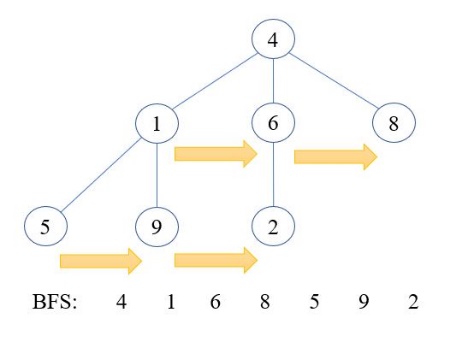
**حل 8 Puzzle با استفاده از جستجوهای ناآگاهانه**

در این پروژه ما قصد داریم که 8 Puzzle را با استفاده از 4 روش ناآگاهانه ی BFS ، DFS ، LDS و IDS حل کنیم. در ادامه به روش کار هر یک از این الگوریتم ها می پردازیم.

1. الگوریتم BFS :

الگوریتم پیمایش اول سطح یا جستجوی اول سطح (Breadth First Search - BFS) از جمله الگوریتم‌های مشهور پیمایش و جستجوی گراف است که در حل مسائل الگوریتمی و هوش مصنوعی کاربرد دارد. این الگوریتم برای پیمایش و جستجوی گراف از یک صف برای نگهداری ترتیب جستجو استفاده می‌کند.

[](http://www.algorithmha.ir/جستجوی-اول-سطح-bfs/)

الگوریتم BFS با وارد کردن گره مبدأ به صف پردازش شروع شده و تا خالی نشدن این صف مراحل زیر را تکرار می‌شود:

1) عنصر جلوی صف را به عنوان گره جاری انتخاب و از صف حذف کن.

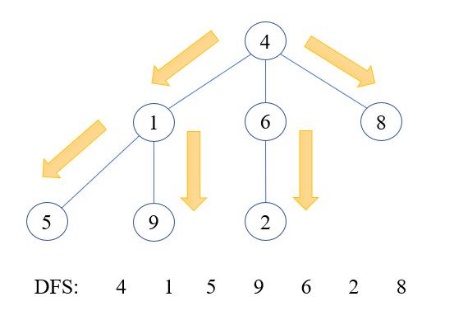
2) گره جاری را پردازش کن.

3) گره‌های مجاور گره جاری که پردازش نشده و در صف پردازش نیز قرار ندارند به این صف اضافه کن.

منظور از پردازش، هر عملی روی گره است که پیمایش یا جستجو به آن نیت صورت گرفته است.

2. الگوریتم DFS :

الگوریتم جستجوی اول عمق (Depth First Search - DFS) یا نام‌های دیگری همچون جستجو در عمق، پیمایش اول عمق، پیمایش عمق اول الگوریتمی مشابه الگوریتم جستجوی اول سطح (BFS) برای پیمایش گراف است. این دو الگوریتم خواص و کاربردهای مشترک بسیاری دارند و تفاوت اصلی در این است که در هر تکرار الگوریتم DFS تنها یکی از گره‌های مجاور گره پردازش شده برای مرحله بعد انتخاب می‌شود. به این ترتیب، الگوریتم DFS به جای صف از یک پشته برای مشخص کردن مسیر پیمایش استفاده می‌کند.

[](http://www.algorithmha.ir/جستجوی-اول-عمق-dfs/)

الگوریتم DFS با فرض انتخاب گره مبدأ به عنوان گره جاری از مراحل زیر تشکیل یافته است:

1) گره جاری را به پشته اضافه کن.

2) گره جاری را پردازش کن.

3) از گره‌های مجاور گره جاری یک گره پیمایش نشده را به عنوان گره جاری انتخاب کرده و برو به مرحله (1).

4) اگر همه گره‌های مجاور گره جاری پیمایش شده‌اند، گره بالای پشته را به عنوان گره جاری از پشته حذف کرده و برو به مرحله (3).

5) اگر گرهی در پشته وجود ندارد، اجرای الگوریتم را متوقف کن.

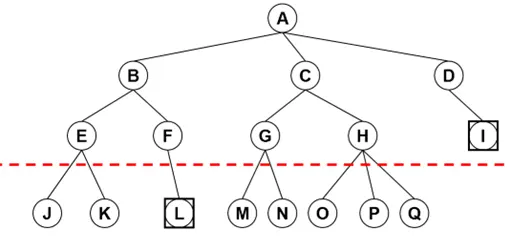
منظور از پردازش، هر عملی روی گره است که پیمایش یا جستجو به آن نیت صورت گرفته است.

3. الگوریتم LDS :

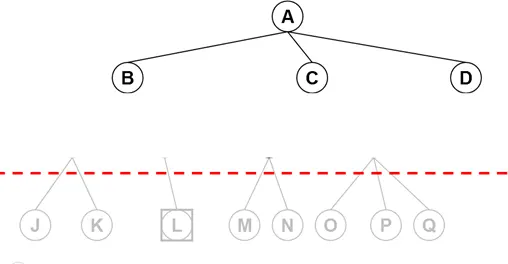
روش این الگوریتم جستجوی با عمق محدود است کهدر آن اساس کار همان الگوریتم جستجوی عمقی است با این تفاوت که به جای اجرای الگوریتم روی کل درخت جستجو ، جستجو را تا یک عمق معینی از درخت انجام می دهیم ، در این حالت شانس رسیدن به جواب هایی که در عمق کمتری از درخت قرار دارند افزایش پیدا می کند. هرچند در این روش جواب هایی که در عمق پایین تری از عمق تعیین شده قرار دارند از دسترس خارج خواهند شد.  حال به بررسی این روش بر روی مثال زیر می پردازیم.

اگر فرض کنیم درخت جستجویی به شکل زیر داریم می خواهیم جستجوی با عمق محدود را بر روی آن اجرا کنیم. محدوده عمق جستجو را با خط قرمز مشخص می کنیم.

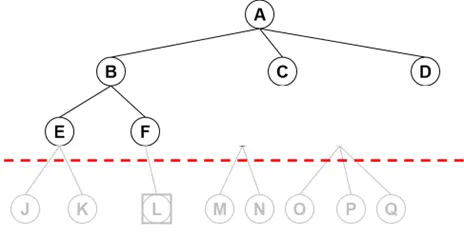
**مرحله اول**



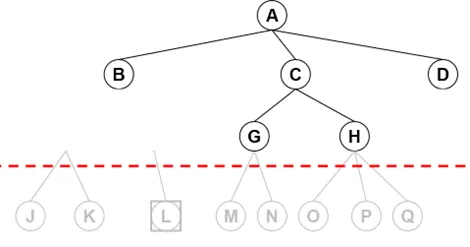
**مرحله دوم**



**مرحله سوم**



**مرحله چهارم**



**مرحله پنجم**

A picture containing diagram, line, white, plot

Description automatically generated

در مثال فوق به خوبی مشخص است که الگوریتم دیگر گرفتار عمق نامحدود نمی شود و همینطور به جواب **I** رسیدیم که در لایه دوم از درخت جستجو قرار دارد. اما به بررسی کارایی این الگوریتم می پردازیم:

* **کامل بودن**: خير ( اگر سطحی ترين هدف در خارج از عمق محدود قرار داشته باشد، اين راهبرد کامل نخواهد بود)
* **بهينگی**: خير ( با توجه به کامل نبودن قطعا بهبینه نیز نخواهد بود)
* **پيچيدگی زمانی**: به صورت نمایی
* **پيچيدگی فضا**: به صورت خطی

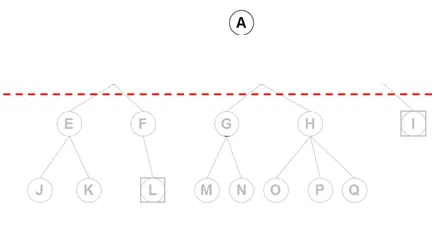
4. الگوریتم IDS :

روش این الگوریتم جستجوی عمیق کننده تکراری می باشد.روش جستجوی با عمق محدود مشکل گرفتاری در عمق نامحدود را حل کرد اما یه مشکل بزرگ داشت. در این روش ما جواب هایی که ممکن بود جواب های بهتری هم باشند اما خارج از محدوده تعیین شده باشند از دست می دادیم. برای برطرف کردن این ایراد باز اصلاحی در ساختار الگوریتم جستجوی عمقی می دهیم تا در عین حال که از مزایای الگوریتم با عمق محدود بهره مند می شویم بتوانیم شانس انتخاب را به جواب های در عمق بیشتر نیز بدهیم.

در این روش مثل جستجوی با عمق محدود ، جستجو را تا یک عمق مشخصی انجام می دهیم ، در مرحله بعد محدوده عمق را بیشتر و دوباره جستجوی عمقی را روی آن اجرا می کنیم. این کار را با افزایش عمق جستجو می توانیم بارها و بارها انجام دهیم تا زمانی که به جواب مطلوب برسیم یا اینکه کل درخت پیمایش شود.

در مثال بالایی روش جستجوی عمیق کننده تکراری را اجرا می کنیم. محدوده عمق جستجو در هر مرحله با خط چین قرمز نمایش داده می شود.

**مرحله 1**

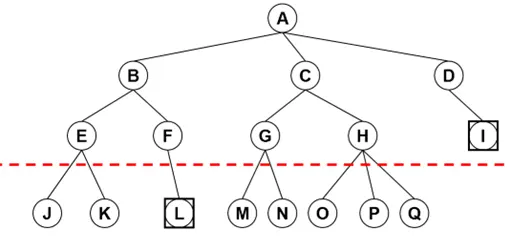


**مرحله 2**

A diagram of a tree

Description automatically generated with low confidence

**مرحله 3**

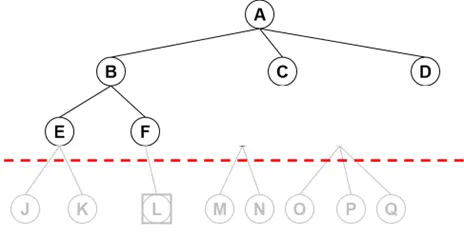


**مرحله 4**

A diagram of a tree

Description automatically generated with low confidence

**مرحله 5**



**مرحله 6**

A diagram of a tree

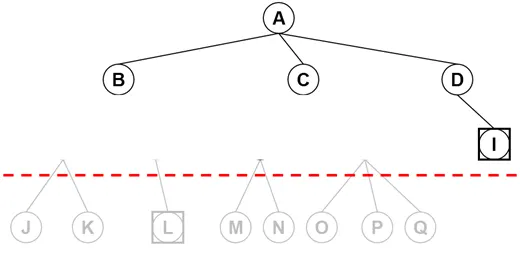
Description automatically generated with low confidence

**مرحله 7**

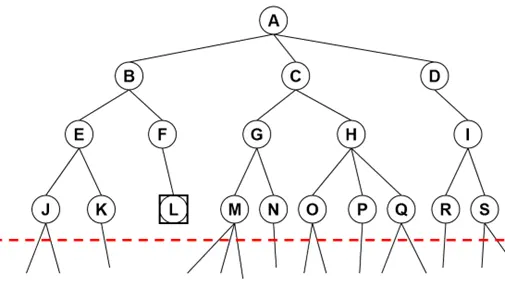
A diagram of a tree

Description automatically generated with low confidence

**مرحله 8**



**مرحله 9**



در مراحل بعدی نیز همان کارهایی که از مراحل 1 تا نه انجام شد تکرار خواهد شد، در اولین نگاه در کنار همه مزایای این روش یک مشکل عمده به چشم می خورد و آن تکرار چندین باره یک عمل و بست تکراری گره ها در هر بار افزایش مقدار عمق می باشد. دوباره کاری عمده مشکل این الگوریتم می باشد. حالا به بررسی کارایی این الگوریتم می پردازیم.

* **کامل بودن**: بله ( در صورتی که فاکتور انشعاب محدود باشد)
* **بهينگی**: بله ( وقتی که هزينه مسير، تابعی غير نزولی از عمق گره باشد )
* **پيچيدگی زمانی**: به صورت نمایی
* **پيچيدگی فضا**: به صورت خطی

پس از آشنایی با روش کار الگوریتم های مورد نظر ، اقدام به کد نویسی هر یک از این الگوریتم ها برای حل 8 Puzzle می نماییم.

ابتدا ما حالت هدف و مقادیر ثابت ردیف و ستون و حداقل و حداکثر ابعاد را تعریف می کنیم:

# Travis Chamness  
# BFS Puzzle Solution by Graph Mode  
# Sept 20th 2021  
#TODO - implement to allow user to add their own Goal.  
GOAL = [[0, 1, 2], [3, 4, 5], [6, 7, 8]]  
ROW = 0  
COL = 1  
MIN\_DIMENSION = 0  
MAX\_DIMENSION = 2

سپس کلاسی تحت عنوان گره یا Node ایجاد می کنیم تا به وسیله ی آن گره های درخت پیمایش را ایجاد کنیم:

#Taken from Lab 3  
class Node:

در ابتدای تعریف هر کلاس در زبان پایتون معمولاً تابع \_\_init\_\_ را ایجاد می کنند که بیانگر شرایط و صفات اوّلیّه ی کلاس می باشد و می تواند حکم سازنده ی کلاس (Constructor) را داشته باشد:

def \_\_init\_\_(self, state, start, parent, movement):  
 self.parent = parent  
 self.location = start  
 self.neighbors = [] # Neighbors from this current state of the graph  
 self.state = state # Puzzle at this current state  
 self.movement = movement # Movement in the puzzle resulting in the state

در ادامه ی بخش بالا برای الگوریتم های DLS و IDS از صفت زیر برای انتساب عمق جستجو استفاده می شود:

self.depth = depth # Current depth of the branch

همانطور که مشاهده می کنید ، کلیه ی متغیرهای ورودی این تابع به متغیر self آن نسبت داده شدند و مقدار این متغیرها به صفات اصلی این کلاس تبدیل شدند.

در ادامه تابع مقایسه ی وضعیت تعریف می شود که در آن وضعیت دو گره متفاوت بررسی و مقایسه می شود:

#utility for comparing nodes  
 def compare\_state(self, state):  
 same = True  
 for i, row in enumerate(self.state):  
 if same:  
 for j,val in enumerate(row):  
 if val != state[i][j]:  
 same = False  
 break  
 else:  
 break  
 return same

در این بخش تابع بررسی هدف بودن وضعیت حاضر تعریف می شود:

#Tests current state against the GOAL state  
 def goal\_test(self):  
 return self.compare\_state(GOAL)

در این بخش تابع کپی گره تعریف می شود که وظیفه ی آن ایجاد یک گره فرزند جدید با استفاده از مشخصات قبلی ، مانند وضعیت ، موقعیت ، والد و حرکت می باشد:

def copy(self):  
 newNode = Node(self.state, self.location, self.parent, self.movement)  
 return newNode

در این بخش ما تابع مقایسه را تعریف می کنیم که کار آن بررسی برابری دو گره است و مباحث حرکت و والد گره را لحاظ نمی کند:

# Compares two nodes for whether or not they are the same  
 # Does not consider the movement, because a movement may not have happened yet, and certainly should not happen again.'  
 # Also will not consider parent because the parent is irrelevant to the same state being found  
 def compare(self, o):  
 if o == None:  
 return False  
 elif o is self:  
 return True  
 else:  
 return self.location == o.location and self.compare\_state(o.state)

در این بخش ما تابع بررسی قرار داشتن گره در بسته مورد نظر در مراحل مختلف را تعریف می کنیم:

def not\_in\_closed(currentNode, closed):  
 in\_closed = False  
 for node in closed:  
 if in\_closed:  
 break  
 else:  
 in\_closed = currentNode.compare(node)  
 return in\_closed

در این بخش تابع خواندن و دریافت پازل مورد نظر کاربر تعریف می شود:

#Reads puzzle from file or user  
def create\_puzzle():  
 #For user specified puzzle  
 # maze\_name = input("Enter puzzle name(Example - puzzle.txt): ")  
 #For Hardcoded puzzle use  
 # maze\_name = "puzzle1.txt"  
 # maze\_name = "puzzle2.txt"  
 maze\_name = "puzzle3.txt"  
 # maze\_name = "puzzle4BFS.txt"  
 file = open(maze\_name, "r")  
 lines = file.readlines()  
 puzzle = []  
 for line in lines:  
 arr = []  
 for character in line:  
 if character != '\n':  
 arr.append(int(character))  
 else:  
 break  
 puzzle.append(arr)  
 return puzzle

در این بخش پازل مورد نظر بررسی و خانه ی خالی آن برای شروع حرکت و حلّ مشخّص می شود:

#prints the puzzle and identifies the starting position.  
def print\_puzzle\_id\_start(puzzle, find\_start = False):  
 for i, row in enumerate(puzzle): # Technical Row of the puzzle which is enumerated with identifier i  
 print()  
 for j, val in enumerate(row): # Technical Col of the puzzle which is enumerated with identifier j  
 if val != '\n':  
 if val == 0:  
 print(" ", end=' ') #IDE says branch never entered?  
 start = [i,j] # Utilize the Row Column shape of the puzzle to describe the starting location with i,j  
 else:  
 print(str(val), end=' ')  
 print('\n')  
 # Optionally return the start location, defaults as off  
 if find\_start:  
 return start  
 else:  
 return ''

در این بخش تابع پسین حل این پازل ایجاد می شود که شامل موراد BFS ، DFS ، DLS و IDS می شود:

1. BFS:  
  
def bfs\_successor\_func(currentNode):  
 #UP  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[ROW] != MIN\_DIMENSION:  
 temp = copyState[copyLocation[ROW] - 1][copyLocation[COL]]  
 copyState[copyLocation[ROW]-1][copyLocation[COL]] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[ROW] = copyLocation[ROW] - 1  
 newNode = Node(copyState, copyLocation, currentNode, 'U')  
 currentNode.neighbors.append(newNode)  
 #Left  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[COL] != MIN\_DIMENSION:  
 temp = copyState[copyLocation[ROW]][copyLocation[COL] - 1]  
 copyState[copyLocation[ROW]][copyLocation[COL] - 1] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[COL]= copyLocation[COL] - 1  
 newNode = Node(copyState, copyLocation, currentNode, 'L')  
 currentNode.neighbors.append(newNode)  
 #Down  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[ROW] != MAX\_DIMENSION:  
 temp = copyState[copyLocation[ROW] + 1][copyLocation[COL]]  
 copyState[copyLocation[ROW]+1][copyLocation[COL]] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[ROW] += 1  
 newNode = Node(copyState, copyLocation, currentNode, 'D')  
 currentNode.neighbors.append(newNode)  
 #Right  
 if copyLocation[COL] != MAX\_DIMENSION:  
 temp = copyState[copyLocation[ROW]][copyLocation[COL]+1]  
 copyState[copyLocation[ROW]][copyLocation[COL]+1] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[COL] += 1  
 newNode = Node(copyState, copyLocation, currentNode, 'R')  
 currentNode.neighbors.append(newNode)  
  
 return currentNode

2 , 3 , 4. DFS and DLS and IDS:

def dfs\_successor\_func(currentNode):  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 #Order reversal of Operation that mimics the stack for U, L, D, R order  
 #Right  
 if copyLocation[COL] != MAX\_DIMENSION:  
 temp = copyState[copyLocation[ROW]][copyLocation[COL]+1]  
 copyState[copyLocation[ROW]][copyLocation[COL]+1] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[COL] += 1  
 newNode = Node(copyState, copyLocation, currentNode, 'R')  
 currentNode.neighbors.append(newNode)  
 #Down  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[ROW] != MAX\_DIMENSION:  
 temp = copyState[copyLocation[ROW] + 1][copyLocation[COL]]  
 copyState[copyLocation[ROW]+1][copyLocation[COL]] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[ROW] += 1  
 newNode = Node(copyState, copyLocation, currentNode, 'D')  
 currentNode.neighbors.append(newNode)  
 #Left  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[COL] != MIN\_DIMENSION:  
 temp = copyState[copyLocation[ROW]][copyLocation[COL] - 1]  
 copyState[copyLocation[ROW]][copyLocation[COL] - 1] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[COL]= copyLocation[COL] - 1  
 newNode = Node(copyState, copyLocation, currentNode, 'L')  
 currentNode.neighbors.append(newNode)  
 #UP  
 copyLocation = currentNode.location.copy()  
 copyState = [row[:] for row in currentNode.state]  
 if copyLocation[ROW] != MIN\_DIMENSION:  
 temp = copyState[copyLocation[ROW] - 1][copyLocation[COL]]  
 copyState[copyLocation[ROW]-1][copyLocation[COL]] = copyState[copyLocation[ROW]][copyLocation[COL]]  
 copyState[copyLocation[ROW]][copyLocation[COL]] = temp  
 copyLocation[ROW] = copyLocation[ROW] - 1  
 newNode = Node(copyState, copyLocation, currentNode, 'U')  
 currentNode.neighbors.append(newNode)  
 return currentNode

همان گونه که مشاهده می کنید ، در روش BFS ابتدا جهت های بالا ، چپ ، پایین و راست به ترتیب بررسی می شوند و در سایر روش ها (DFS , DLS , IDS) ابتدا جهت های راست ، پایین ، چپ و بالا بررسی می شوند.

در این بخش تابع مسیر پر کردن پازل با گره تعریف می شود:

def populate\_path(node):  
 path = []  
 currentNode = node.copy()  
 while currentNode != None:  
 path.insert(0, currentNode.movement)  
 currentNode = currentNode.parent  
 return path

در این بخش تابع اضافه کردن گره های جاری به حاشیه تعریف می شود:

def append\_to\_fringe(fringe, currentNode):  
 for node in currentNode.neighbors:  
 fringe.append(node.copy())  
 return fringe

در این بخش ما تابع حل کننده ی پازل را برای روش های مختلف تعریف می کنیم:

1. BFS:

def bfs\_solution(puzzle):  
 find\_start = True  
 goalFound = False  
 start = print\_puzzle\_id\_start(puzzle, find\_start)  
 path = []  
 closed = []  
 currentNode = None  
 head = Node(puzzle, start, None, None)  
 fringe = [head]  
 iteration = 0  
 while not goalFound and fringe:  
 iteration += 1  
 in\_closed = False  
 #Dequeue the 0th place node from the fringe  
 currentNode = fringe.pop(0)  
 #Test the current node state against the goal state  
 goalFound = currentNode.goal\_test()  
 print(print\_puzzle\_id\_start(currentNode.state))  
 #Determine if the current node is in the closed set  
 in\_closed = not\_in\_closed(currentNode, closed)  
 if not goalFound and not in\_closed:  
 closed.append(currentNode)  
 currentNode = bfs\_successor\_func(currentNode)  
 fringe = append\_to\_fringe(fringe, currentNode)  
 elif goalFound:  
 path = populate\_path(currentNode)  
 print(iteration)  
 print(path)  
 return path, len(path) – 1

2. DFS:

def dfs\_solution(puzzle):  
 find\_start = True  
 goalFound = False  
 start = print\_puzzle\_id\_start(puzzle, find\_start)  
 print("Start Location: ",start)  
 path = []  
 closed = []  
 head = Node(puzzle, start, None, None) #initialized head on Fringe  
 currentNode = None  
 fringe = [head]  
 while not goalFound and fringe:  
 in\_closed = False  
 currentNode = fringe.pop()  
 goalFound = currentNode.goal\_test()  
 print(print\_puzzle\_id\_start(currentNode.state))  
 in\_closed = not\_in\_closed(currentNode, closed)  
 if not goalFound and not in\_closed:  
 closed.append(currentNode)  
 currentNode = dfs\_successor\_func(currentNode)  
 fringe = append\_to\_fringe(fringe, currentNode)  
 elif goalFound:  
 path = populate\_path(currentNode)  
 #append to fringe  
 print(path)  
 return path, len(path) - 1

3 , 4. DLS and IDS:

def dls\_solution(puzzle, limit):  
 find\_start = True  
 goalFound = False  
 start = print\_puzzle\_id\_start(puzzle, find\_start)  
 print("\nStart Location:",start)  
 path = []  
 closed = []  
 head = Node(puzzle, start, None, None, 0) #initialized head on Fringe  
 currentNode = None  
 fringe = [head]  
 while not goalFound and fringe:  
 in\_closed = False  
 currentNode = fringe.pop()  
 goalFound = currentNode.goal\_test()  
 print("{} \nNodeDepth: {}".format(print\_puzzle\_id\_start(currentNode.state), currentNode.depth))  
 in\_closed = not\_in\_closed(currentNode, closed)  
 if not goalFound and currentNode.depth < limit and not in\_closed:  
 closed.append(currentNode)  
 currentNode = dfs\_successor\_func(currentNode)  
 fringe = append\_to\_fringe(fringe, currentNode)  
 elif goalFound:  
 path = populate\_path(currentNode)  
 #append to fringe  
 print(path)  
 return path, len(path) - 1

همان گونه که در روش های DLS و IDS مشاهده می کنید ، در این دو روش در هر مرحله میزان عمق پیمایش شده بررسی می شود تا از میزان عمق مشخص شده بیشتر نباشد.

برای روش IDS یک تابع دسته بندی کننده ی جواب های حاصل تعریف می کنیم که این جواب ها با توجّه به قسمت قبلی با استفاده از روش DLS به دست می آیند:

def dls\_solution\_wrapper(depth):  
 path = []  
 for i in range(depth):  
 if not path:  
 iterationDepth = i + 1  
 path = dls\_solution(create\_puzzle(), iterationDepth)  
 else: break  
 return path, len(path) – 1

برای اجرای برنامه از دستور زیر با فراخوانی های مختلف مشخص شده برای هر روش استفاده می کنیم:

if \_\_name\_\_ == '\_\_main\_\_':

1. BFS:  
 bfs\_solution(create\_puzzle())  
  
2. DFS:

dfs\_solution(create\_puzzle())

3 , 4. DLS and IDS:

dls\_solution(create\_puzzle(), 3)