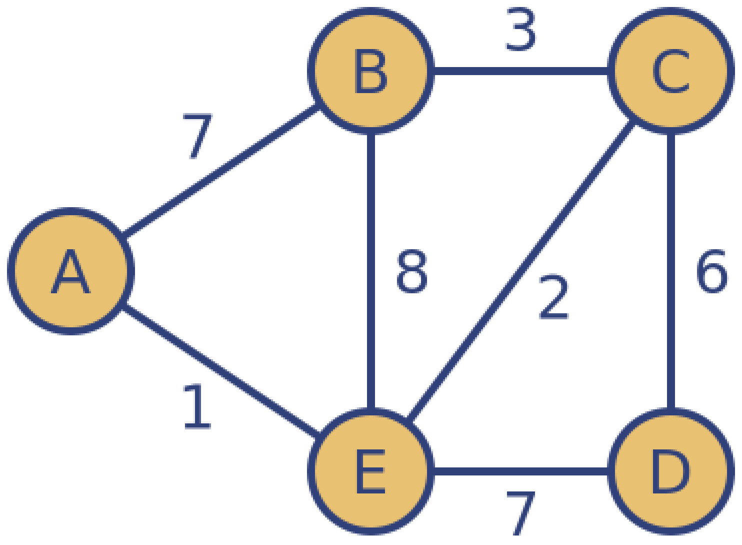


Here’s an undirected graph. For finding the shortest path BFS(Breadth Fast search),DFS(Depth Fast Search) and Dijkstra’s algorithm will be very useful but each has its own specifications. In order to find the shortest path from A to D:

BFS(Breadth Fast Search):

This algorithm will explore all the neighbours level by level. It guarantees the **shortest path in terms of edges** not weights. In order to find the path we need to set a queue where we will store the weights level by level.

Step	Queue (Front → Back)	Visited	Current Node	Action
1	[A]	{}	A	Start
2	[B, E]	{A}	A	Enqueue neighbors
3	[E]	{A, B}	B	Visit B, enqueue (C, E)

4	[E, C, E]	{A, B}	Added neighbors	
5	[C, E]	{A, B, E}	E	Visit E, enqueue (C, D)
6	[C, E, C, D]	{A, B, E}	Added neighbors	
7	[E, C, D]	{A, B, E, C}	C	Visit C
8	[C, D]	{A, B, E, C, D}	D	Found

Here we start from A and try to find its neighbours and put them into the queue. Whether it's the shortest or not we try to put every neighbour elements in the queue. Then for each queued elements we try to find again its neighbour elements and push them into the queue. Eventually we will find that there will be some elements that are still yet to be pointed out. We will then write it down as the path like the others.

SO after iterating through each element the shortest path we find is:

A -> B -> E -> C -> D

DFS(Depth Fast Search):

DFS does not guarantee the shortest path. It explores deeply across the graph. In order to find the shortest path from A -> D :

Step	Stack (Top → Bottom)	Visited	Current Node	Action
1	[A]	{}	A	Start
2	[E, B]	{A}	A	Push neighbors (E, B)
3	[E]	{A, B}	B	Visit B, push (C, E)
4	[E, C, E]	{A, B}	B	Added neighbors
5	[E, C]	{A, B, E}	E	Visit E, push (C, D)
6	[E, C, D, C]	{A, B, E}	E	Added neighbors

7	[E, C, D]	{A, B, E, C}	C	Visit C
8	[E, C, D]	{A, B, E, C, D}	D	Found target

In this case we'll use a stack where we will push the elements or destinations of the graph. For example:

We stand on A and find its one neighbour. This time we won't push all the neighbours. We'll choose only one and push it into the stack. After exploring its neighbour we'll suspend the previous element and try to find the next element's only one neighbour. Eventually we'll see that we can't find the next element. At that time we'll try backtracking. And while backtracking we'll cut the previous elements that have been explored previously. And thus in this way we'll try to explore each element until while doing backtracking we see that there is no element that is to be explored and thus we'll find the shortest path.

The shortest path we find : **A–E–C–D .**This path has total weight 1 + 2 + 6 = 9.

But DFS does not guarantee shortest path.

Dijkstra Algorithm:

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a starting node to all other nodes in a graph with non-negative edge weights. It works on only weighted graphs with non-negative weights. It finds the shortest path from source to all nodes

The data structure we can use here is: min-heap priority queue. But we can also use set data structure to make it more efficient in some cases.

Time Complexity: O((V+E) log V) with binary heap

In this algorithm we try to set the weights as infinite telling that they are not explored yet which will be the initial setup like this:

Node	Distance from A	Previous
A	0	—
B	∞	—
C	∞	—
D	∞	—
E	∞	—

Now we will go each element and try to find its shortest neighbours and avoid other longer weighted neighbours.
In this case Heap initially:
→ **[(0, A)]**

Here's a table to demonstrate how the interaction is working and which parts are set to relax before going to the next element :

Step	Visited Node	Heap Before	Heap After	Distance Updates
1	A	[(0, A)]	[(1, E), (7, B)]	B=7, E=1
2	E	[(1, E), (7, B)]	[(3, C), (7, D), (7, B)]	From E: C=3 (1+2), D=8 (1+7), B stays 7
3	C	[(3, C), (7, D), (7, B)]	[(7, B), (7, D)]	From C: D=9 (3+6) but D=8 already better
4	B	[(7, B), (7, D)]	[(7, D)]	From B: E=1 (better exists), C=3 (better exists)
5	D	[(7, D)]	[]	End

So we try to find the neighbours and after finding it we try to find the shortest weights of those neighbours and try to add the previous weights in order to find the shortest path. And eventually we can reach our destination by iterating through each node and adding its previous weights.

Final shortest path we'll get:

Node	**Shortest Distance**	**Path**
A	0	A
B	7	A–B
C	3	A–E–C
D	8	A–E–D
E	1	A–E

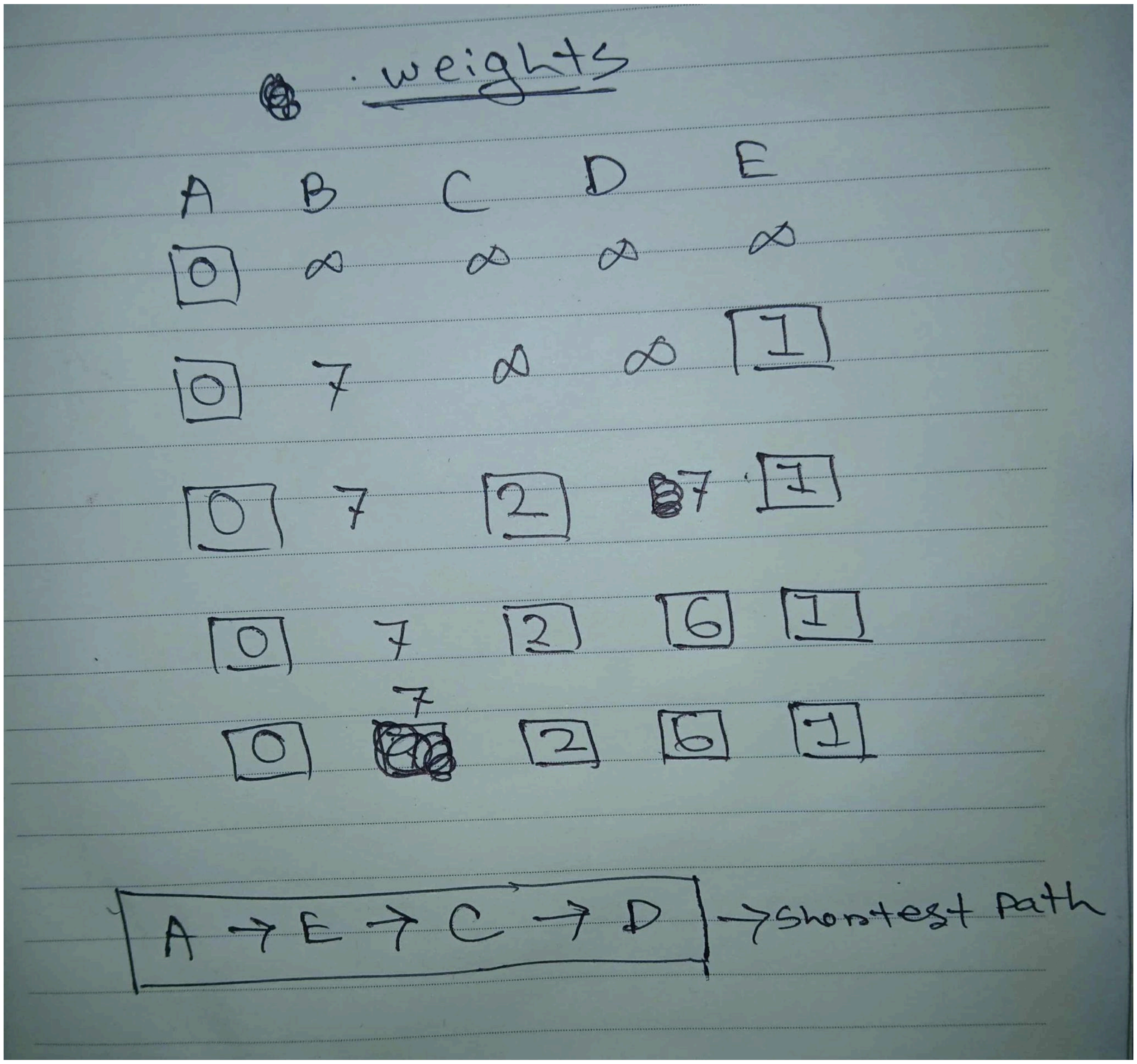
So the shortest path we obtain is :

A -> E -> C -> D

Here's another way we can implement the Dijkstra algorithm to find the shortest path:
The box parts are indicating the relaxation state of the nodes. In this table we try to find the shortest nodes and try to box those parts so that it stays still and we try to find the other neighbours.

SO the Shortest path we can obtain is:

A -> E -> C -> D



· weights

A	B	C	D	E
$\boxed{0}$	∞	∞	∞	∞
$\boxed{0}$	7	∞	∞	$\boxed{1}$
$\boxed{0}$	7	$\boxed{2}$	7	$\boxed{1}$
$\boxed{0}$	7	$\boxed{2}$	$\boxed{6}$	$\boxed{1}$
$\boxed{0}$	7	$\boxed{2}$	$\boxed{6}$	$\boxed{1}$

$\boxed{A \to E \to C \to D}$ → Shortest path