Week 9
Graph Algorithms

LM Data Structures, Algorithms, and Databases (34141)

Dr Ahmad Ibrahim
a.ibrahim@bham.ac.uk
March 11, 2024

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

Slides by Martin Escardo

Topics by Week

Week	Date	Topic
1	15 Jan	Searching algorithms
2	22 Jan	Binary Search Tree
3	29 Jan	Balancing Trees – AVL Tree
4	5 Feb	Databases – Conceptual Design
5	14 Feb	Databases – Logical Design & Relational Algebra
6	19 Feb	Consolidation Week
7	26 Feb	Complexity analysis, Stacks, Queues, Heaps
8	4 Mar	Sorting Algorithms, Hash tables
9	11 Mar	Graph Algorithms
10	18 Mar	Databases – Normalization
		Easter break and Eid break
11	22 Apr	Databases – Concurrency
12	29 Apr	Assessment Support Week

Timetable & Office hours

Day	Time	Event	Location
Monday	4:00-5:00pm	Online support session*	Online*
Tuesday	4:00-5:00pm	Office hour 1 (by appointment)*	Online*
Wednesday	-	-	-
Thursday	4:00-5:00pm	Office hour 2 (by appointment)*	Online*
			Auditorium
			Auditorium

Ramadan Timetable
March 17th and March 24th 2024
Sunday 1:00-4:00pm

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

***Zoom link:** https://bham-ac-uk.zoom.us/j/81310444523?pwd=T01tZlZGdmdUL2lkeHZsVFpjcWxUUT09

Assessments

Assessments (Test 1, Test 2, Test 3): **20%**

Exam: **80%**

Week 10

Test 3

Not available until 20 Mar at 16:00 | Due 21 Mar at 16:00 | -/20 pts

Important Note (Ramadan)

4:00pm UAE Time

(12:00 noon UK Time)

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

Past Exam papers and Week 12

Past exam papers are available at:

https://canvas.bham.ac.uk/courses/73245

Week 12 (assessment support week)

LM Data Structures, Algorithms, and Databases (34141)
Date: Monday 29th April
Time: 6:00-8:00pm
Venue: Auditorium

Recap

Last Week

- Sorting Algorithm
    - Insertion Sort
    - Heapsort
    - Merge sort (Divide and conquer)
    - Quick Sort (Divide and conquer)
- HashTables

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

This Week

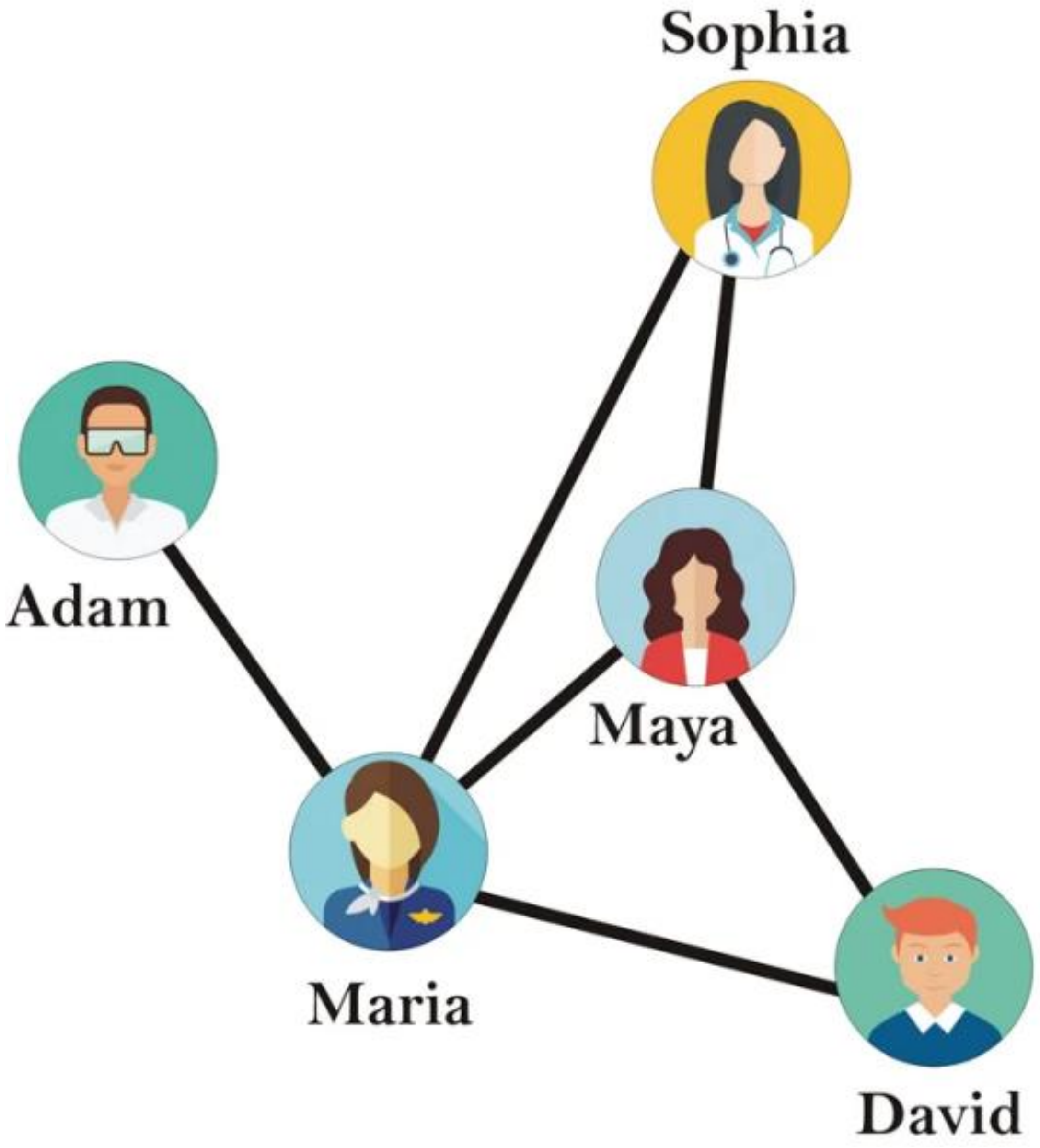
Graphs

Graph representation

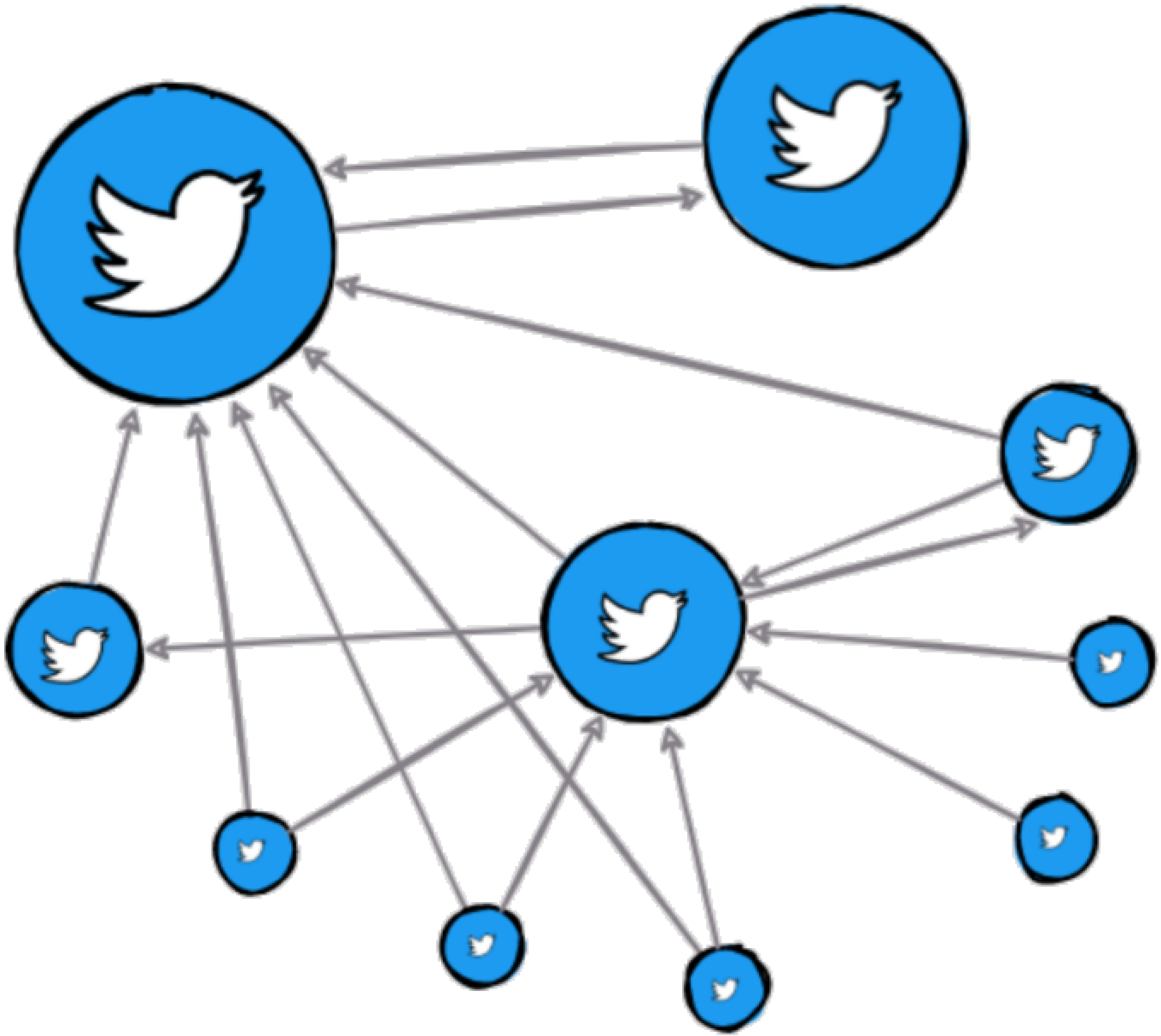
- adjacency matrix
- adjacency list

Shortest path - Dijkstra’s algorithm

Minimal spanning tree - Jarnik-Prim algorithm

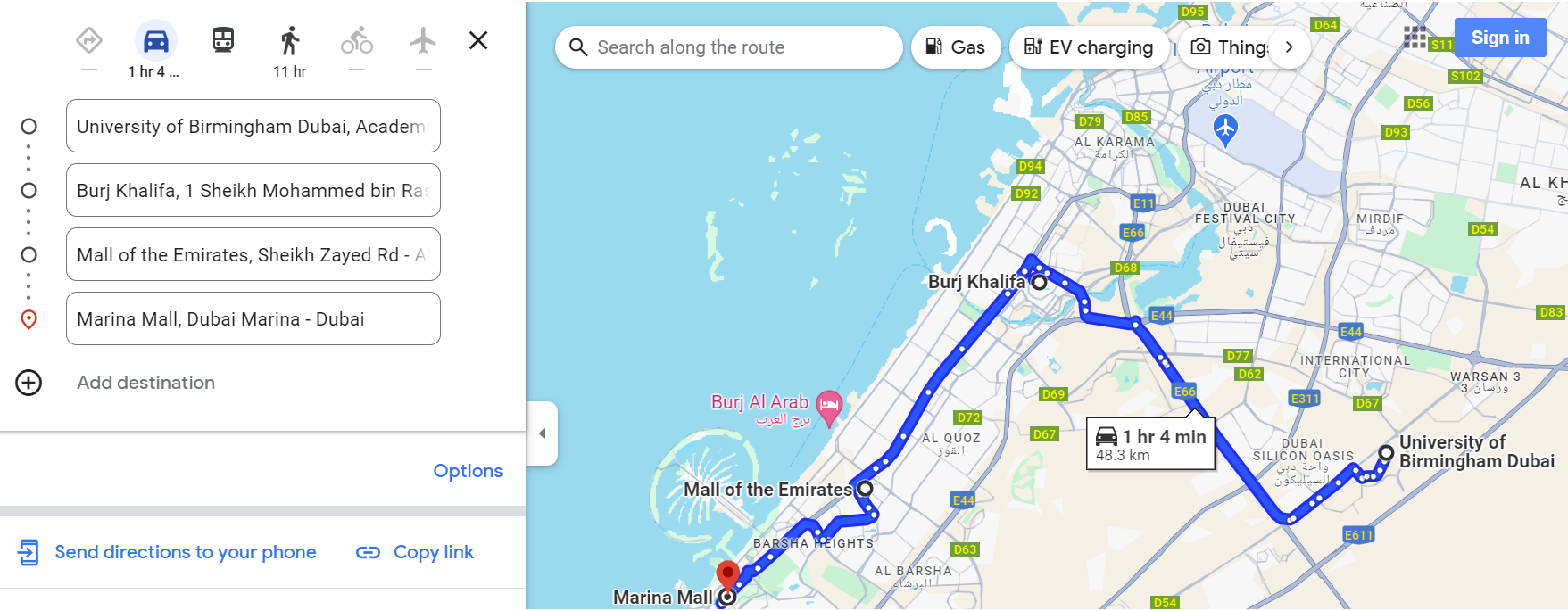


Facebook - Social Network Graph



Twitter - Social Network Graph

ROADS & TRANSPORT AUTHORITY
RTA
Green Line
Red Line
Station
Interchange Station
Airport
Abu Hail
Al Qiyadah
Abu Baker Al Siddique
Palm Deira
Stadium
AL Ras
Baniyas Square
Salah Al Din
Al Nahada
Union
Al Rigga
Dubai Airport Free Zone
Deira City Centre
Al Ghubaiba
GGICO
Al Qusais
Al Fahidi
Airport Terminal 1
Etisalat
Burjuman
Airport Terminal 2
Al Karama
Oud Metha
Al Jafiliya
Dubai Healthcare City
Emirates
World Trade Centre
Al Jadaf
Emirates Towers
Creek
Rashidiya
Financial Centre
Burj Khalifa/Dubai Mall
Business Bay
Noor Islamic Bank
First Gulf Bank
Mall of the Emirates
Sharaf DG
Dubai Internet City
Nakheel
Dubai Marina
Jumeirah Lakes Towers
Nakheel Harbour & Tower
Ibn Battuta
Energy
Danube
Jebel Ali
مترو دبي
Dubai metro
RTA

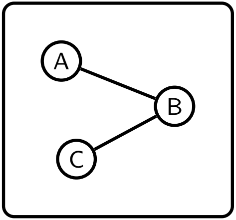

1 hr 4 ...
11 hr
University of Birmingham Dubai, Academ
Burj Khalifa, 1 Sheikh Mohammed bin Ra
Mall of the Emirates, Sheikh Zayed Rd - A
Marina Mall, Dubai Marina - Dubai
Add destination
Options
Send directions to your phone
Copy link
Search along the route
Gas
EV charging
Sign in
Burj Khalifa
Burj Al Arab
Mall of the Emirates
Marina Mall
University of Birmingham Dubai
1 hr 4 min
48.3 km
AL KARAMA
DUBAI FESTIVAL CITY
MIRDIF
INTERNATIONAL CITY
WARSAN 3
DUBAI SILICON OASIS
AL QUOZ
BARSHA HEIGHTS
AL BARSHA

Google Maps

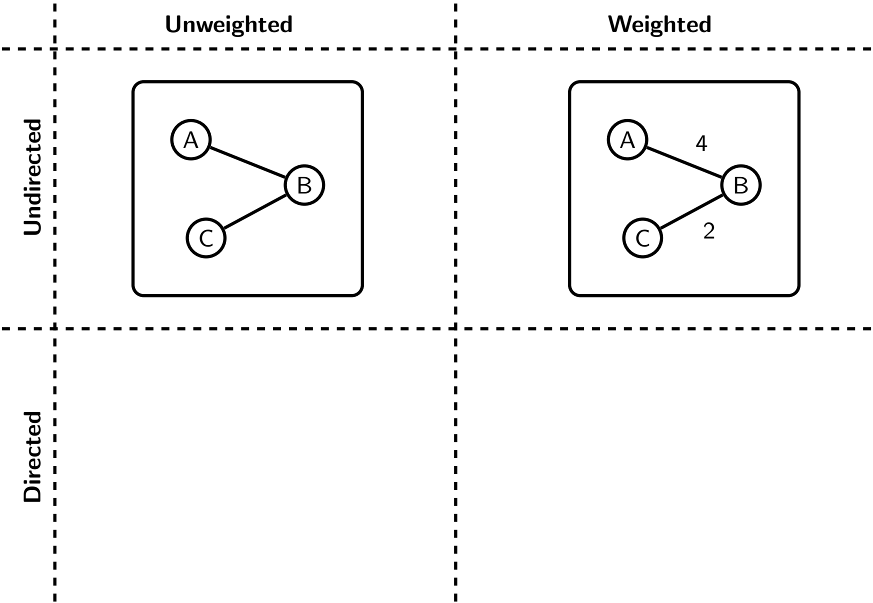
A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:

	Unweighted	**Weighted**
Undirected		
Directed		

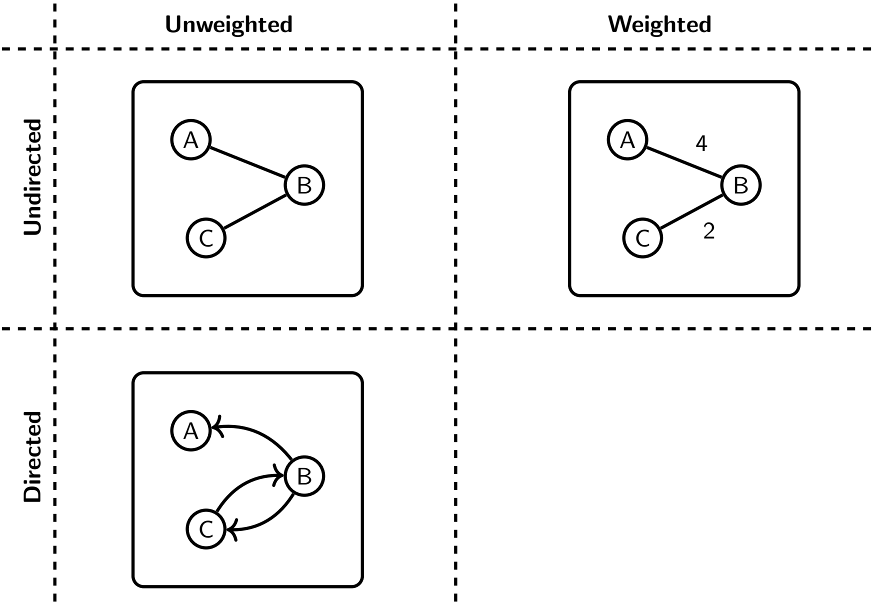
A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:

	Unweighted	**Weighted**
Undirected		
Directed		

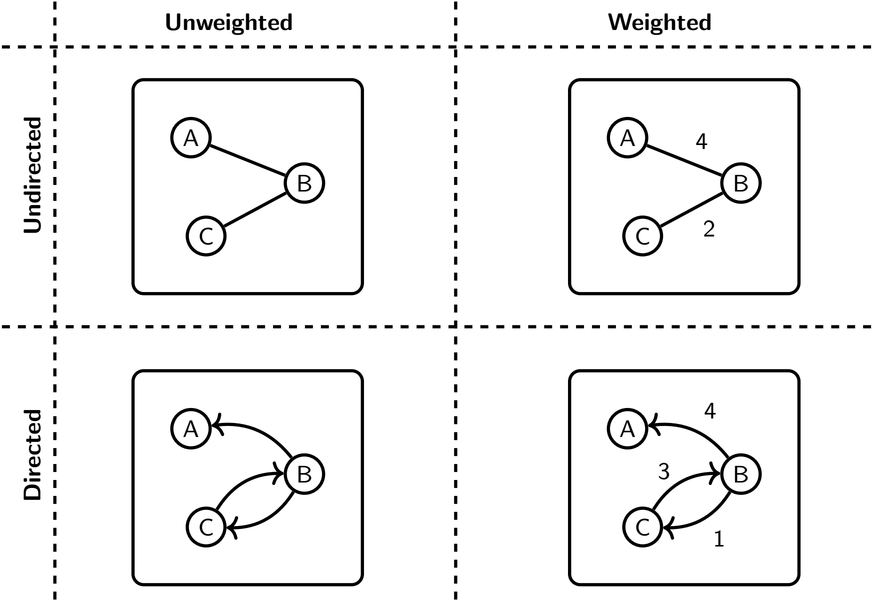
A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:



A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:



A **graph** is formed of a (finite) set of vertices/nodes and a set of edges between them. We distinguish four types of graphs:



If the graph is undirected, an edge between nodes `u` and `w` can be thought of as having two edges `u` $\rightarrow$ `w` and `w` $\rightarrow$ `u`.

Examples

Undirected unweighted graph:

- vertices = registered people on Facebook
- edges = friendships between people (it is mutual!)

Directed unweighted graph:

- vertices = registered people on Twitter
- edges = who is following who

Undirected weighted graph:

- vertices = train stops/stations
- edges = rail lines connecting train stops together with their length

This Week

Graphs

Graph representation

adjacency matrix

adjacency list

Shortest path - Dijkstra’s algorithm

Minimal spanning tree - Jarnik-Prim algorithm

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

Graph represented as an adjacency matrix

Assume that graph's vertices are numbered $V = \{0, 1, 2, ..., n-1\}$.
Adjacency matrix `G` is a two-dimensional array/matrix $n \times n$ described as follows.

Unweighted graphs:

- `G[v][w]` $= 1$ if there is an edge going from `v` to `w`
- `G[v][w]` $= 0$ if there is no such edge

Graph represented as an adjacency matrix

Assume that graph's vertices are numbered $V = \{0, 1, 2, ..., n-1\}$. *Adjacency matrix* `G` is a two-dimensional array/matrix $n \times n$ described as follows.

Unweighted graphs:

- `G[v][w]` $= 1$ if there is an edge going from `v` to `w`
- `G[v][w]` $= 0$ if there is no such edge

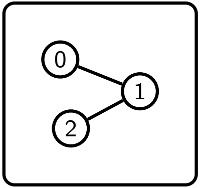
Weighted graphs:

- `G[v][w]` $=$ weight of the edge going from `v` to `w`
- `G[v][w]` $= \infty$ if there is no such edge
- `G[v][v] = 0`

Remark: The graph is undirected if `G[v][w] = G[w][v]` for all vertices `v` and `w`.

Example: Adjacency matrix

Unweighted undirected:

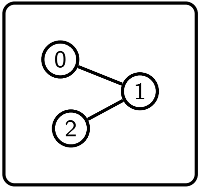


$$\texttt{G} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

For example: `G[2][0] = 0` and `G[2][1] = 1`.

Example: Adjacency matrix

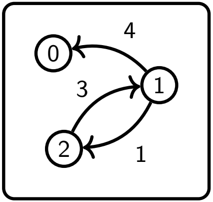
Unweighted undirected:



$$\texttt{G} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

For example: `G[2][0] = 0` and `G[2][1] = 1`.

Weighted directed:



$$\texttt{G} = \begin{pmatrix} 0 & \infty & \infty \\ 4 & 0 & 1 \\ \infty & 3 & 0 \end{pmatrix}$$

For example: `G[2][0] = infty` and `G[2][1] = 3`.

Graph represented as adjacency lists

To represent a graph on vertices $V = \{0, 1, 2, \ldots, n-1\}$ by *adjacency lists* we have an array `N` of n-many linked lists (one list for every vertex).

Unweighted:

- `N[v]` is the list of *neighbours* of `v`.

  (`w` is a neighbour of `v` if there is an edge `v` $\to$ `w`)

Graph represented as adjacency lists

To represent a graph on vertices $V = \{0, 1, 2, \ldots, n-1\}$ by *adjacency lists* we have an array `N` of n-many linked lists (one list for every vertex).

Unweighted:

- `N[v]` is the list of *neighbours* of `v`.

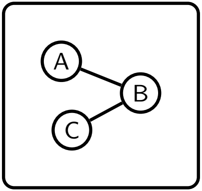
  (`w` is a neighbour of `v` if there is an edge `v` $\to$ `w`)

Weighted:

- `N[v]` is the list of *neighbours* of `v` together with the weight of the edge that connects them with `v`.

Example: adjacency lists

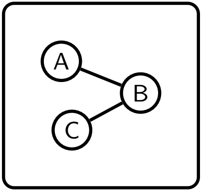
Unweighted undirected:



`N[v]`	neighbours
A	B
B	A, C
C	B

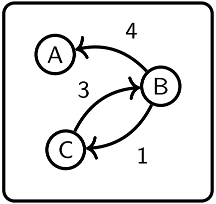
Example: adjacency lists

Unweighted undirected:



N[v]	neighbours
A	B
B	A, C
C	B

Weighted directed:

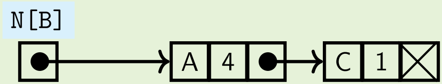


N[v]	neighbours & weights
A	
B	(A, 4), (C,1)
C	(B, 3)

We said that representing a graph by adjacency lists means that we will have an array `N` of *n*-many linked list (where *n* is the number of vertices). Then, for example, `N[2]` stores the address of the head of the linked list of all neighbours of the `2`nd vertex. If we name our vertices by letters `A`, `B`, `C`, for example, we need to find a way to assign indexes of the array `N` to the letters `A`, `B`, `C`. One way to do this is to use hash tables.

However, in the example given here, we don't care how this is done. We assume that we have lists of neighbours stored in `N[A]`, `N[B]`, `N[C]`.

In the weighted case, `N[B]` also stores the weights of the edges:



But instead of drawing this we just say that `N[B]` stores the list (A, 4), (C,1).

A graph is **sparse** if m is approximately equal to n.

An example of a sparse graph would be the graph of Facebook users with edges representing friendships. Facebook has hundreds of millions of users but each user has only a few hundreds of friends. In other words, every vertex of the graph has only a few hundreds of neighbours.

From the table we see that checking whether an edge exists is much faster for adjacency matrix. On the other hand, if our graph is sparse, then the allocated space of adjacency lists is much smaller than adjacency matrix and also traversing neighbours is faster for adjacency lists than adjacency matrix.

Comparison of those two methods

Set $n =$ the number of vertices, $m =$ the total number of edges.

	Adjacency matrix	**Adjacency lists**
Checking if there is an edge `v` $\rightarrow$ `w`:	Reading `G[v][w]` (which is in $\mathcal{O}(1)$).	Checking if `w` is in in the list `N[v]`.

Comparison of those two methods

Set n = the number of vertices, m = the total number of edges.

	Adjacency matrix	**Adjacency lists**
Checking if there is an edge `v` $\rightarrow$ `w`:	Reading `G[v][w]` (which is in $\mathcal{O}(1)$).	Checking if `w` is in in the list `N[v]`.
Allocated space:	n arrays of size n $= \mathcal{O}(n \times n)$ space.	n linked lists storing m edges in total $= \mathcal{O}(n + m)$ space.

Comparison of those two methods

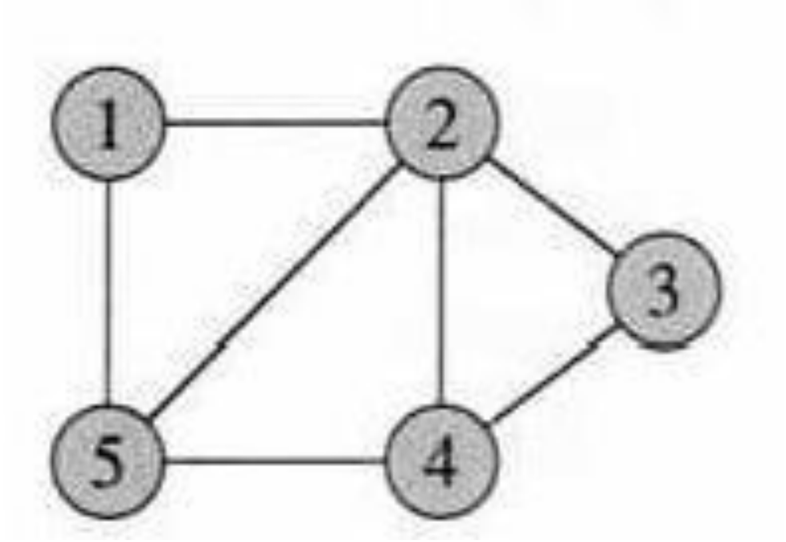
Set n = the number of vertices, m = the total number of edges.

	Adjacency matrix	**Adjacency lists**
Checking if there is an edge `v` $\rightarrow$ `w`:	Reading `G[v][w]` (which is in $\mathcal{O}(1)$).	Checking if `w` is in in the list `N[v]`.
Allocated space:	n arrays of size n $= \mathcal{O}(n \times n)$ space.	n linked lists storing m edges in total $= \mathcal{O}(n + m)$ space.
Traversing `v`'s neighbours:	Traversing all `G[v][0]`, `G[v][1]`, .., `G[v][n-1]`. $= \mathcal{O}(n)$ time.	Traversing only the linked list `N[v]`.

In the third case (with adjacency lists) we only traverse the actual neighbours of `v`. This is better whenever the graph is **sparse** (= not dense), that is, if there are relatively few edges.

Exercise

Generate the adjacency matrix and adjacency list representations for the given **Unweighted Undirected Graph**.



Exercise

Consider a graph with 1000 vertices and an average of 10 outgoing edges per vertex.

How much space do the two representations of the graphs require?

Adjacency matrix

Adjacency list

This Week

Graphs

Graph representation

- adjacency matrix
- adjacency list

Shortest path - Dijkstra’s algorithm

Minimal spanning tree - Jarnik-Prim algorithm

Paths and shortest paths

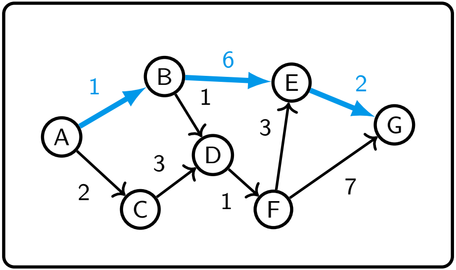
A **path** from `v` to `z` is a sequence of edges

`v` → `w1` → `w2` → ... → `z`

connecting `v` with `z`.

Example

1. $A \rightarrow B \rightarrow E \rightarrow G$



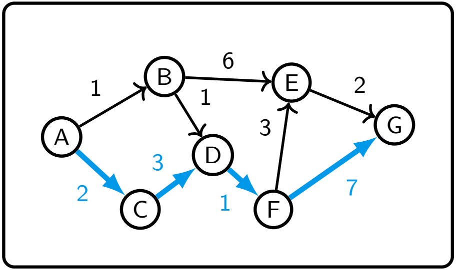
A **path** from `v` to `z` is a sequence of edges

$$\texttt{v} \to \texttt{w1} \to \texttt{w2} \to \dots \to \texttt{z}$$

connecting `v` with `z`.

Example

1. $A \to B \to E \to G$
2. $A \to C \to D \to F \to G$
3. ...



Paths and shortest paths

A **path** from `v` to `z` is a sequence of edges

`v` $\rightarrow$ `w1` $\rightarrow$ `w2` $\rightarrow$ ... $\rightarrow$ `z`

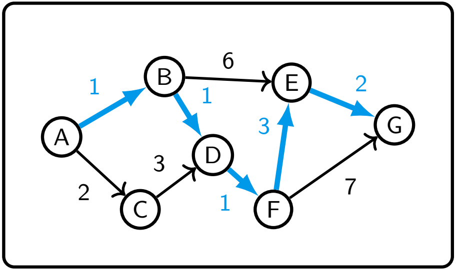
connecting `v` with `z`.

The **shortest path** is the path such that the sum of weights of its edges is the minimal such. (In unweighted graphs, set weights to 1.)

Example

1. $A \rightarrow B \rightarrow E \rightarrow G$
2. $A \rightarrow C \rightarrow D \rightarrow F \rightarrow G$
3. ...

The shortest: $A \rightarrow B \rightarrow D \rightarrow F \rightarrow E \rightarrow G$



For each vertex `w` of the graph other than `v`, we keep track of the following:

i. `d[w] =` the shortest distance from `v` to `w` so far
(Initially: ∞, except `d[v]` $= 0$)

ii. `p[w] =` the predecessor on the path from `v`
(initially: `w` itself, just a convention)

iii. `f[w] =` is computation of `d[w]` *finished*?
(initially: `false`)

The algorithm (idea):

1. Set `w` = `v`.
2. Set `f[w] = true`   (mark `w` as *finished*).
3. Update `d[u]` and `p[u]` of neighbours of `w`:

   For every neighbour `u` of `w` such that `d[w] + weight(w,u) < d[u]`,

   set `d[u] = d[w] + weight(w,u)` and `p[u] = w`.
4. Stop if `z` is finished, else set `w =` the yet unfinished vertex with the smallest `d[w]`.
5. Go to step 2.

(Where `weight(w,u)` is the weight of the edge `w` $\to$ `u`.)

The input of the algorithm is a graph (represented as an adjacency matrix or adjacency lists) and two vertices `v` and `z`. The aim is to find the shortest path from `v` to `z`.

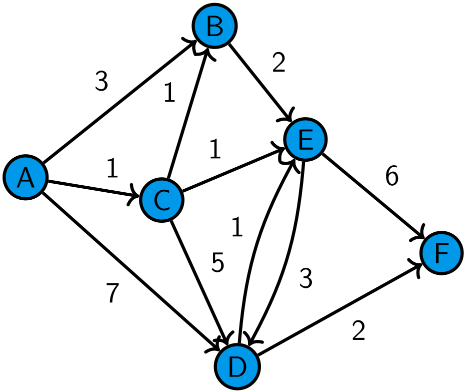
As the algorithm runs it changes the values `d[w]`, `p[w]` and `f[w]`. Initially `d[w] = infinity`, `p[w] = w` and `f[w] = false` for every vertex `w`.

The arrays `d` and `f` obeys the following *invariant*:

- `d[w]` is the length of the shortest path from `v` to `w` when using only the finished vertices (i.e. those `w` such that `f[w] == true`).
- If `w` is finished then `d[w]` is the actual length of the shortest path from `v` to `w`.

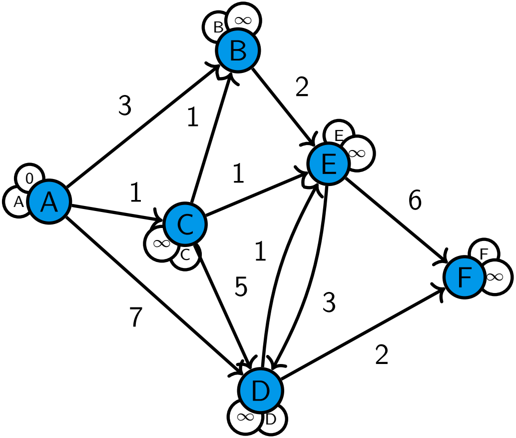
After the algorithm finishes, we compute the found shortest path by using the array `p`. Lastly, `weight(w,u)` is the weight of the edge `w` $\rightarrow$ `u` obtained from the adjacency matrix/lists of the graph.

Example: Execution of Dijkstra's algorithm



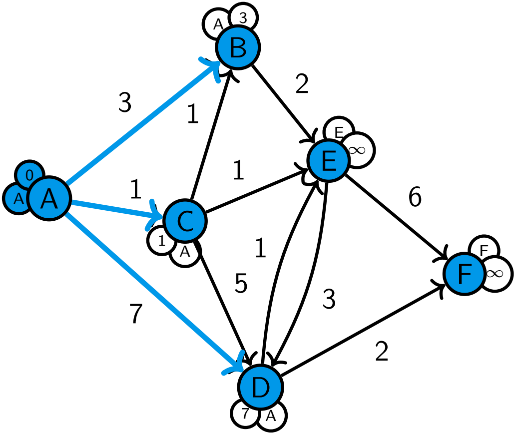
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	



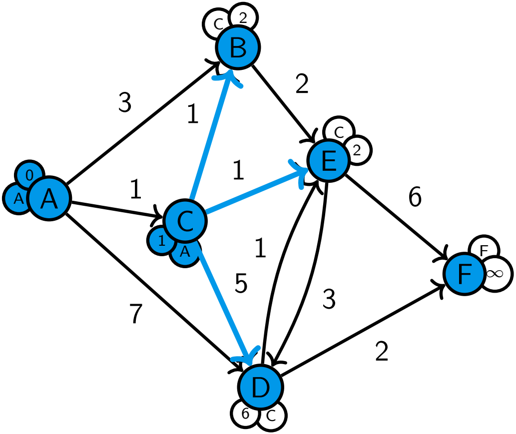
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A



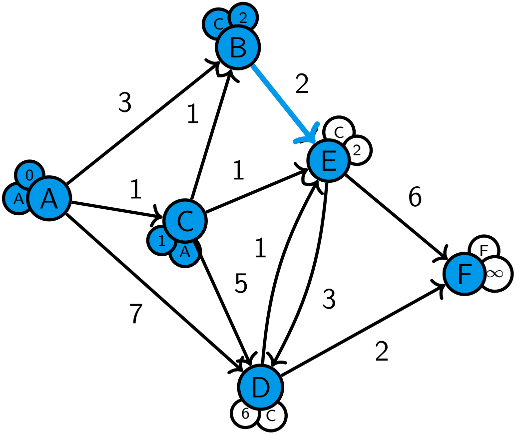
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C



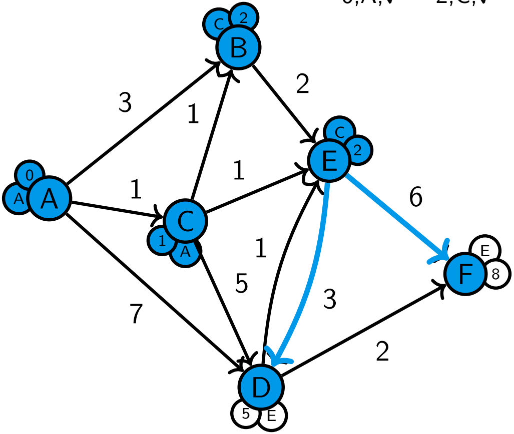
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞,F	B



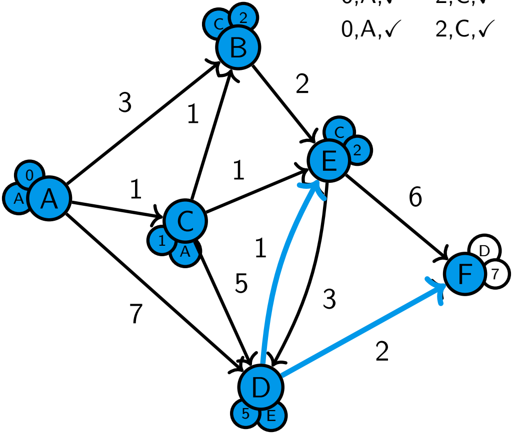
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞,F	B
0,A,✓	2,C,✓	1,A,✓	5,E	2,C,✓	8,E	E



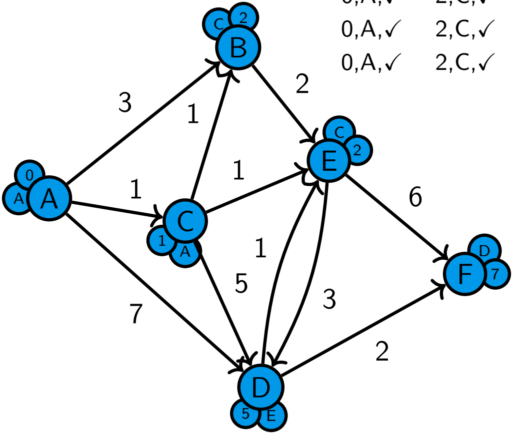
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞,F	B
0,A,✓	2,C,✓	1,A,✓	5,E	2,C,✓	8,E	E
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D	D



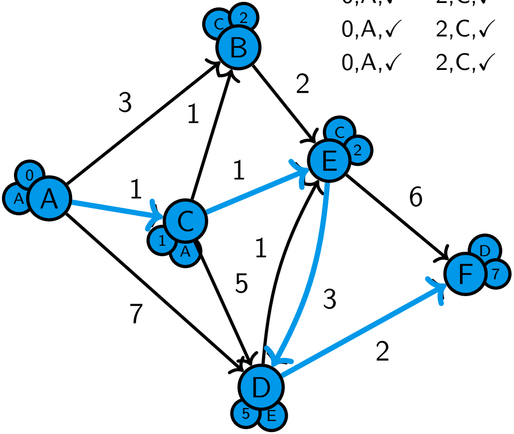
Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞,F	B
0,A,✓	2,C,✓	1,A,✓	5,E	2,C,✓	8,E	E
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D	D
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D,✓	F



Example: Execution of Dijkstra's algorithm

A	B	C	D	E	F	finished
0,A	∞,B	∞, C	∞,D	∞,E	∞,F	
0,A,✓	3,A	1,A	7,A	∞,E	∞,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞,F	B
0,A,✓	2,C,✓	1,A,✓	5,E	2,C,✓	8,E	E
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D	D
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D,✓	F



The shortest path from A to F is obtained (in the reversed order) by reading out `p[w]`'s, starting from F:

$A \to C \to E \to D \to F.$

Every iteration of the algorithm corresponds to one row in the table and each such row shows the content of the three arrays `d[-]`, `p[-]` and `f[-]`. (Check marks denote finished vertices.)

In the graph, the two circles adjacent to a vertex mark the current state of `d[w]` and `p[w]`. They turn blue whenever the vertex is marked as finished.

Dijkstra's time complexity (adjacency matrix)

$n =$ the number of vertices, $\quad m =$ the total number of edges.

We do the following *up to* n-times:

2. Find `w` which is unfinished and with the smallest `d[w]`.
3. Mark `w` as finished.
4. Update every neighbour of `w`.

Representing the graph by an *adjacency matrix*, means that it takes $\mathcal{O}(n)$ to do step 4.

We can also do step 2. in $\mathcal{O}(n)$ by going through all vertices.

$\implies$ The time complexity is $\mathcal{O}(n^2)$.

Dijkstra's time complexity (adjacency lists)

We do the following *up to n*-times:

2. Find `w` which is unfinished and with the smallest `d[w]`.
3. Mark `w` as finished.
4. Update every neighbour of `w`.

With *adjacency lists*, executions of step 4. will (in total) update

neighbours of the 1st selected `w`,
neighbours of the 2nd selected `w`,
neighbours of the 3nd selected `w`,
...

Over all iterations combined we update *m*-many times! $\Rightarrow \mathcal{O}(m)$

Dijkstra's time complexity (adjacency lists)

We do the following *up to n*-times:

2. Find `w` which is unfinished and with the smallest `d[w]`.
3. Mark `w` as finished.
4. Update every neighbour of `w`.

With *adjacency lists*, executions of step 4. will (in total) update

neighbours of the 1st selected `w`,
neighbours of the 2nd selected `w`,
neighbours of the 3nd selected `w`,
...

Over all iterations combined we update m-many times! $\Rightarrow \mathcal{O}(m)$

Speeding up steps 2. and 4.:

Use min-priority queue! The priority of `u` is `d[u]`.

- We call `deleteMin` once per iteration, i.e. up to n-times.
- Whenever `d[u]` changes, we `update` the priority of `u`.

$\Longrightarrow$ The total time complexity

$$\mathcal{O}(n \times \text{"cost of deleteMin"} \ + \ m \times \text{"cost of update"})$$

What omitted in the analysis is the time complexity of initialising the heap. This is usually done by `heapify` and its time complexity was always $\mathcal{O}(n)$ for all heaps we had. Alternatively, we can do `insert` n-times which will result in the time complexity $\mathcal{O}(n \log n)$ or $\mathcal{O}(n)$ depending on the heap that we are using. Either way, the initialisation will not play any role in the total time complexity.

Dijkstra's time complexity – comparison

Adjacency matrices	Adjacency lists	
	Binary Heaps	Fibonacci Heaps
$\mathcal{O}(n^2)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}((n\log n)+m)$

Min-priority queues:

- Binary heaps: both `update` and `deleteMin` are in $\mathcal{O}(\log n)$.
- Fibonacci heaps: `update` is in $\mathcal{O}(1)$ and `deleteMin` is in $\mathcal{O}(\log n)$ (both amortized).

Remark: Dijkstra's algorithm works only if all weights are ≥ 0.

Remark: If the graph is *dense*, that is if the number of edges is approximately n^2, then using adjacency lists together with binary heaps has the time complexity $\mathcal{O}((n + n^2) \log n) = \mathcal{O}(n^2 \log n)$ which is slower than just using adjacency matrices. This problem disappears when using Fibonacci heaps where, for dense graphs, we the time complexity becomes $\mathcal{O}(n \log n + n^2) = \mathcal{O}(n^2)$.

On the other hand, if the graph is not dense, using adjacency lists with binary heaps or Fibonacci heaps is faster than using adjacency matrices.

Dijkstra's algorithm (pseudocode with adjacency matrix)

```
dijkstra_with_matrix(int[][] G, int v, int z) {
    n = G.length;
    d = new int[n]; p = new int[n]; f = new bool[n];

    for (int w=0; w<n; w++) {
        d[w] = infty;    p[w] = w;    f[w] = false;
    }
    d[v] = 0;

    while (true) {
        w = min_unfinished(d, f);
        if (w == -1) break;

        for (int u=0; u<n; u++) update(w, u, d, p);
        f[w] = true;
    }
    // Compute output in a required form:
    return compute_result(v, z, G, d, p);
}
```

```
int min_unfinished(int[] d, bool[] f) {
    int min = +infty;
    int idx = -1;

    for (int i=0; i<d.length; i++) {
        if (f[i] == false && d[i] < min) {
            idx = i;
            min = d[i]
        }
    }

    return idx;
}
```

```
void update(w, u, G, d, p) {
    if (d[w] + G[w][u] < d[u]) {
        d[u] = d[w] + G[w][u];
        p[u] = w;
    }
}
```

Dijkstra's algorithm (pseudocode with adjacency lists)

```
dijkstra_with_lists(List<Edge>[] N, int v, int z) {
    n = G.length;
    d = new int[n];    p = new int[n];
    Q = new MinPriorityQueue();

    for (int w=0; w<n; w++) {
        d[w] = infty;    p[w] = w;
        Q.add(w, d[w]);
    }
    d[v] = 0;
    Q.update(v, 0);

    while (Q.notEmpty()) {
        w = Q.deleteMin()

        for (Edge e : N[w]) { // iterate over edges to neighbours
            u = e.target;
            if (d[w] + e.weight < d[u]) { // should we update?
                d[u] = d[w] + e.weight;
                p[u] = w;
                Q.update(u, d[u]);
            }
        }
    }

    return compute_result(v, z, G, d, p);
}
```

```
class Edge {
  // target node
  int target;

  int weight;
}
```

The initialisation happens on lines 6–9.

Lines 10–11 make sure that the first selected `w` will be `v`.

We use the class `Edge` to store neighbours together with the weight of the edge that connects them. For example, if the vertex `A` has neighbours B, C and D with the edge $A \to B$ of weight 3, $A \to C$ of weight 1, and $A \to D$ of weight 8, then we will have that the linked list `N[v]` stores `Edge(B, 3)`, `Edge(C, 1)` and `Edge(D, 8)`.

This Week

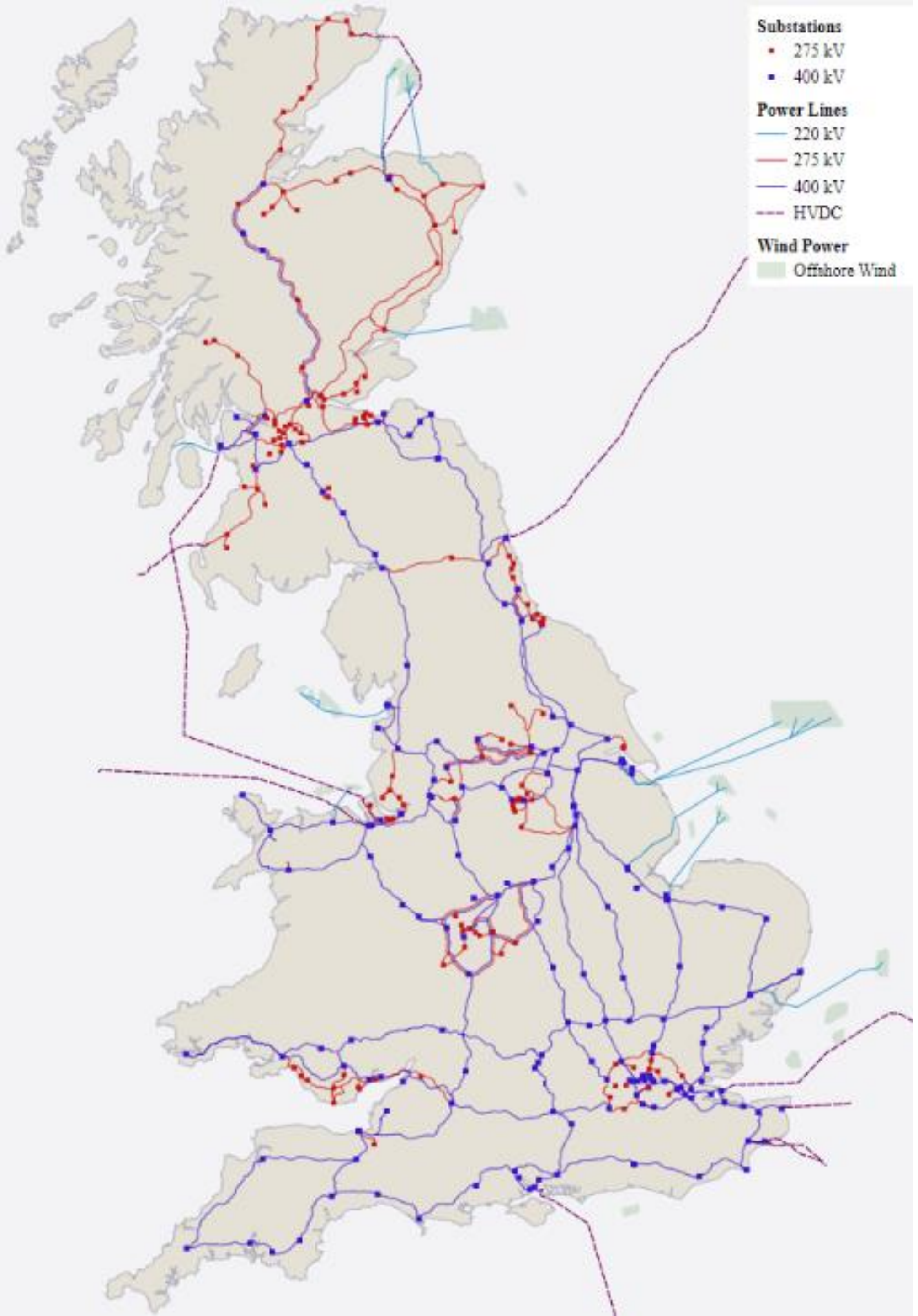
Graphs

Graph representation

- adjacency matrix
- adjacency list

Shortest path - Dijkstra's algorithm

Minimal spanning tree - Jarnik-Prim algorithm



National Grid (Great Britain)

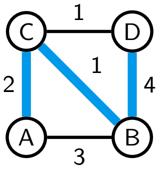
Minimal spanning tree

Such that there is a path between any two vertices.

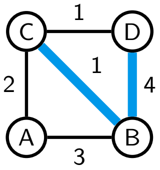
Assumption: Consider only *undirected* and *connected* graphs!

A **spanning tree** is a minimal possible selection of edges which connects all vertices. (That is, a spanning tree does not contain any cycles.)

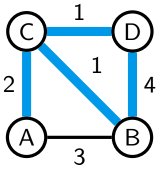
Example



Spanning tree.



Not a spanning tree.
A is not connected.



Not a spanning tree.
Any of the edges CV, CD, BD could be removed.

Minimal spanning tree

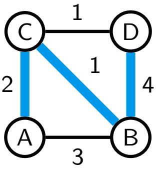
Such that there is a path between any two vertices.

<u>Assumption: Consider only *undirected* and *connected* graphs!</u>

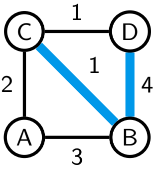
A **spanning tree** is a minimal possible selection of edges which connects all vertices. (That is, a spanning tree does not contain any cycles.)

Minimum spanning tree is a spanning tree such that the sum of weights of its edges is the minimal such.

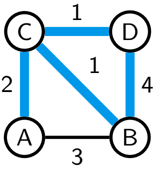
Example



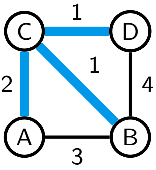
Spanning tree.



Not a spanning tree. A is not connected.



Not a spanning tree. Any of the edges CV, CD, BD could be removed.



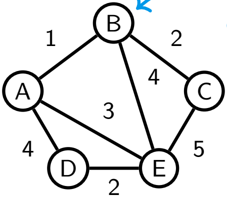
Minimum spanning tree!

Example: Execution of Jarník-Prim algorithm

Idea: Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

Example

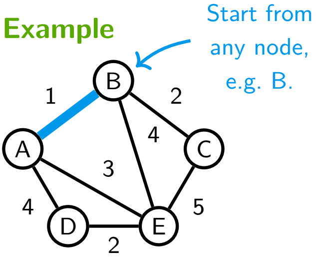
Start from any node, e.g. B.



Example: Execution of Jarník-Prim algorithm

Idea: Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

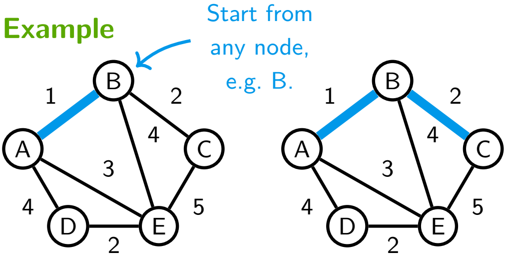
Example



Example: Execution of Jarník-Prim algorithm

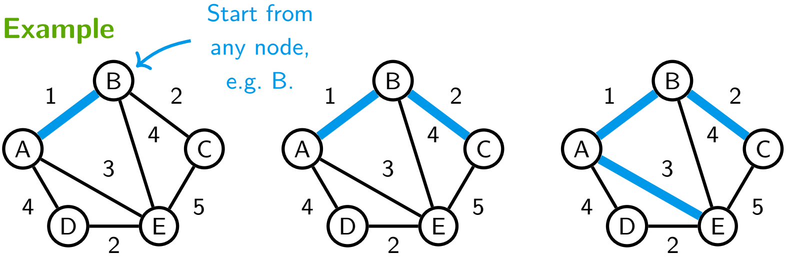
Idea: Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

Example



Idea: Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

Example

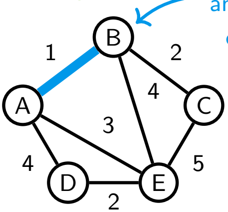


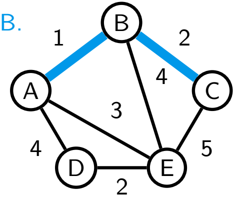
Example: Execution of Jarník-Prim algorithm

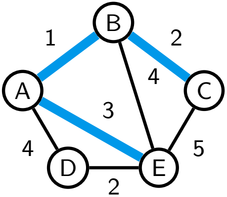
Idea: Iteratively extend the tree with an edge which has the smallest weight and which connects a yet unconnected node.

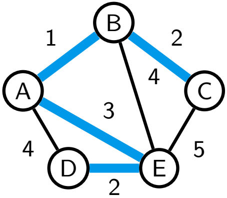
Example

Start from any node, e.g. B.









The minimal spanning tree consists of edges: AB, BC, AE, ED.

Jarník-Prim algorithm for finding the minimal spanning tree

For each vertex `w` of the graph, we keep track of the following:

i. `d[w] =` the current distance from the *tree* (initially: ∞)
ii. `p[w] =` the vertex which connects to the tree (initially: `w`)
iii. `f[w] =` has `w` been added to the tree? (initially: `false`)

The algorithm (idea):

1. Set `d[0] = 0`. (vertex `0` could be replaced by any other vertex)
2. Set `w =` the yet unfinished node with the smallest `d[w]`.
3. Set `f[w] = true` (mark `w` as *finished*).
4. Update `d[u]` and `p[u]` of neighbours of `w`:

   For every neighbour `u` of `w` such that `weight(w,u) < d[u]`, set `d[u] = weight(w,u)` and `p[u] = w`.
5. If there are still some nodes unfinished, go to step 2.

(Where `weight(w,u)` is the weight of the edge `w` $\to$ `u`.)

Jarník-Prim's algorithm works similarly to Dijkstra's algorithm. The differences are marked by red. Altough the principle is similar, the interpretation of the execution and the result is different. The main idea of Jarník-Prim is that we are building a "spanning tree" iteratively, in steps.

The following is an invariant for Jarník-Prim:

1. The vertices marked as finished are connected/added to the tree.
2. For those vertices which are not connected yet, `d[w]` denotes the smallest weight of an edge that connects `w` to the tree.
3. `p[w]` denotes the vertex of the tree such that the edge between `w` and `p[w]` is the edge with weight `d[w]`.

In step 1. we pick an arbitrary vertex (stored on 0th position) and mark its distance as 0. As a consequence, we start building tree from this vertex.

After the algorithm finishes, i.e. all vertices are marked finished, we can read out the spanning tree from the array `p[-]`. To obtain the minimum spanning tree, for every vertex `w` (except for `w == 0`), add the edge `w` — `p[w]`.

Jarník-Prim's time complexity

The time complexity is the same as for Dijkstra's algorithm!

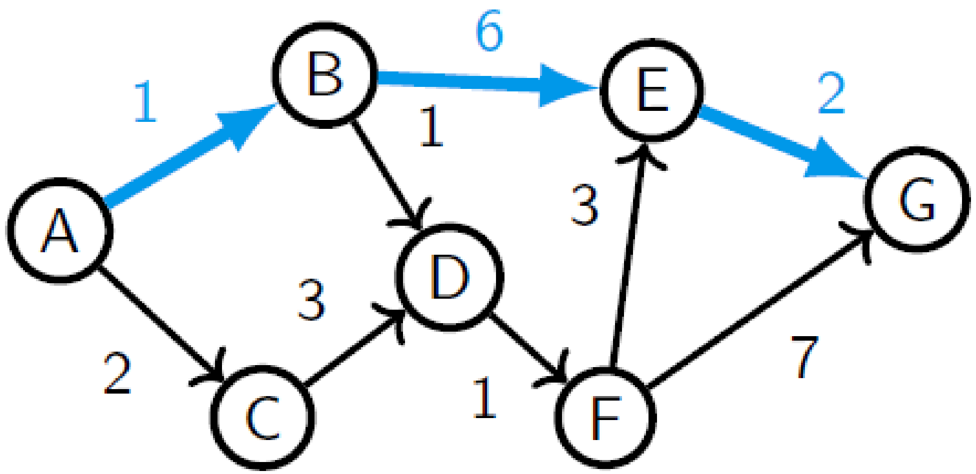
Adjacency matrices	Adjacency lists	
	Binary Heaps	Fibonacci Heaps
$\mathcal{O}(n^2)$	$\mathcal{O}((n+m)\log n)$	$\mathcal{O}((n\log n)+m)$

Remark: Unlike Dijkstra's algorithm, Jarník-Prim's algorithm would also work for graphs with edges that have negative weights.

Exercise

In the graph below,

1. Which vertices do not have paths from B?
2. What is the shortest path from A to F in the above graph?

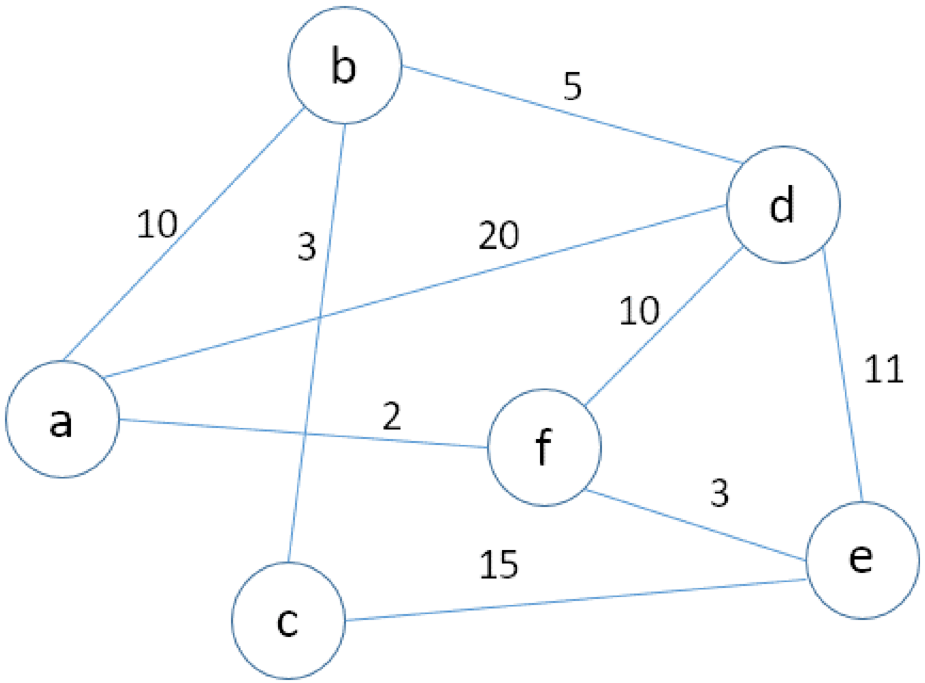


Exercise

Consider the undirected graph below. If we are trying to calculate the **shortest path**, what will be the first node selected

From the node a to every other node

From the node b to every other node



Weekly Reading

Data Structures and Algorithm Analysis by Clifford A. Shaffer (3rd Ed)

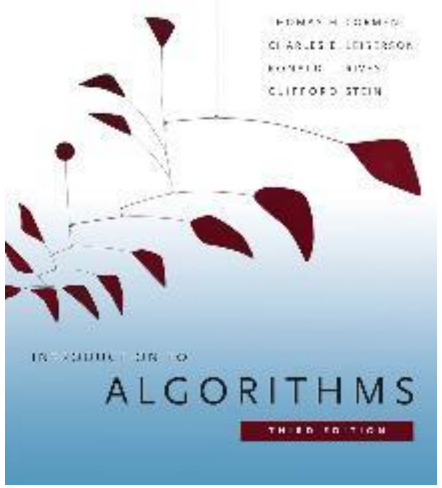
Chapter 11 Graphs

https://people.cs.vt.edu/~shaffer/Book/JAVA3elatest.pdf

Introduction to Algorithms by Cormen (4th Ed)

Chapter 20, 21, 22

https://ebookcentral.proquest.com/lib/bham/detail.action?docID=6925615



Next Week

Week	Date	Topic
1	15 Jan	Searching algorithms
2	22 Jan	Binary Search Tree
3	29 Jan	Balancing Trees – AVL Tree
4	5 Feb	Databases – Conceptual Design
5	14 Feb	Databases – Logical Design & Relational Algebra
6	19 Feb	Consolidation Week
7	26 Feb	Complexity analysis, Stacks, Queues, Heaps
8	4 Mar	Sorting Algorithms, Hash tables
9	11 Mar	Graph Algorithms
10	18 Mar	Databases – Normalization
		Easter break and Eid break
11	22 Apr	Databases – Concurrency
12	29 Apr	Assessment Support Week

The Dubai Student Survey

Provide valuable feedback on your academic and overall experience for your chance to win a UoBD prize bundle including a hoodie.

Scan the QR code to complete. The survey closes Sunday 5 May 2024.

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

Improvements from DSS 2022-2023

- #1 (PT) Students experienced late enrollments of up to 4 weeks.
  - Developed improved coordination process with Prog Admin Team → 2023-24: Timely enrollments of on-time arrivals.
  - Further improvement coming in 2024-25!
- #2 Improve resolution of inter-module dependencies.
  - Streamlined PT study plans coming (w.e.f. 2024-25).
- #3 Demand for more and timely feedback
  - More formative feedback opportunities (Kahoot! / MS Forms quizzes).
  - Fewer delays in grading.
- #4 Better support in labs.
  - Introduced Lab TAs.
- #5 Brought in more industry guest talks through UoBD ACM Student chapter
  - <u>Industry Talks:</u> intel US, intel Dubai, NVIDIA, VISA Research, etc. → More in pipeline for next year.
  - <u>Workshops:</u> Blockchain, git, Python, Web development, Game development.
  - <u>Hackathons & Competitions:</u> Hackathon, Gulf Prog Competition.
- #6 Improved communication.
  - Majorly updated induction.
  - Revamped Student Handbooks.

Takeaway

Your considered feedback helps us improve processes and deploy new resources effectively.

The DSS is an important feedback channel.

Tell us what we are doing well ☑ – Tell us where we need to improve ☒

Thank you.

Questions?

UNIVERSITY OF BIRMINGHAM | DUBAI دبي

Attendance

UNIVERSITY OF BIRMINGHAM | DUBAI دبي