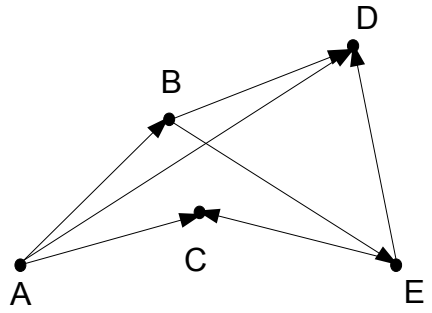
1) For the given graph: a) give it's adjacency matrix representation and adjacency list representation (using either Python's lists or Python's dictionaries).



Adjacency matrix representation:

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	0	1	1
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	1	1	0

Adjacency list representation:
A → B, C, D, E
B → D, E
C →
D →
E → C, D

Adjacency list using Python's lists:

```
G = [
[ 'A', [ 'B', 'C', 'D', 'E' ]],
[ 'B', [ 'D', 'E' ]],
[ 'C', [ ]],
[ 'D', [ ]],
[ 'E', [ 'C', 'D' ]] ]
```

Adjacency list using Python's dictionary:

```
G = {
 'A' : { 'B', 'C', 'D', 'E' },
 'B' : { 'D', 'E' },
 'C' : { },
 'D' : { },
 'E' : { 'C', 'D' } }
```

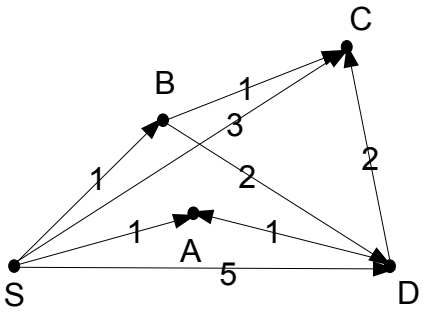
b) in-degree of D = **3** in-degree of A = **0**

out-degree of B = **2** out-degree of C = **0**

c) Does it have cycles? If yes, list them.

No it doesn't have cycles.

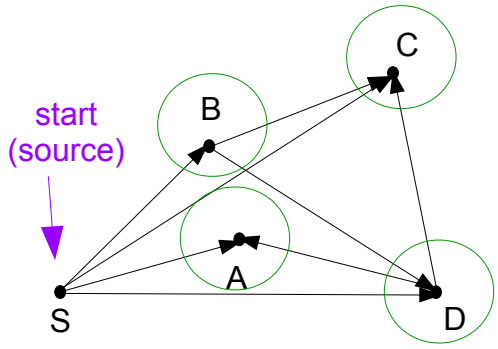
2) For the following weighted directed graph, give its adjacency list representation



Adjacency list representation:
S → A(1), B(1), C(3), D(5)
A →
B → C(1), D(2)
C →
D → A(1), C(2)

Note that we were not asked to give adjacency list representation via Python's list or Python's dictionary, therefore, we don't do it.

3) For the given unweighted directed graph, use the unweighted shortest path algorithm (**BFS**).



queue: ~~S~~
v: S w: A then B then C then D

	S	A	B	C	D
parent	None	~~None~~ S	~~None~~ S	~~None~~ S	~~None~~ S
distance	0	1	1	1	1

queue: D, C, B, ~~A~~
v: A

	S	A	B	C	D
parent	None	S	S	S	S
distance	0	1	1	1	1

queue: D, ~~C~~, B
v: B w: C then D (but their parents are not none)

	S	A	B	C	D
parent	None	S	S	S	S
distance	0	1	1	1	1

queue: D, ~~C~~
c: C

	S	A	B	C	D
parent	None	S	S	S	S
distance	0	1	1	1	1

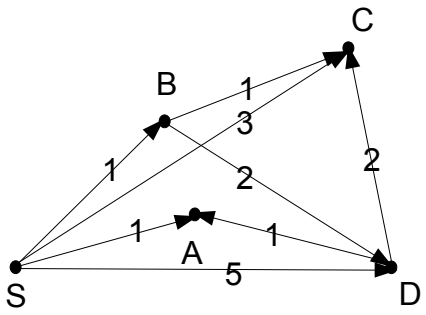
queue: ~~D~~
v: D w: C then D (but their parents are not none)

	S	A	B	C	D
parent	None	S	S	S	S
distance	0	1	1	1	1

Give the path from S to C: S → C

Give the path from S to D: S → D

4) For the given weighted directed graph, use the weighted shortest path algorithm (**Dijkstra's**).



0 ∞ ∞ ∞ ∞
priority queue: ~~S~~, A, B, C, D S is dequeued,
since it has the smallest distance

	S	A	B	C	D
parent	None	~~None~~ S	~~None~~ S	~~None~~ S	~~None~~ S
distance	0	~~infty~~ 1	~~infty~~ 1	~~infty~~ 3	~~infty~~ 5

1 1 3 5
priority queue: ~~A~~, B, C, D A is dequeued,
(since it has the smallest distance + alphabetical order)

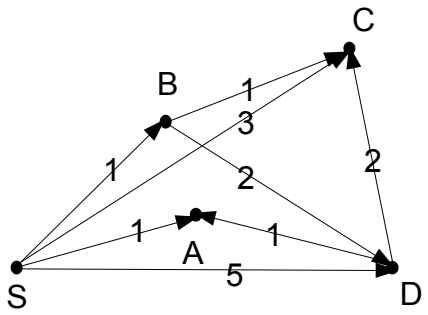
	S	A	B	C	D
parent	None	S	S	S	S
distance	0	1	1	3	5

A doesn't have any vertices accessible from it.

1 3 5
priority queue: ~~B~~, C, D B is dequeued,
(since it has the smallest distance)

	S	A	B	C	D
parent	None	S	S	~~S~~ B	~~S~~ B
distance	0	1	1	~~3~~ 2	~~5~~ 3

dist. from B to C is 1+1=2; 3 > 2, hence update info
dist. from B to D is 1+2=3; 5 > 3, hence update info



priority queue: ~~2~~ C, ~~3~~ D C is dequeued,
(since it has the smallest distance)

	S	A	B	C	D
parent	None	S	S	B	B
distance	0	1	1	2	3

dist. from B to C is 1+1=2; 3 > 2, hence update info
dist. from B to D is 1+2=3; 5 > 3, hence update info

priority queue: ~~3~~ D D is dequeued,
(since it is the only vertex left in the queue)

	S	A	B	C	D
parent	None	S	S	B	B
distance	0	1	1	2	3

dist. from D to A is 3+1=4; 1 > 4 is false, hence nothing is changed
dist. from D to C is 3+2=5; 2 > 5 is false, hence nothing is changed

The table is ready to be used. For example, the shortest path from S to D is S → B → D and
The shortest path from S to C is S → B → C