Unformed Search

Muxi Chen

Email: mxchen21@cse.cuhk.edu.hk

Office: SHB1013, CUHK

The Chinese University of Hong Kong

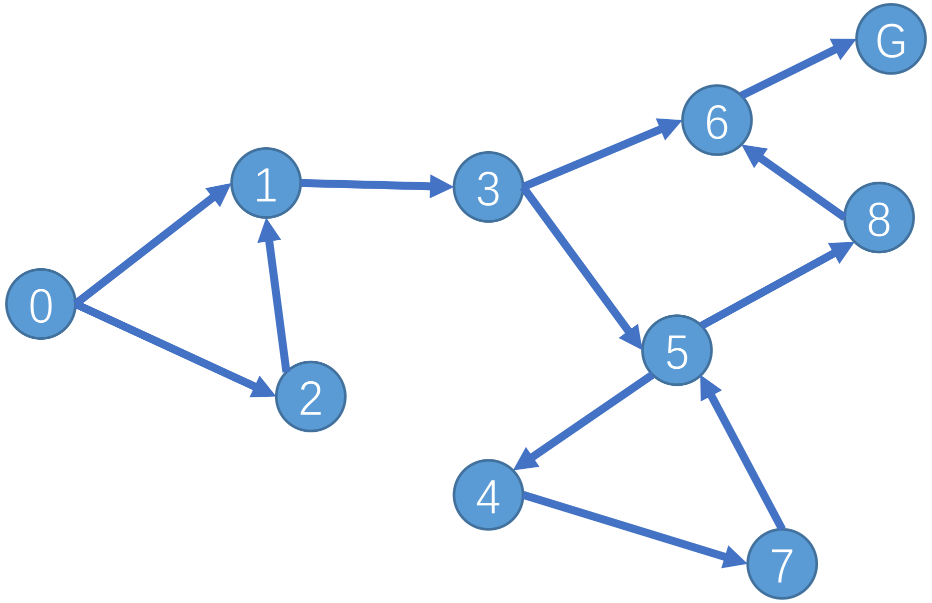
November 14, 2021

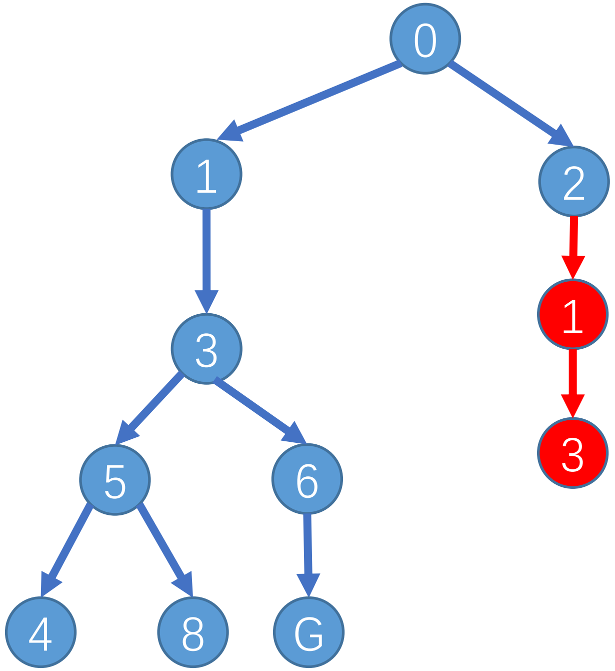
Outline

- Depth-first search
- Breadth-first search
- Iterative Deepening Search
- Uniform-Cost Search

Graph Search

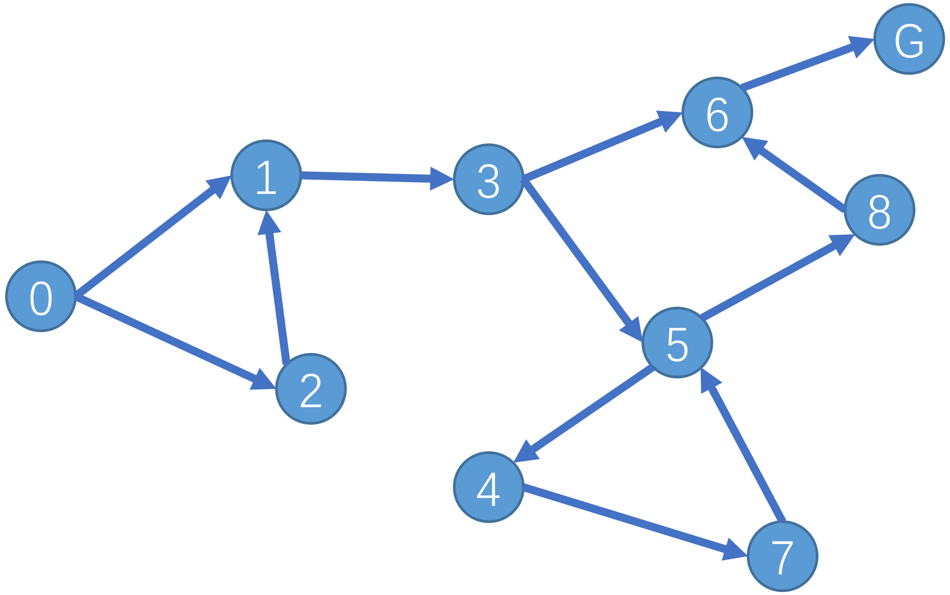
- Idea: never expand a state twice





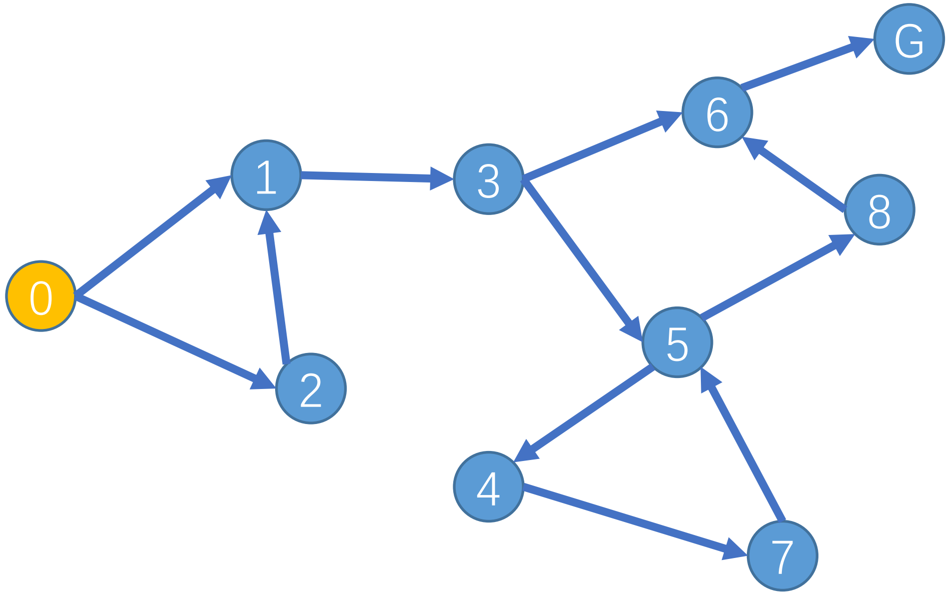
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



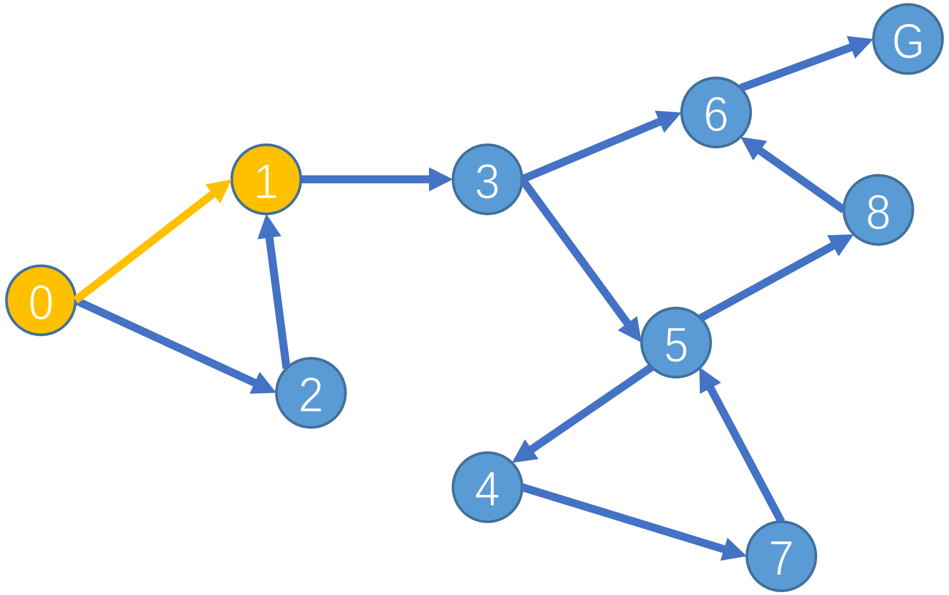
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



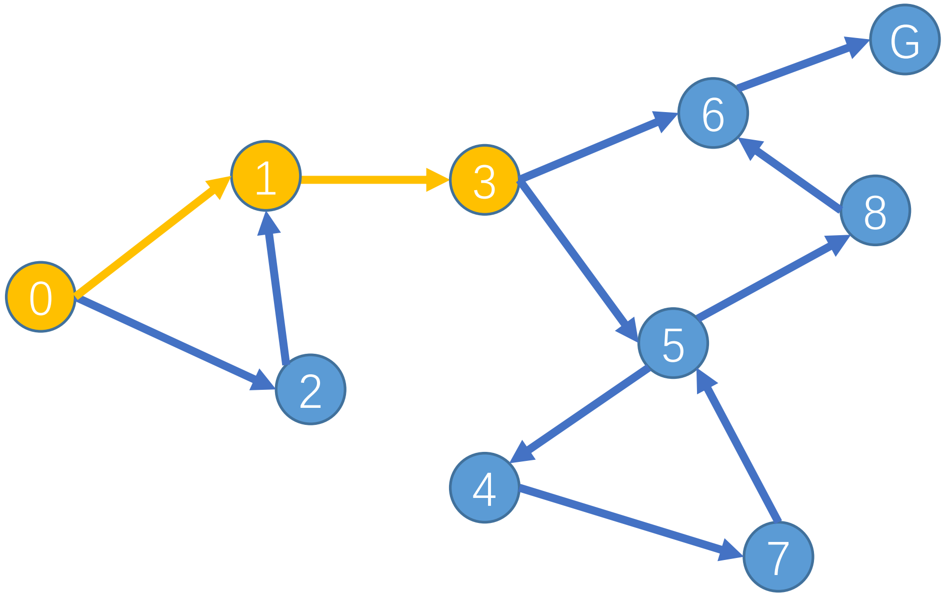
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



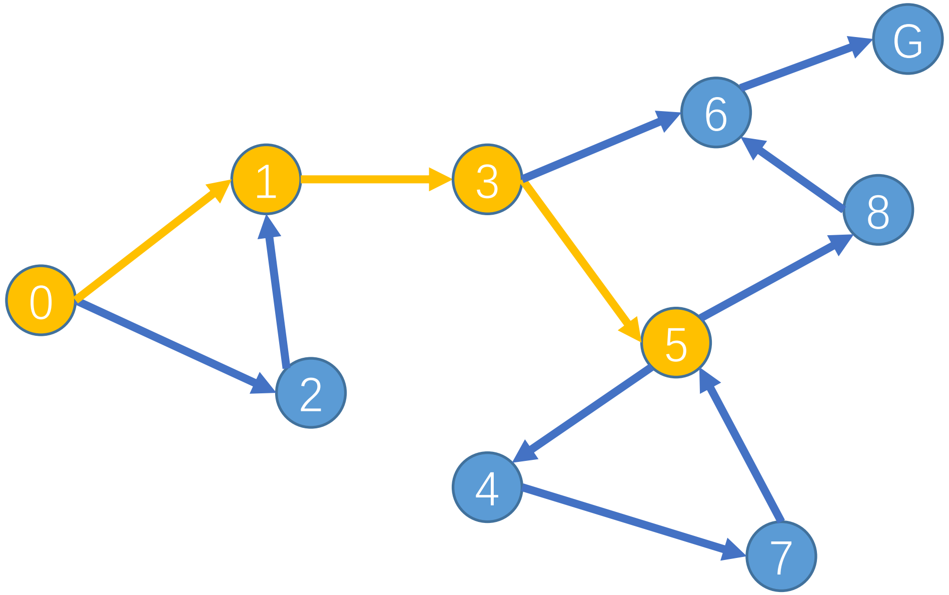
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



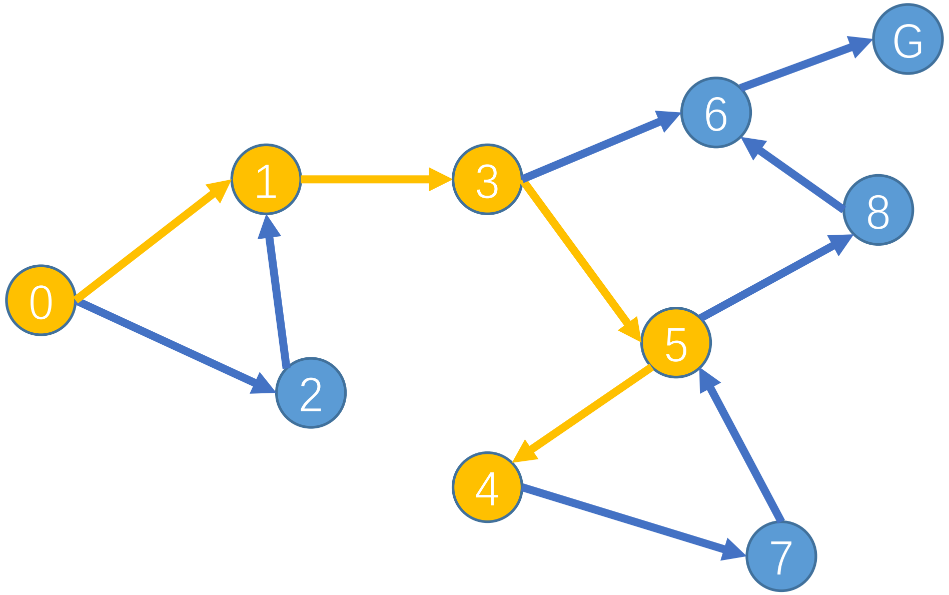
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



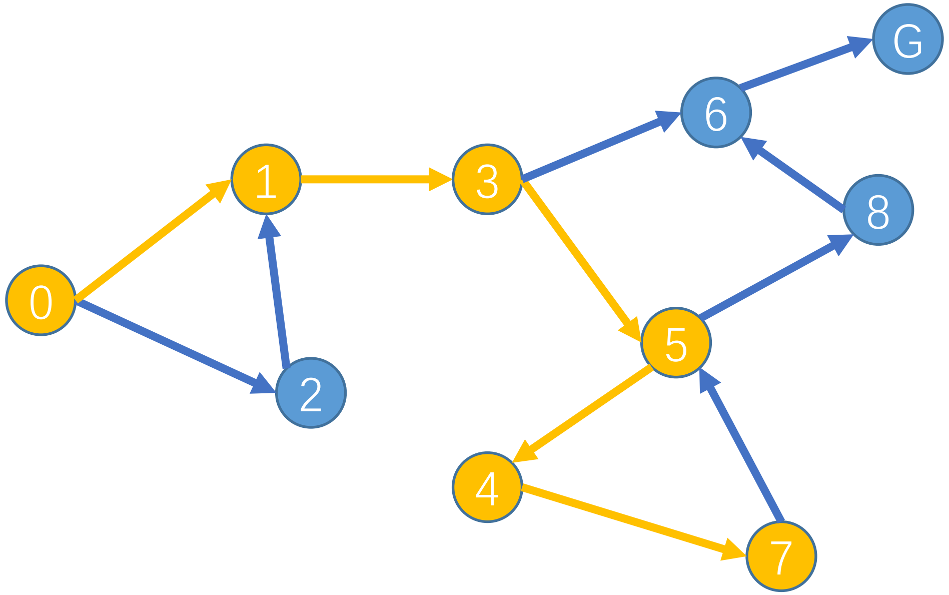
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



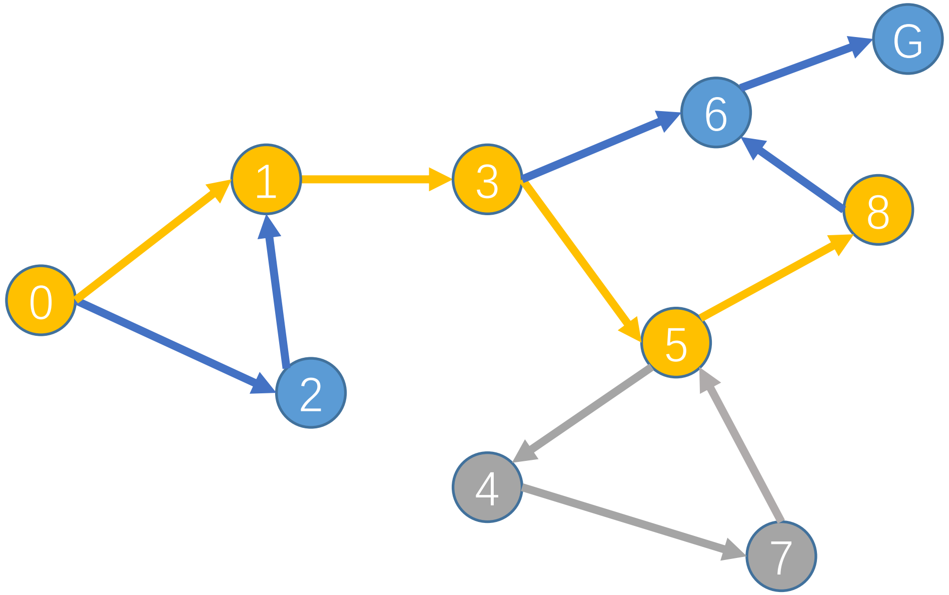
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



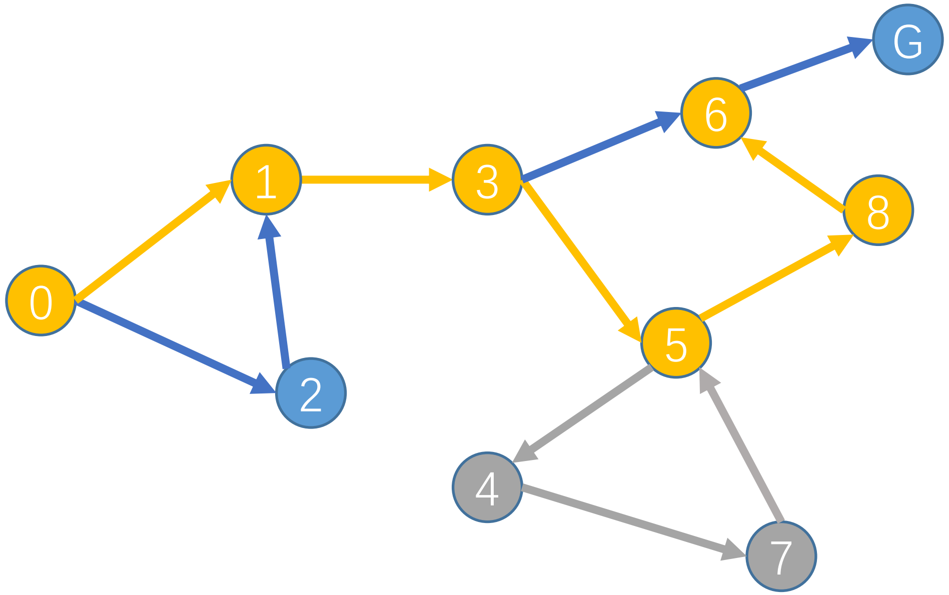
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



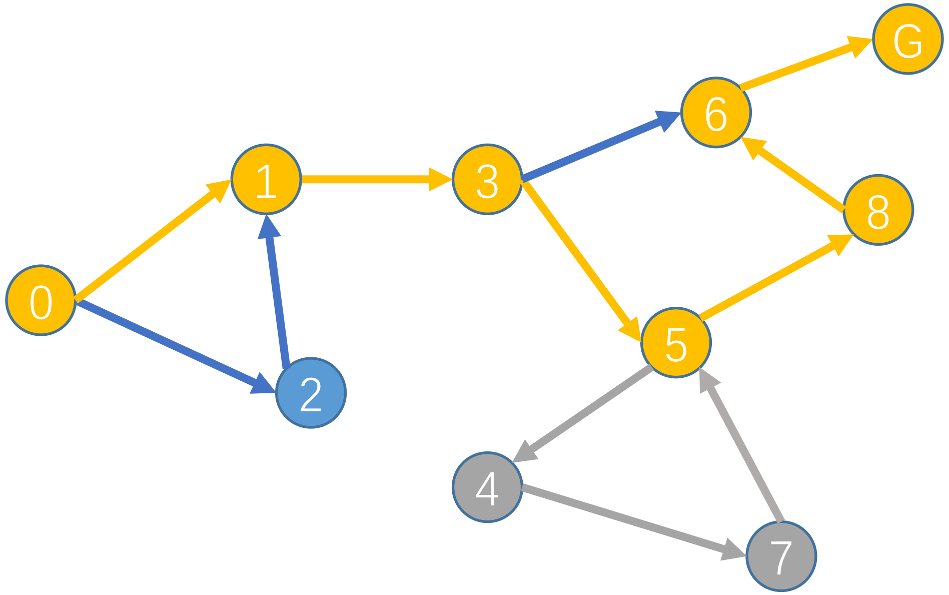
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



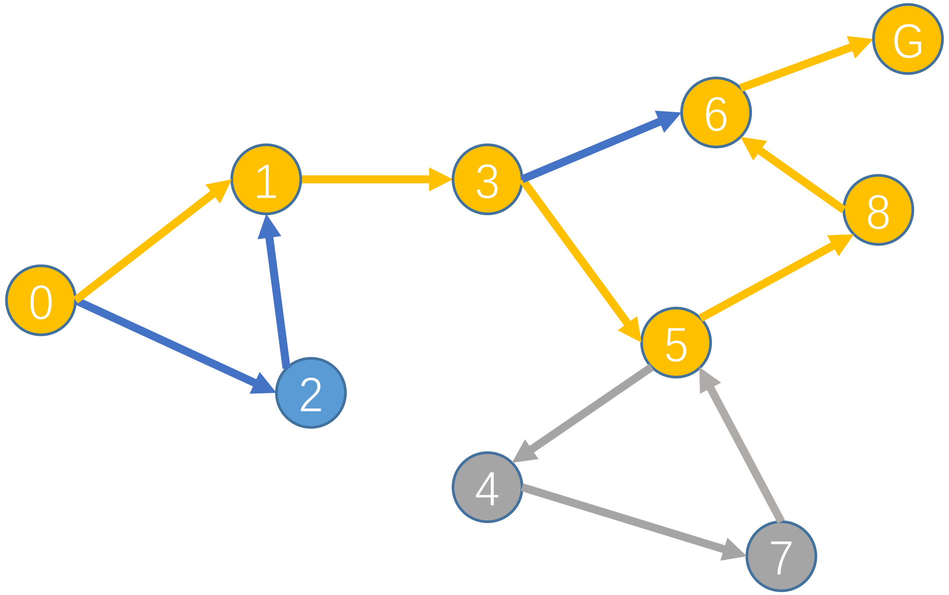
Depth-first search

- Always expands one of the nodes at the deepest level of the tree
- Only when the search hits a dead end (a non-goal node with no expansion) does the search go back and expand nodes at shallower level and so on



Recursive algorithm for DFS

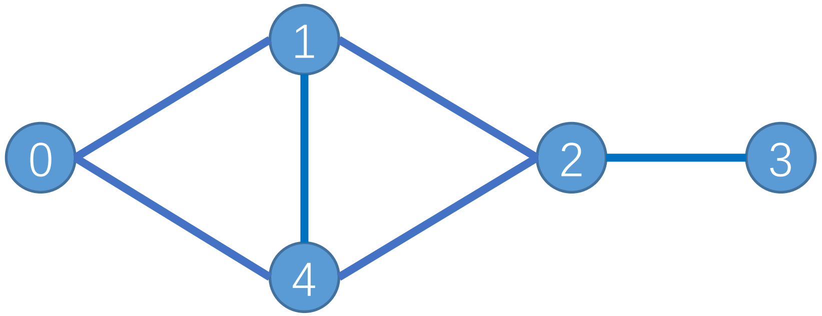
- Naturally recursive structure
  - Visit a node
  - Run DFS on its neighbors

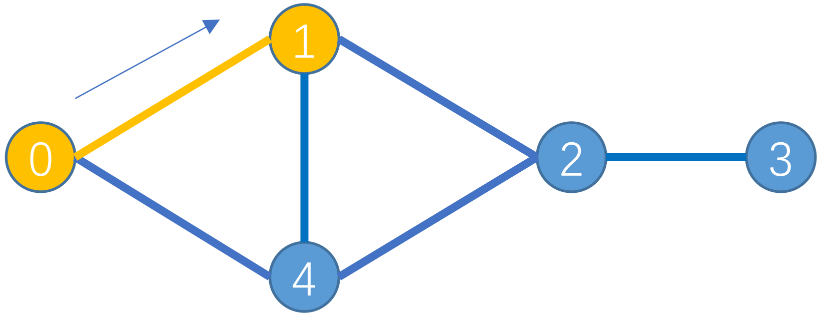


Recursive algorithm for DFS

Goal: Traverse the graph

```
Def DFS_recursive(node):
    visit(node)
    For v in Neighbors(node):
        DFS_recursive(v)
```

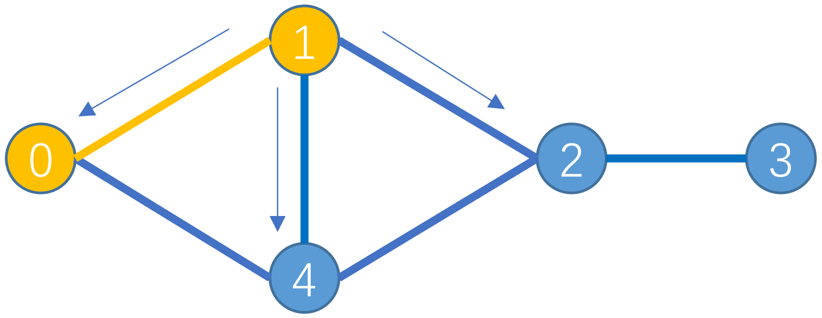




Recursive algorithm for DFS

Goal: Traverse the graph

```
Def DFS_recursive(node):
    visit(node)
    For v in Neighbors(node):
        DFS_recursive(v)
```

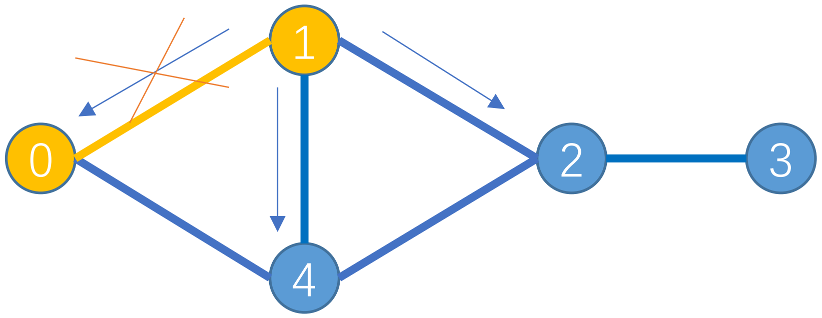


Node 0 will be visited again!
Graph search do not visit a node twice

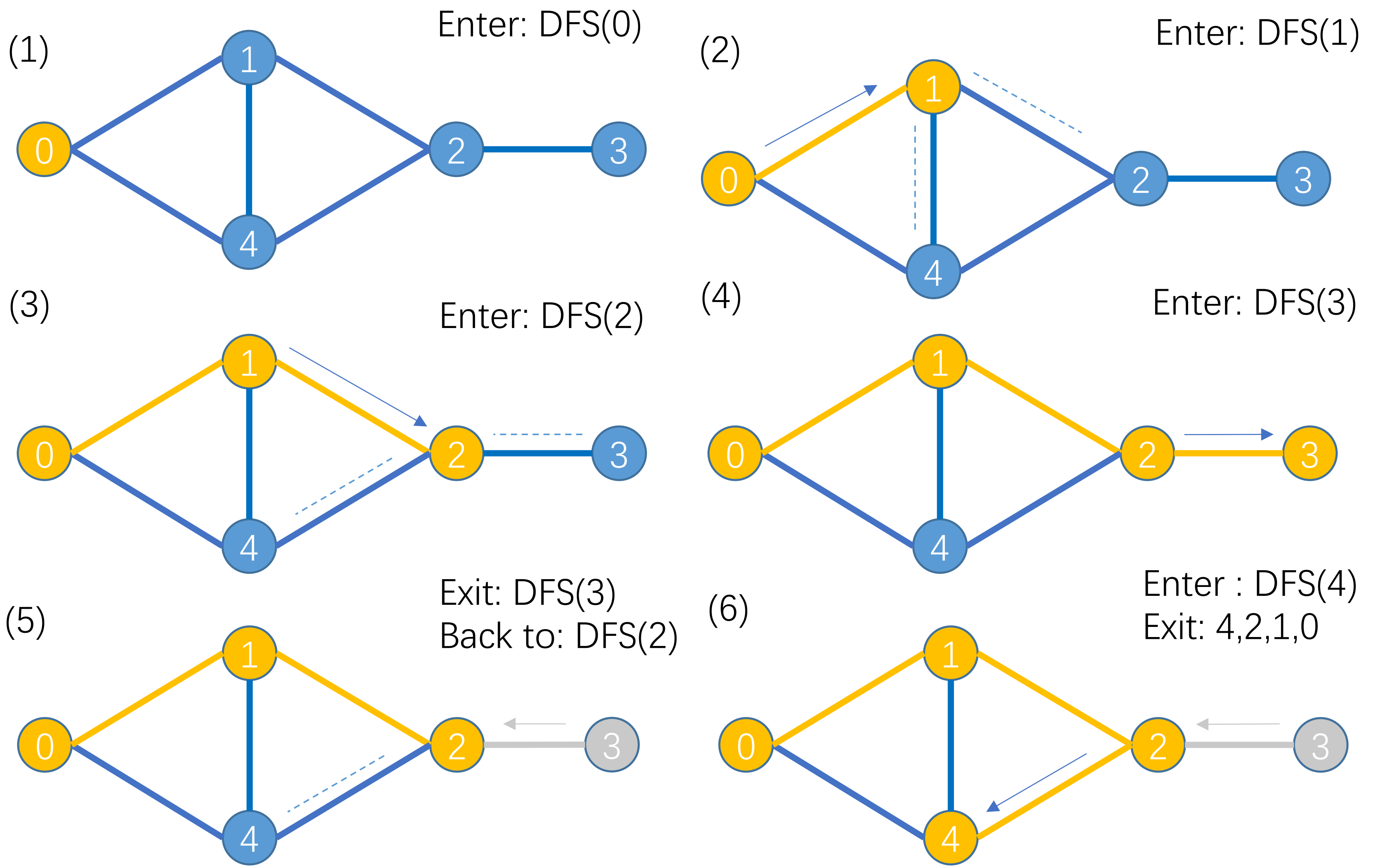
Recursive algorithm for DFS

Only call DFS on unvisited node

```
visited_list = [ ]
Def DFS_recursive(node):
    visit(node)
    visited_list.append(node)
    For v in Neighbors(node):
        if v not in visited_list:
            DFS_recursive(v)
```



Recursive algorithm for DFS



(1) Enter: DFS(0)

(2) Enter: DFS(1)

(3) Enter: DFS(2)

(4) Enter: DFS(3)

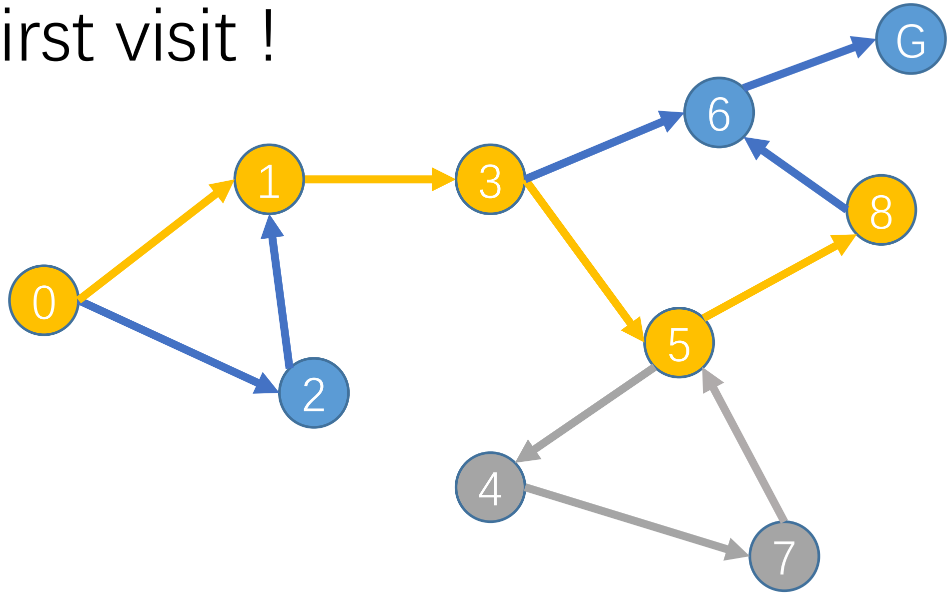
(5) Exit: DFS(3)
Back to: DFS(2)

(6) Enter : DFS(4)
Exit: 4,2,1,0

```
visited_list = [ ]
Def DFS_recursive(node):
        visit(node)
        visited_list.append(node)
        For v in Neighbors(node):
                if v not in visited_list:
                        DFS_recursive(v)
```

Iterative algorithm for DFS

- Always expands one of the nodes at the deepest level of the tree
    - 0 > 1 > 3 > 5
- Meet a node with no expansion, go back to the node at a shallower level
    - After visit node 7, back to visit node 5’ s neighbor -–- node 8
- Last found node, first visit !



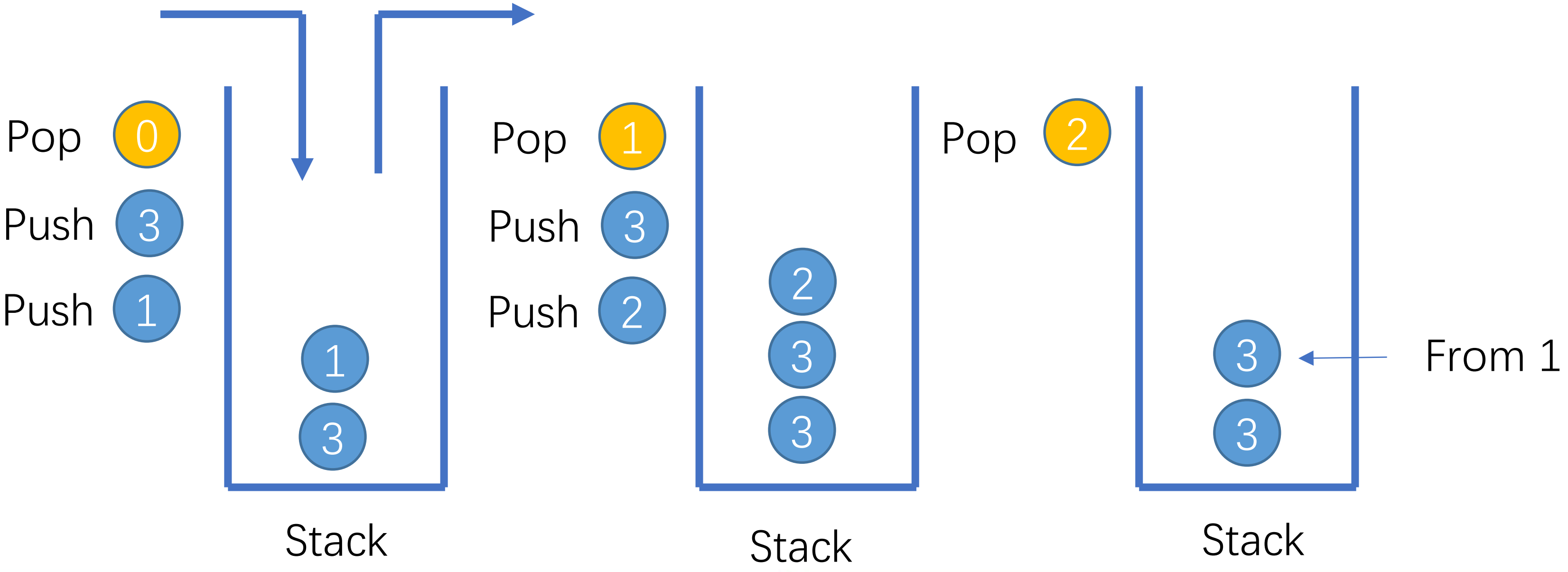
Iterative algorithm for DFS

Stack matches the process of DFS

Push: Store neighbors of current node

Pop: Choose one neighbor of last node

Last in First out: Expand nodes at deepest level



Pop 0

Push 3

Push 1

Stack

Pop 1

Push 3

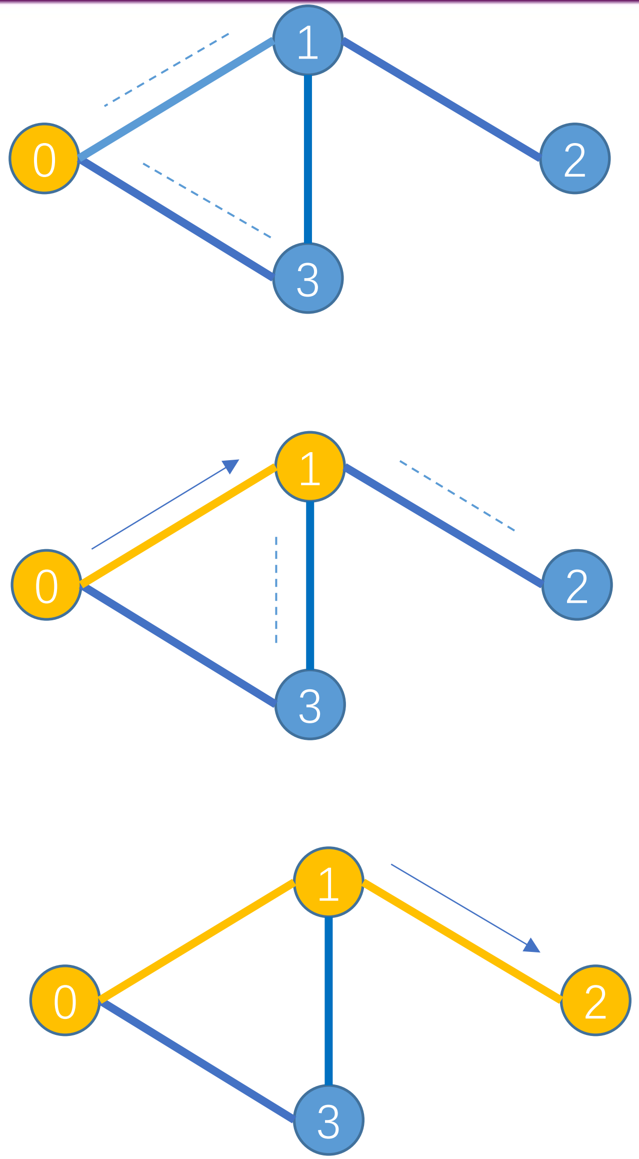
Push 2

Stack

Pop 2

From 1

Stack

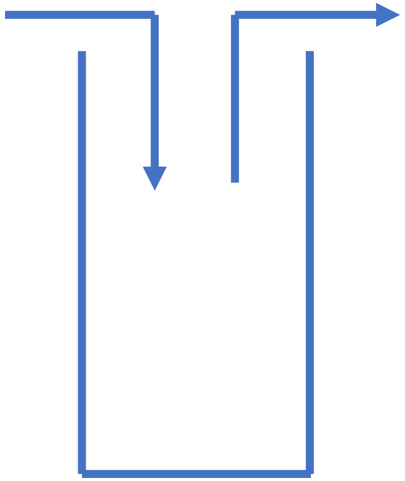


Iterative algorithm for DFS

Use stack to keep track of nodes

Goal: Traverse the graph

```
visited_list = [ ]
Def DFS_iterative(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    stack.push(v)
```

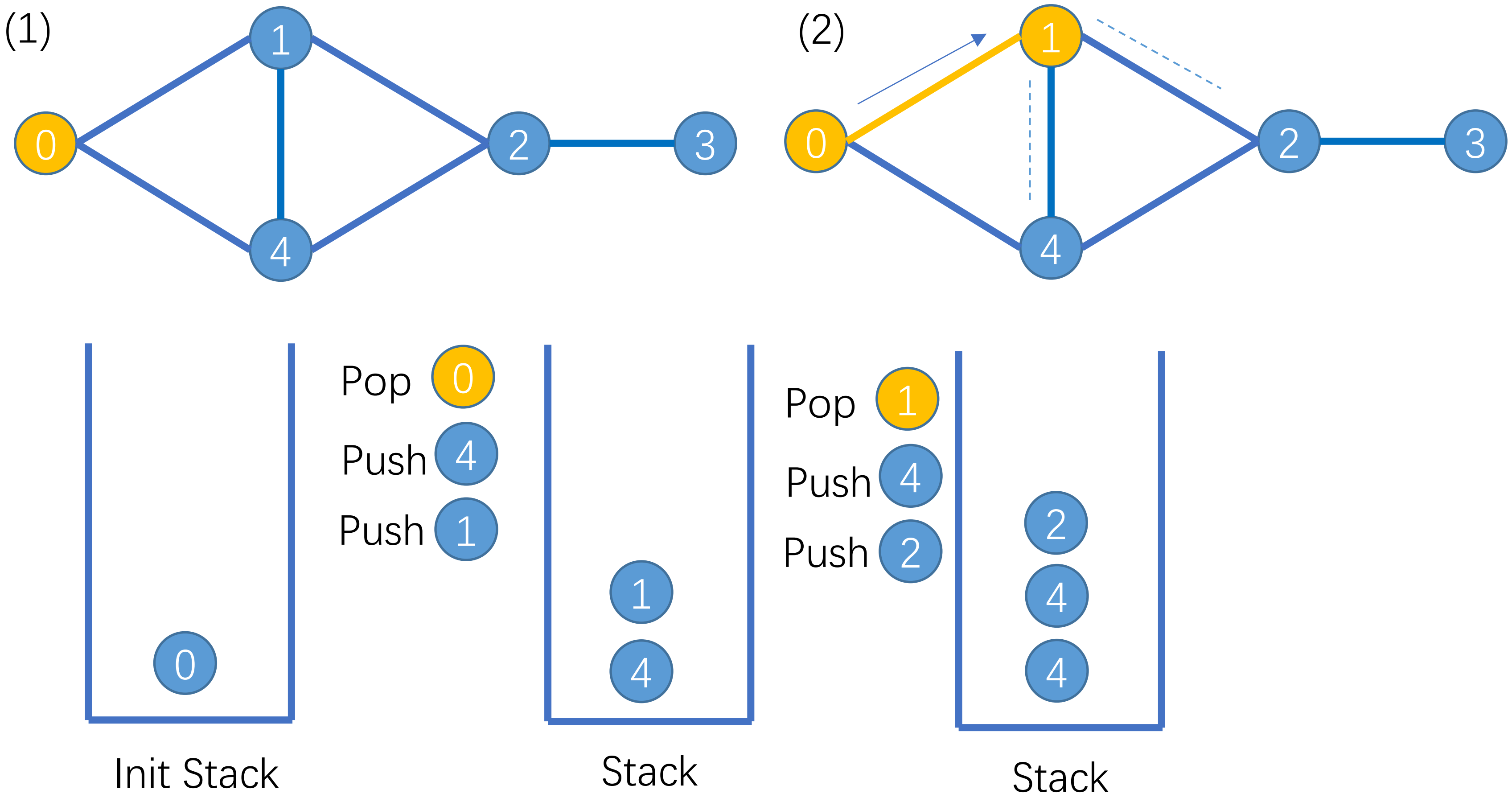


Stack

Stack matches the process of DFS
Push: Store neighbors of current node
Pop: Choose one neighbor of last node
Last in First out: Expand nodes at shallower level

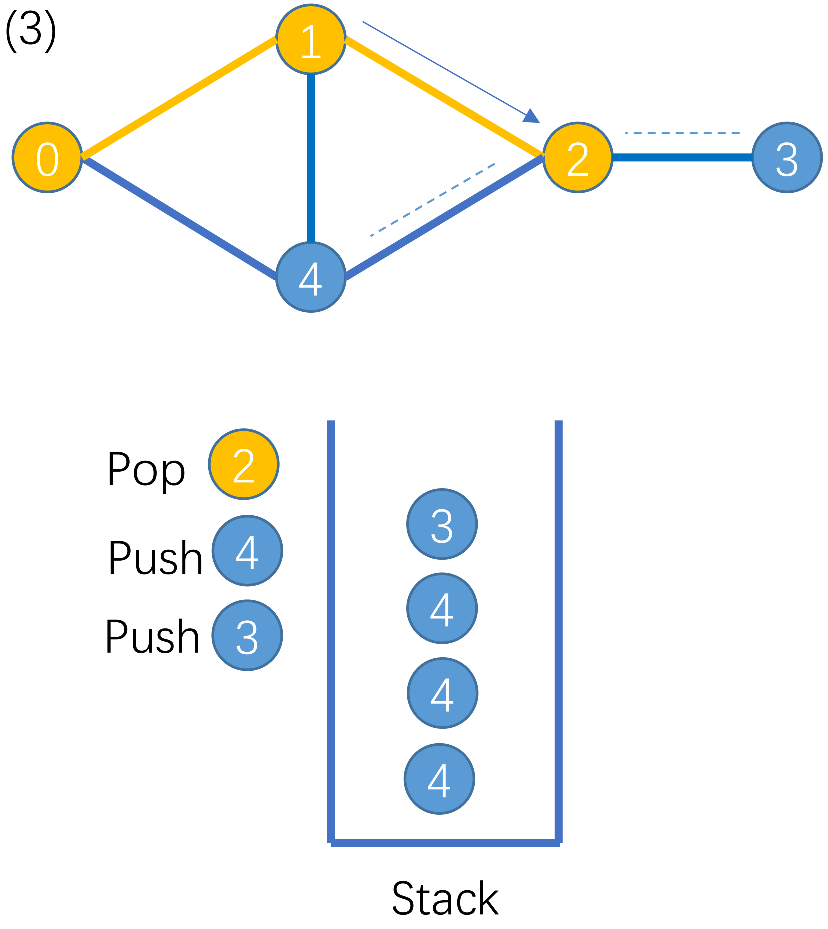
Iterative algorithm for DFS

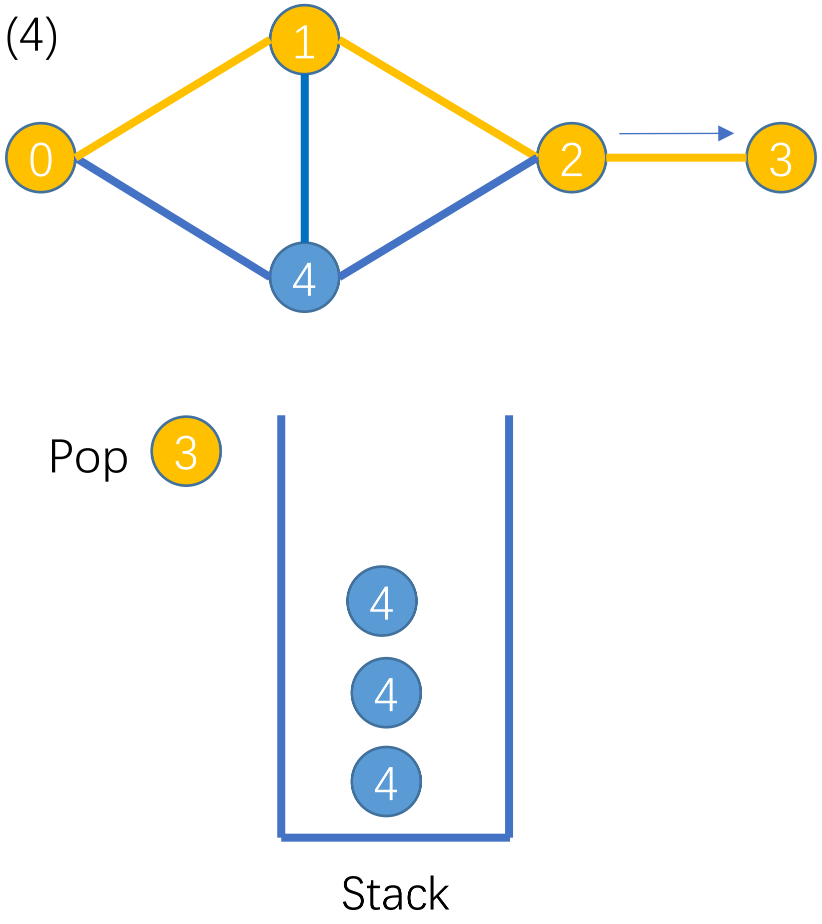
```
visited_list = [ ]
Def DFS_iterative(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    stack.push(v)
```



Iterative algorithm for DFS

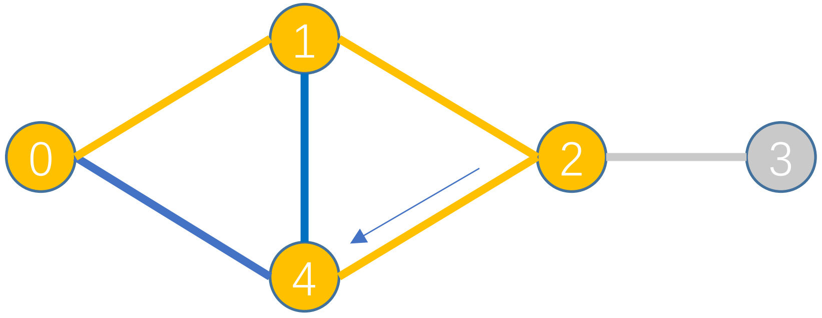
```
visited_list = [ ]
Def DFS_iterative(node):
    **stack** = **stack**.push(node)
    while stack is not empty:
        node = **stack**.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    **stack**.push(v)
```



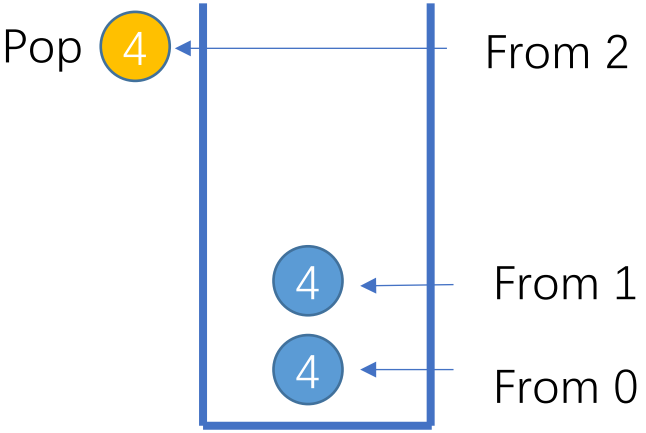


Iterative algorithm for DFS

```
visited_list = [ ]
Def DFS_iterative(node):
        stack = stack.push(node)
        while stack is not empty:
                node = stack.pop()
                if node not in visited_list:
                      visit(node)
                      visited_list.append(node)
                      For v in Neighbors(node):
                            if v not in visited_list:
                                  stack.push(v)
```



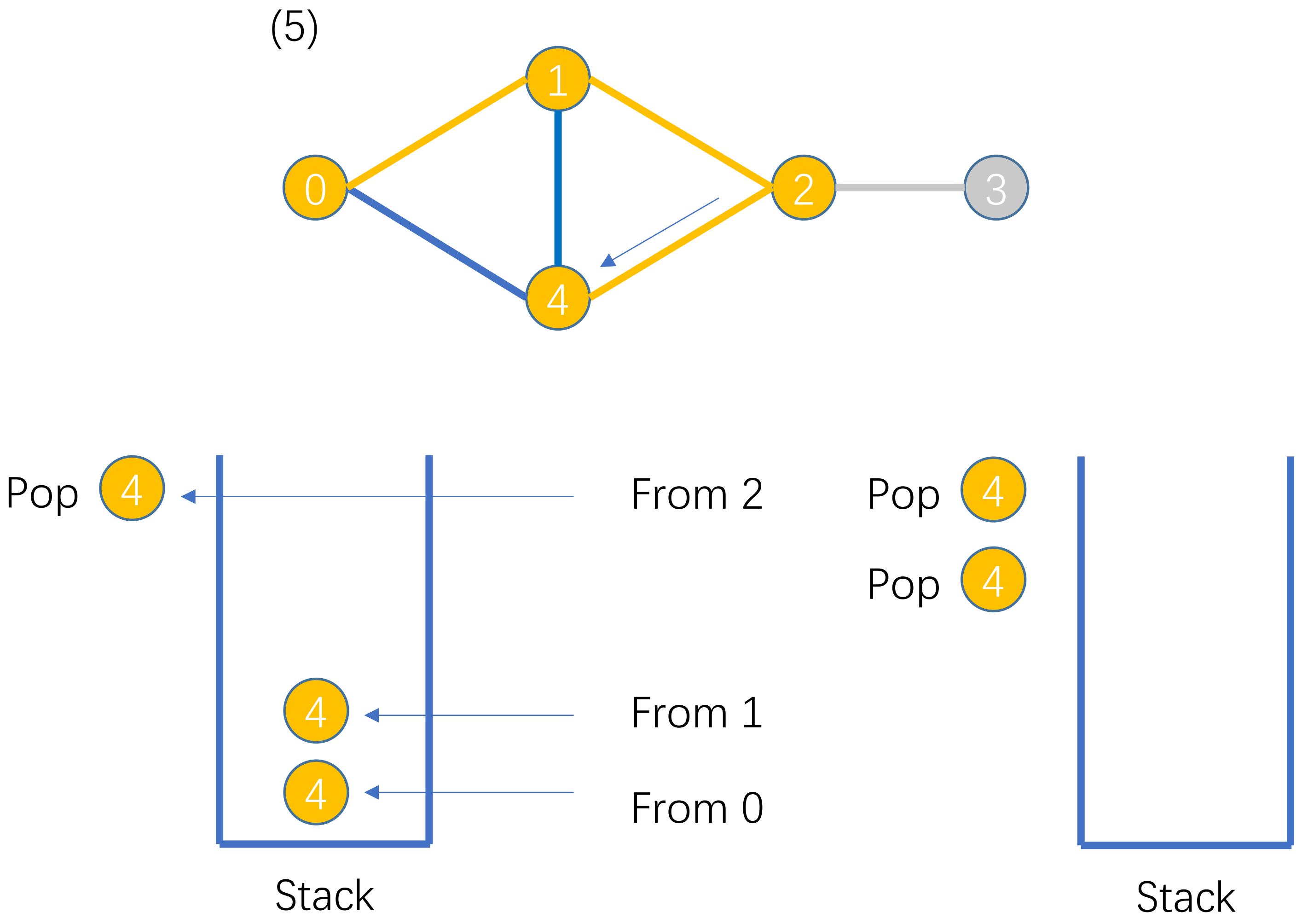
Edge from last parent



Stack

Iterative algorithm for DFS

```
visited_list = [ ]
Def DFS_iterative(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    stack.push(v)
```

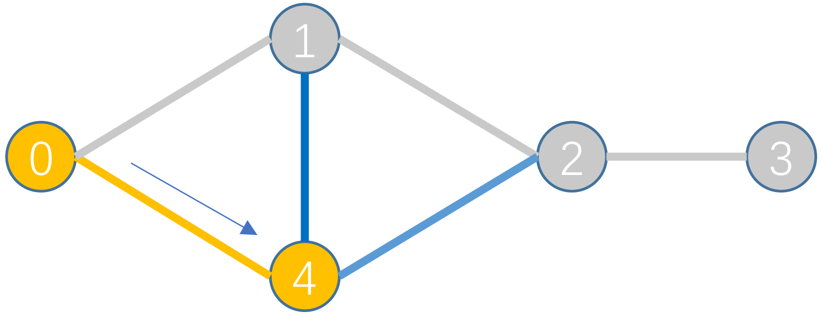


DFS : Path to Goal Node

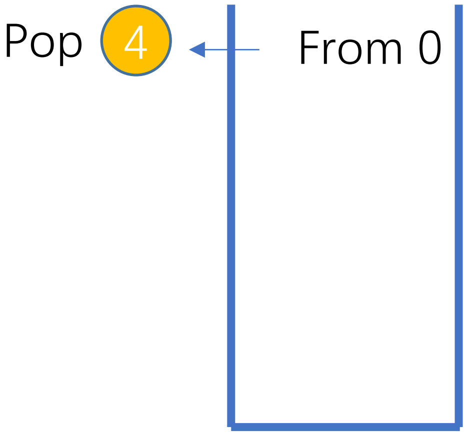
```
visited_list = [ ]
Def DFS_iterative(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    stack.push(v)
```

→

```
visited_list = [ ]
Def DFS_iterative2(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()

        visit(node)
        For v in Neighbors(node):
            if v not in visited_list:
                visited_list.append(v)
                stack.push(v)
```



Edge from first parent



Same node will not be push twice -> Node pop from stack will not repeat

When Node 4 is our goal, this method can produce shorter path!

Recursive or Iterative

Goal: Traverse the graph
Both run: O(V+E)

```
visited_list = [ ]
Def DFS_recursive(node):
    visit(node)
    visited_list.append(node)
    For v in Neighbors(node):
        if v not in visited_list:
            DFS_recursive(v)
```

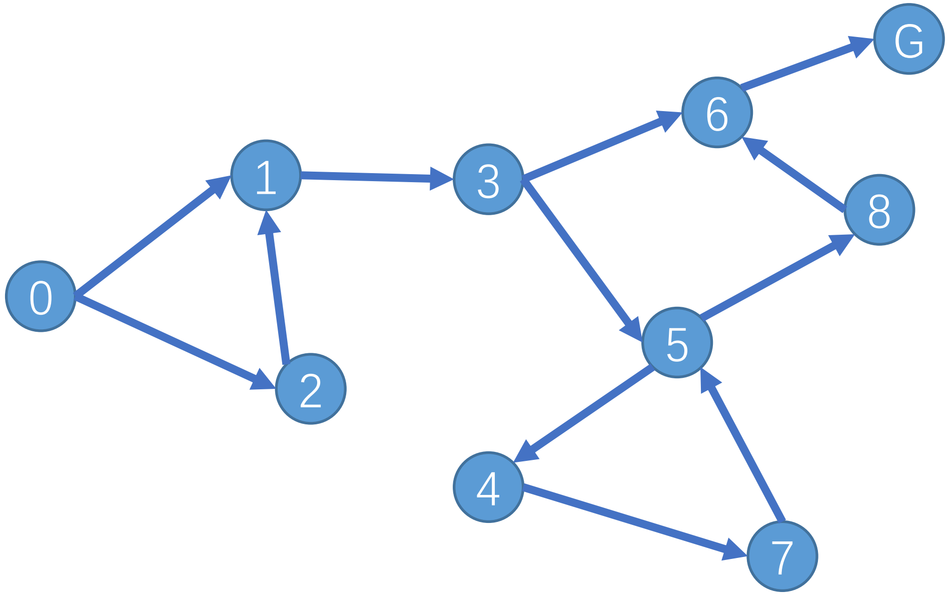
Cleaner and easier to read

```
visited_list = [ ]
Def DFS_iterative(node):
    stack = stack.push(node)
    while stack is not empty:
        node = stack.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    stack.push(v)
```

More generalizable

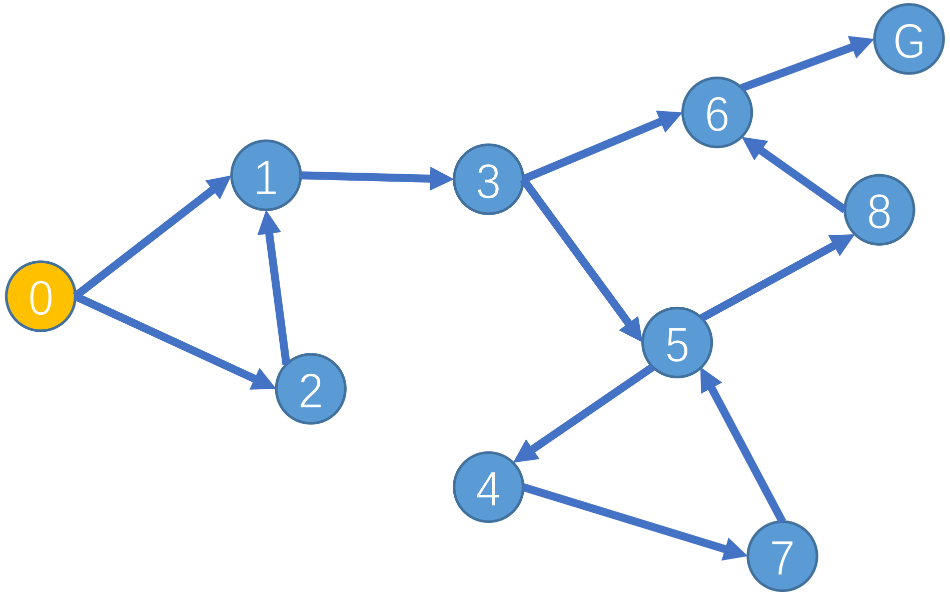
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



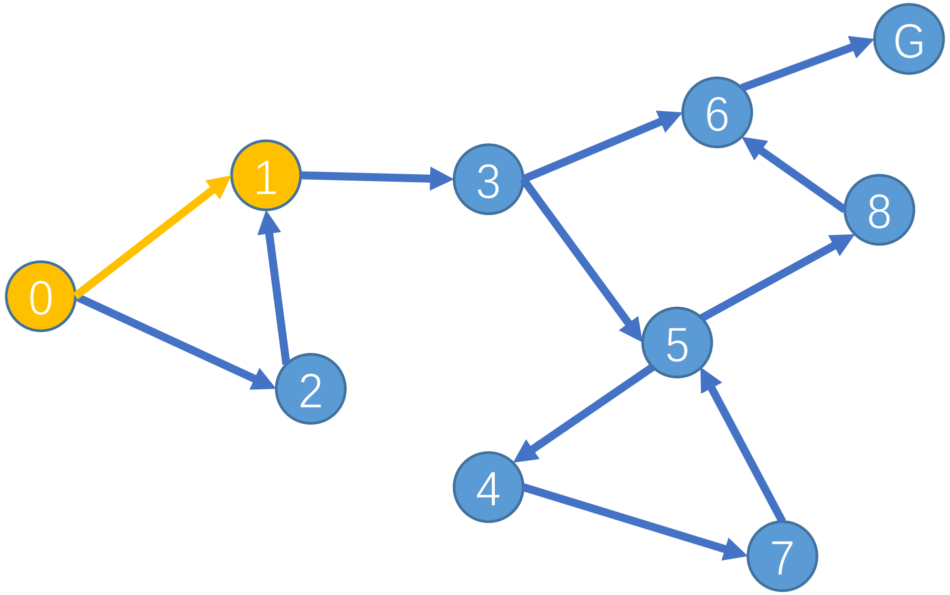
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



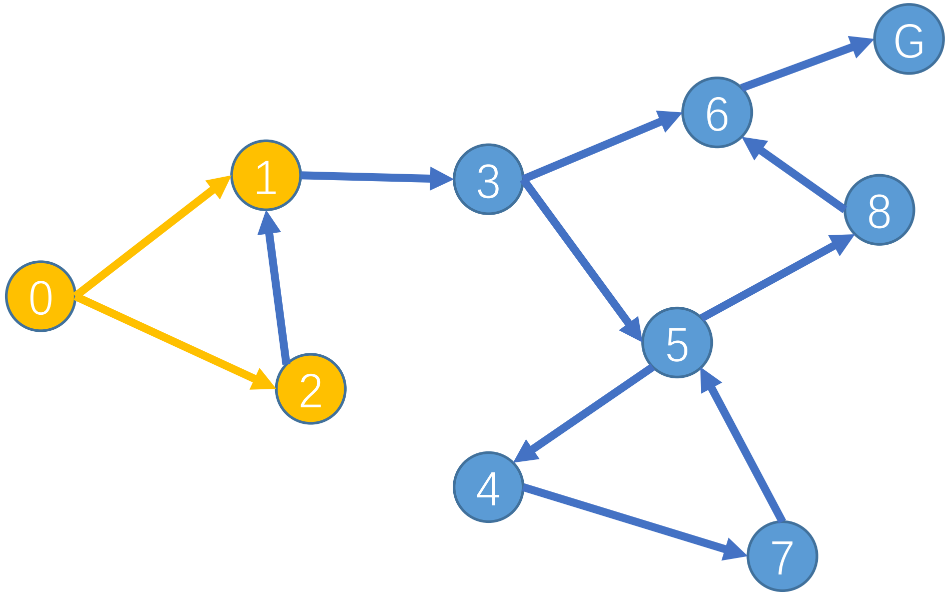
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



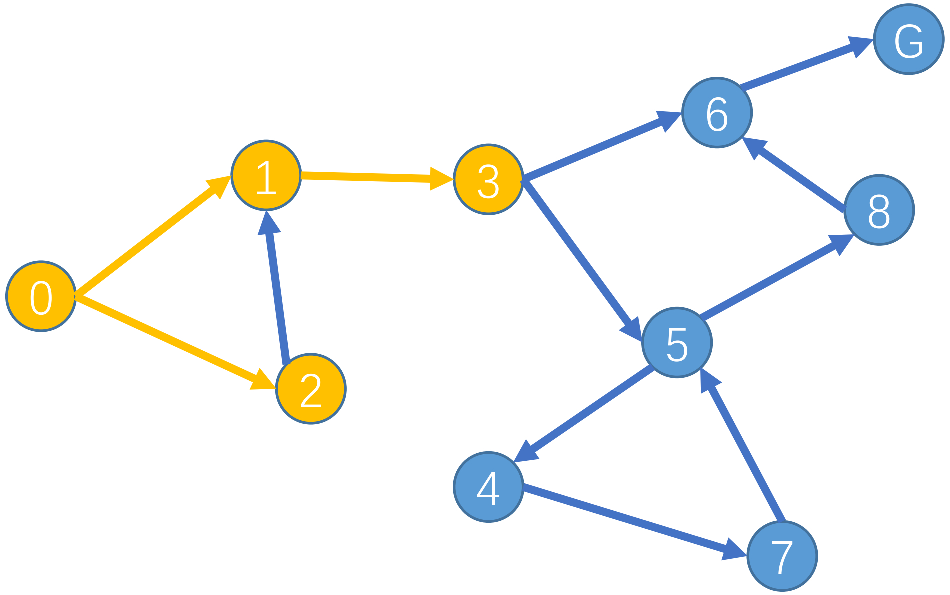
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



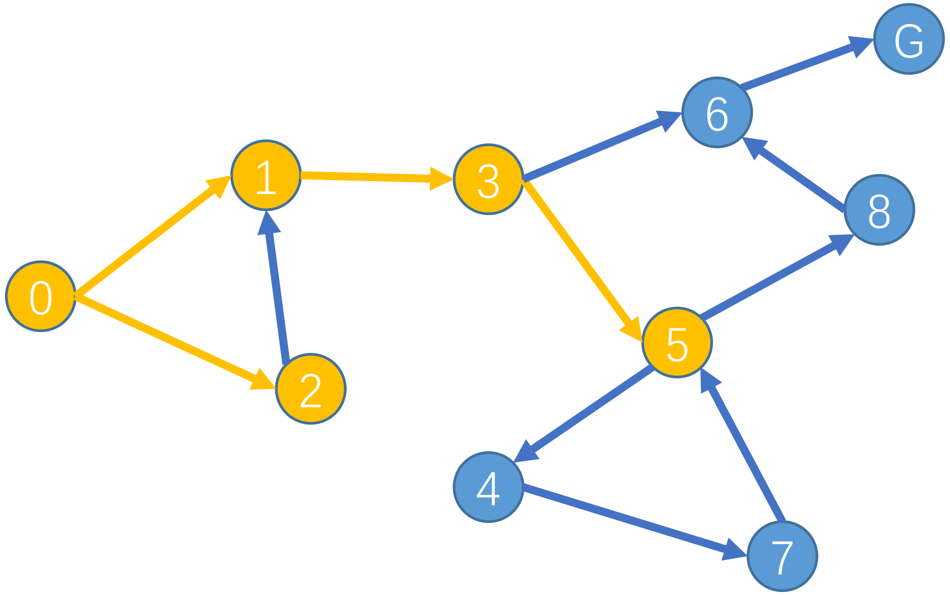
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



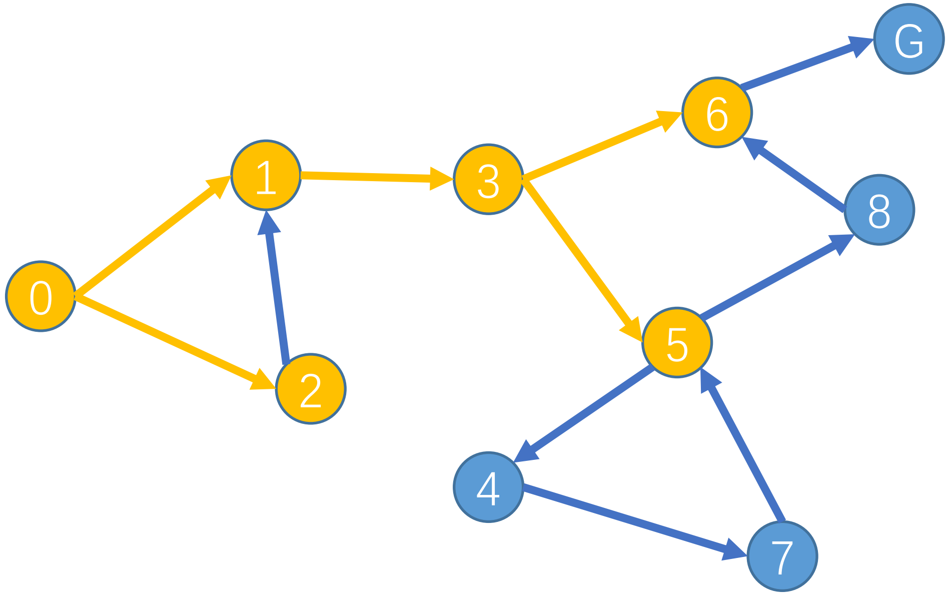
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



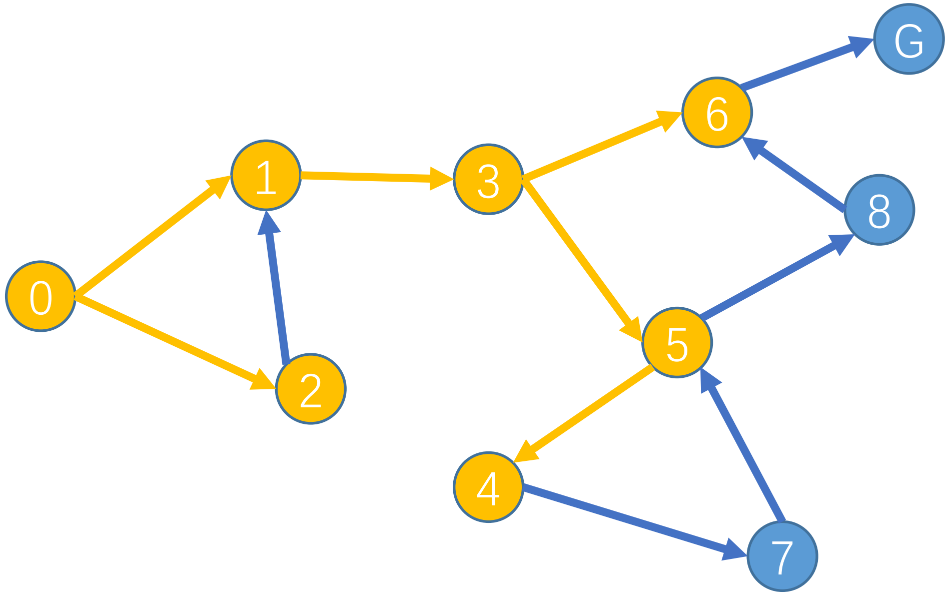
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



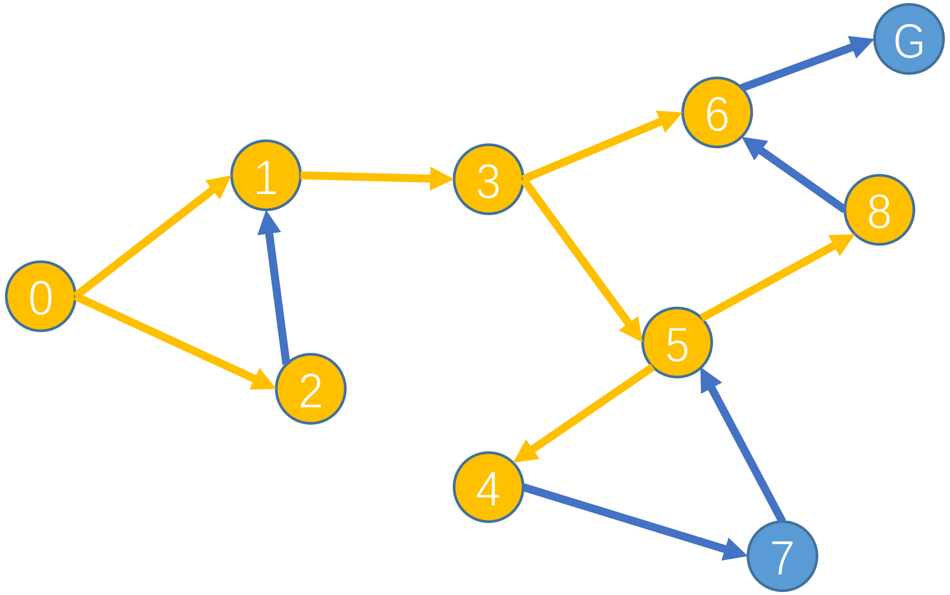
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



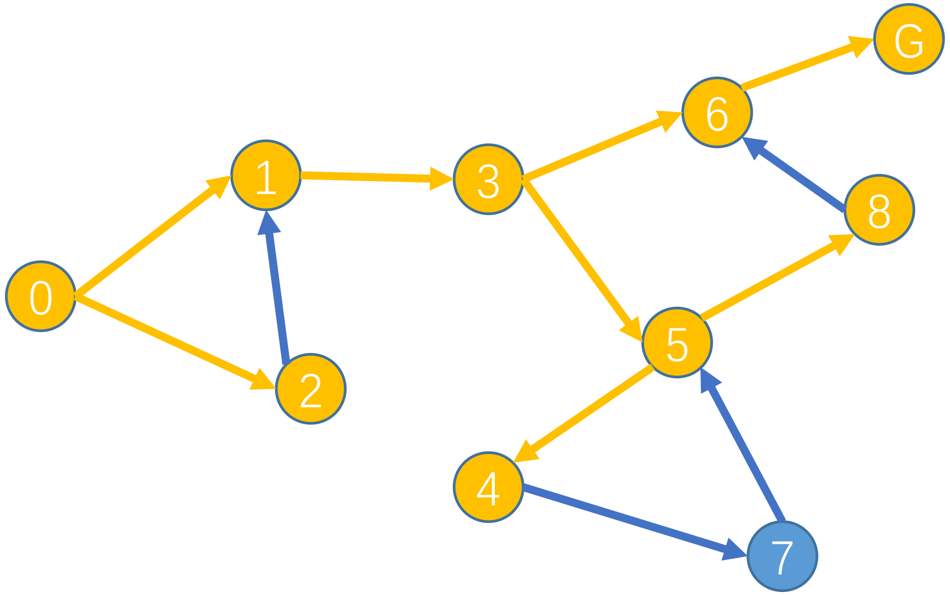
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



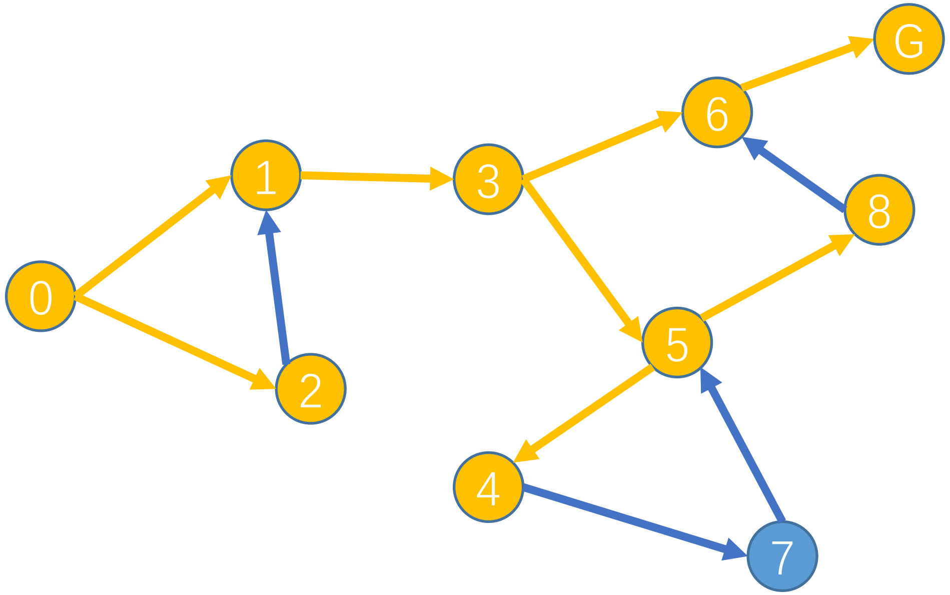
Breadth-first search

- The root node is expanded first, then all the nodes generated by the root node are expanded next, and then their successors, and so on
- All the nodes at depth d in the search tree are expanded before the nodes at depth d + 1



Breadth-first search

- Not naturally recursive
- First found first visit
    - Visit nodes at depth d -> find nodes at d+1 -> visit nodes at d+1



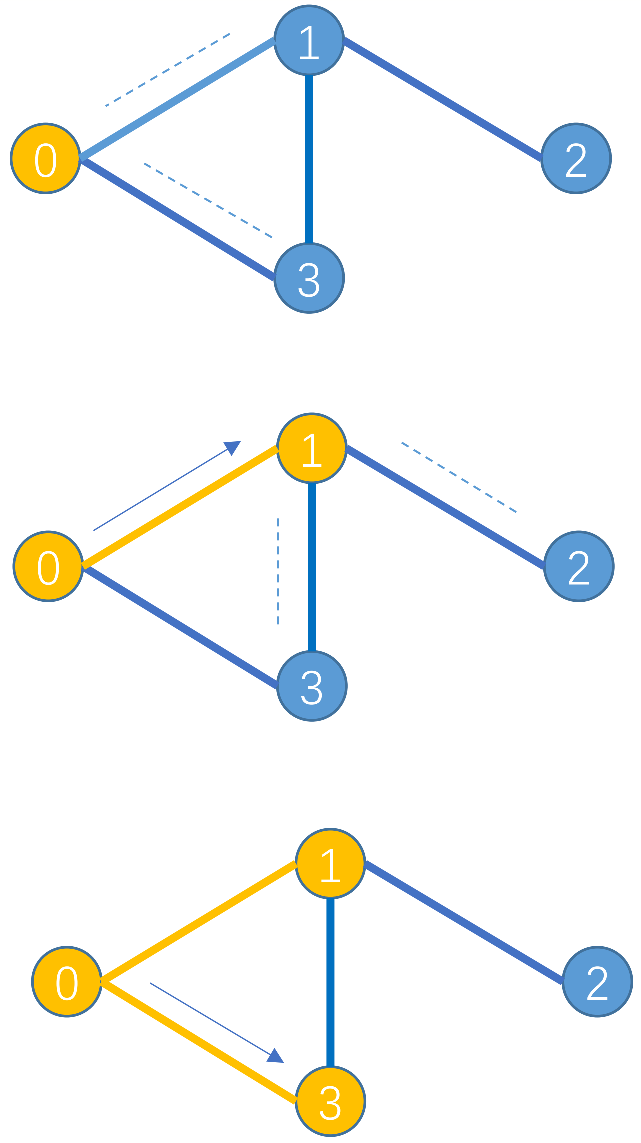
Iterative algorithm for BFS

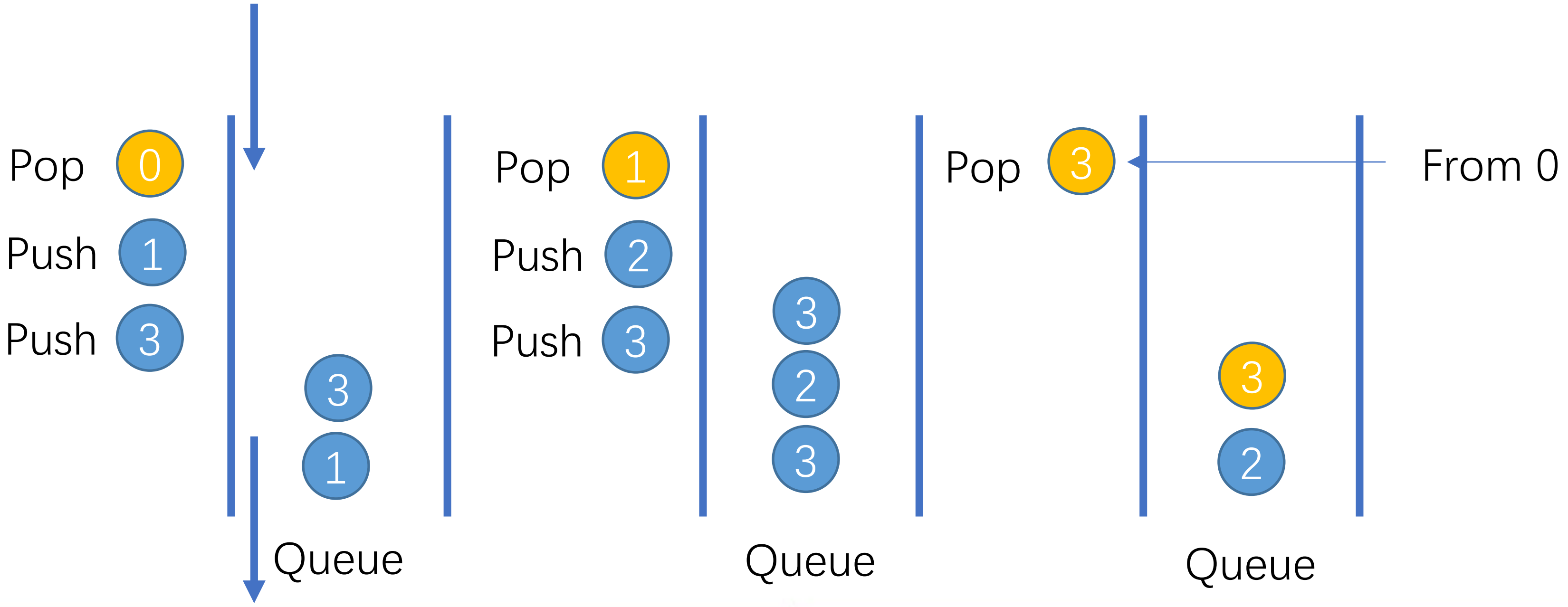
Queue matches the process of BFS

Push: Store neighbors of current node

Pop: Choose one neighbor earliest found

First in First out: Expand nodes at shallowest level



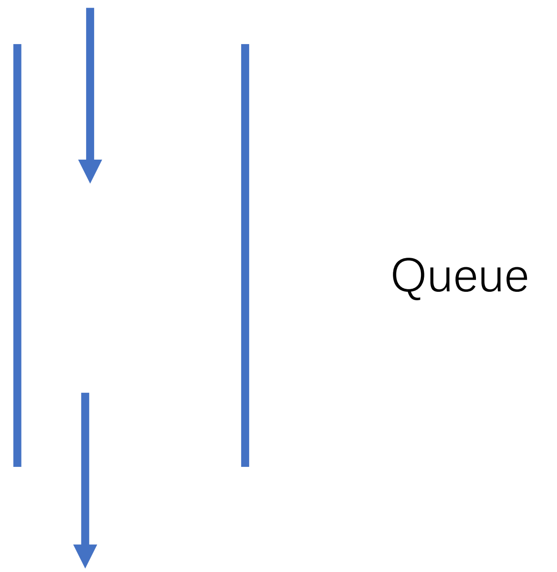


Iterative algorithm for BFS

Use queue to keep track of nodes

Goal: Traverse the graph

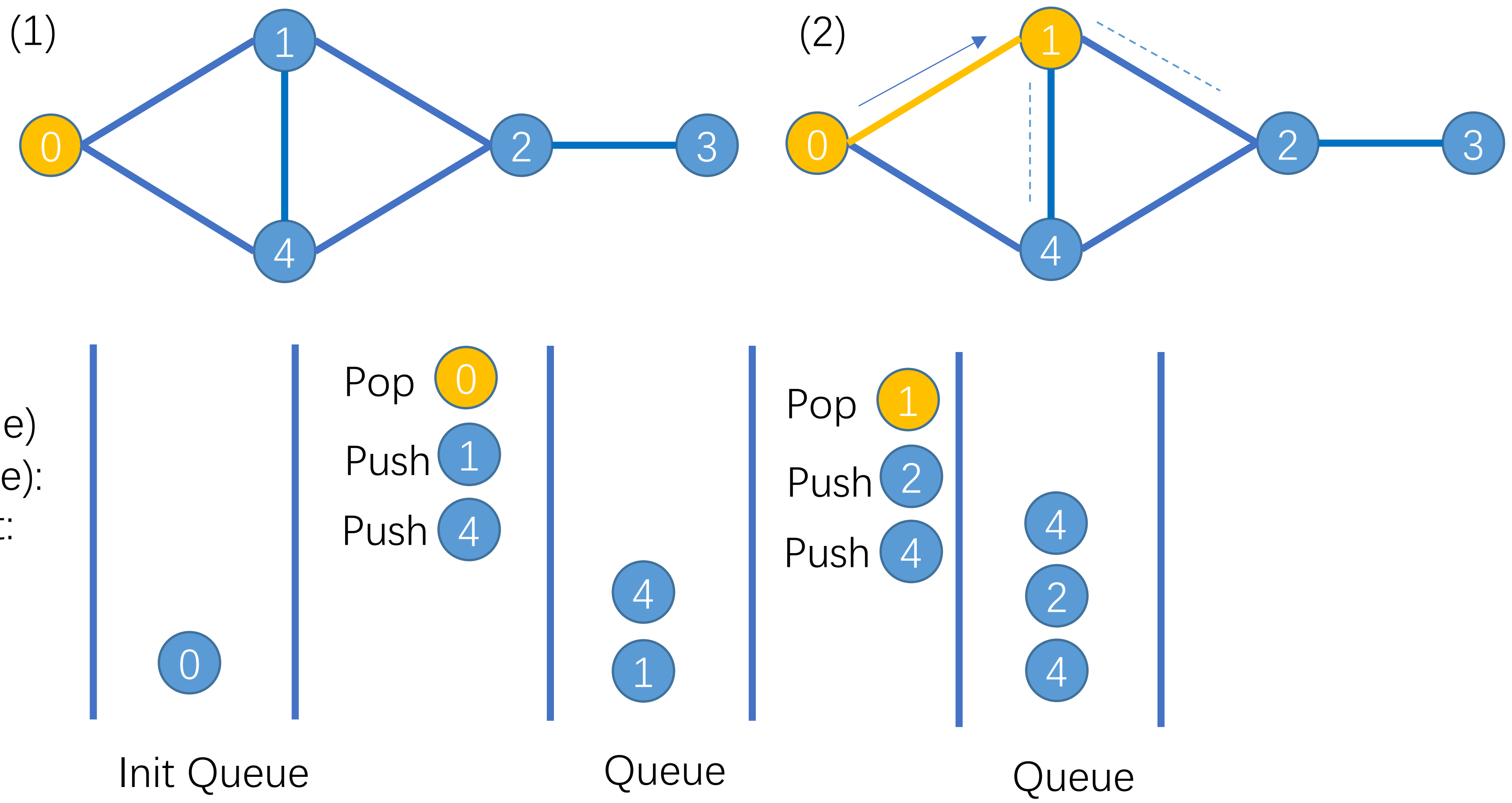
```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    queue.push(v)
```



Queue matches the process of BFS
Push: Store neighbors of current node
Pop: Choose one neighbor earliest found
First in First out: Expand nodes at shallower level

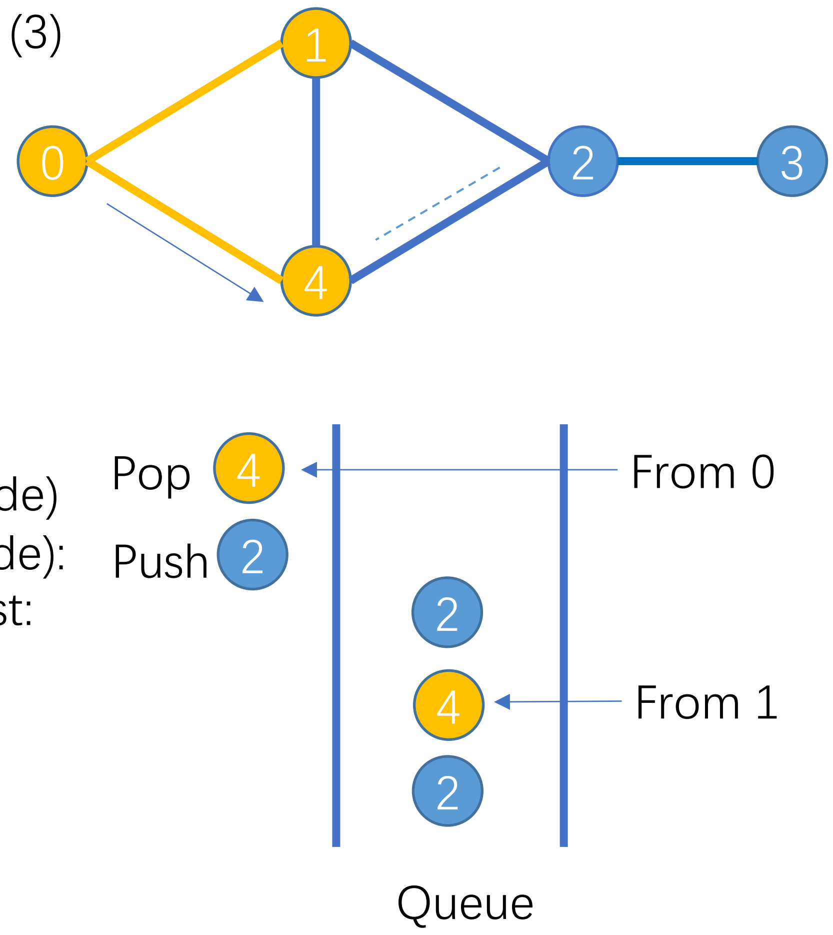
Iterative algorithm for BFS

```
visited_list = [ ]
Def BFS_iterative(node):
    **queue** = **queue**.push(node)
    while **queue** is not empty:
        node = **queue**.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    **queue**.push(v)
```



Iterative algorithm for BFS

```
visited_list = [ ]
Def BFS_iterative(node):
    **queue** = **queue**.push(node)
    while **queue** is not empty:
        node = **queue**.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    **queue**.push(v)
```


(3)

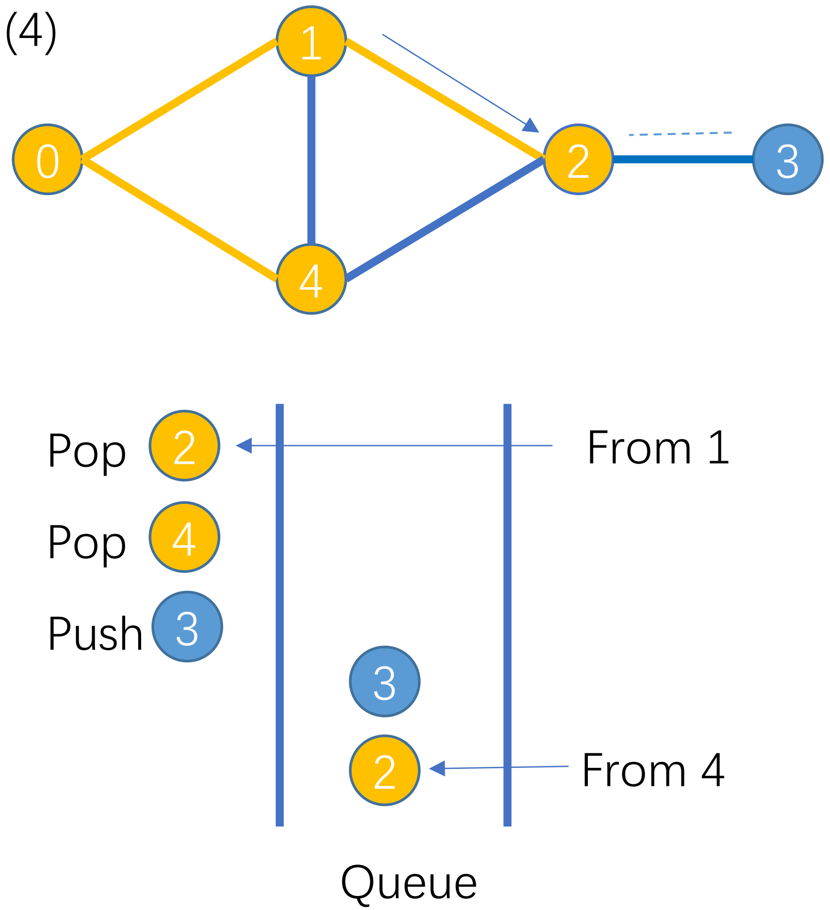
Pop 4

Push 2

From 0

From 1

Queue


(4)

Pop 2

Pop 4

Push 3

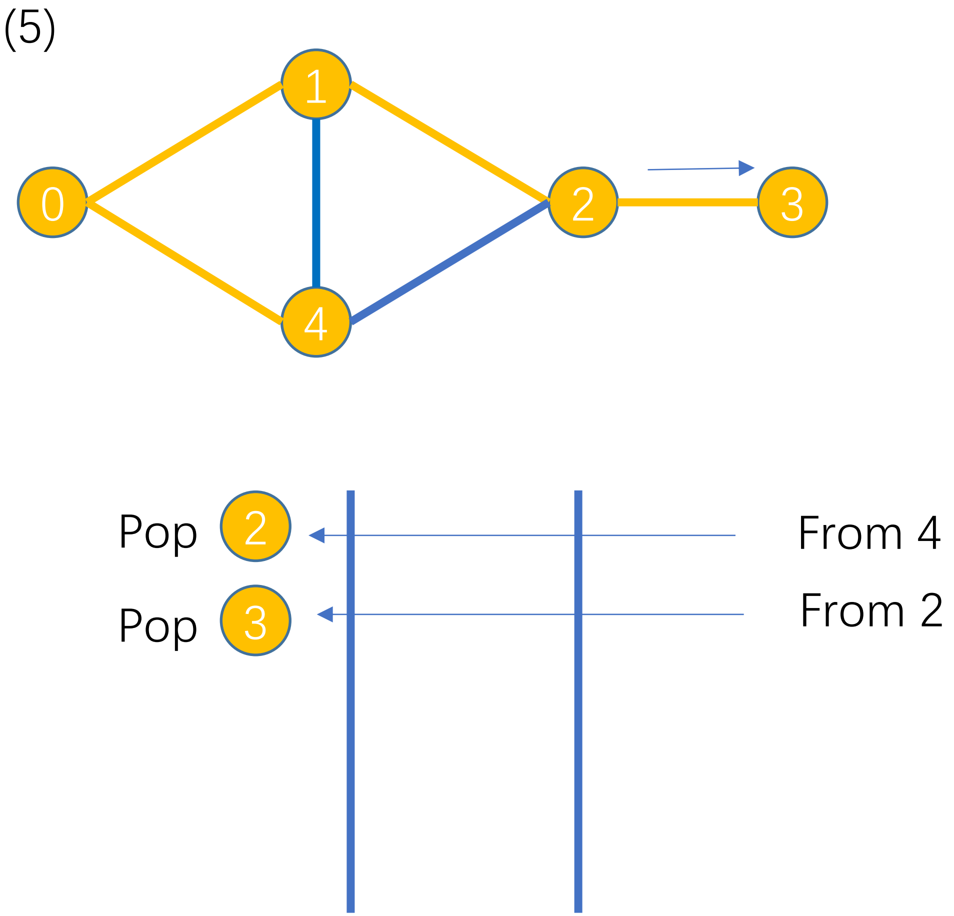
From 1

From 4

Queue

Iterative algorithm for BFS

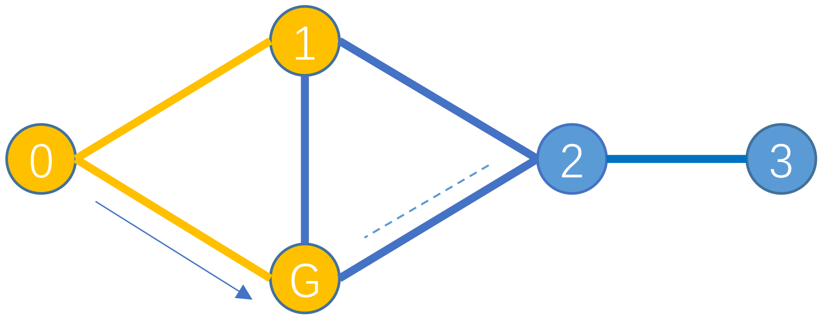
```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    queue.push(v)
```

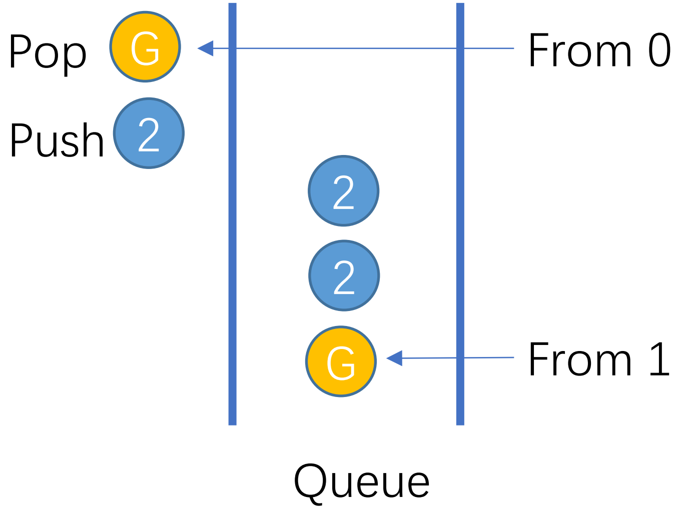


BFS : Path to Goal Node

```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    queue.push(v)
```

```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:
        node = queue.pop()
        visit(node)

        For v in Neighbors(node):
            if v not in visited_list:
                visited_list.append(node)
                queue.push(v)
```

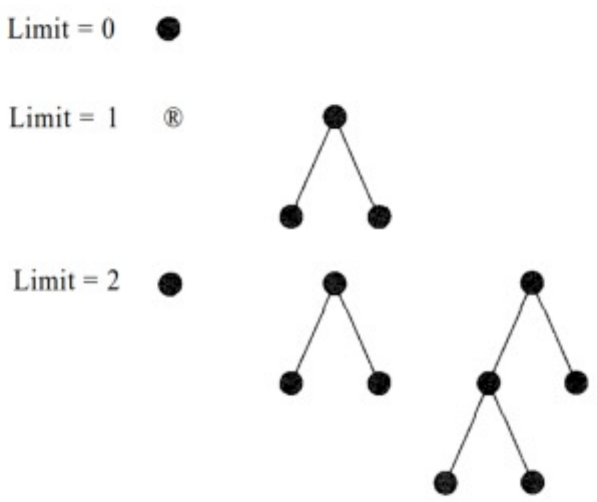


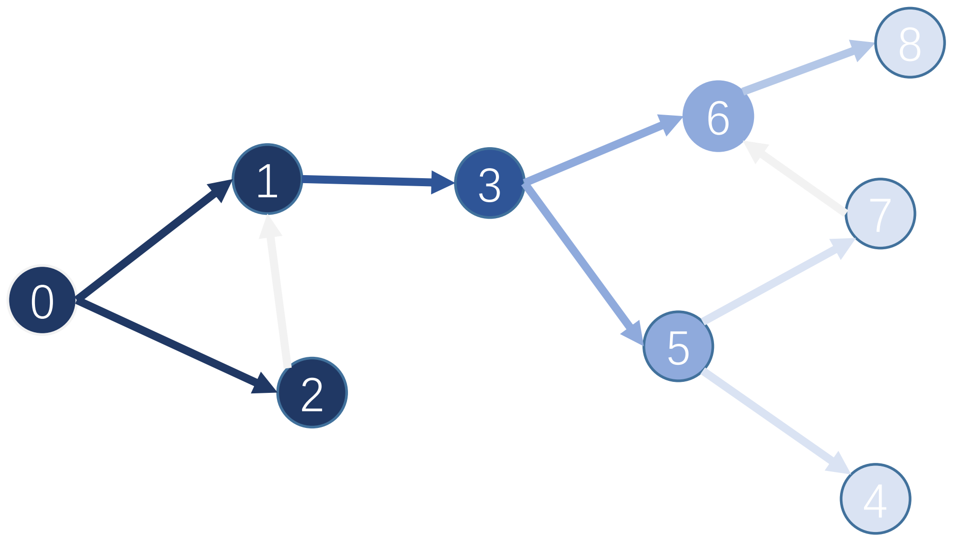


First found first visit: They are the same

Iterative Deepening Search

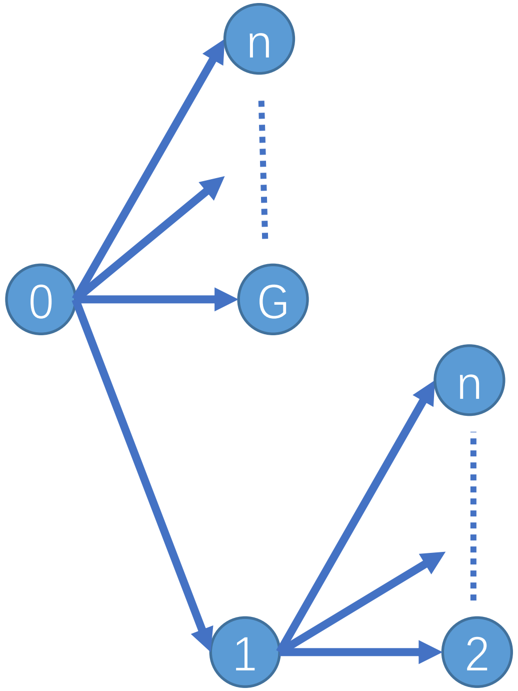
- Iterative deepening search is a strategy that sidesteps the issue of choosing the best depth limit by trying all possible depth limits: first depth 0, then depth 1, then depth 2, and so on.
- Do DFS with limited depth 0, depth 1, depth 2





Iterative Deepening Search

- Why iterative deepening
  - Suppose **n** is a very large number
  - To find goal G
- BFS uses too much space
  - To find the goal, BFS needs to enqueue n nodes
- DFS takes too much time
  - DFS spends excessively long time on node 1



BFS:

```
For v in Neighbors(node):
    if v not in visited_list:
        queue.push(v)
```

DFS:

```
For v in Neighbors(node):
    if v not in visited_list:
        DFS_recursive(v)
```

Iterative Deepening Search

DFS:

```
visited_list = [ ]
Def DFS_recursive(node):
    visit(node)
    visited_list.append(node)
    For v in Neighbors(node):
        if v not in visited_list:
            DFS_recursive(v)


DFS _recursive(start_node)
```

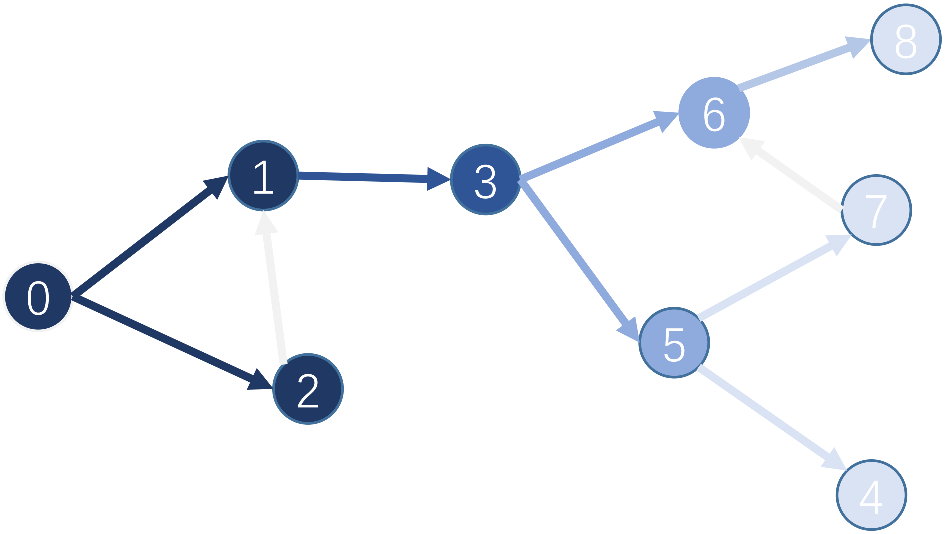
IDDFS:

```
visited_list = [ ]
Def IDDFS(node, depth):
    visit(node)
    visited_list.append(node)
    depth += 1
    if depth > max_depth:
       return
    For v in Neighbors(node):
        if v not in visited_list:
            IDDFS(v, depth)


# Suppose max_depth is a global var
For i in range(M):
    max_depth = i
    IDDFS(start_node, 0)
```

Iterative Deepening Search

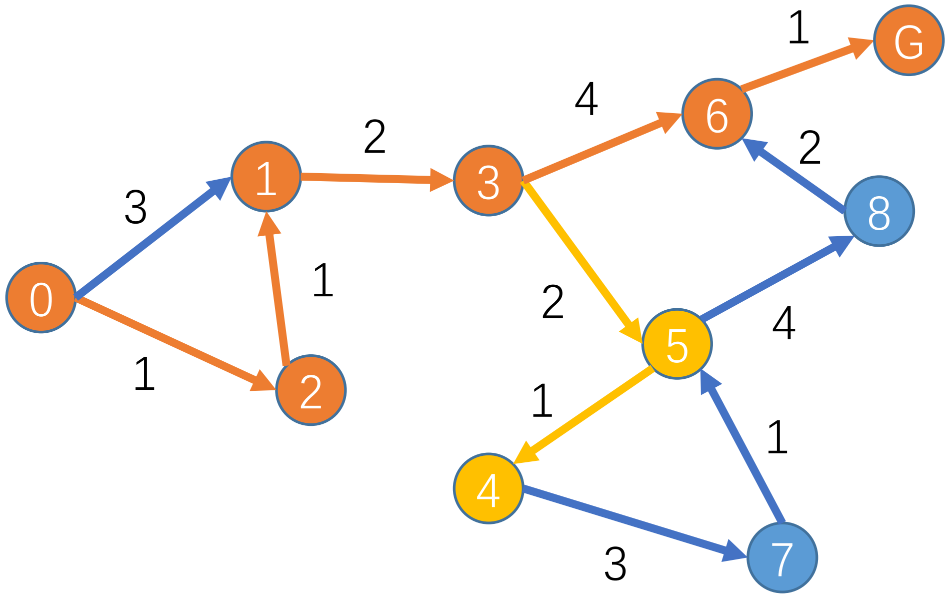
IDDFS:

```
visited_list = [ ]
Def IDDFS(node, depth):
        visit(node)
        visited_list.append(node)
        depth += 1
        if depth > max_depth:
            return
        For v in Neighbors(node):
                if v not in visited_list:
                        IDDFS(v, depth)


 For i in range(5):
        max_depth = i
        IDDFS(start_node, 0)
```



Uniform-Cost Search

- Uniform cost search modifies the breadth-first strategy by always expanding the lowest-cost node on the fringe



Cost:

0	0	5	6
1	2	6	8
2	1	7	10
3	4	8	10
4	7	G	9

Sequence:

0 2 1 3 5 4 6 G

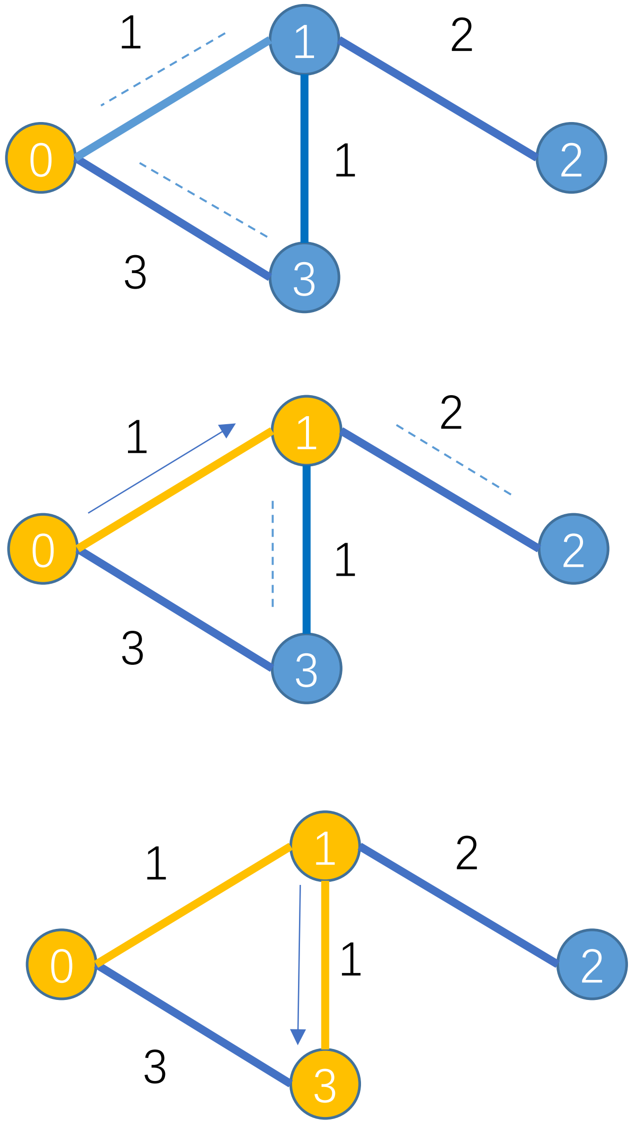
Uniform-Cost Search

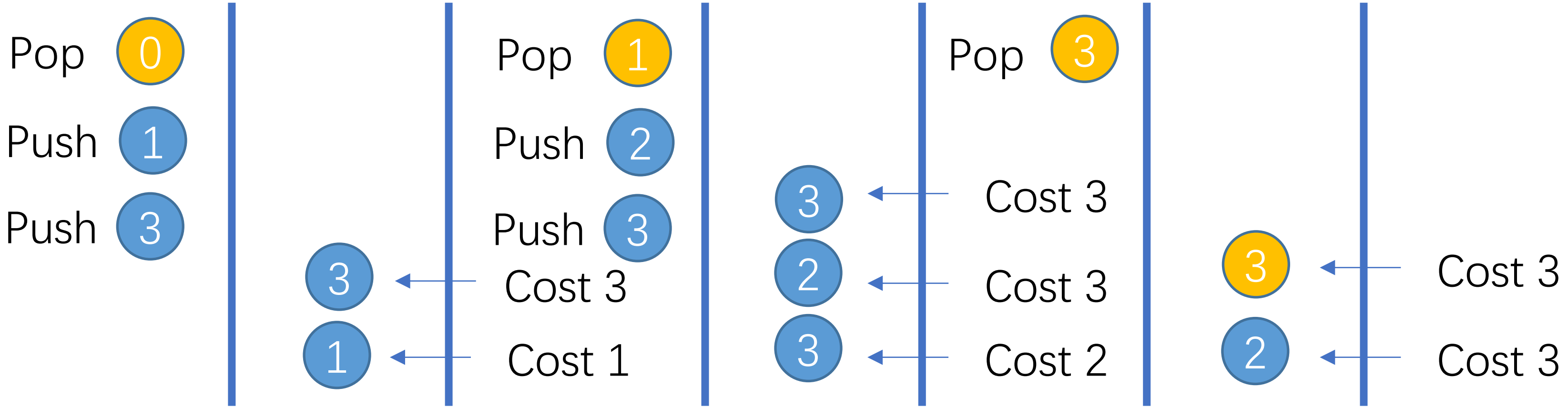
Priority Queue matches the process of UCS

Push: Store neighbors of current node

Pop: Choose nodes with lowest cost

Priority: Sorting nodes with cost




Pop 0

Push 1

Push 3

3 Cost 3

1 Cost 1

Pop 1

Push 2

Push 3

3 Cost 3

2 Cost 3

3 Cost 2

Pop 3

3 Cost 3

2 Cost 3

Priority Queue

Priority Queue

Priority Queue

Uniform-Cost Search

BFS:

```
visited_list = [ ]
Def BFS_iterative(node):
    queue = queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    queue.push(v)
```

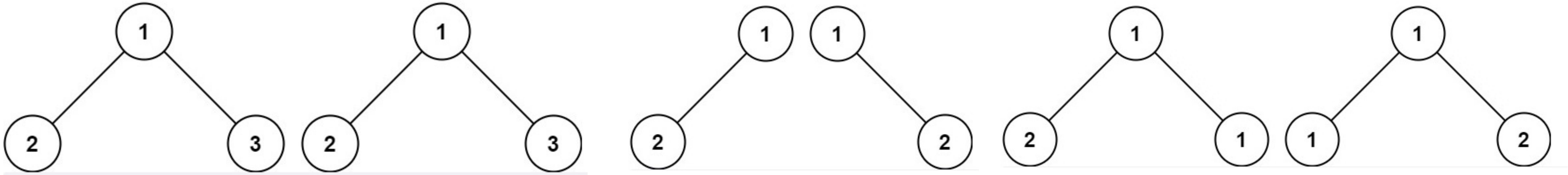
UCS:

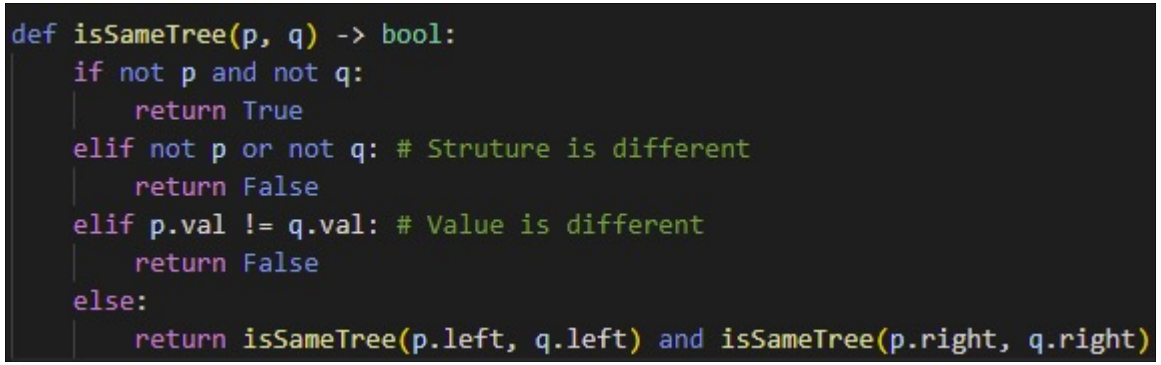
```
visited_list = [ ]
Def UCS(node):
    queue = priority_queue.push(node)
    while queue is not empty:
        node = queue.pop()
        if node not in visited_list:
            visit(node)
            visited_list.append(node)
            For v in Neighbors(node):
                if v not in visited_list:
                    v.cost = node.cost + Cost(node, v)
                    queue.push(v)
                    queue.sort()
```

Simple Question

Given two trees, check if two trees are the same: 1) same structure 2) nodes have the same value

node.val: the value of the node
node.left/node.right: the left/right child of the node

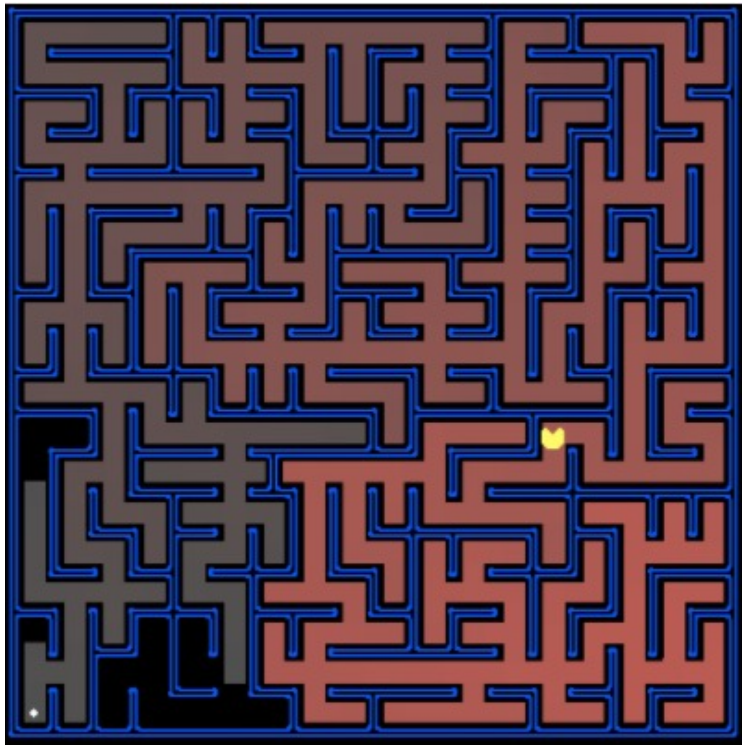




```
def isSameTree(p, q) -> bool:
    if not p and not q:
        return True
    elif not p or not q: # Struture is different
        return False
    elif p.val != q.val: # Value is different
        return False
    else:
        return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

Assignment 4

In this project, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will **build general search algorithms** and apply them to Pacman scenarios. (All in python)

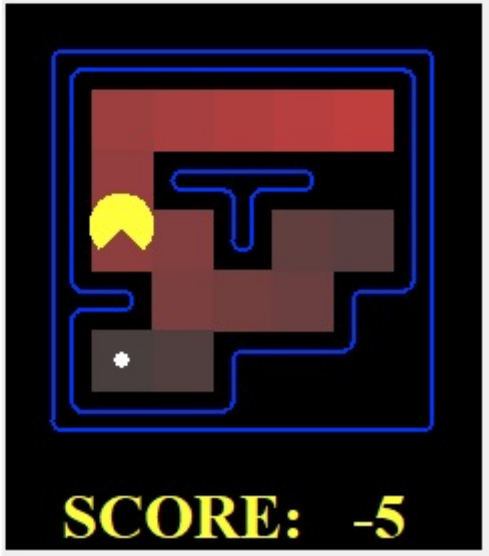


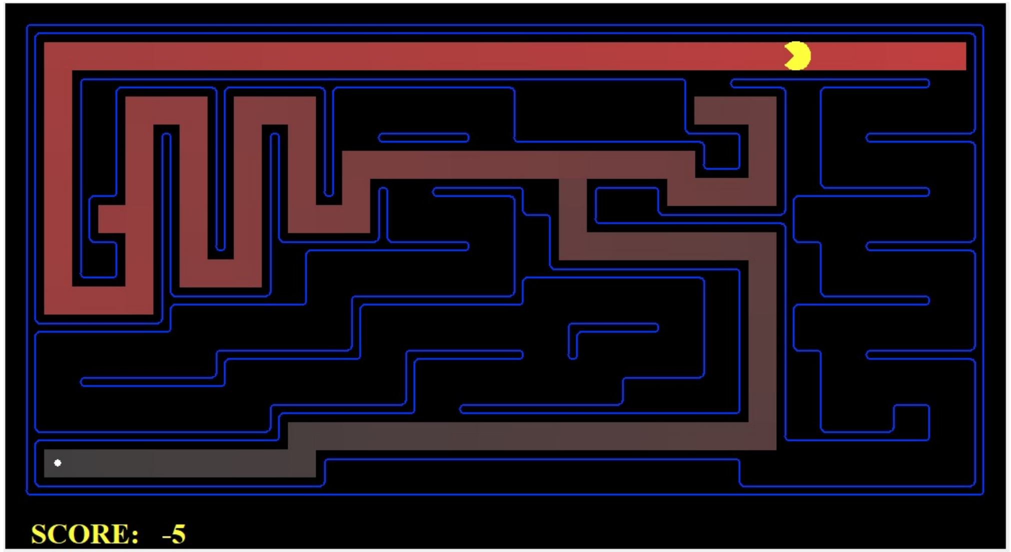
Assignment 4

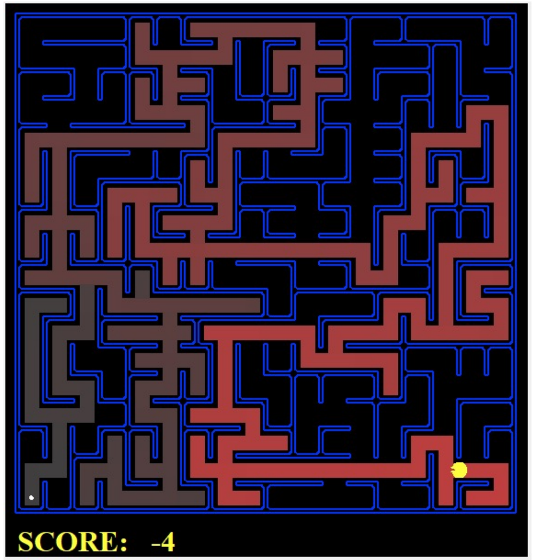
- DFS recursive and iterative
- BFS
- Uniform Cost Search
- A*

Assignment 4

- Your Pacman should pass the maze and find the goal
- You should find a way to store path toward the goal







Thank you!