm
    - Find cycles with negative weights

Example


Original graph
B
E
-4
3
4
A
5
C
6
F
8
H
-3
2
D
3
G
7
-6


Shortest-paths
3
-1
B
-4
E
4
0
5
11
-∞
A
5
C
6
F
8
H
-3
2
D
3
G
7
-6
-∞
-∞

Representing Shortest Paths

- Often we wish to compute vertices on shorterst path, not only weights
  - A few representations are possible
- Array of predecessors v.pred
  - $\forall v \in V \quad v.pred = \begin{cases} parent(v) \text{ if } \exists \\ NULL \text{ otherwise} \end{cases}$
- Predecessor's sub-graph
  - $G_{pred}=(V_{pred}, E_{pred})$, where
    - $V_{pred} = \{v \in V: v.pred \neq NULL\} \cup \{s\}$
    - $E_{pred} = \{(v.pred, v) \in E : v \in V_{pred} - \{s\}\}$

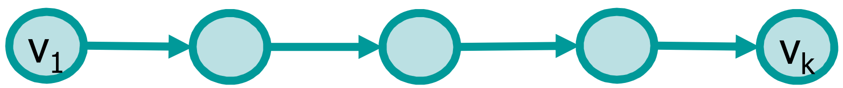
Attribute pred (predecessor) for each vertex

Representing Shortest Paths

- Shortest-Paths Tree
  - G′ = (V′, E′)
    - Where V′ $\subseteq$ V && E′ $\subseteq$ E
    - V′ is the set of vertices reachable from s
    - S is the tree root
    - $\forall$v$\in$V′ the unique simple path from s to v in G′ is a minimum weight from s to v in G

Theoretical Background

- Lemma
  - Sub-paths of shortest paths are shortest paths
  - G = (V, E)
    - Directed, weighted $w: E \rightarrow R$
  - $P = <v_1, v_2, \ldots, v_k>$
    - Is a shortest path from v_1 to v_k
  - $\forall i, j \; 1 \leq i \leq j \leq k, \; p_{ij} = <v_i, v_{i+1}, \ldots, v_j>$
    - Sub-path of p from v_i to v_j
  - The p_{ij} is a shortest path from v_i to v_j



Theoretical Background

- Corollary
  - G = (V, E)
    - Directed, weighted w: E→R
  - A shortest path p from s to v may be decomposed into
    - A shortest sub-path from s to u
    - An edge (u,v)
  - Then
    - $\delta(s,v) = \delta(s,u) + w(u,v)$

Theoretical Background

- Lemma
  - $G = (V, E)$
    - Directed, weighted $w: E \rightarrow R$
  - $\forall (u,v) \in E$
    - $\delta(s,v) \leq \delta(s,u) + w(u,v)$
  - A shortest path from s to v cannot have a weight larger than the path formed by a shortest path from s to u and an edge (u, v)

Relaxation

- The algorithms we are going to anayze use the technique of **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

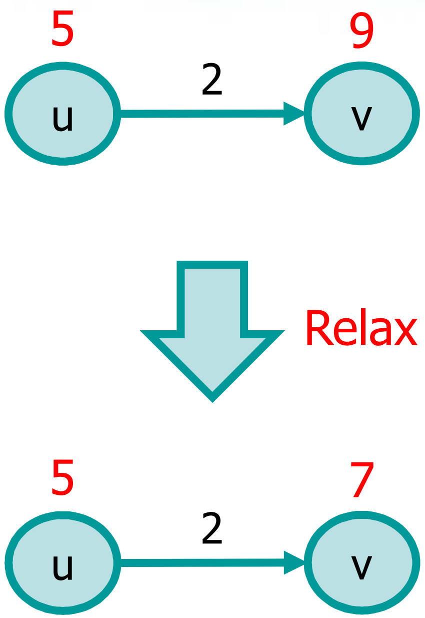
v.dist
= shortest path estimate =
upper bound on the weight of
a shortest path from s to v

Relaxation

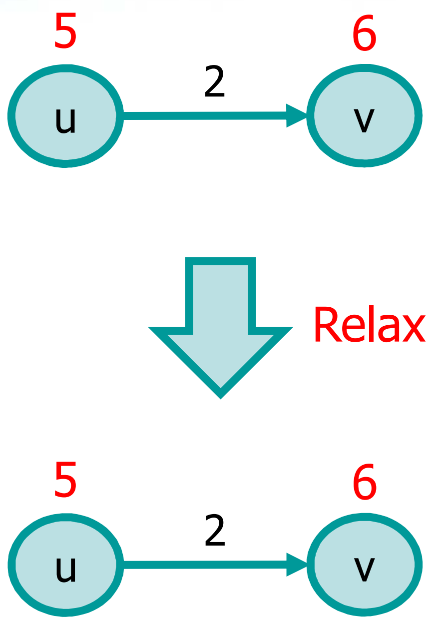
- Relaxation
  - Update v.dist and v.pred by testing whether it is possibile to improve the shortest path to v found so far by going through the edge e = (u,v), where w(u,v) is the weigth of the edge

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

Example



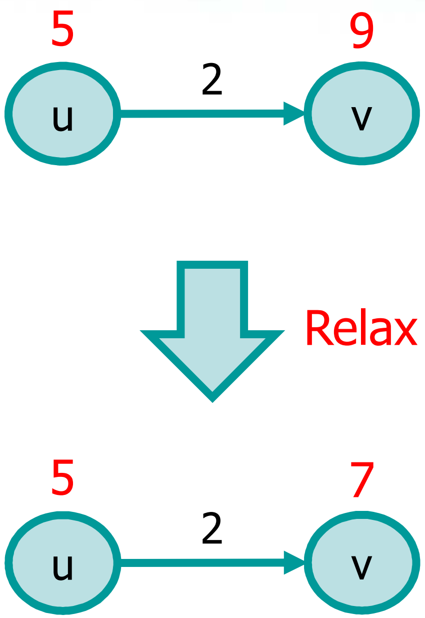

```
v.dist = 9
u.dist = 5
w(u,v) = 2
v.dist >
  u.dist + w(u,v)
```



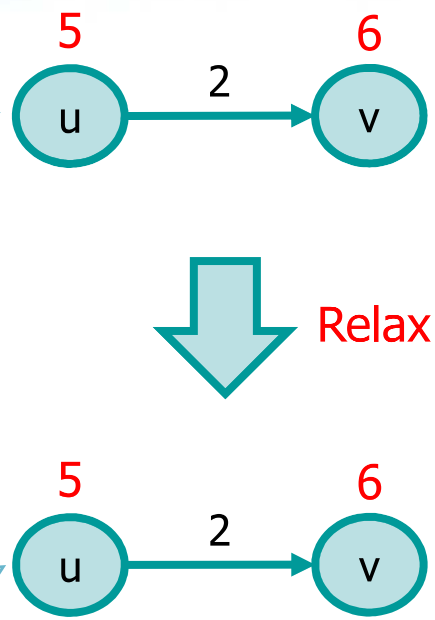
Shortest path from s to v = shortest path from s to u + edge (u,v)

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

Example




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



Relaxation has no effect

```
v.dist = unchanged
       = 6
v.pred = unchanged
```

Properties

- Lemma
  - Given G=(V,E)
  - Directed, weighted w: $E \rightarrow R$, with e = (u,v) $\in$ E
- After relaxing e = (u,v) we have
  - v.dist $\leq$ u.dist + w (u, v)
- That is, after relaxing e, v.d cannot increase
  - Either v.dist is unchanged (relaxation with no effect)
  - Or v.dist is decreased (effective relaxation)

Properties

- Lemma
  - Given G=(V,E), directed, weighted w: E→R, with source s ∈ V
  - After a proper initialization of v.dist and v.pred
- $\forall$ v ∈ V v.dist ≥ δ(s, v)
  - For all relaxation steps on the edges
  - When v.dist = δ(s,v), then v.dist does not change any more

Properties

- Lemma
  - Given G=(V,E) directed, weighted w: E→R, with source s ∈ V
  - After a proper initialization of v.dist and v.pred
- The shortest path from s to v is made-up of
  - Path from s to u
  - Edge e = (u, v)
- Application of relaxation on e=(u, v)
  - If before relaxation u.dist = δ(s, u)
  - After relaxation v.dist = δ(s, v)

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
  - Stop when Q is 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 = ϕ
  Q = V
  while Q ≠ ϕ
    u = extract_min (Q)
    S = S ∪ {u}
    for each vertex v ∈ adjacency list of u
      relax (u, v, w)
```

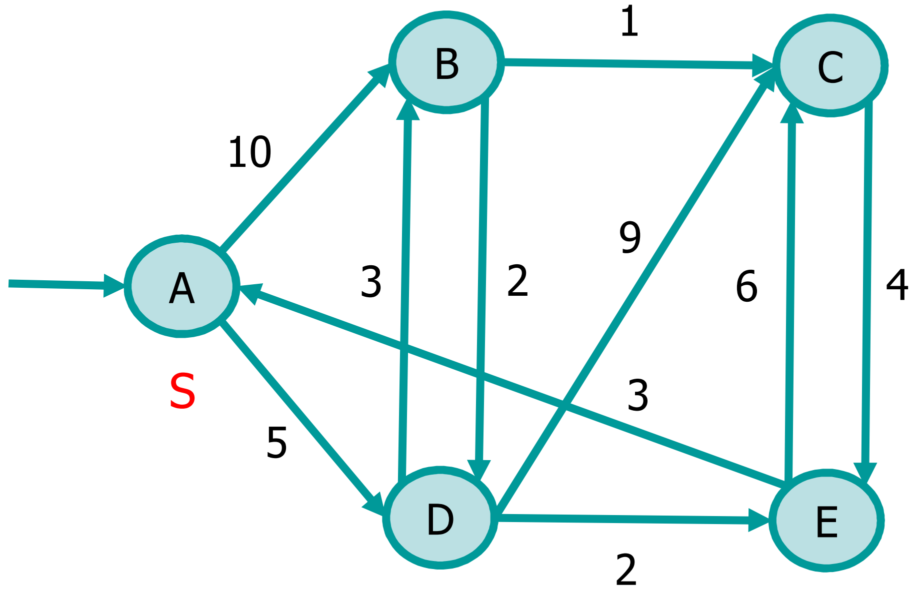
For all vertices starting from s

Extract vertex with minimum distance

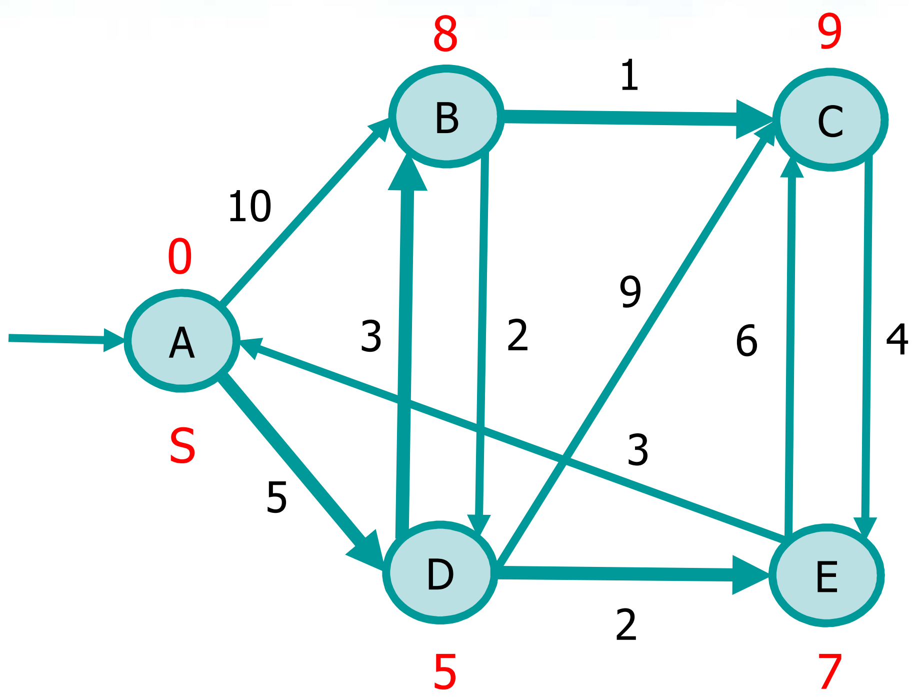
Insert if in S

Relax all adjancency vertices

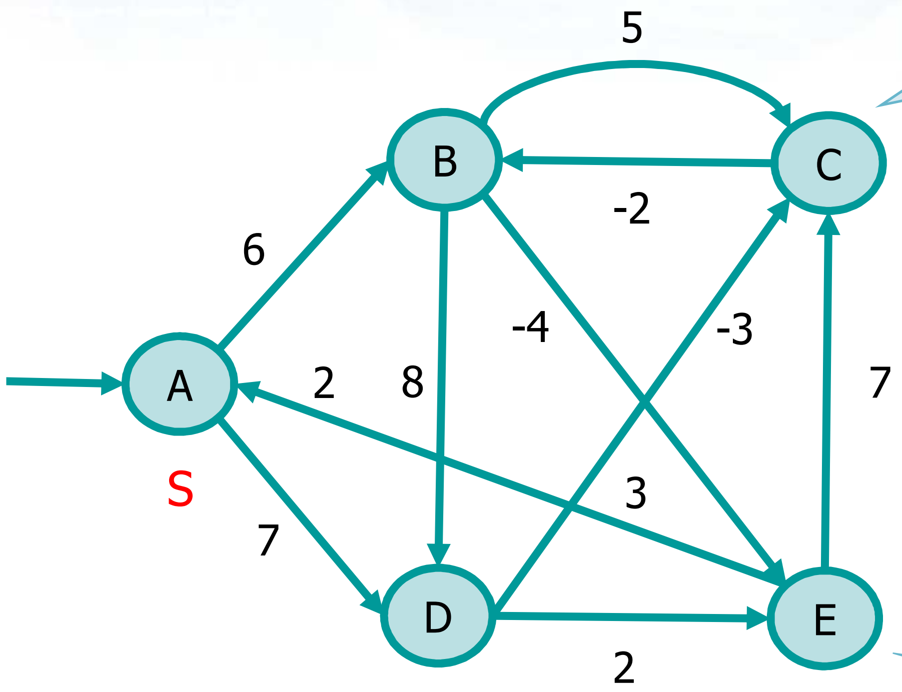
Example 1



Example 1



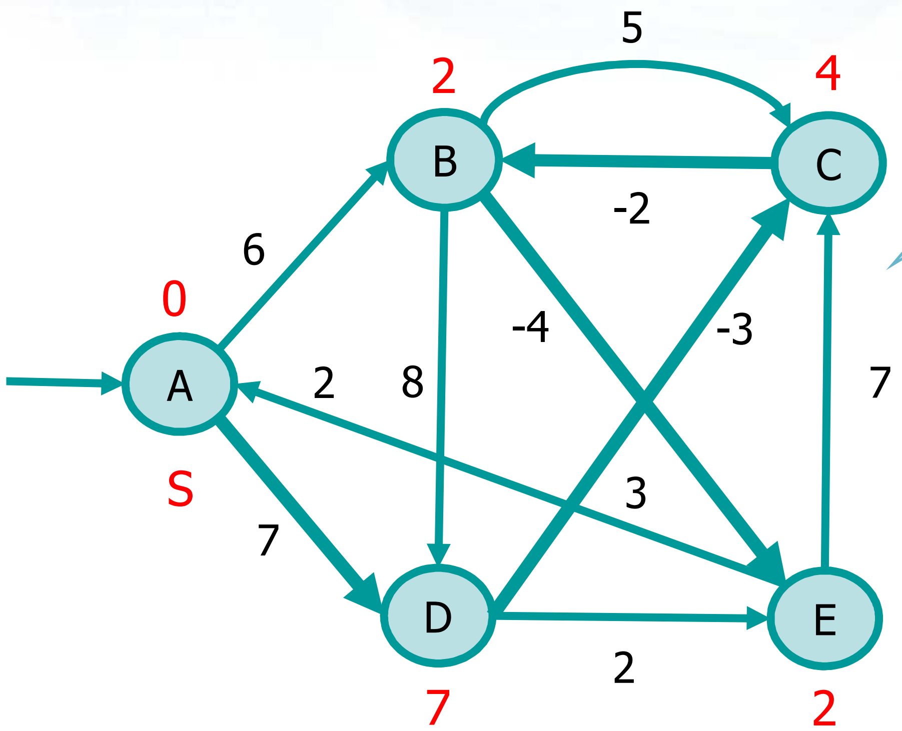
Example 2: Negative edges



There are edges with negative weight

There are no cycles with negative weight

Example 2: Negative edges



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

Implementation

```
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;
};
```

Graph ADT
(same used for Kruskal's algorithm)

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

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

O(|V|)

Executed |V| times

O (lg |V|) → **O(|V| log|V|)**

Overall O(|E|)

O(lg |V|) → **O(|E|log|V|)** due to PQ change

Overall running time complexity T(n) = O((|V|+|E|) · lg |V|)

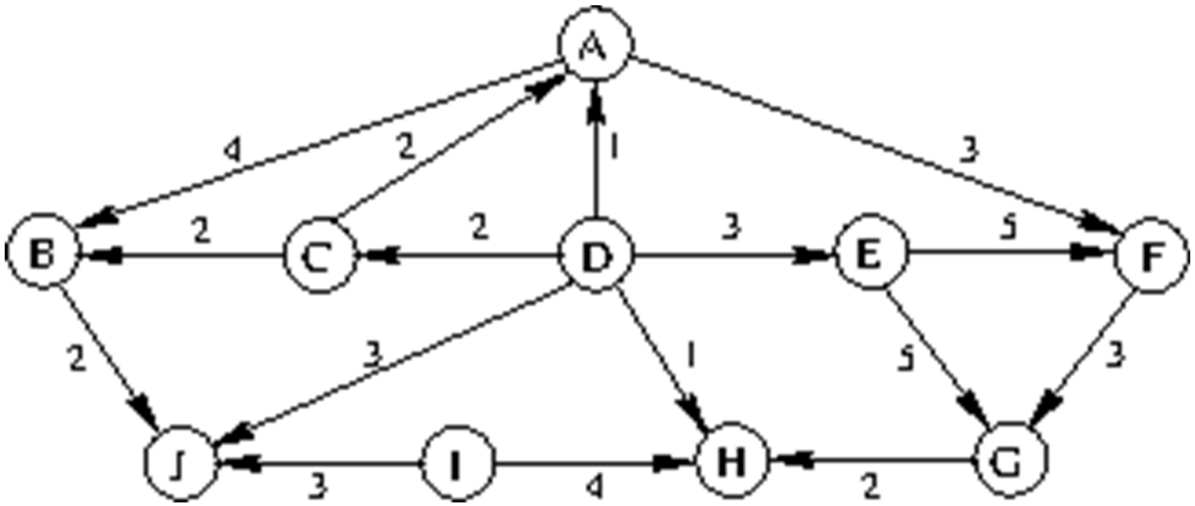
Complexity

- In general
  - $T(n) = O((|V|+|E|) \cdot \lg |V|)$
- This can be reduced to
  - $T(n) = O(|E| \cdot \lg |V|)$

  if all vertices are reachable from the source s

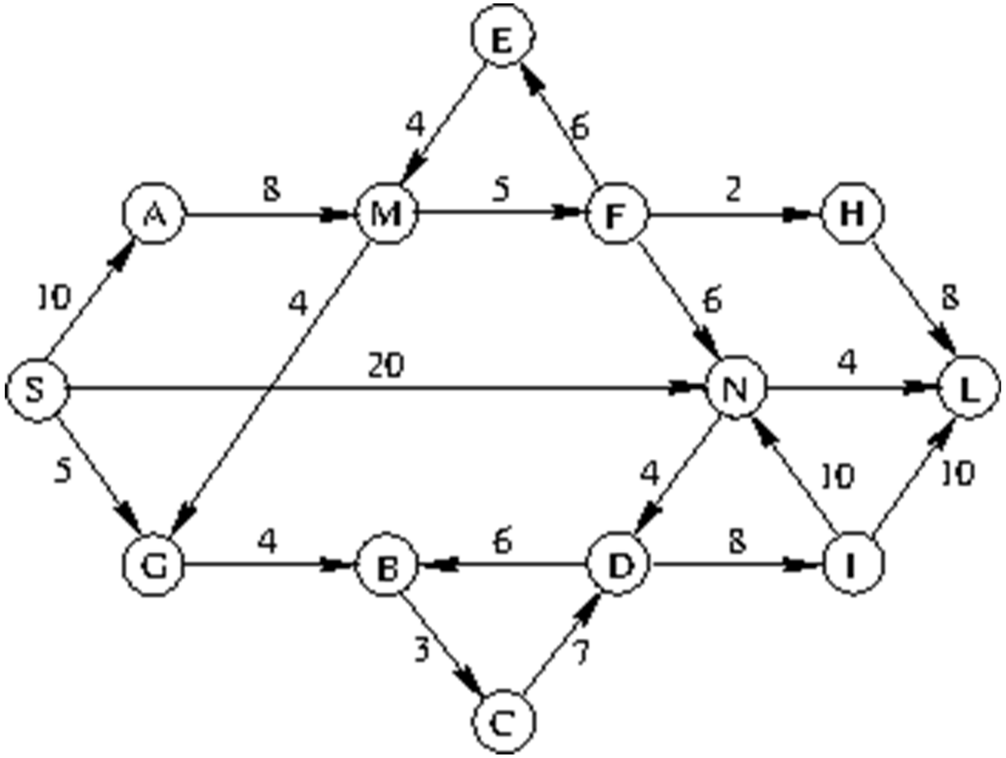
Exercise

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



Exercise

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



Bellman-Ford's Algorithm

- Bellman-Ford may run on graphs
  - With negative weight edges
  - If there is a cycle with negative weight it detects it
  - It applies relaxation more than once for all edges
  - |V|-1 step of relaxation on all edges
  - At the i-th relaxation step either
    - It decreases at least one estimate

    or

    - It has already found an optimal solution and it can stop returning an optimum solution

Pseudo-code

Pseudo-code

```
sssp_Bellman_Ford (G, w, s)
  initialize_single_source (G, s)
  for i = 1 to |V| – 1
    for each edge (u, v) ∈ E
      relax (u, v, w)
  for each edge (u, v) ∈ E
    if ( v.dist > (u.d + w(u, v)) )
      return FALSE
return TRUE
```

Iterates |V|-1 times

Relaxes all edges

Checks for negative weight cycles

Returns FALSE if a negative weight cycle is detected

Returns TRUE otherwise

Pseudo-code

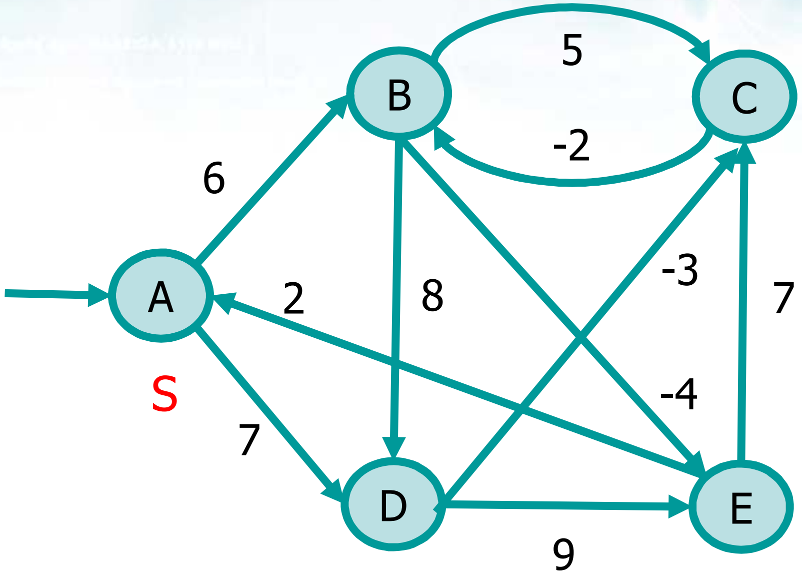
Pseudo-code

After |V|-1 iterations, all vertices reachable from s have been reached with the shortest path

```
sssp_Bellman_Ford (G, w, s)
  initialize_single_source (G, s)
  for i = 1 to |V| – 1
    for each edge (u, v) ∈ E
      relax (u, v, w)
  for each edge (u, v) ∈ E
    if ( v.dist > (u.d + w(u, v)) )
      return FALSE
return TRUE
```

Proof
With |V| vertices the longest simple path includes |V| vertices, that is |V|-1 edges. All of them are relaxed in |V|-1 iterations. Thus, all paths are the shortest ones for the property of relaxation

Example 1

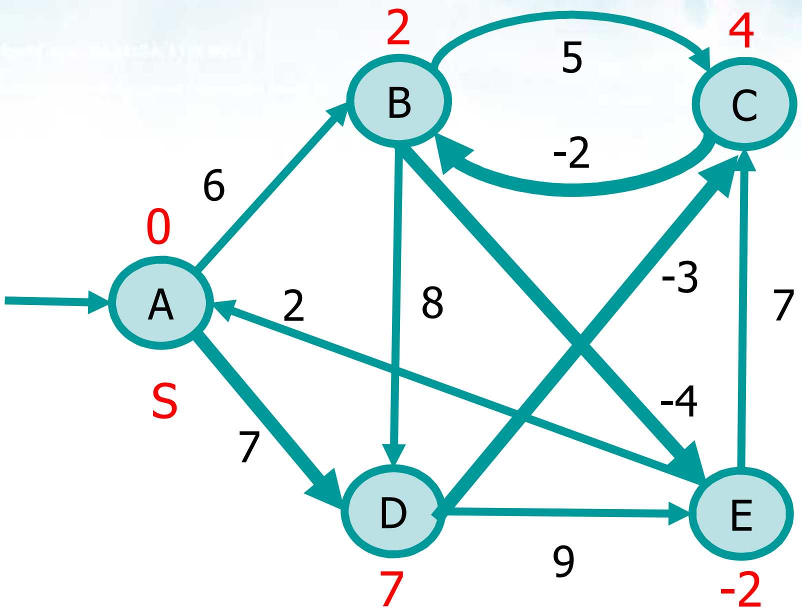


Lessicographic order of the edges
(A, B)
(A, D)
(B, C)
(B, D)
(B, E)
(C, B)
(D, C)
(D, E)
(E, A)
(E, C)

	#0	#1	#2	#3	#4
A					
B					
C					
D					
E					

Step #
(5 vertices → 4 iterations)

Example 1

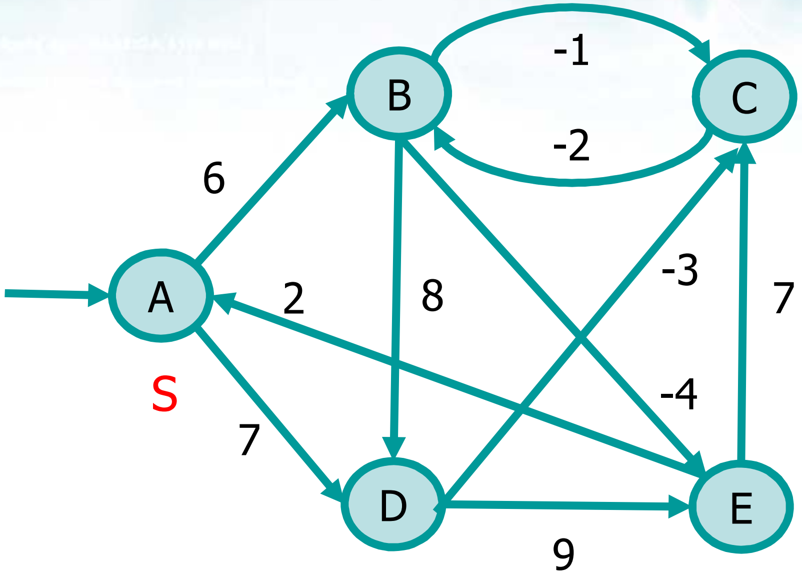


Lessicographic order of the edges
(A, B)
(A, D)
(B, C)
(B, D)
(B, E)
(C, B)
(D, C)
(D, E)
(E, A)
(E, C)

	#0	#1	#2	#3	#4
A	0	0	0	0	0
B	∞	6	2	2	2
C	∞	11→4	4	4	4
D	∞	7	7	7	7
E	∞	2	2	-2	-2

Step #
(5 vertices → 4 iterations)

Example 1

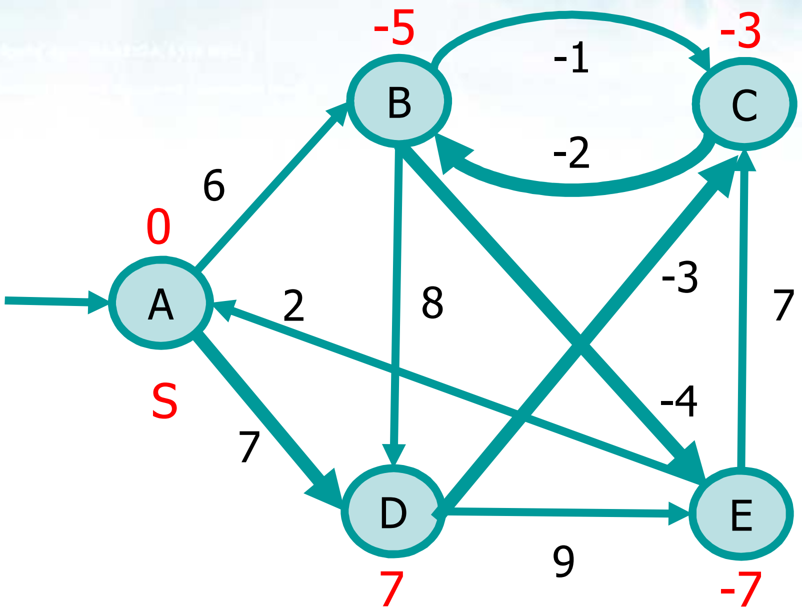


Lessicographic order of the edges
(A, B)
(A, D)
(B, C)
(B, D)
(B, E)
(C, B)
(D, C)
(D, E)
(E, A)
(E, C)

	#0	#1	#2	#3	#4
A					
B					
C					
D					
E					

Step #
(5 vertices → 4 iterations)

Example 1

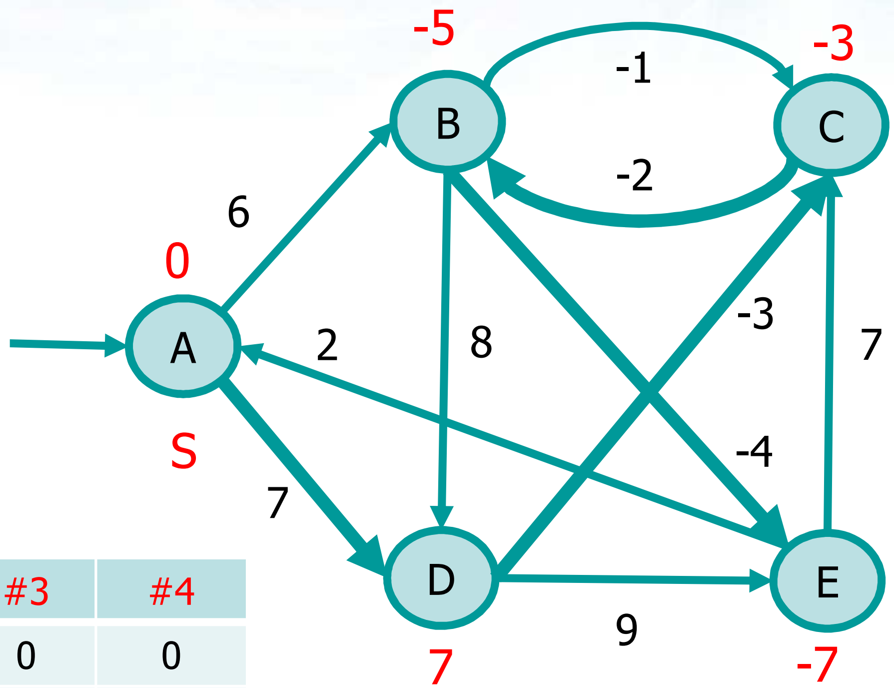


	#0	#1	#2	#3	#4
A	0	0	0	0	0
B	∞	6→3	1	-2	-5
C	∞	5→4	3	0	-3
D	∞	7	7	7	7
E	∞	2	-1	-3	-7

Lessicographic order of the edges
(A, B)
(A, D)
(B, C)
(B, D)
(B, E)
(C, B)
(D, C)
(D, E)
(E, A)
(E, C)

At the next iteration, edges BC and CB would make B and C reachable in -8 and -6

Example 2: Negative cycles



Step #
(5 vertices → 4 iterations)

	#0	#1	#2	#3	#4
A	0	0	0	0	0
B	∞	6→3	1	-2	-5
C	∞	5→4	3	0	-3
D	∞	7	7	7	7
E	∞	2	-1	-3	-7

At the next iteration, edges BC and CB would make B and C reachable in -8 and -6

Implementation

Graph ADT
(same used for Prim's algorithm)

```
typedef struct graph_s graph_t;
typedef struct vertex_s vertex_t;
typedef struct edge_s edge_t;


struct graph_s {
  vertex_t *g;
  int nv;
};
```

```
struct edge_s {
  int weight;
  int dst;
  edge_t *next;
};

struct vertex_s {
  int id;
  int color;
  int dist;
  int disc_time;
  int endp_time;
  int pred;
  int scc;
  edge_t *head;
};
```

Array of vertex of lists of edges

Implementation

Client
(code extract)

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

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

if (sssp_bellman_ford (g, i) != 0) {
  fprintf (stdout, "Negative weight loop detected!\n");
} else {
  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

```
int sssp_bellman_ford (graph_t *g, int i) {
  edge_t *e;
  int k, stop=0;
  g->g[i].dist = 0;
  for (k=0; k<g->nv-1 && !stop; k++){
    stop = 1;
    for (i=0; i<g->nv; i++) {
      if (g->g[i].dist != INT_MAX) {
        e = g->g[i].head;
        while (e != NULL) {
          if (g->g[i].dist+e->weight < g->g[e->dst].dist) {
            g->g[e->dst].dist = g->g[i].dist+e->weight;
            g->g[e->dst].pred = i;
            stop = 0;
          }
          e = e->next;
        }
      }
    }
  }
```

For each edge in the graph

Relax the connected nodes

Move to next edge

Implementation

```
  if (!stop) {
    for (i=0; i<g->nv; i++) {
      if (g->g[i].dist != INT_MAX) {
         e = g->g[i].head;
         while (e != NULL) {
           if (g->g[i].dist+e->weight < g->g[e->dst].dist) {
             return 1;
           }
           e = e->next;
         }
      }
    }
  }

  return 0;
}
```

Verify negative weight loops

Relax the connected nodes

Complexity

Pseudo-code

```
sssp_Bellman_Ford (G, w, s)
  initialize_single_source (G, s)
  for i = 1 to |V| - 1
    for each edge (u, v) ∈ E
      relax (u, v, w)
  for each edge (u, v) ∈ E
    if ( v.dist > (u.d + w(u, v)) )
      return FALSE
return TRUE
```

O (|V|)

Executed |V|-1 times

Executed |E| times

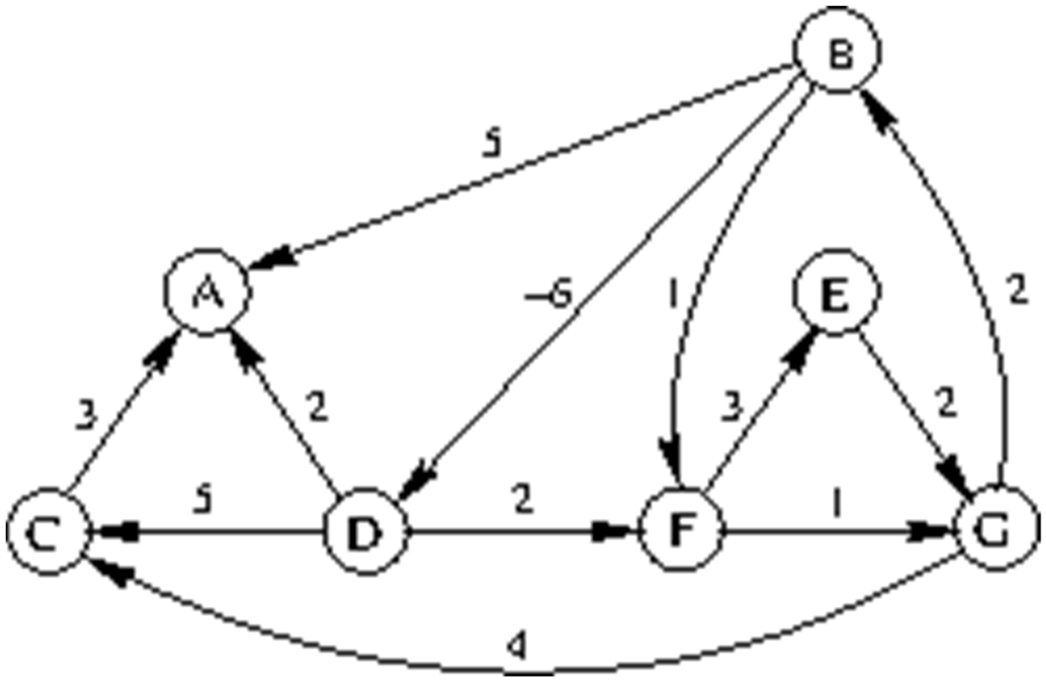
O(1) → **O(|E|·|V|)**

Executed |E| times → **O(|E|)**

Overall running time complexity
T(n) = O(|V| · |E|)

Exercise

- Given the following graph apply Bellman-Ford's algorithm from vertex B



Exercise

- Given the following graph apply Bellman-Ford's algorithm from vertex A