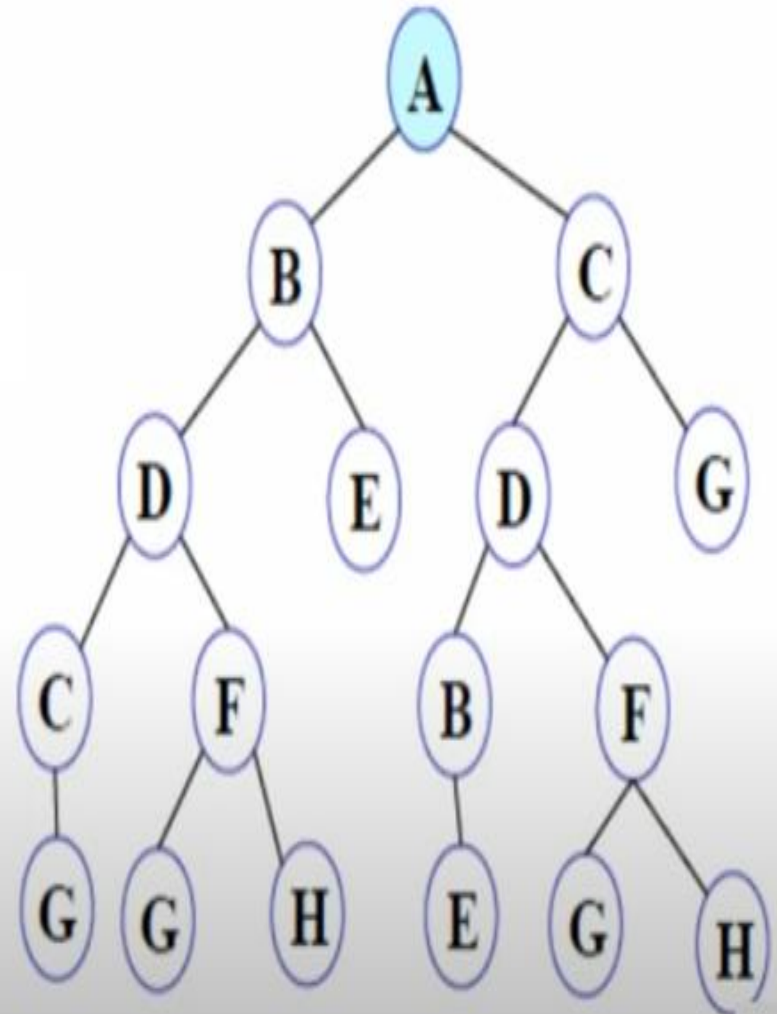
Types of Search Algorithms in Artificial Intelligence

1. **Uninformed search** **(blind search)**

   uses no information about the problem to guide the search and therefore may not be very efficient.

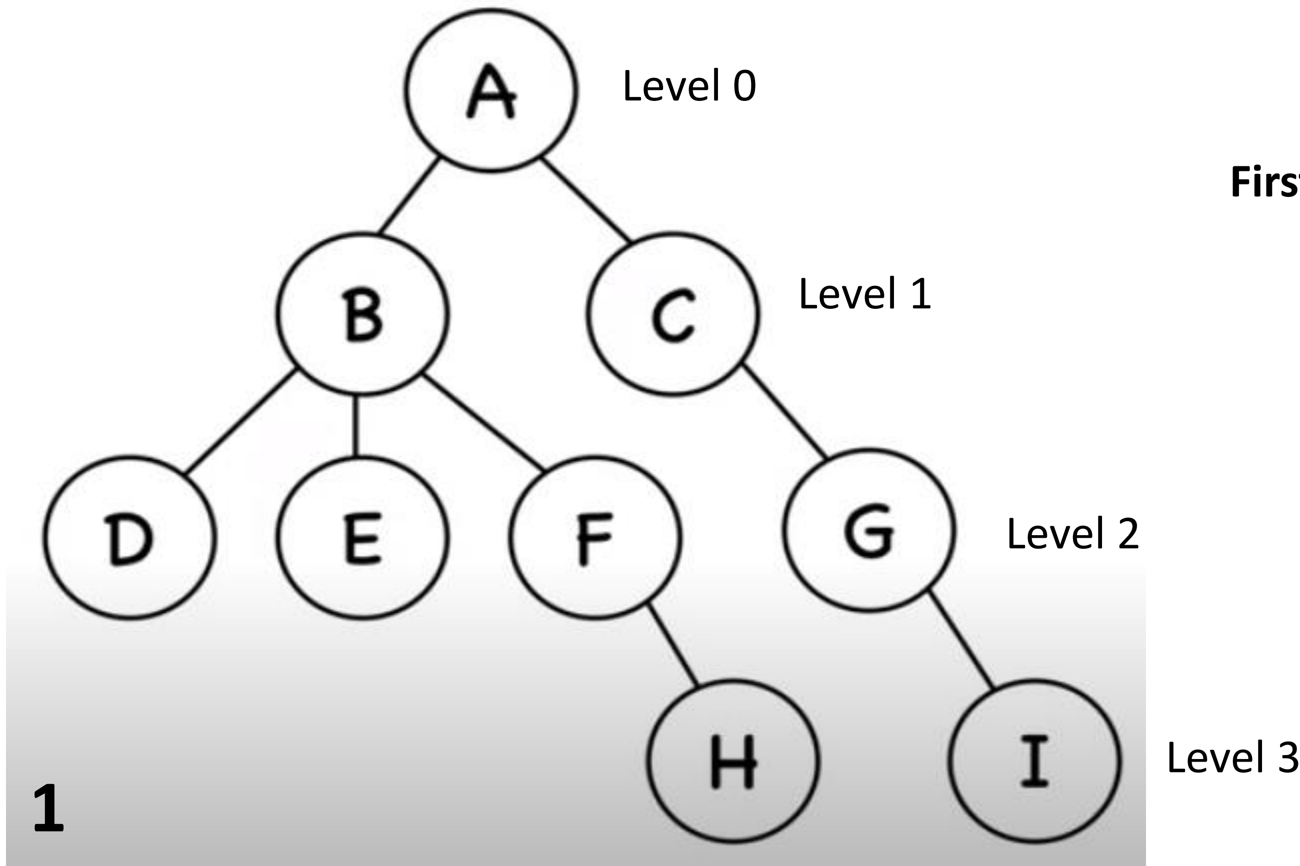


خوارزميات البحث غير المطلعة: **والمعروفة أيضًا باسم خوارزميات البحث الأعمى، هي فئة من خوارزميات البحث التي تستكشف مساحة المشكلة دون استخدام أي معلومات محددة حول المشكلة بخلاف تعريف المشكلة نفسها. بعض الامثلة عن هذا النوع من الخوارزميات:**

1. Breadth-First Search (BFS):

ملاحظة: مبدا العمل هو FIFO

consider the graph below. Find path from node A to node G using BFS.



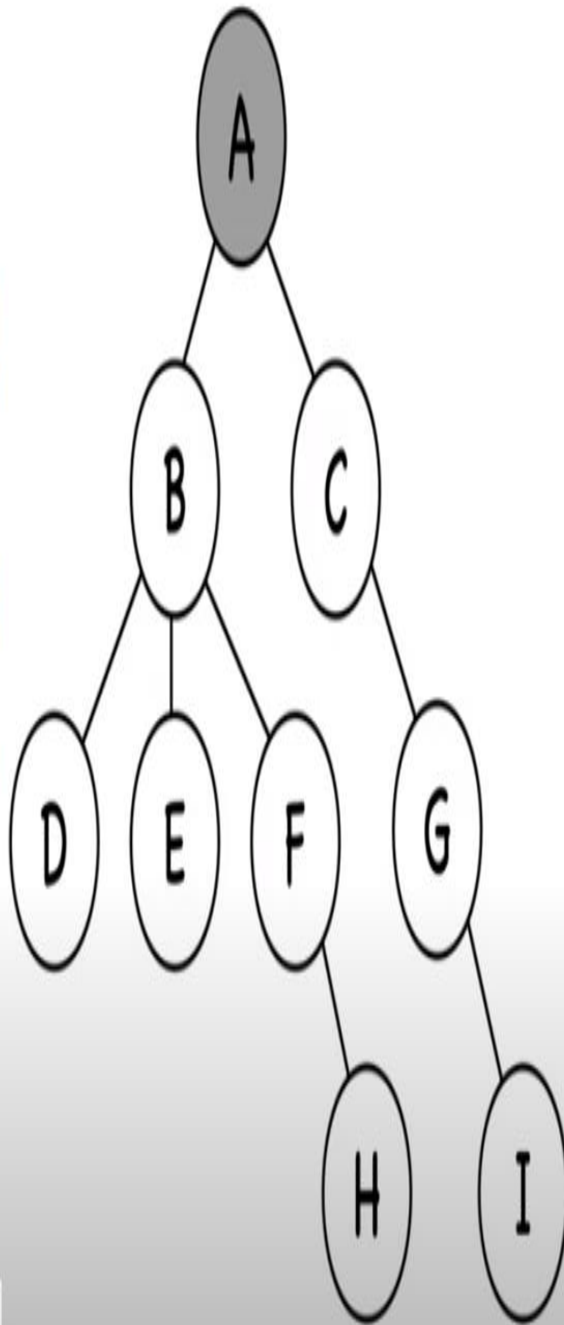
First emplty Queue = [ ]

[A]

نزور نود البداية وهي A ثم نضعها في الكيو ... هل A=G كلا

اذن نستخرج A من الكيو ونضع اطفالها من اليسار الى اليمين

كما في الشكل



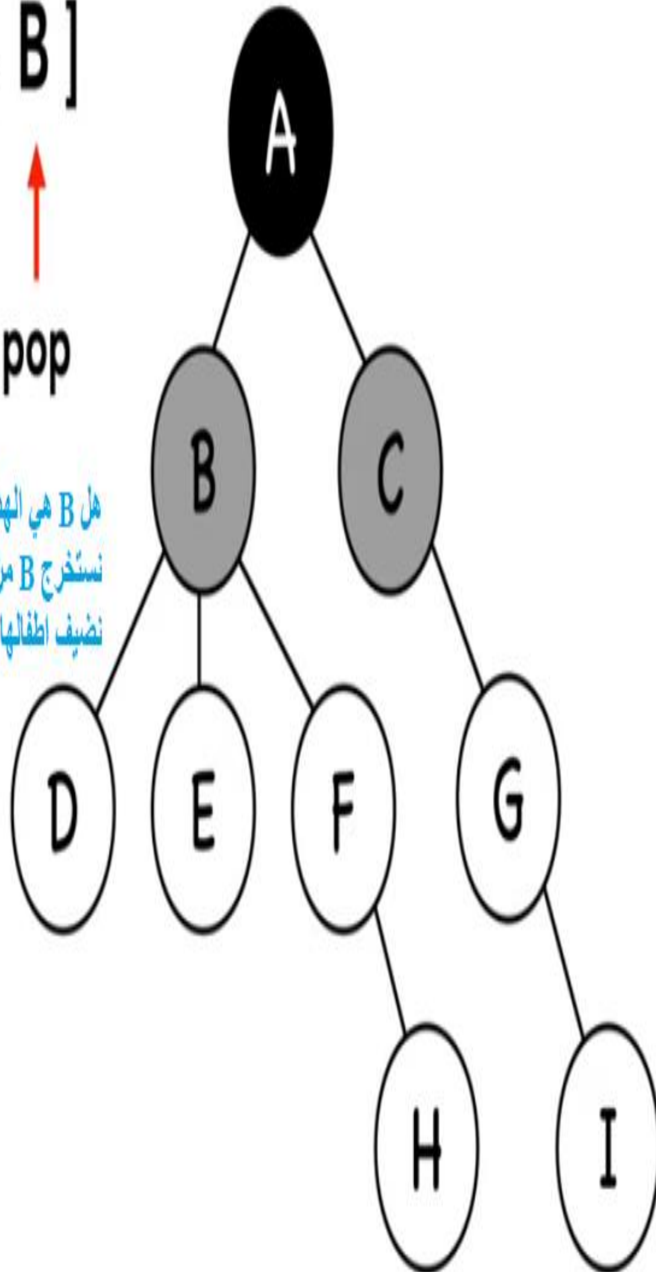
2

Black = Visited

[C, B]

A B C D E F G H I

3



[F, E, D, C]

A
B
C

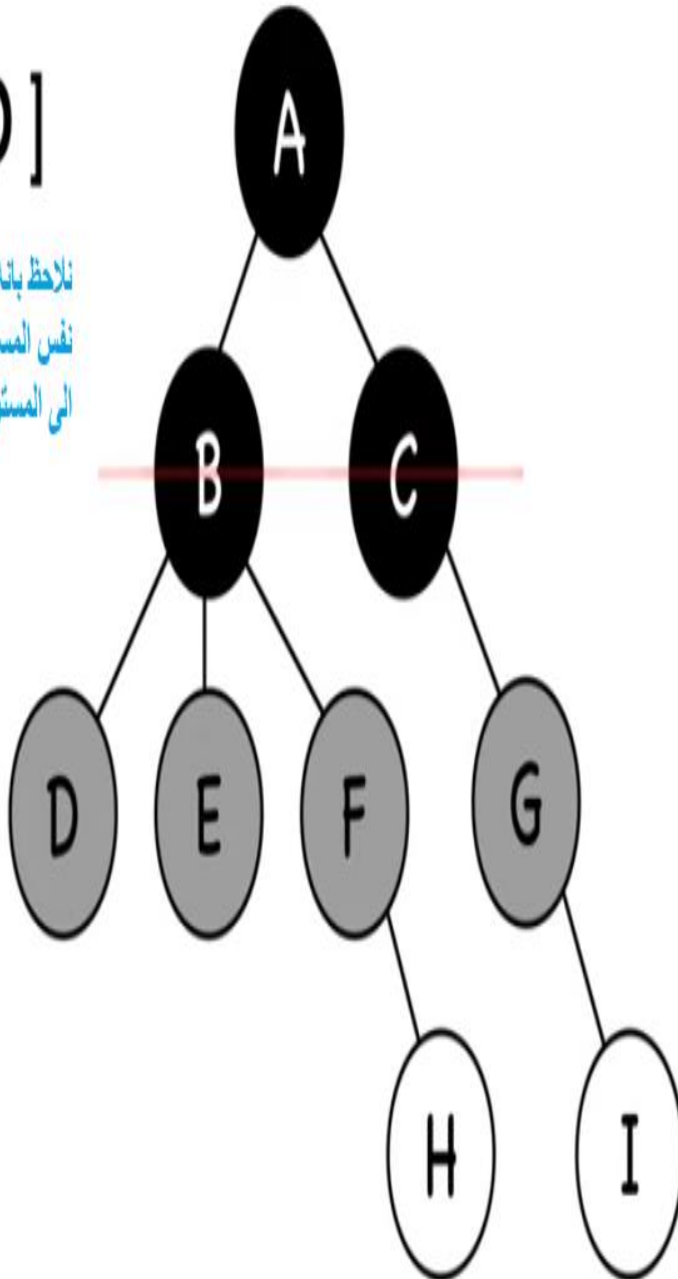
ناخذ النود C هل هي الهدف ؟ كلا
اذن نستخرج C ونأشر عليها بالاسود
نضيف اطفال C الى أخر الكيو من جهة اليسار

D
E
F
G
H
I

5

[G, F, E, D]

نلاحظ بانه تم زيارة ال NODES ضمن نفس المستوى الواحد وبعد اكمالها ننتقل الى المستوى الادنى.

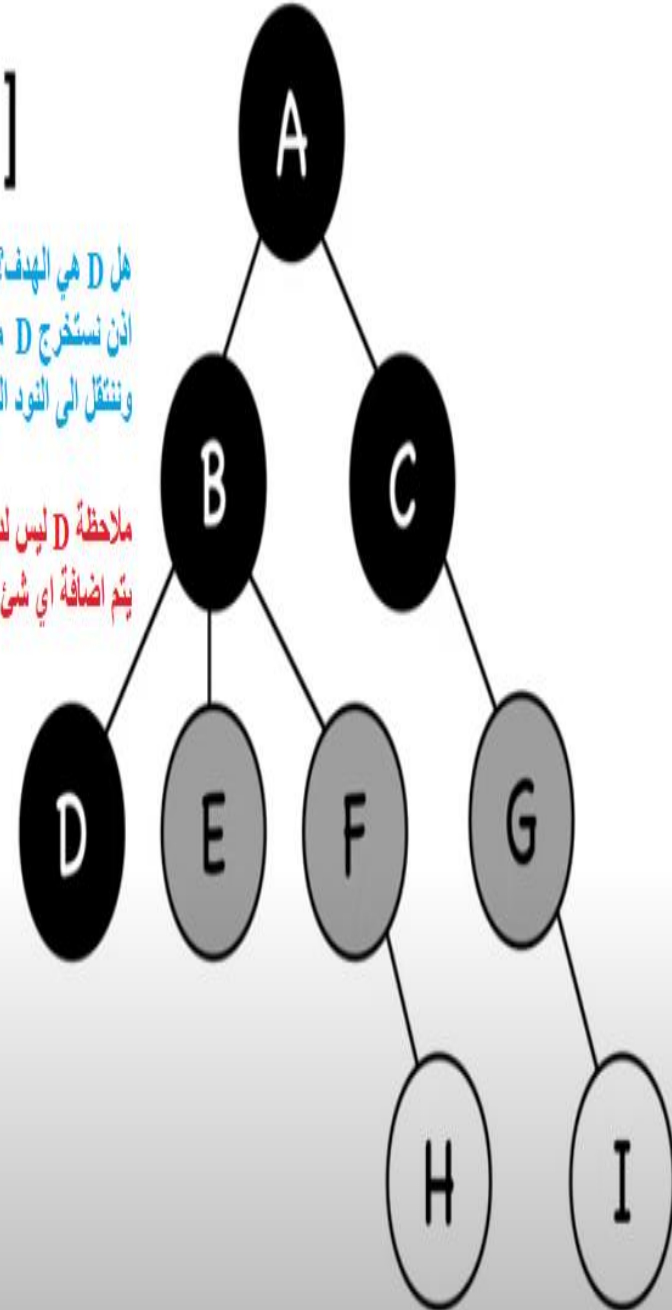


6

[G, F, E]

هل D هي الهدف؟ كلا
اذن نستخرج D من الكيو ونلون بالاسود وننتقل الى النود التي تليها بالكيو.

ملاحظة D ليس لديها لواحق - اطفال - لذا لم يتم اضافة اي شئ الى الكيو



7

[G, F]

8

[G]

9

[H, G]

A

B C

D E F G

H I

10

[H, G]

pop

A

B C

D E F G

H I

11

[H]

A — Level 0

هنا نستخرج نود G ونتوقف لانها الهدف وبهذا يكون ال PATH

PATH = [A,B,C,D,E,F,G]

B C — Level 1

D E F G — Level 2

H I — Level 3

ملاحظة اذا لم يحدد نود الهدف في السؤال فيجب ان نكمل الحل الى اخر نود موجودة بال **graph**.

ملاحظة: اذا اعطى في السؤال graph فيجب اولا ان نقوم بتحويله الى tree ومن ثم نحل باستخدام ال algorithm.

Code for Breadth First Search algorithm in python

temp.py × | untitled15.py ×

```python
from collections import deque

class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

    def bfs(self, start_node):
        visited = set()
        queue = deque()

        queue.append(start_node)
        visited.add(start_node)

        while queue:
            current_node = queue.popleft()
            print(current_node, end=' ')

            for neighbor in self.graph[current_node]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)

# Example usage:
if __name__ == "__main__":
    g = Graph()
    g.add_edge('A', ['B', 'C'])
    g.add_edge('B', ['A', 'D', 'E'])
    g.add_edge('C', ['A', 'F'])
    g.add_edge('D', ['B'])
    g.add_edge('E', ['B', 'F'])
    g.add_edge('F', ['C', 'E'])

    print("Breadth-First Traversal (starting from 'A'):")
    g.bfs('A')
```

I'll describe each line of the code in detail:

1. `class Graph:`: This line defines a Python class named `Graph`.

2. `def __init__(self):`: This is the constructor method for the `Graph` class. It is called when you create a new instance of the `Graph` class. Inside the constructor, an empty dictionary is created to store the graph data, and it is assigned to the instance variable `self.graph`.

3. `def add_edge(self, node, neighbors):`: This is a method for adding edges to the graph. It takes two arguments: `node` (the starting node) and `neighbors` (a list of nodes that are neighbors of the starting node). This method adds an entry to the graph dictionary where the key is the starting node (`node`) and the value is the list of neighbors (`neighbors`).

4. `def dfs(self, start_node, visited=None):`: This method is used to perform a depth-first search (DFS) traversal of the graph starting from a given `start_node`. It takes two arguments: `start_node` (the node to start the traversal from) and an optional `visited` set to keep track of visited nodes.

5. `if visited is None:`: This line checks if the `visited` set is not provided as an argument. If not, it initializes it as an empty set.

6. `if start_node not in visited:`: This line checks if the `start_node` has not been visited yet. If it hasn't, the code inside the if block will be executed.

7. `print(start_node, end=' ')`: This line prints the `start_node` to the console, separated by a space (using the `end` parameter of the `print` function). This is part of the DFS traversal and helps you see the order in which nodes are visited.

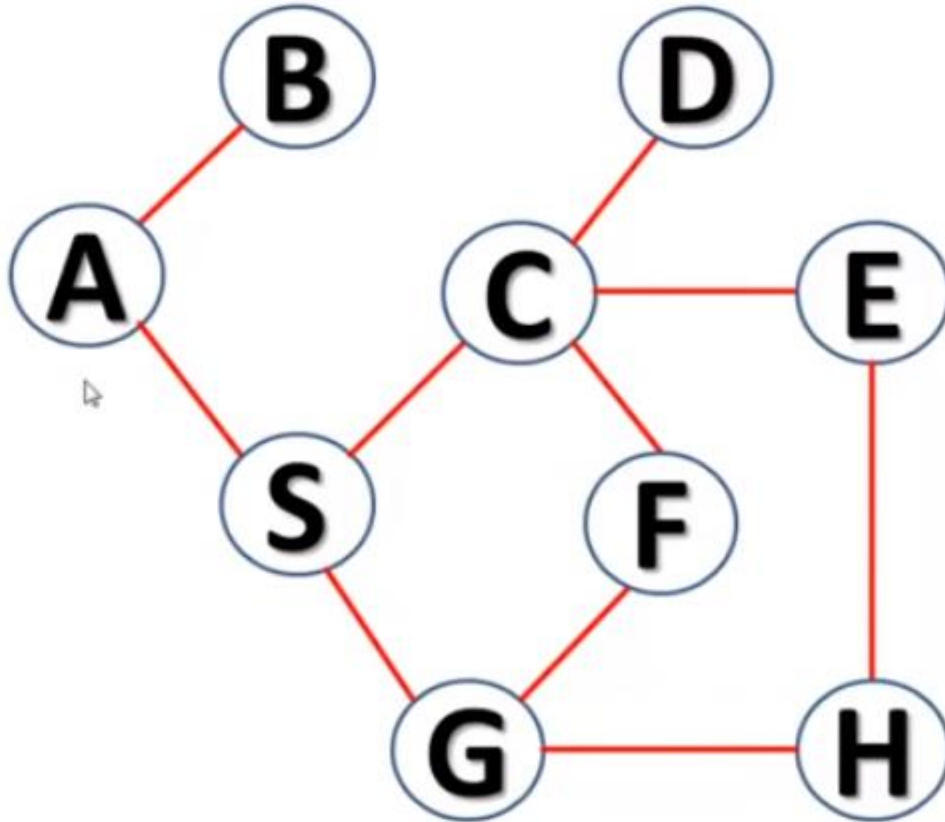
8. `visited.add(start_node)`: This line adds the `start_node` to the `visited` set to mark it as visited.

9. `for neighbor in self.graph[start_node]:`: This line iterates through the neighbors of the `start_node`, which are stored in the `self.graph` dictionary.

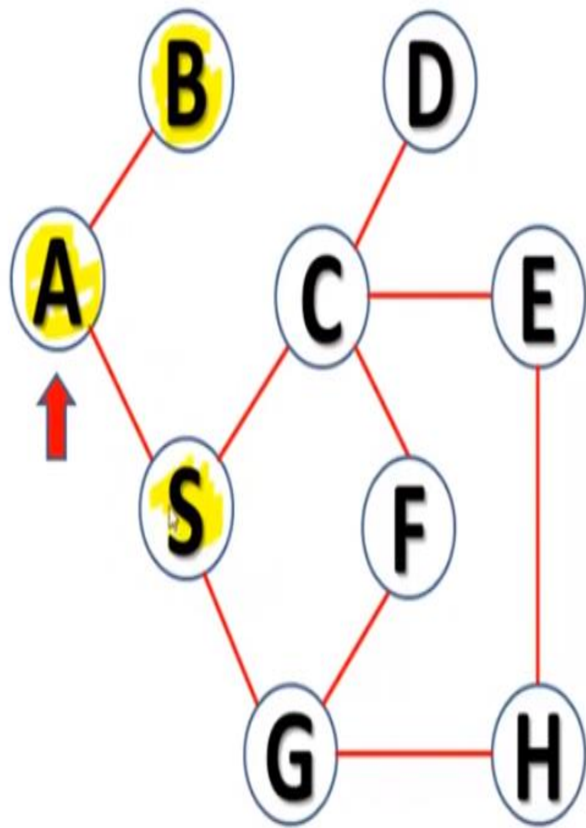
10. `self.dfs(neighbor, visited)`: This line recursively calls the `dfs` method on each neighbor of the `start_node`. This recursion allows the traversal to continue to unvisited neighbors in a depth-first manner.

Another BFS example

هنا لم يحدد ال goal state و
A is start state



Queue Status

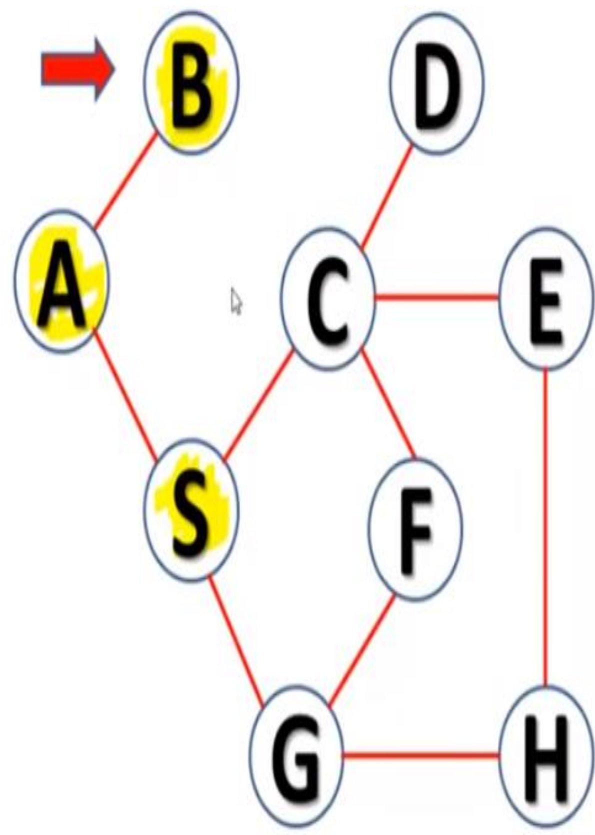


Queue Status

[S B]

OUTPUT : A B S

2

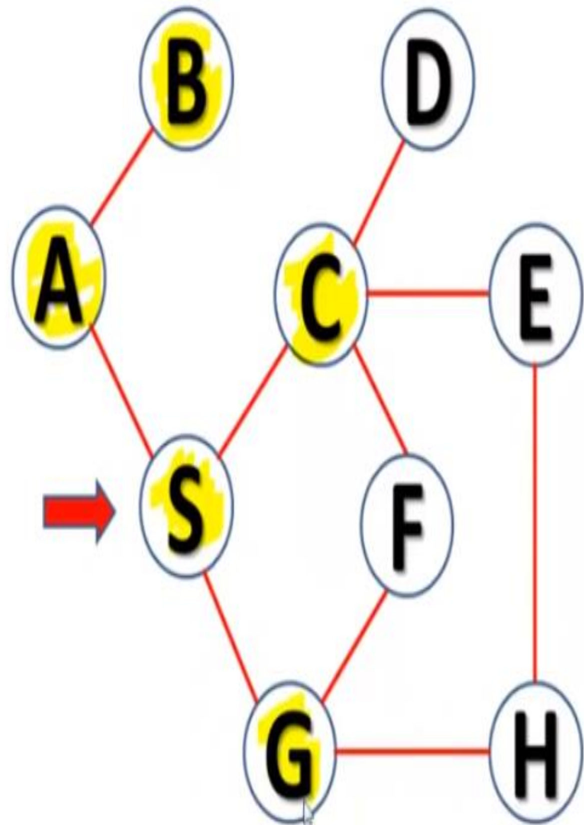


Queue Status

[S]

OUTPUT : A B S

3

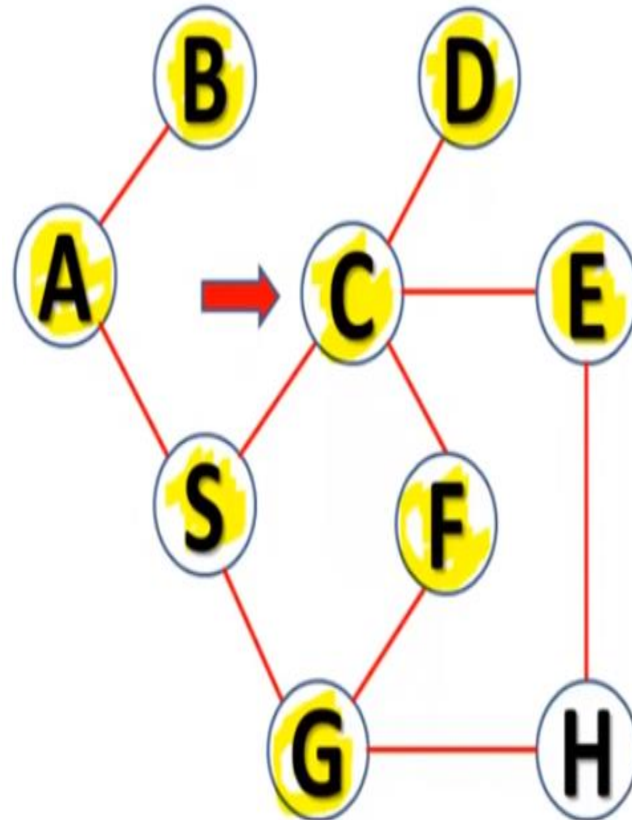


Queue Status

[G C]

OUTPUT : A B S C G

4

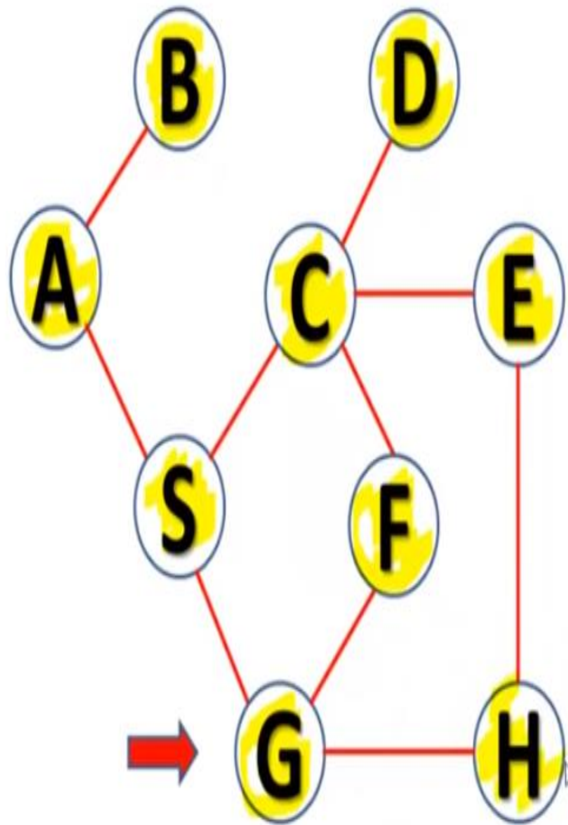


Queue Status

[F E D G]

OUTPUT : A B S C G D E F

5

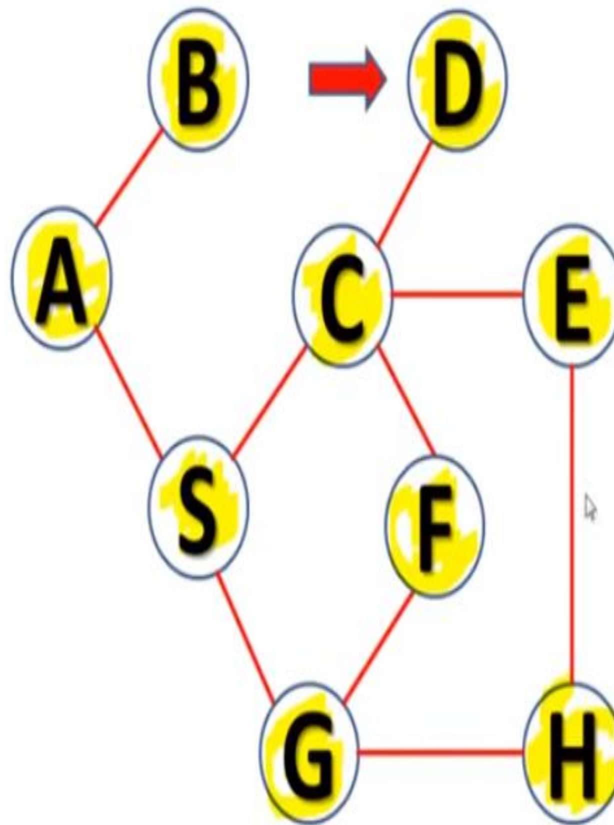


Queue Status

[H F E D]

OUTPUT : A B S C G D E F H

6

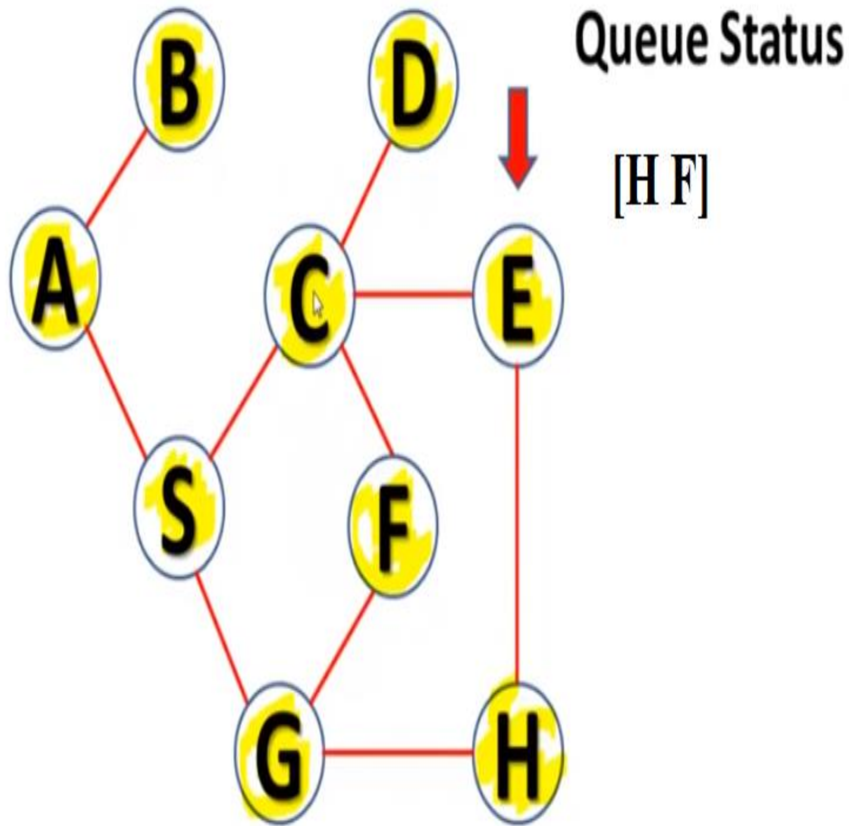


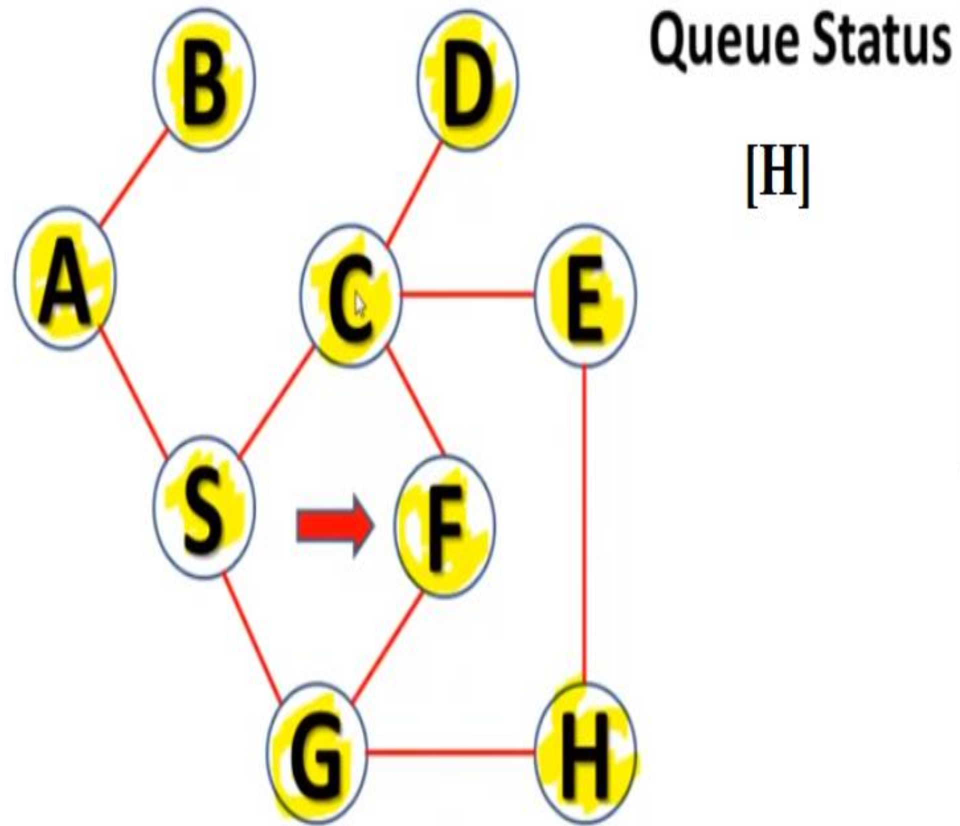
Queue Status

[H FH E]

OUTPUT : A B S C G D E F H

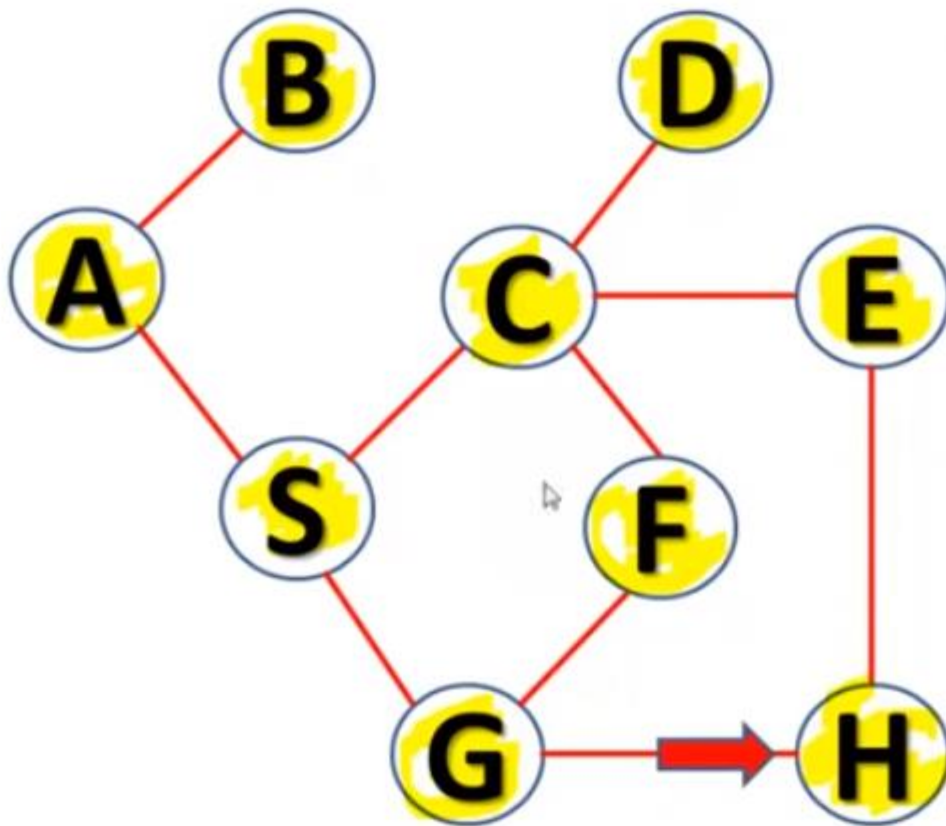
7





8

9



Queue Status

[ ]

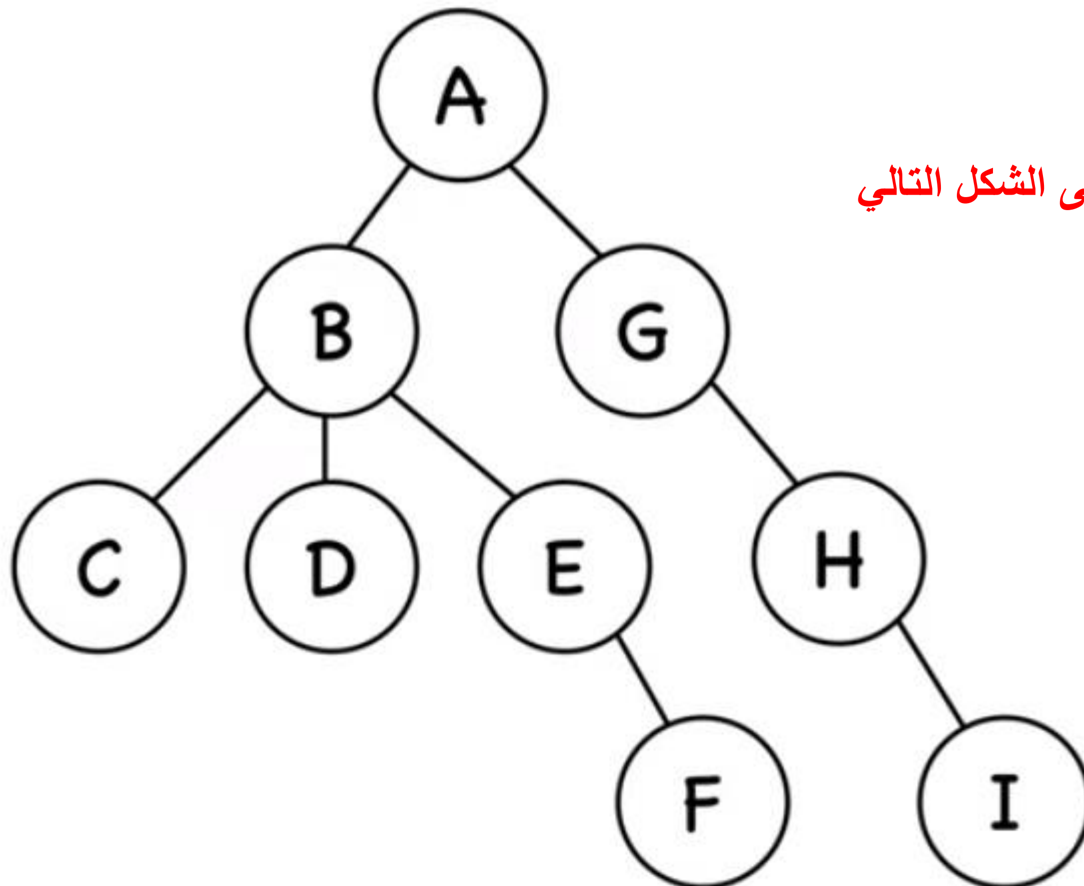
OUTPUT : A B S C G D E F H

2- Depth First Search Algorithm

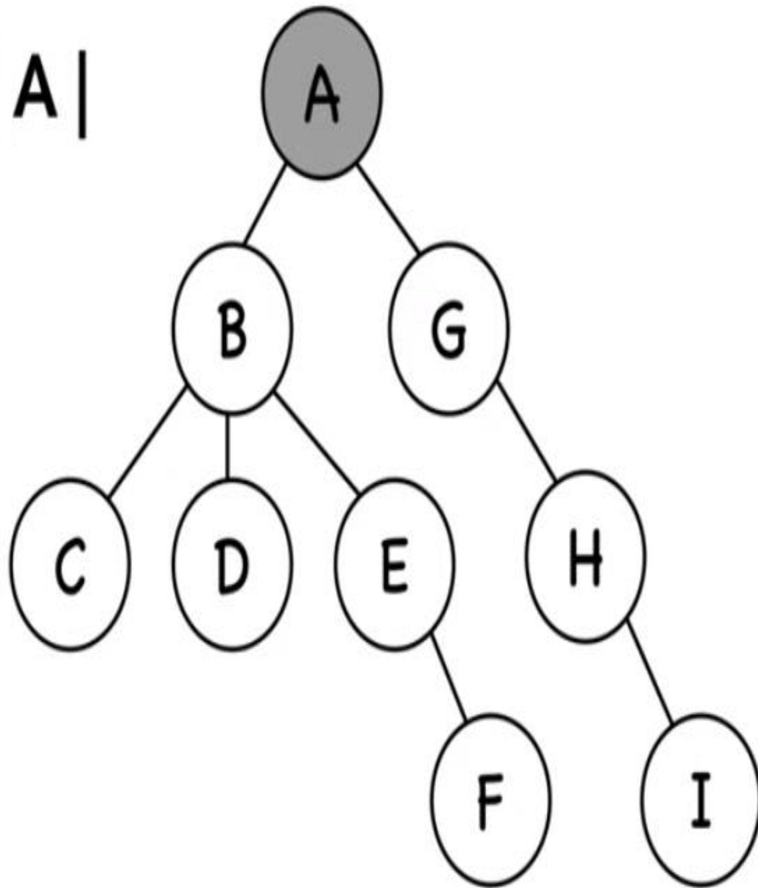
algorithm for searching a graph

depth = vertical before horizontal

stack

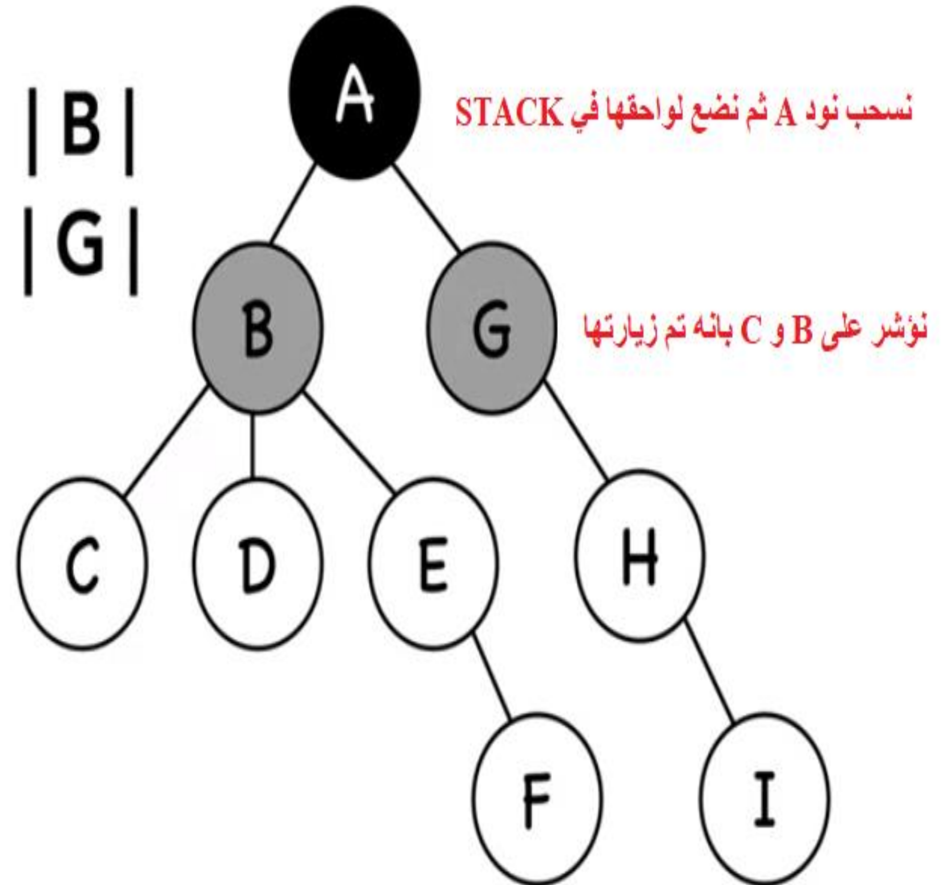


سنقوم بتطبيق DFS على الشكل التالي



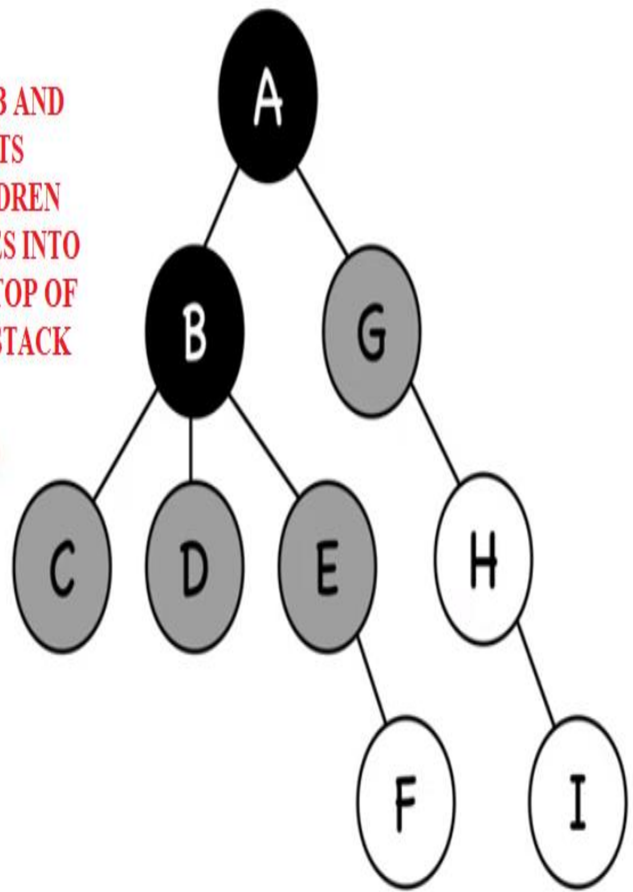
2

POP A FROM THE STACK

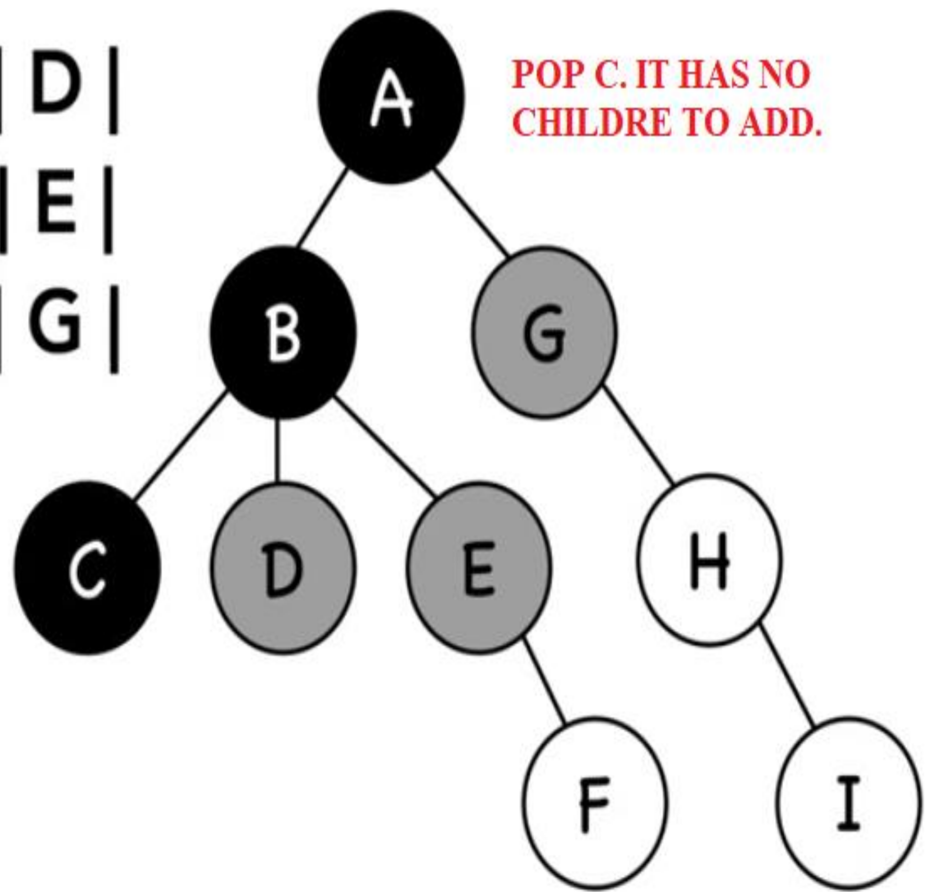


3

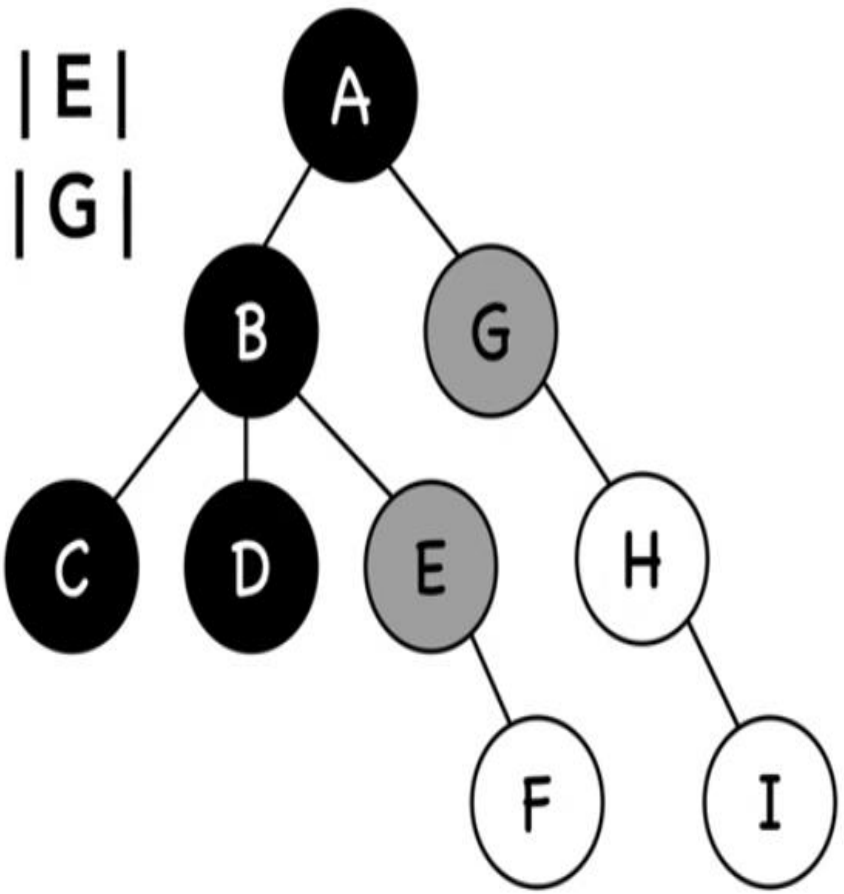
|C|
|D|
|E|
|G|



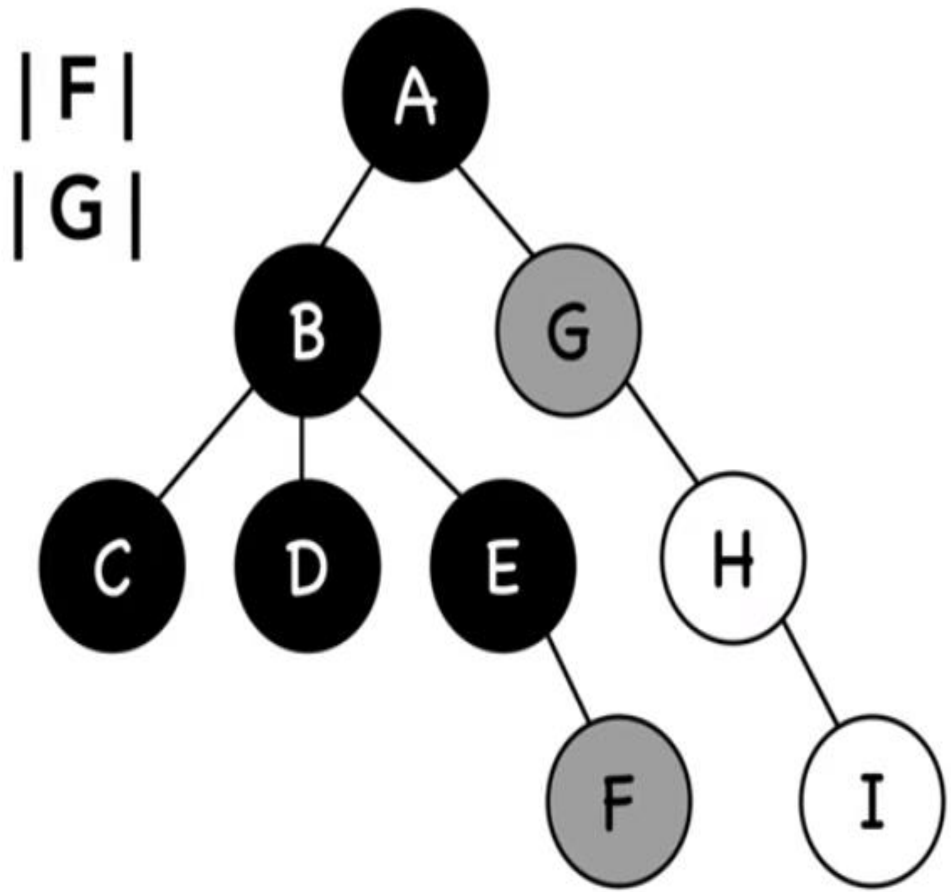
4



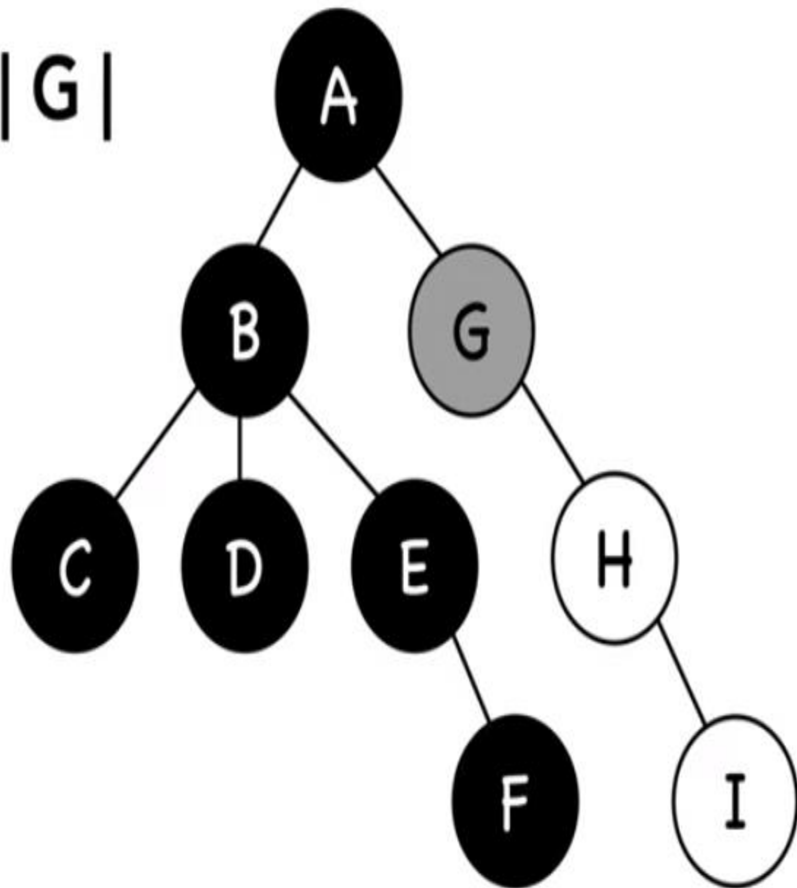
5



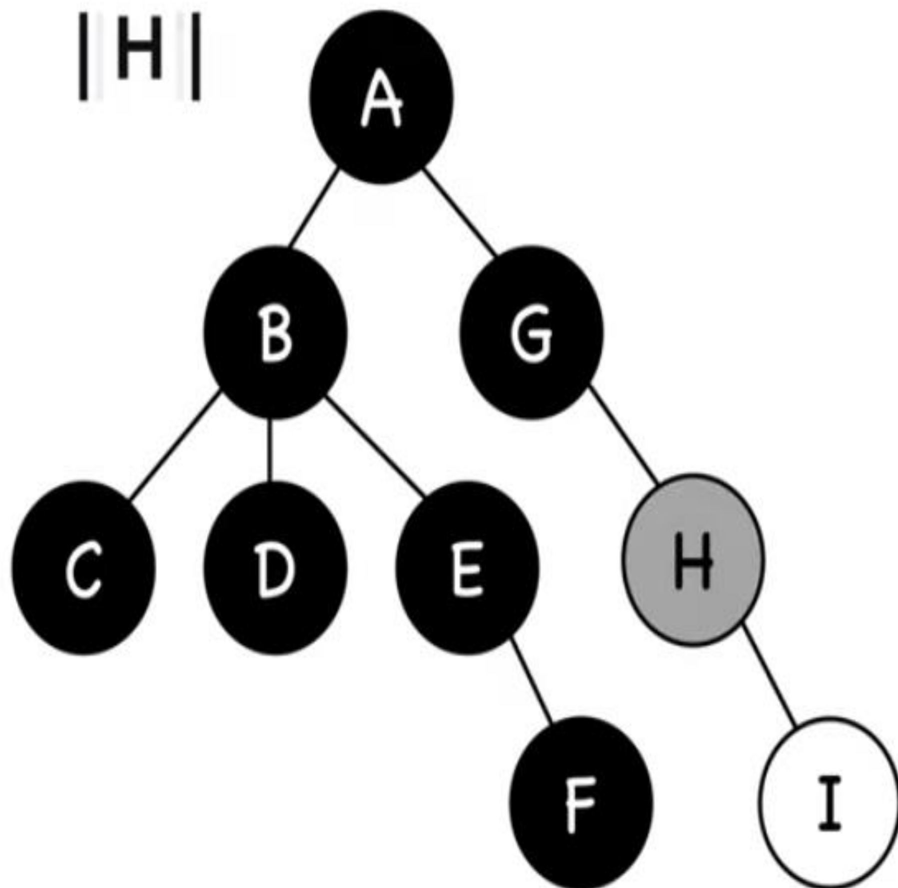
6



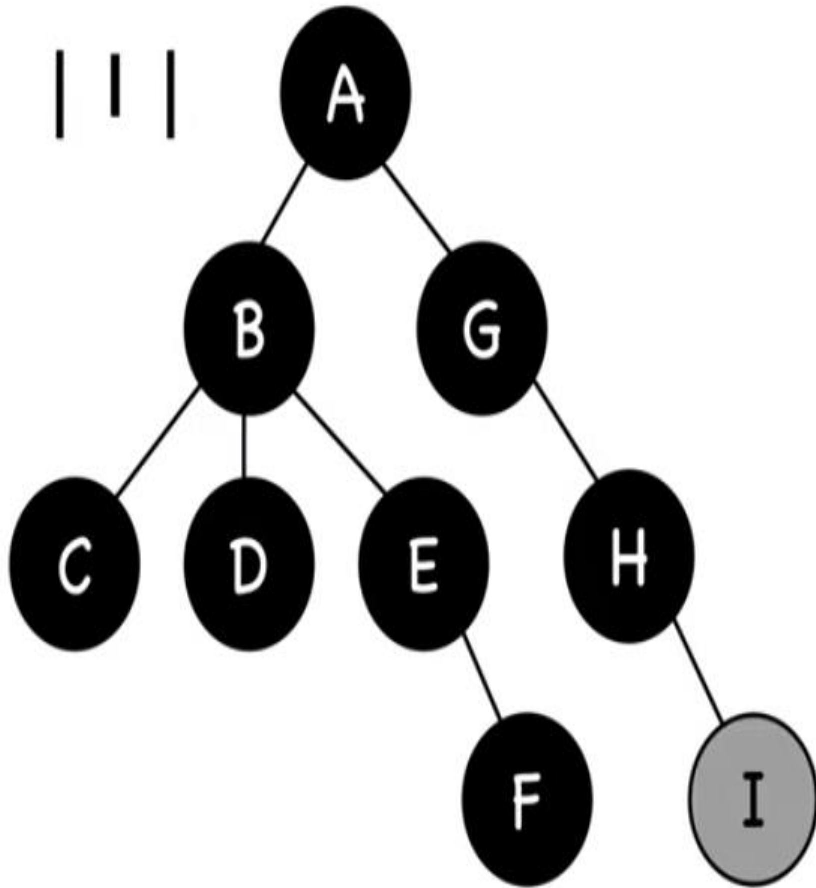
7



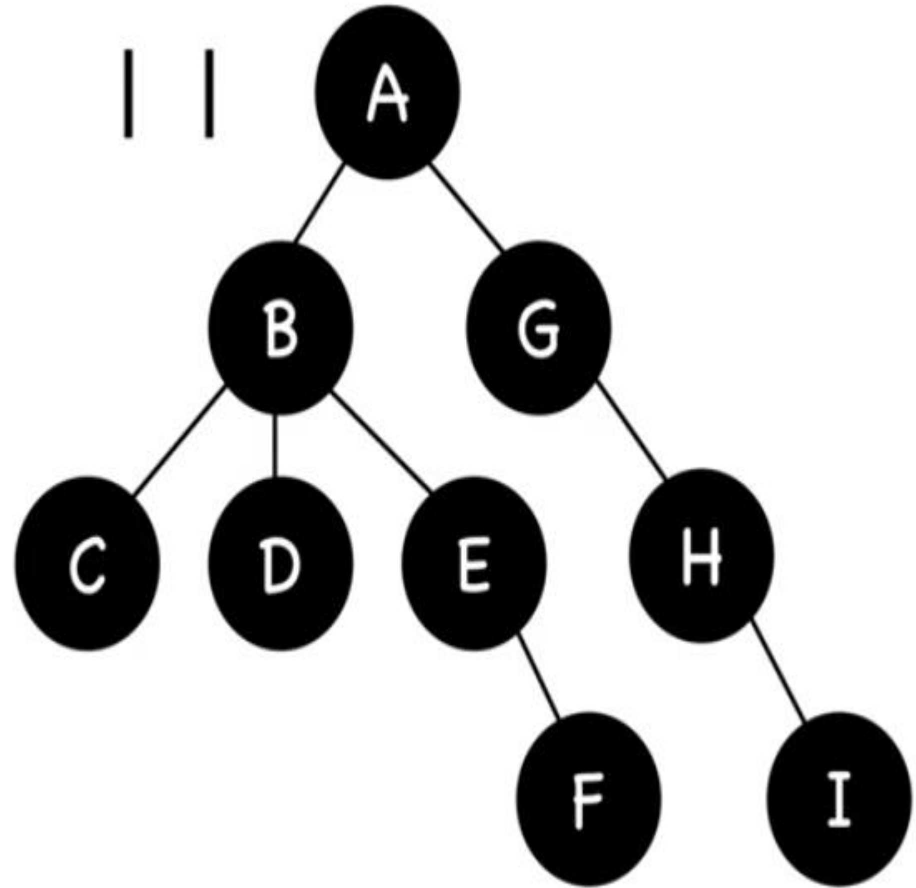
|G|
A
B
G
C
D
E
H
F
I



|H|
A
B
G
C
D
E
H
F
I



10



PATH = A B C D E F G H I

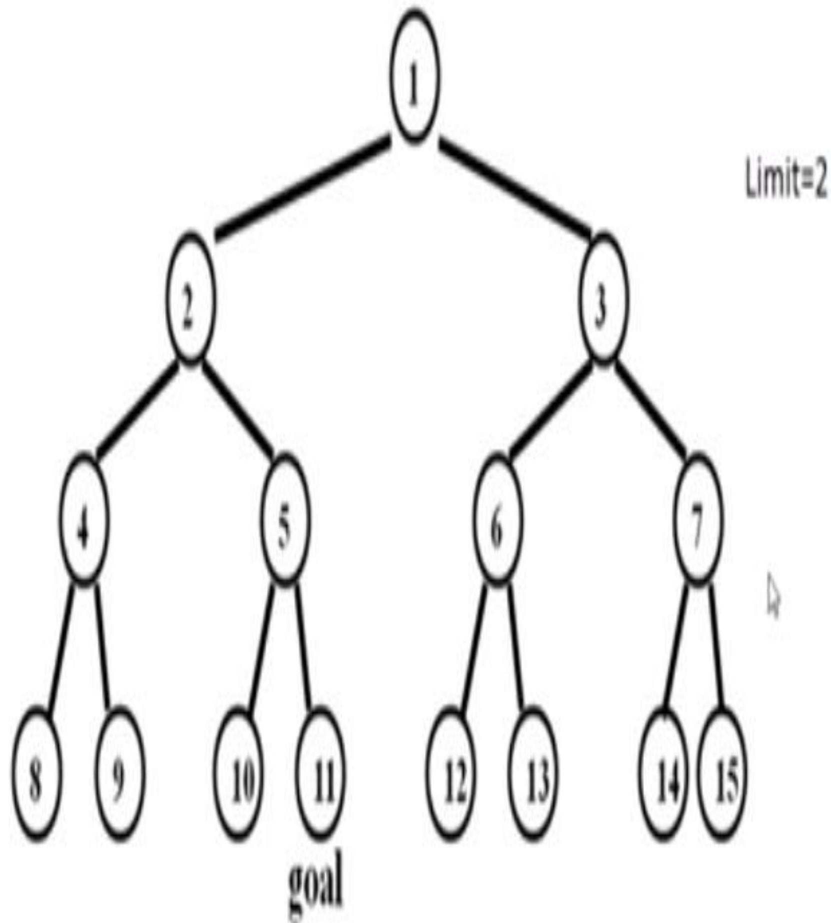
9

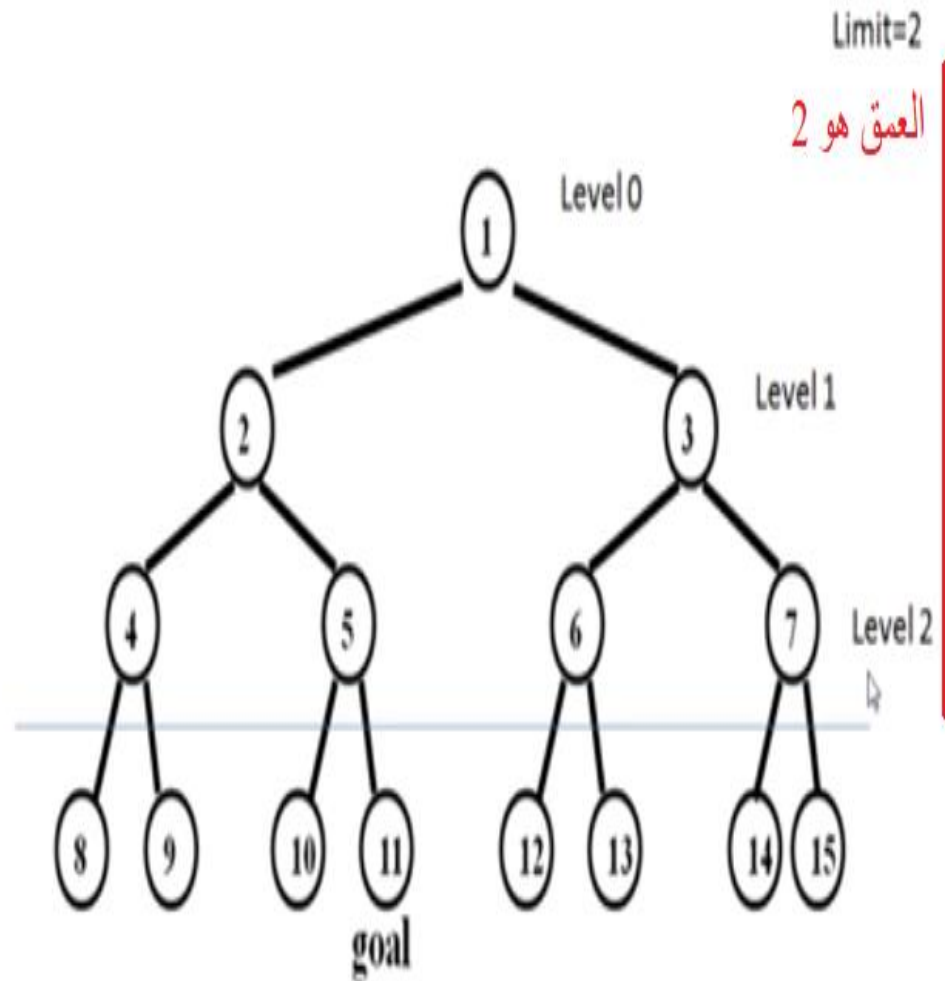
Code for DFS Algorithm in python

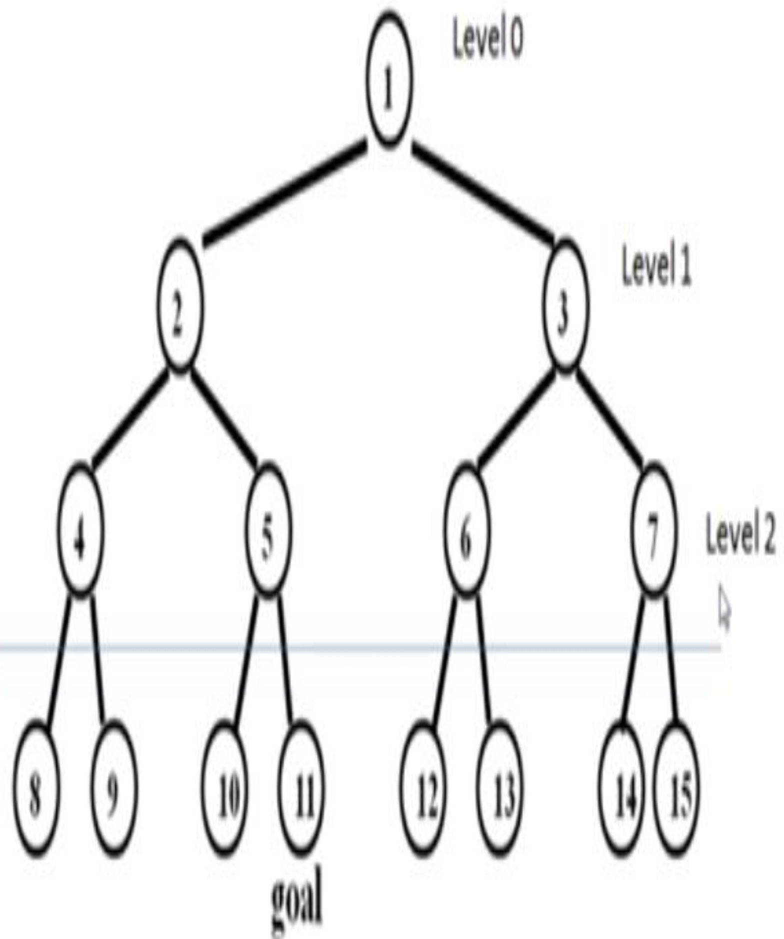
temp.py × untitled15.py ×

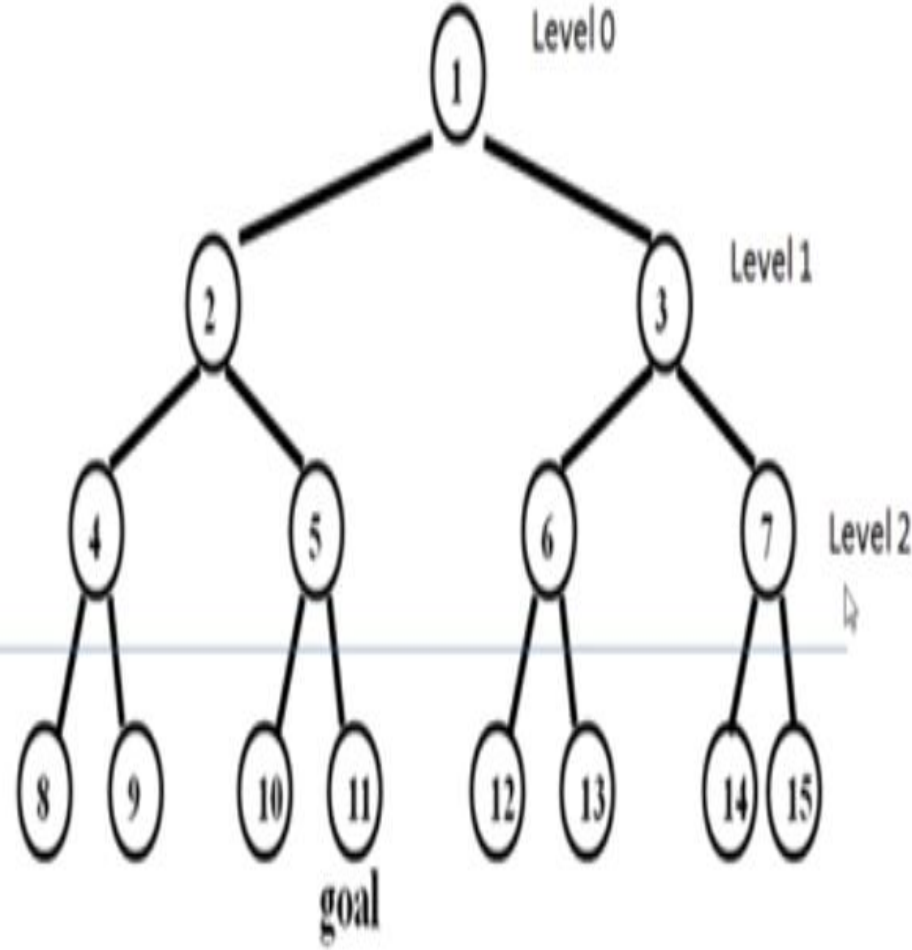
```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, node, neighbors):
        self.graph[node] = neighbors

    def dfs(self, start_node, visited=None):
        if visited is None:
            visited = set()

        if start_node not in visited:
            print(start_node, end=' ')
            visited.add(start_node)

            for neighbor in self.graph[start_node]:
                self.dfs(neighbor, visited)

# Example usage:
if __name__ == "__main__":
    g = Graph()
    g.add_edge('A', ['B', 'C'])
    g.add_edge('B', ['A', 'D', 'E'])
    g.add_edge('C', ['A', 'F'])
    g.add_edge('D', ['B'])
    g.add_edge('E', ['B', 'F'])
    g.add_edge('F', ['C', 'E'])

    print("Depth-First Traversal (starting from 'A'):")
    g.dfs('A')
```

3.Depth Limited Search Algorithm

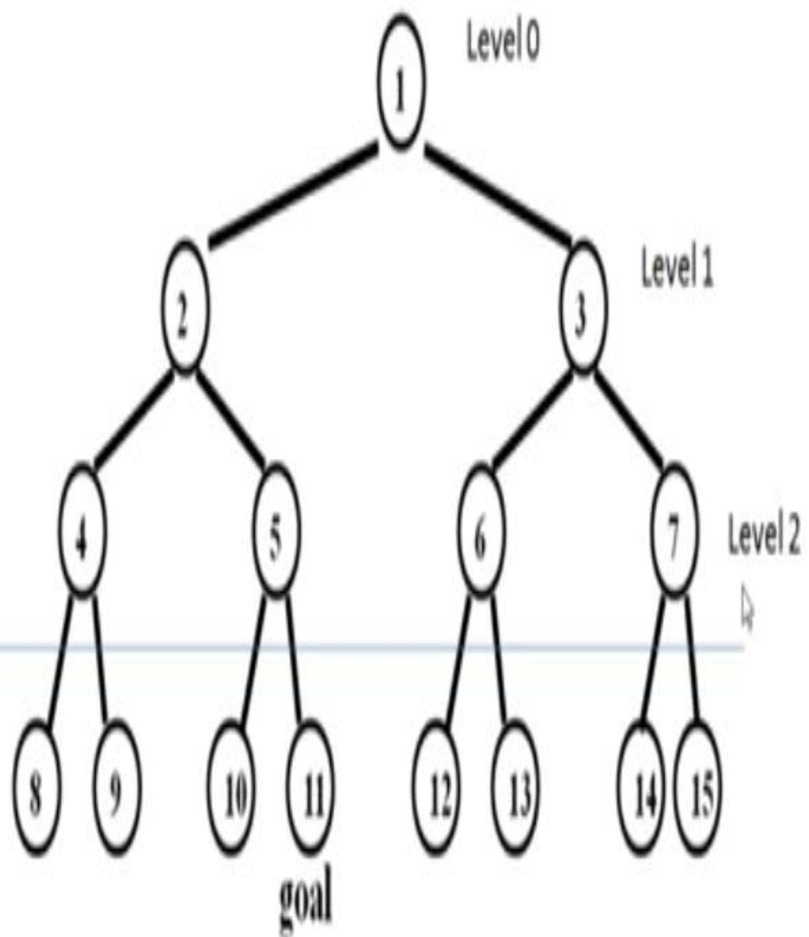





Limit=2
Level 0
Level 1
Level 2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
goal
queue = [2 3]

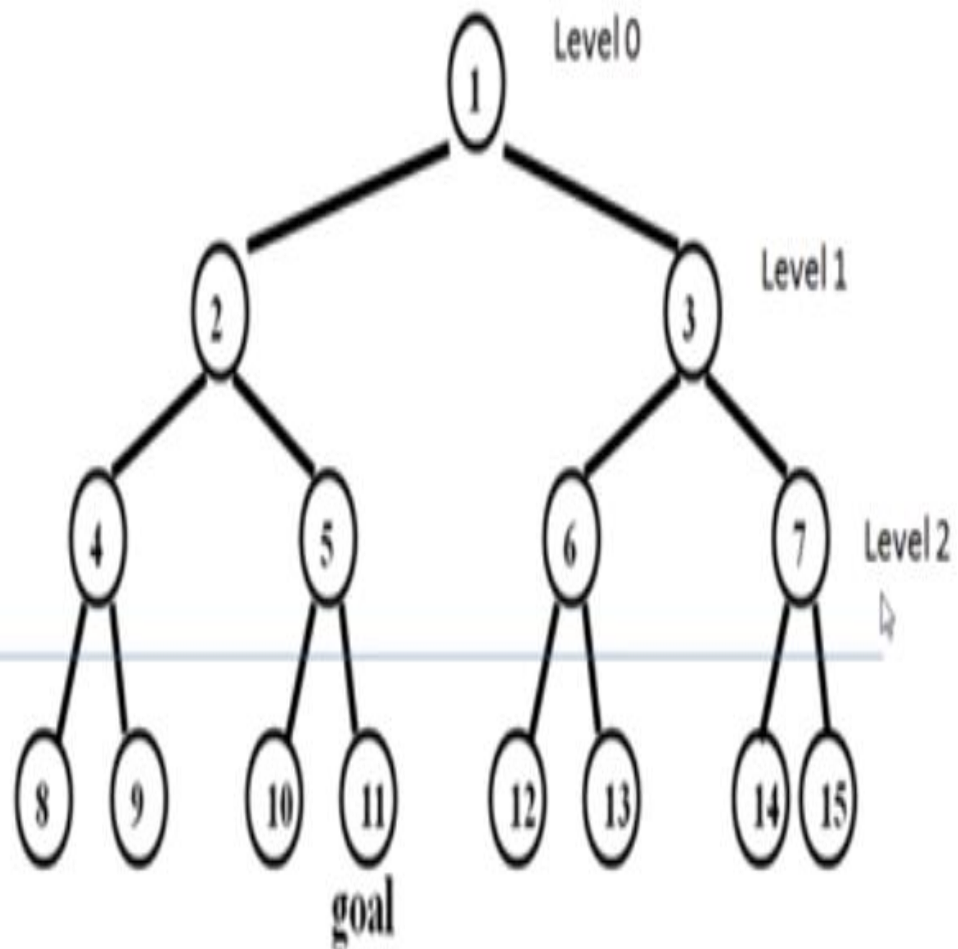

Limit=2
Level 0
Level 1
Level 2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
goal
queue = [4 5 3]

Limit=2

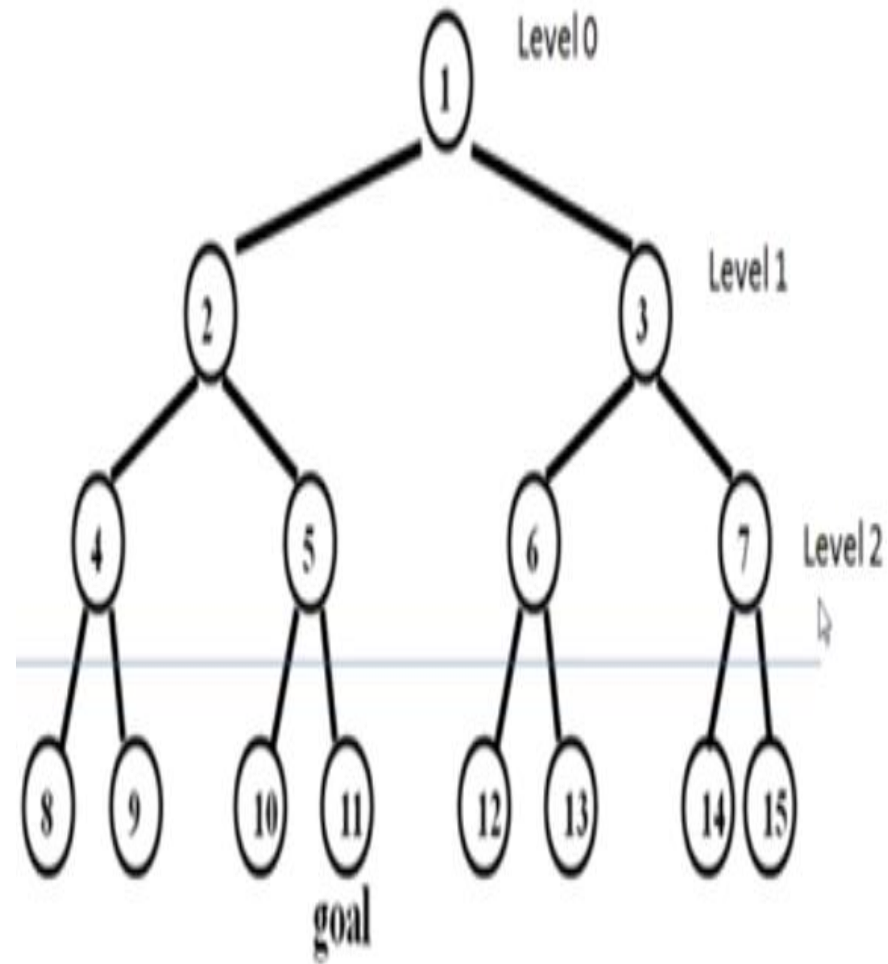


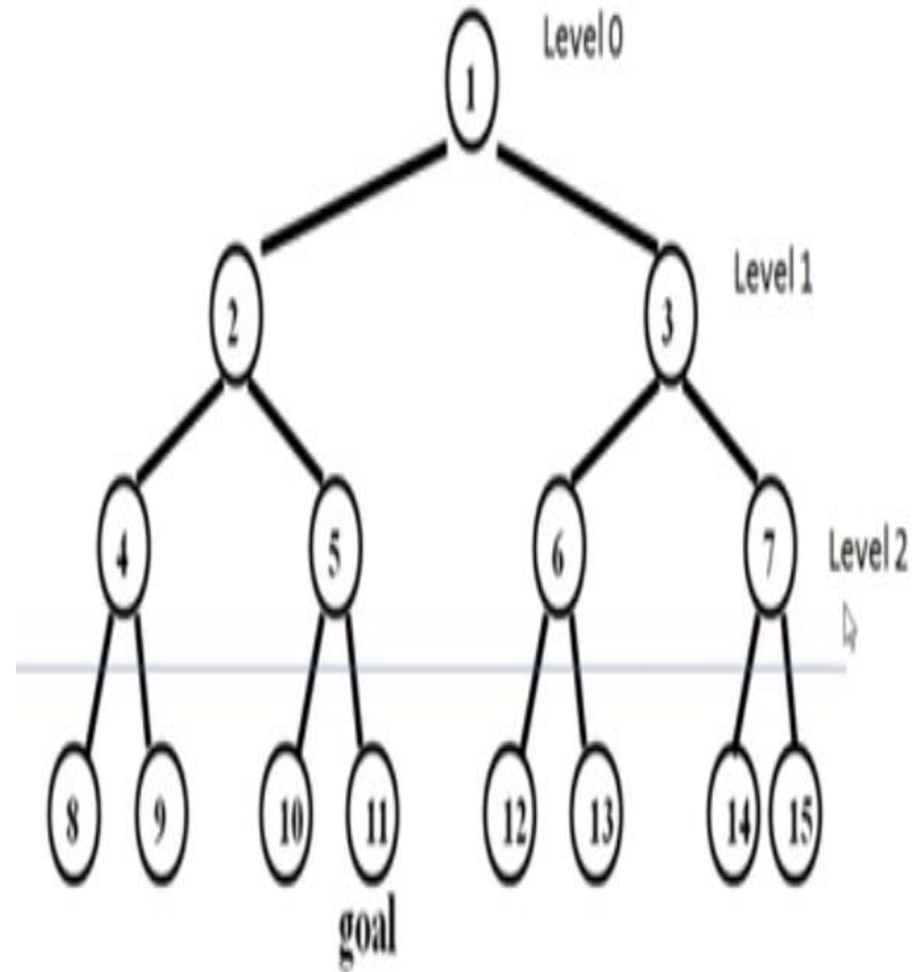
queue = [5 3]

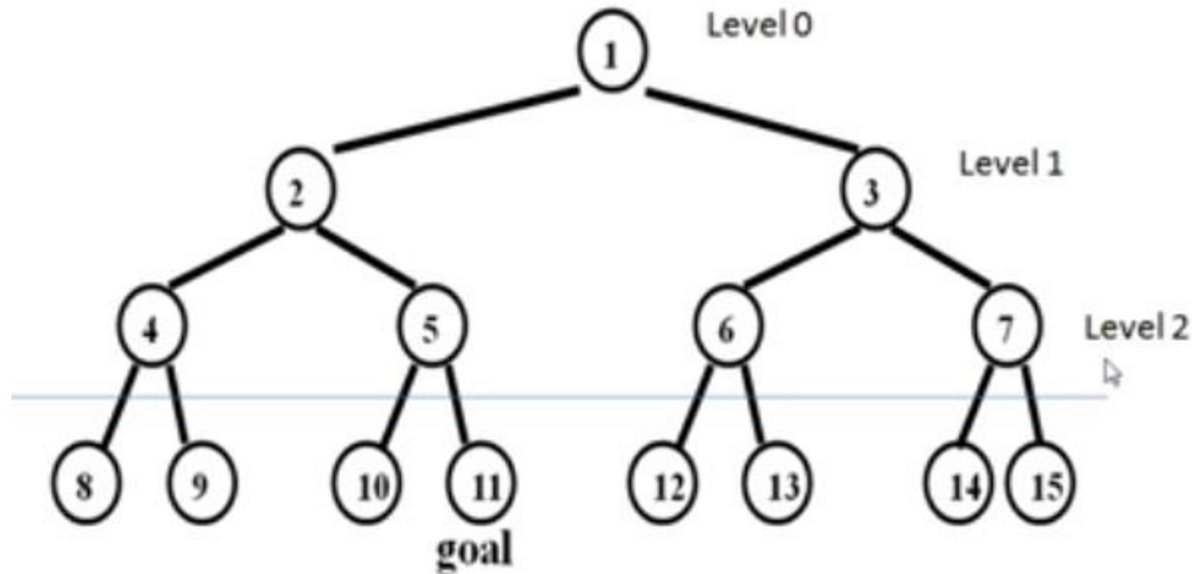
Limit=2



queue = [3]


Limit=2
Level 0
Level 1
Level 2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
goal
queue = [6 7]


Limit=2
Level 0
Level 1
Level 2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
goal
queue = [7]



No goal found so incomplete.

- This limitation helps avoid infinite loops in cases where a solution may not exist or to ensure that the search does not become overly time-consuming.

4. Uniform-Cost Search (UCS)

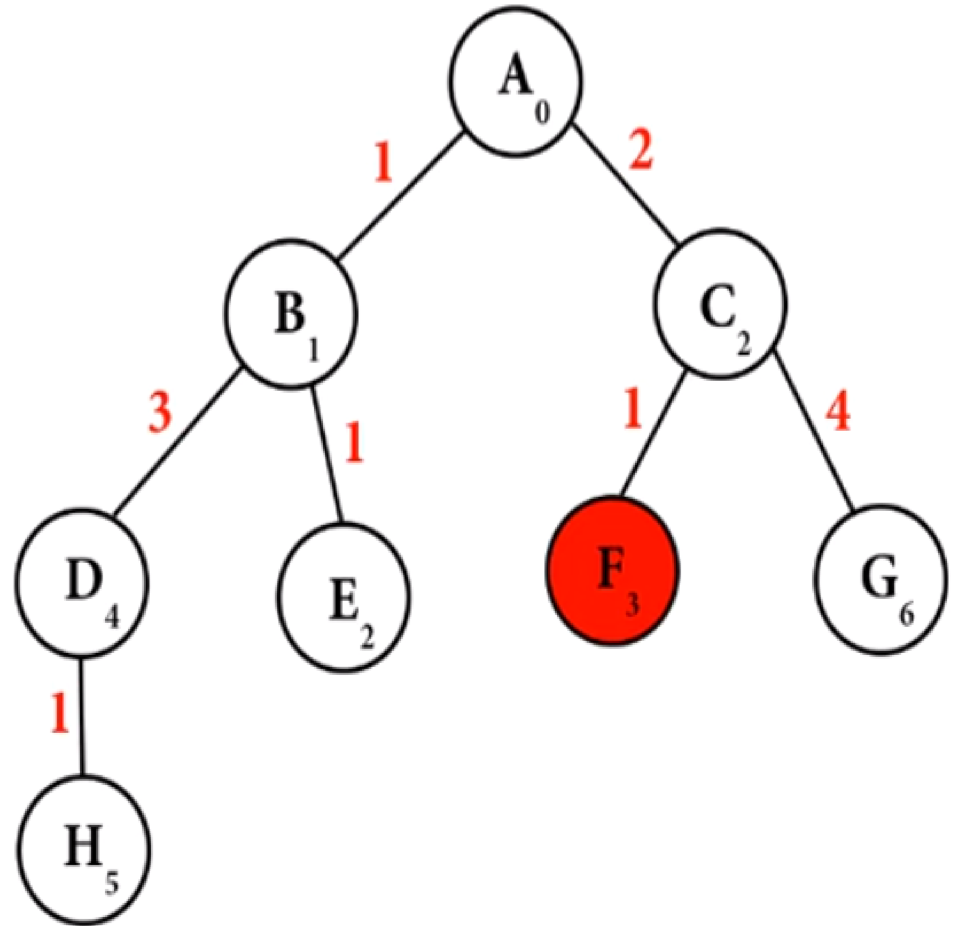
اولا نحسب كلفة الوصول لكل node

بما ان A ليس قبلها node اذن كلفة الوصول اليها = صفر

كلفة الوصول من A الى B هي 1=1+0

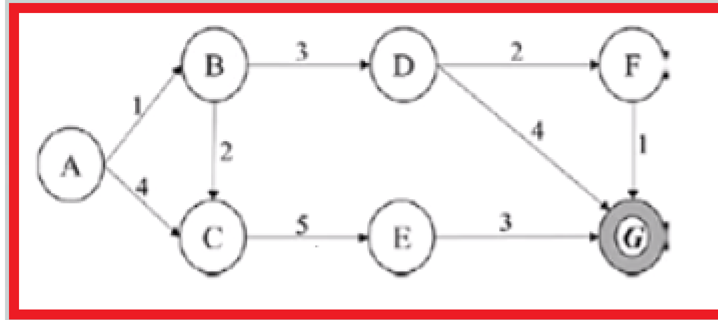
كلفة الوصول الى D هي 4 =3+1

closed	node	arranged
—	A_0	
A_0	B_1 C_2	
B_1	C_2 D_4 E_2 →	C_2 E_2 D_4
C_2	E_2 D_4 F_3 G_6 →	E_2 F_3 D_4 G_6
E_2	F_3 D_4 G_6	
F_3	Goal	



نضع A مع كلفتها تحت عمود ال node. هل Aهي ال Goal؟ لا . اذن نضع A تحت عمود closed ونضع اطفالها مع كلفهم تحت عمود ال node وبشكل مرتب من الاصغر كلفة الى الاكبر(مرتبة مسبقا). هل B هي الهدف؟ كلا. اذن نضع B تحت عمود ال closed ونضيف اطفالها مع كلفهم تحت عمود ال node ثم نرتب من الاصغر الى الاكبر. بما ان C و E كلفتها متساوية فتكون الاسبقية لل C لانها من مستوى level اعلى. نستمر بنفس الاسلوب الى ان نجد الهدف او نزور كل ال nodes.

Home work#1: use UCS to solve the problem?



Informed Search Algorithms

1. A* Algorithm

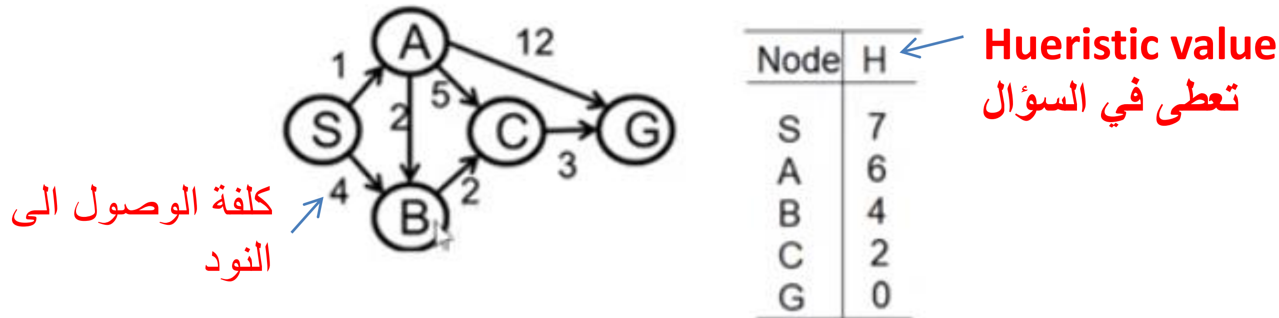
The A* algorithm is a popular heuristic search algorithm that **combines the advantages of both BFS and DFS.** It uses a heuristic function that estimates the cost of reaching the goal from a given node. A* uses a priority queue to prioritize nodes to explore based on a combination of the actual cost to reach a node (g-cost) and the estimated cost to reach the goal (h-cost). The algorithm guarantees finding the optimal solution if the heuristic is admissible (i.e., it never overestimates the true cost).

Heuristic Search:

Heuristic search algorithms, like A*, are especially useful when dealing with **large search spaces** where uninformed search strategies can be **impractical**. By using heuristics to estimate the cost, these algorithms can efficiently guide the search towards the most promising paths while avoiding unnecessary exploration.

A* Search

Use A* search algorithm to find the path between S and G



Node	H
S	7
A	6
B	4
C	2
G	0

هنالك 3 مصطلحات مهمة جدا يجب معرفتها والتمييز بينها وهي

1- G-cost

2- H-cost

3- F-cost

G-cost : the G-cost to move from S to A to C = 1 + 5 = 6

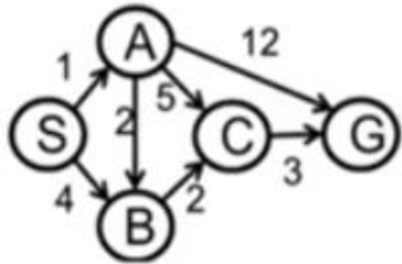
H-cost : the H value for the last node in the path. The H-cost to move from S to A to C= 2

F-cost: G-cost + H-cost = 6 + 2 = 8

Solution idea:

- Use a priority queue (like uniform-cost search)
- Pop the element with least F-cost
- If two elements have the same F-cost, use alphabetic order
- Remember: If a node is already visited we don't expand it again

يستخدم الاسبقية
ياخذ الاقل F-cost
اذا تساوت F-cost اذن نتبع الحروف الابجدية
يتم زيارة كل نود مرة واحدة فقط



Node	H
S	7
A	6
B	4
C	2
G	0

Solution steps:

Current	priority queue
	[S] (7) ← F-cost = 0 + 7
[S] (7)	[S,A] (7) [S,B] (8)
[S,A] (7)	[S,B] (8) [S,A,B] (7) [S,A,C] (8) [S,A,G] (13)
[S,A,B] (7) ← نختار الاقل	[S,B] (8) [S,A,C] (8) [S,A,G] (13) [S,A,B,C] (7)
[S,A,B,C] (7)	~~[S,B]~~ (8) visited ~~[S,A,C]~~ (8) visited [S,A,G] (13) [S,A,B,C,G] (8)
[S,A,B,C,G] (8)	

Solution:
S -> A -> B -> C -> G

G-cost = 8
H-cost = 0
F-cost = 8

خوارزميات البحث الذكية

أنواع خوارزميات البحث

خوارزميات البحث المنظم (حدسية)
informed Search Algorithm

خوارزميات البحث الغير منظم
Uninformed
Search Algorithm

خوارزميات اللعب ضد الحاسب
Gaming algorithms

<u>مثال اخر عن الخوارزميات التجريبية</u>:

Hill Climbing Algorithm (Manhattan Distance)

خوارزميات البحث الذكية

8-puzzle

الحالة الإبتدائية للمشكلة

Initial state

1	3	4
2	5	7
6	8	

الهدف المراد تحقيقه

Goal state

	1	2
5	4	3
8	7	6

1	2	3
8		4
7	6	5

1	2	3
4	5	6
7	8	

العمليات (الانتقالات)

Actions

1	3	4
2	5	7
6	8	

يمكنني عمل 2 حركة مختلفة

تحريك الفراغ للأعلى و لليسار

1	2	4
3	5	
6	8	7

يمكنني عمل 3 حركات مختلفة

1	2	3
5		6
4	8	7

يمكنني عمل 4 حركات مختلفة

تحريك الفراغ للأعلى و الأسفل واليمين واليسار

1- نقوم بتمثيل الاحتمالات.
2- نحسب القيم التجريبية لكل حالة (manhattan distance) عدد الحرات اللازمة لتحريك ل قطعة ليست في مكانها الصحيح.
3- نختار القيم التجريبية الاقل حتى نصل الى الهدف.
وكما موضح في ادناه

خوارزميات البحث الذكية

8-puzzle

حالات المشكلة :states

جميع الحالات المختلفة التي نمر بها للوصول إلى الحل

يمكنني القيام ب 4 حركات

up
left
down
right

يمكنني القيام ب 3حركات

يمكنني القيام ب 3حركات

تحريك الفراغ للأعلى والأسفل واليسار

down
left
right
right
up
down
up
left
right
up
down
left

يمكنني القيام ب 2حركة

...

...

كيفية حساب Manhattan distance:

خوارزميات البحث الذكية

(0,0) (0,1) (0,2)

3	2	8
4	5	6
7	1	

الحالة الإبتدائية للمشكلة

Initial state

(0,0) → (0,2)

Manhattan Distance

md=|0-0|+|0-2|

md = 0+2

md =2

1	2	3
4	5	6
7	8	

الهدف المراد تحقيقه

Goal state

خوارزميات البحث الذكية

3	2	8
4	5	6
7	1	

الحالة الإبتدائية للمشكلة

Initial state

(0,2) → (2,1)

Manhattan Distance

md=|0-2|+|2-1|

md = 2 +1

md =3

1	2	3
4	5	6
7	8	

الهدف المراد تحقيقه

Goal state

نحسب MD لكل الارقام التي ليست في مكانها الصحيح ونجمع النواتج حتى نجد قيمة كل حالة

الحل النهائى: