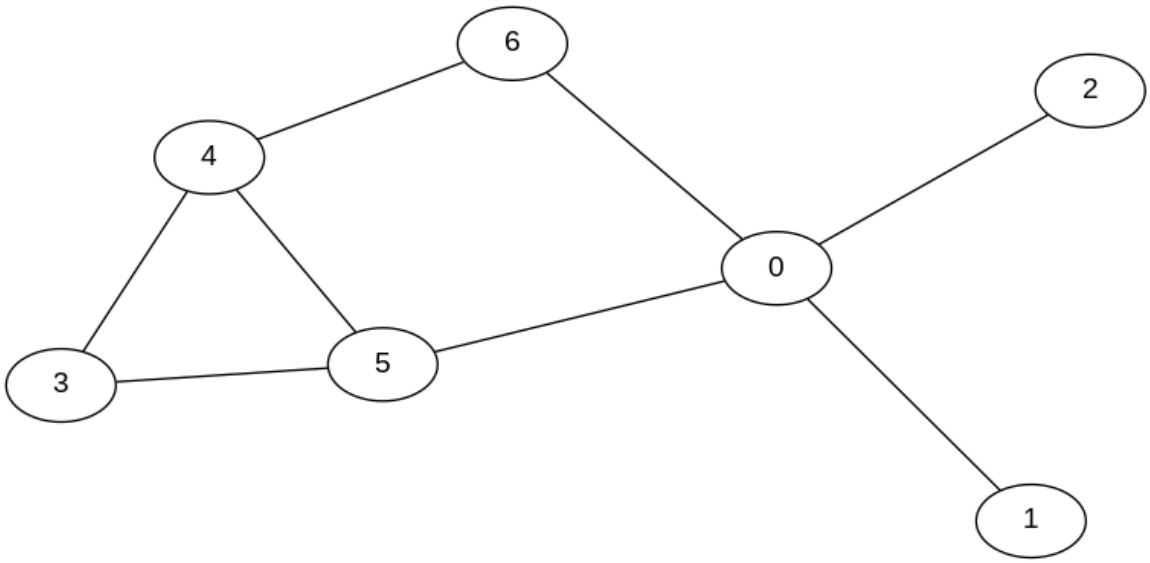
Lecture 11.3 : Searching graphs (again)

Introduction

- We present an approach to searching through a graph called *breadth-first search*.
- We previously saw that depth-first search (DFS) is a recursive algorithm that uses back-tracking to identify and explore novel paths.
- An alternative to DFS is breadth-first search (BFS).
- BFS does not use recursion but instead makes use of a queue.
- BFS can be used to find all vertices connected to a given vertex.
- BFS can be used to find a path betwen two vertices (should one exist).
- Before we code it, let's look at BFS in action so we can see how it works.
- Here's a video I made.
- Here's a website with BFS animations.
- Here's a website with lots of algorithm animations.

Our graph



Graph description

- As usual, we describe a graph with a simple text file where the first line defines the number of vertices in the graph and the following lines define the edges (i.e. which vertices are connected to which).

```
$ cat graph01.txt
7
0 1
0 2
0 5
0 6
3 4
```

```
3 5
4 5
4 6
```

Basic graph class

- Our basic graph class looks as follows.

```
class Graph(object):

   def __init__(self, V):
      self.V = V
      self.adj = {}
      for v in range(V):
         self.adj[v] = list()

   def addEdge(self, v, w):
      self.adj[v].append(w)
      self.adj[w].append(v)
```

Coding BFS

- We will not add a new method to the `Graph` class but will instead create a new `BFSPaths` class.
- The input to the `BFSPaths` class is the graph we wish to explore using BFS and a starting vertex.

```
class BFSPaths(object):

   def __init__(self, g, s):
      self.g = g
      self.s = s
      self.marked = [False for _ in range(g.V)]
      self.parent = [False for _ in range(g.V)]
      self.bfs(s)

   def bfs(self, s):
      queue = [s]
      self.marked[s] = True

      while queue:
         v = queue.pop(0)
         for w in self.adj[v]:
            if not self.marked[w]:
               queue.append(w)
               self.parent[w] = v
               self.marked[w] = True

   # Return True if there is a path from s to v
   def hasPathTo(self, v):
      return self.marked[v]

   # Return path from s to v (or None should one not exist)
   def pathTo(self, v):
      if not self.hasPathTo(v):
         return None
      path = [v]
      while v != self.s:
         v = self.parent[v]
         path.append(v)
      return path[::-1]
```

Applying BFS to a graph

```
from graph import Graph, BFSPaths

with open('graph01.txt') as f:

   V = int(f.readline())

   g = Graph(V)

   for line in f:

       v, w = [int(t) for t in line.strip().split()]
       g.addEdge(v, w)

   paths = BFSPaths(g, 0)

   print(paths.hasPathTo(6))

   print(paths.pathTo(6))
```

```
True
[0, 6]
```