CSC 106
Graph Algorithms

Summer 2016

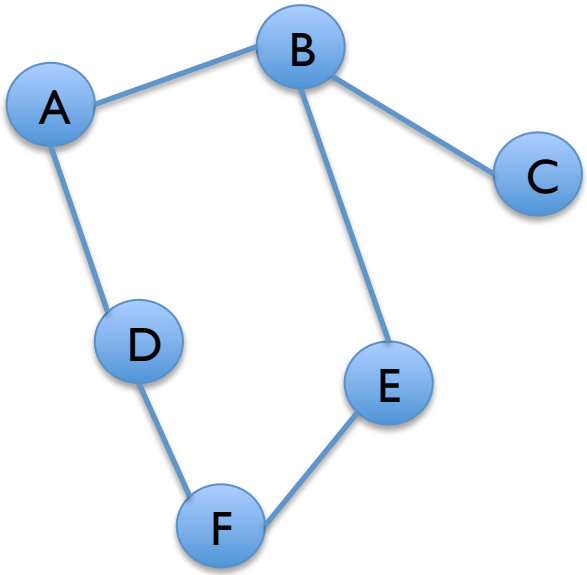
Announcements

- EXAM 2
  - Come get your exam 2, if you did not get it yet
  - Questions?

- Assignment 4 and 5 due ...

Graphs

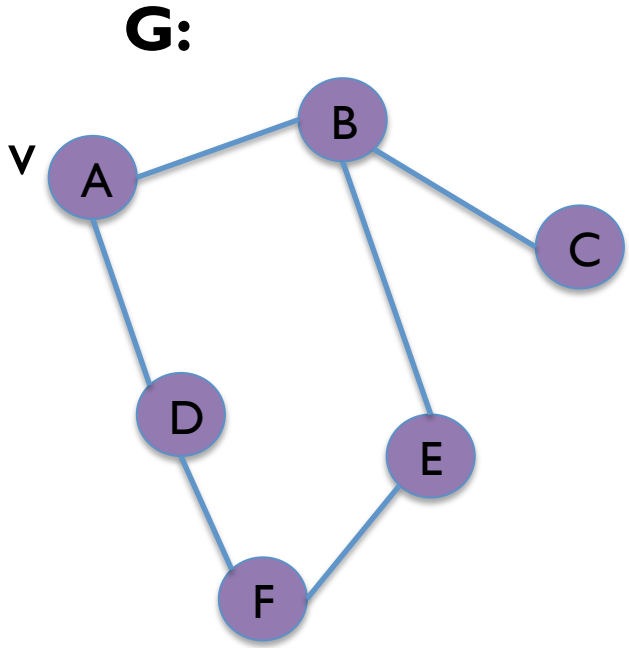
- A graph is:
  - a collection of vertices,
  - connected by edges.



Graphs

- Graph **G** has:
  - a collection of vertices **v**:

    {A, B, C, D, E, F}



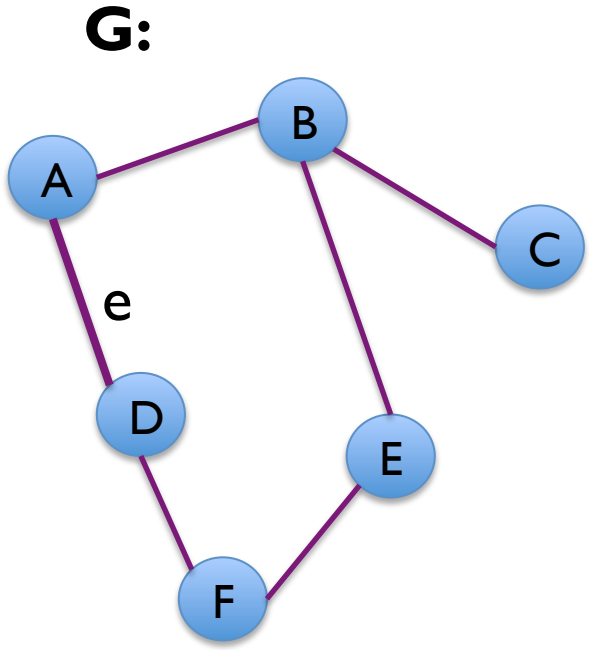
Graphs

- Graph **G** has:
  - a collection of vertices **v**:

    {A, B, C, D, E, F}

  - connected by edges **e**:

    {(A,B), (A,D), (B,C), (B,E),
    (D,F), (E,F)}



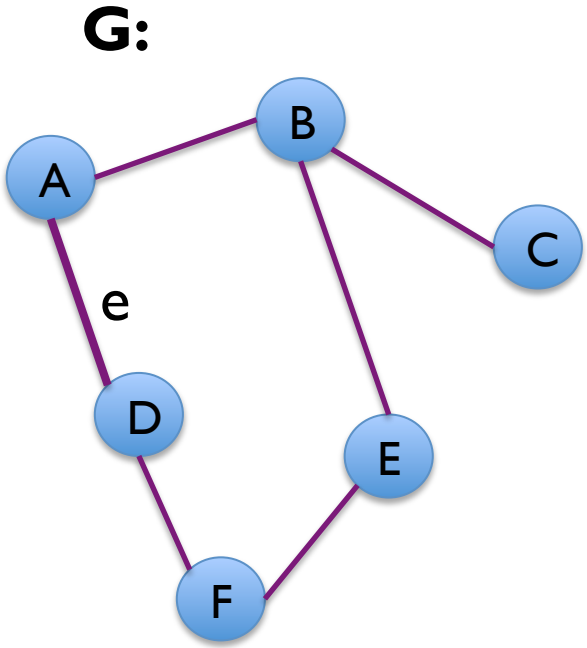
Graphs

- A Graph is an ordered pair G = (*V*, *E*) where:
  - *V* is a set of vertices

    {A, B, C, D, E, F}

  - *E* is a set of edges

    {(A,B), (A,D), (B,C), (B,E), (D,F), (E,F)}

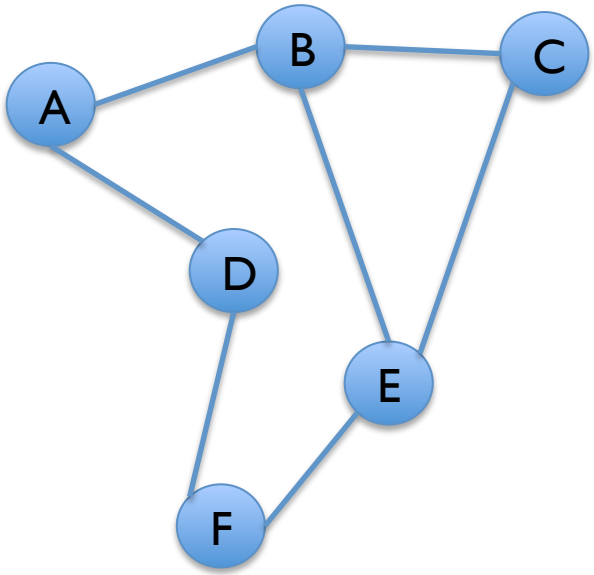


Graphs

- The *order* of a graph is the cardinality of *V*:
  - *V* = {A, B, C, D, E, F}
  - |*V*| = 6 = order of G

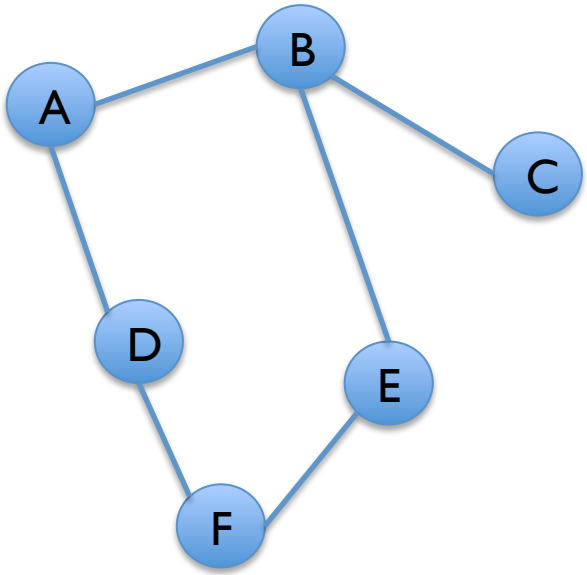
- The *size* of a graph is the cardinality of *E*:

  {(A,B), (A,D), (B,C), (B,E),
  (C,E), (D,F), (E,F)}
  - |*E*| = 7 = size of G



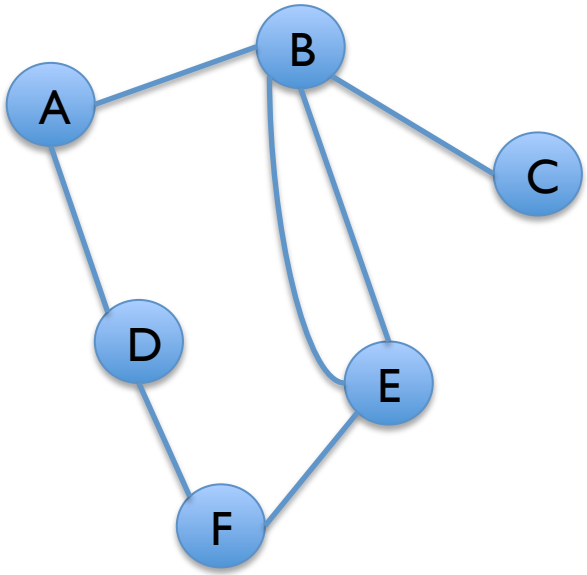
Graphs

- The degree of a vertex is the number of edges incident to that vertex:
  - The degree of A is 2
  - The degree of B is 3



Graphs

- There may be multiples edges between two vertices



Graph Properties

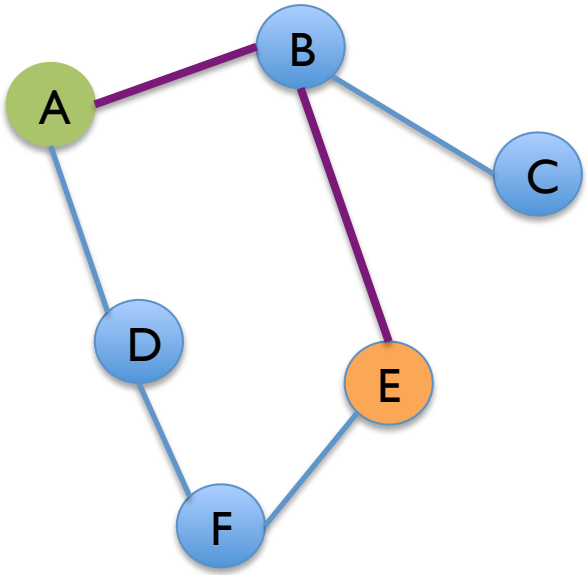
- A *path* is a sequence of vertices and edges (without repetition) that connects two vertices
  - A path between A and E or (AE-path) is:

    A, (A,B), B, (B,E), E

  often written as

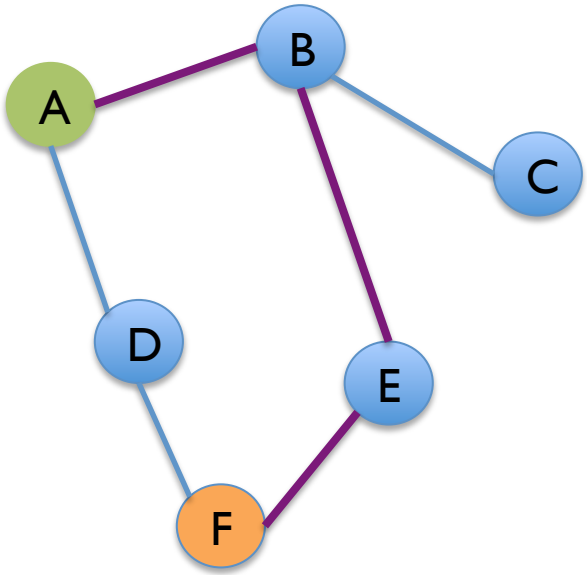
    A, B, E

  implying (A,B), (B,E)



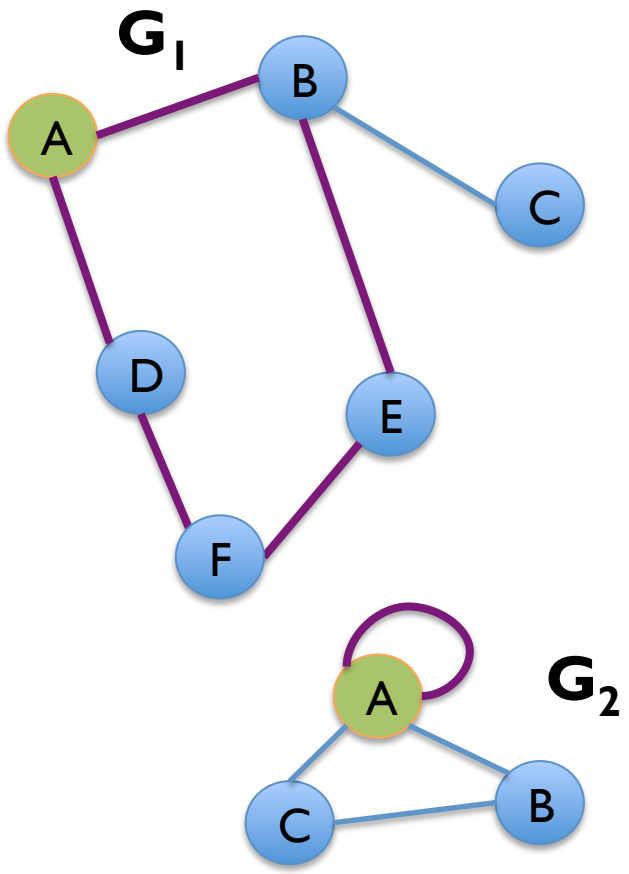
Graph Properties

- The *length* of a path is the number of edges in the path
  - A single edge is a path of length 1
  - The *AF*-path illustrated here has length 3



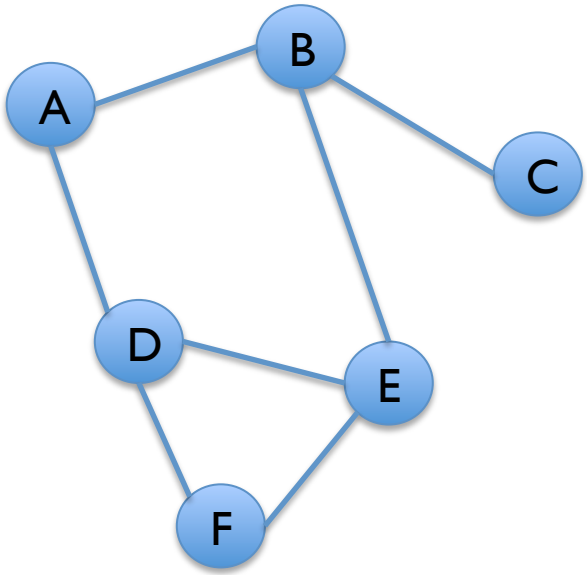
Graph Properties

- A *cycle* is a path of minimum length 1, starting and ending at the same vertex



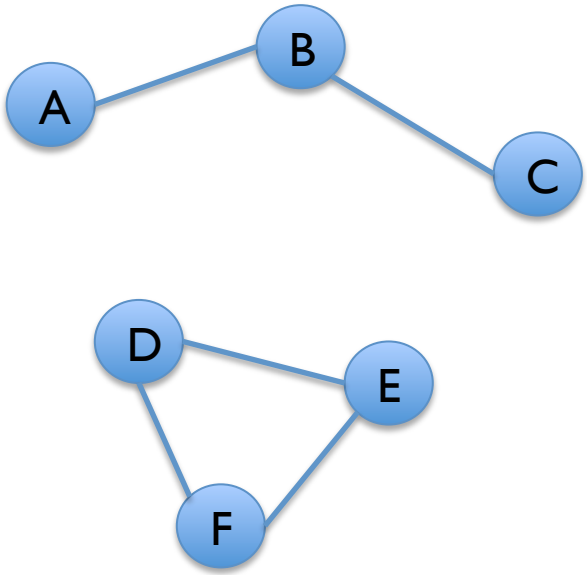
Graph Properties

- A graph G is *connected* if, for every pair of vertices *u* and *v*, there exist at least one *uv*-path
  - This graph is connected



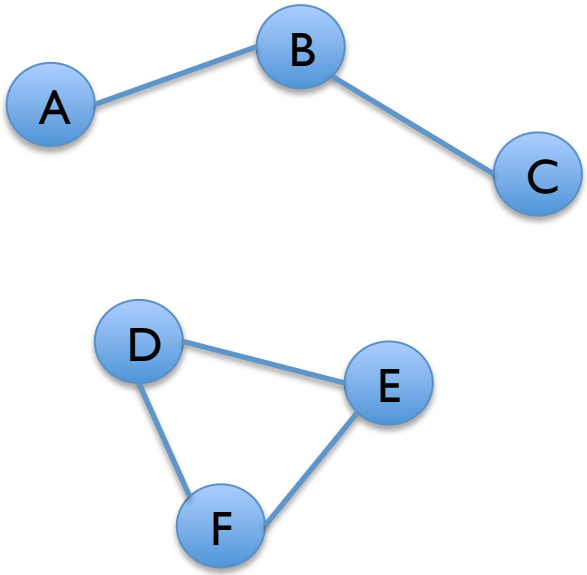
Graph Properties

- A graph G is *connected* if, for every pair of vertices *u* and *v*, there exist at least one *uv*-path
  - This graph is not connected



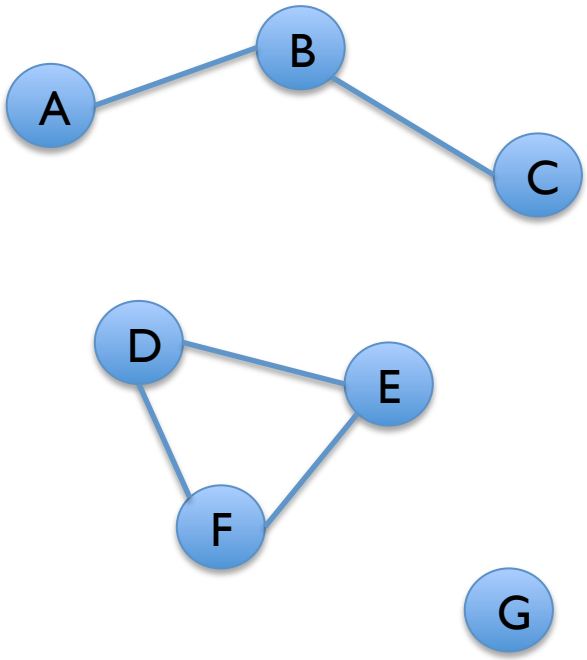
Graph Properties

- A disconnected graph consists of at least two connected components
  - This graph has 2 components



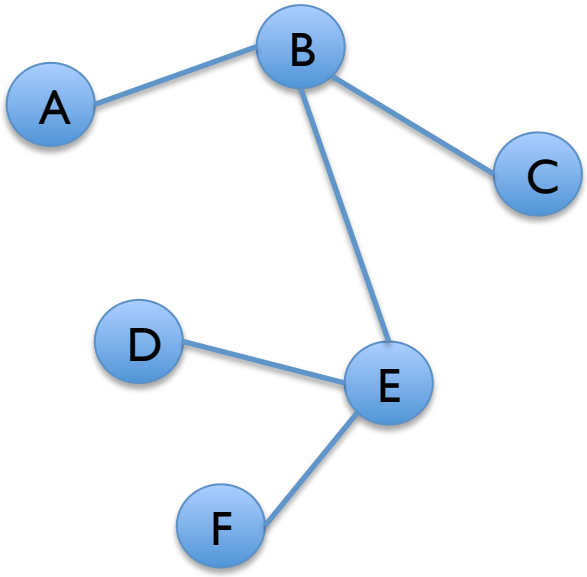
Graph Properties

- A disconnected graph consists of at least two connected components
  - This graph has 3 components



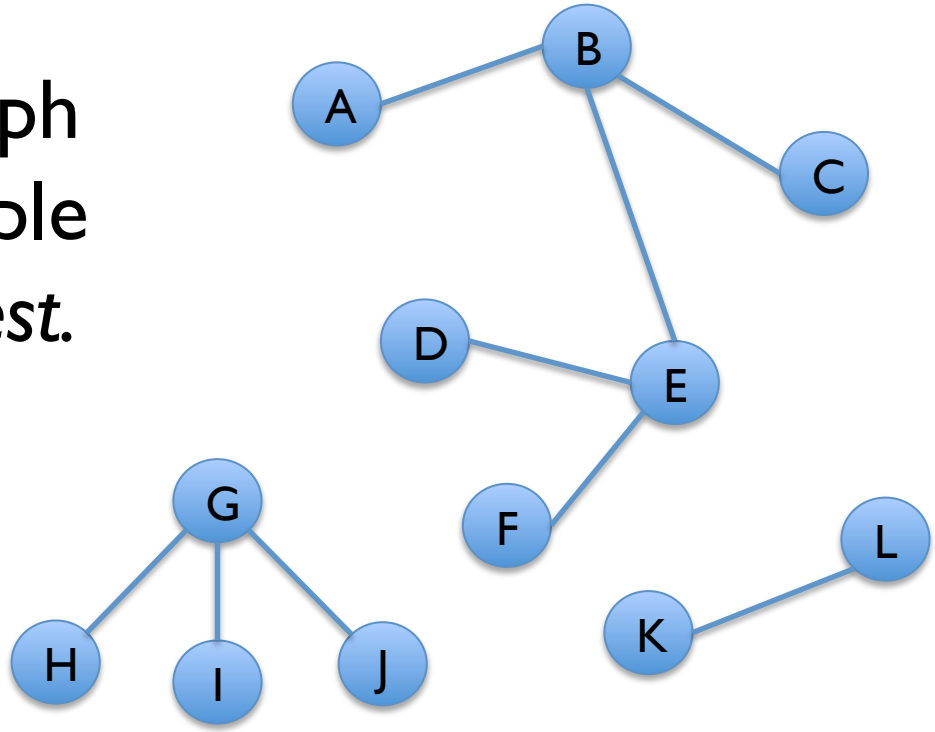
Trees

- A connected graph with no cycles is called a *tree*
- There exist only one path between any given two vertices



Trees

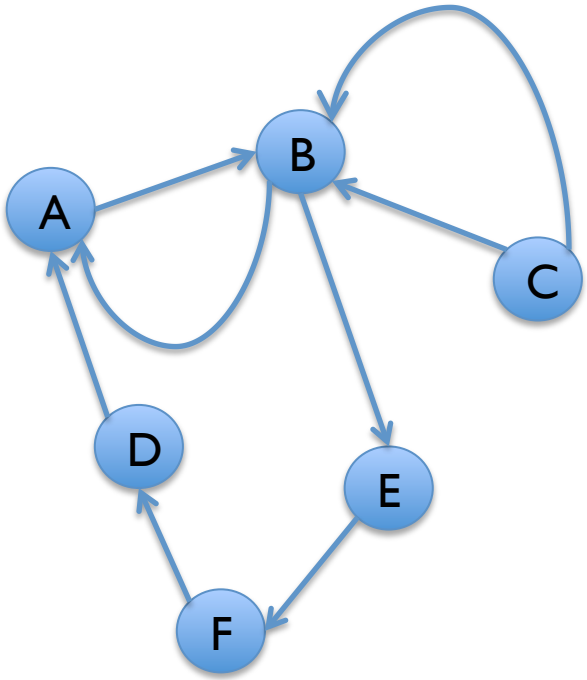
- A disconnected graph composed by multiple trees is called a *forest*.



Directed Graph

- A *directed graph*, or *digraph*, is a graph whose edges have a direction

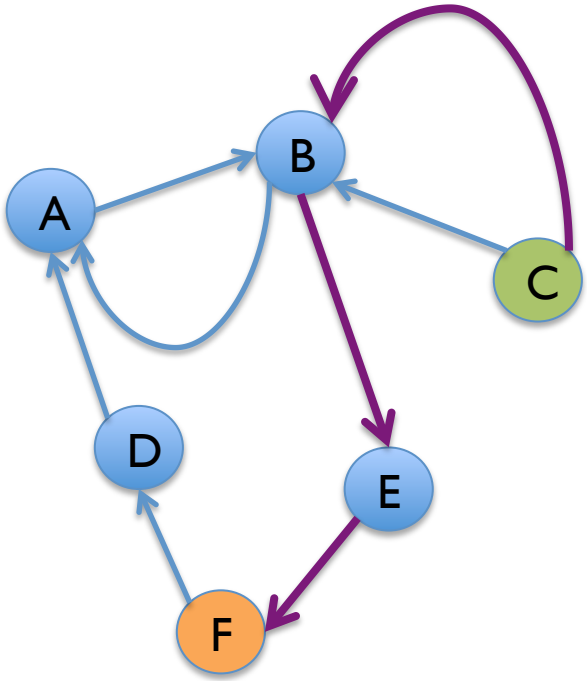
- Edges in directed graphs are called *directed edges* or *arcs*
  - note that (A,B) is not the same edge as (B,A)



Directed Path

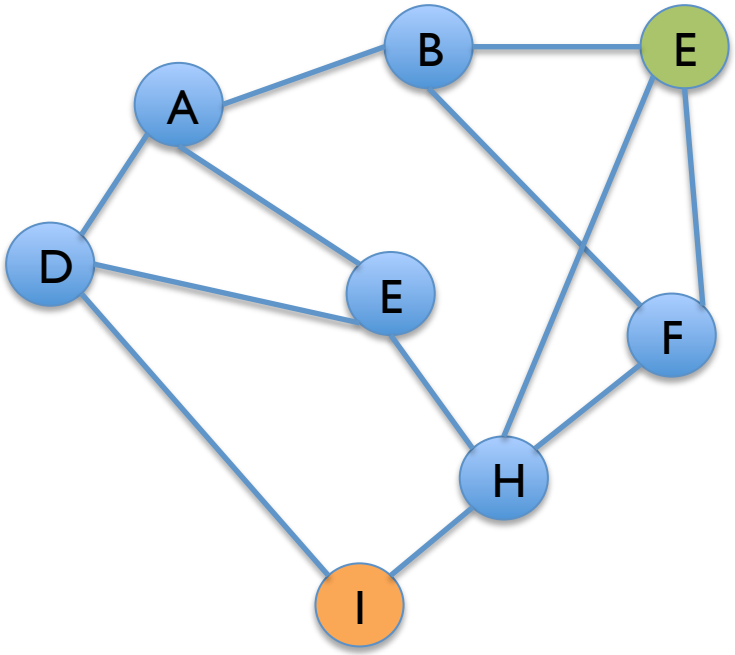
- A *directed path*, is a path following the direction of the edges.

- The endpoints of a directed path are usually referred to as the *source* and *destination*



Graph Traversals

- Say we are looking for a path between E and I
- What should we do?

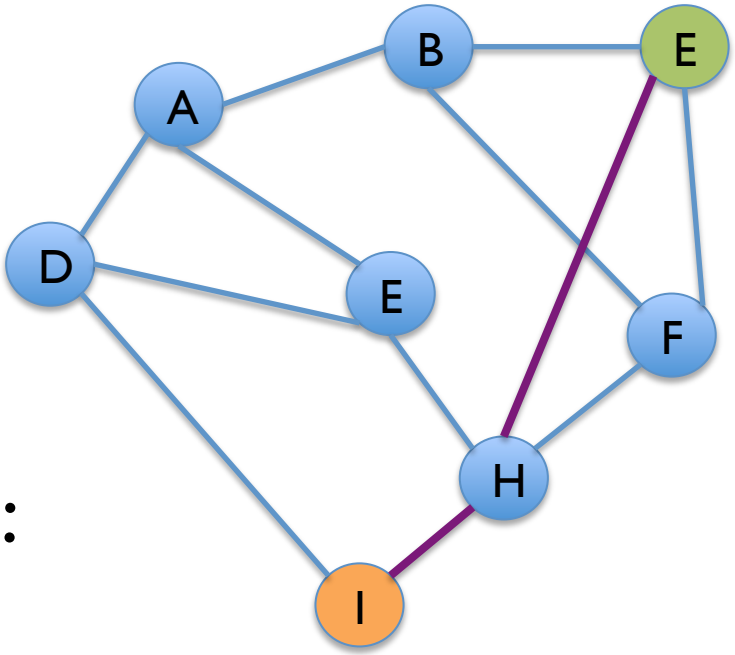


Graph Traversals

- Say we are looking for a path between E and I
- What should we do?
- We can easily see a path:

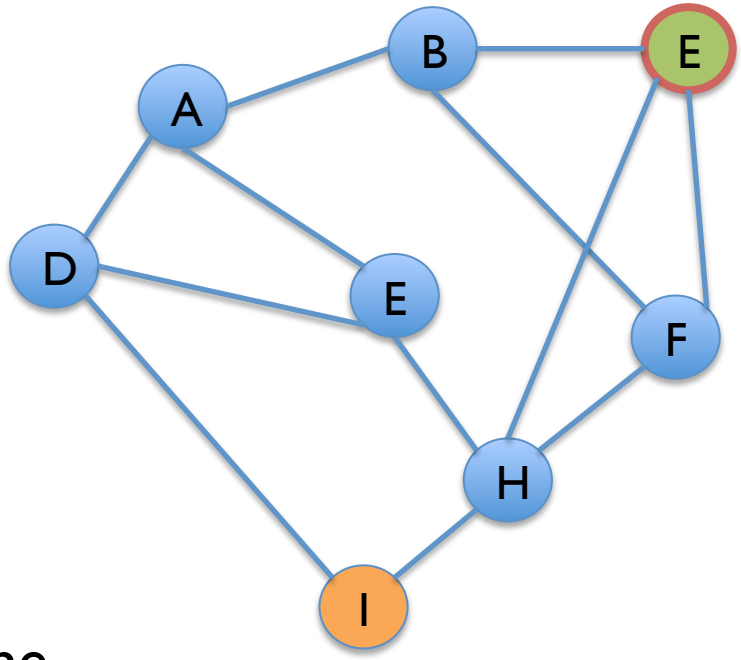
  EI-path = (E, H, I)

  But what do computer see?



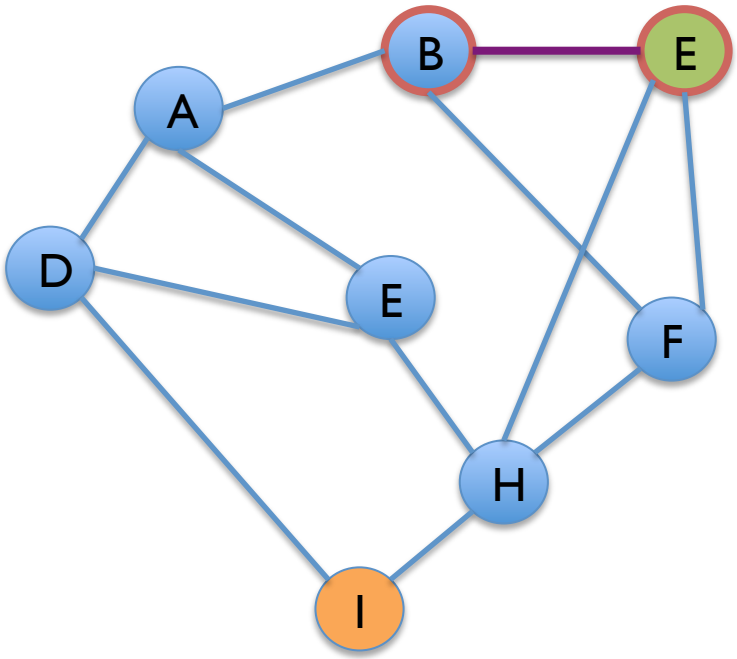
Graph Traversals

- Computers need to explore one thing at the time:
  - Start at E
    - mark it as *seen*
    - Are we there yet?
    - NO? walk to another vertex and check if this one is our destination



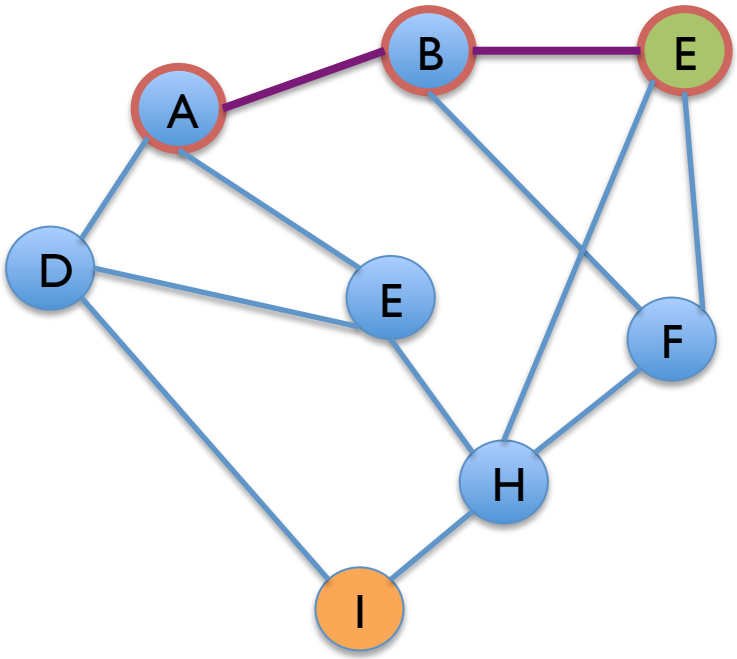
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - No? mark it as *seen*
    - Are we there yet?
    - NO? walk to another vertex and check if this one is our destination



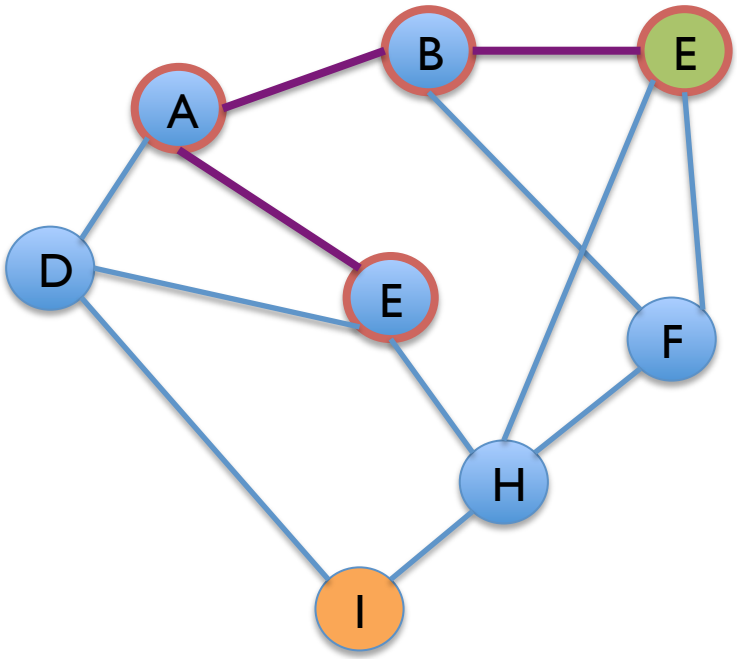
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - No? mark it as *seen*
    - Are we there yet?
    - NO? walk to another vertex and check if this one is our destination



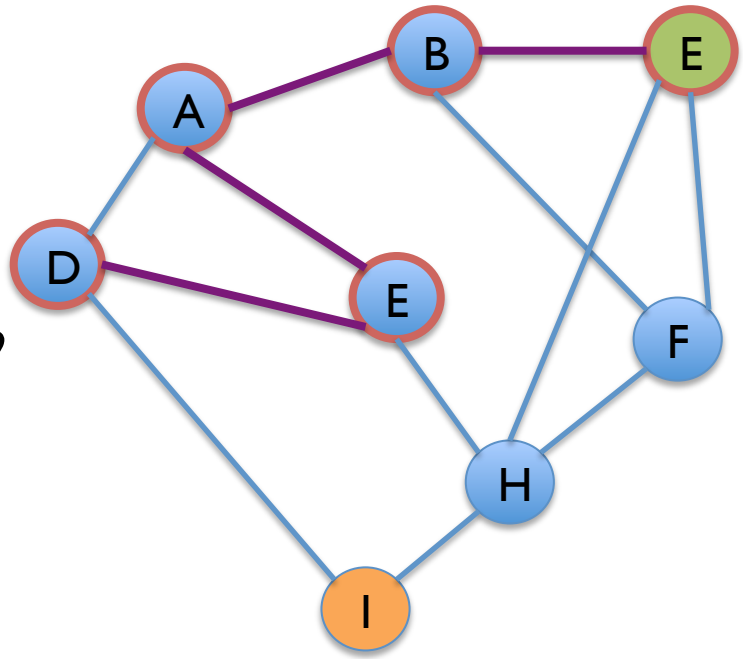
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - mark it as *seen*
    - Are we there yet?
    - NO? walk to another vertex and check if this one is our destination



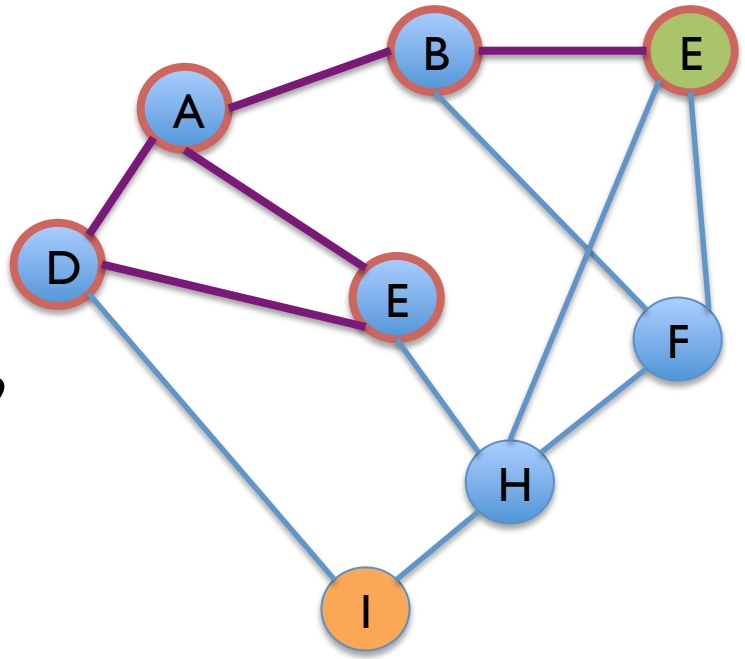
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - mark it as *seen*
    - Are we there yet?
    - NO? walk to another vertex and check if this one is our destination



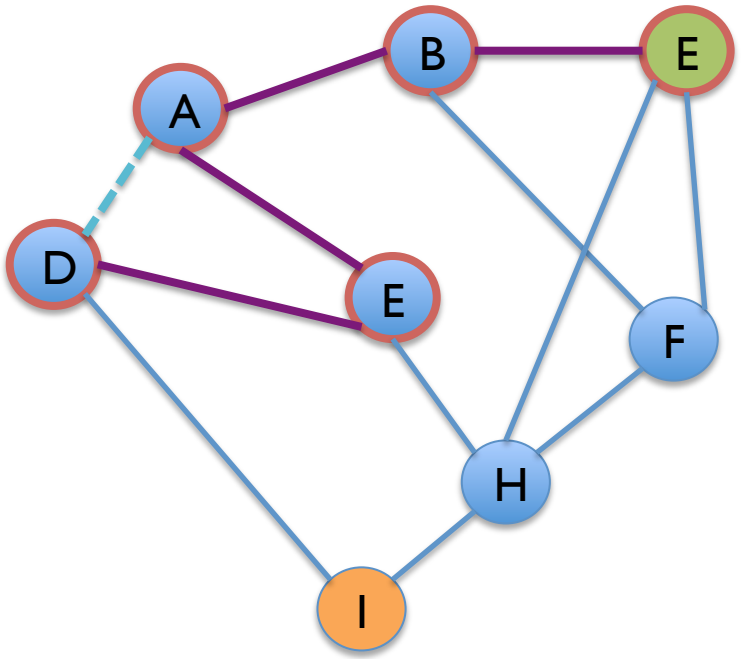
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - Yes? Backtrack



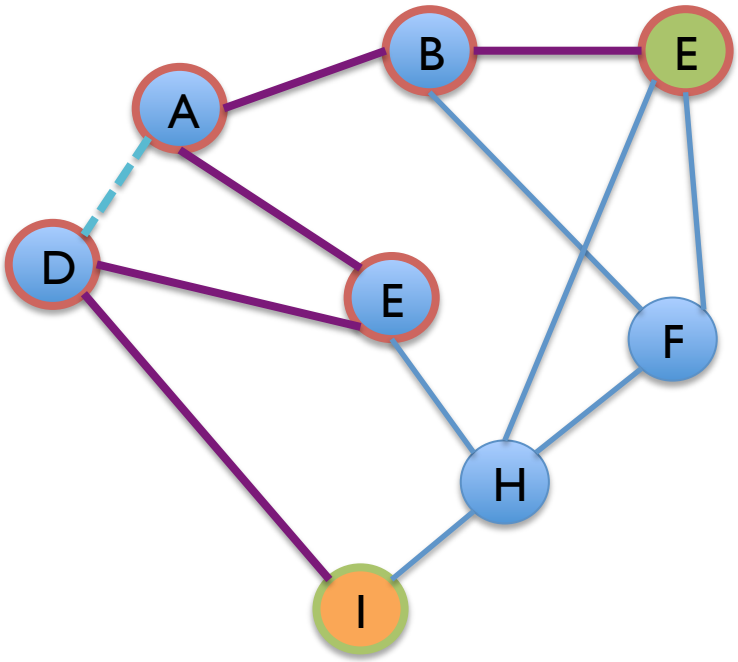
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Is there any other way from here?
    - Yes? Try it



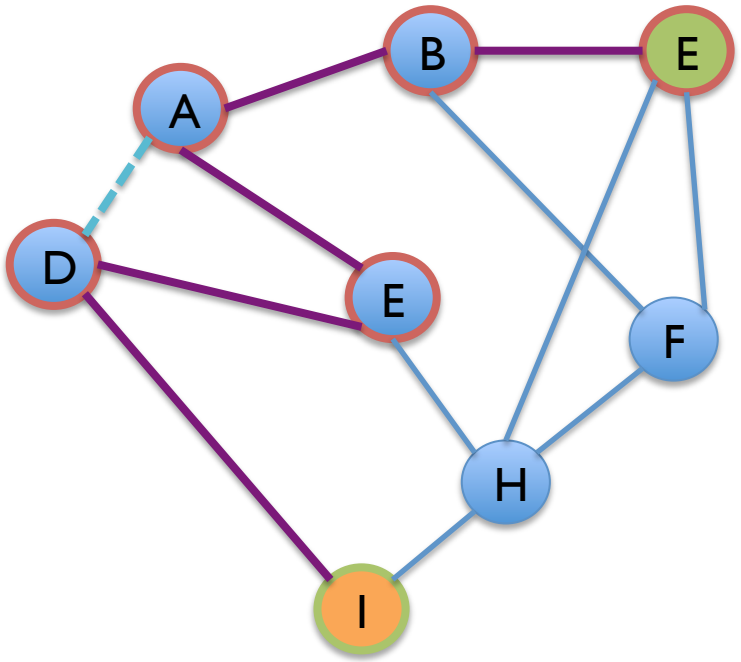
Graph Traversals

- Computers need to explore one thing at the time:
  - Walk to a neighbour
    - Have we seen this before?
    - mark it as *seen*
    - Are we there yet?
    - YES? About time!
      - path is the set of edges followed during the walk



Graph Traversals

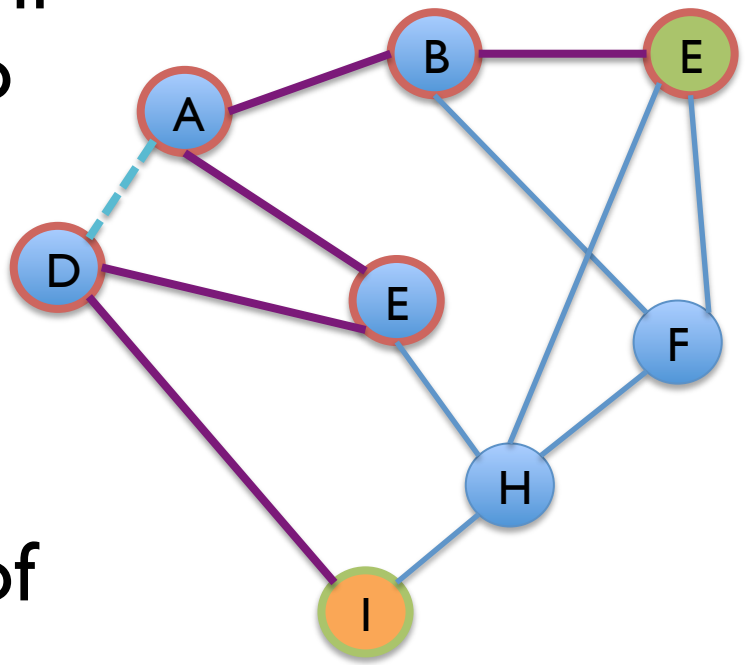
- Algorithms which explore graphs by walking along edges are called *graph traversal* algorithms.
- There are multiple graph traversal algorithms.
- The one we followed corresponds to a *Depth First Search* (DFS)



Depth First Search

- DFS can be used to find if a path exists between to vertices.

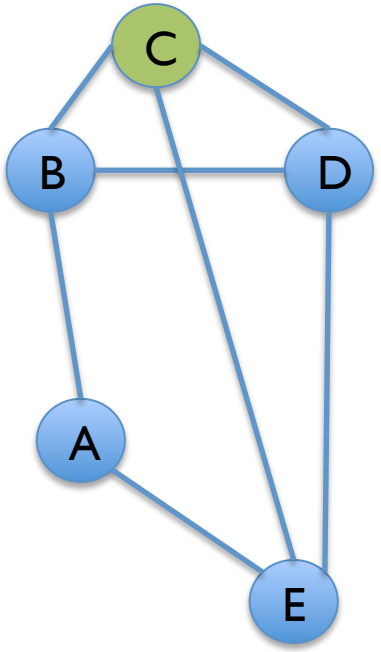
- In general, it is use to explore all vertices in a connected component of a graph, starting from a given *source* vertex *v*.



Depth First Search

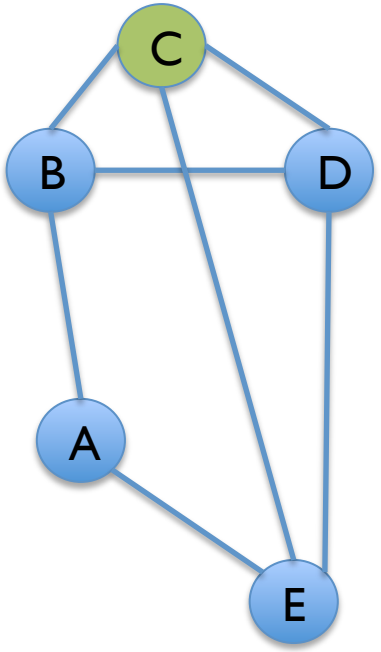
- To explore and backtrack easily, DFS is typically implemented using a stack

- It can also be implemented recursively (which implicitly uses a stack...)



Depth First Search

```
DFS(Graph G, Source vertex s)
    Push s in a stack  # Last In First Out
    WHILE the stack is not empty
        current = pop a vertex from stack
        IF current is not marked as visited
            Mark current as visited
            FOR each neighbour n of current
                IF n is not marked as visited
                push n in the stack
```



Depth First Search

DFS(G, *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

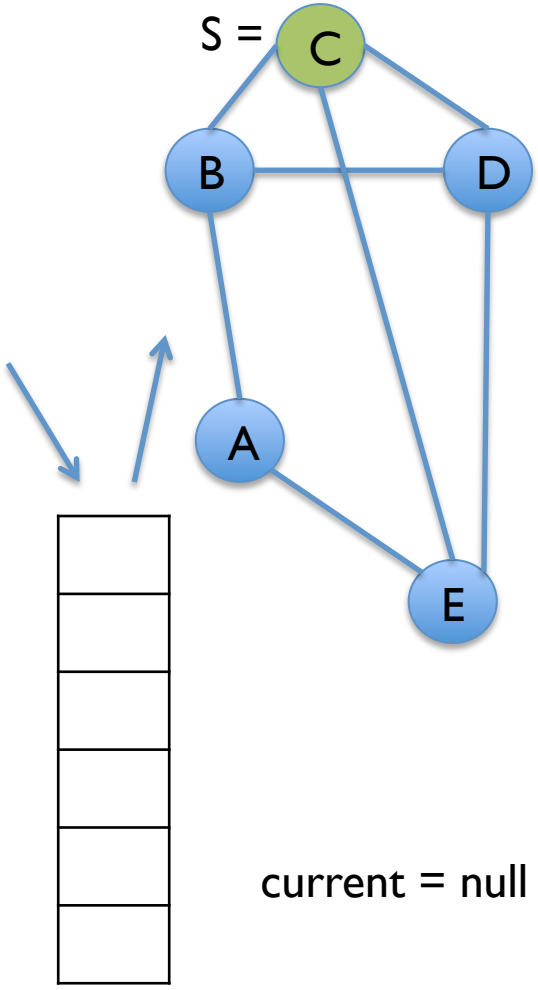
IF current is not marked as visited

Mark current as visited

FOR each neighbour n of current

IF n is not marked as visited

push n in the stack



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

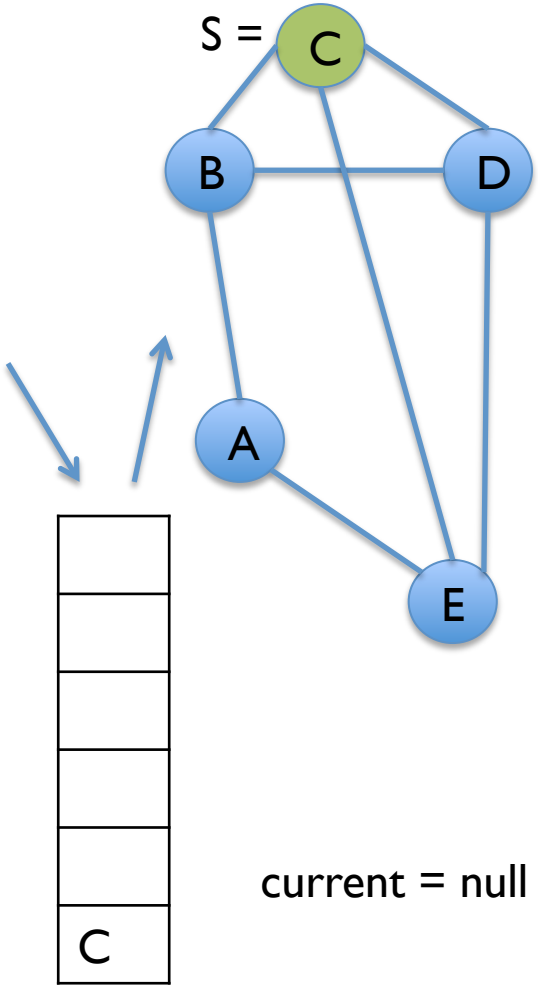
IF current is not marked as visited

Mark current as visited

FOR each neighbour n of current

IF n is not marked as visited

push n in the stack



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

IF current is not marked as visited

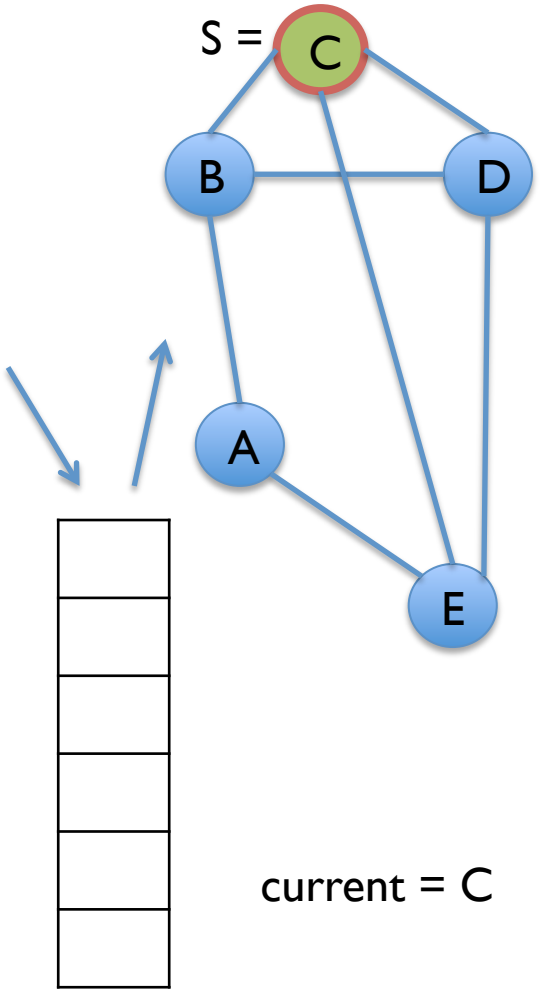
Mark current as visited*

FOR each neighbour n of current

IF n is not marked as visited

push n in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

IF current is not marked as visited

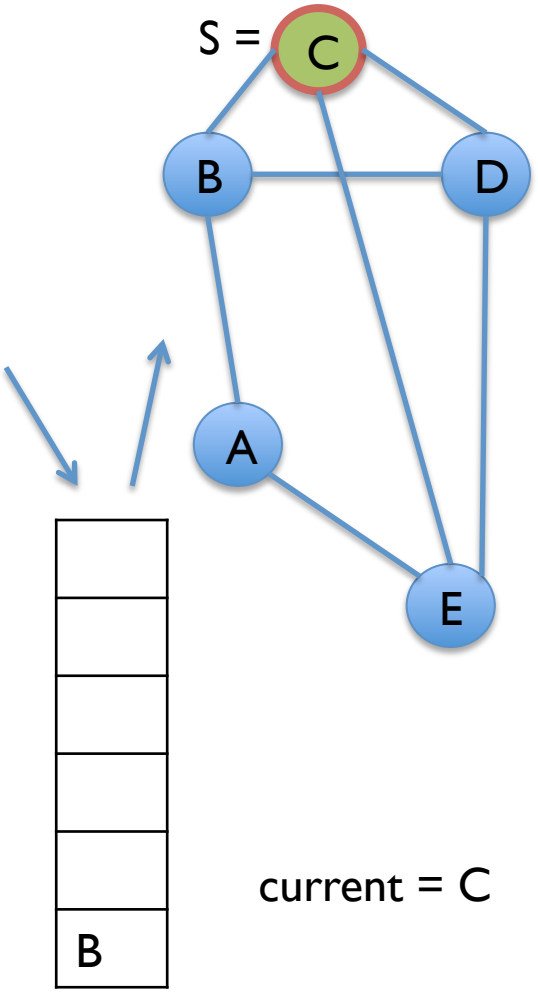
Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack

* let's explore in alphabetical order



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

IF current is not marked as visited

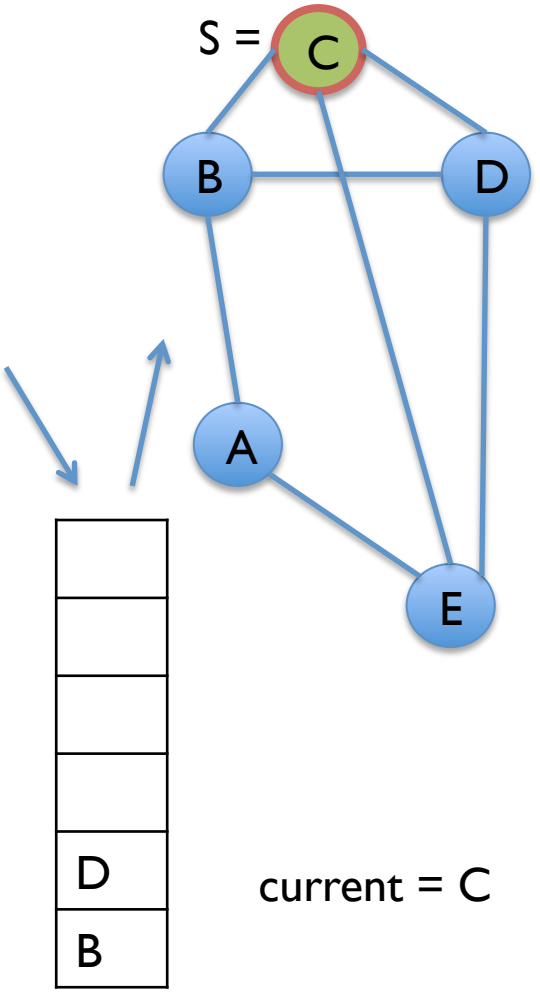
Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack

* let's explore in alphabetical order



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

IF current is not marked as visited

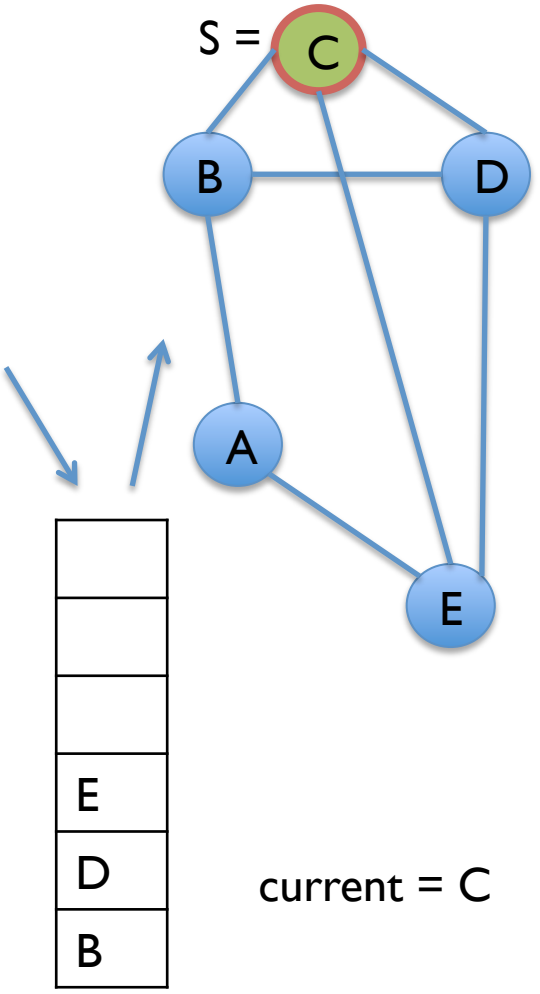
Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack

* let's explore in alphabetical order



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack

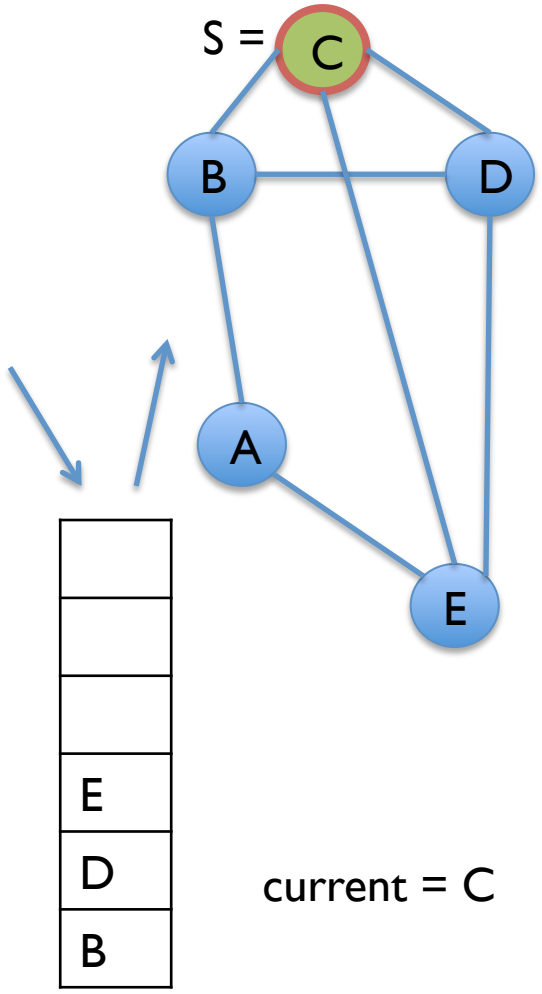
IF current is not marked as visited

Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

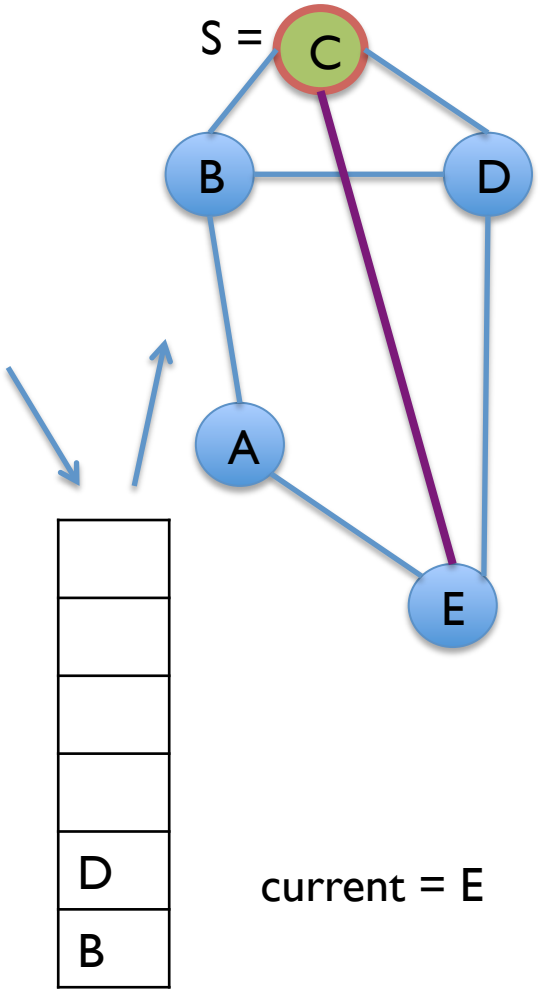
Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

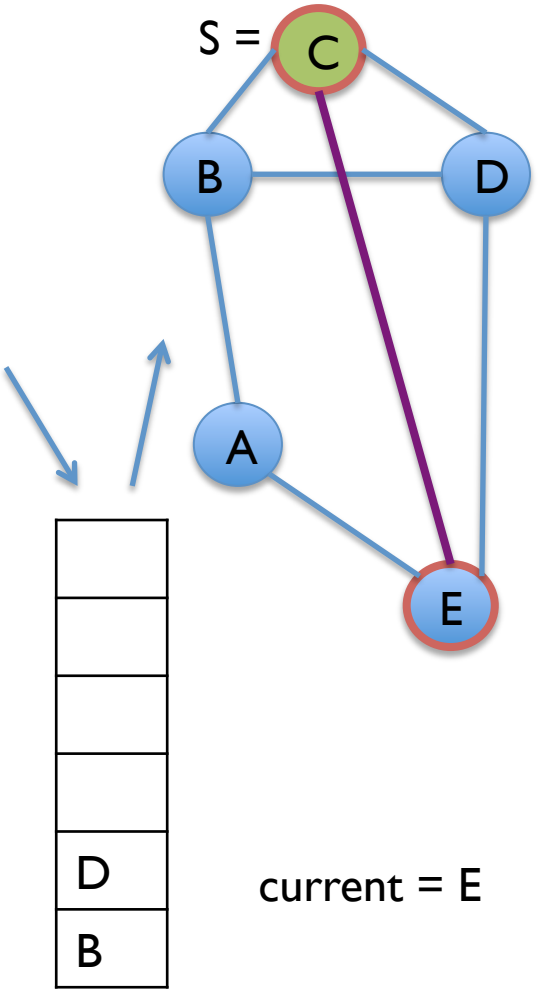
Mark current as visited

FOR each neighbour v of current*

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

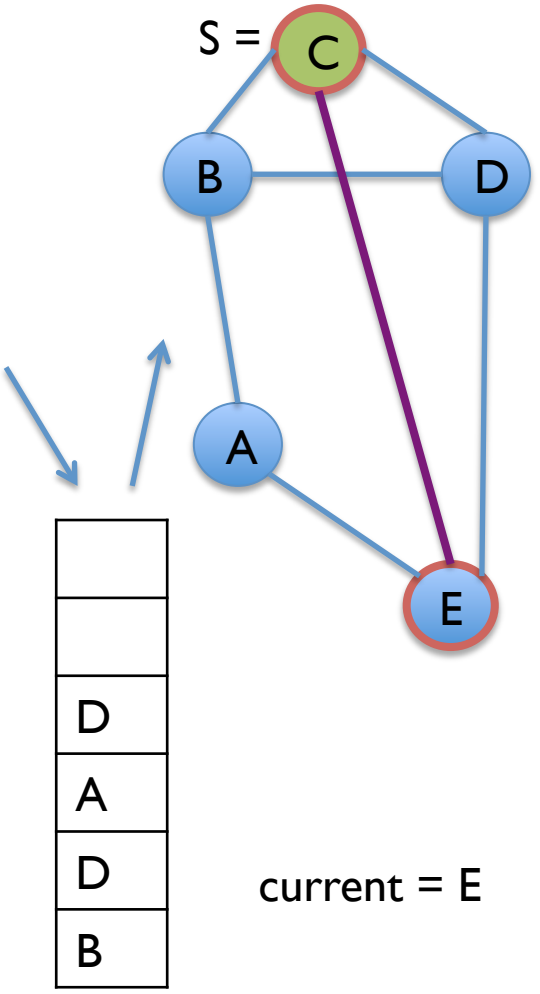
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

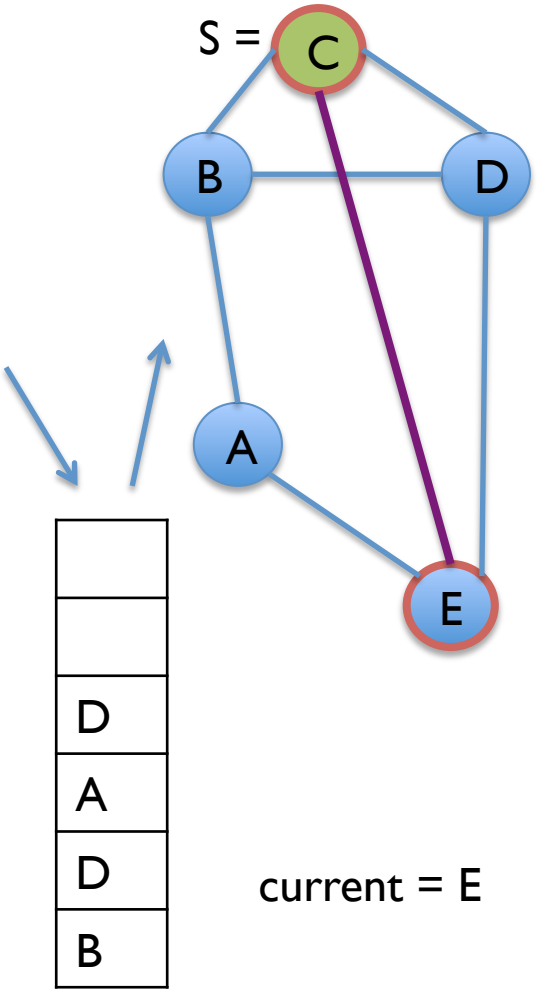
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

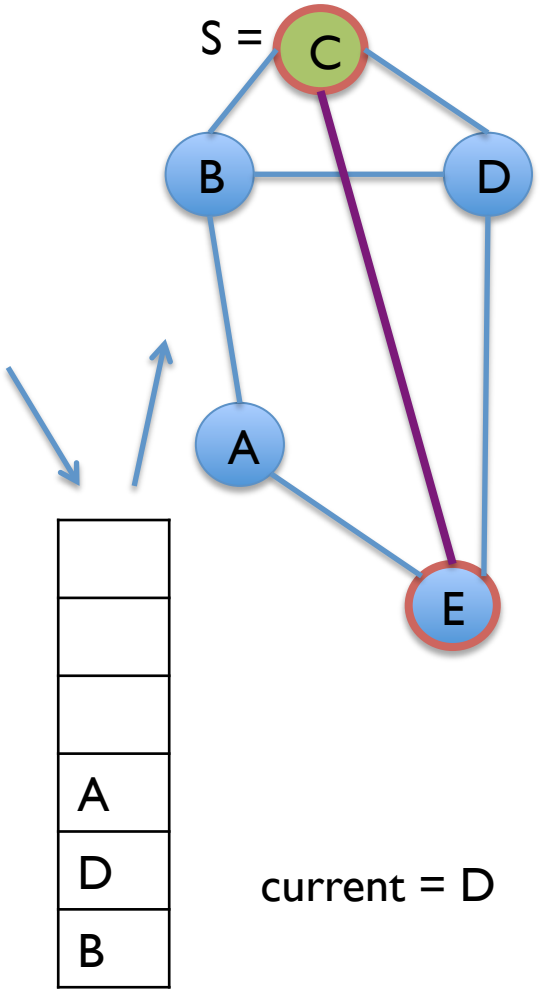
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

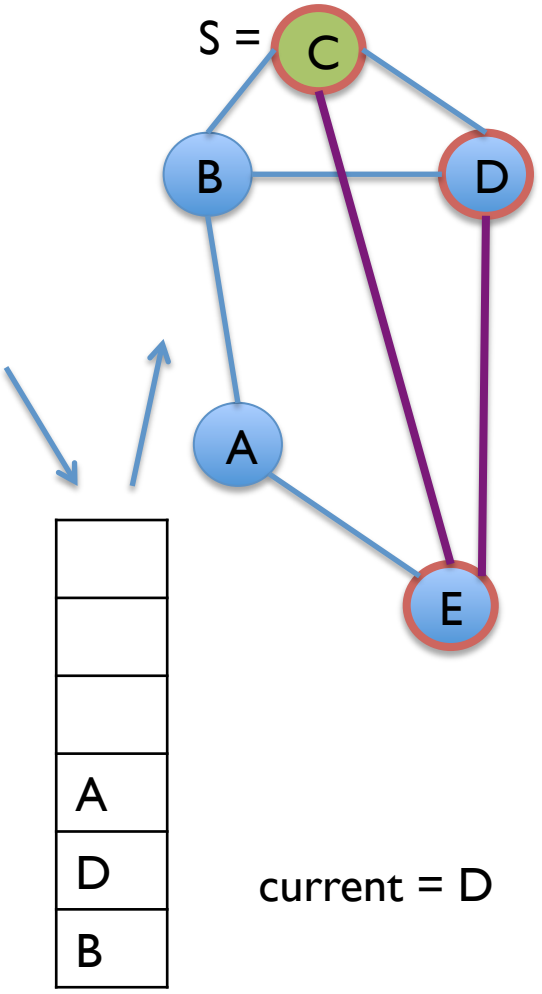
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

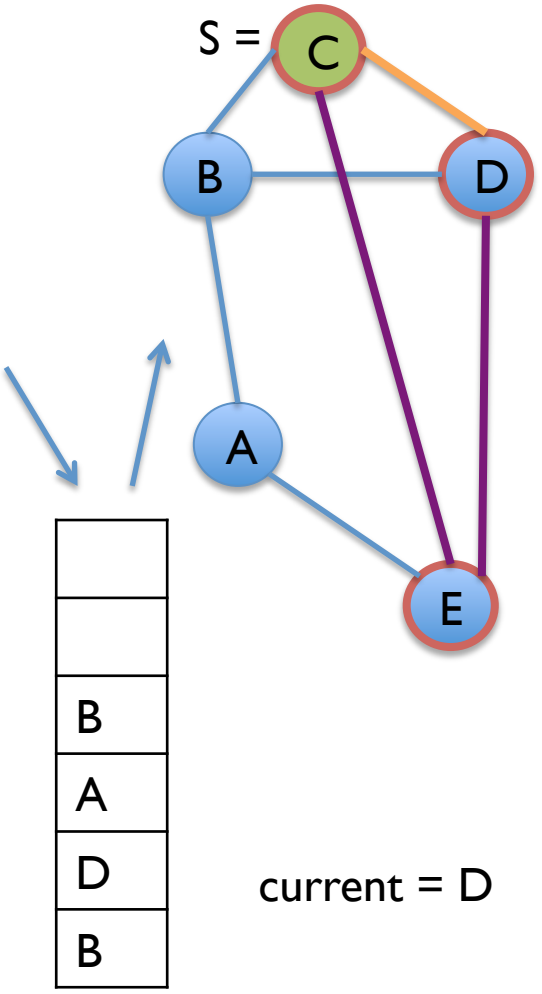
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

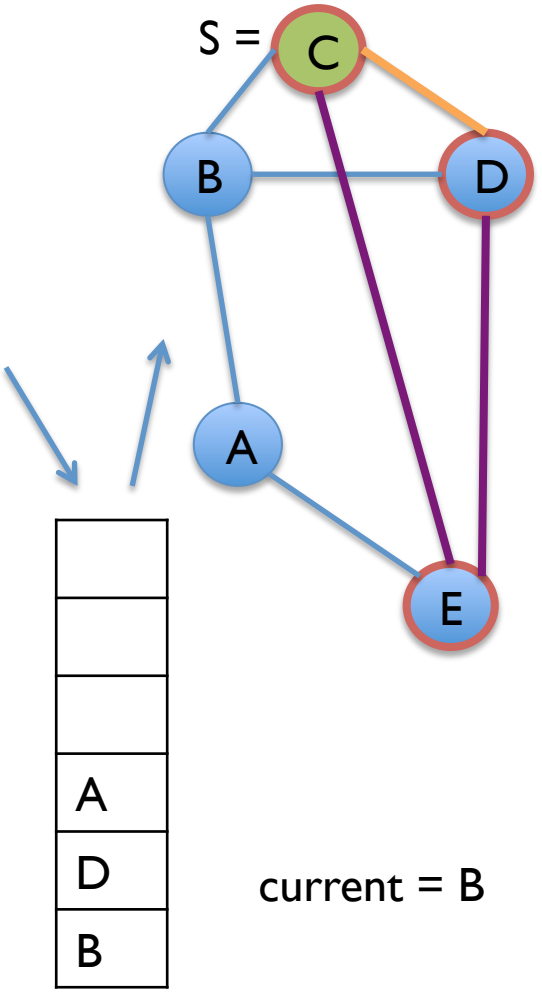
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

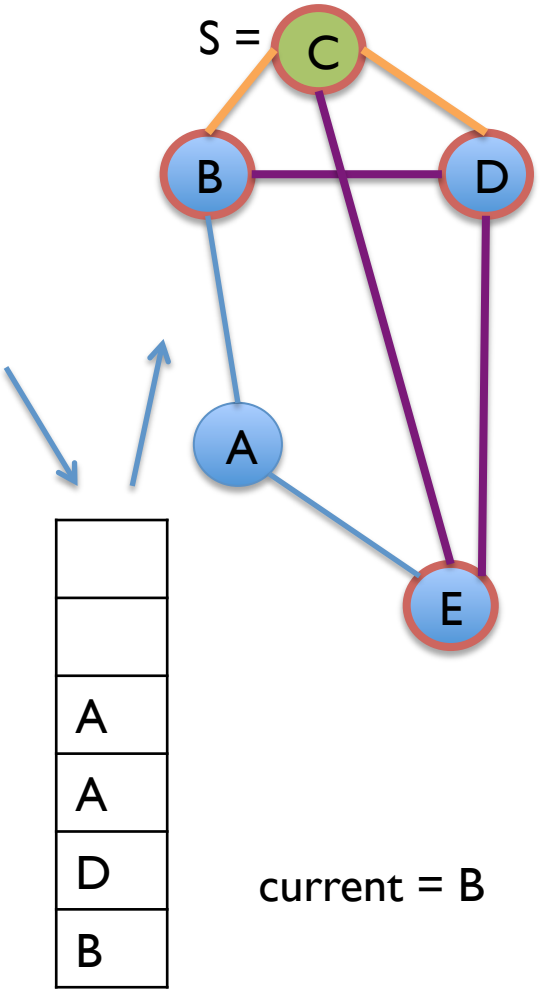
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

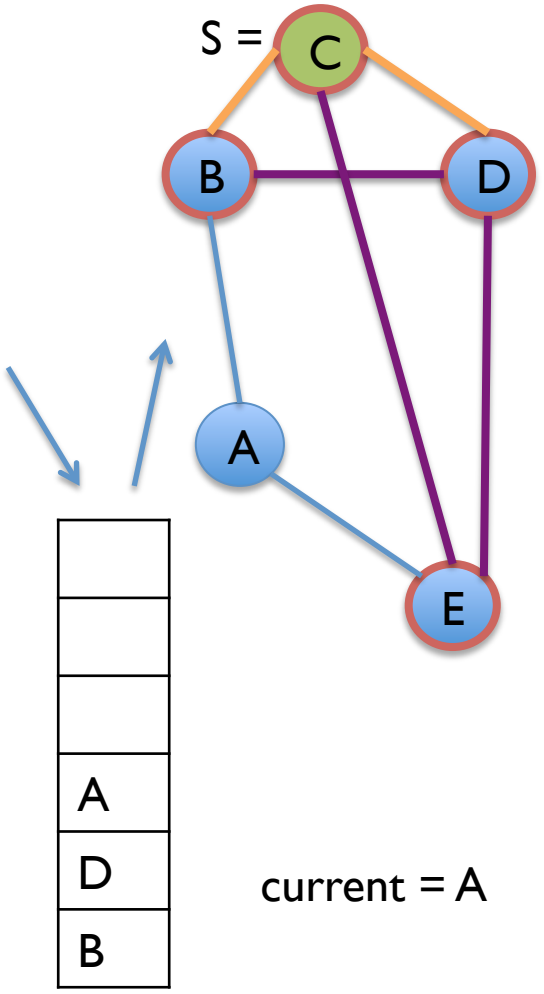
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

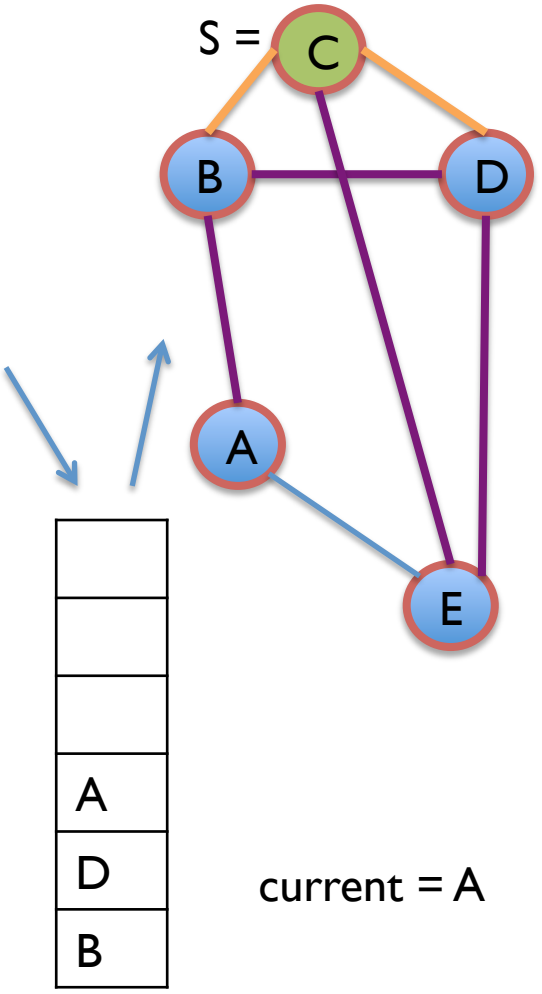
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

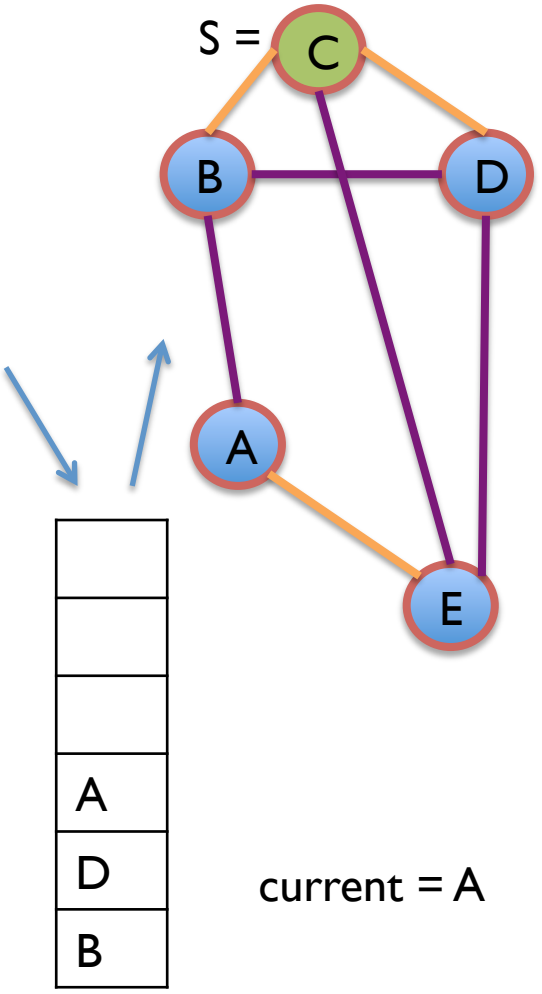
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

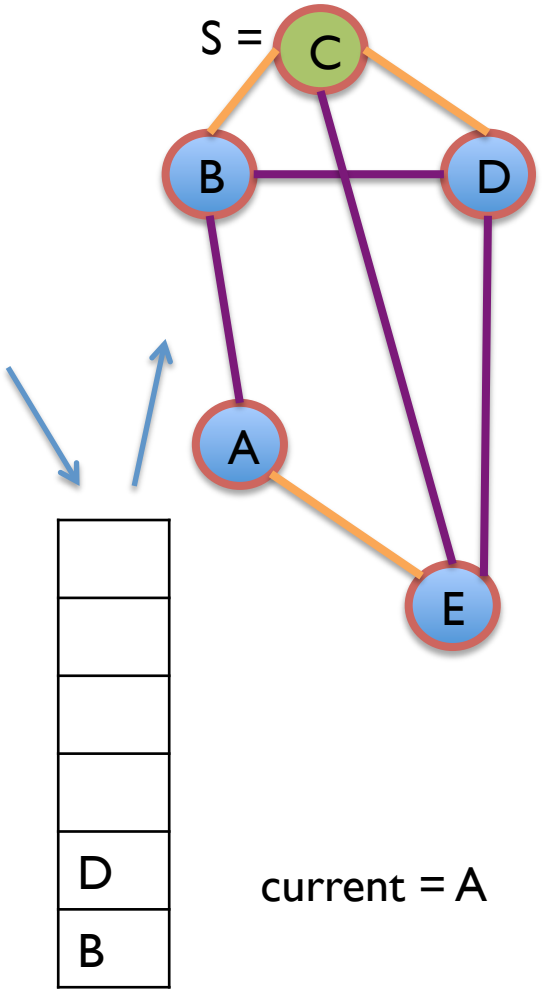
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

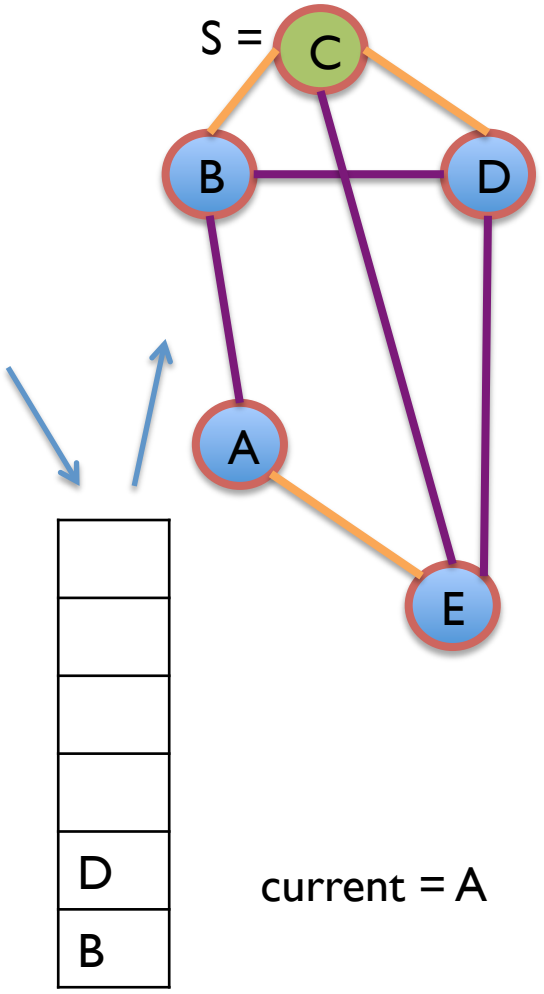
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

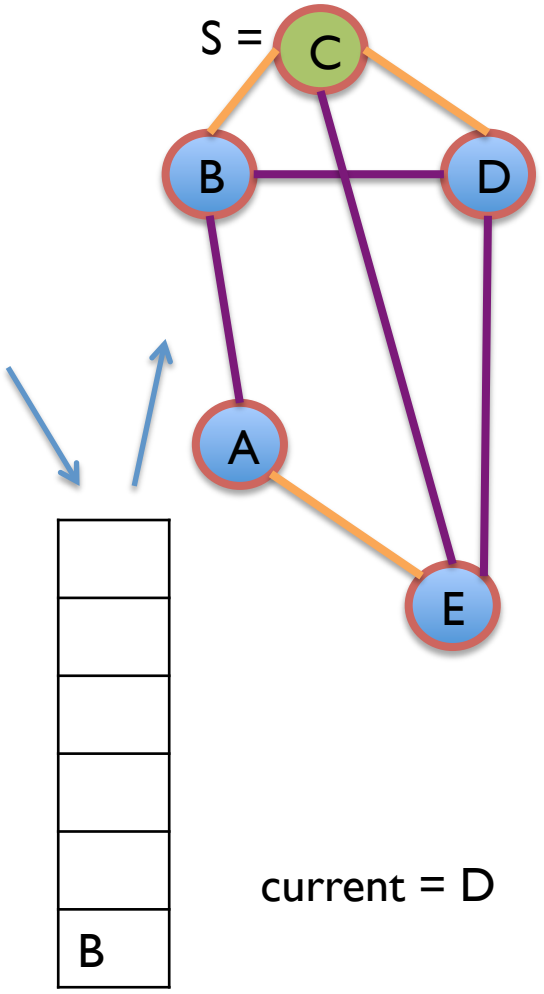
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

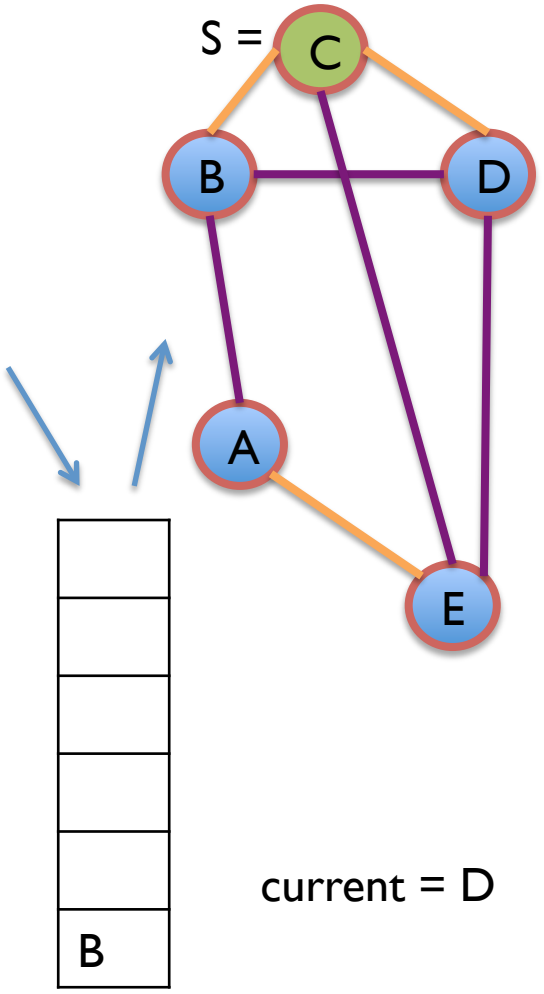
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

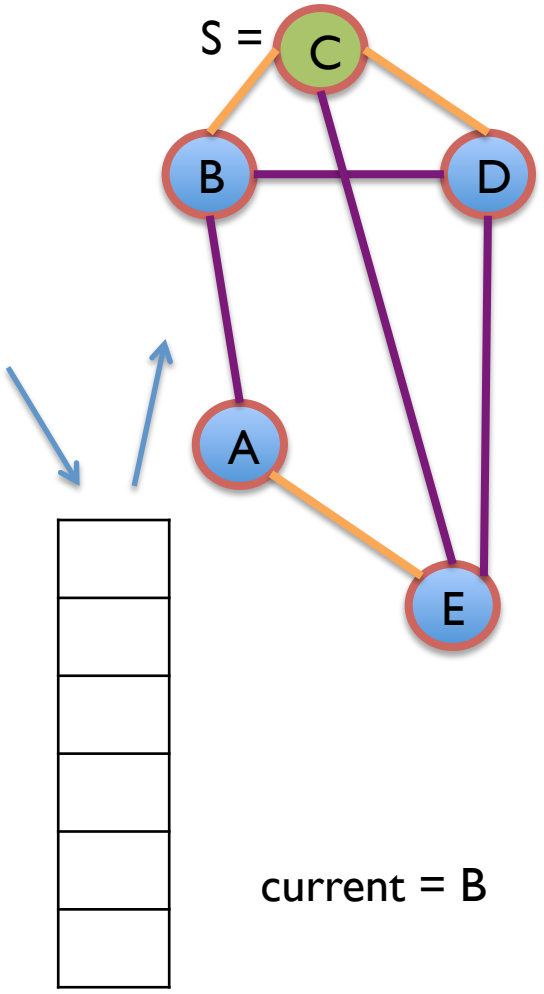
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

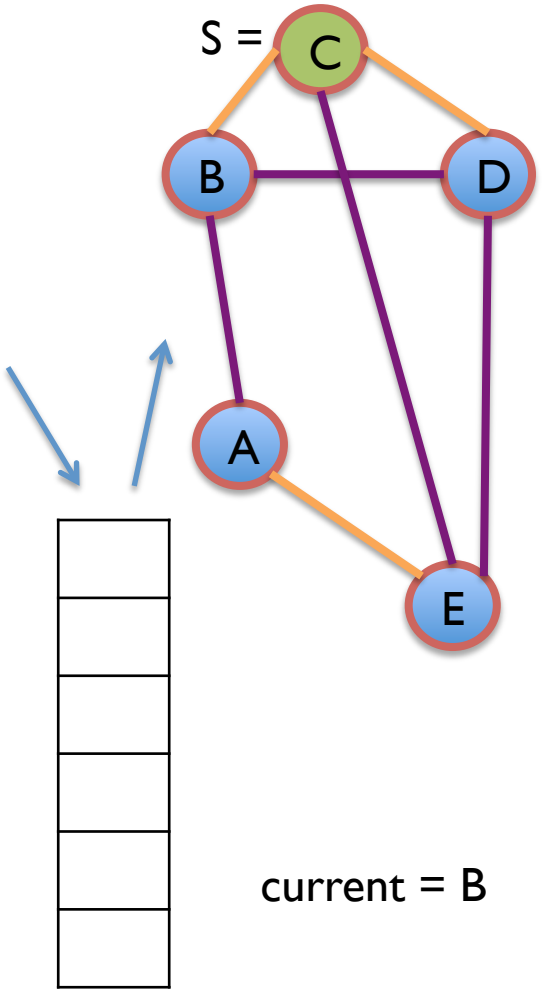
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

* keep track of where you came from



Depth First Search

DFS(G, s = *C*)

Push s in a stack

WHILE the stack is not empty

current = pop a vertex from stack*

IF current is not marked as visited

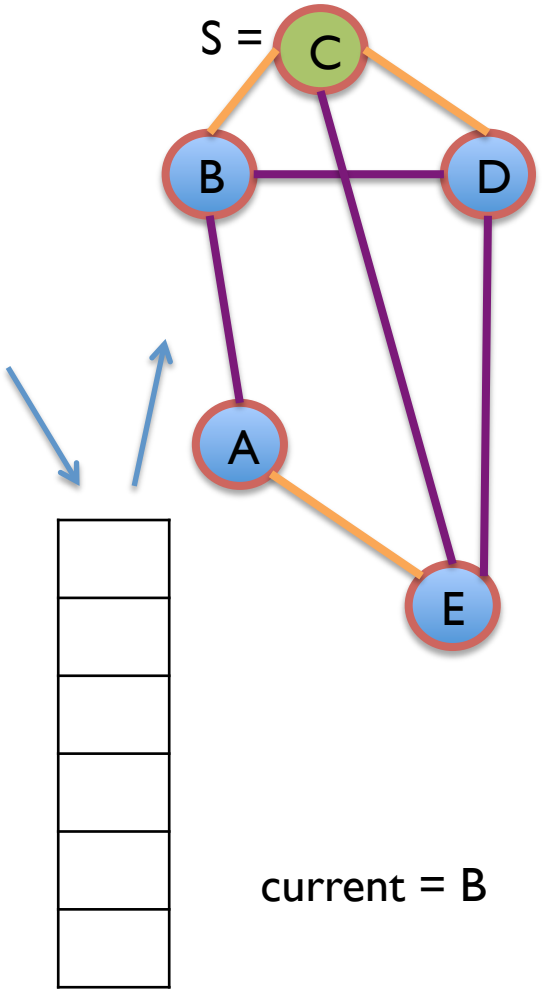
Mark current as visited

FOR each neighbour v of current

IF v is not marked as visited

push v in the stack

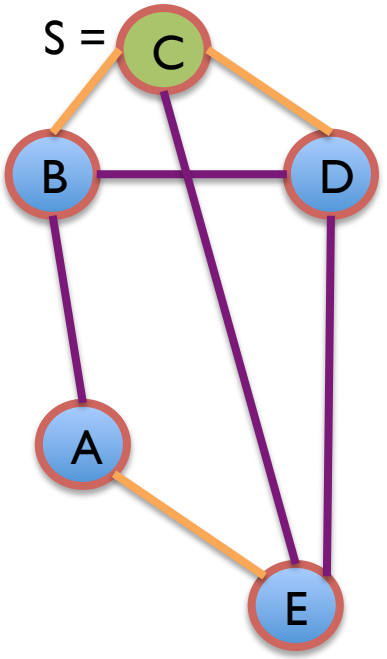
DONE!



Depth First Search

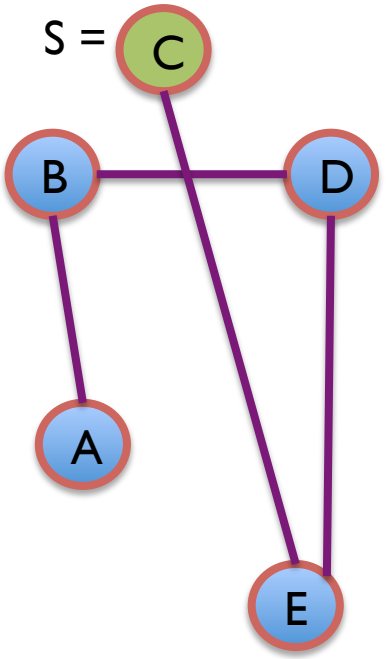
- Edges used to performed the traversal are called *tree edges*, or *discovery edges*

- Edges connected that would take us back on an explored part of our path are called *back edges*
  - note: if there exists a back edge, the graph must contain a cycle



Depth First Search

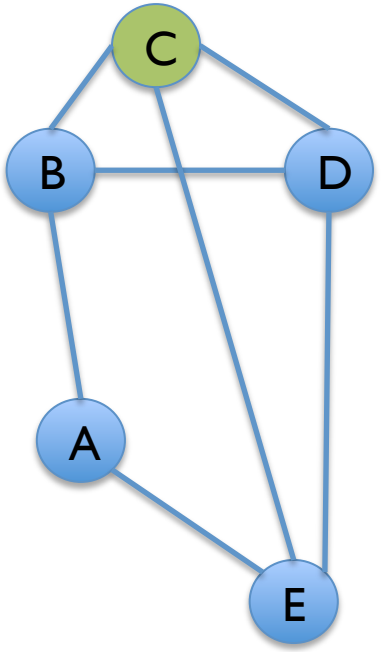
- Edges used to performed the traversal are called *tree edges*, or *discovery edges*
- Edges connected that would take us back on an explored part of our path are called *back edges*
  - note: if there exists a back edge, the graph must contain a cycle
- Keeping only the tree edges yields the DFS tree



Breadth-First Search

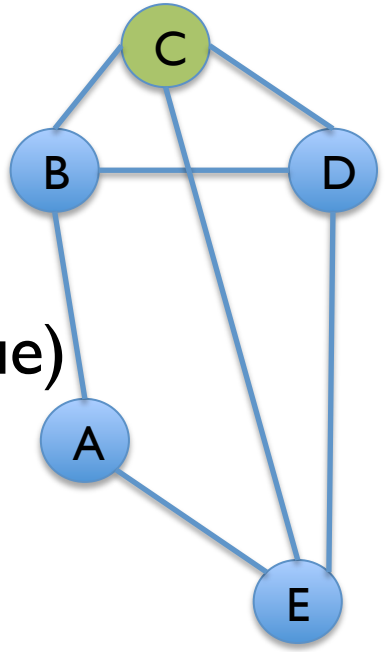
- To explore and backtrack easily, DFS is typically implemented using a stack

- What is we explored using a queue instead?



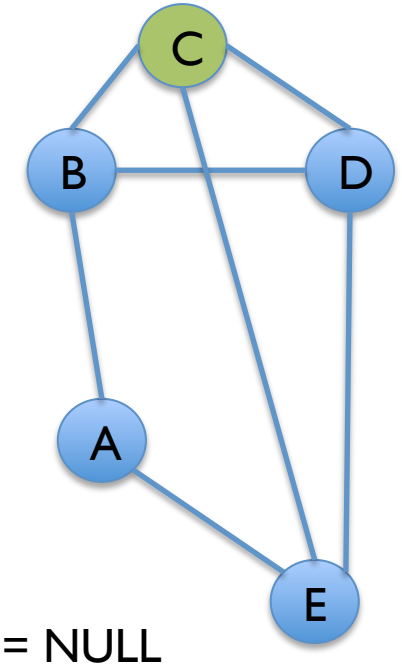
Breadth-First Search

```
BFS(Graph G, Source vertex s)
    mark s
    Put s in a queue  # First In First Out
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



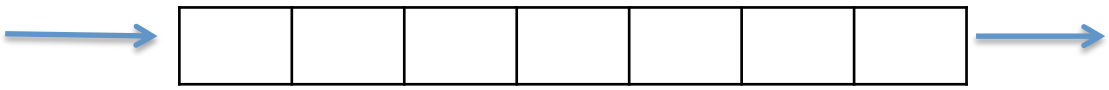
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



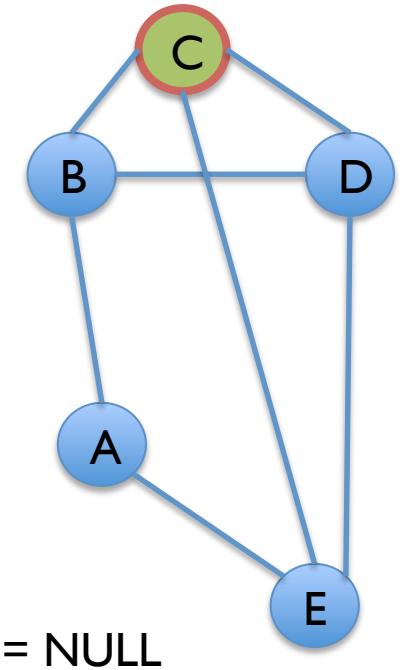
current = NULL

queue:



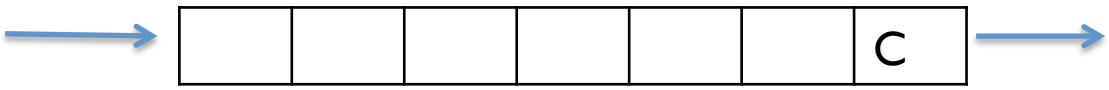
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



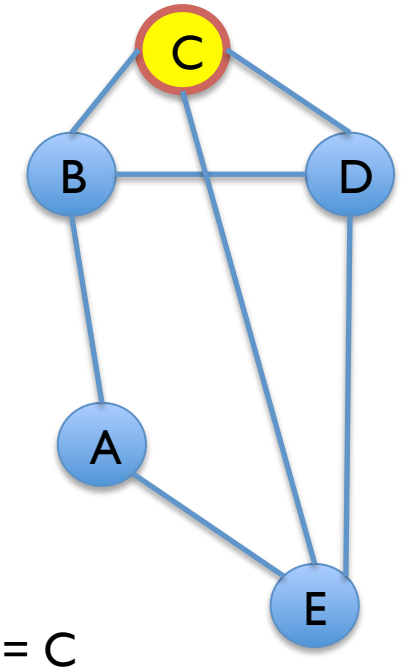
current = NULL

queue:



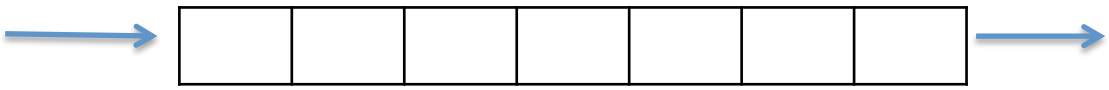
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



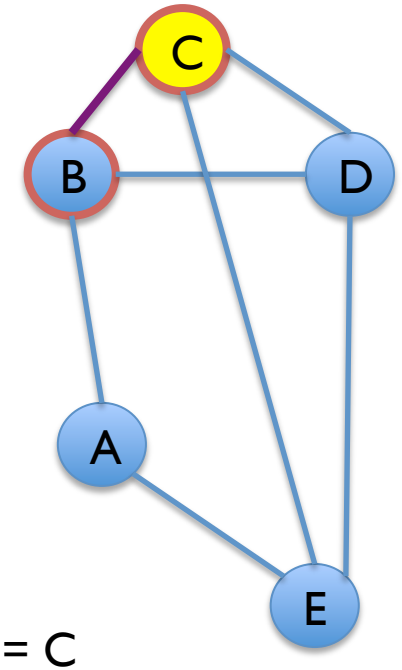
current = C

queue:



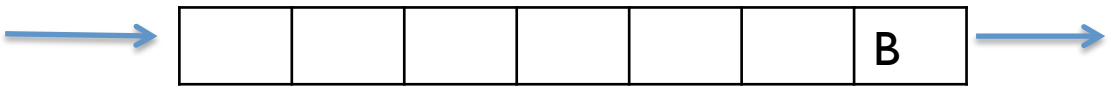
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current *
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



current = C

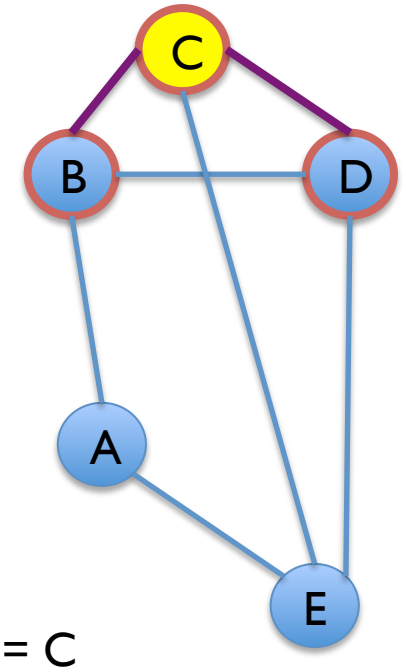
queue:



*In alphabetical order

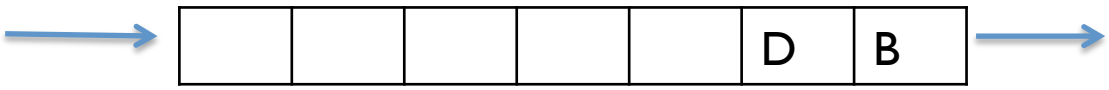
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current *
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



current = C

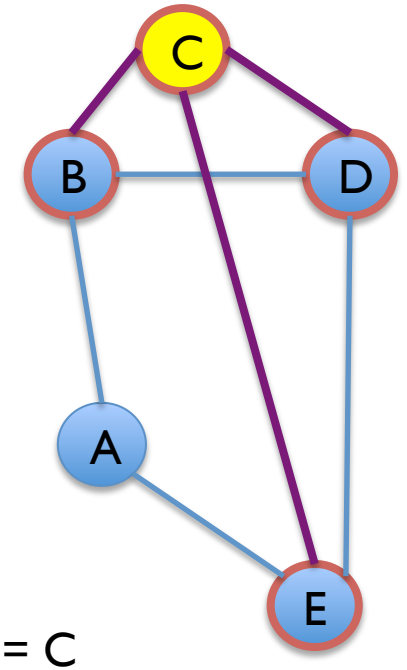
queue:



*In alphabetical order

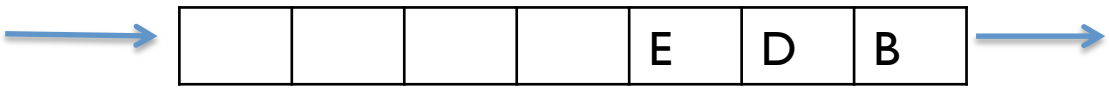
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current *
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



current = C

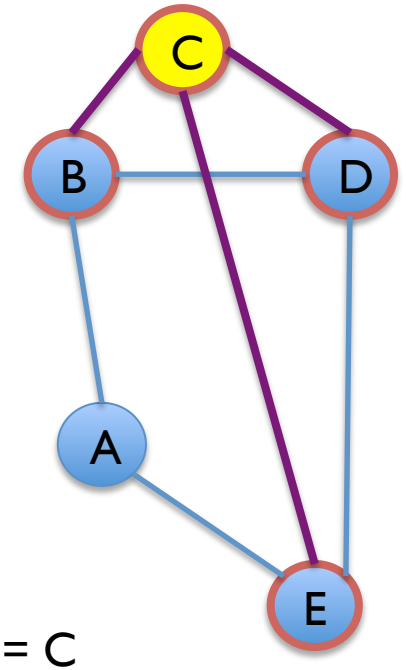
queue:



*In alphabetical order

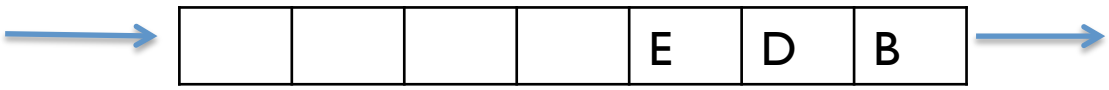
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current *
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



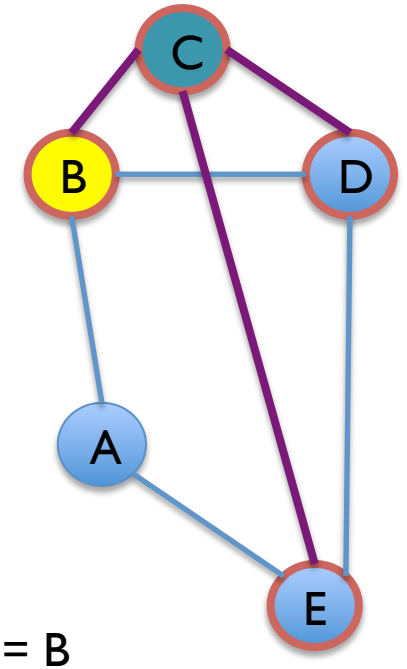
current = C

queue:



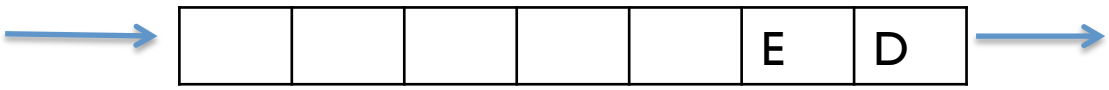
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current *
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



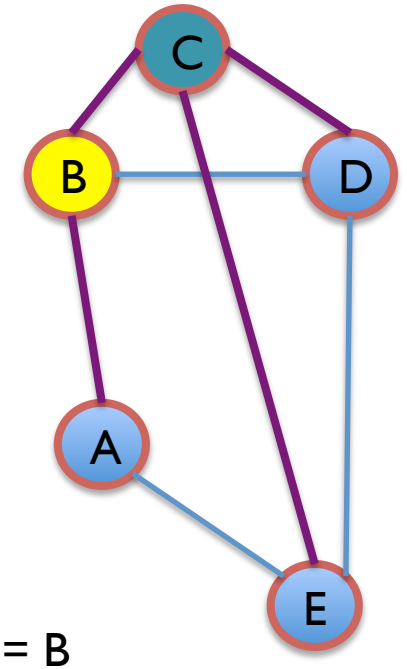
current = B

queue:



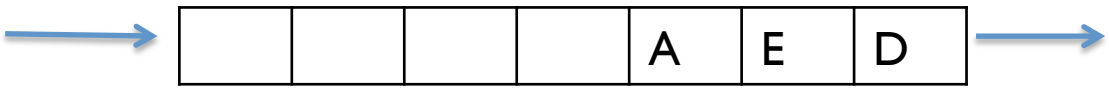
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



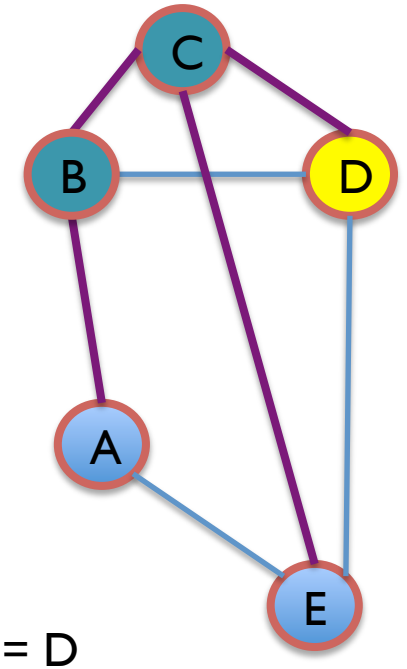
current = B

queue:



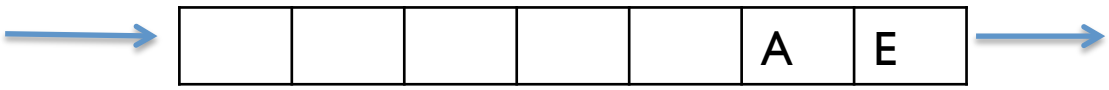
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



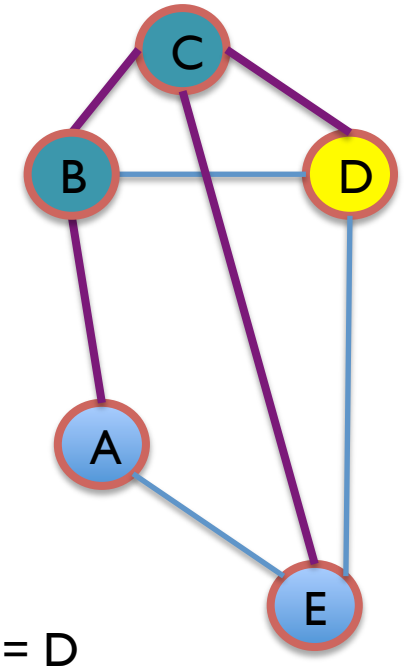
current = D

queue:



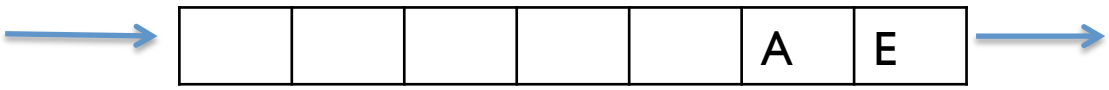
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



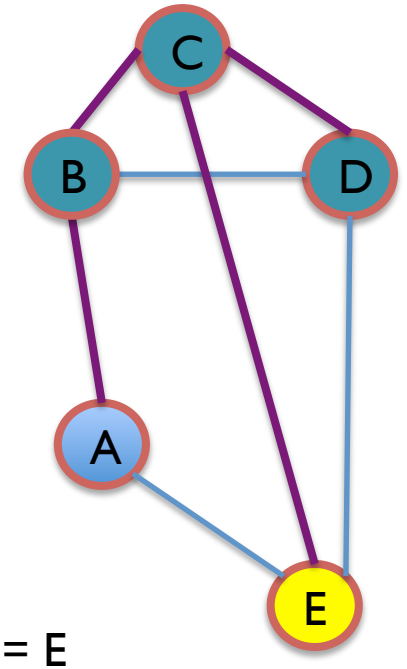
current = D

queue:



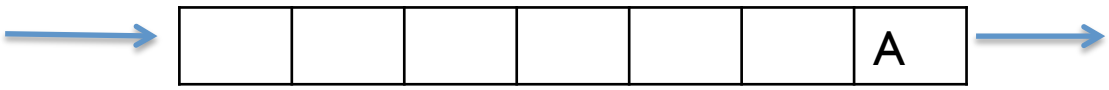
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



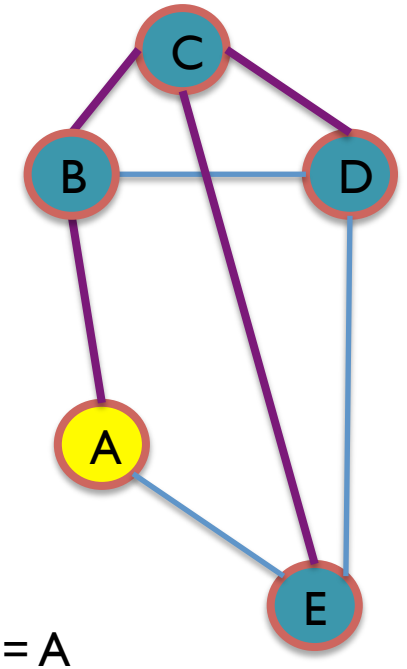
current = E

queue:



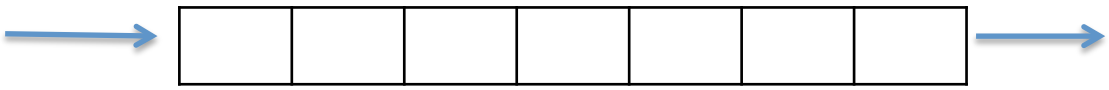
Breadth-First Search

```
BFS(G, s = C)
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```



current = A

queue:

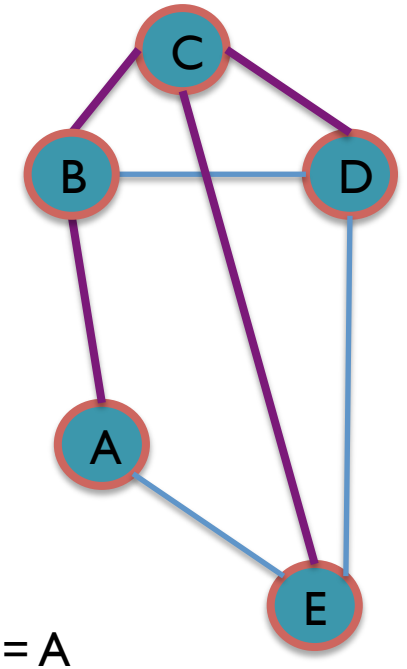


Breadth-First Search

BFS(G, s = C)

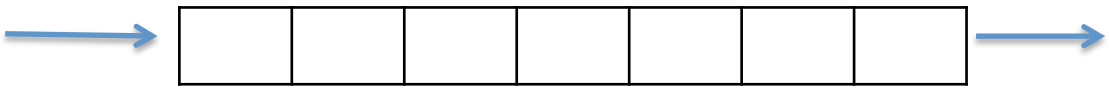
```
    mark s
    Put s in a queue
    WHILE the queue is not empty
        current = next vertex in queue (dequeue)
        FOR each neighbour v of current
            IF v is not marked
                parent of v = current
                mark v
                put v in the queue
```

DONE!



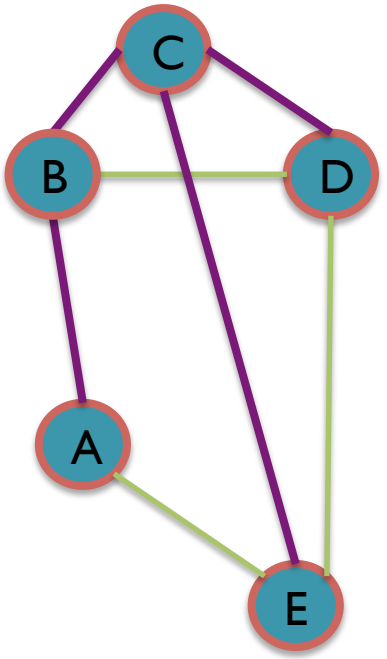
current = A

queue:



Breadth-First Search

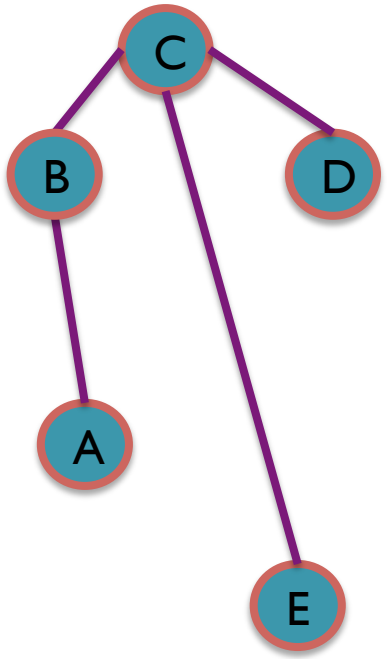
- Edges that would take cross path with a different branch of the BFS tree are call *cross edges*.



Breadth-First Search

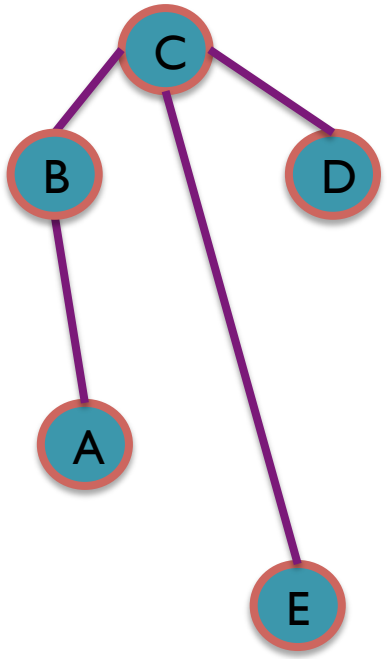
- Edges that would take cross path with a different branch of the BFS tree are call *cross edges*.

- Keeping only the tree edges yields the BFS spanning tree.



Breadth-First Search

- Edges that would take cross path with a different branch of the BFS tree are call *cross edges*.



- Keeping only the tree edges yields the BFS spanning tree.

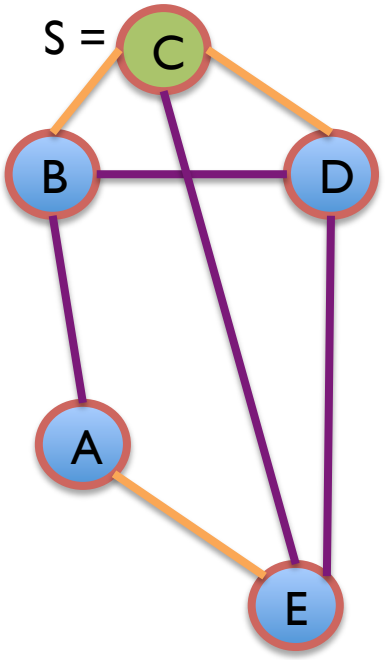
- This tree represent the shortest paths from the source, to every other vertices in the connected component.

Back Edges vs. Cross Edges

- What is the difference between back edges and *cross edges*?

- A back edge (as we have seen in DFS) is an edge that would take us back to an ancestor vertex.

- In this graph, vertex *C* is a *parent* of vertex *E*, who is a parent of *D*. Therefore, vertex *C* is an ancestor of vertex *D*.



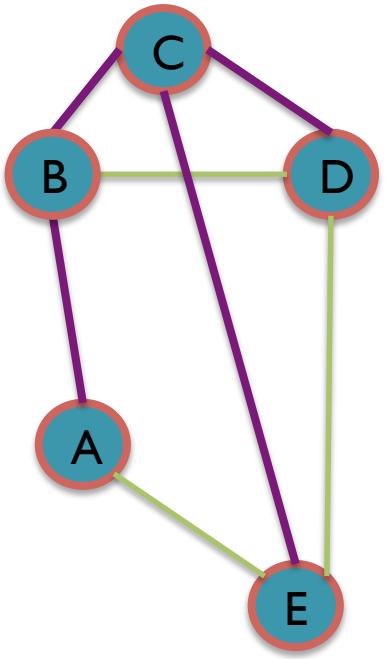
DFS

Back Edges vs. Cross Edges

- What is the difference between back edges and *cross edges*?

- A cross edge (as we have seen in BFS) is an edge that would take us to a vertex who is not an ancestor and has been visited (or marked) before.

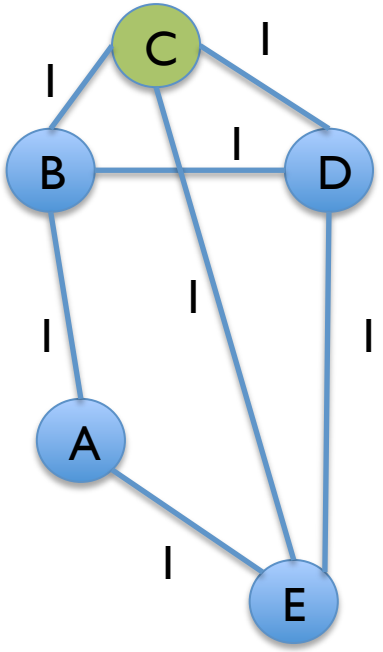
- In this graph, vertex *C* is a *parent* of both vertex *B*, and *D*. Therefore *B*, and *D* are *siblings*, and share a common ancestor, but one is not the ancestor of the other.



BFS

Weighted Graphs

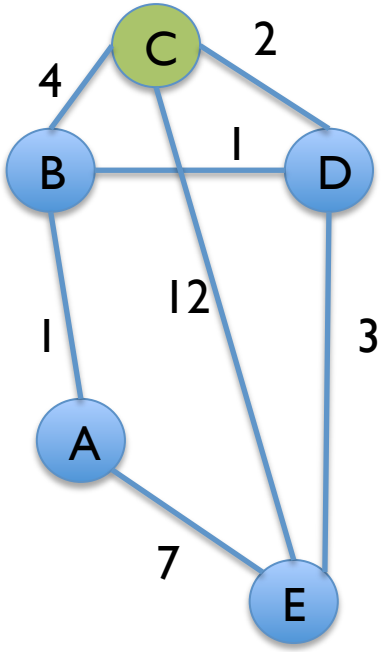
- So far, we looked at unweighted graphs.
  - The *(C,E)* edge is assumed to be as long as the *(C,B)* edge.
  - We can say that all edges have a weight of 1 (or all have equal weight).



Weighted Graphs

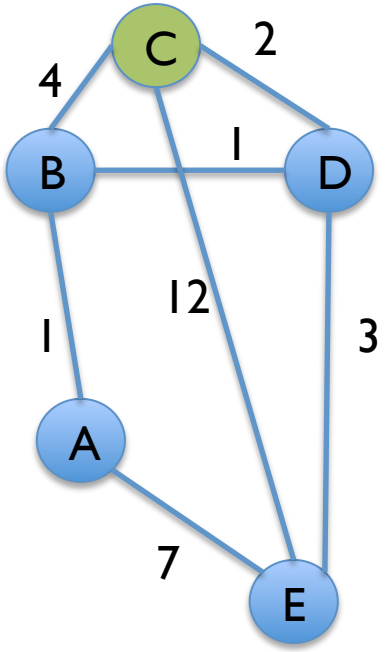
- Weighted graphs are often used to represent the *distance* between two things (literally and figuratively).

- They are also use to represent *cost* (again, both literally and figuratively).



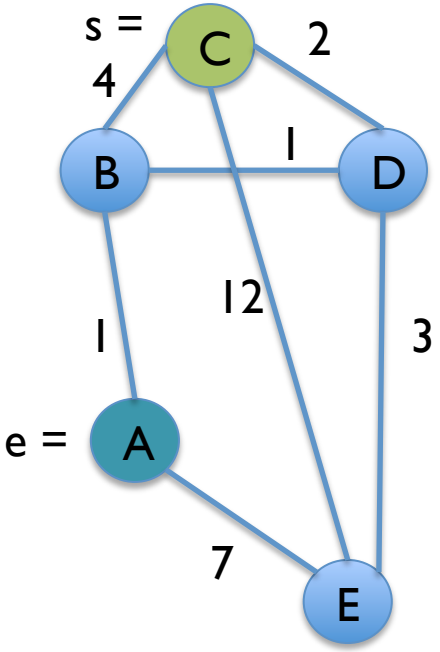
Finding the Shortest Path in Weighted Graphs

- Cities *A*, *B*, *C*, *D*, and *E*, have been relying on satellite for their internet services.
- An internet provider has decided to 'wire' the cities.
- Cable is expensive, so the company is looking for the cheapest route.



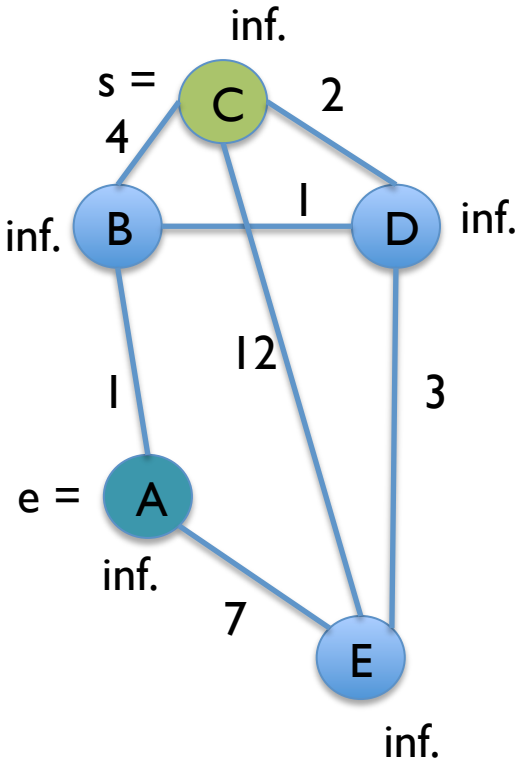
Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```



Dijkstra's Algorithm

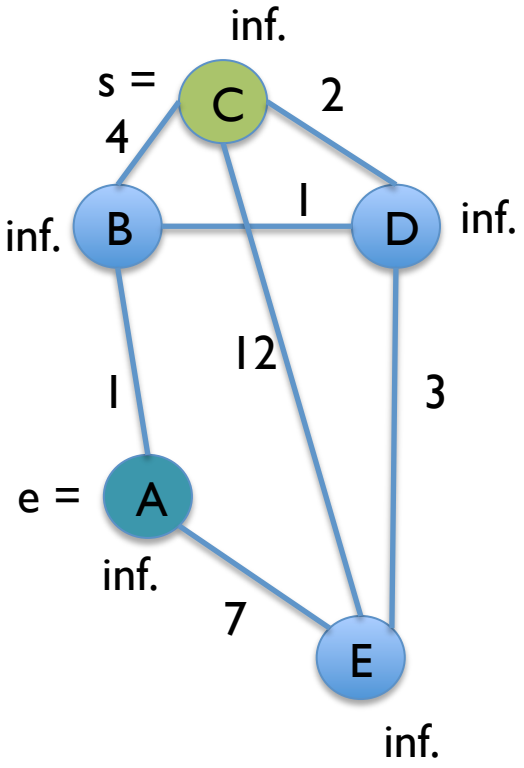
```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```



	A	B	C	D	E
marked					
dist.	inf.	inf.	inf.	inf.	inf.
prev.	null	null	null	null	null

Dijkstra's Algorithm

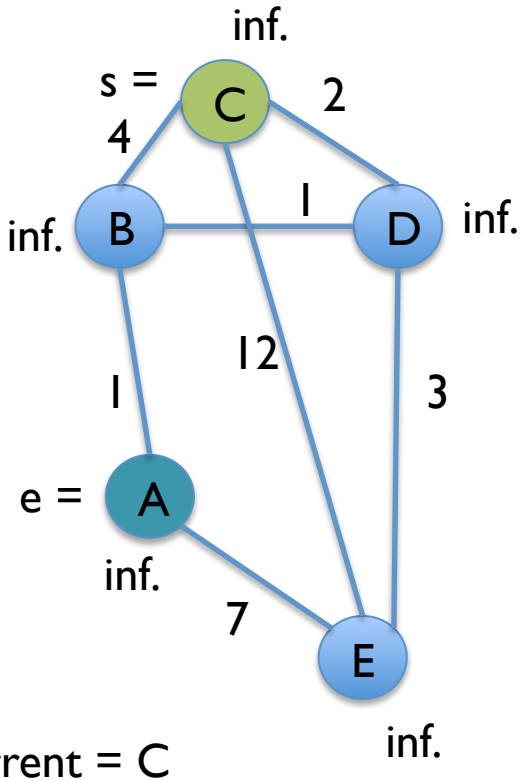
```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```



	A	B	C	D	E
marked					
dist.	inf.	inf.	0	inf.	inf.
prev.	null	null	null	null	null

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

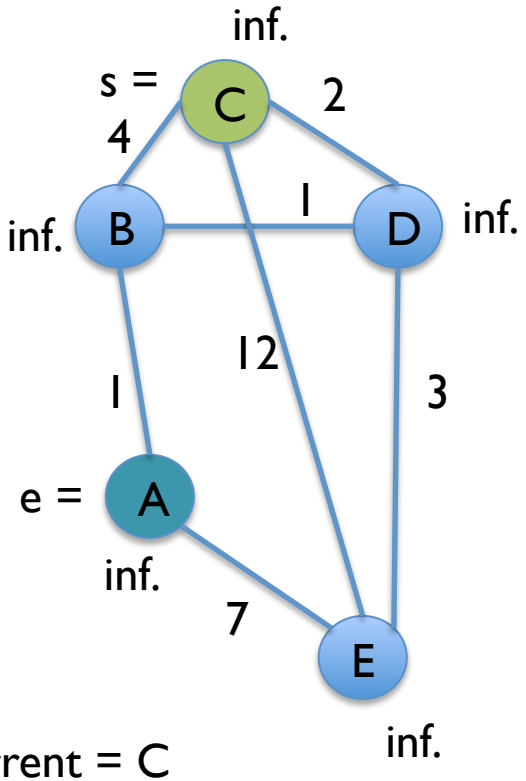


current = C

	A	B	C	D	E
marked					
dist.	inf.	inf.	0	inf.	inf.
prev.	null	null	null	null	null

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

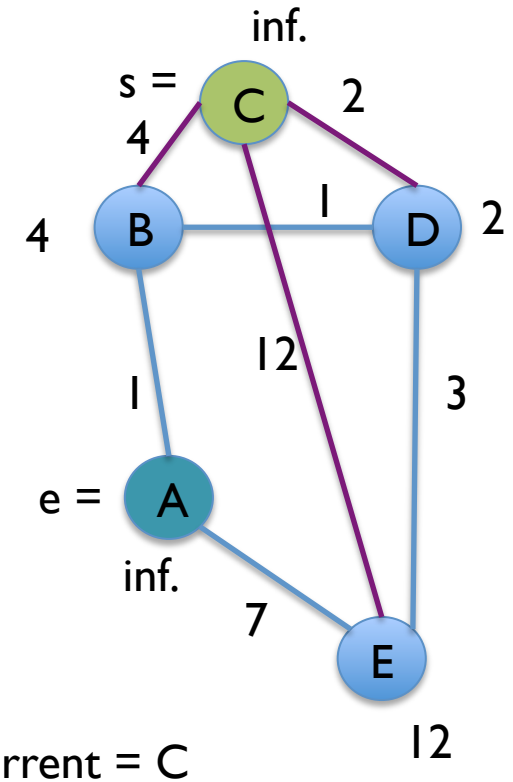


current = C

	A	B	C	D	E
marked					
dist.	inf.	inf. (4)	0	inf. (2)	inf.
prev.	null	null	null	null	null

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

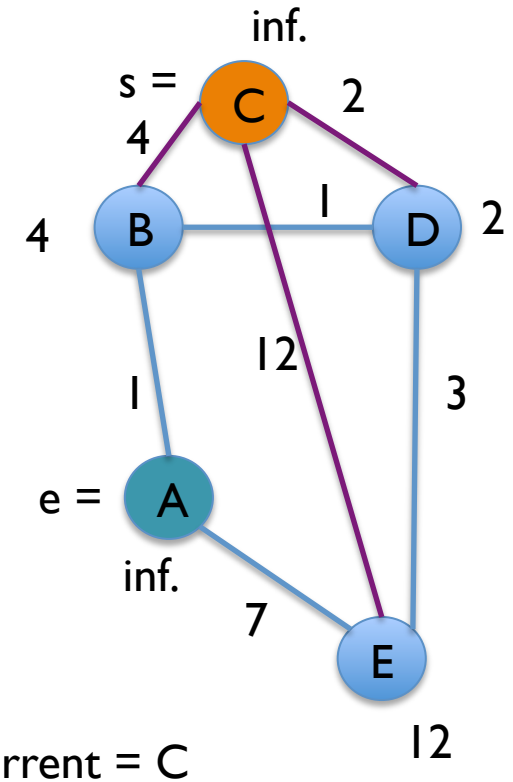


current = C

	A	B	C	D	E
marked					
dist.	inf.	4	0	2	12
prev.	null	C	null	C	C

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

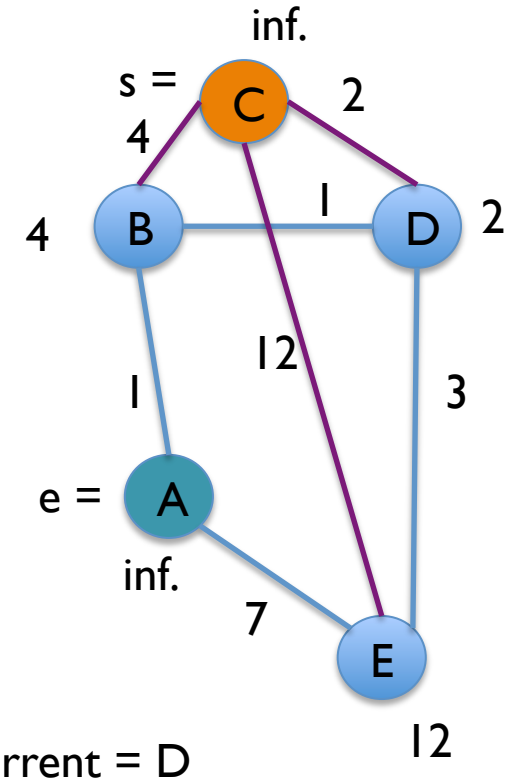


current = C

	A	B	C	D	E
marked			x		
dist.	inf.	4	0	2	12
prev.	null	C	null	C	C

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

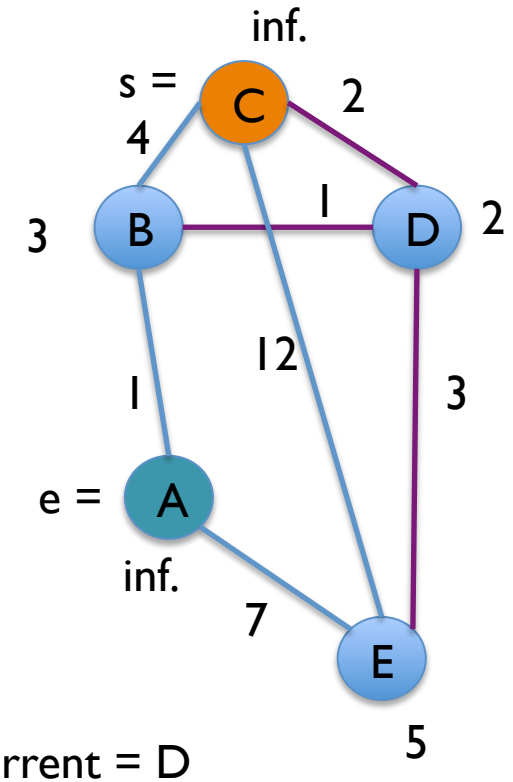


current = D

	A	B	C	D	E
marked			x		
dist.	inf.	4<(2+1)?	0	2	12<(2+3)?
prev.	null	C	null	C	C

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

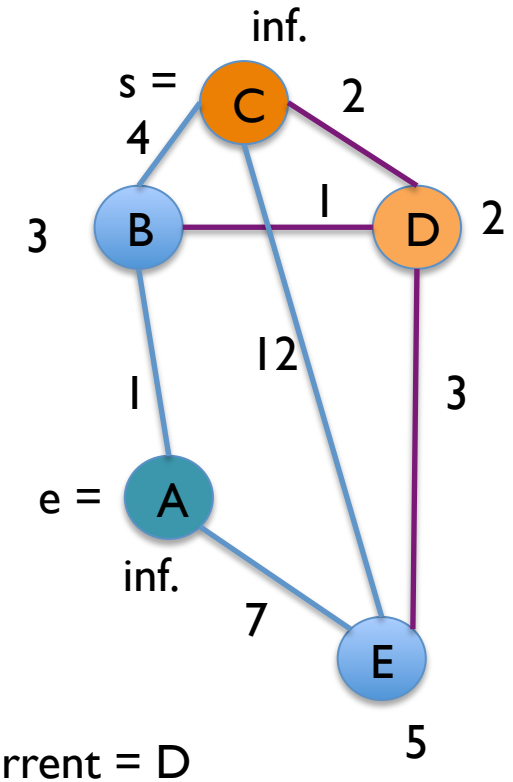


current = D

	A	B	C	D	E
marked			x		
dist.	inf.	3	0	2	5
prev.	null	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

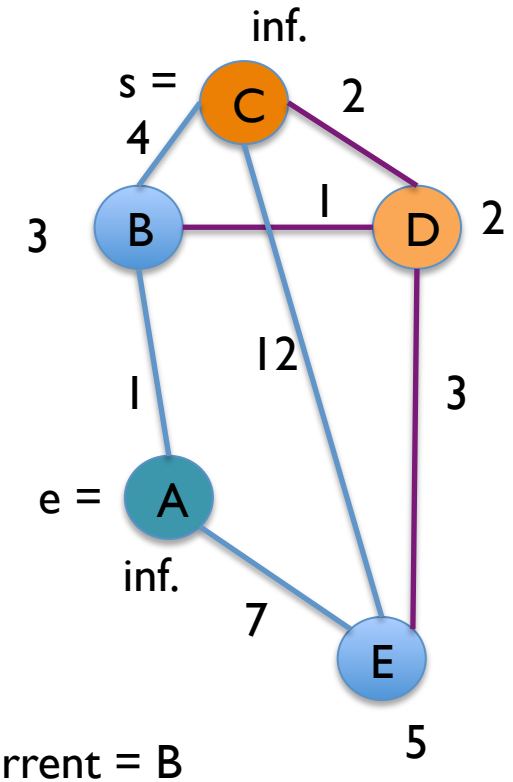


current = D

	A	B	C	D	E
marked			x	x	
dist.	inf.	3	0	2	5
prev.	null	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

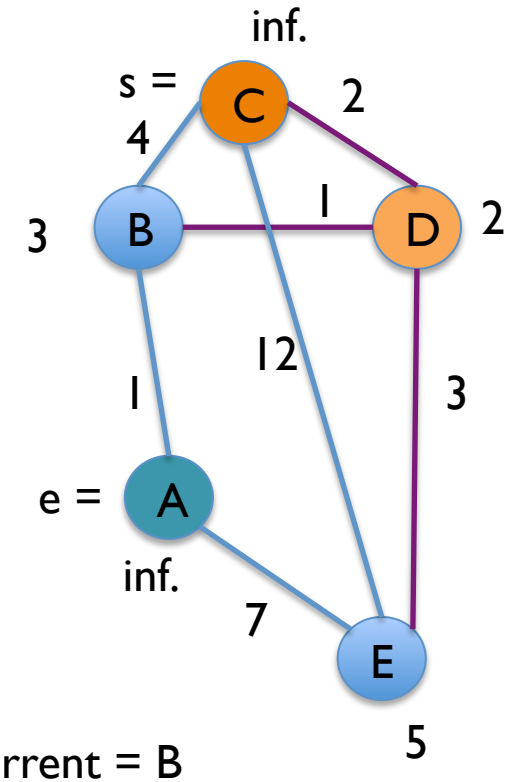


current = B

	A	B	C	D	E
marked			x	x	
dist.	inf.	3	0	2	5
prev.	null	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

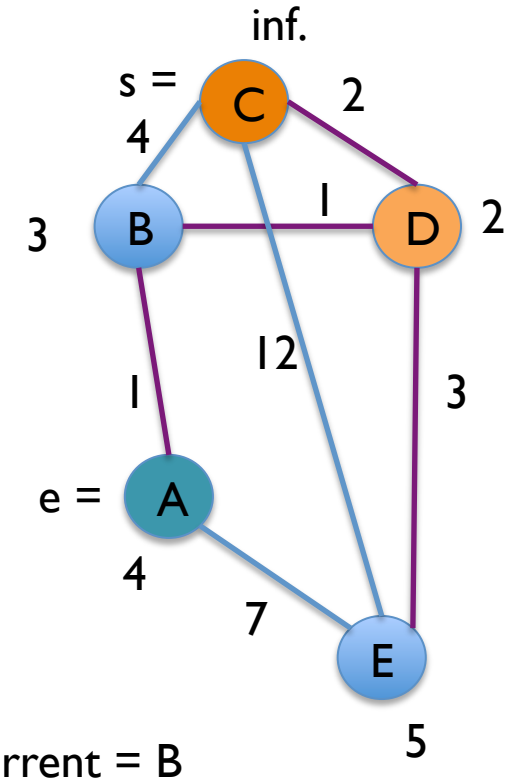


current = B

	A	B	C	D	E
marked			x	x	
dist.	inf.(3+1)	3	0	2	5
prev.	null	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

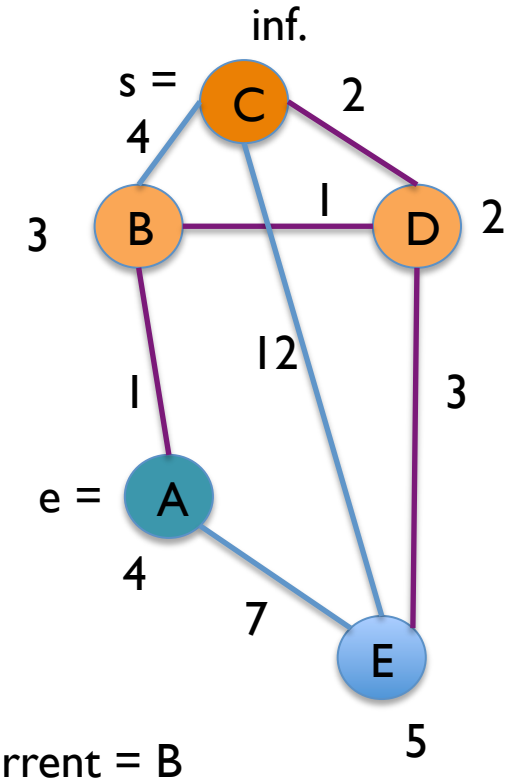


current = B

	A	B	C	D	E
marked			x	x	
dist.	4	3	0	2	5
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

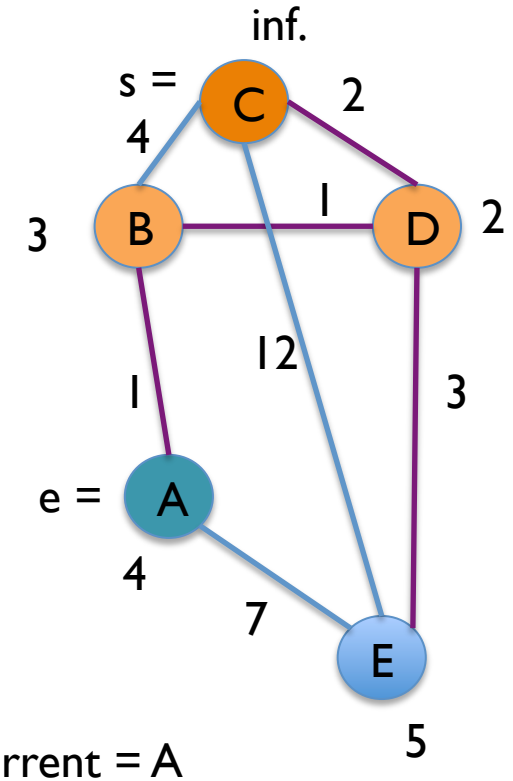


current = B

	A	B	C	D	E
marked		x	x	x	
dist.	4	3	0	2	5
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

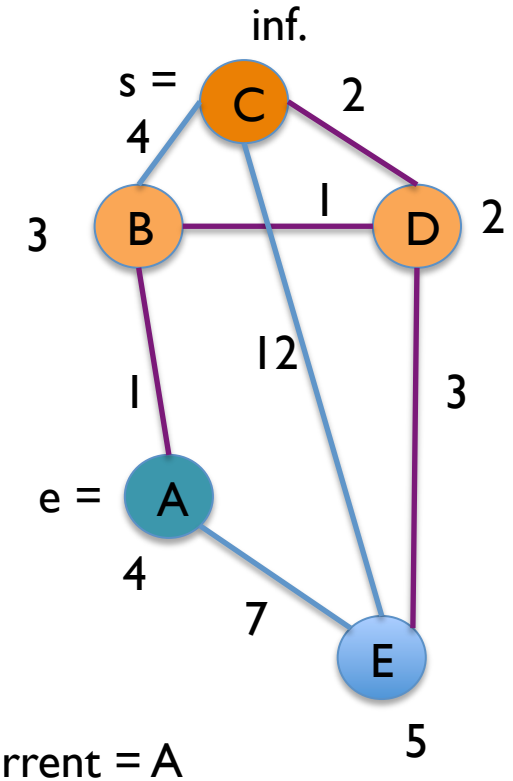


current = A

	A	B	C	D	E
marked		x	x	x	
dist.	4	3	0	2	5
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

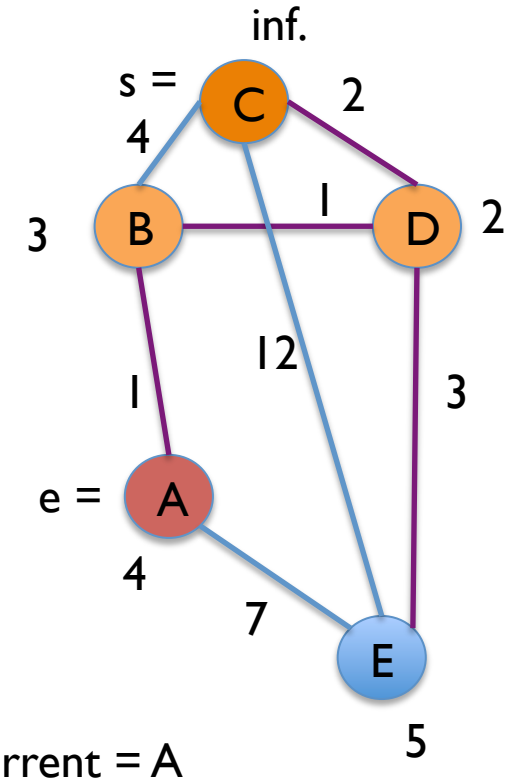


current = A

	A	B	C	D	E
marked		x	x	x	
dist.	4	3	0	2	5 (4+7)
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

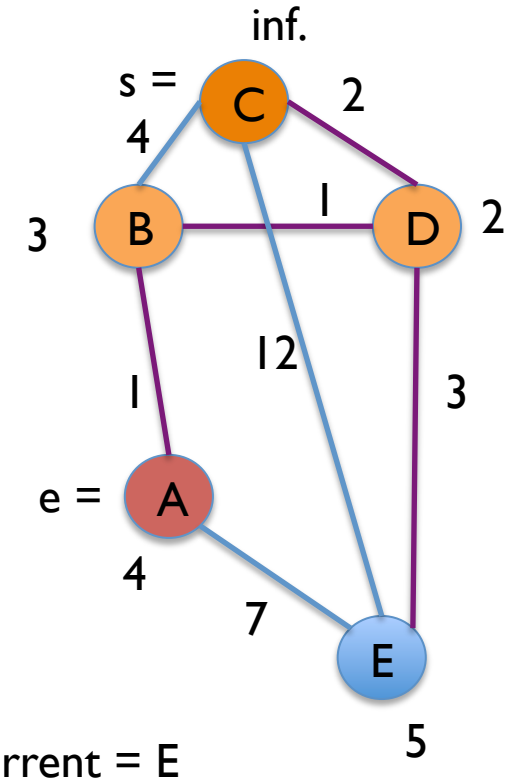


current = A

	A	B	C	D	E
marked	x	x	x	x	
dist.	4	3	0	2	5
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```

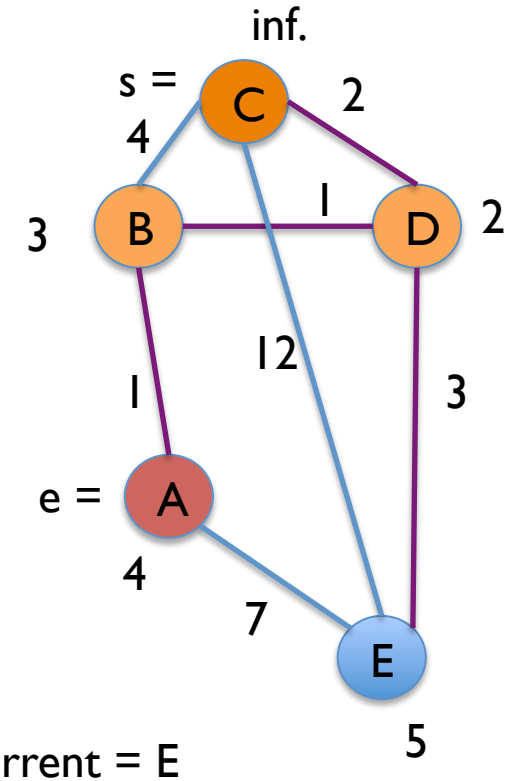


current = E

	A	B	C	D	E
marked	x	x	x	x	x
dist.	4	3	0	2	5
prev.	B	D	null	C	D

Dijkstra's Algorithm

```
Dijkra(Graph G, Source vertex s, End vertex e)
FOR each vertex in V
    distance[v] = infinity  // distance from s to vertex v
    previous[v] = undefined
distance[s] = 0 // distance from s to s
WHILE some vertex are not marked
    current = unmarked vertex with smallest distance
    FOR each unmarked neighbour n of current
        Calculate distance from s coming from current
        IF calculated distance < distance[n]
            distance[n] = calculated distance
            previous of n = current
    mark current
return distance[ ], previous[ ]
```



current = E

	A	B	C	D	E
marked	x	x	x	x	x
dist.	4	3	0	2	5
prev.	B	D	null	C	D