PSO 11

Kosaraju, Dijkstra, Bellman Ford

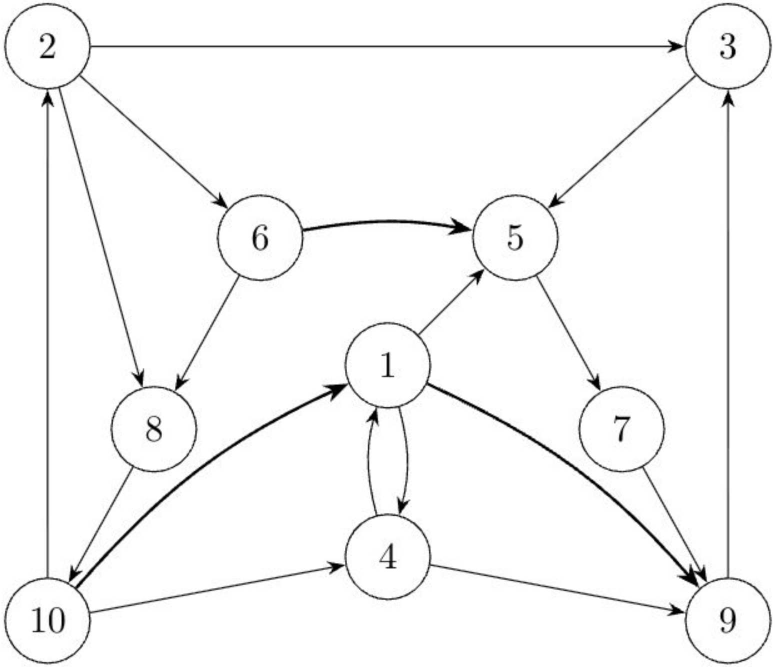
Austin, TX
city hall

Fun fact
People from Austin
HATE City Hall

Question 1

(Kosaraju's algorithm)

(a) Run phase 1 of Kosaraju's algorithm and show the L stack at the end of phase 1. (Note: You should assume that that we loop through nodes in numerical order (ascending) and that each adjacency list are also sorted in ascending order e.g., the adjacency list for node 1 is $(4, 5, 9)$)



kosaraju's alg
finding
SCCs.

(b) Run phase 2 of Kosaraju's algorithm and list the strongly connected components (in topological order).

Question 2

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

2. Given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, but all other edge weights are non-negative, and there are no negative-weighted cycles. Can the Dijkstra's algorithm correctly find all the shortest paths from s in this graph?

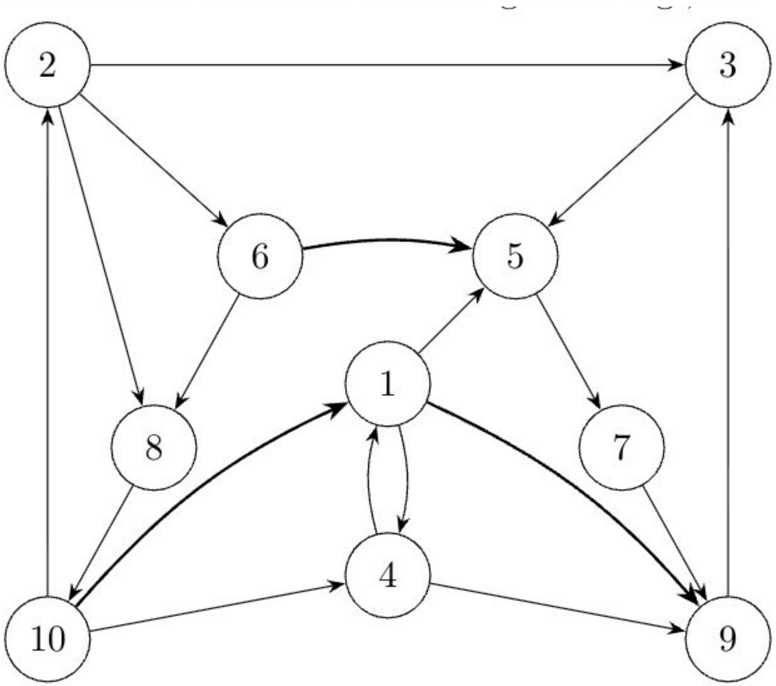
3. Your classmate claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path. Show that him/her is mistaken by constructing a directed graph for which the Dijkstra's algorithm could relax the edges of a shortest path out of order.

Hint: The shortest path between two vertices in the graph is not necessarily unique.

Question 3

(Bellman-Ford algorithm)

1. Why does the Bellman-Ford algorithm only require $|V| - 1$ passes?
2. Why will the last pass $(|V| - 1)$ through the edges will determine if there are any negative weight cycles or not?



Kosaraju's Algorithm

outputs: SCC

How does it work:

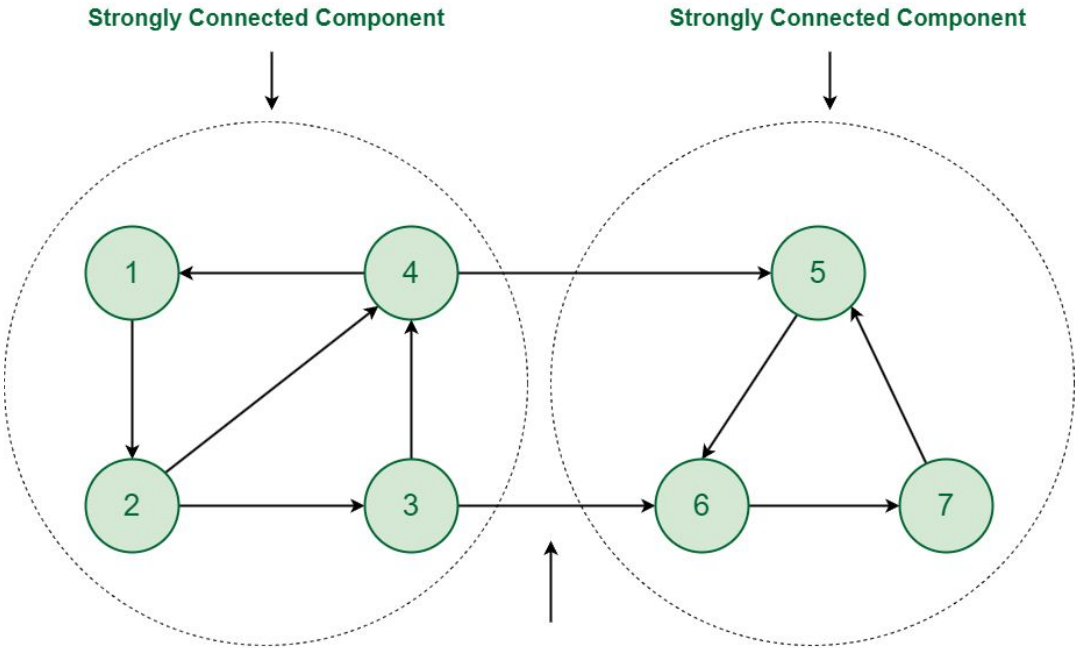
one-way

Phase 1: find 'weak' connectivity (dfs)

Phase 2: filter out any weak parts.

(dfs on reverse graph)

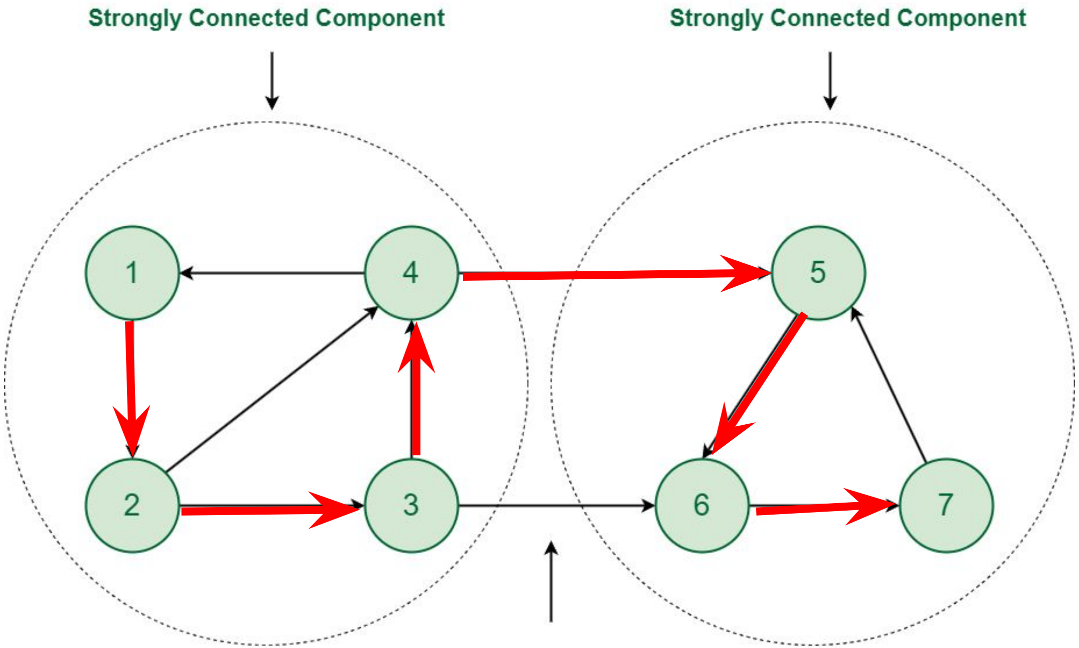
Intuition of Kosaraju's (Phase 1)



Phase 1: Find all *unidirectional* connected components with DFS

Intuition of Kosaraju's (Phase 1)

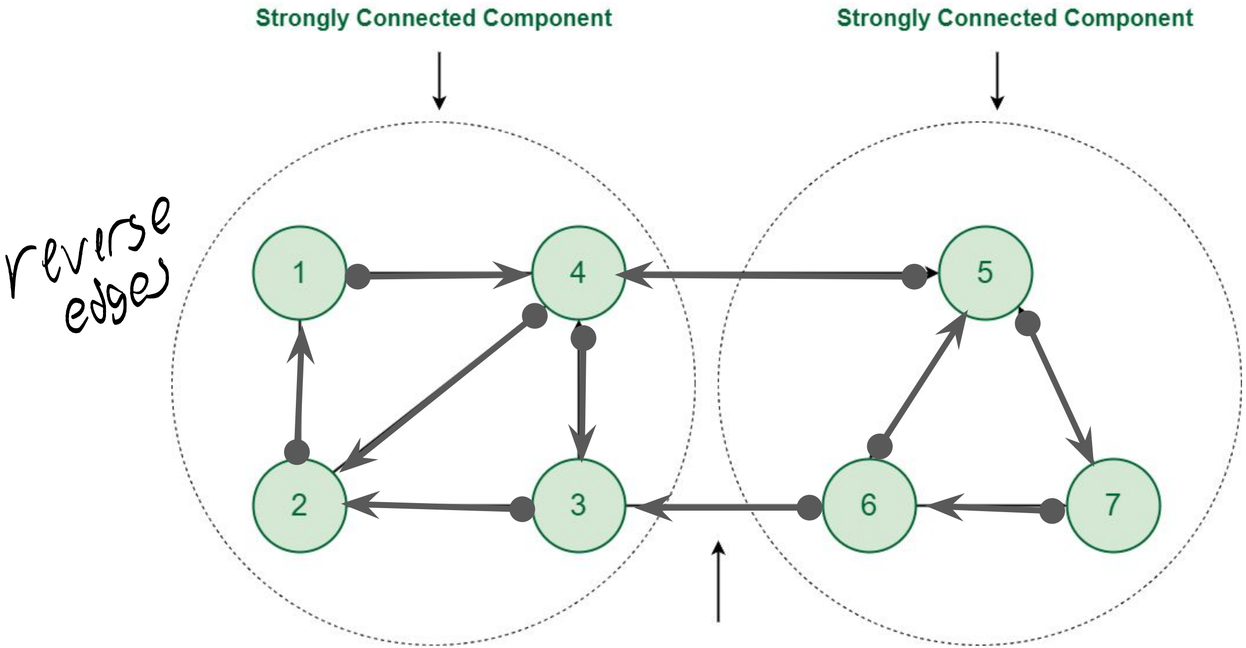
dfs(1)



Phase 1: Find all *unidirectional* connected components with DFS

Intuition of Kosaraju's (Phase 2)

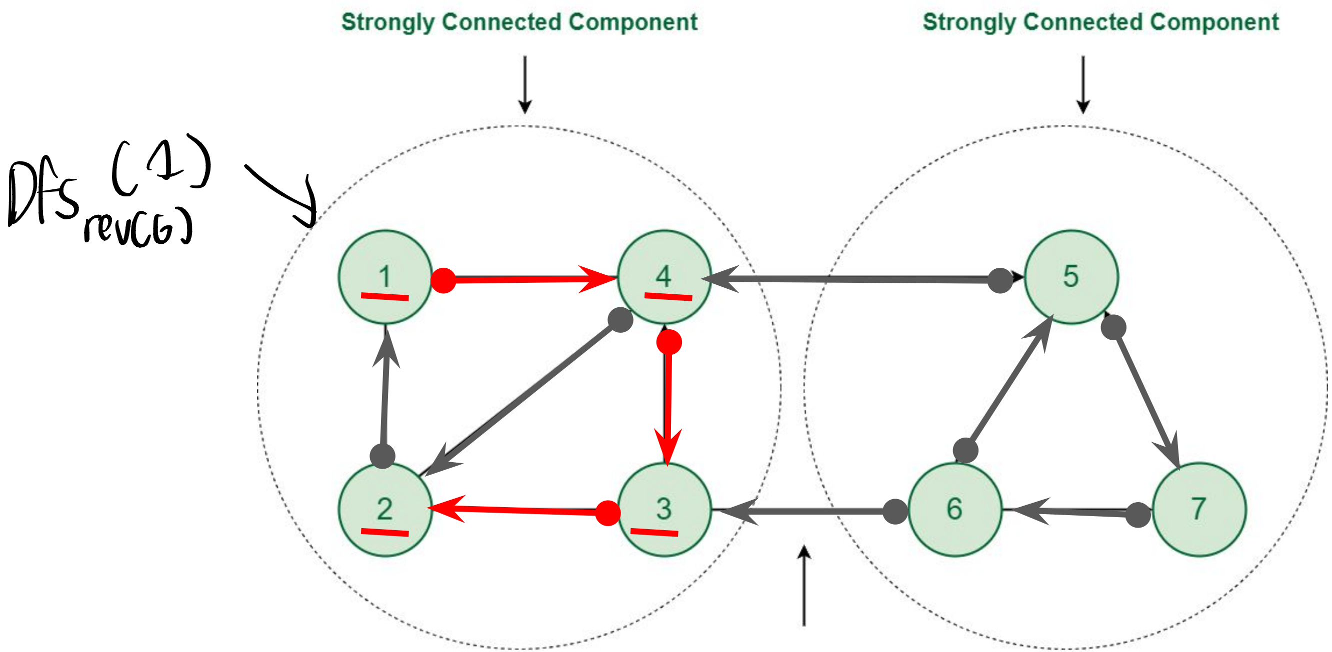
Dfs trace: 1,2,3,4,5,6,7



Phase 2: Find SCCs within the unidirectional components by reverse DFS

Intuition of Kosaraju's (Phase 2)

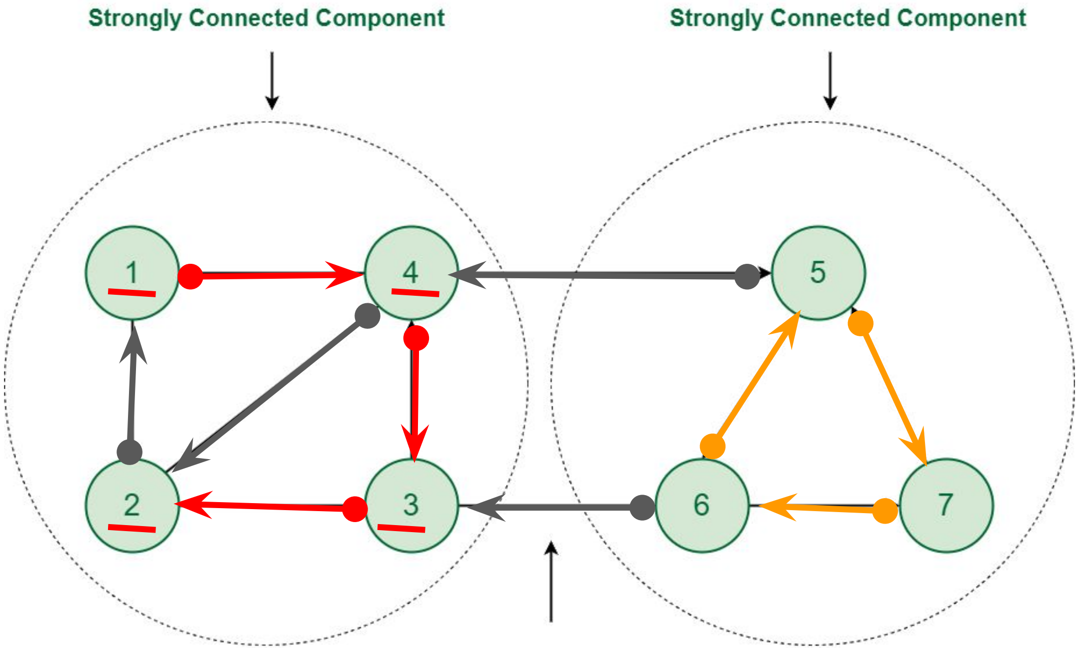
Dfs trace: 1,2,3,4,5,6,7



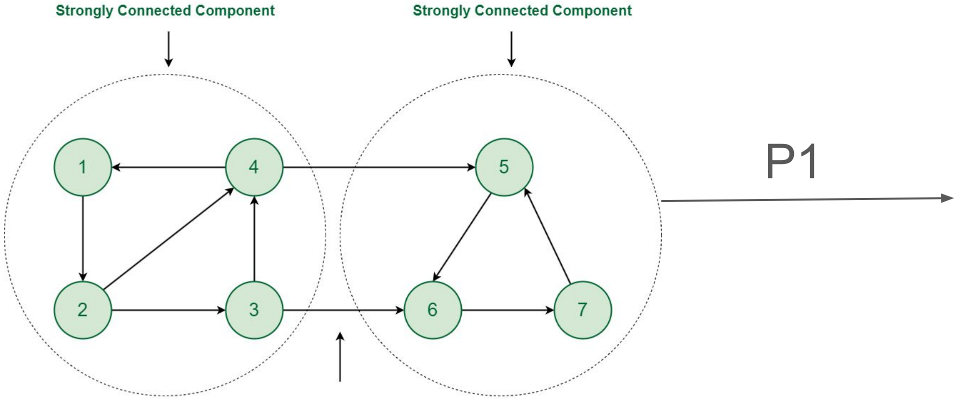
Phase 2: Find SCCs within the unidirectional components by reverse DFS

Intuition of Kosaraju's (Phase 2)

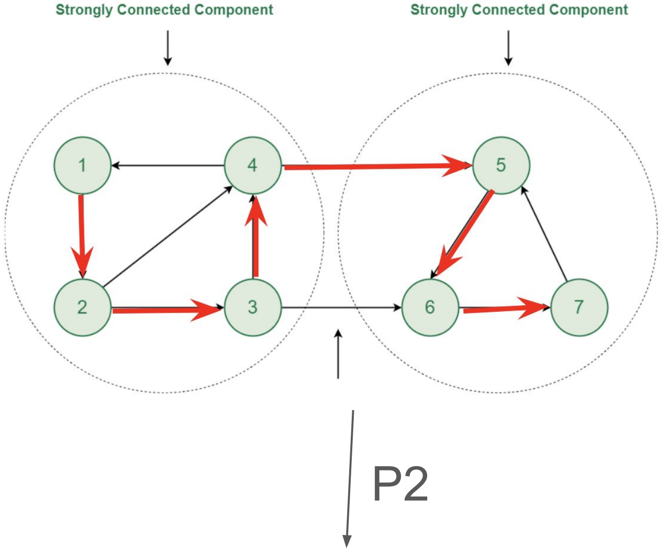
Dfs trace: 1,2,3,4,5,6,7

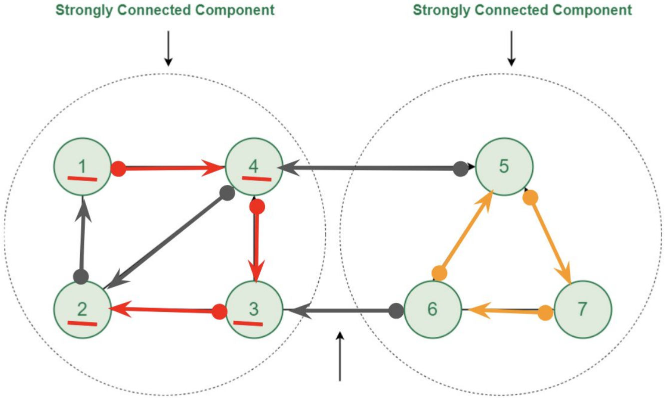


Phase 2: Find SCCs within the unidirectional components by reverse DFS

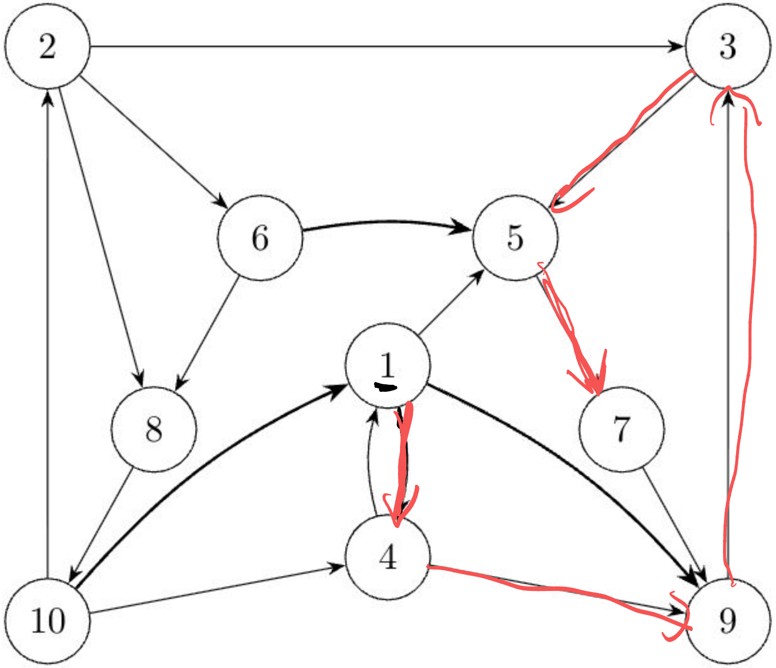


reverse edges
||
filter out any
unidirectional paths.





Phase 1: finding all connections



From v = 1,...,10:

Run DFS(v)

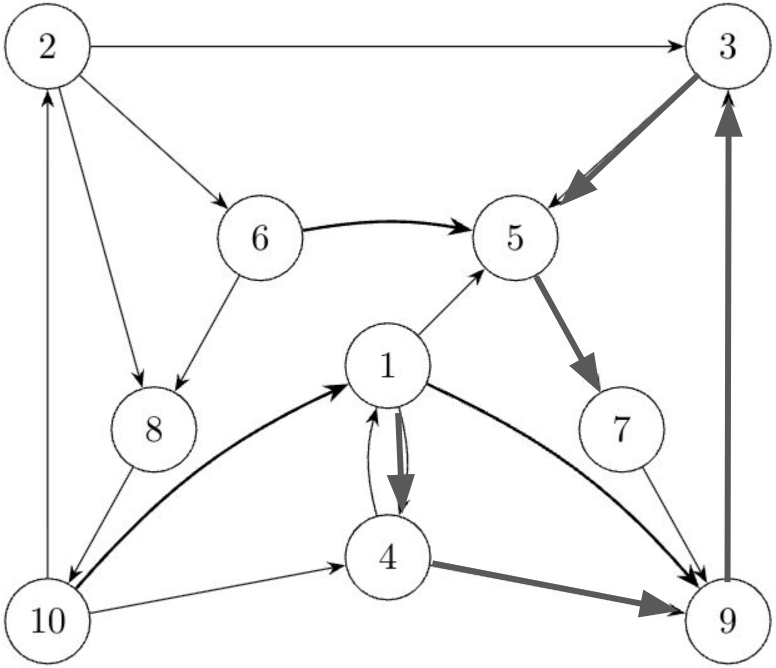
Transfer seen to L stack

//break once all nodes marked

7
5
3
9
4
1

Seen

Phase 1: finding all connections

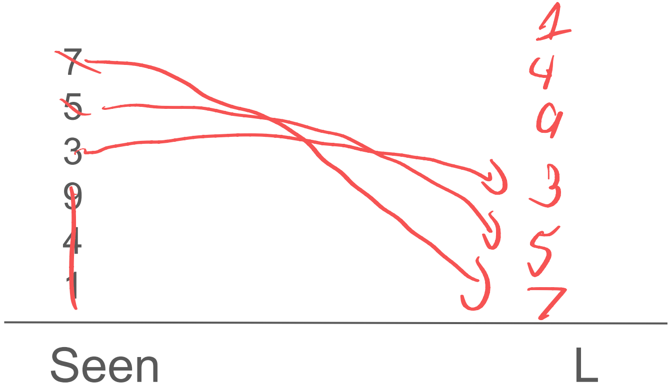


From v = 1,...,10:

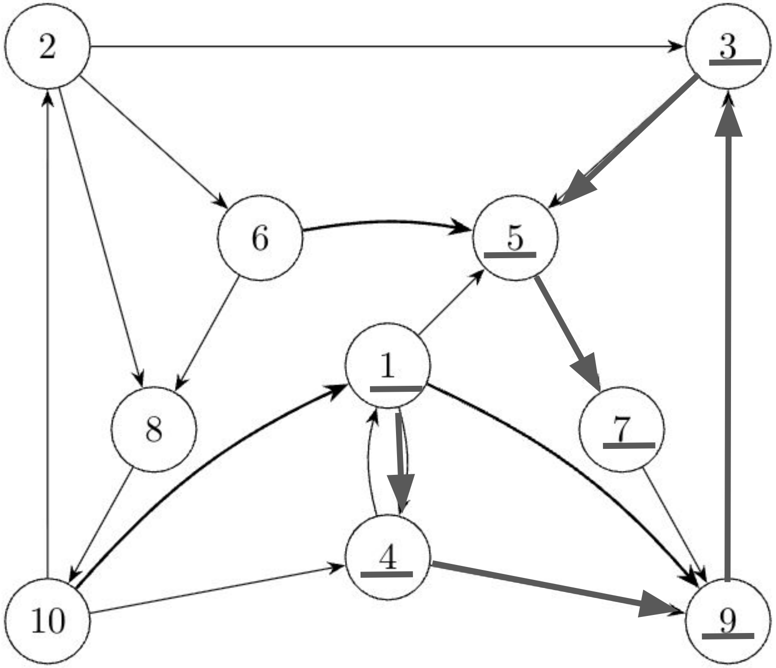
Run DFS(v)

Transfer seen stack to L stack

//break once all nodes marked



Phase 1: finding all connections



From v = 1,...,10:

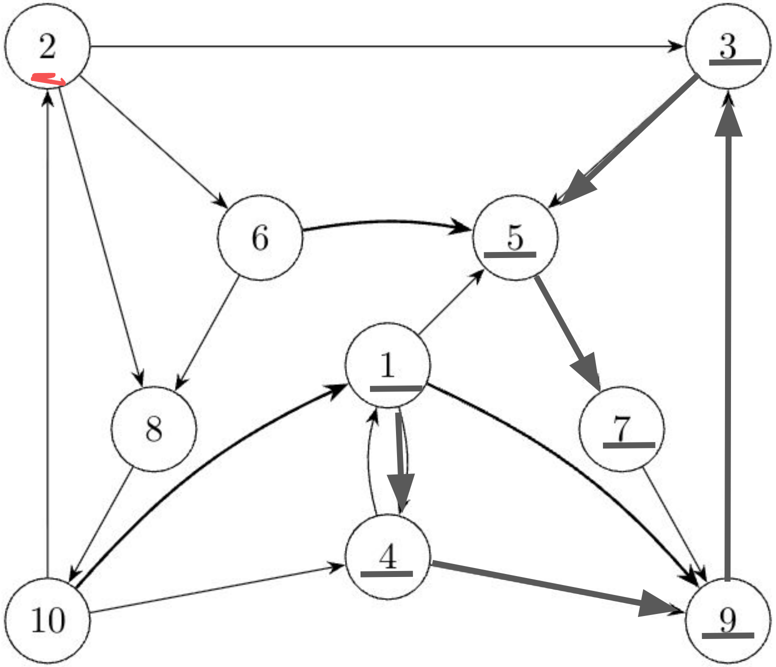
Run DFS(v)

Transfer seen stack to L stack

//break once all nodes marked

Seen	L
	1
	4
	9
	3
	5
	7

Phase 1: finding all connections



From v = 1,...,10:

Run DFS(v)

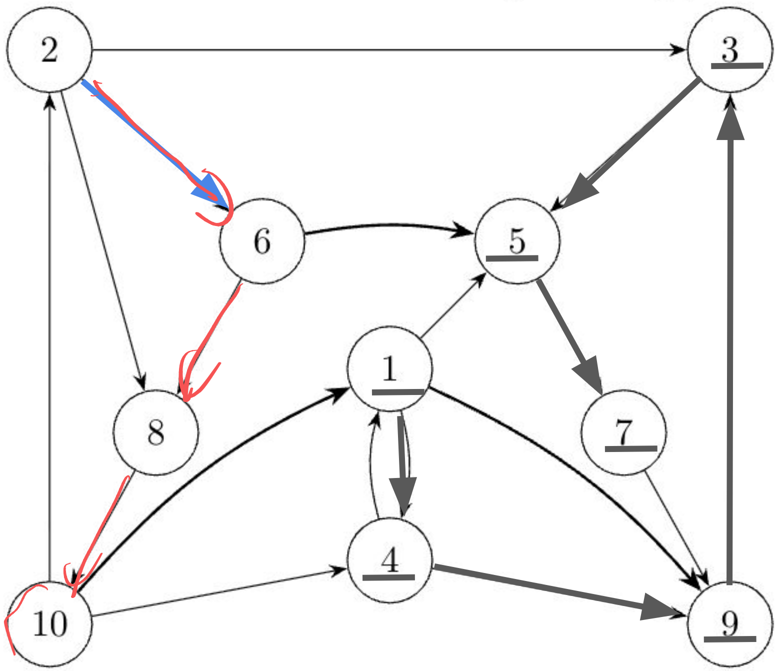
Transfer seen stack to L stack

//break once all nodes marked

Skip to next unseen (2)

Seen	L
	1
	4
	9
	3
	5
	7

Phase 1: finding all connections



From v = 1,...,10:

Run DFS(v)

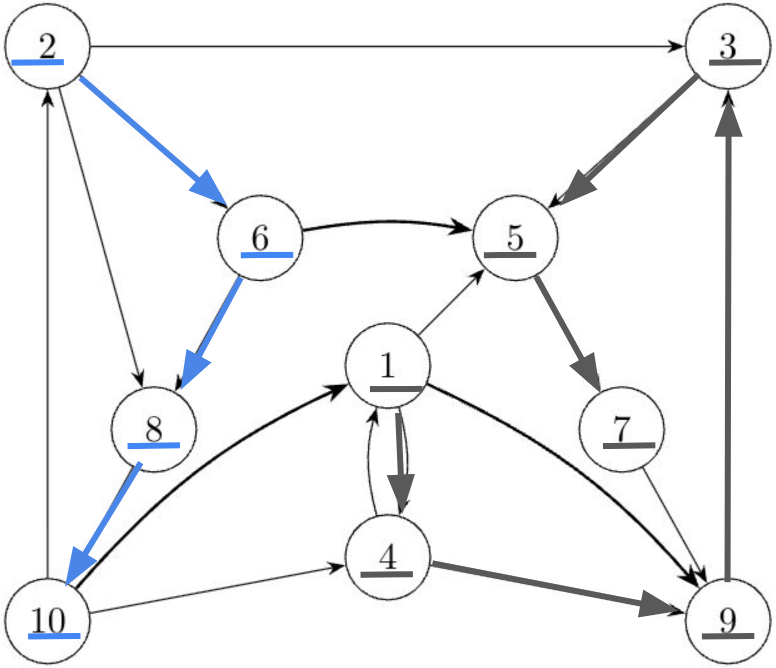
Transfer seen stack to L stack

//break once all nodes marked

Skip to next unseen (2)

Seen	L
10	1
8	4
6	9
2	3
	5
	7

Phase 1: finding all connections



From v = 1,...,10:

Run DFS(v)

Transfer seen stack to L stack

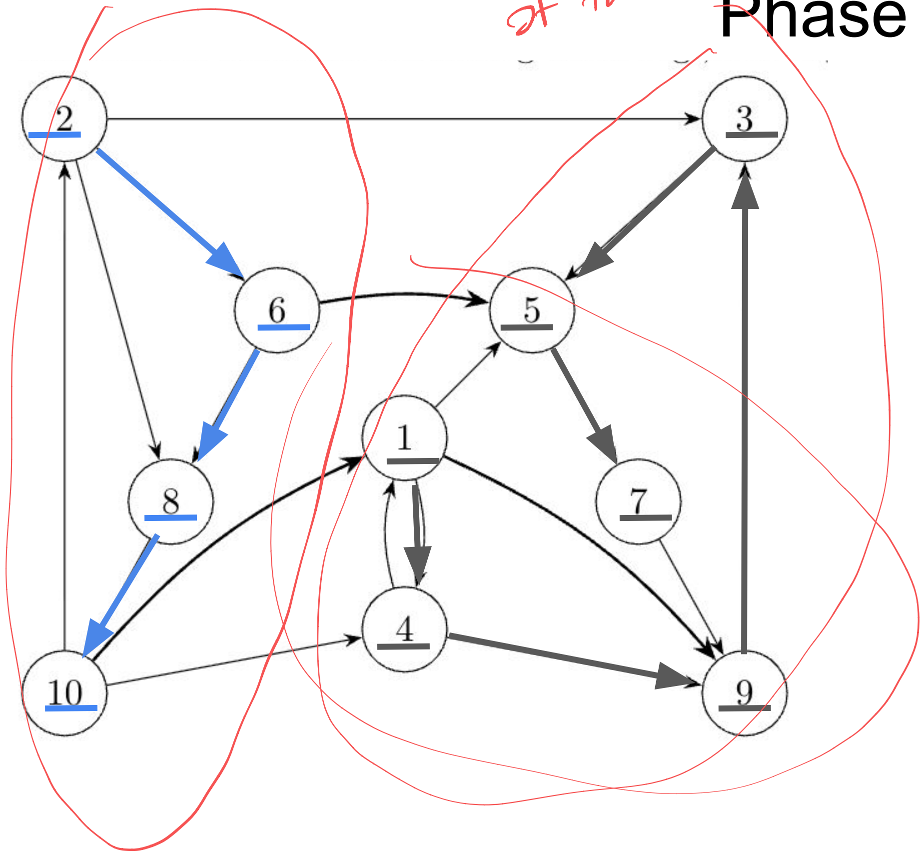
//break once all nodes marked

Skip to next unseen (2)

Seen	L
	2
	6
	8
	10
	1
	4
10	9
8	3
6	5
2	7

. SCCs are subsets of the dfs components from p1.

Phase 1: finding all connections



From v = 1,...,10:

Run DFS(v)

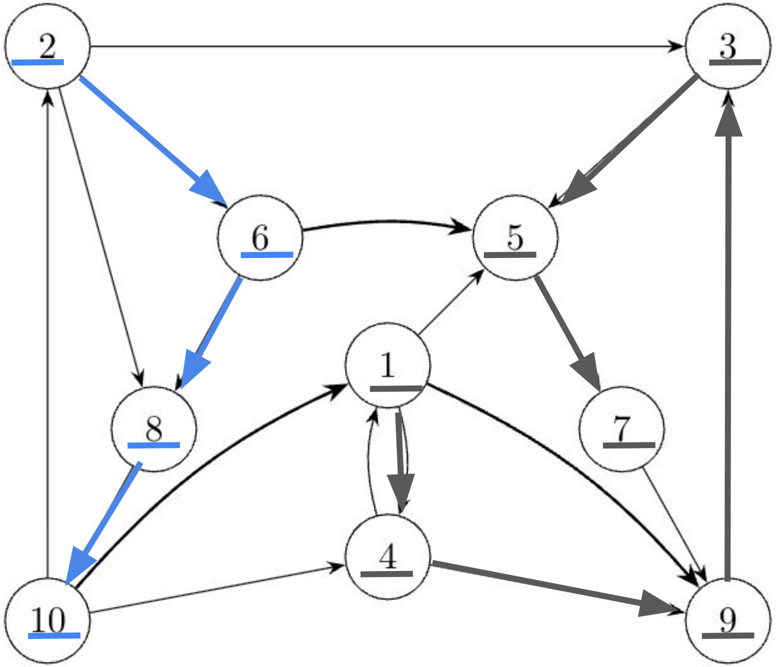
Transfer seen stack to L stack

//break once all nodes marked

2
6
8
10
1
4
9
3
5
7

Seen L

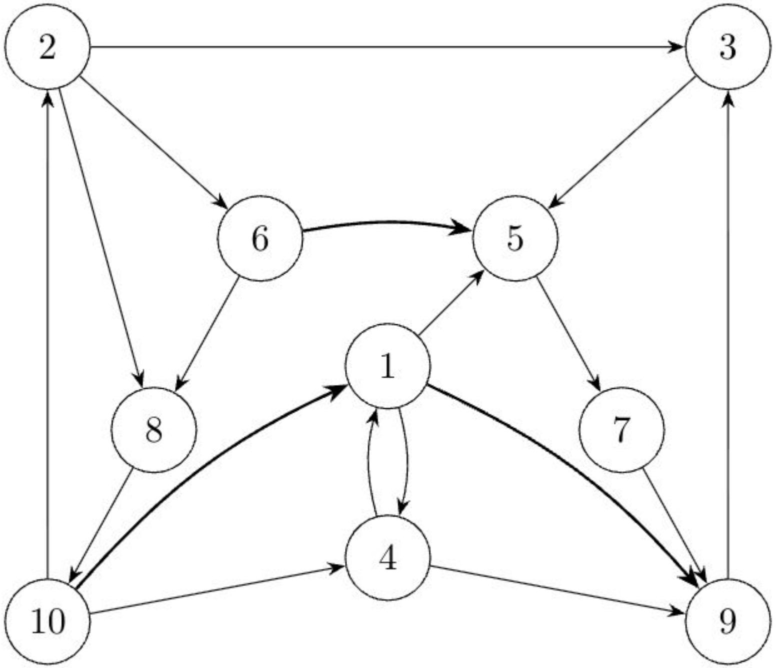
Phase 1: finding all connections



Insight from before:
SCCs are inside these DFS traversals

Seen	L
	2
	6
	8
	10
	1
	4
	9
	3
	5
	7

Phase 2: Reverse graph to filter out non-SCCs

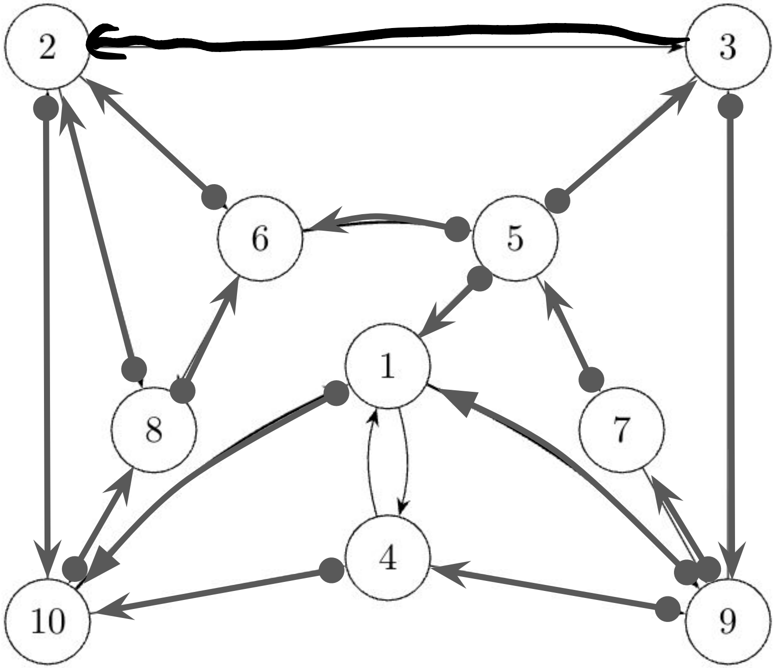


1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

2
6
8
10
1
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs



PRO TIP: Bring a sharpie to the midterm

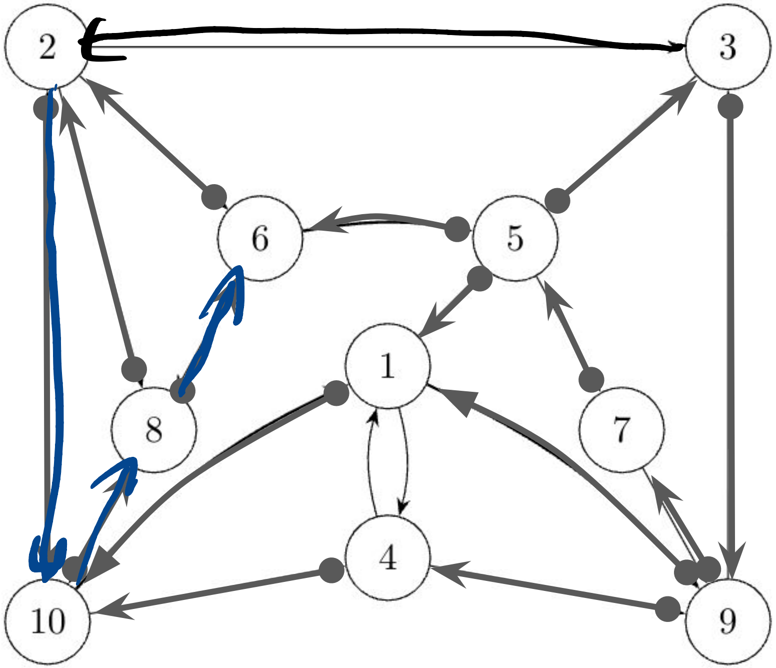
1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

2
6
8
10
1
4
9
3
5
7
L

ok...

too wide

Phase 2: Reverse graph to filter out non-SCCs



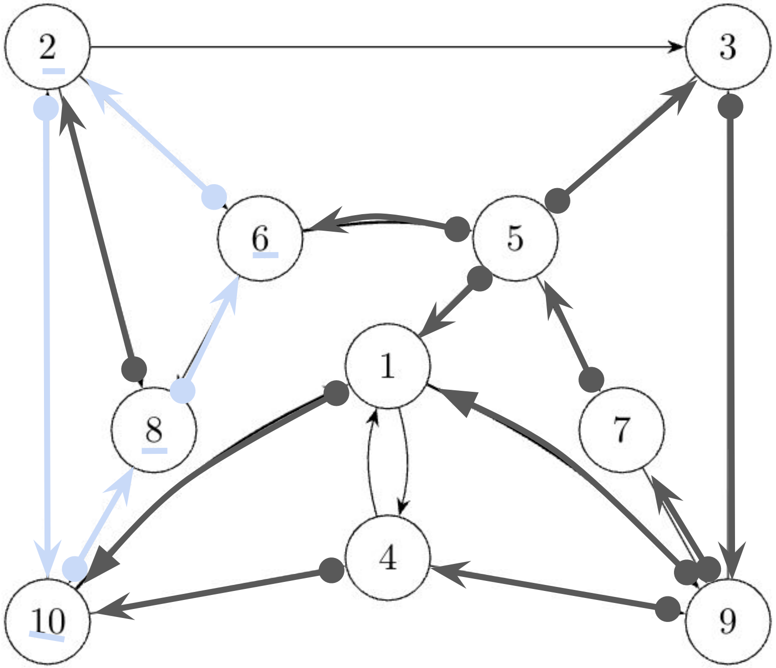
1. Reverse edges
2. While unmarked vertices:
    a. i = L.pop()
    b. //if i is marked, continue to next loop iter.
    c. dfs(i) //and mark as SCC j, j++

2
6
8
10
1
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs

light blue = SCC #2

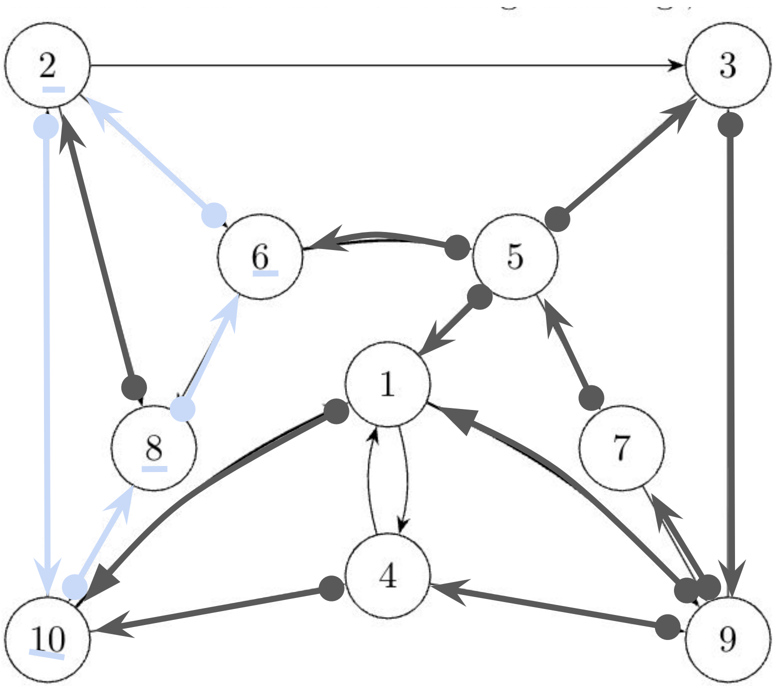


1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

6
8
10
1
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs

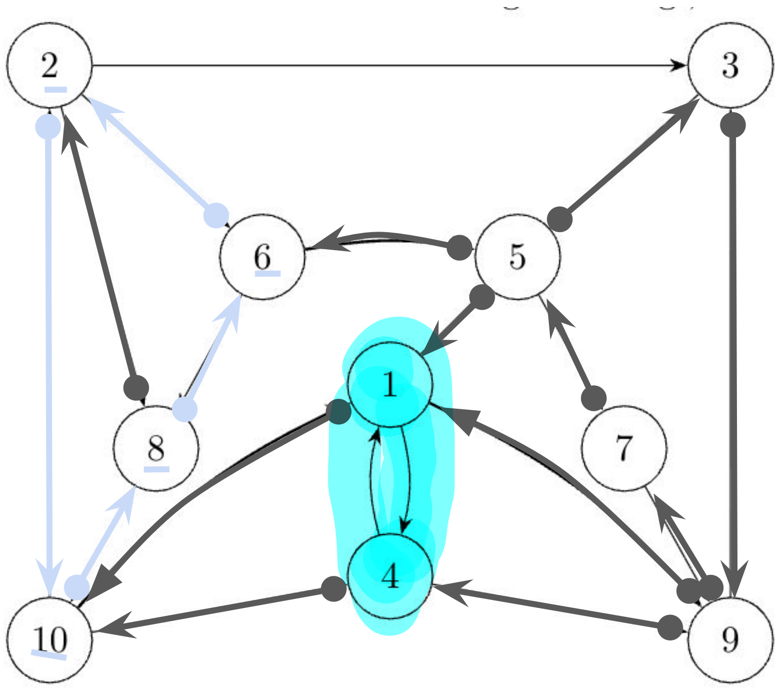


1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

~~6~~
~~8~~
~~10~~
1
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs

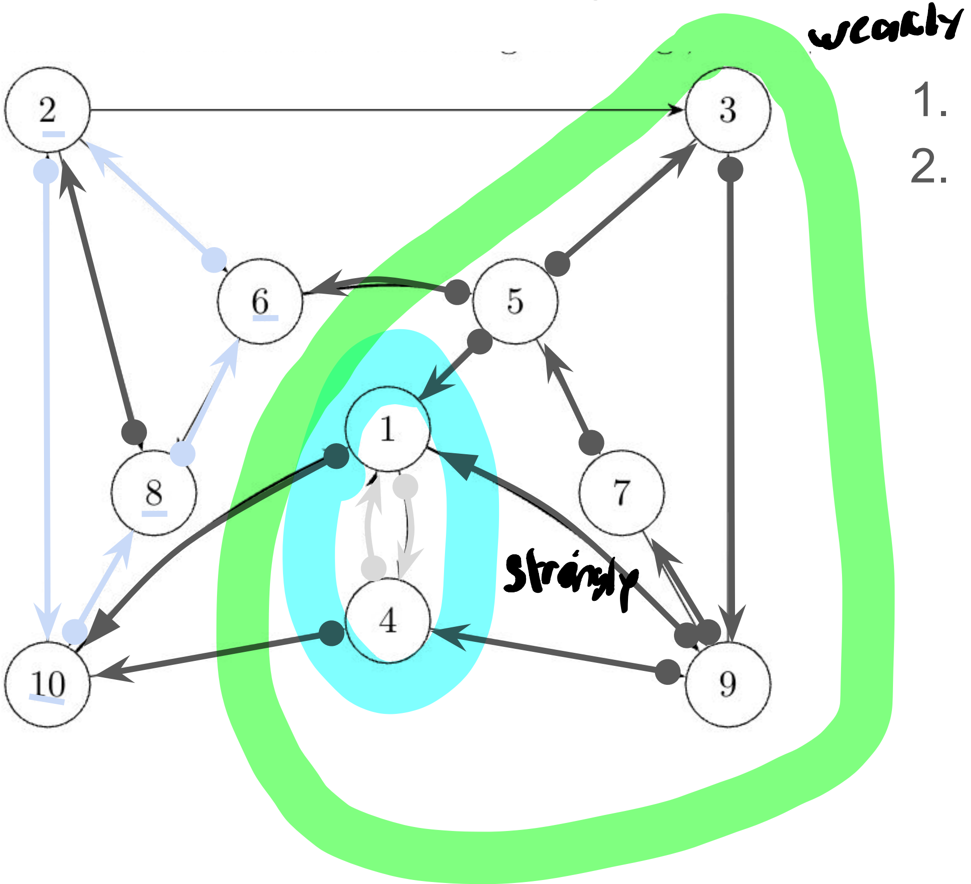


1. Reverse edges
2. While unmarked vertices:
    a. i = L.pop()
    b. //if i is marked, continue to next loop iter.
    c. dfs(i) //and mark as SCC j, j++

1
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs

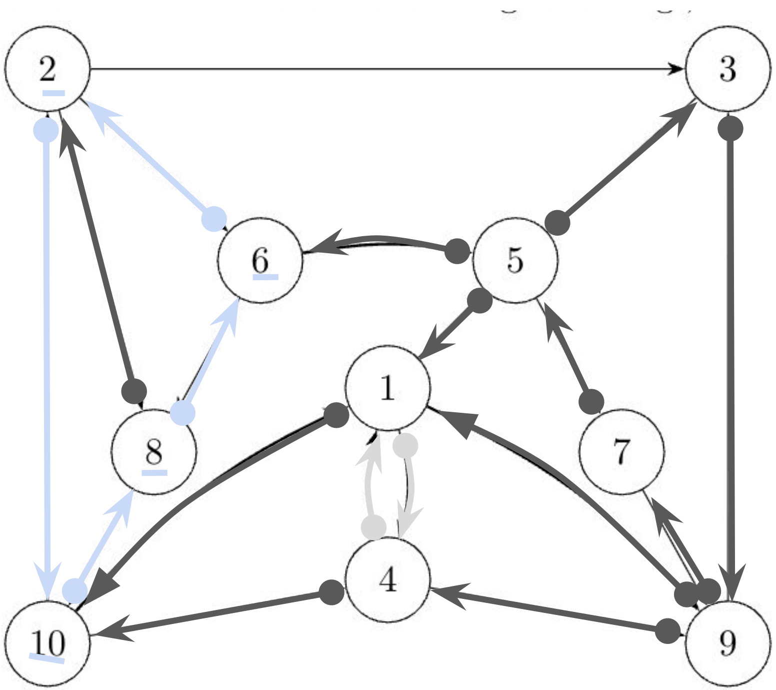


1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

~~1~~
4
9
3
5
7

L

Phase 2: Reverse graph to filter out non-SCCs

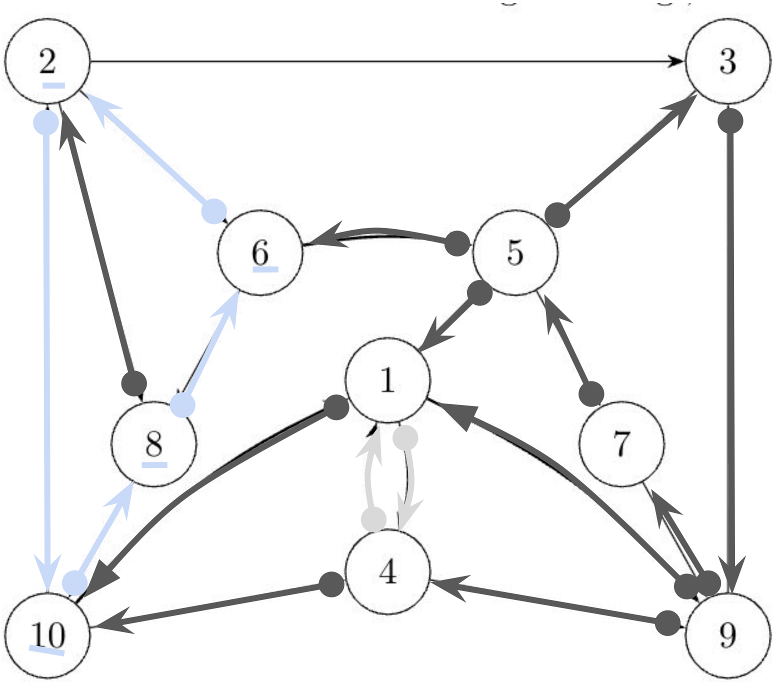


1. Reverse edges
2. While unmarked vertices:
    a. i = L.pop()
    b. //if i is marked, continue to next loop iter.
    c. dfs(i) //and mark as SCC j, j++

~~1~~
~~4~~
9
3
5
7

L

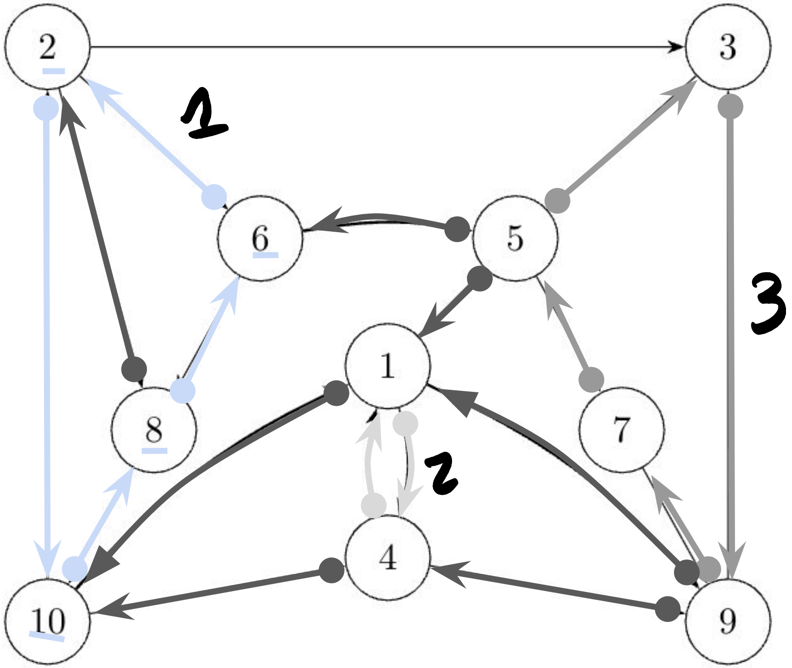
Phase 2: Reverse graph to filter out non-SCCs



1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

9
3
5
7
L

Phase 2: Reverse graph to filter out non-SCCs

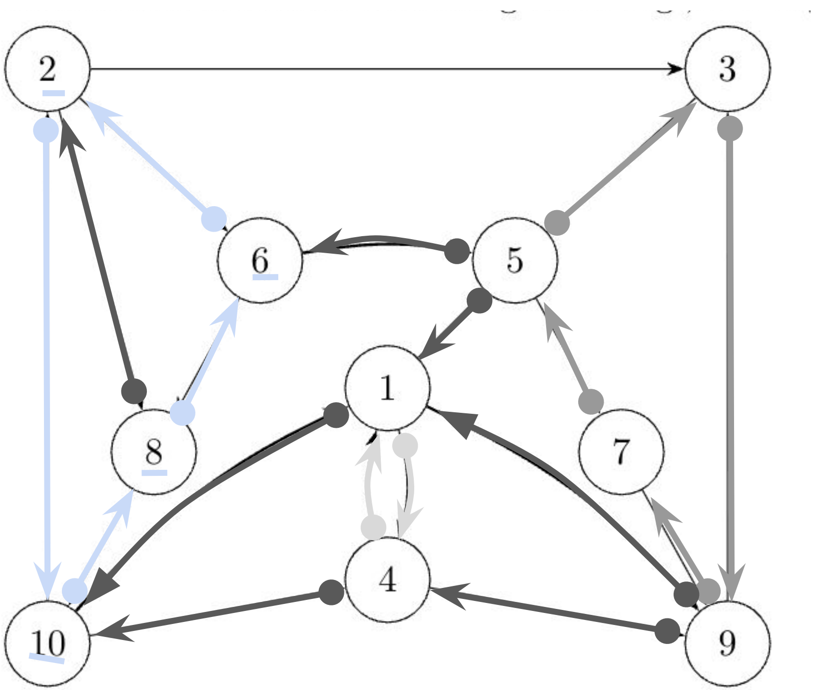


1. Reverse edges
2. While unmarked vertices:
   a. i = L.pop()
   b. //if i is marked, continue to next loop iter.
   c. dfs(i) //and mark as SCC j, j++

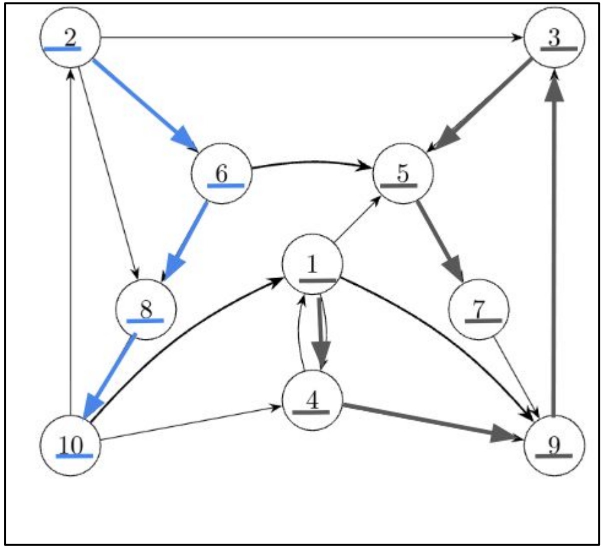
O(n+m) time.

~~3~~
~~5~~
~~7~~
L

Summary: 3 SCCs



Recall the first phase



Partitioned the second (black) dfs traversal into 2 SCCs

Question 2

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

2. Given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, but all other edge weights are non-negative, and there are no negative-weighted cycles. Can the Dijkstra's algorithm correctly find all the shortest paths from s in this graph?

3. Your classmate claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path. Show that him/her is mistaken by constructing a directed graph for which the Dijkstra's algorithm could relax the edges of a shortest path out of order.

Hint: The shortest path between two vertices in the graph is not necessarily unique.

Dijkstra - Find shortest paths between (input starting vertex s) and all other vertices.

- **Single source shortest paths**

$(s) \rightarrow v \in V$

Dijkstra

```
algorithm DijkstraShortestPath(G(V,E), s ∈ V)

   let dist:V → ℤ
   let prev:V → V
   let Q be an empty priority queue

   dist[s] ← 0
   for each v ∈ V do
      if v ≠ s then
         dist[v] ← ∞
      end if
      prev[v] ← -1
      Q.add(dist[v], v)
   end for

   while Q is not empty do
      u ← Q.getMin()
      for each w ∈ V adjacent to u still in Q do
         d ← dist[u] + weight(u, w)
         if d < dist[w] then
            dist[w] ← d
            prev[w] ← u
            Q.set(d, w)
         end if
      end for
   end while

   return dist, prev
end algorithm
```

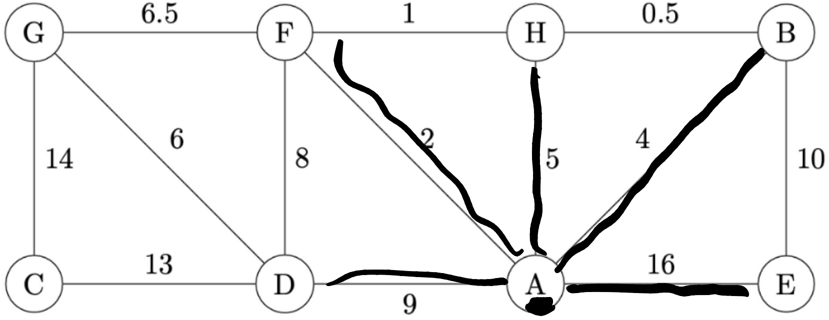
For a vertex s, finds shortest paths to all vertices. At each step..

- Consider current closest vertex u (priority queue)
- Greedily update path lengths to u's neighbors
  - **"Relaxing the edge"**
- Mark as visited

obs: once V is no longer in PQ its distance is fixed.

For your studies, a walkthrough of how Dijstra works..

Start at A

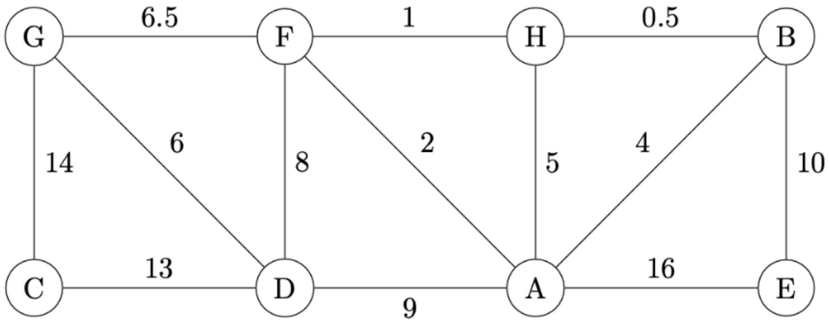


	A	B	C	D	E	F	G	H
dist	0	~~∞~~ 4	∞	~~∞~~ 9	~~∞~~ 16	~~∞~~ 2	∞	~~∞~~ 5
prev	~~-1~~ A	~~-1~~ A	-1	~~-1~~ A	~~-1~~ A	~~-1~~ A	-1	~~-1~~ A

Q
(0, A)
(~~∞~~ 4, B)
(∞, C)
(~~∞~~ 9, D)
16 (~~∞~~, E)
2 (~~∞~~, F)
(∞, G)

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

Update based on A’s edges



	A	B	C	D	E	F	G	H
dist	0	4	∞	9	16	2	∞	5
prev	-1	A	-1	A	A	A	-1	A

Q

(2, F)

(4, B)

(9, D)

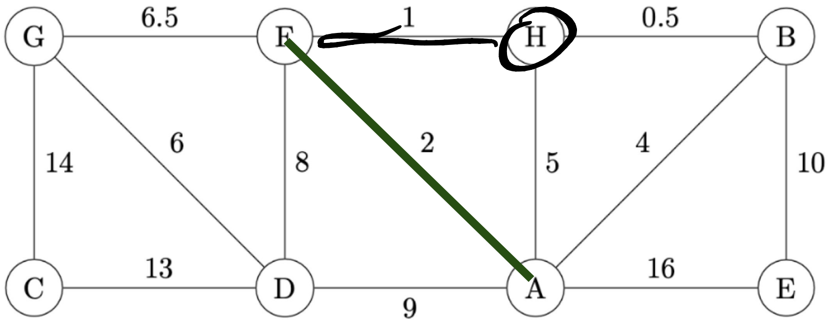
(16, E)

(∞, C)

(∞, G)

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

Next on the prior. Q is F



	A	B	C	D	E	F	G	H
dist	0	4	∞	9	16	2	∞	~~5~~ 3
prev	-1	A	-1	A	A	A	-1	~~A~~ F

Q
(2, F)
(4, B)
(9, D)
(16, E)
(∞, C)
(∞, G)

$$\underbrace{\overset{2}{dist[F]} + \overset{1}{e(F,H)}}_{=\,3}$$

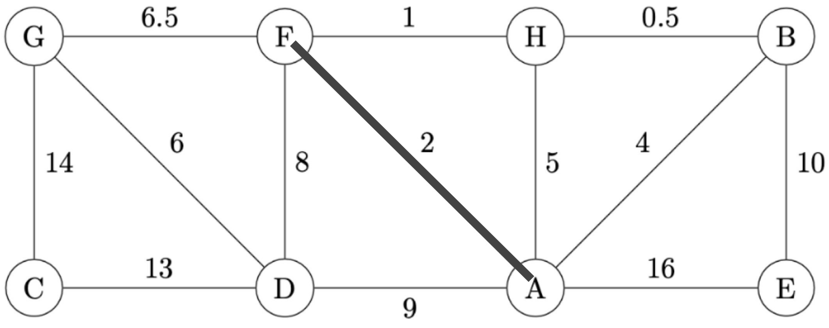
$$dist[H] \overset{?}{>} dist[F] + e(F,H)$$

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

Why didn't we update D?

dist[F] + e(F,D) = 10

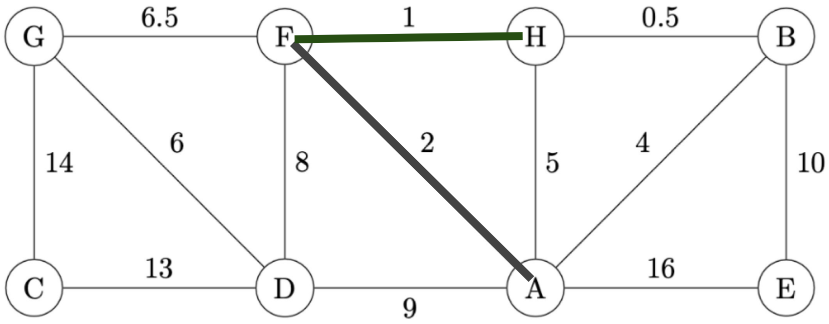
2 & 8



	A	B	C	D	E	F	G	H
dist	0	4	∞	9	16	2	8.5	3
prev	-1	A	-1	A	A	A	F	F

Q
(3, H)
(4, B)
(9, D)
(16, E)
(∞, C)
(8.5, G)

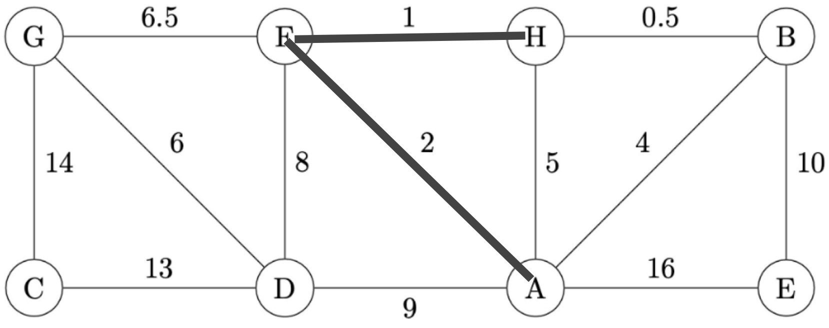
```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```



	A	B	C	D	E	F	G	H
dist	0	4	∞	9	16	2	8.5	3
prev	-1	A	-1	A	A	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

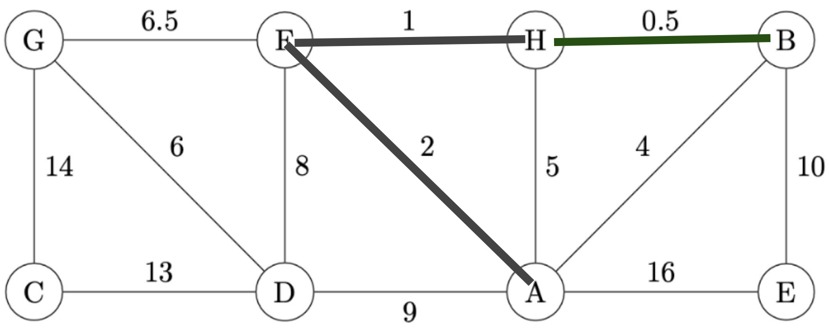
Q
(3, H)
(4, B)
(9, D)
(16, E)
(∞, C)
(8.5, G)



	A	B	C	D	E	F	G	H
dist	0	3.5	∞	9	16	2	8.5	3
prev	-1	H	-1	A	A	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

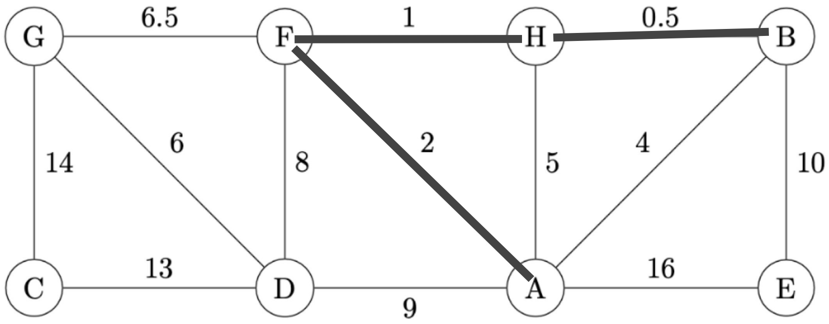
Q
(3.5, B)
(8.5, G)
(9, D)
(16, E)
(∞, C)



	A	B	C	D	E	F	G	H
dist	0	3.5	∞	9	16	2	8.5	3
prev	-1	H	-1	A	A	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

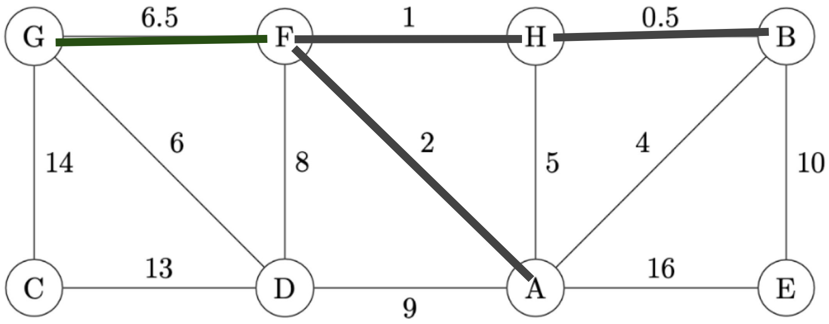
Q
(3.5, B)
(8.5, G)
(9, D)
(16, E)
(∞, C)



	A	B	C	D	E	F	G	H
dist	0	3.5	∞	9	13.5	2	8.5	3
Prev	-1	H	-1	A	B	A	F	F

Q
(8.5, G)
(9, D)
(13.5, E)
(∞, C)

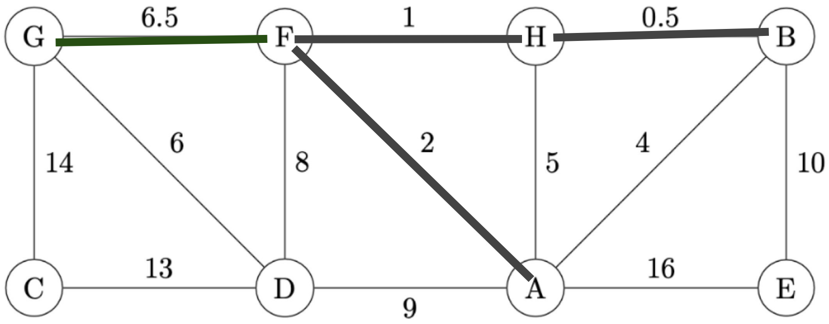
```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```



	A	B	C	D	E	F	G	H
dist	0	3.5	∞	9	13.5	2	8.5	3
Prev	-1	H	-1	A	B	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

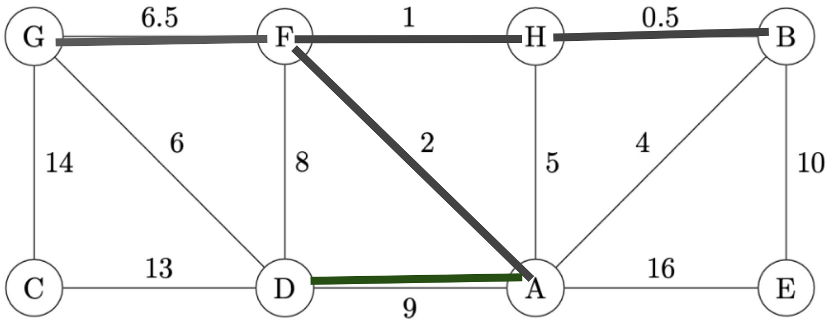
Q
(8.5, G)
(9, D)
(13.5, E)
(∞, C)



	A	B	C	D	E	F	G	H
dist	0	3.5	22.5	9	13.5	2	8.5	3
Prev	-1	H	G	A	B	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

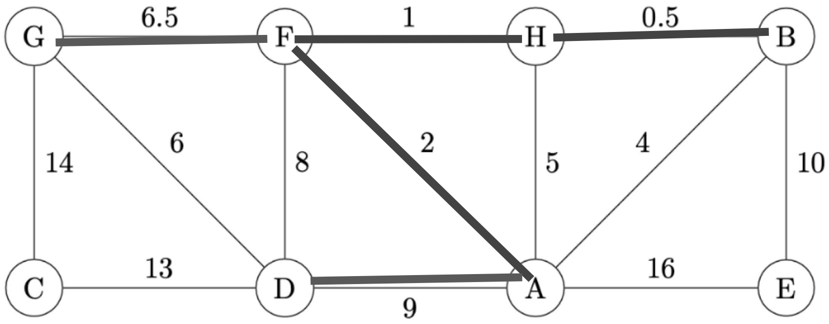
Q
(9, D)
(13.5, E)
(22.5, C)



	A	B	C	D	E	F	G	H
dist	0	3.5	22.5	9	13.5	2	8.5	3
Prev	-1	H	G	A	B	A	F	F

Q
(9, D)
(13.5, E)
(22.5, C)

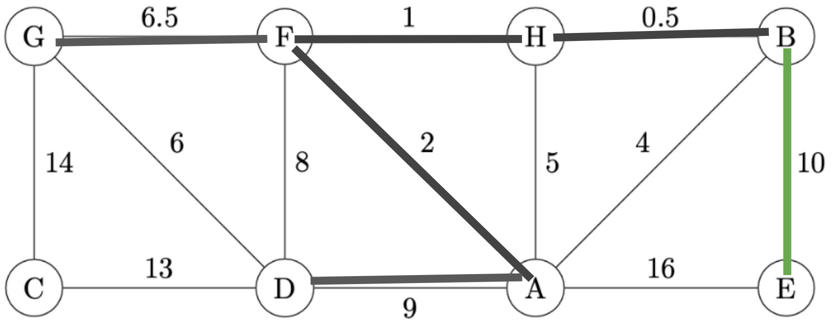
```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```




```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

	A	B	C	D	E	F	G	H
dist	0	3.5	22	9	13.5	2	8.5	3
Prev	-1	H	D	A	B	A	F	F

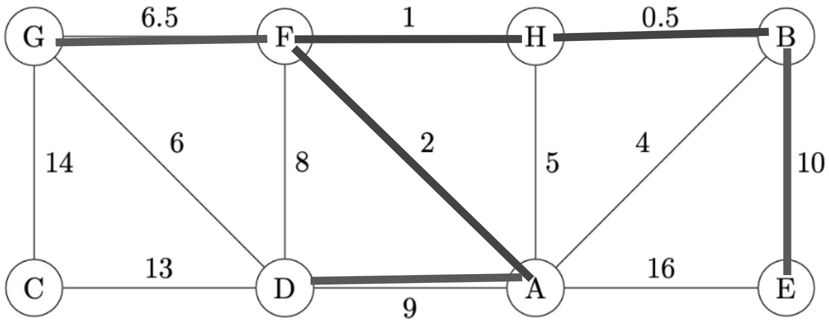
	Q	
	(13.5, E)	
	(22 , C)	



	A	B	C	D	E	F	G	H
dist	0	3.5	22	9	13.5	2	8.5	3
Prev	-1	H	D	A	B	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

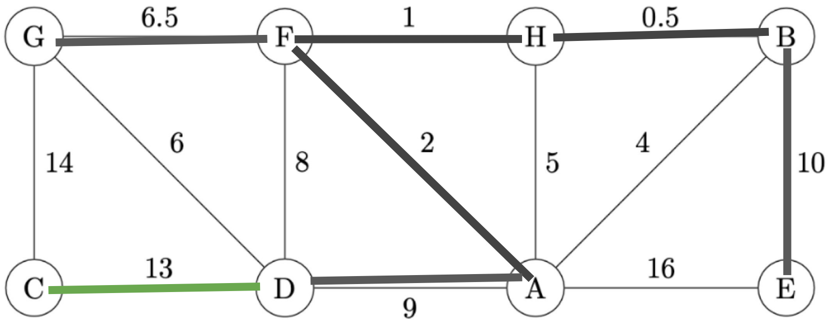
Q
(13.5, E)
(22 , C)




```
for each w ∈ V adjacent to u still in Q do
   d ← dist[u] + weight(u, w)
   if d < dist[w] then
      dist[w] ← d
      prev[w] ← u
      Q.set(d, w)
```

	A	B	C	D	E	F	G	H
dist	0	3.5	22	9	13.5	2	8.5	3
Prev	-1	H	D	A	B	A	F	F

Q
(22 , C)



	A	B	C	D	E	F	G	H
dist	0	3.5	22	9	13.5	2	8.5	3
Prev	-1	H	D	A	B	A	F	F

```
for each w ∈ V adjacent to u still in Q do
    d ← dist[u] + weight(u, w)
    if d < dist[w] then
        dist[w] ← d
        prev[w] ← u
        Q.set(d, w)
```

Q
(22 , C)

Question 2

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

negative weights

Dijkstra may not work..

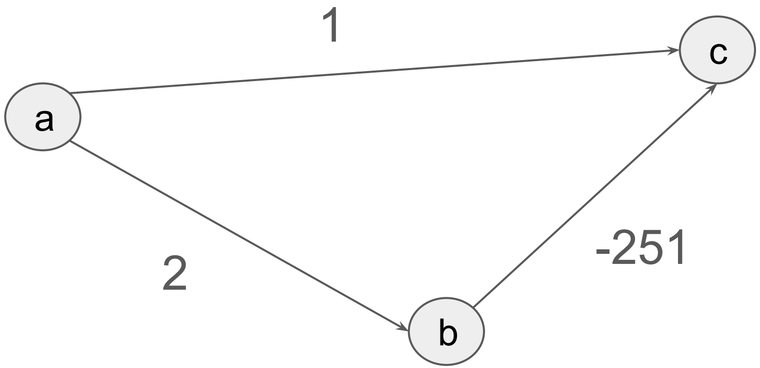
Question 1

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

Intuition: once a vertex is removed from the pQ, its shortest path is fixed.

negative edge *may* be at the end, Dijkstra won't encounter it in time



pQ

1. (a,0)
2. (b,inf)
3. (c,inf)

	A	B	C
dist	0	inf	inf
prev	A	-1	-1

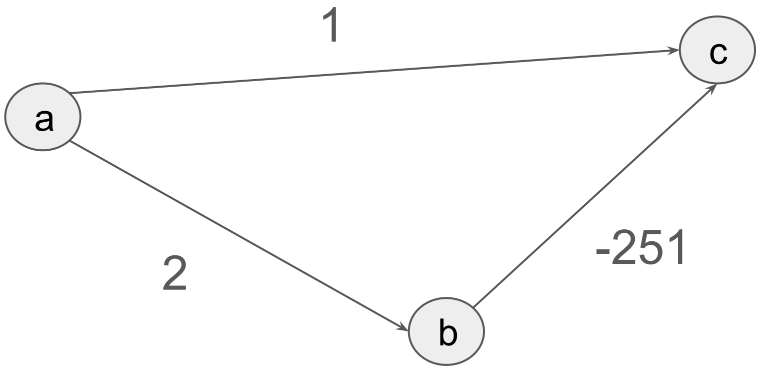
Question 1

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

Intuition: once a vertex is removed from the pQ, its shortest path is fixed.

negative edge *may* be at the end, Dijkstra won't encounter it in time



pQ

1. (c,**1**)
2. (b,**2)**

	A	B	C
dist	0	2	1
prev	A	-1	-1

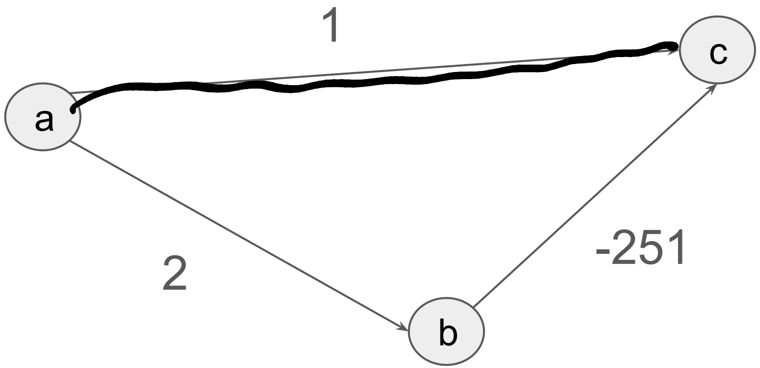
Question 1

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

Intuition: once a vertex is removed from the pQ, its shortest path is fixed.

negative edge *may* be at the end, Dijkstra won't encounter it in time



pQ

1. ~~(c,1)~~
2. (b,**2**)

	A	B	C
dist	0	2	1
prev	A	-1	-1

We take c off here, but there is clearly a shorter (negative) path!

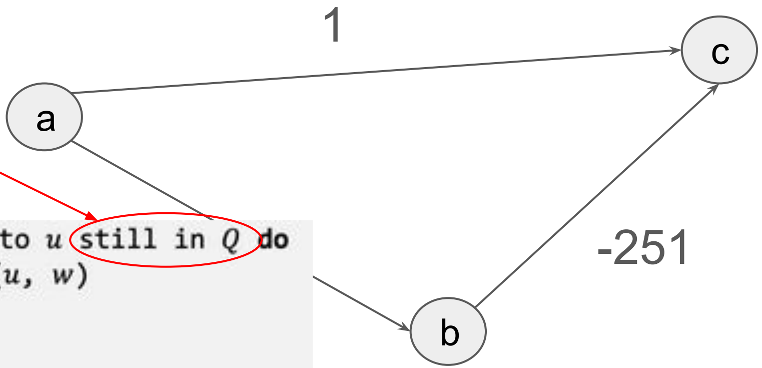
Question 1

(Dijkstra's algorithm)

1. Give a simple example of a directed graph with negative-weighted edges for which Dijkstra's algorithm produces an incorrect answer.

Intuition: once a vertex is removed from the pQ, its shortest path is fixed.

negative edge *may* be at the end, Dijkstra won't encounter it in time



pQ

1. (b,**2)**

```
or each w ∈ V adjacent to u still in Q do
  d ← dist[u] + weight(u, w)
  if d < dist[w] then
     dist[w] ← d
     prev[w] ← u
     Q.set(d, w)
```

		B	C
dist	0	2	1
prev	A	-1	-1

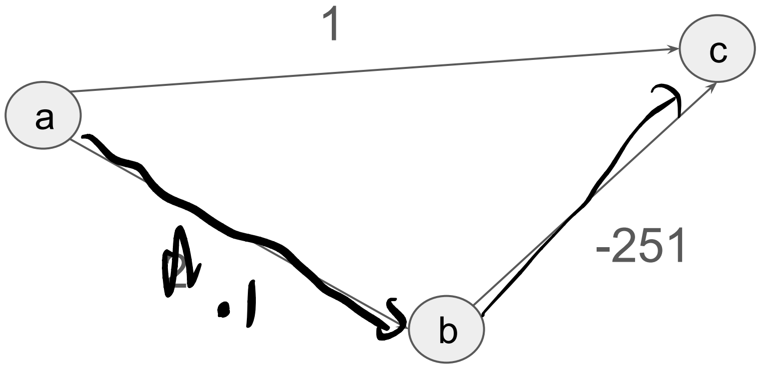
By the time we encounter b and find the path to c, c no longer in the pQ!!!

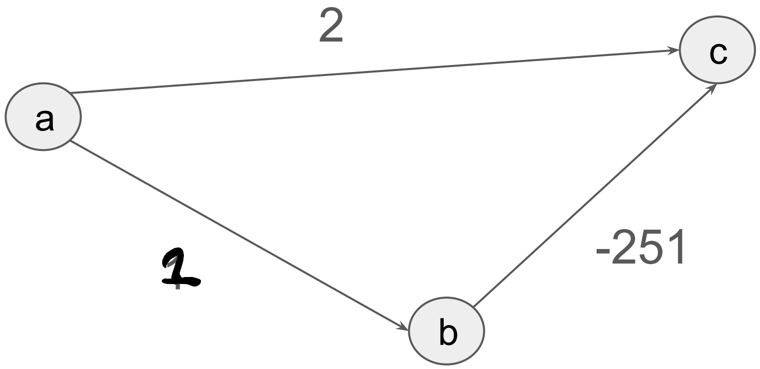
2. Given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, but all other edge weights are non-negative, and there are no negative-weighted cycles. Can the Dijkstra's algorithm correctly find all the shortest paths from s in this graph?

This seems plausible…

Can we update the previous example so

that it actually finds the shortest path a -> c



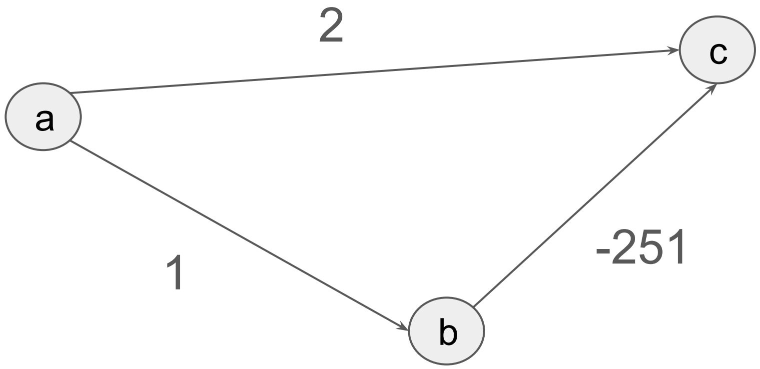


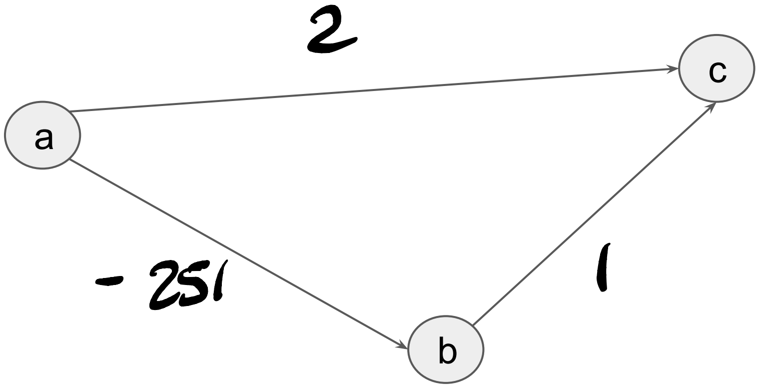
pQ

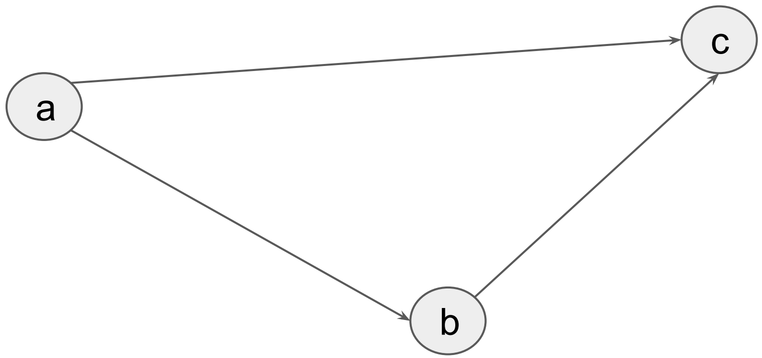
1.
2.
3.

	A	B	C
dist			
prev			

Other examples that work



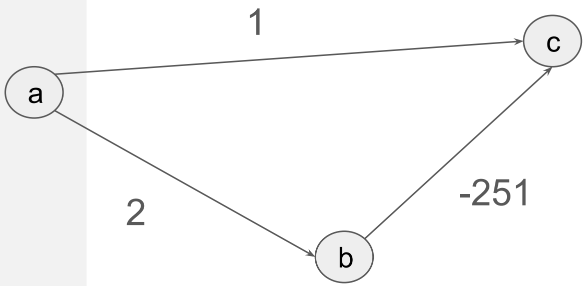




2. Given a weighted, directed graph $G = (V, E)$ in which edges that leave the source vertex s may have negative weights, but all other edge weights are non-negative, and there are no negative-weighted cycles. Can the Dijkstra's algorithm correctly find all the shortest paths from s in this graph?

Another way to see this

```
algorithm DijkstraShortestPath(G(V,E), s ∈ V)

   let dist:V → ℤ
   let prev:V → V
   let Q be an empty priority queue

   dist[s] ← 0
   for each v ∈ V do
      if v ≠ s then
         dist[v] ← ∞
      end if
      prev[v] ← -1
      Q.add(dist[v], v)
   end for

   while Q is not empty do
      u ← Q.getMin()
      for each w ∈ V adjacent to u still in Q do
         d ← dist[u] + weight(u, w)
         if d < dist[w] then
            dist[w] ← d
            prev[w] ← u
            Q.set(d, w)
         end if
      end for
   end while

   return dist, prev
end algorithm
```



Update step correct regardless if negative edge

We only have an error **if we don't update (see prev example)**

3. Your classmate claims that Dijkstra's algorithm relaxes the edges of every shortest path in the graph in the order in which they appear on the path. Show that him/her is mistaken by constructing a directed graph for which the Dijkstra's algorithm could relax the edges of a shortest path out of order.

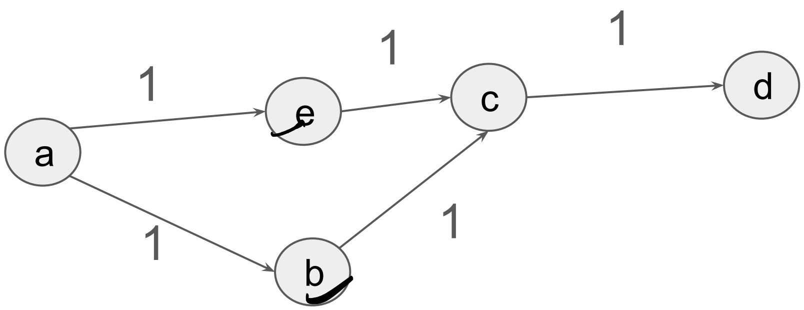
Hint: The shortest path between two vertices in the graph is not necessarily unique.

Relaxing an edge: checking if the current best known distance is better than the distance of the current edge

Basically, this step

```
algorithm DijkstraShortestPath(G(V,E), s ∈ V)

   let dist:V → ℤ
   let prev:V → V
   let Q be an empty priority queue

   dist[s] ← 0
   for each v ∈ V do
      if v ≠ s then
         dist[v] ← ∞
      end if
      prev[v] ← -1
      Q.add(dist[v], v)
   end for

   while Q is not empty do
      u ← Q.getMin()
      for each w ∈ V adjacent to u still in Q do
         d ← dist[u] + weight(u, w)
         if d < dist[w] then
            dist[w] ← d
            prev[w] ← u
            Q.set(d, w)
         end if
      end for
   end while

   return dist, prev
end algorithm
```

Hint: The shortest path between two vertices in the graph is not necessarily unique.



pQ

1.
2.
3.
4.
5.

(order depends on priority)

	A	B	C	D	E
dist					
prev					

I can do

a → e → c → d

or

a → b → c → d

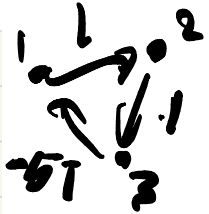
Question 2

(Bellman-Ford algorithm)

For the Bellman-Ford algorithm, explain

1. why it only requires $|V| - 1$ passes?
2. why the last pass $(|V| - 1)$ through the edges will determine if there are negative weight cycles or not?

$|\text{path}| \leq |V| - 1$



dijstra but for neg. weight edges.

```
Bellman-Ford (G, s):
    dist [] = ∞
    prev[] = -1
    dist [s] = 0
    for i = 1, ..., n-1:
        for each edge e = (u,v) ∈ E :
            d ← dist [u] + w(e)
            if d < dist [v]:
                dist [v] = d
                prev [v] = u

    for each e = (u,v) ∈ E :
        if Dist(u) + w(e) < d[v]:
            negative cycle
```

take another step

dijkstra step

→ finds all shortest paths of length i

neg cycle check