GRAPHS

Algorithms and Data Structures 2
Exercise – 2023W

Martin Schobesberger, Markus Weninger, Markus Jäger,
Florian Beck, Achref Rihani

Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

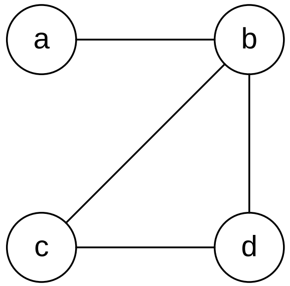
VERTICES AND EDGES

G = (V, E)

- **V... Set of vertices (or nodes)**
- **E... Set of edges**

Example:

- V = {a, b, c, d}
- E = {{a, b}, {b, d}, {c, d}, {c, b}}



Two vertices are **adjacent**, if they are connected by an edge.

An edge *e* connecting two vertices *x* and *y* is called **incident** to *x* and *y*.

MOTIVATION

Graphs are one of the most basic data structures. Many problems can be characterized by graphs, such as:

- **Electric power grid**
  - Nodes: power distributors, transformer stations, etc.
  - Edges: wires
  - No defined direction → undirected graph
  - Cycles are possible
- **Material flow in manufacturing companies**
  - Nodes: workstations
  - Edges: band-conveyors
  - Raw materials only flow in one direction → directed graph
  - Limited capacity of band-conveyors → weighted graph
  - No cycles
- **Social distance in a set of persons**
  - Nodes: Humans
  - Edges: Relations

DEFINITION / TERMS

Degree of a vertex:

- Number of vertices that are adjacent to it
  (which is not necessarily equal to the number of edges)

Path: Sequence of adjacent vertices

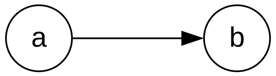
- simple: No vertex occurs more than once.
- cyclic: At least one vertex occurs more than once.

Cyclic graph:

- Contains at least one cyclic path (otherwise: **acyclic** graph)

Directed edge: Connection from *a* to *b*

- Directed graph: Contains only directed edges



Loop:

- Edge (v, v) for vertex v

DEFINITION / TERMS

Component: connected part of a graph

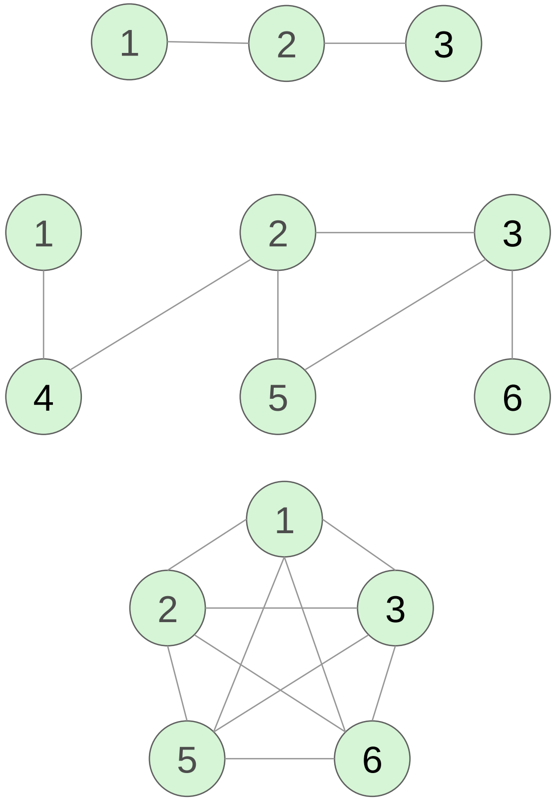
Connectivity (1)

- Two vertices are called connected if there is a path (i.e., a sequence of edges) between them.

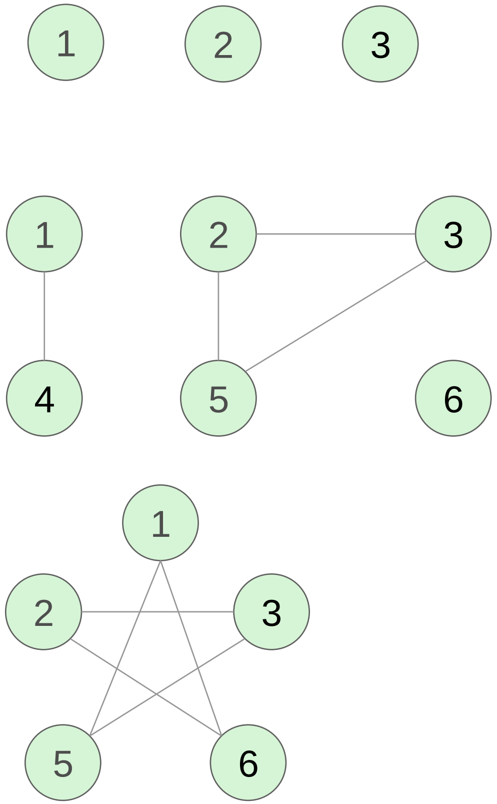
- Connected graph: Each pair of vertices in the graph is connected. This means that there is a path between every pair of vertices.

- Complete graph: Each pair of vertices is adjacent to each other (number of edges= $n(n-1)/2$)

example



counter example



DEFINITION / TERMS

Connectivity (2)

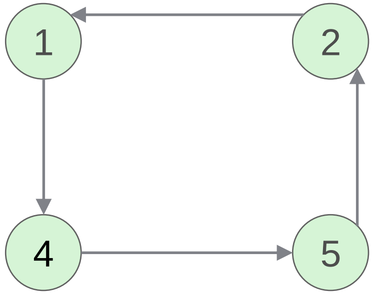
- Strongly connected directed graph is a directed graph in which a *directed* path between every pair of vertices exists
- Weakly connected directed graph is a directed graph whose underlying *undirected* graph is connected, i.e., if replacing all directed edges with undirected edges leads to a connected graph.

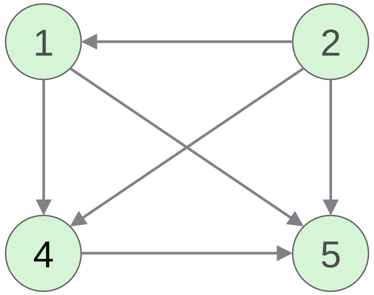
Tree: Connected, acyclic, undirected graph

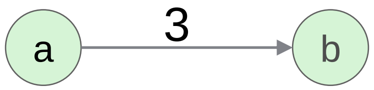
Forest: Set of trees

Weighted graph: Contains weighted edges.

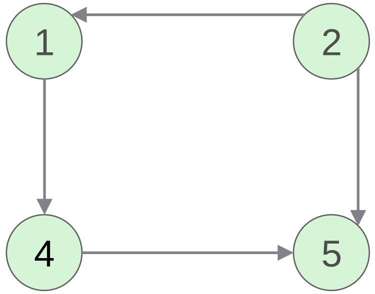
example

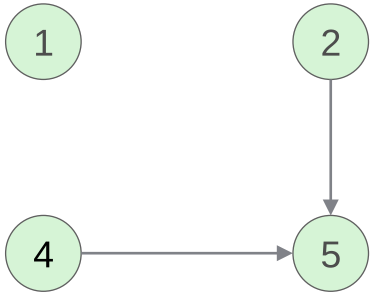






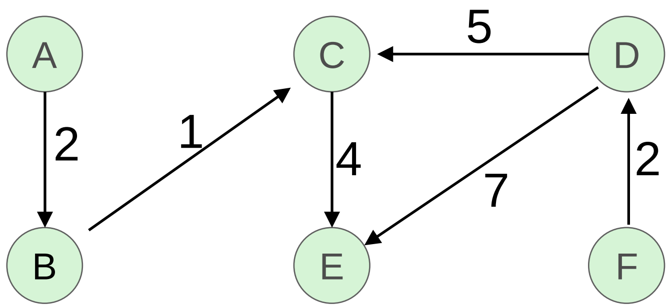
counter example





DEFINITION / TERMS

Example:

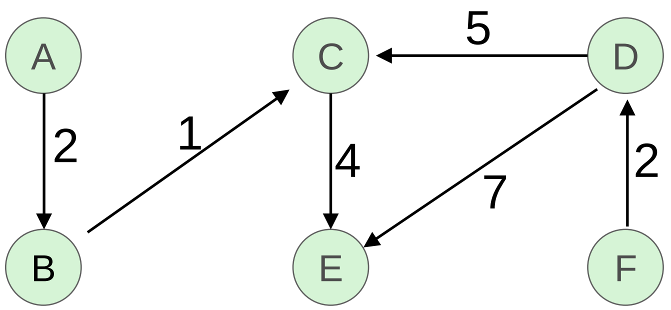


In-degree Out-degree

Degree(„C“):	(2,1)
Weighted:	yes
Directed:	yes
Cyclic:	no
Loops:	0
Connected:	weak
Tree:	no

Graphs can be represented in form of a:
- **Edge list**
- **Adjacency matrix**

DATA STRUCTURES



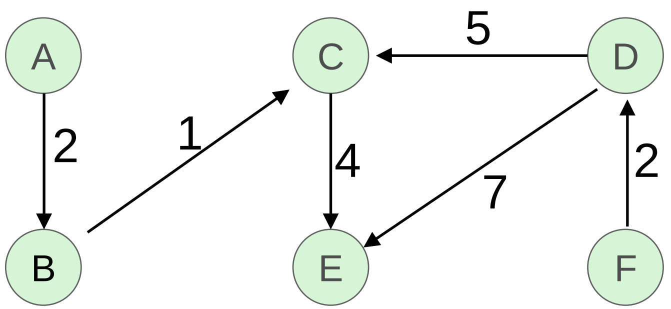
Edge list

- **Principle**: 2 data structures (for vertices and edges)
  - Array/List for vertices (add new vertices at the end)

```
class Vertex {
   Object content
   toString() {...}
}
Vertex vertices[] // in Graph class
```

index	0	1	2	3	4	5
vertex	A	B	C	D	E	F

DATA STRUCTURES



Edge list

- **Principle**: 2 data structures (for vertices and edges)
  - Array/List for vertices (add new vertices at the end)

```
class Vertex {
   Object content
   toString() {...}
}
Vertex vertices[] // in Graph class
```

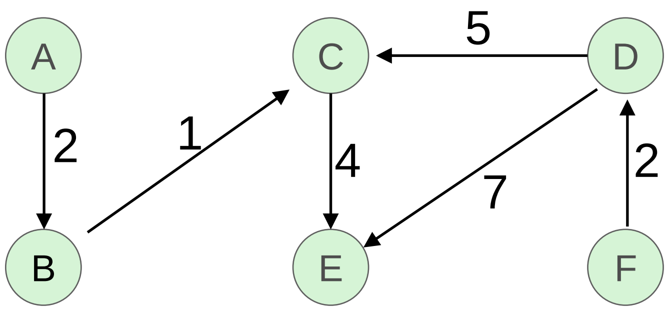
index	0	1	2	3	4	5
vertex	A	B	C	D	E	F

  - Array/List for edges

```
class Edge {
   Vertex first, second  // the edge's vertices
   int weight   // edge weight
}
Edge edges[] // in Graph class
```

index	0	1	2	3	4	5
edge	A B 2	B C 1	C E 4	D C 5	D E 7	F D 2

DATA STRUCTURES



Adjacency matrix

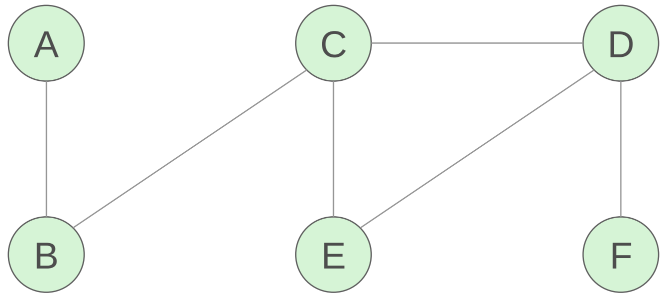
- **Principle**: Graph with *n* vertices is represented by an *n* x *n* matrix
  - Each vertex is assigned to an index.
  - Relation of the vertices are entered in the matrix.
  - True is entered in the i^{th} row and j^{th} column if vertices *i* and *j* are connected by an **unweighted edge**, otherwise false.
  - For **weighted graphs** enter the edge weight
  - The adjacency matrix is symmetrical if the graph does not contain **any directed edges**.
  - The **main diagonal** remains free if the graph contains no **loops**
  - If there is no edge, for example -1 can be entered.

	A	B	C	D	E	F
A		2				
B			1			
C				4		
D			5		7	
E						
F				2		

TRAVERSAL

Two ways of traversing graphs (i.e., visiting all edges):

- **Breadth First Search (BFS)**
- **Depth First Search (DFS)**



DFS/BFS can be used to check:

- Is a graph *G* connected?
- Number of components in *G*?
- Is *G* cyclic?

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)

Principle

- Start with any vertex *v*:
  - Visit *v*
  - Traverse (recursively) any unvisited vertex connected to *v.*

Implementation hint

- Usage of an auxiliary array/set to note which vertices have already been visited.

HINT

The following slides only account for undirected graphs

- In the example, we also want to ignore "trivial cycles“
  (a single edge in an undirected graph automatically forms a "trivial cycle“)

- Directed graphs can use similar techniques, yet certain details may need to be adjusted

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

Is graph *G* connected (method `boolean isConnected()`)**?**

Does graph *G* contain cycles (method `boolean isCyclic()`)**?**

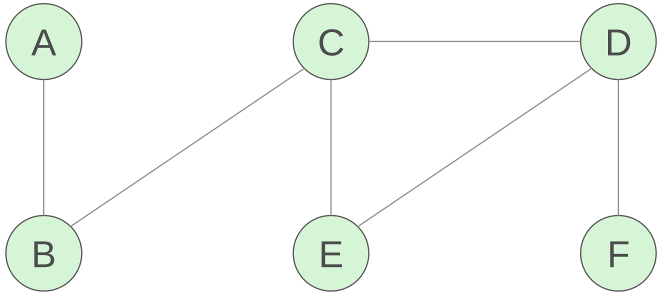
1. Mark **node (v)** as **visited**
   (Set value in auxiliary array at index of v to true).

2. Determine the set of all vertices ***AD(v)*** that are adjacent to ***v***
   (How to create AD(v): Iterate over edge list and determine adjacent vertices).

3. For ***each vertex n in AD(v), if n has not yet been visited,*** go back to step 1 (***v = n***).
   (Can be implemented recursively. If we encounter an already visited node, the graph contains a cycle. In the following example we also have cycle_candidates(v) which are all vertices connected to v, except for the one that called DFS(v)).

→ If the **auxiliary array is completely filled** at the end, the **graph is connected**.

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

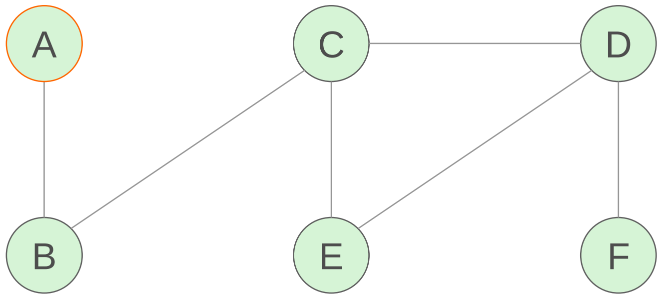


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	F	F	F	F	F	F

Auxiliary array for visited nodes

DFS(A): start vertex A

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

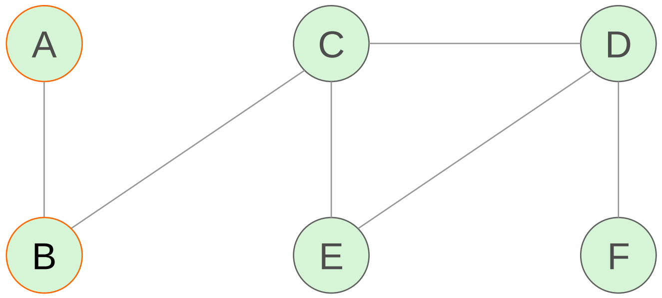


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	F	F	F	F	F

DFS(A): start vertex A

mark A / check B (not visited yet)

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

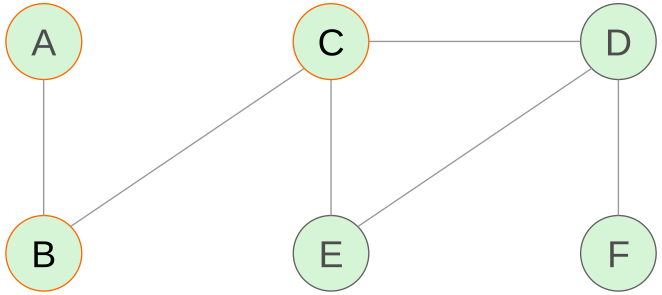


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	F	F	F	F

DFS(B): A → B

mark B / check A (already visited), C

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



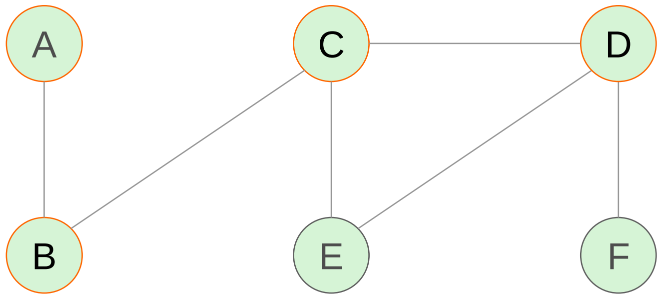
index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	F	F	F

DFS(C): A → B → C

mark C / check B (already visited), D, E

No general visiting order
→ Implementation dependent

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

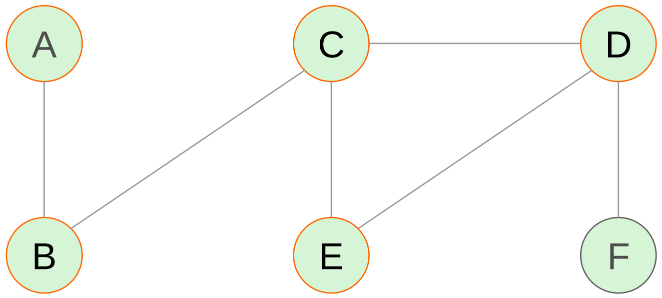


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	F	F

DFS(D): A → B → C → D

mark D / check C (already visited), E, F

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

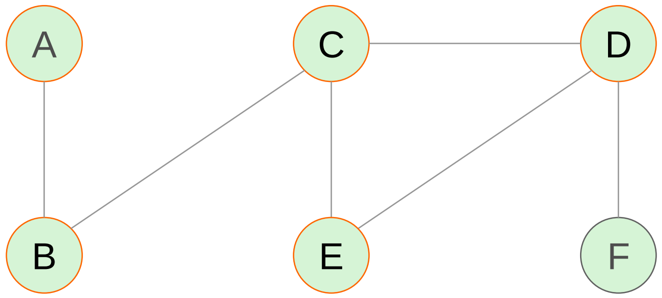


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	F

DFS(E): A → B → C → D → E

mark E / check C, D (cycle candidates („visited without last“): A, B, C.
D is no candidate because it was the last one visited – this is to prevent trivial cycles)

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



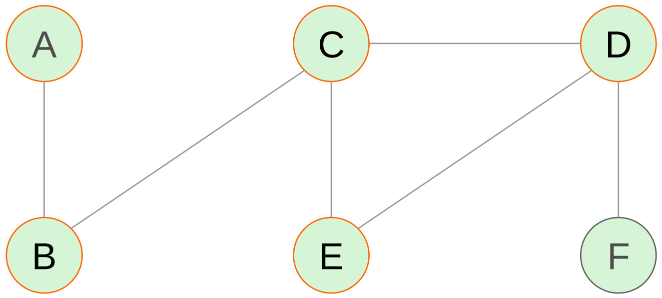
index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	F

DFS(E): A → B → C → D → E

overlap

mark E / check C, D (cycle candidates („visited without last“): A, B, C)

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	F

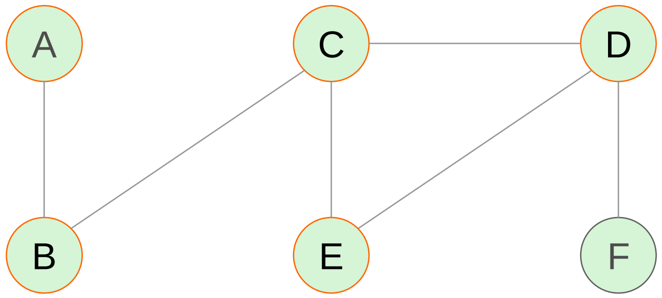
DFS(E): A → B → C → D → E

Overlap between the vertex to be checked (adjacent) and cycle candidate (visited but not last) → **cyclic graph!**

overlap

mark E / check C, D (cycle candidates („visited without last“): A, B, C)

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	F

Overlap between the vertex to be checked (adjacent) and cycle candidate (visited but not last) → **cyclic graph!**

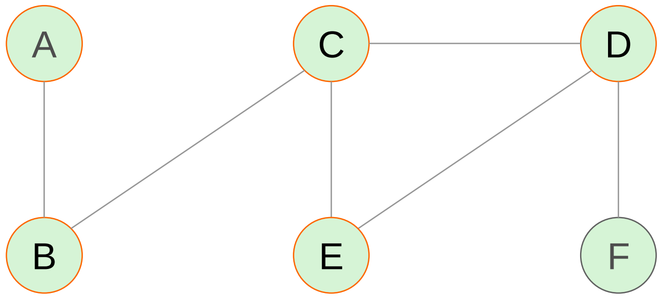
DFS(E): A → B → C → D → E

overlap

mark E / check C, D (cycle candidates („visited without last“): A, B, C)

Vertices C, D already marked, **therefore go back in recursion** to vertex D and visit F from there.

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

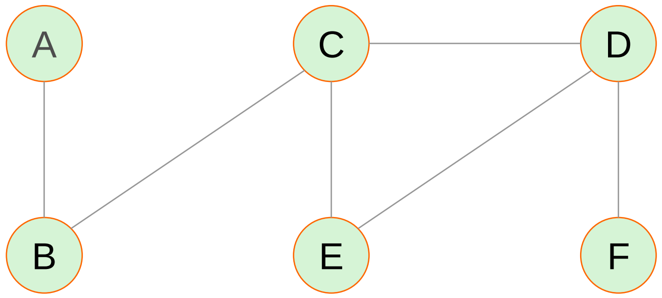


index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	F

*DFS(**D**):* A → B → C → **D** → E

mark D / check E (already visited), F

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	T

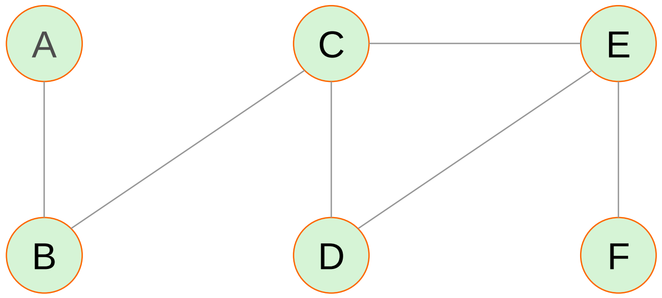
DFS(F): A → B → C → D → E
→ F

mark F / check D (cycle candidate: A,C,B,E)

Vertex D already marked, **therefore:**

- **go back in recursion to D** (no more unvisited neighbors to visit from E)
    - **go back in recursion to C** (no more unvisited neighbors to visit from C)
        - **go back in recursion to B** (no more unvisited neighbors to visit from B)
            - **go back in recursion to A** (no more unvisited neighbors to visit from A)
                - **end DFS**

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS



index	1	2	3	4	5	6
vertex	A	B	C	D	E	F
Visited	T	T	T	T	T	T

DFS(F): A → B → C → D → E
→ F

mark F / check E (cycle candidate: A,C,B,D)

Auxiliary array filled completely
→ graph connected!

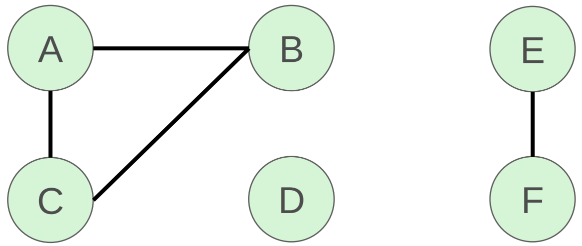
Vertex E already marked, **therefore:**

- **go back in recursion to E** (no more unvisited neighbors to visit from E)
    - **go back in recursion to** C (no more unvisited neighbors to visit from C)
        - **go back in recursion to** B (no more unvisited neighbors to visit from B)
            - **go back in recursion to A** (no more unvisited neighbors to visit from A)
                - end DFS

TRAVERSAL :: DEPTH FIRST SEARCH (DFS) UNDIRECTED GRAPHS

What is the number of components in the graph (method `int getNumOfComponents()`)?

Graph:



- DFS is called once (starting with vertex A) and return the following array (= 1. component)

vertex	A	B	C	D	E	F
visited	*T*	*T*	*T*	*F*	*F*	*F*

Then call DFS until all fields are marked (continuing with the next unmarked vertex, here D)

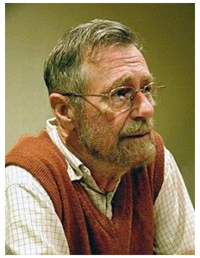
- DFS ends for the 2. time (= 2. component)

vertex	A	B	C	D	E	F
visited	*T*	*T*	*T*	*T*	*F*	*F*

- DFS ends for the 3. time (= 3. component)

vertex	A	B	C	D	E	F
visited	*T*	*T*	*T*	*T*	*T*	*T*

GRAPHS :: DIJKSTRA

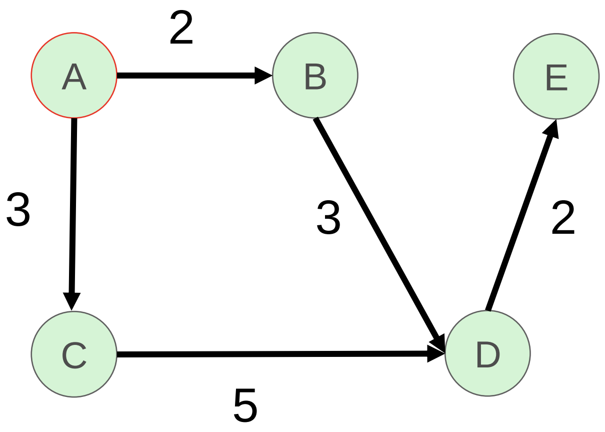


Edsger Wybe Dijkstra (1930-2002)

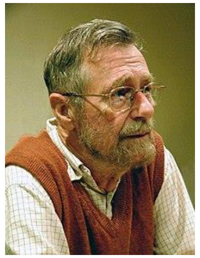
Dutch Computer Scientist
1972 Turing Award

"Computer science is no more about computers than astronomy is about telescopes."

Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.



GRAPHS :: DIJKSTRA

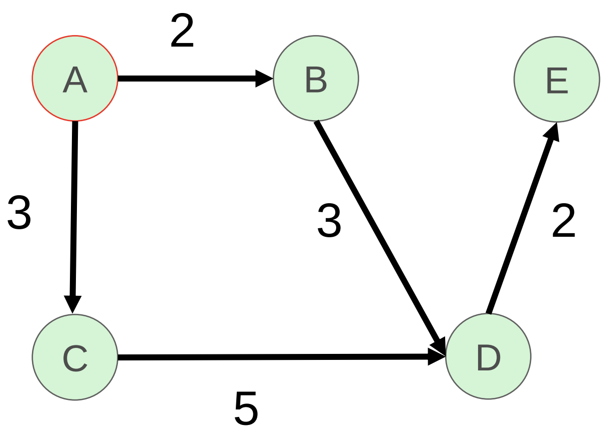


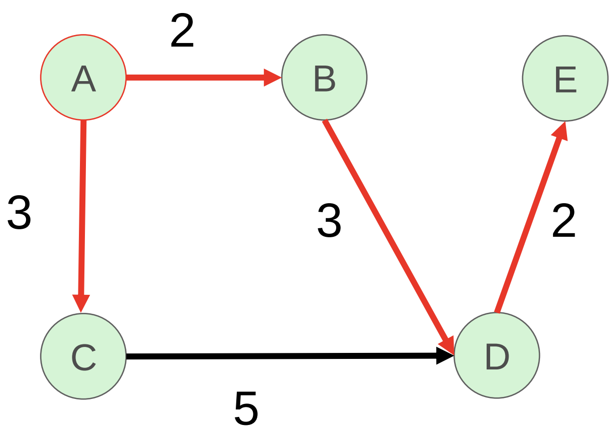
Edsger Wybe Dijkstra (1930-2002)

Dutch Computer Scientist
1972 Turing Award

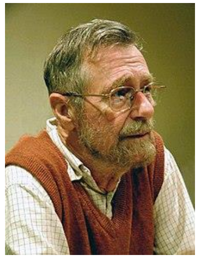
"Computer science is no more about computers than astronomy is about telescopes."

Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.





GRAPHS :: DIJKSTRA



Edsger Wybe Dijkstra (1930-2002)

Dutch Computer Scientist
1972 Turing Award

"Computer science is no more about computers than astronomy is about telescopes."

Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.

- Shortest Path = Path with the minimum total edge weight

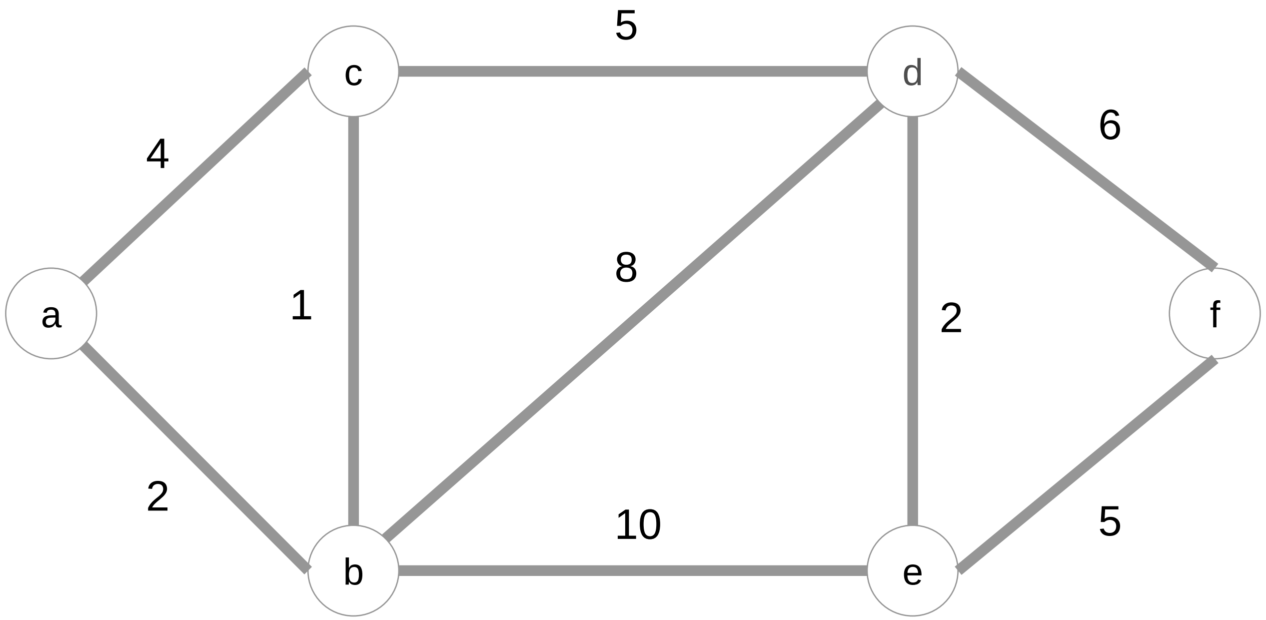
Preconditions:

- Edge weights must not be negative

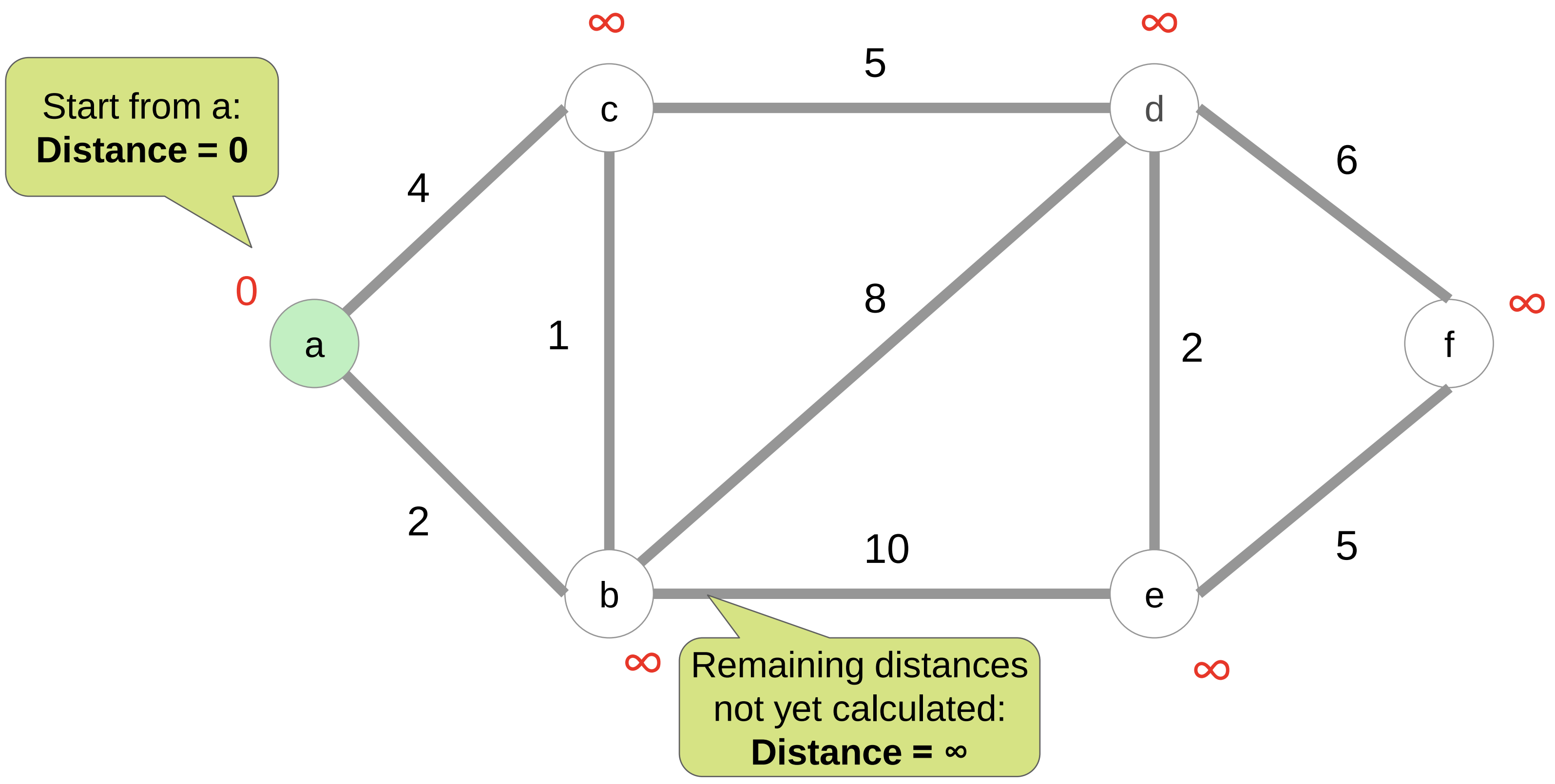
Example application:

- Routing protocols

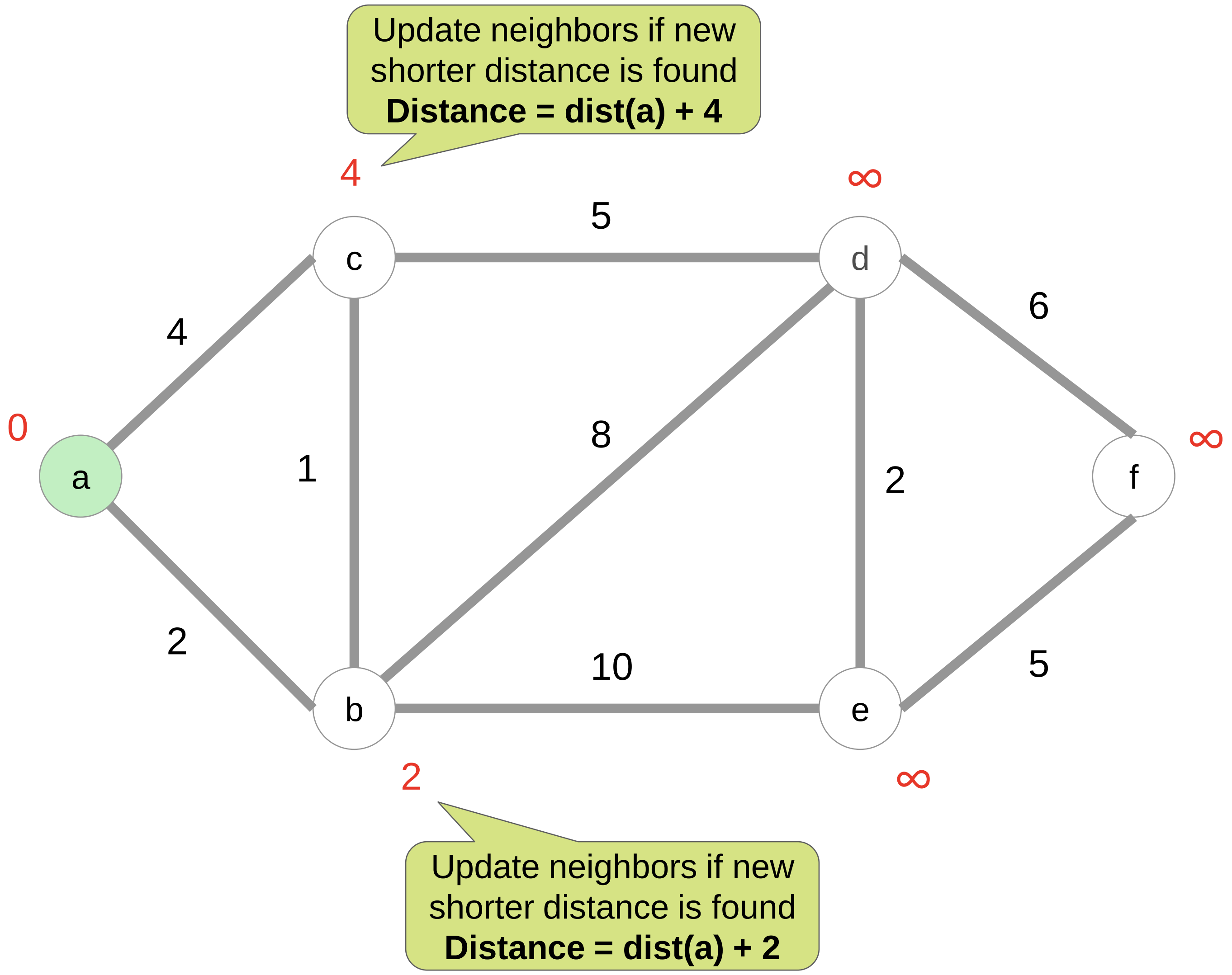
GRAPHS :: DIJKSTRA :: SHORTEST PATH



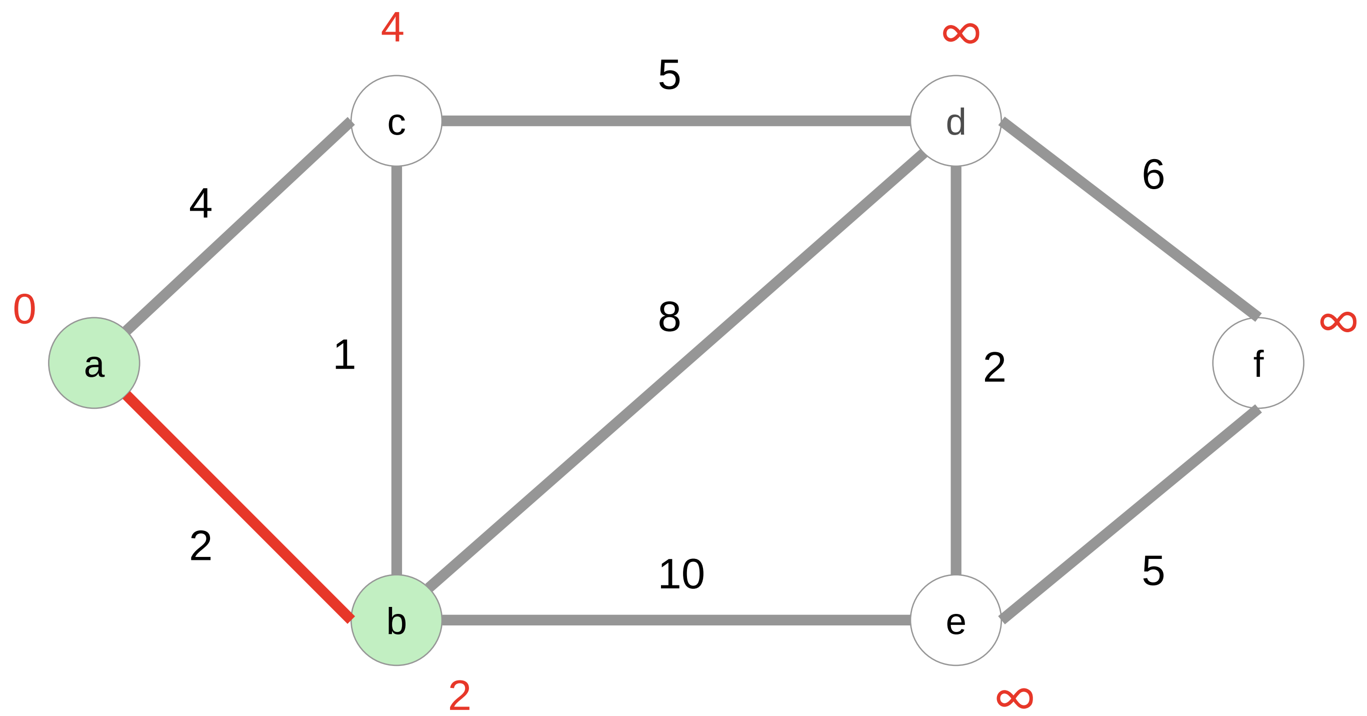
GRAPHS :: DIJKSTRA :: SHORTEST PATH



GRAPHS :: DIJKSTRA :: SHORTEST PATH



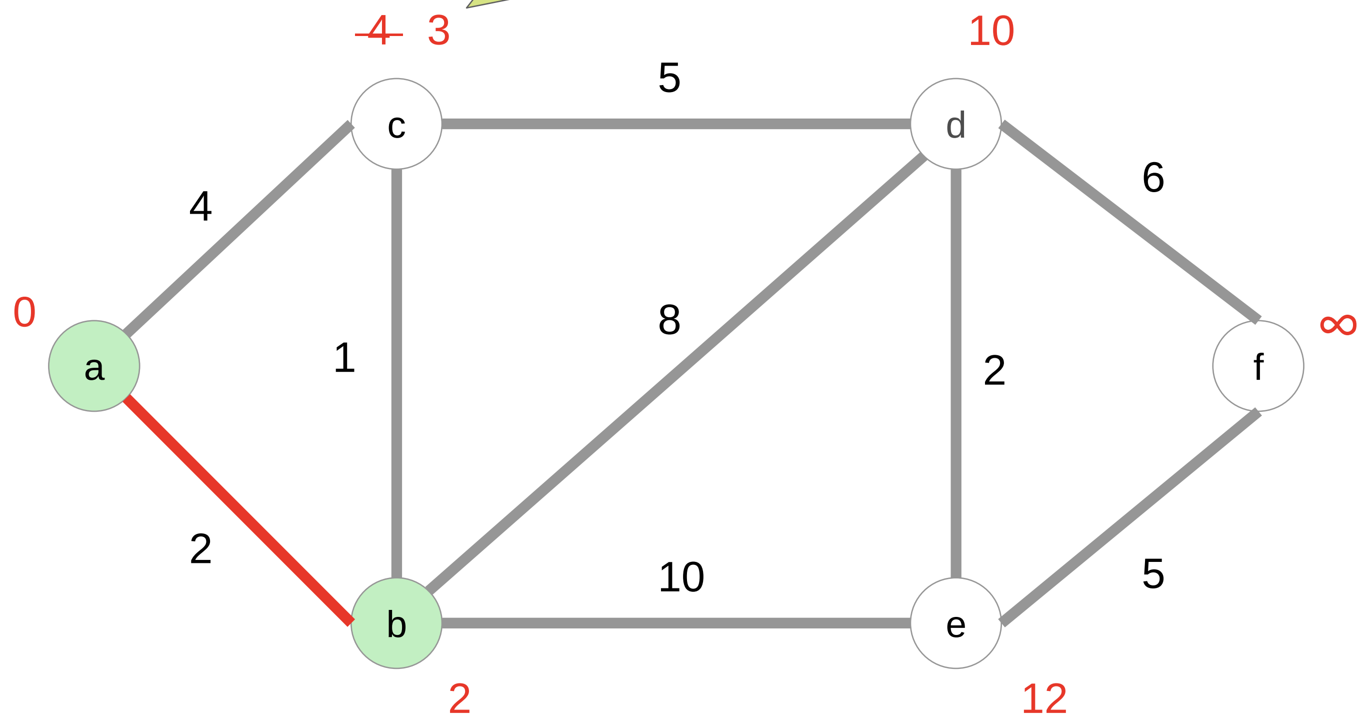
GRAPHS :: DIJKSTRA :: SHORTEST PATH



Continue with **unvisited vertex** that has **smallest dist**

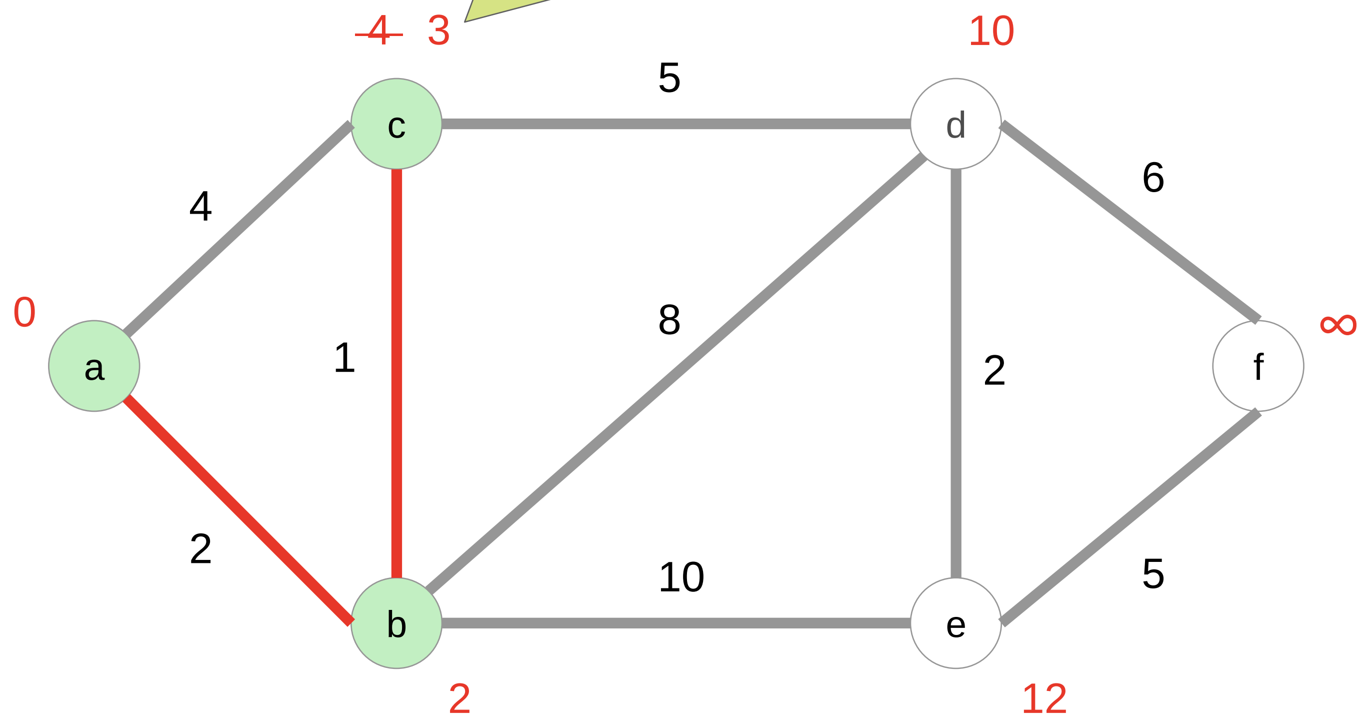
GRAPHS :: DIJKSTRA :: SHORTEST PATH

Update neighbors if new shorter distance is found
Distance = dist(b) + 1



GRAPHS :: DIJKSTRA :: SHORTEST PATH

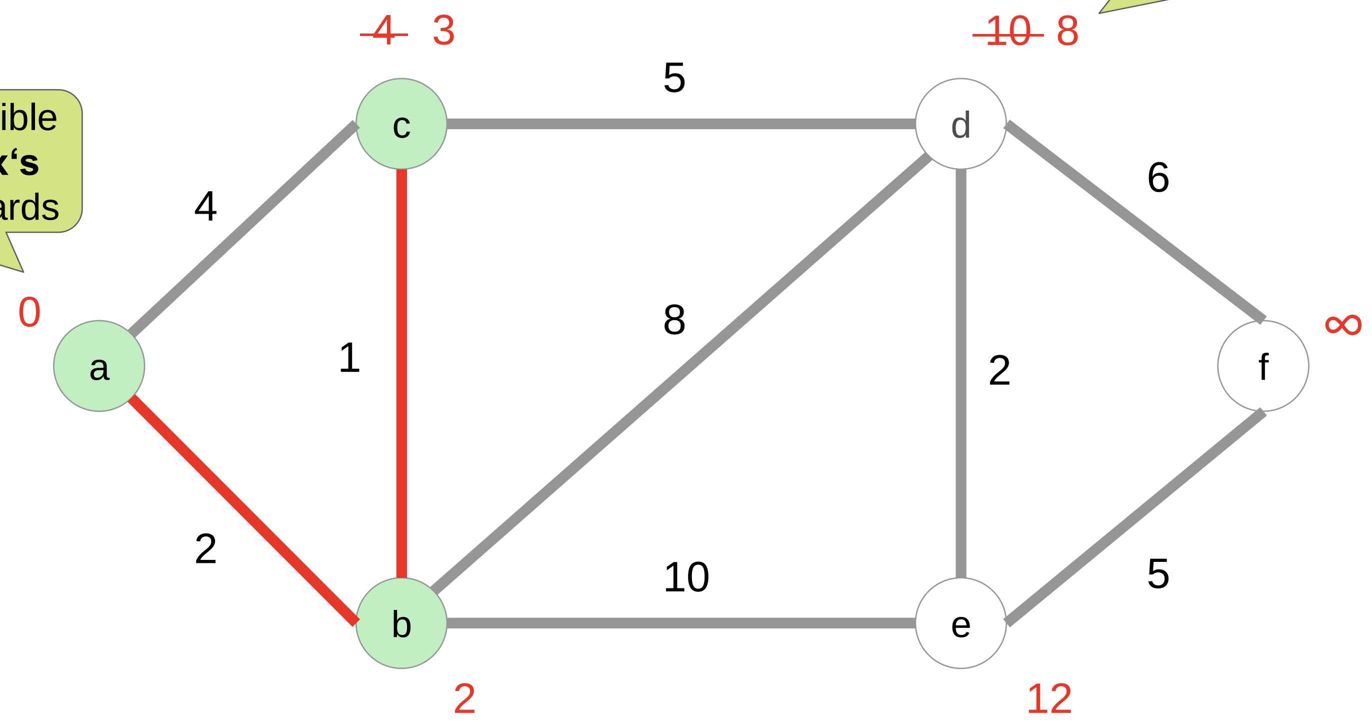
Continue with **unvisited vertex** that has **smallest dist**



GRAPHS :: DIJKSTRA :: SHORTEST PATH

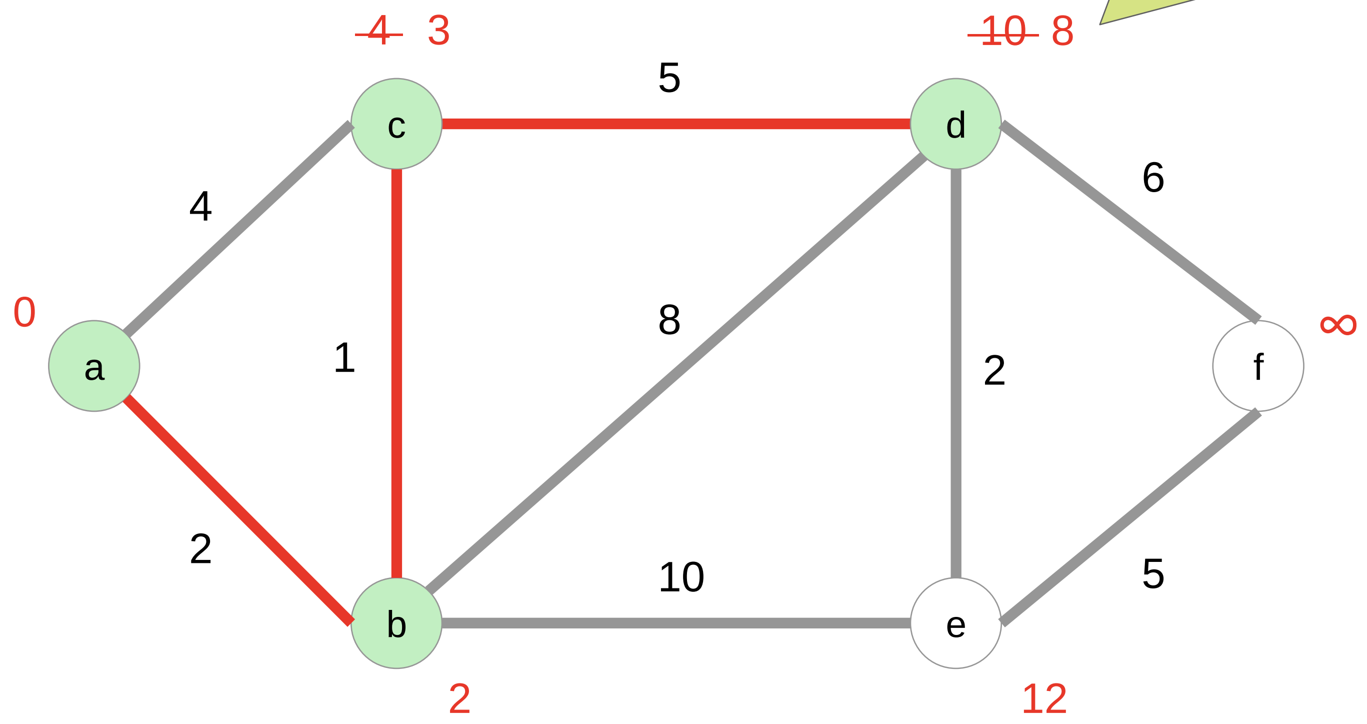
Hint: It is **never** possible that a **visited vertex's dist changes** afterwards

Update neighbors if new shorter distance is found
Distance = dist(c) + 5

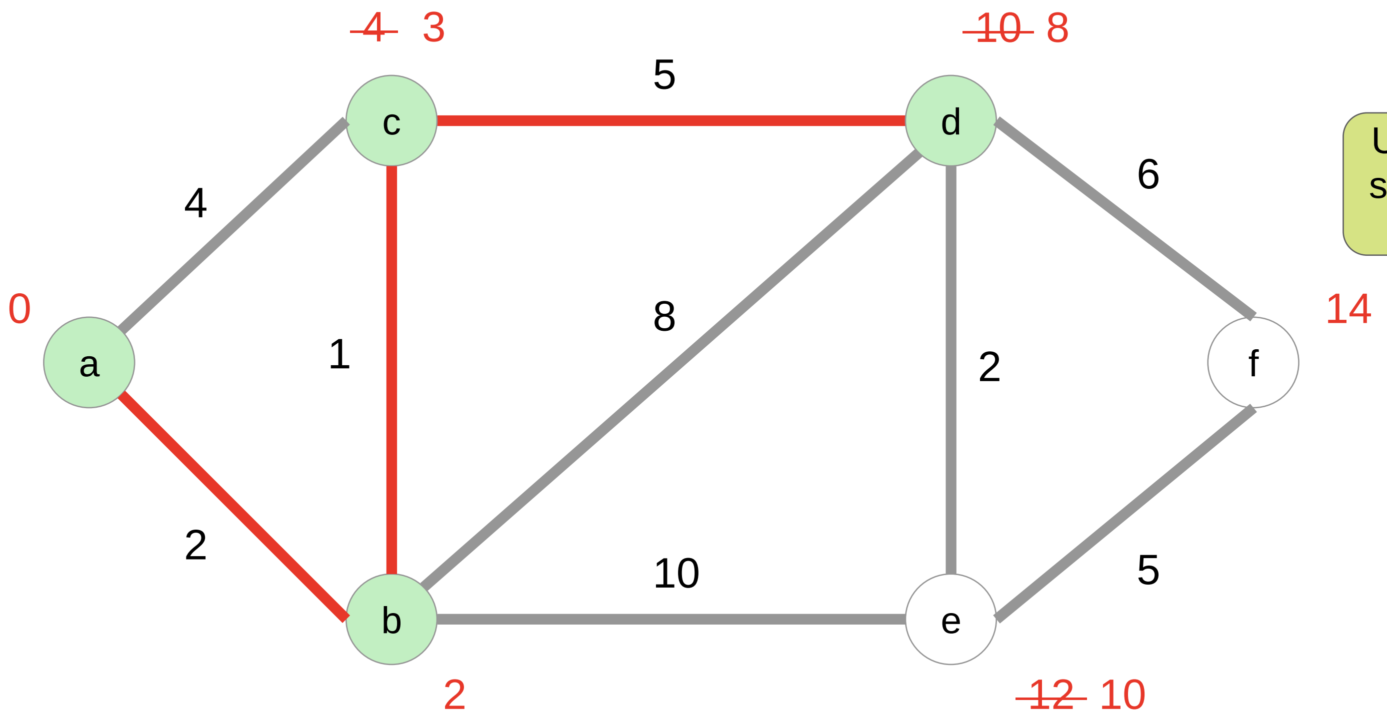


GRAPHS :: DIJKSTRA :: SHORTEST PATH

Continue with **unvisited vertex** that has **smallest dist**

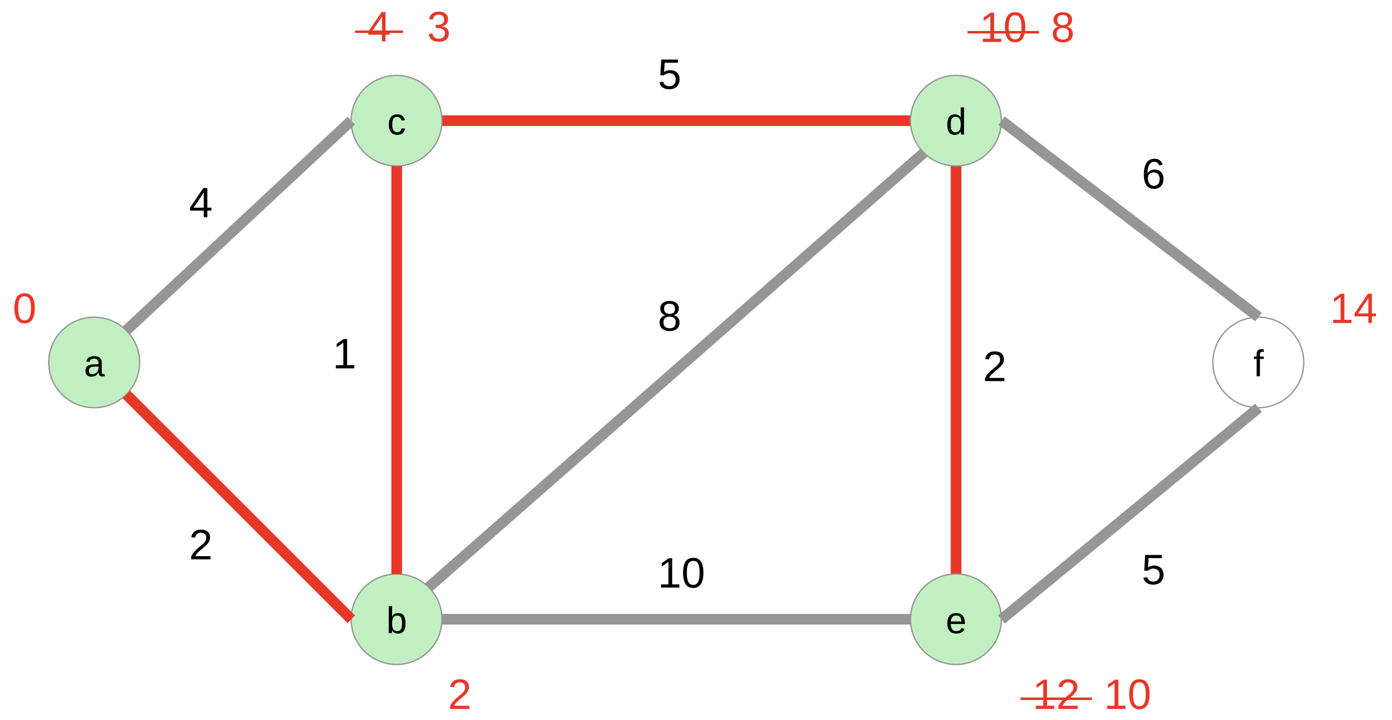


GRAPHS :: DIJKSTRA :: SHORTEST PATH



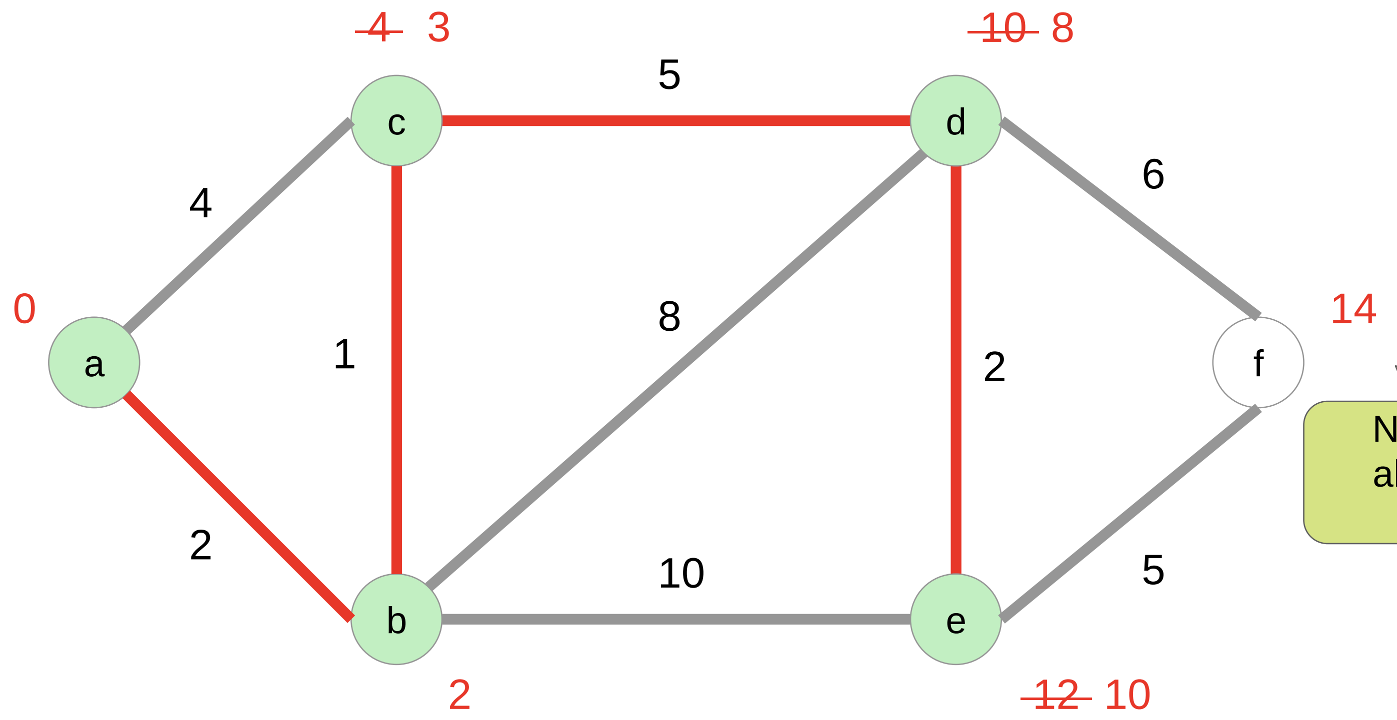
Update neighbors if new shorter distance is found
Distance = dist(d) + 6

GRAPHS :: DIJKSTRA :: SHORTEST PATH



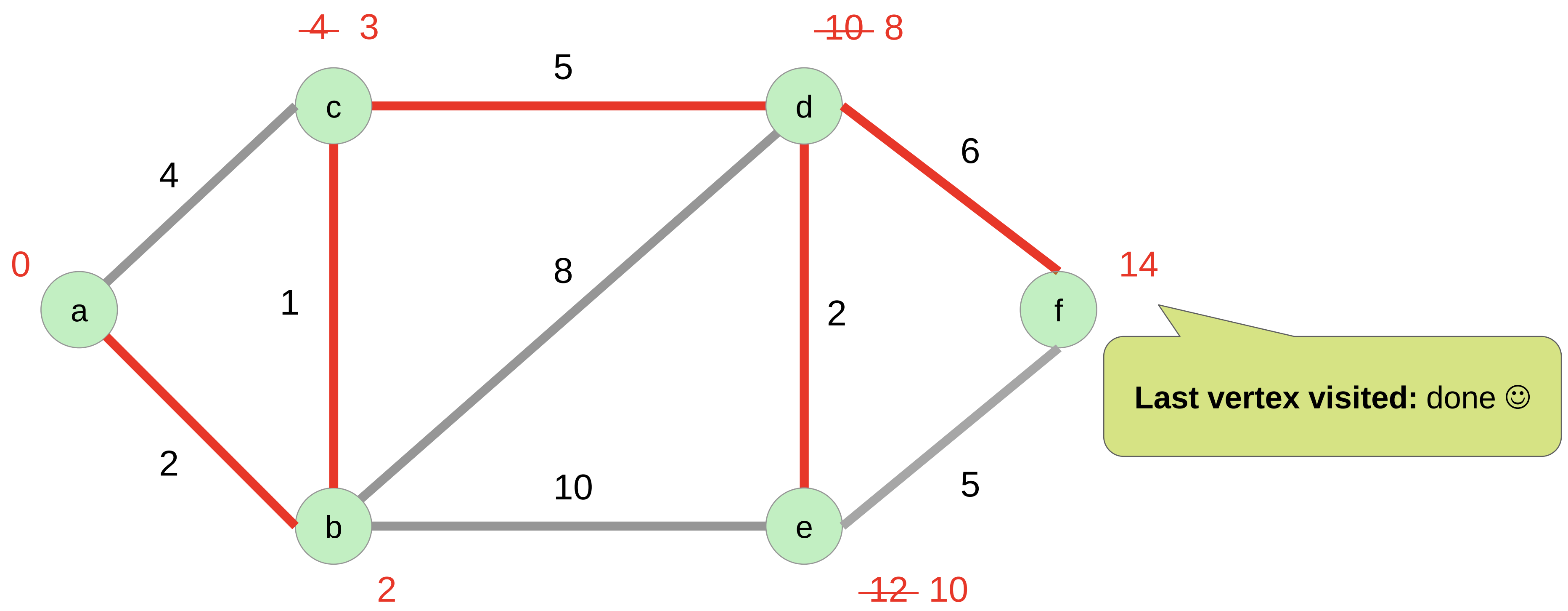
Continue with **unvisited vertex** that has **smallest dist**

GRAPHS :: DIJKSTRA :: SHORTEST PATH

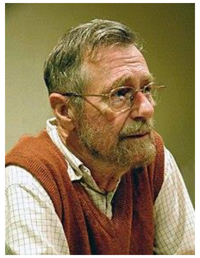


Nothing to update because we already found shorter distance
14 < dist(e) + 5

GRAPHS :: DIJKSTRA :: SHORTEST PATH



GRAPHS :: DIJKSTRA



Edsger Wybe Dijkstra (1930-2002)

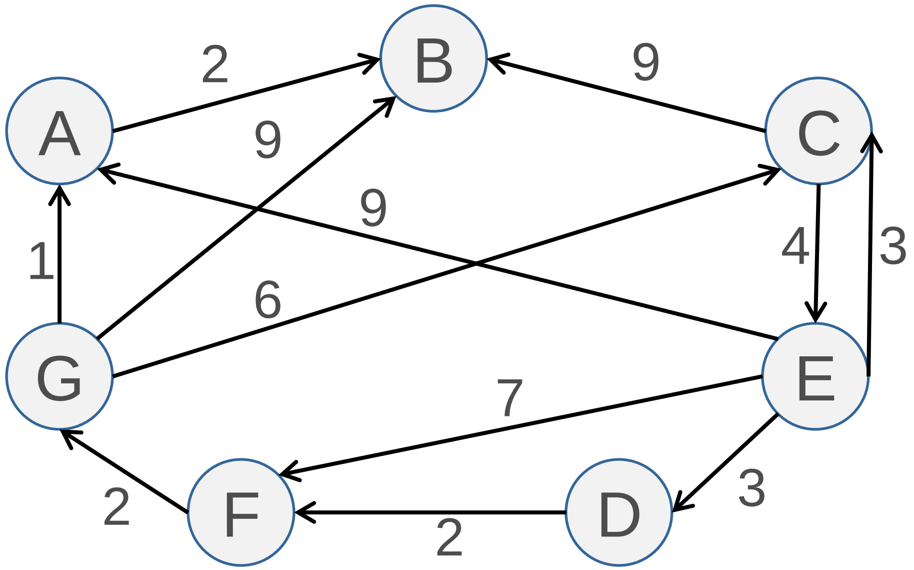
Dutch Computer Scientist
1972 Turing Award

"Computer science is no more about computers than astronomy is about telescopes."

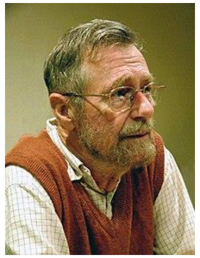
Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.
- Shortest Path = Path with the minimum total edge weight

Preconditions:
- Edge weights must not be negative

Example application:
- Routing protocols



GRAPHS :: DIJKSTRA



Edsger Wybe Dijkstra (1930-2002)

Dutch Computer Scientist
1972 Turing Award

"Computer science is no more about computers than astronomy is about telescopes."

Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.

- Shortest Path = Path with the minimum total edge weight

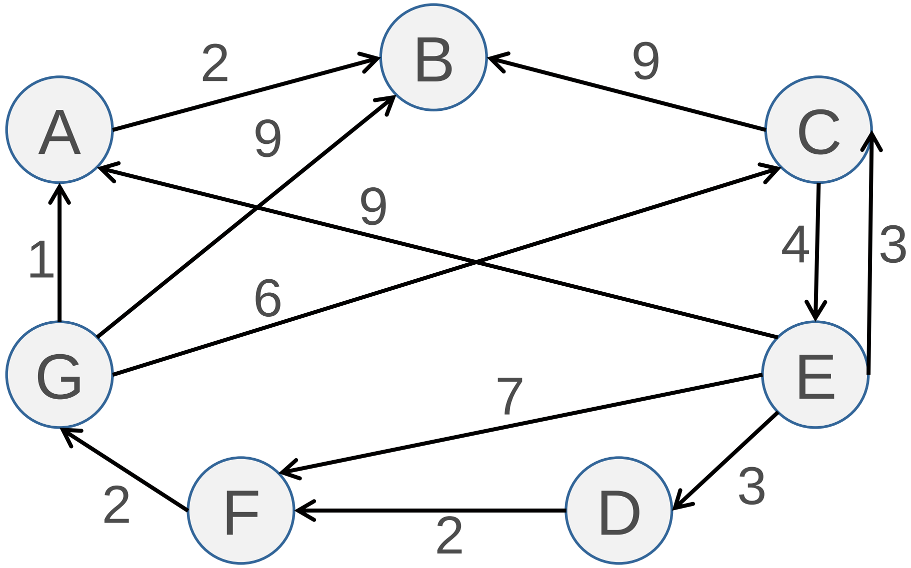
Preconditions:

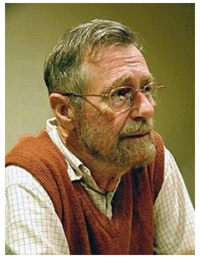
- Edge weights must not be negative

Example application:

- Routing protocols

	A	**B**	**C**	**D**	**E**	**F**	**G**
A		2					
B							
C		9			4		
D						2	
E	9		3	3		7	
F							2
G	1	9	6				





Edsger Wybe Dijkstra (1930-2002)

Dutch Computer Scientist
1972 Turing Award

"Computer science is no more about computers than astronomy is about telescopes."

GRAPHS :: DIJKSTRA

Find the shortest path in **weighted graphs** from vertex *x* to all accessible vertices.

- Shortest Path = Path with the minimum total edge weight

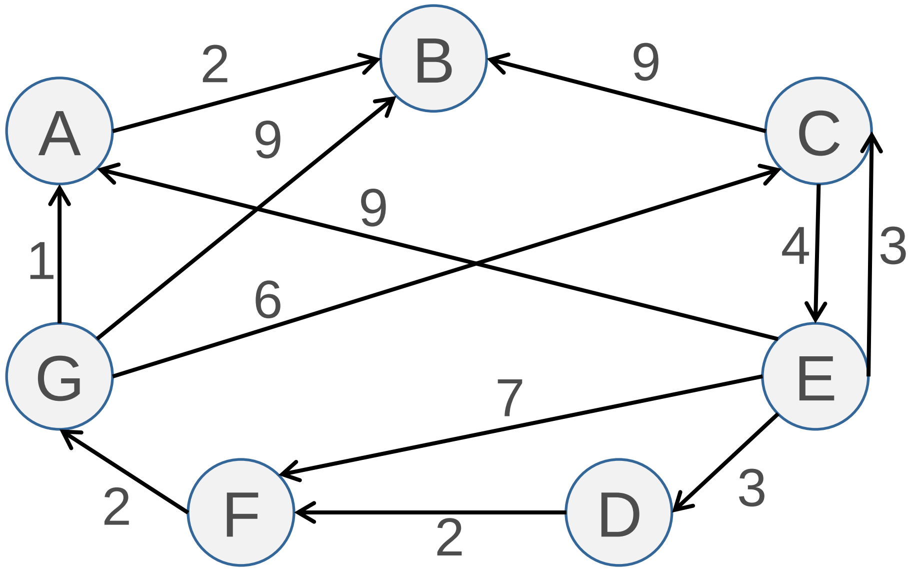
Preconditions:

- Edge weights must not be negative

Example application:

- Routing protocols

	A	B	C	D	E	F	G
A		2					
B							
C		9			4		
D						2	
E	9		3	3		7	
F							2
G	1	9	6				



The matrix is to be read line by line. For example, in the 1st line there is a directed edge with weight 2 from vertex *A* to the vertex *B*.

GRAPHS :: DIJKSTRA

Three set of vertices (either VV or NV is optional, can be calculated on the fly):

- All vertices (**V**) → Given by graph
- Already visited vertices (**VV**) → „Cloud of Vertices“ (in the end contains all reachable vertices)
- Not yet visited vertices (**NV**)

1. Search starts with some vertex *x*

- ***x*** is put into the set **VV** and removed from set **NV**
- Determine the distances to adjacent vertices
- Store current minimum distances in array *d* (length of d = number of vertices)

- **Initialization:** $d(i) = \infty$
- **First step**: **d(i) = weight(x, i)** ... for vertices ***i*** that are adjacent to ***x***

GRAPHS :: DIJKSTRA

2. Next step

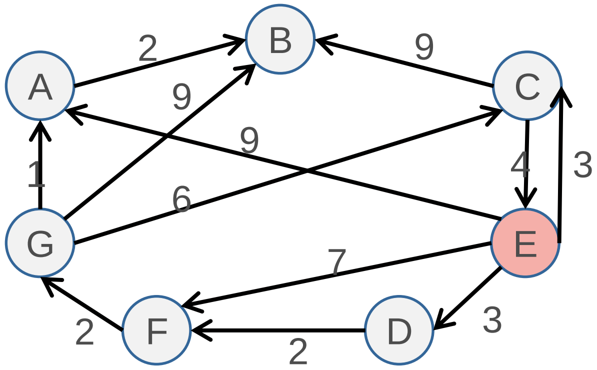
- Select vertex from **NV** with smallest minimum distance (that is not ∞) as new start vertex ***w***.
- Continue until the set **NV** is empty (or all vertices in NV have a minimum distance of ∞).
- As soon as a vertex is in set **VV**, its minimum distance is never changed anymore.

In each step: d(i) = min(d(i), d(w) + weight(w,i)) ... for all nodes *i* in **NV** adjacent to ***w***

d(i) ... Current minimum distance from vertex ***i*** to the start vertex ***x***.

d(w) ... Minimum distance from current vertex ***w*** to the start vertex ***x***.

GRAPHS :: DIJKSTRA



2. Next step

- Select vertex from **NV** with smallest minimum distance (that is not ∞) as new start vertex ***x***.
- Continue until the set **NV** is empty.
- As soon as a vertex is in set **VV**, its minimum distance is never changed anymore.

In each step: d(i) = min(d(i), d(w) + weight(w,i)) ... for all nodes *i* in **NV** adjacent to ***w***

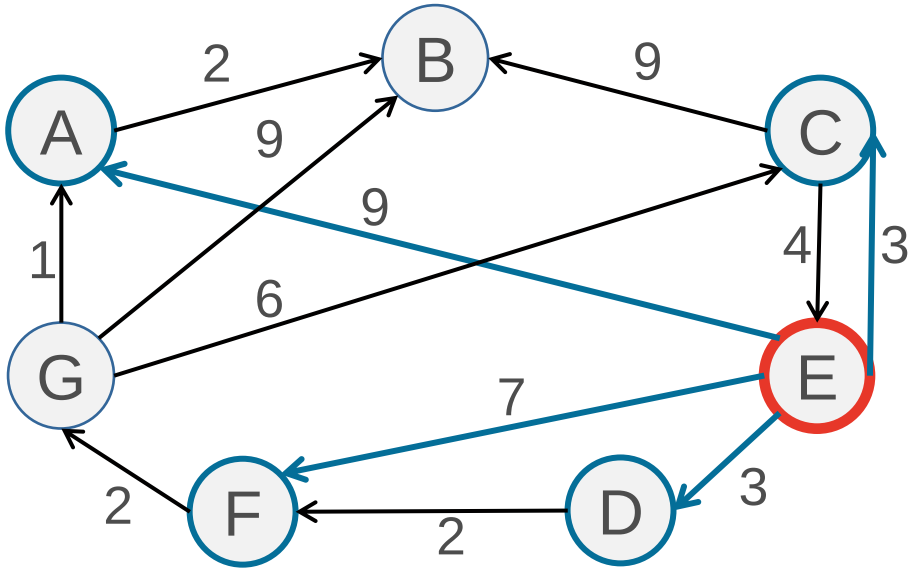
d(i) ... Current minimum distance from vertex ***i*** to the start vertex ***x***.

d(w) ... Minimum distance from current vertex ***w*** to the start vertex ***x***.

Example:

Calculate the distances of the shortest paths from vertex **E** to all other vertices in the graph, using the shortest path algorithm of Dijkstra.

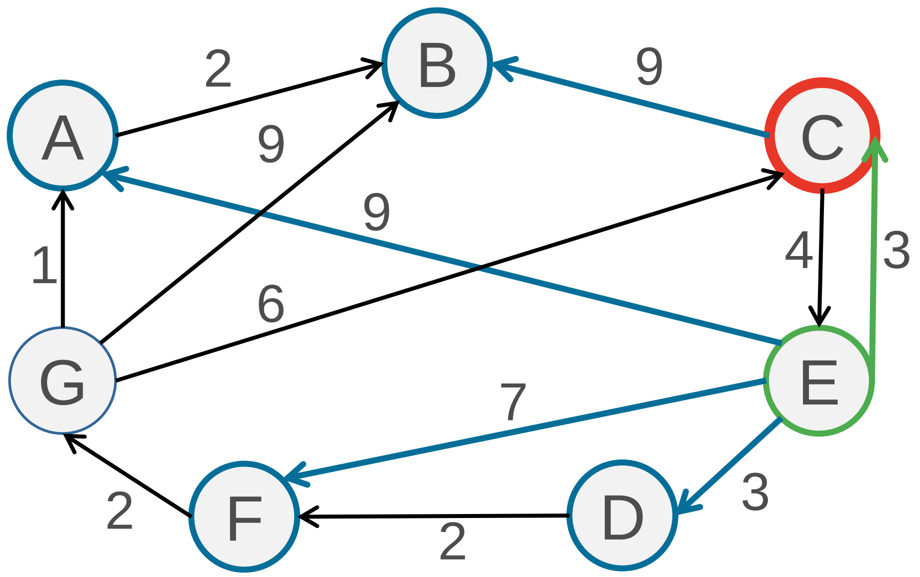
GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3

- Initialize: For each vertex ***i*** adjacent to start vertex ***x*** → *d(i) = weight(x, i), shortest_path(i) = [x, i]*
  - others → *d(i) = ∞, shortest_path(i) = []*
- Next vertex → **NV** vertex with **local min**
  - **Local min**: NV with **min. distance < ∞**
  - **In this case: C** or **D**

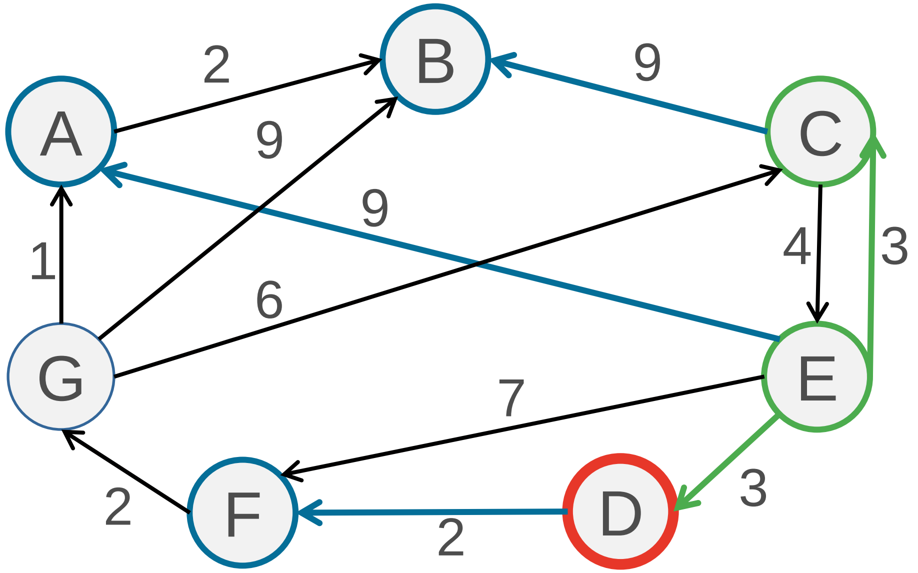
GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{E,C}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3

- d(B) = min(d(B), d(C) + 9) = min(∞, 12) = 12
  - shortest_path(B) changes from [] to [E,C,B] (which is shortest_path(C) + B)
- d(E): Vertex E is already in ***VV***; therefore no changes anymore

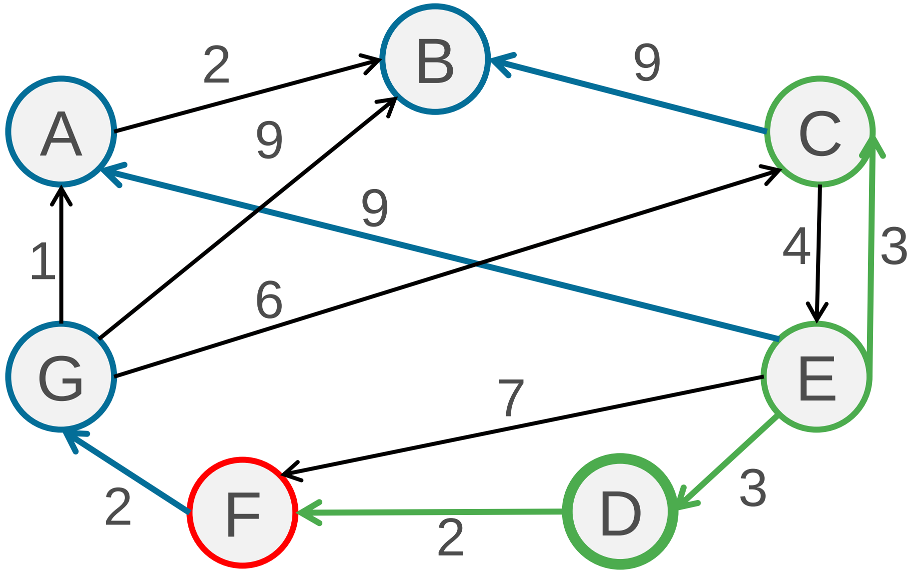
GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	5	∞	5

- d(F): min(d(F), d(D) + 2) = min(7, 3 + 2) = 5
  - shortest_path(F) changes from [E,F] to [E,D,F] (which is shortest_path(D) + F)

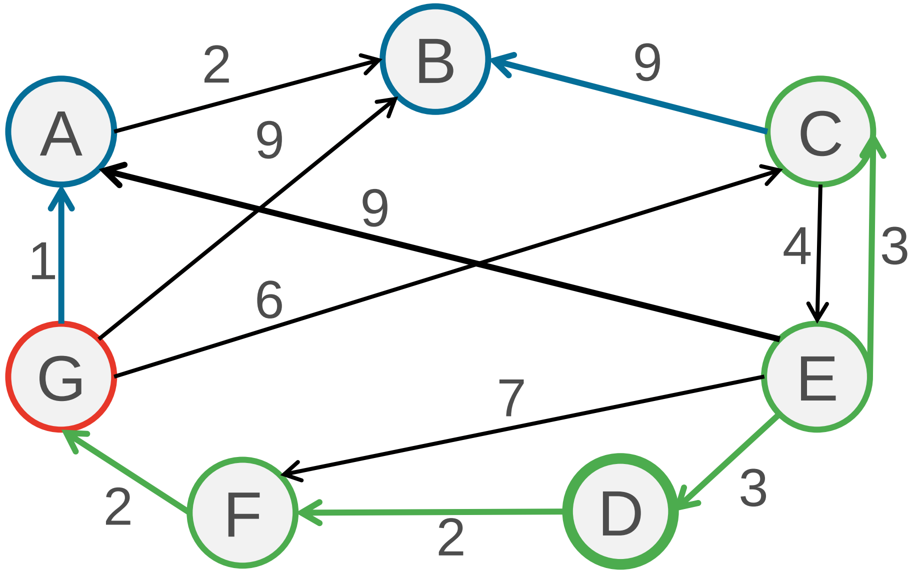
GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	5	∞	5
F	{C,D,E,F}	{A,B,G}	9	12	.	.	.	.	7	7

- d(G) = min(d(G), d(F) + 2) = min(∞, 7) = 7
  - shortest_path(G) changes from [] to [E,D,F,G] (which is shortest_path(F) + G)

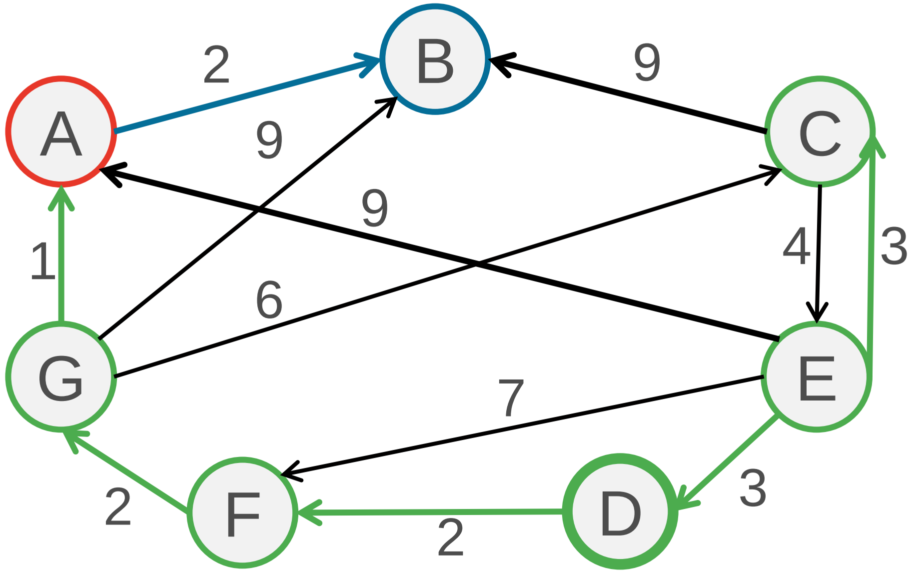
GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	5	∞	5
F	{C,D,E,F}	{A,B,G}	9	12	.	.	.	.	7	7
G	{C,D,E,F,G}	{A,B}	8	12	.	.	.	.	.	8

- d(A): min(d(A), d(G) + 1) = min(9, 7 + 1) = 8
  - shortests_path(A) changes from [E,A] to [E,D,F,G,A] (which is shortest_path(G) + A)

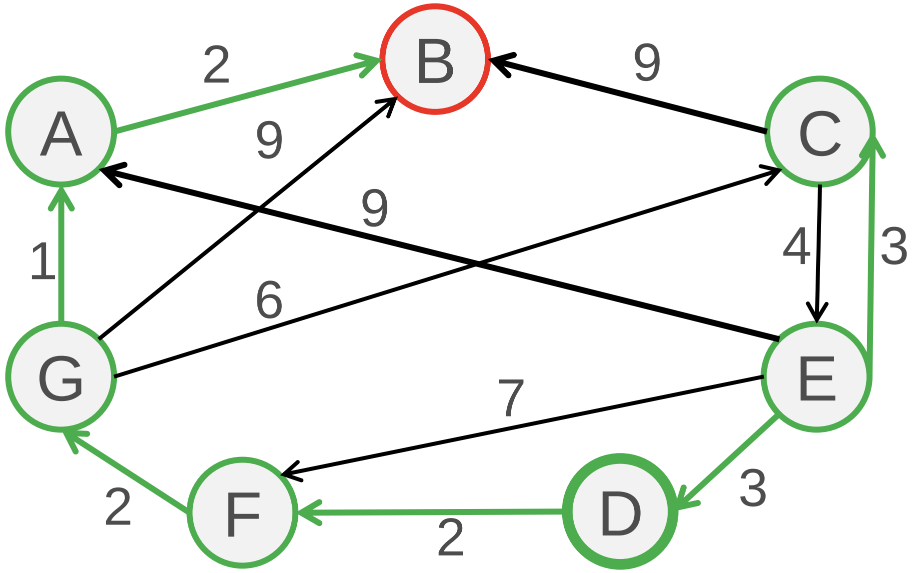
GRAPHS :: DIJKSTRA



- d(B): d(A) + 2
- shortests_path(B) changes from [E,C,B] to [E,D,F,G,A,B]

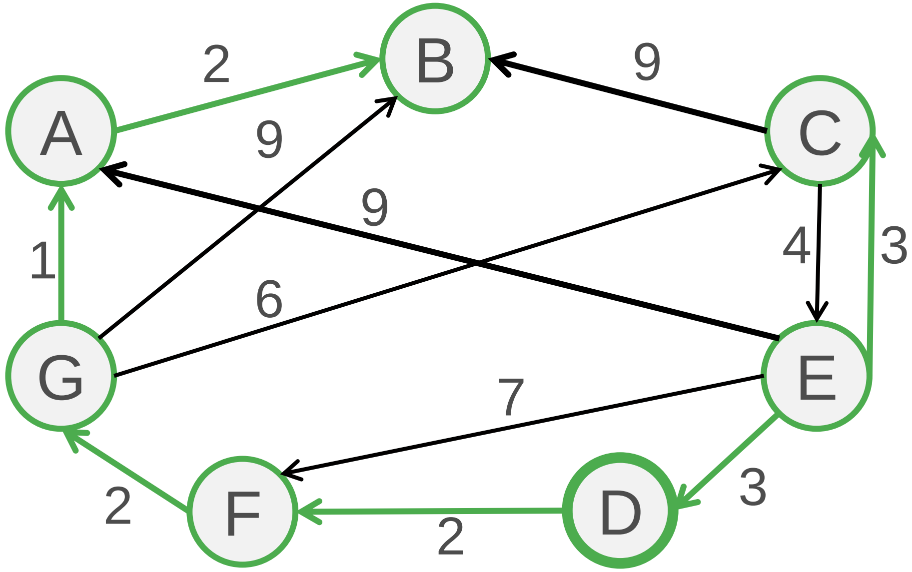
Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	5	∞	5
F	{C,D,E,F}	{A,B,G}	9	12	.	.	.	.	7	7
G	{C,D,E,F,G}	{A,B}	8	12	.	.	.	.	.	8
A	{A,C,D,E,F,G}	{B}	.	10	.	.	.	.	.	10

GRAPHS :: DIJKSTRA



Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	3	3	0	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	3	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	5	∞	5
F	{C,D,E,F}	{A,B,G}	9	12	.	.	.	.	7	7
G	{C,D,E,F,G}	{A,B}	8	12	.	.	.	.	.	8
A	{A,C,D,E,F,G}	{B}	.	10	.	.	.	.	.	10
B	{*}	{}	.	.	.	.	.	.	.	-

GRAPHS :: DIJKSTRA

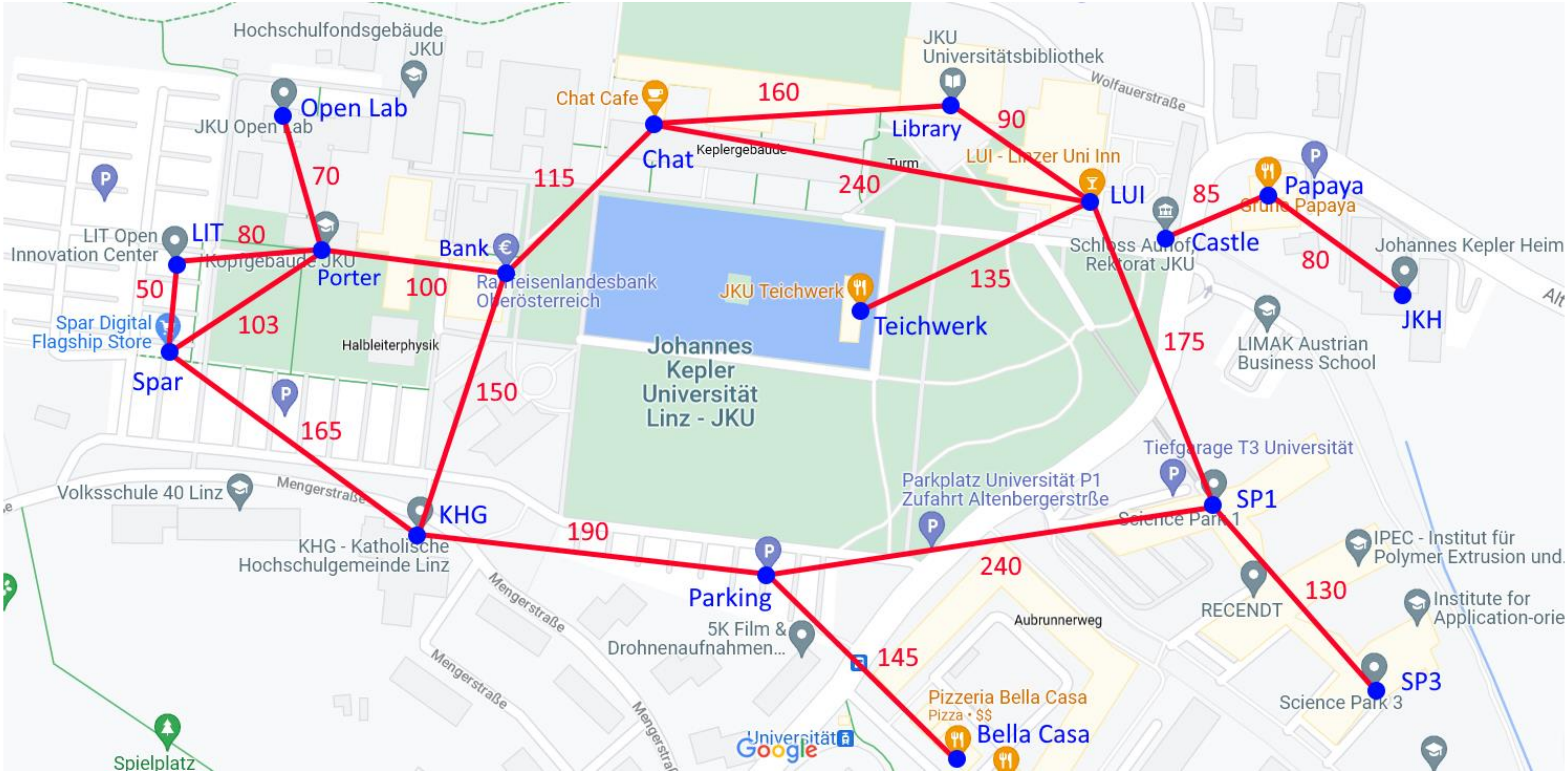


Shortest Paths from starting vertex ***E*** to all other vertices in the directed, weighted graph by Dijkstra

Current vertex	Already visited (VV)	Not yet visited (NV)	d (A)	d (B)	d (C)	d (D)	d (E)	d (F)	d (G)	Local min.
E	{E}	{A,B,C,D,F,G}	9	∞	$\boxed{3}$	3	$\boxed{0}$	7	∞	3
C	{C,E}	{A,B,D,F,G}	9	12	.	$\boxed{3}$	.	7	∞	3
D	{C,D,E}	{A,B,F,G}	9	12	.	.	.	$\boxed{5}$	∞	5
F	{C,D,E,F}	{A,B,G}	9	12	.	.	.	.	$\boxed{7}$	7
G	{C,D,E,F,G}	{A,B}	$\boxed{8}$	12	.	.	.	.	.	8
A	{0,2,3,4,5,6}	{B}	.	$\boxed{10}$	.	.	.	.	.	10
B	{*}	{}	.	.	.	.	.	.	.	-

ASSIGNMENT 04

ASSIGNMENT 04



HINTS FOR ASSIGNMENT 04 (CS)

For the implementation of this assignment a method such as the following is recommended:

```
void dijkstra(V cur, HashSet<V> visited, HashMap<V, Integer> distances, HashMap<V, ArrayList<V>> paths)
```

- **`cur`** ... current vertex
- **`visited`** ... a HashSet which notes already visited vertices
- **`distances`** ... Map (nVertices entries), which stores the min. distance to each vertex. (Suggestion: initialize all vertices with `Integer.MAX_VALUE` except "from" vertex with 0)
- **`paths`** … the shortest path (i.e., the list of vertices on the path) to each vertex (Hint: when visiting vertex *x* and updating the minimum distances to its neighbors - if a new minimum distance is found for neighbor y, then the shortest path to y is *(shortest path to x, followed by y)*

HINTS FOR ASSIGNMENT 04 (AI)

For the implementation of this assignment a method such as the following is recommended:

```
def dijkstra(cur: Vertex, visited_set, distances: dict, paths: dict)
```

- **`cur`** ... current vertex
- **`visited_set`** ... a set which notes already visited vertices
- **`distances`** ... Dictionary (number of vertices entries), which stores the min. distance to each vertex. (Suggestion: initialize all vertices with `sys.maxsize` except "from" vertex with 0)
- **`paths`** … the shortest path (i.e., the list of vertices on the path) to each vertex (Hint: when visiting vertex *x* and updating the minimum distances to its neighbors - if a new minimum distance is found for neighbor y, then the shortest path to y is *(shortest path to x, followed by y)*

GRAPHS

Algorithms and Data Structures 2
Exercise – 2023W

Martin Schobesberger, Markus Weninger, Markus Jäger,
Florian Beck, Achref Rihani

Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at