6.009: Fundamentals of Programming

Tutorial: Graphs and Paths
Undirected vs. directed graphs

Undirected:
- Vertices: V = {a, b, c, d}
- Edges: E = {{a, b}, {a, c}, {b, c}, {b, d}, {c, d}}

Directed:
- Vertices: V = {a, b, c}
- Edges: E = {(a, c), (b, c), (c, b), (b, a)}
Using matrix to represent graphs

- A → B, C, D
- B → A, C, E
- C → B, D
- D → C, F
- E → D
- F → C, E

Matrix representation:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Using dicts to represent graphs

```
graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['D'],
    'D': ['C'],
    'E': ['F'],
    'F': ['C']
}
```
def find_path(graph, start, end, path=[]):
    # extend current path being careful to create fresh copy since the incoming path is shared!
    path = path + [start]
    if start == end:
        # reached destination so return path
        return path
    # consider all neighbors of start as the next step in path. Use '.get' to handle the case where node as no neighbors.
    for node in graph.get(start, []):
        # avoid cycles by not including node more than once
        if node not in path:
            # recursively find path starting at neighbor
            newpath = find_path(graph, node, end, path)
            if newpath is not None:
                # yah! found a path.
                return newpath
    return None
find_path(graph,'A','D')

```python
find_path('A','D',[])
    find_path('B','D',['A'])
        find_path('C','D',['A','B'])
            find_path('D','D',['A','B','C'])
                return ['A','B','C','D']
            return ['A','B','C','D']
        return ['A','B','C','D']
    return ['A','B','C','D']
return ['A','B','C','D']

graph = {
    'A': ['B', 'C'],
    'B': ['C', 'D'],
    'C': ['D'],
    'D': ['C'],
    'E': ['F'],
    'F': ['C']
}
```

return ['A','B','C','D']
```python
find_path('A','E',[])  
    find_path('B','E',['A'])
        find_path('C','E',['A','B'])
            find_path('D','E',['A','B','C'])
                return None
            return None
        return None
    find_path('D','E',['A','B'])
        find_path('C','E',['A','B','D'])
            return None
        return None
    return None
find_path('C','E',['A'])
    find_path('D','E',['A','C'])
        return None
    return None
return None
```

```python
graph = { 'A': [ 'B', 'C' ],  
         'B': [ 'C', 'D' ],  
         'C': [ 'D' ],  
         'D': [ 'C' ],  
         'E': [ 'F' ],  
         'F': [ 'C' ]}
```

Follows each path to its end:

*depth-first search (dfs)*
find_all_paths(graph, start, end)

# return a list of all paths between start and end,
# where each path is a list of nodes
def find_all_paths(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        # reached destination, return a list containing this path
        return [path]
    # start a list of paths that we discover
    paths = []
    for node in graph.get(start, []):
        if node not in path:
            # continue path finding with neighbor.
            # accumulate list of paths that we find
            paths.extend(find_all_paths(graph, node, end, path))
    return paths

Extends list with contents of list arg
# find the shortest path (length = # of nodes in path)

def find_shortest_path(graph, start, end, path=[]):
    path = path + [start]
    if start == end:
        return path
    shortest = None
    for node in graph.get(start, []):
        if node not in path:
            newpath = find_shortest_path(graph, node, end, path)
            if newpath is not None:
                if shortest is None or len(newpath) < len(shortest):
                    shortest = newpath
    return shortest

Tests all paths!
Can take exponential time...
Graphs with weighted edges

find_shortest_path(graph,'A','C')
A→C  weight = 7
A→B→C  weight = 6
A→B→D→C  weight = 5
A→B→D→E→F→C  weight = 11

Do we have to try all paths?
find_least_weight_path(graph, start, end)

def find_least_weight_path(graph, start, end, path=[], weight=0):
    path = path + [start]
    if start == end:
        return path, weight
    shortest_path = None
    shortest_weight = None
    for node, w in graph.get(start, []):
        if node not in path:
            newpath, neww = find_least_weight_path(graph, node, end, path, weight+w)
            if newpath is not None:
                if shortest_path is None or neww < shortest_weight:
                    shortest_path = newpath
                    shortest_weight = neww
    return shortest_path, shortest_weight

Only works when weights ≥ 0
Breadth-first Search

• Key idea: consider all paths of length $N$ before considering paths of length $N + 1$

• Benefit: can find shortest path without having to visit all paths. Why?
find_path_bfs(graph, start, end)

# return path from start to end as a list of nodes
# return None if there is no path

def find_path_bfs(graph, start, end):
    # consider collections.deque for efficient pop(0) and append operations...
    paths = [[[start]]]  # list of active paths
    visited = {start}  # set of visited nodes

    while len(paths) > 0:
        # retrieve next active path from list   [for deque use paths.popleft()]
        path = paths.pop(0)
        # consider all edges from last node in path
        for n in graph.get(path[-1], []):
            # avoid cycles by not visiting a node twice
            if n in visited: continue
            # extend current path by one edge, mark node as visited
            npath = path + [n]
            visited.add(n)
            # are we at destination?
            if n == end: return npath
            # nope, so save npath for later exploration
            paths.append(npath)

    # no more active paths!
    return None

How to extend to weighted graphs?
find_path_bfs(graph, 'A', 'F')

List of paths at each iteration:

1. [[ 'A' ]]
2. [[ 'A', 'B' ], [ 'A', 'C' ]]
3. [[ 'A', 'C' ], [ 'A', 'B', 'D' ]]
4. [[ 'A', 'B', 'D' ]]
5. [[ 'A', 'B', 'D', 'E' ]]

Returns: ['A', 'B', 'D', 'E', 'F']
Depth-first Search

- Follow path until you get stuck
- Backtrack along breadcrumbs until reach unexplored neighbor
- Explore forward from nearest previous branch point
- Careful not to repeat a vertex

*Hint: it’s like exploring a maze on foot!*
find_path_dfs(graph, start, end)

• Earlier, we saw a recursive implementation for DFS
• Another common pattern is to use a “work queue” DFS implementation similar to our BFS:
  – Implementation is the same as BFS, EXCEPT:
    Pop from end of work queue rather than front of work queue!
  – Essentially, our work queue implementation of BFS or DFS manages “partial” solutions through the queue, rather than Python managing them through the stack of procedure calls (in the recursive procedure DFS implementation)
find_path_dfs(graph, start, end)

# return path from start to end as a list of nodes
# return None if there is no path

def find_path_dfs(graph, start, end):
    paths = [[[start]]]  # list of active paths
    visited = {start}  # set of visited nodes

    while len(paths) > 0:
        # retrieve next active path from list
        path = paths.pop(-1)
        # consider all edges from last node in path
        for n in graph.get(path[-1], []):
            # avoid cycles by not visiting a node twice
            if n in visited:
                continue
            # extend current path by one edge, mark node as visited
            npath = path + [n]
            visited.add(n)
            # are we at destination?
            if n == end:
                return npath
            # nope, so save npath for later exploration
            paths.append(npath)

    # no more active paths!
    return None

Work on most recently added path next!
find_path_dfs(graph, 'A', 'F')

Returns: ['A', 'B', 'C', 'D', 'E', 'F']

NOT THE SHORTEST PATH!