گراف بدون جهت
UNDIRECTED GRAPH

فرشته دهقانی

سرفصل مطالب

- ✓گراف و اهمیت مطالعه گراف
- ✓اصطلاحات گراف
- ✓نحوه نمایش گراف
- ✓پیمایش عمقی (dfs) و سطحی (bfs) گراف

گراف

- ✓مجموعه ای از رئوس که به وسیله تعدادی یال به صورت دوبدو به یکدیگر متصل شده اند
- ✓گراف یک شاخه جذاب و چالش برانگیز در علوم کامپیوتر است که هزاران کاربرد عملی و صدها الگوریتم شناخته شده دارد

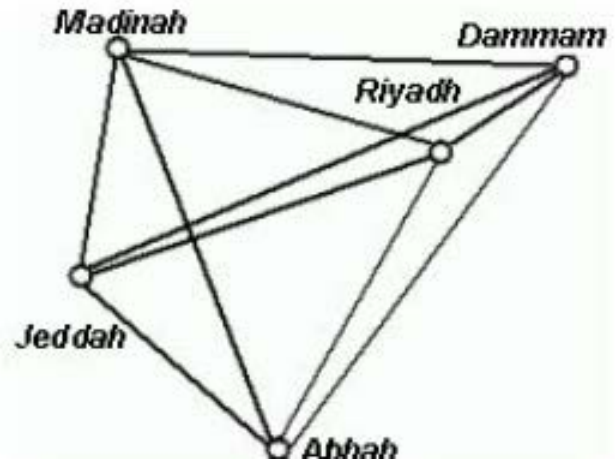
مثالهایی از کاربرد گراف

گراف	رأس	یال
مخابرات	تلفن، کامپیوتر	کابل و فیبر نوری
مدار	گیت، ثبات، پردازنده	سیم
علوم مالی	سهام، ارز	تعاملات
حمل و نقل	تقاطع، فرودگاه، مترو	خیابان، خطوط هوایی و زیرزمینی
اینترنت	شبکه‌های محلی	اتصالات
بازی	وضعیت صفحه	حرکات قانونی
روابط اجتماعی	فرد	دوستی
شبکه‌های عصبی	نورون	سیناپس
شبکه‌ی پروتئین	پروتئین	تعامل بین پروتئین‌ها

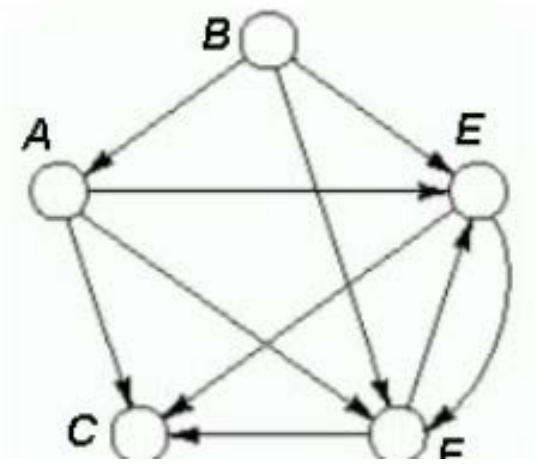
اصطلاحات

- ✓مسیر (path): دنباله ای از رئوی به هم متصل شده به وسیله یال ها
- ✓دو راس به هم متصل (Connected): وجود مسیر بین دو راس
- ✓دور (cycle): مسیر که از یک راس شروع و به همان راس ختم می شود
- ✓انواع گراف: بدون جهت، جهت دار و وزن دار

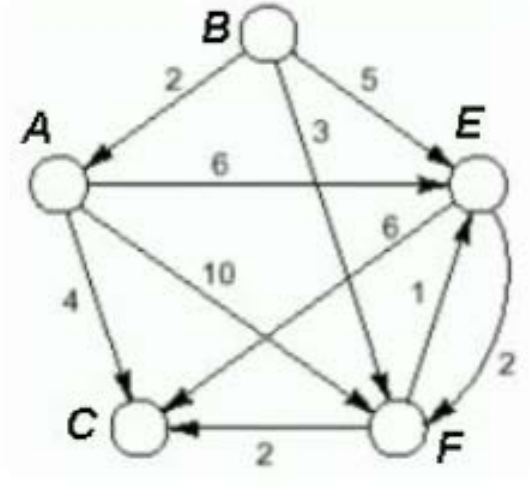
Undirected



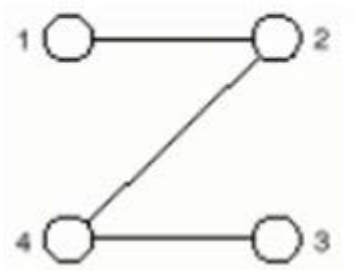
Directed (Digraph)



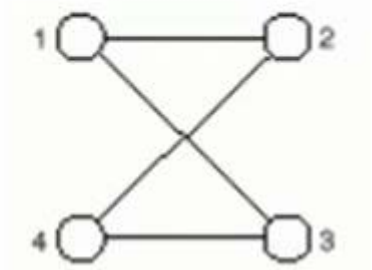
Weighted



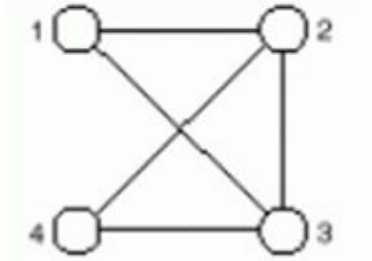
Path



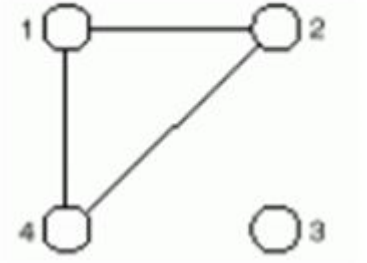
Cycle

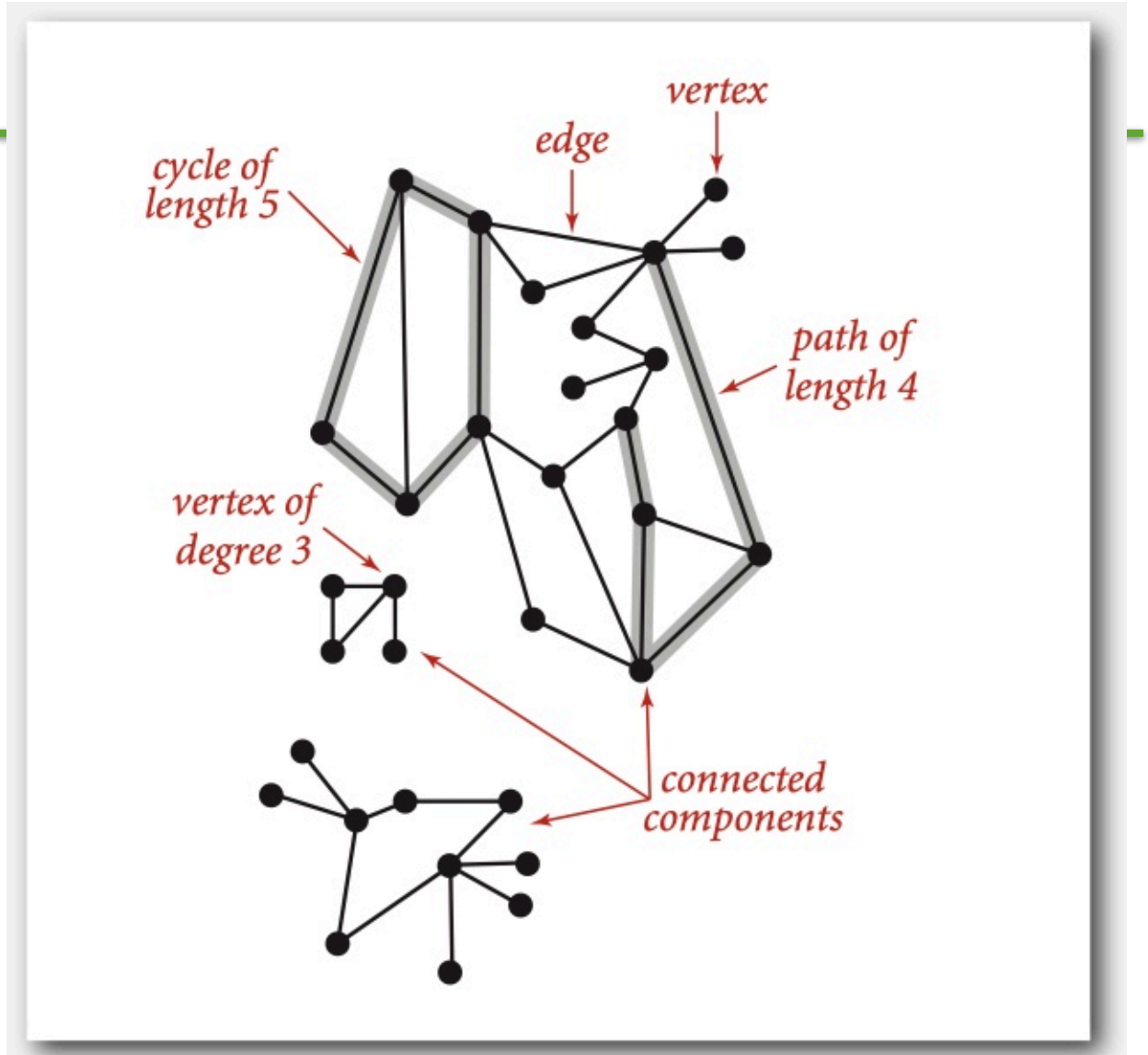


Connected



Disconnected





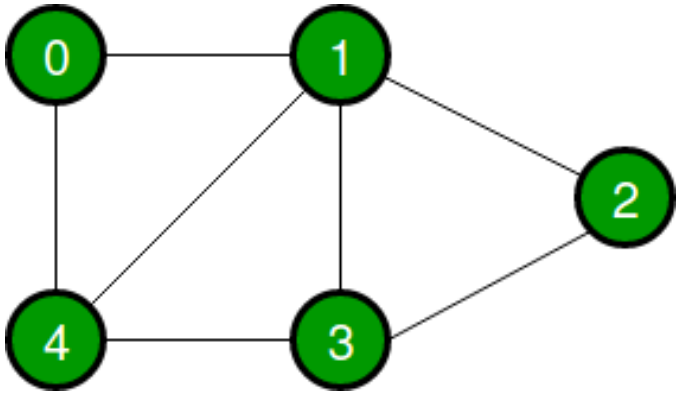
چندتا مسایل معروف گراف

- ✓**مسیر**: آیا بین دو راس s و t، مسیری وجود دارد؟
- ✓**کوتاهترین مسیر:** کوتاهترین مسیر میان s و t کدام است؟
- ✓**دور**: آیا گراف شامل دور است؟
- ✓**تور اویلری**: آیا دوری وجود دارد که از هر یال دقیقا یک بار بگذرد؟
- ✓**تور هامیلتونی**: آیا دوری وجود دارد که از هر راس یک بار بگذرد؟
- ✓**همبندی**: آیا روشی برای متصل کردن همه راس ها وجود دارد؟
- ✓**درخت پوشای کمینه**: کم هزینه ترین روش برای متصل کردن همه رئوس گراف چیست؟
- ✓**دوهمبندی**: با حذف چه راسی، گراف ناهمبند می شود
- ✓**مسطح بودن**: آیا میتوان گراف را به گونه ای رسم کرد که هیچ دو یالی از روی یکدیگر نگذرند
- ✓**یکریختی**: آیا دو گراف داده شده بیانگر یک گراف است؟

نحوه نمایش گراف

- ✓لیست یال (Edge List)
- ✓ماتریس همسایگی (Adjacency Matrix)
- ✓لیست همسایگی (Adjacency List)

لیست یالها



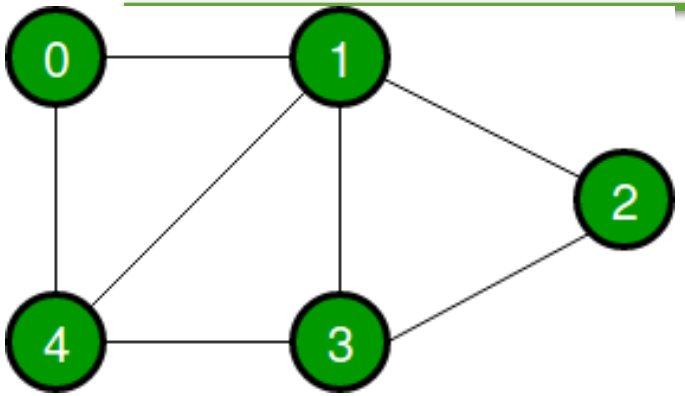
0	1
0	4
1	4
1	3
1	2
2	3
3	4

مزایا و معایب

- ✓اضافه کردن یال راحت تر
- ✓مصرف فضای حافظه : O(|E|)

- ✓عیب: پیدا کردن یال میان دو راس : O(|E|)

ماتریس همسایگی



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

- ماتریس دوبعدی با اندازه V*V (V: تعداد راس های گراف)
- adj[i][j]: نشان دهنده یالی میان راس i به راس J است.
- متقارن در گراف بدون جهت
- مناسب برای نمایش گراف وزن دار (adj[i][j]=w)

مزایا و معایب ماتریس همسایگی

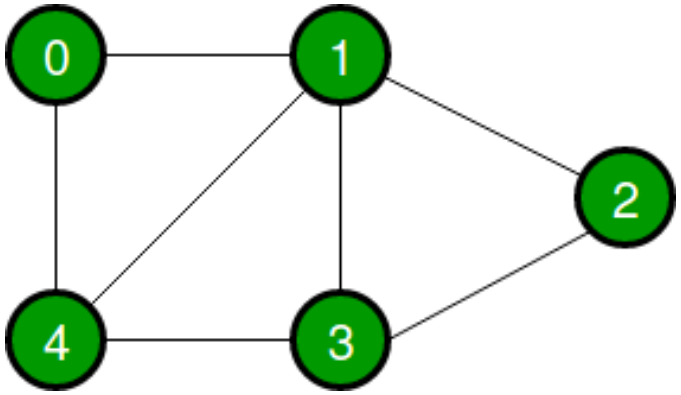
- ✓مزیت: راحت بودن پیاده سازی و دنبال کردن گراف
  - ✓حذف یال و وجود یال میان دو راس: $O(1)$

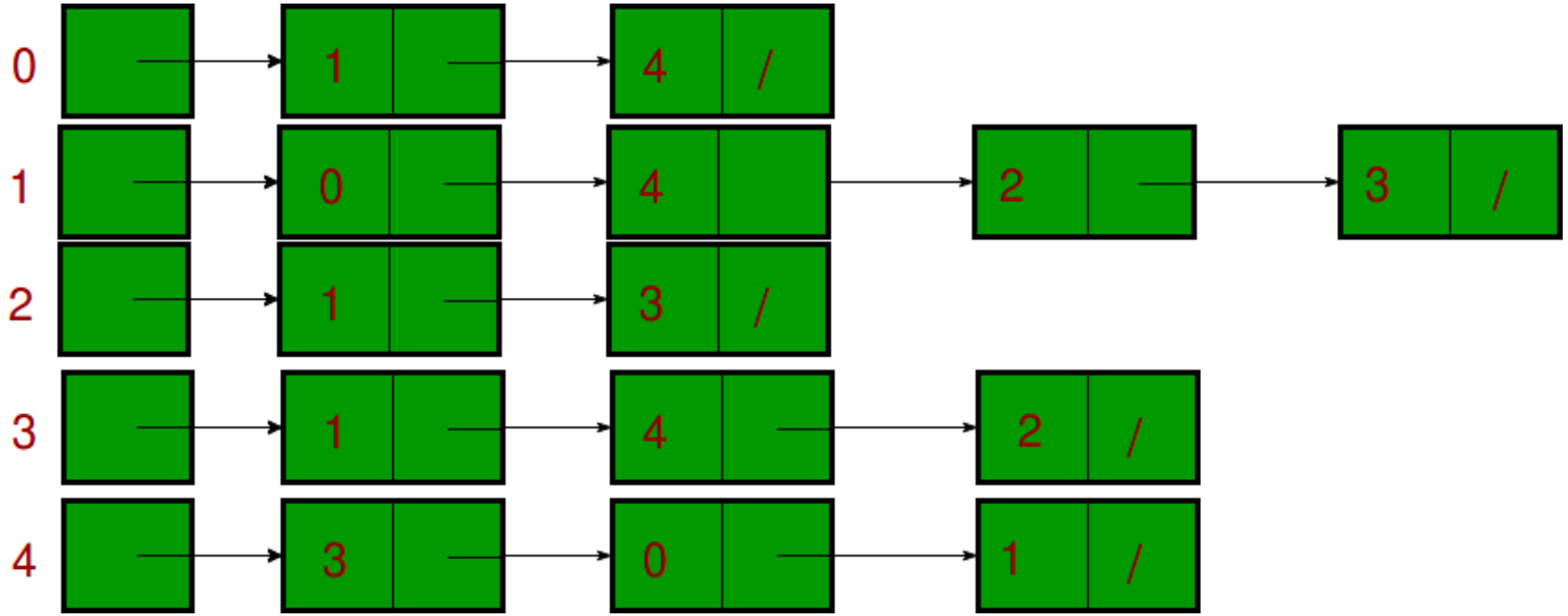
- ✓مصرف حافظه زیاد : $O(V^2)$ حتی با وجود اسپارس بودن گراف (یال کم در گراف)
- ✓اضافه کردن یک راس سخت تر

لیست همسایگی

- آرایه ای (به اندازه تعداد راس ها) از لیست ها
- Array[i]: لیستی از راس های مجاور با گره iام

لیست مجاورت



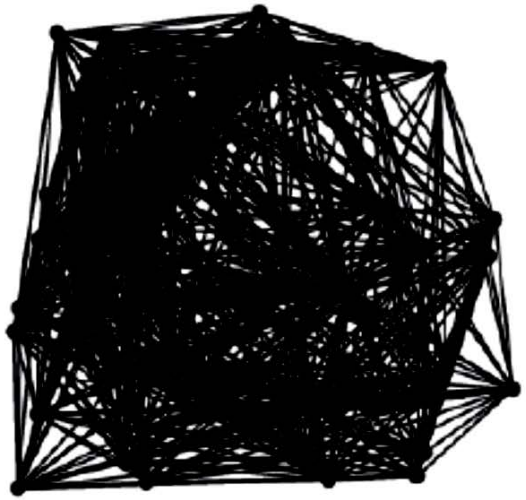


مزایا و معایب

- ✓مصرف حافظه: O(|V|+|E|)
- ✓در بدترین حالت تعداد یالها برابر |V^2| و در نتیجه مصرف حافظه O(V^2)
- ✓اضافه کردن یک راس راحت تر

✓در دنیای واقعی گراف ها بیشتر به سمت گراف خلوت متمایل هستند.

گراف متراکم (E = 1000)



گراف خلوت (E = 200)

دو گراف (V = 50)

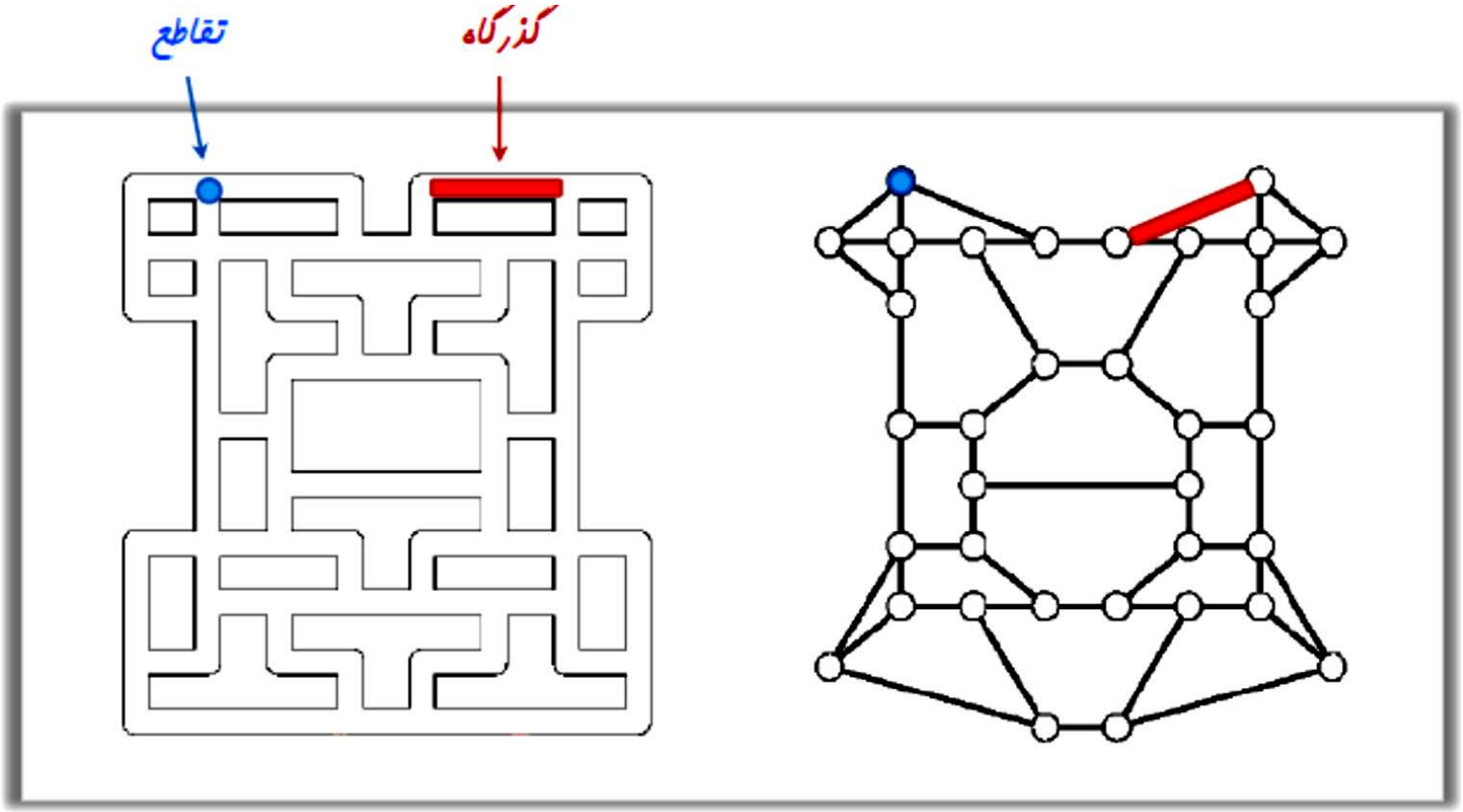
روش نمایش گراف	حافظه	افزودن یال	بررسی وجود یال بین رئوس V و W	حلقه زدن روی رئوس مجاور V
لیست یالها	E	1	E	E
ماتریس همسایگی	V^2	1	1	V
لیست همسایگی	E + V	1	degree(v)	degree(v)

پیمایش و جستجوی گراف

- ✓ پیمایش عمقی
- ✓ پیمایش سطحی

- ✓ نکته: باید از دور در پیمایش اجتناب شود

مثال- مسیر پرپیچ و خم



- ✓راس ها: تقاطع ها
- ✓یال: گذرگاهها

هدف: کاوش تمامی تقاطع‌ها

پیمایش عمقی
DEPTH FIRST SEARCH

- ✓هدف: جستجوی عمقی یک گراف به صورت منظم
- ✓استفاده از پشته

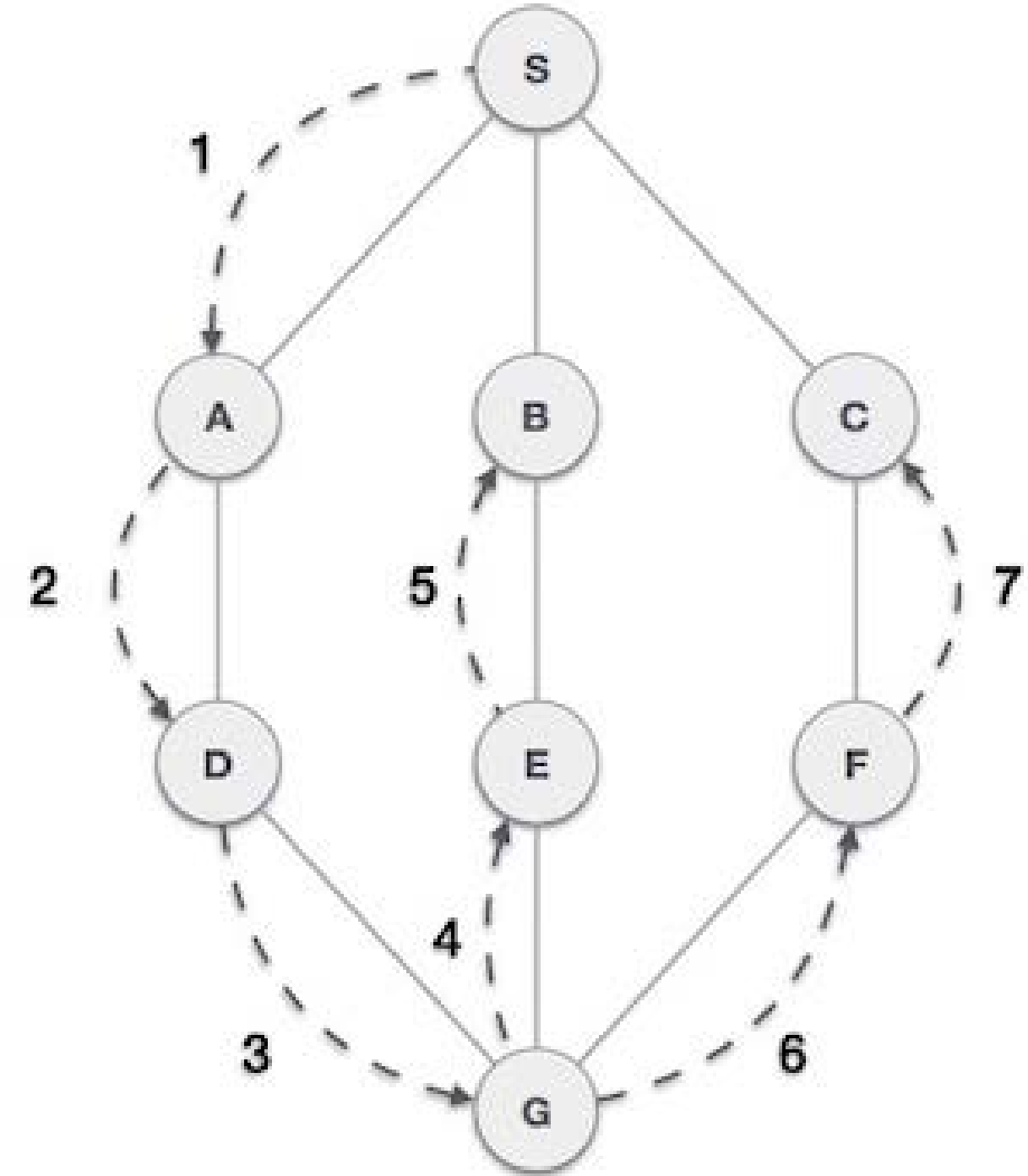
DFS (to visit a vertex v)

Mark v as visited.

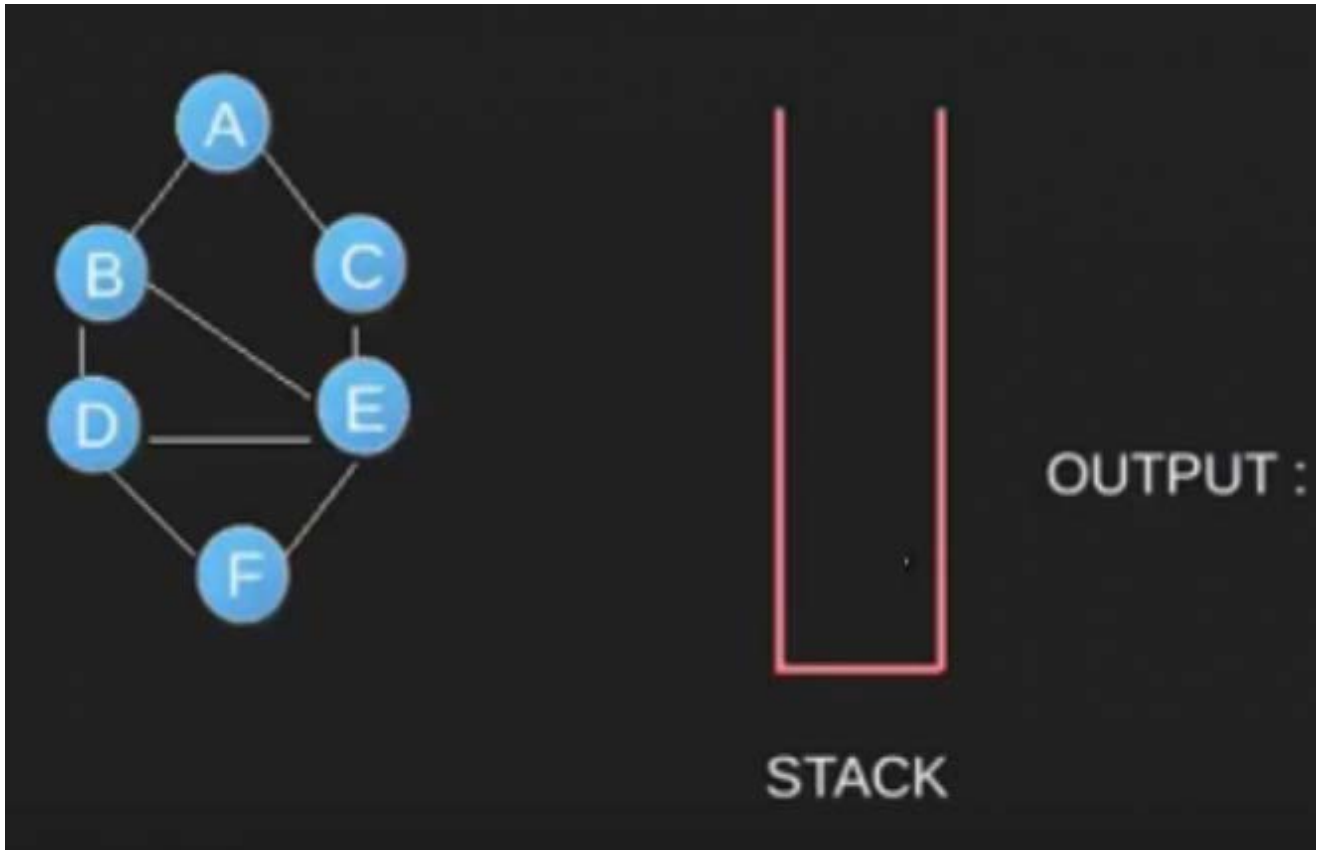
Recursively visit all unmarked vertices w adjacent to v.

روش کار

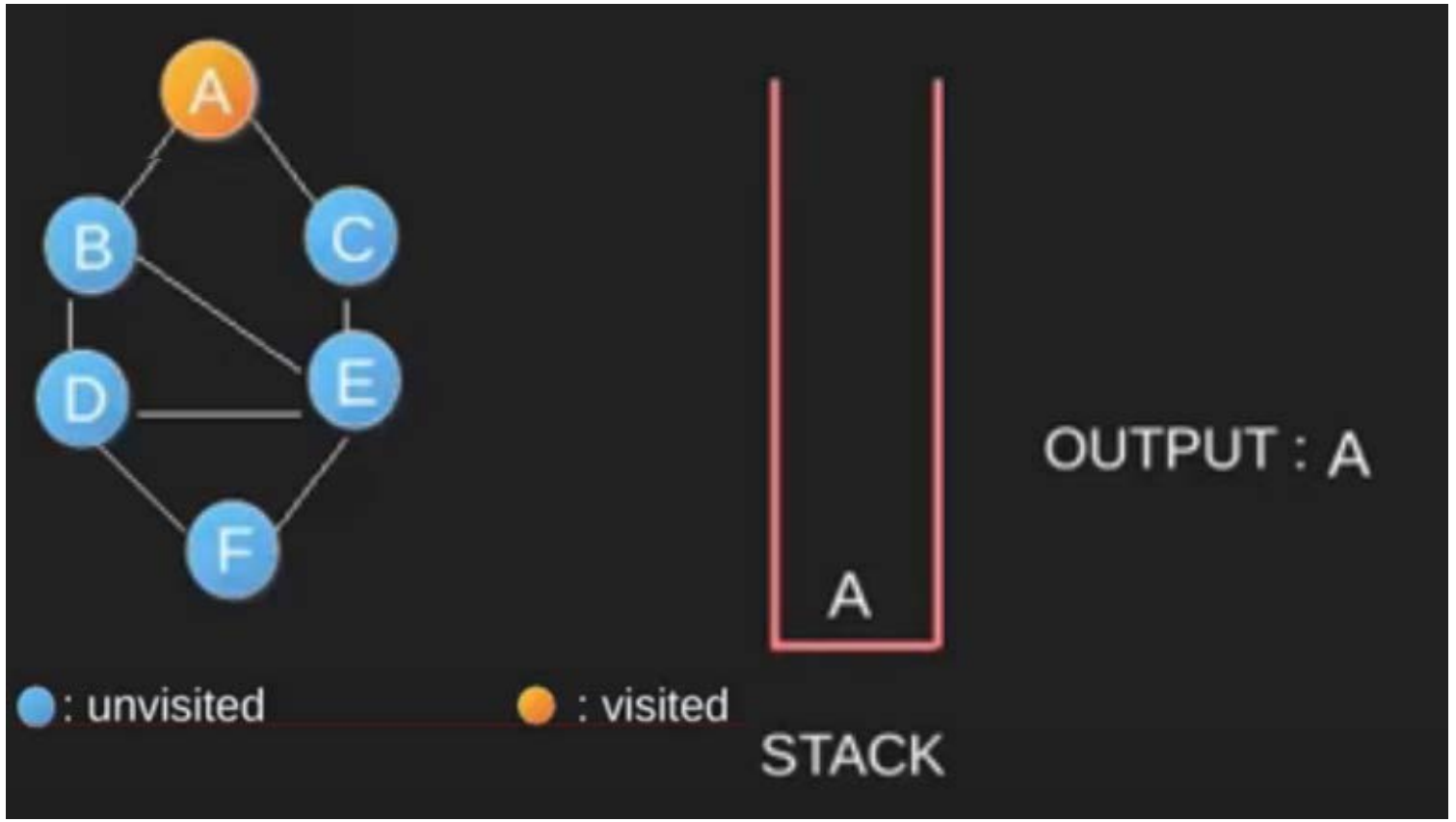
۱- گره جاری را به پشته اضافه کن.
۲- گره جاری را پردازش کن.
۳- از گره‌های مجاور گره جاری یک گره پیمایش نشده را به عنوان گره جاری انتخاب کرده و برو به مرحله‌ی ۱.
۴- اگر همه‌ی گره‌های مجاور گره جاری پیمایش شده‌اند، گره بالای پشته را به عنوان گره جاری از پشته حذف کرده و برو به مرحله‌ی ۳.
۵- اگر گرهی در پشته وجود ندارد، اجرای الگوریتم را متوقف کن.


S
A
B
C
D
E
F
G
1
2
3
4
5
6
7

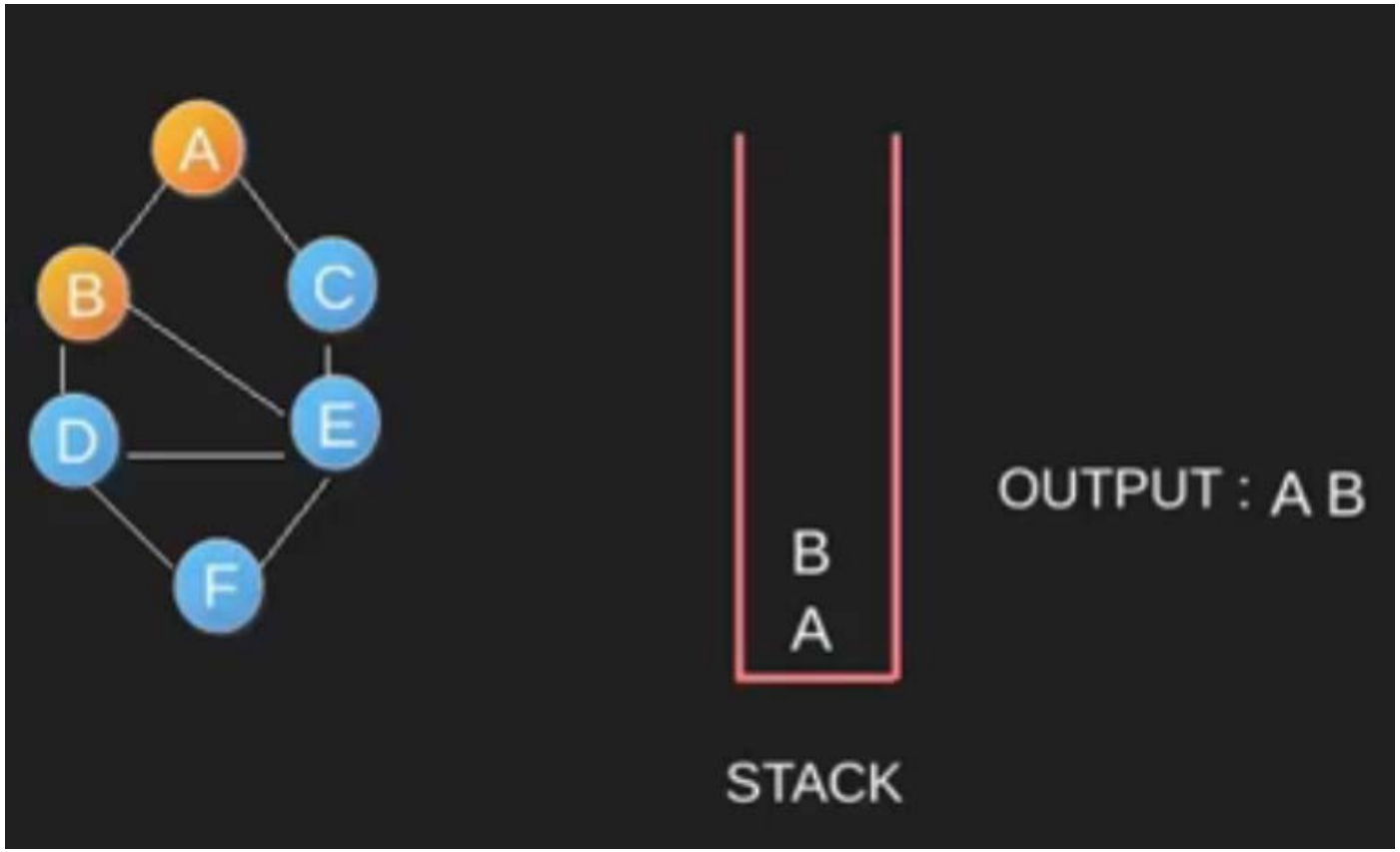
پیمایش عمقی...



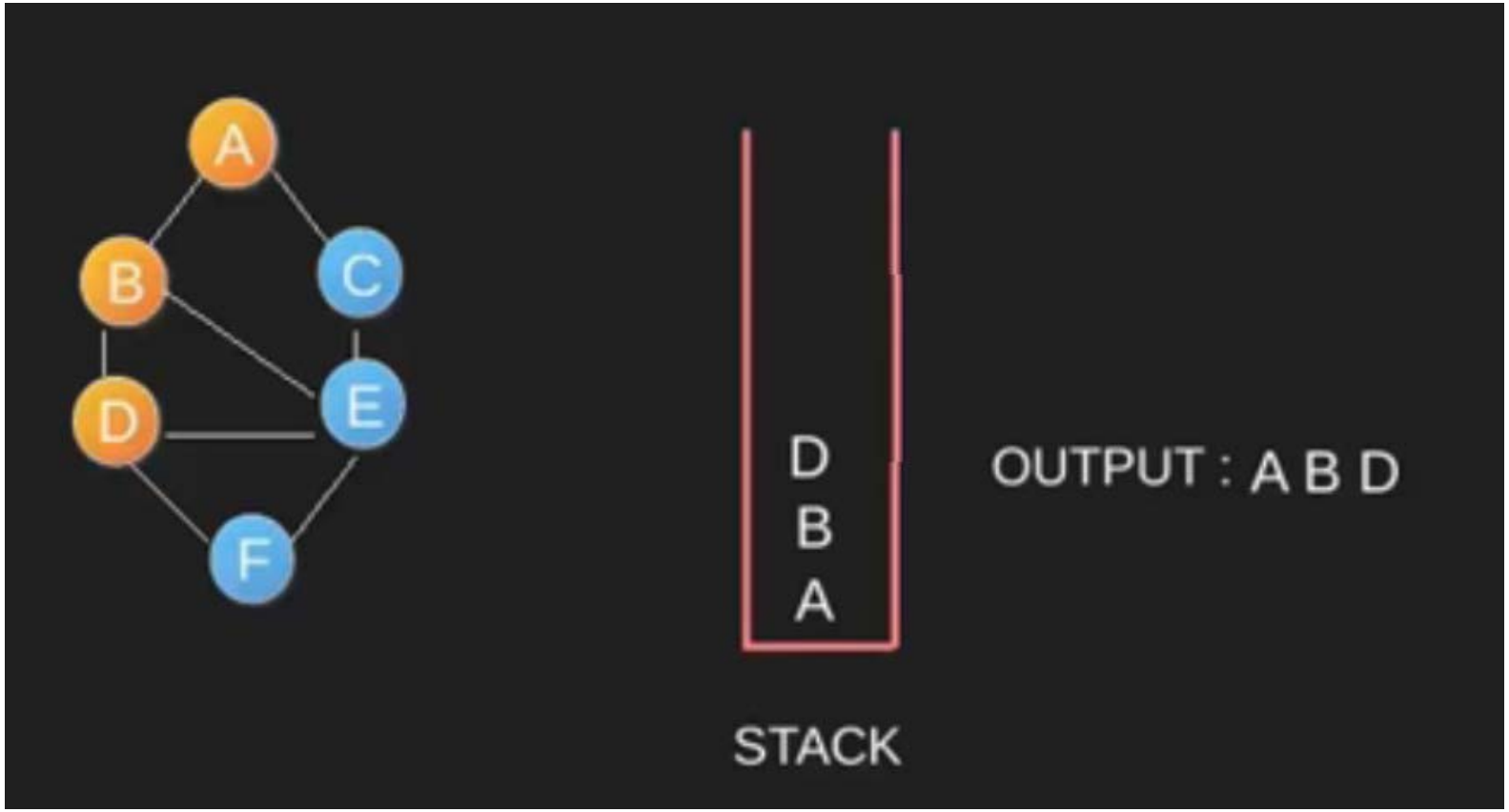
پیمایش عمقی...



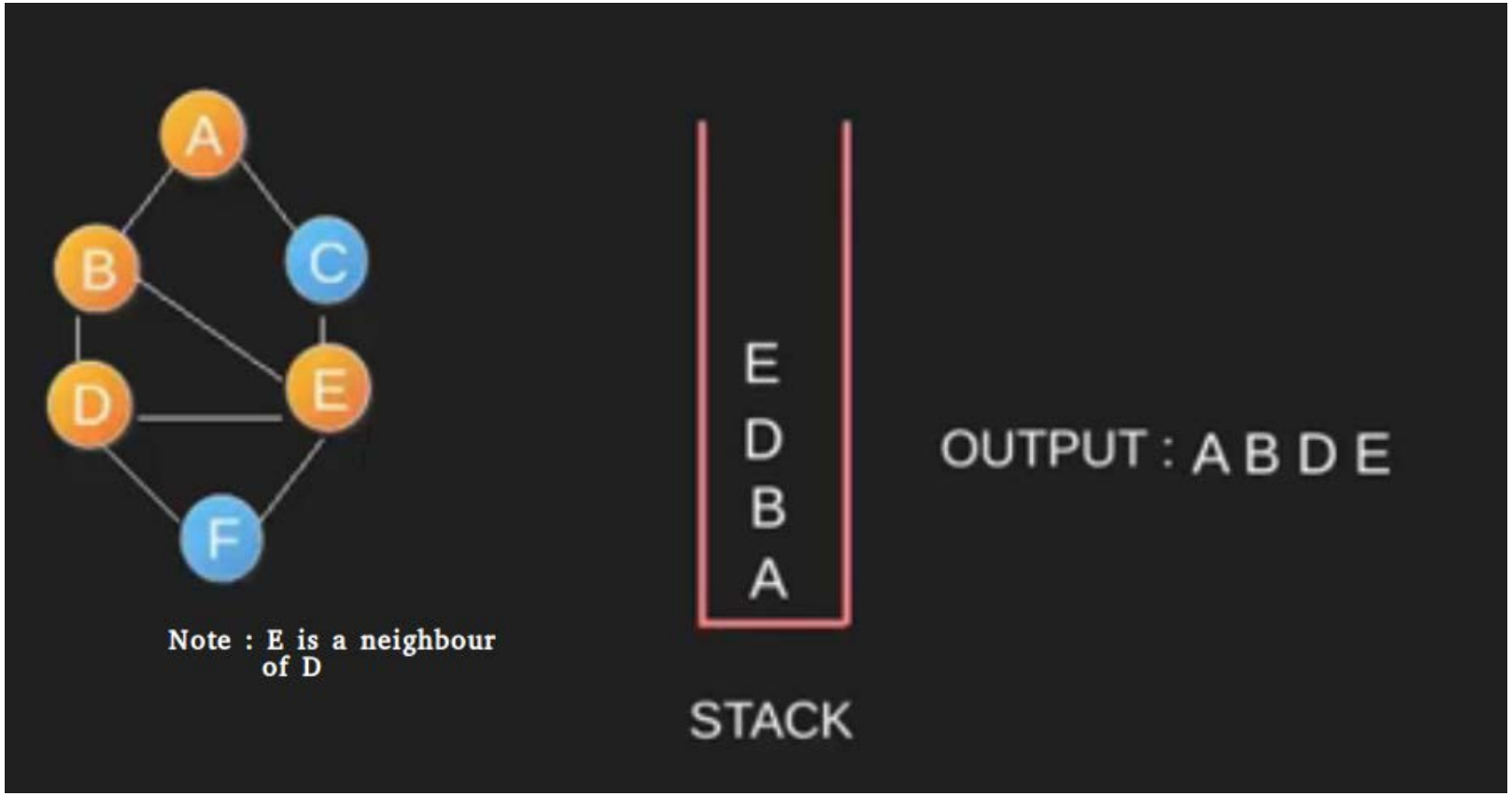
پیمایش عمقی...



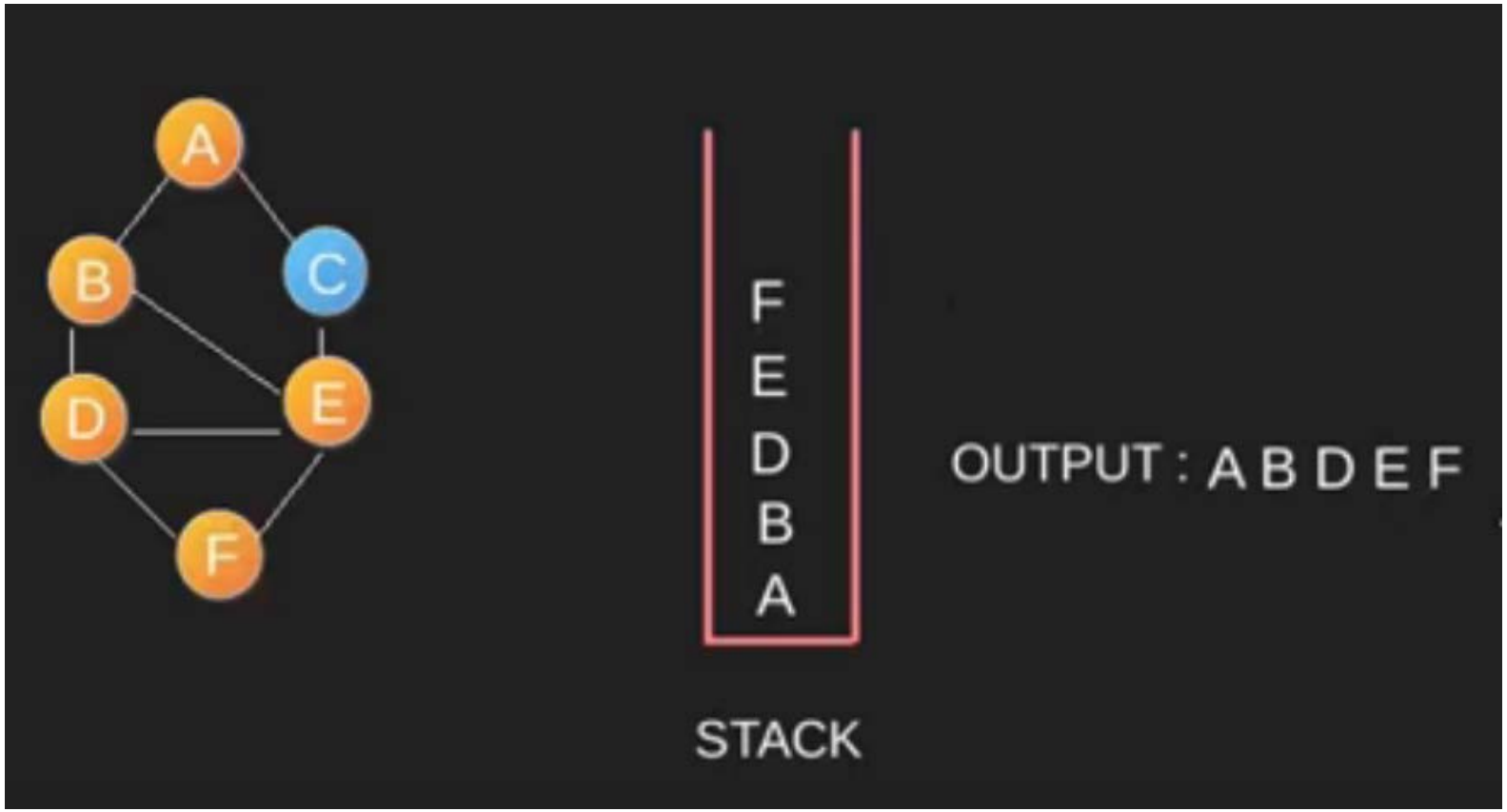
پیمایش عمقی...



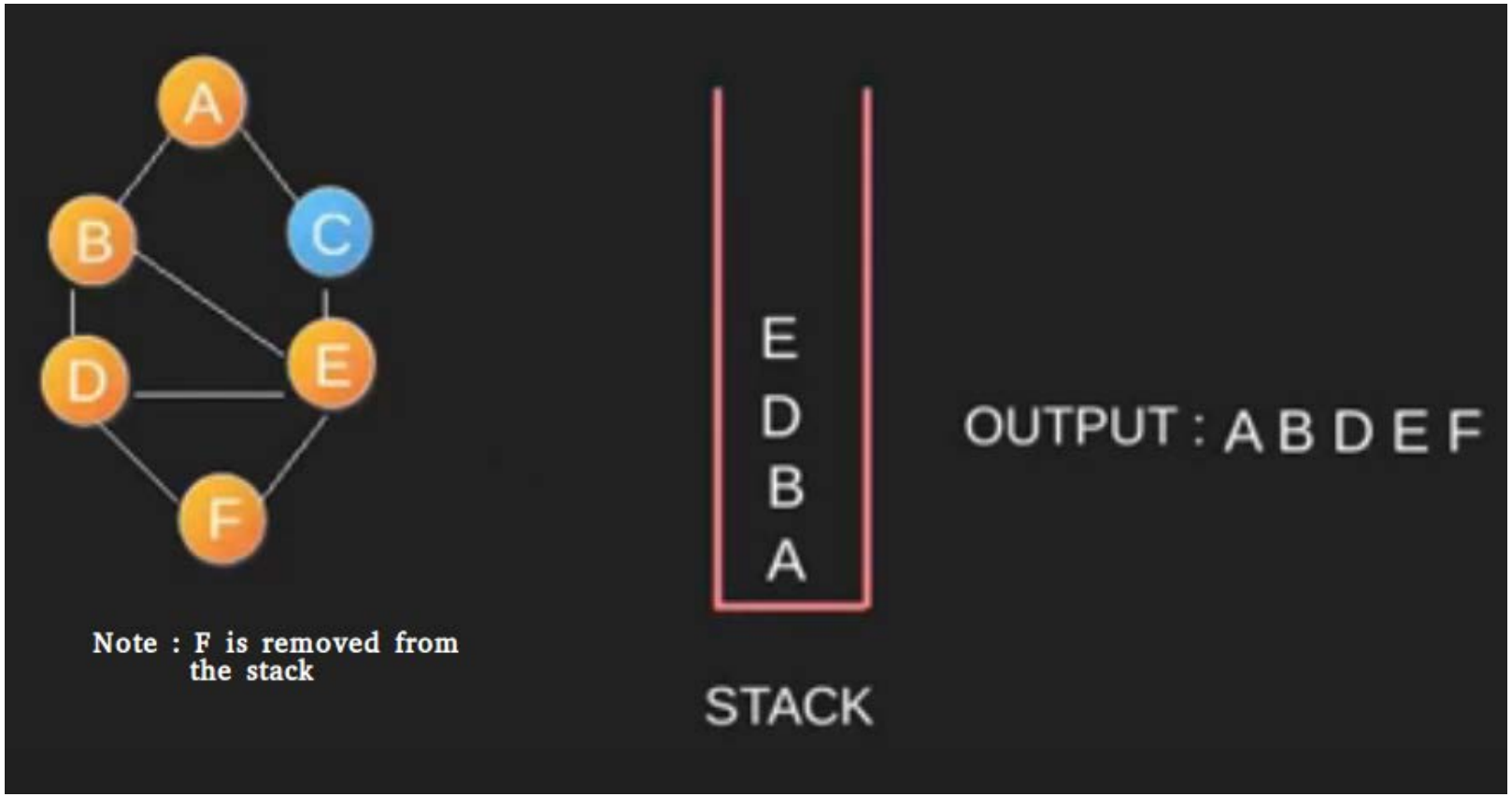
پیمایش عمقی...



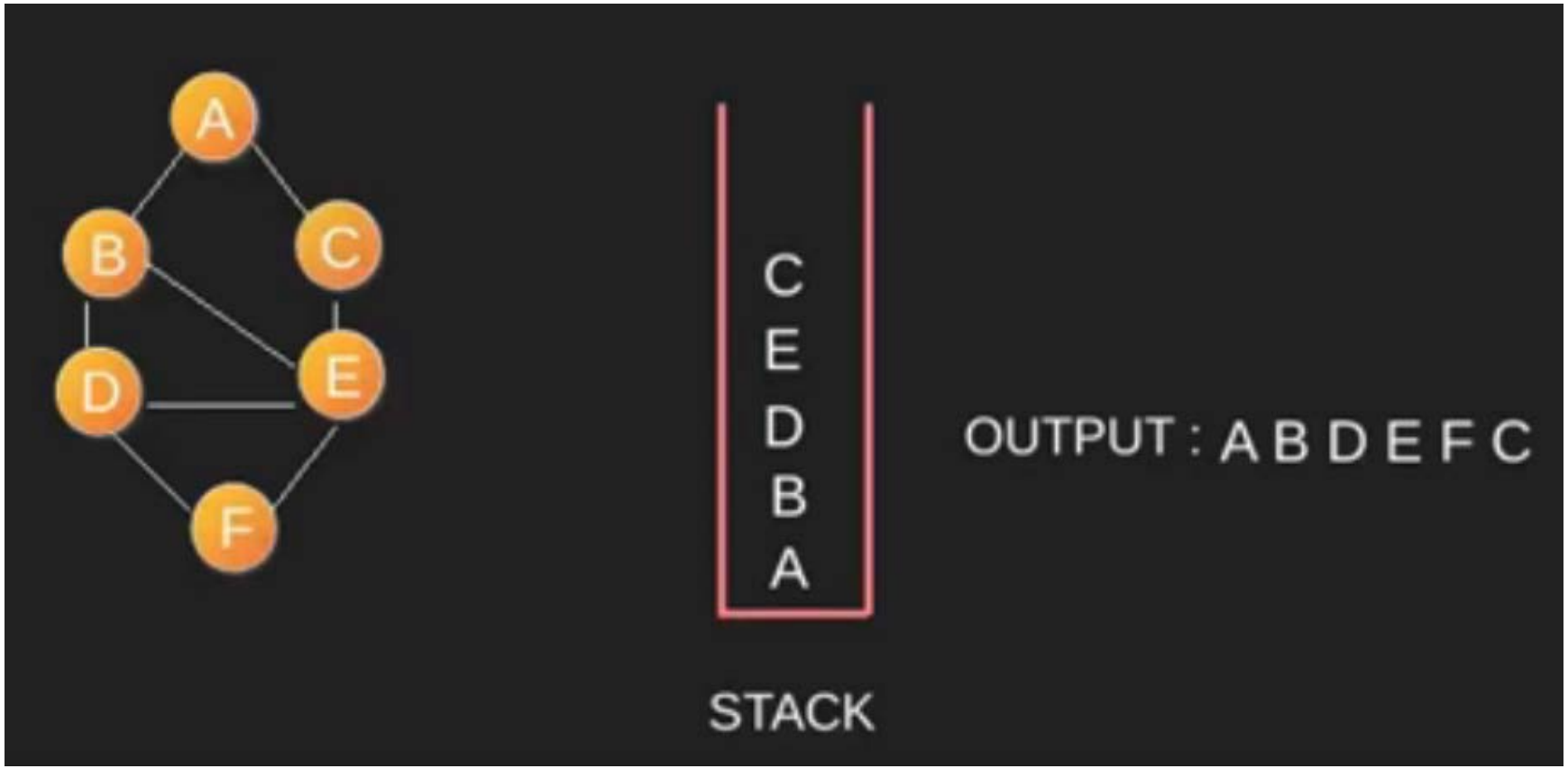
پیمایش عمقی...



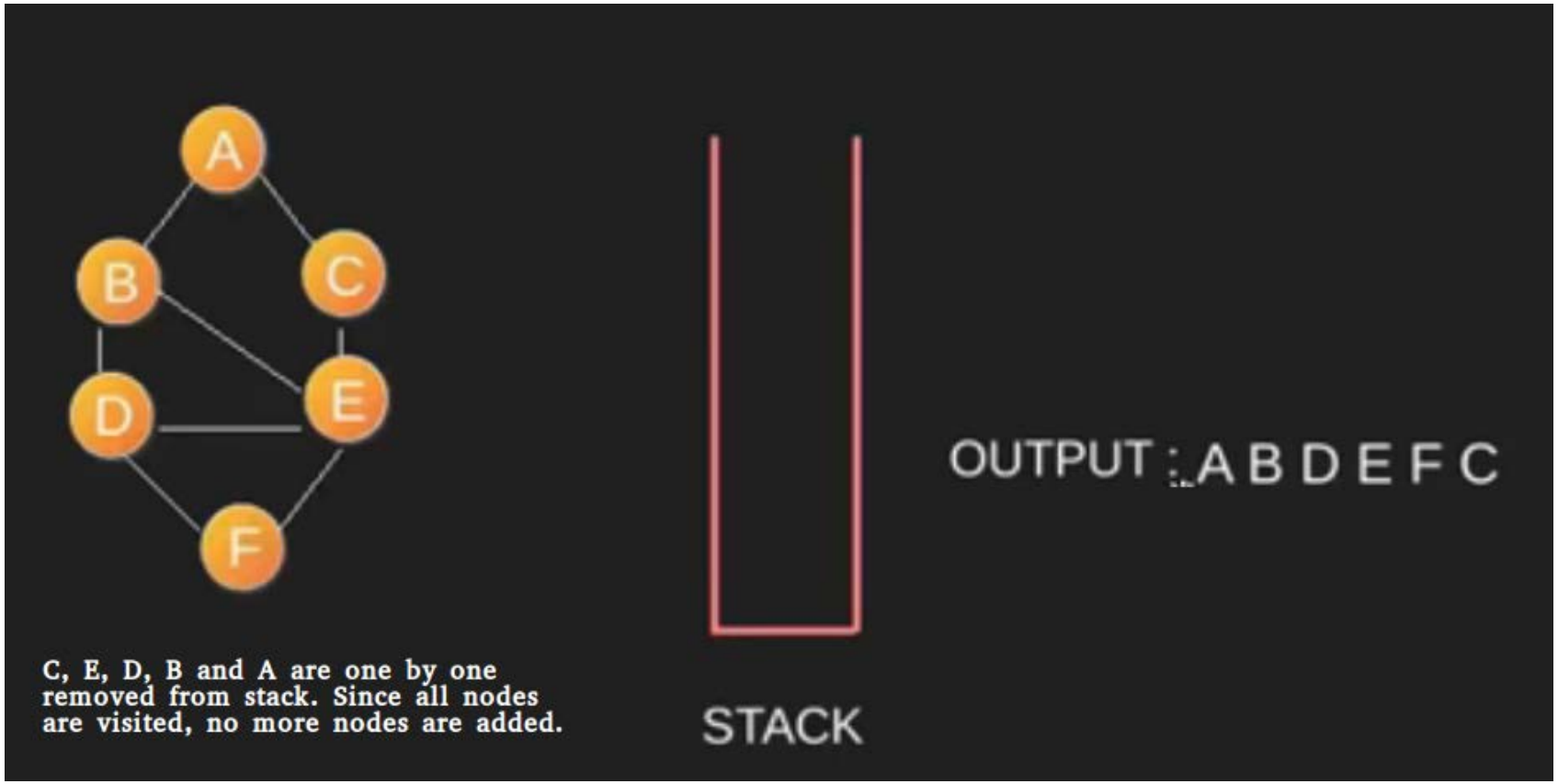
پیمایش عمقی...

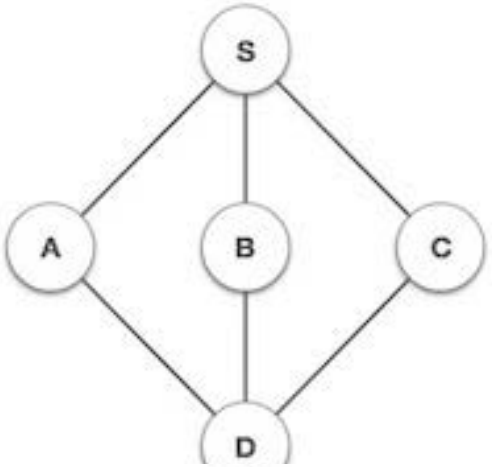

A
B
C
D
E
F
Note : F is removed from the stack
E
D
B
A
STACK
OUTPUT : A B D E F

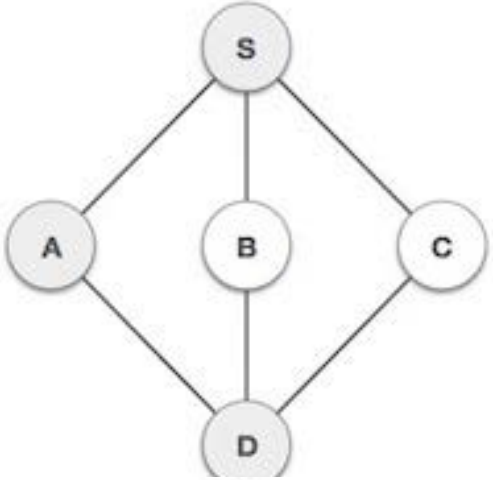
پیمایش عمقی...

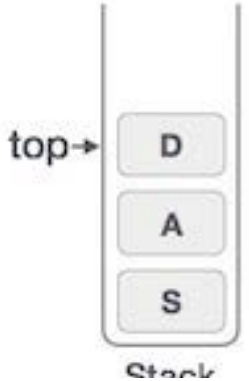


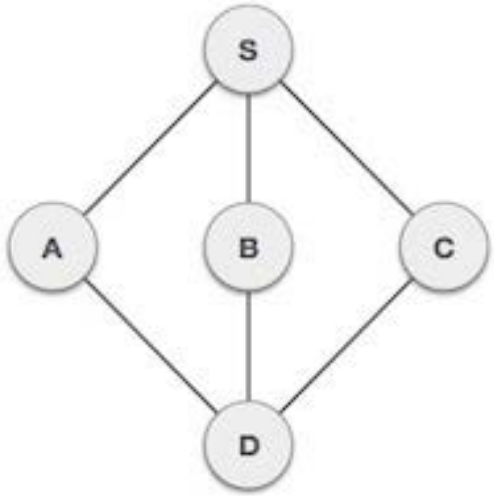
پیمایش عمقی...

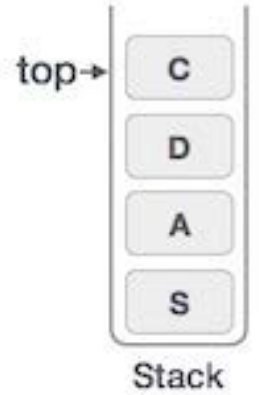

A
B
C
D
E
F
OUTPUT : A B D E F C
STACK
C, E, D, B and A are one by one removed from stack. Since all nodes are visited, no more nodes are added.

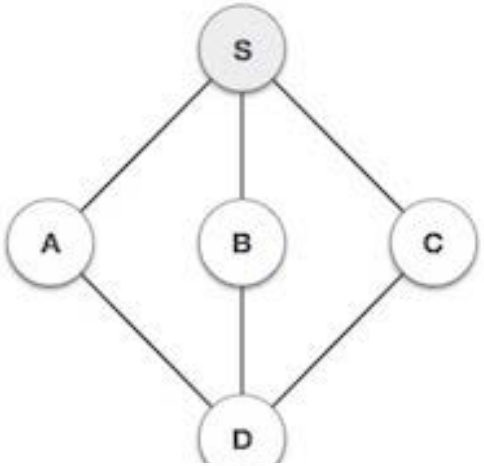

S
A
B
C
D

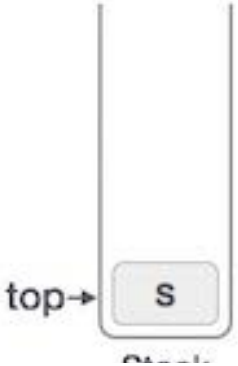

S
A
B
C
D

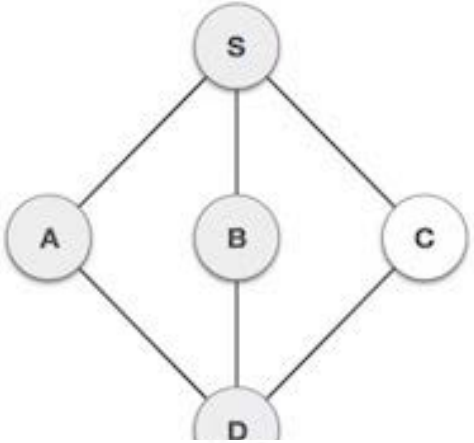

top
D
A
S

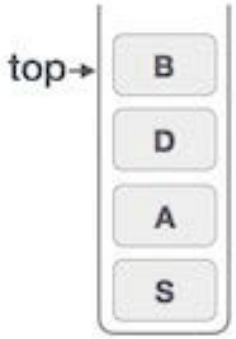

S
A
B
C
D

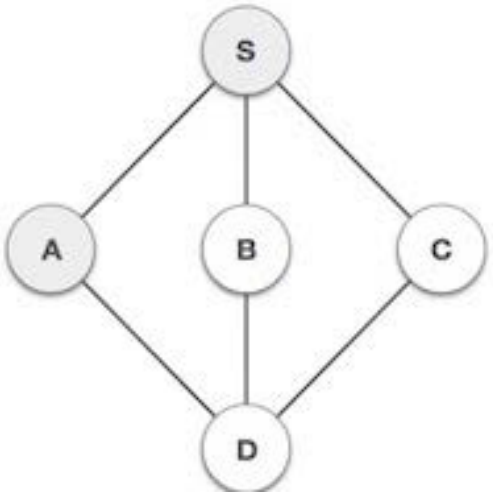

top
C
D
A
S
Stack

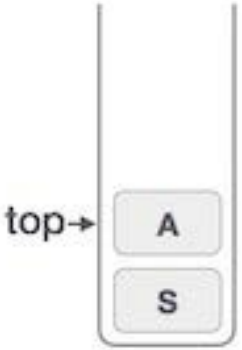

S
A
B
C
D

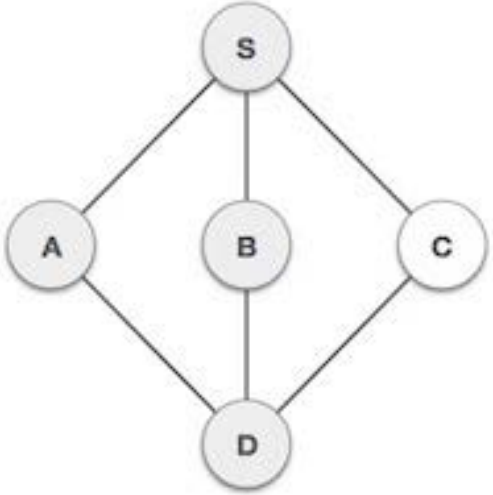

top
S

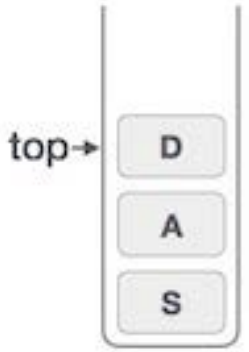

S
A
B
C
D


top
B
D
A
S


S
A
B
C
D

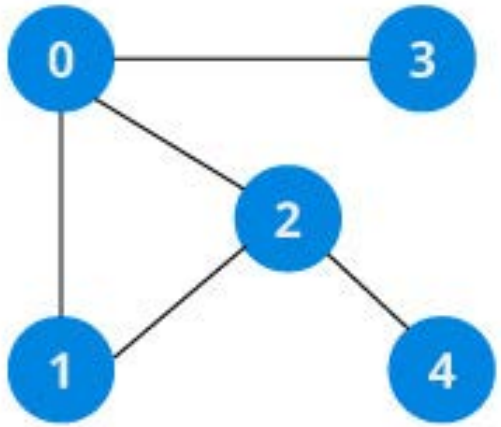

top
A
S
Stack


S
A
B
C
D


top
D
A
S
Stack

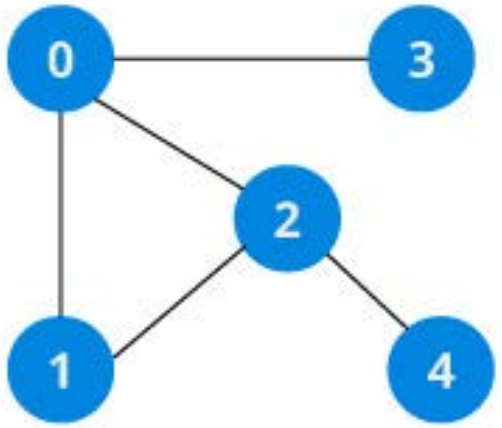
```python
def dfs(graph, node):
    visited = [node]
    stack = [node]
    while stack:
        node = stack[-1]
        if node not in visited:
            visited.extend(node)
        remove_from_stack = True
        for next in graph[node]:
            if next not in visited:
                stack.extend(next)
                remove_from_stack = False
                break
        if remove_from_stack:
            stack.pop()
    return visited

print (dfs(graph1, 'A'))
```

✓غیربازگشتی و
با استفاده از stack



```
# A function used by DFS
def DFSUtil(self, v, visited):

    # Mark the current node as visited
    # and print it
    visited[v] = True
    print(v, end = ' ')

    # Recur for all the vertices
    # adjacent to this vertex
    for i in self.graph[v]:
        if visited[i] == False:
            self.DFSUtil(i, visited)

# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self, v):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.graph))

    # Call the recursive helper function
    # to print DFS traversal
    self.DFSUtil(v, visited)
```

✓بازگشتی



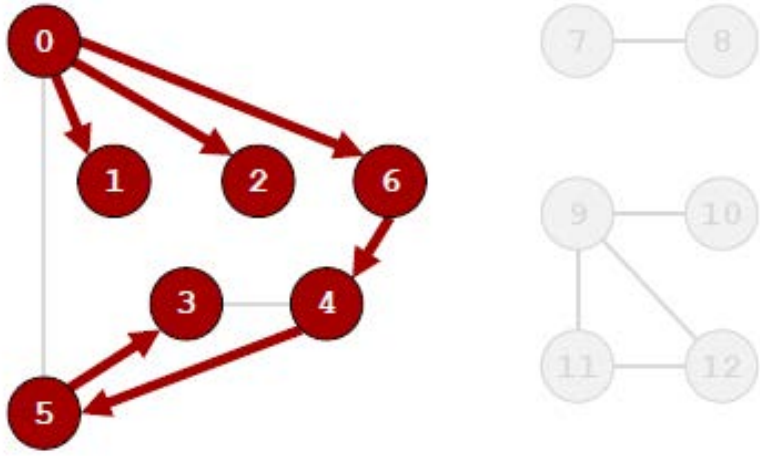
کاربرد DFS

- ✓یافتن تمام رئوس متصل به راس مبدا
- ✓یافتن مسیر بین دو راس (در صورت وجود)
- ✓تولید درخت پوشا
- ✓تشخیص دور در گراف
- ✓تشخیص دو بخشی بودن گراف
- ✓حل مارپیچ های تک مسیر

تمرین

v	marked[]	prev[]
0	T	-
1	T	0
2	T	0
3	T	5
4	T	6
5	T	4
6	T	0
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

- ✓برنامه ای بنویسید که با استفاده از پیمایش DFS تمام روئوس قابل دسترسی توسط راس 0 را چاپ کند
- ✓همچنین مسیر قابل دسترسی راس ۵ از راس 0 را چاپ کند



رئوس قابل دسترسی از رأس ۰

جستجوی سطحی
BREADTH FIRST SEARCH

- DFS: رئوس رویت نشده در بالای پشته قرار میگیرد
- BFS: رئوس رویت نشده در انتهای صف قرار میگیرد

BFS (from source vertex s)

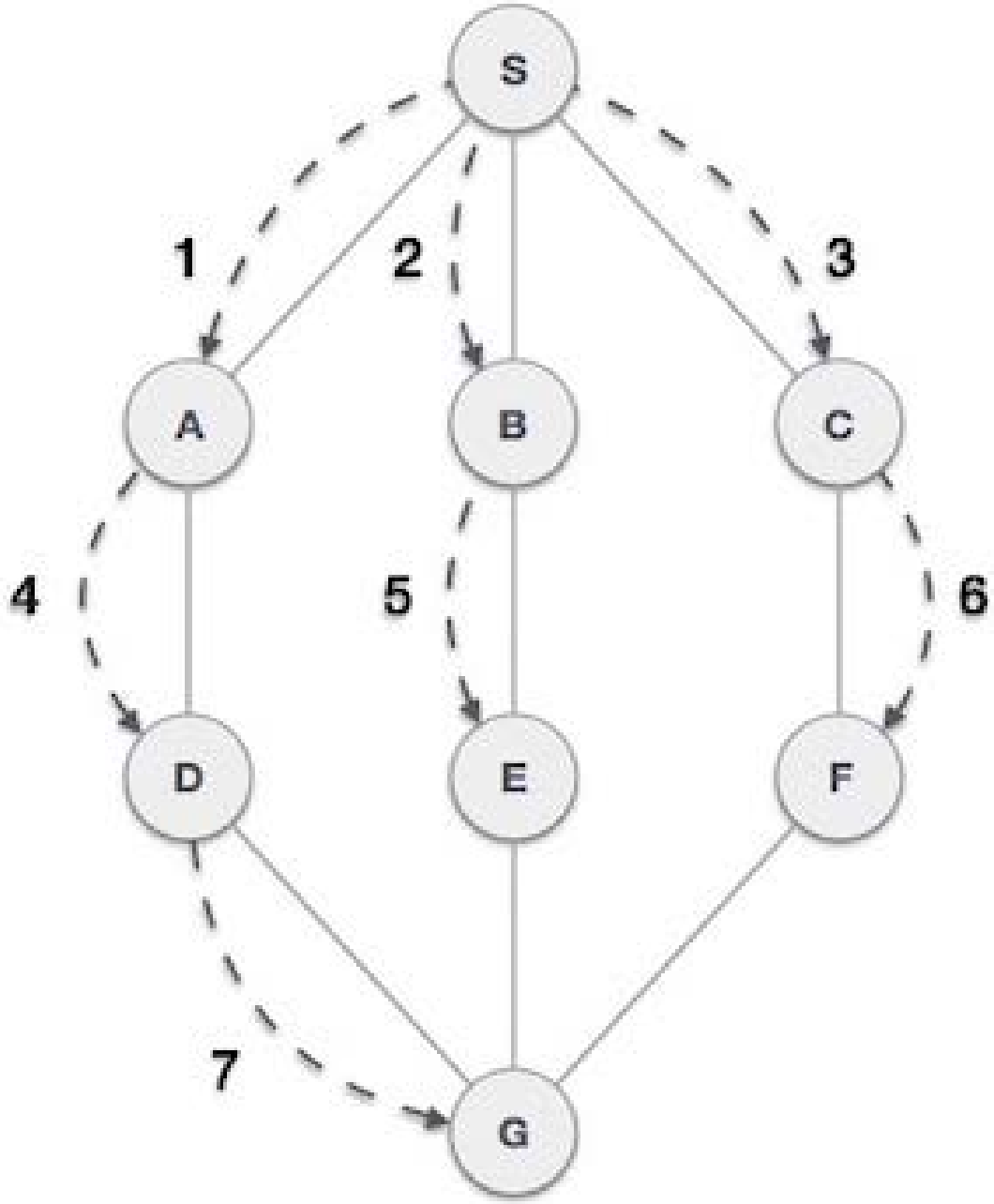
put s onto a FIFO queue and mark s as visited.

Repeat until the queue is empty:

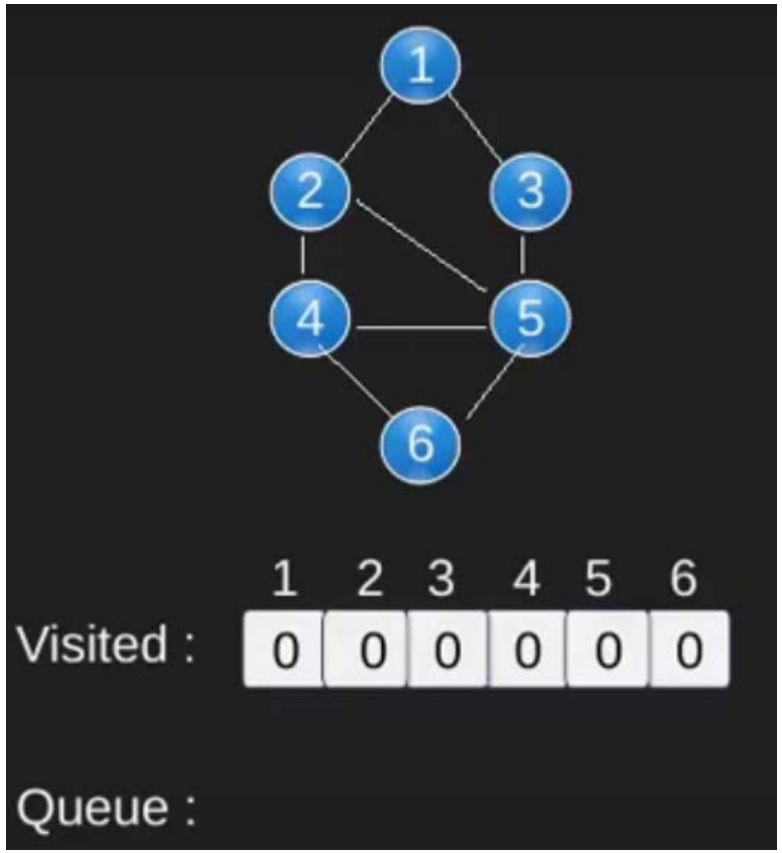
- remove the least recently added vertex v
- add each of v's unvisited neighbors to the queue,
  and mark them as visited.

روش کار

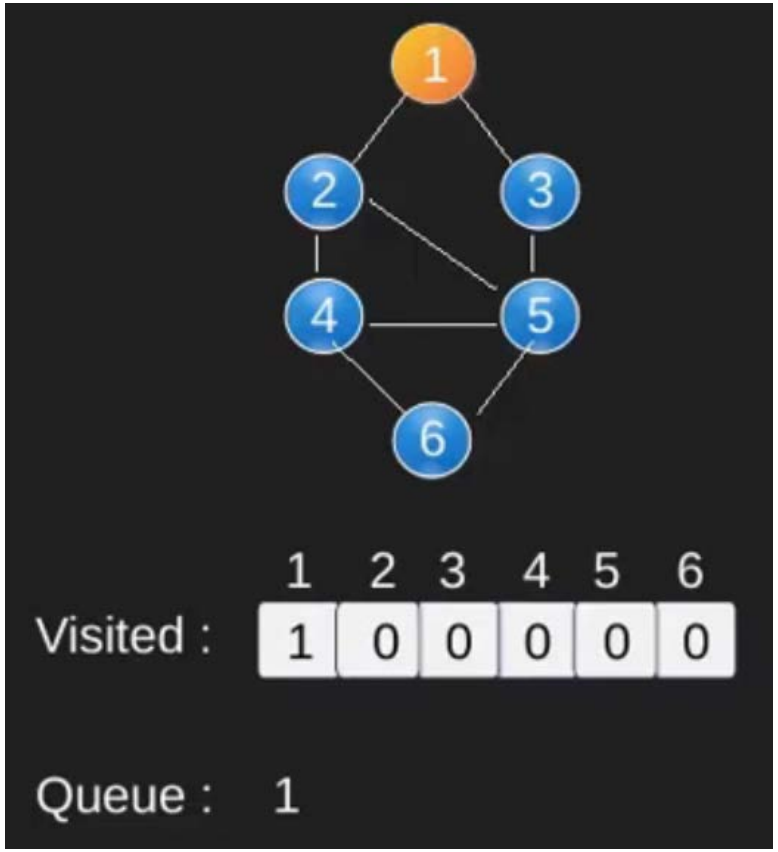
- ✓ ۱- عنصر جلوی صف را به عنوان گره جاری انتخاب و از صف حذف کن.
- ✓ ۲- گره جاری را پردازش کن.
- ✓ ۳- گره‌های مجاور گره جاری که پردازش نشده و در صف پردازش نیز قرار ندارند به این صف اضافه کن.


S
1
2
3
A
B
C
4
5
6
D
E
F
7
G

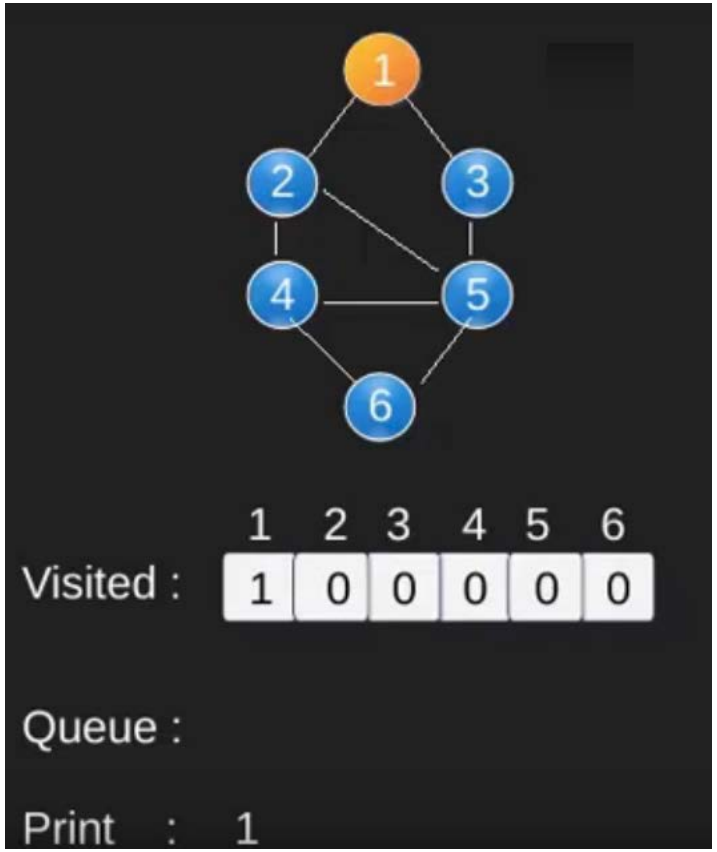
پیمایش سطحی...



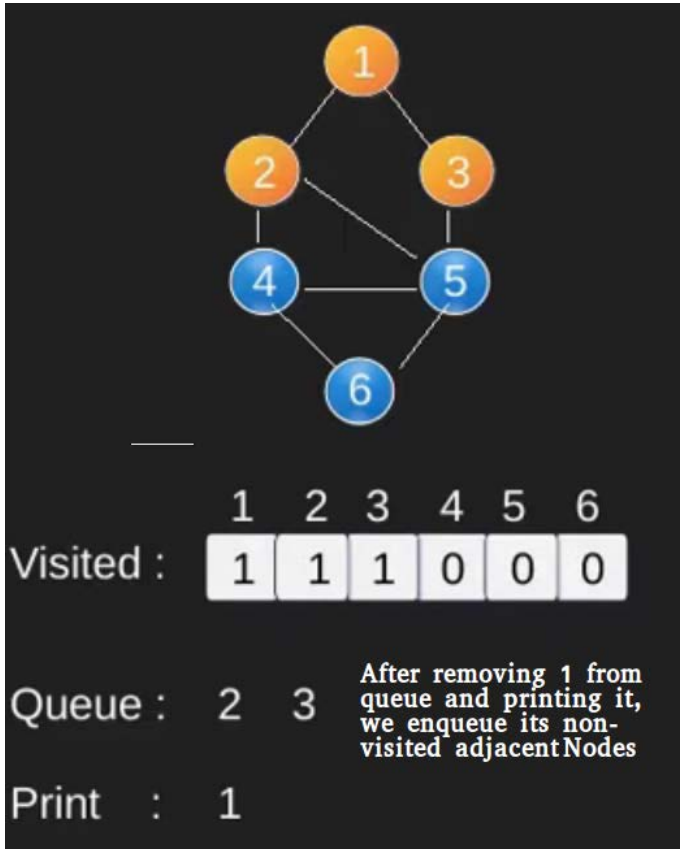
پیمایش سطحی...



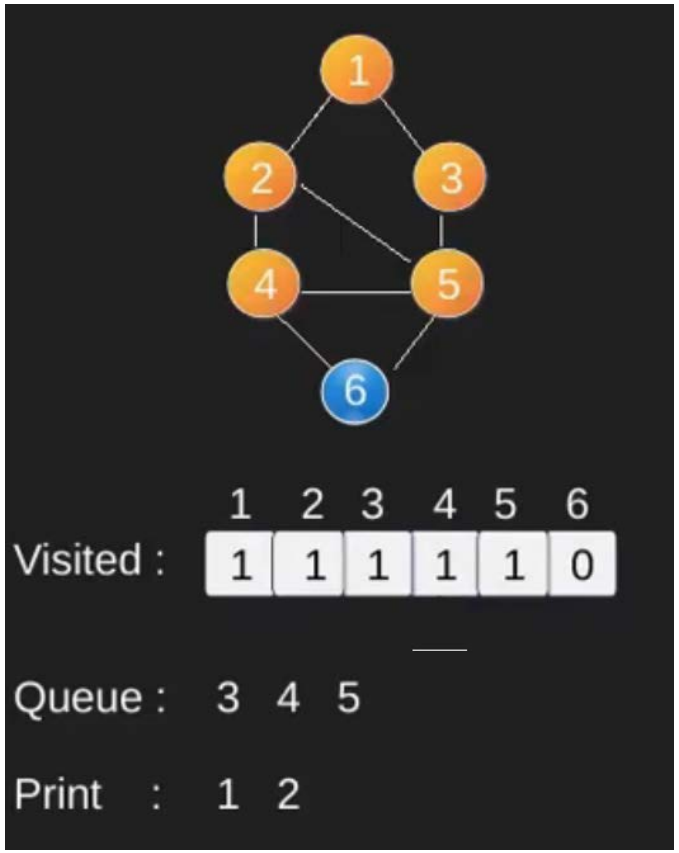
پیمایش سطحی...



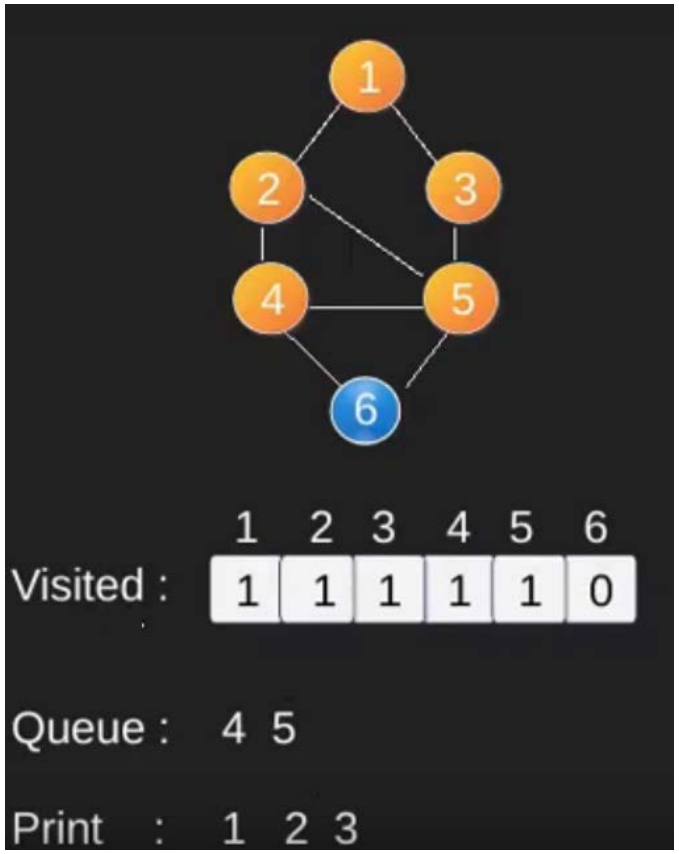
پیمایش سطحی...


1
2
3
4
5
6
1 2 3 4 5 6
Visited :
1 1 1 0 0 0
Queue : 2 3
After removing 1 from queue and printing it, we enqueue its non-visited adjacent Nodes
Print : 1

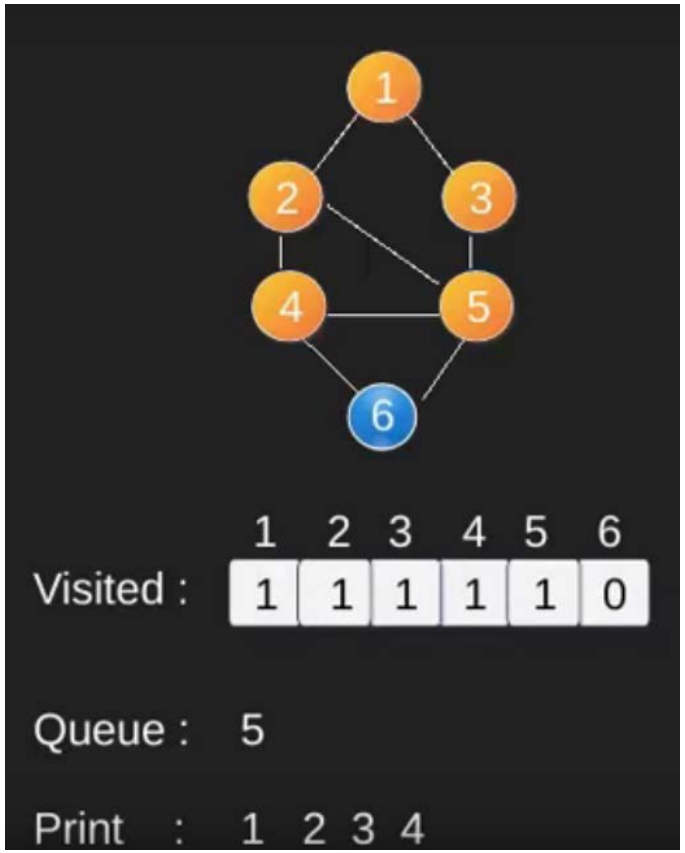
پیمایش سطحی...



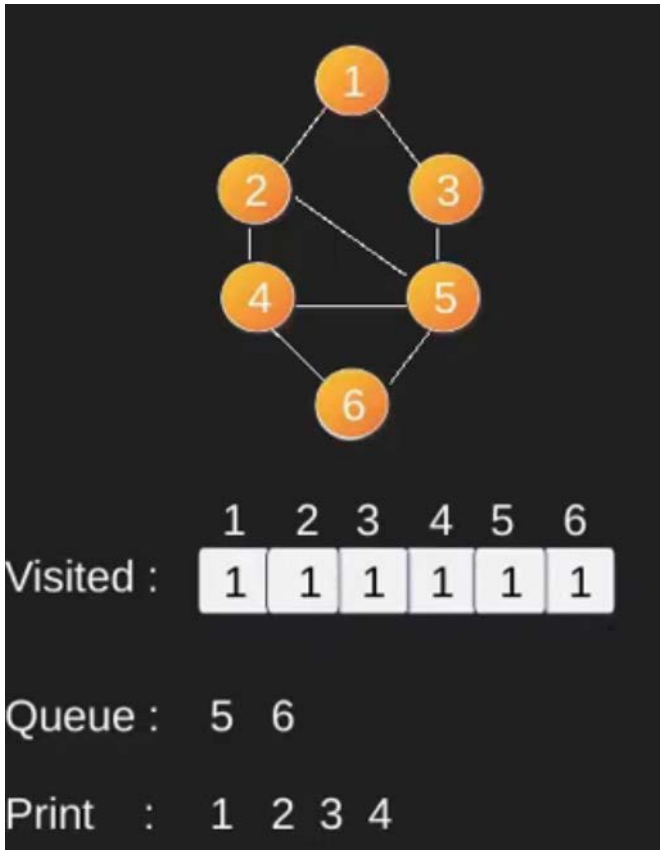
پیمایش سطحی...



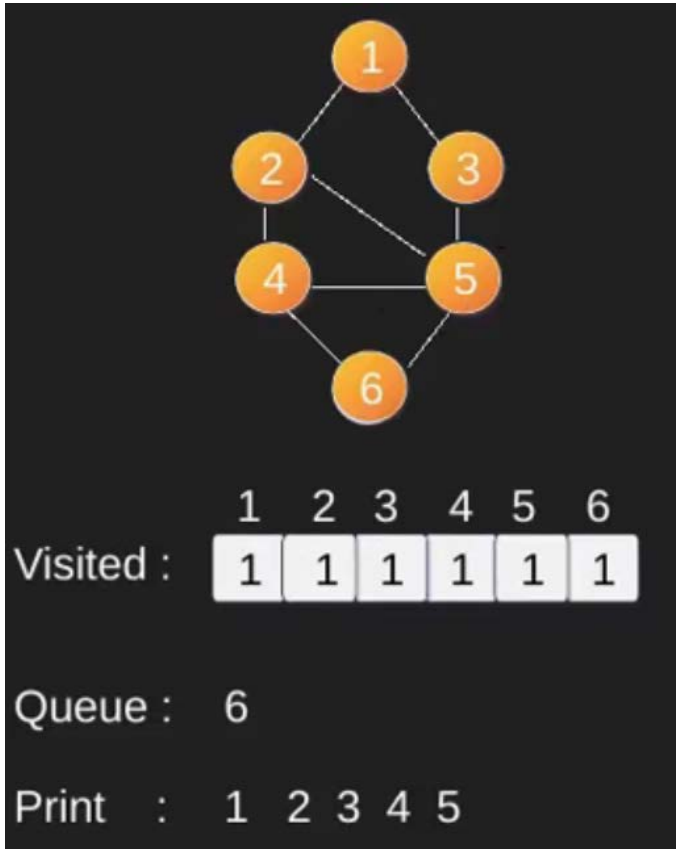
پیمایش سطحی...



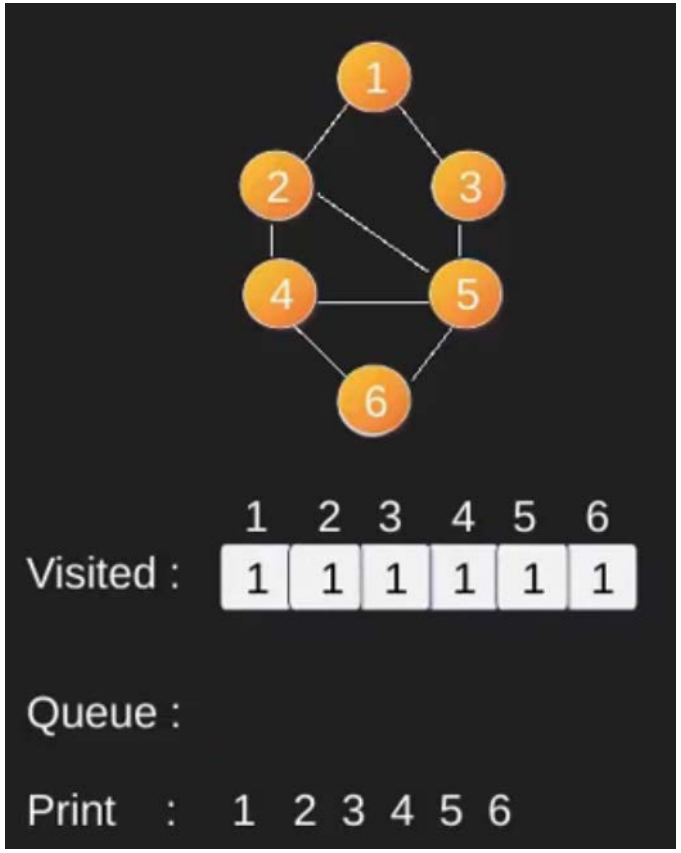
پیمایش سطحی...

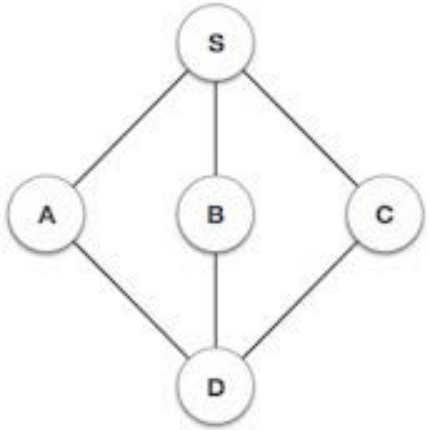


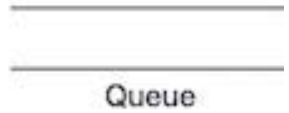
پیمایش سطحی...

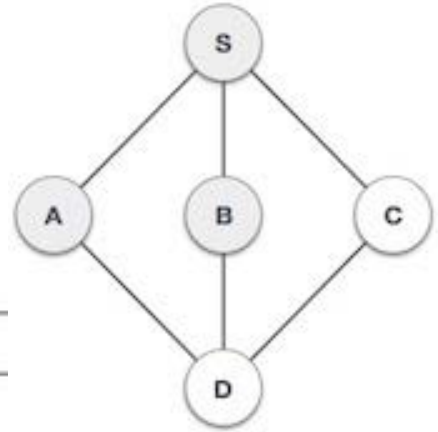


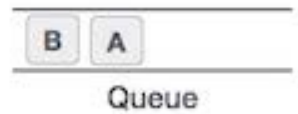
پیمایش سطحی...

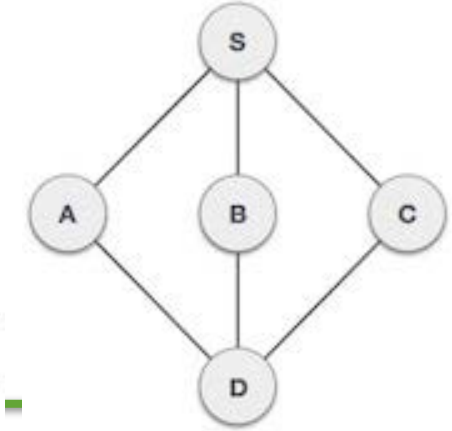


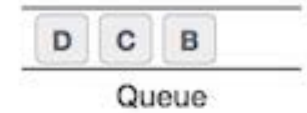

S
A
B
C
D

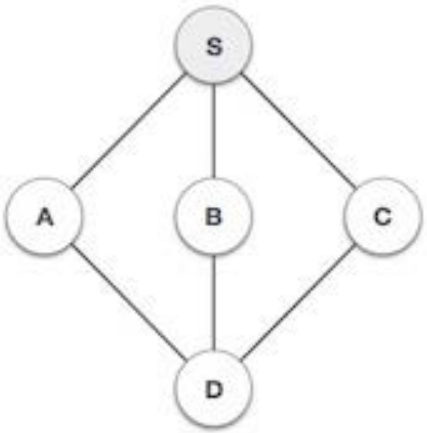

Queue

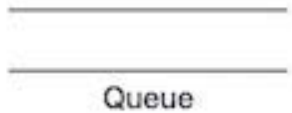

S
A
B
C
D

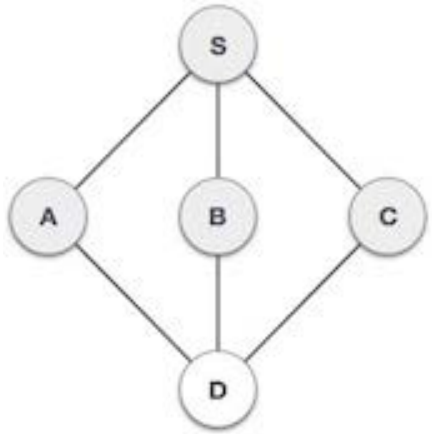

B
A
Queue

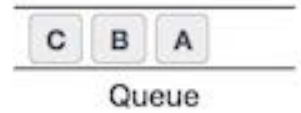

S
A
B
C
D

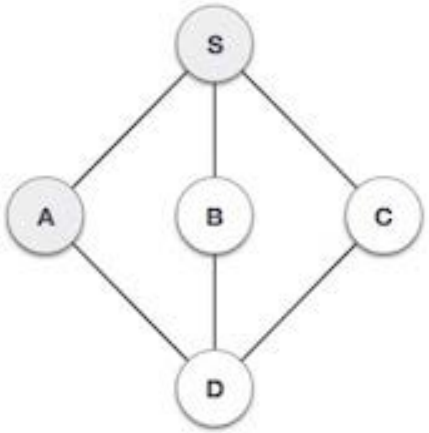

D
C
B
Queue

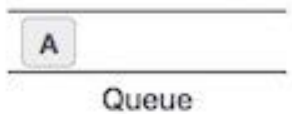

S
A
B
C
D

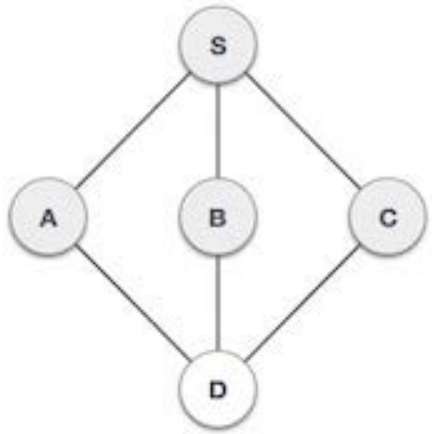

Queue

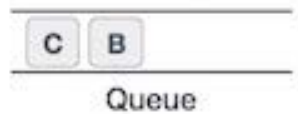

S
A
B
C
D


C
B
A
Queue


S
A
B
C
D


A
Queue


S
A
B
C
D


C
B
Queue

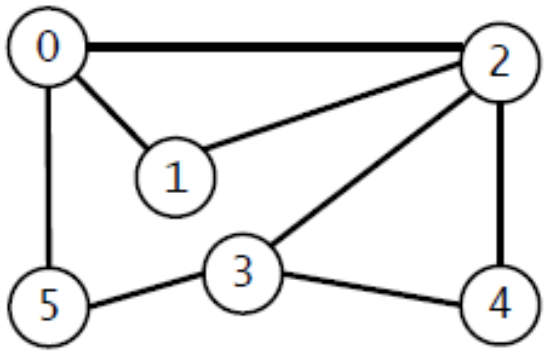
```
def BFS(self, s):

    # Mark all the vertices as not visited
    visited = [False] * (len(self.graph))

    # Create a queue for BFS
    queue = []

    # Mark the source node as
    # visited and enqueue it
    queue.append(s)
    visited[s] = True

    while queue:

        # Dequeue a vertex from
        # queue and print it
        s = queue.pop(0)
        print (s, end = " ")

        # Get all adjacent vertices of the
        # dequeued vertex s. If a adjacent
        # has not been visited, then mark it
        # visited and enqueue it
        for i in self.graph[s]:
            if visited[i] == False:
                queue.append(i)
                visited[i] = True
```

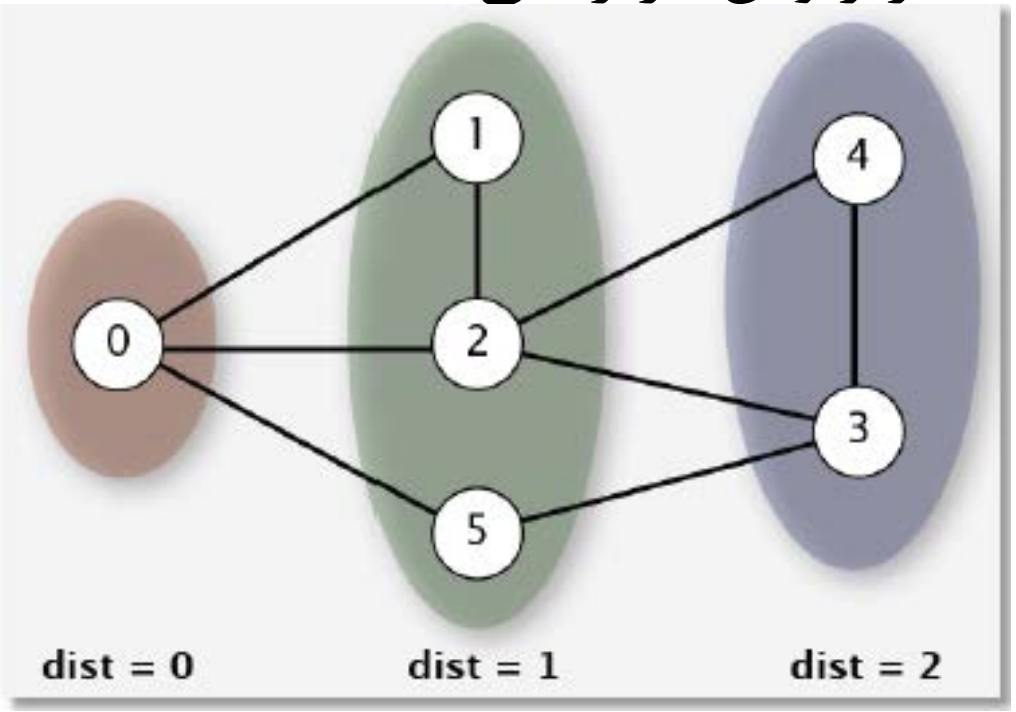
تمرین

- ✓برنامه پیمایش سطحی را به صورت بازگشتی بنویسید
- ✓کوتاهترین مسیر از گره مبدا به بقیه گره ها رو چاپ کنید

کاربرد پیمایش سطحی

✓در یک گراف همبند، کوتاهترین مسیر (بر حسب تعداد یال) از راس مبدا s تا سایر رئوس در زمانی متناسب با V+E محاسبه می شود

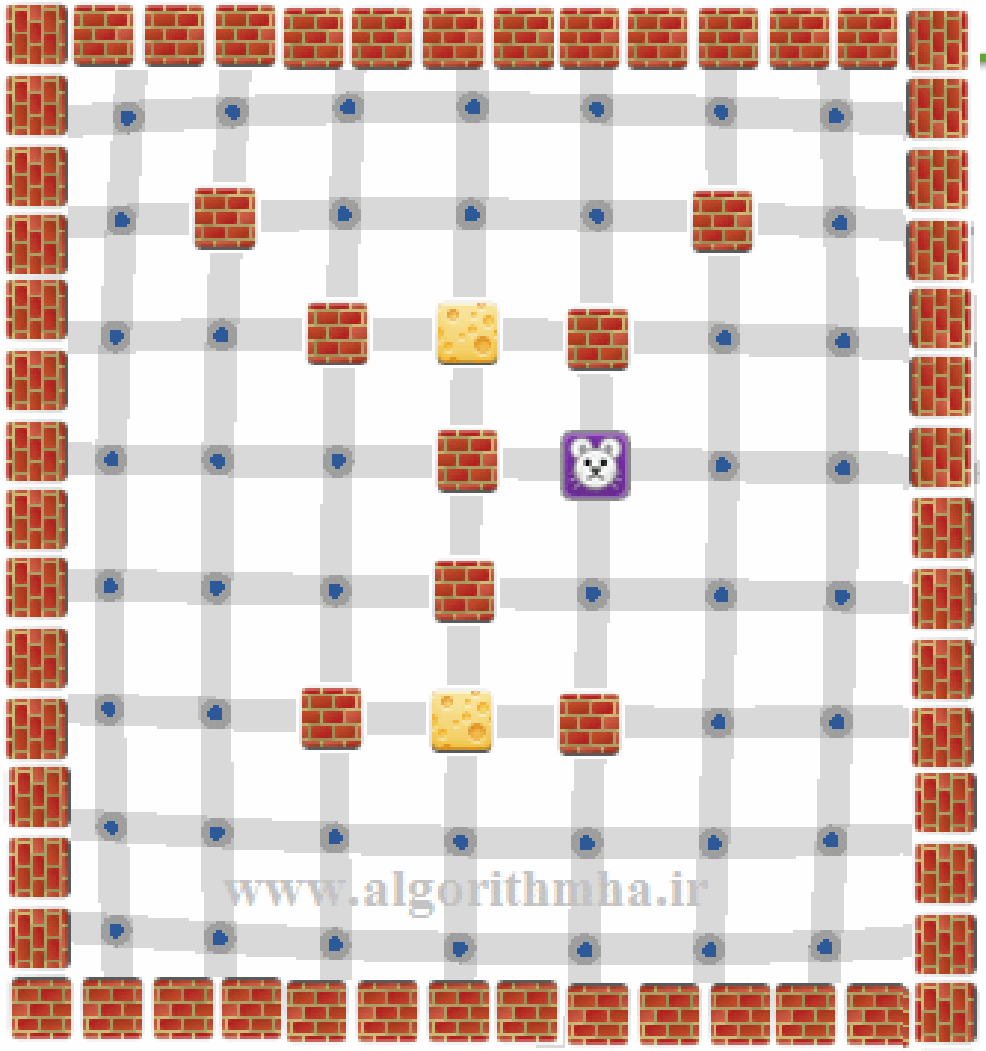




کاربرد BFS

- ✓تشخیص همبندی گراف یا مؤلفه‌های همبندی آن
- ✓تولید یک درخت پوشا برای گراف (درخت پوشای تولید شده با الگوریتم BFSبرای یک گراف وزن‌دار لزوما درخت پوشای کمینه نیست)
- ✓برای پیمایش صفحات مارپیچ (Maze)

نزدیکترین پنیر



نکات BFS و DFS

- مرتبه زمانی BFS و DFS:
  - برای گراف $G=(E,V)$ برابر O(|E|+|V|) است؛ چرا که این الگوریتم در بزرگترین حالت تمامی گره‌ها را پیمایش کرده و نیاز به بررسی تمامی یال‌ها دارد. این مرتبه‌ی اجرایی در یک گراف همبند به صورت $O(|E|)$بوده و در حالت کلی متناسب با تعداد یال‌ها حداکثر از مرتبه‌ی O(n^2) است.
- در الگوریتم BFS اگر گراف بدون وزن باشد، مسیر مشخص شده توسط الگوریتم به طور حتم کوتاهترین مسیر از گره مبدأ به مقصد است؛ اما در مورد الگوریتمDFS این خاصیت لزوما برقرار نیست.
- الگوریتم BFSراهکاری است که در یافتن کوتاهترین مسیر در مسیرهای مارپیچ (Mazeکاربرد دارد. اگر مسیر رسیدن به هر نقطه از مارپیچ یکی باشد (دوری در مارپیچ نباشد یا گراف معادل مارپیچ یک درخت باشد) می‌توان از الگوریتم DFS نیز استفاده کرد. چرا که در چنین حالتی تنها یک مسیر به مقصد وجود داشته و کوتاه بودن آن مطرح نیست.

تمرین

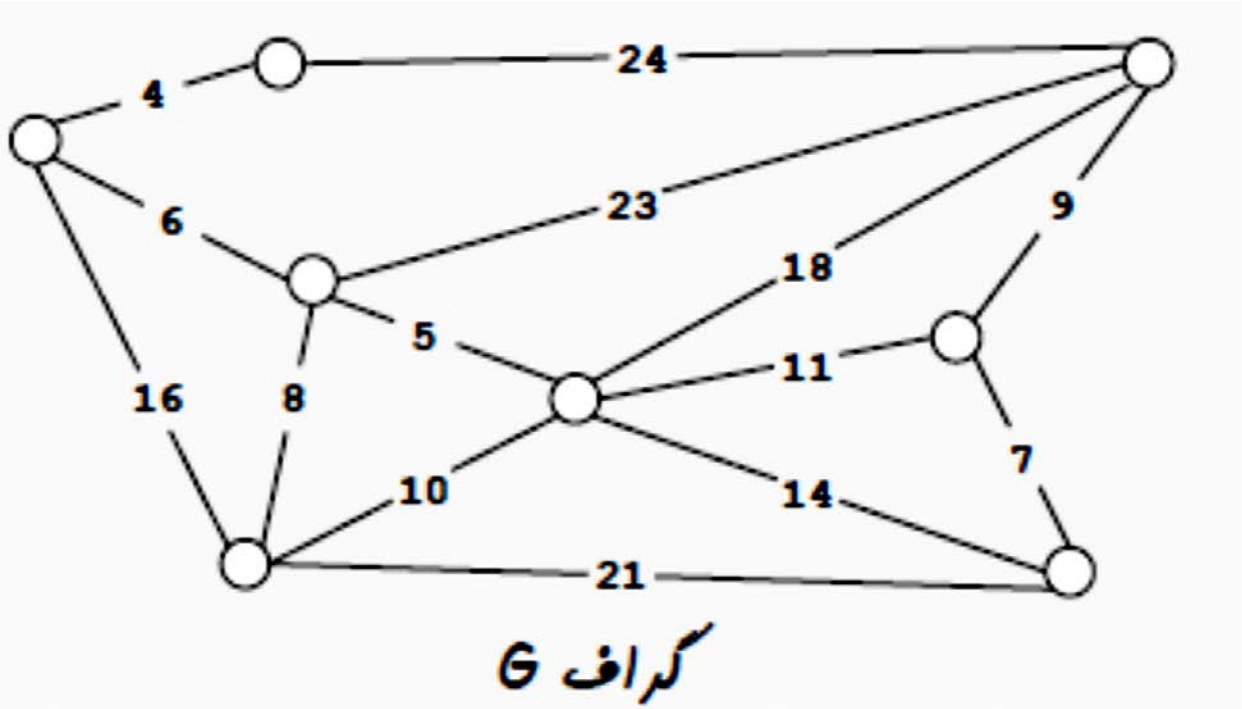
- با استفاده از پیمایش سطحی، تعداد کوتاهترین مسیر قابل دسترسی از گره مبدا تا بقیه گره ها را چاپ کنید

درخت پوشا
SPANNING TREE

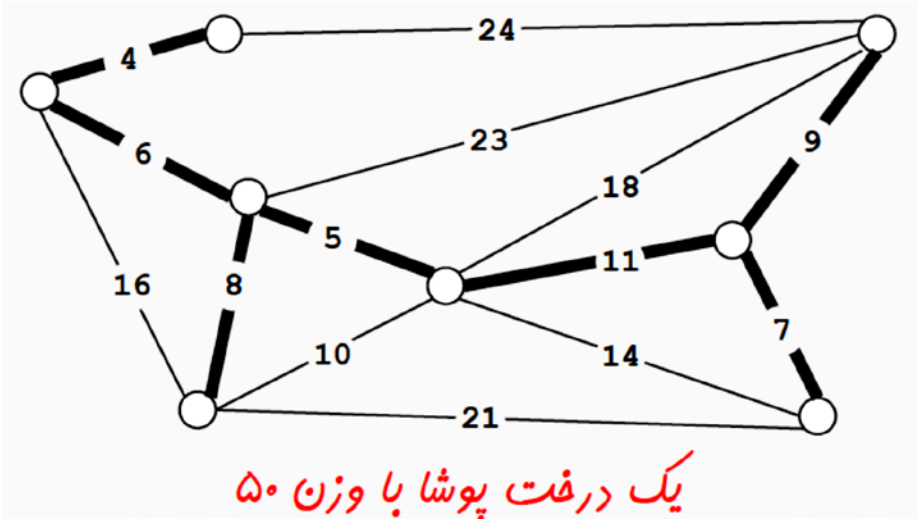
- یک گراف همبند G می‌تواند بیش از یک درخت پوشا داشته باشد.
- همه درخت‌های پوشای گراف G تعداد یکسانی از یال‌ها و رئوس را دارند.
- درخت پوشا هیچ دوری ندارد.
- با حذف یک یال از درخت پوشا، به گراف غیر همبند تبدیل می‌شود، یعنی درخت پوشا دارای کمینه اتصال‌های ممکن است.
- افزودن یک یال به درخت پوشا موجب ایجاد یک مدار یا طوقه می‌شود، یعنی درخت پوشا در حالت بیشینه غیر دوری (maximally acyclic) است.
- یک گراف کامل می‌تواند در بیشینه حالت خود n^{n-2} عدد درخت پوشا داشته باشد. (n تعداد گره ها)

درخت پوشای کمینه
MINIMUM SPANING TREE

- ورودی: گراف بدون جهت و همبند G که وزن یالهای آن مثبت است
- تعریف: یک درخت پوشا از گراف G، یک زیر گراف **همبند** و **بدون دور** از گراف
- هدف: یافتن درخت پوشا با کمترین هزینه



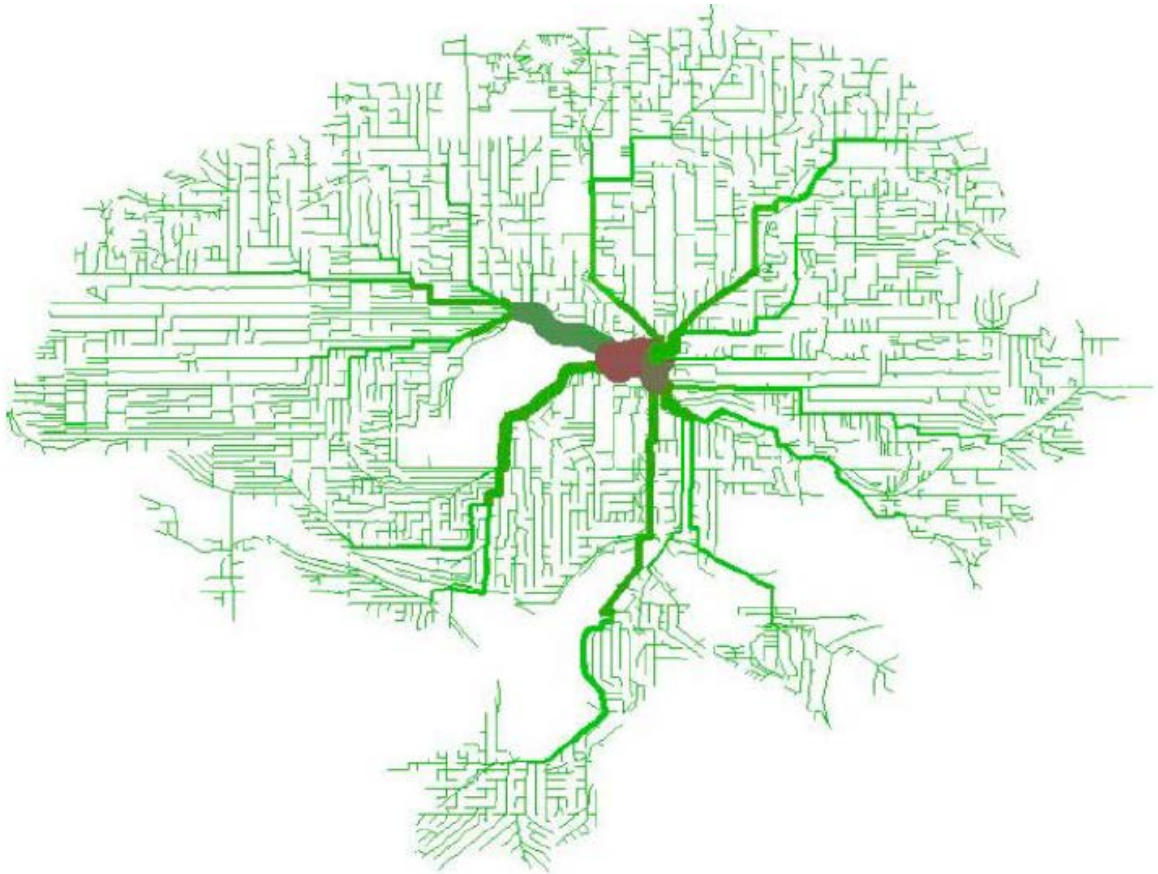
گراف G



یک درخت پوشا با وزن ۵۰

مثال از MSP

- درخت پوشای کمینه برای مسیرهای دوچرخه سواری در شهر سیاتل



کاربرد MST

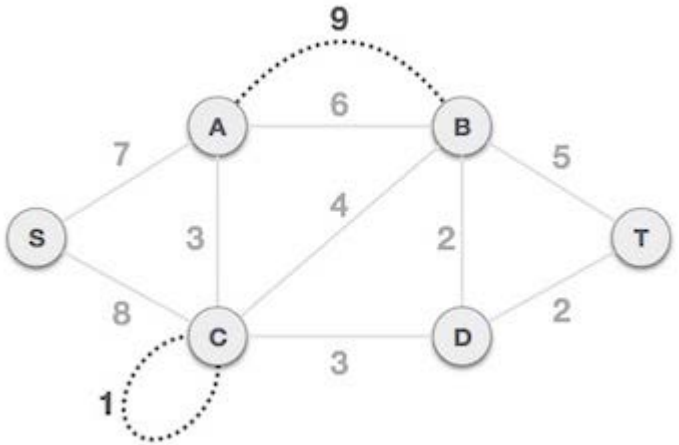
- ✓ طراحی شبکه (ارتباطی، الکتریکی، کامپیوتری، کابلی، جاده ای)
- ✓ الگوریتم های تقریبی برای مسائل ان-پی کامل (فروشنده ی دوره گرد)
- ✓ یافتن شبکه ی راه ها در تصاویر هوایی و ماهواره ای
- ✓ تشخیص چهره به صورت بلادرنگ

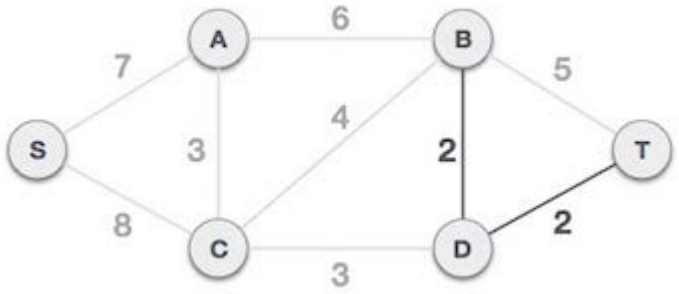
دو روش معروف با رویکرد حریصانه

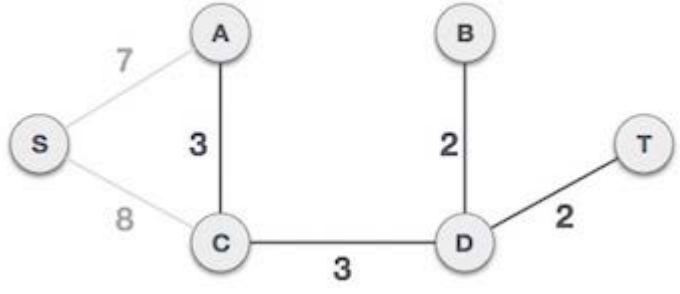
- ✓ پرایم
- ✓ کروسکال

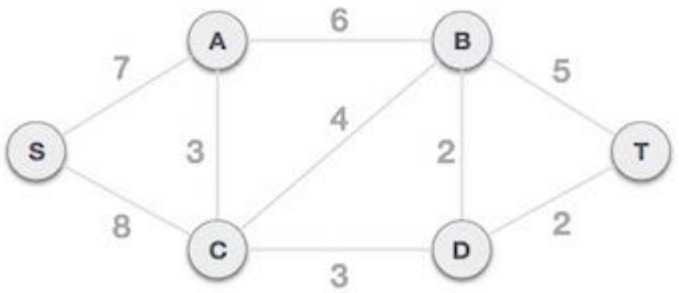
کروسکال

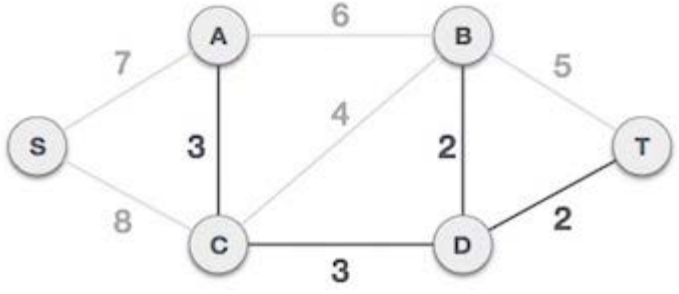
- ✓یالهای طوقه و موازی رو حذف میکنیم (حذف یال با وزن بیشتر در یالهای موازی)
- ✓یالها را به ترتیب صعودی وزن در نظر بگیر؛
- ✓کم وزنترین یال باقیمانده را به درخت **T** اضافه کن مگر آن که انجام این کار باعث ایجاد دور شود

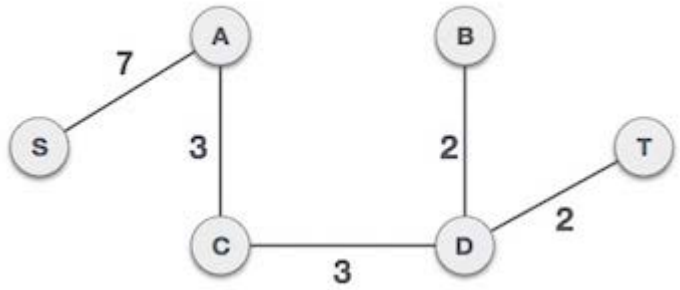




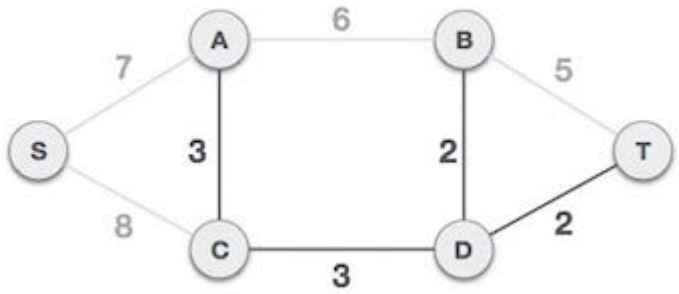




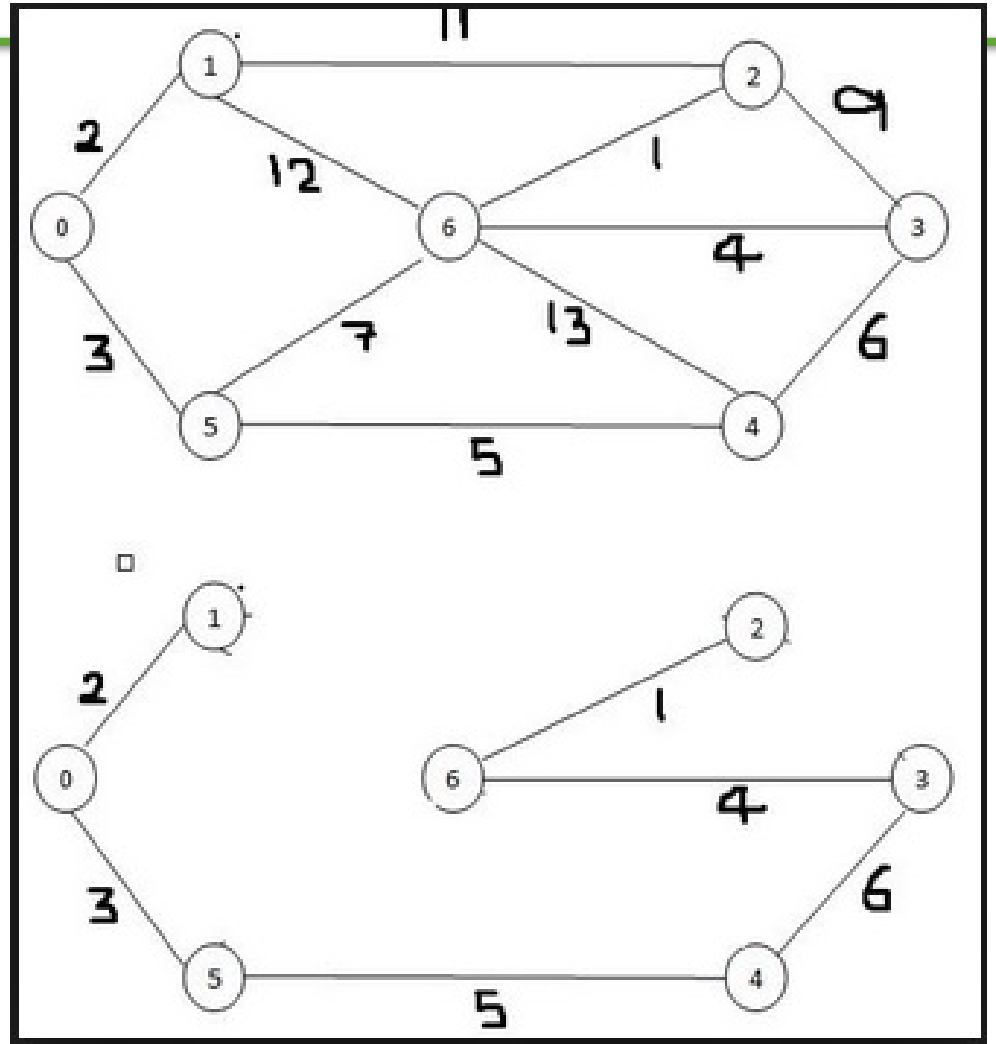




B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

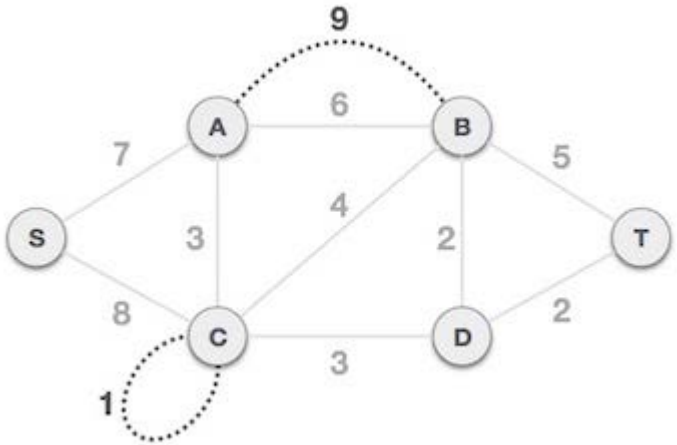


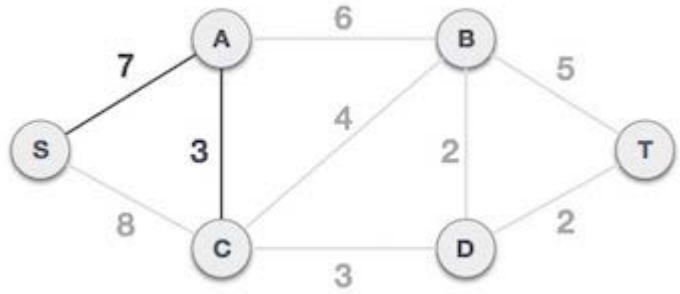
کروسکال

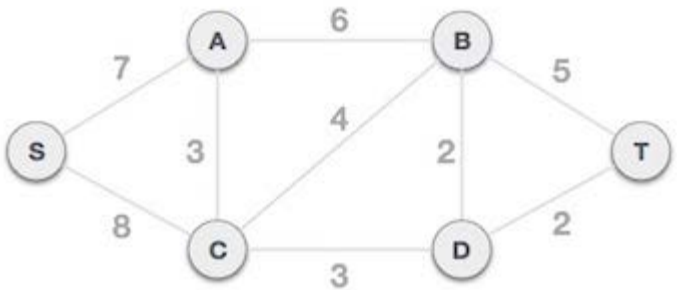


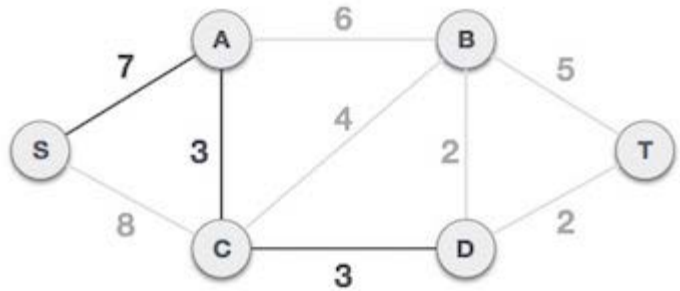
پرایم

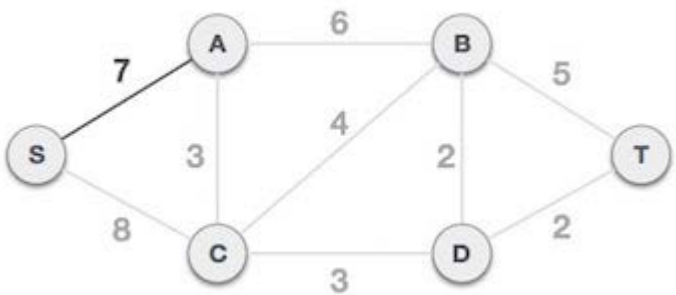
- ✓یالهای طوقه و موازی رو حذف میکنیم (حذف یال با وزن بیشتر در یالهای موازی)
- ✓با رأس دلخواه شروع کن و درخت **T** را به طور حریصانه بزرگ کن؛
- ✓در هر مرحله، یال با کمترین وزن را که دقیقا یک رأس آن در **T**است، به **T**اضافه کن

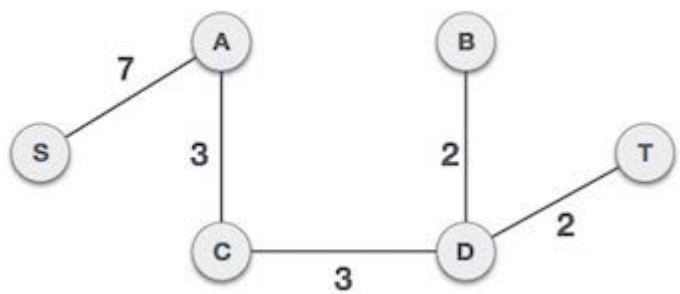

9
A
B
6
7
5
4
S
3
2
T
8
C
D
2
3
1


A
B
6
7
5
4
S
3
2
T
8
C
D
2
3

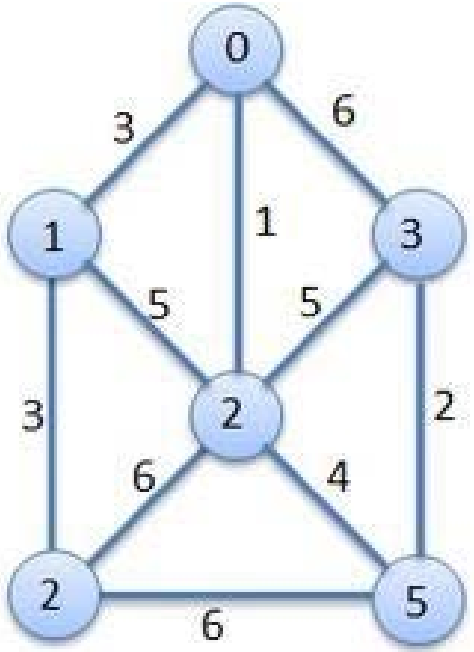

A
B
6
7
5
4
S
3
2
T
8
C
D
2
3

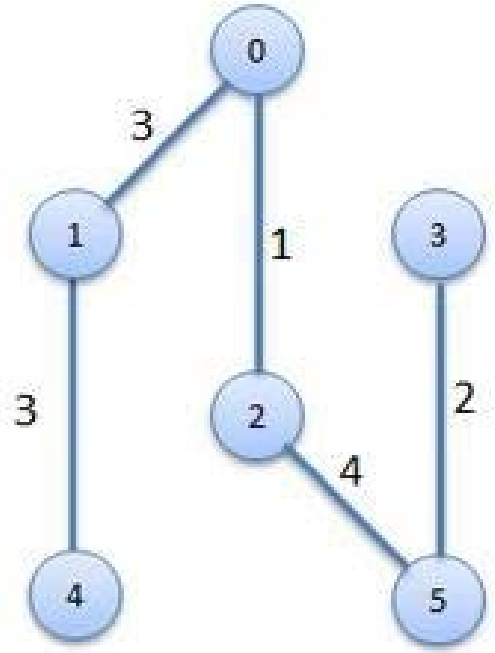

A
B
6
7
5
4
S
3
2
T
8
C
D
2
3


A
B
6
7
5
4
S
3
2
T
8
C
D
2
3


A
B
7
S
3
2
T
C
D
2
3

پرایم





مقایسه پرایم و کروسکال

دو الگوریتم prim و kruskal درصورتی دو درخت متفاوت تولید میکنند که چندین یال با هزینه‌های مساوی داشته باشیم ولی همیشه هزینه درختهای تولید شده برابر و کمینه است. در الگوریتم Prim در ابتدا یک گره بوده و کم کم در حین الگوریتم گسترش میابد تا به درخت پوشای کمینه تبدیل شود درحالیکه در الگوریتم kruskal چند درخت (جنگل) وجود دارد که در انتهای الگوریتم درختهای جنگل به هم پیوند خورده تا تبدیل به درخت پوشای کمینه شوند.