Graphs Traversal and Topological Sort

Traversing Graphs

Goal: visit each node in a graph in a systematic way

Non trivial because:

- Non-linear
- Non-hierarchical

Breadth-first Search

Exploring one layer at a time

- Nodes are visited in order of increasing distance from s
- First visit nodes at distance 1 from s
- Then visit nodes at distance 2 from s
- Continue until all nodes have been visited.

How can this be achieved?

- Put nodes that have been discovered but not yet explored in a **queue**
- Keep a record of which nodes have been discovered
- Construct a search tree that records search

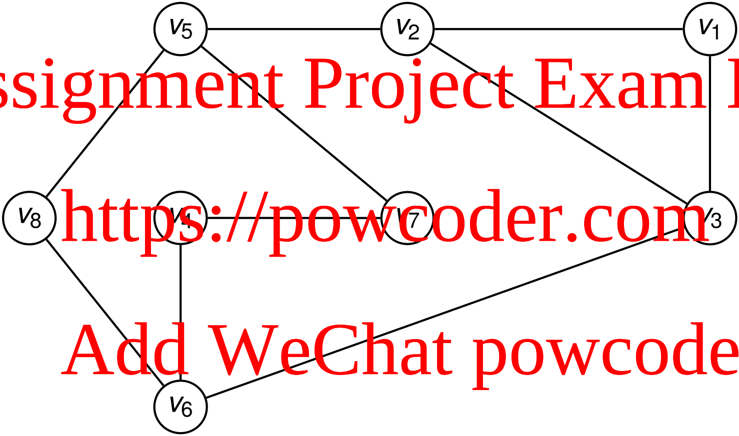
Breadth-first Search

$BFS(G, s)$:

let Q be a queue containing just the node s
let $discovered(s) = true$
let $discovered(v) = false$ *for all* $v \in V - \{s\}$
let $T = (V, \{\})$

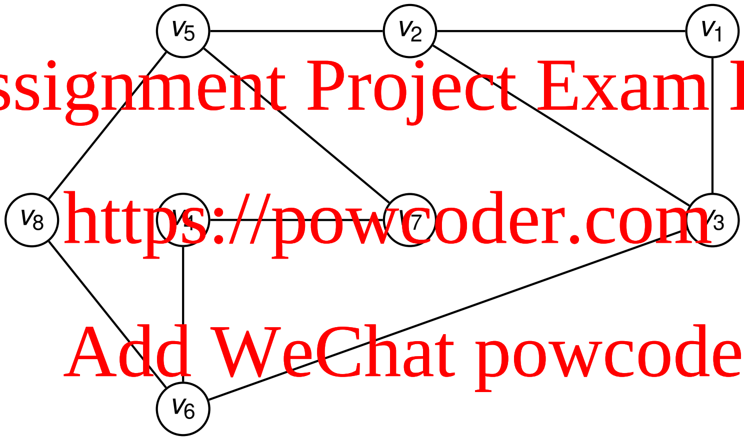
while Q is not empty
    remove v from the front of Q
    for each edge $\{v, w\}$ *in E where not* $discovered(w)$
        let $discovered(w) = true$
        add w to the back of Q
        add edge $\{v, w\}$ *to edges in T*

Illustration of BFS



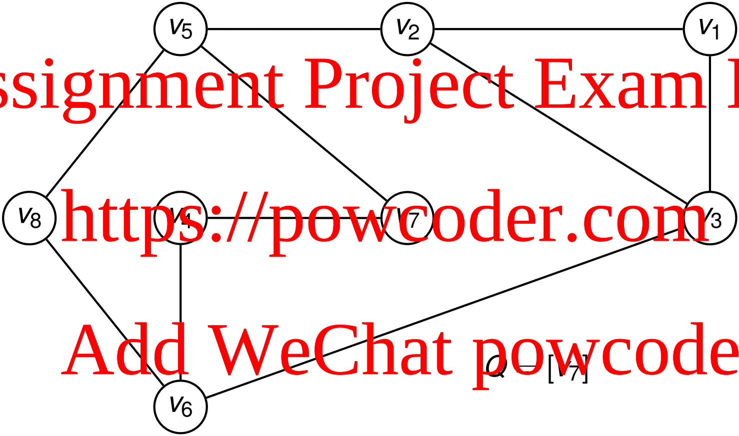
Run BFS starting at v_7

Illustration of BFS



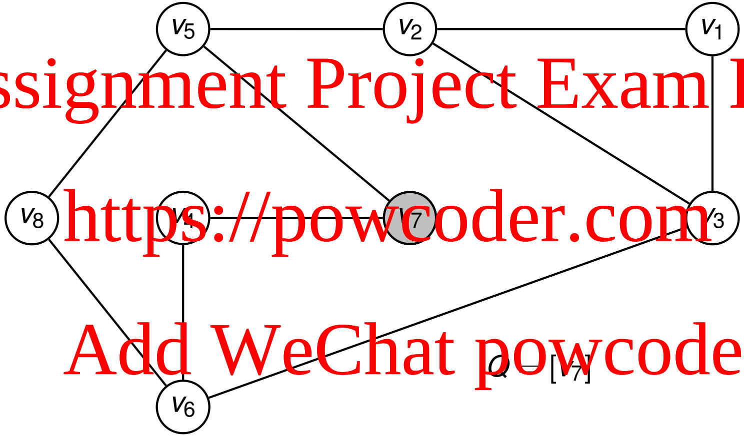
Initialize Q

Illustration of BFS



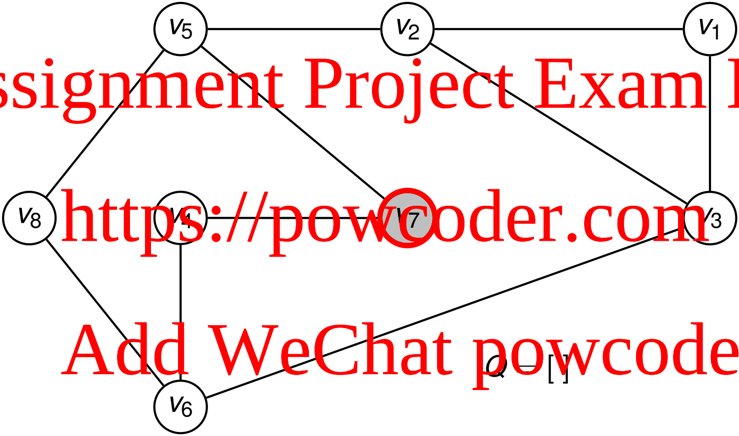
Let $discovered(v_7) = true$

Illustration of BFS



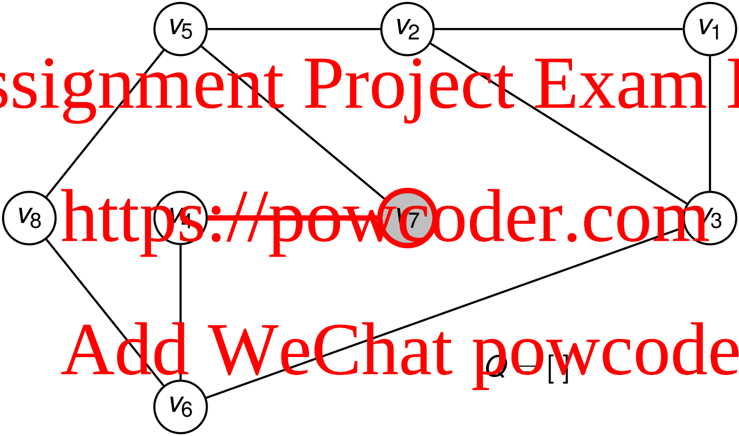
Remove v_7 from front of Q

Illustration of BFS



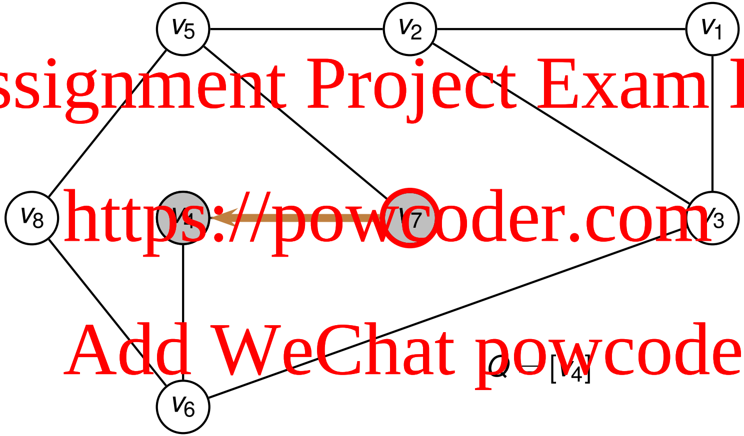
Consider edge $\{v_7, v_4\}$ since v_4 not yet discovered

Illustration of BFS



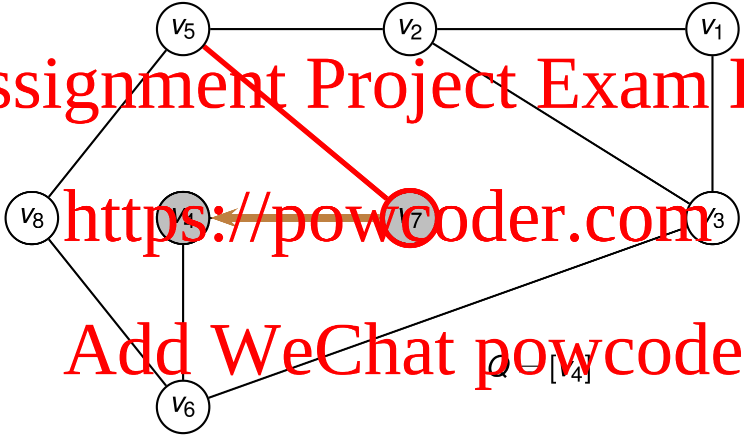
Let $discovered(v_4) = true$, add $\{v_7, v_4\}$ to T and v_4 to Q

Illustration of BFS



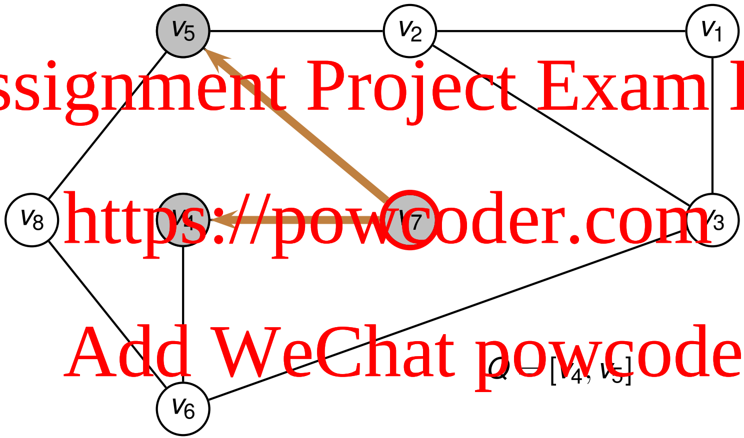
Consider edge $\{v_7, v_5\}$ since v_5 not yet discovered

Illustration of BFS



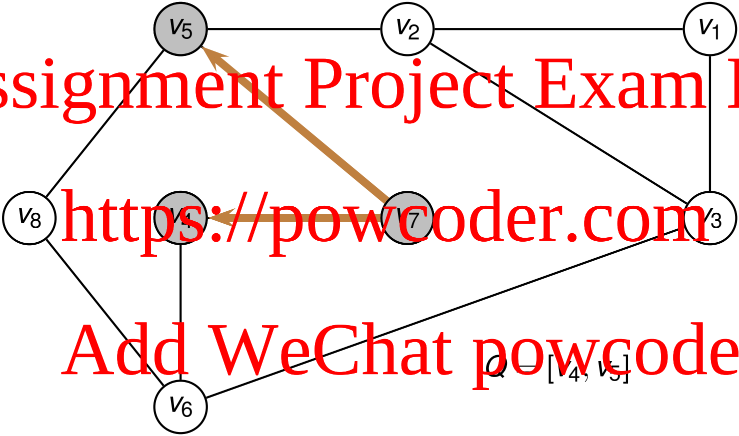
Let *discovered*$(v_5) =$ *true*, add $\{v_7, v_5\}$ to T and v_5 to Q

Illustration of BFS



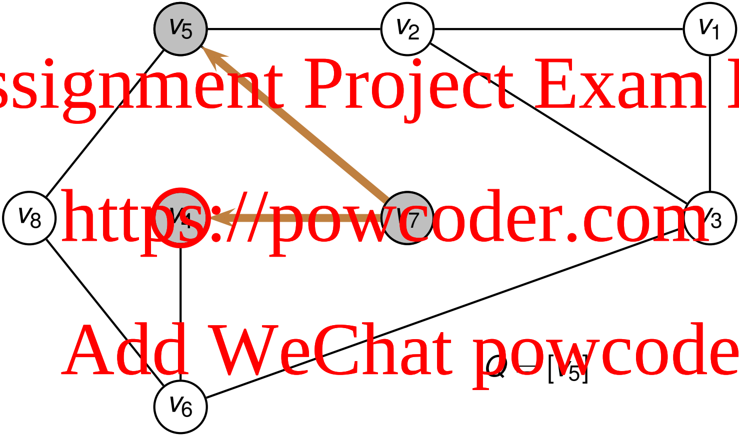
No more edges from v_7

Illustration of BFS



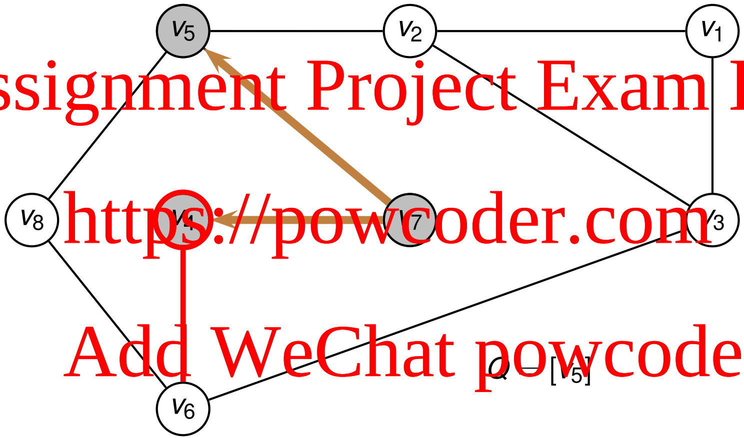
Remove v_4 from front of Q

Illustration of BFS



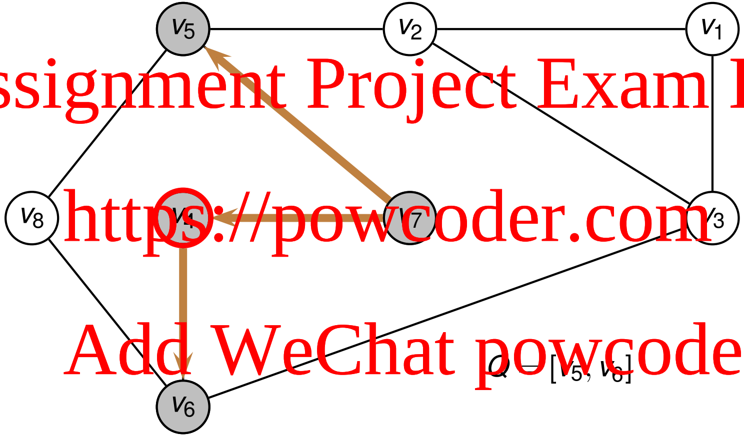
Consider edge $\{v_4, v_6\}$ because v_6 not yet discovered

Illustration of BFS



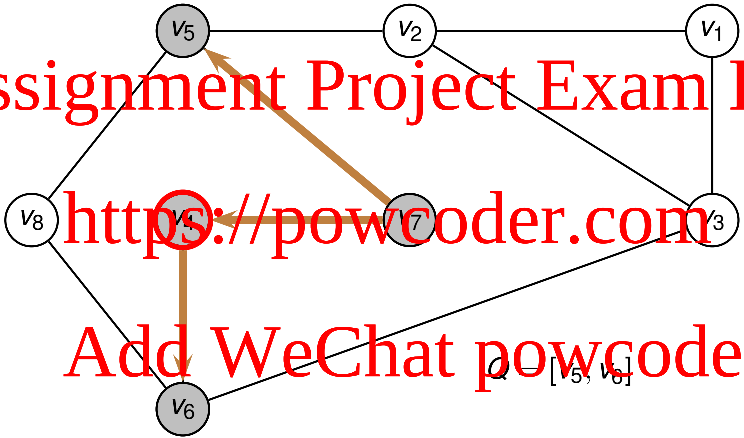
Let *discovered*$(v_6) =$ *true*, add $\{v_4, v_6\}$ to T and v_6 to Q

Illustration of BFS



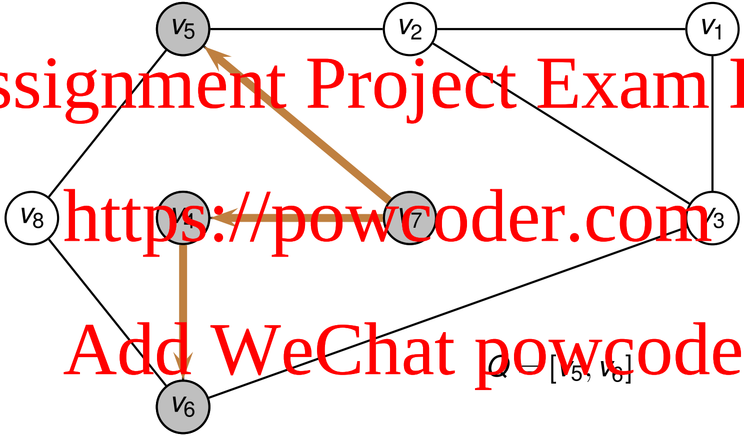
Don't consider $\{v_4, v_7\}$ because v_7 already discovered

Illustration of BFS



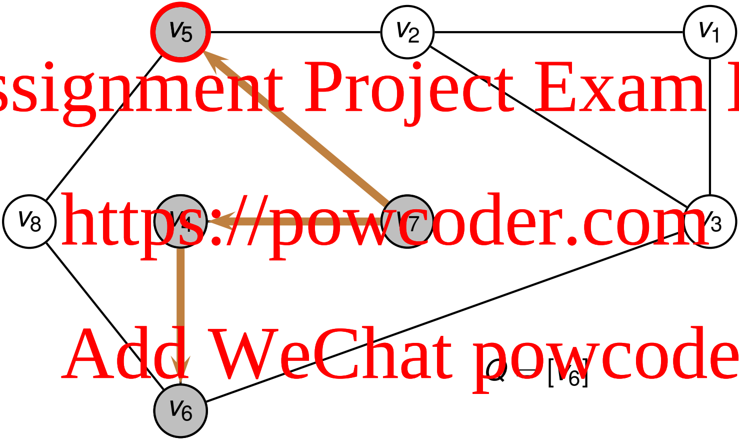
No more edges from v_4

Illustration of BFS



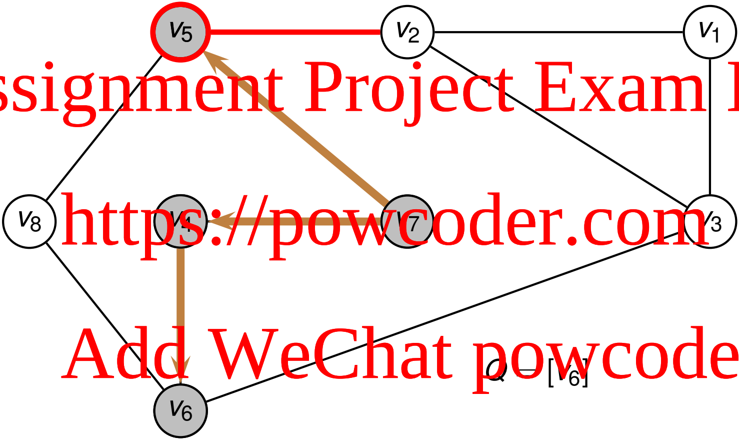
Remove v_5 from front of Q

Illustration of BFS



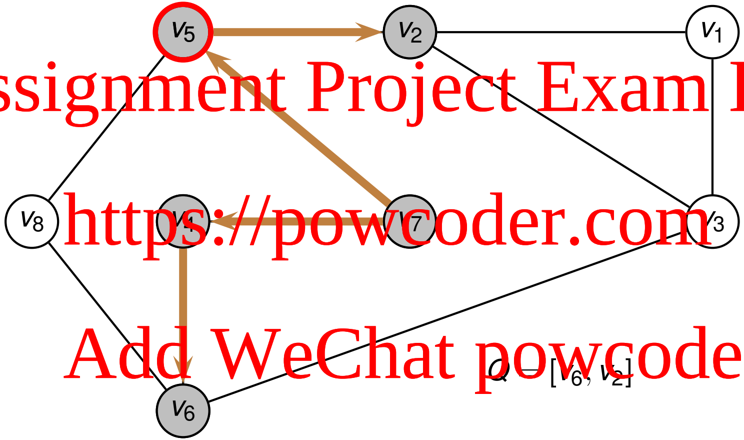
Consider edge $\{v_5, v_2\}$ because v_2 not yet discovered

Illustration of BFS



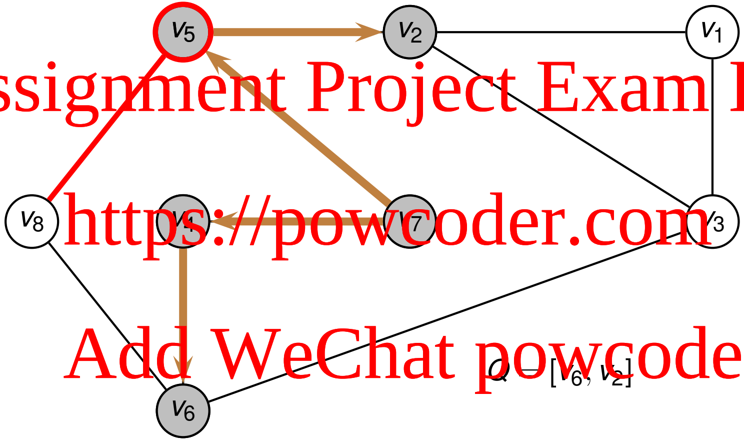
Let *discovered*$(v_2) = $ *true*, add $\{v_5, v_2\}$ to T and v_2 to Q

Illustration of BFS



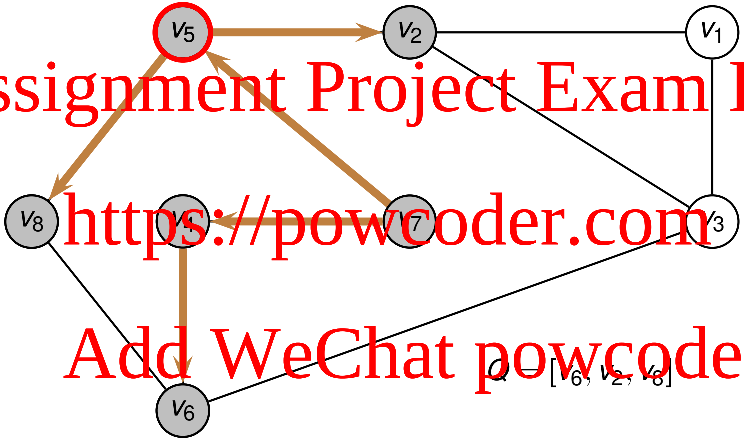
Consider $\{v_5, v_8\}$ because v_8 not yet discovered

Illustration of BFS



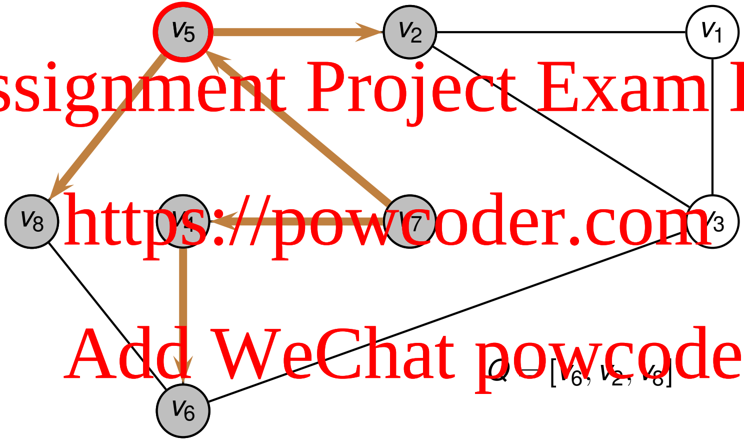
Let *discovered*$(v_8) =$ *true*, add $\{v_5, v_8\}$ to T and v_8 to Q

Illustration of BFS



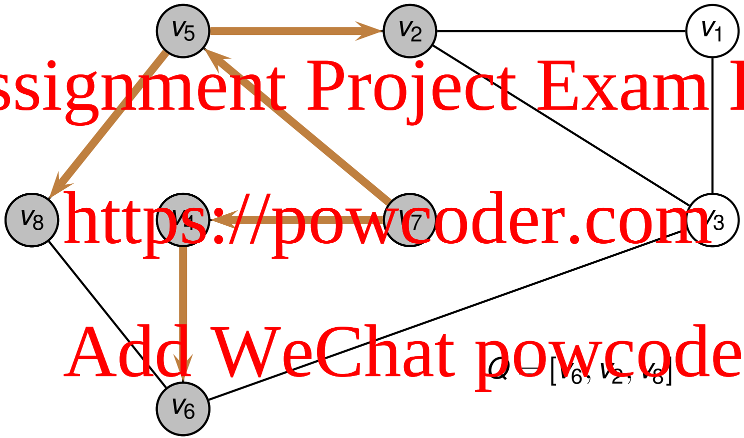
Don't consider $\{v_5, v_7\}$ because v_7 already discovered

Illustration of BFS



No more edges from v_5

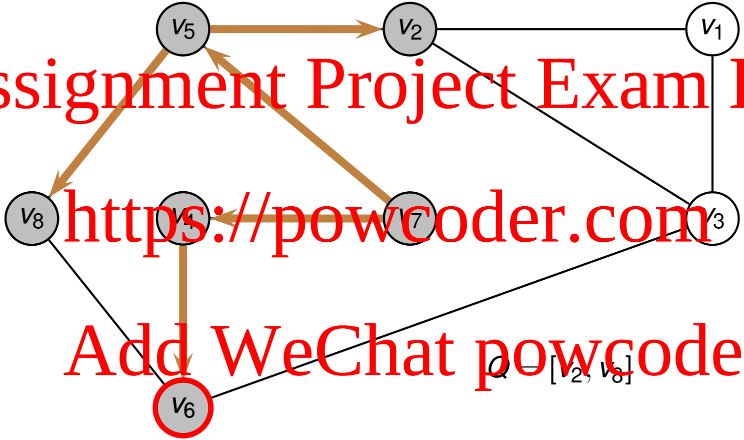
Illustration of BFS



$Q = [v_6, v_2, v_8]$

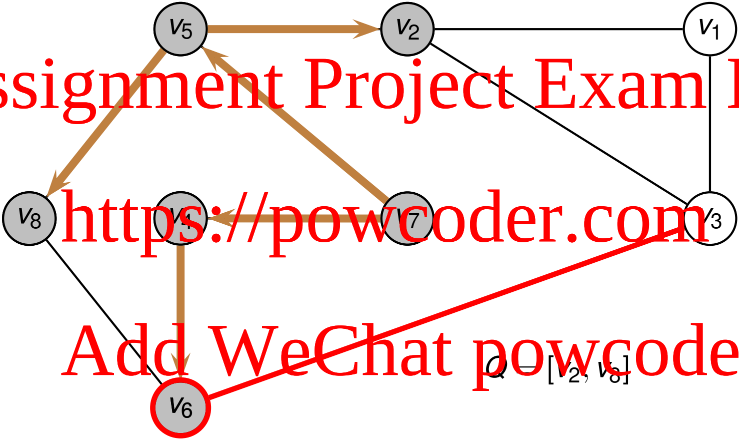
Remove v_6 from front of Q

Illustration of BFS



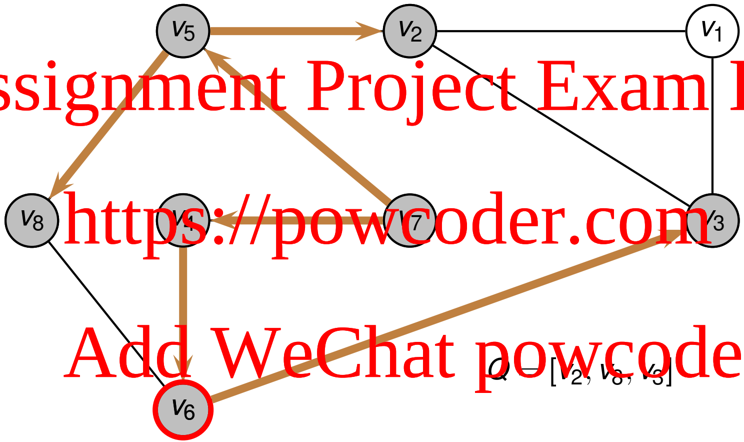
Consider edge $\{v_6, v_3\}$ because v_3 not yet discovered

Illustration of BFS



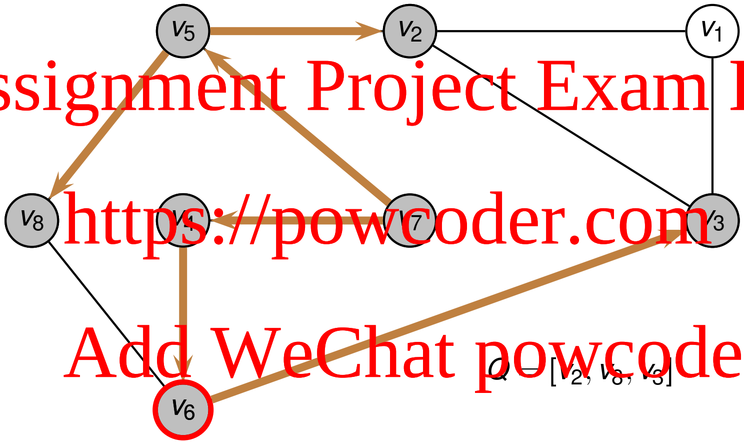
Let $\mathit{discovered}(v_3) = \mathit{true}$, add $\{v_6, v_3\}$ to T and v_3 to Q

Illustration of BFS



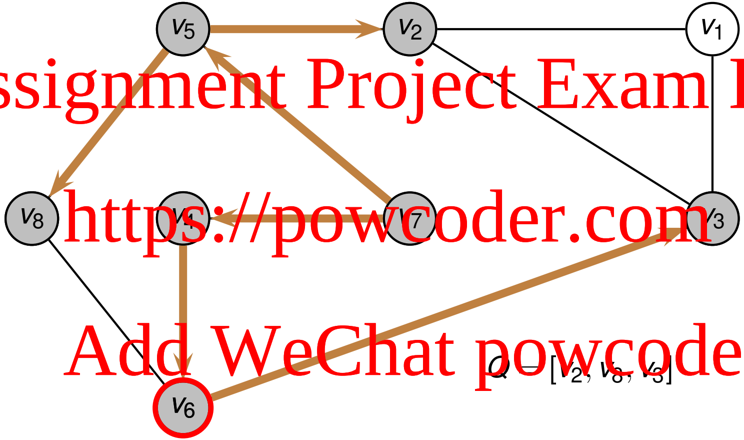
Don't consider $\{v_6, v_4\}$ because v_4 already discovered

Illustration of BFS



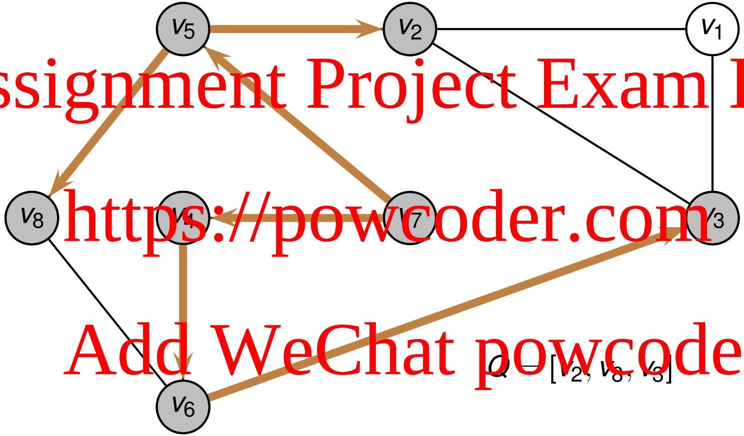
Don't consider $\{v_6, v_8\}$ because v_8 already discovered

Illustration of BFS



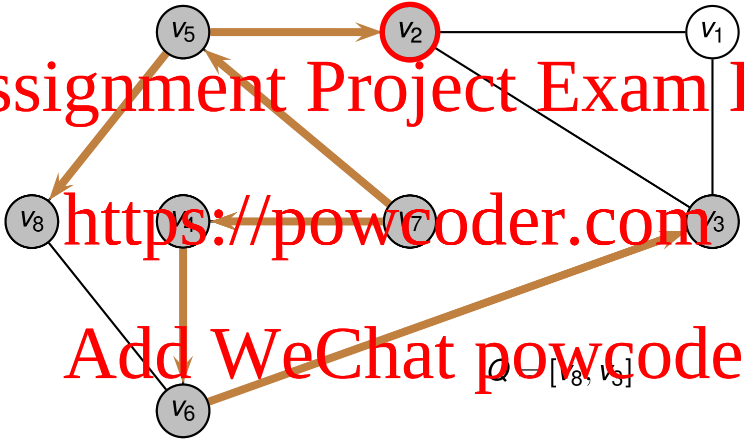
No more edges from v_6

Illustration of BFS



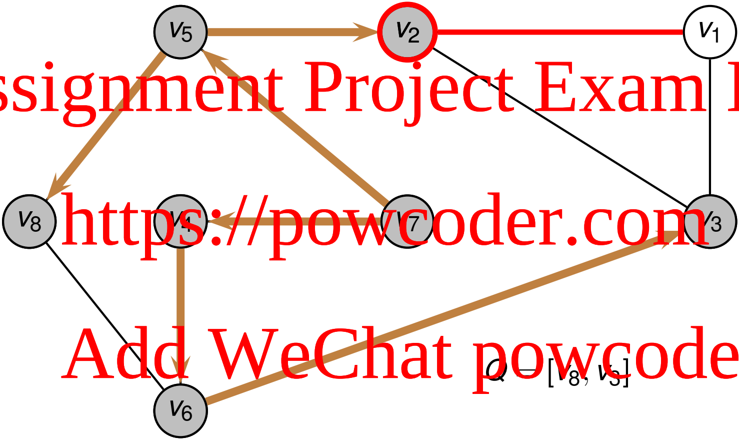
Remove v_2 from front of Q

Illustration of BFS



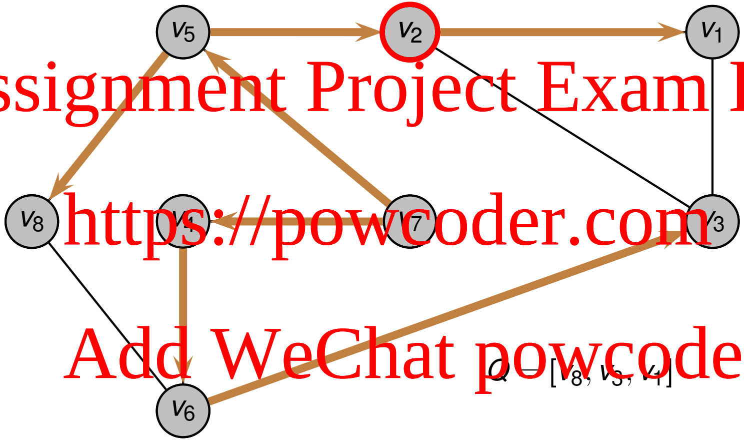
Consider edge $\{v_2, v_1\}$ because v_1 not yet discovered

Illustration of BFS



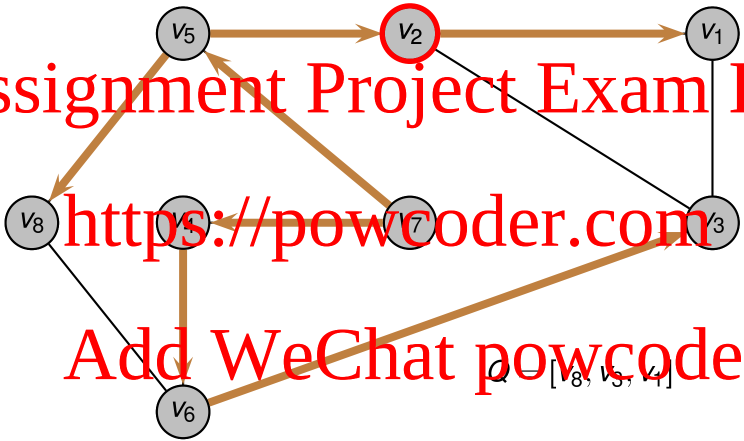
Let $discovered(v_1) = true$, add $\{v_2, v_1\}$ to T and v_1 to Q

Illustration of BFS



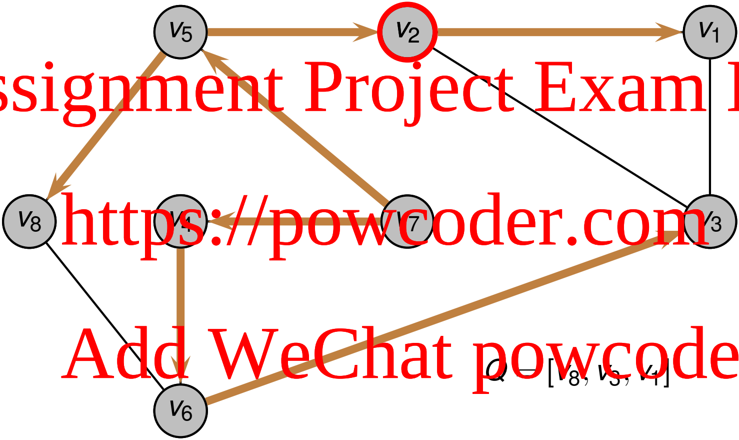
Don't consider $\{v_2, v_3\}$ because v_3 already discovered

Illustration of BFS



Don't consider $\{v_2, v_5\}$ because v_5 already discovered

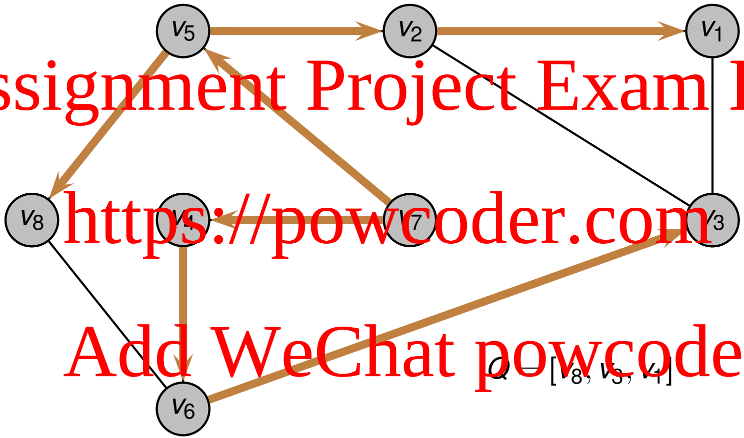
Illustration of BFS



$Q = [v_8, v_3, v_1]$

No more edges from v_2

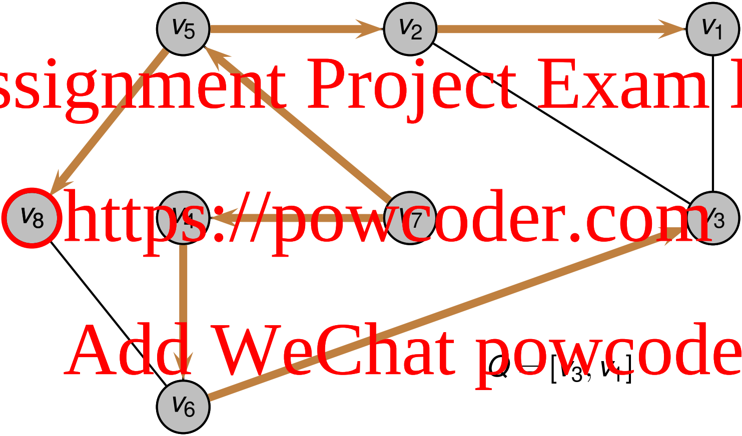
Illustration of BFS



$Q = [v_8, v_3, v_1]$

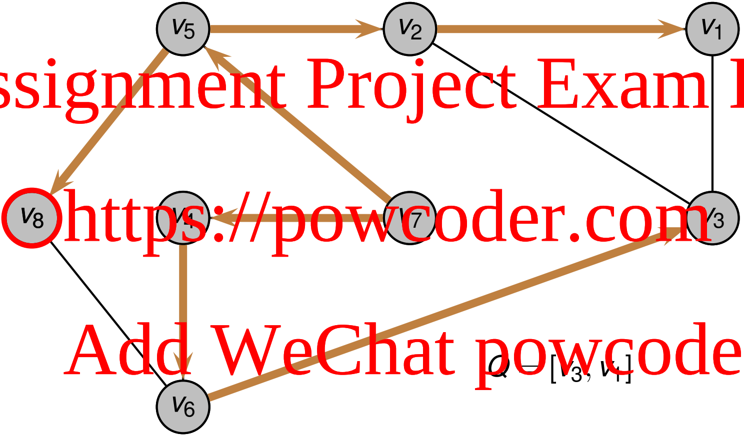
Remove v_8 from front of Q

Illustration of BFS



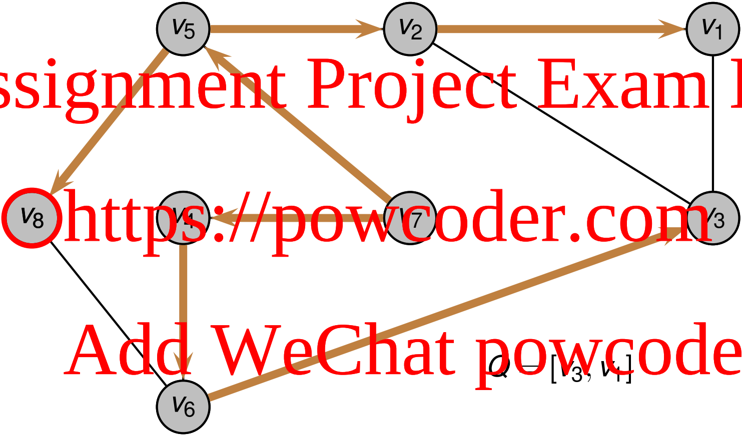
Don't consider $\{v_8, v_5\}$ because v_5 already discovered

Illustration of BFS



Don't consider $\{v_8, v_6\}$ because v_6 already discovered

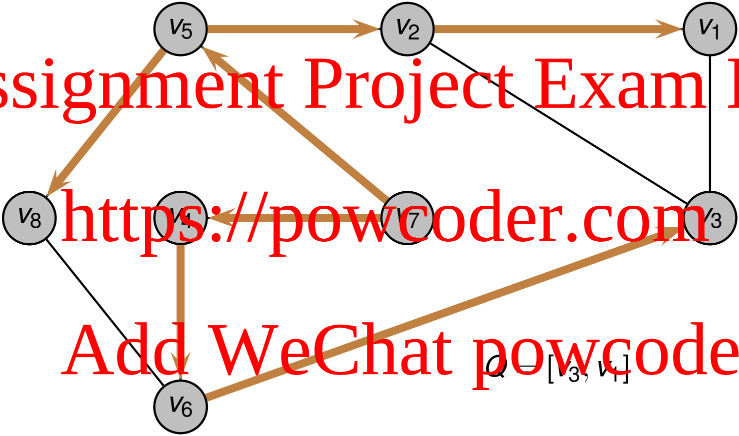
Illustration of BFS



$Q = [v_3, v_4]$

No more edges from v_8

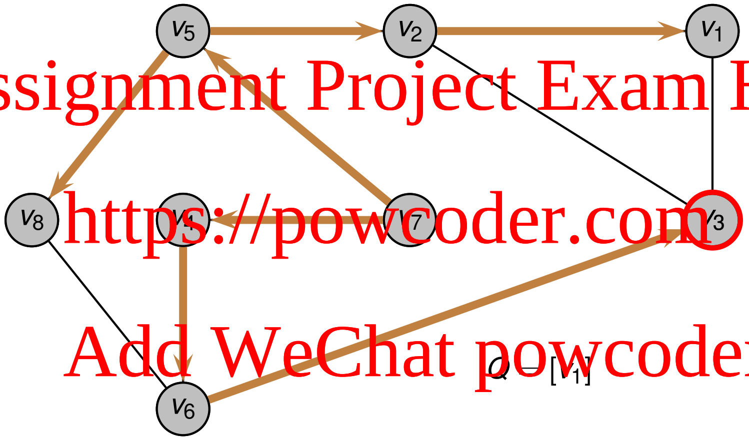
Illustration of BFS



$Q = [v_3, v_1]$

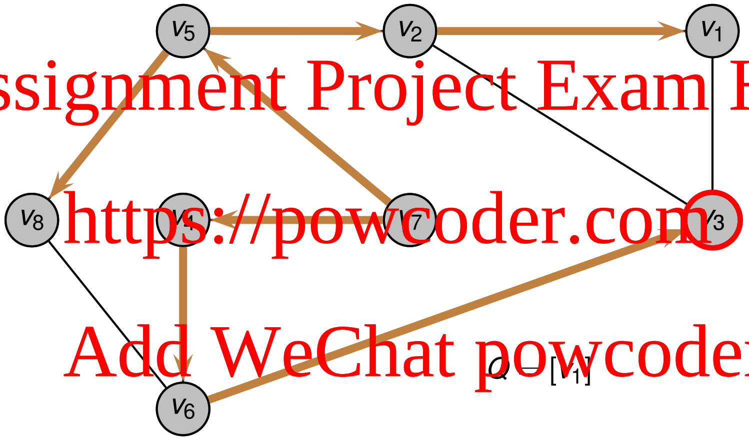
Remove v_3 from front of Q

Illustration of BFS



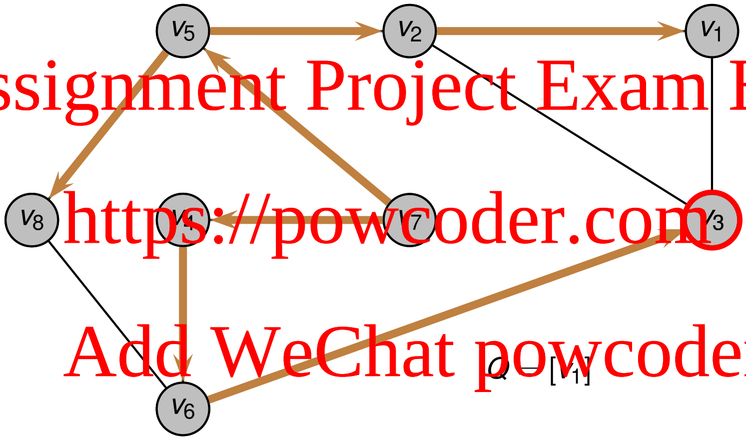
Don't consider $\{v_3, v_1\}$ because v_1 already discovered

Illustration of BFS



Don't consider $\{v_3, v_2\}$ because v_2 already discovered

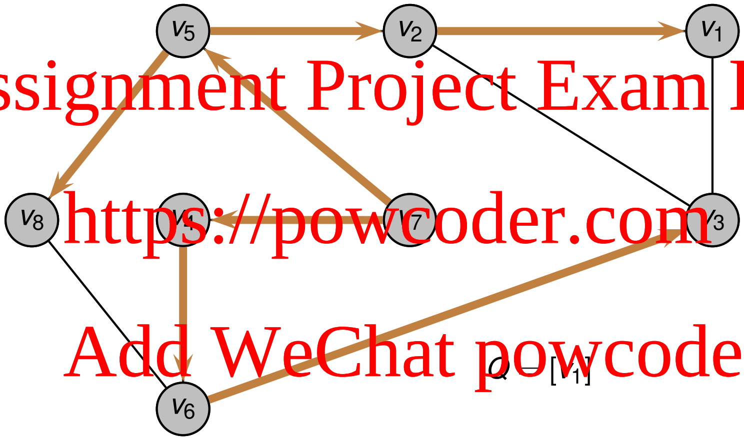
Illustration of BFS



$Q = [v_1]$

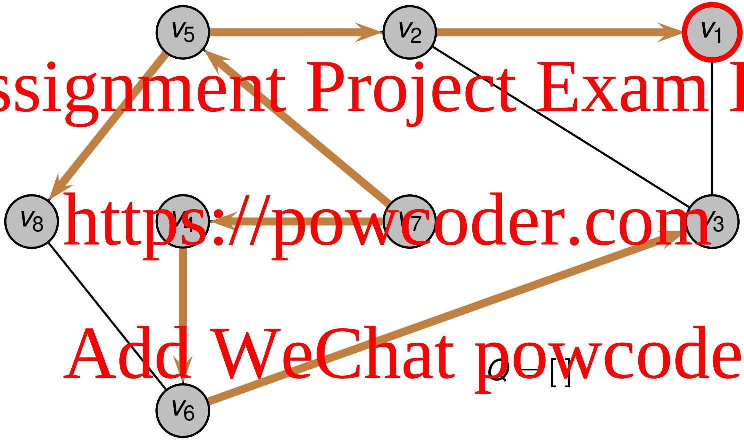
No more edges from v_3

Illustration of BFS



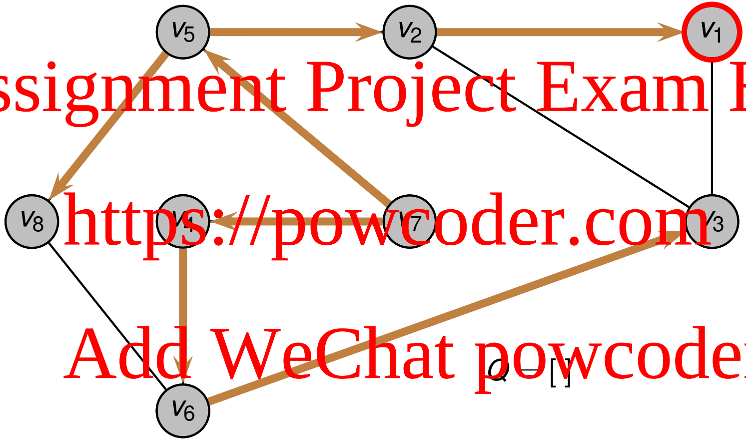
Remove v_1 from front of Q

Illustration of BFS



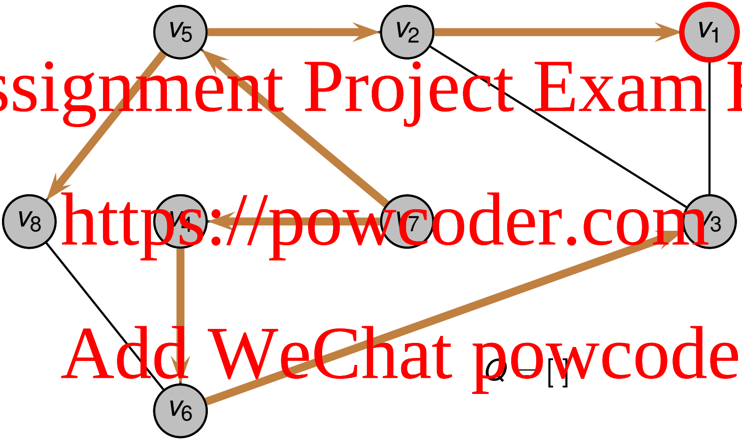
Don't consider $\{v_1, v_2\}$ because v_2 already discovered

Illustration of BFS



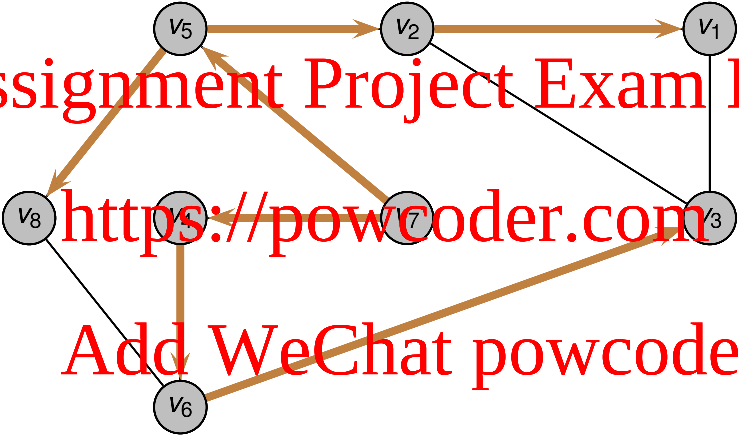
Don't consider $\{v_1, v_3\}$ because v_3 already discovered

Illustration of BFS



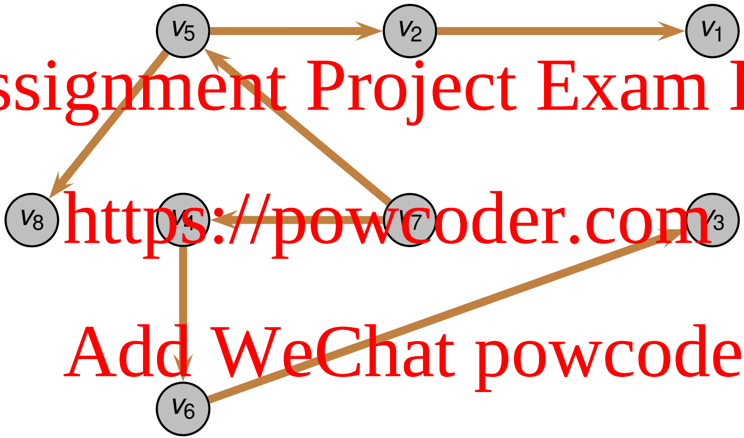
No more edges from v_1

Illustration of BFS



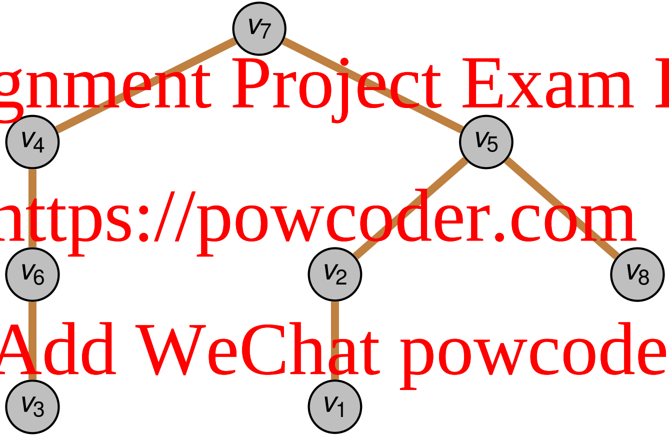
Q is empty so traversal complete

Illustration of BFS



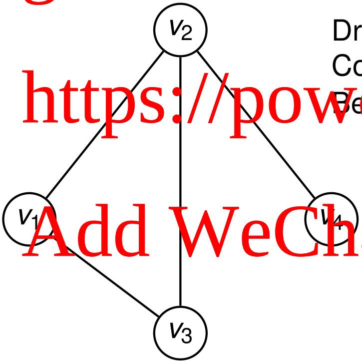
Here is the breadth first search tree for this run

Illustration of BFS



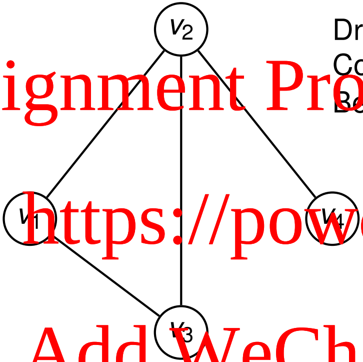
Same tree, but arranged in more usual way

Example for You



Draw the breadth-first search tree
Consider vertices in order of subscript
Begin traversal at v_1

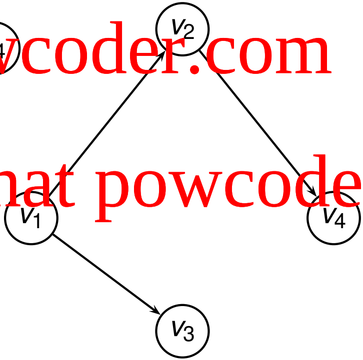
Example for You



Draw the breadth-first search tree

Consider vertices in order of subscript

Begin traversal at v_1

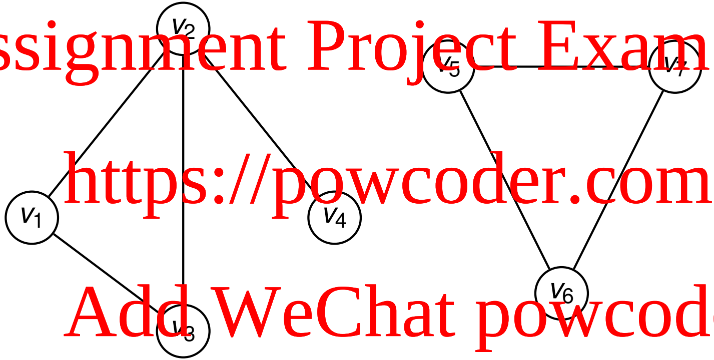


Breadth-first Search

$BFS(G, s)$:

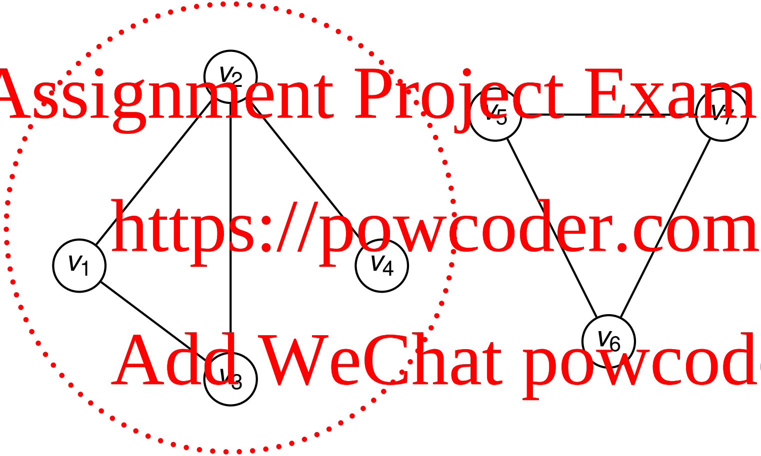
let Q *be a queue containing just the node* s
let $discovered(s) = true$
let $discovered(v) = false$ *for all* $v \in V - \{s\}$
let $T = (V, \{\})$
while Q *is not empty*
  remove v *from the front of* Q
  for each edge $\{v, w\}$ *in* E *where not* $discovered(w)$
    let $discovered(w) = true$
    add w *to the back of* Q
    add edge $\{v, w\}$ *to edges in* T

Components of Graphs



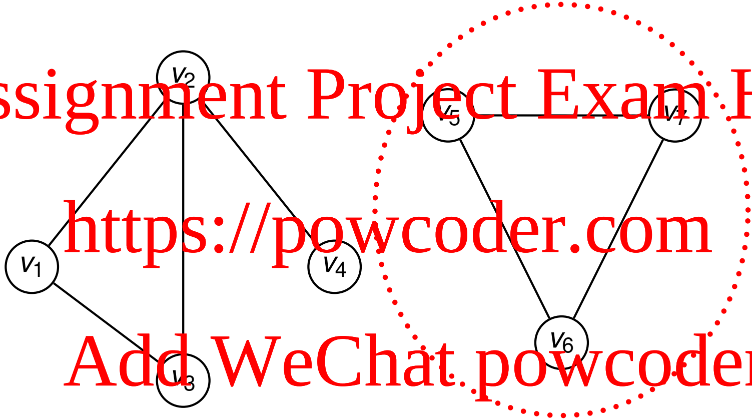
Is this one or two graphs?

Components of Graphs



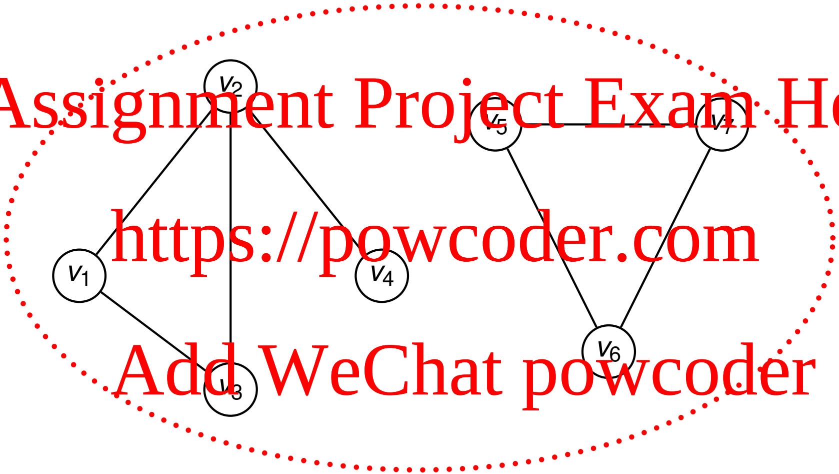
This is one of the components

Components of Graphs



And this is the other components

Components of Graphs



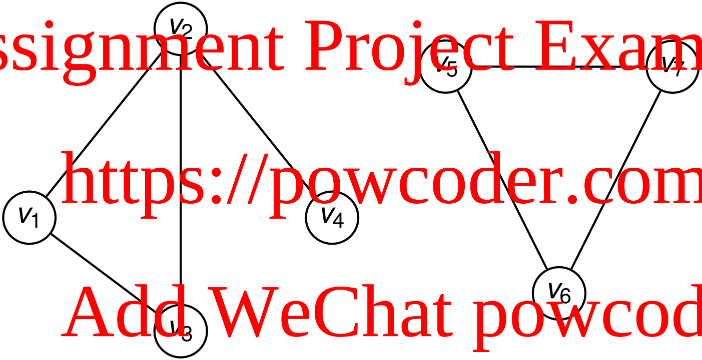
But remember, its all one graph

Components of Graphs (cont.)

Definition of a component of a graph $G = (V, E)$

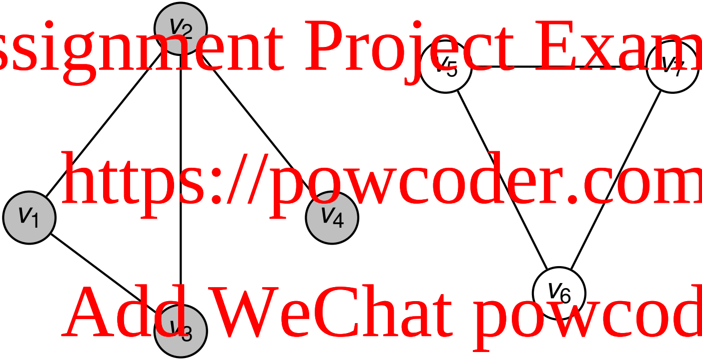
- A subset of the vertices in G and all associated edges from G
- Must be connected
  — path between any pair of vertices
- Must be maximal
  — cannot be enlarged and remain connected

BFS and Multiple Component Graphs



Question: Suppose BFS run on this graph starting at v_3?

Components of Graphs (cont.)



Answer: BFS would find only nodes in the component containing v_3

Revised BFS Algorithm

$BFS(G)$:

$let\ discovered(v) = false\ for\ all\ v \in V$

$let\ T = (V, \{\})$

$for\ each\ v \in V$

$\quad if\ not\ discovered(v)$

$\quad\quad BFS(G, v)$

Revised BFS Algorithm (cont.)

BFS(G, s) :

let Q *be a queue containing just the node* s
let discovered$(s) =$ *true*
while Q *is not empty*
    remove v *from the front of* Q
    for each edge $\{v, w\}$ *in* E *where not discovered*(w)
        let discovered$(w) =$ *true*
        add w *to the back of* Q
        add edge $\{v, w\}$ *to edges in* T

BFS Forest

- T can consist of more than one tree
- One tree in T for each component of G
- T is actually Breadth-First Search **Forest**
- Number of edges in T will be $n - k$
  — where G contains k components

Running Time of BFS

- What is a good measure of progress?
- Number of vertices whose edges have been considered
- Increases by 1 on each iteration of the *while* loop
- This means that there n iterations of the loop
- How much time spent on each iteration?
- Depends on number of edges to be considered

Running Time of BFS (cont.)

- Number of steps within loop is $\Theta(\max(1, \mathit{outdegree}(v)))$
- $\mathit{outdegree}(v)$ is number of edges from v to some other vertex w
- We know that in total there are m edges to consider
- Total across all iterations of loop:

$$\sum_{v \in V} \max(1, \mathit{outdegree}(v)) = \Theta(n + m)$$

Depth-first Search

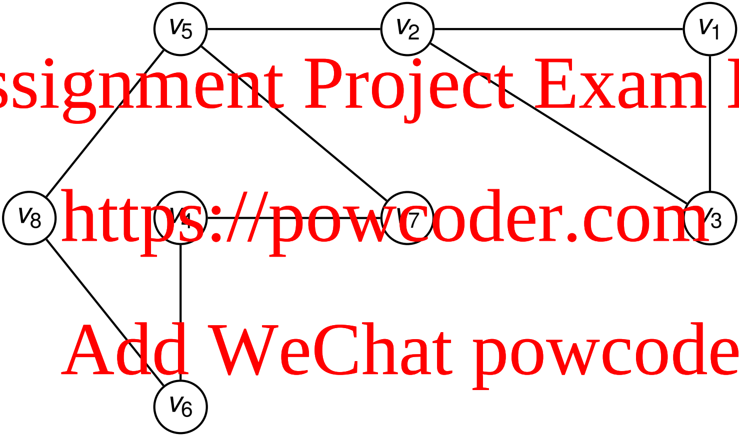
A different strategy for systematic exploration of a graph

- Uses a **stack** to hold discovered nodes
- Record nodes as *explored* once popped from stack
- Only explore nodes taken from stack that haven't already been explored
- Record edges in search tree by remembering the latest parent of each node
- Parent of a node can change up to the point it is explored

Depth-first Search

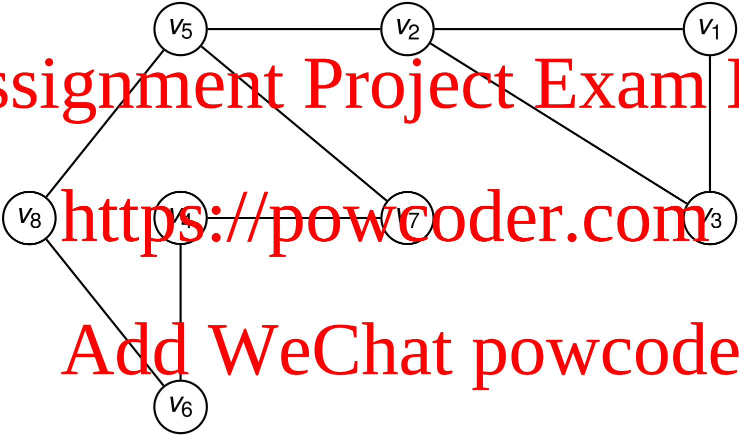
```
DFS(G, s) :
let S be a stack containing just the node s
let explored(v) = false for all v ∈ V
while S is not empty
    pop v from the top of S
    if not explored(v) then
        for each edge {v, w} in E where not explored(w)
            push w onto S
            let parent(w) = v
        let explored(v) = true
```

Illustration of DFS



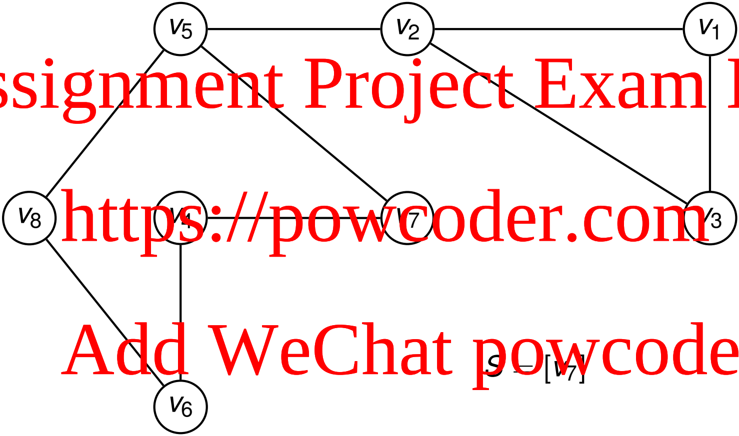
Run DFS starting at v_7

Illustration of DFS



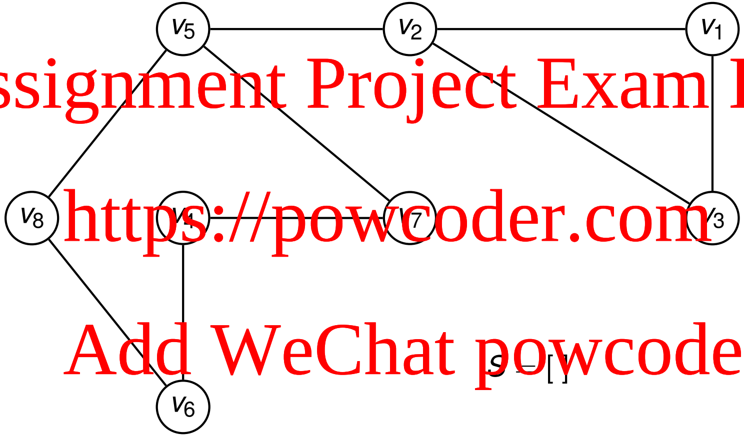
Initialize S

Illustration of DFS



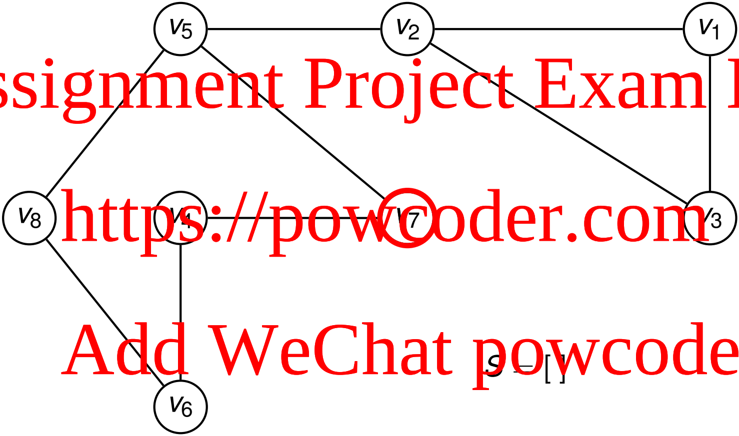
Pop v_7 from top of S

Illustration of DFS



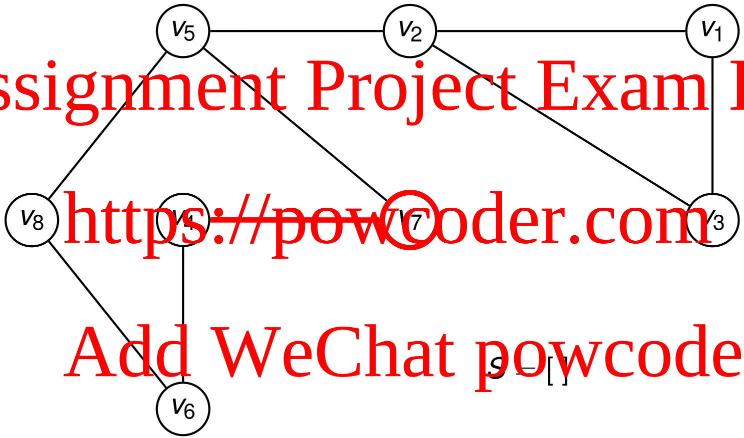
v_7 not yet explored so let's explore it

Illustration of DFS



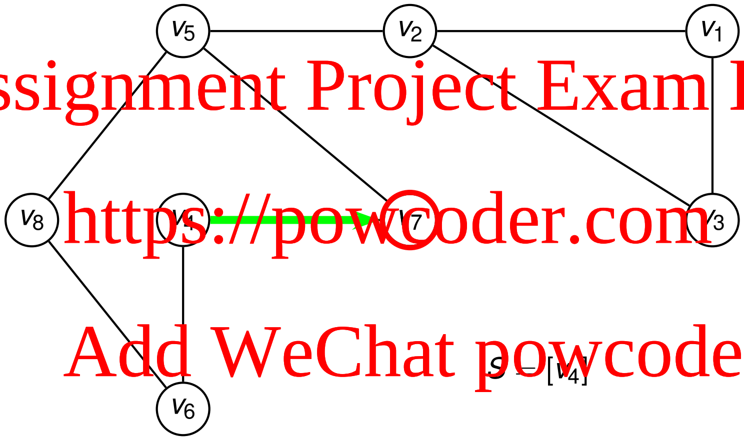
Consider edge $\{v_7, v_4\}$ since v_4 not yet explored

Illustration of DFS



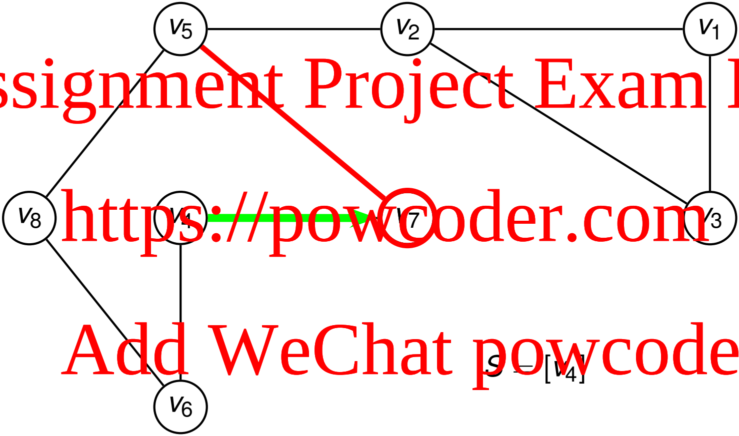
Push v_4 onto S and let $parent(v_4) = v_7$

Illustration of DFS



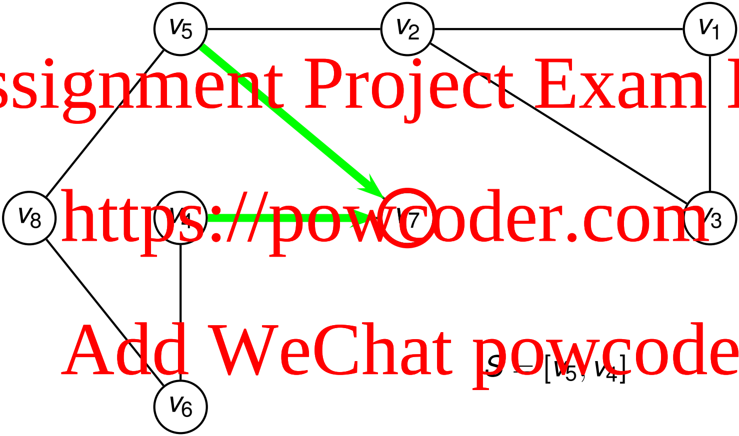
Consider edge $\{v_7, v_5\}$ since v_5 not yet explored

Illustration of DFS



Push v_5 onto S and let $parent(v_5) = v_7$

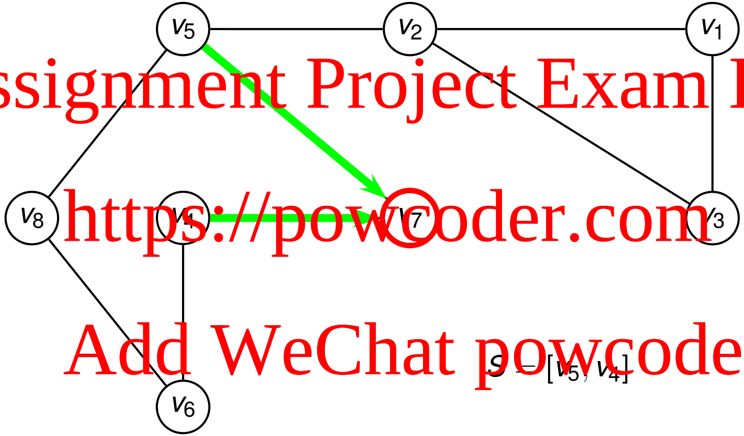
Illustration of DFS



$S = [v_5, v_4]$

No more edges from v_7

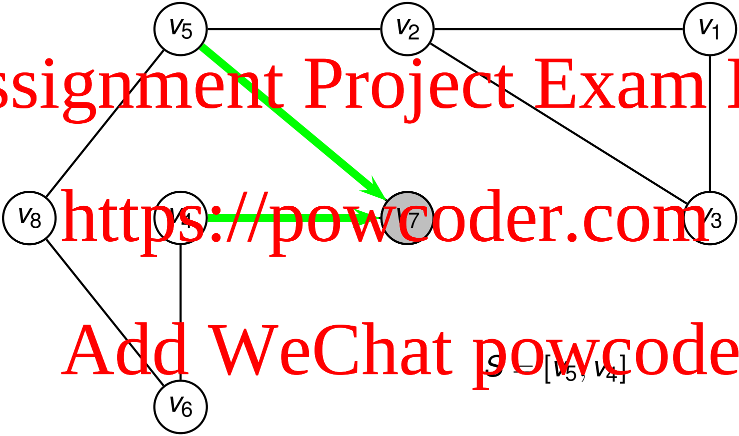
Illustration of DFS



$S = [v_5, v_4]$

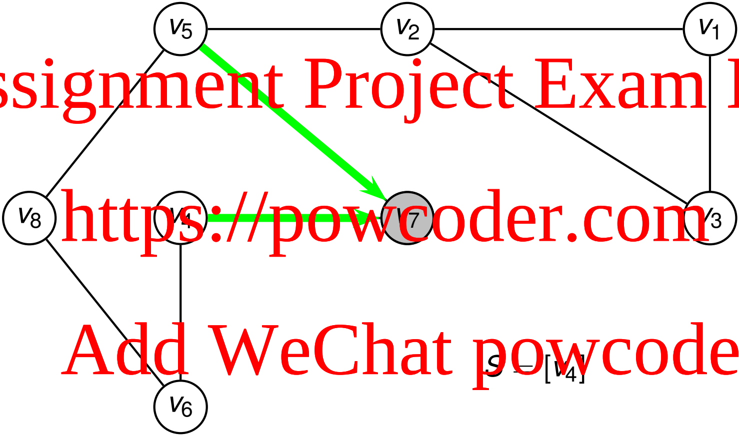
Let $\mathit{explored}(v_7) = \mathit{true}$

Illustration of DFS



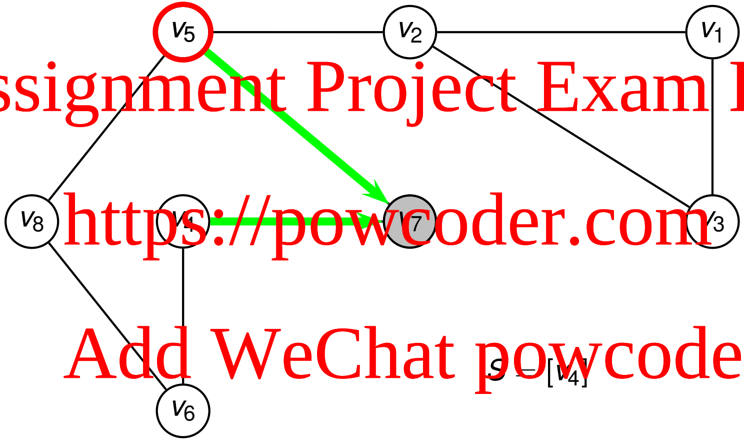
Pop v_5 from top of S

Illustration of DFS



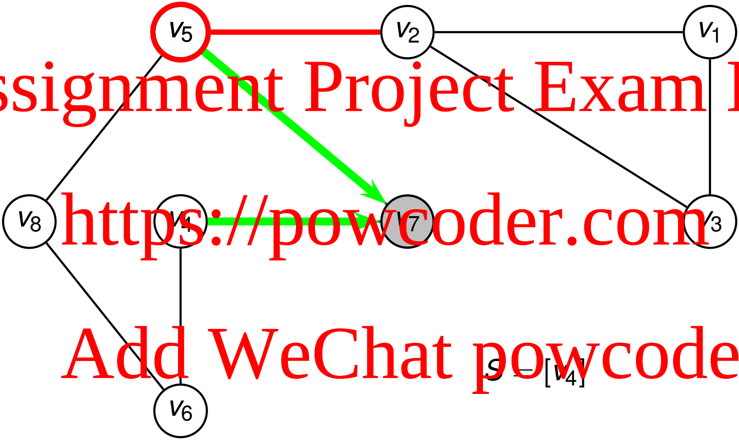
v_5 not yet explored so let's explore it

Illustration of DFS



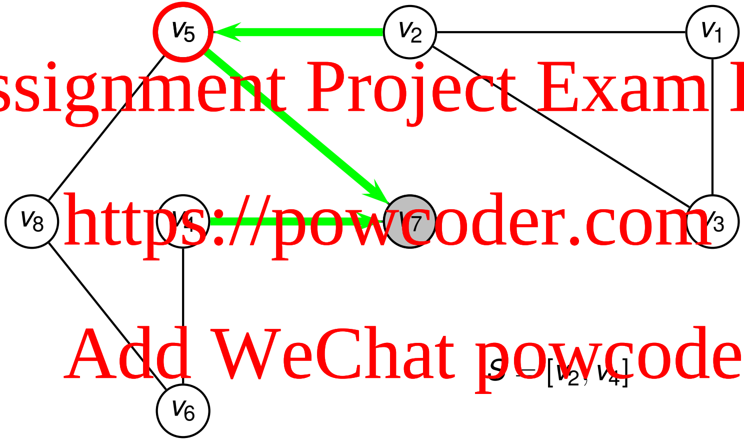
Consider edge $\{v_5, v_2\}$ since v_2 not yet explored

Illustration of DFS



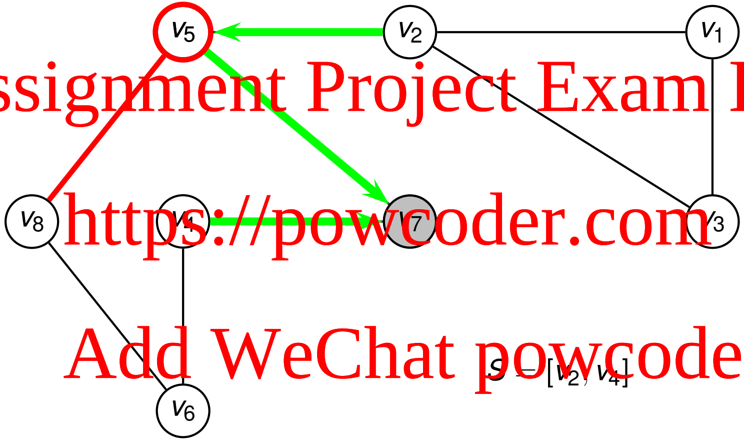
Push v_2 onto S and let $parent(v_2) = v_5$

Illustration of DFS



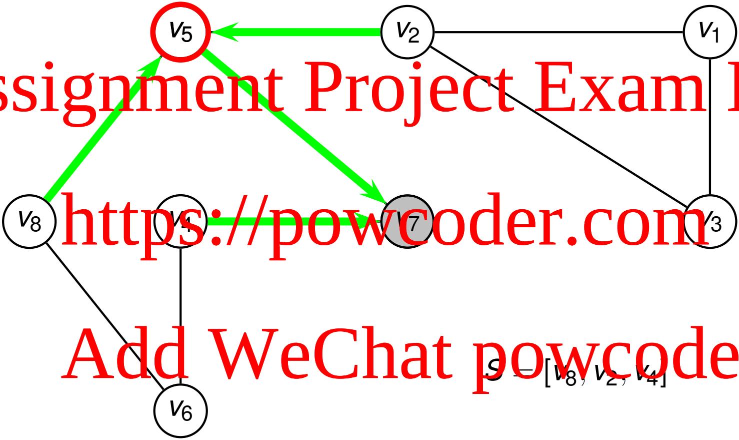
Consider $\{v_5, v_8\}$ since v_8 not yet explored

Illustration of DFS



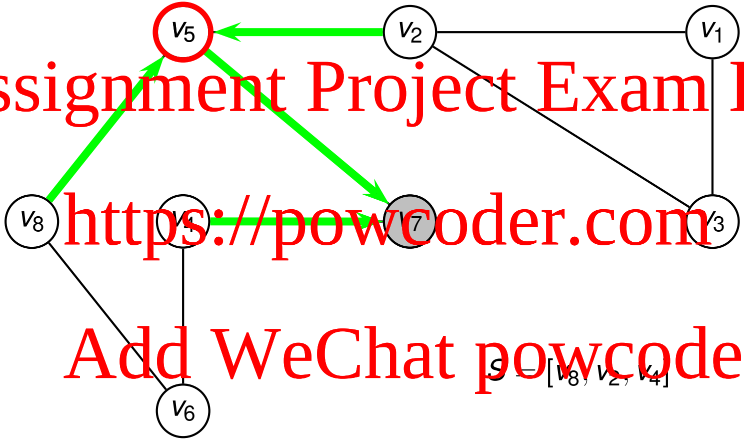
Push v_8 onto S and let $parent(v_8) = v_5$

Illustration of DFS



Don't consider $\{v_5, v_7\}$ because v_7 already explored

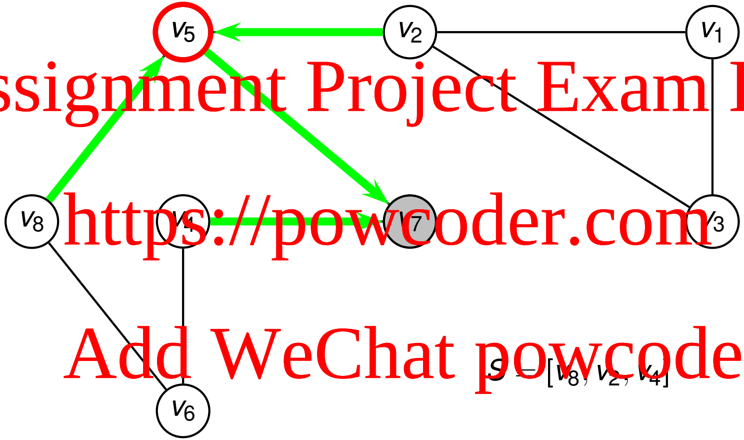
Illustration of DFS



$S = [v_8, v_2, v_4]$

No more edges from v_5

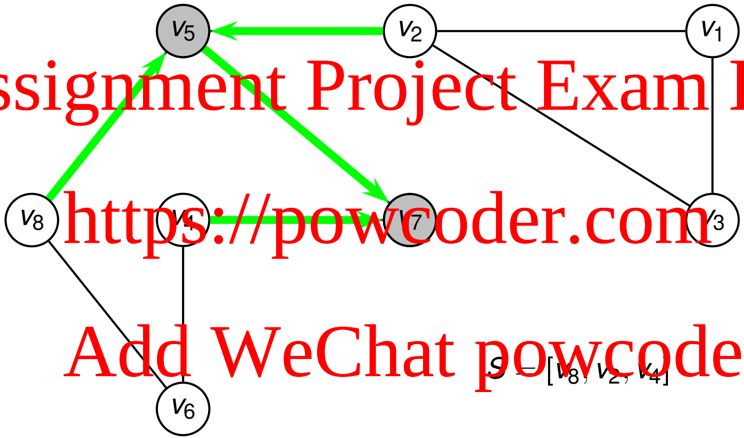
Illustration of DFS



$S = [v_8, v_2, v_4]$

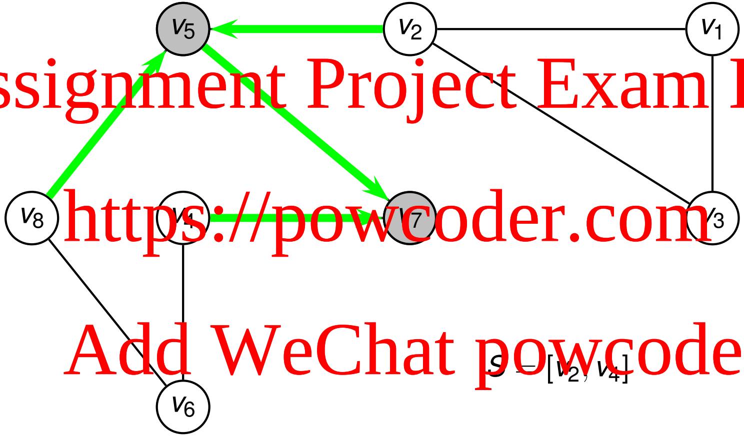
Let $explored(v_5) = true$

Illustration of DFS



Pop v_8 from top of S

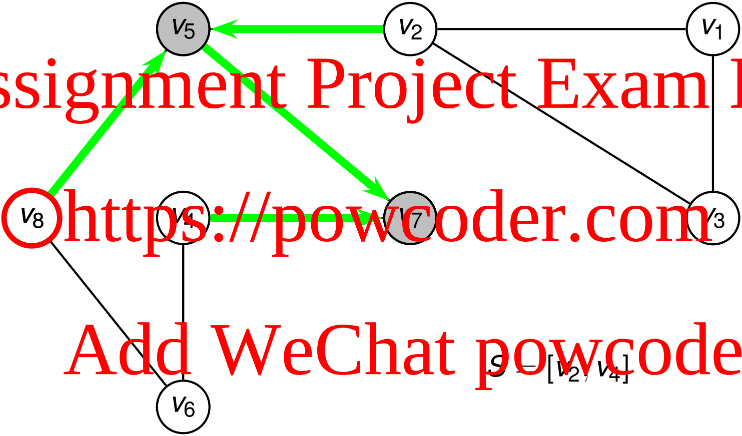
Illustration of DFS



$S = [v_2, v_4]$

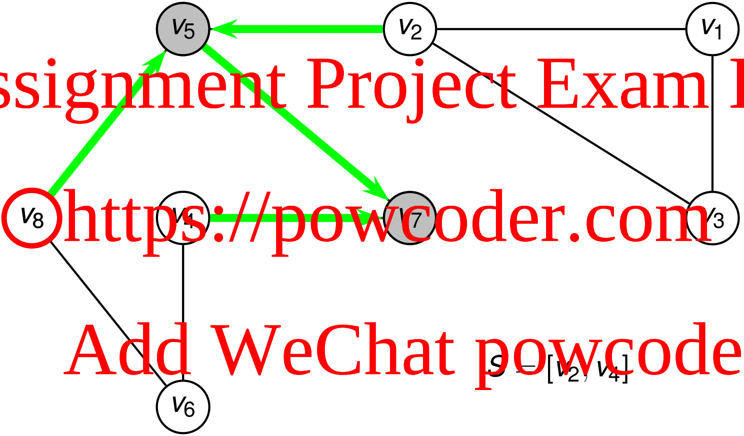
v_8 not yet explored so let's explore it

Illustration of DFS



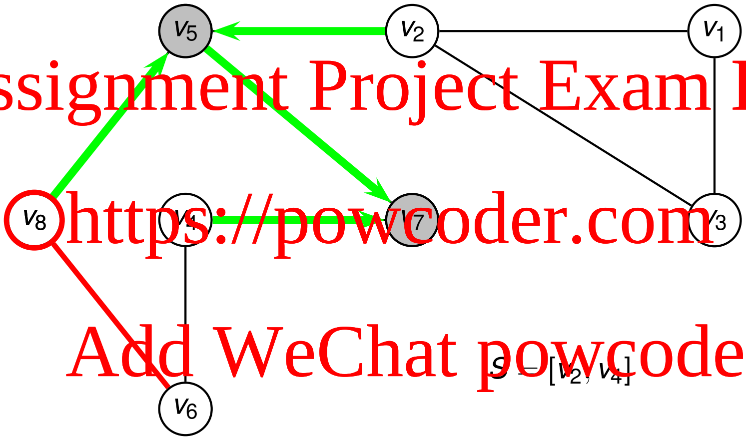
Don't consider $\{v_8, v_5\}$ because v_5 already explored

Illustration of DFS



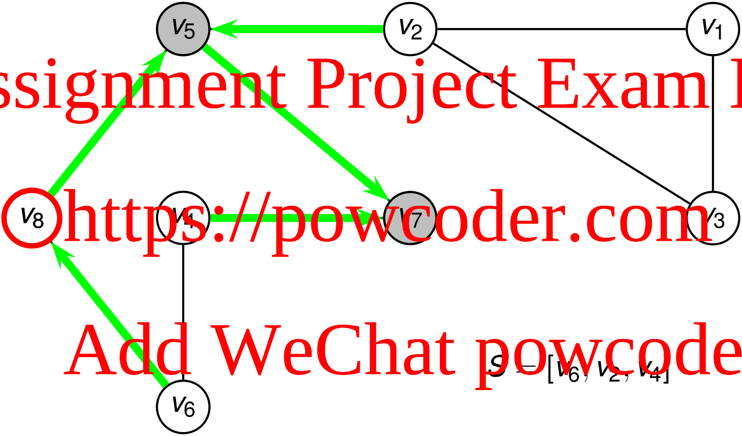
Consider edge $\{v_8, v_6\}$ since v_6 not yet explored

Illustration of DFS



Push v_6 onto S and let $parent(v_6) = v_8$

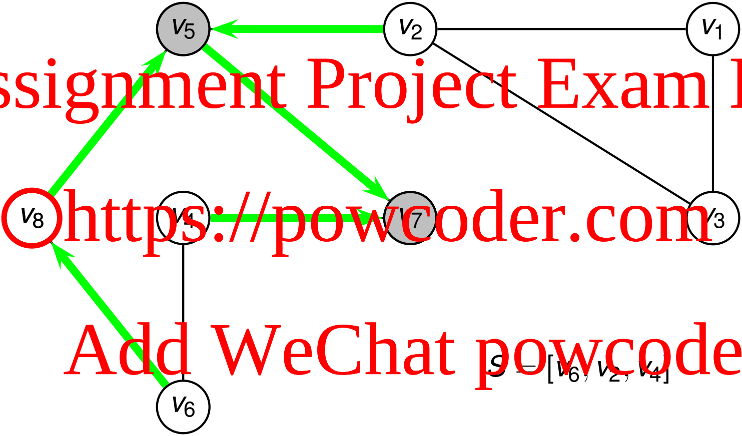
Illustration of DFS



$S = [v_6, v_2, v_4]$

No more edges from v_8

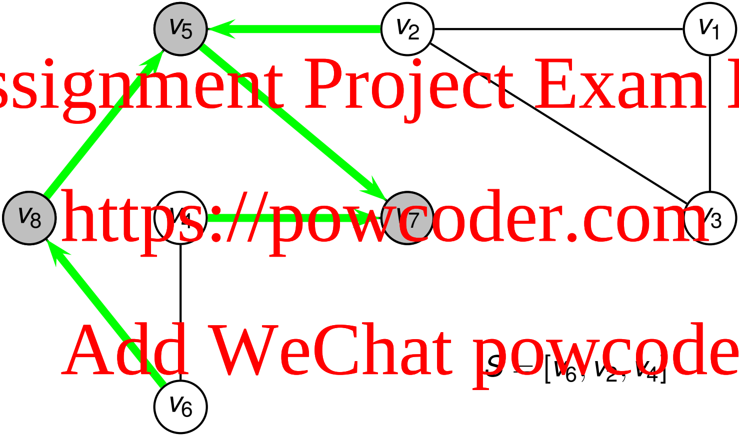
Illustration of DFS



$S = [v_6, v_2, v_4]$

Let $\mathit{explored}(v_8) = \mathit{true}$

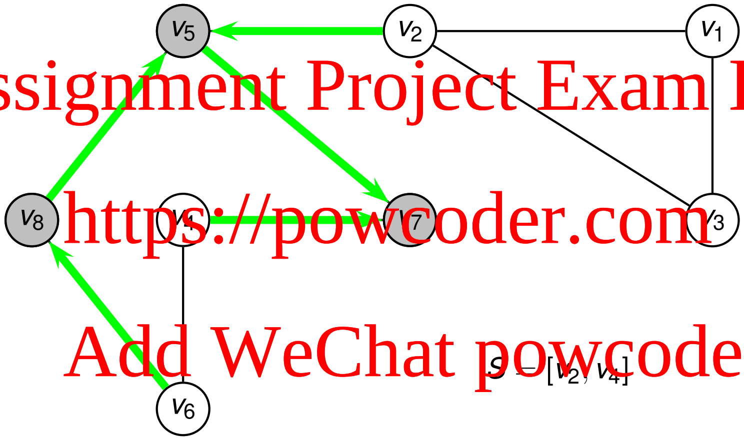
Illustration of DFS



$S = [v_6, v_2, v_4]$

Pop v_6 from top of S

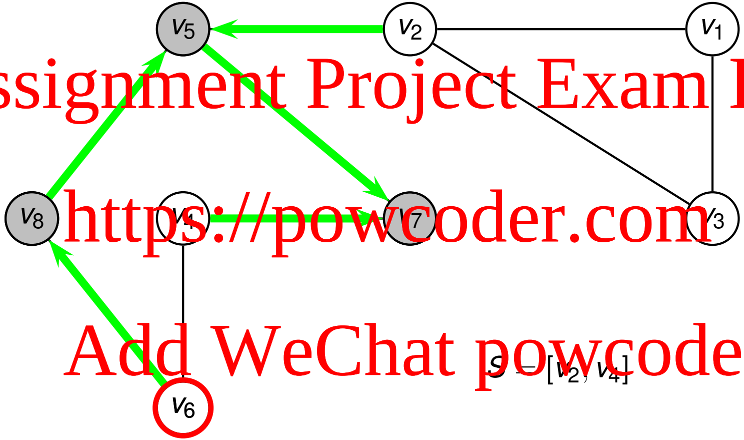
Illustration of DFS



$S = [v_2, v_4]$

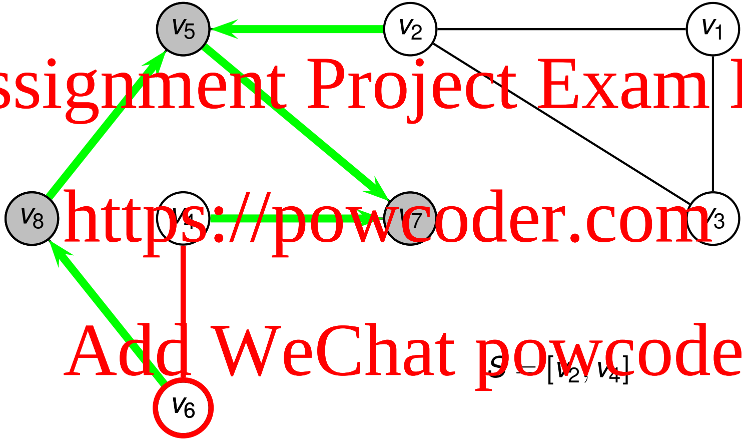
v_6 not yet explored so let's explore it

Illustration of DFS



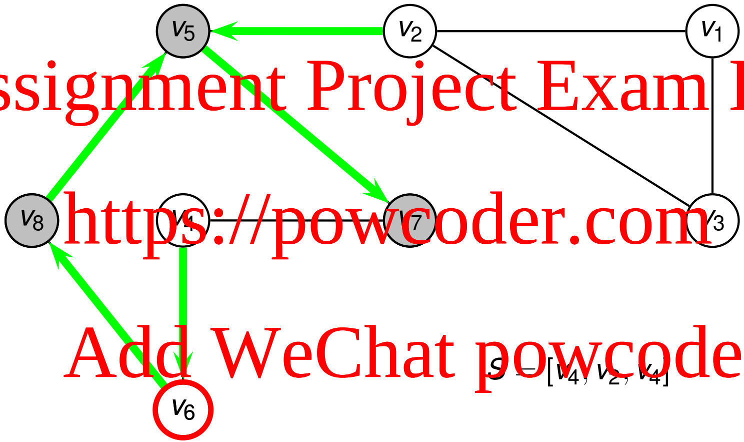
Consider edge $\{v_6, v_4\}$ since v_4 not yet explored

Illustration of DFS



Push v_4 onto S and let $parent(v_4) = v_6$

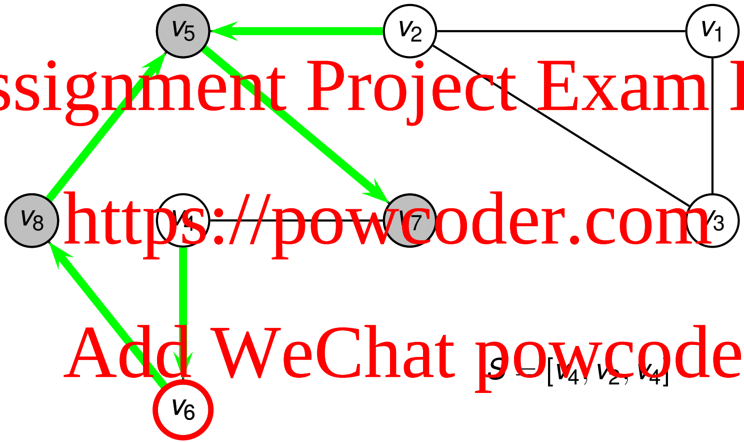
Illustration of DFS



$S = [v_4, v_2, v_4]$

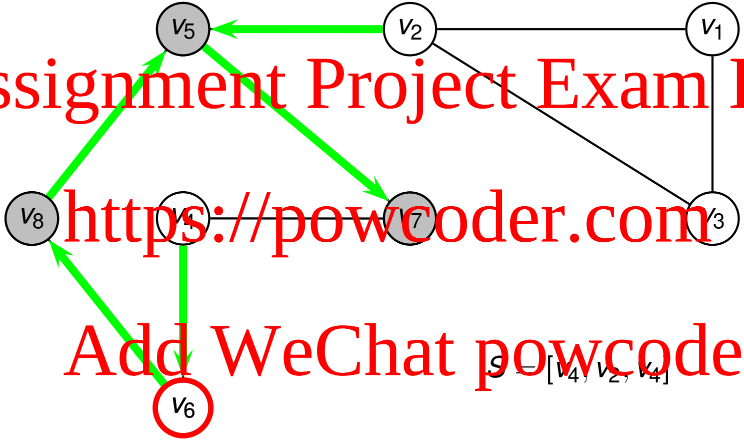
Don't consider $\{v_6, v_8\}$ because v_8 already explored

Illustration of DFS



No more edges from v_6

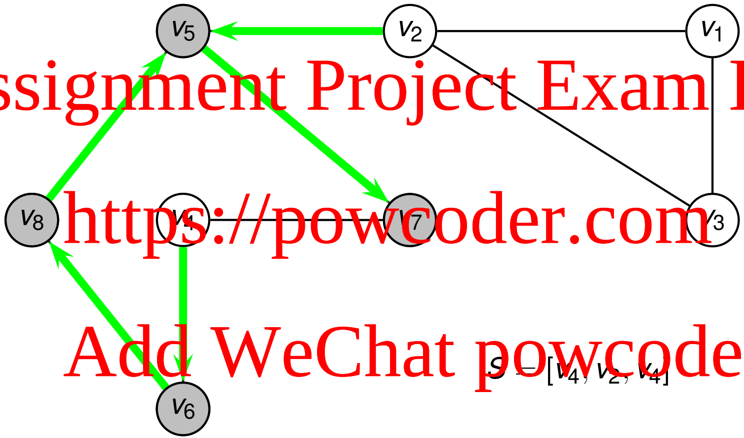
Illustration of DFS



$S = [v_4, v_2, v_4]$

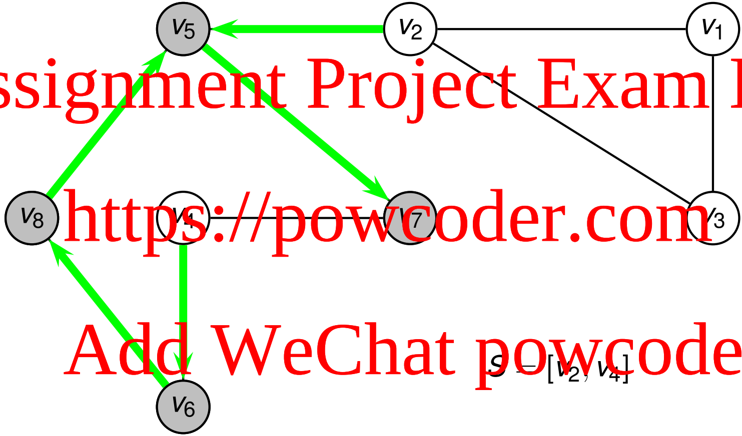
Let $explored(v_6) = true$

Illustration of DFS



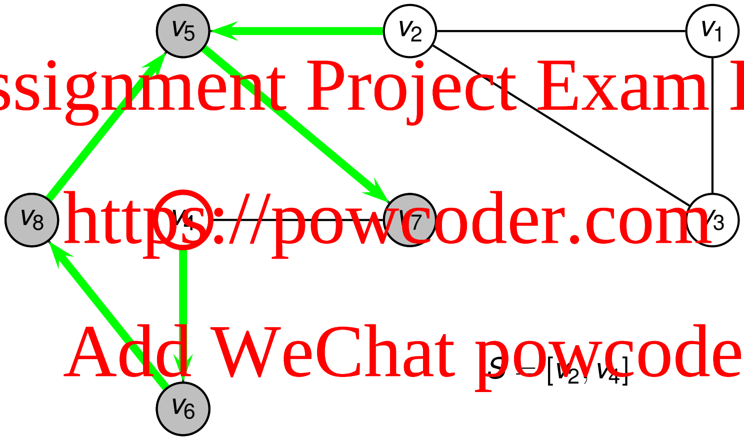
Pop v_4 from top of S

Illustration of DFS



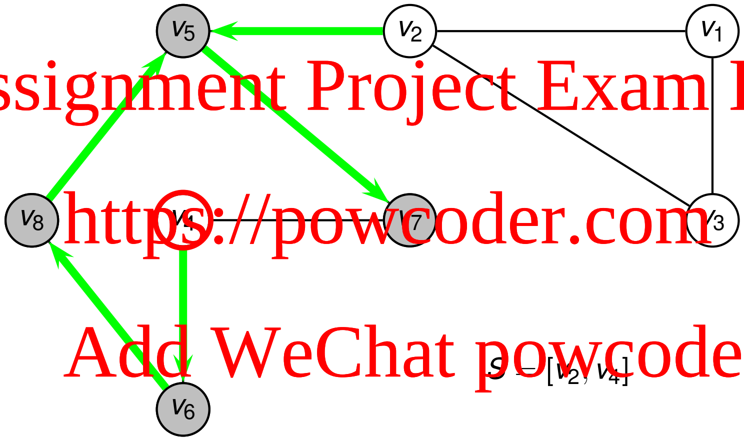
v_4 not yet explored so let's explore it

Illustration of DFS



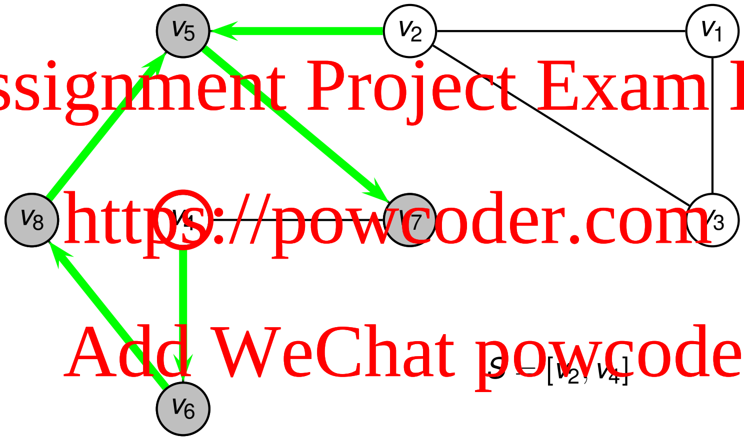
Don't consider $\{v_4, v_6\}$ because v_6 already explored

Illustration of DFS



Don't consider $\{v_4, v_7\}$ because v_7 already explored

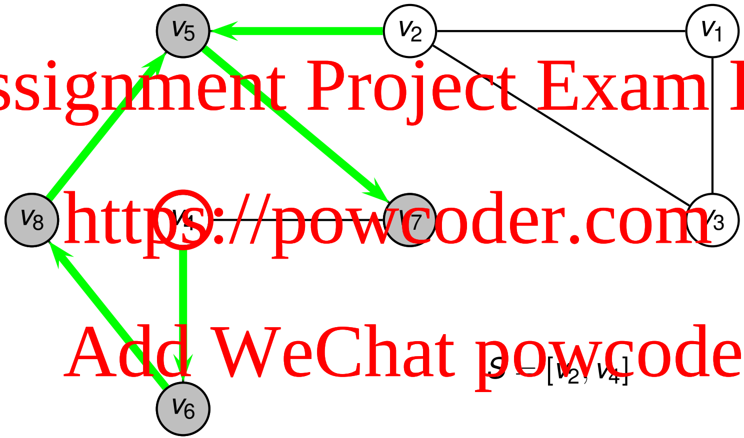
Illustration of DFS



$S = [v_2, v_4]$

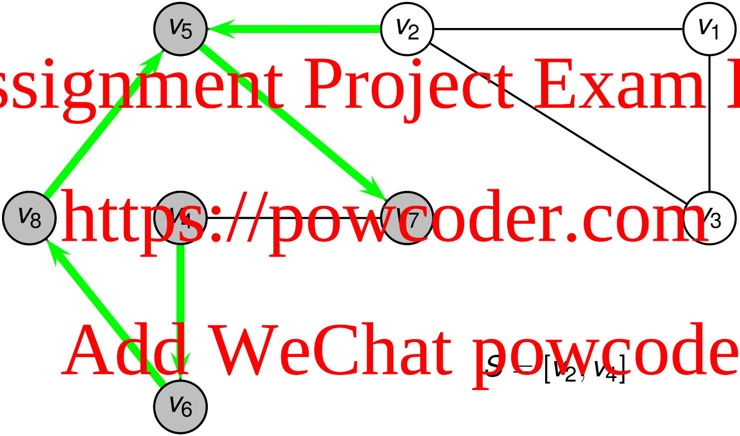
No more edges from v_4

Illustration of DFS



Let $explored(v_4) = true$

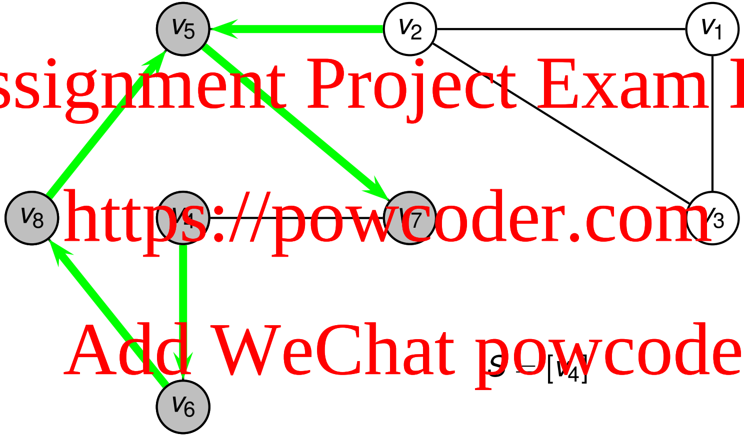
Illustration of DFS



$S = [v_2, v_4]$

Pop v_2 from top of S

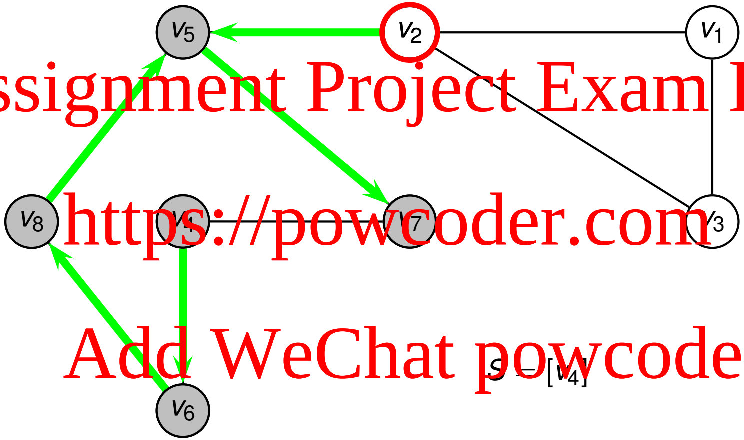
Illustration of DFS



$S = [v_4]$

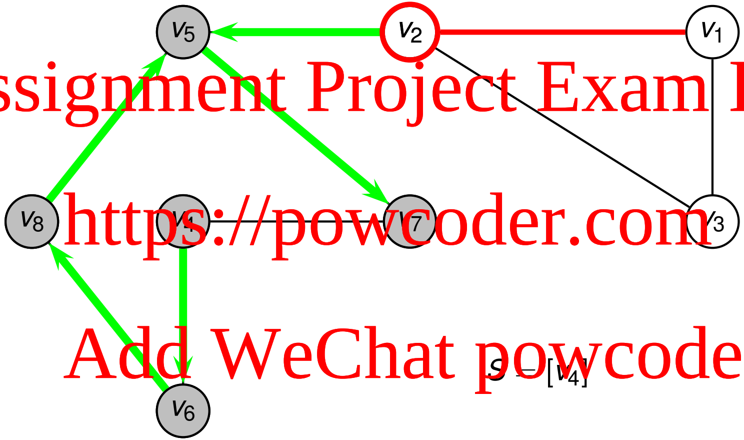
v_2 not yet explored so let's explore it

Illustration of DFS



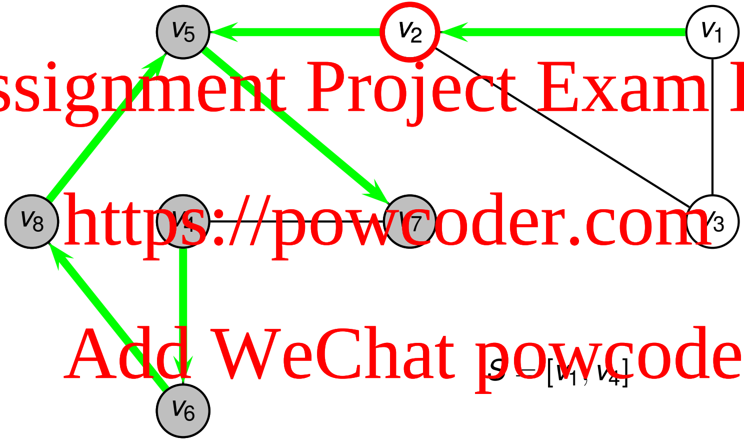
Consider edge $\{v_2, v_1\}$ since v_1 not yet explored

Illustration of DFS



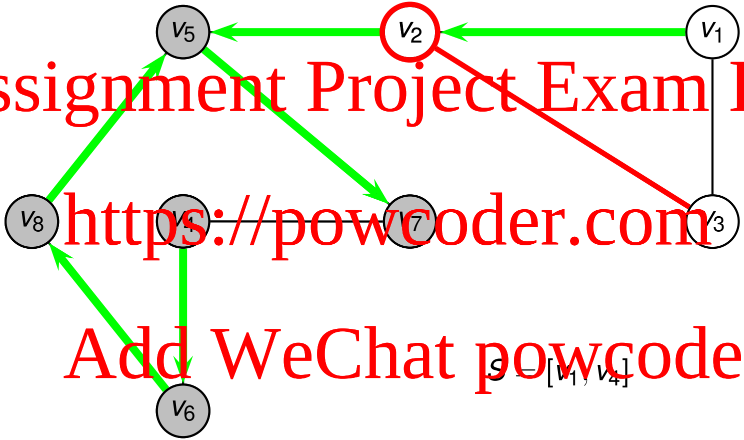
Push v_1 onto S and let $parent(v_1) = v_2$

Illustration of DFS



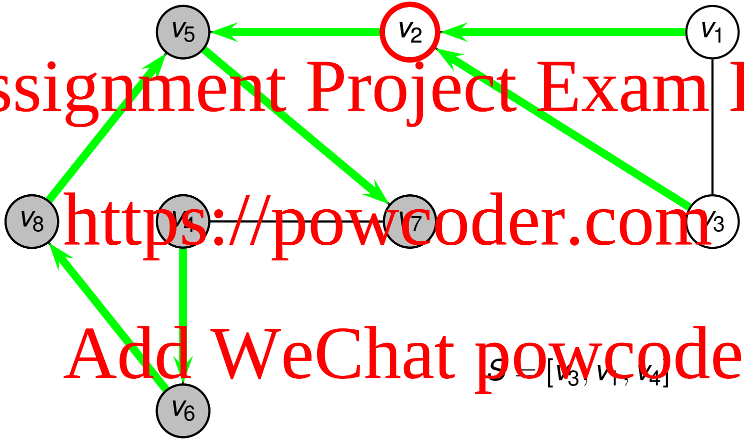
Consider edge $\{v_2, v_3\}$ since v_3 not yet explored

Illustration of DFS



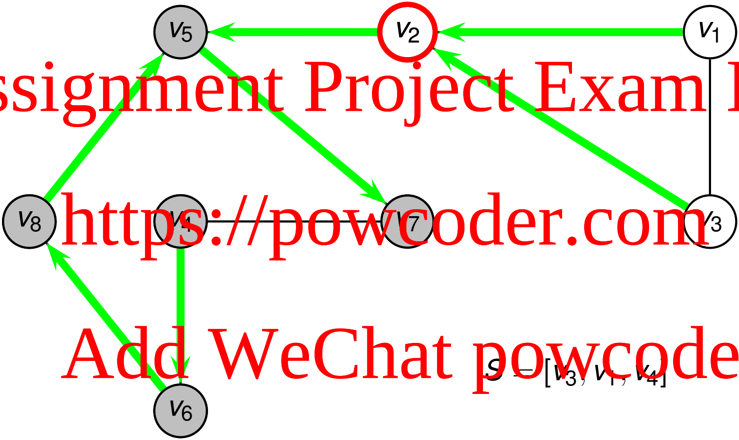
Push v_3 onto S and let $parent(v_3) = v_2$

Illustration of DFS



Don't consider $\{v_2, v_5\}$ because v_5 already explored

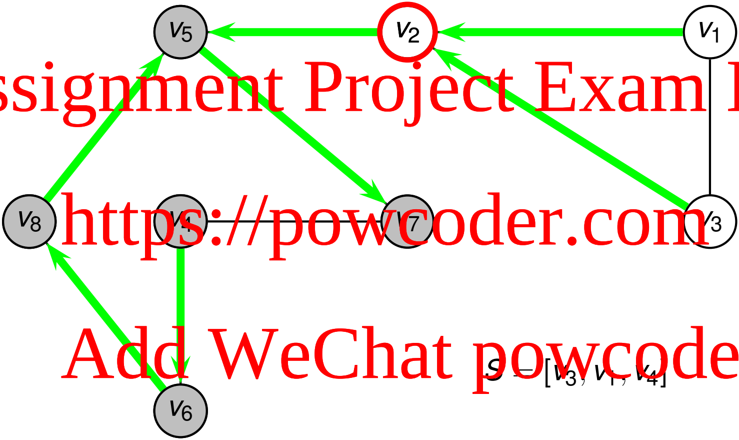
Illustration of DFS



$S = [v_3, v_1, v_4]$

No more edges from v_2

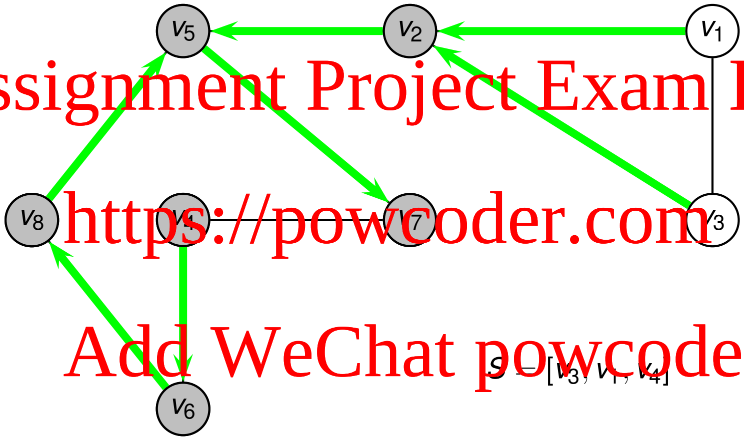
Illustration of DFS



$S = [v_3, v_1, v_4]$

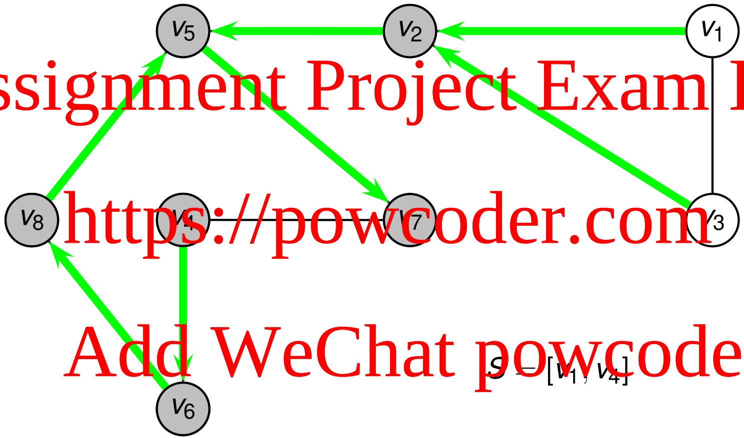
Let $explored(v_2) = true$

Illustration of DFS



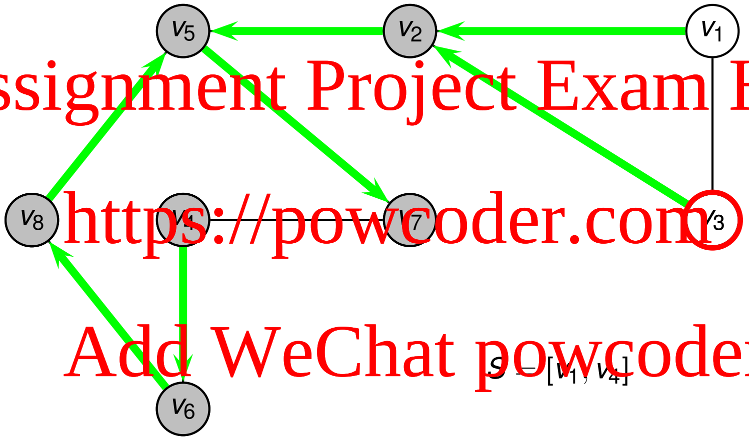
Pop v_3 from top of S

Illustration of DFS



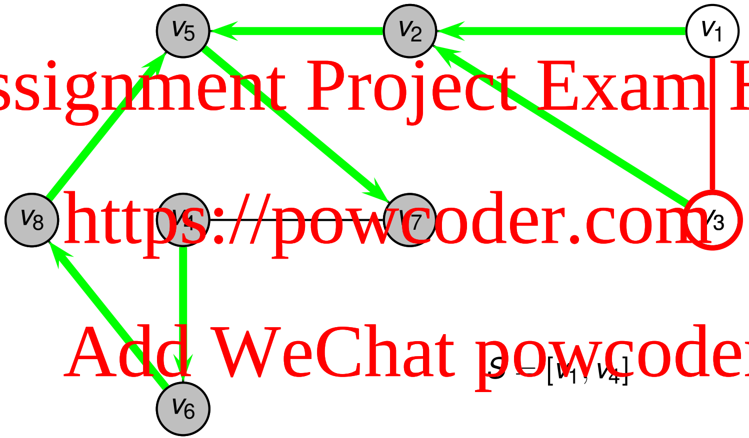
v_3 not yet explored so let's explore it

Illustration of DFS



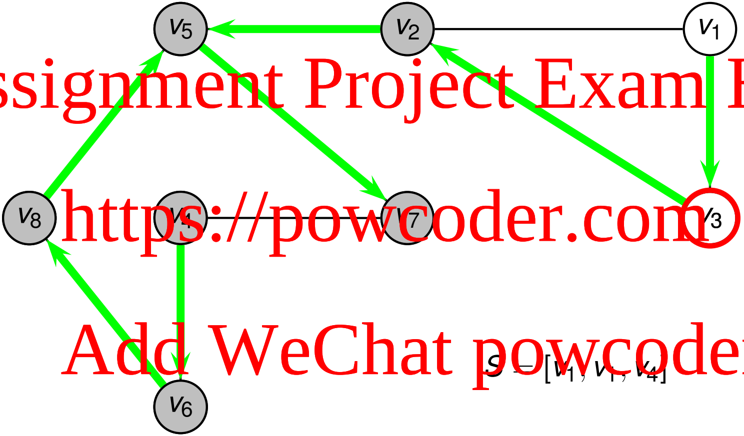
Consider edge $\{v_3, v_1\}$ since v_1 not yet explored

Illustration of DFS



Push v_1 onto S and let $parent(v_1) = v_3$

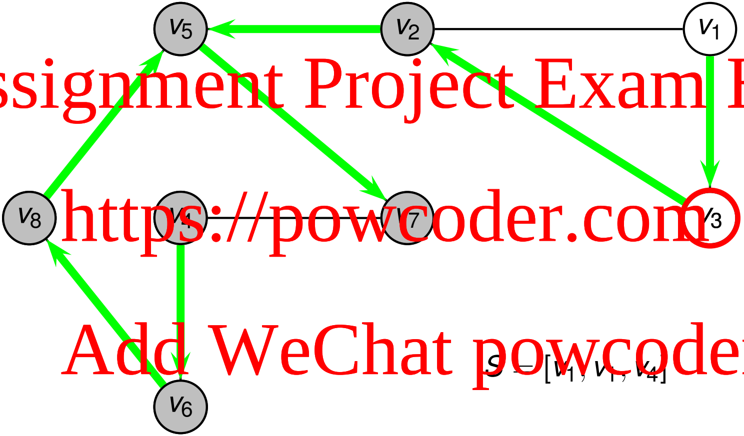
Illustration of DFS



$S = [v_1, v_1, v_4]$

Don't consider $\{v_3, v_2\}$ because v_2 already explored

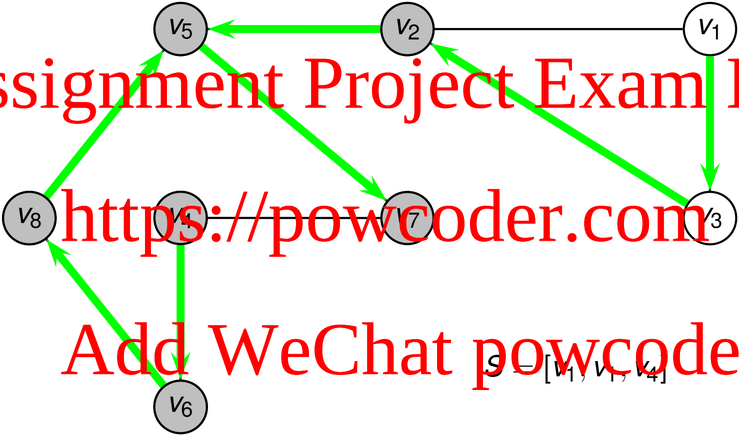
Illustration of DFS



$S = [v_1, v_1, v_4]$

No more edges from v_3

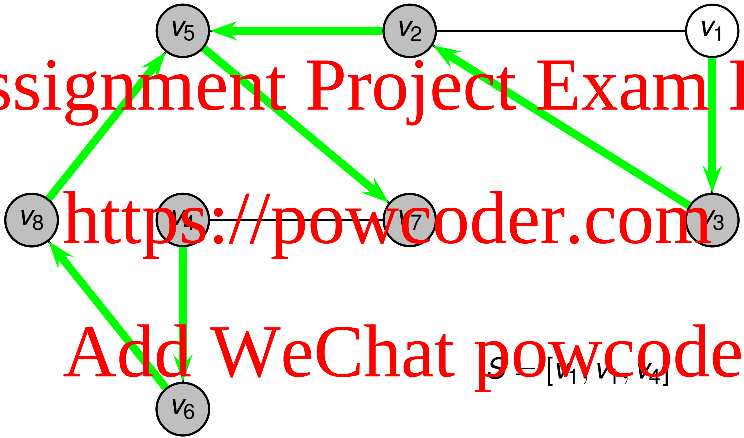
Illustration of DFS



$S = [v_1, v_1, v_4]$

Let $\textit{explored}(v_3) = \textit{true}$

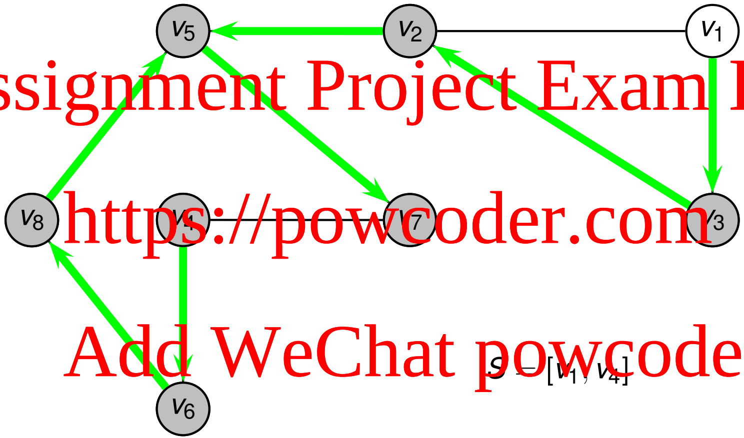
Illustration of DFS



$S = [v_1, v_1, v_4]$

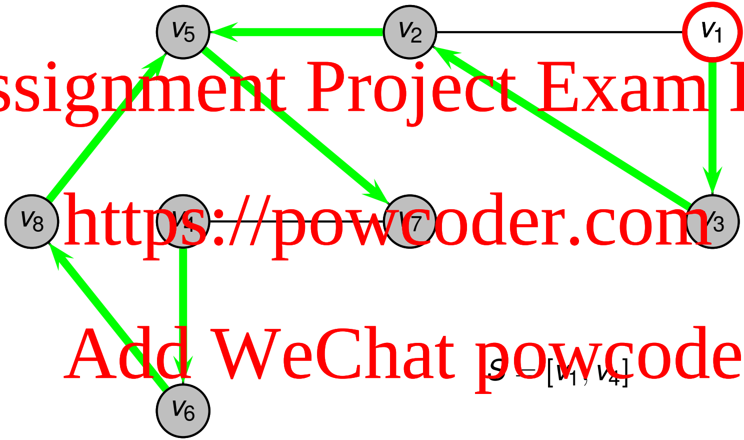
Pop v_1 from top of S

Illustration of DFS



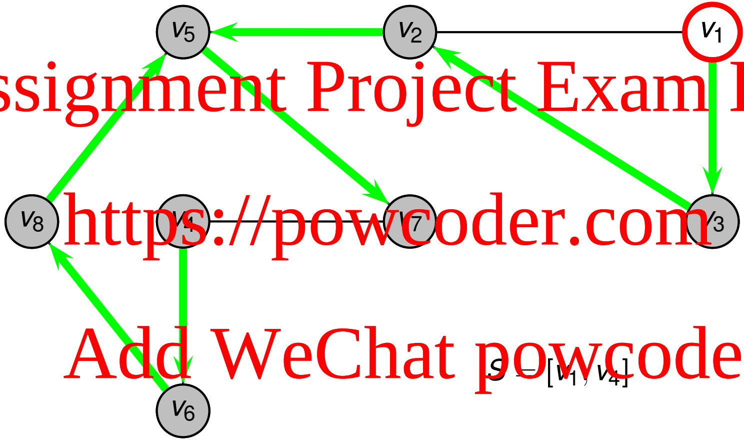
v_1 not yet explored so let's explore it

Illustration of DFS



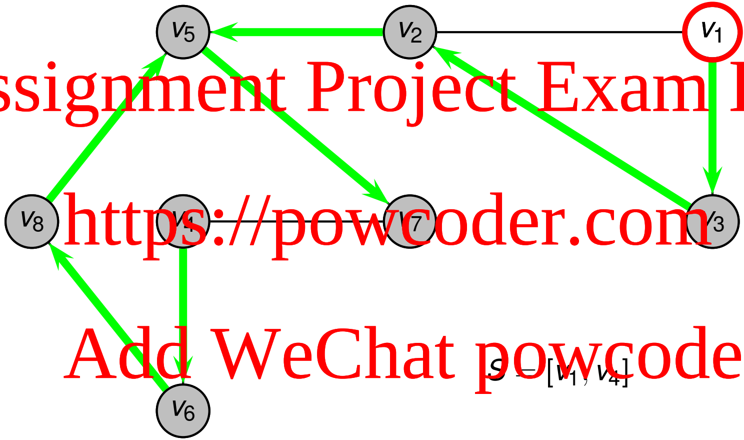
Don't consider $\{v_1, v_2\}$ because v_2 already explored

Illustration of DFS



Don't consider $\{v_1, v_3\}$ because v_3 already explored

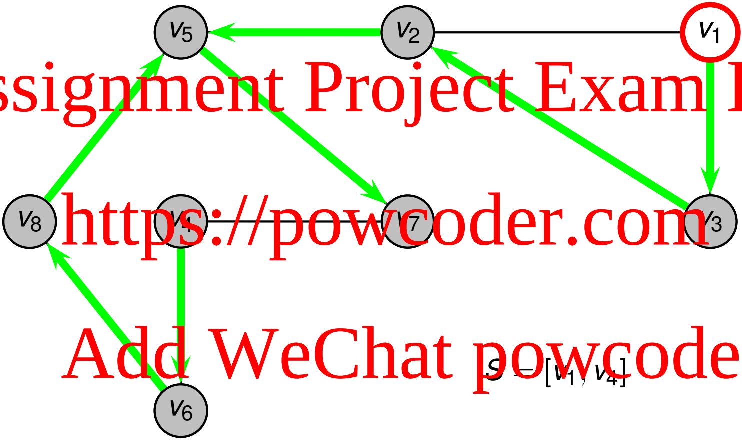
Illustration of DFS



$S = [v_1, v_4]$

No more edges from v_1

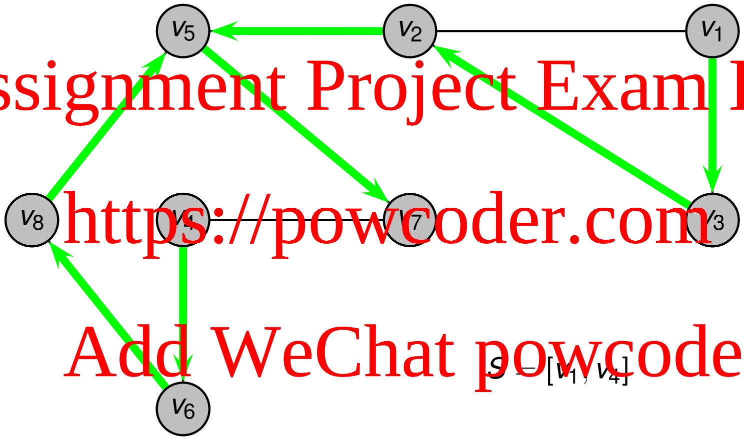
Illustration of DFS



$S = [v_1, v_4]$

Let $explored(v_1) = true$

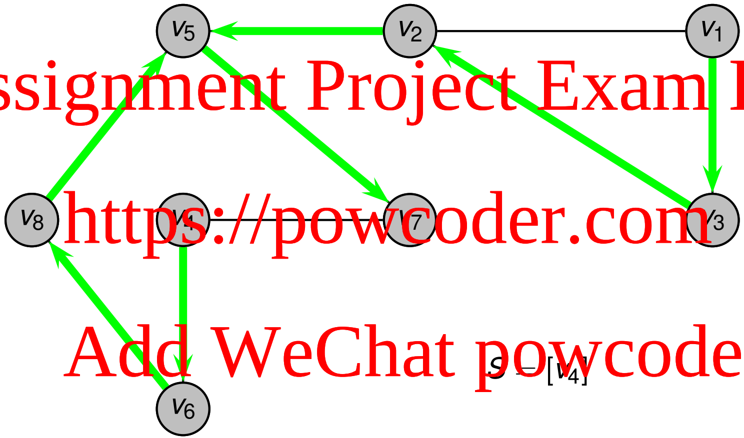
Illustration of DFS



$S = [v_1, v_4]$

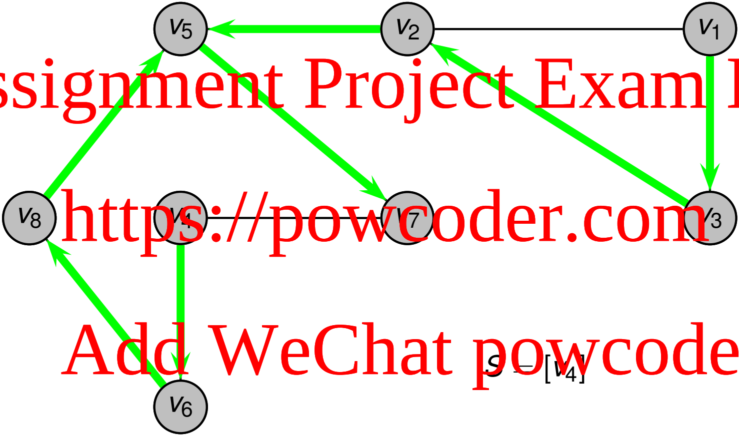
Pop v_1 from top of S

Illustration of DFS



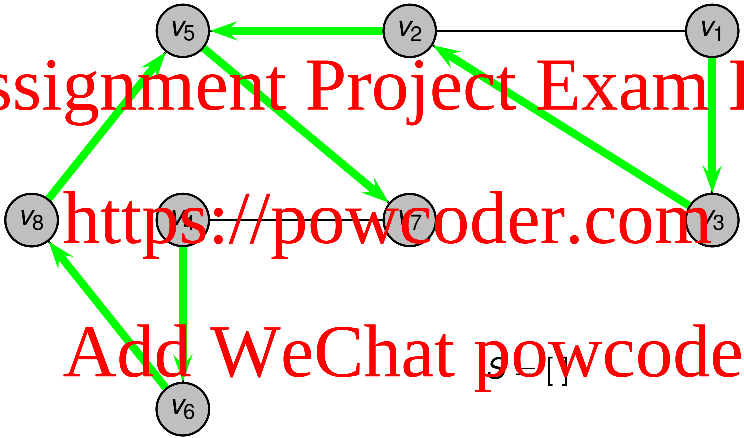
v_1 already explored so no need to consider it

Illustration of DFS



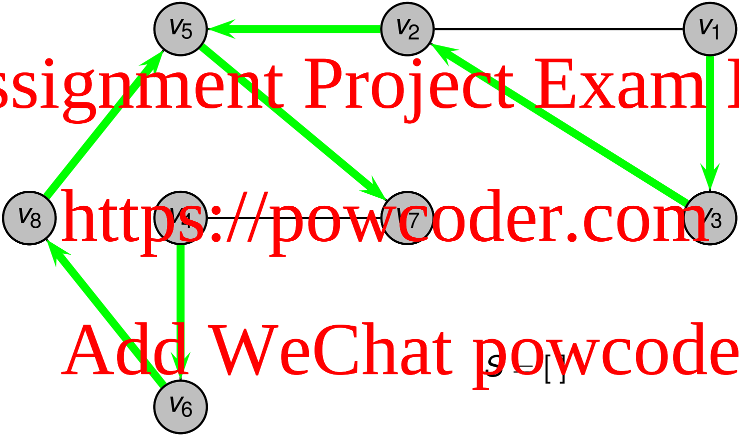
Pop v_4 from top of S

Illustration of DFS



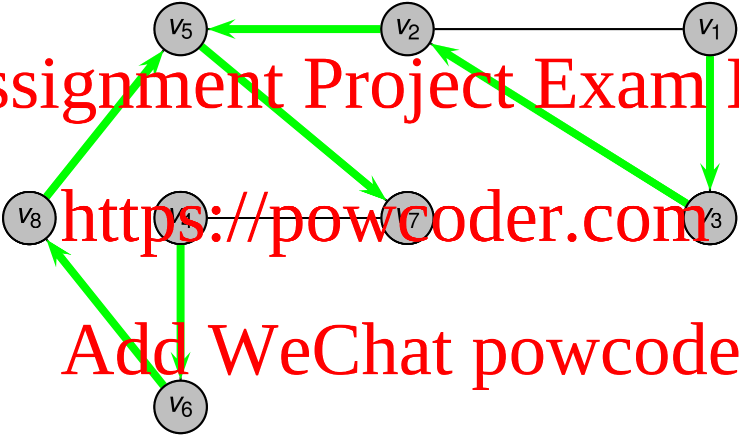
v_4 already explored so no need to consider it

Illustration of DFS



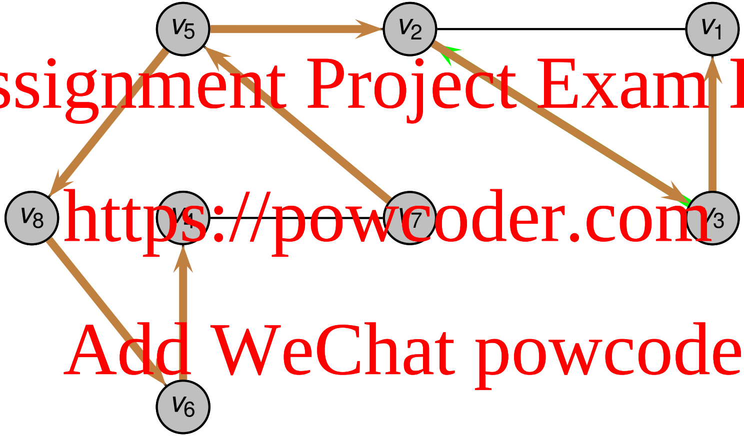
S is empty so search is complete

Illustration of DFS



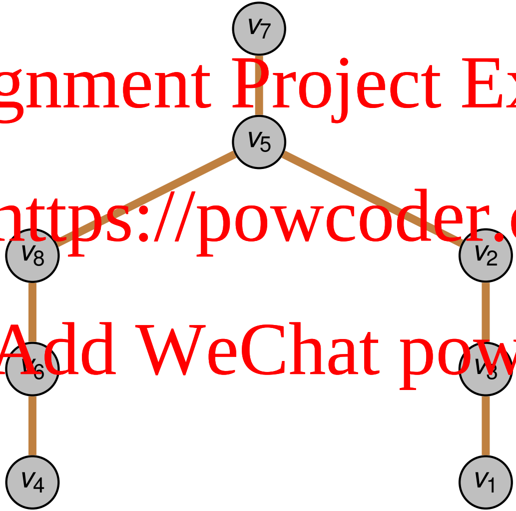
Obtain tree by reversing directionality of all parent edges

Illustration of DFS

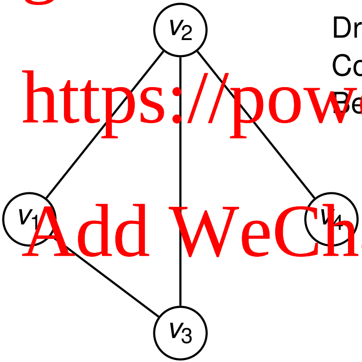


Re-position nodes for clarity

Illustration of DFS



Example for You

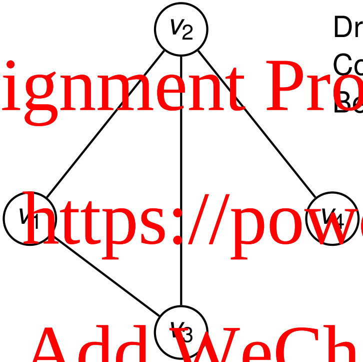


Draw the depth-first search tree

Consider vertices in order of subscript

Begin traversal at v_1

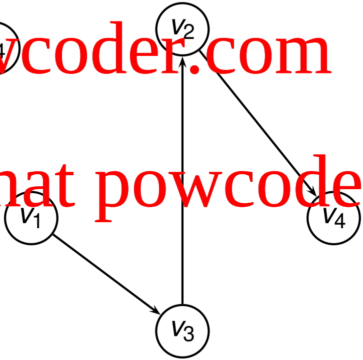
Example for You



Draw the depth-first search tree

Consider vertices in order of subscript

Begin traversal at v_1



DFS Algorithm

```
DFS(G, s) :
let S be a stack containing just the node s
let explored(v) = false for all v ∈ V
while S is not empty
    pop v from the top of S
    if not explored(v) then
        for each edge {v, w} in E where not explored(w)
            push w onto S
            let parent(w) = v
        let explored(v) = true
```

Running Time of DFS

- What is a good measure of progress?
- Number of vertices whose edges have been considered not adequate
- For some iterations this will not increase
  — when explored vertex removed from stack
- Number of edge **destinations** considered always increases
- Each directed edge is directed into the destination vertex
- Increases by at least 1 on each iteration of the *while* loop

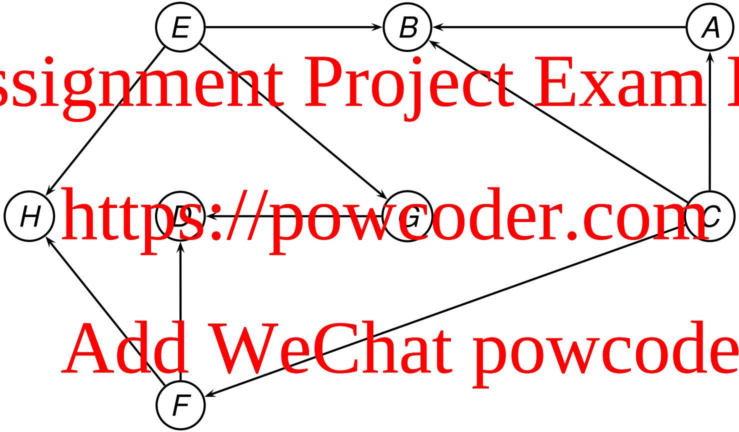
Running Time of DFS (cont.)

- This means that there $\Theta(m)$ iterations of the loop
- How much time spent on each iteration?
- Depends on number of edges to be considered
- Number of steps within loop is $\Theta(\max(1, \mathit{outdegree}(v)))$
- We know that in total there are m edges to consider
- Total across all iterations of loop:

$$\sum_{v \in V} \max(1, \mathit{outdegree}(v)) = \Theta(n + m)$$

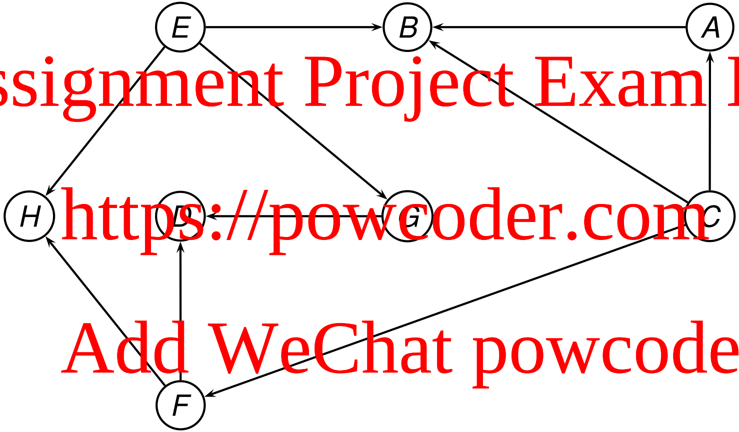
Topological Sorting

Working with Dependencies



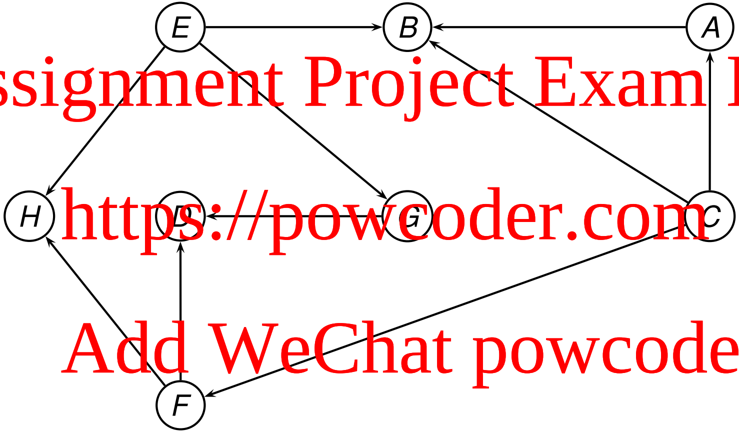
Directed Graphs often used to encode **dependencies**

Working with Dependencies



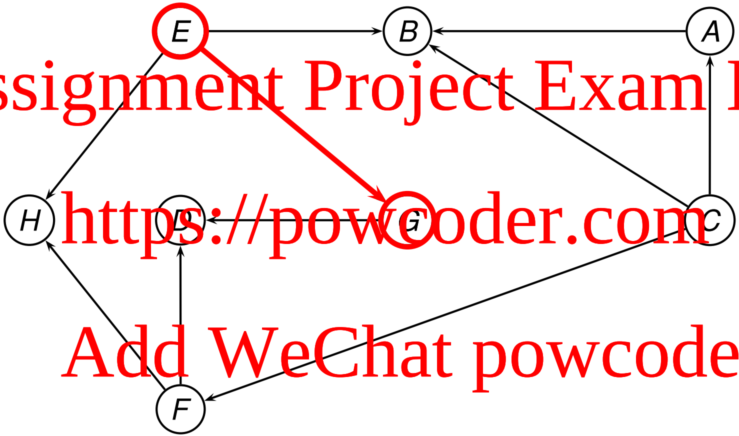
Nodes correspond to tasks

Working with Dependencies



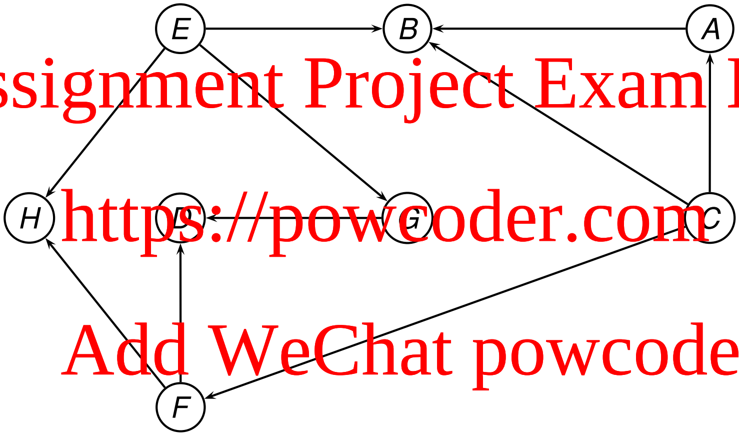
Edges encode constraints on order in which tasks can be scheduled

Working with Dependencies



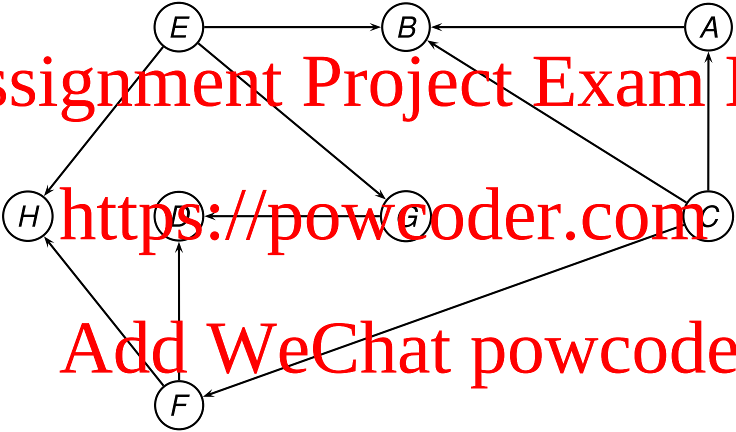
Task E must be completed before task G is started

Working with Dependencies



Cycles are undesirable

Working with Dependencies



No scheduling would be possible that satisfies constraints

Topological Sorting of DAGs

Topological Sorting Problem:

Given a DAG, $G = (V, E)$, find a topological sort of G.

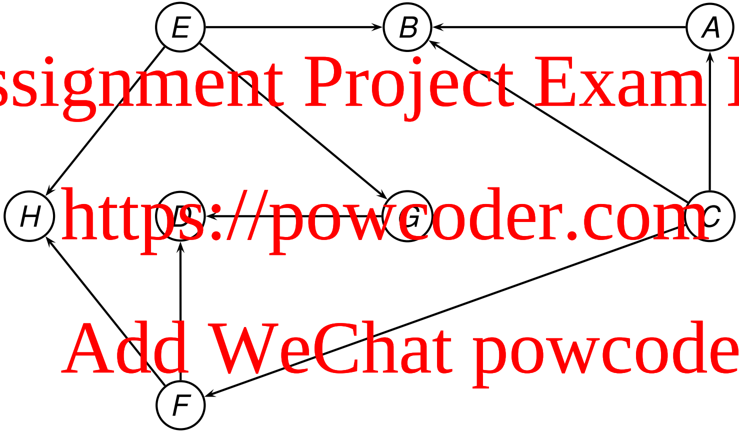
Topological Ordering:

A topological ordering of a DAG, $G = (V, E)$, is a permutation of the vertices in V that is compatible with each edge in E

A permutation $(v_1, \ldots, v_n)$ is ***compatible*** *with edge (v, w) if*

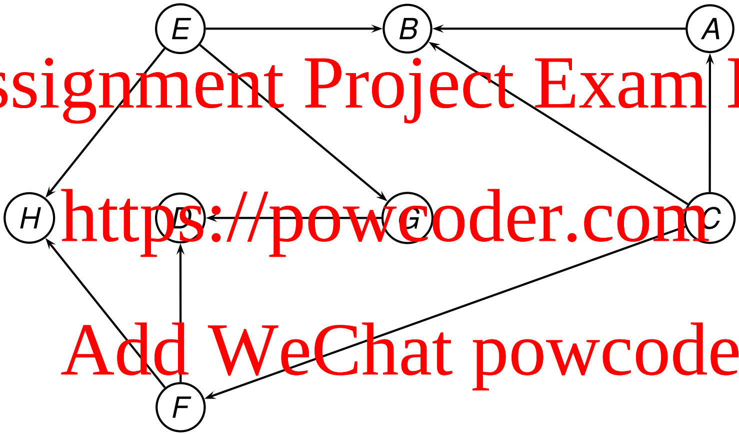
$$v = v_i, \quad w = v_j \quad \text{and} \quad i < j$$

Examples of Topological Sorts



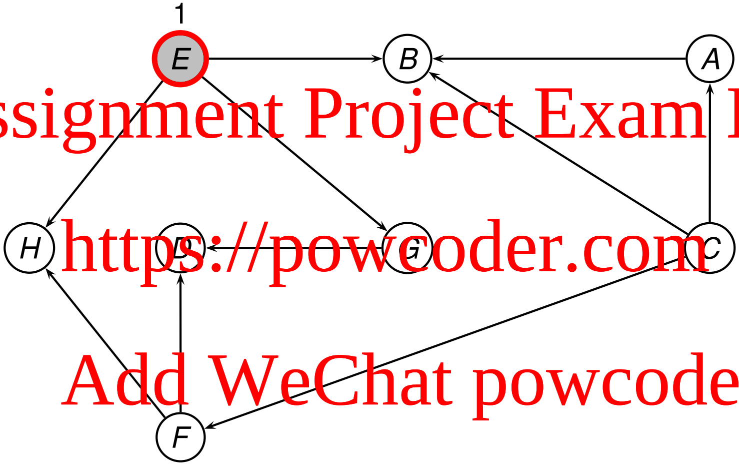
Let’s find a topological sort of this DAG

Examples of Topological Sorts

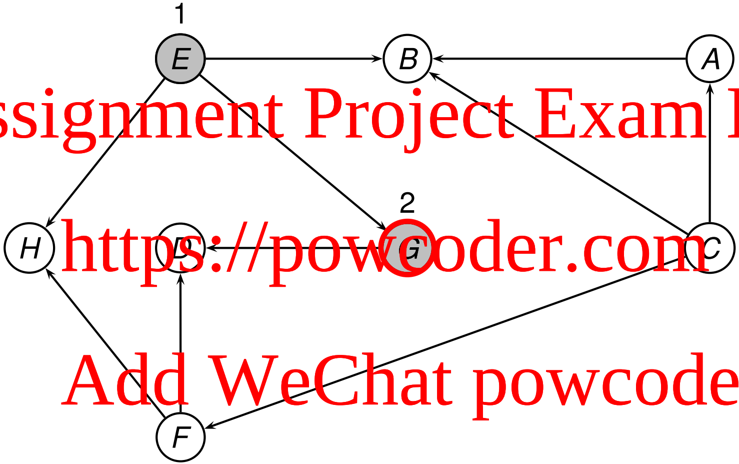


We could schedule C or E, let's choose to schedule E

Examples of Topological Sorts

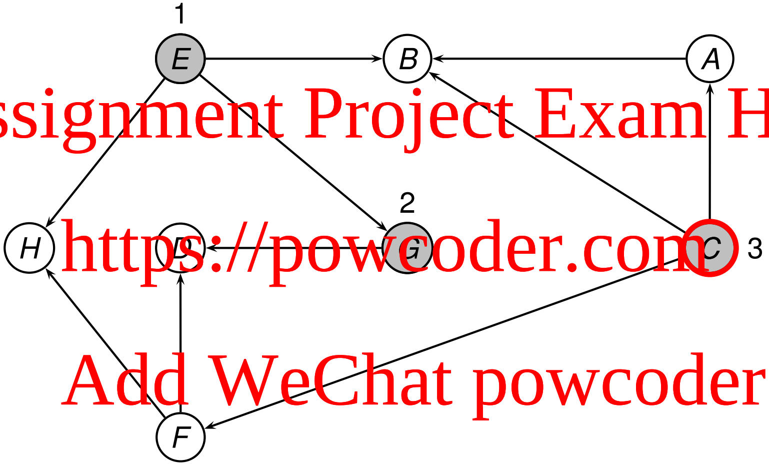


Examples of Topological Sorts



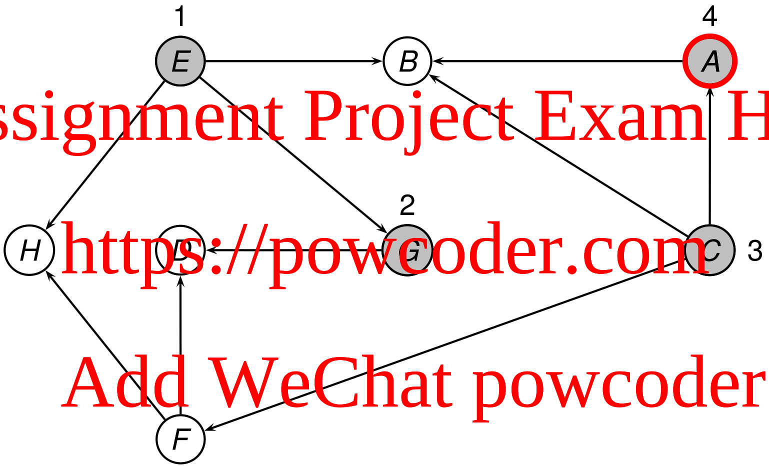
Now we could schedule G or C, let's choose G

Examples of Topological Sorts



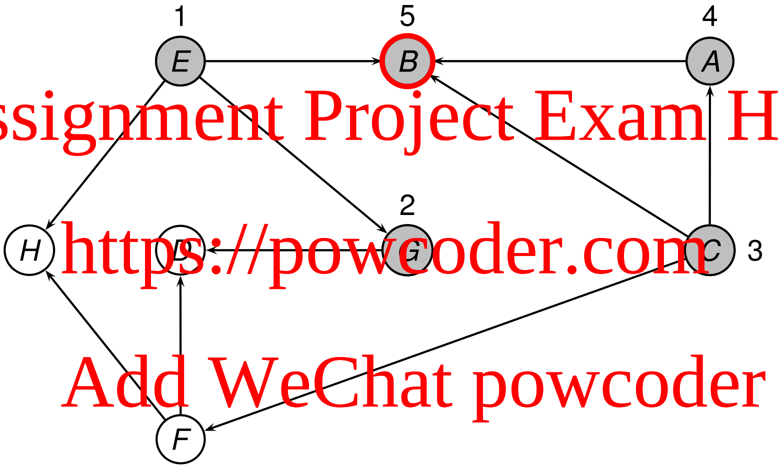
Now let's schedule *C*

Examples of Topological Sorts



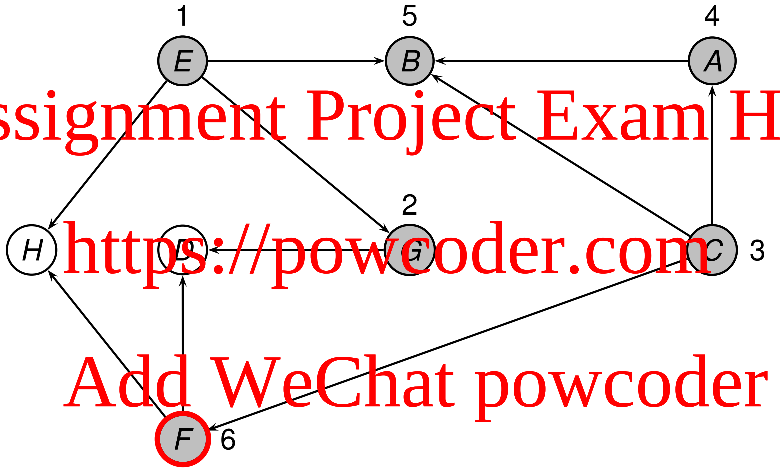
Now let's schedule A

Examples of Topological Sorts



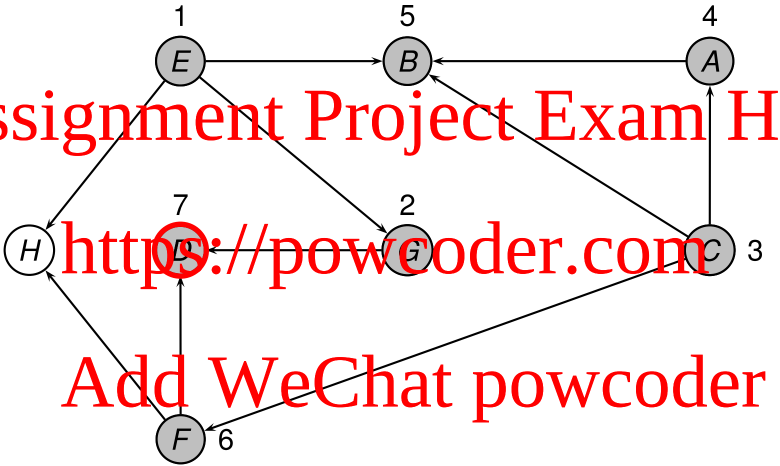
Now let's schedule *B*

Examples of Topological Sorts



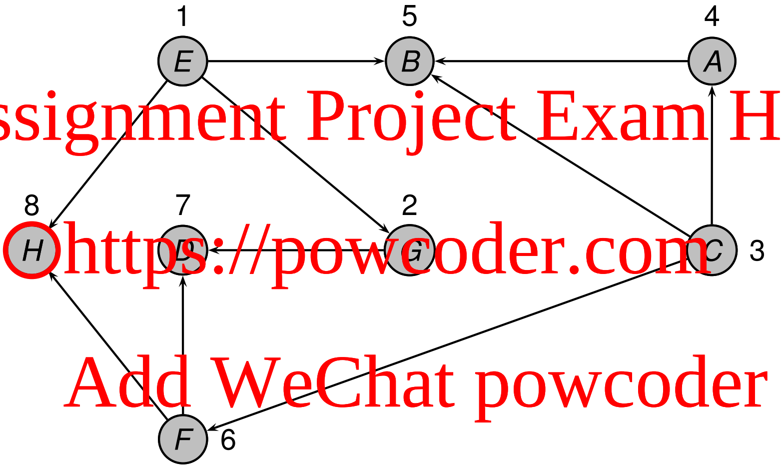
Now let's schedule F

Examples of Topological Sorts



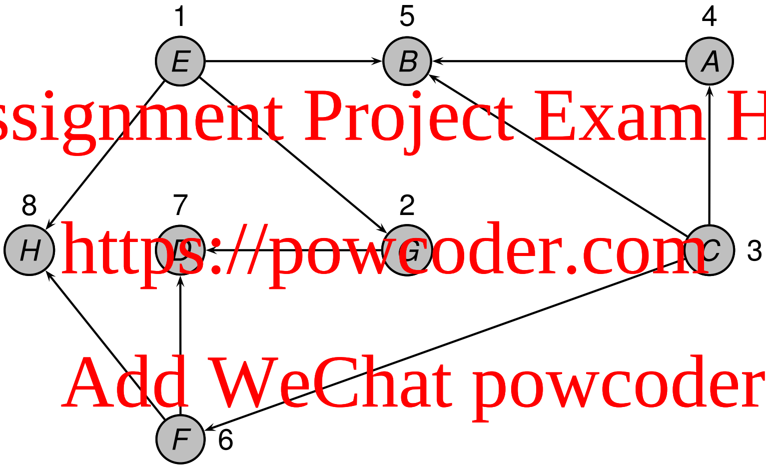
Now let's schedule D

Examples of Topological Sorts



Finally, let's schedule *H*

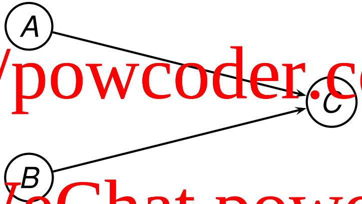
Examples of Topological Sorts



So the topological sort is: (E, G, C, A, B, F, D, H)

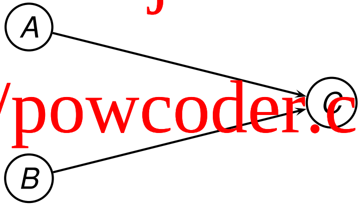
Question for you

- Give all topological sorts of this graph.



Question for you

- Give all topological sorts of this graph.



- (A, B, C)
- (B, A, C)

Question for you

- Give a graph containing 5 vertices with exactly one topological sort

Question for you

- Give a graph containing 5 vertices with exactly one topological sort

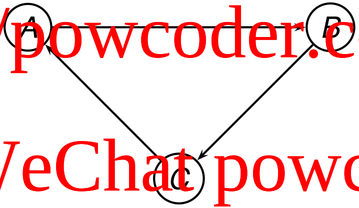
$$A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$$

Question for you

- Give a graph containing 3 vertices with no topological sorts

Question for you

- Give a graph containing 3 vertices with no topological sorts



Topological Sorting Algorithm

Question: When can a task be safely scheduled?

Answer: when everything that it depends on has already been scheduled

- We need to start with a task that depends on nothing
- This corresponds to a node without any **incoming** edges

The Incoming Edge Count

For each node the total incoming edge count is a measure of how "close" it is to a task that is ready to be scheduled

Once a task X is scheduled, can reduce incoming edge count for all tasks that depend on X.

In effect, the edges out of X are deleted once X has been scheduled.

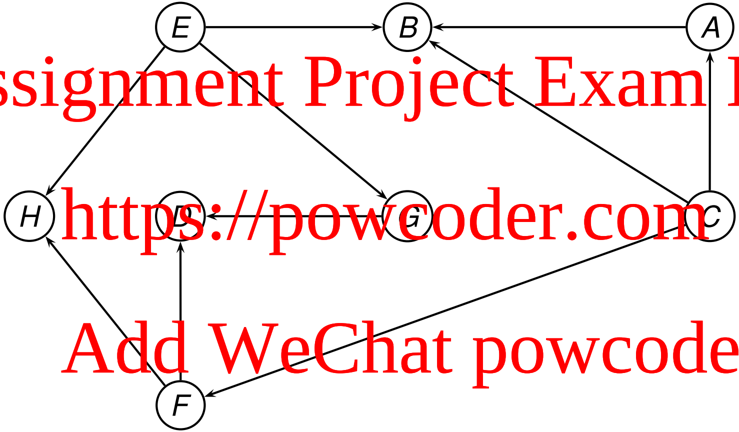
Once incoming edge count for a node reduces to 0 it can be scheduled

Topological Sort Algorithm

TopologicalSort(G) :

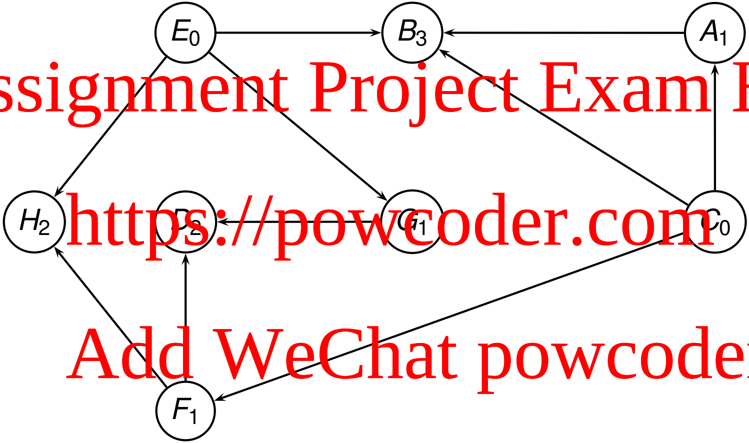
    let S *be an empty stack*
    for each vertex $v \in V$
        compute indegree(v)
        if indegree(v) $= 0$ *then push* v *onto* S
    while S *is not empty*
        pop u *from* S
        schedule u
        for each $(u, w) \in E$
            decrement indegree(w) *by* 1
            if indegree(w) $= 0$ *then*
                push w *onto* S

Illustration of Topological Sort Algorithm



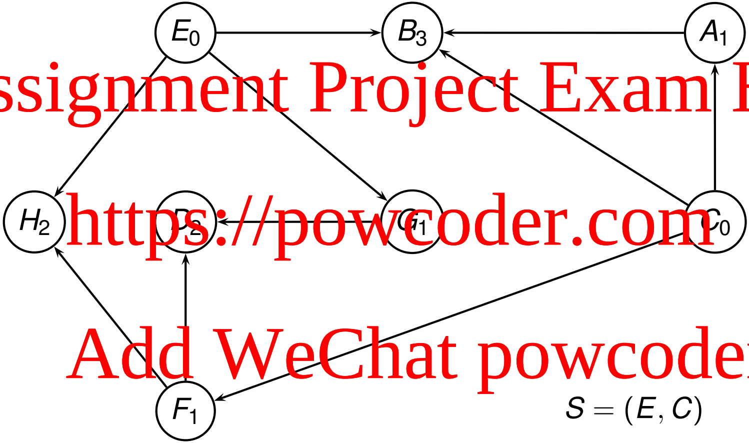
Compute indegree of each vertex

Illustration of Topological Sort Algorithm



Push vertices with 0 indegree on stack

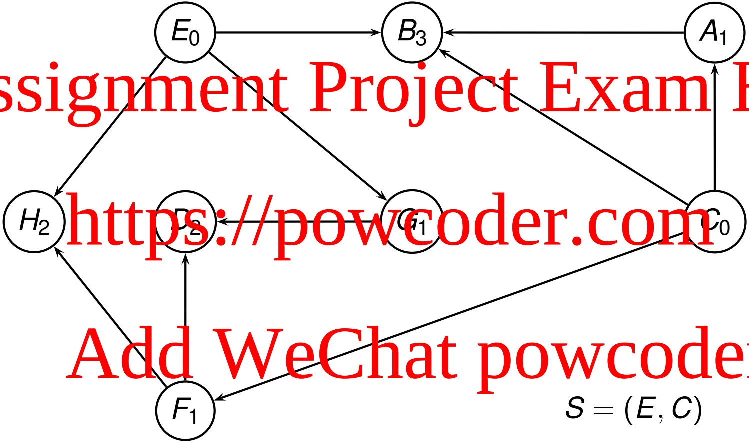
Illustration of Topological Sort Algorithm



$S = (E, C)$

Push vertices with 0 indegree on stack

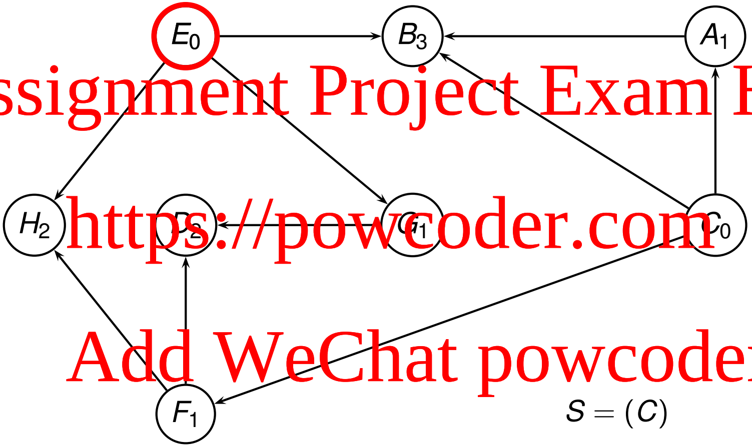
Illustration of Topological Sort Algorithm



$S = (E, C)$

Pop E from stack

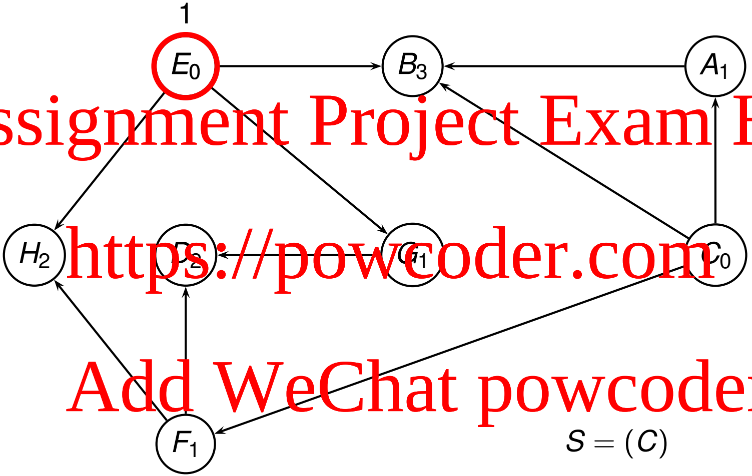
Illustration of Topological Sort Algorithm



$S = (C)$

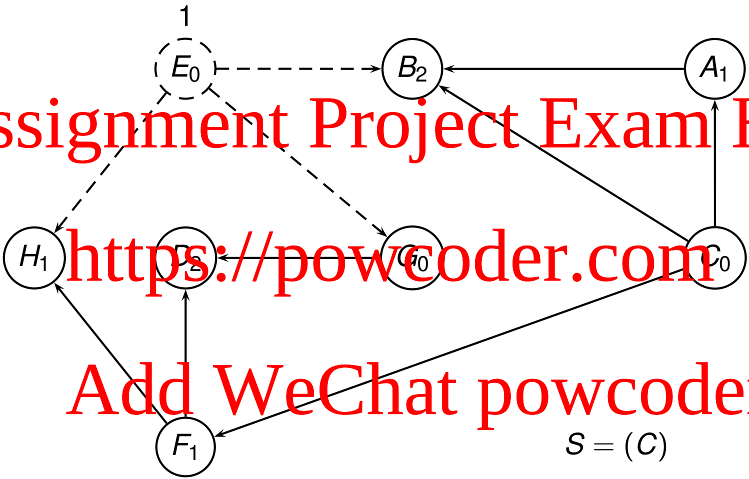
Schedule E

Illustration of Topological Sort Algorithm



"Remove" all edges out of E, updating indegree values

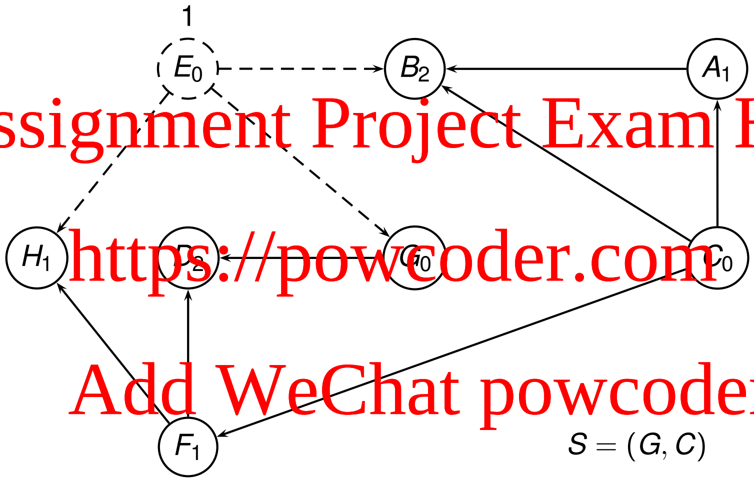
Illustration of Topological Sort Algorithm



$S = (C)$

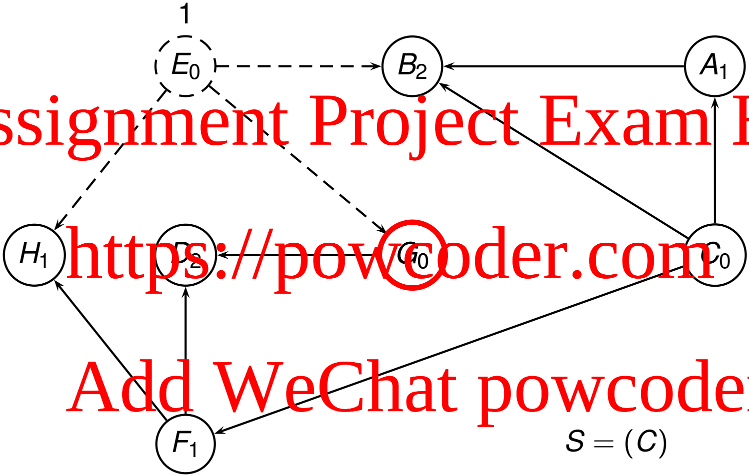
Push G onto stack

Illustration of Topological Sort Algorithm



Pop G from stack

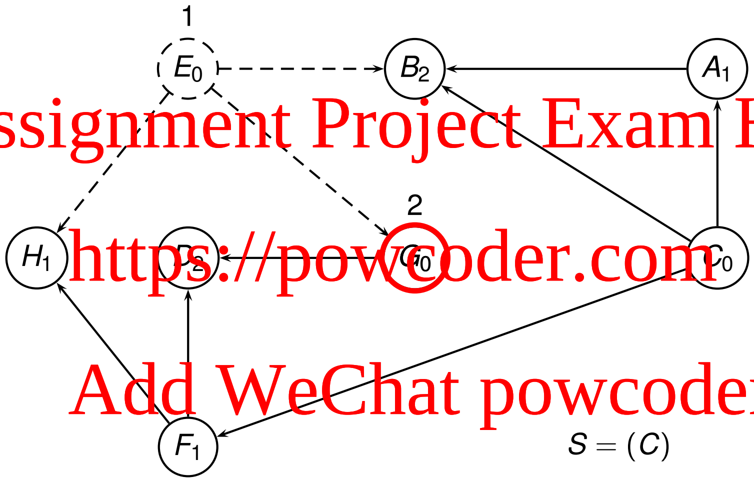
Illustration of Topological Sort Algorithm



$S = (C)$

Schedule G

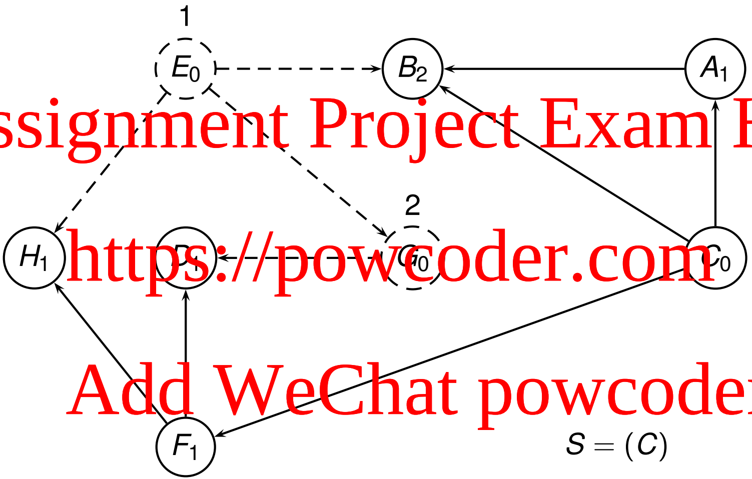
Illustration of Topological Sort Algorithm



$S = (C)$

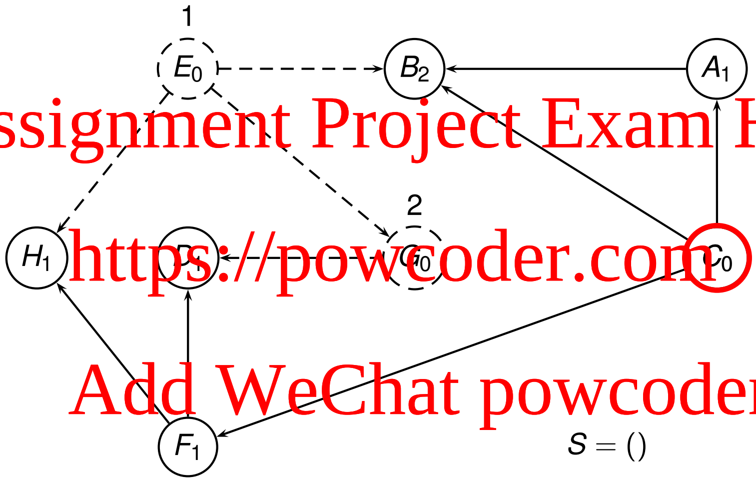
“Remove” all edges out of G, updating indegree values

Illustration of Topological Sort Algorithm



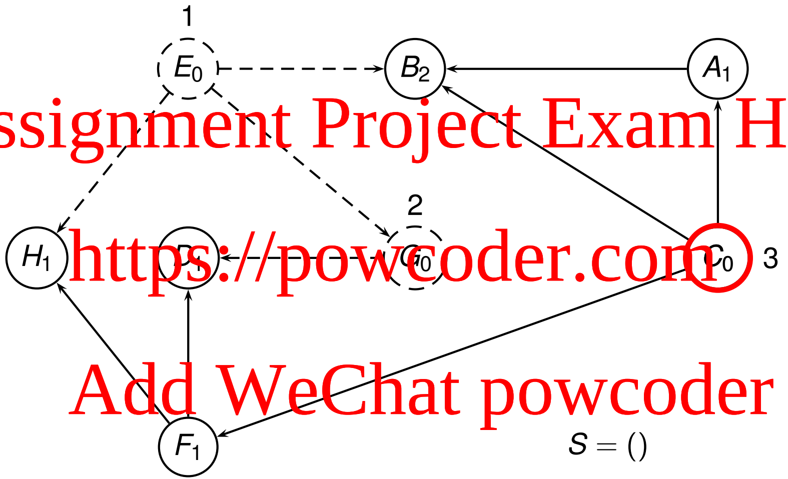
Pop C from stack

Illustration of Topological Sort Algorithm



Schedule C

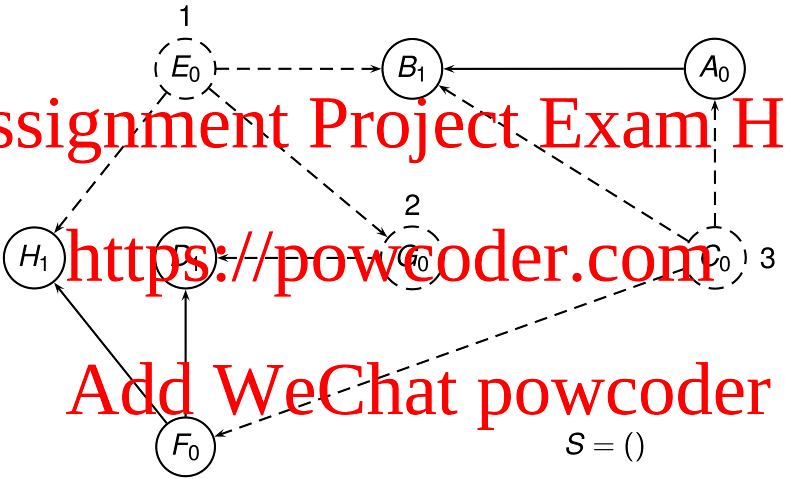
Illustration of Topological Sort Algorithm



$S = ()$

“Remove” all edges out of C, updating indegree values

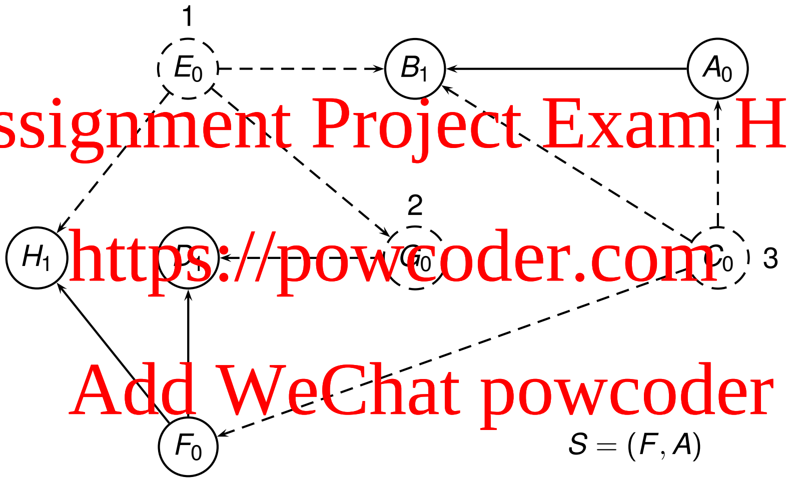
Illustration of Topological Sort Algorithm



$S = ()$

Push both A and F onto stack

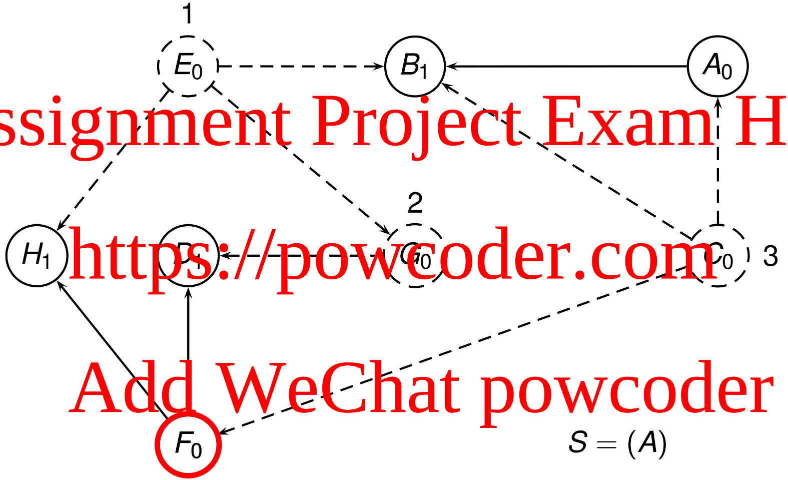
Illustration of Topological Sort Algorithm



$S = (F, A)$

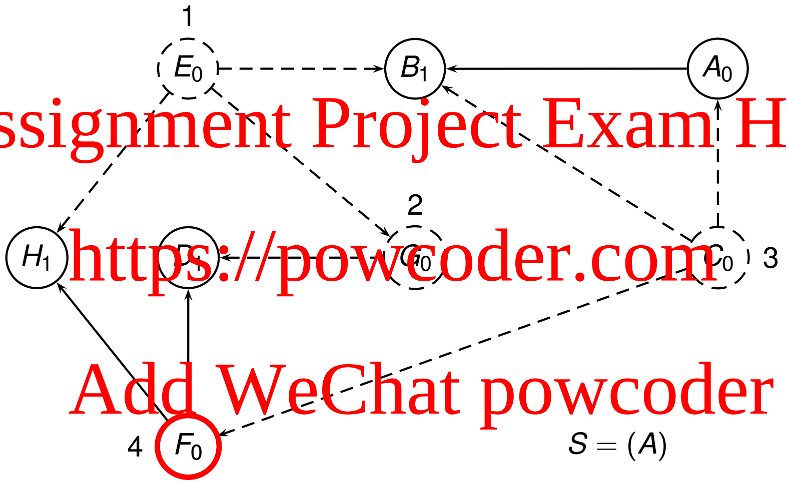
Pop F from stack

Illustration of Topological Sort Algorithm



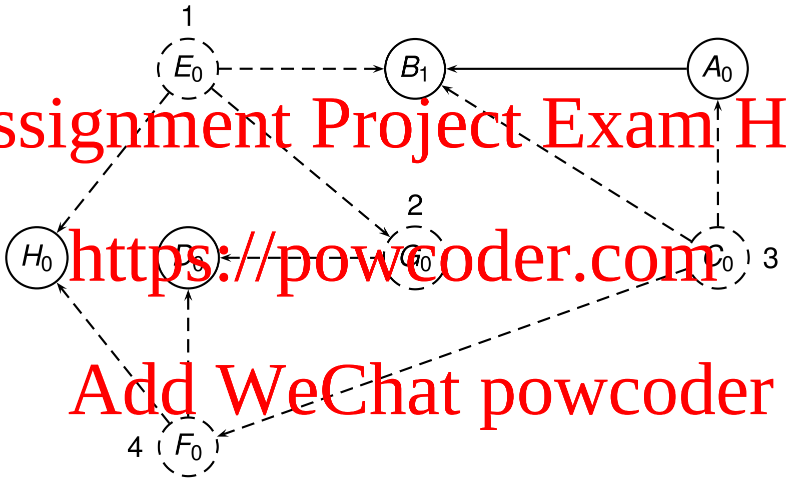
Schedule F

Illustration of Topological Sort Algorithm



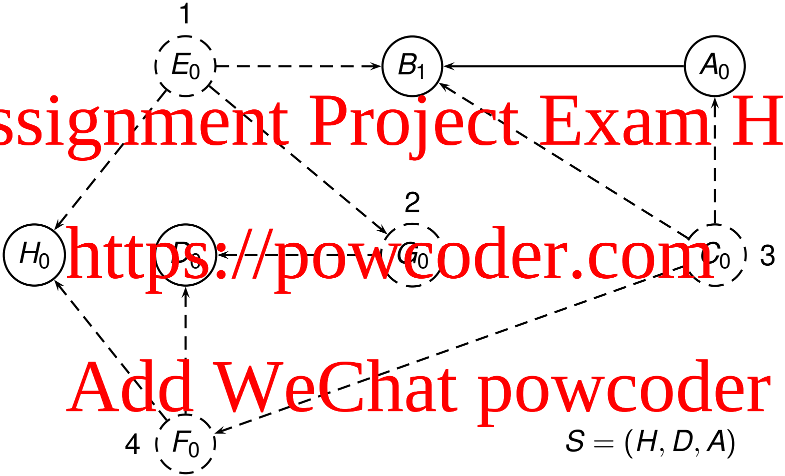
"Remove" all edges out of F, updating indegree values

Illustration of Topological Sort Algorithm



Push both D and H onto stack

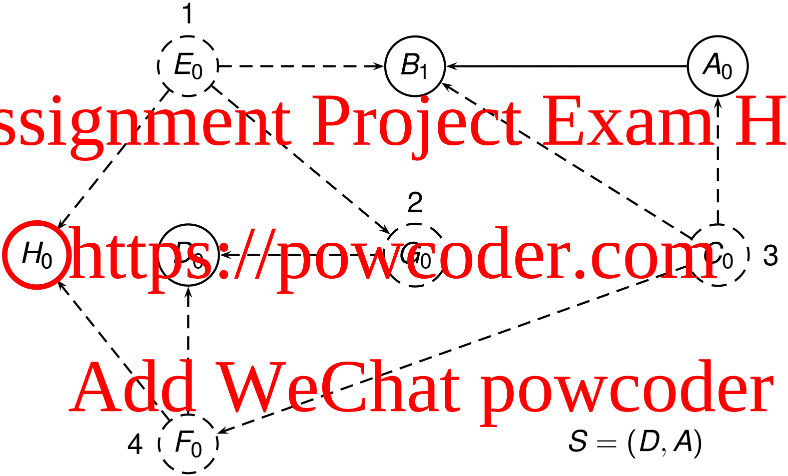
Illustration of Topological Sort Algorithm



$S = (H, D, A)$

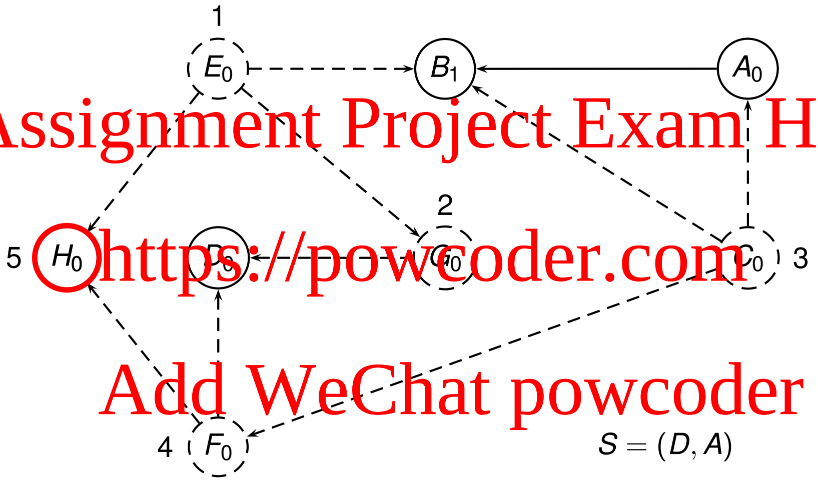
Pop H from stack

Illustration of Topological Sort Algorithm



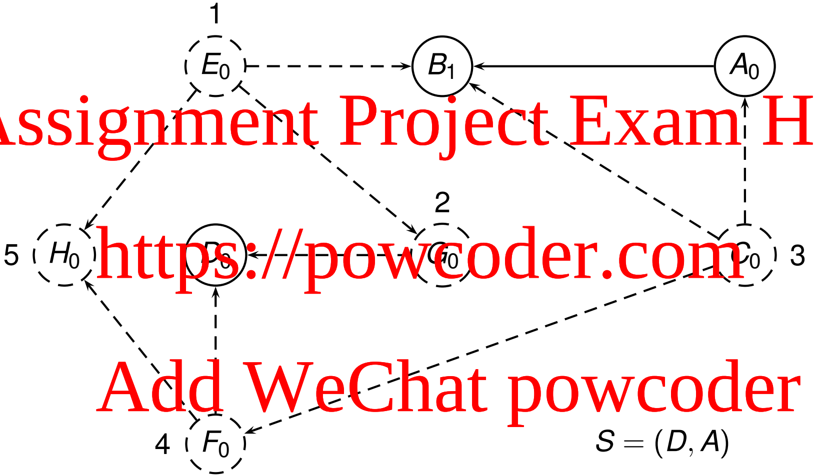
Schedule H

Illustration of Topological Sort Algorithm



No edges out of H to “remove” — no indegree values to update

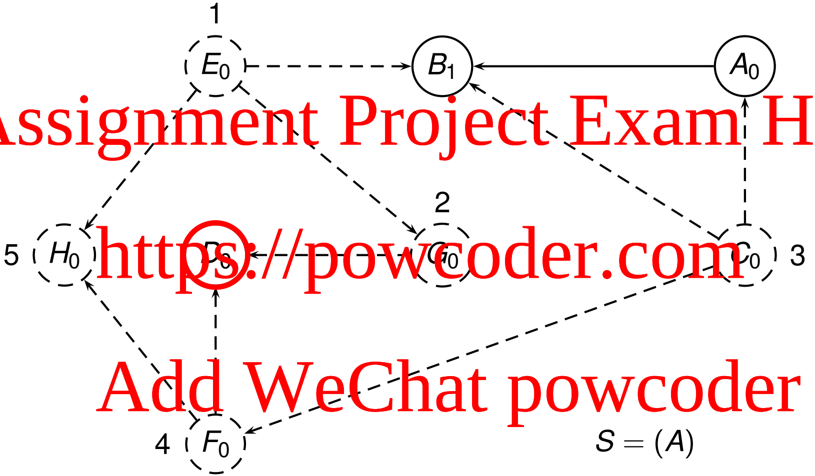
Illustration of Topological Sort Algorithm



$S = (D, A)$

Pop D from stack

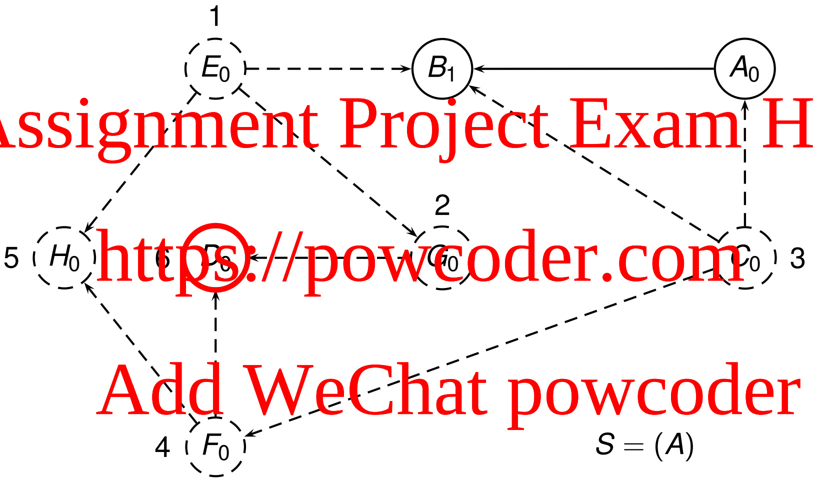
Illustration of Topological Sort Algorithm



$S = (A)$

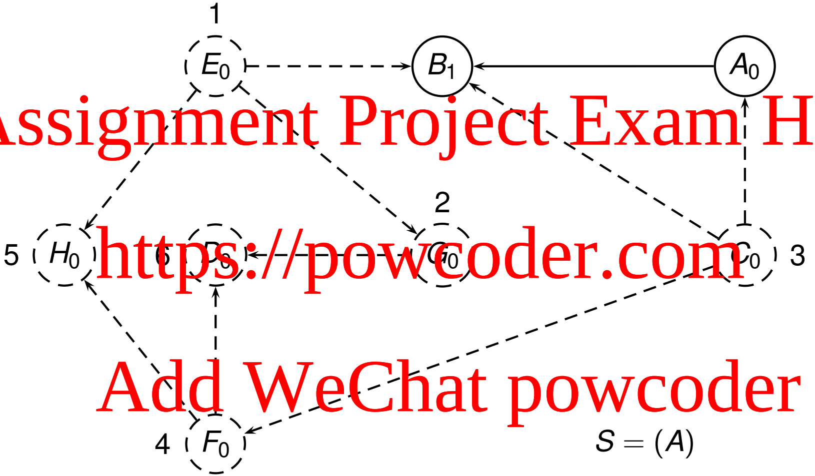
Schedule D

Illustration of Topological Sort Algorithm



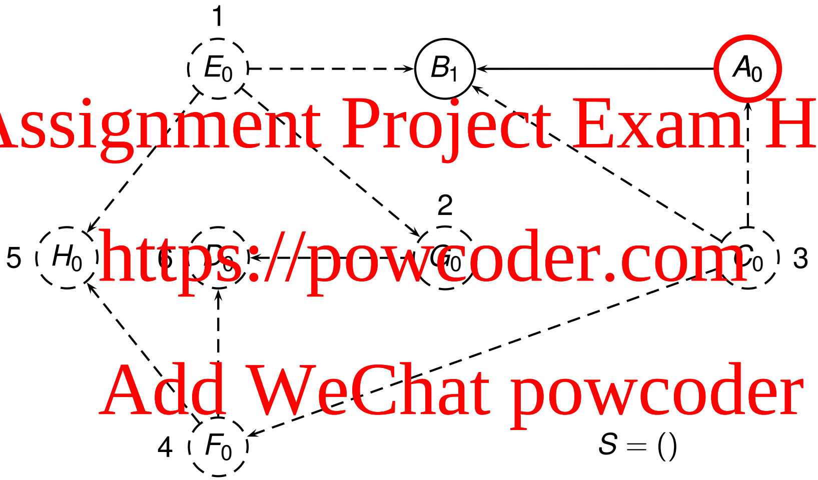
No edges out of D to “remove” — no indegree values to update

Illustration of Topological Sort Algorithm



Pop A from stack

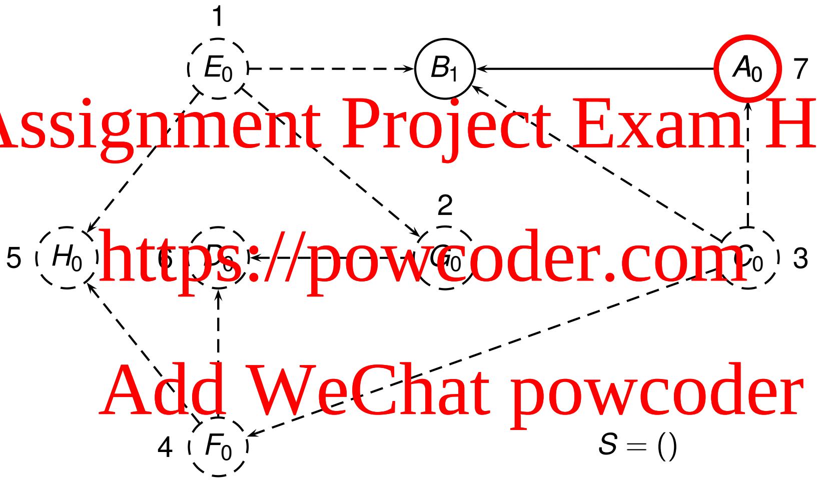
Illustration of Topological Sort Algorithm



$S = ()$

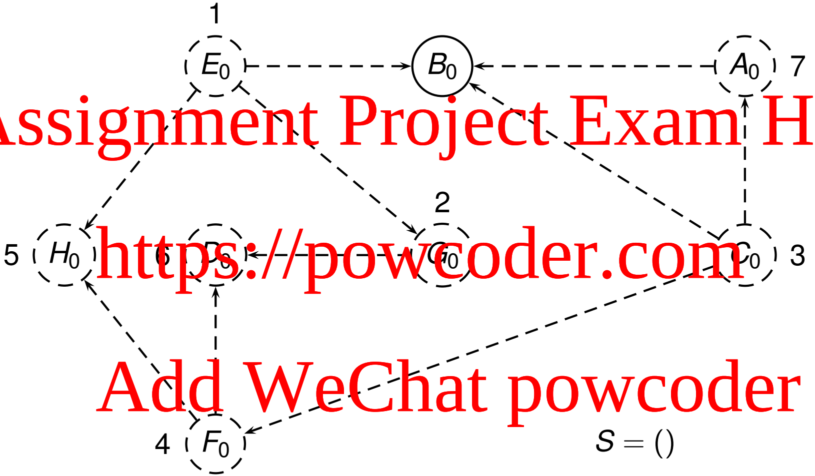
Schedule A

Illustration of Topological Sort Algorithm



"Remove" all edges out of A and update indegree values

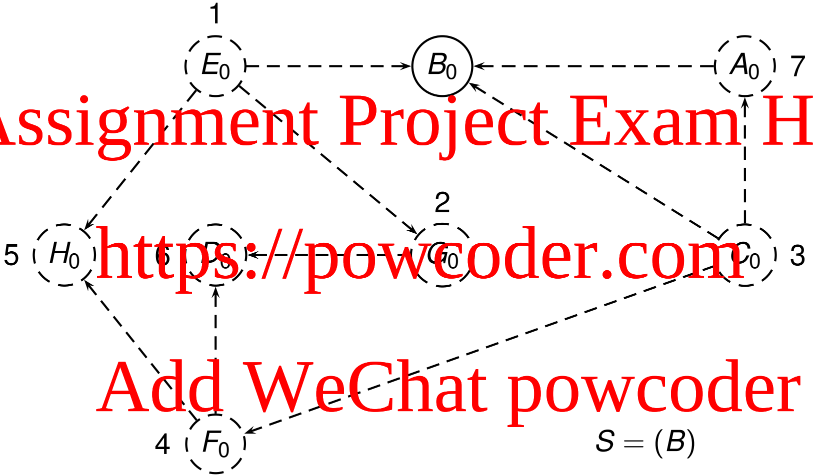
Illustration of Topological Sort Algorithm



$S = ()$

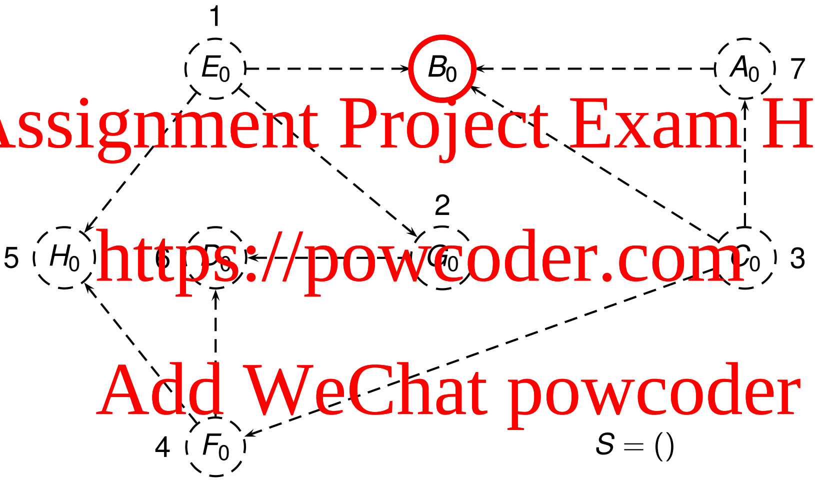
Push B onto stack

Illustration of Topological Sort Algorithm



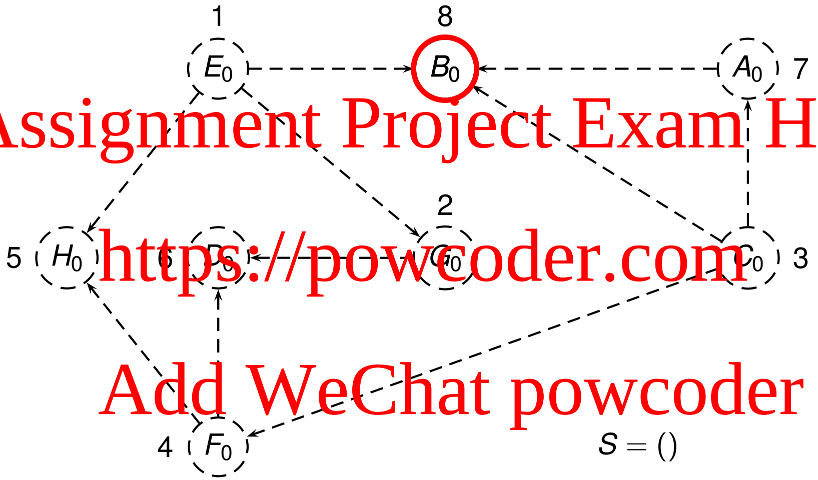
Pop B from stack

Illustration of Topological Sort Algorithm



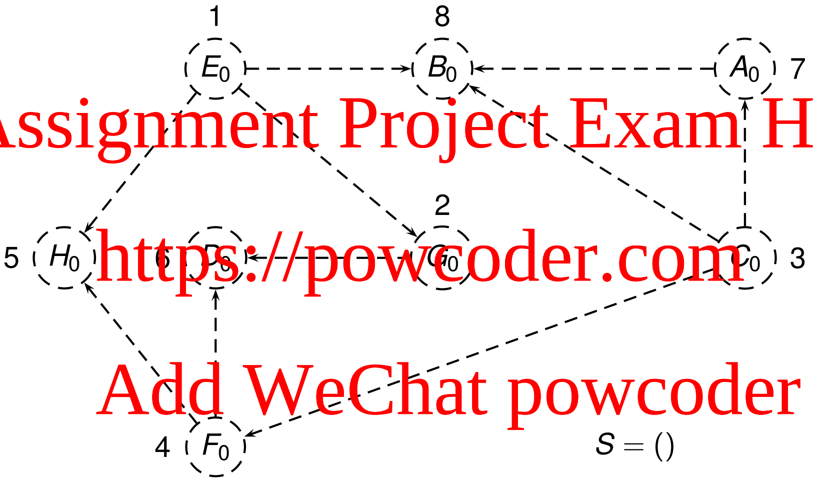
Schedule B

Illustration of Topological Sort Algorithm



No edges out of B to "remove" — no indegree values to update

Illustration of Topological Sort Algorithm



$S = ()$

All done!

Question for you

- What would happen if this topological sort algorithm was given a cyclic graph as input?

Question for you

- What would happen if this topological sort algorithm was given a cyclic graph as input?

The algorithm would terminate without scheduling any of the edges within a cycle.

Question for you

- What would happen when the algorithm is run on a graph with just one topological sort?

Question for you

- What would happen when the algorithm is run on a graph with just one topological sort?

Throughout the execution of the algorithm, the stack would contain no more than one element.

Topological Sort Algorithm (again)

```
TopologicalSort(G) :
  let S be an empty stack
  for each vertex v ∈ V
    compute indegree(v)
    if indegree(v) = 0 then push v onto S
  while S is not empty
    pop u from S
    schedule u
    for each (u, w) ∈ E
      decrement indegree(w) by 1
      if indegree(w) = 0 then
        push w onto S
```

Running Time

- Time to compute initial indegree values is $\Theta(n + m)$
  — traverse all adjacency lists
- How many times is the *while* loop executed?
- Measure of progress is number of scheduled vertices
  — always increases by 1 for each iteration
- Number of steps within loop is $\Theta(\max(1, \mathit{outdegree}(v)))$
- We know that in total there are m edges to consider
- Total across all iterations of loop:

$$\sum_{v \in V} \max(1, \mathit{outdegree}(v)) = \Theta(n + m)$$

Question for you

- Would the algorithm be correct if a queue rather than a stack was used to hold the schedulable vertices?

Question for you

- Would the algorithm be correct if a queue rather than a stack was used to hold the schedulable vertices?

Yes, the order of elements within the stack (or the queue) is irrelevant

Question for you

- Would the use of queues rather than stacks have an impact on the asymptotic running time of the algorithm?

Question for you

- Would the use of queues rather than stacks have an impact on the asymptotic running time of the algorithm?

No, though stacks are slightly more efficient to implement.

Question for you

- How could the algorithm be adpated to cater for a scenario where some jobs are more urgent than others?

Question for you

- How could the algorithm be adpated to cater for a scenario where some jobs are more urgent than others?

Use a priority queue rather than a stack.