Computational Optimization
ISE 407

Lecture 17

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 will first generalize the algorithm from last time for finding a simple path in a graph.
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.

```python
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
```
Labeling Components

• The set of all nodes connected to a given node by a path is called a \textit{component}.

• How easy is it to determine all of the nodes in the same component as a given node?

```python
def DFSRecursion(G, v, pred, component_num = 0):
    G.set_node_attr(v, 'component', component_num)
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'component') == None:
            pred[n] = v
            DFS(G, n, pred, component_num)
    return

def DFS(G, v, component_num = 0):
    for n in G.get_node_list():
        G.set_node_attr(n, 'component', None)
    DFSRecursion(G, v, [], component_num)
    return
```
Depth-first Search for General Graphs

• The algorithm we have just seen is a generalization of the depth-first search method we saw earlier for trees.

• We will see why it is called this shortly.

• Associated with the search is a search tree that can be used to visualize the algorithm.

• At the time a node \( n \) is discovered, we can record \( v \) as its predecessor.

• 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?
Complexity of Depth-first Search

- How do we analyze a DFS algorithm?
- How many recursive calls are there?
- How does the graph data structure affect the running time?
  - Adjacency matrix
  - Adjacency list
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.

```python
def label_component(G):
    component_num = 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, component_num)
            component_num += 1
    return
```

- What is the complexity of this algorithm?
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, note that the notion of a component is different here.

```python
def DFSRecursion(G, v, pred):
    G.set_node_attr(v, 'color', 'green')
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'color') == 'red':
            pred[n] = v
            DFS(G, n, pred)
    return

def DFS(G, v):
    for n in G.get_node_list():
        G.set_node_attr(n, 'color', 'red')
        DFSRecursion(G, v, [])
    return
```

- What nodes will this search reach?
Depth-first Search in Directed Graphs

• 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 predecessor.

• The set of arcs consisting of each node and its predecessor forms a tree rooted at $v$.

  – We call the arcs in the tree tree arcs.
  – The remaining arcs can be either
    * Back arcs: Those connecting a vertex to an ancestor
    * Down arcs: Those connecting a vertex to a descendant
    * Cross arcs: Those connecting a vertex to a vertex that is neither a descendant nor an ancestor.
Node Order and Arc Type

• Also as with undirected graphs, we can order the nodes in two different ways: postorder and preorder.

• As before, we refer to the preorder number of a node as its discovery time and the postorder number as its finishing time.

• We can identify the type of an arc as follows.

  – It is a back arc if it leads to a node with a later finishing time.
  – Otherwise, it is a cross arc if it leads to a node with an earlier discovery time and a down arc if it leads 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

• Depth-first search is so called because the node selected in each step is a neighbor of the node that is farthest from the root (in the tree).

• This is convenient because it allows a simple recursive implementation.

• Could we search the graph in a different “order”? 
General Graph Search Algorithm

def search(self, root, q = Stack()):
    if display == None:
        display = self.display_mode
    if isinstance(q, Queue):
        addToQ = q.enqueue
        removeFromQ = q.dequeue
    elif isinstance(q, Stack):
        addToQ = q.push
        removeFromQ = q.pop
    visited = {}
    addToQ(root)
    while not q.isEmpty():
        current = removeFromQ()
        self.process_node(current, q)
        for n in current.get_neighbors():
            if not n in visited:
                visited[n] = True
                self.process_edge(current, n)
                addToQ(n)
General Search Algorithm

- The algorithm is a template for a whole class of algorithms.
  - If $Q$ is a stack (LIFO), we are doing depth-first search, as before.
  - If $Q$ is a queue (FIFO), we are doing 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 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?