Algorithms in Systems Engineering
ISE 172

Lecture 18

Dr. Ted Ralphs

References for Today's Lecture

- Required reading
  - Sections 7.4-7.7
- References
  - CLRS Chapter 22
  - R. Sedgewick, *Algorithms in C++* (Third Edition), 1998.

Connectivity in Graphs

- An undirected graph is said to be *connected* if there is a path between any two vertices in the graph.
- A graph that is not connected consists of a set of *connected components* that are the maximal connected subgraphs.
- Given a graph, one of the most basic questions one can ask is whether vertices i and j are in the same component.
- In other words, is there a path from i to j?
- Such questions might arise in the design of a network or circuit.
- They may not be that easy to answer!
- One approach is to use a data structure for storing *disjoint sets*.

Finding a Simple Path

- We now revisit the question of whether there is a path connecting a given pair of vertices in a graph.
- Using the operations in the `Graph` class, we can answer this question directly using a recursive algorithm.
- We must pass in a vector of bools to track which nodes have been visited.

```
def SPath(G, v, w, visited = {})
    if v == w:
        return True
    visited[v] = True
    for n in v.get_neighbors():
      if not visited[n]:
         if SPath(G, n, w, visited):
             return True
    return False
```

Finding a Hamiltonian Path

- Now let's consider finding a path connecting a given pair of vertices that also visits every other vertex in between (called a *Hamiltonian path*).
- We can easily modify our previous algorithm to do this by passing an additional parameter `d` to track the path length.
- What is the change in running time?

Finding a Hamiltonian Path (code)

```
def HPath(G, v, w = None, d = None, visited = {})
    if d == None:
        d = G.get_node_num()
    if v == w:
        return d == 0
    if w == None:
        w = v
    visited[v] = True
    for n in v.get_neighbors():
      if not visited[n]:
        if SPath(G, n, w, d-1, visited):
            return True
    visited[v] = False
    return False
```

Hard Problems

- We have just seen an example of two very similar problem, one of which is hard and one of which is easy.
- In fact, there is no known algorithm for finding a Hamiltonian path that takes less than an exponential number of steps.
- This is our first example of a problem which is easy to state, but for which no known efficient algorithm exists.
- Many such problems arise in graph theory and it's difficult to tell which ones are hard and which are easy.
- Consider the problem of finding an *Euler path*, which is a path between a pair of vertices that includes every *edge* exactly once.
- Does this sound like a hard problem?

Depth-first Search

- With an algorithm called *depth-first search*, we can compute the number of connected components of a graph, and label each node with a component number.
- This allows us to answer the question of whether two nodes are in the same connected component in constant time.
- Here is the basic depth-first search function for finding all the nodes in the same component as a given starting node `v`.

```
def DFS(G, v, compNum, comps):
    comps[v] = compNum
    for n in v.get_neighbors():
        if v not in comps:
            DFS(G, t, compNum, comp)
```

- This is almost exactly the same algorithm as depth-first search of a tree.
- What is the difference?
- What is the running time of this algorithm?

Ordering Nodes by Depth-first Search

- The process of performing DFS produces two natural orderings of the nodes of a graph, by *discovery order* and by *finishing order*.
  - *Discovery order* is the the order in which nodes are encountered (the order in which recursive calls to search from each node are made).
  - *Finishing order* is the the order in which we *return from* the recursive call made from each node.
- Note that these are similar to the pre- and post-order we discussed with trees.
- In fact, discovery order is exactly the same as pre-order with respect to the search tree (and similarly for post-order and finishing order).

Component Labeling

- To label all vertices with a component number, we simply call DFS iteratively on each vertex that has not been labeled.

```
def Label(G):
    numComps = 0
    comps = {}
    for s in G.get_node_list():
        if not comps[v]:
            numComps += 1
            DFS(G, v, compNum, comp)
    return numComps
```

General Graph Search Algorithms

- The goal of component labeling is simply to discover all nodes of the graph in any order.
- Using a simple recursive algorithm works well for this purpose.
- What if we want to search the graph with some sort of purpose in mind?
- Graph search from a root node `r` is a general procedure in which we perform the following loop.
  - Add `r` to the list of "unprocessed" vertices (`Q`).
  - Repeat until no vertices remain on the list `Q`.
    - Choose a vertex `v` from `Q`.
    - (Optionally) process `v`.
    - For each arc `(v, w)` incident to `v`,
      - (Optionally) process (v, w) and
      - Add `w` to the list of unprocessed vertices.
- Depending on the order in which we process nodes and what we do at each step, we get a variety of different algorithms.

General Graph Search Algorithm

```
def search(self, root, q = Stack()):
    if isinstance(q, Queue):
        addToQ = q.enqueue
        removeFromQ = q.dequeue
    elif isinstance(q, Stack):
        addToQ = q.push
        removeFromQ = q.pop
    visited = {}
    visited [root] = True
    addToQ(root)
    while not q.isEmpty():
        current = removeFromQ()
        for n in current.get_neighbors():
            if not n in visited:
                visited[n] = True
                addToQ(n)
```

Search Trees and Forests

- Consider searching a connected undirected graph $G = (V, E)$.
- The process of searching G can be captured by constructing a tree T called the *search tree*.
- The search tree is a subgraph which represents the paths that were followed in exploring the graph.
- T is constructed as the search evolves by adding an edge connecting the vertex currently being processed to any vertex not yet processed.
- This graph must be connected and acyclic, and is hence a tree.
- We can view it as a rooted tree by taking the root to be the start vertex.
- In graphs with multiple components, we can similarly obtain *search forests*.
- To keep track of the tree as it is constructed, we use a *predecessor dictionary*.

Breadth-first versus Depth-first Search

- Using a `Queue` to hold the list of unprocessed nodes results in visiting the node in *breadth-first* search order (first in, first out).
- Using a `Stack` results in visiting the nodes in *depth-first* search order (last in, first out) (well, sort of...).

```
def bfs(self, root, display = None):
    self.search(root, display, Queue())

def dfs(self, root, display = None):
    self.search(root, display, Stack())
```

- We have seen that that depth-first search can be implemented either recursively or iteratively using a stack.
- This is an illustration of the general principle we talked about earlier in the course.
- Note, however, that recurive depth-first search is *slightly* different from general graph search with a stack (how?).

Directed Graphs

- Up until now, we've concentrated on undirected graphs.
- In a directed graph, or *digraph*, the connections between the vertices are ordered pairs called *arcs*.
- The set of vertices is typically denoted N and the set of arcs is denoted A with $m = |A| \leq n(n-1)$.
- A directed graph $G = (N, A)$ is then composed of a set of vertices N and a set of arcs $A \subseteq V \times V$.
- If $a = (i, j) \in A$, then
  - i is called the *tail* of a and j is called the *head* of a,
  - a is said to be *incident from* i and *incident to* j, and
  - i and j are said to be *adjacent* vertices.
- For a given digraph, there is an *underlying undirected graph* obtained by ignoring the directions of the arcs (and eliminating parallel edges).

More Terminology

- Let $G = (N, A)$ be a digraph.
- A *subgraph* of G is a digraph composed of an arc set $A' \subseteq A$ along with all incident vertices.
- A subset V' of V, along with all incident arcs is called an *induced subgraph*.
- A *directed path* in G is a sequence $v_0, \ldots, v_p$ of vertices such that for each $i \in 0, \ldots, p-1$, $(v_i, v_{i+1}) \in A$.
- A directed path is *simple* if no vertex occurs more than once in the sequence.
- A *directed cycle* is a directed path that is simple, except that the first and last vertices are the same.
- A *directed tour* is a directed cycle that includes all the vertices.

Data Structures for Digraphs

- Data structures for digraphs are similar to those for undirected graphs.
- As before, there are two basic choices
  - Adjacency matrix
  - Adjacency lists
- These are implemented in similar fashion, except that
  - In the case of an adjacency matrix, the matrix is no longer symmetric.
  - In the case of an adjacency list, each arc appears only on the adjacency list of its tail vertex.

Connectivity in Digraphs

- A digraph is *connected* if the underlying undirected graph is connected.
- A digraph is *strongly connected* if for each pair of vertices i and j, there is a directed path from i to j and a directed path from j to i.
- A digraph that is not strongly connected consists of a set of *strongly connected components* that are the maximal strongly connected subgraphs.
- Given a digraph, one of the most basic questions one can ask is whether there is a path from vertex i to vertex j.
- We can answer this question as we did before using a modified version of graph search.
- In graph search for directed graphs, we only examine the arcs that are incident from the vertex being processed.
- Performing graph search starting at vertex r results in the processing of all the vertices to which there is a path from r.

Graph Search for Directed Graphs

- When the graph is directed, we have to pay attention to the directions of the arcs.
- To do so, we simply replace `get_neighbors` with `get_out_neighbors`.
- With directed graphs, some of the questions we have asked have to be re-formulated.
  - The question "is there a path from x to y" is different from "is there a path from y to x"
  - The definitions of connected components have to be generalized.
- We cannot use graph search directly to find the (strongly) connected components.
- As before, this graph search process results in construction of a directed search tree in which there is a path from r to each other vertex.
- Such a directed tree is said to be *directed away from* r.

Graph Search for Digraphs

- (Directed) graph search from vertex r:

```
def search(self, root, q = Stack()):
    if isinstance(q, Queue):
        addToQ = q.enqueue
        removeFromQ = q.dequeue
    elif isinstance(q, Stack):
        addToQ = q.push
        removeFromQ = q.pop
    visited = {}
    visited [root] = True
    addToQ(root)
    while not q.isEmpty():
        current = removeFromQ()
        for n in current.get_out_neighbors():
            if not n in visited:
                visited[n] = True
                addToQ(n)
```

Topological Ordering

- In a directed graph, the arcs can be thought of as representing *precedence constraints*.
- In other words, an arc (i, j) represents the constraint that node i must come before node j.
- Given a graph $G = (N, A)$ with the nodes labeled with distinct numbers 1 through n, let $order(i)$ be the label of node i.
- Then, this labeling is a *topological ordering* of the nodes if for every arc $(i, j) \in A$, $order(i) < order(j)$.
- Can all graphs be topologically ordered?

Directed Acyclic Graphs

- We have just seen that *precedence graphs* should be both directed and acyclic.
- Given a directed acyclic graph (DAG), we would like to be able to determine an ordering of vertices that is a *topological sort*.
- Given a DAG, DFS can be used to perform a topological sort.
- A topological sort can be easily obtained by examining the finishing times of the nodes (how?).