Shortest paths

László Papp

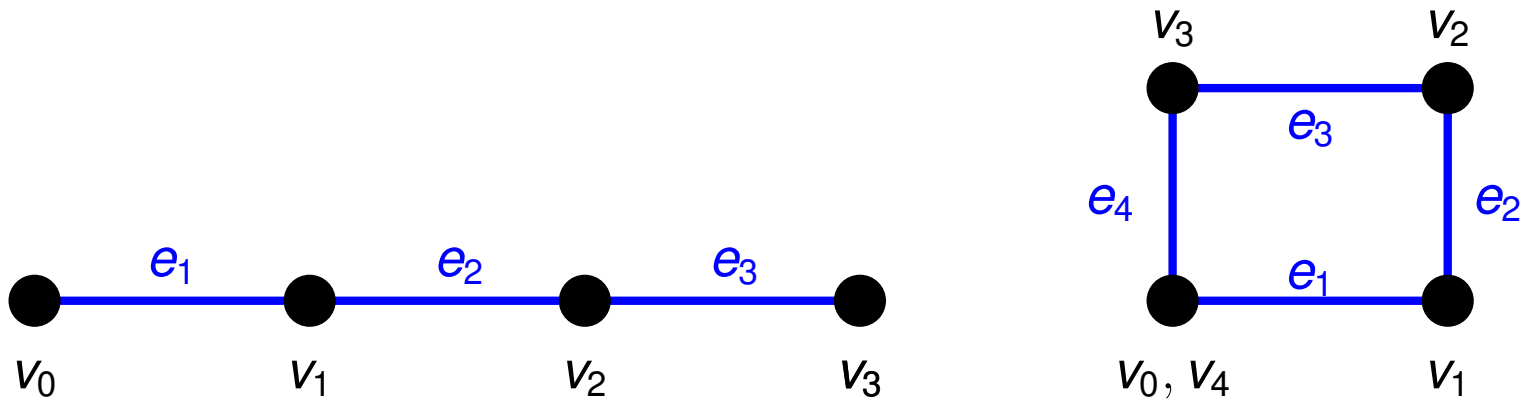
BME

31st of March, 2023

Distance in a graph

Definition: The **length** of a path or a cycle is the number of its edges.

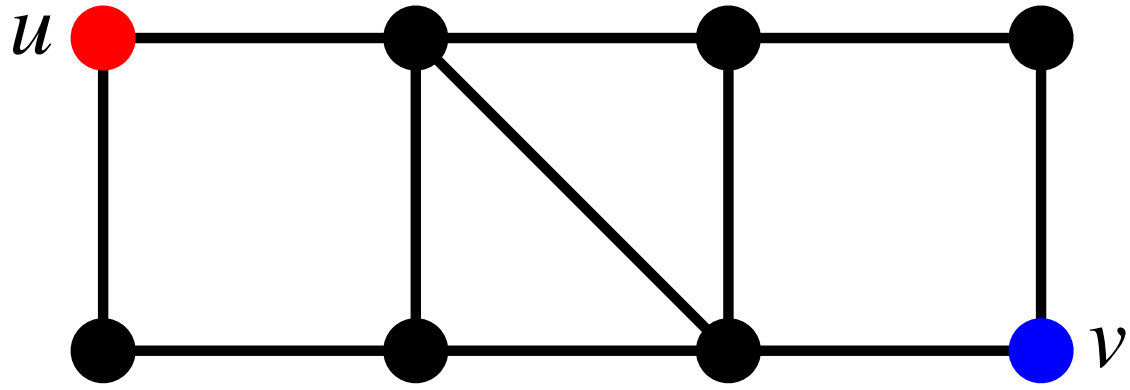
Example: The lenght of the path below is 3 and the lenght of the cycle below is 4.



Definition: The **distance between vertices** u **and** v in a (directed) graph is the length of the shortest (directed) path which starts from u and ends at v. If there is no (directed) path between u and v, then we say that the distance between u and v is infinite. We denote this value by $dist(u, v)$.

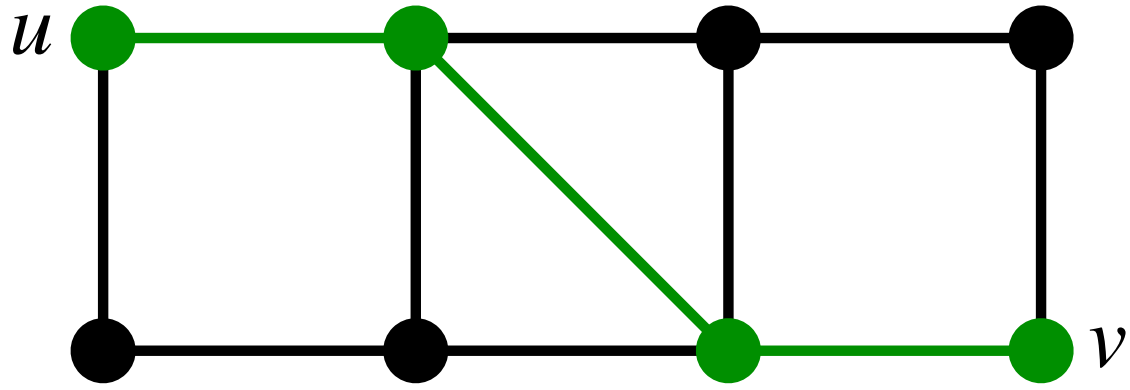
Remark: If the graph is directed, then the values $dist(u, v)$ and $dist(v, u)$ can be different!

Distance in graph: Examples

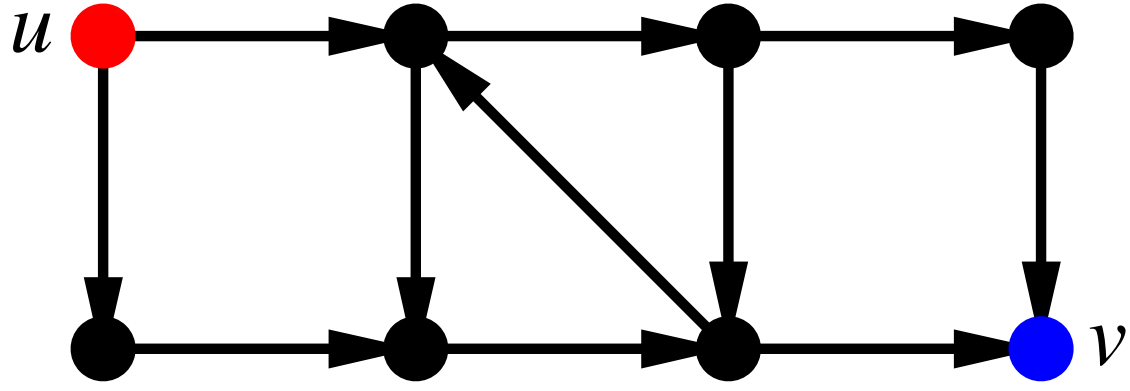


$dist(u, v) = ?$

Distance in graph: Examples

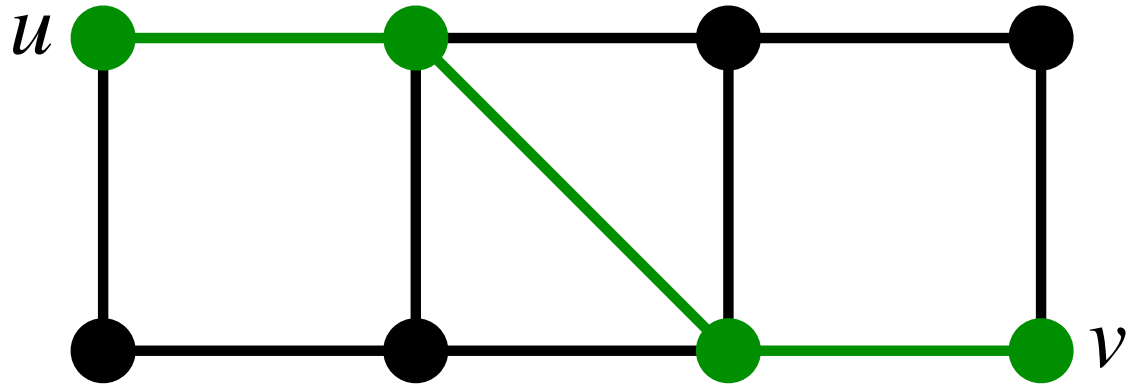


$dist(u, v) = 3$

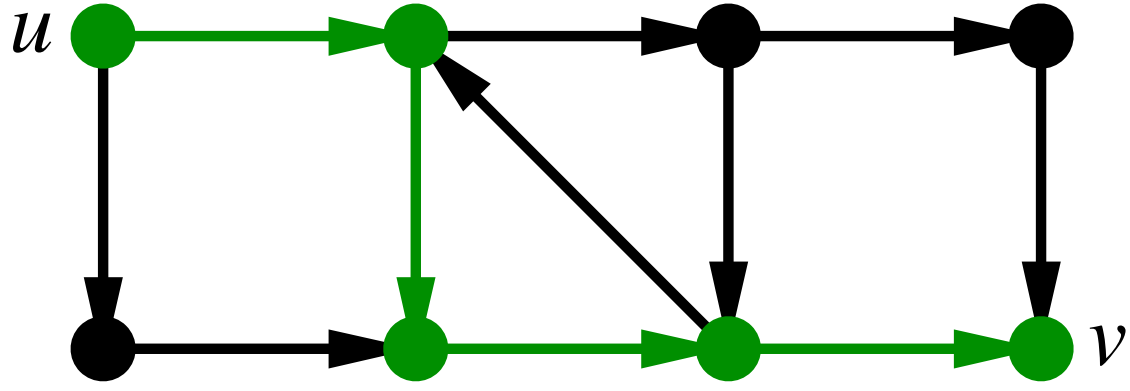


$dist(u, v) = ?$ $dist(v, u) = ?$

Distance in graph: Examples



$dist(u, v) = 3$



$dist(u, v) = 4 \quad dist(v, u) = \infty$

Breadth-first search:

To calculate $dist(v, w)$, we traverse the graph by an algorithm called Breadth-first search (BFS). We start it from a vertex v.
A naive description of the algoirthm:

"After we visit the starting vertex we visit its neighbors, then we visit the neighbors of the neighbors, etc.. . . "

The BFS algorithm uses a queue (FIFO list: first in first out) which has two operations:

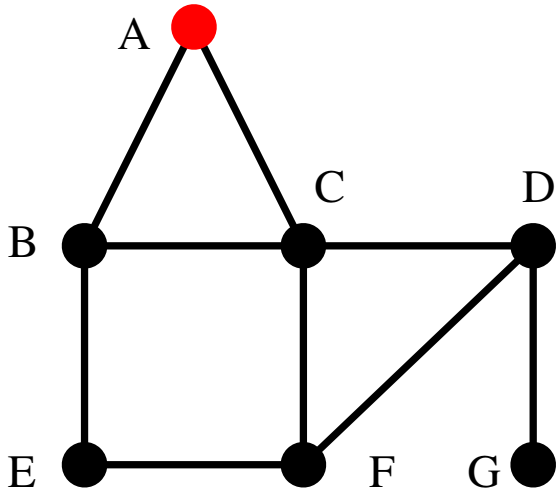
- **Enqueue(x)**: Add element x to the end of the list.
- **Dequeue()**: Remove and return the first element of the list.

The algorithm uses an array or a labelling to store wether a vertex has been explored or not.
The algorithm also builds up a traversal tree, which is a directed cycleless graph and it shows that what edges were used to visit new vertices. An alternative is to store parent labels for vertices instead. If a vertex u is explored by crossing edge v, u, then we say that v is the parent of u.

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

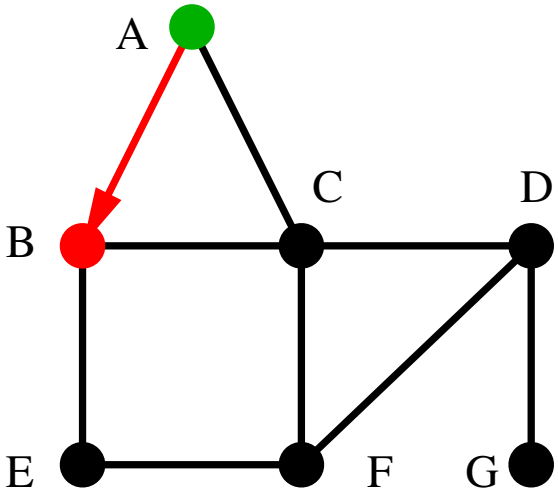


The root is A. The blue edges form the traversal tree.
Q: [A]

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v,u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

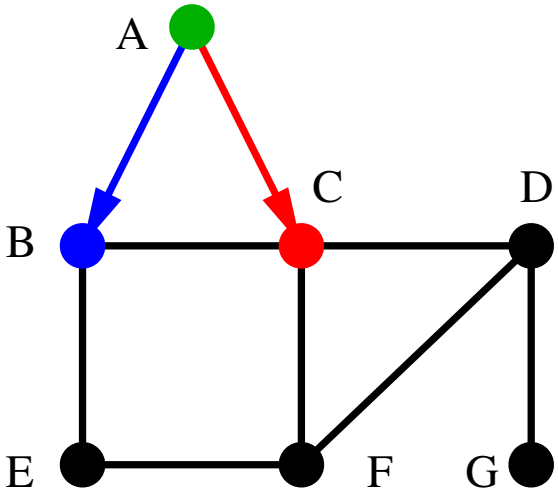


The root is A. The blue edges form the traversal tree.
Q: [B]
$p(B) = A$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

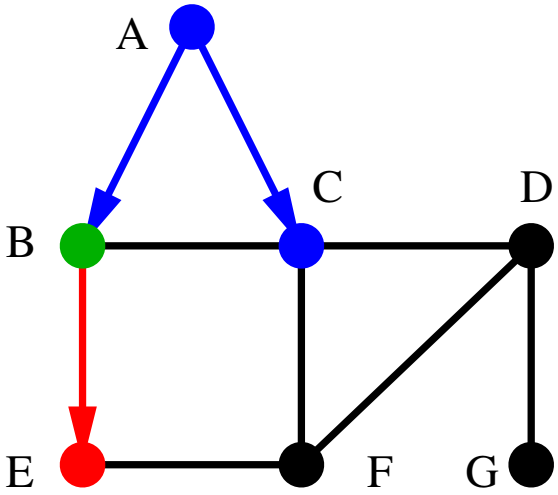


The root is A. The blue edges form the traversal tree.
Q: [B, C]
$p(B) = A$, $p(C) = A$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

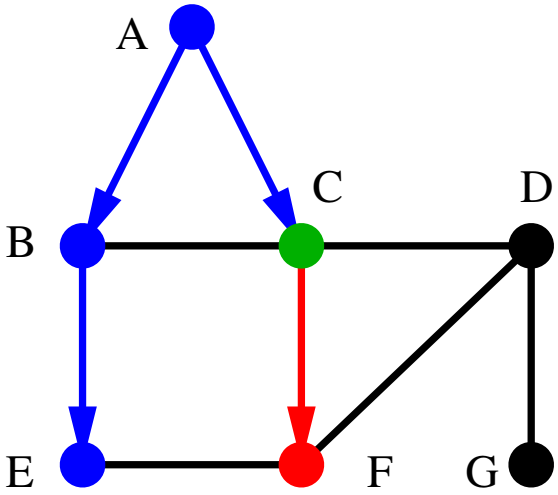


The root is A. The blue edges form the traversal tree.
Q: [C, E]
$p(B) = A$, $p(C) = A$, $p(E) = B$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

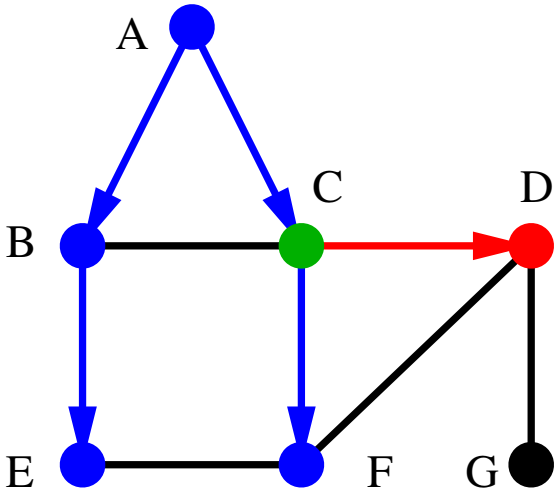


The root is A. The blue edges form the traversal tree.
Q: [E, F]
$p(B) = A$, $p(C) = A$, $p(E) = B$, $p(F) = C$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

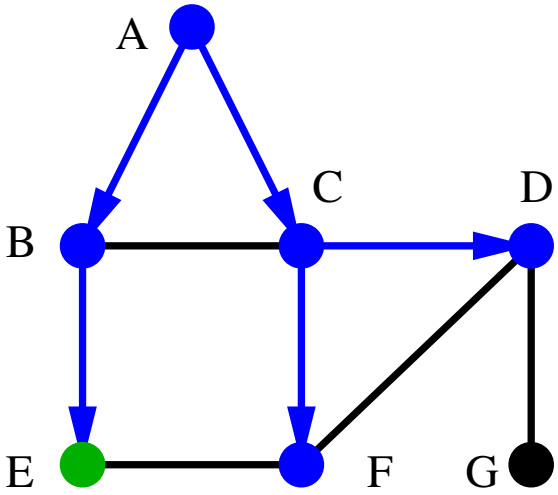


The root is A. The blue edges form the traversal tree.
Q: [E, F, D]
$p(B) = A$, $p(C) = A$, $p(E) = B$,
$p(F) = C$, $p(D) = C$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

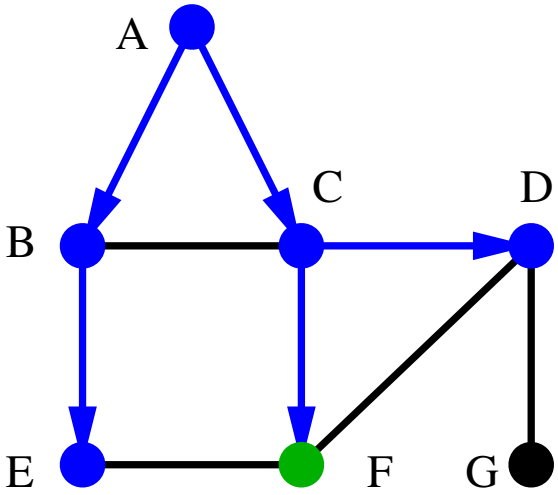


The root is A. The blue edges form the traversal tree.
Q: [F, D]
$p(B) = A$, $p(C) = A$, $p(E) = B$,
$p(F) = C$, $p(D) = C$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

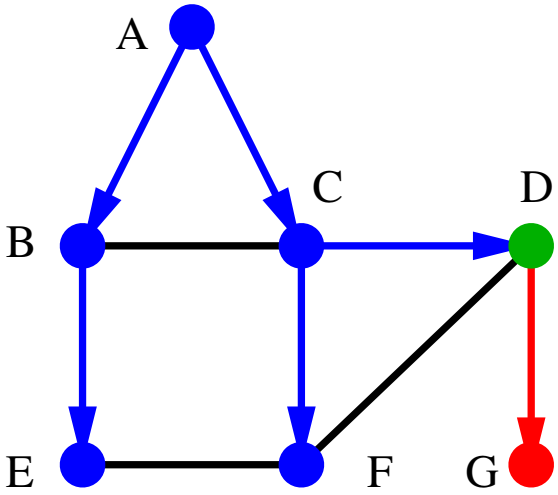


The root is A. The blue edges form the traversal tree.
Q: [D]
$p(B) = A$, $p(C) = A$, $p(E) = B$,
$p(F) = C$, $p(D) = C$

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.

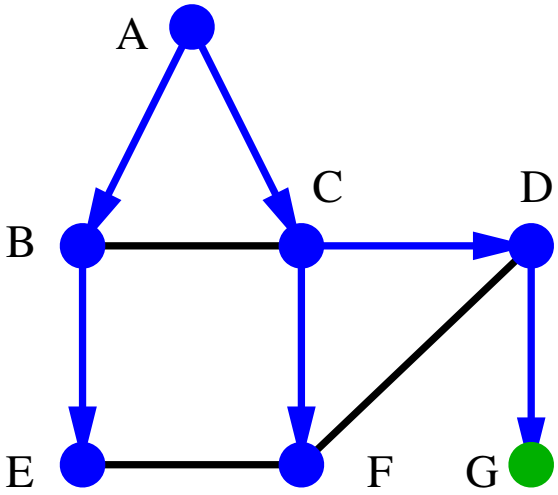


The root is A. The blue edges form the traversal tree.
Q: [G]
$p(B) = A$, $p(C) = A$, $p(E) = B$,
$p(F) = C$, $p(D) = C$,
$p(G) = D$.

Breadth-first search, BFS

Input: A graph G (or a directed graph $\vec{G}$) and a start vertex v_0 which is usually called as the root.

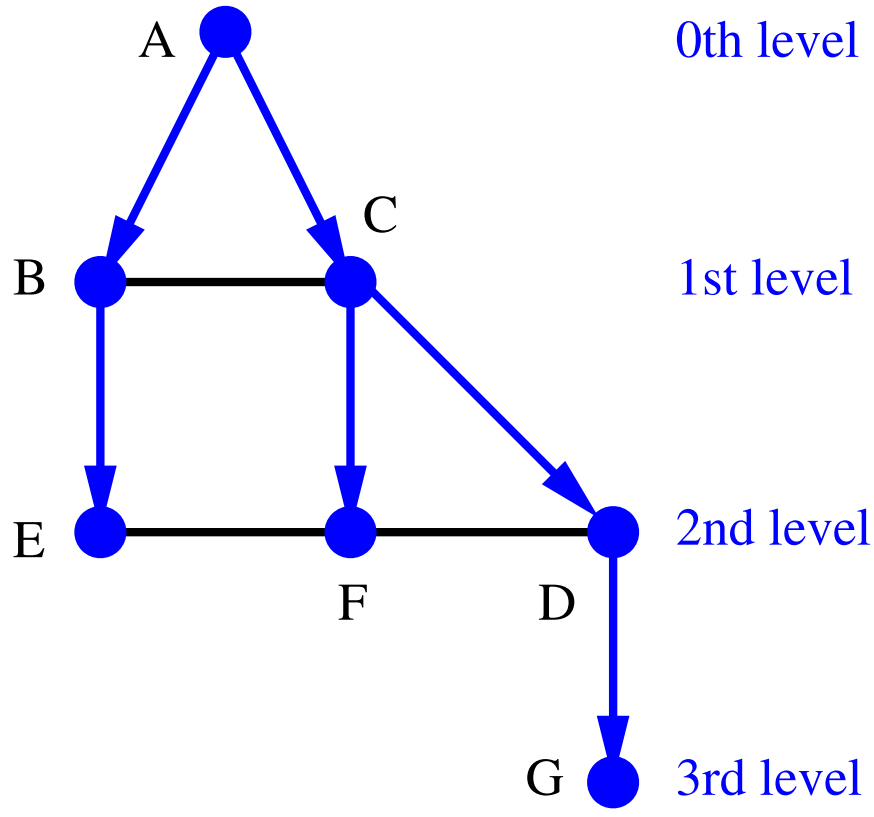
1. Add v_0 to queue Q (Enqueue(v_0)) and label v_0 as explored.
2. Let v be the first element of Q and remove it from Q ($v :=$Dequeue()).
3. If v has a not explored (out-)neighbor u, then label u as explored, add u to the end of Q (Enqueue(u)), add $\{\vec{v, u}\}$ to the traversal tree, set $p(u) = v$ and go to step 3.
4. If the traversal que Q is nonempty, then go to step 2, otherwise STOP.



The root is A. The blue edges form the traversal tree.
Q: [ ]
$p(B) = A$, $p(C) = A$, $p(E) = B$,
$p(F) = C$, $p(D) = C$,
$p(G) = D$.

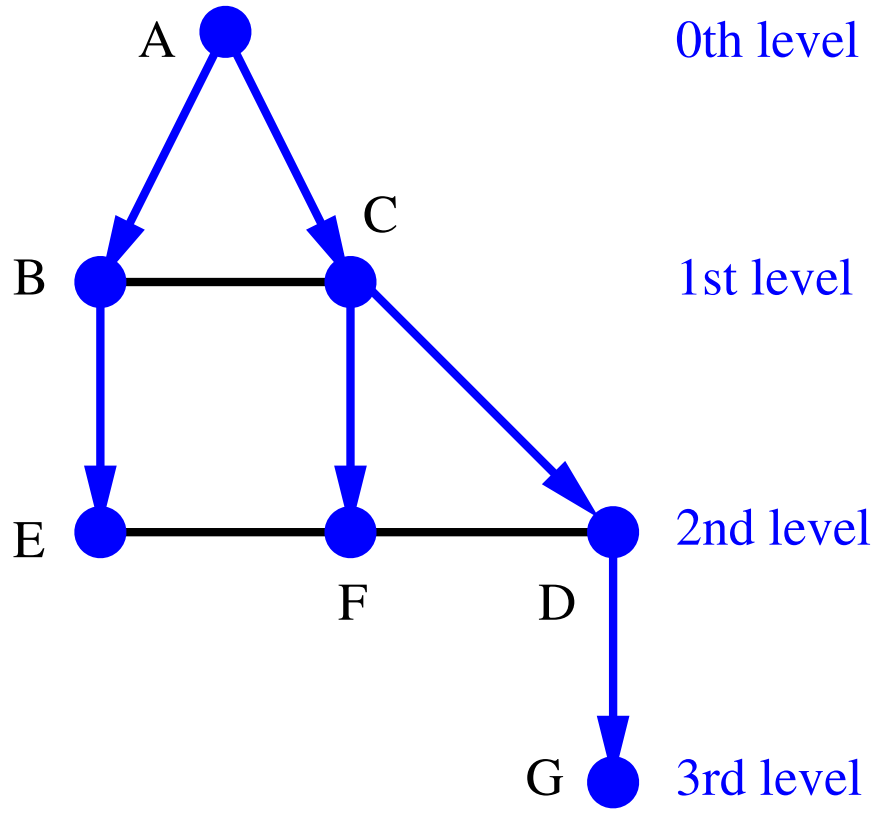
The BFS tree

The directed path contained in the BFS tree between A and G can be obtained by the reversal of $G, p(G) = D, p(p(G)) = C, p(p(p(G))) = A$.
Let's say that the level of a vertex v is the number of times that p() need to be called to v to obatin the root vertex A, which is equals to the length of the path connecting A nad v in the BFS tree.



The BFS tree

The directed path contained in the BFS tree between A and G can be obtained by the reversal of $G, p(G) = D, p(p(G)) = C, p(p(p(G))) = A$.
Let's say that the level of a vertex v is the number of times that p() need to be called to v to obatin the root vertex A, which is equals to the length of the path connecting A nad v in the BFS tree.



Claim: The level of v equals to $dist(A, v)$.
Corollary: The vertices contained at level k are the set of vertices whose distance from the root is exactly k.

Analyzing the *BFS*

Question: What is the time complexity of the BFS?

Hand-shaking lemma

In any finite graph the sum of the degrees equals twice the number of the edges: $\sum_{v \in V(G)} d(v) = 2e(G)$.

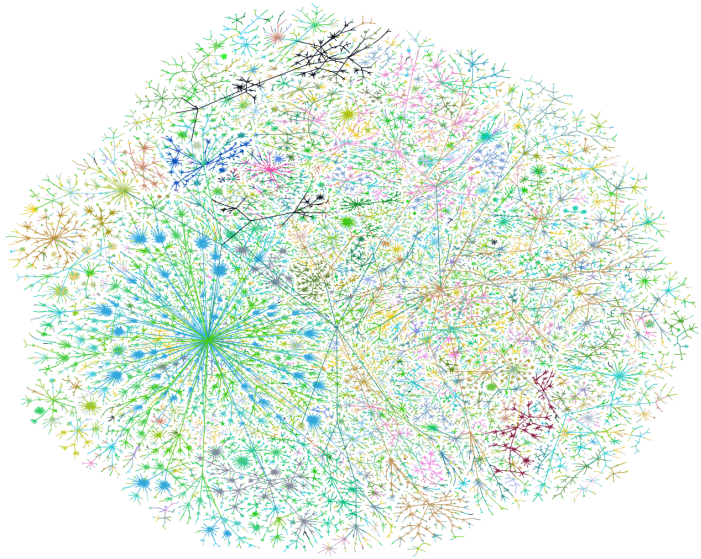
During the BFS we explore each vertex at most once and for each explored vertex we look for all of its (out-)neighbors. If the graph is given by an adjacency list, then looking for all the neighbors of a vertex v takes $d(v)$ operations. So these requires at most $n + \sum_{v \in V(G)} d(v) = n + 2e \in O(n + e)$ operations.
Adding an element to the end of the traversal queue Q and removing the first element of Q can be done in constant steps. Each vertex is added and deleted at most once, therefore the usage of the traversal queue requires $O(n)$ steps. Setting the labeling of all vertices can be done in O(n) as well.
All in all the time complexity of the BFS algorithm is in $O(n + e)$.

Another application of BFS:

We can decide algorihmically whether a given undirected graph is connected by running the *BFS* algorithm. If BFS visits all the vertices, then the graph is connected, otherwise it is not.

Corollary

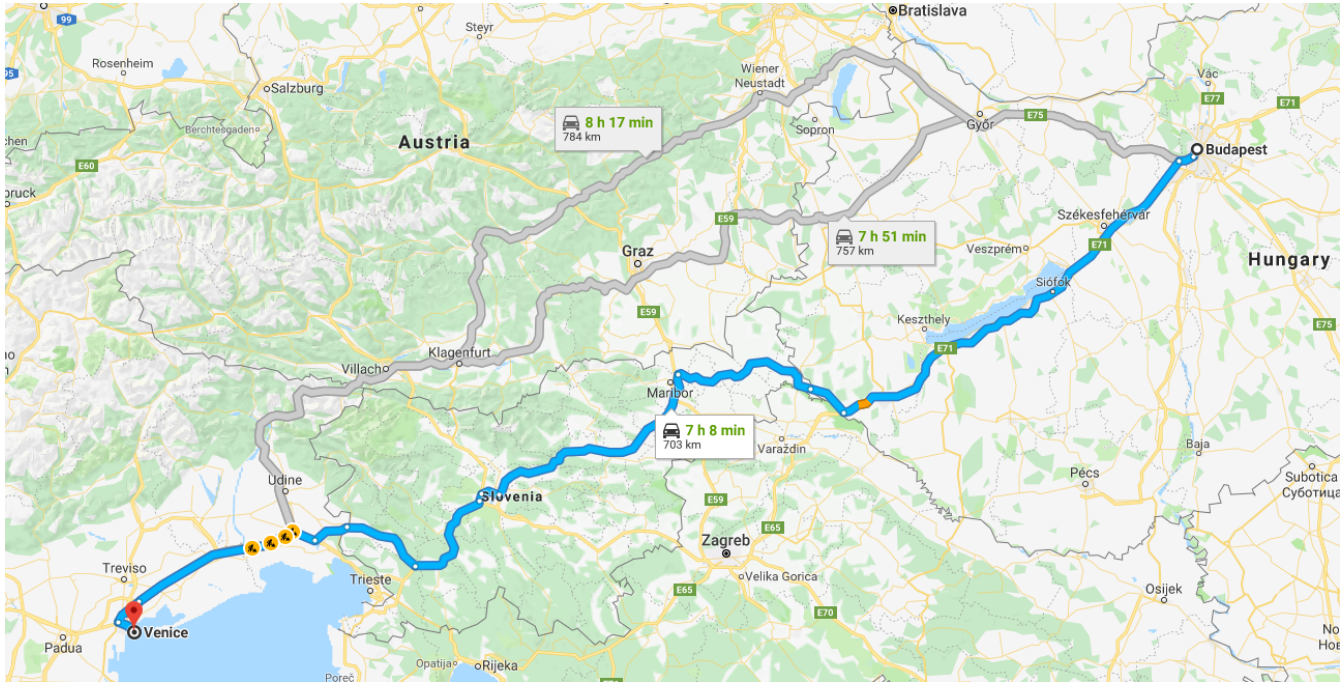
We can decide whether a graph is connected in polynomial time.



The BFS is a kind of building-block of many algorithms. Next week we will see an algorithm which uses the BFS algorithm several times.

Shortest path in real world applications

The distance is not the number of edges, each edge has its own length, which can be any real number. In this course we are speaking about non-negative lengths.



So a length function $l : E(G) \to \mathbb{R}^+$ is given, which tells the length of each edge.

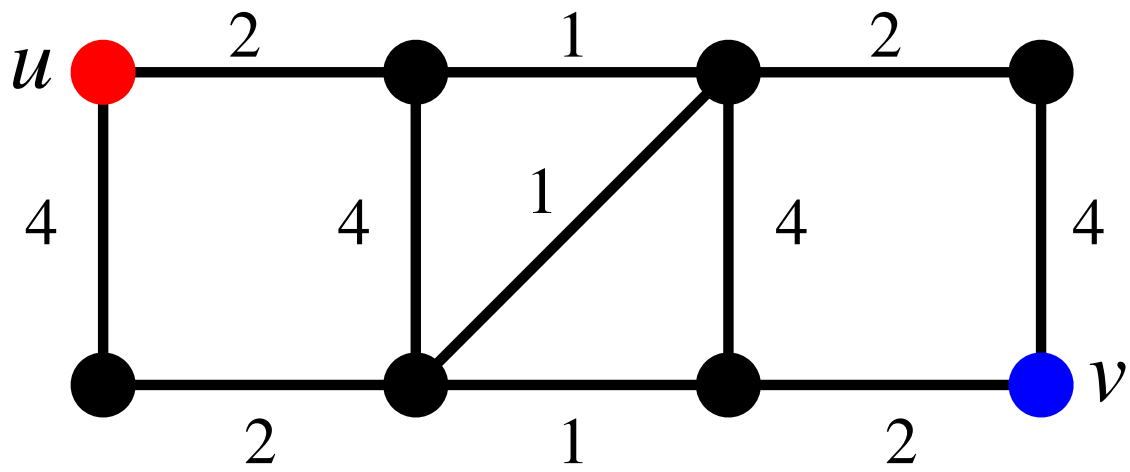
Distance given by a length function

Definition

The **length (given by the function l) of a path** containing edges $e_1, e_2, \ldots, e_k$ is $\sum_{i=1}^{k} l(e_i)$. So it is the sum of the length of its edges.

The **distance** between vertices u and v in a graph (in a directed graph) is the length of a shortest (directed) path between u and v.

Example:



$dist(u, v) = ?$

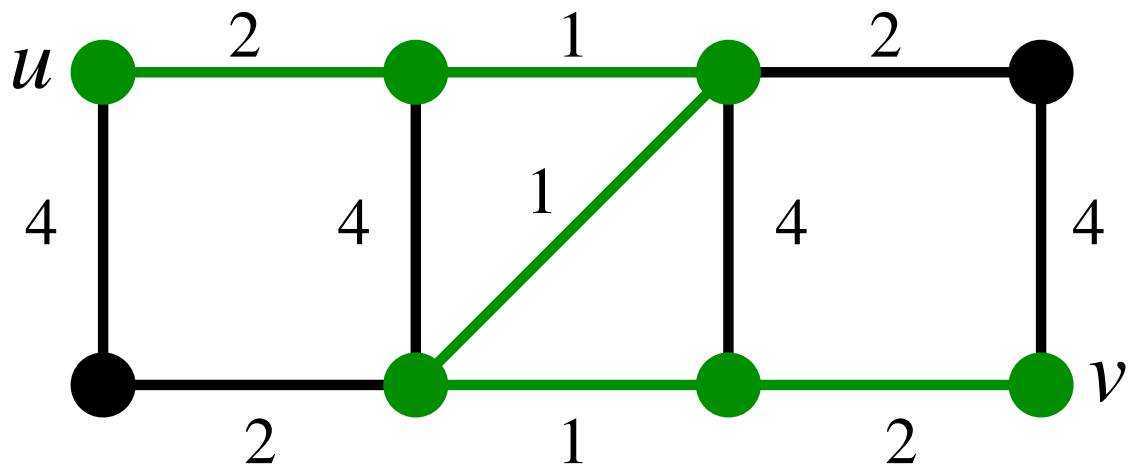
Distance given by a length function

Definition

The **length (given by the function l) of a path** containing edges $e_1, e_2, \ldots, e_k$ is $\sum_{i=1}^{k} l(e_i)$. So it is the sum of the length of its edges.

The **distance** between vertices u and v in a graph (in a directed graph) is the length of a shortest (directed) path between u and v.

Example:

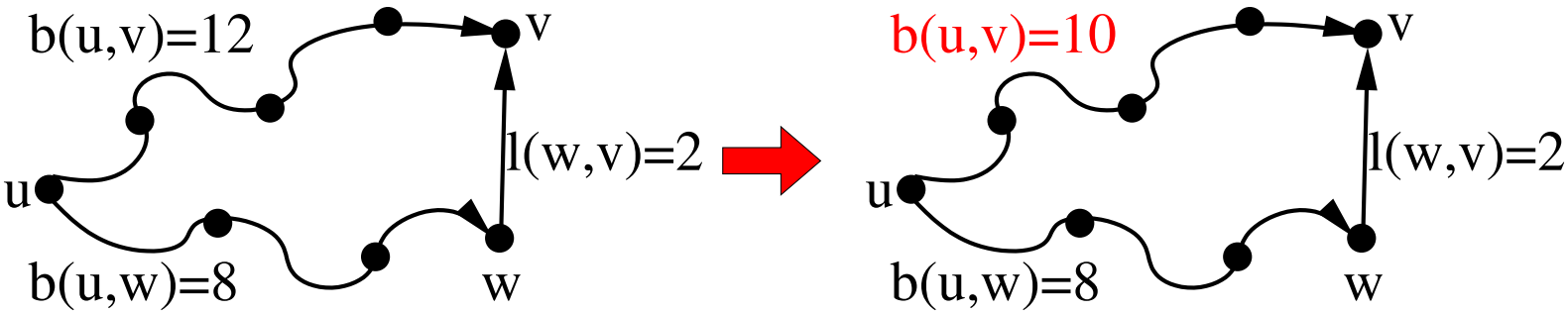


$dist(u, v) = 7$

Calculating the shortest path

There are many different algorithms which find the shortest path between a given pair of vertices. Most of them use the concept of **improving an upper bound by an edge**:

Assume that we have an upper bound on each distance in the graph, so we have a function b which satisfies that $b(u, v) \geq dist(u, v)$ for any u, v pair of vertices.



If $\{w, v\}$ is an edge and $b(u, w) + l(\{w, v\}) < b(u, v)$, then we replace $b(u, v)$ with $b(u, w) + l(\{w, v\})$ and we obtain a smaller upper bound function.

Dijkstra's algorithm

Input: Let G be an undirected (or directed) graph and l be a non-negative length function over the edge set of G, so $l : E(G) \to \mathbb{R}^+ \cup 0$ and let $s \in V(G)$ be a given vertex.

The algorithm determines the $dist(s, v)$ distance for any vertex v. During the algorithm we update the $d(v)$ value which is an upper bound on $dist(s, v)$. We place the vertices one by one to a set called FINISHED. If vertex v is in FINISHED it means that $d(v) = dist(s, v)$. During this procedure we also calculate a mapping $V(G) \to V(G) \cup \emptyset$. The semantics of this F is the following: If $dist(s, v) < \infty$, then there is a shortest s, v (directed) path whose last edge is $\{F(v), v\}$. (F() works like the parent function p() in BFS.)

At the end of the algorithm $d(v) = dist(s, v)$ for all the vertices. The algorithm also gives a tree which, for each vertex v, contains a shortest path between s and v.

Dijkstra's algorithm

Input: Undirected or directed graph G, non-negative length function $l : E(G) \to \mathbb{R}^+ \cup 0$ and a vertex s.

0. FINISHED$= \{s\}$, $d(s) := 0$, $d(v) = \infty$ for any vertex v which is not s and let $u := s$.

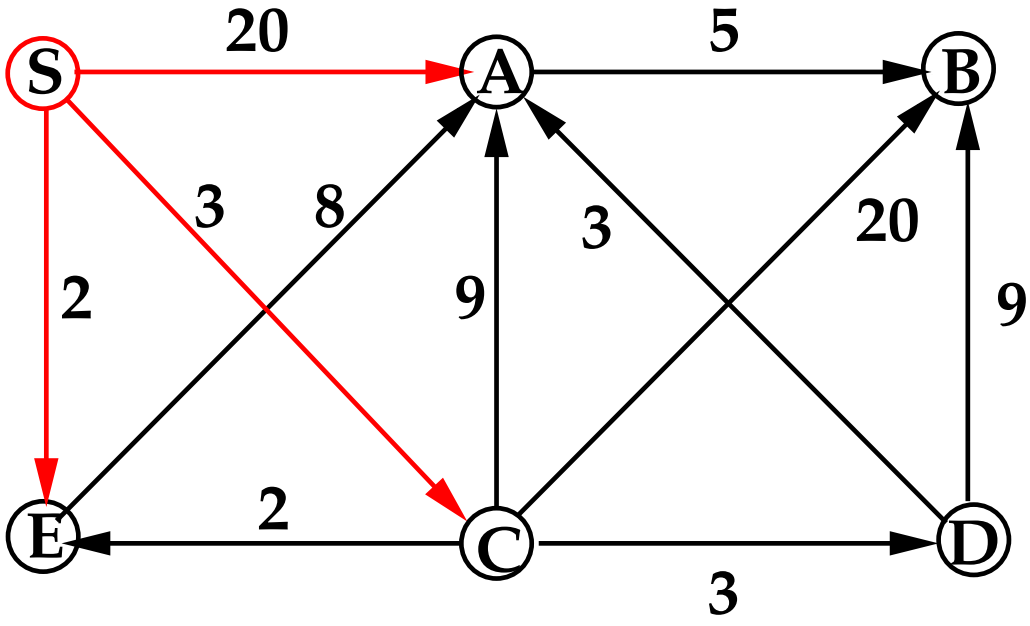
1. Try to improve the $d()$ upper bound by all the edges $\{u, \vec{v}\}$ (directed edges {u,v} in case of a directed graph) where v is not in FINISHED: If $d(u) + l(\{u, v\}) < d(v)$, then set $d(v)$ to $d(u) + l(\{u, v\})$ and $F(v) := u$.

2. Set u to be the vertex whose $d()$ is minimal among the vertices which are not contained in FINISHED. Add u to FINISHED and mark the edge $\{F(u), u\}$.

3. If FINISHED does not contain all the vertices, then go to step 1.

Output: The functions $d(v)$, $F(v)$ and the set of marked edges.

Example for Dijkstra's algorithm

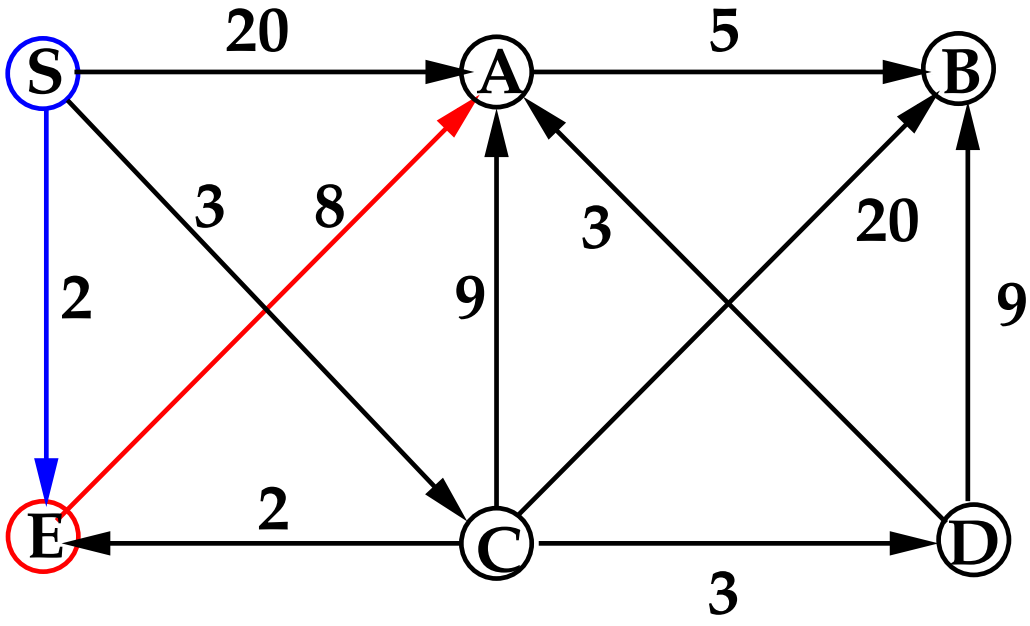


FINISHED={S}

d()						
S	0	0				
A	∞	20				
B	∞	∞				
C	∞	3				
D	∞	∞				
E	∞	2				

F()					
S					
A	S				
B					
C	S				
D					
E	S				

Example for Dijkstra's algorithm

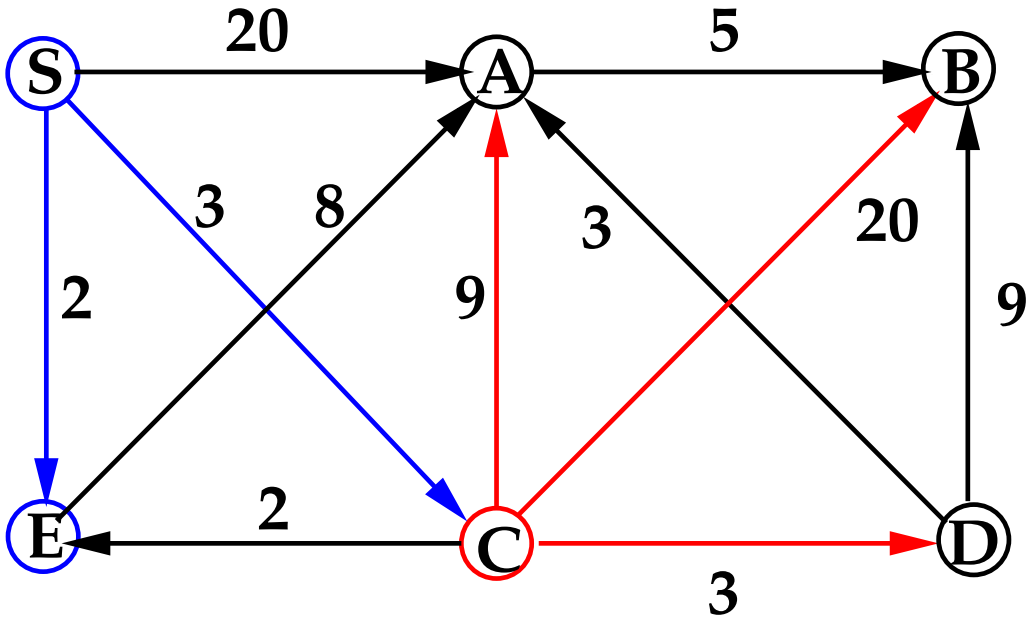


FINISHED={S,E}

d()						
S	0	0	0			
A	∞	20	10			
B	∞	∞	∞			
C	∞	3	3			
D	∞	∞	∞			
E	∞	2	2			

F()					
S					
A	S	E			
B					
C	S	S			
D					
E	S	S			

Example for Dijkstra's algorithm

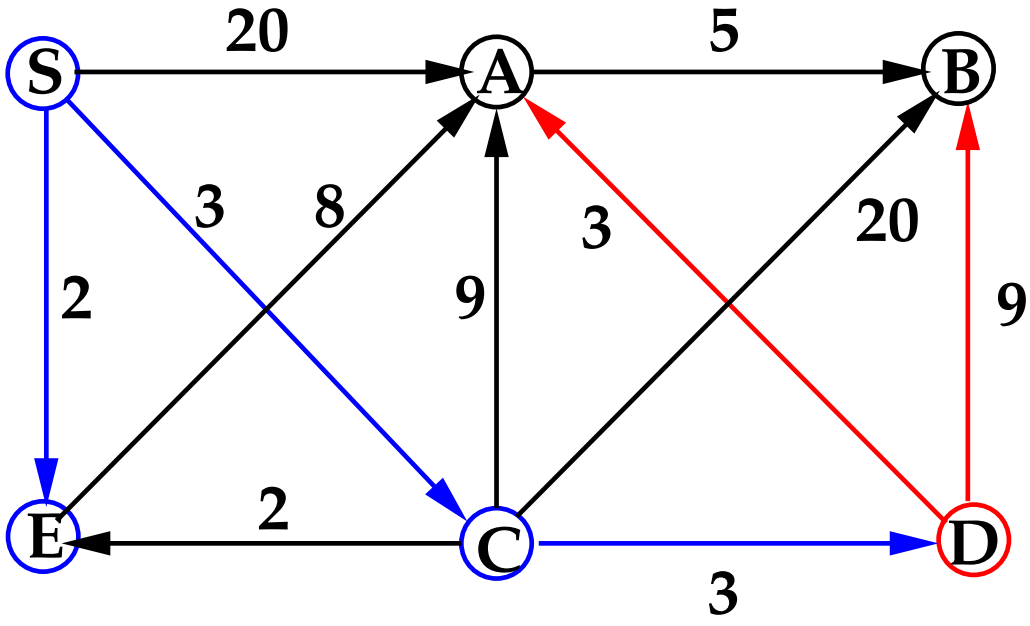


FINISHED={S,E,C}

d()						
S	0	0	0	0		
A	∞	20	10	10		
B	∞	∞	∞	23		
C	∞	3	3	3		
D	∞	∞	∞	6		
E	∞	2	2	2		

F()					
S					
A	S	E	E		
B			C		
C	S	S	S		
D			C		
E	S	S	S		

Example for Dijkstra's algorithm

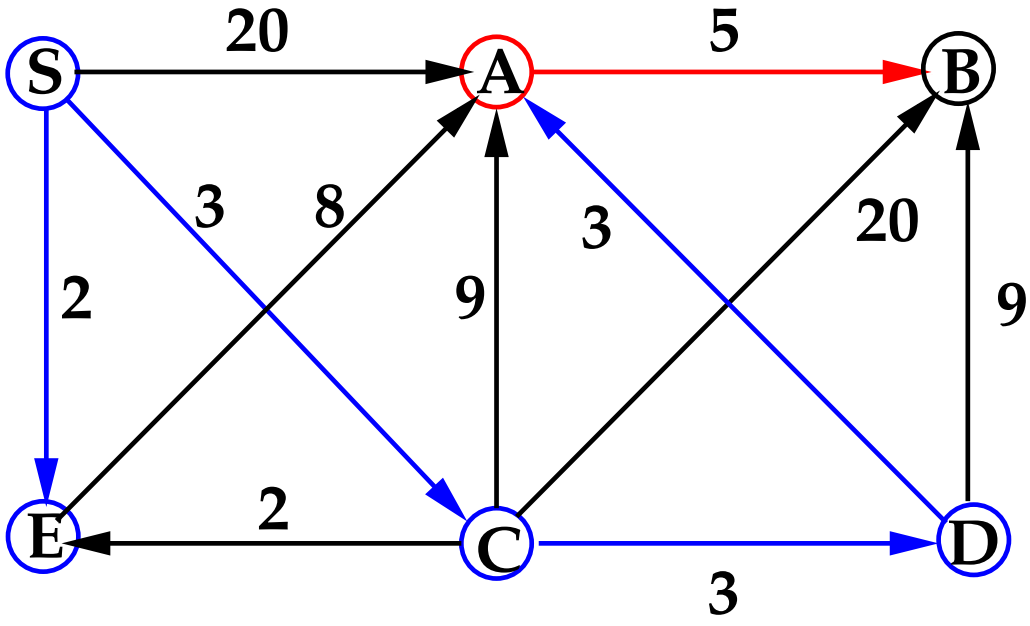


FINISHED={S,E,C,D}

d()						
S	0	0	0	0	0	
A	∞	20	10	10	9	
B	∞	∞	∞	23	15	
C	∞	3	3	3	3	
D	∞	∞	∞	6	6	
E	∞	2	2	2	2	

F()					
S					
A	S	E	E	D	
B			C	D	
C	S	S	S	S	
D			C	C	
E	S	S	S	S	

Example for Dijkstra's algorithm

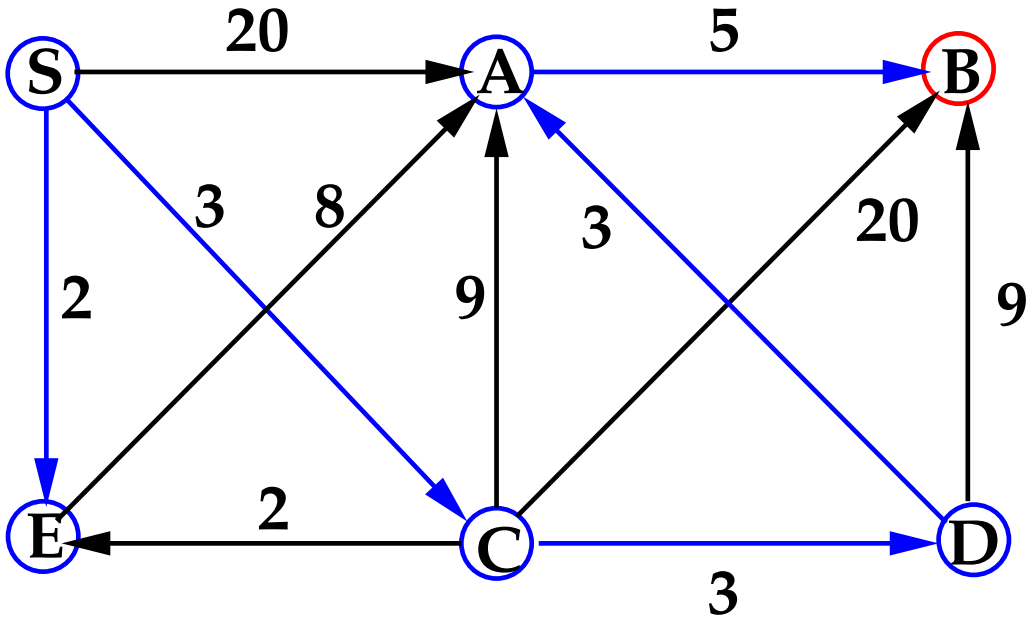


FINISHED={S,E,C,D,A}

d()						
S	0	0	0	0	0	0
A	∞	20	10	10	9	9
B	∞	∞	∞	23	15	14
C	∞	3	3	3	3	3
D	∞	∞	∞	6	6	6
E	∞	2	2	2	2	2

F()					
S					
A	S	E	E	D	D
B			C	D	A
C	S	S	S	S	S
D			C	C	C
E	S	S	S	S	S

Example for Dijkstra's algorithm



FINISHED={S,E,C,D,A,B}
dist(S,B)=14 and a shortest path between *S* and *B* is SCDAB.

d()						
S	0	0	0	0	0	0
A	∞	20	10	10	9	9
B	∞	∞	∞	23	15	14
C	∞	3	3	3	3	3
D	∞	∞	∞	6	6	6
E	∞	2	2	2	2	2

F()					
S					
A	S	E	E	D	D
B			C	D	A
C	S	S	S	S	S
D			C	C	C
E	S	S	S	S	S

How fast is Dijkstra's algorithm?

Let n be the number of vertices. After a vertex has been put to FINISHED we try to improve among some edges which are incident to it, therefore their number is less than n. We add all the n vertices to the FINISHED set, so all the improvement of $d()$ can be done in $O(n^2)$.
When we choose that which vertex should be added to FINISHED, we have to find a vertex whose current d() value is minimal. We can do it in O(n) steps. We do this n times, therefore this part of the algorithm can be done in $O(n^2)$ time.
Marking the edges of the tree containing the shortest paths can be done in $O(n^2)$ steps.
All the steps can be done in $O(n^2)$ time.

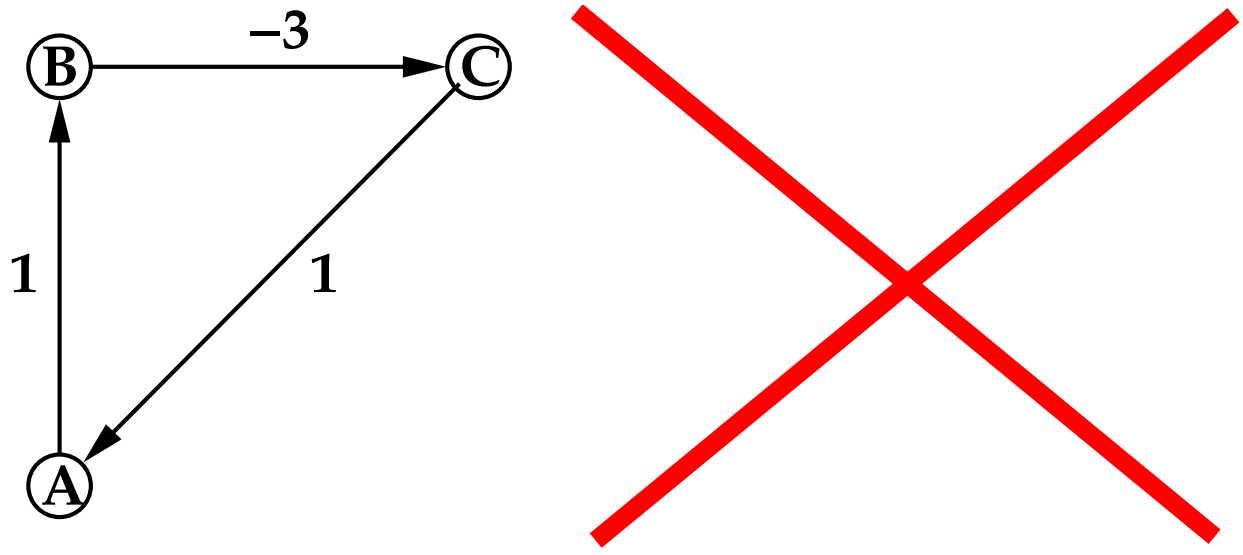
How fast is Dijkstra's algorithm?

Let n be the number of vertices. After a vertex has been put to FINISHED we try to improve among some edges which are incident to it, therefore their number is less than n. We add all the n vertices to the FINISHED set, so all the improvement of $d()$ can be done in $O(n^2)$.
When we choose that which vertex should be added to FINISHED, we have to find a vertex whose current d() value is minimal. We can do it in O(n) steps. We do this n times, therefore this part of the algorithm can be done in $O(n^2)$ time.
Marking the edges of the tree containing the shortest paths can be done in $O(n^2)$ steps.
All the steps can be done in $O(n^2)$ time.

Corollary: We can calculate the shortest paths starting from a vertex in polinomial time if there are no negative edge lengths.

Finding shortest paths in case of negative edge length

We can calculate the distance between any two vertex in polynomial time if negative edge lengths are allowed but there is no directed cycle of negative length. Dijkstra does not work for that case. If you are interested, then google for Bellman-Ford algorithm or Floyd's algorithm.



There is no known polynomial-time algorithm which find a shortest path between two vertices in a graph which contains a directed cycle of negative length.