MATH 580A 7/5 Notes

1 Recap and Introduction

Last lecture we covered how to write lambda functions and at this point we have covered most of the foundational aspects of Python needed for computational work. There are some additional aspects of the language, primarily related to object-oriented programming, that we will skip since it won't be applicable for our purposes.

This week we will focus on graphs, which are a fundamental structure in combinatorics and computer science. We will look at some classic algorithms on graphs and in the assignment see a variety of applications of them. As we move towards the more mathematical aspect of the course, we'll also start to use more libraries in common use by the mathematical community. For graphs, NetworkX will be the library of choice.

2 Using NetworkX For Graphs

In the previous lectures, we have seen how we can represent directed graphs on our own using dictionaries in Python. Although very flexible, it can be cumbersome to extend this representation to handle assigning weights to each edge and attributes to each node. Furthermore, we don't have an easy way to write our dictionary representation into a file to store nor do we have a way to load graphs from files. We will transition to using the external library NetworkX which provides much more fully featured graph objects in Python to work. The library implements representations for graphs, directed graphs, and multigraphs. It also has many standard graph algorithms built-in and has functions to generate various graph types such as trees. NetworkX is integrated with Matplotlib for displaying graphs and is also the backing library in SageMath, a platform for doing mathematical computation built on top of Python and various Python libraries.

Listing 1 demonstrates how to define graphs and directed graphs in NetworkX. The function `nx.Graph()` creates a new empty graph. We can use `<graph>.add_node(<node_name>)` in order to add a single node or `<graph>.add_nodes_from(<node_list>)` to add a list of nodes. Similarly, `add_edge()` and `add_edges_from()` is used to add edges to our graph. In NetworkX, edges are represented as tuples of node labels `(<from>, <to>)`. NetworkX has many built-in graphs and we could have called `nx.octahedral_graph()` to obtain the same graph we constructed in lines 3 through 19.

Lines 23 and 24 demonstrate adding *node attributes*. In general, you can access the nodes in a graph via `<graph>.nodes` and treat it as a dictionary whose keys are the node names. The value of this dictionary is a node object which you can think of as another dictionary to assign arbitrary attributes to. Here, nodes `"A"` and `"D"` get assigned an extra attribute called `"comment"`. More generally, you might use attributes to store information on a graph such as the distance of a node from a certain starting node. Although you can set attributes on each node or edge individually,

```
1  import networkx as nx
2
3  octahedron_graph = nx.Graph()
4  octahedron_graph.add_node("A")
5  octahedron_graph.add_nodes_from(["B", "C", "D", "E", "F"])
6  octahedron_graph.add_edge("A", "B")
7  octahedron_graph.add_edges_from([
8    ("A", "C"),
9    ("A", "F"),
10   ("A", "E"),
11   ("B", "C"),
12   ("B", "D"),
13   ("B", "F"),
14   ("C", "E"),
15   ("C", "D"),
16   ("D", "E"),
17   ("D", "F"),
18   ("E", "F"),
19 ])
20
21 directed_graph = nx.DiGraph()
22 directed_graph.add_nodes_from(["A", "B", "C", "D"])
23 directed_graph.nodes["A"]["comment"] = "source"
24 directed_graph.nodes["D"]["comment"] = "sink"
25 directed_graph.add_edges_from([
26   ("A", "B"),
27   ("A", "C"),
28   ("B", "C"),
29   ("C", "B"),
30   ("B", "D"),
31   ("C", "D"),
32 ])
33 nx.set_edge_attributes(directed_graph, 1, "weight")
34 directed_graph.edges[("A","B")]["weight"] = 2
35 directed_graph.edges[("B","C")]["weight"] = 2
36 directed_graph.edges[("C","B")]["weight"] = 2
37
38 petersen_graph = nx.read_adjlist("graphs/petersen_graph.txt")
```

Listing 1: Defining Graphs in NetworkX.

you can also set an attribute to all the nodes or edges at the same time. Line 33 shows how to use the `set_edge_attributes()` function in order set the `"weight"` attribute of all edges in the graph to 1. We then override the weight attribute on a few of the edges in the subsequent lines.

Line 38 demonstrates how to create a graph from data stored in a file. There a number of file formats that NetworkX supports. Here we demonstrate the simplest one, the adjacency list file

format. The file `"graphs/petersen_graph.txt"` starts off with the lines `A B F` and `B C G`. The first node of each line is the starting node and an edge is drawn between it and ever subsequent node. Thus, these first two lines define 4 edges `(A, B)`, `(A, F)`, `(B, C)`, and `(C, G)`.

3 Breadth-First Search

Our first graph algorithm will be breadth-first search, often abbreviated to BFS. This is a general purpose algorithm for traversing an entire graph and doing some computation. At a high level, BFS starts with a designated *root* node and examines each node of the graph in turn, where all the nodes at distance d from the node are examined "simultaneously", and then all the nodes at distance $d+1$ are examined simultaneously, and so on.

Fig. 1 demonstrates what an example BFS traversal may look like on a graph where A is the designated root node. The traversal first examines the root node A, then it examines the distance 1 nodes from A which are B and C. Finally, it examines the distance 2 nodes from A which are E and F. Equivalently, the distance 2 nodes are the nodes distance 1 from B that have not already been traversed.

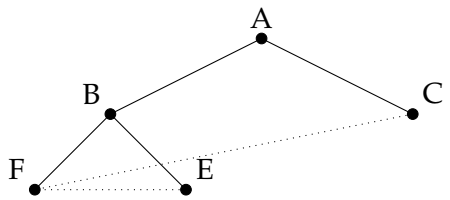

Figure 1: Example BFS Traversal.

An analogy for BFS is how a forest fire spreads. Imagine each node as a tree and nodes that are joined by an edge as trees that are next to each other. At the initial iteration of BFS, the root node is lit on fire. At the next step, each of the root's neighbors catch fire. Then the neighbors of the neighbors catch fire and so on. But once a tree has caught on fire, it burns out and will no longer catch on fire again. The ith iteration looks at the "ring of fire" a distance i from the root.

In order to implement BFS, we will maintain a `used = set()` of nodes that we have already visited and do not need to visit again. We will also have a list `todo = []` of nodes that are at the current distance i from the root. After examining each node in `todo` and performing some computation, we will look at all the neighbors of `todo`. If they have not yet been used, we will add them to the `next_todo` list. Once we are done iterating over `todo` and examining all the nodes at distance i, we will then replace `todo` with `next_todo` which gives us the nodes at distance $i+1$ that we need to examine. Note that BFS is *not* a recursive algorithm.

Listing 2 is a sample implementation of using BFS to compute the maximum distance from a given root node. We initialize `used` and `todo` to each only contain the root node to begin with. Lines 8 through 19 are the main BFS traversal. As described above, we iterate over each node at the current depth, iterate over each of their neighbors, and append them to `next_todo` and add them to `used` if the neighbor has not been used yet. At each iteration, we increment a `depth` counter and the maximum distance that we're looking for will be one less than our final `depth` value. Of note is that `<graph>.neighbors(<node>)` will return all the nodes connected to `<node>` by an edge. For a directed graph, this function will only contain those nodes connected by an

```python
import networkx as nx

def bfs(G, root):
  used = set([root])
  todo = [root]
  depth = 0

  while len(todo) > 0:
    next_todo = []
    depth += 1

    for node in todo:
      for neighbor in G.neighbors(node):
        if neighbor in used:
          continue

        used.add(neighbor)
        next_todo.append(neighbor)
    todo = next_todo

  return depth-1

G = nx.grid_graph(dim=(4,4))
root = (1,1)
print(f"Maximum distance from node {root} is {bfs(G, root)}")
```

Listing 2: BFS Implementation.

edge starting at <node>.

Do BFS exercises. The first one is about returning if a graph contains any cycles and the second one is about returning the number of connected components in a graph.

4 Depth-First Search

Another type of graph traversal is known as depth-first search (DFS). In depth-first search, rather than examining all nodes at distance i simultaneously, we pick the first node we haven't used yet and try to walk down as far a path along it as possible. Once we are done with this path, we will walk down the next path, and so on.

An analogy for DFS is to imagine the graph as a labyrinth. To traverse the entire labyrinth, we can start walking down any of its corridors. Imagine each unit square in the labyrinth is a node and two adjacent squares are joined by an edge as long as there is no wall blocking them. As we walk down the corridors, we will leave behind some breadcrumbs. At some point we might reach a dead end with no where to go or end up looping back around. Then we will follow our breadcrumbs back and find the most recent fork where there was an alternative path we can take that we haven't yet walked down, because there are no breadcrumbs. Eventually we'll get

to traverse the entire labyrinth this way.

DFS is commonly implemented recursively and used in conjunction with backtracking to explore a space of possibilities and find a solution. In the labyrinth analogy, we do some computations at each step as we walk along the labyrinth and if at some point we run into a contradiction or some other issue, we will collect all our breadcrumbs when we return to the nearest fork. The point of this is that we'll pretend we haven't walked along all the squares yet.

Listing 3 shows a DFS implementation with backtracking. This implementation is used to compute whether or not it's possible to color a graph `G` with `num_colors` colors. If it is possible, it will also augment each vertex with a `"color"` attribute at the end. A valid graph coloring is one where each vertex is assigned a color and no two vertices joined by an edge are colored the same. Fig. 2 shows a valid 3-coloring of a graph on the left and an invalid 3-coloring of a graph on the right.

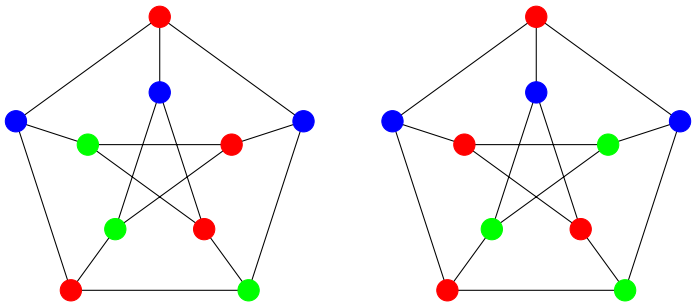

Figure 2: Valid and Invalid 3-coloring of the Petersen Graph.

In our DFS function, `v` is the vertex that we are currently looking at. In the labyrinth analogy, each call of `dfs()` corresponds to taking a step in the labyrinth and `v` is where in the labyrinth we are. Returning from the function is equivalent to backtracking along our breadcrumbs and `used` is a set of nodes that keeps track of where we have dropped breadcrumbs so far.

At each step in our DFS, we need to consider coloring the current node some integer color in the range `0, 1, ..., num_colors - 1`. This is the outermost for-loop starting at line 6. Our return variable `ret` will store whether or not we can finish the coloring of the graph.

On lines 10 through 15, we first check to see if we immediately run into a conflict by coloring the node `v` using the color `c`. We look at all of its neighbors, and if any of them have been colored `c`, then we reached a conflict and `ret` should be set to `False`. If there is a conflict already, there is no point in traversing the labyrinth further by recursively calling `dfs()`, so lines 14 and 15 check for a conflict and use `continue` to move on to the next iteration of the loop if there is one.

If our coloring is tentatively okay, then we move on further into the labyrinth on lines 17 through 22. We look at each of the neighbors of `v` in turn. If they are in `used`, then they've already been colored so we skip calling `dfs()` on them. Otherwise, line 20 calls `dfs()` using the neighbor `n` as the new current node. If it's not possible to color the graph when traversing down that node (i.e. a contradiction is forced at some point), then there's no hope for coloring the graph with the current color `c` assigned to `v`. So we set `ret` to `False` and don't recurse into the other neighbors.

Observe that a key property we use here is that the order in which we color nodes won't affect whether not the graph can be colored with `num_colors`. This means we can call `dfs()` on the neighbors of `v` and backtrack the moment one of them fails. In particular, coloring certain

```python
import networkx as nx

def dfs(G, v, used, num_colors=4):
  used.add(v)

  for c in range(num_colors):
    G.nodes[v]["color"] = c

    ret = True
    for n in G.neighbors(v):
      if G.nodes[n]["color"] == c:
        ret = False
        break
    if not ret:
      continue

    for n in G.neighbors(v):
      if n in used:
        continue
      if not dfs(G, n, used, num_colors):
        ret = False
        break

    if ret:
      break

  if not ret:
    used.remove(v)

  return ret

num_colors = 4
G = nx.read_adjlist("graphs/four_color_example.txt")
print(f"Can be {num_colors}-colored?", dfs(G, "A", set(), num_colors))
```

Listing 3: DFS Implementation For Graph Coloring.

neighbors first won't affect the coloring procedure when we try to color other neighbors.

Finally, lines 27 and 28 are part of the backtracking logic. If we failed to color the graph at this point, we pick up our breadcrumb be removing `v` from the `used` set. It's possible that after we backtrack and color some of the previous vertices differently, we will then be able to color the graph at the vertex `v`.