References for Today’s Lecture

• Required reading
  – Sections 18.1–18.6, 19.2, 19.6, 19.8

• References
  – AMO Section 3.4
  – CLRS Chapter 22
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 again consider undirected graphs to start.

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

• The set of all nodes connected to a given node by a path is called a *component*.

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

```python
def DFS(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:
            DFS(G, n, pred, component_num)
    return
```
Depth-first Search

• The algorithm we have just seen is known as \textit{depth-first search}.

• We will see why it is called this shortly.

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

• At the time a node \( n \) is \textit{discovered}, we can record \( v \) as its \textit{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 \textit{tree edges}.
  – The remaining edges connect a vertex with an ancestor in the tree that is not its parent and are called \textit{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

- The nodes can be ordered in two ways during the depth-first search.
  - 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 DFS(G, v):
    G.set_node_attr(v, 'color', 'green')
    for n in G.get_neighbors(v):
        if G.get_node_attr(n, 'color') == 'red':
            DFS(G, n, pred)
    return

for n in G.get_node_list():
    G.set_node_attr(n, 'color', 'red')
    DFS(G, v)
```

- What nodes will be colored green after DFS is called?
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**
Directed Acyclic Graphs

- A *directed acyclic graph* (DAG) is a directed graph containing no directed cycles.
- DAGs can be interpreted as specifying precedence relations or a (partial) order on the nodes.
- Directed cycles can be detected in directed graphs by using DFS.
- A graph is a DAG if and only if it contains no back arc.
Topological Ordering

• In a DAG, we interpret the arcs 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 \(\text{order}(i)\) be the label of node \(i\).

• Then, this labeling is a *topological ordering* of the nodes if for every arc \((i, j) \in A\), \(\text{order}(i) < \text{order}(j)\).

• Can all graphs be topologically ordered?
**Topological Ordering**

The following algorithm will detect presence of a directed cycle or produce a topological ordering of the nodes.

**Input:** Directed acyclic graph $G = (N, A)$

**Output:** The array $order$ is a topological ordering of $N$.

```
count ← 1
while \{v ∈ N : I(v) = 0\} ≠ ∅ do
    let v be any vertex with I(v) = 0
    order[v] ← count
    count ← count + 1
    delete v and all outgoing arcs from $G$
end while
if $N = ∅$ then
    return success
else
    report failure
end if
```

Can this be implemented efficiently?
Topological Ordering Algorithm

• Correctness of algorithm
  1. If $G$ has a cycle...
  2. If $G$ is acyclic...

• Running time of the algorithm
Topological Ordering with DFS

- How might we topologically order a graph using DFS?
Connectivity in Directed Graphs

- Determining connectivity in directed graphs is more involved than in undirected graphs.

- Although it is not obvious how to do it, we can find the strongly connected components of a graph in linear time.
  - Use DFS to compute the finishing time for each vertex
  - Compute the reverse (transpose) of the graph.
  - Do DFS on the transpose, but explore each vertex in decreasing order of finish time.

- This can be implemented very efficiently.