CS101 Algorithms and Data Structures

Topological Sort

Textbook Ch 22.4

Topological Sort

In this topic, we will discuss:

- Motivations
- Review the definition of a directed acyclic graph (DAG)
- Describe a topological sort and applications
- Prove the existence of topological sorts on DAGs
- Describe an abstract algorithm for a topological sort
- Do a run-time and memory analysis of the algorithm
- Describe a concrete algorithm
- Define critical times and critical paths

Outline

- Topological sorting
  - Definitions
  - Algorithm
- Finding the critical path

Motivation

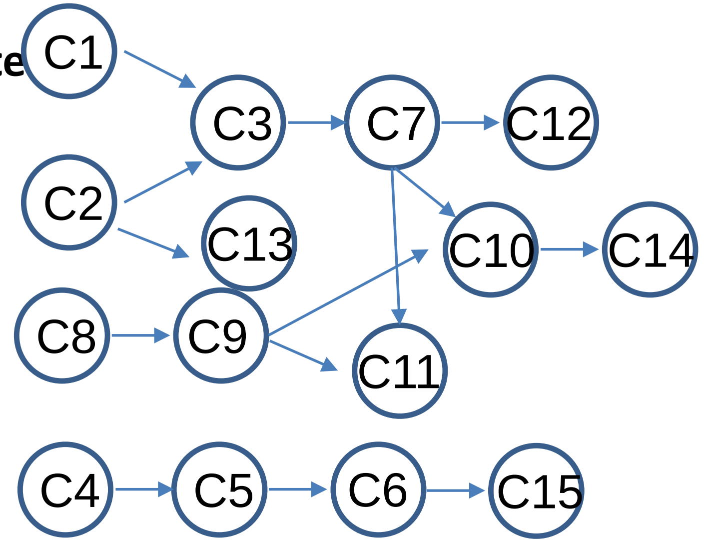
Dependency between tasks: one task is required to be done before the other task can be done

Dependencies form a partial ordering

- A partial ordering on a finite number of objects can be represented as a directed acyclic graph (DAG)

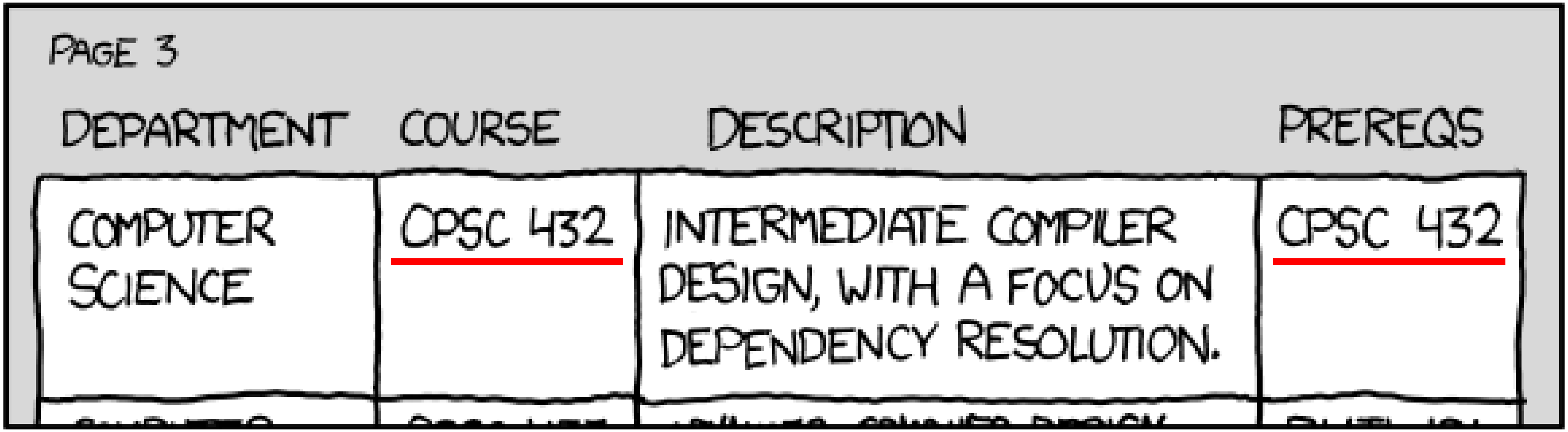
SIST course curriculum

Course No	Course title	Prerequisite
C1	Programming	None
C2	Discrete mathematics	None
C3	Data structure	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear algebra	C5
C7	Algorithm	C3
C8	Logic and computer design basis	None
C9	Computer composition	C8
C10	Operating system	C7,C9
C11	Compilation principle	C7,C9
C12	Database	C7
C13	Computing theory	C2
C14	Computer network	C10
C15	Numerical analysis	C6



Motivation

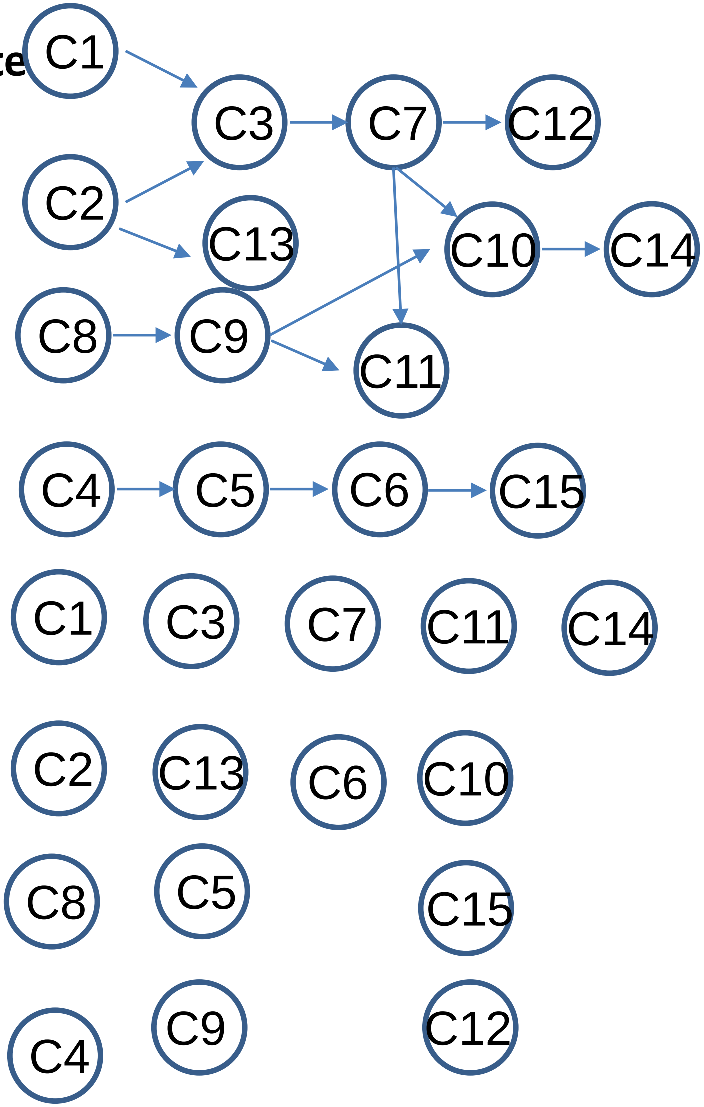
Cycles in dependencies can cause issues...



http://xkcd.com/754/

SIST course curriculum

Course No	Course title	Prerequisite
C1	Programming	None
C2	Discrete mathematics	None
C3	Data structure	C1, C2
C4	Calculus I	None
C5	Calculus II	C4
C6	Linear algebra	C5
C7	Algorithm	C3
C8	Logic and computer design basis	None
C9	Computer composition	C8
C10	Operating system	C7,C9
C11	Compilation principle	C7,C9
C12	Database	C7
C13	Computing theory	C2
C14	Computer network	C10
C15	Numerical analysis	C6



Topological sorting

Given a set of tasks with dependencies, is there an order in which we can complete the tasks?

A topological sorting of the vertices in a DAG is an ordering

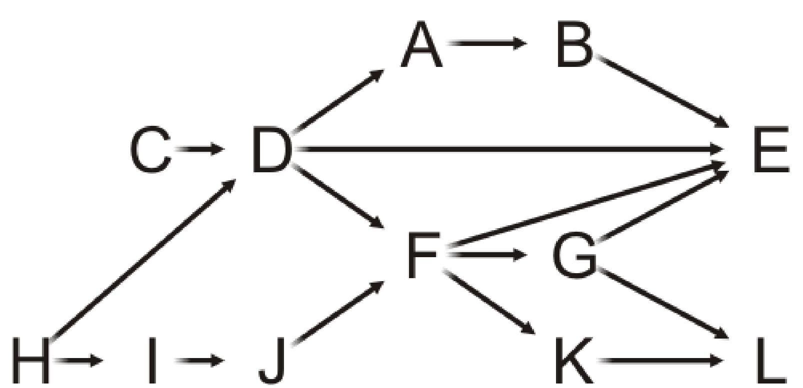
$$v_1, v_2, v_3, \ldots, v_{|V|}$$

such that v_j appears before v_k if there is a path from v_j to v_k

Example

Given this DAG, a topological sort is

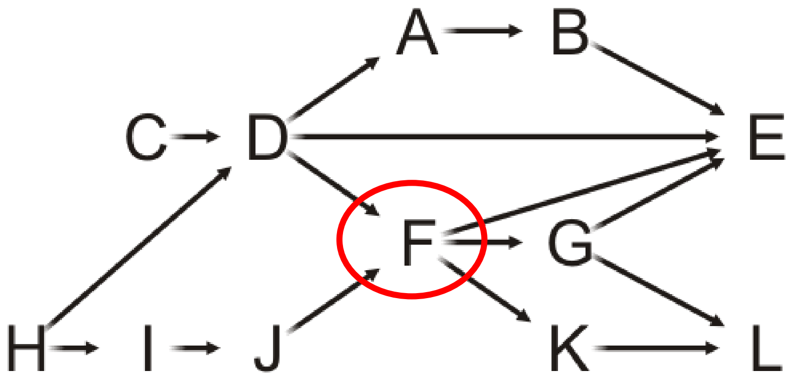
H, C, I, D, J, A, F, B, G, K, E, L



Example

For example, there are paths from H, C, I, D and J to F, so all these must come before F in a topological sort

H, **C**, **I**, **D**, **J**, A, **F**, B, G, K, E, L



Clearly, this sorting need not be unique

Applications

Taking courses

- The courses must be taken in an order such that the prerequisites of a course are taken before that course

Applications

Consider you getting ready for a dinner out

You must wear the following:

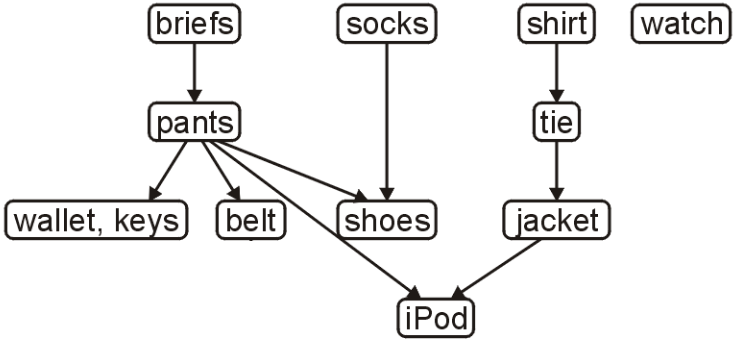
- jacket, shirt, briefs, socks, tie, etc.

There are certain constraints:

- the pants really should go on after the briefs,
- socks are put on before shoes

Applications

The following is a task graph for getting dressed:



Many people would go like this (a possible topological sort):

briefs, shirt, socks, pants, belt, tie, jacket, wallet, keys, iPod, watch, shoes

Another topological sort is:

briefs, pants, wallet, keys, belt, socks, shoes, shirt, tie, jacket, iPod, watch

Topological Sort

Theorem:

A graph is a DAG if and only if it has a topological sorting

Proof strategy:

Such a statement is of the form $a \leftrightarrow b$ and this is equivalent to:

$a \rightarrow b$ and $b \rightarrow a$

Topological Sort

First, we need a two lemmas:

– A DAG always has at least one vertex with in-degree zero
    - That is, it has at least one *source*

Proof by contradiction:

– If we cannot find a vertex with in-degree zero, we will show there must be a cycle
– Start with any vertex and define a list $L = (v)$
– Then iterate this loop $|V|$ times:
    - The first vertex ℓ_1 in the list L does not have in-degree zero
    - So we can find a vertex w such that (w, ℓ_1) is an edge
    - Add w to the list: $L = (w, \ell_1, \ldots, \ell_k)$
– By the pigeon-hole principle, at least one vertex must appear twice
    - This forms a cycle; hence a contradiction, as this is a DAG

Topological Sort

First, we need a two lemmas:

- Any sub-graph of a DAG is a DAG

Proof:

- If a sub-graph has a cycle, that same cycle must appear in the super-graph
- We assumed the super-graph was a DAG
- This is a contradiction

∴ the sub-graph must be a DAG

Topological Sort

We will start with showing $a \rightarrow b$:

If a graph is a DAG, it has a topological sort

Proof by induction:

A graph with one vertex is a DAG and it has a topological sort

Assume a DAG with n vertices has a topological sort

A DAG with $n + 1$ vertices must have at least one vertex v of in-degree zero

Removing the vertex v and consider the vertex-induced sub-graph with the remaining n vertices

- If this sub-graph has a cycle, so would the original graph—contradiction
- Thus, the graph with n vertices is also a DAG, therefore it has a topological sort

Add the vertex v to the start of the topological sort to get one for the graph of size $n + 1$

Topological Sort

Next, we will show that $b \rightarrow a$:

If a graph has a topological ordering, it must be a DAG

We will show this by showing the contrapositive: $\neg a \rightarrow \neg b$:

If a graph is not a DAG, it does not have a topological sort

By definition, it has a cycle: $(v_1, v_2, v_3, \ldots, v_k, v_1)$

- In any topological sort, v_1 must appear before v_2, because (v_1, v_2) is a path
- However, there is also a path from v_2 to v_1: $(v_2, v_3, \ldots, v_k, v_1)$
- Therefore, v_2 must appear in the topological sort before v_1

This is a contradiction, therefore the graph cannot have a topological sort

$\therefore a \leftrightarrow b$: A graph is a DAG if and only if it has a topological sorting

Outline

- Topological sorting
  - Definitions
  - Algorithm
- Finding the critical path

Topological Sort

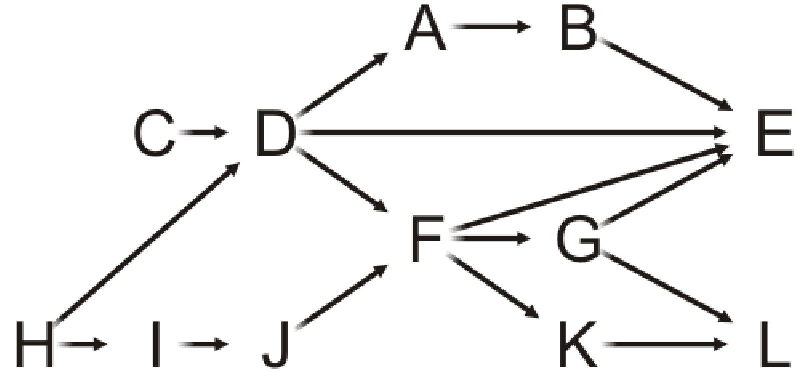
Idea:

- Given a DAG V, iterate:
  - Find a vertex v in V with in-degree zero
  - Let v be the next vertex in the topological sort
  - Continue iterating with the vertex-induced sub-graph $V \setminus \{v\}$

Example

On this graph, iterate the following $|V| = 12$ times

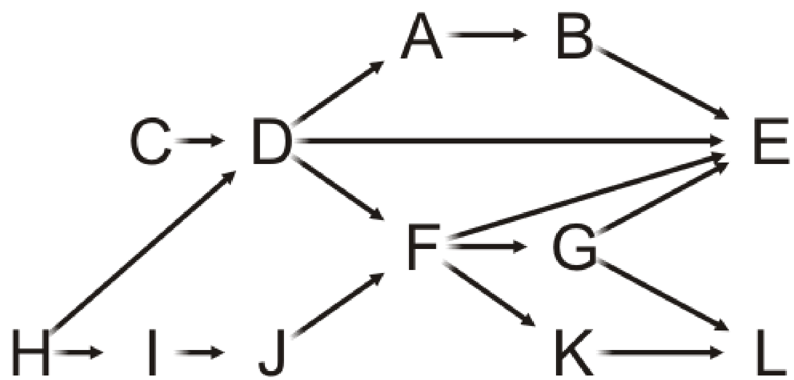
- Choose a vertex v that has in-degree zero
- Let v be the next vertex in our topological sort
- Remove v and all edges connected to it



Example

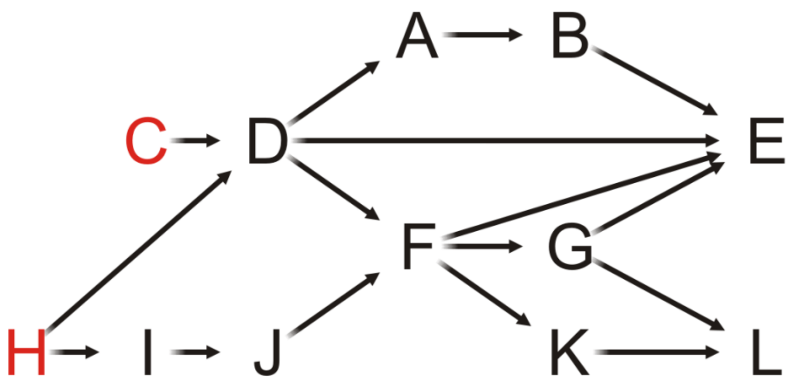
Let's step through this algorithm with this example

– Which task can we start with?



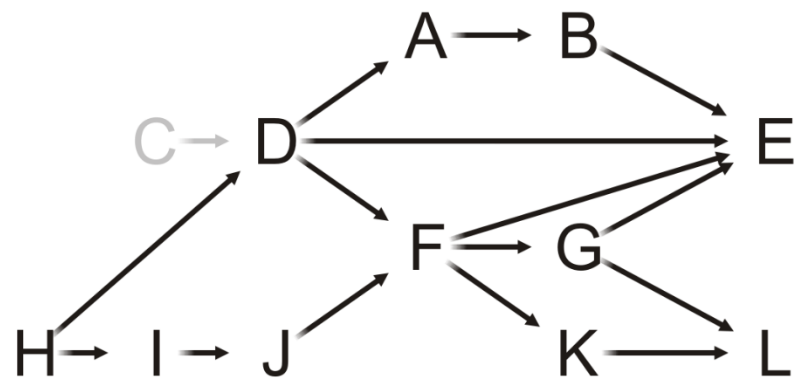
Example

Of Tasks C or H, choose Task C



Example

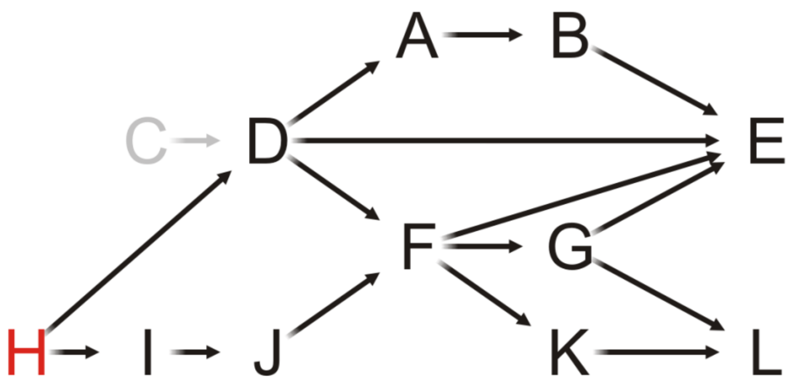
Having completed Task C, which vertices have in-degree zero?



C

Example

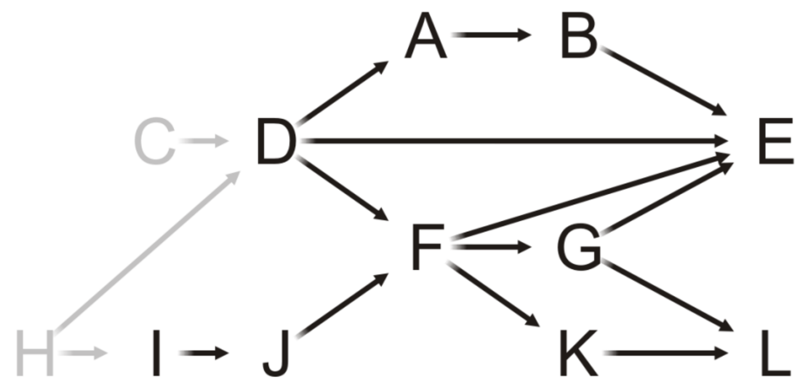
Only Task H can be completed, so we choose it



C

Example

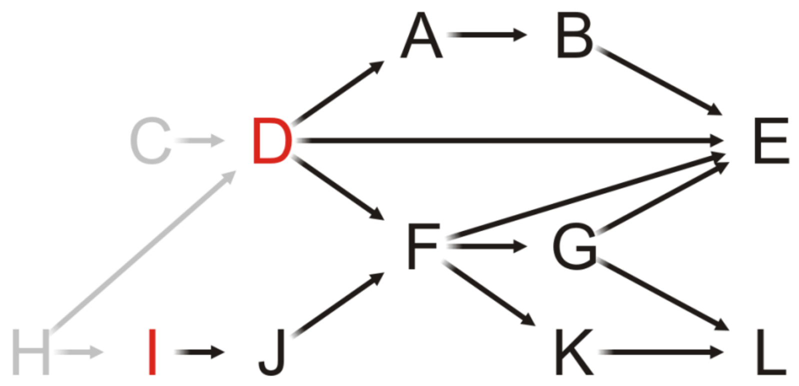
Having removed H, what is next?



C, H

Example

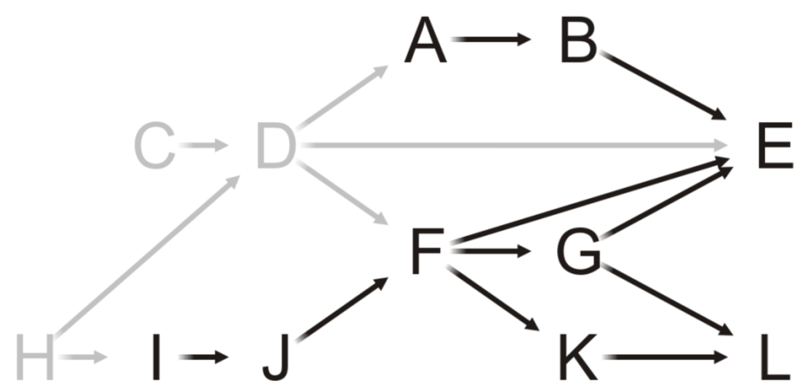
- Both Tasks D and I have in-degree zero
  - Let us choose Task D



C, H

Example

We remove Task D, and now?

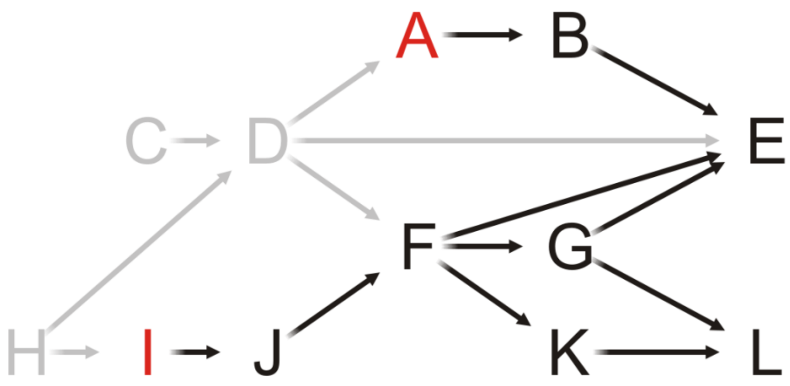


C, H, D

Example

Both Tasks A and I have in-degree zero

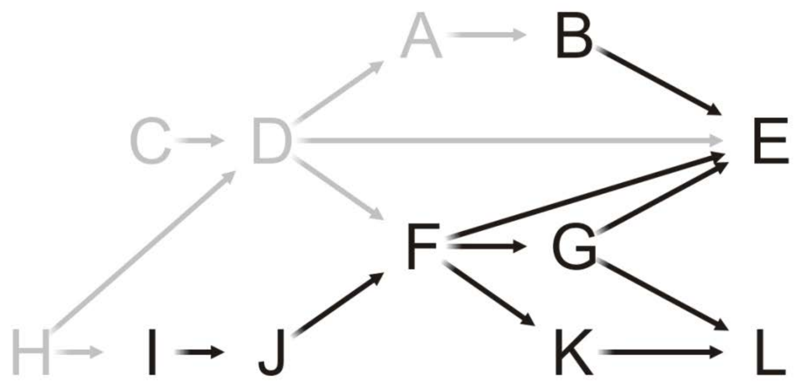
- Let's choose Task A



C, H, D

Example

Having removed A, what now?

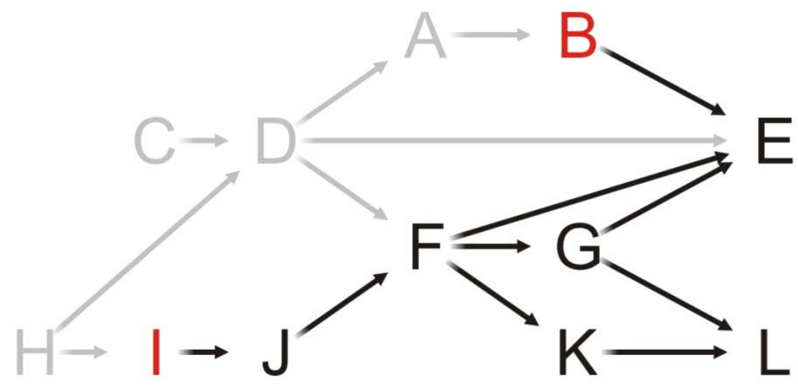


C, H, D, A

Example

Both Tasks B and I have in-degree zero

– Choose Task B

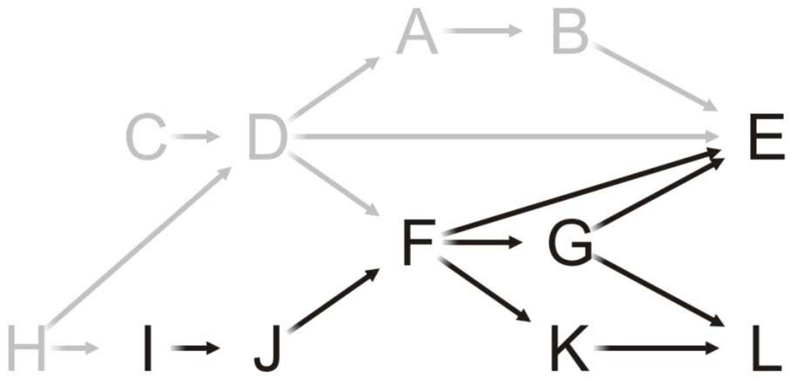


C, H, D, A

Example

Removing Task B, we note that Task E still has an in-degree of two

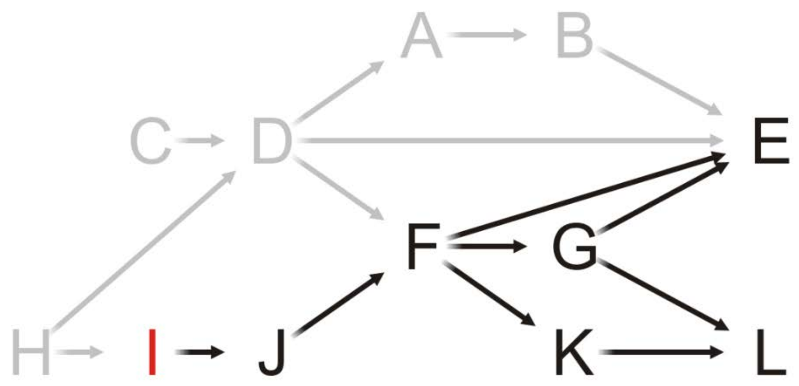
– Next?



C, H, D, A, B

Example

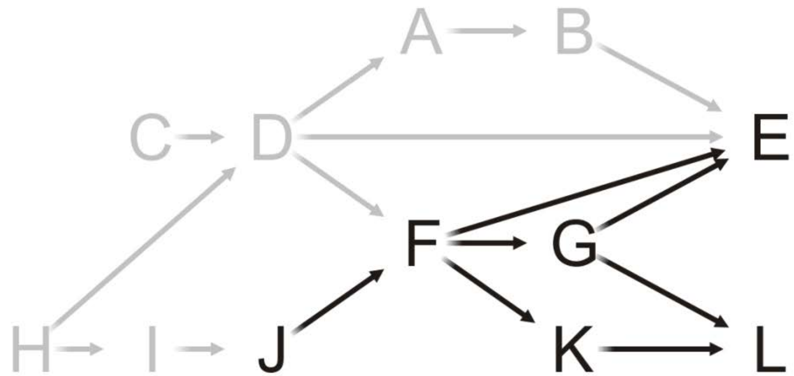
As only Task I has in-degree zero, we choose it



C, H, D, A, B

Example

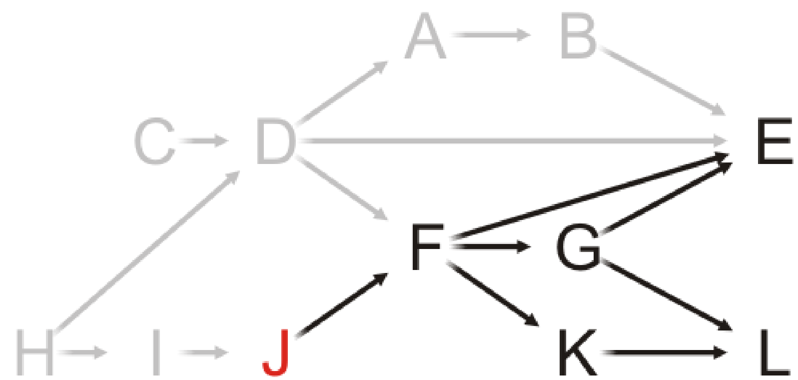
Having completed Task I, what now?



C, H, D, A, B, I

Example

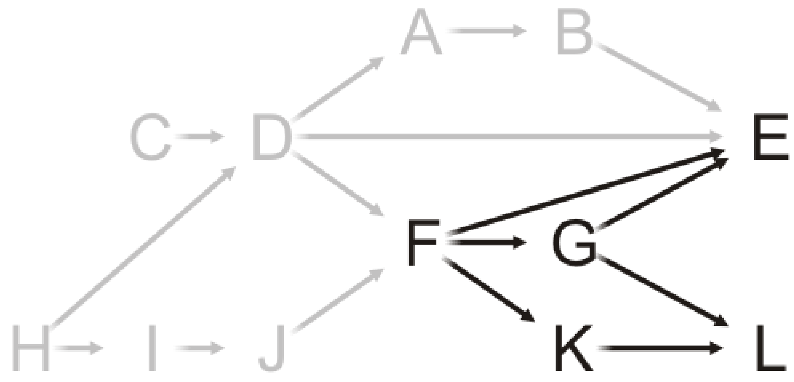
Only Task J has in-degree zero: choose it



C, H, D, A, B, I

Example

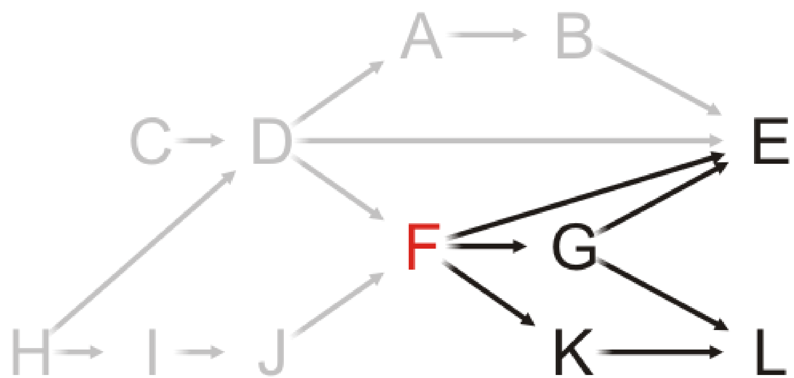
Having completed Task J, what now?



C, H, D, A, B, I, J

Example

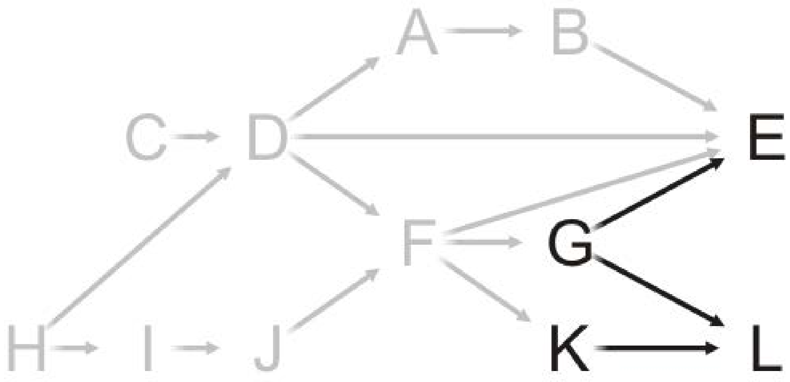
Only Task F can be completed, so choose it



C, H, D, A, B, I, J

Example

What choices do we have now?

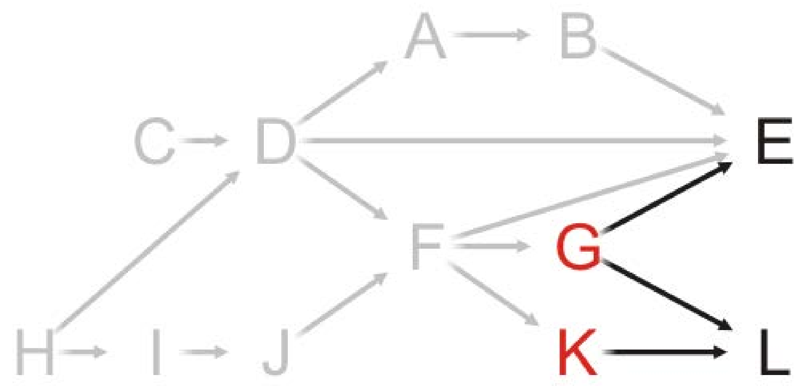


C, H, D, A, B, I, J, F

Example

We can perform Tasks G or K

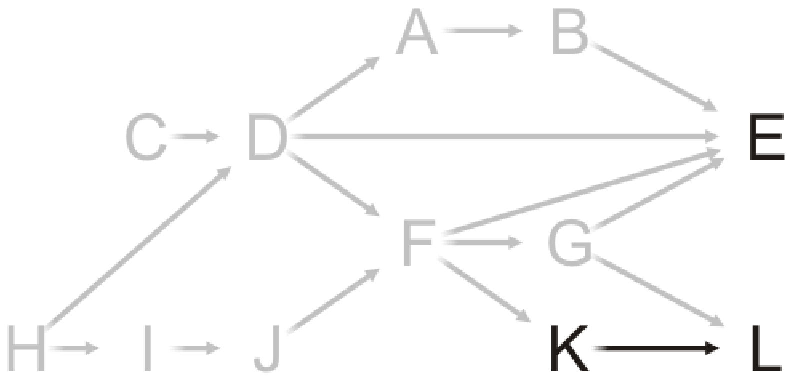
- Choose Task G



C, H, D, A, B, I, J, F

Example

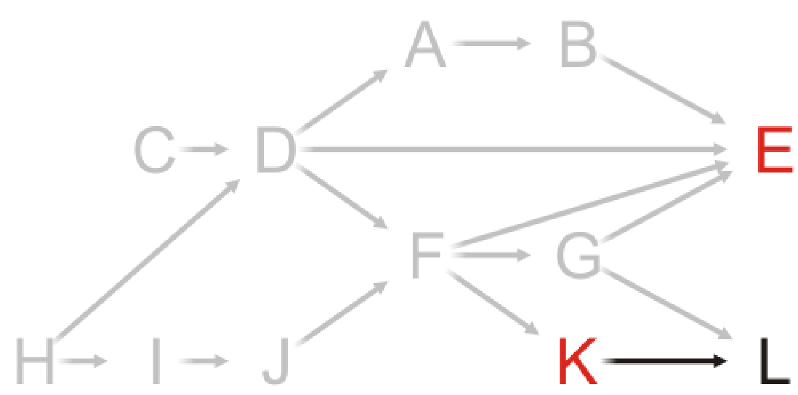
Having removed Task G from the graph, what next?



C, H, D, A, B, I, J, F, G

Example

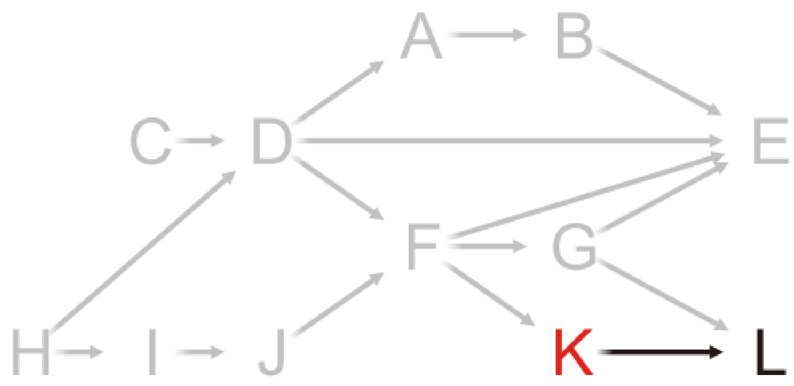
Choosing between Tasks E and K, choose Task E



C, H, D, A, B, I, J, F, G

Example

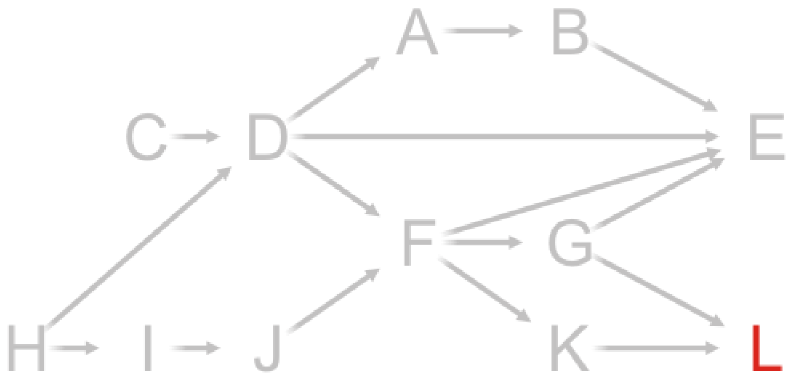
At this point, Task K is the only one that can be run



C, H, D, A, B, I, J, F, G, E

Example

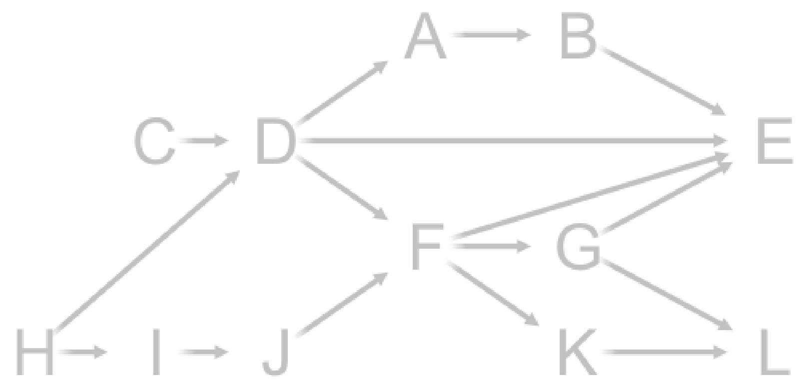
And now that both Tasks G and K are complete, we can complete Task L



C, H, D, A, B, I, J, F, G, E, K

Example

There are no more vertices left

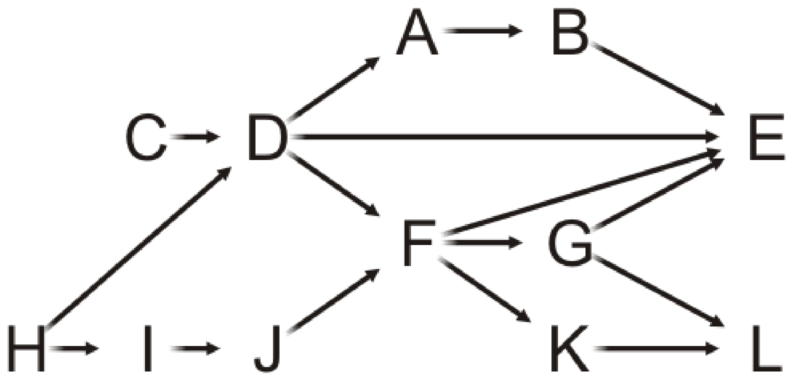


C, H, D, A, B, I, J, F, G, E, K, L

Example

Thus, one possible topological sort would be:

C, H, D, A, B, I, J, F, G, E, K, L

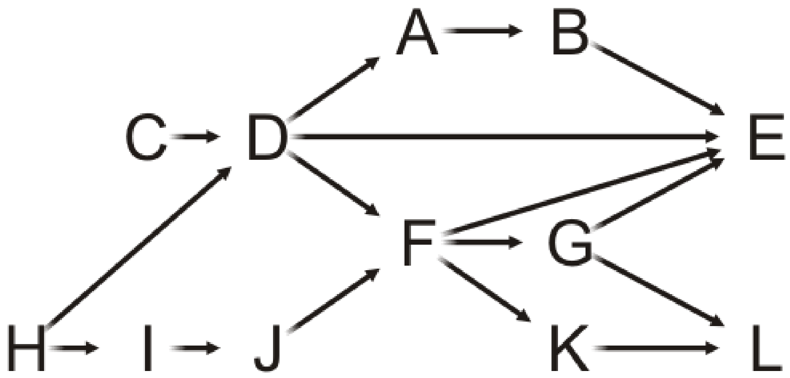


Example

Note that topological sorts need not be unique:

C, H, D, A, B, I, J, F, G, E, K, L

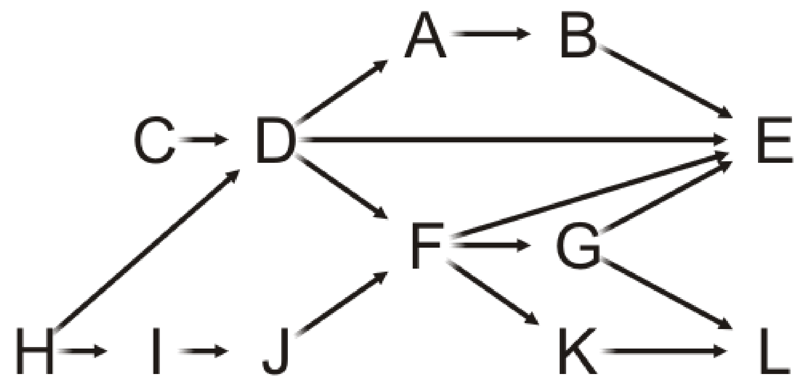
H, I, J, C, D, F, G, K, L, A, B, E



Analysis

What are the tools necessary for a topological sort?

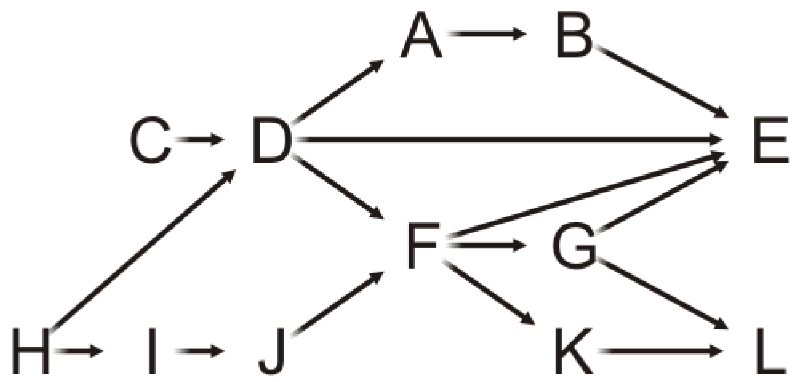
- We must know and be able to update the in-degrees of each of the vertices
- We could do this with a table of the in-degrees of each of the vertices
- This requires $\Theta(|V|)$ memory



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

We must iterate at least $|V|$ times, so the run-time must be $\Omega(|V|)$

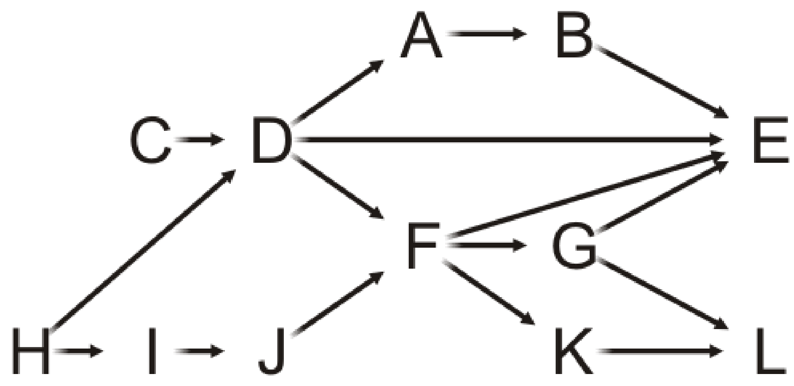


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

We need to find vertices with in-degree zero

- We could loop through the table with each iteration
- The run time would be $O(|V|^2)$

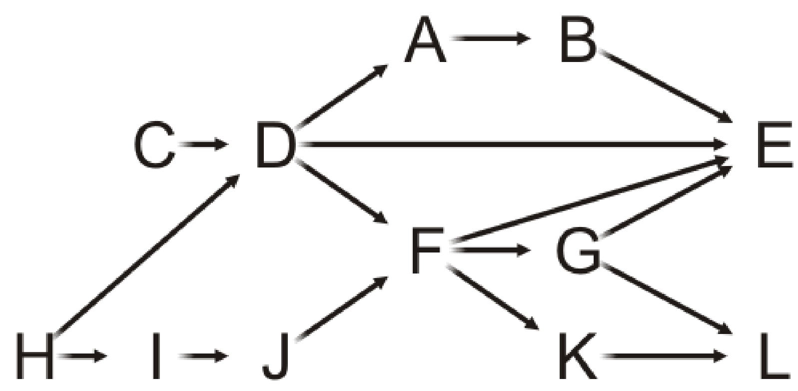


A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

A better approach

- Use a queue (or other container) to temporarily store those vertices with in-degree zero
- Each time the in-degree of a vertex is decremented to zero, push it onto the queue



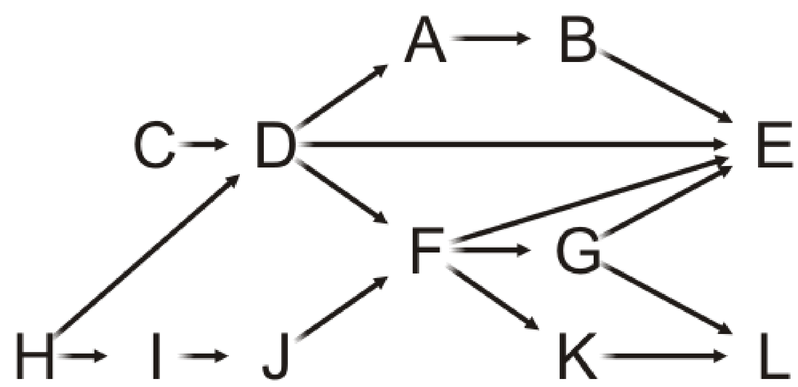
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What are the run times associated with the queue?

- Initially, we must scan through each of the vertices: $\Theta(|V|)$
- For each vertex, we will have to push onto and pop off the queue once, also $\Theta(|V|)$

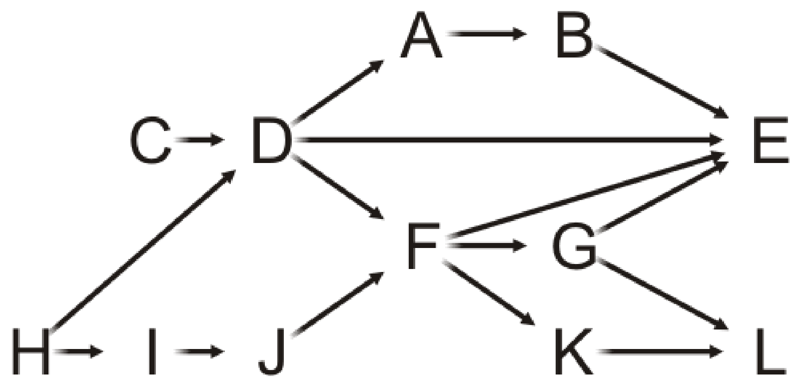
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2



Analysis

Finally, every time we remove a vertex v, all its edges shall also be removed and the in-degree table be updated

- The run time of these operations is $\Omega(|E|)$
- If we are using an adjacency matrix: $\Theta(|V|^2)$
- If we are using an adjacency list: $\Theta(|E|)$



Here, $|E| = 16$

A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	+ 2

16

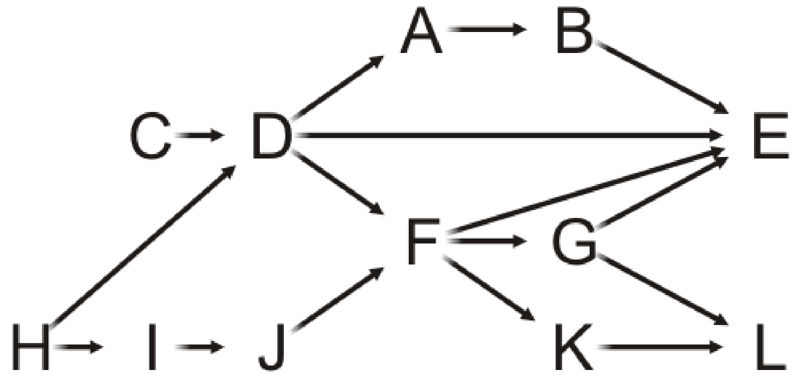
Analysis

Therefore, the run time of a topological sort is:

$\Theta(|V| + |E|)$ if we use an adjacency list

$\Theta(|V|^2)$ if we use an adjacency matrix

and the memory requirements is $\Theta(|V|)$



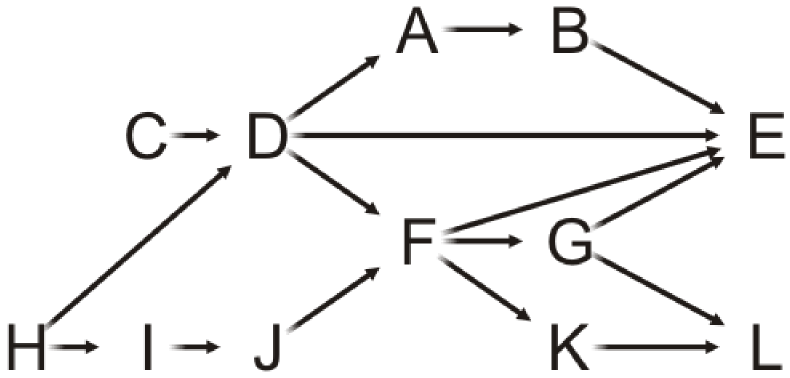
A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Analysis

What happens if at some step, all remaining vertices have an in-degree greater than zero?

– There must be at least one cycle within that sub-set of vertices

Consequence: we now have an $\Theta(|V| + |E|)$ algorithm for determining if a graph has a cycle



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Implementation

Thus, to implement a topological sort:

- Allocate memory for and initialize an array of in-degrees
- Create a queue and initialize it with all vertices that have in-degree zero

While the queue is not empty:

- Pop a vertex from the queue
- Decrement the in-degree of each neighbor
- Those neighbors whose in-degree was decremented to zero are pushed onto the queue

Implementation

We will use an array implementation of our queue

Because we place each vertex into the queue exactly once

- We must never resize the array
- We do not have to worry about the queue cycling

Most importantly, however, because of the properties of a queue

- When we finish, the underlying array stores the topological sort

Implementation

The operations with our queue

– Initialization

```
Type array[vertex_size()];
int ihead = 0, itail = -1;
```

– Testing if empty:

```
ihead == itail + 1
```

– For push

```
++itail;
array[itail] = next vertex;
```

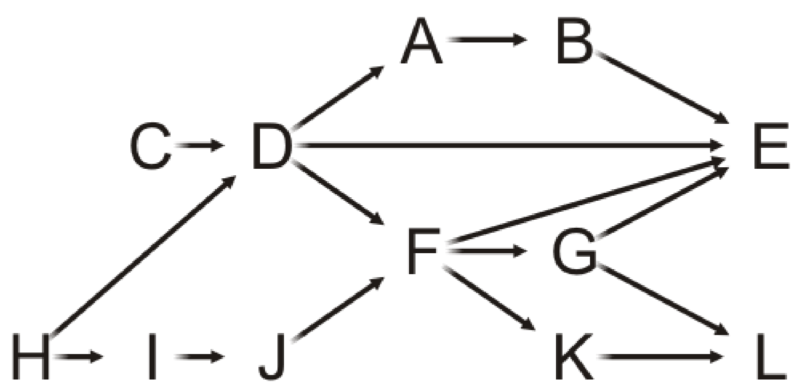
– For pop

```
Type current_top = array[ihead];
++ihead;
```

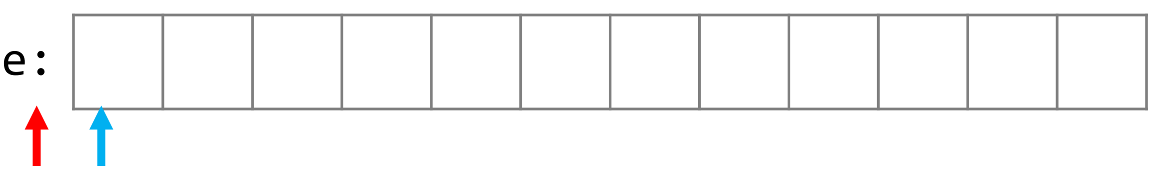
Example

With the previous example, we initialize:
- The array of in-degrees
- The queue

A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2



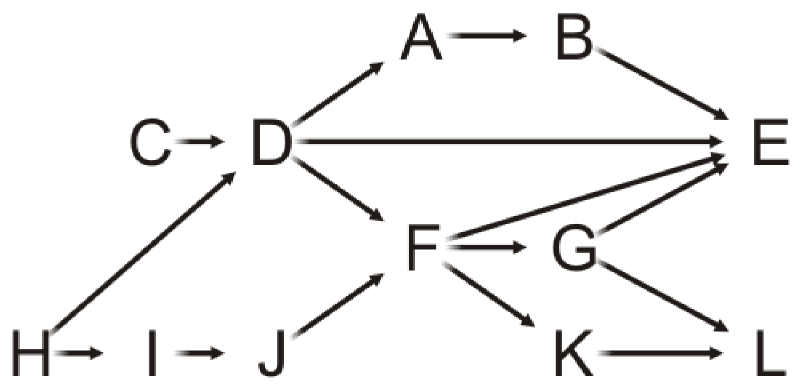
Queue:



The queue is empty

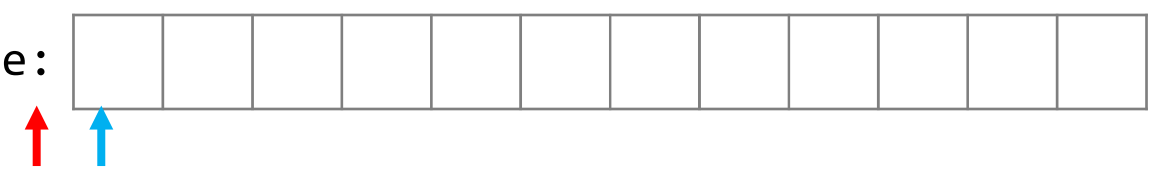
Example

Stepping through the array, push all source vertices into the queue



A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

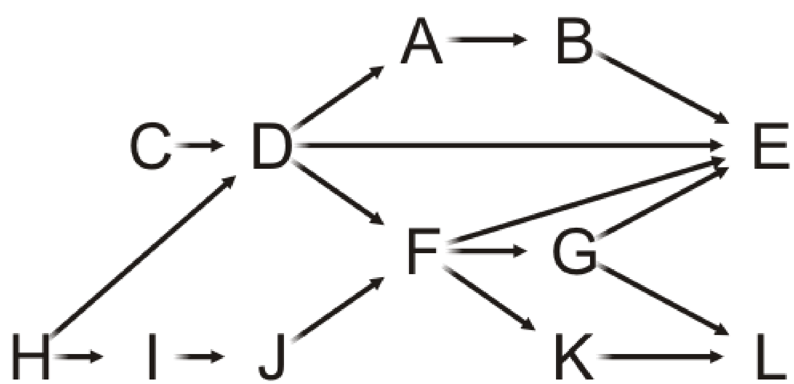
Queue:



The queue is empty

Example

Stepping through the table, push all source vertices into the queue



A	1
B	1
C	**0**
D	2
E	4
F	2
G	1
H	**0**
I	1
J	1
K	1
L	2

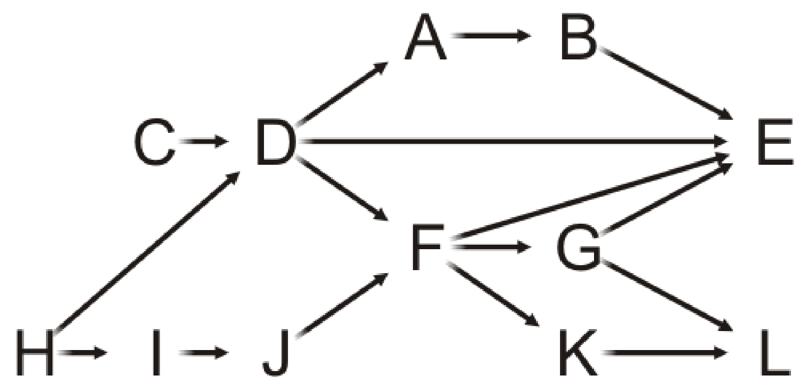
Queue:

C	**H**										

The queue is empty

Example

Pop the front of the queue



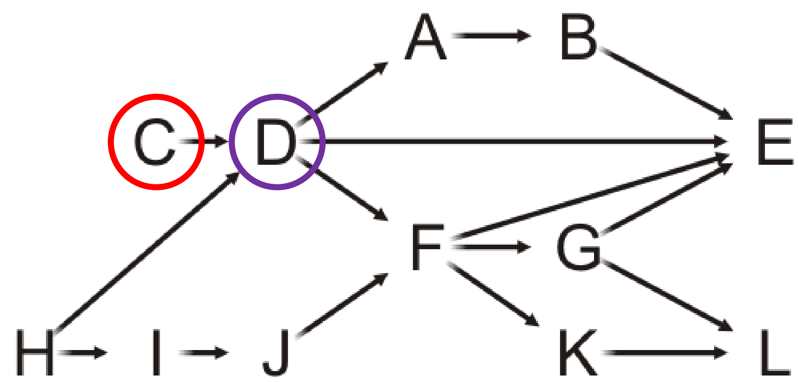
Queue:	C	H										

A	1
B	1
C	0
D	2
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- C has one neighbor: D



Queue:

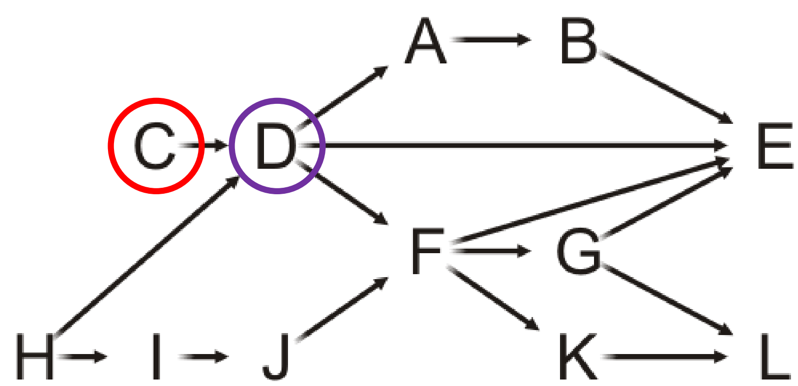
C	H										

A	1
B	1
C	**0**
D	**2**
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- C has one neighbor: D
- Decrement its in-degree



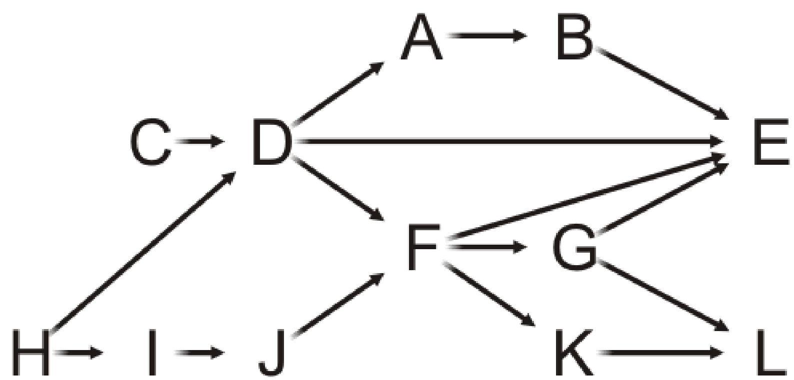
Queue:

C	H										

A	1
B	1
C	**0**
D	**1**
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue



Queue:

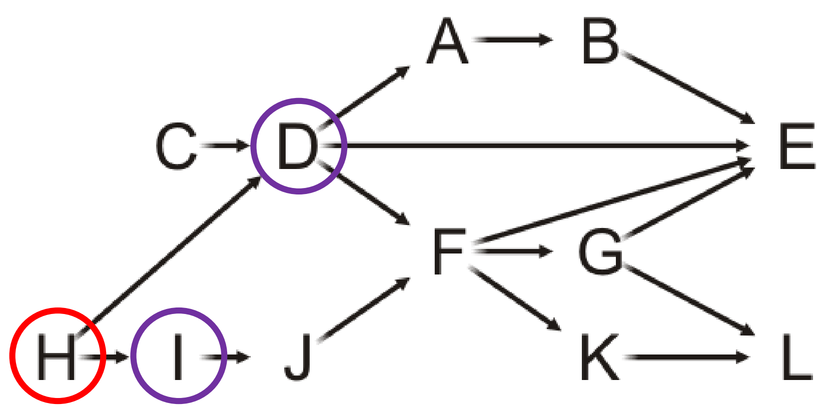
C	H										

A	1
B	1
C	0
D	1
E	4
F	2
G	1
H	0
I	1
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I



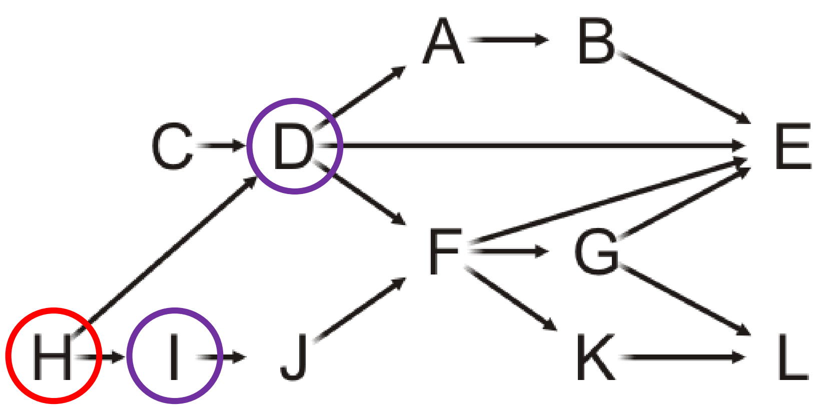
Queue:

C	H										

A	1
B	1
C	0
D	**1**
E	4
F	2
G	1
H	**0**
I	**1**
J	1
K	1
L	2

Example

Pop the front of the queue
- H has two neighbors: D and I
- Decrement their in-degrees



Queue:

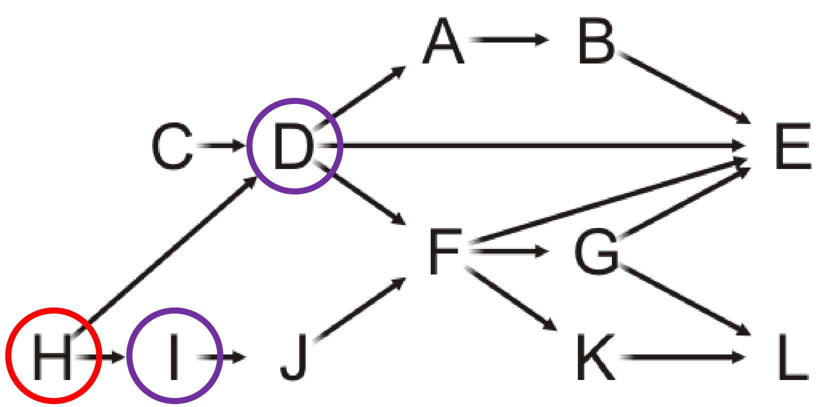
C	H										

A	1
B	1
C	0
D	**0**
E	4
F	2
G	1
H	**0**
I	**0**
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees
  - Both are decremented to zero, so push them onto the queue



Queue:

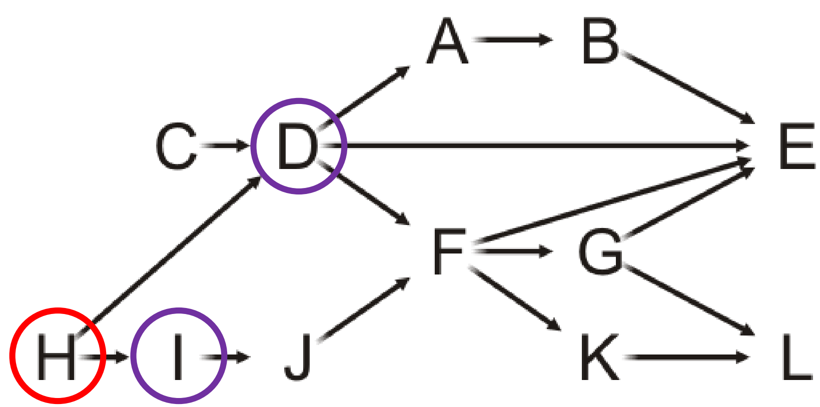
C	H										

A	1
B	1
C	0
D	**0**
E	4
F	2
G	1
H	**0**
I	**0**
J	1
K	1
L	2

Example

Pop the front of the queue

- H has two neighbors: D and I
- Decrement their in-degrees
  - Both are decremented to zero, so push them onto the queue



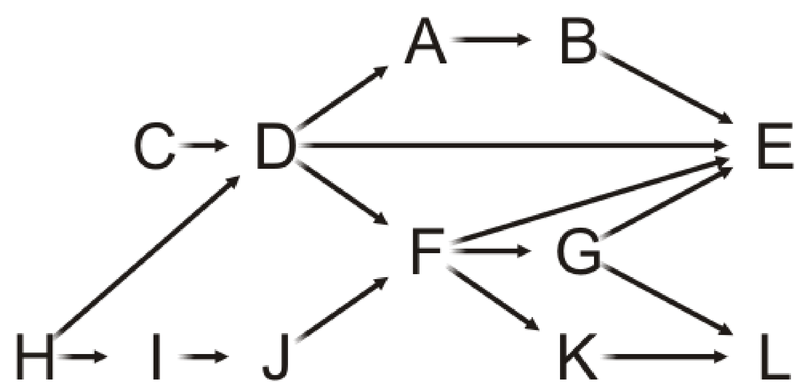
Queue:

C	H	D	I								

A	1
B	1
C	0
D	**0**
E	4
F	2
G	1
H	**0**
I	**0**
J	1
K	1
L	2

Example

Pop the front of the queue



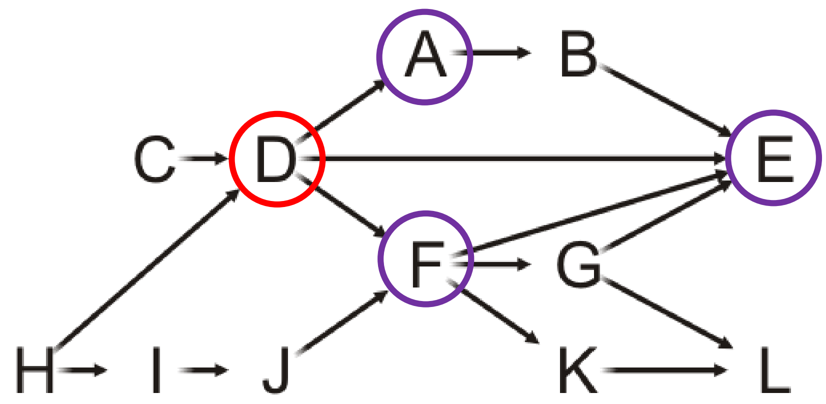
Queue:												
	C	H	D	I								

A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

– D has three neighbors: A, E and F



Queue:

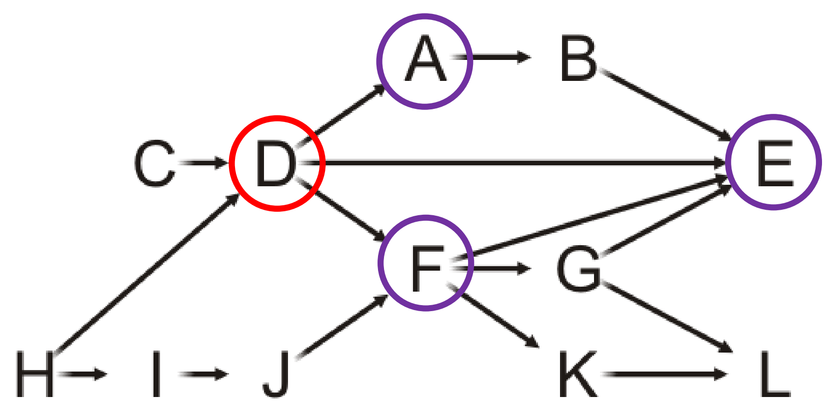
C	H	D	I								

A	1
B	1
C	0
D	0
E	4
F	2
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- D has three neighbors: A, E and F
- Decrement their in-degrees



Queue:

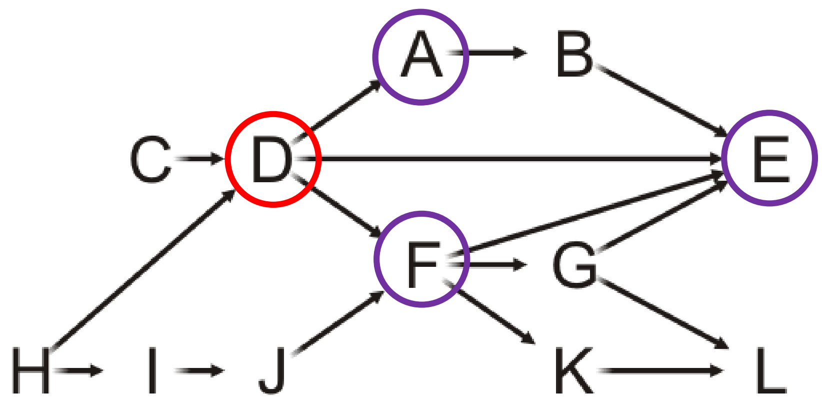
C	H	D	I								

A	**0**
B	1
C	0
D	**0**
E	**3**
F	**1**
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

– D has three neighbors: A, E and F
– Decrement their in-degrees
  - A is decremented to zero, so push it onto the queue



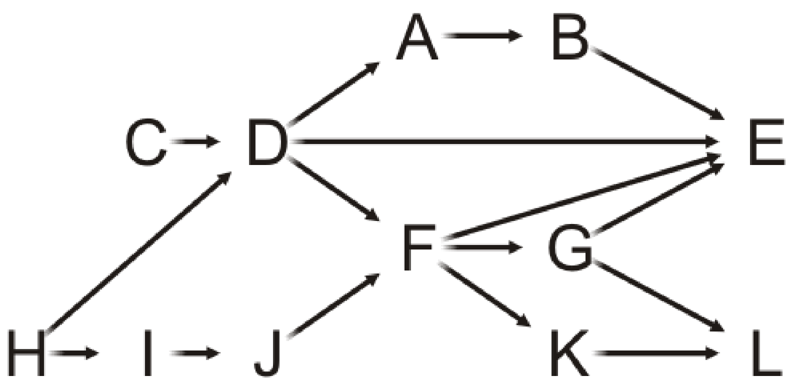
Queue:

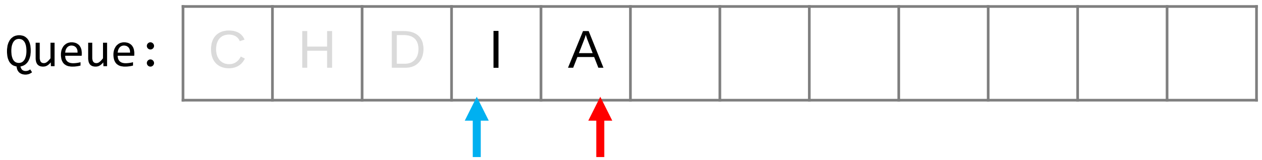
C	H	D	I	A							

A	**0**
B	1
C	0
D	**0**
E	**3**
F	**1**
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue



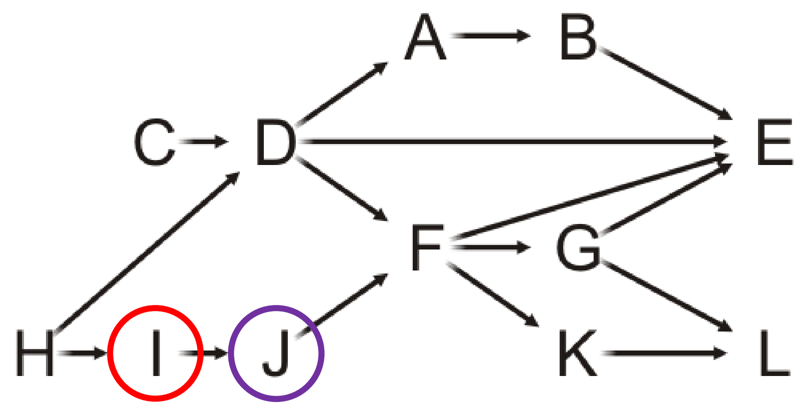


A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	1
K	1
L	2

Example

Pop the front of the queue

- I has one neighbor: J



Queue:

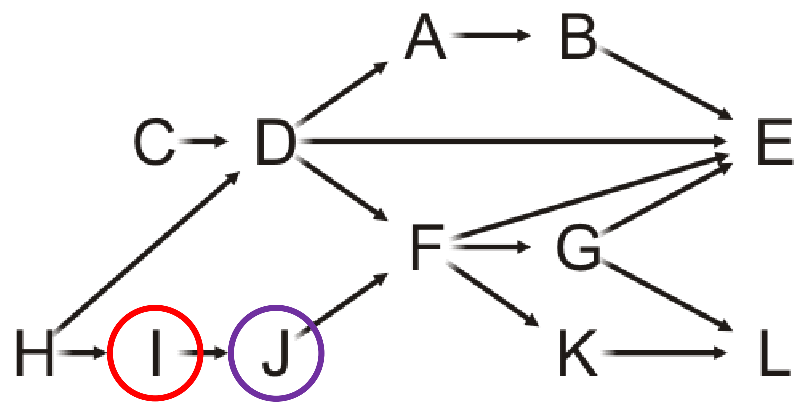
C	H	D	I	A							

A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	**0**
J	**1**
K	1
L	2

Example

Pop the front of the queue

– I has one neighbor: J
– Decrement its in-degree



A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	**0**
J	**0**
K	1
L	2

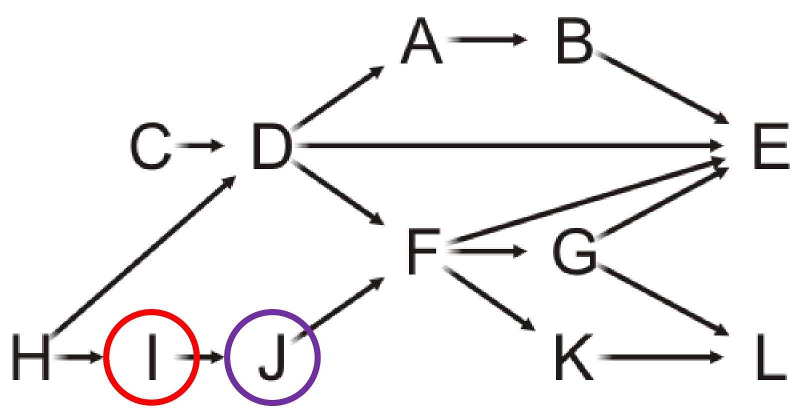
Queue:

C	H	D	I	A							

Example

Pop the front of the queue

- I has one neighbor: J
- Decrement its in-degree
  - J is decremented to zero, so push it onto the queue



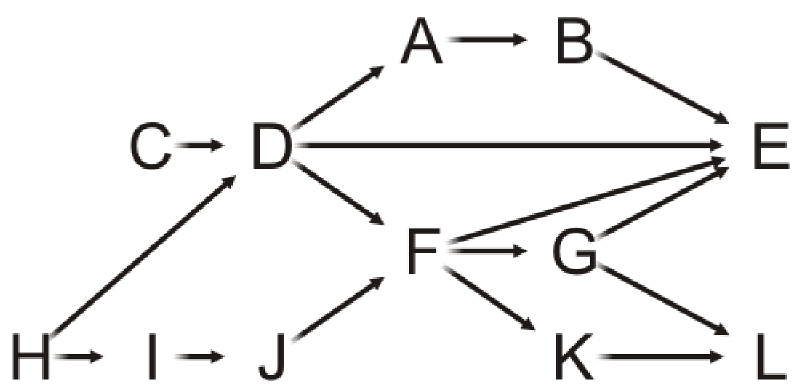
A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	**0**
J	**0**
K	1
L	2

Queue:

C	H	D	I	A	J						

Example

Pop the front of the queue



Queue:

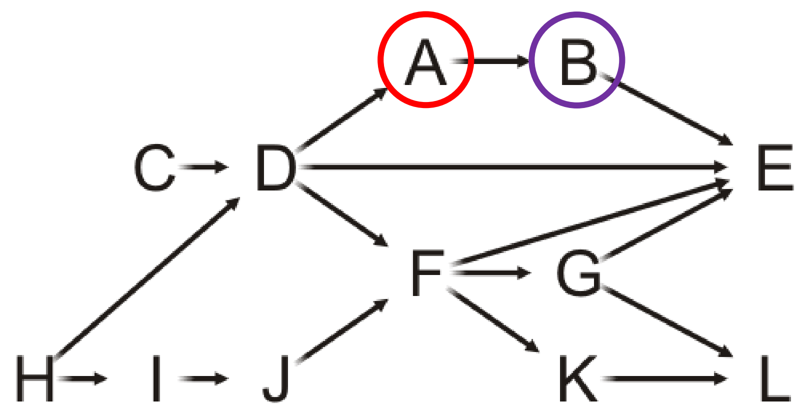
C	H	D	I	A	J						

A	0
B	1
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

– A has one neighbor: B



Queue:

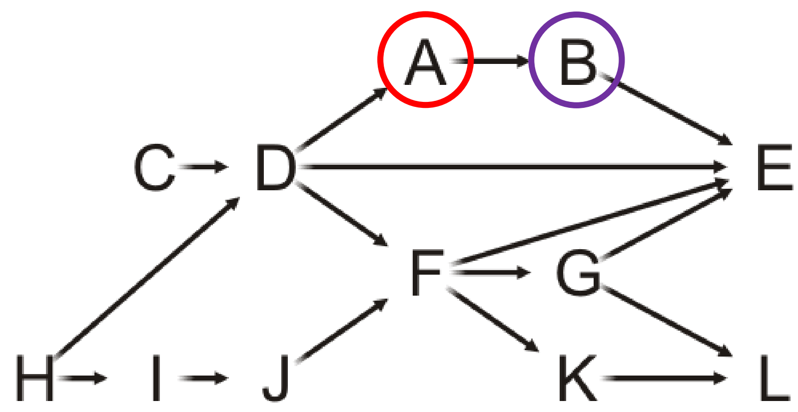
C	H	D	I	A	J						

A	**0**
B	**1**
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B
- Decrement its in-degree



Queue:

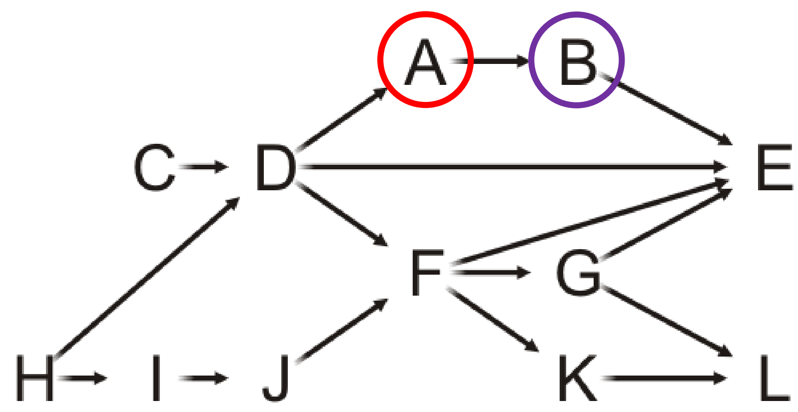
C	H	D	I	A	J						

A	**0**
B	**0**
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

- A has one neighbor: B
- Decrement its in-degree
  - B is decremented to zero, so push it onto the queue



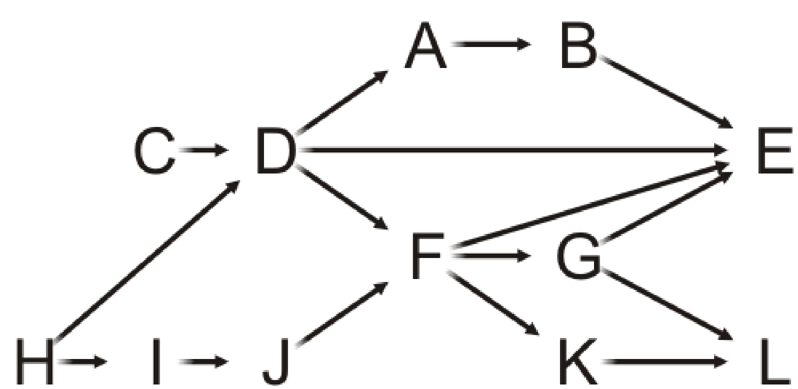
Queue:

C	H	D	I	A	J	B					

A	**0**
B	**0**
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue



Queue:

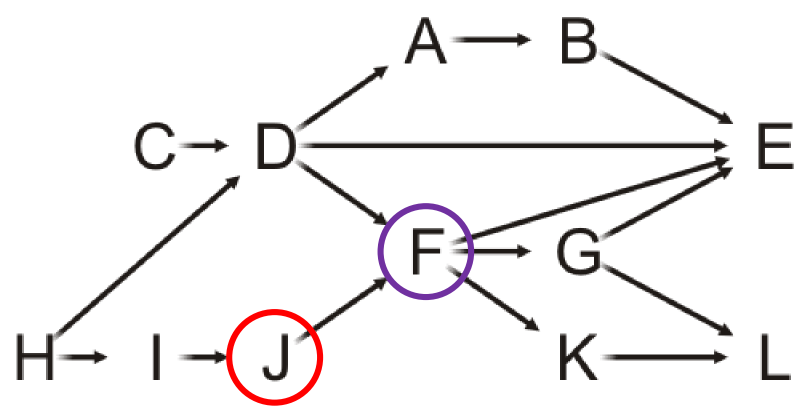
C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	1
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

– J has one neighbor: F



Queue:

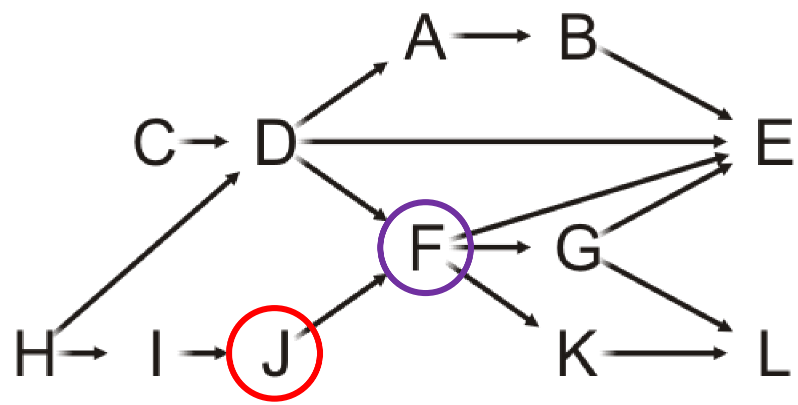
C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	**1**
G	1
H	0
I	0
J	**0**
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F
- Decrement its in-degree



Queue:

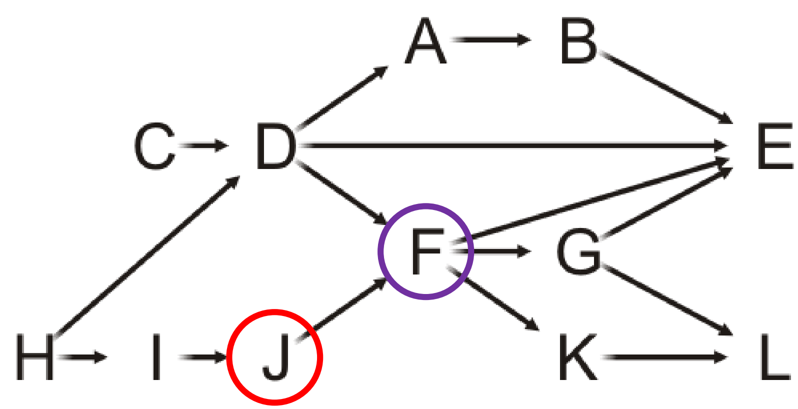
C	H	D	I	A	J	B					

A	0
B	0
C	0
D	0
E	3
F	**0**
G	1
H	0
I	0
J	**0**
K	1
L	2

Example

Pop the front of the queue

- J has one neighbor: F
- Decrement its in-degree
  - F is decremented to zero, so push it onto the queue



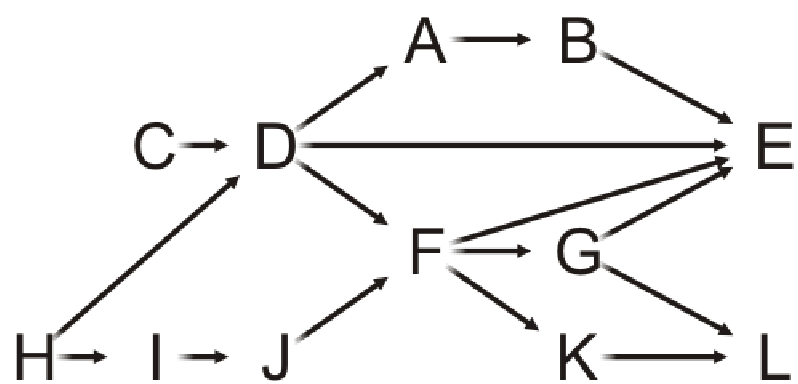
Queue:

C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	3
F	**0**
G	1
H	0
I	0
J	**0**
K	1
L	2

Example

Pop the front of the queue



Queue:

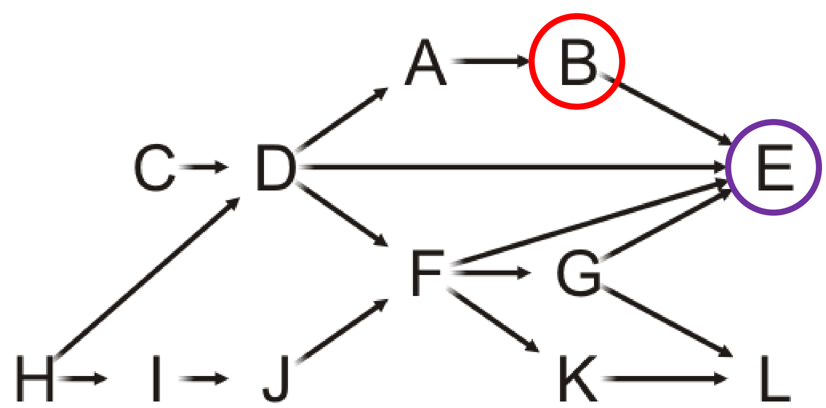
C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	3
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

– B has one neighbor: E



A	0
B	**0**
C	0
D	0
E	**3**
F	0
G	1
H	0
I	0
J	0
K	1
L	2

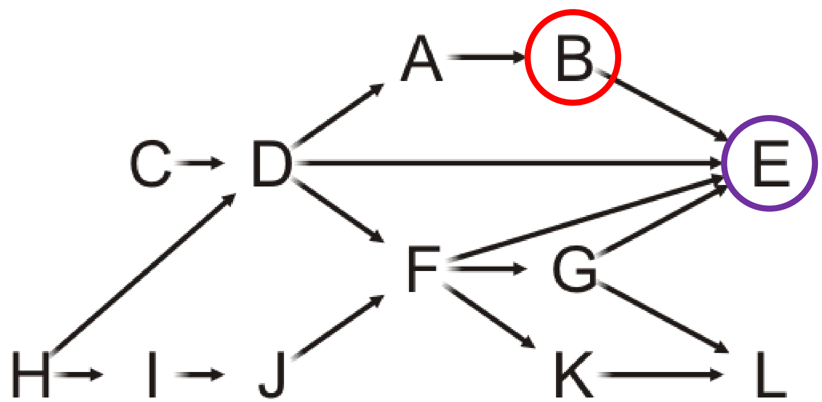
Queue:

C	H	D	I	A	J	B	F				

Example

Pop the front of the queue

- B has one neighbor: E
- Decrement its in-degree



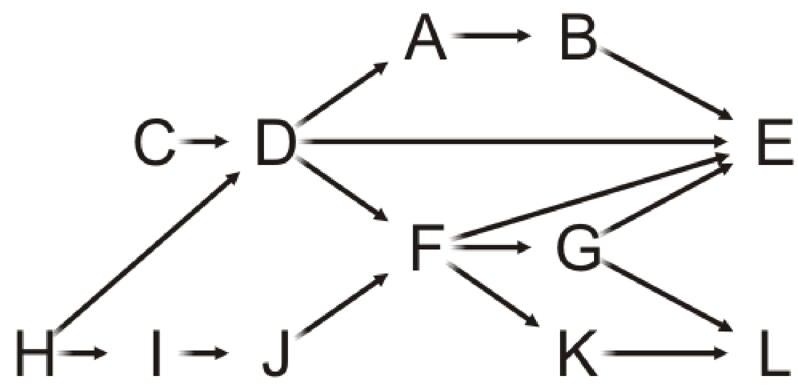
Queue:

C	H	D	I	A	J	B	F				

A	0
B	**0**
C	0
D	0
E	**2**
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue



Queue:

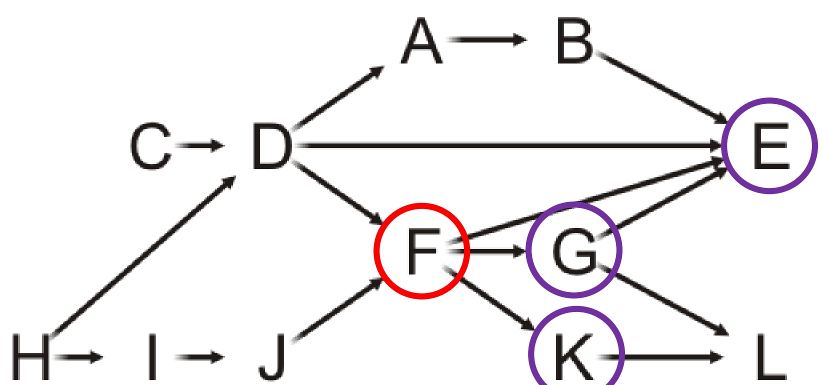
C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	2
F	0
G	1
H	0
I	0
J	0
K	1
L	2

Example

Pop the front of the queue

– F has three neighbors: E, G and K



Queue:

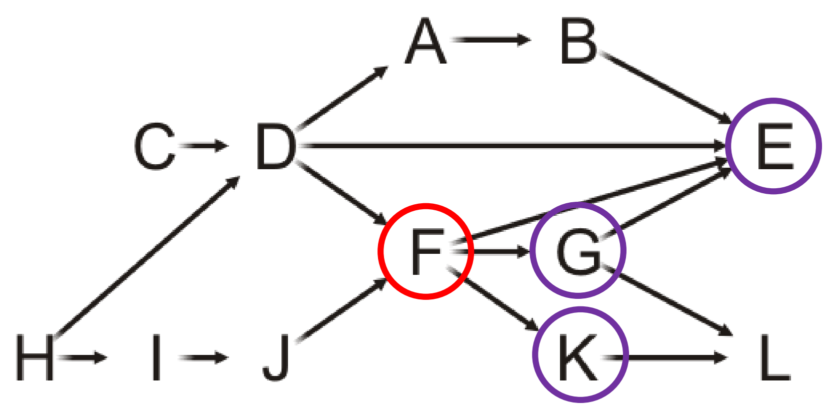
C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	**2**
F	**0**
G	**1**
H	0
I	0
J	0
K	**1**
L	2

Example

Pop the front of the queue

- F has three neighbors: E, G and K
- Decrement their in-degrees



Queue:

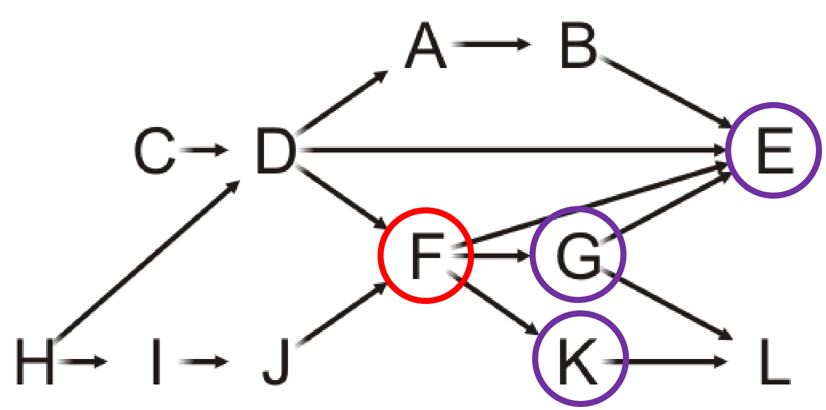
C	H	D	I	A	J	B	F				

A	0
B	0
C	0
D	0
E	**1**
F	**0**
G	**0**
H	0
I	0
J	0
K	**0**
L	2

Example

Pop the front of the queue

- F has three neighbors: E, G and K
- Decrement their in-degrees
  - G and K are decremented to zero,
    so push them onto the queue



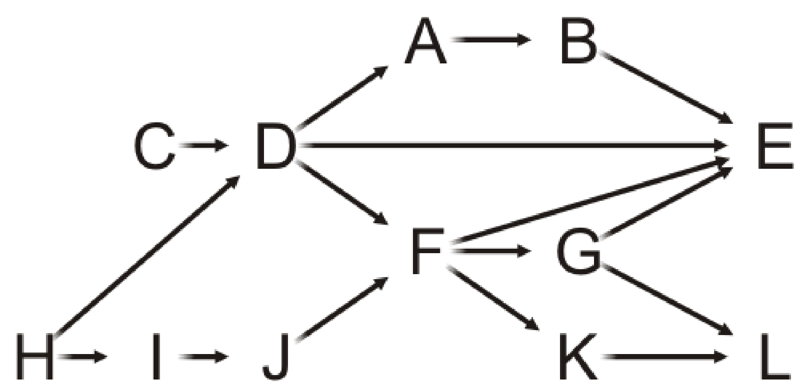
Queue:

C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue



Queue:

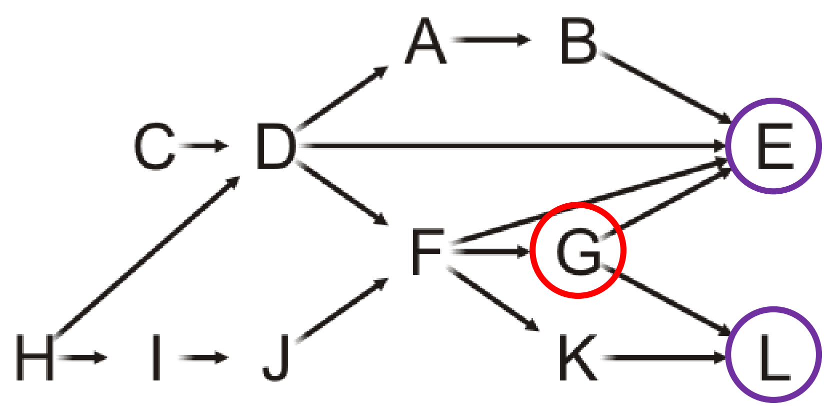
C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	1
F	0
G	0
H	0
I	0
J	0
K	0
L	2

Example

Pop the front of the queue

- G has two neighbors: E and L



Queue:

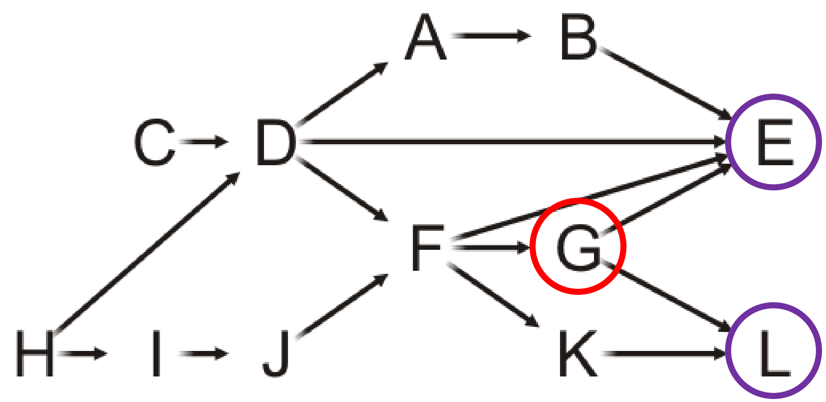
C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	**1**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**2**

Example

Pop the front of the queue

- G has two neighbors: E and L
- Decrement their in-degrees



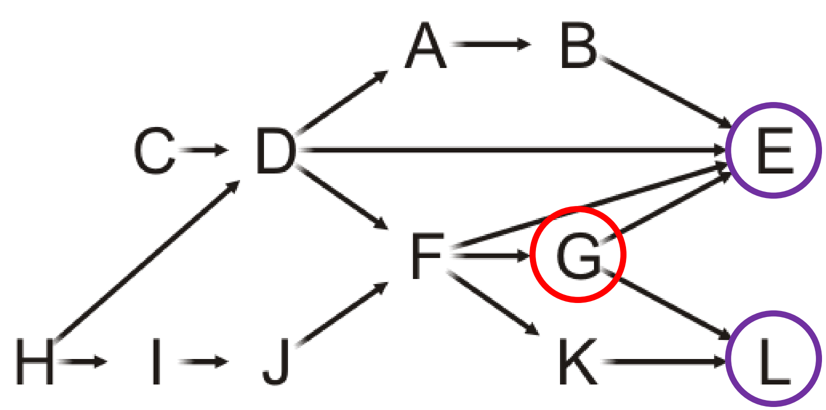
Queue:	C	H	D	I	A	J	B	F	G	K		

A	0
B	0
C	0
D	0
E	**0**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**1**

Example

Pop the front of the queue

- G has two neighbors: E and L
- Decrement their in-degrees
  - E is decremented to zero, so push it onto the queue



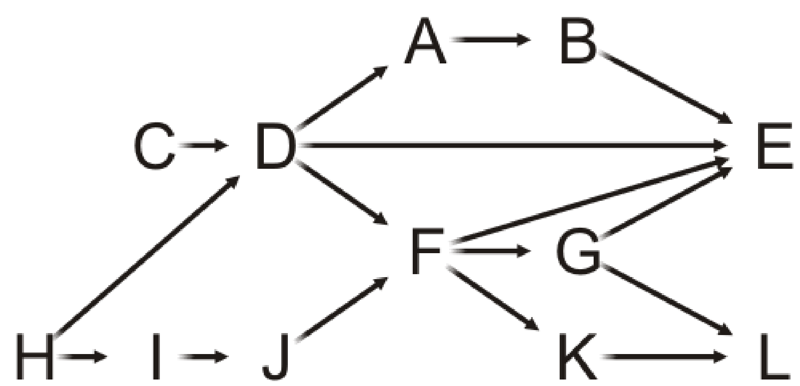
Queue:

C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	**0**
F	0
G	**0**
H	0
I	0
J	0
K	0
L	**1**

Example

Pop the front of the queue



Queue:

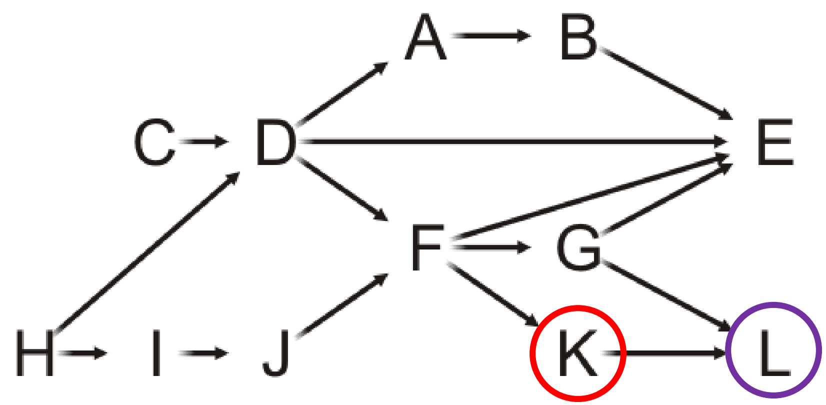
C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	1

Example

Pop the front of the queue

– K has one neighbors: L



Queue:

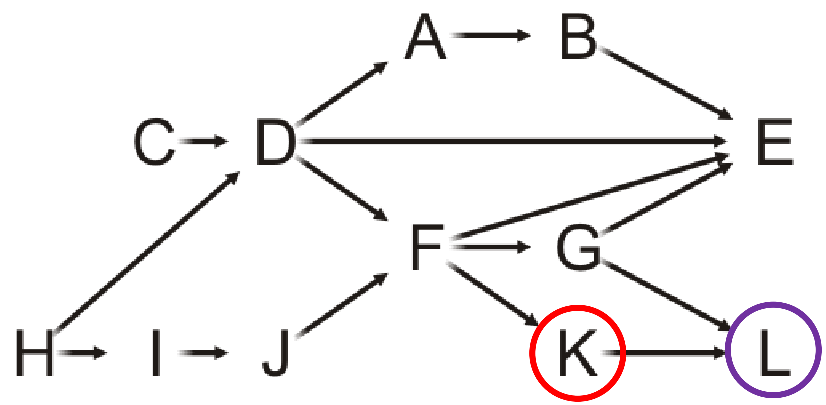
C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**1**

Example

Pop the front of the queue

- K has one neighbors: L
- Decrement its in-degree



Queue:

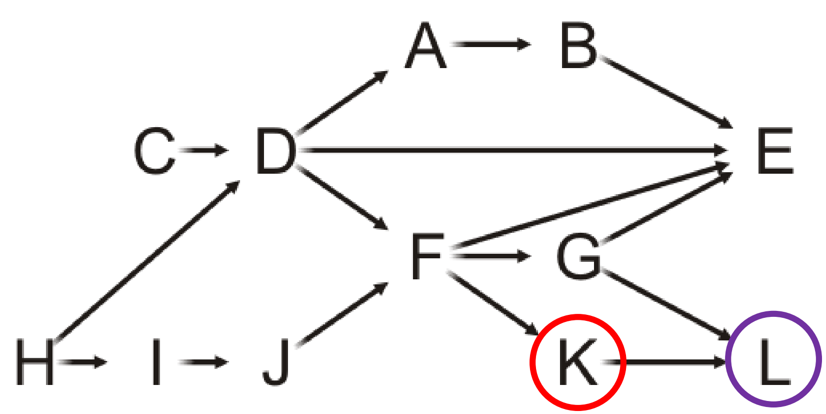
C	H	D	I	A	J	B	F	G	K	E	

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**0**

Example

Pop the front of the queue

- K has one neighbors: L
- Decrement its in-degree
  - L is decremented to zero, so push it onto the queue



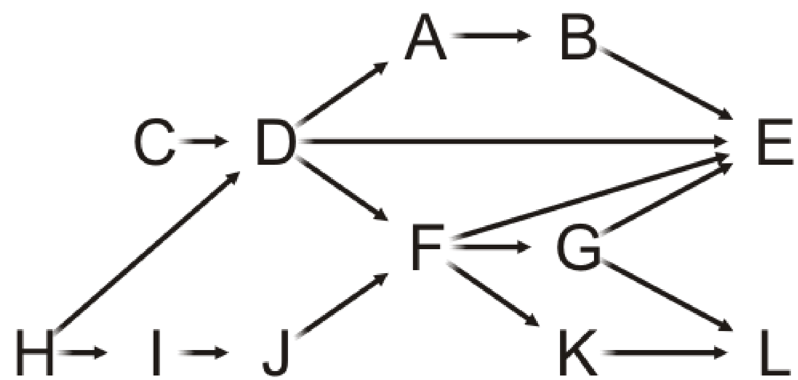
Queue:

C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	**0**
L	**0**

Example

Pop the front of the queue



Queue:

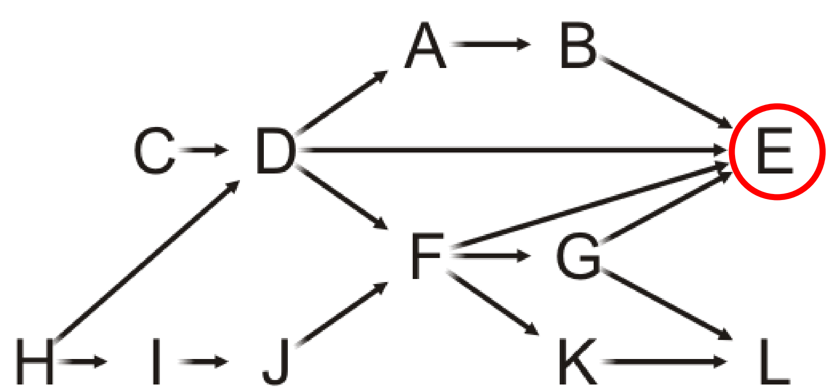
C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue

- E has no neighbors—it is a *sink*



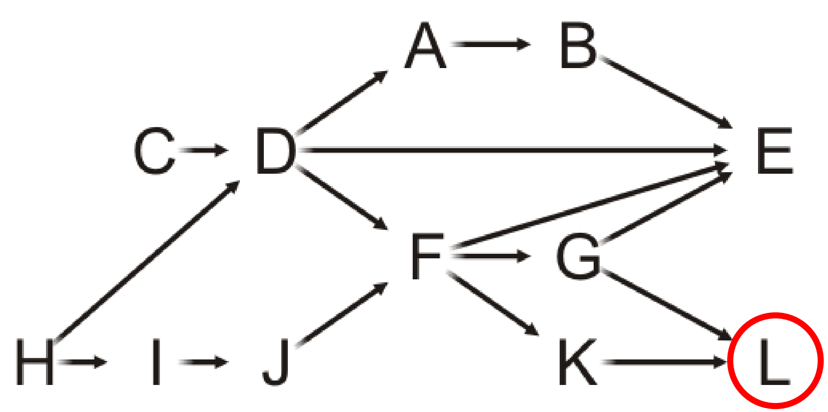
Queue:

C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	**0**
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

Pop the front of the queue



A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

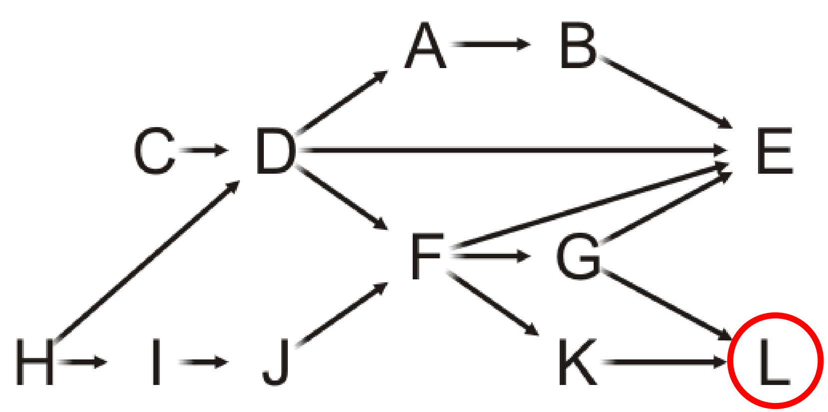
Queue:

C	H	D	I	A	J	B	F	G	K	E	L

Example

Pop the front of the queue

- L has no neighbors—it is also a *sink*



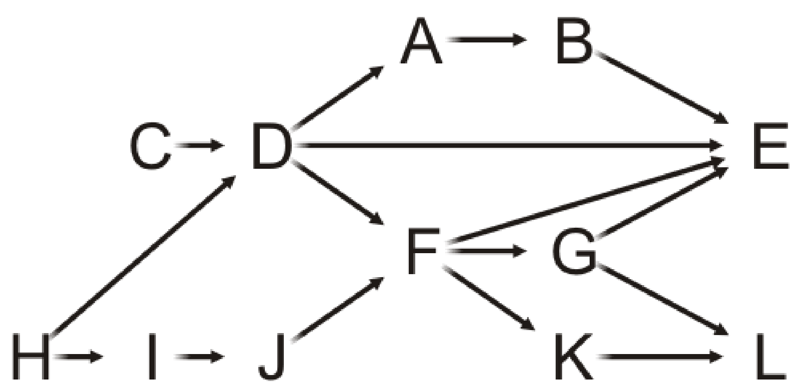
Queue:

C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	**0**

Example

The queue is empty, so we are done



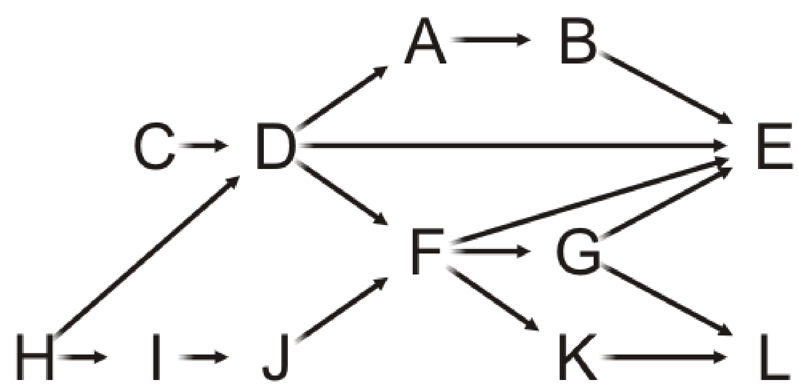
Queue:

C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

The array used for the queue stores the topological sort



C	H	D	I	A	J	B	F	G	K	E	L

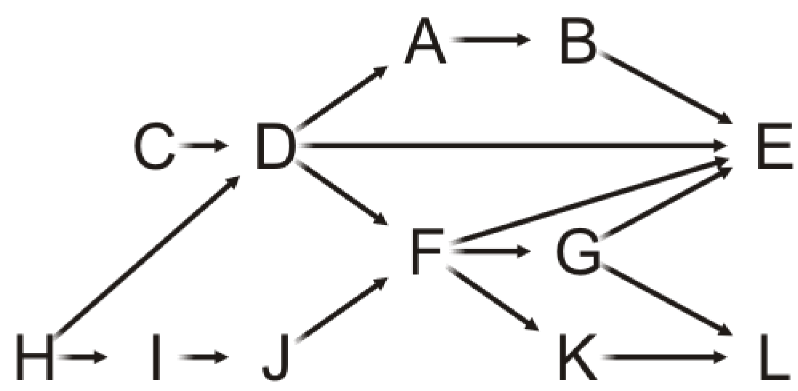
A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

Example

The array used for the queue stores the topological sort

– Note the difference in order from our previous sort?

C, H, D, A, B, I, J, F, G, E, K, L



C	H	D	I	A	J	B	F	G	K	E	L

A	0
B	0
C	0
D	0
E	0
F	0
G	0
H	0
I	0
J	0
K	0
L	0

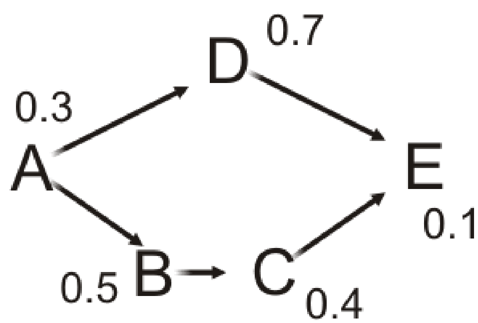
Outline

- Topological sorting
  - Definitions
  - Algorithm
- Finding the critical path

Critical path

Suppose each task has a performance time associated with it

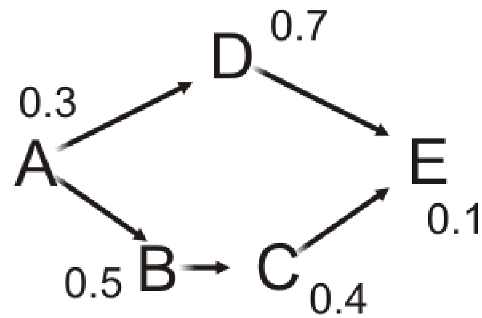
– If the tasks are performed serially, the time required to complete the last task equals to the sum of the individual task times



– These tasks require $0.3 + 0.7 + 0.5 + 0.4 + 0.1 = 2.0\ \mathrm{s}$ to execute serially

Critical path

In many cases, however, we could perform tasks in parallel

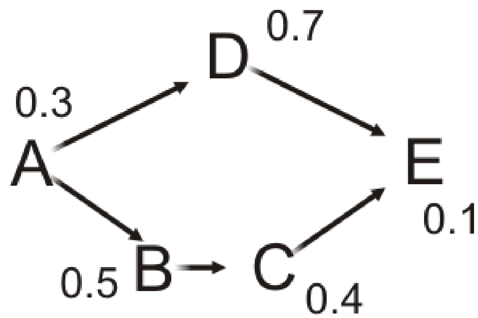


- Computer tasks can be executed in parallel (multi-processing)
- Different tasks can be completed by different teams in a company

Critical path

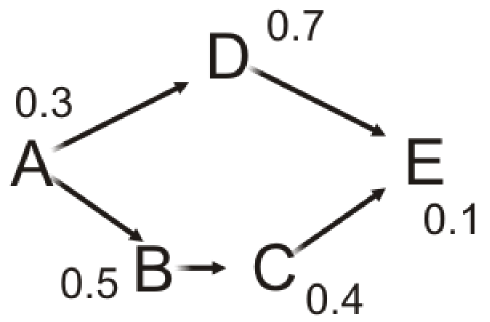
Suppose Task A completes

- We can now execute Tasks B and D in parallel



Critical path

Note that, Task E cannot execute until Task C completes, and Task C cannot execute until Task B completes



– The least time in which these five tasks can be completed is

$$0.3 + 0.5 + 0.4 + 0.1 = 1.3 \text{ s}$$

– This is called the *critical time of all tasks*

– The path (A, B, C, E) is said to be the *critical path*

Critical path

The *critical time* of each task is the earliest time that it could be completed after the start of execution

The *critical path* is the sequence of tasks determining the minimum time needed to complete the project

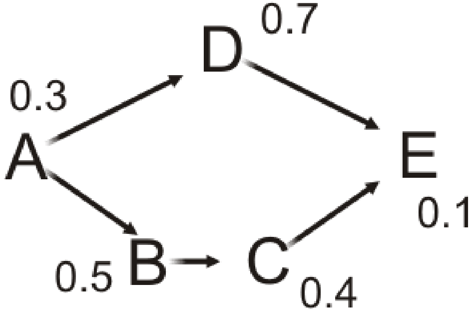
- If a task on the critical path is delayed, the entire project will be delayed

Finding the critical path

Tasks that have no prerequisites have a critical time equal to the time it takes to complete that task

For tasks that depend on others, the critical time will be:

- The maximum critical time that it takes to complete a prerequisite
- Plus the time it takes to complete this task



In this example, the critical times are:

- Task A completes in 0.3 s
- Task B must wait for A and completes after 0.8 s
- Task D must wait for A and completes after 1.0 s
- Task C must wait for B and completes after 1.2 s
- Task E must wait for both C and D, and completes after

$$\max(1.0, 1.2) + 0.1 = 1.3 \text{ s}$$

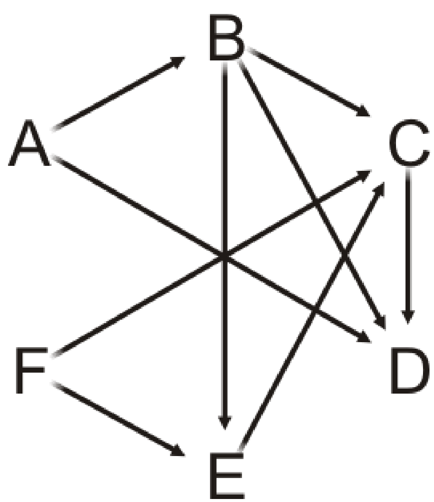
Finding the critical path

To find the critical time/path, we run topological sorting and require the following additional information:

- We must know the execution time of each task
- We will have to record the critical time for each task
  - Initialize these to zero
- We will need to know the previous task with the longest critical time to determine the critical path
  - Set these to null

Finding the critical path

Suppose we have the following times for the tasks



Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	Ø
B	1	6.1	0.0	Ø
C	3	4.7	0.0	Ø
D	3	8.1	0.0	Ø
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

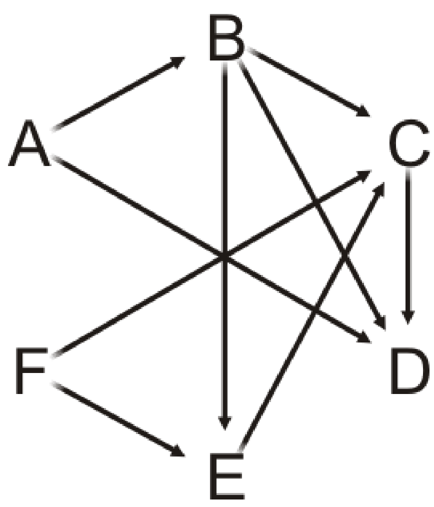
Finding the critical path

Each time we pop a vertex v, in addition to what we already do:

- For v, add the task time onto the critical time for that vertex:
  - That is the critical time for v
- For each adjacent vertex w:
  - If the critical time for v is greater than the currently stored critical time for w
    - Update the critical time with the critical time for v
    - Set the previous pointer to the vertex v

Finding the critical path

So we initialize the queue with those vertices with in-degree zero



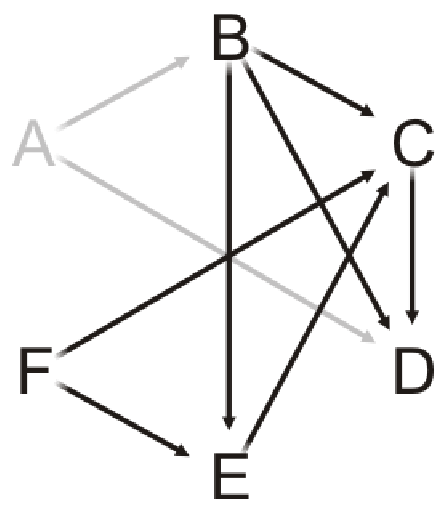
Queue

A	**F**		

Task	In-degree	Task Time	Critical Time	Previous Task
A	**0**	5.2	0.0	Ø
B	1	6.1	0.0	Ø
C	3	4.7	0.0	Ø
D	3	8.1	0.0	Ø
E	2	9.5	0.0	Ø
F	**0**	17.1	0.0	Ø

Finding the critical path

Pop Task A and update its critical time 0.0 + 5.2 = 5.2



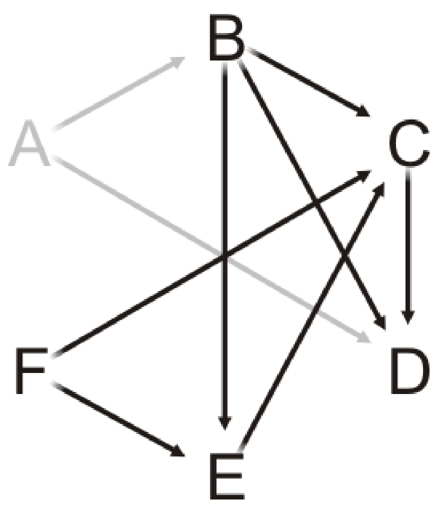
Queue

F			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	0.0	Ø
B	1	6.1	0.0	Ø
C	3	4.7	0.0	Ø
D	3	8.1	0.0	Ø
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

Finding the critical path

Pop Task A and update its critical time 0.0 + 5.2 = 5.2



Queue

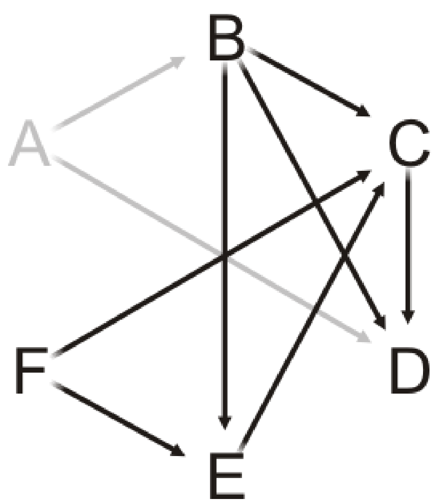
F			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	**5.2**	Ø
B	1	6.1	0.0	Ø
C	3	4.7	0.0	Ø
D	3	8.1	0.0	Ø
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

Finding the critical path

For each neighbor of Task A:

- Decrement the in-degree, push if necessary, and check if we must update the critical time



Queue

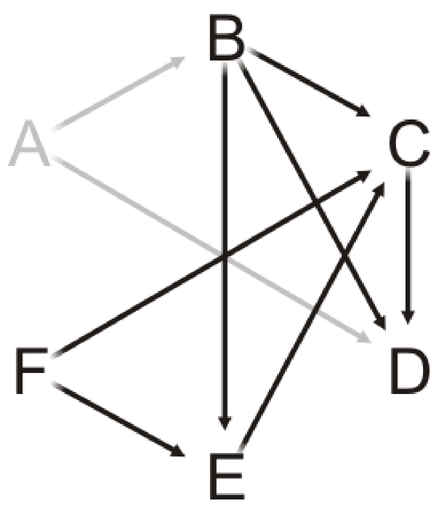
F			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	1	6.1	0.0	Ø
C	3	4.7	0.0	Ø
D	3	8.1	0.0	Ø
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

Finding the critical path

For each neighbor of Task A:

- Decrement the in-degree, push if necessary, and check if we must update the critical time



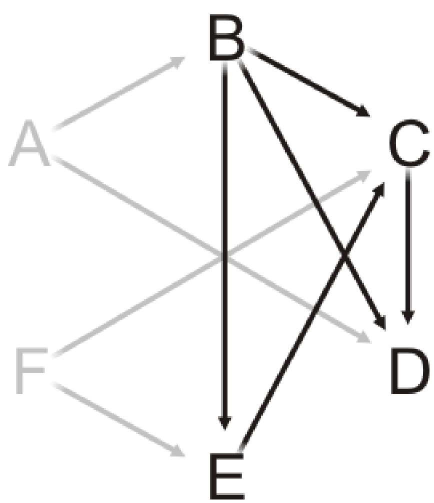
Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	**0**	6.1	**5.2**	**A**
C	3	4.7	0.0	Ø
D	**2**	8.1	**5.2**	**A**
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

Queue

F	**B**		

Finding the critical path

Pop Task F and update its critical time 0.0 + 17.1 = 17.1



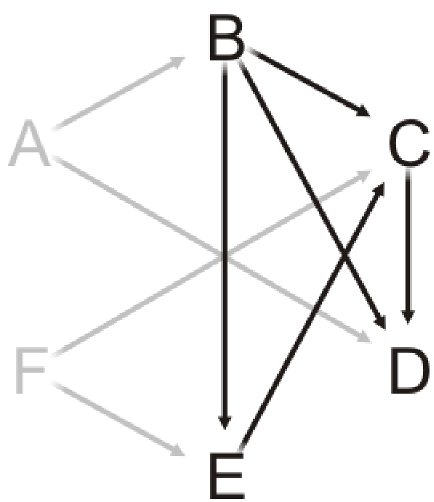
Queue

B			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	5.2	A
C	3	4.7	0.0	Ø
D	2	8.1	5.2	A
E	2	9.5	0.0	Ø
F	0	17.1	0.0	Ø

Finding the critical path

Pop Task F and update its critical time 0.0 + 17.1 = 17.1



Queue

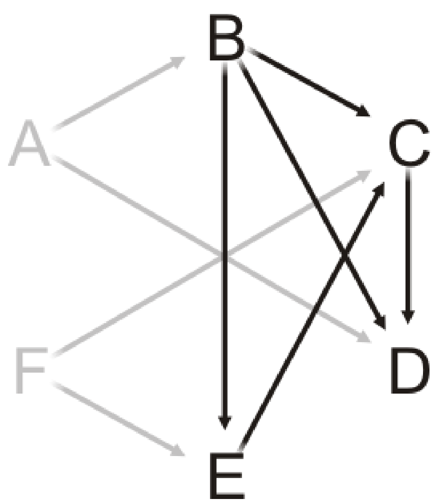
B			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	5.2	A
C	3	4.7	0.0	Ø
D	2	8.1	5.2	A
E	2	9.5	0.0	Ø
F	0	17.1	**17.1**	Ø

Finding the critical path

For each neighbor of Task F:

- Decrement the in-degree, push if necessary, and check if we must update the critical time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	5.2	A
C	3	4.7	0.0	Ø
D	2	8.1	5.2	A
E	2	9.5	0.0	Ø
F	0	17.1	17.1	Ø

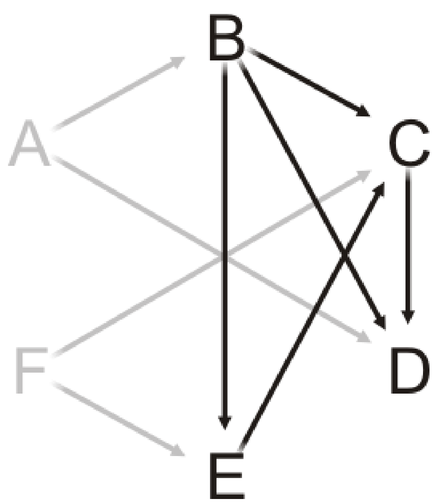
Queue

B			

Finding the critical path

For each neighbor of Task F:

– Decrement the in-degree, push if necessary, and check if we must update the critical time



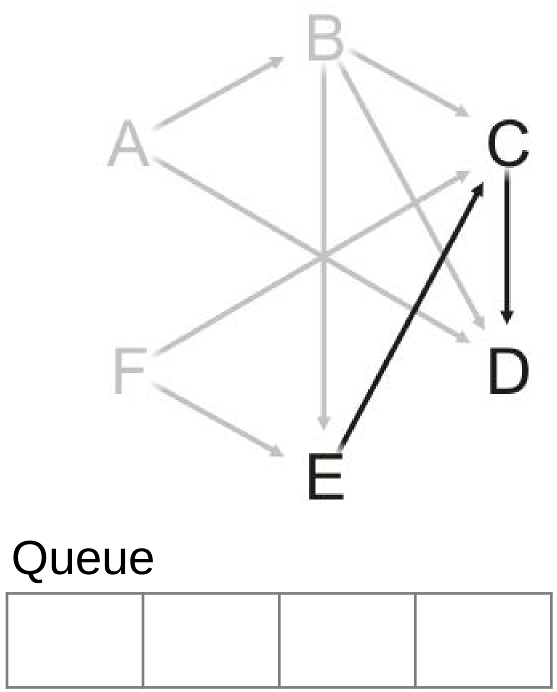
Queue

B			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	5.2	A
C	**2**	4.7	**17.1**	**F**
D	2	8.1	5.2	A
E	**1**	9.5	**17.1**	**F**
F	0	17.1	17.1	Ø

Finding the critical path

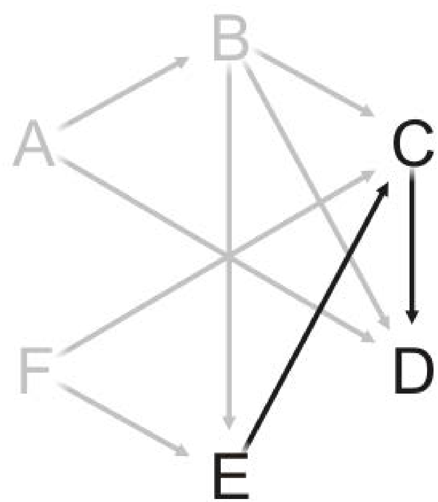
Pop Task B and update its critical time 5.2 + 6.1 = 11.3



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	5.2	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task B and update its critical time 5.2 + 6.1 = 11.3



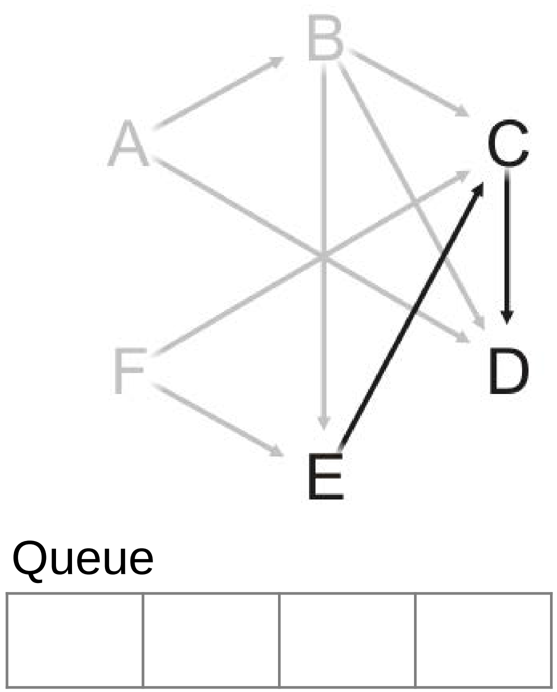
Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	**11.3**	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task B:

– Decrement the in-degree, push if necessary, and check if we must update the critical time

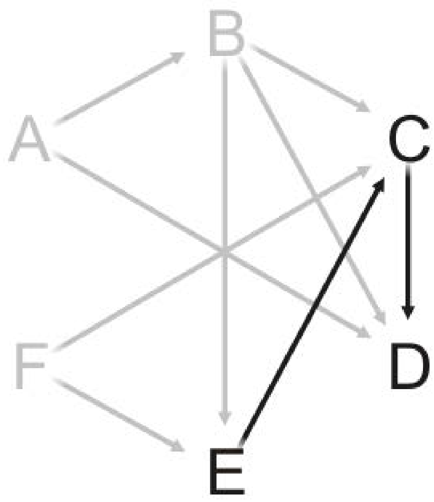


Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	2	4.7	17.1	F
D	2	8.1	5.2	A
E	1	9.5	17.1	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task F:

- Decrement the in-degree, push if necessary, and check if we must update the critical time
- Both C and E are waiting on F



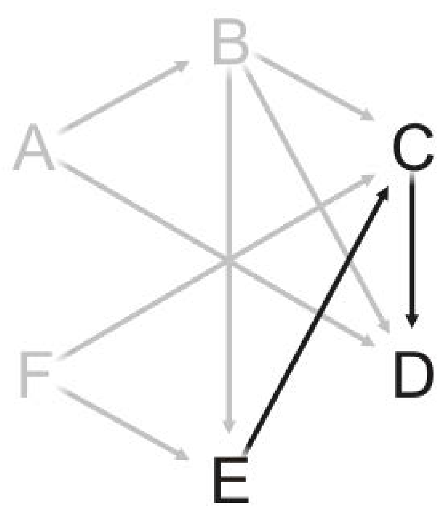
Queue

E			

Task	**In-degree**	**Task Time**	**Critical Time**	**Previous Task**
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	**1**	4.7	17.1	F
D	**1**	8.1	**11.3**	**B**
E	**0**	9.5	17.1	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task E and update its critical time 17.1 + 9.5 = 26.6

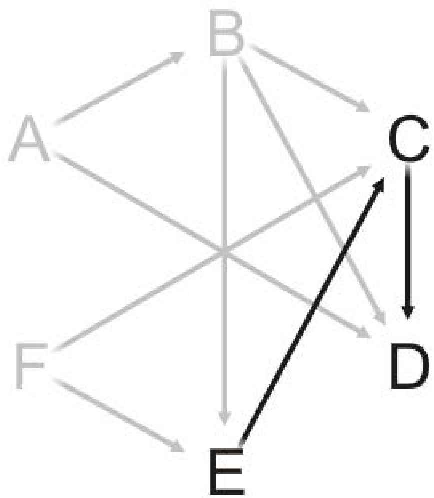


Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	17.1	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task E and update its critical time 17.1 + 9.5 = 26.6



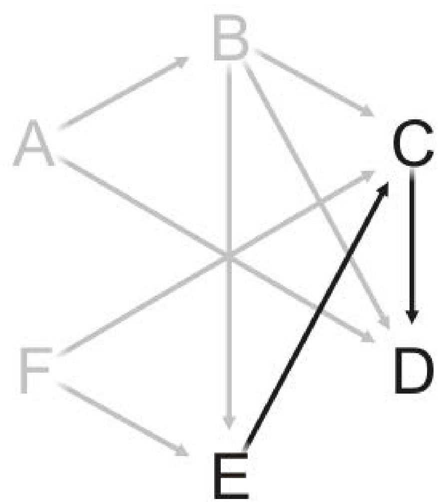
Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	**26.6**	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task E:

- Decrement the in-degree, push if necessary, and check if we must update the critical time



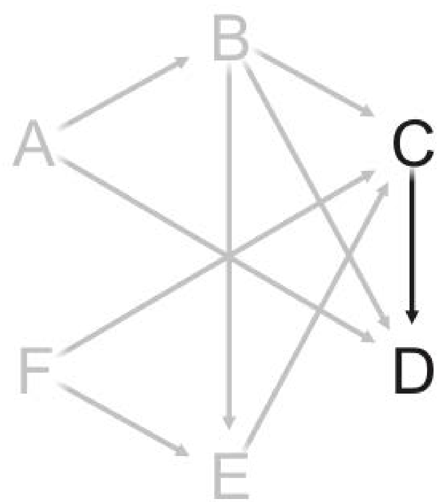
Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	1	4.7	17.1	F
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task E:

– Decrement the in-degree, push if necessary, and check if we must update the critical time



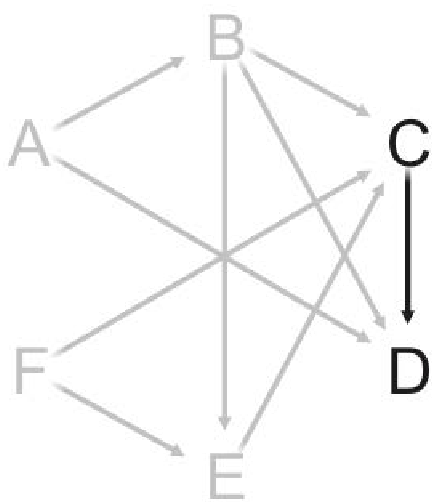
Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	**0**	4.7	**26.6**	**E**
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Queue

C			

Finding the critical path

Pop Task C and update its critical time 26.6 + 4.7 = 31.3

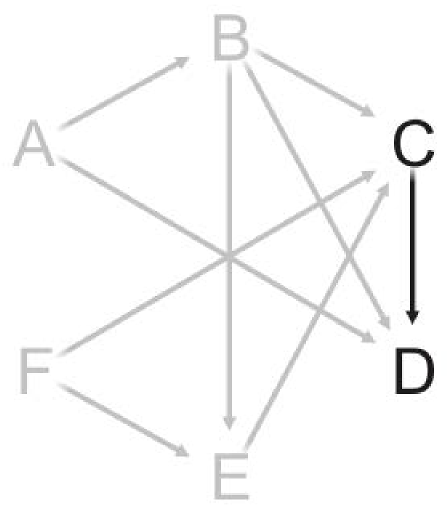


Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	26.6	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task C and update its critical time 26.6 + 4.7 = 31.3



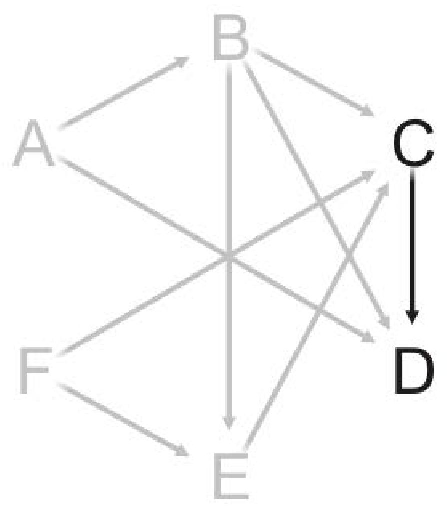
Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	**31.3**	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task C:

– Decrement the in-degree, push if necessary, and check if we must update the critical time



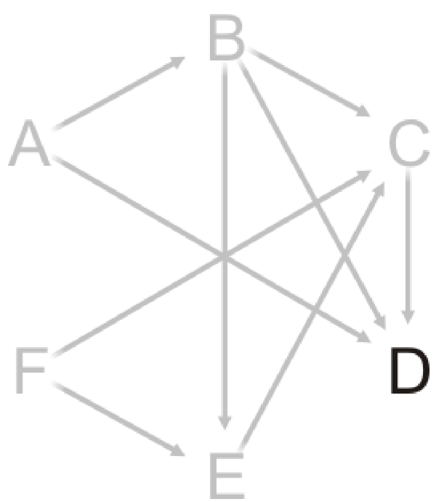
Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	1	8.1	11.3	B
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

For each neighbor of Task C:

- Decrement the in-degree, push if necessary, and check if we must update the critical time



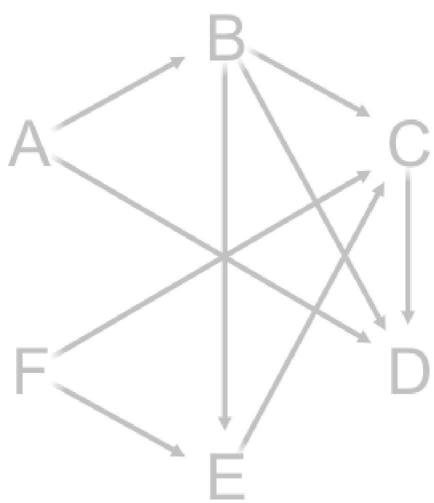
Queue

D			

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	**0**	8.1	**31.3**	**C**
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task D and update its critical time 31.3 + 8.1 = 39.4

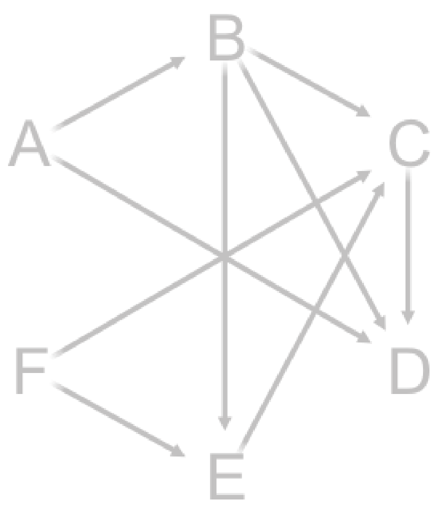


Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	31.3	C
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

Pop Task D and update its critical time 31.3 + 8.1 = 39.4

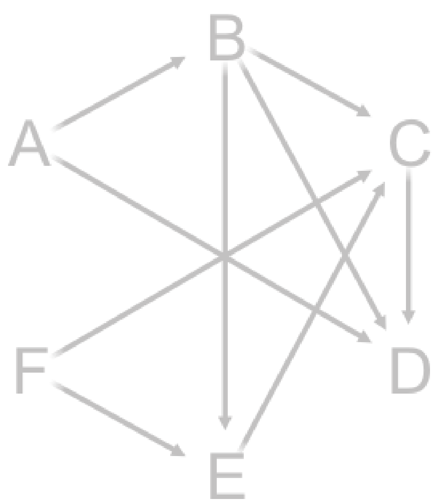


Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	**39.4**	C
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

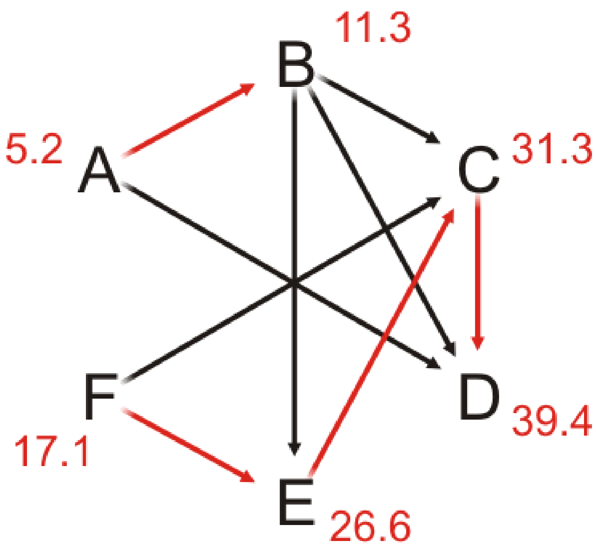
Task D has no neighbors and the queue is empty
- We are done



Queue

Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

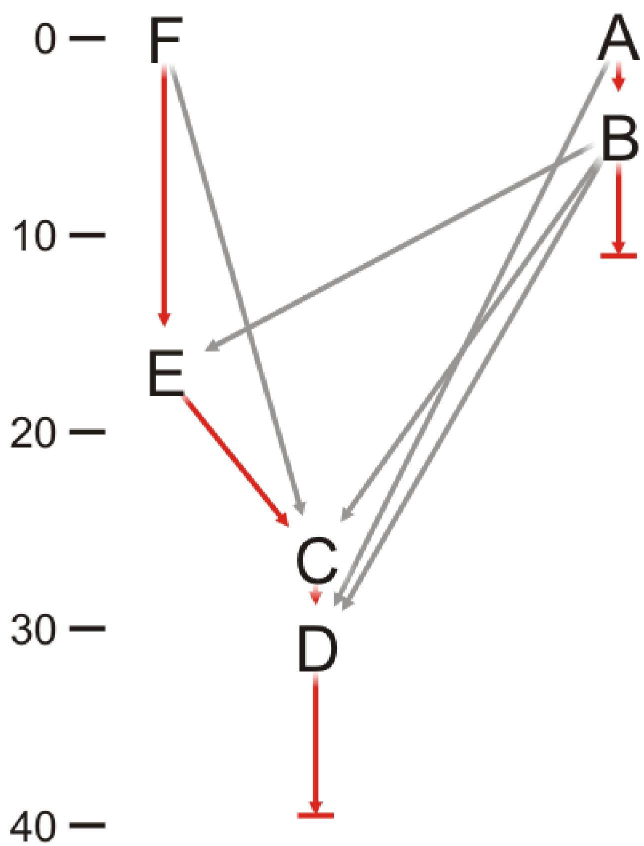
Finding the critical path



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

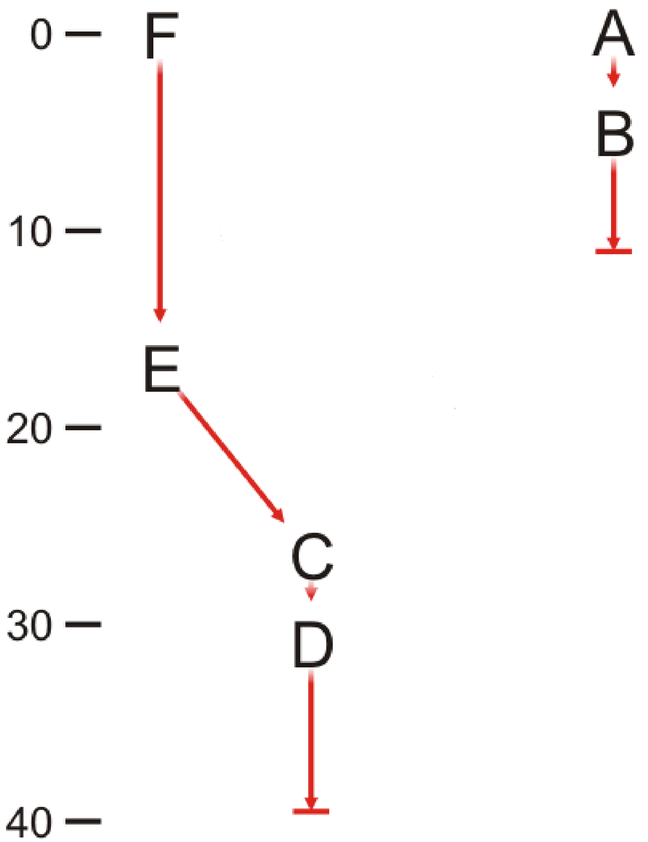
We can also plot the completing of the tasks in time



Task	In-degree	Task Time	Critical Time	Previous Task
A	0	5.2	5.2	Ø
B	0	6.1	11.3	A
C	0	4.7	31.3	E
D	0	8.1	39.4	C
E	0	9.5	26.6	F
F	0	17.1	17.1	Ø

Finding the critical path

Incidentally, the task and previous task defines a forest using the parental tree data structure



Task	Previous Task
A	Ø
B	A
C	E
D	C
E	F
F	Ø

Summary

In this topic, we have discussed topological sorts

- Sorting of elements in a DAG
- Implementation
  - A table of in-degrees
  - Select that vertex which has current in-degree zero
- We defined critical paths
  - The implementation requires only a few more table entries