Computational Optimization
ISE 407

Lecture 19

Dr. Ted Ralphs

Search Algorithms

- *Search algorithms* are fundamental techniques applied to solve a wide range of optimization problems.
- Search algorithms attempt to find all the nodes in a network satisfying a particular property.
- Examples
  - Find nodes that are reachable by directed paths from a source node.
  - Find nodes that can reach a specific node along directed paths
  - Identify the connected components of a network
  - Identify directed cycles in network
- Let us consider undirected graphs to start.
- We first consider an algorithm for finding a simple path between two given nodes in a graph.

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 of whether there is a path from i to j 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 = None)
    if visited == None:
       visited = {}
    if v == w:
       return [v]
    visited[v] = True
    for n in v.get_neighbors():
       if n not in visited:
          path = SPath(G, n, w, visited)
             if path != None
                path.append(n)
                return path
    return None
```

Connectivity in Graphs

- The algorithm on the last slide allows us to answer the question of whether two nodes are *connected* in a undirected graph.
- We can also ask more generally whether an entire graph is "connected."
- 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.
- This is the same as asking whether there a path from i to j and this question can thus be answered by the algorithm on the previous slide.
- More generally, we might want to construct a partition of the set of all vertices into sets of vertices that are in the same component.
- We can generalize the path-finding algorithm to solve this more general problem.

Finding All Nodes in a Single Component

- The set of all nodes in the same component as a given node v are the nodes connected to v by a path in the graph.
- How easy is it to determine all of the nodes in the same component as v?
- This can be done by an extension of the recursive depth-first search algorithm we saw earlier from trees to more general graphs.
- Starting at v, we recursively search from each neighbor of v, avoiding vertices that have already been "discovered."

```
def DFSRecursion(G, v, component_label = '0'):
    G.set_node_attr(v, 'component', component_label)
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'component') == None
            DFSRecursion(G, n, component_label)
    return
```

Search Tree

- The algorithm we have just seen implicitly constructs a *search tree*.
- The search tree is an acyclic subgraph of the original graph connecting all nodes in the same component as v.
- A node is discovered
- When a neighbor n of v is *discovered*, we record v as its *predecesor*.
- The set of edges consisting of each node and its predecessor forms a tree rooted at v.
  - We call the edges in the tree *tree edges*.
  - The remaining edges connect a vertex with an ancestor in the tree that is not its parent and are called *back edges*.
- Why must every edge be either a tree edge or a back edge?

Constructing the Search Tree

```
def DFS(G, v, comp_label = '0')
    for n in G.get_node_list():
        G.set_node_attr(n, 'component', None)
    DFSRecursion(G, v, {}, comp_label)
    return

def DFSRecursion(G, v, pred, comp_label, dtime = 0, ftime = 0):
    G.set_node_attr(v, 'component', comp_label)
    for n in G.get_neighbors(v):
        if n in pred:
            pred[n] = v
            G.set_node_attr(n, 'discover', dtime)
            dtime += 1
            DFSRecursion(G, n, pred, comp_label, dtime, ftime)
            G.set_node_attr(n, 'finish', ftime)
            ftime += 1
    return
```

Node Ordering

- As in depth-first search for trees, the nodes can be ordered in two ways.
  - Preorder: The order in which the nodes are first *discovered* (discovery time).
  - Postorder: The order in which the nodes *finished* (the recursive calls on all neighbors return).
- These orders will be referred to in various algorithms we'll study.

Labeling All Components

- To label all components, we loop through all the nodes in the graph and start labeling the component of any node we find that has not already been labeled.

```
def label_components(self):
    comp_label = 0
    for n in G.get_node_list():
        G.set_node_attr(n, 'component', None)
    for n in G.get_node_list():
        if G.get_node_attr(n, 'component') is None:
            DFS(G, n, comp_label)
        comp_label += 1
    return
```

Complexity of Depth-first Search for Single Component

- How do we analyze a DFS algorithm?
- Start with complexity of finding a single component.
  - How many recursive calls are there?
  - What is the total running time of all recursive calls?
- We cannot answer these questions in general without knowing the data structure being used to store the graph.
- How does the graph data structure affect the running time?
  - Adjacency matrix
  - Adjacency list

Component Labeling in General Graphs

- The depth-first search algorithm for undirected graphs can be easily generalized to directed graphs.
- However, the notion of a "component" is different.
- In a directed graph, we may have a directed path from i to j, but not from j to i.
- a graph is called *strongly connected* if there is a path from i to j and from j to i for all pairs (i, j) of nodes.
- The strongly connected components of a directed graph are the maximal strongly connected subgraphs.
- We may also consider the components of the *underlying undirected graph* in some cases.

Depth-first Search in Directed Graphs

- DFS in a directed graph is very similar to DFS in an undirected graph.
- The main difference is that each arc is only encountered once during the search.
- Also, this simple algorithm cannot be used to find all nodes in the same *strongly connected* component as a given node v.

Depth-first Search in Directed Graphs

```
def DFS(G, v, comp_label = '0')
    for n in G.get_node_list():
        G.set_node_attr(n, 'component', None)
    DFSRecursion(G, v, {}, comp_label)
    return

def DFSRecursion(G, v, pred, comp_label, dtime = 0, ftime = 0):
    G.set_node_attr(v, 'component', comp_label)
    for n in G.get_out_neighbors(v):
        if n in pred:
            pred[n] = v
            G.set_node_attr(n, 'discover', dtime)
            dtime += 1
            DFSRecursion(G, n, pred, comp_label, dtime, ftime)
            G.set_node_attr(n, 'finish', ftime)
            ftime += 1
    return
```

Node Order and Arc Type

- As with undirected graphs, DFS in directed graphs produces a *search tree* that is *directed out* from the initial node (an *out tree*).
- At the time a node n is *discovered*, we record v as its *predecesor*.
- The finishing time is the time at which the recursive calls on all neighbors complete, as in the undirected case.
- As before, we refer to the preorder as the ordering of nodes by discovery time and the postorder as the ordering of nodes by their finishing time.
- The set of arcs consisting of each node and its predecessor forms the search tree.
- The arcs not in the tree can be either
  - Back arcs: Those connecting a node to an ancestor.
  - Down arcs: Those connecting a node to a descendant.
  - Cross arcs: All other arcs.
- Back arcs connect to an ancestor with a later finishing time.
- Cross arc leads to a node with an earlier discovery time.
- Down arcs lead to a node with a later discovery time.

Problems Solvable With DFS (Undirected Graphs)

- Cycle Detection: The discovery of a back edge indicates the existence of a cycle.
- Simple Path
- Connectivity
- Component Labeling
- Spanning Forest
- Two-colorability, bipartiteness, odd cycle

General Graph Search

- The order in which the nodes are discovered in depth-first search can differ, depending on the order in which the neighbors are listed.
- The algorithm is constrained, however, to construct a maximal path from the root node before backtracking.
- This is why the algorithm is called "depth-first" search—it constructs a search tree with maximal depth.
- This method of exploring the nodes of the graph is efficient and convenient, since it allows a simple recursive implementation.
- However, in many cases, we want to search the graph in a different "order" to achieve a different goal.
- This leads to the notion of a *general graph search*.

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
    pred = {}
    addToQ(root)
    while not q.isEmpty():
        current = removeFromQ()
        self.process_node(current, q)
        for n in current.get_neighbors():
            self.process_edge(current, n)
            if n not in q:
                pred[n] = current
                addToQ(n)
```

General Search Algorithm

- The algorithm is a template for a whole class of algorithms.
  - If Q is a *queue*, for example, we are doing *breadth-first search*.
  - If Q is a *stack*, we are doing a modified version of *breadth-first search*.
  - In other cases, we will want to maintain Q as a priority queue.
- What problem does breadth-first search of a graph solve?

Complexity of Search Algorithm

- The search proceeds differently depending on which element `v` is selected from `q` in each iteration.
- `q` must be ordered in some way by storing it in an appropriate data structure.
  - If `q` is a *queue*, elements are inserted at one end and removed from the other and we get FIFO ordering.
  - If `q` is a *stack*, elements are inserted and deleted from the the same end and we get LIFO ordering.
- The efficiency of the algorithm can be affected by
  - the data structure used to maintain `q`,
  - what additional steps are required in `process_node`, and
  - what additional steps are required in `process_edge`.

Aside: 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?

Aside: 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
```

Aside: 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?