Graph-BFS-DFS

December 5, 2017

0.1 BFS

- O(V + E)

```
def bfs (vertex):
    Queue queue
    vertex set visited true
    queue.enqueue(vertex)

    while queue not empty:
        actual = queue.dequeue()

        for v in actual neighbours:
            if v is not visited:
                v set visited true
                queue.enqueue(v)
```

```
In [9]: class Node(object):

          def __init__(self, name):
              self.name = name;
              self.adjacencyList = [];
              self.visited = False;
              self.predecessor = None;

        class BreadthFirstSearch(object):

          def bfs(self, startNode):

            queue = [];
            queue.append(startNode)
            startNode.visited = True;

            while queue:

              actualNode = queue.pop(0);
              print("%s " % actualNode.name)
```

```
            for node in actualNode.adjacencyList:
                if not node.visited:
                    node.visited = True
                    queue.append(node);
```

```
In [10]: node1 = Node("A");
         node2 = Node("B");
         node3 = Node("C");
         node4 = Node("D");
         node5 = Node("E");

         node1.adjacencyList.append(node2);
         node1.adjacencyList.append(node3);
         node2.adjacencyList.append(node4);
         node4.adjacencyList.append(node5);

         bfs = BreadthFirstSearch();
         bfs.bfs(node1);
```

```
A
B
C
D
E
```

```
In [ ]:
```

```
In [ ]:
```

0.2 DFS

```
def dfs(vertex):
    Stack stack
    vertex set visited True
    stack.push(vertex)

    while stack not empty:
        actual = stack.pop()

        for v in actual neighours:
            if v is not visited:
                v set visited True
                stack.push(v)


def dfs(vertex):
```

```
    vertex set visisted True
    print vertex

    for v in vertex neighbours:
        if v is not visited:
            dfs(v)
```

In [14]:
```python
class Node(object):
  def __init__(self, name):
    self.name = name;
    self.adjacenciesList = [];
    self.visited = False;
    self.predecessor = None;

class DepthFirstSearch(object):

  def dfs(self, node):

    node.visited = True;
    print("%s" % node.name);

    for n in node.adjacenciesList:
      if not n.visited:
        self.dfs(n)
```

In [16]:
```python
node1 = Node("A");
node2 = Node("B");
node3 = Node("C");
node4 = Node("D");
node5 = Node("E");

node1.adjacenciesList.append(node2);
node1.adjacenciesList.append(node3);
node2.adjacenciesList.append(node4);
node4.adjacenciesList.append(node5);

dfs = DepthFirstSearch();
dfs.dfs(node1);
```

```
A
B
D
E
C
```

In [ ]:

0.3 Memory Management

- BFS:
  - ***Space complexity: O(N)***. Because at the leaves -> if we have N items stored in the balanced tree ~ then there will be N/2 leave nodes
  - So we have to store O(N) items if we want to traverse a tree that contains N items!!!
- DFS:
  - Here we have to backtrack (pop item from stack): so basically we just have to store as many items n the stack as the height of the tree -> which is log(N)!!!
  - ~ so the memory complexity will be O(logN)
- That's why depth-first search is preferred most of the times. There may be some situations where BFS is better ~ artificial intelligence, robot movements

```
In [ ]:
```