

Shortest Paths and Dijkstra's Algorithm

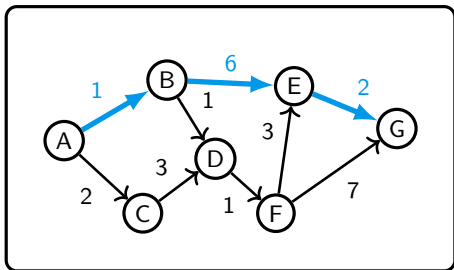
Paths and shortest paths

Recall: A **path** is a sequence of vertices v_1, v_2, \dots, v_n such that v_i and v_{i+1} are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from A to B is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from A to B . Note that there may be multiple different shortest paths from A to B . (In unweighted graphs, set weights to 1.)

Example

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



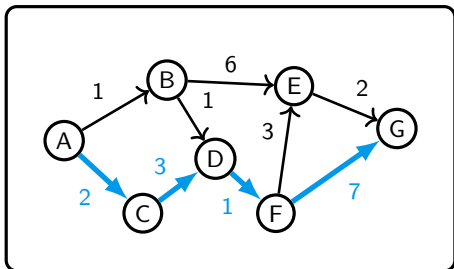
Paths and shortest paths

Recall: A **path** is a sequence of vertices v_1, v_2, \dots, v_n such that v_i and v_{i+1} are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from A to B is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from A to B . Note that there may be multiple different shortest paths from A to B . (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. ...



Paths and shortest paths

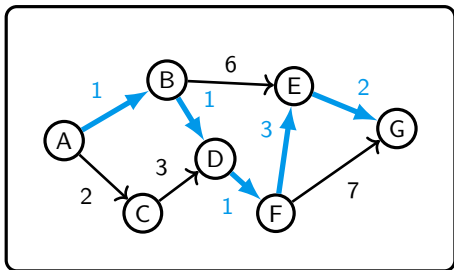
Recall: A **path** is a sequence of vertices v_1, v_2, \dots, v_n such that v_i and v_{i+1} are connected by an edge for all $1 \leq i \leq n-1$.

A **shortest path** from A to B is a path for which the sum of the weights along the path is less than or equal to the sum of the weights along any other path from A to B . Note that there may be multiple different shortest paths from A to B . (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: $A \rightarrow B \rightarrow$
 $D \rightarrow F \rightarrow E \rightarrow G$



Dijkstra's algorithm to find the shortest path from v to z

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

The algorithm (idea):

- 1: set $d[v] = 0$ (i.e. start on v)
- 2: while there are unfinished vertices:
- 3: set $w =$ the yet unfinished vertex with the smallest $d[w]$
- 4: set $f[w] = \text{true}$ (i.e. mark w as *finished*)
- 5: for every neighbour u of w :
- 6: if $d[w] + \text{weight}(w,u) < d[u]$:
- 7: set $d[u] = d[w] + \text{weight}(w,u)$ and $p[u] = w$

(Where $\text{weight}(w,u)$ is the weight of the edge $w \rightarrow u$)

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 $d[w] = \text{infinity}$, $p[w] = w$ and $f[w] = \text{false}$ for every vertex w .

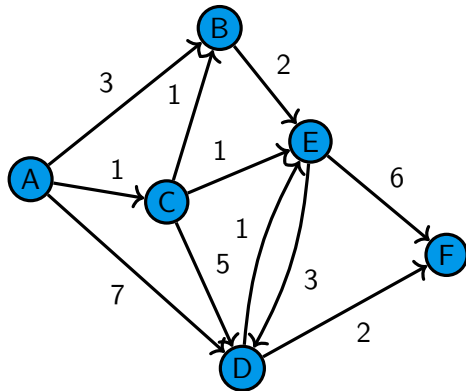
The arrays d and f obeys the following *invariant*:

- $d[w]$ is the length of the shortest path from v to w when using only the finished vertices (i.e. those w such that $f[w] == \text{true}$).
- 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, $\text{weight}(w,u)$ is the weight of the edge $w \rightarrow u$ obtained from the adjacency matrix/lists of the graph.

Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

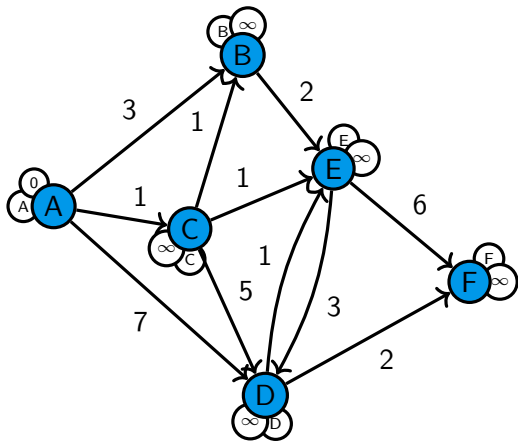


Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

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

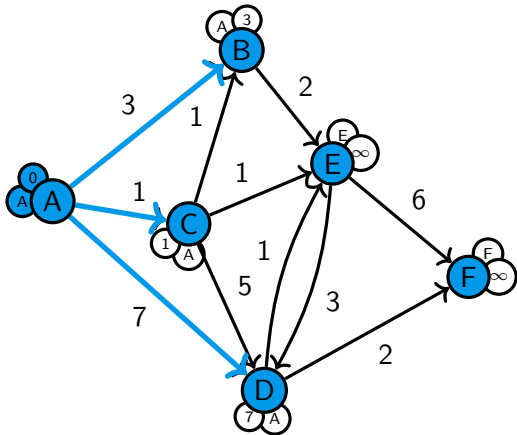
finished



Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

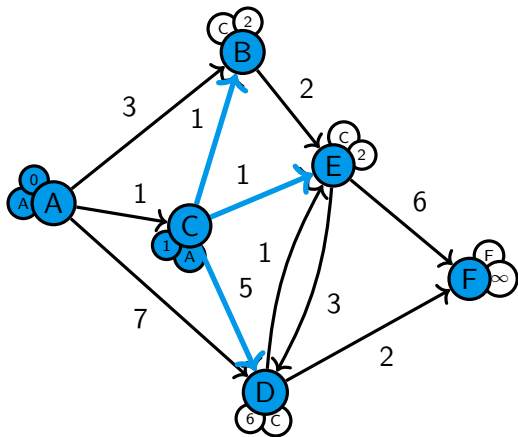
A	B	C	D	E	F	finished
0,A	∞ ,B	∞ , C	∞ ,D	∞ ,E	∞ ,F	
0,A,✓	3,A	1,A	7,A	∞ ,E	∞ ,F	A



Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

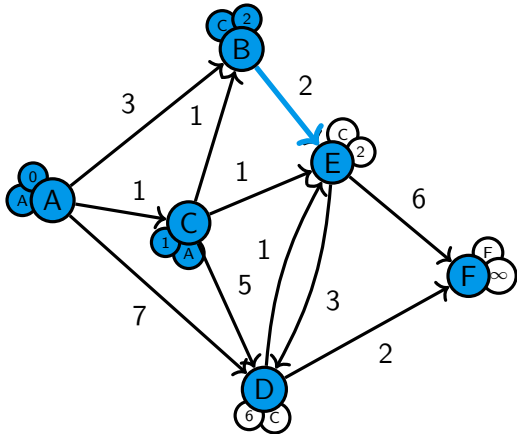
A	B	C	D	E	F	finished
0,A	∞ ,B	∞ , C	∞ ,D	∞ ,E	∞ ,F	
0,A,✓	3,A	1,A	7,A	∞ ,E	∞ ,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞ ,F	C



Example: Execution of Dijkstra's algorithm

Shortest Path $A \rightarrow F$

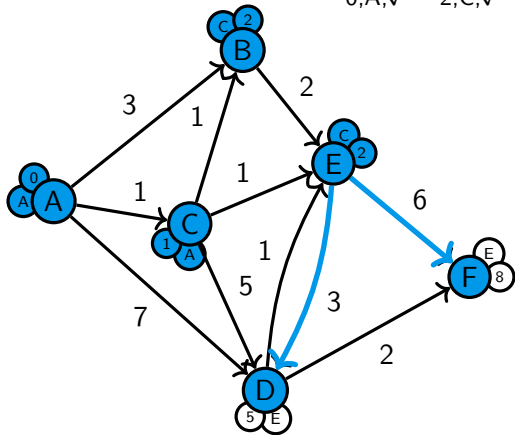
A	B	C	D	E	F	finished
0,A	∞ ,B	∞ ,C	∞ ,D	∞ ,E	∞ ,F	
0,A,✓	3,A	1,A	7,A	∞ ,E	∞ ,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞ ,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞ ,F	B



Example: Execution of Dijkstra's algorithm

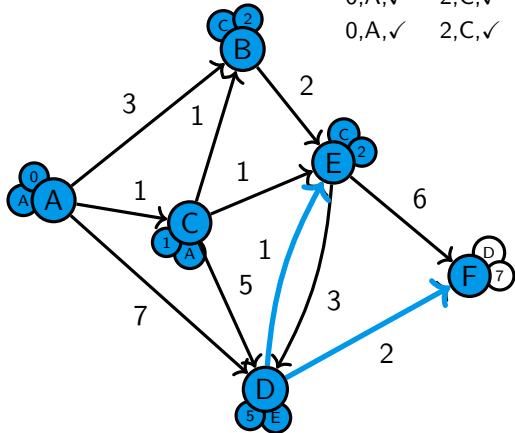
Shortest Path $A \rightarrow F$

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

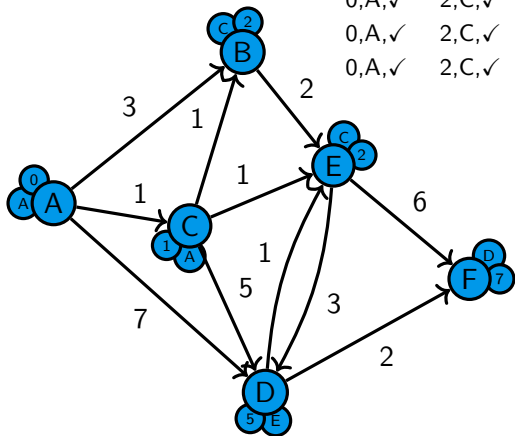
Shortest Path $A \rightarrow F$



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

Shortest Path $A \rightarrow F$

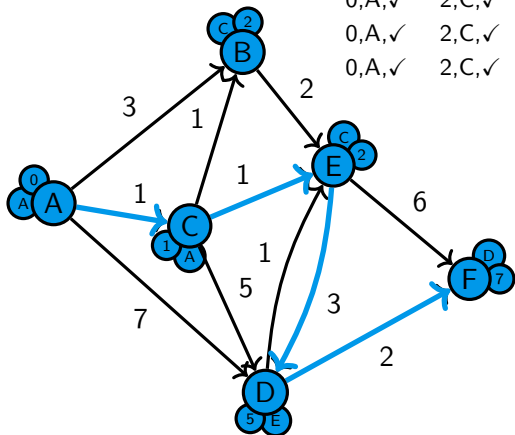


A	B	C	D	E	F	finished
0,A	∞ ,B	∞ , C	∞ ,D	∞ ,E	∞ ,F	
0,A,✓	3,A	1,A	7,A	∞ ,E	∞ ,F	A
0,A,✓	2,C	1,A,✓	6,C	2,C	∞ ,F	C
0,A,✓	2,C,✓	1,A,✓	6,C	2,C	∞ ,F	B
0,A,✓	2,C,✓	1,A,✓	5,E	2,C,✓	8,E	E
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D	D
0,A,✓	2,C,✓	1,A,✓	5,E,✓	2,C,✓	7,D,✓	F

Example: Execution of Dijkstra's algorithm

finished

Shortest Path $A \rightarrow F$



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

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

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

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, m = the total number of edges.

We do the following *up to* n times:

- Mark w as finished.
- Update every neighbour of w .
- Find w which is unfinished and with the smallest $d[w]$.

Representing the graph by an *adjacency matrix*, means that, over all n outer loops, it takes:

- $O(n)$ to do step a
- $O(n^2)$ to do step b
- $O(n^2)$ to do step c by going through all vertices.

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

Dijkstra's time complexity (adjacency lists)

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

- Mark w as finished.
- Update every neighbour of w .
- Find w which is unfinished and with the smallest $d[w]$.

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

neighbours of the 1st selected w ,
neighbours of the 2nd selected w ,
neighbours of the 3rd selected w ,

...

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

Representing the graph by an *adjacency list*, means that, over all n outer loops, it takes:

- $O(n)$ to do step a
- $O(m)$ to do step b
- $O(n^2)$ to do step c by going through all vertices.

\Rightarrow The time complexity is $O(n^2)$ (Note: $m \leq n^2$ in a simple graph)

Dijkstra's time complexity (adjacency lists)

Speeding up step c

Use min-priority queue: The priority of u is $d[u]$.

- Initialise the queue by inserting all nodes into it
- Call `deleteMin` to find the unfinished node with smallest $d[w]$
 - once per iteration, i.e. up to n times in total
- Whenever $d[u]$ changes, we `update` the priority of u .

⇒ total time complexity of step c

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

- Using Binary Heap: $O(n \log n + m \log n)$
- Using Fibonacci Heap: $O(n \log n + m)$

What is omitted in the analysis is the time complexity of initialising the heap. This is usually done by `heapify` and its time complexity was always $O(n)$ for all heaps we had. Alternatively, we can do `insert` n -times which will result in the time complexity $O(n \log n)$ or $O(n)$ 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
$O(n^2)$	$O((n + m) \log n)$	$O((n \log n) + m)$

Min-priority queues:

- Binary heaps: both `update` and `deleteMin` are in $O(\log n)$.
- Fibonacci heaps: `update` is in $O(1)$ and `deleteMin` is in $O(\log n)$ (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, m , is approximately n^2 , then using adjacency lists together with binary heaps has the time complexity $O((n + n^2) \log n) = O(n^2 \log n)$ which is slower than just using adjacency matrices. This problem disappears when using Fibonacci heaps where, for dense graphs, the time complexity becomes $O(n \log n + n^2) = O(n^2)$.

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)

```
1  dijkstra_with_matrix(int [][] G, int v, int z) {
2      n = G.length;
3      d = new int[n]; p = new int[n]; f = new bool[n];
4
5      for (int w = 0; w < n; w++) {
6          d[w] = infty;    p[w] = w;    f[w] = false;
7      }
8      d[v] = 0;
9
10     while (true) {
11         w = min_unfinished(d, f);
12         if (w == -1)
13             break;
14
15         for (int u = 0; u < n; u++)
16             update(w, u, d, p);
17
18         f[w] = true;
19     }
20     // compute results in desired form
21     return compute_result(v, z, G, d, p);
22 }
```



```
1 int min_unfinished(int[] d, bool[] f) {  
2     int min = infity;  
3     int idx = -1;  
4  
5     for (int i=0; i < d.length; i++) {  
6         if ( (not f[i]) && d[i] < min) {  
7             idx = i;  
8             min = d[i]  
9         }  
10    }  
11  
12    return idx;  
13 }
```

```
1 void update(w, u, G, d, p) {  
2     if (d[w] + G[w][u] < d[u]) {  
3         d[u] = d[w] + G[w][u];  
4         p[u] = w;  
5     }  
6 }
```

Dijkstra's algorithm (pseudocode with adjacency lists)

```
1  dijkstra_with_lists(List<Edge>[] N, int v, int z) {
2      n = G.length;
3      d = new int[n];    p = new int[n];
4      Q = new MinPriorityQueue();
5
6      for (int w = 0; w < n; w++) {
7          d[w] = infty;    p[w] = w;
8          Q.add(w, d[w]);
9      }
10     d[v] = 0;
11     Q.update(v, 0);
12
13     while (Q.notEmpty()) {
14         w = Q.deleteMin();
15
16         for (Edge e : N[w]) { // iterate over edges to neighbours
17             u = e.target;
18             if (d[w] + e.weight < d[u]) { // should we update?
19                 d[u] = d[w] + e.weight;
20                 p[u] = w;
21                 Q.update(u, d[u]);
22             }
23         }
24     }
25     return compute_result(v, z, G, d, p);
26 }
```

```
1  class Edge {
2      // target node
3      int target;
4
5      int weight;
6  }
```

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 $A \rightarrow B$ of weight 3, $A \rightarrow C$ of weight 1, and $A \rightarrow D$ of weight 8, then we will have that the linked list `N[v]` stores `Edge(B, 3)`, `Edge(C, 1)` and `Edge(D, 8)`.