- برای استفاده از پکیج نتورک ایکس در وی اس کد باید اول افزونه ی پایتون را در وی اس کد نصب کنیم سپس برنامه اصلی پایتون را در سیستم خود نصب میکنیم حال برای شروع کار از منوی فایل یک فایل با پسوند .py ایجاد میکنیم.

- برای شروع کار باید دو پکیج را به پروژه اضافه کنیم .

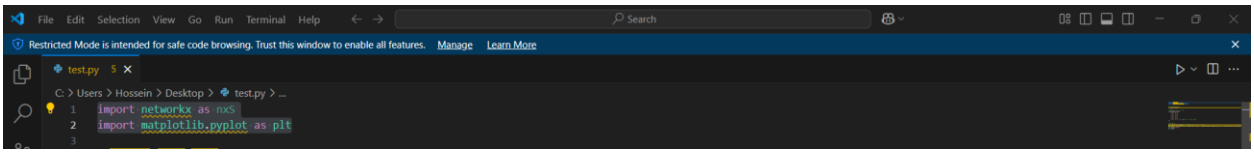
network X :1

matplotlib.pyplot:2

- ترمینال را از منوی ترمینال باز میکنیم و دستور pip install networkx اجرا میکنیم .

- حال فایل امده برای نوشتن دستورات هست ابتدا این دو پکیج را اضافه میکنیم .

```
import networkx as nxS
import matplotlib.pyplot as plt
```

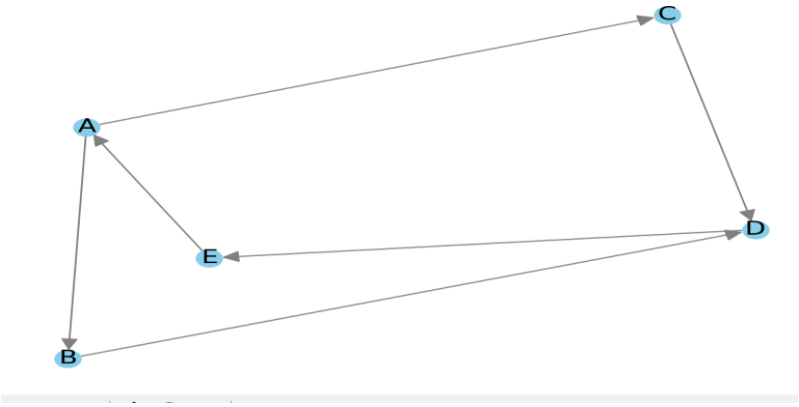


- حال یک گراف جهت دار رسم میکنیم(ساده).

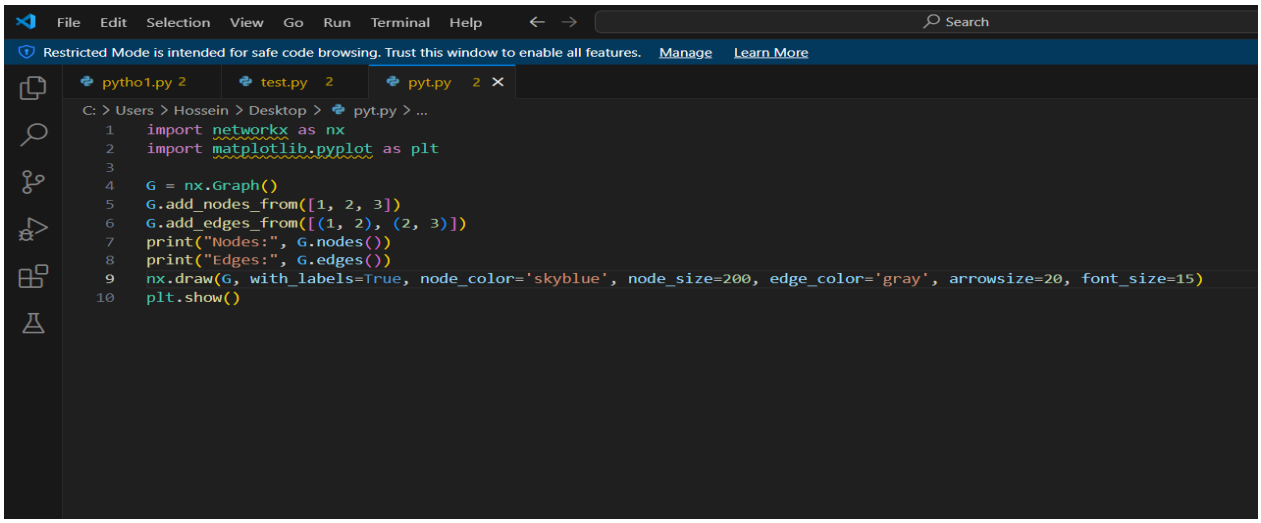
```python
import networkx as nx
import matplotlib.pyplot as plt

# ساخت گراف جهت دار
G = nx.DiGraph()

# (جهت یا) یال ها افزودن
G.add_edges_from([
    ('A', 'B'),
    ('A', 'C'),
    ('B', 'D'),
    ('C', 'D'),
    ('D', 'E'),
    ('E', 'A')  # بیشتر جذابیت برای حلقه یک
])

# گراف رسم
pos = nx.spring_layout(G)  # بهتر رسم برای گره ها موقعیت
nx.draw(G, with_labels=True, node_color='skyblue', node_size=200, edge_color='gray', arrowsize=20, font_size=15)
plt.title("lrng")
plt.show()
```

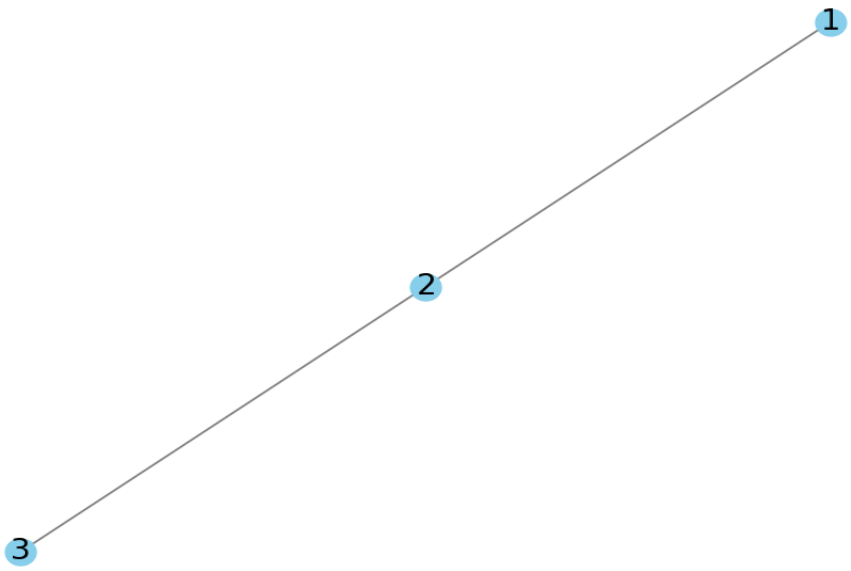
- با ترمینال به محل ذخیره فایل میرویم و با دستور python test.py برنامه را اجرا میکنیم .

.

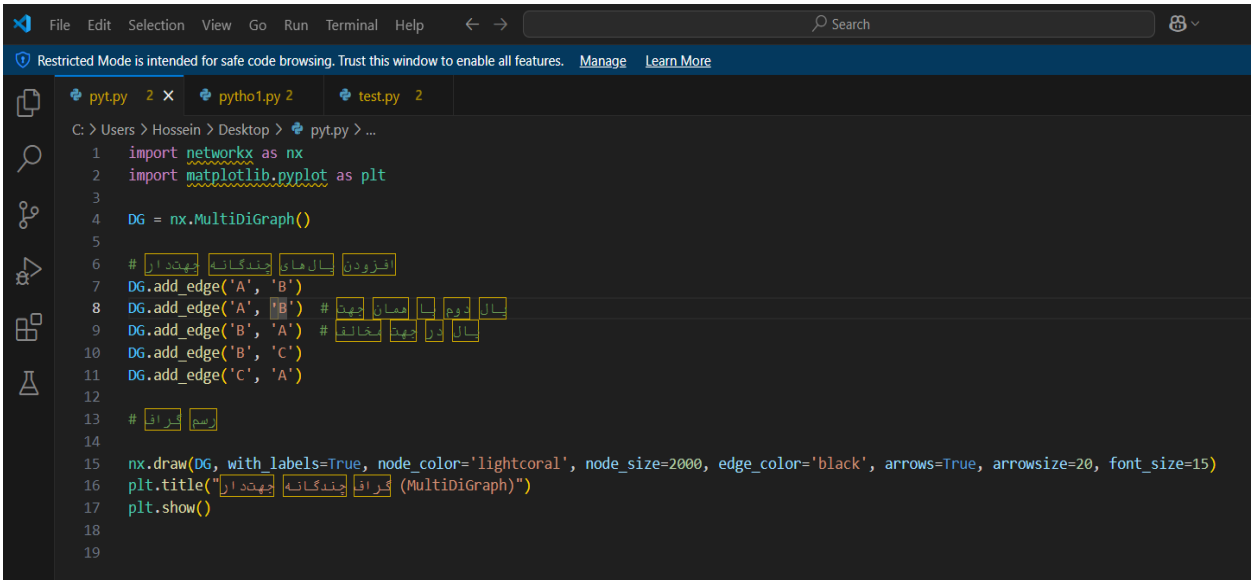


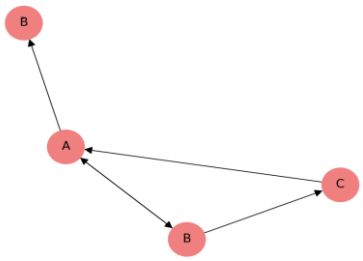
- گراف بدونه جهت (ساده) .>>

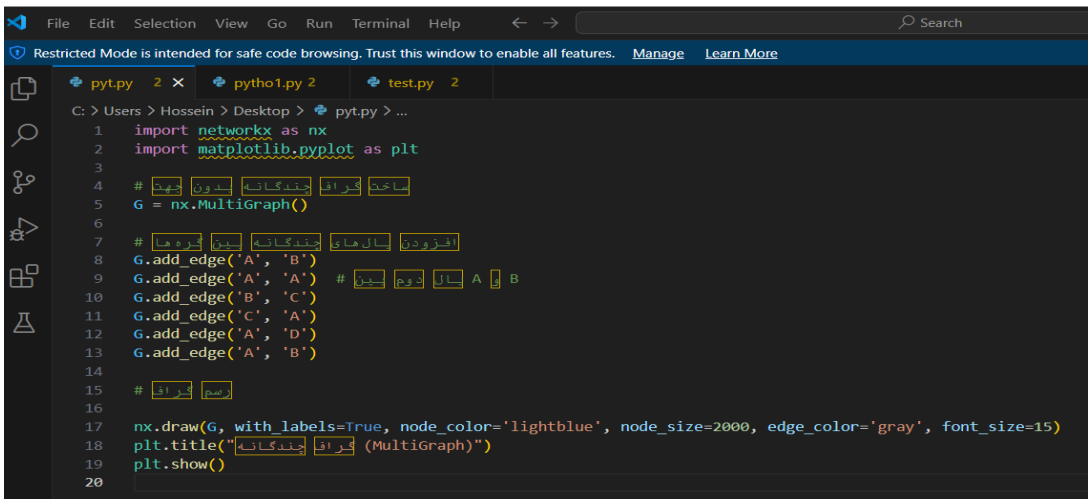


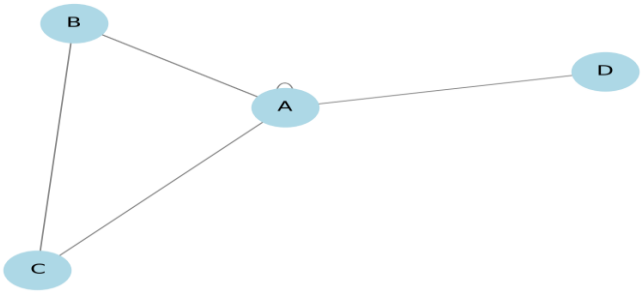


- گراف چند گانه(multigraph): یعنی گرافی که بین دو گره می‌تونه چند یال(با جهت یا بدون جهت) داشته باشه.

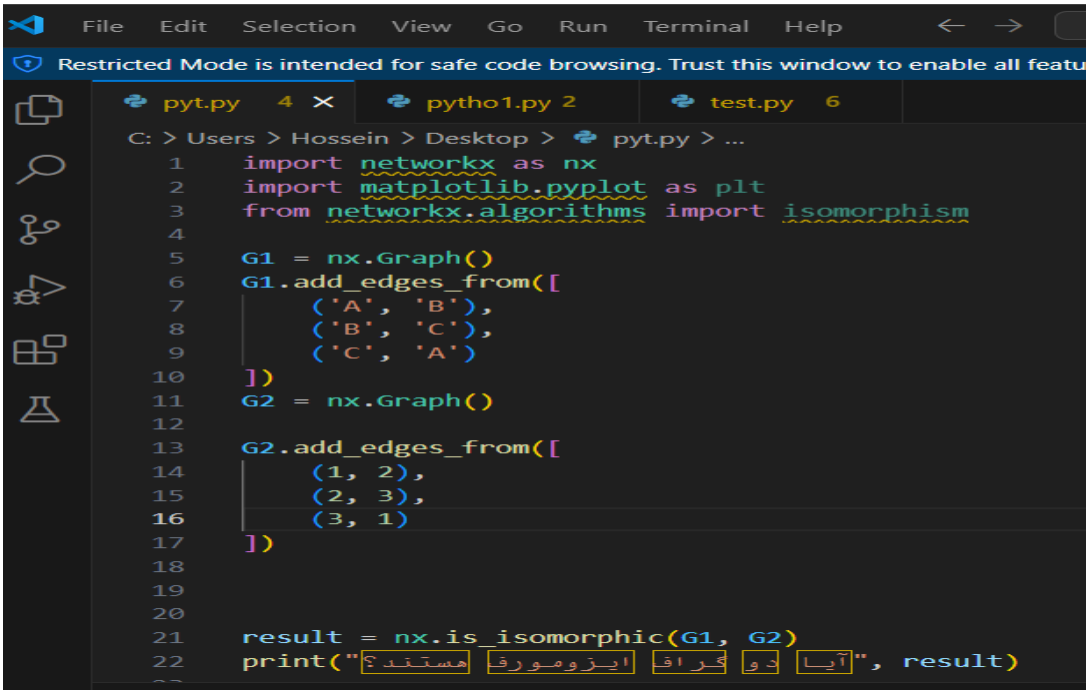

File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
pyt.py 2
pytho1.py 2
test.py 2
C: > Users > Hossein > Desktop > pyt.py > ...
import networkx as nx
import matplotlib.pyplot as plt
DG = nx.MultiDiGraph()
# جهت‌دار چندگانه یال‌های افزودن
DG.add_edge('A', 'B')
DG.add_edge('A', 'B') # جهت همان با دوم یال
DG.add_edge('B', 'A') # مخالف جهت در یال
DG.add_edge('B', 'C')
DG.add_edge('C', 'A')
# گراف رسم
nx.draw(DG, with_labels=True, node_color='lightcoral', node_size=2000, edge_color='black', arrows=True, arrowsize=20, font_size=15)
plt.title("جهت‌دار چندگانه گراف (MultiDiGraph)")
plt.show()


B
A
C
B


File Edit Selection View Go Run Terminal Help
Search
Restricted Mode is intended for safe code browsing. Trust this window to enable all features. Manage Learn More
pyt.py 2
pytho1.py 2
test.py 2
C: > Users > Hossein > Desktop > pyt.py > ...
import networkx as nx
import matplotlib.pyplot as plt
# جهت بدون چندگانه گراف ساخت
G = nx.MultiGraph()
# گره‌ها بین چندگانه یال‌های افزودن
G.add_edge('A', 'B')
G.add_edge('A', 'A') # بین دوم یال A و B
G.add_edge('B', 'C')
G.add_edge('C', 'A')
G.add_edge('A', 'D')
G.add_edge('A', 'B')
# گراف رسم
nx.draw(G, with_labels=True, node_color='lightblue', node_size=2000, edge_color='gray', font_size=15)
plt.title("چندگانه گراف (MultiGraph)")
plt.show()

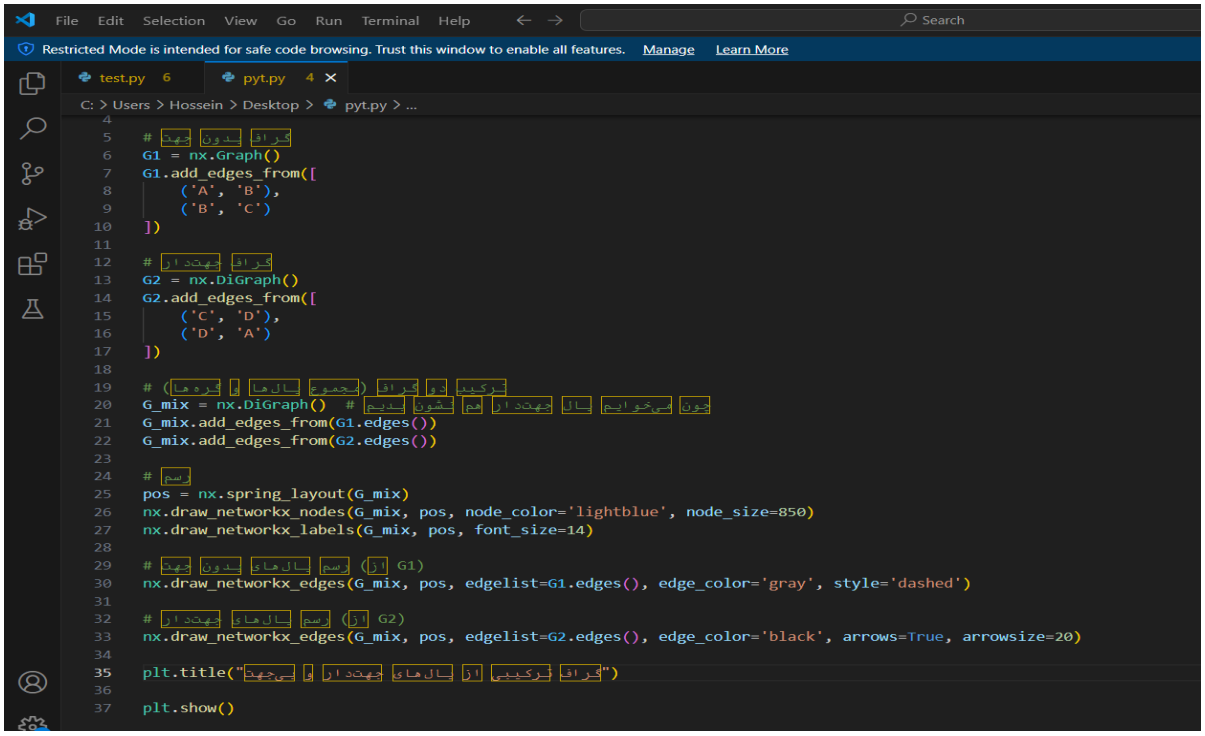

B
D
A
C

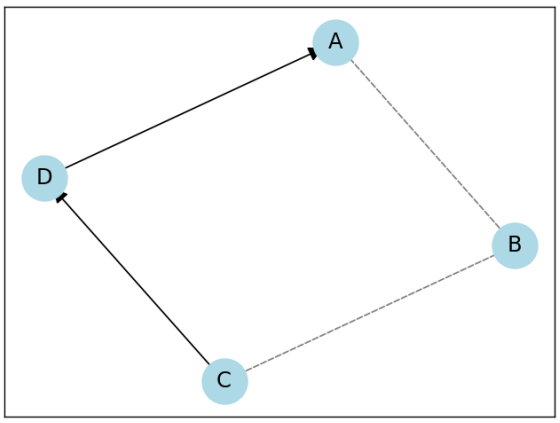
- گراف ایزومورفیک (Isomorphic Graphs) : ساختار آن‌ها یکسان باشد، یعنی فقط نام گره‌ها فرق کند، ولی اتصال بین گره‌ها (یال‌ها) دقیقاً یکسان باشد



خروجی true و false

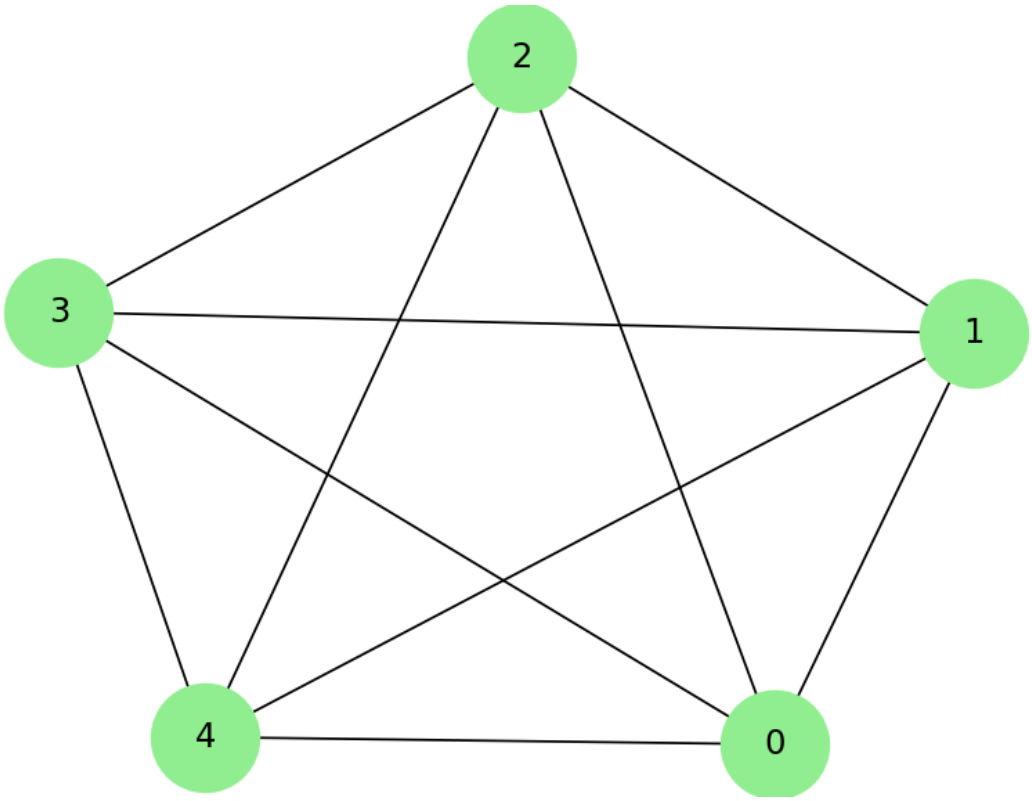
- گراف میکس : گراف ترکیبی از جهت‌دار و بدون جهت





- گراف کامل : بین هر دو گره یک یال وجود دارد.

```
G = nx.complete_graph(5)  # گره 5 با گراف کامل

pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000, font_size=14)
plt.title("جهت بدون کامل گراف (K5)")
plt.axis('off')
plt.show()
```

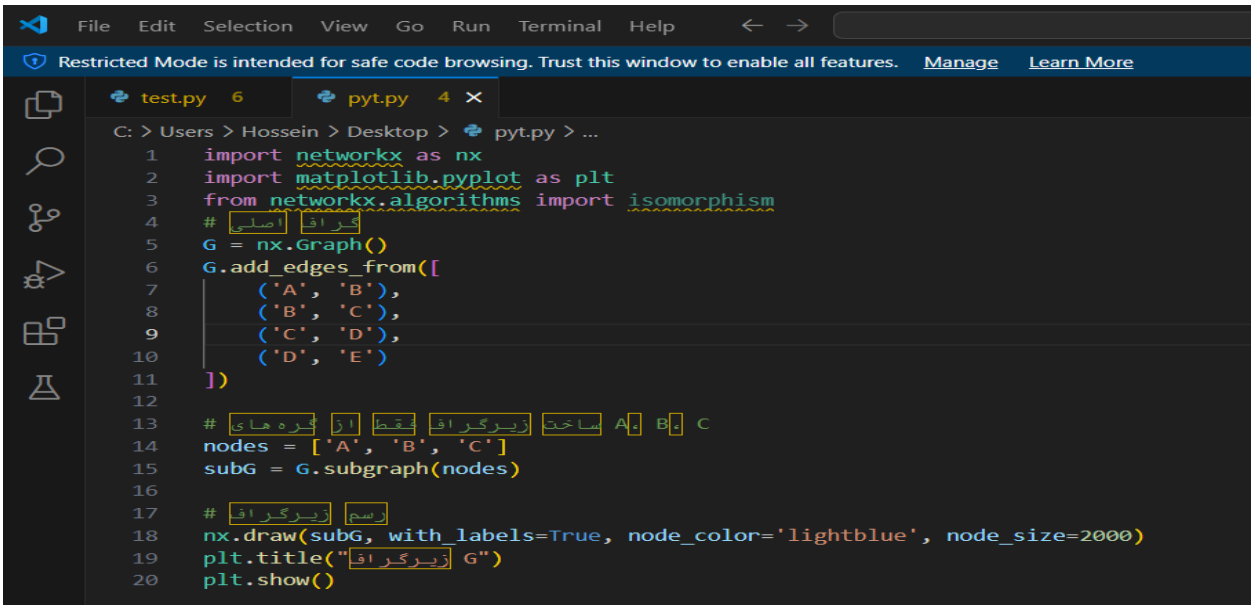


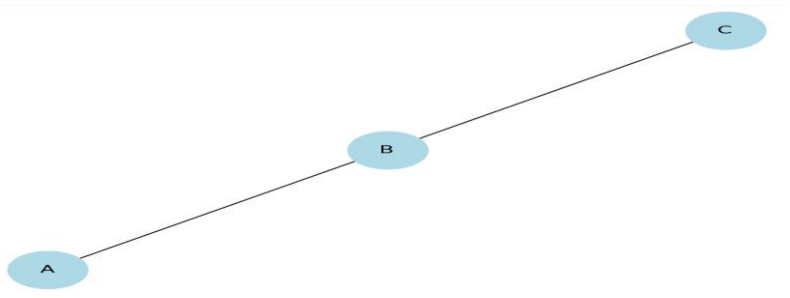
هم بند بودن یا نبودن : بین هر دو گره آن، حداقل یک مسیر وجود دارد.

```python
G = nx.Graph()
G.add_edges_from([
    (1, 2), (2, 3), (3, 4)
])

print("آیا گراف همبند است؟", nx.is_connected(G))


DG = nx.DiGraph()
DG.add_edges_from([
    (1, 2), (2, 3), (3, 1)
])

print("آیا گراف قویاً همبند است؟", nx.is_strongly_connected(DG))
print("آیا گراف ضعیفاً همبند است؟", nx.is_weakly_connected(DG))
```

خروجی true و false

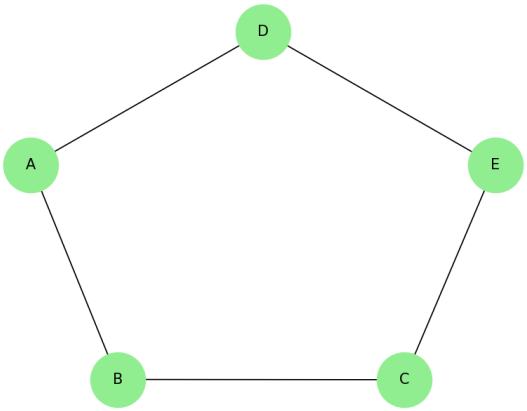
- زیر گراف : یک بخش از گراف اصلی که شامل بعضی از گره‌ها و یال‌های بین آن‌ها است.





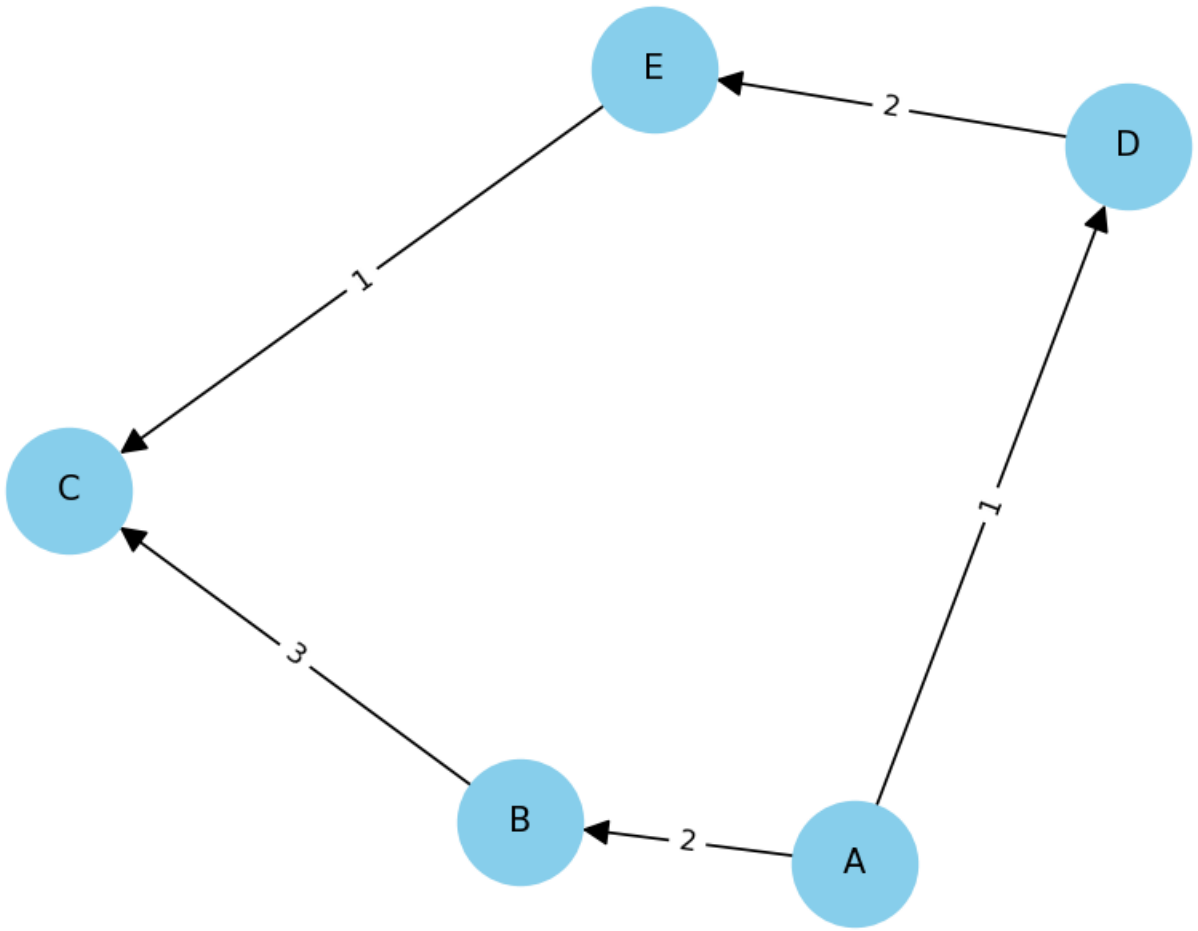
- کوتاه ترین مسیر در گراف بدونه جهت (برابر بودن وزن همه یال ها)

```python
# ساخت گراف بدون جهت
G = nx.Graph()
G.add_edges_from([
    ('A', 'B'),
    ('B', 'C'),
    ('A', 'D'),
    ('D', 'E'),
    ('E', 'C')
])

# رسم گراف
pos = nx.spring_layout(G)
nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000)
plt.title("گراف بدون جهت")
plt.axis('off')
plt.show()

# یافتن کوتاه‌ترین مسیر بین A و C
path = nx.shortest_path(G, source='A', target='C')
length = nx.shortest_path_length(G, source='A', target='C')

print("کوتاه‌ترین مسیر از A به C:", path)
print("طول مسیر:", length)
```



کوتاه‌ترین مسیر در گراف جهت‌دار (Directed Graph): یعنی مسیری با کمترین تعداد یال (در گراف بدون وزن) یا کمترین مجموع وزن (در گراف وزن‌دار)، با رعایت جهت‌ها.

```python
# ساخت گراف جهت‌دار وزن‌دار
G = nx.DiGraph()
G.add_weighted_edges_from([
    ('A', 'B', 2),
    ('A', 'D', 1),
    ('B', 'C', 3),
    ('D', 'E', 2),
    ('E', 'C', 1)
])

# رسم گراف
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='skyblue', node_size=2000, arrows=True, arrowsize=20)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)

plt.title("گراف جهت‌دار وزن‌دار")
plt.axis('off')
plt.show()

# محاسبه کوتاه‌ترین مسیر از A به C
path = nx.shortest_path(G, source='A', target='C', weight='weight')
length = nx.shortest_path_length(G, source='A', target='C', weight='weight')

print("کوتاه‌ترین مسیر از A به C:", path)
print("طول مسیر:", length)
```



- درجه رأس (Degree of a Node): تعداد یال‌هایی که به آن متصل است.

```python
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'),
    ('A', 'C'),
    ('B', 'C'),
    ('C', 'A'),
])

# نمایش درجه ها
for node in G.nodes:
    print(f"{node}: in-degree = {G.in_degree(node)}, out-degree = {G.out_degree(node)}")
```

```
PS C:\Users\Hossein\desktop>  python pyt.py
A: in-degree = 1, out-degree = 2
B: in-degree = 1, out-degree = 1
C: in-degree = 2, out-degree = 1
PS C:\Users\Hossein\desktop>
```

خروجی :

- الگوریتم دایکسترا (Dijkstra’s Algorithm) : پیدا کردن کوتاه‌ترین مسیر از یک گره به بقیه گره‌ها در گراف وزن‌دار بدون وزن منفی.

```python
G = nx.DiGraph()
G.add_weighted_edges_from([
    ('A', 'B', 2),
    ('B', 'C', 3),
    ('A', 'D', 1),
    ('D', 'C', 1)
])

path = nx.dijkstra_path(G, source='A', target='C')
length = nx.dijkstra_path_length(G, source='A', target='C')

print("مسیر:", path)
print("طول مسیر:", length)
```

- مسیر اویلری (Eulerian Path): مسیری که هر یال گراف دقیقاً یک‌بار طی بشه (ممکنه گره‌ها تکرار بشن)

- مدار اویلری (Eulerian Circuit): مسیری که هر یال دقیقاً یک‌بار طی بشه و مبدأ و مقصد یکی باشن

- شرط وجود در گراف بدون جهت:

مدار اویلری:
تمام رئوس باید درجه زوج داشته باشن
مثلاً: 2، 4، 6 ...

مسیر اویلری (اما نه مدار):
دقیقاً دو رأس درجه فرد داشته باشن
بقیه درجه زوج باشن.

```
iG.add_edges_from([
    ('A', 'B'),
    ('B', 'C'),
    ('C', 'D'),
    ('D', 'A'),
    ('A', 'C')
])

print("مدار اویلری؟", nx.is_eulerian(iG))  # مدار؟
print("مسیر اویلری؟", nx.has_eulerian_path(iG))  # مسیر؟
```

- شرط وجود در گراف جهت‌دار:

مدار اویلری:
برای هر رأس: درجه ورود = درجه خروج

مسیر اویلری:

دقیقاً یک رأس باشه که:

درجه خروج = درجه ورود + 1 (شروع مسیر)
و دقیقاً یک رأس که:

درجه ورود = درجه خروج + 1 (پایان مسیر)

```python
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'),
    ('B', 'C'),
    ('C', 'D'),
    ('D', 'A')
])
nx.draw(G, with_labels=True, node_color='lightblue', node_size=2000, arrows=True, arrowsize=20)
plt.show()

# اویلری مدار بررسی
if nx.is_eulerian(G):
    print("✅ دارد اویلری مدار")
    for u, v in nx.eulerian_circuit(G):
        print(f"{u} → {v}")
# اویلری مسیر بررسی
elif nx.has_eulerian_path(G):
    print("✅ دارد اویلری مسیر") ···
else:
    print("❌ دارد اویلری مدار نه و مسیر نه")
```

- دور هامیلتیونی (Hamiltonian Cycle)

تعریف: دور هامیلتیونی (یا چرخه هامیلتیونی)، مسیر بسته‌ای است که:
از هر رأس دقیقاً یک‌بار عبور می‌کند،و در نهایت به رأس شروع برمی‌گردد.

- مسیر هامیلتیونی (Hamiltonian Path)

تعریف: مسیر هامیلتیونی، مسیری در یک گراف است که از هر رأس دقیقاً یک‌بار عبور کند (بدون تکرار رأس‌ها)، اما لازم نیست به رأس شروع برگردد.

```
# ساخت گراف
G = nx.Graph()
G.add_edges_from([
    (1, 2), (2, 3), (3, 4),
    (4, 1), (1, 3)
])

nodes = list(G.nodes)

print("بررسی مسیرها و مدارهای هامیلتونی:\n")

for path in permutations(nodes):
    # بررسی مسیر هامیلتونی
    is_path = True
    for i in range(len(path) - 1):
        if not G.has_edge(path[i], path[i+1]):
            is_path = False
            break

    if is_path:
        print("✅ مسیر هامیلتونی:", path)

        # بررسی اینکه آیا مسیر به رأس اول برمی‌گردد و دور هامیلتونی است
        if G.has_edge(path[-1], path[0]):
            print("🔁 ➡ این مسیر یک دور هامیلتونی** است:", path + (path[0],))
```

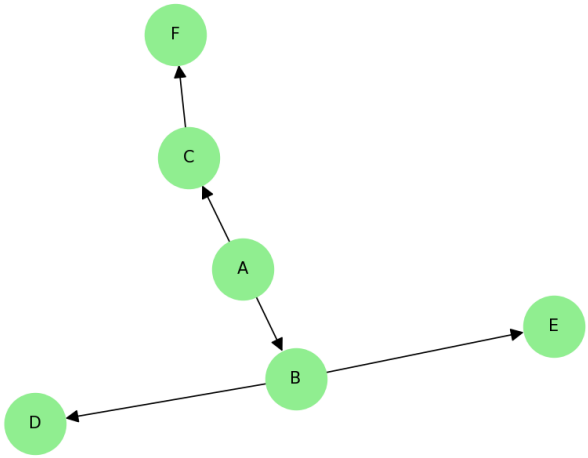
- برای گراف‌ها، ماتریس هم‌سایگی (Adjacency Matrix) یک نمایش ماتریسی از ارتباط بین گره‌هاست.

```
6   # ساخت گراف
7   G = nx.DiGraph()
8   G.add_edges_from([
9       ('A', 'B'),
10      ('B', 'C'),
11      ('C', 'D'),
12      ('D', 'A')
13  ])
14
15  # گرفتن ماتریس هم‌سایگی به صورت آرایه NumPy
16  adj_matrix = nx.to_numpy_array(G, dtype=int, nodelist=sorted(G.nodes()))
17
18  # چاپ ماتریس
19  print("ماتریس هم‌سایگی:")
20  print(adj_matrix)
```

```
PS C:\Users\Hossein\desktop>
ی‌گی اسم ه  سی‌رت ام:
[[0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]
 [1 0 0 0]]
```

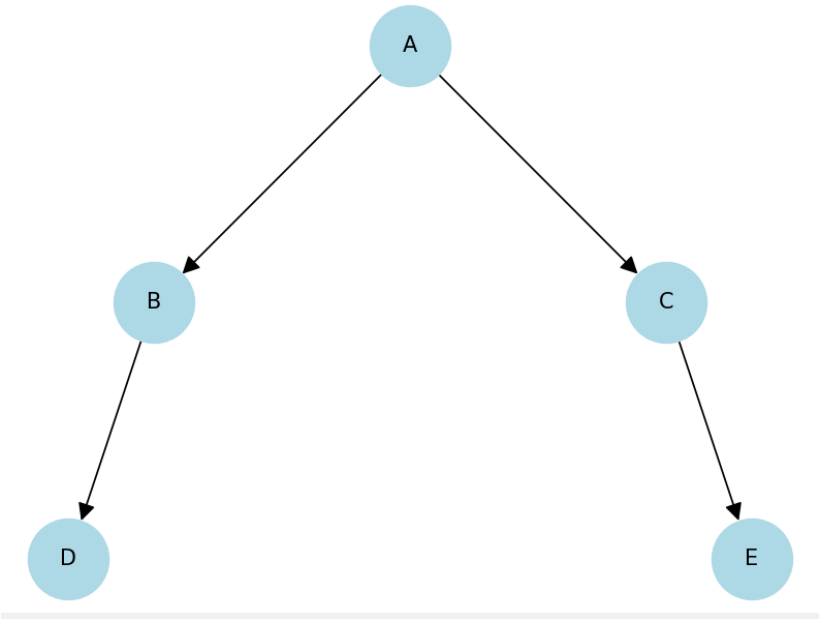
- Directed Tree (درخت جهت‌دار) : درخت جهت‌دار یک نوع گراف جهت‌دار است که ویژگی‌های زیر را دارد:
  1. یک گره ریشه (Root) دارد که هیچ یال ورودی ندارد.
  2. از ریشه می‌توان با دنبال کردن یال‌ها به تمام گره‌های دیگر رسید.a
  3. گراف بدون دور (cycle) است.
  4. هر گره (غیر از ریشه) دقیقاً یک یال ورودی دارد.

```python
# جهت دار درخت ساخت
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'),
    ('A', 'C'),
    ('B', 'D'),
    ('B', 'E'),
    ('C', 'F')
])

# درخت رسم
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000, arrows=True, arrowsize=20)
plt.title("جهت دار درخت (Directed Tree)")
plt.axis('off')
plt.show()
```



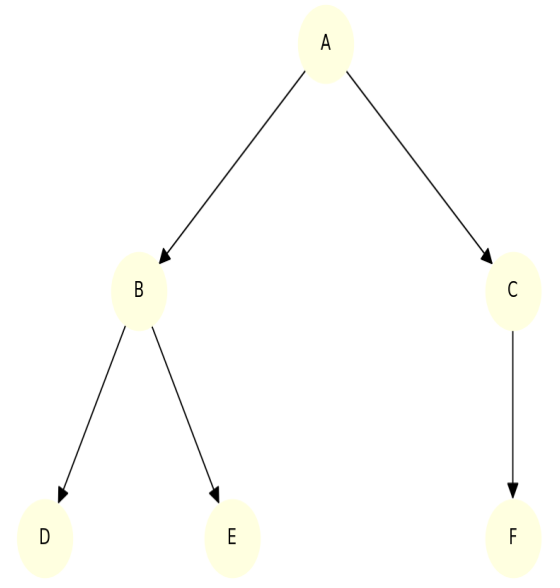
- Binary Tree (درخت دودویی) چیست؟ درخت دودویی ساختاری درختی است که:
  1. هر گره می‌تواند حداکثر دو فرزند داشته باشد:
  2. یک فرزند چپ (Left Child)
  3. یک فرزند راست (Right Child)

```python
# ساخت درخت دودویی
G = nx.DiGraph()
edges = [
    ('A', 'B'), ('A', 'C'),
    ('B', 'D'),
    ('C', 'E')
]
G.add_edges_from(edges)

# موقعیت دلخواه برای ظاهر دودویی
pos = {
    'A': (0, 3),
    'B': (-1.5, 2), 'C': (1.5, 2),
    'D': (-2, 1), 'E': (2, 1)
}

# رسم درخت
nx.draw(G, pos, with_labels=True, node_color='lightblue', node_size=2000, arrows=True, arrowsize=20)
plt.title("درخت دودویی (Binary Tree)")
plt.axis('off')
plt.show()
```



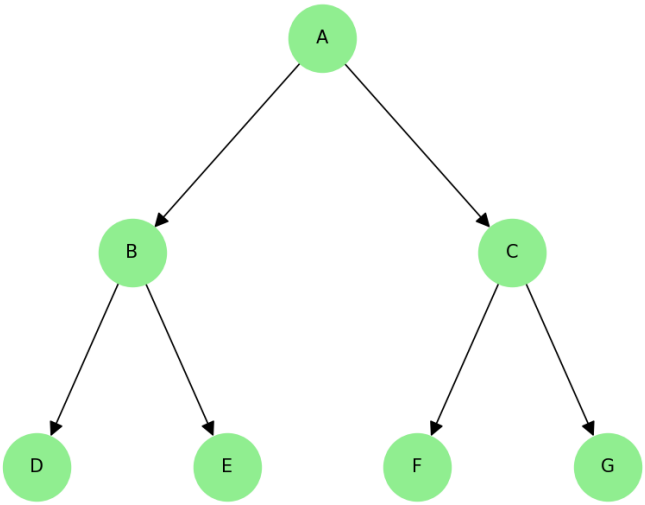
- درخت مرتب (Ordered Tree): نوعی درخت ریشه‌دار هست که در اون
  1. هر گره می‌تونه چند فرزند داشته باشه
  2. ترتیب فرزندان اهمیت داره (یعنی فرق هست بین فرزندان [B, C] و [C, B])

```python
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'), ('A', 'C'),
    ('B', 'D'), ('B', 'E'),
    ('C', 'F')
])

# موقعیت دستی برای حفظ ترتیب
pos = {
    'A': (0, 2),
    'B': (-1, 1), 'C': (1, 1),
    'D': (-1.5, 0), 'E': (-0.5, 0),
    'F': (1, 0)
}

nx.draw(G, pos, with_labels=True, node_color='lightyellow', node_size=2000, arrows=True, arrowsize=20)
plt.title("Ordered Tree (درخت مرتب)")
plt.axis('off')
plt.show()
```



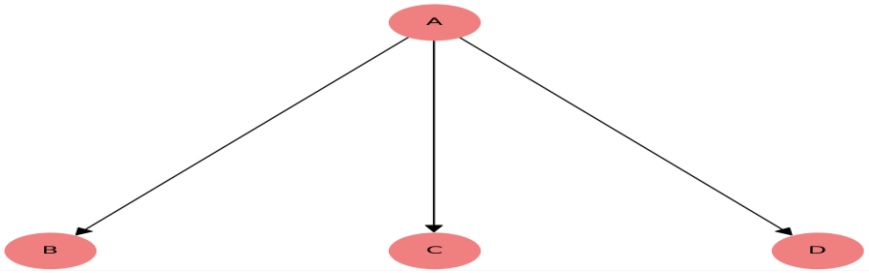
- درخت دودویی کامل (Full Binary Tree) نوعی درخت دودوییه که در اون:
  1. هر گره یا دقیقاً دو فرزند داره یا هیچ فرزندی نداره.
  2. یعنی هیچ گره‌ای فقط یک فرزند نداره.

```python
G = nx.DiGraph()
edges = [
    ('A', 'B'), ('A', 'C'),
    ('B', 'D'), ('B', 'E'),
    ('C', 'F'), ('C', 'G')
]
G.add_edges_from(edges)

# موقعیت گره‌ها برای رسم تمیز
pos = {
    'A': (0, 3),
    'B': (-2, 2), 'C': (2, 2),
    'D': (-3, 1), 'E': (-1, 1),
    'F': (1, 1), 'G': (3, 1)
}

# رسم گراف
nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000, arrows=True, arrowsize=20)
plt.title("Full Binary Tree (درخت دودویی کامل)")
plt.axis('off')
plt.show()
```



- Positional Tree یا درخت موقعیتی نوعی درخت مرتب (Ordered Tree) هست که:

علاوه بر ترتیب بین فرزندان، موقعیت دقیق هر فرزند در یک مکان خاص (Position) تعریف شده.

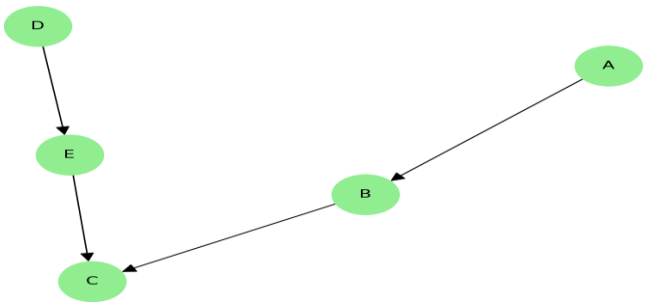


```python
G = nx.DiGraph()
edges = [
    ('A', 'B'),  # pos 1
    ('A', 'C'),  # pos 2
    ('A', 'D')   # pos 3
]
G.add_edges_from(edges)

pos = {
    'A': (0, 2),
    'B': (-2, 1),
    'C': (0, 1),
    'D': (2, 1)
}

nx.draw(G, pos, with_labels=True, node_color='lightcoral', node_size=2000, arrows=True, arrowsize=20)
plt.title("Positional Tree (درخت موقعیتی)")
plt.axis('off')
plt.show()
```

- یک گراف جهت‌دار (Directed Graph) وقتی متصل ضعیف (Weakly Connected) محسوب می‌شه که:

1. اگر جهت یال‌ها را نادیده بگیریم، گراف بدون جهت حاصل، متصل باشد.
2. یعنی شاید از A به B مسیر مستقیم نباشه، اما اگه جهت‌ها رو حذف کنیم، همه گره‌ها به هم وصل باشن.

```python
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'),
    ('B', 'C'),
    ('E', 'C'),
    ('D', 'E')
])

# رسم گراف
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_color='lightgreen', node_size=2000, arrows=True, arrowsize=20)
plt.title("گراف جهت‌دار متصل ضعیف")
plt.axis('off')
plt.show()

# بررسی اتصال ضعیف
is_weakly = nx.is_weakly_connected(G)
print("آیا گراف متصل ضعیف است؟", is_weakly)
```



- دو گراف جهت‌دار (DiGraph) رو یک‌ریخت (isomorphic) می‌گیم اگر:

بتونیم برچسب گره‌ها رو بازچینش کنیم طوری که ساختار یال‌ها دقیقاً یکسان باقی بمونه (با جهت‌ها).

- **شرط‌ها:**

1. دو گراف جهت‌دار G و H یک‌ریخت هستن اگر:
2. تعداد گره‌ها برابر باشه
3. جهت و اتصال یال‌ها یکی باشه
4. فقط اسم گره‌ها فرق کنه

```
# ساخت دو گراف جهت‌دار
G1 = nx.DiGraph()
G1.add_edges_from([('A', 'B'), ('B', 'C'), ('A', 'D'), ('D', 'E'), ('E', 'C')])

G2 = nx.DiGraph()
G2.add_edges_from([('X', 'Y'), ('Y', 'Z'), ('X', 'W'), ('W', 'Q'), ('Q', 'Z')])

# بررسی یکریختی
GM = isomorphism.DiGraphMatcher(G1, G2)
print("آیا گرافها یکریخت هستند؟", GM.is_isomorphic())
```

خروجی : true وfalse

گراف یک ریخت غیر جهت دار نیز همین طور است .

- گراف همبند قوی (Strongly Connected Graph) چیست؟ گرافی که بتوان از هر راس آن به راس های دیگر رفت و دسرسی داشت برعکس همبند ضعیف .

```
G = nx.DiGraph()
G.add_edges_from([
    ('A', 'B'),
    ('B', 'C')
])

print("همبند قوی؟", nx.is_strongly_connected(G))   # False
print("همبند ضعیف؟", nx.is_weakly_connected(G))   # True
```

- گراف جهت دار متصل یک طرفه:همان گراف همبند ضعیف میتواند باشد یعنی از یک راس میتوان به راس دیگه رفت اما برگشتی ندارد فقط از یک رفت وصل است .

- گراف دوبخشی یعنی:

  می‌تونی همهٔ رأس‌ها (نقطه‌ها) رو به دو دسته تقسیم کنی

  طوری که هیچ یالی (خطی) بین دو رأس از یه دسته نباشه.

```
G = nx.Graph()
G.add_edges_from([
    ('A', '1'),
    ('A', '2'),
    ('B', '2'),
    ('B', '3')
])

print(nx.is_bipartite(G))  # خروجی: True
```

- هدف الگوریتم کراسکال:

1. پیدا کردن یک درخت پوشای کمینه (MST)

→ یعنی کمترین مجموعه‌ی یال‌ها که همه رأس‌ها رو وصل کنه و دور نداشته باشه.

➢ مراحل اجرای الگوریتم کراسکال:

مرتب کردن یال‌ها بر اساس وزن:

A-B → 1

B-C → 2

A-C → 3

ساخت MST (درخت پوشا):

مرحله 1:

A-B رو انتخاب می‌کنیم ← وزن = 1

(A و B متصل شدن)

مرحله 2:

B-C رو انتخاب می‌کنیم ← وزن = 2

(C هم به جمع وصل می‌شه)

مرحله 3:

A-C وزنش 3 هست، ولی اگر انتخابش کنیم، یه دور ایجاد می‌شه پس رد می‌کنیم.

نتیجه نهایی:

درخت پوشای کمینه شامل یال‌های:

A-B (وزن 1)

B-C (وزن 2)

مجموع وزن = 1 + 2 = 3

```python
edges = [('A', 'B', 1), ('B', 'C', 2), ('A', 'C', 3)]
nodes = ['A', 'B', 'C']
class UnionFind:
    def __init__(self, nodes):
        self.parent = {n: n for n in nodes}

    def find(self, node):
        if self.parent[node] != node:
            self.parent[node] = self.find(self.parent[node])
        return self.parent[node]

    def union(self, u, v):
        ru, rv = self.find(u), self.find(v)
        if ru != rv:
            self.parent[rv] = ru
            return True
        return False

def kruskal(nodes, edges):
    uf = UnionFind(nodes)
    edges.sort(key=lambda x: x[2])
    mst = []

    for u, v, w in edges:
        if uf.union(u, v):
            mst.append((u, v, w))
    return mst

mst = kruskal(nodes, edges)
print("درخت پوشای کمینه:", mst)
```