ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐHQG-HCM
KHOA CÔNG NGHỆ THÔNG TIN

Phương pháp toán cho Trí tuệ nhân tạo

Lab 02:

Giảng viên lý thuyết
PGS. TS. Nguyễn Đình Thúc

Sinh viên thực hiện
Nguyễn Thị Thu Hằng - 22C15027
Lê Nhựt Nam - 22C11067

Tháng 02 năm 2023

0.1 Tổng quan

Thiết kế kiểu dữ liệu trừu tượng Graph với hai lớp, tương đương với việc sử dụng danh sách kề.

- Graph - chứa danh sách các đỉnh

```python
class Graph:
    def __init__(self):
        """Phương thức khởi tạo đồ thị rỗng, mới."""
        # Danh sách các đỉnh
        self.vertList = {}
        # Số lượng đỉnh trong đồ thị
        self.numVertices = 0

```

  – Phương thức thêm đỉnh addVertex.

```python
def addVertex(self, key: Union[int, str]) -> Vertex:
    """Phương thức thêm đỉnh có `key` vào đồ thị

    Tham số:
        key (Union[int, str]): key của đỉnh cần thêm vào đồ thị.

    Trả về:
        Vertex: đối tượng thuộc lớp Vertex thể hiện đỉnh vừa thêm vào đồ thị.
    """
    self.numVertices = self.numVertices + 1
    newVertex = Vertex(key)
    self.vertList[key] = newVertex
    return newVertex

```

  – Phương thức lấy một đỉnh getVertex.

```python
def getVertex(self, n: Union[int, str]) -> Union[Vertex, None]:
    """Phương thức lấy một đỉnh có khóa hay nhãn là `n` ra từ danh sách các
    ↪  đỉnh của đồ thị.

    Tham số:
        n (Union[int, str]): key của đỉnh cần lấy ra từ đồ thị.

    Trả về:
```

```
        Union[Vertex, None]: Trả về đối tượng thuộc lớp Vertex thể hiện đỉnh cần lấy ra từ đồ thị.
        Nếu đỉnh không tồn tại, trả về None.
    """
    if n in self.vertList:
        return self.vertList[n]
    else:
        return None

```

– Phương thức thêm một cạnh addEdge.

```
def addEdge(self, f: Union[int, str], t: Union[int, str], weight: int = 0):
        """Phương thức thêm cạnh, có hướng vào đồ thị nối hai đỉnh `f` và `t`.
        Tham số:
            f (Union[int, str]): Khóa của đỉnh bắt đầu.
            t (Union[int, str]): Khóa của đỉnh kết thúc.
            weight (int, optional): Trọng số của cạnh nối hai đỉnh. Mặc định là 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)
```

– Phương thức lấy tất cả các đỉnh getVertices.

```
    def getVertices(self) -> list:
        """Phương thức trả về danh sách tất cả các đỉnh trong đồ thị.

        Tham số:
            list: Danh sách tất ca các đỉnh trong đồ thi.
        """
        return self.vertList.keys()
```

– Phương thức iter duyệt qua các đối tượng đỉnh trong đồ thị.

```
def __iter__(self):
        """Phương thức duyệt các đối tượng đỉnh trong đồ thị.
```

```
3
4           Trả về:
5               SupportsNextT@iter: Bộ duyệt duyệt các đối tượng đỉnh trong đồ
    ↪  thị.
6           """
7           return iter(self.vertList.values())
```

- Một số phương thức hỗ trợ xây dựng đồ thị
  - Từ dữ liệu nhập tay

```
1  @classmethod
2      def build_graph_from_file(cls, filename, delimiter = " "):
3          """ Phương thức xây dựng đồ thị từ danh sách kề
4
5          Tham số:
6              filename: file chứa danh sách kề
7
8          Trả về:
9              Đồ thị
10         """
11         g = cls()
12         edge_list = dict()
13         for line in open(filename,'U'):
14             L = line.strip().split(delimiter)
15             if (L[0] not in g.getVertices()):
16                     g.addVertex(L[0])
17             for i in range(1, len(L)):
18                 if (L[i] not in g.getVertices()):
19                     g.addVertex(L[i])
20                 g.addEdge(L[0], L[i])
21         return g
```

  - Từ dữ liệu tập tin

```
1      @classmethod
2      def build_graph_from_edge_list(cls, d):
3          """ Phương thức xây dựng đồ thị từ danh sách kề
4
5          Tham số:
6              d: danh sách kề
```

```
7
8           Trả về:
9               Đồ thị
10          """
11          g = cls()
12          for v1, v2_list in d.items():
13              if (v1 not in g.getVertices()):
14                      g.addVertex(v1)
15              for v2 in v2_list:
16                  if (v2 not in g.getVertices()):
17                      g.addVertex(v2)
18                  g.addEdge(v1, v2)
19
20          return g
```

- Vertex - mỗi đỉnh trong đồ thị

```
1  class Vertex:
2      def __init__(self, key: Union[int, str]):
3          """Phương thức khởi tạo.
4          Chỉ thiết lập id, gồm một string là key được truyền cho và một từ điển
   ↪  connectTo
5          Tham số:
6              key (Union[int, str]): Khóa hay nhãn của đỉnh được truyền cho.
7          """
8          # Khóa hay nhãn
9          self.id = key
10         # Từ điển connectTo
11         self.connectedTo = {}
12
```

– Phương thức addNeighbor để thêm một cung vào đồ thị.

```
1  def addNeighbor(self, nbr: Union[int, str], weight: int = 0):
2          """Phương thức thêm một cung vào đồ thị.
3
4          Tham số:
5              nbr (Union[int, str]): Khóa hay nhãn của lân cận đầu vào.
6              weight (int, optional): Trọng số của cung sau khi khởi tạo. Mặc
   ↪  định là 0.
7          """
```

```
8         self.connectedTo[nbr] = weight
```

– Phương thức getConnections để lấy tất cả các đỉnh trong danh sách kề.

```
1 def getConnections(self) -> dict:
2         """Phương thức trả về tất cả các đỉnh trong danh sách kề, biểu diễn
          ↪  bởi biến connectedTo.
3
4         Trả về:
5             dict: Từ điển connectedTo chứa dữ liệu tất cả các đỉnh trong danh
      ↪  sách kề.
6         """
7         return self.connectedTo.keys()
```

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

```
1 ef getWeight(self, nbr) -> int:
2         """Phương thức lấy trọng số của cạnh được truyền theo tham số.
3
4         Tham số:
5             nbr (_type_): khóa hay nhãn thể hiện của đỉnh lân cận với đối
      ↪  tượng đỉnh.
6
7         Trả về:
8             int: Trọng số của cạnh được truyền theo tham số.
9         """
10        return self.connectedTo[nbr]
```

0.2 Thiết kế lớp trừu tượng Whatever first search

Whatever first search là một thuật toán tổng quát để thể hiện ba loại thuật toán chính trong bài toán duyệt đồ thị. Thuật toán duyệt đồ thị tổng quát lưu trữ một tập hợp lực lượng các cạnh trong một cấu trúc dữ liệu tổng quát gọi là "bag". Một tính chất quan trọng của một "bag" là ta có thể đưa một phần tử vào trong đó và sau đó lấy phần tử bên trong ra.

Bằng cách thay đổi cấu trúc dữ liệu bag, ta có thể cài đặt được các thuật toán sau:

- Nếu bag là cấu trúc dữ liệu ngăn xếp (stack), ta cài đặt được thuật toán Depth First Search.
- Nếu bag là cấu trúc dữ liệu hàng đợi (queue), ta cài đặt được thuật toán Breadth First Search.
- Nếu bag là cấu trúc dữ liệu hàm đợi ưu tiên (priority queue), ta cài đặt được thuật toán Djisktra.

Thuật toán 1 Thuật toán Whatever first search

Khởi tạo các biến đánh dấu
Đặt phần tử s vào một empty bag.
while bag không rỗng **do**
    Lấy một đỉnh v từ trong bag.
    if v chưa được đánh dấu **then**
        Đánh dấu v
        for each edge vw **do**
            Đặt w vào bag
        end for
    end if
end while

Cài đặt

```python
class WhateverFirstSearch(ABC):
    def __init__(self, start, G):
        self.start = start
        self.G = G
    @abstractclassmethod
    def search(self):
        raise NotImplementedError
```

Nếu gọi $m = |V|$, $n = |E|$, và t lần lượt là số lượng đỉnh, số lượng cạnh, và thời gian thêm/ xóa phần tử trong bag.

- Việc khởi tạo các biến đánh dấu tốn chi phí $O(m)$ do có bấy nhiêu đỉnh thì tạo mảng rộng bấy nhiêu.
- Vòng for ở trong cùng thực thi chính xác một lần cho mỗi những đỉnh đã được đánh dấu, do đó nó sẽ chạy nhiều nhất m lần, $O(m)$
- Mỗi cạnh uv được đặt vào bag chính xác hai lần: một lần với cạnh u, v, một lần với v, u. Do đó, lệnh put trong vòng for chạy nhiều nhất $2n$ lần.
- Việc lấy một phần tử ra khỏi bag tốn nhiều nhất $2n + 1$ lần ($+1$ là pha lấy cuối cùng để biết nó rỗng)

Tùy vào cấu trúc lưu trữ của đồ thị mà thuật toán có độ phức tạp khác nhau:

- Cấu trúc danh sách kề: $O(m + tn)$
- Cấu trúc ma trận kề: $O(m^2 + tn)$

0.3 Thuật toán Breadth First Search

Khi cấu trúc bag được cài đặt bằng cấu trúc hàng đợi (queue), ta có một cách duyệt gọi là thuật toán Breadth First Search. Cấu trúc hàng đợi được cài đặt với các phương thức push và pop trong độ phức tạp $O(1)$, do đó độ phức tạp thuật toán là $O(m+n)$

```python
class BFS(WhateverFirstSearch):
    def __init__(self, start, G):
        super().__init__(start, G)

    def search(self):
        queue = []      # Initialize a queue
        visited = []    # List for visited nodes.
        queue.append(self.start)
        visited.append(self.start)
        current_node_id = self.start
        while len(queue) > 0:
            current_node_id = queue.pop(0)
            print(current_node_id)
            for neighbor in self.G.getVertex(str(current_node_id)).getConnections():
                id_node_neighbor = neighbor.id
                if id_node_neighbor not in visited:
                    queue.append(id_node_neighbor)
                    visited.append(id_node_neighbor)
```

Cách chạy thực nghiệm

1/ Với dữ liệu nhập tay

```python
# Khởi tạo dữ liệu
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

# Khởi tạo đồ thị
g = Graph()
g = g.build_graph_from_edge_list(graph)
```

```
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
#print(g.getVertex(str(5)).getConnections())

print("Following is the Breadth-First Search")
dfs = BFS(5, g).search()

# Kết quả: 5 3 7 2 4 8
```

2/ Với dữ liệu nhập từ tập tin

```
# File: graph.txt
# 5 3 7
# 3 2 4
# 7 8
# 2
# 4 8
# 8
print("Test load graph from file")
g = Graph()
g = g.build_graph_from_file("graph.txt")
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
print("Following is the Breadth-First Search")
dfs = BFS(5, g).search()

# Kết quả: 5 3 7 2 4 8
```

0.4 Thuật toán Depth First Search

Khi cấu trúc bag được cài đặt bằng cấu trúc ngăn xếp (stack), ta có một cách duyệt gọi là thuật toán Depth First Search. Cấu trúc hàng đợi được cài đặt với các phương thức push và pop trong độ phức tạp $O(1)$, do đó độ phức tạp thuật toán là $O(m+n)$

```
class DFS(WhateverFirstSearch):
    def __init__(self, start, G):
        super().__init__(start, G)

```

```
    def search(self):
        stack = []
        visited = []
        stack.append(self.start)
        current_node_id = self.start
        while len(stack) > 0:
            current_node_id = stack.pop()
            print(current_node_id)
            for neighbor in self.G.getVertex(str(current_node_id)).getConnections():
                id_node_neighbor = neighbor.getId()
                # print(id_node_neighbor)
                if id_node_neighbor not in visited:
                    stack.append(id_node_neighbor)
                    visited.append(id_node_neighbor)
```

Cách chạy thực nghiệm

1/ Với dữ liệu nhập tay

```
# Khởi tạo dữ liệu
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

# Khởi tạo đồ thị
g = Graph()
g = g.build_graph_from_edge_list(graph)
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
#print(g.getVertex(str(5)).getConnections())

print("Following is the Depth-First Search")
dfs = DFS(5, g).search()

# Kết quả: 5 7 8 3 4 2
```

2/ Với dữ liệu nhập từ tập tin

```
# File: graph.txt
# 5 3 7
# 3 2 4
# 7 8
# 2
# 4 8
# 8
print("Test load graph from file")
g = Graph()
g = g.build_graph_from_file("graph.txt")
for v in g:
    for w in v.getConnections():
        print("( %s , %s )" % (v.getId(), w.getId()))
print("Following is the Depth-First Search")
dfs = DFS(5, g).search()

# Kết quả: 5 7 8 3 4 2
```