RME 3111 - Lab 1: BFS

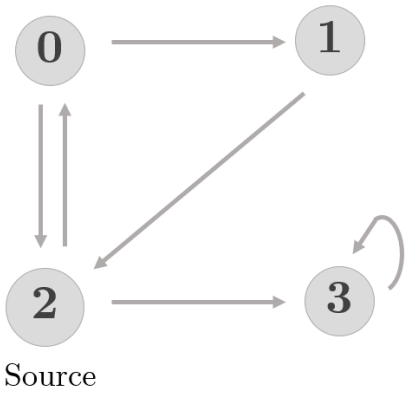
Course Instructor: Dr. Sejuti Rahman

Md. Arban Hossain (SH-092-005)

27 Jan 2023

Part A

Here is the given graph.



Pen and Paper Simulation

Before implementing the code, let's search through the graph manually documenting our process. Use vertex #2 as our source and vertex #1 as our destination.

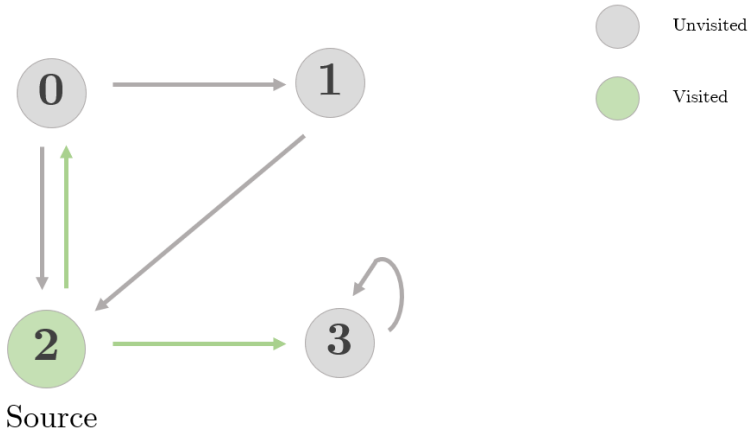
To start the Breadth First Search, push the source vertex into a queue **Q**. Use a *set* **Visited** to keep track of the visited vertices. At each iteration of the search, pop an item from the queue, mark it as visited, and check each of its neighboring vertices. We have found our path if one of the neighbors is the destination. Otherwise, push the neighbors into the queue if they have not been visited and resume the process. Terminate the process when we have: (a) reached our destination, or (b) visited all vertices reachable from the

Before starting the search, the states of **Q** and **Visited** are:
Q: 2
Visited: {}

Iteration 1

- Pop an item (2) from **Q**. Add it to **Visited**.
- Check its neighbors. In this case, 3 and 0. None of them is the destination. None of them is in **Visited**.
- Push 3 and 0 into **Q**.

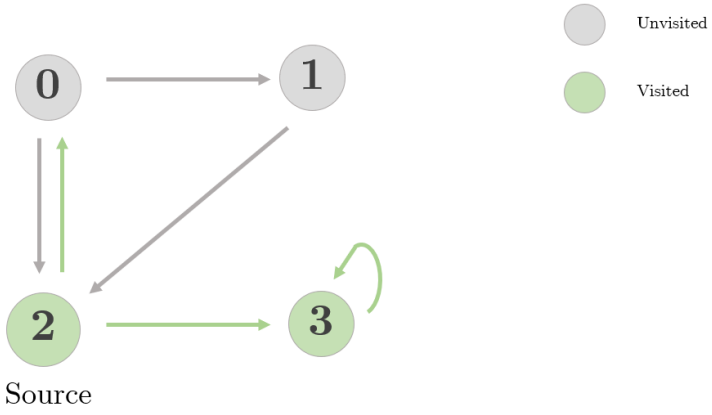
Q: 3 0
Visited: {2}



Iteration 2

- Pop an item (3) from **Q**. Add it to **Visited**.
- Check its neighbors. In this case, 3. It is not the destination. It is in **Visited**. So do nothing.

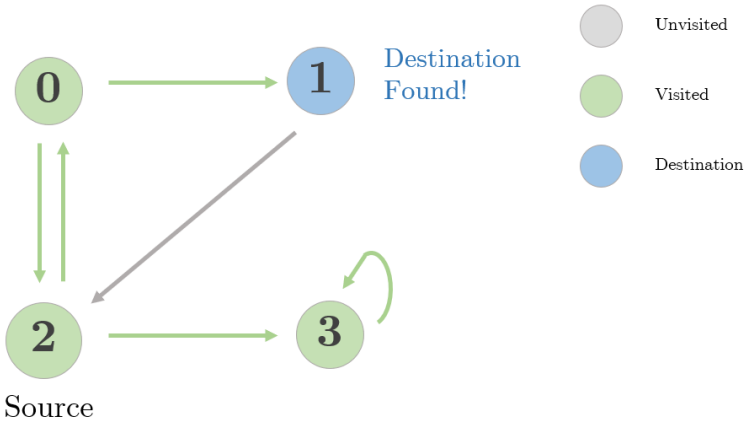
Q: 0
Visited: {2, 3}



Iteration 3

- Pop an item (0) from **Q**. Add it to **Visited**.
- Check its neighbors. In this case, 2 and 1. One of them (1) is the destination. Add it to **Visited**. We can now terminate our search and return a **True** value.

Q: Empty
Visited: {2, 3, 0, 1}



Part B

Source Code

Python3 source code for the problem follows.

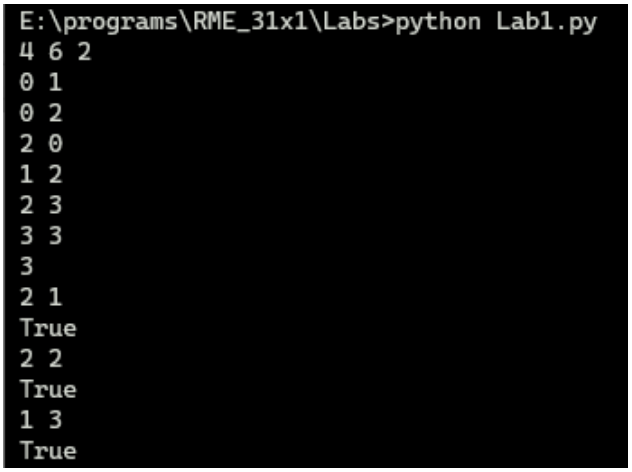
```
### Lab1.py

class Node:

  def __init__(self, elem):
    self.value = elem
    self.next = None

class Queue:

  def __init__(self):
    self.head = None
    self.tail = None

  # O(1)
  def is_empty(self):
    return self.head == None

  # O(1)
  def enqueue(self, elem):
    node = Node(elem)
    if self.head == None:
      self.head = node
      self.tail = node
    else:
      self.tail.next = node
      self.tail = node

  # O(1)
  def dequeue(self):
    if self.head == None: return None
    else:
      node = self.head
      self.head = self.head.next
      return node.value

class Vertex:

  def __init__(self, name, value=None):
    self.value = value
```

```
    self.name = name

# Space: O(V) for vertices. O(E) for edges. Total O(V+E)
class Graph:

  def __init__(self):
    self.adjacency_list = {}
    self.vertices = {}

  def add_directed_edge(self, v1, v2):
    # add vertex in vertices if not already there
    if v1.name not in self.vertices:
      self.vertices[v1.name] = v1
    if v2.name not in self.vertices:
      self.vertices[v2.name] = v2

    # add v2 in v1's adjacency list
    if v1.name in self.adjacency_list:
      self.adjacency_list[v1.name].append(v2.name)
    else:
      self.adjacency_list[v1.name] = [v2.name]

    if v2.name not in self.adjacency_list:
      self.adjacency_list[v2.name] = []

  # Time: O(V+E)
  # Space: O(V)
  def path_exists_bfs(self, v1, v2):
    queue = Queue()
    queue.enqueue(v1)
    visited = set()
    while not queue.is_empty():
      current = queue.dequeue()
      if current not in visited:
        visited.add(current)
        if current == v2:
          return True
        for neighbor in self.adjacency_list[current]:
          if neighbor == v2:
            return True
          queue.enqueue(neighbor)
    return False
```

```
if __name__ == "__main__":
  no_of_vertices, edges, source = map(int, input().split())

  graph = Graph()

  for _ in range(edges):
    v1, v2 = [Vertex(x) for x in input().split()]
    graph.add_directed_edge(v1, v2)

  for _ in range(int(input())):
    v1, v2 = [x for x in input().split()]
    print(graph.path_exists_bfs(v1, v2))
```

Output

Using the provided input file, the output of the program was:



Complexity Analysis

Time Complexity

The function **path_exists_bfs()** iterates over all the vertices at most once. For each vertex, it iterates over all the edges it is connected to. During each iteration, it takes **O(1)** time for the *push* and *pop* operations of the queue. Looking up an item in a Python *set* also takes **O(1)** time. Considering the graph has **V** vertices and **E** edges, in the worst case it will take **O(V+E)** time to finish the search.

Space Complexity

The adjacency list is implemented using a Python dictionary. During each iteration, all neighboring vertices of the current vertex are present in the queue. In the worst case, this number can be equal to the number of total vertices in the graph. So it takes **O(V)** space.

Links

The source files can be found at github.com/arbanhossain/RME_31x1.