# Graphs and Network Flows
# IE411

## Quiz 1 Review

Dr. Ted Ralphs

# Graphs and Data structures

- The essence of a graph is a ground set of elements $N$ and a set $A$ of pairs of those elements (ordered or unordered) that represent relationships among the elements
- Data structures
  - Node-Arc Incidence Matrix
  - Node-Node Adjacency Matrix
  - Adjacency List
  - Forward Star (Reverse Star)

# Computational Complexity: What is the objective?

- Complexity analysis is aimed at answering two types of questions.
  - How hard is a given problem?
  - How efficient is a given algorithm for a given problem?
- The usual measure of efficiency is *running time*, usually defined as the number of elementary operations required (more on this later).
- The running time will differ by instance, algorithm, and computing platform.
- How should we measure the performance so that we can select the “best” algorithm from among several?

# What do We Measure?

Three methods of analysis:

- Empirical analysis
  - Try to determine how algorithms behave in practice
- Average-case analysis
  - Try to determine the expected number of steps an algorithm will take analytically.
- Worst-case analysis
  - Provide an upper bound on the number of steps an algorithm can take on *any* instance.

# Asymptotic Analysis

- So far, we have determined that our measure of running time will be a function of instance size (a positive integer).
- Determining the exact function is still problematic at best.
- We will only really be interested in approximately how quickly the function grows "in the limit".
- To determine this, we will use *asymptotic analysis*.
- Order relations

$$f(n) \in O(g(n)) \Leftrightarrow \exists c \in \mathbb{R}_+, n_0 \in \mathbb{Z}_+ \text{ s.t. } f(n) \leq cg(n) \; \forall n \geq n_0.$$

- In this case, we say $f$ *is order* $g$ or $f$ *is 'big O' of* $g$.
- Using this relation, we can divide functions into classes that are all *of the same order*.

# Running Time and Complexity

- Running time is a measure of the efficiency of an algorithm.
- Computational complexity is a measure of the difficulty of a problem.
- The computational complexity of a problem is the running time of the best possible algorithm.
- In most cases, we cannot prove that the best known algorithm is the also the best possible algorithm.
- We can therefore only provide an upper bound on the computational complexity in most cases.
- That is why complexity is usually expressed using “big O” notation.
- A case in which we know the exact complexity is comparison-based sorting, but this is unusual.

# 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

# Basic Search Algorithm

This is the basic search algorithm.

**Input:** Graph $G = (N, A)$

$Q \leftarrow \{s\}$
**while** $Q \neq \emptyset$ **do**
    let $v$ be any element of $Q$
    remove $v$ from $Q$
    mark $v$
    **for** $v' \in A(v)$ **do**
        **if** $v'$ is not marked **then**
            $Q \leftarrow Q \cup \{v'\}$
        **end if**
    **end for**
**end while**

# 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?

# 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` $\leftarrow 1$
**while** $\{v \in N : I(v) = 0\} \neq \emptyset$ **do**
  let $v$ be any vertex with $I(v) = 0$
  `order`$[v] \leftarrow$ `count`
  `count` $\leftarrow$ `count + 1`
  delete $v$ and all outgoing arcs from $G$
**end while**
**if** $V = \emptyset$ **then**
  return `success`
**else**
  report `failure`
**end if**

# Shortest Path Problem

- The shortest path problem underlies virutally all network flow problems.
- Variants
  - Single Source
    - Acyclic
    - Non-negative arc lengths
    - Arbitrary arc lengths
  - All Pairs

**Definition 1.** *Given a directed network* $G = (N, A)$ *with an arc length* $c_{ij}$ *associated with each arc* $(i, j) \in A$ *and a distinguished node* $s$*, the* shortest path problem *is to determine a shortest length directed path from node* $s$ *to every node* $i \in N - \{s\}$*.*

# Shortest Path Algorithms

- Label Setting (Chapter 4)
  - one label becomes permanent during each iteration
  - acyclic with arbitrary arc lengths OR non-negative arc lengths
- Label Correcting (Chapter 5)
  - all labels are temporary until last iteration
  - more general graphs including negative arc lengths
- Both are iterative; they differ in label update procedure and convergence procedure.

# Optimality Conditions

**Theorem 1. [5.1]** *For every node $j \in N$, let $d(j)$ denote the length of some directed path from the source node to node $j$. Then, the numbers $d(j)$ represent the shortest path distances* **if and only if** *they satisfy the following for all $(i, j) \in A$:*

$$d(j) \leq d(i) + c_{ij}.$$

**Theorem 2.** *For every pair of nodes $[i, j] \in N \times N$, let $d[i, j]$ represent the length of some directed path from node $i$ to node $j$ satisfying $d[i, i] = 0 \ \forall i \in N$ and $d[i, j] \leq c_{ij} \ \forall (i, j) \in A$. These distances represent* shortest path distances *if and only if they satisfy*

$$d[i, j] \leq d[i, k] + d[k, j] \ \forall i, j, k \in N.$$

# Dijkstra's Algorithm

**Input:** An acyclic network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}_+^A$
**Output:** $d(i)$ is the length of a shortest path from node $s$ to node $i$ and pred(i) is the immediate predecessor of $i$ in an associated shortest paths tree.

$S := \emptyset$
$\bar{S} := N$
$d(i) \leftarrow \infty \forall i \in N$
$d(s) \leftarrow 0$ and $pred(s) \leftarrow 0$
**while** $|S| < n$ **do**
  let $i \in \bar{S}$ be the node for which $d(i) = min\{d(j) : j \in \bar{S}\}$
  $S \leftarrow S \cup \{i\}$
  $\bar{S} \leftarrow \bar{S} \setminus \{i\}$
  **for** $(i, j) \in A(i)$ **do**
    **if** $d(j) > d(i) + c_{ij}$ **then**
      $d(j) \leftarrow d(i) + c_{ij}$ and $pred(j) \leftarrow i$
    **end if**
  **end for**
**end while**

# General Label-Correcting Algorithms

Maintain a distance label $d(j)$ for all nodes $j \in N$

- If $d(j)$ is infinite, the algorithm has not found a path joining the source node to node $j$.
- If $d(j)$ is finite, it is the distance from the source node to that node along $some$ path (upper bound).
- No label is permanent until the algorithm terminates.

# All-Pairs Label-Correcting Algorithm

**Input:** A network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}^A$
**Output:** $d[i, j]$ is the length of a shortest path from node $i$ to node $j$ for pairs $i$ and $j$.

$d[i, j] \leftarrow \infty$ for all $[i, j] \in N \times N$
$d[i, j] \leftarrow 0$ for all $i \in N$
**for** $(i, j) \in A$ **do**
  $d[i, j] \leftarrow c_{ij}$
  **while** $\exists(i, j, k)$ satisfying $d[i, j] > d[i, k] + d[k, j]$ **do**
    $d[i, j] := d[i, k] + d[k, j]$
  **end while**
**end for**

# Floyd-Warshall Algorithm

**Input:** A network $G = (N, A)$ and a vector of arc lengths $c \in \mathbb{Z}^A$

**Output:** $d[i, j]$ is the length of a shortest path from node $i$ to node $j$ for pairs $i$ and $j$.

**for** $(i, j) \in N \times N$ **do**
  $d[i, j] \leftarrow \infty$ and $pred[i, j] \leftarrow 0$
**end for**
**for** $i \in N$ **do**
  $d[i, i] \leftarrow 0$
**end for**
**for** $(i, j) \in A$ **do**
  $d[i, j] \leftarrow c_{ij}$ and $pred[i, j] := i$
**end for**
**for** $k = 1$ to $n$ **do**
  **for** $[i, j] \in N \times N$ **do**
    **if** $d[i, j] > d[i, k] + d[k, j]$ **then**
      $d[i, j] \leftarrow d[i, k] + d[k, j]$
      $pred[i, j] \leftarrow pred[k, j]$
    **end if**
  **end for**
**end for**

# Maximum Flow Problem

Given a network $G = (N, A)$ with a non-negative capacity $u_{ij}$ associated with each arc $(i, j) \in A$ and two nodes $s$ and $t$, find the maximum flow from $s$ to $t$ that satisfies the arc capacities.

$$
\begin{aligned}
\text{Maximize} \quad & v && (1)\\
\text{subject to} \quad & \sum_{j:(s,j)\in A} x_{sj} - \sum_{j:(j,s)\in A} x_{js} = v && (2)\\
& \sum_{j:(i,j)\in A} x_{ij} - \sum_{j:(j,i)\in A} x_{ji} = 0 \quad \forall i \in N \setminus \{s,t\} && (3)\\
& \sum_{j:(t,j)\in A} x_{tj} - \sum_{j:(j,t)\in A} x_{jt} = -v && (4)\\
& x_{ij} \le u_{ij} \quad \forall (i,j) \in A && (5)\\
& x_{ij} \ge 0 \quad \forall (i,j) \in A && (6)
\end{aligned}
$$

# Residual Network

- Suppose that an arc $(i, j)$ with capacity $u_{ij}$ carries $x_{ij}$ units of flow.
- Then, we can send up to $u_{ij} - x_{ij}$ additional units of flow.
- We can also send up to $x_{ij}$ units of flow backwards, canceling the existing flow and decreasing the flow cost.
- The *residual network* $G(x^0)$ is defined with respect to a given flow $x^0$ and consists of arcs with positive residual capacity.
- Note that if for some pair of nodes $i$ and $j$, $G$ already contains both $(i, j)$ and $(j, i)$, the residual network may contain parallel arcs with different residual capacities.

# Cuts

- A *cut* is a partition of the node set $N$ into two parts $S$ and $\bar{S} = N \setminus S$.
- An $s - t$ *cut* is defined with respect to two distinguished nodes $s$ and $t$ and is a cut $[S, \bar{S}]$ such that $s \in S$ and $t \in \bar{S}$.
- A *forward arc* with respect to a cut is an arc $(i, j)$ with $i \in S$ and $j \in \bar{S}$.
- A *backward arc* with respect to a cut is an arc $(i, j)$ with $i \in \bar{S}$ and $j \in S$.

**Property 1. [6.1]** *The value of any feasible flow is less than or equal to the capacity of any cut in the network.*

# Generic Augmenting Path Algorithm

- An *augmenting path* is a directed path from the source to the sink in the *residual* network.
- The *residual capacity* of an augmenting path is the minimum residual capacity of any arc in the path, which we denote by $\delta$.
  - By definition, $\delta > 0$.
  - When the network contains an augmenting path, we can send additional flow from the source to the sink.

**Theorem 3. [6.4]** *A flow $x^*$ is a maximum flow if and only if the residual network $G(x^*)$ contains no augmenting path.*

# Generic Augmenting Path Algorithm

**Input:** A network $G = (N, A)$ and a vector of capacities $u \in \mathbb{Z}^A$
**Output:** $x$ represents the maximum flow from node $s$ to node $t$

$x \leftarrow 0$
**while** $G(x)$ contains a directed path from $s$ to $t$ **do**
  identify an augmenting path $P$ from $s$ to $t$
  $\delta \leftarrow \min\{r_{ij} : (i,j) \in P\}$
  augment the flow along $P$ by $\delta$ units and update $G(x)$ accordingly.
**end while**

# Identifying an Augmenting Path

- Use search technique to find a directed path in $G(x)$ from $s$ to $t$
  - At any step, partition nodes into *labeled* and *unlabeled*
  - Iteratively select a labeled node and scan its arc adjacency list in $G(x)$ to reach and label additional nodes
  - When sink becomes labeled, augment flow, erase labels and repeat
  - Terminate when all labeled nodes have been scanned and sink remains unlabeled

# Distance Labels

A *distance function* $d : N \rightarrow Z^+ \cup \{0\}$ with respect to the residual capacity $r_{ij}$ is *valid* with respect to a flow $x$ if it satisfies:

$$\begin{aligned} d(t) &= 0 \\ d(i) &\leq d(j) + 1 \ \forall (i, j) \in G(x) \end{aligned}$$

**Property 2. [7.1]** *If the distance labels are valid, $d(i)$ is a lower bound on the length of the shortest (directed) path from node $i$ to node $t$ in the residual network.*

**Property 3. [7.2]** *If $d(s) \geq n$, then the residual network contains no directed path from $s$ to $t$.*

Distance labels are *exact* if $d(i)$ equals the length of the shortest path from $i$ to $t$ in $G(x)$ for all $i \in N$.

# Shortest Augmenting Path Algorithm

- Always augments flow along a shortest path from $s$ to $t$ in $G(x)$
- Proceeds by augmenting flows along admissible paths
- Constructs an admissible path incrementally – adding one arc at a time
- Maintains a partial admissible path and iteratively performs *advance* or *retreat* operations from current node
- Repeats operations until partial admissible path reaches sink node

# Basic Idea of Preflow-Push Algorithm

- Select an active node $i$
- Try to remove the excess $e(i)$ by pushing flow on admissible arcs (push flow to neighbors of $i$ that are closer to $t$ as measured by $d$)
- If active node $i$ has no admissible arcs, increase its distance label
- Terminate when there are no active nodes

# Generic Preflow-Push Algorithm

**algorithm** *preflow-push*
**begin**
  *preprocess*
  **while** the network contains an active node **do**
    select an active node $i$
    *push/relabel*($i$)
**end**

# Specific Implementations

By specifying different rules for selecting active nodes for push/relabel, we can derive different algorithms, each with different worst-case complexity.

**FIFO** : examine active nodes in FIFO order ($\mathcal{O}(n^3)$)

**Highest-Label** : always push from an active node with highest value of distance label ($\mathcal{O}(n^2\sqrt{m})$)

**Excess Scaling** : push flow from node with sufficiently large excess to node with sufficiently small excess ($\mathcal{O}(nm + n^2 log\ U)$)

# Summary of Maximum Flow Algorithms

| | |
|---|---|
| Labeling Algorithm | $\mathcal{O}(nmU)$ |
| Capacity Scaling Algorithm | $\mathcal{O}(nmlog\ U)$ |
| Generic Preflow-Push Algorithm | $\mathcal{O}(n^2m)$ |
| FIFO Preflow-Push Algorithm | $\mathcal{O}(n^3)$ |
| Highest-Label Preflow-Push Algorithm | $\mathcal{O}(n^2\sqrt{m})$ |
| Excess Scaling Algorithm | $\mathcal{O}(nm + n^2log\ U)$ |