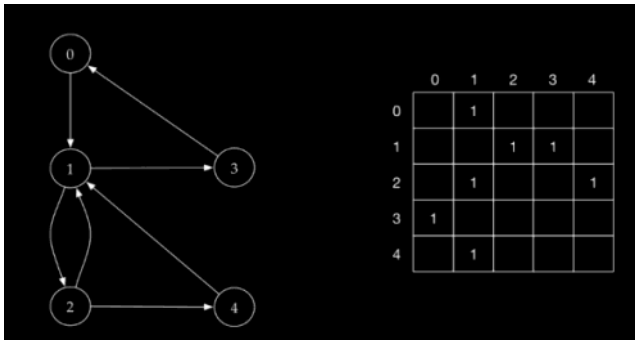
Aula PL #02

26 de setembro de 2023 16:07

Adjacency matrices

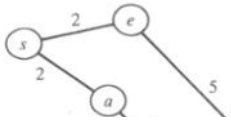


0 → 1
1 → 2 → 3
2 → 1 → 4
3 → 0
4 → 1

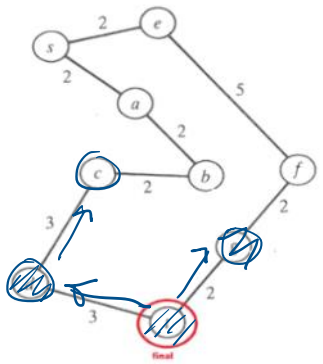
```python
class Node:
    def __init__(self, name, max_distance):
        self._name = name
        self._adjacent = {}
        self._distance = max_distance
        self._visited = False
        self._previous = None

    def add_neighbour(self, neighbour, weight=0):
        self._adjacent[neighbour] = weight

    def get_adjacent(self):
        return self._adjacent.items()

    def get_name(self):
        return self._name

    def get_weight(self, neighbour):
        return self._adjacent[neighbour]

    @property
    def distance(self):
        return self._distance

    @distance.setter
    def distance(self, d):
        self._distance = d

    @property
    def visited(self):
        return self._visited

    @visited.setter
    def visited(self, v):
        self._visited = v

    @property
    def previous(self):
        return self._previous

    @previous.setter
    def previous(self, p):
        self._previous = p

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

    def __lt__(self, other):
        return self.distance < other.get_distance()
```

```python
class Graph:
    def __init__(self, max_value, is_directed):
        self.nodes_map = {}  # key -> Node
        self.max_value = max_value
        self.is_directed = is_directed
        self.num_nodes = 0
```

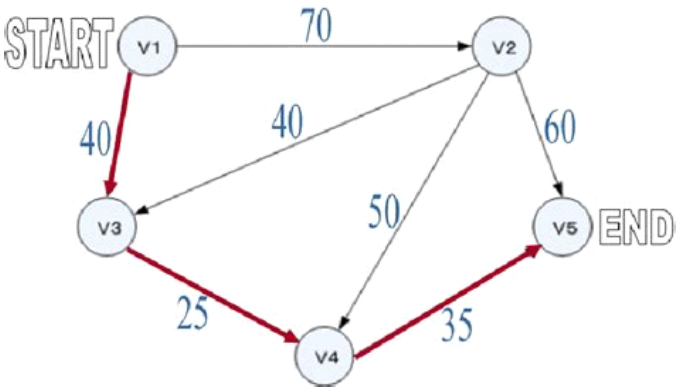
```python
7 usages
def add_edge(self, from_node, to_node, weight=0):
    if from_node not in self.nodes_map:
        self.add_node(from_node)
    if to_node not in self.nodes_map:
        self.add_node(to_node)
    self.nodes_map[from_node].add_neighbour(self.nodes_map[to_node], weight)
    if not self.is_directed:  # if undirected
        self.nodes_map[to_node].add_neighbour(self.nodes_map[from_node], weight)
```



```
        self.nodes_map[from_node].add_neighbour(self.nodes_map[to_node], weight)
        if not self.is_directed:  # if undirected
            self.nodes_map[to_node].add_neighbour(self.nodes_map[from_node], weight)
```



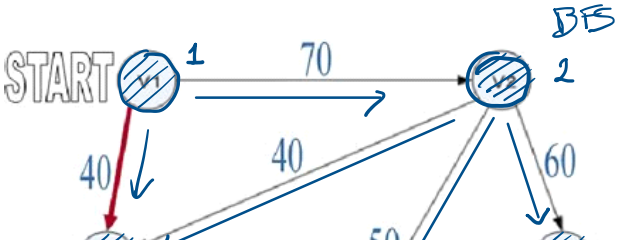
Visited: t, d, g
Result: t, d,
Queue: g, c

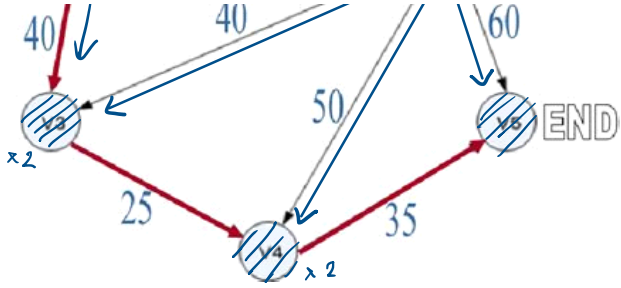



```
1 ['V1', 'V3', 'V4', 'V5']
2 ['V1', 'V3', 'V2', 'V4', 'V5']
3 ['V1', 'V3', 'V4', 'V5', 'V2']
```

1 → DIJKSTRA
2 → BFS
3 → DFS

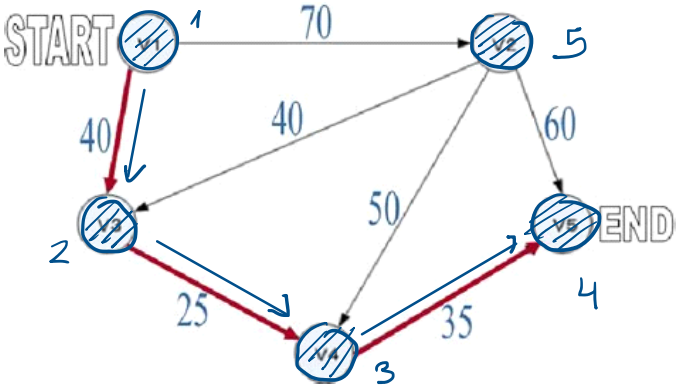
```
1 usage
def bfs(self, start):
    start_node_obj = self.get_node(start)
    if not start_node_obj:
        return None

    path = []
    self._reset_nodes()

    queue = deque()
    queue.append(start_node_obj)
    start_node_obj.visited = True

    while queue:
        current_node = queue.popleft()
        path.append(current_node.get_name())
        for (neighbour, _) in current_node.get_adjacent():
            if not neighbour.visited:
                neighbour.visited = True
                queue.append(neighbour)

    return path
```



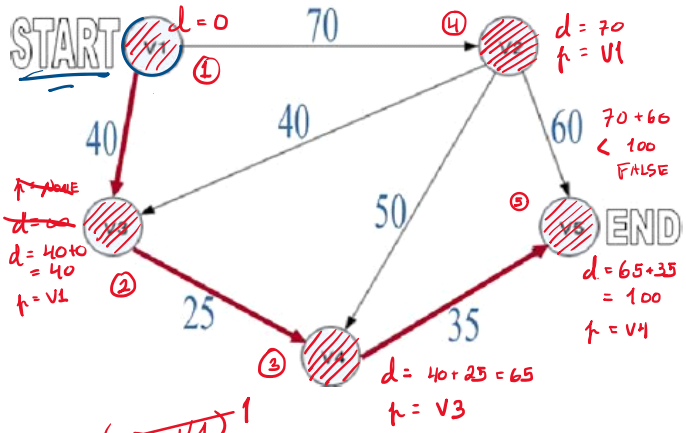


Queue : ~~V1~~, ~~V2~~, ~~V3~~, ~~V4~~, ~~V5~~

```
1 usage
def dfs(self, start):
    start_node_obj = self.get_node(start)
    if not start_node_obj:
        return None

    self._reset_nodes()

    def dfs_recursive(node, p):
        node.visited = True
        p.append(node.get_name())

        for (neighbour, _) in node.get_adjacent():
            if not neighbour.visited:
                dfs_recursive(neighbour, p)

    path = []
    dfs_recursive(start_node_obj, path)
    return path
```



```
1 usage
def dijkstra(self, start):

    if start not in self.nodes_map:
        raise ValueError("Start node must exist in the graph.")

    self._reset_nodes()

    start_node_obj = self.nodes_map[start]
    start_node_obj.distance = 0

    priority_queue = [(0, start_node_obj)]

    while priority_queue:
        (current_distance, current_node) = heapq.heappop(priority_queue)
        if not current_node.visited:
            current_node.visited = True
            for (neighbour, weight) in current_node.get_adjacent():
                if not neighbour.visited:
                    distance = current_distance + weight
                    if distance < neighbour.distance:
                        neighbour.distance = distance
                        neighbour.previous = current_node
                        heapq.heappush(priority_queue, _item: (distance, neighbou
```



~~(0, V1)~~ 1

~~(40, V3)~~ 2

~~(70, V2)~~ 4

~~(65, V4)~~ 3

~~(100, V5)~~ 5

Prioridade: Distância