Lab 3: Các thuật toán phân hoạch đồ thị

Phương pháp toán cho Trí tuệ nhân tạo

Trần Xuân Lộc - 22C11064, Lê Nhựt Nam - 22C11067

26/03/2023

Mục lục

1 Thông tin chung

- Thành viên:
  - Trần Xuân Lộc - 22C11064
  - Lê Nhựt Nam - 22C11067
- Bảng phân công công việc:

Công việc	Người thực hiện
Tái cấu trúc đồ thị theo yêu cầu của thầy	Xuân Lộc
Viết hàm và tài liệu cho thuật toán `BFS`, `Kerninghan-Lin`, `Fiduccia-Mattheyses`, và `Spectral Bisection`	Nhựt Nam
Viết hàm và tài liệu cho thuật toán `BFS`	Xuân Lộc

2 Cấu trúc mã nguồn

- Tại đây mô tả về schema của các class và thông tin (tóm tắt, đầu vào, đầu ra) của các hàm.

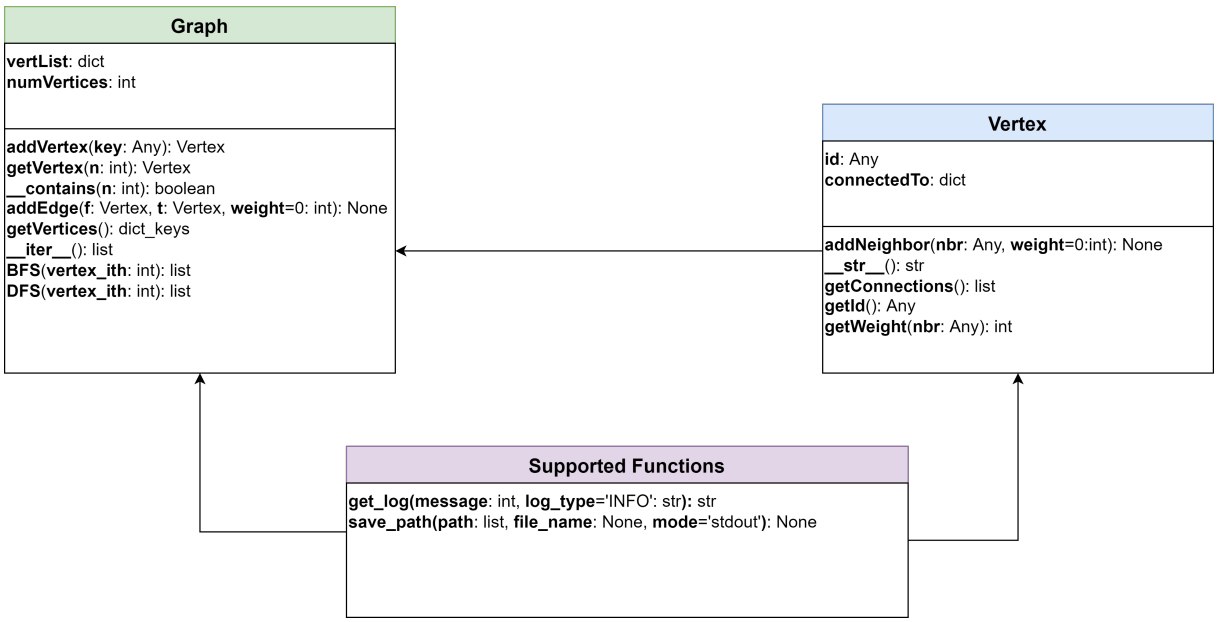

Figure 1: Tổ chức dữ liệu đồ thị.

2.1 Tổ chức dữ liệu đồ thị của lớp `Graph`

Trong phần này, nhóm sẽ trình bày cách tổ chức lưu trữ dữ liệu đồ thị vào trong lớp `Graph` bao gồm các thành phần lưu trữ dữ liệu và mô tả các hàm thực thi phục vụ cho lớp. Tệp `graph.py` lưu trữ thông tin cấu hình chi tiết và mã nguồn.

2.1.1 Cấu trúc dữ liệu đồ thị

- `self.vertList`: Biến có kiểu dữ liệu từ điển, chứa danh sách đỉnh của đồ thị. Mỗi phần tử trong từ điển có khóa là định danh của đỉnh (`id`) và giá trị là một đối tượng có kiểu dữ liệu `Vertex`.
- `self.numVertices`: Biến có kiểu dữ liệu là số nguyên, xác định số đỉnh của đồ thị.

2.1.2 Các hàm thành phần

- Hàm `addVertex(self, key)`:

  - Mô tả: Hàm thêm một đỉnh vào cấu trúc dữ liệu đồ thị.
  - Tham số:
    * `key`: Định danh của một đỉnh.
  - Trả về: Đỉnh vừa được thêm vào dưới dạng một đối tượng `Vertex`.

- Hàm `getVertex(self, n)`:
  - Mô tả: Hàm lấy thông tin của đỉnh có định danh `n` của đồ thị
  - Tham số:
    * `n`: Định danh (`id`) của đỉnh trong đồ thị.
  - Trả về:
    * Đối tượng `Vertex` có định danh `n`, nếu đỉnh `n` có tồn tại trong đồ thị.
    * `None`, nếu trong đồ thị không tồn tại đỉnh có định danh `n`.

- Hàm `__contains__(self, n)`:
  - Mô tả: Hàm kiểm tra đỉnh có định danh `n` có tồn tại trong đồ thị hay không.
  - Tham số:
    * `n`: Định danh (`id`) của một đỉnh.
  - Trả về:
    * `True`, nếu đỉnh `n` tồn tại trong đồ thị.
    * `False`, nếu đỉnh `n` không tồn tại trong đồ thị.

- Hàm `addEdge(self, f, t, weight=0)`:
  - Mô tả: Hàm thêm một cạnh có trọng số `weight` (mặc định bằng 0) đi từ đỉnh `f` đến đỉnh `t`. Nếu một trong hai đỉnh không tồn tại trong đồ thị thì thêm đỉnh đó vào đồ thị.
  - Tham số:
    * `f`: Định danh của đỉnh xuất phát.
    * `t`: Định danh của đỉnh đích.
    * `weight`: Trọng số của đỉnh được thêm vào. Mặc định bằng 0.

- Hàm `getVertices(self)`:
  - Mô tả: Hàm lấy thông tin của toàn bộ đỉnh trong đồ thị.
  - Tham số: Không.
  - Trả về:
    * Trả về đối tượng `dict_key`, chứa danh sách định danh của toàn bộ đỉnh trong đồ thị.

- Hàm `__iter__(self)`:
  - Mô tả: Hàm hỗ trợ việc duyệt qua mọi đỉnh trong đồ thị.

  - Tham số: Không.
  - Trả về:
    * Đối tượng có kiểu dữ liệu `iterator`, hỗ trợ việc duyệt qua mọi đỉnh trong đồ thị.

- Hàm `BFS(self, vertex_ith)`:
  - Mô tả: Hàm duyệt qua tất cả các đỉnh trong đồ thị bằng thuật toán `BFS` với đỉnh bắt đầu là `vertex_ith`.
  - Tham số:
    * `vertex_ith`: Định danh (`id`) của đỉnh bắt đầu.
  - Trả về:
    * Thứ tự đỉnh được duyệt qua bởi thuật toán `BFS`.

- Hàm `DFS(self, vertex_ith)`:
  - Mô tả: Hàm duyệt qua tất cả các đỉnh trong đồ thị bằng thuật toán `DFS` với đỉnh bắt đầu là `vertex_ith`.
  - Tham số:
    * `vertex_ith`: Định danh (`id`) của đỉnh bắt đầu.
  - Trả về:
    * Thứ tự đỉnh được duyệt qua bởi thuật toán `DFS`.

- Hàm `save_path(path: list, file_name=None, mode='stdout')`:
  - Mô tả: Hàm hỗ trợ lưu kết quả dưới dạng tệp hoặc hiển thị kết quả ra màn hình.
  - Tham số:
    * `path`: Danh sách lưu lại các thứ tự duyệt các nút của các thuật toán.
    * `file_name`: Biến lưu giá trị tên tệp lưu kết quả, nếu `mode='write_to_file'` nhưng giá trị biến rỗng sẽ báo lỗi cho người dùng.
    * `mode`: Biến mang cấu hình hiện thị ra màn hình nếu `mode='stdout'` (giá trị mặc định) hoặc ghi kết quả vào tệp `mode='write_to_file'`.
  - Trả về: Không

Trong bài lab này, chúng em thiết kế thêm một số hàm hỗ trợ tính toán ma trận kề, bậc của đỉnh và ma trận Laplacian cho đồ thị

- Hàm `compute_adjacency_matrix(self, )`:
  - Trả về: mảng numpy thể hiện biểu diễn ma trận kề của đồ thị.

- Hàm `degree_nodes(self, adjacency_matrix)`:
  - Trả về: numpy vector thể hiện bậc của các đỉnh trong đồ thị.

- Hàm `compute_laplacian_matrix(self, )`:
  - Trả về: mảng numpy thể hiện biểu diễn ma trận Laplacian của đồ thị.

2.2 Tổ chức dữ liệu đỉnh của lớp `Vertex`

Tiếp theo, nhóm sẽ trình bày cách tổ chức lưu trữ dữ liệu của các đỉnh bao gồm định danh đỉnh và tập hợp các láng giềng kề với đỉnh đó. Ngoài ra, chúng em cũng mô tả thêm các hàm thực thi hỗ trợ cho lớp này và các hàm này được lưu trong tệp `vertex.py`.

2.2.1 Cấu trúc dữ liệu của lớp đỉnh `Vertex`

- `self.id`: Biến lưu trữ định danh của một đỉnh, có kiểu dữ liệu bất kì - `Any`, không có ràng buộc có thể là kiểu số hoặc chuỗi tùy ý.
- `self.connectedTo`: Biến lưu trữ tập hợp các láng giềng có kề với đỉnh hiện tại, kiểu dữ liệu lưu trữ là từ điển - `dict`.

Trong bài lab này chúng em bổ sung một số thuộc tính cho cấu trúc dữ liệu này để thuận tiện hơn trong quá trình cài đặt thuật toán phân hoạch đồ thị - `self.level`: Lưu trữ level của đỉnh sau khi được viếng thăm bởi thuật toán BFS, sau đó được dùng cho thuật toán phân hoạch đồ thị dựa trên ngưỡng - `self.partition_label`: thuộc tính nhãn phân hoạch dùng để xác định đỉnh nằm trong tập phân hoạch nào. Biến này sử dụng cho việc cài đặt thuật toán: Fiduccia-Mattheyses, và Kerninghan-Lin - `self.external_cost` và `self.internal_cost`: thuộc tính chi phí nội tại và chi phí ngoại tại của phân hoạch. Các biến này sử dụng cho việc cài đặt thuật toán: Fiduccia-Mattheyses, và Kerninghan-Lin

2.2.2 Các hàm thành phần

- Hàm `addNeighbor(self, nbr, weight=0)`:
  - Mô tả: Hàm thêm láng giềng `nbr` có liên kết với đỉnh hiện tại với trọng số mặc định `weight=0`.
  - Tham số:
    - `nbr`: láng giềng của định đang xét.
    - `weight`: giá trị thể hiện trọng số liên kết.
  - Trả về: Không
- Hàm `__str__(self)`:
  - Mô tả: Hàm mô tả đỉnh thông qua các thông số lưu trữ.

  - Tham số: Không
  - Trả về: Chuỗi bao gồm định danh và tập hợp các đỉnh kề của đỉnh đó.

- Hàm `getConnections(self)`:
  - Mô tả: hàm trả về
  - Tham số: Không
  - Trả về: mảng các đỉnh có liên kết với đỉnh hiện tại thông qua giá trị của biến `self.connectedTo`.

- Hàm `getId(self)`:
  - Mô tả: Hàm trả về định danh của đỉnh hiện tại.
  - Tham số: Không
  - Trả về: Định danh của đỉnh hiện tại của đỉnh với kiểu dữ liệu bất kì `Any`.

- Hàm `getWeight(self, nbr)`:
  - Mô tả: Hàm trả về trọng số liên kết của đỉnh `nbr` với đỉnh đang xét.
  - Tham số:
    * `nbr`: láng giềng của đỉnh hiện tại.
  - Trả về: Trọng số của cạnh giữa đỉnh hiện tại và `nbr`.

2.3 Các hàm hỗ trợ

Phần này sẽ tập trung mô tả các hàm hỗ trợ ghi dữ liệu và hiển thị dữ liệu cho người dùng thông báo tình trạng thực thi các hàm của các lớp. Các hàm này được lưu trong tệp `support.py`.

- Hàm `save_path(path: list, file_name=None, mode='stdout')`:
  - Mô tả: Hàm hỗ trợ lưu kết quả dưới dạng tệp hoặc hiển thị kết quả ra màn hình.
  - Tham số:
    * `path`: Danh sách lưu lại các thứ tự duyệt các nút của các thuật toán.
    * `file_name`: Biến lưu giá trị tên tệp lưu kết quả, nếu `mode='write_to_file'` nhưng giá trị biến rỗng sẽ báo lỗi cho người dùng.
    * `mode`: Biến mang cấu hình hiện thị ra màn hình nếu `mode='stdout'` (giá trị mặc định) hoặc ghi kết quả vào tệp `mode='write_to_file'`.
  - Trả về: Không

- Hàm `get_log(message, log_type='INFO')`:
  - Mô tả: Hàm hỗ trợ ghi thông tin thực thi của hàm bao gồm loại nhật kí ghi, thời gian thực thi và thông điệp muốn ghi lại.

- Tham số:
  * `path`: Danh sách lưu lại các thứ tự duyệt các nút của các thuật toán.
  * `log_type`: Biến mang cấu hình loại nhật kí được thực thi với giá trị mặc định là `log_type='stdout'`, một số loại nhật kí khác như `WARNING`, `ERROR`, `DEBUG`
- Trả về: Chuỗi lưu trữ nhật kí hoặc thông tin thực thi tại thời điểm gọi hàm.

2.4 Mô tả thuật toán DFS, BFS

2.4.1 Thuật toán DFS

- Ý tưởng thuật toán: Bắt đầu từ đỉnh xuất phát đi xa nhất có thể, đến khi không thể đi được nữa thì quay lui (backtracking). Chính vì vậy, có thể cài đặt thuật toán này bằng đệ quy hoặc sử dụng một ngăn xếp.
- Thuật toán được cài đặt như sau:

```
def DFS(self, vertex_ith: int):
    """depth first search function, start from `vertex_ith`
    Args: vertex_ith (int): key of vertex in graph
    Raises: ValueError: can't find a vertex with given key
    Returns: list[int]: the path that DFS agent has gone through
    """
    vertex: Vertex = self.getVertex(vertex_ith)
    if vertex is None:
        message = 'Invalid vertex id, could not found vertex id `'
            + str(vertex_ith) + '` in Graph'
        raise ValueError(get_log(message, log_type='ERROR'))

    closed_set: list[int] = []
    open_set: list[int] = [vertex.getId()]

    while open_set:
        cur_vertex: Vertex = self.getVertex(open_set.pop())
        cur_vertex_id = cur_vertex.getId()

        if cur_vertex_id not in closed_set:
            closed_set.append(cur_vertex_id)
            neighbors = [x.id for x in cur_vertex.getConnections()
                ]

            for neighbor in neighbors:
                if neighbor not in closed_set:
                    open_set.append(neighbor)
    return closed_set
```

- Minh họa thuật toán:
  - Đồ thị:

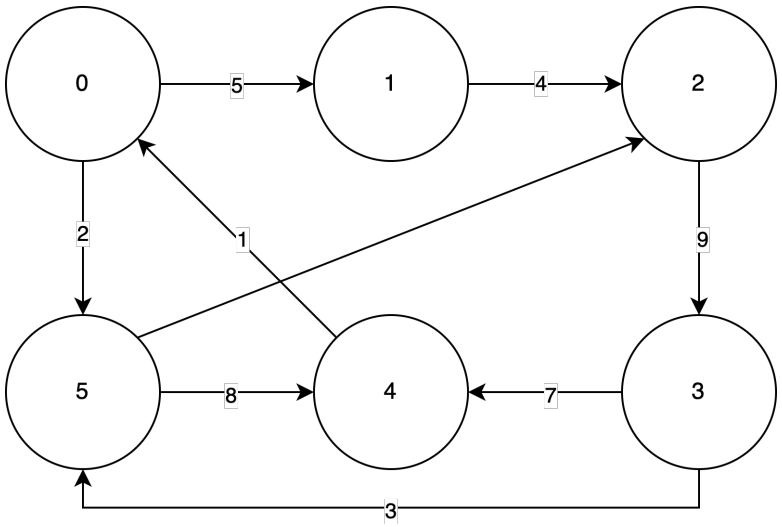

Figure 2: Minh họa đồ thị

- Quá trình duyệt đồ thị:

current node	stack	visited
0	{1,5}	{0}
5	{1,4,2}	{0,5}
2	{1,4,3}	{0,5,2}
3	{1,4}	{0,5,2,3}
4	{1}	{0,5,2,3,4}
1	{}	{0,5,2,3,4,1}

- Kết quả: Thứ tự duyệt của đồ thị là $\{0, 5, 2, 3, 4, 1\}$
- Minh họa bằng cây tìm kiếm:

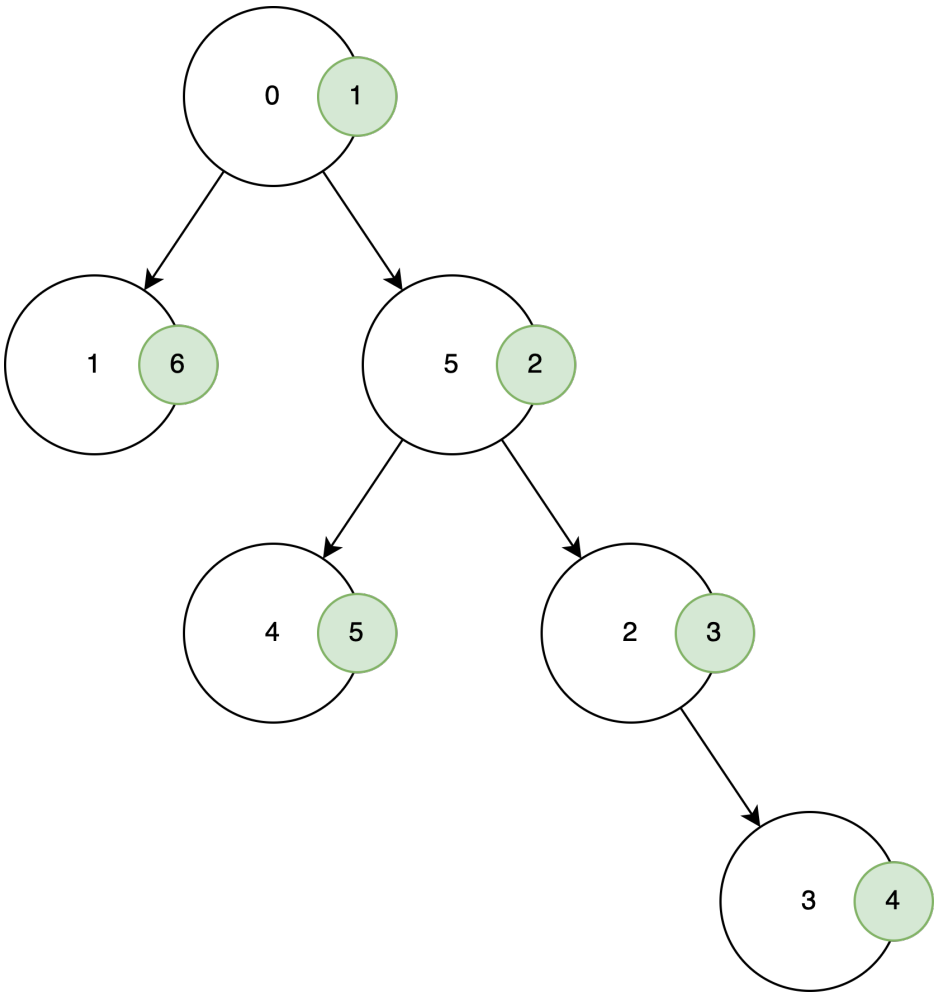

Figure 3: Minh họa thuật toán DFS bằng cây tìm kiếm. Thứ tự duyệt được thể hiện trong hình tròn màu xanh.

2.4.2 Thuật toán BFS

- Ý tưởng thuật toán: Bắt đầu từ đỉnh xuất phát đi rộng nhất có thể, đến khi không thể đi được nữa thì quay lại đi xuống 1 bậc đồ thị để tiếp tục quá trình tương tự. Do đó, ta có thể cài đặt thuật toán này bằng 1 hàng đợi và 1 mảng đánh dấu đã duyệt là đủ.
- Cấu hình thuật toán được thể hiện ở bên dưới.

```
1  def BFS(self, vertex_ith: int):
2      """
3      Module applying Breadth First Search Algorithm.
4
5      :param vertex_ith: the vertex id in Graph
```

```
6       :return: path computed by BFS
7       """
8       # get the vertex `vertex_ith`.
9       vertex = self.getVertex(vertex_ith)
10
11      # checking if not exist `vertex_ith` in Graph then raise error
12      if not vertex:
13          message = 'Invalid vertex id, could not found vertex id `'
                + str(vertex_ith) + '` in Graph'
14          raise ValueError(get_log(message, log_type='ERROR'))
15
16      # get the number of vertices.
17      n = self.numVertices
18
19      # bool array for marking visited or not.
20      visited = [False] * n
21
22      # get the vertex_id for easy management.
23      vertex_id = vertex.getId()
24
25      # initializing a queue to handling which vertex is remaining.
26      queue = [vertex_id]
27
28      # marking the `vertex_id` is visited due to the beginning
           vertex.
29      visited[vertex_id] = True
30
31      # path to track the working state of BFS.
32      path = []
33      while queue:
34          # handling current vertex before removing out of queue.
35          cur_pos = queue[0]
36
37          # appending to path to track.
38          path.append(cur_pos)
39          # remove it out of queue
40          queue.pop(0)
41          # get all neighbors id of current vertex.
42          neighbor_cur_pos = [x.id for x in self.getVertex(cur_pos).
               getConnections()]
43
44          # loop over the neighbor of current vertex.
45          for neighborId in neighbor_cur_pos:
46              # if not visited then push that vertex into queue.
47              if not visited[neighborId]:
48                  visited[neighborId] = True
49                  queue.append(neighborId)
50      return path
```

- Minh họa thuật toán:

- Đồ thị:

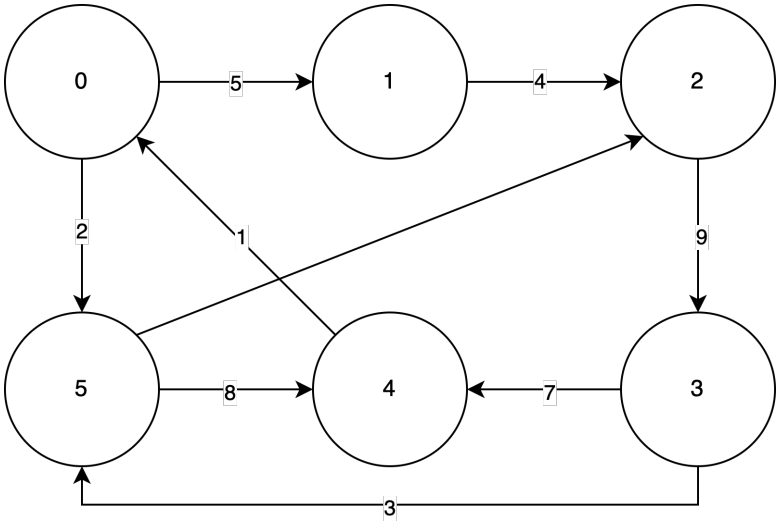

Figure 4: Minh họa đồ thị

- Quá trình duyệt đồ thị:

current node	stack	visited
0	{1,5}	{0}
1	{5,2}	{0,1}
5	{2,4}	{0,1,5}
2	{4,3}	{0,1,5,2}
4	{3}	{0,1,5,2,4}
3	{}	{0,1,5,2,4,3}

- Kết quả: Thứ tự duyệt của đồ thị là $\{0, 1, 5, 2, 4, 3\}$
- Minh họa bằng cây tìm kiếm:

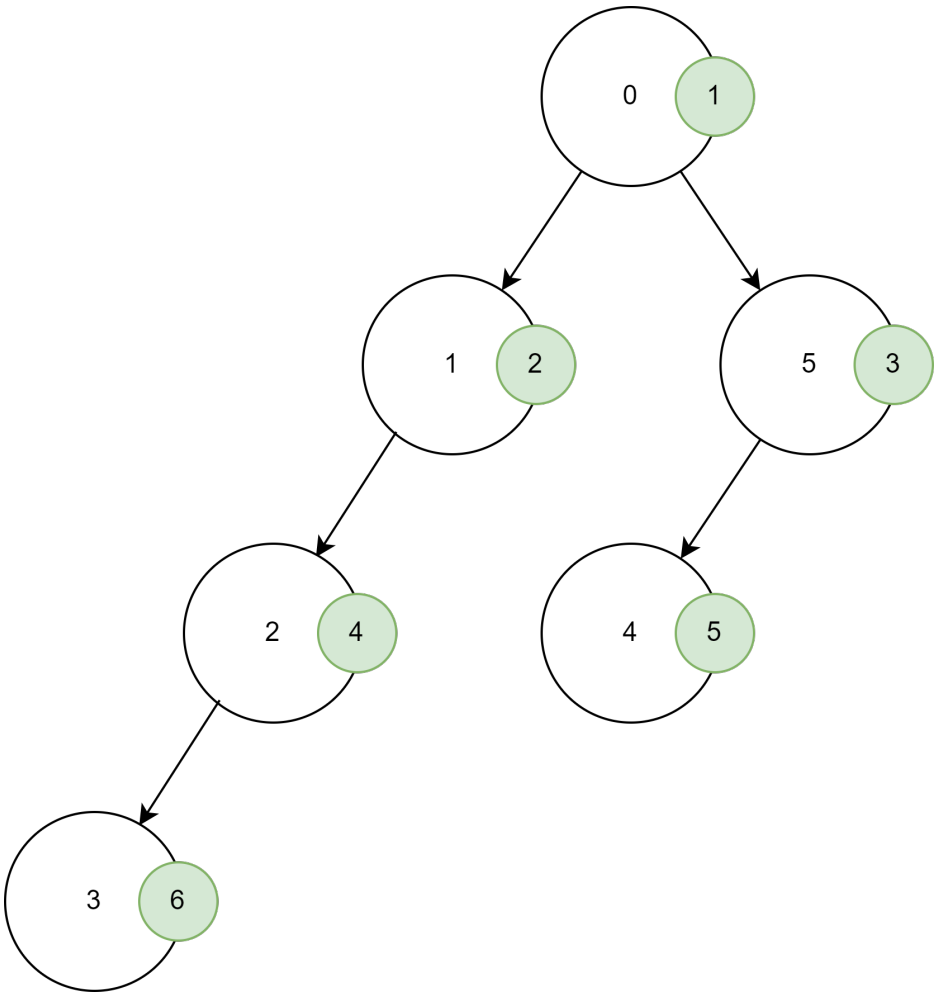

Figure 5: Minh họa thuật toán BFS bằng thuật toán duyệt theo chiều rộng với cây tìm kiếm. Thứ tự duyệt được thể hiện trong hình tròn màu xanh.

2.5 Mô tả thuật toán phân hoạch đồ thị

2.5.1 Phát biểu bài toán

Xem xét đồ thị $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, trong đó $\mathcal{V}$ đại diện cho tập hợp n đỉnh, $\mathcal{E}$ đại diện cho tập hợp các cạnh.

Với một bài toán phân hoạch cân bằng (k, v)m mục tiêu là phân hoạch đồ thị $\mathcal{G}$ thành k thành phần có kích thước lớn nhất là $v \cdot \frac{n}{k}$ trong khi cực tiểu capacity của các cạnh giữa các thành phần phân hoạch (có thể định nghĩa capacity theo nhiều cách).

Trong báo cáo kỹ thuật này, chúng em xem xét bài toán phân hoạch đồ thị thành hai thành phần. Chúng em thực hiện cài đặt các thuật toán như sau:

- Phân hoạch dựa trên BFS
- Kernighan-Lin Algorithm
- Fiduccia-Mattheyses Partitioning Algorithm
- Spectral Bisection
- Tối ưu thuật toán phân hoạch đồ thị bằng thành phần liên thông

2.5.2 Thuật toán phân hoạch dựa trên BFS

Thuật toán tìm kiếm theo chiều rộng có thể sử dụng để giải quyết bài toán phân hoạch đồ thị.

Ý tưởng: Thuật toán BFS duyệt đồ thị theo từng mức (level by level), và bằng cách đánh dấu mỗi đỉnh với level mà nó được viếng thăm. Tập hợp các đỉnh của đồ thị được phân thành hai phần V_1 và V_2 bằng cách đặt những đỉnh mà có level nhỏ hơn hoặc bằng một nhưỡng L được xác định từ trước.

```
def pa_bfs(graph, vertex_ith: int, threshold: int):
    """Algorithm description:
        We can use basic graph search algorithm to solve graph
           partition algorithm :)
        Basically, the well-known BFS (Breadth-First-Search) algorithm
           can be modified to help us divide graph into two parts.
        BFS algorithm traverses the graph level by level and marks each
            vertex with the level in which it was visited.
        After completion of the traversal, the set of vertices of the
           graph is portioned into two parts $V_1$ and $V_2$ by putting
        all vertices with level less than or equal to a pre-determined
           threshold $L$ in the set V1 and putting the remaining
           vertices
        (with level greater than $L$) in the set $V_2$. $L$ is so
           chosen that $|V1|$ is close to $|V2|$.

    References:
```

```
    [1] Graph Partitioning, https://patterns.eecs.berkeley.edu/?page_id
        =571#1_BFS
    """
    L1 = set()
    L2 = set()

    # get the vertex `vertex_ith`.
    vertex = graph.getVertex(vertex_ith)

    # checking if not exist `vertex_ith` in Graph then raise error.
    if not vertex:
        message = 'Invalid vertex id, could not found vertex id `' + \
            str(vertex_ith) + '` in Graph'
        raise ValueError(get_log(message, log_type='ERROR'))

    # get the number of vertices.
    n = graph.numVertices

    # bool array for marking visited or not.
    visited = [False] * n

    # get the vertex_id for easy management.
    vertex_id = vertex.getId()

    # initializing a queue to handling which vertex is remaining.
    queue = [vertex_id]

    # marking the `vertex_id` is visited due to the beginning vertex.
    visited[vertex_id] = True

    path = []  # path to track the working state of BFS.

    level = 0
    while queue:
        # handling current vertex before removing out of queue.
        cur_pos = queue[0]

        # appending to path to track.
        path.append(cur_pos)

        # remove it out of queue
        queue.pop(0)

        # get all neighbors id of current vertex.
        neighbor_cur_pos = [
            x.id for x in graph.getVertex(cur_pos).getConnections()]

        # loop over the neighbor of current vertex.
        for neighborId in neighbor_cur_pos:
            # if not visited then push that vertex into queue.
            if not visited[neighborId]:
```

```
                visited[neighborId] = True
                graph.vertList[neighborId].level = level
                if level >= threshold:
                    L2.add(neighborId)
                else:
                    L1.add(neighborId)
                queue.append(neighborId)
        level += 1

    logging.info("Group A vertices: {}".format(L1))
    logging.info("Group B vertices: {}".format(L2))
    return L1, L2
```

2.5.3 Thuật toán Kernighan-Lin

Thuật toán Kernighan-Lin là một thuật tóa phân hoạch đồ thị dựa trên heuristic được đề xuất vào năm 1970. Nó nhận đầu vào là một đồ thị có trọng số $G = (V, E, w(.))$, trong đó là $w(.)$ là hàm trọng số cạnh, $|v| = 2n$ và một lưỡng phân hoạch khởi tạo (initial bi-partition) (V_1, V_2) của tập hợp các cạnh, trong đó $|V_1| = |V_2| = n$. Mục tiêu thuật toán là tạo ra một phân hoạch mới (V_1', V_2') mà $|V_1'| = |V_2'| = n$ mà tổng chi phí phân hoạch nhỏ hơn hoặc bằng chi phí trước đó.

Thuật toán Kernighan-Lin là một thuật toán phân hoạch cân bằng (balanced partitioning algorithm), tức là hai thành phần được tạo bởi thuật toán có cùng số lượng đỉnh (trong trường hợp tổng quá, có thể dùng thuật ngữ xấp xỉ bằng).

Thuật toán hoán đổi một cách tuần tự các cặp đỉnh cho đến khi đạt được tối ưu phân hoạch cục bộ, độ phức tạp thời gian của thuật toán là $O(N^3)$, trong đó N là số lượng đỉnh trong đồ thị G. Một trong những thuật toán kế thừa của thuật toán này là Fiduccia-Mattheyeses algorithm, cải thiện thời gian thực thi trong $O(|E|)$ và hoạt động tốt trên siêu đồ thị (hypergraph).

Mã giả của thuật toán Kernighan-Lin như sau:

```
Compute T = cost(A,B) for initial A,
    B Repeat
        Compute costs D(n) for all n in N
        Unmark all nodes in N
        While there are unmarked nodes
            Find an unmarked pair (a,b) maximizing gain(a,b)
            Mark a and b (but do not swap them) Update D(n) for all
               unmarked n,
              as though a and b had been swapped
        Endwhile

        Pick m maximizing Gain = Sk=1 to m gain(k)
        If Gain > 0 then … it is worth
        swapping
```

```
            Update newA = A - { a1…,,am } U { b1…,,bm }
            Update newB = B - { b1…,,bm } U { a1…,,am
                } Update T = T - Gain
        endif
    Until Gain <= 0
```

Cài đặt thuật toán bằng Python

Khởi tạo hai thành phần phân hoạch có kích thước bằng nhau

```
# Partition the vertices into two equal-sized groups A and B.
half = graph.numVertices // 2
for ind in range(half):
    graph.vertList[ind].partition_label = 'A'
    graph.vertList[ind + half].partition_label = 'B'
```

Tính toán `D_values` cho một lần duyệt

```
group_a = [v.id for k, v in graph.vertList.items()
                  if v.partition_label == "A"]
group_b = [v.id for k, v in graph.vertList.items()
           if v.partition_label == "B"]

D_values = {}

for idx, vertex in graph.vertList.items():
    for neighbor in vertex.connectedTo:
        if vertex.partition_label == neighbor.partition_label:
            vertex.internal_cost += vertex.getWeight(neighbor)
        else:
            vertex.external_cost -= vertex.getWeight(neighbor)

    D_values.update(
        {idx: graph.vertList[idx].external_cost + graph.vertList[idx].
           internal_cost})
```

Tính toán độ lợi cho tất cả các hoán vị đỉnh có thể có giữa hai thành phần phân hoạch.

```
# Compute the gains for all possible vertex swaps between groups A and
   B.
gains = []
for a in group_a:
    for b in group_b:
        c_ab = graph.vertList[a].getWeight(graph.vertList[b])
        gain = D_values[a] + D_values[b] - (2 * c_ab)
        gains.append([[a, b], gain])
```

Sắp xếp và lấy ra độ lợi lớn nhất.

```
# Sort the gains in descending order and get the maximum gain.
gains = sorted(gains, key=lambda x: x[1], reverse=True)
```

```
max_gain = gains[0][1]

if max_gain <= 0:
    break
```

Lấy ra cặp đỉnh với độ lợi lớn nhất và hoán vị nhãn phân hoạch của chúng.

```
# Get the pair of vertices with the maximum gain and swap their
   partition labels.
pair = gains[0][0]
group_a.remove(pair[0])
group_b.remove(pair[1])

graph.vertList[pair[0]].partition_label = "B"
graph.vertList[pair[1]].partition_label = "A"
```

Cập nhật độ lợi cho các đỉnh trong phân hoạch.

```
# Update the D values of the vertices in groups A and B.
for x in group_a:
    c_xa = graph.vertList[x].getWeight(graph.vertList[pair[0]])
    c_xb = graph.vertList[x].getWeight(graph.vertList[pair[1]])
    D_values[x] += 2 * c_xa - 2 * c_xb

for y in group_b:
    c_ya = graph.vertList[y].getWeight(graph.vertList[pair[0]])
    c_yb = graph.vertList[y].getWeight(graph.vertList[pair[1]])
    D_values[x] += 2 * c_ya - 2 * c_yb

# Update the total gain.
total_gain += max_gain
```

Toàn bộ mã nguồn thuật toán

```
def pa_kl(graph):
    """Algorithm description:
    The KL algorithm is a heuristic algorithm for partitioning a graph
       into two disjoint sets of vertices with roughly equal sizes,
    while minimizing the total weight of the edges between the two sets
       . The algorithm works by iteratively swapping vertices
    between the two sets in a way that reduces the total weight of the
       cut.

    The algorithm starts by dividing the vertices into two sets, A and
       B, with roughly equal sizes.
    Then, it iteratively performs the following steps:
        Step 1: Compute the net gain for moving each vertex from set A
           to set B, and vice versa.
        The net gain is defined as the difference between the sum of
           the weights of the edges
        that connect the vertex to its current set and the sum of the
```

```
            weights of the edges
        that connect the vertex to the other set.

        Step 2: Select the pair of vertices with the highest net gain,
            one from set A and one from set B, and swap them.

        Step 3: Update the net gains for the affected vertices, and
            repeat the process until no more swaps can be made.

    The algorithm terminates when no more swaps can be made that
        improve the total weight of the cut.
    The final partitioning is the one that results in the minimum cut.

    References:
    [1] Graph Partitioning, https://patterns.eecs.berkeley.edu/?page_id
        =571#1_BFS
    """
    # Partition the vertices into two equal-sized groups A and B.
    half = graph.numVertices // 2
    for ind in range(half):
        graph.vertList[ind].partition_label = 'A'
        graph.vertList[ind + half].partition_label = 'B'

    total_gain = 0

    # Keep track of the total gain in each iteration.
    for _ in range(half):
        # Get the vertices in groups A and B and their D values.
        group_a = [v.id for k, v in graph.vertList.items()
                   if v.partition_label == "A"]
        group_b = [v.id for k, v in graph.vertList.items()
                   if v.partition_label == "B"]

        D_values = {}

        for idx, vertex in graph.vertList.items():
            for neighbor in vertex.connectedTo:
                if vertex.partition_label == neighbor.partition_label:
                    vertex.internal_cost += vertex.getWeight(neighbor)
                else:
                    vertex.external_cost -= vertex.getWeight(neighbor)

            D_values.update(
                {idx: graph.vertList[idx].external_cost + graph.
                    vertList[idx].internal_cost})

        # Compute the gains for all possible vertex swaps between
            groups A and B.
        gains = []
        for a in group_a:
            for b in group_b:
```

```
56                  c_ab = graph.vertList[a].getWeight(graph.vertList[b])
57                  gain = D_values[a] + D_values[b] - (2 * c_ab)
58                  gains.append([[a, b], gain])
59
60          # Sort the gains in descending order and get the maximum gain.
61          gains = sorted(gains, key=lambda x: x[1], reverse=True)
62          max_gain = gains[0][1]
63
64          if max_gain <= 0:
65              break
66
67          # Get the pair of vertices with the maximum gain and swap their
                partition labels.
68          pair = gains[0][0]
69          group_a.remove(pair[0])
70          group_b.remove(pair[1])
71
72          graph.vertList[pair[0]].partition_label = "B"
73          graph.vertList[pair[1]].partition_label = "A"
74
75          # Update the D values of the vertices in groups A and B.
76          for x in group_a:
77              c_xa = graph.vertList[x].getWeight(graph.vertList[pair[0]])
78              c_xb = graph.vertList[x].getWeight(graph.vertList[pair[1]])
79              D_values[x] += 2 * c_xa - 2 * c_xb
80
81          for y in group_b:
82              c_ya = graph.vertList[y].getWeight(graph.vertList[pair[0]])
83              c_yb = graph.vertList[y].getWeight(graph.vertList[pair[1]])
84              D_values[x] += 2 * c_ya - 2 * c_yb
85
86          # Update the total gain.
87          total_gain += max_gain
88
89          # break
90
91      # Get the cutset size and the vertex IDs in groups A and B.
92      cutset_size = graph.compute_partition_cost()
93      group_a = [v.id for k, v in graph.vertList.items()
94                 if v.partition_label == "A"]
95      group_b = [v.id for k, v in graph.vertList.items()
96                 if v.partition_label == "B"]
97
98      # Print the results
99      logging.info("Cut size: {}".format(cutset_size))
100     logging.info("Group A vertices: {}".format(group_a))
101     logging.info("Group B vertices: {}".format(group_b))
102
103     return cutset_size, group_a, group_b
```

2.5.4 Thuật toán Fiduccia-Mattheyses Partitioning

2.5.5 Thuật toán Spectral Bisection

Lý thuyết spectral bisection được phát triển vào năm 1970 bởi Fiedler. Nó dựa trên tính toán vector trị riêng của ma trận Laplacian matrix của đồ thị. Với một đồ thị G, chúng ta định nghĩa ma trận Laplacian của nó $L(G)$ như sau:

- Laplacian matrix $L(G)$ của đồ thị $G = (V, E)$ là một ma trận đối xứng, kích thước $|V| \times |V|$ với một dòng và một cột cho mỗi đỉnh, và mỗi vị trí trong ma trận được định nghĩa bởi:
  - $L(G)(i, i)$ = bậc của đỉnh i
  - $L(G)(i, j) = -1$ nếu $i \neq j$ và tồn tại cạnh (i, j)
  - $L(G)(i, j) = 0$, nếu trong trường hợp khác.

Thuật toán Spectral Bisection

- Bước 1: Xây dựng ma trận Laplacian matrix $L(G)$ cho đồ thị đầu vào $G = (V, E)$
- Bước 2: Tính toán vector riêng v_2 tương ứng với trị riêng thứ hai λ_2 của ma trận Laplacian matrix $L(G)$
- Bước 3: Với mỗi đỉnh $i \in V$:
  - nếu $v_2[i] < 0$ đặt đỉnh này vào phân hoạch A
  - ngược lại thì đặt đỉnh này phân hoạch B.

Cài đặt thuật toán bằng Python

```python
def pa_sb(graph):
    """Algorithm description:

    The Spectral Bisection algorithm is a graph partitioning algorithm
       that is based on the eigenvalues and eigenvectors of the graph
       Laplacian matrix.
    The algorithm works by iteratively bisecting the graph into two
       disjoint sets of vertices with roughly equal sizes, while
       minimizing
    the total weight of the edges between the two sets.

    The algorithm starts by computing the eigenvectors corresponding to
        the smallest eigenvalues of the graph Laplacian matrix.
    The eigenvectors are then used to partition the graph into two sets
        of vertices, A and B, by assigning each vertex to
    the set that corresponds to the sign of the corresponding
       eigenvector component. This initial partitioning is not
       guaranteed
```

```
    to be balanced, but is usually close to it.

    Then, the algorithm iteratively improves the partitioning by performing the following steps:
        Step 1: Compute the cut size of the current partitioning.
        Step 2: Compute the eigenvectors corresponding to the second-smallest eigenvalues of the graph Laplacian matrix.
        Step 3: Compute the projection of the current partitioning onto the space spanned by the eigenvectors corresponding
        to the smallest and second-smallest eigenvalues. This projection is used to determine a new partitioning by assigning
        each vertex to the set that corresponds to the sign of the projection.
        Step 4: Compute the cut size of the new partitioning, and select it if it has a smaller cut size than the current partitioning.
        Otherwise, discard the new partitioning and continue with the current one.

    The algorithm terminates when no more improvements can be made, or when a desired balance ratio between the two sets is achieved.
    The final partitioning is the one that results in the minimum cut.

    References:
    [1] Graph Partitioning, https://patterns.eecs.berkeley.edu/?page_id=571#1_BFS
    """
    laplacian_matrix = graph.compute_laplacian_matrix()
    logging.info('Computing the eigenvectors and eigenvalues')
    eigenvalues, eigenvectors = np.linalg.eigh(laplacian_matrix)

    # Index of the second eigenvalue
    index_fnzev = np.argsort(eigenvalues)[1]
    logging.info('Eigenvector for #{} eigenvalue ({}): '.format(
        index_fnzev, eigenvalues[index_fnzev]), eigenvectors[:, index_fnzev])

    # Partition on the sign of the eigenvector's coordinates
    partition = [val >= 0 for val in eigenvectors[:, index_fnzev]]

    # Compute the edges in between
    logging.info('Compute the edges in between two groups.')
    a = [idx for (idx, group_label) in enumerate(partition) if group_label]
    b = [idx for (idx, group_label) in enumerate(partition) if not group_label]

    group_a = [v.id for k, v in graph.vertList.items()
               if v.id in a]
    group_b = [v.id for k, v in graph.vertList.items()
```

```
                if v.id in b]

    logging.info("Group A vertices: {}".format(group_a))
    logging.info("Group B vertices: {}".format(group_b))
    return group_a, group_b
```

2.5.6 Tối ưu thuật toán phân hoạch đồ thị bằng thành phần liên thông

Một trong những cách tối ưu hóa thuật toán phân hoạch đồ thị bằng cách sử dụng thành phần liên thông có thể được thực hiện như sau:

- Bước 1: Xác định thành phần liên thông trong đồ thị bằng cách thuật toán dựa trên DFS hoặc BFS.
- Bước 2: Với mỗi thành phần liên thông, tính toán một trọng số mà thể hiện tính cân bằng của các đỉnh trong mỗi phân hoạch. Một trong những hàm trọng khả thi ở đây là trị tuyệt đố giữa số lượng nút trong mỗi phân hoạch.
- Bước 3: Sắp xếp những thành phần liên thông theo trọng số giảm dần.
- Bước 4: Bắt đầu với thành phần liên thông ứng với trọng số lớn nhất nhất, thực hiện phân hoạch nó thành hai phân hoạch mà có tính cân bằng nhất có thể. Bước này có thể sử dụng một số thuật toán heuristic như Kernighan-Lin hay Fiduccia-Mattheyses.
- Bước 5: Lặp lại bước 4 với thành phần liên thông kế tiếp, thuật toán dừng khi tất cả các thành phần liên thông đã được xử lý.

Cài đặt bằng Python

```
def pa_scc_kl(graph):
    # Identify the connected components in the graph using DFS
    scc_lst = graph.find_strongly_connected_components()

    # Compute the weight of each component
    weights = [abs(len(c) - graph.numVertices/2) for c in scc_lst]

    # Sort the components by weight in descending order
    sorted_scc_lst = [c for _,c in sorted(zip(weights, scc_lst),
        reverse=True)]

    # Initialize the partition as an empty list
    partition = {}

    # Iterate over each connected component and attempt to partition it
    for component in sorted_scc_lst:
        print(component)
        subgraph = Graph()
```

```
18
19          for vertex in component:
20              subgraph.addVertex(vertex)
21              for neighbor in graph.vertList[vertex].connectedTo:
22                  if neighbor.getId() not in component:
23                      subgraph.addVertex(neighbor.getId())
24                      subgraph.addEdge(vertex, neighbor.getId(), graph.
                            vertList[vertex].getWeight(neighbor))
25
26
27          # Partition the subgraph using a heuristic algorithm
28          # such as Kernighan-Lin or Fiduccia-Mattheyses
29          # Here we'll use Kernighan-Lin
30          cutset_size, partition_1, partition_2 = pa_kl(subgraph)
31
32          # Add the partitions to the overall partition
33          partition.setdefault('partition_1', []).append(partition_1)
34          partition.setdefault('partition_2', []).append(partition_2)
35
36      return partition
```

Thuật toán tìm kiếm thành phần liên thông (mạnh) Kosaraju's algorithm

- Bước 1: Thực hiện một Depth First Search (DFS) trên đồ thị gốc và ghi nhớ thứ tự của các đỉnh đã được duyệt.
- Bước 2: Đảo ngược hướng của tất cả các cạnh để hình thành một đồ thị mới.
- Bước 3: Thực hiện DFS trên đồ thị mới vừa hình thành theo thứ tự vừa nhận được ở bước 1. Đánh dấu mỗi nút thuộc cùng một thành phần liên thông.
- Bước 4: Lặp lại từ bước 3 và 4 cho tất cả các nút chưa được duyệt trên đồ thị.

Cài đặt bằng Python

```
1  def find_strongly_connected_components(self):
2      """
3      Kosaraju algorithm
4      """
5      # Perform a Depth First Search (DFS) on the original graph
6      # and keep track of the order in which the nodes are visited.
7      dfspath = self.DFS()
8
9      # Reverse the directions of all edges in the graph to obtain a new
           graph.
10     rg = self.reversing()
11
12     # Perform a DFS on the new graph, visiting the nodes in the reverse
           order obtained in step 1.
13     # As you perform the DFS, mark each node as belonging to the same
           SCC.
```

```
    closed_set: list[int] = []
    scc_set = []

    for idx in range(rg.numVertices):
        if idx not in closed_set:
            # open_set: list[int] = [idx]
            scc = []

            while dfspath:
                cur_vertex: Vertex = rg.getVertex(dfspath.pop())
                cur_vertex_id = cur_vertex.getId()

                if cur_vertex_id not in closed_set:
                    closed_set.append(cur_vertex_id)
                    scc.append(cur_vertex_id)

                    neighbors = [x.id for x in cur_vertex.
                        getConnections()]

                    for neighbor in neighbors:
                        if neighbor not in closed_set:
                            dfspath.append(neighbor)
            scc_set.append(scc)

    return scc_set
```

3 Tài liệu tham khảo

- Một số bài toán cơ bản trong phân tích dữ liệu. (n.d.). Thuc Nguyen Dinh.