VE281

Data Structures and Algorithms

Shortest Path

Learning Objectives:

- Know the shortest path problem
- Know Dijkstra's algorithm and its runtime complexity
- Know the similarity between Prim's algorithm and Dijkastra's algorithm

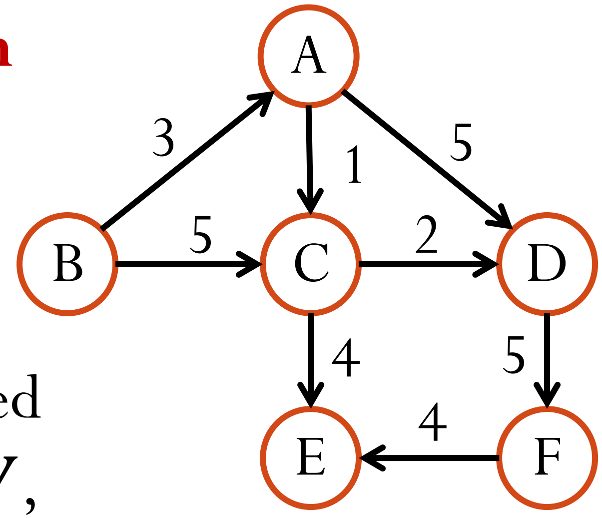
Outline

- Shortest Path Problem
  - Unweighted Graph
  - Dijkstra’s Algorithm

Shortest Path Problem

Introduction

- Given a weighted graph $G = (V, E)$, **path length** is defined as the sum of weights of edges on the path.
  - E.g., length of the path B, C, D, F is 12.
- **Shortest path problem**: given a weighted graph $G = (V, E)$ and two nodes $s, d \in V$, find the shortest path from s to d.
  - Assume G is a directed graph without parallel edges of the same direction
  - For an undirected graph, we can replace each edge by two edges of the same weight but of different directions.



What is the shortest path from B to F?

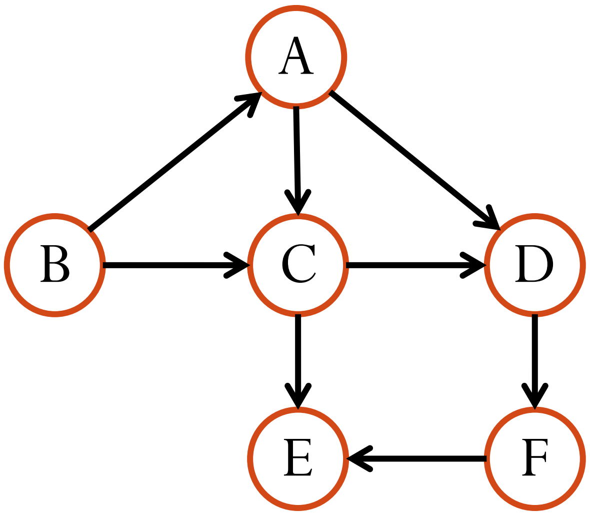
Shortest Path Problem

- The starting node on the path is the **source** node and the ending node is the **destination** node.
- The previous problem is a **single source single destination** problem.
- What we will solve is a **single source all destinations** problem: Given $G = (V, E)$ and a node $s \in V$, find the shortest path from s to **every other** node in G.
  - Single source single destination problem can be solved by solving the single source all destinations problem.
  - However, single source single destination problem is not much easier than the single source all destinations problem.

Shortest Path Problem

A Simple Version: Unweighted Graphs

- For an unweighted graph, path length is defined as the number of edges on the path.
- How do you obtain the shortest path between a pair of nodes?

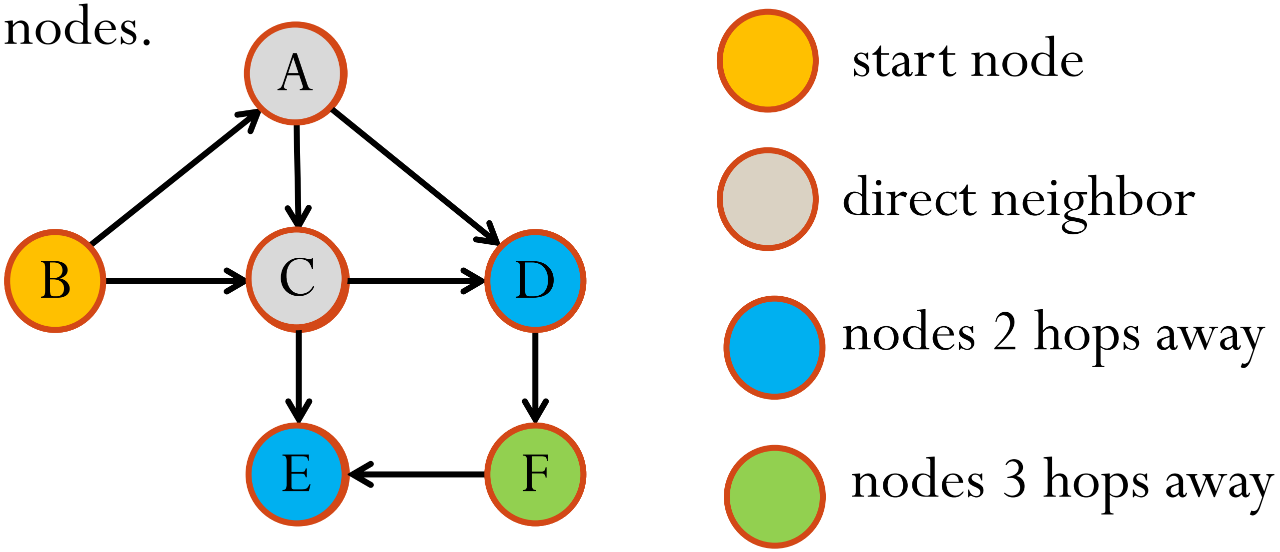


What is the shortest path from B to F?

Using breadth-first search!

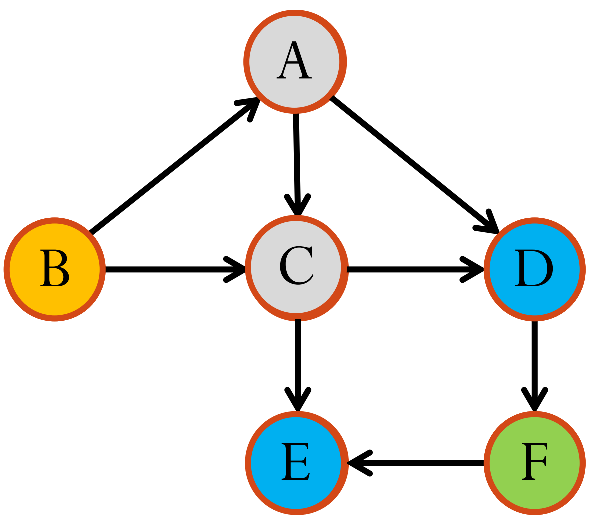
Shortest Path for Unweighted Graphs

- Recall breadth-first search (BFS): Given a start node, visit all directly connected neighbors first, then nodes 2 hops away, 3 hops away, and so on.
  - This is exactly what we want!
  - When the node visited is the destination node, we stop.
  - When the queue becomes empty, there is no path between the two nodes.



Shortest Path for Unweighted Graphs

- Additional bookkeeping
  - Store the distance.
  - Store the **predecessor** on the shortest path, i.e., the previous node on the path.



start node

direct neighbor

nodes 2 hops away

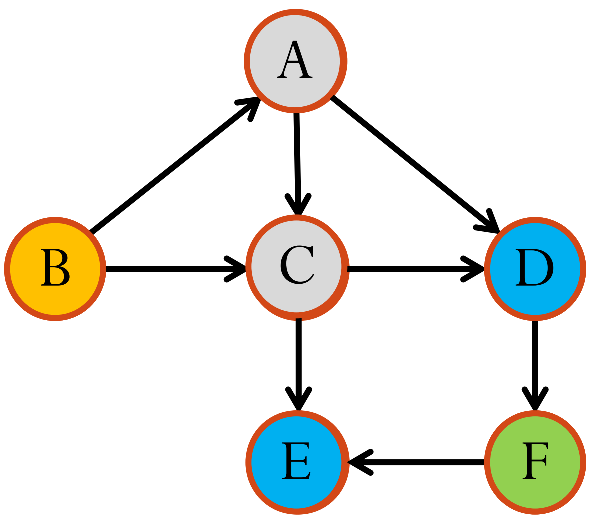
nodes 3 hops away

	A	B	C	D	E	F
dist	1	0	1	2	2	3
pred	B	-	B	A	C	D

Shortest Path for Unweighted Graphs

- We can obtain the shortest path by backtracking.
  - E.g., shortest path from B to F

B→A→D→F



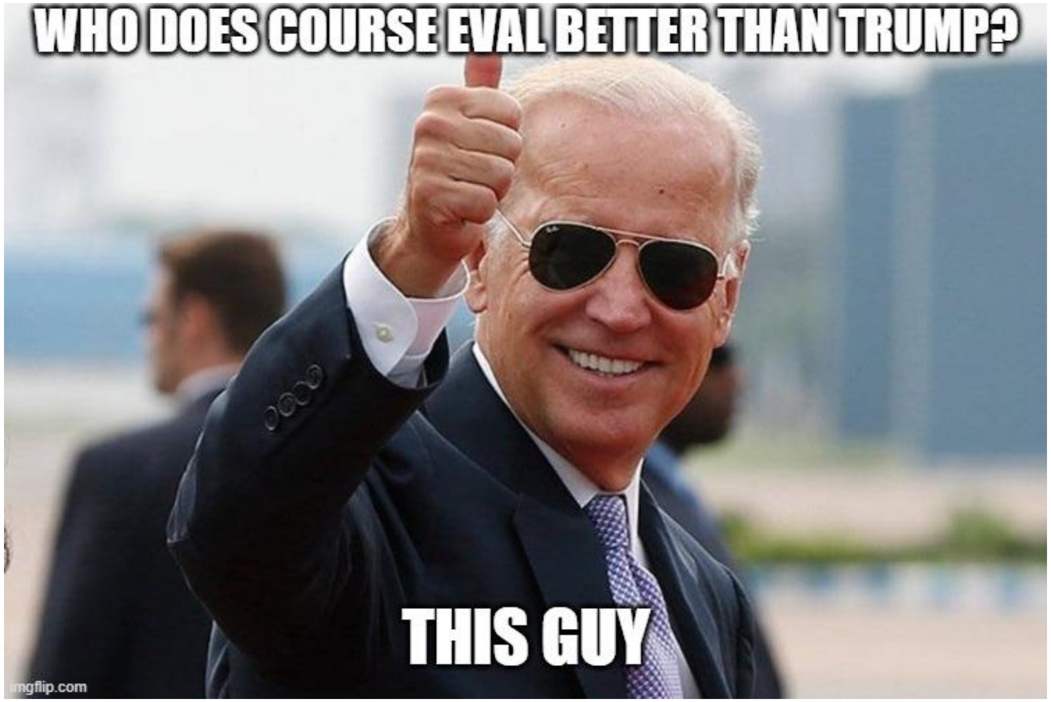
start node

direct neighbor

nodes 2 hops away

nodes 3 hops away

	A	B	C	D	E	F
dist	1	0	1	2	2	3
prev	B	-	B	A	C	D



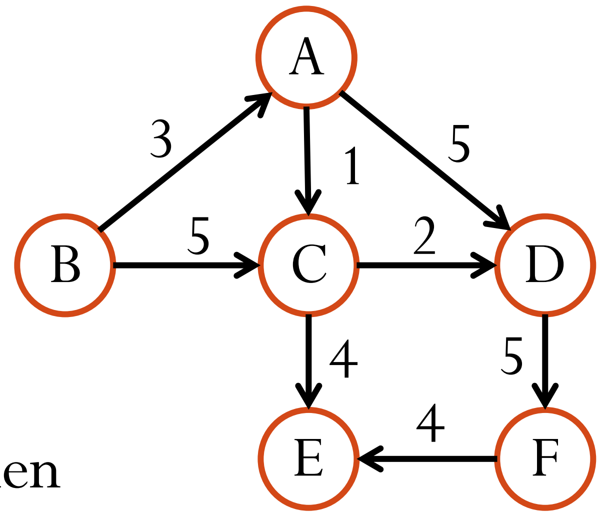
Course Eval

Question 17
Question 18

Thumbs up and subscribe
点赞关注收藏
一件三连

Shortest Path for Weighted Graphs

- The problem becomes more difficult when edges have different weights.
  - Breadth-first search won't work!
  - What is the shortest path from B to F?



- If the weights are **non-negative**, then we can apply **Dijkstra's Algorithm** (more details & examples from Ve203)
  - Works only when all weights are non-negative
  - A greedy algorithm for solving single source all destinations shortest path problem

Dijkstra's Algorithm

- Keep **distance estimate** $D(v)$ and **predecessor** $P(v)$ for each node v.
  - Predecessor: the previous node on the shortest path.

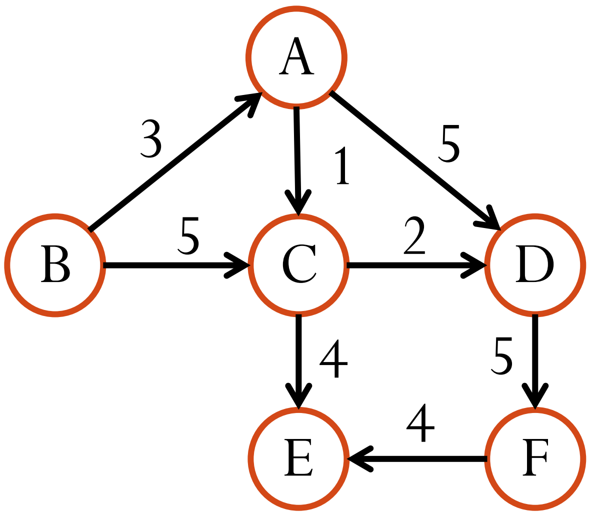
1. Initially, $D(s) = 0$; $D(v)$ for other nodes is $+\infty$; $P(v)$ is unknown.
2. Store all the nodes in a set R.
3. While R is not empty
   1. Choose node v in R such that $D(v)$ is the **smallest**. Remove v from the set R.
   2. Declare that v's shortest distance is known, which is $D(v)$.
   3. For each of v's **neighbors** u that is **still in** R, update distance estimate $D(u)$ and predecessor $P(u)$.

Updating

- For each of v's **neighbors** u that is **still in** R, if $D(v) + w(v,u) < D(u)$, then update $D(u) = D(v) + w(v,u)$ and the predecessor $P(u) = v$.
  - I.e., update $D(u)$ if the path going through v is shorter than the best path so far to u.

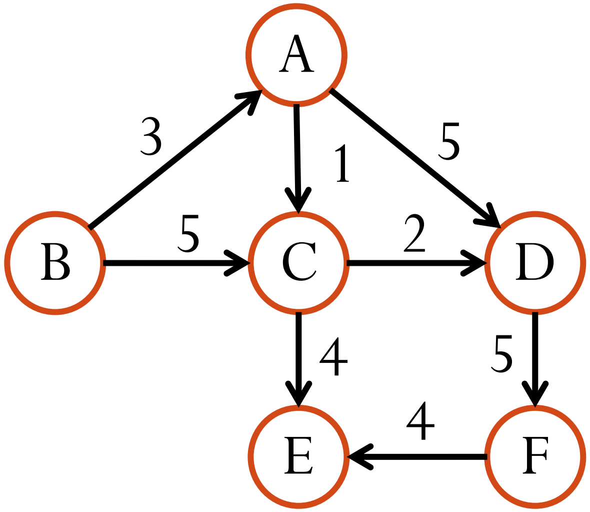
Example A->E

Explored	Unexplored
	A_∞ B_∞ C_∞ D_∞ E_∞ F_∞



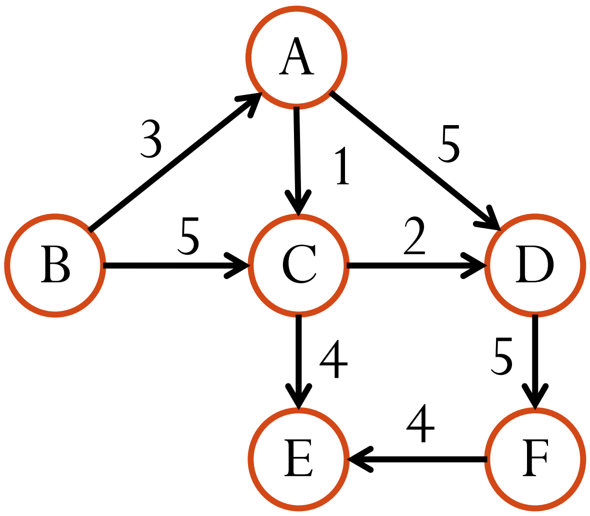
Example

Explored	Unexplored
	A_0^A B_∞ C_∞ D_∞ E_∞ F_∞



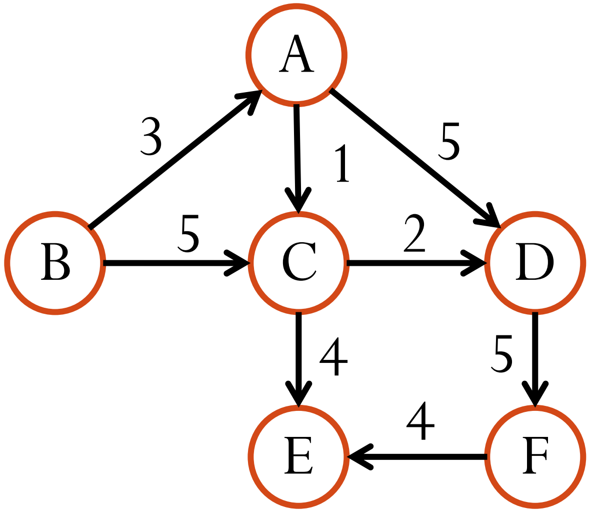
Example

Explored	Unexplored
A_0^A	B_∞ C_∞ D_∞ E_∞ F_∞



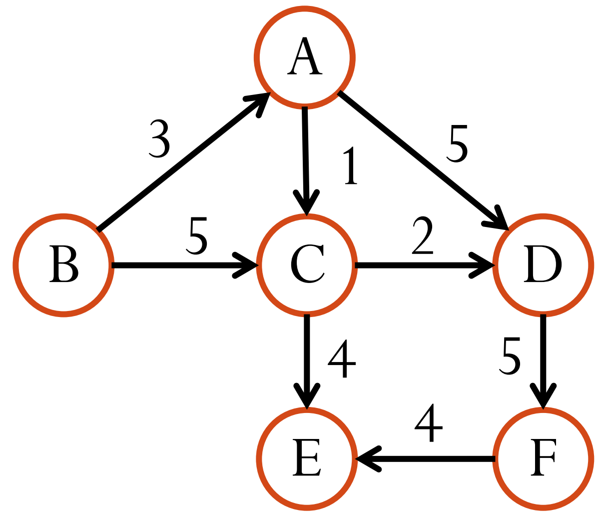
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞



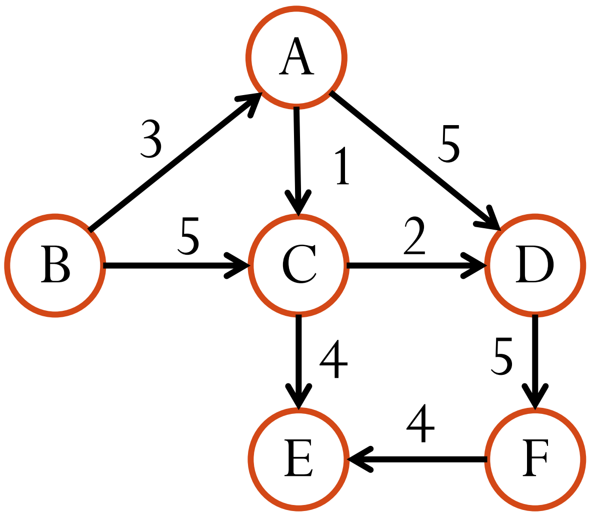
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	



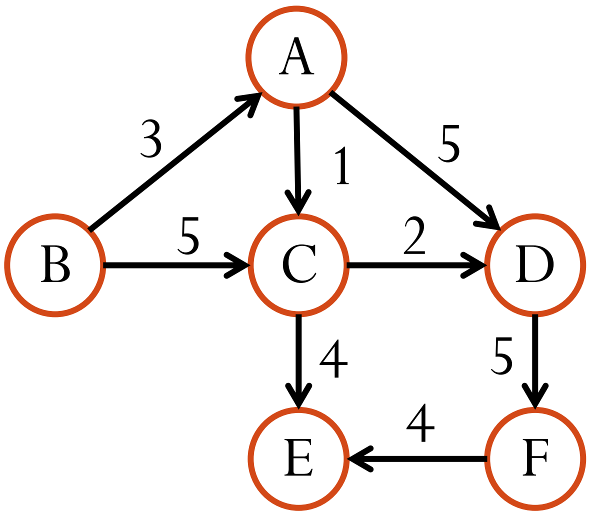
Example

Explored	Unexplored
$A_{\infty}^{0,A}$	B_{∞} C_{1}^{A} D_{5}^{A} E_{∞} F_{∞}
$A_{\infty}^{0,A}$ $C_{\infty}^{A,1}$	B_{∞} D_{3}^{C} E_{C}^{5} F_{∞}



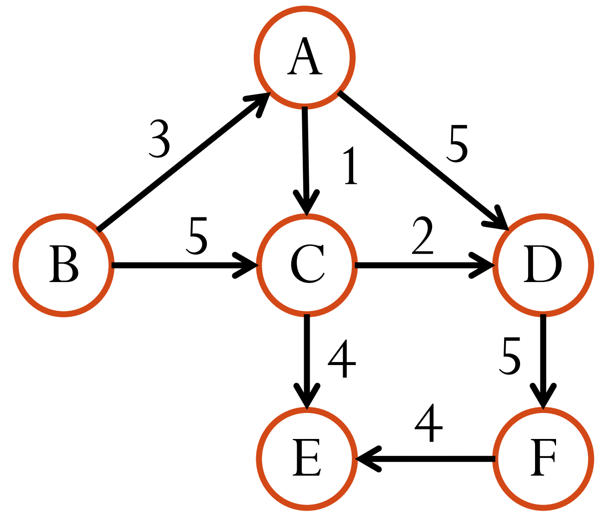
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	B_∞ D_3^C E_C^5 F_∞
A_0^A C_1^A D_3^A	



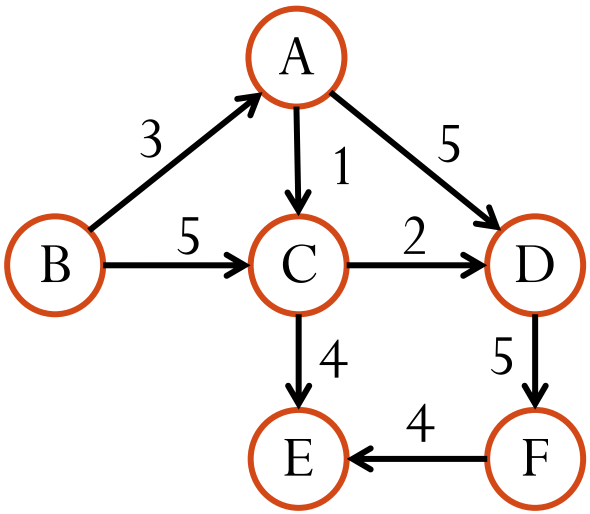
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	B_∞ D_3^C E_C^5 F_∞
A_0^A C_1^A D_3^A	B_∞ E_5^C F_8^D



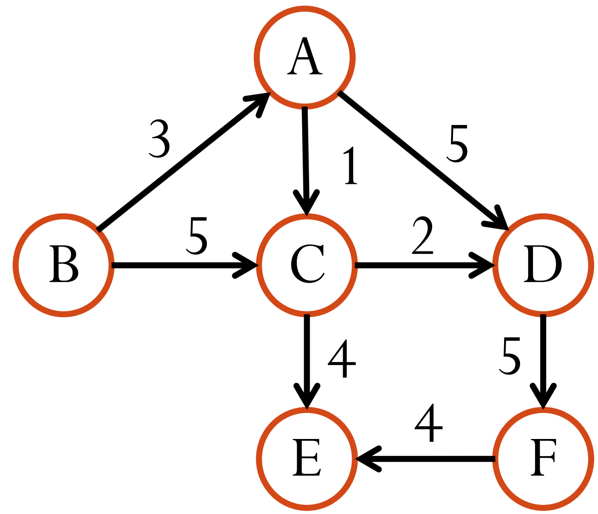
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	B_∞ D_3^C E_C^5 F_∞
A_0^A C_1^A D_3^A	B_∞ E_5^C F_8^D
A_0^A C_1^A D_3^A E_5^C	B_∞ F_8^D



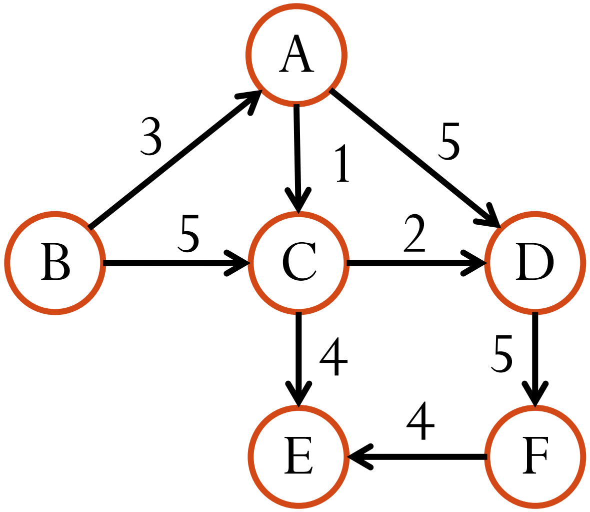
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	B_∞ D_3^C E_C^5 F_∞
A_0^A C_1^A D_3^A	B_∞ E_5^C F_8^D
A_0^A C_1^A D_3^A E_5^C	B_∞ F_8^D
A_0^A C_1^A D_3^A E_5^C F_8^D	B_∞



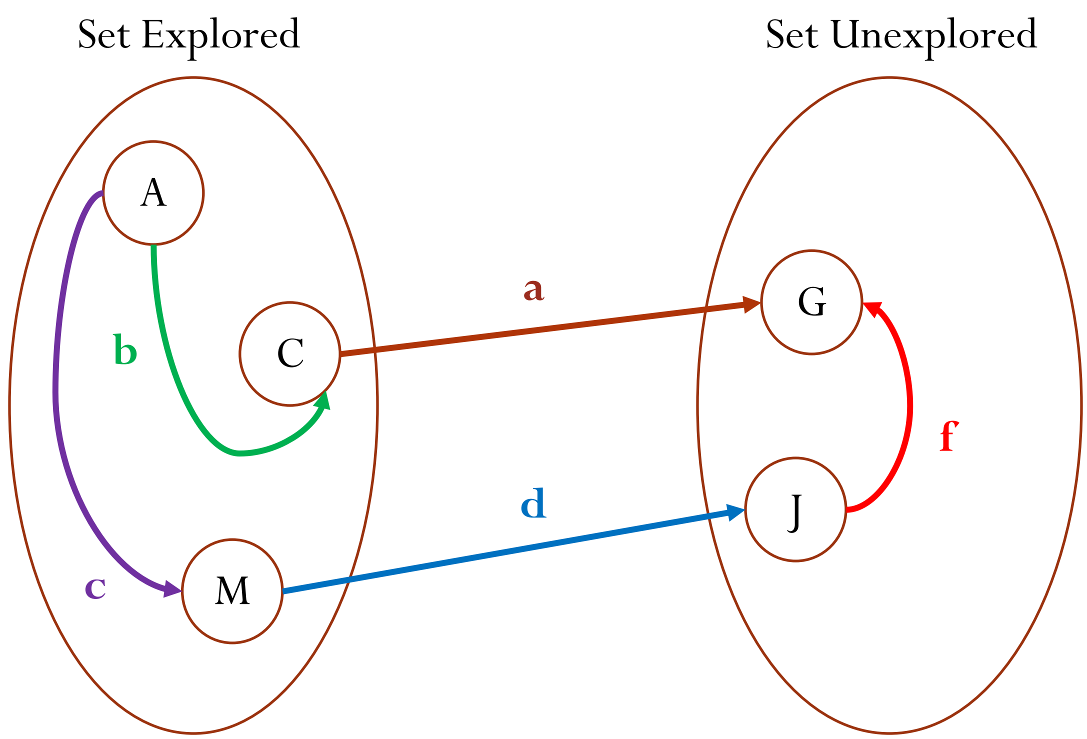
Example

Explored	Unexplored
A_0^A	B_∞ C_1^A D_5^A E_∞ F_∞
A_0^A C_1^A	B_∞ D_3^C E_C^5 F_∞
A_0^A C_1^A D_3^A	B_∞ E_5^C F_8^D
A_0^A C_1^A D_3^A E_5^C	B_∞ F_8^D
A_0^A C_1^A D_3^A E_5^C F_8^D	B_∞
A_0^A C_1^A D_3^A E_5^C F_8^D B_∞	



Why Does it Work?

- Proof by induction
  - All nodes in set Explored already have shortest paths
  - Node in set Unexplored with the smallest distance has the shortest path



b is already the shortest distance to C

b+**a** is already the smallest distance to G through C

$b+a \le c+d$

$b+a \le c+d+f$

Dijkstra's Algorithm v.s. Prim's Algorithm

- Dijkstra's algorithm is similar to Prim's algorithm
  - Prim's algorithm: grow the set of nodes we add to the MST.
  - Dijkstra's algorithm: grow the set of nodes to which we know the shortest path.

Dijkstra's Algorithm

Time Complexity

- Number of times to find the smallest $D(v)$: $|V|$.
  - Each cost? Linear scan: $O(|V|)$; Binary heap: $O(\log|V|)$; Fibonacci heap: $O(\log|V|)$
- Total number of times to update the neighbors: $|E|$.
  - Since each neighbor of each node could be potentially updated.
  - Each cost? Linear scan: $O(1)$; Binary heap: $O(\log|V|)$; Fibonacci heap: $O(1)$
- Total time complexity
  - Linear scan: $O(|E|+|V|^2)=O(|V|^2)$
  - Binary heap: $O(|V|\log|V|+|E|\log|V|)$
  - Fibonacci heap: $O(|V|\log|V|+|E|)$