CS 580: Algorithm Design and Analysis

Jeremiah Blocki
Purdue University
Spring 2018
Recap: Asymptotic Analysis

Five Representative Problems
- Algorithmic Techniques: Greedy, Dynamic Programming, Network Flow,…
- Computationally Intractable Problems: Unlikely that polynomial time algorithm exists.

Formal Definition of Big $O, \Omega, \Theta$ notation
- $T(n) \in O(f(n))$ ---- upper bound
  - Means we can find constants $c, N > 0$ s.t. whenever $n > N$
    $$T(n) < c \times f(n)$$
  - **Intuition:** $c \times f(n)$ upperbounds $T(n)$ for large enough inputs
- $T(n) \in \Omega(f(n))$ ---- lower bound
- $T(n) \in \Theta(f(n))$ ---- lower bound and upper bound

**Polynomial Time function.** $T(n) \in O(n^d)$ for some constant $d$
(d is independent of the input size).
2.4 A Survey of Common Running Times
Linear Time: $O(n)$

Linear time. Running time is proportional to input size.

Computing the maximum. Compute maximum of $n$ numbers $a_1, \ldots, a_n$.

$$
\begin{align*}
\text{max} & \leftarrow a_1 \\
\text{for } i = 2 \text{ to } n \text{ } \{ \\
\quad \text{if } (a_i > \text{max}) \\
\quad \quad \text{max} \leftarrow a_i \\
\text{\}}
\end{align*}
$$
Linear Time: $O(n)$

**Merge.** Combine two sorted lists $A = a_1, a_2, \ldots, a_n$ with $B = b_1, b_2, \ldots, b_n$ into a sorted whole.

Claim. Merging two lists of size $n$ takes $O(n)$ time.

Pf. After each comparison, the length of output list increases by 1.

$$i = 1, \ j = 1$$

while (both lists are nonempty) {
    if ($a_i \leq b_j$) append $a_i$ to output list and increment $i$
    else append $b_j$ to output list and increment $j$
}

append remainder of nonempty list to output list
O(n log n) Time

\(O(n \log n)\) time. Arises in divide-and-conquer algorithms.
also referred to as linearithmic time

**Sorting.** Mergesort and heapsort are sorting algorithms that perform \(O(n \log n)\) comparisons.

**Largest empty interval.** Given \(n\) time-stamps \(x_1, \ldots, x_n\) on which copies of a file arrive at a server, what is largest interval of time when no copies of the file arrive?

**\(O(n \log n)\) solution.** Sort the time-stamps. Scan the sorted list in order, identifying the maximum gap between successive time-stamps.
Quadratic Time: $O(n^2)$

Quadratic time. Enumerate all pairs of elements.

Closest pair of points. Given a list of $n$ points in the plane $(x_1, y_1)$, ..., $(x_n, y_n)$, find the pair that is closest.

$O(n^2)$ solution. Try all pairs of points.

```
min ← (x_1 - x_2)^2 + (y_1 - y_2)^2
for i = 1 to n {
    for j = i+1 to n {
        d ← (x_i - x_j)^2 + (y_i - y_j)^2
        if (d < min) {
            min ← d
        }
    }
}
```

don’t need to take square roots

Remark. $\Omega(n^2)$ seems inevitable, but this is just an illusion.

see chapter 5
Cubic Time: $O(n^3)$

**Cubic time.** Enumerate all triples of elements.

**Set disjointness.** Given $n$ sets $S_1, \ldots, S_n$ each of which is a subset of $1, 2, \ldots, n$, is there some pair of these which are disjoint?

**$O(n^3)$ solution.** For each pairs of sets, determine if they are disjoint.

```plaintext
foreach set $S_i$ {
    foreach other set $S_j$ {
        foreach element $p$ of $S_i$ {
            determine whether $p$ also belongs to $S_j$
        }
        if (no element of $S_i$ belongs to $S_j$)
            report that $S_i$ and $S_j$ are disjoint
    }
}
```
**Independent set of size k.** Given a graph, are there k nodes such that no two are joined by an edge?

\[ \text{poly-time for } k=17, \text{ but not practical} \]

**O(n^k) solution.** Enumerate all subsets of k nodes.

```
foreach subset S of k nodes {
    check whether S in an independent set
    if (S is an independent set)
        report S is an independent set
}
```

- Check whether S is an independent set = \( O(k^2) \).
- Number of k element subsets = \( \binom{n}{k} = \frac{n(n-1)(n-2)\cdots(n-k+1)}{k(k-1)(k-2)\cdots(2)(1)} \leq \frac{n^k}{k!} \).
Independent set. Given a graph, what is maximum size of an independent set?

$O(n^2 2^n)$ solution. Enumerate all subsets.

```
S* ← ∅
foreach subset S of nodes {
    check whether S in an independent set
    if (S is largest independent set seen so far)
        update S* ← S
}
```
Graphs
3.1 Basic Definitions and Applications
Undirected Graphs

Undirected graph. \( G = (V, E) \)
- \( V = \) nodes.
- \( E = \) edges between pairs of nodes.
- Captures pairwise relationship between objects.
- Graph size parameters: \( n = |V|, m = |E| \).

\[ V = \{ 1, 2, 3, 4, 5, 6, 7, 8 \} \]
\[ E = \{ 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6 \} \]
\[ n = 8 \]
\[ m = 11 \]
## Some Graph Applications

<table>
<thead>
<tr>
<th>Graph</th>
<th>Nodes</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>transportation</td>
<td>street intersections</td>
<td>highways</td>
</tr>
<tr>
<td>communication</td>
<td>computers</td>
<td>fiber optic cables</td>
</tr>
<tr>
<td>World Wide Web</td>
<td>web pages</td>
<td>hyperlinks</td>
</tr>
<tr>
<td>social</td>
<td>people</td>
<td>relationships</td>
</tr>
<tr>
<td>food web</td>
<td>species</td>
<td>predator-prey</td>
</tr>
<tr>
<td>software systems</td>
<td>functions</td>
<td>function calls</td>
</tr>
<tr>
<td>scheduling</td>
<td>tasks</td>
<td>precedence constraints</td>
</tr>
<tr>
<td>circuits</td>
<td>gates</td>
<td>wires</td>
</tr>
</tbody>
</table>
World Wide Web

Web graph.

- **Node**: web page.
- **Edge**: hyperlink from one page to another.
9-11 Terrorist Network

Social network graph.

- Node: people.
- Edge: relationship between two people.

Ecological Food Web

Food web graph.
- Node = species.
- Edge = from prey to predator.

**Graph Representation: Adjacency Matrix**

**Adjacency matrix.** n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.
- Two representations of each edge.
- Space proportional to $n^2$.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.
**Graph Representation: Adjacency List**

**Adjacency list.** Node indexed array of lists.

- Two representations of each edge.
- Space proportional to \( m + n \).
- Checking if \((u, v)\) is an edge takes \(O(\text{deg}(u))\) time.
- Identifying all edges takes \(\Theta(m + n)\) time.

Degree = number of neighbors of \(u\)
Paths and Connectivity

Def. A path in an undirected graph \( G = (V, E) \) is a sequence \( P \) of nodes \( v_1, v_2, \ldots, v_{k-1}, v_k \) with the property that each consecutive pair \( v_i, v_{i+1} \) is joined by an edge in \( E \).

Def. A path is simple if all nodes are distinct.

Def. An undirected graph is connected if for every pair of nodes \( u \) and \( v \), there is a path between \( u \) and \( v \).
Def. A cycle is a path \( v_1, v_2, \ldots, v_{k-1}, v_k \) in which \( v_1 = v_k, k > 2 \), and the first \( k-1 \) nodes are all distinct.

cycle \( C = 1-2-4-5-3-1 \)
Trees

Def. An undirected graph is a tree if it is connected and does not contain a cycle.

Theorem. Let $G$ be an undirected graph on $n$ nodes. Any two of the following statements imply the third.

- $G$ is connected.
- $G$ does not contain a cycle.
- $G$ has $n-1$ edges.
Rooted Trees

**Rooted tree.** Given a tree $T$, choose a root node $r$ and orient each edge away from $r$.

**Importance.** Models hierarchical structure.
Phylogeny trees. Describe evolutionary history of species.
Binary Tree

**Def.** A rooted tree in which each node has at most 2 children

**Def.** Height of a tree is the number of edges in the longest path from root to leaf.

```
root r
1  2  3  4  5  6  7
```

**Thm.** Number of nodes in binary tree of height $h$ is $n \leq 2^{h+1} - 1$. 

**Balanced Binary Tree.** Height $h = O(\log n)$
GUI Containment Hierarchy

GUI containment hierarchy. Describe organization of GUI widgets.

Reference: http://java.sun.com/docs/books/tutorial/uiswing/overview/anatomy.html
3.2 Graph Traversal
Connectivity

**s-t connectivity problem.** Given two nodes $s$ and $t$, is there a path between $s$ and $t$?

**s-t shortest path problem.** Given two nodes $s$ and $t$, what is the length of the shortest path between $s$ and $t$?

**Applications.**
- Navigation (Google Maps).
- Maze traversal.
- Kevin Bacon number.
- Fewest number of hops in a communication network.
Breadth First Search

**BFS intuition.** Explore outward from s in all possible directions, adding nodes one "layer" at a time.

**BFS algorithm.**
- $L_0 = \{ s \}$.
- $L_1$ = all neighbors of $L_0$.
- $L_2$ = all nodes that do not belong to $L_0$ or $L_1$, and that have an edge to a node in $L_1$.
- $L_{i+1}$ = all nodes that do not belong to an earlier layer, and that have an edge to a node in $L_i$.

**Theorem.** For each $i$, $L_i$ consists of all nodes at distance exactly $i$ from $s$. There is a path from $s$ to $t$ iff $t$ appears in some layer.
**Property.** Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
**Property.** Let $T$ be a BFS tree of $G = (V, E)$, and let $(x, y)$ be an edge of $G$. Then the level of $x$ and $y$ differ by at most 1.
Theorem. The above implementation of BFS runs in $O(m + n)$ time if the graph is given by its adjacency representation.

Pf.

- Easy to prove $O(n^2)$ running time:
  - at most $n$ lists $L[i]$
  - each node occurs on at most one list; for loop runs $\leq n$ times
  - when we consider node $u$, there are $\leq n$ incident edges $(u, v)$, and we spend $O(1)$ processing each edge

- Actually runs in $O(m + n)$ time:
  - when we consider node $u$, there are $\deg(u)$ incident edges $(u, v)$
  - total time processing edges is $\sum_{u \in V} \deg(u) = 2m$  

  each edge $(u, v)$ is counted exactly twice in sum: once in $\deg(u)$ and once in $\deg(v)$
Connected Component

Connected component. Find all nodes reachable from $s$.

$\text{Connected component containing node 1} = \{ 1, 2, 3, 4, 5, 6, 7, 8 \}.$
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node:** pixel.
- **Edge:** two neighboring lime pixels.
- **Blob:** connected component of lime pixels.
Flood fill. Given lime green pixel in an image, change color of entire blob of neighboring lime pixels to blue.

- **Node:** pixel.
- **Edge:** two neighboring lime pixels.
- **Blob:** connected component of lime pixels.

Click in the picture to fill that area with color.
**Connected Component**

**Connected component.** Find all nodes reachable from \( s \).

---

\( R \) will consist of nodes to which \( s \) has a path
Initially \( R = \{ s \} \)
While there is an edge \((u, v)\) where \( u \in R \) and \( v \notin R \)
   Add \( v \) to \( R \)
Endwhile

---

**Theorem.** Upon termination, \( R \) is the connected component containing \( s \).
- **BFS** = explore in order of distance from \( s \).
- **DFS** = explore in a different way.
3.4 Testing Bipartiteness
**Bipartite Graphs**

**Def.** An undirected graph $G = (V, E)$ is **bipartite** if the nodes can be colored red or blue such that every edge has one red and one blue end.

**Applications.**
- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.
Testing bipartiteness. Given a graph $G$, is it bipartite?

- Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
- Before attempting to design an algorithm, we need to understand structure of bipartite graphs.
Lemma. If a graph $G$ is bipartite, it cannot contain an odd length cycle.

Pf. Not possible to 2-color the odd cycle, let alone $G$. 

![Diagram showing bipartite and non-bipartite graphs](image)

bipartite (2-colorable)  
not bipartite (not 2-colorable)
**Lemma.** Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.

(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (i)

- Suppose no edge joins two nodes in adjacent layers.
- By previous lemma, this implies all edges join nodes on same level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.

Case (i)
Lemma. Let $G$ be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node $s$. Exactly one of the following holds.

(i) No edge of $G$ joins two nodes of the same layer, and $G$ is bipartite.
(ii) An edge of $G$ joins two nodes of the same layer, and $G$ contains an odd-length cycle (and hence is not bipartite).

Pf. (ii)
- Suppose $(x, y)$ is an edge with $x, y$ in same level $L_j$.
- Let $z = \text{lca}(x, y) = \text{lowest common ancestor}$.
- Let $L_i$ be level containing $z$.
- Consider cycle that takes edge from $x$ to $y$, then path from $y$ to $z$, then path from $z$ to $x$.
- Its length is $1 + (j-i) + (j-i)$, which is odd. □
Corollary. A graph $G$ is bipartite iff it contain no odd length cycle.
3.5 Connectivity in Directed Graphs
Directed Graphs

Directed graph. \( G = (V, E) \)
- Edge \((u, v)\) goes from node \(u\) to node \(v\).

Ex. Web graph - hyperlink points from one web page to another.
- Directedness of graph is crucial.
- Modern web search engines exploit hyperlink structure to rank web pages by importance.
Graph Search

Directed reachability. Given a node \( s \), find all nodes reachable from \( s \).

Directed \( s-t \) shortest path problem. Given two node \( s \) and \( t \), what is the length of the shortest path between \( s \) and \( t \)?

Graph search. BFS extends naturally to directed graphs.

Web crawler. Start from web page \( s \). Find all web pages linked from \( s \), either directly or indirectly.
**Strong Connectivity**

**Def.** Node $u$ and $v$ are **mutually reachable** if there is a path from $u$ to $v$ and also a path from $v$ to $u$.

**Def.** A graph is **strongly connected** if every pair of nodes is mutually reachable.

**Lemma.** Let $s$ be any node. $G$ is strongly connected iff every node is reachable from $s$, and $s$ is reachable from every node.

**Pf.** $\Rightarrow$ Follows from definition.

**Pf.** $\Leftarrow$ Path from $u$ to $v$: concatenate $u$-$s$ path with $s$-$v$ path.

Path from $v$ to $u$: concatenate $v$-$s$ path with $s$-$u$ path.  

\[ \text{ok if paths overlap} \]
**Strong Connectivity: Algorithm**

**Theorem.** Can determine if $G$ is strongly connected in $O(m + n)$ time.

**Pf.**
- Pick any node $s$.
- Run BFS from $s$ in $G$.
- Run BFS from $s$ in $G^{rev}$.
- Return true iff all nodes reached in both BFS executions.
- Correctness follows immediately from previous lemma. □

reverse orientation of every edge in $G$

---

![Diagram of strongly connected and not strongly connected graphs](image)

strongly connected

not strongly connected
3.6 DAGs and Topological Ordering
Directed Acyclic Graphs

**Def.** An **DAG** is a directed graph that contains no directed cycles.

**Ex.** Precedence constraints: edge \((v_i, v_j)\) means \(v_i\) must precede \(v_j\).

**Def.** A **topological order** of a directed graph \(G = (V, E)\) is an ordering of its nodes as \(v_1, v_2, \ldots, v_n\) so that for every edge \((v_i, v_j)\) we have \(i < j\).
Precedence Constraints

Precedence constraints. Edge \((v_i, v_j)\) means task \(v_i\) must occur before \(v_j\).

Applications.

- Course prerequisite graph: course \(v_i\) must be taken before \(v_j\).
- Compilation: module \(v_i\) must be compiled before \(v_j\). Pipeline of computing jobs: output of job \(v_i\) needed to determine input of job \(v_j\).
Lemma. If $G$ has a topological order, then $G$ is a DAG.

Pf. (by contradiction)

- Suppose that $G$ has a topological order $v_1, \ldots, v_n$ and that $G$ also has a directed cycle $C$. Let's see what happens.
- Let $v_i$ be the lowest-indexed node in $C$, and let $v_j$ be the node just before $v_i$; thus $(v_j, v_i)$ is an edge.
- By our choice of $i$, we have $i < j$.
- On the other hand, since $(v_j, v_i)$ is an edge and $v_1, \ldots, v_n$ is a topological order, we must have $j < i$, a contradiction. ▪
Directed Acyclic Graphs

Lemma. If $G$ has a topological order, then $G$ is a DAG.

Q. Does every DAG have a topological ordering?

Q. If so, how do we compute one?
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a node with no incoming edges.

Pf. (by contradiction)

- Suppose that $G$ is a DAG and every node has at least one incoming edge. Let's see what happens.
- Pick any node $v$, and begin following edges backward from $v$. Since $v$ has at least one incoming edge $(u, v)$ we can walk backward to $u$.
- Then, since $u$ has at least one incoming edge $(x, u)$, we can walk backward to $x$.
- Repeat until we visit a node, say $w$, twice.
- Let $C$ denote the sequence of nodes encountered between successive visits to $w$. $C$ is a cycle. ▪
Directed Acyclic Graphs

Lemma. If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:

1. Find a node $v$ with no incoming edges and order it first.
2. Delete $v$ from $G$.
3. Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. 

---

DAG

$\text{Lemma.}$ If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:

1. Find a node $v$ with no incoming edges and order it first.
2. Delete $v$ from $G$.
3. Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. 

---

DAG

$\text{Lemma.}$ If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:

1. Find a node $v$ with no incoming edges and order it first.
2. Delete $v$ from $G$.
3. Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. 

---

DAG

$\text{Lemma.}$ If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:

1. Find a node $v$ with no incoming edges and order it first.
2. Delete $v$ from $G$.
3. Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. 

---

DAG

$\text{Lemma.}$ If $G$ is a DAG, then $G$ has a topological ordering.

Pf. (by induction on $n$)

- Base case: true if $n = 1$.
- Given DAG on $n > 1$ nodes, find a node $v$ with no incoming edges.
- $G - \{ v \}$ is a DAG, since deleting $v$ cannot create cycles.
- By inductive hypothesis, $G - \{ v \}$ has a topological ordering.
- Place $v$ first in topological ordering; then append nodes of $G - \{v\}$ in topological order. This is valid since $v$ has no incoming edges.

To compute a topological ordering of $G$:

1. Find a node $v$ with no incoming edges and order it first.
2. Delete $v$ from $G$.
3. Recursively compute a topological ordering of $G - \{v\}$ and append this order after $v$. 

---

DAG
Theorem. Algorithm finds a topological order in $O(m + n)$ time.

Pf.

- Maintain the following information:
  - $\text{count}[w] =$ remaining number of incoming edges
  - $S =$ set of remaining nodes with no incoming edges
- Initialization: $O(m + n)$ via single scan through graph.
- Update: to delete $v$
  - remove $v$ from $S$
  - decrement $\text{count}[w]$ for all edges from $v$ to $w$, and
    add $w$ to $S$ if $c \text{count}[w]$ hits 0
  - this is $O(1)$ per edge