Graph Theory and Algorithms

Ph.D. Course – Marco Viviani

Walks, Paths, Trails, Cycles, Circuits, Connectivity and related Issues
(April 15, 2021 / 14:30-16:30)

TABLE OF CONTENTS

1

A Quick Recap

Recap of Basic Notions

A Quick Recap

- A **graph** is a pair $G = (V, E)$ of sets such that $E \subseteq [V]^2$; thus, the elements of E are 2-element subsets of V.

$$V = \{v_1, v_2, \dots, v_n\}$$
$$E = \{\{v_i, v_k\}\} \quad i, k \in [1, \dots, n]$$

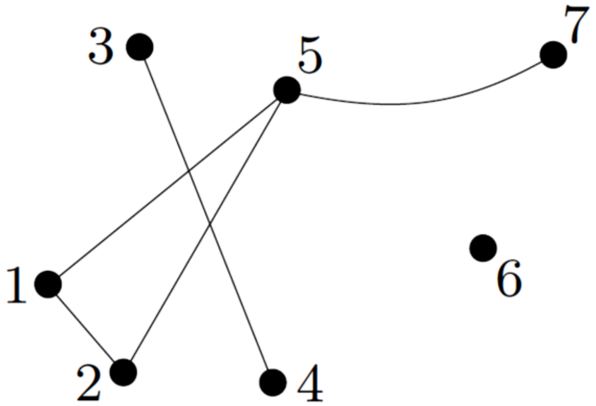
- The elements of V are the **vertices** (or nodes, or points) of the graph G, the elements of E are its **edges** (or lines, or arcs).

- The usual way to **represent a graph** is by drawing a dot for each vertex and joining two of these dots by a line if the corresponding two vertices form an edge.

A Quick Recap ... Cont'd

- The graph G on:

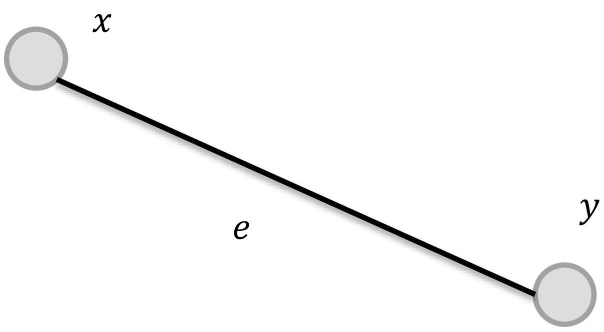
$$V = \{1,\ldots,7\} \text{ with edge set } E = \{\{1,2\},\{1,5\},\{2,5\},\{3,4\},\{5,7\}\}$$

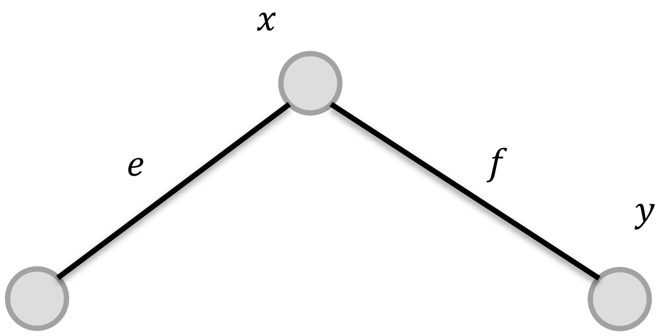


A Quick Recap … Cont'd

- Two **vertices** x, y of G are **adjacent** (or neighbors), if $e = \{x, y\}$ is an edge adjacent of G.

- Two **edges** $e \neq f$ are **adjacent** if they have an end in common.





A Quick Recap ... Cont'd

- **Order of a graph**: its number of vertices $|V|$.
- **Size of a graph**: its number of edges $|E|$.

$$G = (V, E) \rightarrow V = \{1, \ldots, 7\}, E = \{\{1,2\},\{1,5\},\{2,5\},\{3,4\},\{5,7\}\}$$

$$|V| = 7$$

$$|E| = 5$$

2

Some Trivial Definitions

Null and Complete Graphs

Null Graph

- In the mathematical field of graph theory, the term **null graph** may refer either to the **order-zero graph**, or alternatively, to **any edgeless graph**.

- The latter is sometimes called an **empty graph**.

Null Graph (Order-zero Graph)

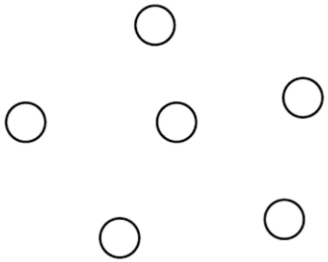
- The **order-zero graph**, denoted as K_0, is the unique graph having no vertices (hence its order is zero).
- It follows that K_0 also has no edges.
- For the order-zero graph $K_0 = G = (\emptyset, \emptyset)$ we simply write $G = \emptyset$.
- A graph of order 0 (or 1) is called **trivial**.

Null Graph (Empty Graph)

- For each natural number n, the edgeless graph (or **empty graph**) $\overline{K_n}$ of order n is the graph with n vertices and zero edges.
- $\overline{K_n} = G = (V, \emptyset)$.

Null Graph (Representations)

- Figure ($\boldsymbol{a}$) illustrates the null (oreder-zero) graph K_0, while ($\boldsymbol{b}$) the null graph (empty graph) $\overline{K_6}$ with six vertices.



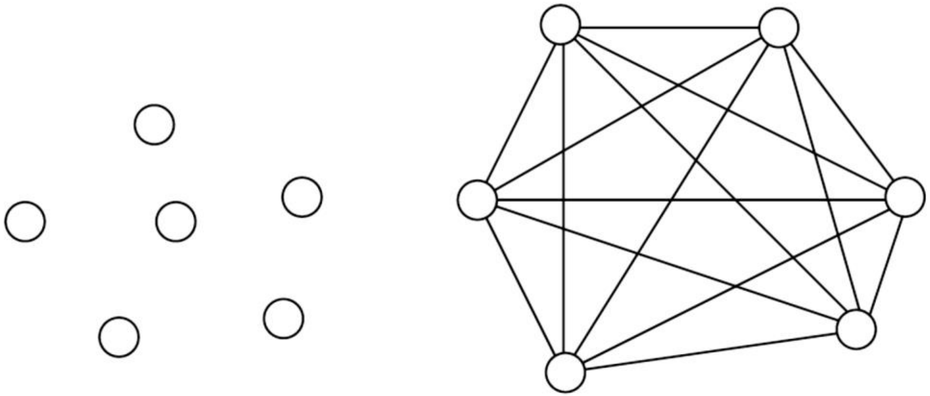
($\boldsymbol{a}$) ($\boldsymbol{b}$)

Complete Graph

- A graph in which each pair of distinct vertices are adjacent is called a **complete graph**.
- A complete graph with n vertices is denoted by K_n.
- K_n contains $\frac{n(n-1)}{2}$ edges.

Complete Graphs … Cont'd

- Figure $(\boldsymbol{b})$ illustrates a complete graph K_6 with six vertices.



$(\boldsymbol{a})$ $(\boldsymbol{b})$

3

Walking on a Graph

Walks, Paths, Trails, Cycles, and Circuits

Walk

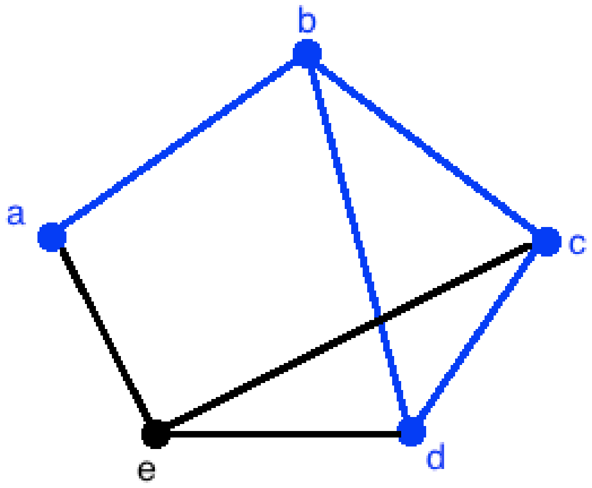
- A **walk** (of length k) in a graph G is a non-empty alternating sequence

$$v_0 e_0 v_1 e_1 \ldots e_{k-1} v_k$$

  of vertices and edges in G such that $e_i = \{v_i, v_{i+1}\}$ for all $i < k$.
- The **length** of a walk is k.

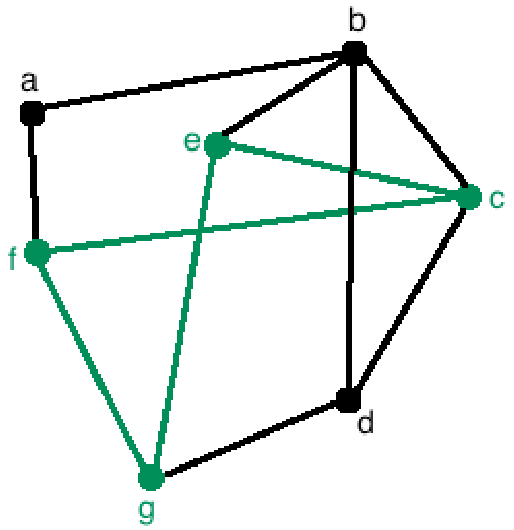
Walk (Example)

- We often refer to a walk by the **natural sequence of its vertices**.
- The walk is denoted as $abcdb$.



Open / Closed Walk

- If the starting vertex is the same as the ending vertex, that is $v_0 = v_k$, the walk is **closed**.
- A walk is considered **open** otherwise.
- $cegfc$ is a closed walk.
- If length of the walk = 0, then it is called as a **trivial walk**.
- Both vertices and edges **can repeat** in a walk whether it is an open or a closed walk.



Path

- A **path** is a non-empty graph $P = (V, E)$ of the form:

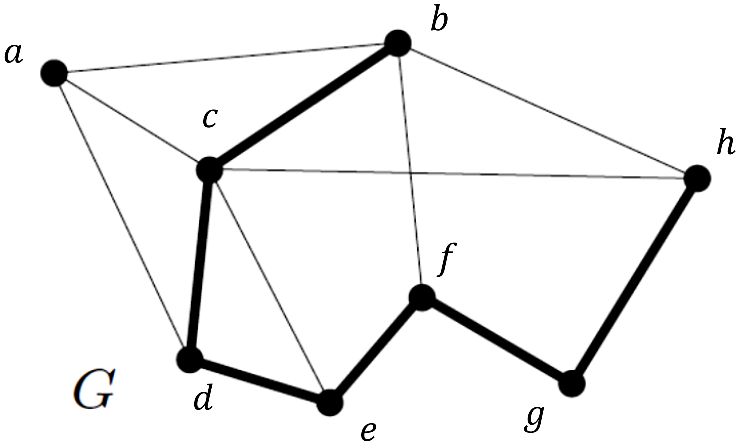
$$V = \{x_0, x_1, \dots, x_k\}$$
$$E = \{\{x_0, x_1\}, \{x_1, x_2\}, \dots, \{x_{k-1}, x_k\}\}$$

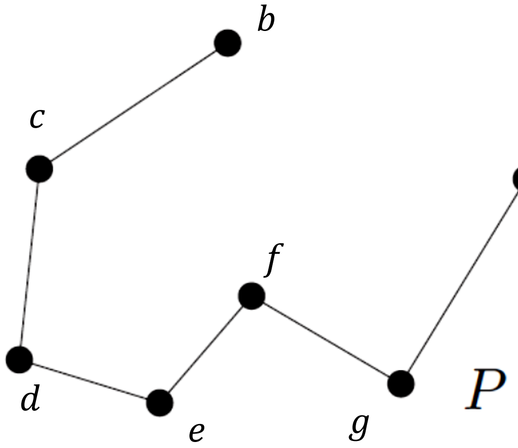
  where the x_i <u>are all distinct</u>.

- The vertices x_0 and x_k are called the **end-vertices** or **ends** of P.
- The vertices $x_1, \dots, x_{k-1}$ are the **inner vertices** of P.

Path (Example)

- A path $P = P^6$ in G





- $P(V, E) \rightarrow V = \{b, c, d, e, f, g, h\}, E = \{\{b, c\}, \{c, d\}, \{d, e\}, \{e, f\}, \{f, g\}, \{g, h\}\}$

Path (A Simpler Definition)

In graph theory, a **path** is defined as an open walk in which:

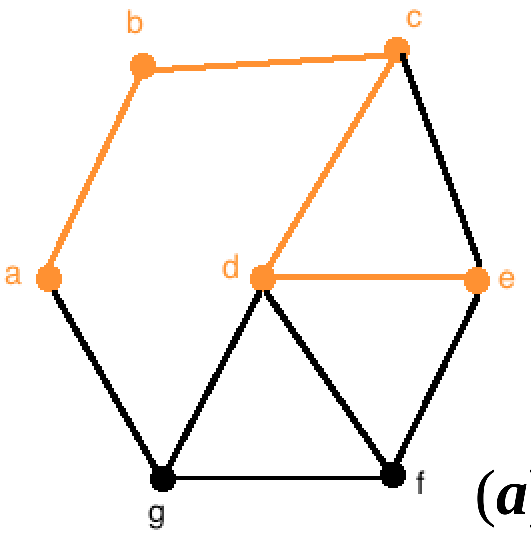
- Neither vertices are allowed to repeat.
- Nor edges are allowed to repeat.

Path … Cont'd

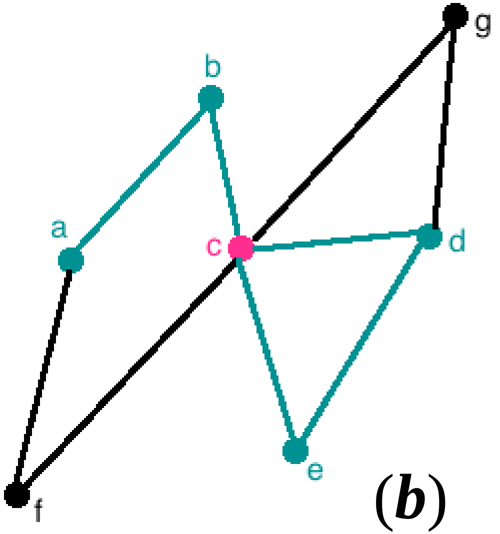
- The number of edges of a path is its **length**.
- The path of length k is denoted by P^k.
- We often refer to a path by the **natural sequence of its vertices**, writing, say, $P = x_0 x_1 \dots x_k$, and calling P a path from x_0 to x_k (as well as between x_0 and x_k).
  - More precisely, by one of the two natural sequences: $x_0 x_1 \dots x_k$ and $x_k x_{k-1} \dots x_0$, we denote the same path.

Path (Example)

- A path $abcde$ ($\boldsymbol{a}$) and … what about $abcdec$ ($\boldsymbol{b}$)?



$(\boldsymbol{a})$



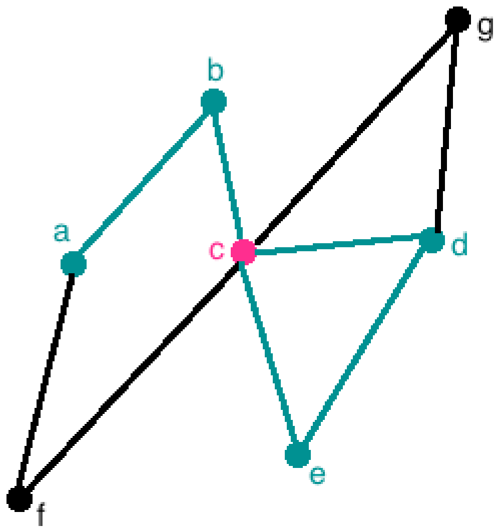
$(\boldsymbol{b})$

Trail

In graph theory, a **trail** is defined as an open walk in which:

- Vertices may repeat.
- Edges are not allowed to repeat.

- $abcdec$ is a trail.



Weight of a Walk (a Path, a Trail)

- **RECAP**: a **weighted graph** associates a value (weight) with every edge in the graph.
- The **weight of a walk** (or trail or path) in a weighted graph is the sum of the weights of the traversed edges.
- Sometimes the words **cost**, or **length**, are used instead of weight.

Directed Walk, Path, Trail

- A **directed walk** is a sequence of edges directed in the same direction which joins a sequence of vertices.
- A **directed path** is a directed walk in which all vertices are distinct.
- A **directed trail** is a directed walk in which all edges are distinct.
- A **weighted directed graph** associates a value (weight) with every edge in the directed graph.
- The **weight of a directed walk** (or trail or path) in a weighted directed graph is the sum of the weights of the traversed edges.

Cycle

A possible formal definition

- If $P = x_0 \ldots x_{k-1}$ is a path and $k \geq 3$, then the graph $C = P + x_{k-1}x_0$ is called a **cycle**.

More simply… In graph theory, a **cycle** is defined as a closed walk in which:

- Neither vertices (except possibly the starting and ending vertices) are allowed to repeat.
- Nor edges are allowed to repeat.

Cycle … Cont'd

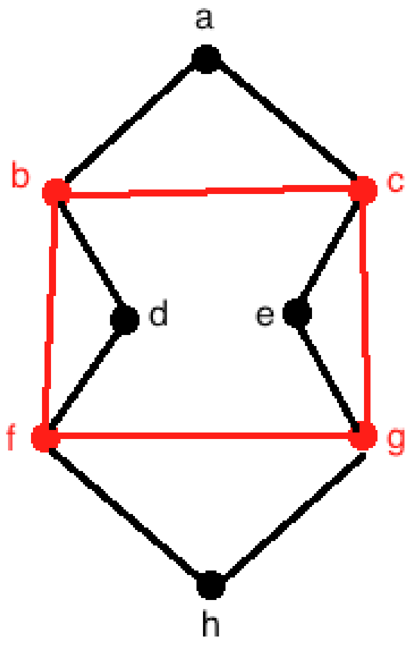
- As with paths, we often denote a cycle by its **(cyclic) sequence of vertices**.
- A cycle C might be written as $x_0 \ldots x_{k-1} x_0$.
- The **length of a cycle** is its number of edges (or vertices).
- The cycle of length k is called a k-cycle and denoted by C^k.

Cycle … Cont'd

- The **minimum length of a cycle** (contained) in a graph G is the **girth** (*calibro*) $g(G)$ of G.
- The **maximum length of a cycle** in G is its **circumference** $c(G)$.
- If G does not contain a cycle, we set the former to ∞, the latter to zero.
  - $g(G) = \infty$
  - $c(G) = 0$

Cycle (Example)

- The closed walk $bcgf$ is a cycle.



Cycle ... Cont'd

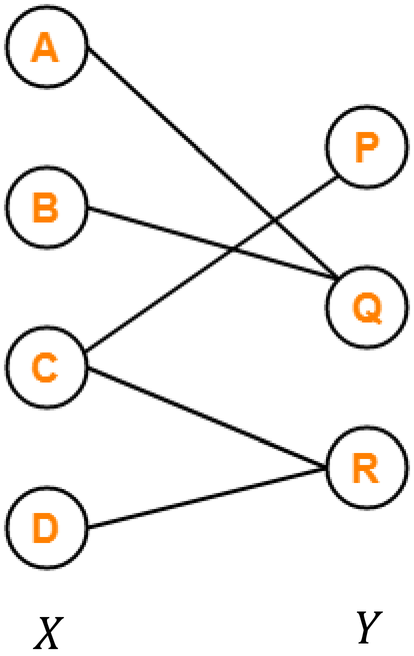
- A **cycle is odd** if its length is odd.
- A **cycle is even** if its length is even.

Bipartite Graps and Cycles

RECAP: In graph theory, a **bipartite graph** is a graph where:

- Vertices can be divided into two disjoint and independent sets X and Y.
- Such that every edge connects a vertex in X to one in Y.
- None of the vertices belonging to the same set join each other.

RECAP: A **complete bipartite graph** (or biclique) is a special kind of bipartite graph where every vertex of the first set is connected to every vertex of the second set.



Bipartite Graps and Cycles … Cont'd

- Bipartite graphs can be characterized in terms of **odd cycles** as follows.
- A graph G is **bipartite** if and only if G **does not contain any odd cycle**.
- Visual demonstration.

Circuit

In graph theory, a **circuit** is defined as a closed walk in which:

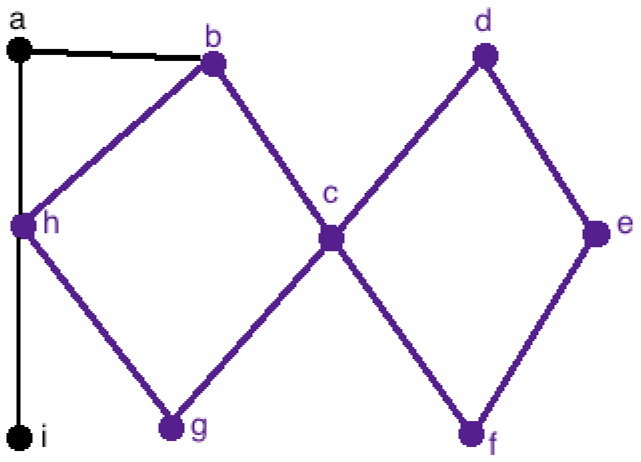
- Vertices may repeat.
- But edges are not allowed to repeat.

OR

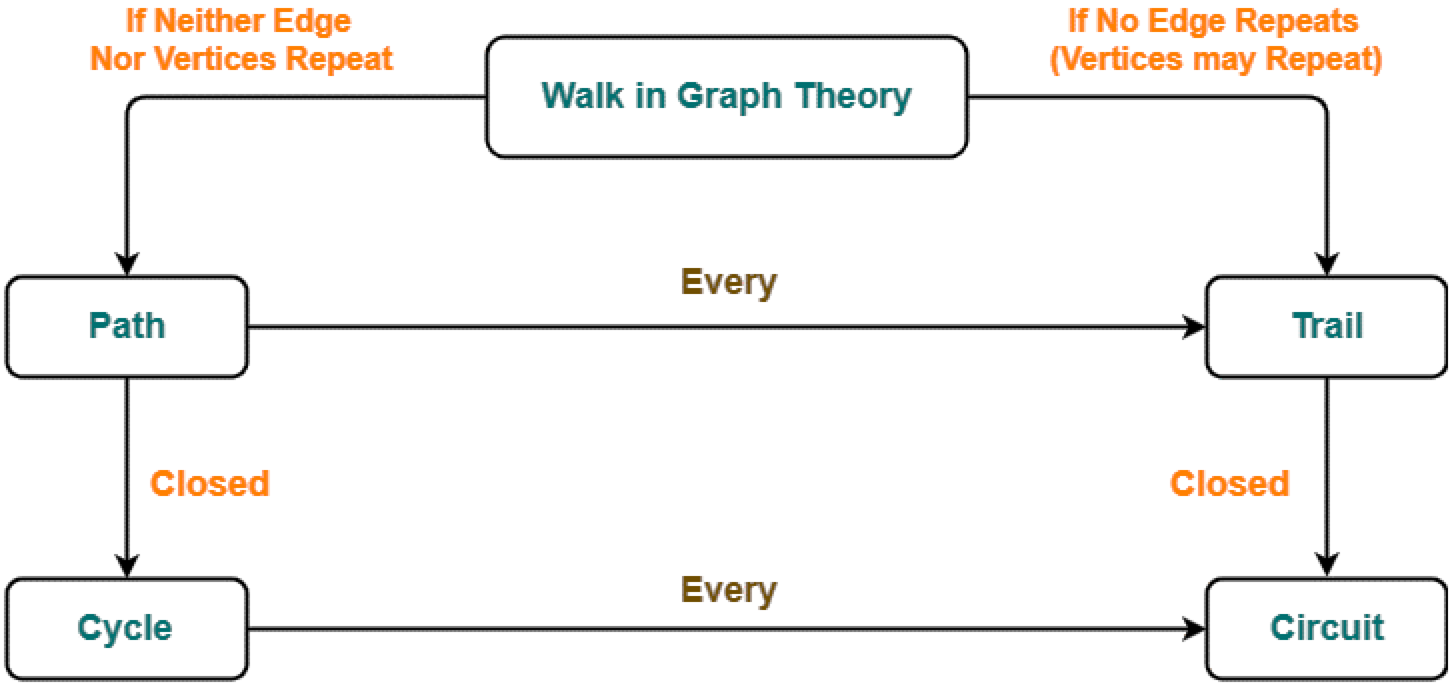
- In graph theory, a closed trail is called as a **circuit**.

Circuit (Example)

- There are no edges repeated in the walk $hbcdefcgh$, hence the walk is certainly a trail and, since it is closed, it is a circuit.



To recap...

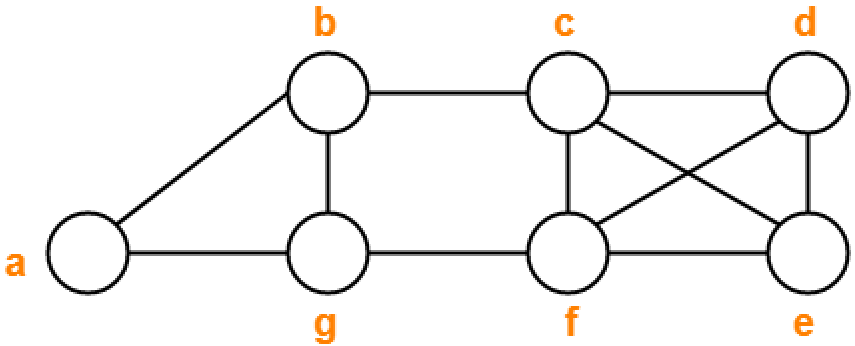

If Neither Edge
Nor Vertices Repeat
Walk in Graph Theory
If No Edge Repeats
(Vertices may Repeat)
Path
Every
Trail
Closed
Closed
Cycle
Every
Circuit

Exercises

- Consider the graph in the figure.

- For those sequences of vertices that are walks, decide whether they are a path, a trail, a cycle or a circuit.

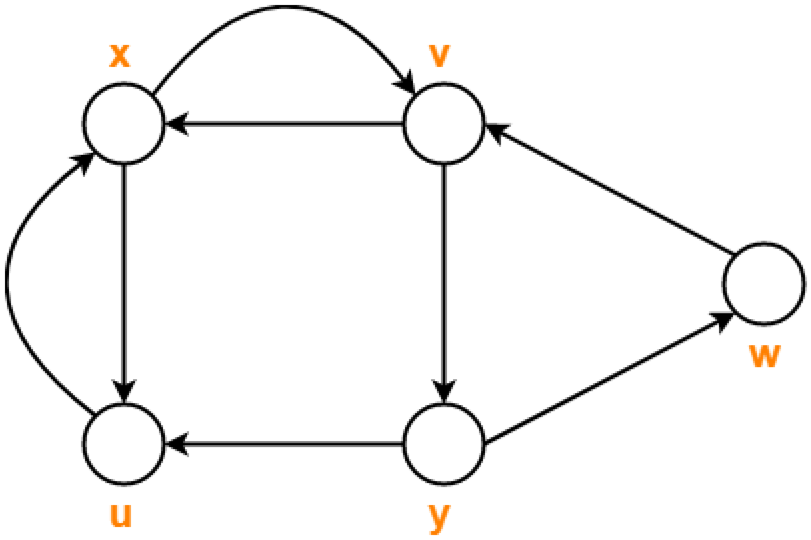
  - a , b , g , f , c , b *Trail*
  - b , g , f , c , b , g , a *Walk*
  - c , e , f , c *Cycle*
  - c , e , f , c , e *Walk*
  - a , b , f , a *Not a walk*
  - f , d , e , c , b *Path*
  - b, g, f, c, e, d, c, b *Circuit*



Exercises … Cont'd

- Consider the following sequences of vertices:
  a. x, v, y, w, v
  b. x, u, x, u, x
  c. x, u, v, y, x
  d. x, v, y, w, v, u, x

- Which are directed walks? a. and b.
- What are the lengths of directed walks? 4
- Which directed walks are also directed paths? none
- Which directed walks are also directed cycles? none



4

Algorithms

Dijkstra’s and Floyd-Warshall algorithms, Random Walks

Finding Paths

- Several algorithms exist to find **shortest and longest paths** in graphs, with the important distinction that the former problem is computationally much easier than the latter.

- The **longest path problem** is the problem of **finding a path of maximum length** between two vertices in a given graph.

- The **shortest path problem** is the problem of **finding a path of minimum length** between two vertices in a given graph.

- The **length of a path** may either be measured by its number of edges, or (in weighted graphs) by the sum of the weights of its edges.

Longest and Shortest Paths (Complexity)

- The **longest path problem is NP-hard** and the decision version of the problem, which asks whether a path exists of at least some given length, is NP-complete.
  - However, it has a **linear time solution** for **Directed Acyclic Graphs**, which has important applications in finding the critical path in scheduling problems.

- The **shortest path problem** can be solved in **polynomial time** in graphs without negative-weight cycles.

Shortest Path Problems

- The **Single-Source Shortest Path (SSSP)** problem consists of finding the shortest paths between a given vertex v and all other vertices in the graph.
  - Algorithms such as **Breadth-First-Search (BFS)** for unweighted graphs or **Dijkstra's** solve this problem.
- The **All-Pairs Shortest Path (APSP)** problem consists of finding the shortest path between all pairs of vertices in the graph.
  - To solve this second problem, one can use the **Floyd-Warshall algorithm** or apply the **Dijkstra's algorithm** to each vertex in the graph.

The Dijkstra's Algorithm

- The **Dijkstra's algorithm** works **only for connected (directed or undirected) graphs**.
- Dijkstra algorithm works only for those graphs that **do not contain any negative weight edge**.
- The actual Dijkstra's algorithm **does not output the shortest paths**.
  - It only provides the value or cost of the shortest paths.
  - By making minor modifications in the actual algorithm, the shortest paths can be easily obtained.

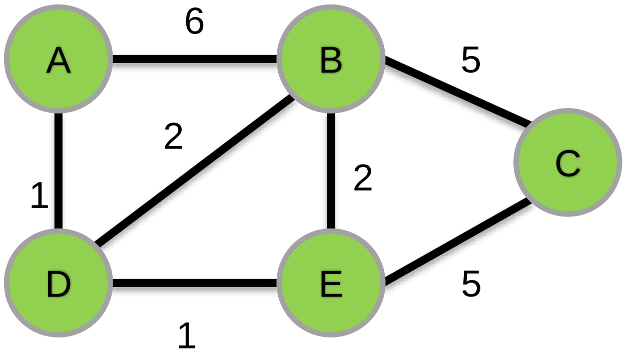
Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, *1*(1), 269-271.

Basics of Dijkstra's Algorithm

- Dijkstra's Algorithm **starts with a source node**, and it analyzes the graph to find the shortest path between that node and all the other nodes in the graph.
- The algorithm keeps track of the currently known shortest distance from each node to the source node and it **updates** these values if it finds a shorter path.
- Once the algorithm has found the shortest path between the source node and another node, that node is marked as **"visited"** and added to the path.
- The process continues until all the nodes in the graph have been **added to the path**. This way, we have a path that connects the source node to all other nodes following the shortest path possible to reach each node.

Dijkstra's Algorithm – Example

- Let us consider a graph with weighted edges.
- This graph can either be directed or undirected.
- Here we will use an undirected graph.



Dijkstra's Algorithm – Initialization

- Let $\boldsymbol{s}$ the node at which we are starting be called the **start vertex**.

For each vertex of the given graph, two variables are defined as:

- $\boldsymbol{\Pi}[v]$ which denotes the **predecessor** of vertex v
- $\boldsymbol{d}[v]$ which denotes the **shortest distance** of vertex v from the source vertex.

Furthermore:

- Create a set $\boldsymbol{Q}$ of all the unvisited nodes called the **unvisited set**.

Dijkstra's Algorithm – Initialization

Dijkstra's algorithm will assign **some initial values** and will try to improve them step by step.

Initially, the value of the considered variables is set as:

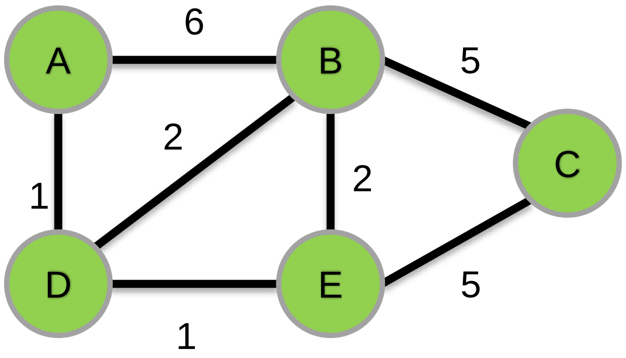
- The value of variable 'Π' for each vertex is set to NIL i.e., $\boldsymbol{\Pi[v] = \text{NIL}}$
- The value of variable 'd' for source vertex is set to 0 i.e., $\boldsymbol{d[s] = 0}$
- The value of variable 'd' for remaining vertices is set to ∞ i.e., $\boldsymbol{d[v] = \infty}$

Furthermore:

- Mark all nodes as unvisited, i.e., $\boldsymbol{Q = V}$.

Dijkstra's Algorithm – Running Example (Start)

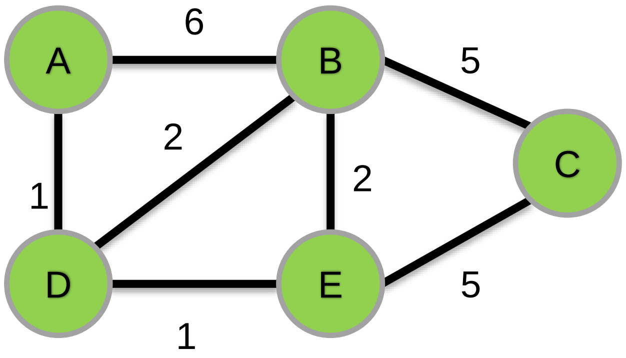
- $Q = V = \{A, B, C, D, E\}$
- $d[A] = 0,\ d[B] = d[C] = d[D] = d[E] = \infty$
- $\Pi[A] = \Pi[B] = \Pi[C] = \Pi[D] = \Pi[E] = \mathrm{NIL}$



Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

Dijkstra's Algorithm – Running Example (Cont'd)

- Visit the unvisited vertex with the smallest distance from the start vertex.

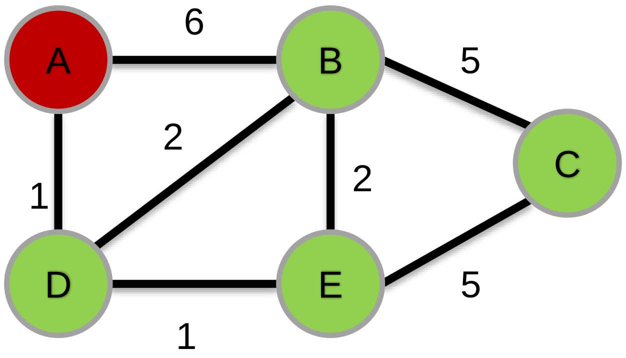


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- Visit the unvisited vertex with the smallest distance from the start vertex.
  - *The first time, it is the start vertex itself.*

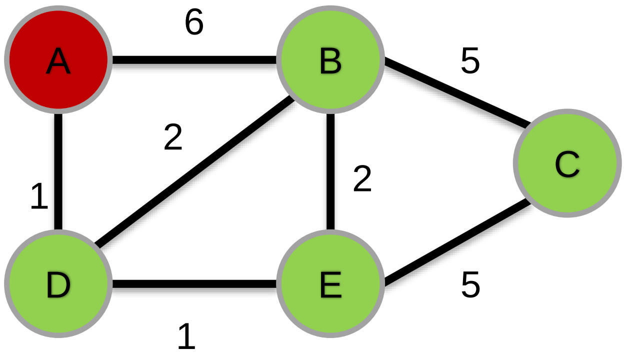


Vertex	Shortest distance from A	Previous vertex
A	$\boxed{0}$	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.

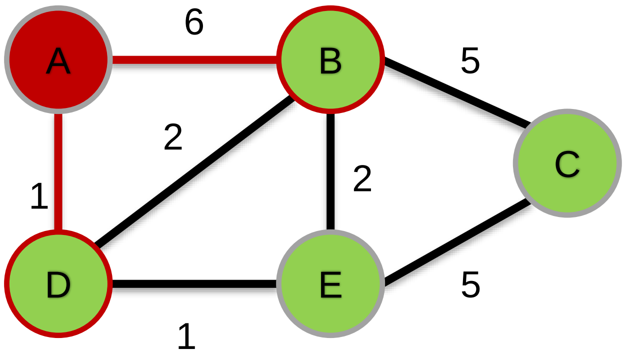


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.
  - *Its unvisited neighbors are B and D.*

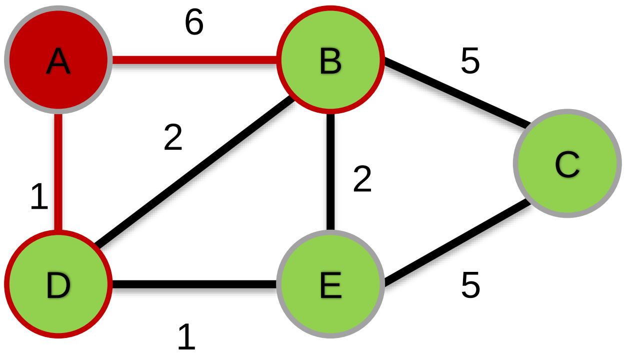


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, calculate the distance of each neighbor from the start vertex.
  - I.e., $d[A] + dist(A, B)$, $d[A] + dist(A, D)$

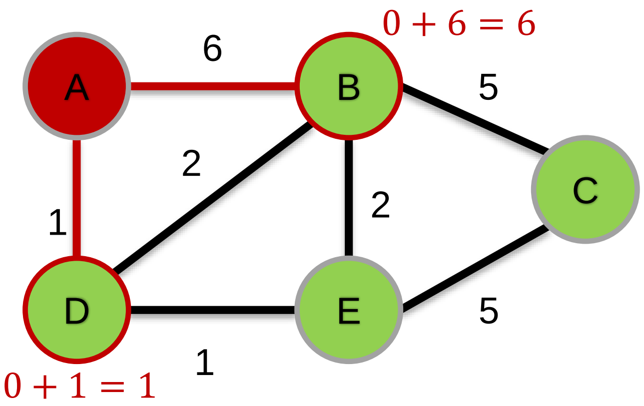


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, calculate the distance of each neighbor from the start vertex.
  - I.e., $d[A] + dist(A, B)$, $d[A] + dist(A, D)$

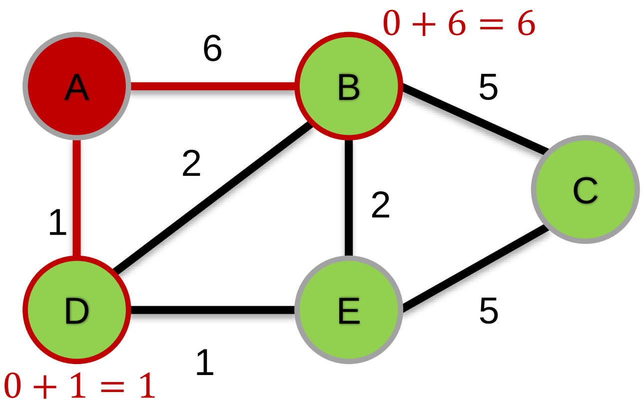


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- If the calculated distance is less then the know distance for the neighbors, update the shortest distance.
  - E.g, if $d[A] + dist(A, B) < d[B] \rightarrow d[B] = d[A] + dist(A, B)$

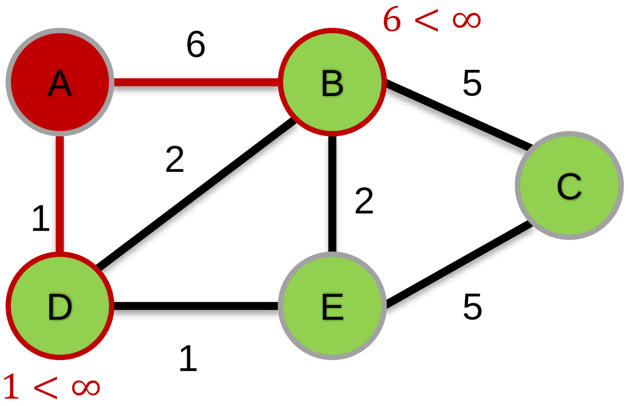


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	∞	NIL
C	∞	NIL
D	∞	NIL
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- If the calculated distance is less then the know distance for the neighbors, update the shortest distance.
  - E.g, if $d[A] + dist(A, B) < d[B] \rightarrow d[B] = d[A] + dist(A, B)$

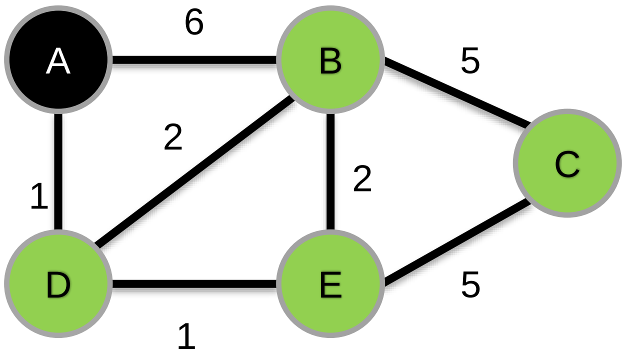


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	**6**	**A**
C	∞	NIL
D	**1**	**A**
E	∞	NIL

- $Q = \{A, B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- When we are done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

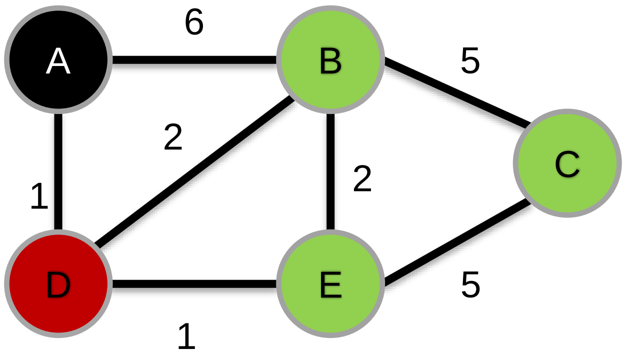


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	6	A
C	∞	NIL
D	1	A
E	∞	NIL

- $Q = \{B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- Visit the unvisited vertex with the smallest distance from the start vertex.
  - *This time, the vertex is D.*

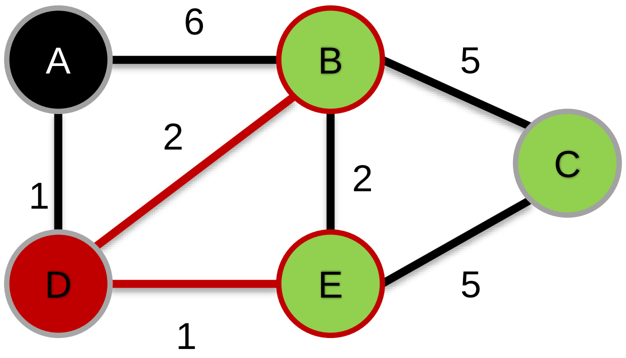


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	6	A
C	∞	NIL
D	$\boxed{1}$	A
E	∞	NIL

- $Q = \{B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.
  - *Its unvisited neighbors are B and E.*

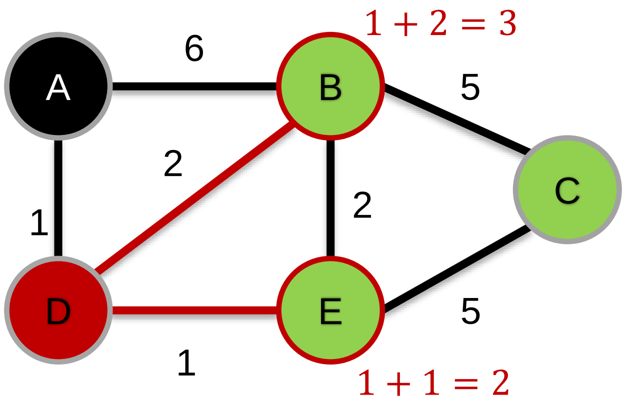


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	6	A
C	∞	NIL
D	1	A
E	∞	NIL

- $Q = \{B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, calculate the distance of each neighbor from the start vertex.
  - I.e., $d[D] + dist(D, B)$, $d[D] + dist(D, E)$

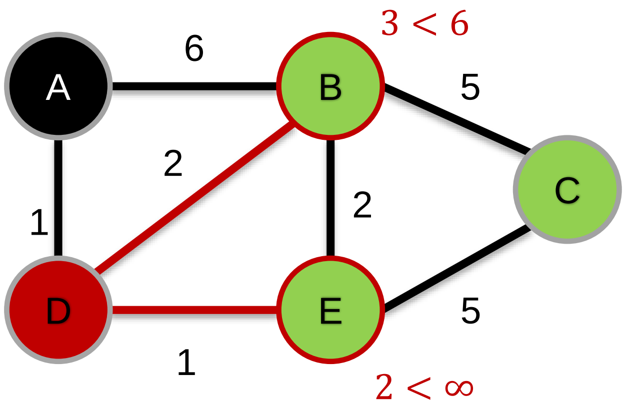


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	6	A
C	∞	NIL
D	1	A
E	∞	NIL

- $Q = \{B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- If the calculated distance is less then the know distance for the neighbors, update the shortest distance.
  - E.g, if $d[D] + dist(D, B) < d[B] \rightarrow d[B] = d[D] + dist(D, B)$

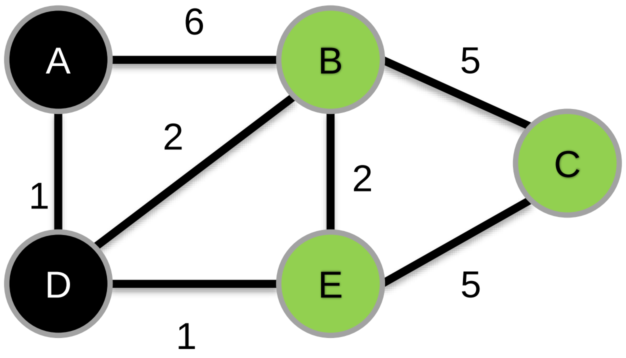


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	∞	NIL
D	1	A
E	2	D

- $Q = \{B, C, D, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- When we are done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

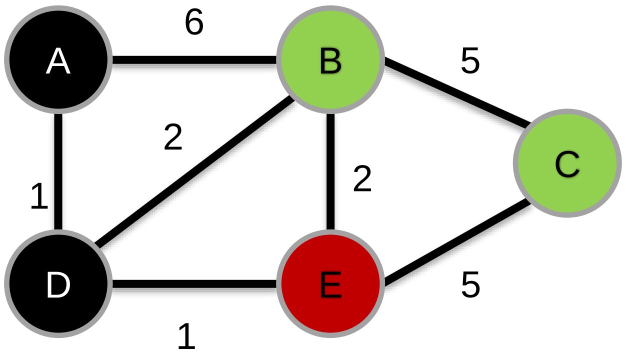


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	∞	NIL
D	1	A
E	2	D

- $Q = \{B, C, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- Visit the unvisited vertex with the smallest distance from the start vertex.
  - *This time, the vertex is E.*

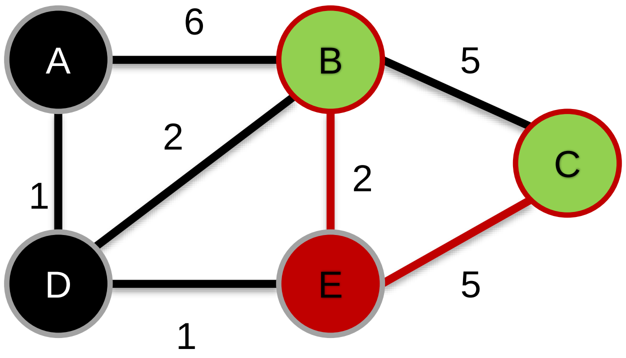


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	∞	NIL
D	1	A
E	$\boxed{2}$	D

- $Q = \{B, C, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.
  - *Its unvisited neighbors are B and C.*

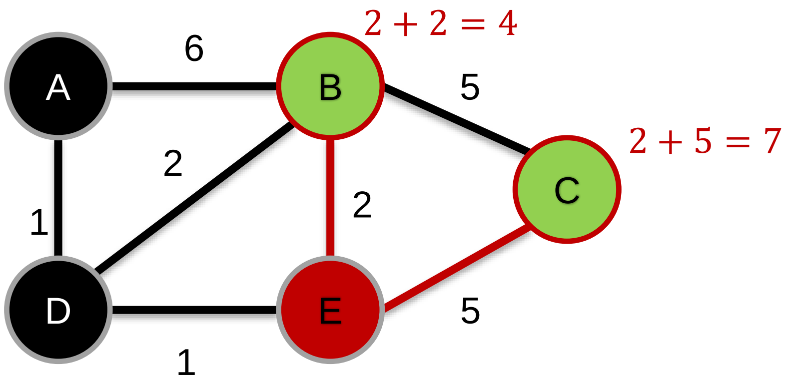


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	∞	NIL
D	1	A
E	2	D

- $Q = \{B, C, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, calculate the distance of each neighbor from the start vertex.
  - I.e., $d[E] + dist(E, B)$, $d[E] + dist(E, C)$

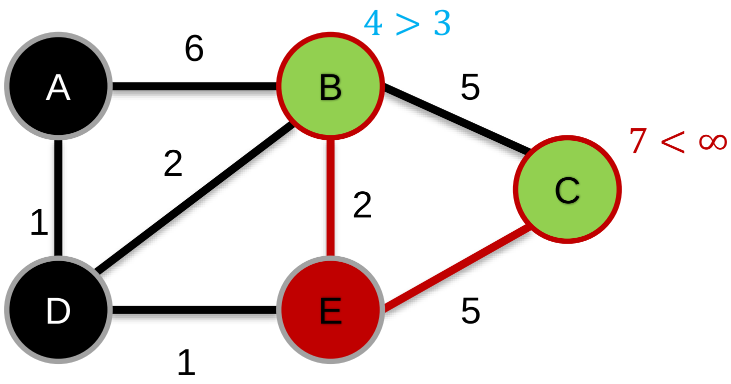


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	∞	NIL
D	1	A
E	2	D

- $Q = \{B, C, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- If the calculated distance is less then the know distance for the neighbors, update the shortest distance.
  - E.g, if $d[E] + dist(E,B) < d[B] \rightarrow d[B] = d[E] + dist(E,B)$

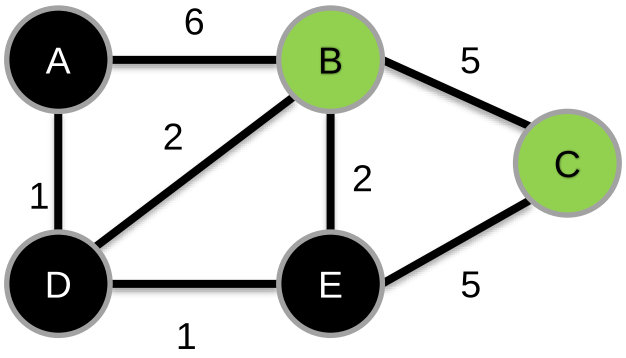


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{B, C, E\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- When we are done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

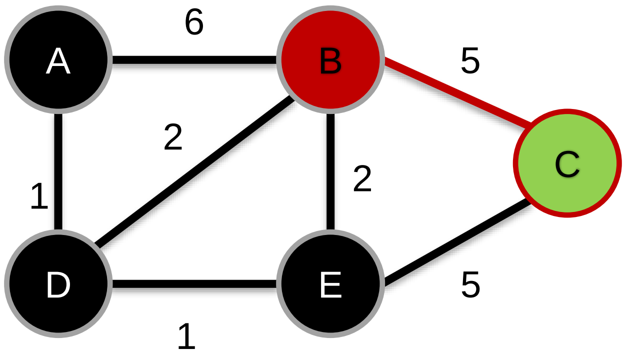


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{B, C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.
  - *Its only unvisited neighbor is C.*

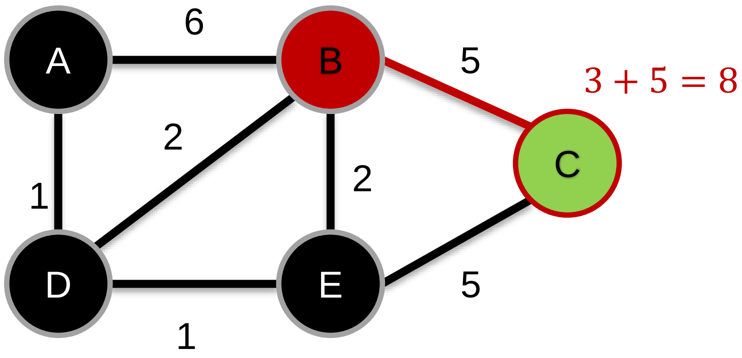


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{B, C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, calculate the distance of each neighbor from the start vertex.
  - I.e., $d[B] + dist(B, C)$

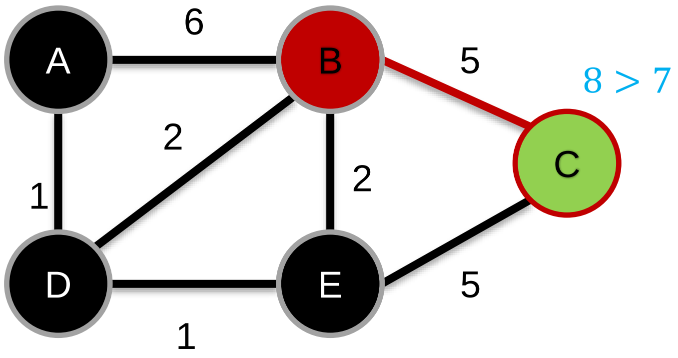


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{B, C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- If the calculated distance is less then the know distance for the neighbors, update the shortest distance.
  - E.g, if $d[B] + dist(B, C) < d[C] \rightarrow d[C] = d[B] + dist(B, C)$

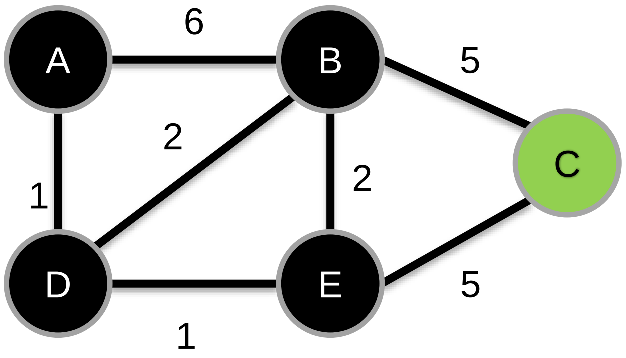


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{B, C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- When we are done considering all the unvisited neighbors of the current node, mark the current node as visited and remove it from the unvisited set. A visited node will never be checked again.

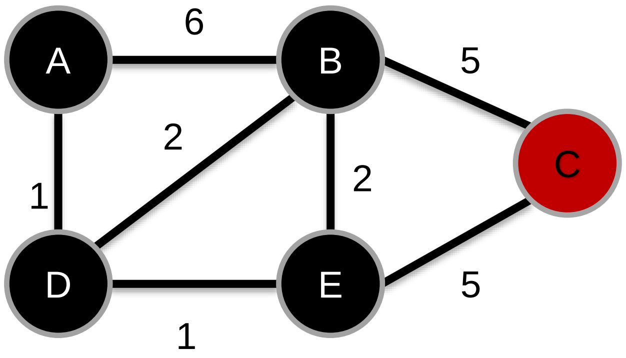


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- Visit the unvisited vertex with the smallest distance from the start vertex.
  - *This time, the vertex is C.*

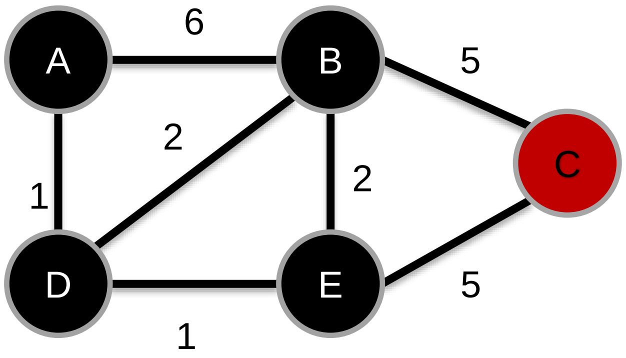


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	$\boxed{7}$	E
D	1	A
E	2	D

- $Q = \{C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- For the current vertex, examine its unvisited neighbors.
  - *NO unvisited neighbors.*

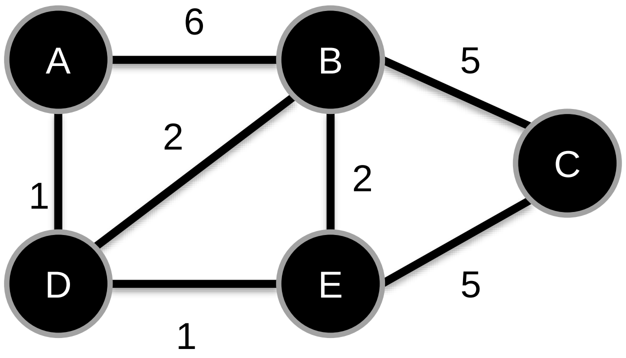


Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

- $Q = \{C\}$

Dijkstra's Algorithm – Running Example (Cont'd)

- Remove the current vertex from the list of unvisited vertices.

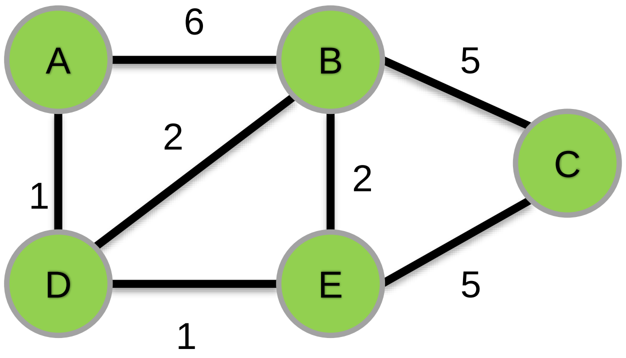


- $Q = \{\}$

Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

Dijkstra's Algorithm – Running Example (Cont'd)

- We have the shortest distance from A to every other vertex



Vertex	Shortest distance from A	Previous vertex
A	0	NIL
B	3	D
C	7	E
D	1	A
E	2	D

Dijkstra's Algorithm – Pseudocode

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:
        dist[v] ← INFINITY
        prev[v] ← NIL
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]

        remove u from Q

        for each neighbor v of u:           // only v that are still in Q
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

The Floyd-Warshall Algorithm

- The **Floyd-Warshall algorithm** is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph.
- This algorithm works for both the **directed** and **undirected** weighted graphs.
- It works for graphs with positive or negative edge weights, but it does not work for the **graphs with negative cycles** (where the sum of the edges in a cycle is negative).

Floyd, R. W. (1962). Algorithm 97: shortest path. *Communications of the ACM*, *5*(6), 345.

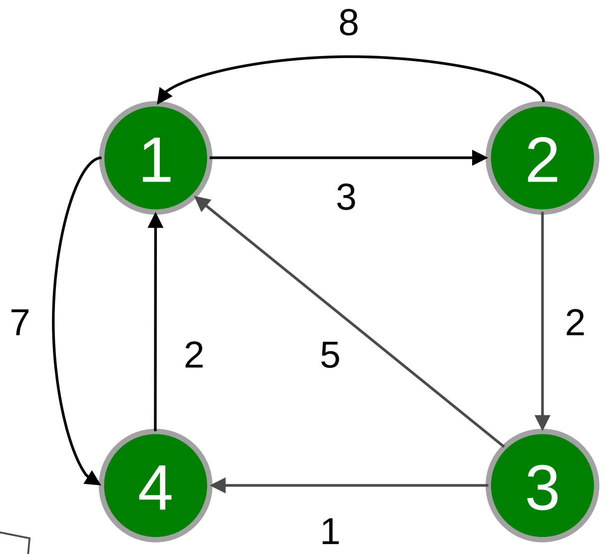
Floyd-Warshall Algorithm – Step 1

- Create an **adjacency matrix** A^0 of dimension $n * n$ where n is the number of vertices. The row and the column are indexed as i and j respectively.

- Each cell $A^0[i][j]$ is filled with the **weight** on the edge from the ith vertex to the adjecent jth vertex.

- If the ith vertex and the jth vertex are **not adjacent**, the value of the cell is left as **infinity**.

Floyd-Warshall Algorithm – Step 1 (Example)



$$A^0 = \begin{array}{c|c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array}$$

Floyd-Warshall Algorithm – Step 2

- Now, create a matrix $\boldsymbol{A^1}$ using matrix A^0.

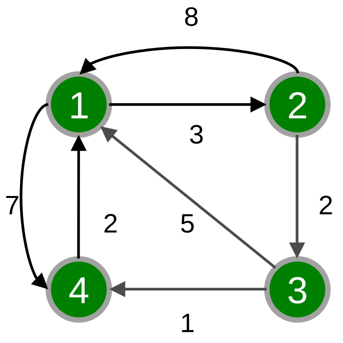
- The elements in the first column and the first row are left as they are.

- The remaining cells are filled in the following way:
  - In this step, k is vertex 1. We calculate the distance from source vertex to destination vertex through this vertex k.
  - $A^1[i][j]$ is filled with $(A^0[i][k] + A^0[k][j])$ if $(A^0[i][j] > A^0[i][k] + A^0[k][j])$.

- That is, if the direct distance from the source to the destination is greater than the path through the vertex k, then the cell is filled with $A[i][k] + A[k][j]$.

Floyd-Warshall Algorithm – Step 2 (Example)

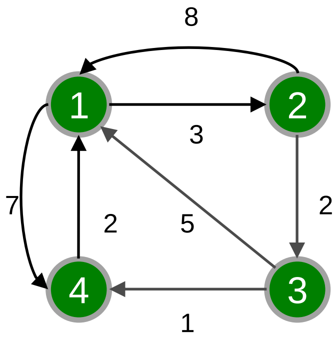
- $A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$
- $\boldsymbol{A^1[2][3]} = \min(\boldsymbol{A^0[2][3]}, \boldsymbol{A^0[2][1]} + \boldsymbol{A^0[1][3]})$



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix} \end{array}$$

Floyd-Warshall Algorithm – Step 2 (Example)

- $A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$
- $\boldsymbol{A^1[2][4]} = \min(\boldsymbol{A^0[2][4]}, \boldsymbol{A^0[2][1]} + \boldsymbol{A^0[1][4]})$



$$A^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & \infty \\ 5 & \infty & 0 & 1 \\ 2 & \infty & \infty & 0 \end{bmatrix} \end{array} \qquad A^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & & \\ 5 & & 0 & \\ 2 & & & 0 \end{bmatrix} \end{array} \Rightarrow \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 8 & \infty & 0 \end{bmatrix} \end{array}$$

Floyd-Warshall Algorithm – Further Steps

- The algorithm is applied until $k = n$ (number of vertices)
- Pseudocode:

```
n = no of vertices
A = matrix of dimension n*n
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            A^k[i,j] = min(A^(k-1)[i,j], A^(k-1)[i,k] + A^(k-1)[k, j])
return A
```

Floyd-Warshall Algorithm – Further Steps (Examples)

$$A^1 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & \infty & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & \infty & 0 \end{bmatrix} \end{array} \qquad A^2 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & & \\ 8 & 0 & 2 & 15 \\ & 8 & 0 & \\ & 5 & & 0 \end{bmatrix} \end{array} \qquad \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$$

Floyd-Warshall Algorithm – Further Steps (Examples)

$$A^2 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 5 & 7 \\ 8 & 0 & 2 & 15 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array} \qquad A^3 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & & 5 & \\ & 0 & 2 & \\ 5 & 8 & 0 & 1 \\ & & 7 & 0 \end{bmatrix} \end{array} \Rightarrow \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}$$

$$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$$

Floyd-Warshall Algorithm – Further Steps (Examples)

$$
A^3 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 7 & 0 & 2 & 3 \\ 5 & 8 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}
\qquad
A^4 = \begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & & & 6 \\ & 0 & & 3 \\ & & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}
\Rightarrow
\begin{array}{c c} & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 3 & 5 & 6 \\ 5 & 0 & 2 & 3 \\ 3 & 6 & 0 & 1 \\ 2 & 5 & 7 & 0 \end{bmatrix} \end{array}
$$

$$A^k[i,j] = \min(A^{k-1}[i,j], A^{k-1}[i,k] + A^{k-1}[k,j])$$

Dijkstra's VS Floyd–Warshall

- **Dijkstra's algorithm** is one example of a single-source shortest or SSSP algorithm, i.e., given a source vertex it finds shortest path from source to all other vertices.
- **Floyd Warshall algorithm** is an example of all-pairs shortest path algorithm, meaning it computes the shortest path between all pair of nodes.

Dijkstra's VS Floyd–Warshall ... Cont'd

- Time Complexity of Dijkstra's Algorithm: $O(E \log V)$
- Time Complexity of Floyd-Warshall: $O(V^3)$
- We can use Dijskstra's shortest path algorithm for finding all pair shortest paths by running it for every vertex. But time complexity of this would be $O(VE \log V)$ which can go $(V^3 \log V)$ in worst case.

Random Walk - Origins

- The concept of **random walk** was firstly introduced by Pearson in 1905 [1].
- Spitzer [2] gives a complete review of random walks for mathematical researchers and clearly presents the mathematical principles of random walks.

[1] Pearson, K. (1905). The problem of the random walk. Nature, 72(1867), 342-342.
[2] Spitzer, F. (2013). Principles of random walk (Vol. 34). Springer Science & Business Media.

Classical Random Walks

- A random walk is known as a **random process**.
- It describes a walk consisting of a **succession of random steps** on some mathematical space, which can be denoted as

$$\{\xi_t, t = 0, 1, 2, \dots\}$$

- ξ_t is a **random variable** describing the position of a random walk after t steps.
- The sequence can also be regarded as **a special category of Markov chain**.

Random Walk Agorithms

- A **random walk algorithm** provides random walks in a graph.
- A random walk start at one node, choose a neighbor to navigate to at random or based on a provided probability distribution, and then do the same from that node, keeping the resulting walk in a list.
  - *It's similar to how a drunk person traverses a city.*

Random Walk Agorithms ... Cont'd

- From the perspective of graph representation, let $G = (V, E)$ be a connected graph, where V is the vertex set and E is the edge set.
- The **adjacency matrix** of G is denoted as $A \in \mathrm{R}^{n \times n}$, where n is the number of nodes in G.
- A_{ij} denotes the weight of edge from the node i to the node j.
- The **transition probability** (single step) from node i to node j on the graph can be defined as:

$$p_{ij} = \frac{A_{ij}}{\sum_{j \in \mathrm{V}} A_{ij}}$$

5

Connectivity (next lesson)

Eulerian and Hamiltonian Graphs, The Travelling Salesperson Problem

6

Possible Assignements

Some Possible Assignements

- Discuss the linear time solution for **longest path detection** in Directed Acyclic Graphs.
- Discuss the **PageRank algorithm** (which is based on Random Walks).
- Discuss a specific solution to the **Travelling Salesperson Problem** (*Next Lesson*).
- *You can either present and discuss one of the above-mentioned problems, and/or present an implementation of the algorithm.*