ĐỒ THỊ VÀ THUẬT GIẢI TRÊN ĐỒ THỊ

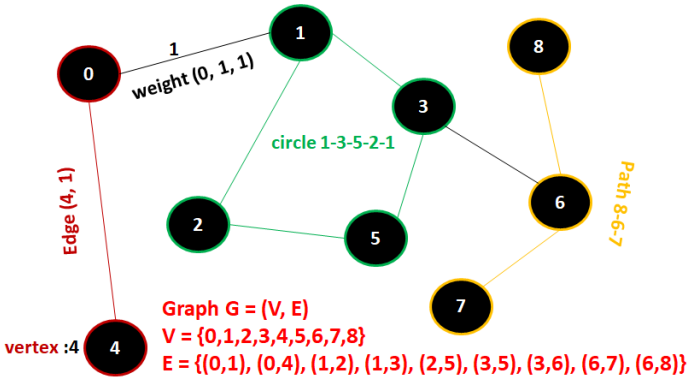
1. Khái niệm và Định nghĩa

Đỉnh – vertex. Một đỉnh, hay còn gọi là nút, là một thành phần cơ bản của đồ thị. Đỉnh có thể được đặt tên gợi nhớ, hay đơn giản chỉ là đánh số, gọi là "*khóa*". Một đỉnh cũng có thể có thông tin bổ sung. Ta gọi thông tin bổ sung này là "*tải trọng*".

Cạnh – edge. Một cạnh, hay còn gọi là cung, là một thành phần cơ bản khác của đồ thị. Một cạnh (v, w) nối hai đỉnh v và w cho biết giữa chúng có mối quan hệ. Các cạnh có thể là một chiều hoặc hai chiều. Nếu mọi cạnh trong đồ thị đều là một chiều, nghĩa là (v, w) khác (w, v), ta nói đồ thị có hướng, hoặc đơn giản là đồ thị, ngược lại là đồ thị vô hướng, (v, w) ≡ (w, v).

Trọng số - weight. Các cạnh có thể có trọng số cho biết "*chi phí*" đi từ đỉnh này sang đỉnh khác. Ví dụ như trong đồ thị nối sân bay này với sân bay khác, trọng số trên cạnh có thể biểu thị khoảng cách hay thời gian bay giữa hai sân bay.

Đồ thị - graph. Một đồ thị G là một bộ đôi G = (V, E), với V là tập các đỉnh và E là tập các cạnh. Mỗi cạnh là một bộ (v, w), w, v ∈ V. Có thể thêm thành phần thứ ba vào bộ (v, w, d) biểu diễn trọng số của cung (v, w). Đồ thị con s, s = (v, e), là tập hợp các cạnh e và đỉnh v sao cho e ⊂ E và v ⊂ V.



Đường đi hay lộ trình – path. Đường đi trong đồ thị là một dãy các đỉnh được nối với nhau qua các cạnh. Về hình thức, một đường đi path = [w_1, w_2, ..., w_n] sao cho (w_i, w_{i+1}) ∈ E với mọi i = 1,…, n – 1. Độ dài đường đi trong đồ thị không trọng số là số cạnh trong đường đi, ở đây là n – 1. Độ dài đường đi trong đồ thị có trọng số là tổng trọng số của tất cả các cạnh trong đường đi.

Chu trình - circle. Chu trình trong đồ thị là một đường đi bắt đầu và kết thúc tại cùng một đỉnh. Đồ thị không có chu trình được gọi là đồ thị phi chu trình (acyclic graph). Đồ thị có hướng không có chu trình

được gọi là đồ thị phi chu trình có hướng - DAG (directed acyclic graph).

2. Kiểu dữ liệu trừu tượng đồ thị

Kiểu dữ liệu trừu tượng đồ thị - Graph ADT (Graph Abstract Data Type) được định nghĩa với các phương thức như sau.

. Graph(): phương thức tạo một đồ thị mới, rỗng.

. addVertex(v): phương thức thêm đỉnh v vào đồ thị.

. addEdge(v, w): thêm cạnh, có hướng vào đồ thị nối hai đỉnh v, w.

. addEdge(w, w, d): thêm cạnh (v, w,d), có trọng số d, có hướng vào đồ thị nối hai đỉnh v, w.

. getVertex(v): tìm đỉnh trong đồ thị có khoá hay nhãn là v.

. getVertices(): trả về danh sách tất cả các đỉnh trong đồ thị.

. in: trả về True cho một câu lệnh của đỉnh có ngữ nghĩa dạng "vertex in graph", nếu đỉnh đã cho nằm trong đồ thị, False ngược lại.

Từ định nghĩa hình thức đồ thị với các thành phần của đồ thị như mô tả trên, có hai cách cài đặt kiểu dữ liệu trừu tượng đồ thị (GraphADT) trong Python: *ma trận kề* và *danh sách kề*.

2.1. Cấu trúc dữ liệu cơ sở

2.1.1. Ma trận kề - **adjacent matrix**

Cách đơn giản nhất để cài đặt đồ thị là sử dụng mảng hai chiều để biểu diễn *ma trận kề* (adjacency matrix). Trong cài đặt dùng ma trận kề, mỗi hàng i và cột i đại diện cho đỉnh nhãn i trong đồ thị. Giá trị được lưu trong ô (v, w), là giao điểm của hàng v và cột w cho biết có cạnh từ đỉnh v đến đỉnh w (khác 0) hay không (bằng 0). Khi hai đỉnh v vàw được nối bởi một cạnh (ô (v, w) khác 0), ta nói v và w kề nhau.

Cài đặt dùng ma trận kề sẽ là giải pháp tốt nếu đồ thị có số lượng cạnh lớn. Trong biểu diễn bằng ma-trận kề, số cạnh tối đa có thể có là $|V|^2$. Tuy nhiên, trong thực tế, số cạnh thường nhỏ hơn $|V|^2$ rất nhiều, và ma-trận thường là ma-trận thưa (với rất nhiều số 0 thể hiện không có cạnh nối 2 đỉnh tương ứng với ô đó).

2.1.2. Danh sách kề - **adjacent list**

Cách khác hiệu quả hơn về không gian là sử dụng *danh sách kề* (adjacency list).

Trong cài đặt dùng danh sách kề, có một danh sách giữ tất cả các đỉnh trong đồ thị và mỗi đỉnh ***Vertex*** liên kết với một danh sách giữ các đỉnh khác kề với nó trong một cấu trúc từ điển với khóa là đỉnh và giá trị là trọng số của cung đó (nếu đồ thị không có trọng thì giá trị bằng 1). Ưu điểm của việc dùng danh sách kề là biểu diễn tiết kiệm không gian lưu trữ khi đồ thị thưa. Danh sách kề cũng giúp thuận tiện khi tìm tất cả các cung kết nối trực tiếp với một đỉnh.

2.2. Cài đặt

Ta sử dụng từ điển để cài đặt danh sách kề. Kiểu dữ liệu trừu tượng **Graph** sẽ có hai lớp

· ***Graph*** – chứa danh sách các đỉnh, và

· ***Vertex*** - cho mỗi đỉnh trong đồ thị.

Mỗi ***Vertex*** là một từ điển tên ***connectTo*** giữ các đỉnh khác kề với nó và trọng số của cạnh tương ứng.

· Hàm khởi tạo – ***init***, chỉ thiết lập ***id***, gồm một ***string*** là ***key*** được truyền cho và một từ điển ***connectedTo***.

· Phương thức ***addNeighbor*** được dùng khi cần thêm một cung vào đồ thị.

· Phương thức ***getConnections*** trả về tất cả các đỉnh trong danh sách kề, biểu diễn bởi biến ***connecTo***.

· Phương thức ***getWeight*** trả về trọng số của cạnh được truyền theo tham số.

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo)

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
```

```
        return self.connectedTo[nbr]
```

Lớp ***Graph*** gồm một từ điển kết buộc tên đỉnh vào đối tượng đỉnh. Graph cũng cung cấp các phương thức để thêm một đỉnh vào đồ thị và nối đỉnh này với đỉnh khác.

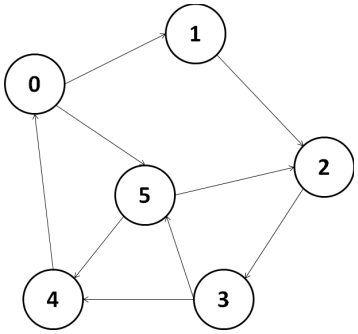
· Phương thức ***getVertices*** trả về tên nhãn tất cả các đỉnh trong đồ thị.

· Ngoài ra, phương thức ***__iter__*** dùng để duyệt qua các đối tượng đỉnh trong đồ thị.

Hai phương thức này cho phép duyệt các đỉnh theo tên hoặc theo đối tượng.

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

Ví dụ. Tạo một đồ thị 6 đỉnh, đánh nhãn từ 0 đến 5, như hình vẽ dưới, rồi hiển thị các đỉnh đó.




```
( 0 , 1 )
( 0 , 5 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```

· Lưu ý rằng đối với mỗi khóa từ 0 đến 5, tạo một thể hiện của Vertex.

· Tiếp theo, thêm các cạnh cho đồ thị.

· Cuối cùng, hiển thị đồ thị.

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None
```

```
    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
#main
g = Graph()
for i in range(6):
  g.addVertex(i)
g.addEdge(0,1,5)
g.addEdge(0,5,2)
g.addEdge(1,2,4)
g.addEdge(2,3,9)
g.addEdge(3,4,7)
g.addEdge(3,5,3)
g.addEdge(4,0,1)
g.addEdge(5,4,8)
g.addEdge(5,2,1)
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
```

Bây giờ, bạn hãy hoàn thiện cấu trúc dữ liệu trừu tượng đồ thị với các phương thức như mô tả cũng như các phương thức khác bạn cho là cần thiết cho các úng dụng của bạn.

3. Bài toán minh hoạ

Ta sẽ tìm hiểu một số thuật toán quan trọng trên đồ thị qua *trò chơi thang từ*, một mở rộng *bài toán mê cung* ta đã biết.

Ở bài toán thang từ này, ta được cho 2 từ: từ ***nguồn*** và từ ***đích***. Thay lần lần các chữ cái để từ ***nguồn*** đến được ***đích*** theo nguyên tắc mỗi lần chỉ được thay 1 chữ cái trong từ hiện tại sao cho từ mới cũng có nghĩa (từ điển). Chẳng hạn, chuyển từ "FOOL" thành từ "SAGE", ta có dãy:

FOOL □ POOL □ POLL □ POLE □ PALE □ SALE □ SAGE.

Rõ ràng, đây là bài toán tìm kiếm hướng đích mà ta đã biết. Ta sẽ giải bằng cách sử dụng đồ thị. Quy trình chung giải một bài toán dùng đồ thị như sau.

Bước 1: *Biểu diễn* bài toán dưới dạng đồ thị.

Bước 2: Sử dụng thuật toán tìm kiếm thích hợp trên đồ thị.

3.1. Xây dựng đồ thị thang từ

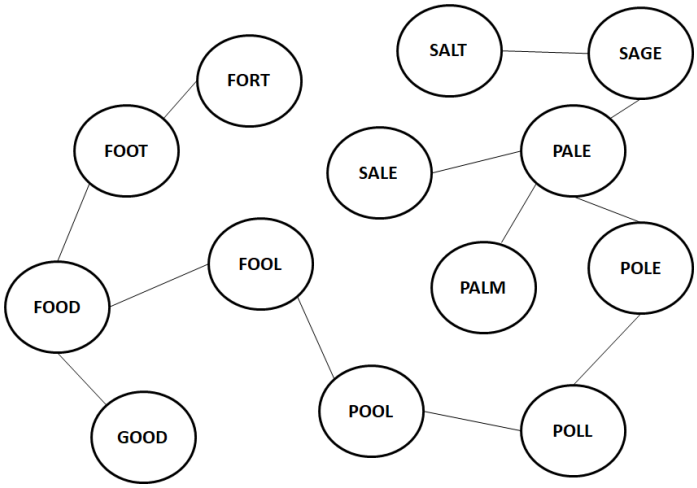
Ta cần xây dựng một đồ thị với các đỉnh là các từ có nghĩa, giữa 2 từ chỉ khác nhau một chữ cái sẽ có 1 cạnh nối. Nếu xây dựng được đồ thị như vậy, bất kỳ đường đi nào từ từ nguồn sang từ đích đều là một lời giải cho trò chơi thang từ. Đây là đồ thị vô hướng và không có trọng số.

Không mất tính tổng quát, ta giả định rằng ta đã có danh sách các từ có cùng độ dài. Để bắt đầu, ta tạo một đỉnh là một từ trong danh sách. Rồi so sánh từng từ trong danh sách với nhau. Nếu hai từ chỉ khác nhau đúng một chữ cái, một cạnh nối chúng được thêm vào đồ thị. Nếu số từ trong danh sách lớn, tiếp cận này không hiệu quả. Đại khái, việc so sánh một từ với mọi từ khác trong danh sách là một thuật toán $O(n^2)$. Như vây, nếu có 5.100 từ, n^2 là hơn 26 triệu so sánh. May mắn thay, số các từ có chiều dài cố định trong bất kỳ từ điển nào đều không quá lớn, vì thế, cách giải này khả thi.

Ta sẽ giải bài toán cho trường hợp tổng quát. Trước hết, phân hoạch từ điển thành các nhóm từ có cùng chiều dài. Để minh hoạ, giả sử có các nhóm, mỗi nhóm có một từ gồm bốn chữ cái. Nhãn danh sách từ điển được khởi tạo là 1 từ có 4 chữ cái trong đó có một chữ cái được thay thế bằng ký tự đại diện '_'. Khi xử lý mỗi từ, ta so sánh từ đó với từng nhóm, theo cách đó cả "pope" và "pops" sẽ khớp với "pop_", hay "pope" và "pole" sẽ khớp với "po_e". Khi tìm thấy nhóm phù hợp, đặt từ vào nhóm đó. Khi tất cả các từ đặt trong các nhóm thích hợp, ta biết rằng mọi từ trong nhóm phải được nối với nhau.

Trong Python, ta có thể cài đặt ý tưởng trên bằng cách sử dụng từ điển. Các nhãn của nhóm là các khóa trong từ điển. Giá trị tương ứng với khóa đó là danh sách các từ. Khi xây dựng từ điển xong, ta có thể tạo đồ thị. Bắt đầu tạo một đỉnh cho mỗi từ trong đồ thị. Sau đó, tạo các cạnh giữa tất cả các đỉnh mà các từ cùng khóa trong từ điển.

Ví dụ. Xây dựng đồ thị thang từ như hình dưới.



FOOD
FOOT
FOOL
FORT
GOOD
PALE
PALM
POLE
POLL
POOL
SAGE
SALE
SALT

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]

class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
```

```
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
#Build word graph
def buildGraph(wList):
    d = {}
    g = Graph()
    #phân hoạch các từ cùng độ dài chỉ khác nhau 1 ký tự
    for line in wList: #lấy từng từ trong từ điển
        word = line[:-1]
        for i in range(len(word)):
            bucket = word[:i] + '_' + word[i+1:]
            if bucket in d:
                d[bucket].append(word)
            else:
                d[bucket] = [word]
    #thêm các đỉnh và các cạnh cho các từng trong cùng bucket
    for bucket in d.keys():
        for word1 in d[bucket]:
            for word2 in d[bucket]:
                if word1 != word2:
                    g.addEdge(word1,word2)
    return g

#main
wList = ["FOOD", "FOOT", "FOOL", "FORT",
        "GOOD",
        "PALE", "PALM", "POLE", "POLL", "POOL",
        "SAGE", "SALE", "SALT"]
g = buildGraph(wList)

for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
```

Trở lại trò chơi thang từ: nhập hai từ nguồn và đích, ta sẽ thiết kế một số thuật toán giải ***bài toán tìm đường đi*** từ từ nguồn đến từ đích được cho.

3.2. Tìm kiếm theo chiều rộng

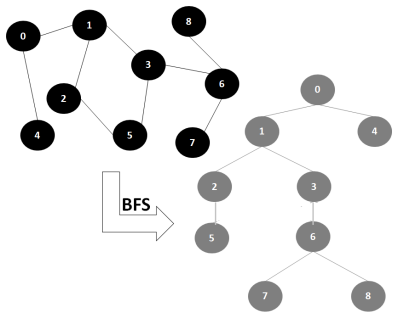
Với đồ đã xây dựng, thuật toán đầu tiên được thiết kế để tìm ra *lời giải ngắn nhất*, là số bước chuyển ít nhất từ từ nguồn tới từ đích, cho bài toán thang từ là "tìm kiếm theo chiều rộng" – ***BFS*** (Breadth First Search), một trong những thuật toán tìm kiếm trên đồ thị trực quan, dễ hiểu.

Cho một đồ thị G và một đỉnh xuất phát s, BFS tiến hành bằng cách khám phá các cạnh trong đồ thị để tìm tất cả các đỉnh trong G có đường đi từ s đến. Ta có thể hình dung thuật toán BFS *đang* xây dựng một cây, cao dần theo cấp độ. BFS thêm tất cả các "nút con" của nút s trước khi xem xét đến "nút cháu".

Để kiểm soát tiến trình xây dựng cây, BFS sẽ tô màu mỗi đỉnh bằng màu trắng, xám hay đen. Tất cả các đỉnh được khởi tạo màu trắng. Đỉnh trắng là đỉnh chưa được khám phá. Khi một đỉnh được phát hiện lần đầu, nó có màu xám, và khi BFS đã khám phá trọn vẹn một đỉnh, nó tô màu đen. Điều này có nghĩa là khi một đỉnh có màu đen, không có đỉnh trắng nào liền kề với nó, còn một nút màu xám có thể có một số đỉnh màu trắng liền kề với nó, cho biết vẫn còn những đỉnh kề để khám phá. BFS cũng sử dụng thêm 2 biến, một để giữ giá trị là khoảng cách từ đỉnh start đến đỉnh s đang xét, và một để đánh dấu đỉnh s vừa được đi đến từ đỉnh nào.

Xuất phát từ đỉnh s = *start*, là đỉnh nguồn được cho, BFS tô *start* màu xám, đánh dấu *start* đang được xem xét. Hai giá trị khoảng cách và đỉnh trước được khởi tạo tương ứng là 0 và *none*. Đưa đỉnh s = *start* vào hàng đợi. Tiến trình tìm kiếm bắt đầu với các đỉnh trong hàng đợi như sau: BFS khám phá từng nút ở đầu hàng đợi qua danh sách kề của nút *currentVert*. Khi kiểm tra nút *nbr* trong danh sách kề, nếu là màu trắng (white), đỉnh chưa được khám phá thì thực hiện các bước chính của BFS.

(BFS - các bước chính)
(1) Nút *nbr* mới, chưa được khám phá, sẽ được tô màu xám.
(2) Nút trước của *nbr* được chọn làm nút hiện *currentVert*
(3) Khoảng cách đến *nbr* được tăng 1, *currentVert* + 1
(4) *nbr* được thêm vào cuối hàng đợi.



Đoạn mã sau sử dụng biểu diễn đồ thị bằng danh sách kề. Sử dụng thêm một *hàng đợi*, để quyết định đỉnh nào sẽ được khám phá tiếp theo.

```
def bfs(g,start):
  start.setDistance(0)
  start.setPred(None)
  vertQueue = Queue()
  vertQueue.enqueue(start)
  while (vertQueue.size() > 0):
    currentVert = vertQueue.dequeue()
    for nbr in currentVert.getConnections():
      if (nbr.getColor() == 'white'):
```

```
            nbr.setColor('gray')
            nbr.setDistance(currentVert.getDistance() + 1)
            nbr.setPred(currentVert)
            vertQueue.enqueue(nbr)
        currentVert.setColor('black')
```

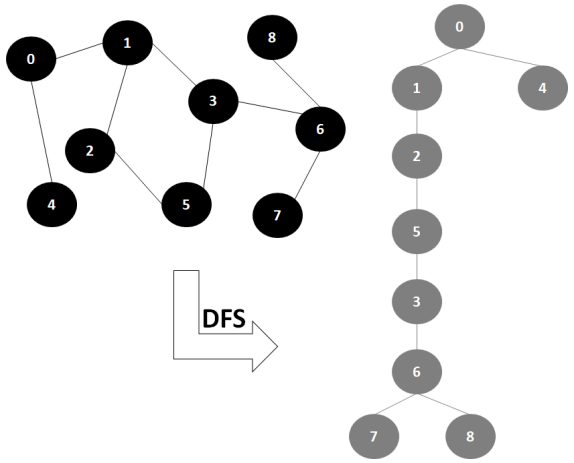
Hãy hoàn chỉnh chương trình bằng cách mở rộng lớp Vertex thêm ba biến (varname) mới: khoảng cách (distance), đỉnh trước (pred) và màu (color), sau đó cung cấp thêm các phương thức lấy giá trị (get<varname>), và gán giá trị (set<varname>) tương ứng cho từng biến mới này.

Khi đã thi hành BFS, có thể bắt đầu ở bất kỳ đỉnh nào trong cây tìm kiếm BFS và lần theo nút trước quay trở lại gốc là đỉnh xuất phát. Hàm sau chỉ ra cách lần theo các liên kết nút trước để in ra thang từ.

```
def traverse(y):
    x = y
    while (x.getPred()):
        print(x.getId())
        x = x.getPred()
    print(x.getId())

traverse(g.getVertex('sage'))
```

3.3. Tìm kiếm theo chiều sâu

Tìm kiếm theo chiều sâu – ***DFS*** (Deep First Search), tạo ra cây theo cách phát triển từng nhánh sâu nhất trước thay vì phát triển từng tầng như ở BFS. Lớp ***DFSGraph*** với các phương thức cài đặt ở dưới tìm kiếm, càng sâu càng tốt, kết nối càng nhiều nút trong đồ thị càng tốt và phân nhánh khi cần thiết.



DFS có thể sẽ tạo ra nhiều hơn một cây, gọi là *rừng*. Như BFS, đầu tiên, DFS sử dụng các cung trước để xây dựng cây. Ngoài ra, DFS sẽ sử dụng hai biến bổ sung trong lớp Vertex. Các biến này giữ số lần khám phá (time) và kết thúc (finish). Số lần khám phá là số bước trong thuật toán tính từ đỉnh đầu tiên. Số lần kết thúc là số bước trong thuật toán trước khi một đỉnh màu đen.

```
class DFSGraph(Graph):
    def __init__(self):
        super().__init__()
        self.time = 0

    def dfs(self):
        for aVertex in self:
            aVertex.setColor('white')
            aVertex.setPred(-1)
        for aVertex in self:
            if aVertex.getColor() == 'white':
                self.dfsvisit(aVertex)

    def dfsvisit(self,startVertex):
        startVertex.setColor('gray')
        self.time += 1
        startVertex.setDiscovery(self.time)
        for nextVertex in startVertex.getConnections():
            if nextVertex.getColor() == 'white':
                nextVertex.setPred(startVertex)
                self.dfsvisit(nextVertex)
        startVertex.setColor('black')
        self.time += 1
        startVertex.setFinish(self.time)
```

Hãy mở rộng của lớp Vertex thêm hai biến mới và cài đặt các phương thức tương ứng. Bạn cũng thay đổi hàm traverse để có thể in kết quả.

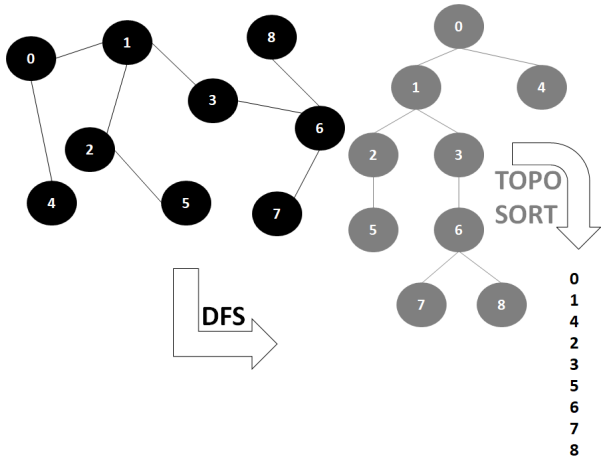
4. Một số thuật giải đồ thị phổ biến và ứng dụng

4.1. Sắp xếp tôpô

Xét ví dụ làm bánh xèo. Công thức khá đơn giản. Nguyên liệu gồm 1 quả trứng, 1 cốc bột, 1 thìa dầu và 4 cốc sữa. Để làm bánh xèo, phải làm nóng chảo, trộn các nguyên liệu với nhau rồi múc hỗn hợp lên chảo nóng. Khi bánh bắt đầu nổi bọt, lật mặt bánh và để cho đến khi bánh có màu vàng nâu.

Cái khó của làm bánh xèo là phải biết làm gì trước làm gì sau. Để giúp quyết định thứ tự chính xác nên thực hiện từng bước làm bánh, đó là ý tưởng thuật toán đồ thị ***sắp xếp tôpô*** .

Sắp xếp tôpô nhận đầu vào là một đồ thị G có hướng, không chu trình và tạo ra một thứ tự tuyến tính trên các đỉnh của G sao cho nếu G chứa cạnh (v, w) thì đỉnh v theo thứ tự sẽ xuất hiện trước đỉnh w.



Đồ thị có hướng không chu trình được sử dụng trong nhiều ứng dụng cần độ ưu tiên của các sự kiện. Làm bánh xèo; lập lịch dự án phần mềm; đồ thị ưu tiên trong truy vấn cơ sở dữ liệu và nhân ma trận;… là các ví dụ.

Sắp xếp tôpô là một biến thể đơn giản của DFS. Thuật toán sắp xếp tôpô như sau:

. Gọi thi hành dfs(g) cho đồ thị g. Lý do chính ta gọi DFS là để tính số cạnh (bước) từ gốc đến mỗi đỉnh.

. Lưu danh sách các đỉnh theo thứ tự giảm dần của số cạnh từ gốc đến.

. Trả về danh sách có thứ tự là kết quả của sắp xếp tôpô.

4.2. Thành phần liên thông

Trong thực tế ứng dụng, thường phải xử lý các đồ thị cực lớn (*big-graph*). Chẳng hạn đồ thị kết nối giữa các máy chủ trên Internet hay liên kết giữa các trang web.

Các công cụ tìm kiếm như Google và Bing khai thác các trang web tạo ra một đồ thị có hướng rất lớn. Để biến World Wide Web thành một đồ thị, ta coi một trang là một đỉnh và các liên kết (link) trên trang là các cạnh nối đỉnh này với đỉnh khác.

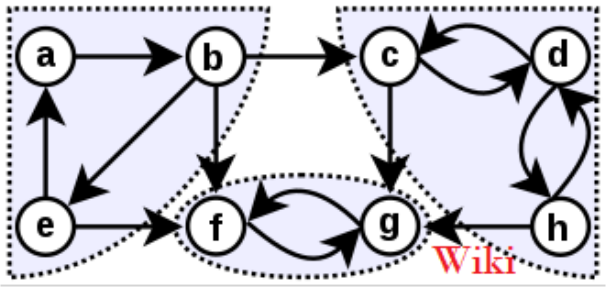
Thuật toán đồ thị có thể giúp tìm các cụm đỉnh có liên kết "cao" trong một đồ thị được gọi là thuật toán các thành phần liên thông "mạnh" – SCC (Strong Connected Components).

Khái niệm "liên kết cao" hay "liên thông mạnh" là các khái niệm có tính định tính và nó phụ thuộc vào tính chất của đồ thị cũng như của bài toán ứng dụng. Chẳng hạn với đồ thị có hướng, ta có thể sử dụng đồ thị chuyển vị như thuật toán giới thiệu ngay dưới; hay có thể dùng bậc của đỉnh, là số đỉnh kề của đỉnh đó như ta dùng trong ứng dụng cũng sẽ được giới thiệu ở dưới. v.v.

Khi các thành phần liên thông mạnh đã được xác định, ta có thể "gom" tất cả các đỉnh trong một thành phần liên thông mạnh thành một đỉnh đại diện.

4.2.1. Một thuật toán dựa trên DFS cho đồ thị có hướng – **DFS-based SCC**

Một thành phần liên thông mạnh, C, của đồ thị có hướng G = (V, E), là tập con lớn nhất của các đỉnh C ⊂ V sao cho với mọi cặp đỉnh v, w ∈ C, có một đường đi từ v đến w và một đường đi từ w tới v: path = $u_0u_1…u_{n-1}u_n$ sao cho u_0 = w, u_n = v (u_0, u_n) ∈ E hay (u_i, u_{i+1}) ∈ E, ∀i = 0, 1,…, n-1.



Có thể xây dựng thuật toán SCC mạnh và hiệu quả dựa trên DFS và khái niệm đồ thị chuyển vị.

Chuyển vị của một đồ thị G, ký hiệu G^T, là đồ thị trong đó tất cả các cạnh trong đồ thị G đã được đảo ngược chiều. Nghĩa là, nếu có một cạnh có hướng từ nút v đến nút w trong đồ thị G thì G^T sẽ có cạnh từ nút w đến nút v.

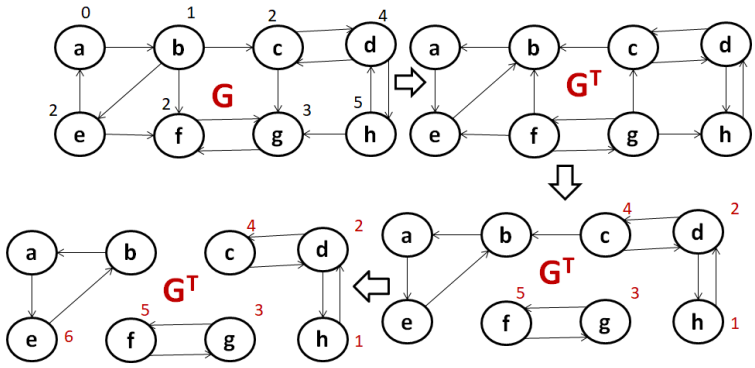
Và thuật toán các thành phần liên thông mạnh cho một đồ thị được mô tả như sau.

Bước 1: Gọi hàm ***dfs*** trên đồ thị G để tính số cạnh (số bước) đến từ gốc được chọn đến mỗi đỉnh.

Bước 2: Tính G^T.

Bước 3: Gọi hàm ***dfs*** cho đồ thị G^T nhưng trong vòng lặp chính của ***DFS***, khám phá từng đỉnh theo thứ tự số cạnh (bước) từ gốc đến theo thứ tự giảm dần.

Bước 4: Mỗi cây trong rừng được tính ở bước 3 là một thành phần liên thông. Xuất id cho mỗi đỉnh trong mỗi cây của rừng để xác định thành phần liên thông tương ứng.

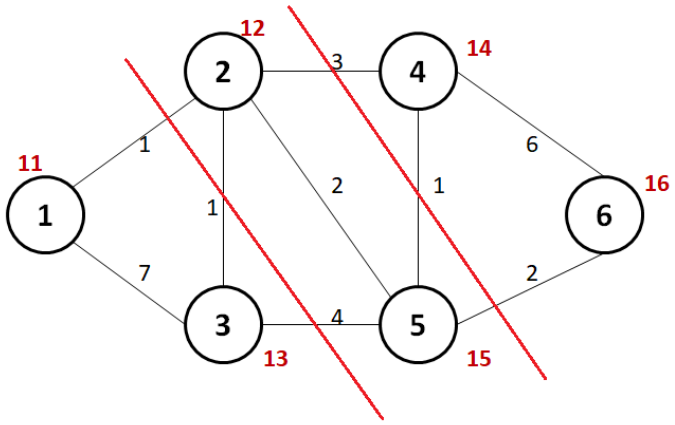


4.2.2. Ứng dụng cho dự báo - **prediction**

Trong thuật toán này, ta xét đồ thị mà các cạnh có trọng số và các đỉnh có tải trọng G = (V, E), trong đó V = {(i, a_i), i = 1, 2,..., |E|}, và E = {(i, j, w_{ij}): i, j ∈ V, w_{ij} ∈ R^+}. Đồ thị này có nhiều ứng dụng trong thực tế như tô màu vùng dịch, phân hoạch các sân bay theo lưu lượng, ...

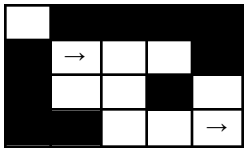
Xét tập đỉnh con khác rỗng S ⊂ V và một phân hoạch $P = (P_i),\ i = 1, ..., p$ của đồ thị G. Đặt $T = \{P_i : S \cap P_i \neq \emptyset,\ i = 1, ..., p\}$ và $w_T = \{a_k,\ k \in T\ V\}$. Bài toán dự báo là tìm tập con T^* của T gồm những đỉnh đạt gần đến tải trọng cực đại của T: $T^* = \{k \in T : a_k \geq \theta w_T,\ 0 < \theta \leq 1\}$. Lưu ý là có thể có một số đỉnh thuộc T^* nhưng không nằm trong S.

Ví dụ. Với tiêu chí phân hoạch dựa trên bậc của đỉnh, giả sử ta có phân hoạch $P = \{P_1, P_2, P_3\}$:
$P_1 = \{(1,11), (3,13)\}$,
$P_2 = \{(2,12), (5,15)\}$,
$P_3 = \{(4,14), (6,16)\}$
và $S = \{(1,11), (5,15)\}$.
$T = P_1 \cup P_2 = \{(1,11), (2,12), (3,13), (5,15)\}$ và $w_T = 15$.
$T^* = \{k \in T: a_k \geq 0.9w_T = 13.5\} = \{5\}$.

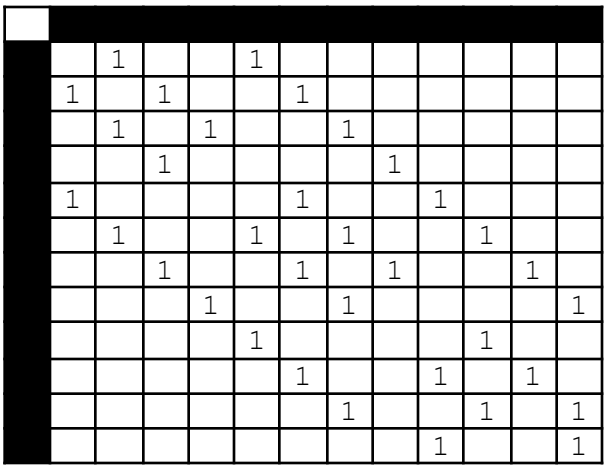


4.3. Tìm đường đi ngắn nhất

Các thuật giải tìm kiếm ta đã xem xét ở trên, BFS hay DFS, là những thuật toán tìm kiếm tổng quát và có thể áp dụng giải bài toán tìm kiếm hướng đích nếu không gian trạng thái có thể được biểu diễn bằng đồ thị, và trạng thái đầu cùng trạng thái kết thúc được cho. Ví dụ bài toán mê cung ta đã xem xét khi phân tích và thiết kế thuật giải có thể được giải bằng sử dụng thuật giải BFS hay DFS, ở đó, mê cung N = mxn ô được biểu diễn bằng đồ thị N đỉnh, mỗi đỉnh u = (i, j) đại diện ô (i, j) của mê cung, và có cạnh nối u = (i, j) đến v = (i', j') nếu v là ô lân cận của ô u và v không có vật cản. Chẳng hạn với mê cung 3x4 như dưới. Theo quy trình giải một bài toán dùng đồ thị, (**Bước 1**) ta có đồ thị 12 đỉnh, dạng ma-trận kề 12x12 tương ứng. Và (**Bước 2**) ta có thể áp dụng thuật giải BFS hay DFS tìm đường đi từ đỉnh 0 đến đỉnh 11.



mê cung 3x4



Như bài tập lập trình, bạn thử viết chương trình (**1**) chuyển từ mê cung sang đồ thị theo ý tưởng nêu trên; và (**2**) áp dụng BFS lẫn DFS cho đồ thị mê cung vừa tạo.

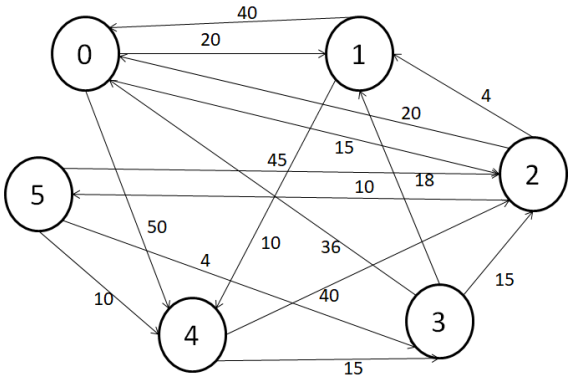
Rõ ràng là để tìm đường đi ngắn nhất, là đường di chuyển của rô-bốt qua ít ô nhất, để đến đích, nếu sử dụng BFS (hay DFS cũng vậy), phải xét tất cả đường đi từ nguồn tới đích và chọn ra lộ trình ngắn nhất. Cách là này là không hiệu quả. Thuật toán **Dijkstra** sau sẽ giải bài toán tìm đường đi ngắn nhất hiệu quả hơn. Sau khi học xong thuật giải **Dijkstra**, bạn cũng hãy thử so sánh độ phức tạp của 2 cách tiếp cận này.

4.3.1. Thuật toán **- Dijkstra**

Dijkstra là một thuật toán lặp tìm đường đi ngắn nhất từ một nút bắt đầu đến tất cả các nút khác trong đồ thị, tương tự với kết quả của tìm kiếm theo chiều rộng - BFS.

Sử dụng biến ***dist*** trong lớp ***Vertex*** để giữ tổng chi phí từ nút bắt đầu đến mỗi nút. Biến ***dist*** giữ tổng trọng lượng hiện tại của đường đi có trọng số nhỏ nhất từ nút bắt đầu đến nút đang xét. Thuật toán duyệt qua mọi đỉnh trong đồ thị; thứ tự thăm các đỉnh được điều khiển bởi hàng đợi. Khoảng cách đến được đỉnh đang xét được xác định theo ***dist***. Khi một đỉnh được tạo lần đầu, ***dist*** được gán giá trị vô cực, trong thực tế cài đặt, chỉ cần định nghĩa vô cực là một số lớn.

Ta minh hoạ thuật giải Dijiktra qua đồ thị ví dụ sau.



node		
	∞.--.20.--.--.--.**2** **0**	2.-.-.-. **2**
	∞.--.04.--.--.--.**0** **4**	2.-.-.-. **2**
→	∞.00.--.--.--.--.**0** **0**	2.-.-.-. **2**
	∞.--.--.--.--.14.**1** **4**	-.-.-.5. **5**
	∞.--.--.70.14.--.**1** **4**	-.0.1.-. **1**
	∞.--.10.--.--.--.**1** **0**	2.-.-.-. **2**

Queue: 2-0-1-5-4-4-3

Đoạn mã sau cài đạt thuật toán Dijkstra. Khi thuật toán kết thúc, các khoảng cách được xác định chính là tổng trọng số các cung đến mỗi đỉnh trong đồ thị.

```
def dijkstra(aGraph,start):
    pq = PriorityQueue()
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in aGraph])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
            newDist = currentVert.getDistance() \
                    + currentVert.getWeight(nextVert)
            if newDist < nextVert.getDistance():
                nextVert.setDistance( newDist )
                nextVert.setPred(currentVert)
                pq.decreaseKey(nextVert,newDist)
```

Cần lưu ý là thuật toán Dijkstra chỉ hoạt động khi các trọng số dương.

4.3.2. Phân tích thuật toán Dijkstra – **big-Oh**

Trước tiên, lưu ý rằng việc xây dựng hàng đợi cần O(|V|) đơn vị thời gian vì ban đầu phải đưa mọi đỉnh của đồ thị vào hàng đợi. Khi hàng đợi được xây dựng, vòng lặp *while* được thực hiện một lần cho mọi đỉnh vì các đỉnh đều được thêm vào lúc đầu. Trong vòng lặp, mỗi lần gọi hàm ***delMin***, mất O(log(|V|)). Kết hợp trong vòng lặp các lệnh gọi ***delMin*** cần O(|V|log(|V|)). Vòng lặp *for* được thực hiện một lần cho mỗi cạnh trong đồ thị, và trong vòng lặp *for*, lệnh gọi ***decreaseKey*** mất O(|E|log(|V|)). Vì vậy, tổng chi phí là O((|V| + |E|)log(|V|))

4.4. Tìm cây khung

Xét bài toán các nhà thiết kế trò chơi trực tuyến và nhà cung cấp dịch vụ Internet phải đối mặt. Họ muốn chuyển một phần thông tin đến bất kỳ ai và tất cả những người đang tham gia một cách hiệu quả. Điều quan trọng là người chơi biết vị trí mới nhất của tất cả những người chơi khác. Điều này rất quan trọng đối với các đài phát tin trên Internet để tất cả đối tượng đang theo dõi đều nhận được tất cả dữ liệu mình cần khi tìm kiếm.

Có một số giải pháp vét cạn cho bài toán này. Ta xem xét qua để hiểu ý nghĩa việc thiết kế thuật toán tốt có ý nghĩa ra sao. Để bắt đầu, người dẫn dắt phát một số thông tin mà tất cả người tham gia cần nhận được. Giải pháp đơn giản nhất là người dẫn giữ danh sách tất cả người nghe và gửi tin nhắn riêng cho từng người.

Một giải pháp vét cạn khác là máy chủ phát một bản sao của thông điệp và để các bộ định tuyến sắp xếp theo mọi thứ. Trong trường hợp này,

chiến lược đơn giản nhất là *gây ngập lụt* tự do. Chiến lược như sau. Mỗi thông báo bắt đầu với giá trị thời gian tồn tại (ttl) được gán một giá trị lớn hơn hoặc bằng số cạnh giữa máy chủ phát sóng và người ở xa nhất tính từ máy chủ. Mỗi bộ định tuyến nhận một bản sao của thông báo và chuyển thông báo đến tất cả các bộ định tuyến lân cận nó. Khi thông báo được truyền thì ttl bị giảm 1. Mỗi bộ định tuyến tiếp tục gửi các bản sao của thông báo đến tất cả các láng giềng của nó cho đến khi giá trị ttl về 0.

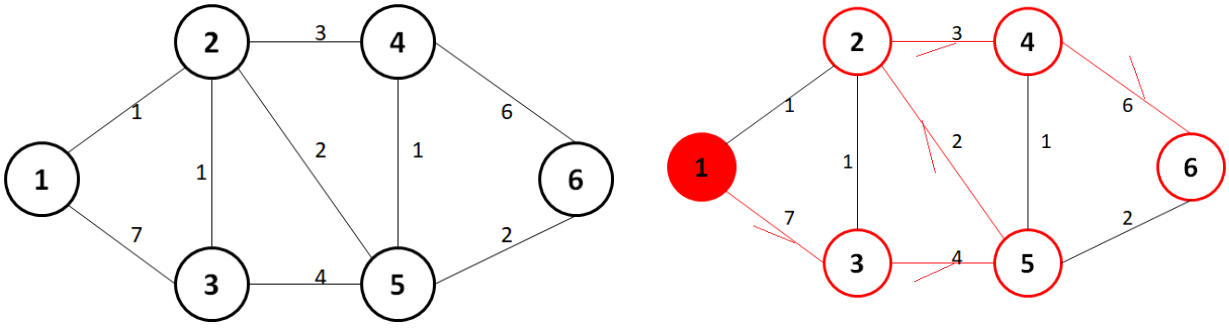
Giải pháp cho vấn đề này nằm ở việc xây dựng một cây khung trọng lượng nhỏ nhất gọi là cây khung tối tiểu. Về mặt hình thức, ta xác định cây khung tối tiểu T cho đồ thị G = (V, E) như sau. T là một tập con không chu trình của E qua tất cả các đỉnh trong V sao cho tổng trọng số của các cạnh trong T là nhỏ nhất.

4.4.1. Thuật toán – **Prim**

Thuật toán **Prime** được dùng để giải quyết vấn đề này. Thuật toán Prim thuộc nhóm "*thuật toán tham lam*" vì ở mỗi bước, ta sẽ chọn bước tiếp theo "rẻ" nhất. Trong trường hợp này, bước tiếp theo rẻ nhất là chọn cạnh có trọng lượng nhỏ nhất. Ý tưởng cơ bản trong việc xây dựng cây khung như sau:

```
WHILE T chưa phải là cây khung

    Tìm cạnh "an toàn" làm cạnh mới

    Thêm cạnh mới vào cây T
```

Bí quyết nằm ở việc "tìm ra cạnh an toàn". Cạnh an toàn là bất kỳ cạnh nào nối một đỉnh nằm trong cây khung với một đỉnh không nằm trong cây khung. Điều này đảm bảo rằng cây sẽ luôn là cây và do đó không có chu trình.



4.4.2. Chương trình - **code**

Thuật toán Prim tương tự như thuật toán Dijkstra ở chỗ cả hai đều sử dụng hàng đợi để cài đặt.

```
def prim(G,start):
    pq = PriorityQueue()
    for v in G:
        v.setDistance(sys.maxsize)
        v.setPred(None)
    start.setDistance(0)
    pq.buildHeap([(v.getDistance(),v) for v in G])
    while not pq.isEmpty():
        currentVert = pq.delMin()
        for nextVert in currentVert.getConnections():
          newCost = currentVert.getWeight(nextVert)
          if nextVert in pq and newCost<nextVert.getDistance():
              nextVert.setPred(currentVert)
              nextVert.setDistance(newCost)
              pq.decreaseKey(nextVert,newCost)
```

Tóm tắt

Ta đã xem xét kiểu dữ liệu trừu tượng đồ thị và một số cách cài đặt của đồ thị. Đồ thị cho phép giải quyết nhiều vấn đề với điều kiện ta có thể biến đổi vấn đề ban đầu về bài toán đồ thị tương đương. Đặc biệt, ta đã thấy rằng đồ thị rất hữu ích để giải quyết các vấn đề trong các lĩnh vực sau đây.

. Tìm kiếm theo chiều rộng để tìm đường đi ngắn nhất không có trọng số.

. Thuật toán Dijkstra cho đường đi ngắn nhất có trọng số.

. Tìm kiếm theo chiều sâu để khám phá đồ thị.

. Các thành phần liên thông mạnh để đơn giản hóa một đồ thị.

. Sắp xếp tôpô để sắp xếp các nhiệm vụ.

. Trọng lượng tối thiểu cây bao trùm để phát thông báo.