Elements of Greedy Algorithms

No general way to tell if a greedy algorithm is optimal, but two key ingredients are:

- **Greedy-choice Property.**
  - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice.
- **Optimal Substructure.**

Recap Greedy Algorithms

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.
- Prove that there’s always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.
- Show that greedy choice and optimal solution to subproblem ⇒ optimal solution to the problem.
- Make the greedy choice and solve top-down.
- May have to preprocess input to put it into greedy order (e.g. sorting activities by finish time).

Activity-selection Problem
For one list per vertex. Consists of an array $Adj$ of $|V|$ lists.

- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to $u$.

### Activity-selection Problem

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

### Graphs

- If $(u, v) \in E$, then vertex $v$ is adjacent to vertex $u$.
- Adjacency relationship is:
  - Symmetric if $G$ is undirected.
  - Not necessarily so if $G$ is directed.
- If $G$ is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then $G$ is a tree.

### Adjacency Lists

- Consists of an array $Adj$ of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to $u$.

### Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is
  \[ \sum_{v \in V} \text{out-degree}(v) = |E| \]
  - Total storage: $\Theta(|V| \cdot |E|)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is
  \[ \sum_{v \in V} \text{degree}(v) = 2|E| \]
  - Total storage: $\Theta(|V| \cdot |E|)$

### Representation of Graphs

- Two standard ways.
  - Adjacency Lists.
  - Adjacency Matrix.

### Types of graphs

- Undirected: edge $(u, v) = (v, u)$; for all $v, (v, v) \notin E$ (No self loops.)
- Directed: $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed.
- Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbb{R}$.
- Dense: $|E| = |V|^2$.
- Sparse: $|E| \ll |V|^2$.
- $|E| = O(|V|^2)$

### Graphs

$G = (V, E)$

- $V$ = set of vertices
- $E$ = set of edges $\subseteq (V \times V)$

#### Activity-selection Problem

<table>
<thead>
<tr>
<th>$i$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_i$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>$f_i$</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

Activities sorted by finishing time.

#### Graphs

- If $(u, v) \in E$, then vertex $v$ is adjacent to vertex $u$.
- Adjacency relationship is:
  - Symmetric if $G$ is undirected.
  - Not necessarily so if $G$ is directed.
- If $G$ is connected:
  - There is a path between every pair of vertices.
  - $|E| \geq |V| - 1$.
  - Furthermore, if $|E| = |V| - 1$, then $G$ is a tree.

#### Adjacency Lists

- Consists of an array $Adj$ of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to $u$.

#### Representation of Graphs

- Two standard ways.
  - Adjacency Lists.
  - Adjacency Matrix.

#### Types of graphs

- Undirected: edge $(u, v) = (v, u)$; for all $v, (v, v) \notin E$ (No self loops.)
- Directed: $(u, v)$ is edge from $u$ to $v$, denoted as $u \rightarrow v$. Self loops are allowed.
- Weighted: each edge has an associated weight, given by a weight function $w : E \rightarrow \mathbb{R}$.
- Dense: $|E| = |V|^2$.
- Sparse: $|E| \ll |V|^2$.
- $|E| = O(|V|^2)$

#### Graphs

$G = (V, E)$

- $V$ = set of vertices
- $E$ = set of edges $\subseteq (V \times V)$

#### Adjacency Lists

- Consists of an array $Adj$ of $|V|$ lists.
- One list per vertex.
- For $u \in V$, $Adj[u]$ consists of all vertices adjacent to $u$. 

#### Storage Requirement

- For directed graphs:
  - Sum of lengths of all adj. lists is
  \[ \sum_{v \in V} \text{out-degree}(v) = |E| \]
  - Total storage: $\Theta(|V| \cdot |E|)$
- For undirected graphs:
  - Sum of lengths of all adj. lists is
  \[ \sum_{v \in V} \text{degree}(v) = 2|E| \]
  - Total storage: $\Theta(|V| \cdot |E|)$
Pros and Cons: adj list

**Pros**
- Space-efficient, when a graph is sparse.
- Can be modified to support many graph variants.

**Cons**
- Determining if an edge \( (u, v) \) \( \in G \) is not efficient.
  - Have to search in \( u \)'s adjacency list. \( \Theta(\text{degree}(u)) \) time.
  - \( \Theta(V) \) in the worst case.

### Adjacency Matrix

- \( |V| \times |V| \) matrix \( A \).
- Number vertices from 1 to \( |V| \) in some arbitrary manner.
- \( A \) is then given by:
  \[
  A[i,j] = \begin{cases} 
  1 & \text{if} \ (i,j) \in E \\
  0 & \text{otherwise}
  \end{cases}
  \]

### Space and Time

**Space:** \( \Theta(V^2) \).
- Not memory efficient for large sparse graphs.
**Time:** to list all vertices adjacent to \( u \): \( \Theta(V) \).
**Time:** to determine if \( (u, v) \in E \): \( \Theta(1) \).
**Can store weights instead of bits for weighted graph.**

### Graph-searching Algorithms (COMP250)

- Searching a graph:
  - Systematically follow the edges of a graph
  to visit the vertices of the graph.
- Used to discover the structure of a graph.
- Standard graph-searching algorithms.
  - Breadth-first Search (BFS).
  - Depth-first Search (DFS).

### Breadth-first Search

- Expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier.
- A vertex is “discovered” the first time it is encountered during the search.
- A vertex is “finished” if all vertices adjacent to it have been discovered.
- Colors the vertices to keep track of progress.
  - White – Undiscovered.
  - Gray – Discovered but not finished.
  - Black – Finished.
- Colors are required only to reason about the algorithm. Can be implemented without colors.
Example (BFS)

Example (BFS)

Example (BFS)

Example (BFS)

Analysis of BFS

- Initialization takes $O(V)$.
- Traversal Loop
  - After initialization, each vertex is enqueued and dequeued at most once, and each operation takes $O(1)$. So, total time for queuing is $O(V)$.
  - The adjacency list of each vertex is scanned at most once. The sum of lengths of all adjacency lists is $\Theta(E)$.
- Summing up over all vertices $\Rightarrow$ total running time of BFS is $O(V+E)$, linear in the size of the adjacency list representation of graph.

Depth-first Search (DFS)

- Explore edges out of the most recently discovered vertex $v$.
- When all edges of $v$ have been explored, backtrack to explore other edges leaving the vertex from which $v$ was discovered (its predecessor).
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.
- If any undiscovered vertices remain, then one of them is chosen as a new source and search is repeated from that source.
Depth-first Search

- **Input:** $G = (V, E)$, directed or undirected. No source vertex given!
- **Output:**
  - 2 timestamps on each vertex. Integers between 1 and $2|V|$.
  - $d[v]$ = discovery time (v turns from white to gray)
  - $f[v]$ = finishing time (v turns from gray to black)
  - $\pi[v]$ : predecessor of $v = u$, such that $v$ was discovered during the scan of $u$’s adjacency list.
- Uses the same coloring scheme for vertices as BFS.

**Pseudo-code**

**DFS**

1. for each vertex $u \in V(G)$
2. do $\text{color}[u] \leftarrow \text{WHITE}$
3. $\pi[u] \leftarrow \text{NIL}$
4. $\text{time} \leftarrow 0$
5. for each vertex $u \in V(G)$
6. do if $\text{color}[u] = \text{WHITE}$
7. then $\text{DFS-Visit}(u)$

**DFS-Visit**(u)

1. $\text{color}[u] \leftarrow \text{GRAY}$
2. $\text{time} \leftarrow \text{time} + 1$
3. $\text{d}[u] \leftarrow \text{time}$
4. for each $v \in \text{Adj}[u]$
5. do if $\text{color}[v] = \text{WHITE}$
6. then $\text{color}[v] \leftarrow \text{gray}$
7. $\text{DFS-Visit}(v)$
8. $\text{color}[u] \leftarrow \text{BLACK}$
9. $\text{f}[u] \leftarrow \text{time} \leftarrow \text{time} + 1$

**Example (DFS)**

- Diagram of a graph and its DFS traversal, showing discovery and finishing times, and predecessor relationships.

- Diagram of another graph and its DFS traversal, illustrating the same process.

- Diagram of a third graph and its DFS traversal, highlighting the same features.
Analysis of DFS

- Loops on lines 1-2 & 5-7 take $\Theta(V)$ time, excluding time to execute DFS-Visit.
- DFS-Visit is called once for each white vertex $v \in V$ when it’s painted gray the first time. Lines 3-6 of DFS-Visit is executed $|\text{Adj}[v]|$ times. The total cost of executing DFS-Visit is $\sum_{v \in V} |\text{Adj}[v]| = \Theta(E)$.
- Total running time of DFS is $\Theta(V+E)$.

Example (DFS)

Parenthesis Theorem

Theorem 1:
For all $u, v$, exactly one of the following holds:
1. $d(u) < f[u] < d(v) < f[v]$ or $d(v) < f[v] < d(u) < f[u]$ and neither $u$ nor $v$ is a descendant of the other.
2. $d(u) < d(v) < f[v] < f[u]$ and $v$ is a descendant of $u$.
3. $d(v) < d(u) < f[u] < f[v]$ and $u$ is a descendant of $v$.

- So $d(u) < d(v) < f[u] < f[v]$ cannot happen.
- Like parentheses:
  - OK: $(())()()$
  - Not OK: $(()()())$

Corollary
$v$ is a proper descendant of $u$ if and only if $d(u) < d(v) < f[v] < f[u]$.

Example (Parenthesis Theorem)

White-path Theorem

Theorem 2
$v$ is a descendant of $u$ if and only if at time $d[u]$, there is a path $u \rightsquigarrow v$ consisting of only white vertices. (Except for $u$, which was just colored gray.)

Example (DFS)

$\text{v, y, and } x \text{ are descendants of } u$. 
Classification of Edges

- **Tree edge**: in the depth-first forest. Found by exploring \((u, v)\).
- **Back edge**: \((u, v)\), where \(u\) is a descendant of \(v\) (in the depth-first tree).
- **Forward edge**: \((u, v)\), where \(v\) is a descendant of \(u\), but not a tree edge.
- **Cross edge**: any other edge. Can go between vertices in same depth-first tree or in different depth-first trees.

**Theorem 3**
In DFS of an undirected graph, we get only tree and back edges. No forward or cross edges.

Identification of Edges

- Edge type for edge \((u, v)\) can be identified when it is first explored by DFS.
- Identification is based on the color of \(v\).
  - White – tree edge.
  - Gray – back edge.
  - Black – forward or cross edge.

Directed Acyclic Graph

- **DAG** – Directed graph with no cycles.
- Good for modeling processes and structures that have a **partial order**:
  - \(a > b\) and \(b > c\) \(\Rightarrow\) \(a > c\).
  - But may have \(a\) and \(b\) such that neither \(a > b\) nor \(b > a\).
- Can always make a **total order** (either \(a > b\) or \(b > a\) for all \(a \neq b\)) from a partial order.

Characterizing a DAG

**Lemma 1**
A directed graph \(G\) is acyclic iff a DFS of \(G\) yields no back edges.

**Proof**:
- \(\Rightarrow\): Show that back edge \(\Rightarrow\) cycle.
  - Suppose there is a back edge \((u, v)\). Then \(v\) is ancestor of \(u\) in depth-first forest.
  - Therefore, there is a path \(v \sim u\), so \(v \sim u \sim v\) is a cycle.
Characterizing a DAG

**Lemma 2**
A directed graph $G$ is acyclic iff a DFS of $G$ yields no back edges.

**Proof (Contd.):**
- $\Leftarrow$: Show that a cycle implies a back edge.
  - At time $d[v]$, vertices of $c$ form a white path $v$ to $u$.
  - By white-path theorem, $u$ is a descendant of $v$ in depth-first forest.
  - Therefore, $(u, v)$ is a back edge.

**Topological Sort**

Want to "sort" a directed acyclic graph (DAG).

Think of original DAG as a **partial order**.

Want a **total order** that extends this partial order.

**Topological Sort**

- Performed on a DAG.
- Linear ordering of the vertices of $G$ such that if $(u, v) \in E$, then $u$ appears somewhere before $v$.

**Topological-Sort ($G$)**
1. call DFS($G$) to compute finishing times $f[v]$ for all $v \in V$
2. as each vertex is finished, insert it onto the front of a linked list
3. return the linked list of vertices

Time: $\Theta(V + E)$.

**Example 1**

Linked List:

$\text{A}$ $\text{B}$ $\text{C}$ $\text{D}$ $\text{E}$
Example 1

Linked List:

A
B
D
C
E
1/4
2/3
E
2/3
1/4
D
5/8
6/7
6/7
C
5/8
B
9/10
A

Example 2

socks
shorts
hose
pants
skates
leg pads
T-shirt
chest pad
sweater
mask
catch glove
blocker
bawng
glove
26
24
23
22
21
14
13
12
11
6
5
4
3
2
1

Correctness Proof

• Just need to show if $(u, v) \in E$, then $f(v) < f(u)$.
• When we explore $(u, v)$, what are the colors of $u$ and $v$?
  – $u$ is gray.
  – Is $v$ gray, too?
    • No, because then $v$ would be an ancestor of $u$.
    • $\Rightarrow$ contradiction of Lemma 2 (DAG has no back edges).
  – Is $v$ white?
    • Then becomes descendant of $u$.
  – Is $v$ black?
    • Then $v$ is already finished.
    • Since we’re exploring $(u, v)$, we have not yet finished $u$.
    • Therefore, $f(v) < f(u)$.

Strongly Connected Components

• $G$ is strongly connected if every pair $(u, v)$ of vertices in $G$ is reachable from one another.
• A strongly connected component (SCC) of $G$ is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$, both $u \leadsto v$ and $v \leadsto u$ exist.

Component Graph

• $G^{SCC} = (V^{SCC}, E^{SCC})$.
• $V^{SCC}$ has one vertex for each SCC in $G$.
• $E^{SCC}$ has an edge if there’s an edge between the corresponding SCC’s in $G$.
• $G^{SCC}$ for the example considered:

$G^{SCC}$ is a DAG

Lemma 2
Let $C$ and $C'$ be distinct SCC’s in $G$, let $u, v \in C$, $u', v' \in C'$, and suppose there is a path $u \leadsto v$ in $G$. Then there cannot also be a path $v' \leadsto u'$ in $G$.

Proof:
• Suppose there is a path $v' \leadsto u'$ in $G$.
• Then there are paths $u \leadsto u' \leadsto v'$ and $v \leadsto v' \leadsto u$ in $G$.
• Therefore, $u$ and $v'$ are reachable from each other, so they are not in separate SCC’s.
Transpose of a Directed Graph

- $G^T = \text{transpose of directed } G$.
  - $G^T = (V, E'), E' = \{(u, v) : (v, u) \in E\}$.
  - $G^T$ is $G$ with all edges reversed.
- Can create $G^T$ in $\Theta(V + E)$ time if using adjacency lists.
- $G$ and $G^T$ have the same SCC’s. ($u$ and $v$ are reachable from each other in $G$ if and only if reachable from each other in $G^T$.)

Algorithm to determine SCCs

1. call DFS($G$) to compute finishing times $f[u]$ for all $u$
2. compute $G^T$ 
3. call DFS($G^T$), but in the main loop, consider vertices in order of decreasing $f[u]$ (as computed in first DFS)
4. output the vertices in each tree of the depth-first forest formed in second DFS as a separate SCC

Time: $\Theta(V + E)$. 

Example

$G$

Example

$G^T$

Example

$G$

Example

$G^T$
**Proof:**

- **Case 1:** $d(C) < d(C')$
  - Let $u$ be the first vertex discovered in $C$.
  - At time $d(u)$, all vertices in $C$ and $C'$ are white. Thus, there exist paths of white vertices from $u$ to all vertices in $C$ and $C'$.
  - By the white-path theorem, all vertices in $C$ and $C'$ are descendants of $u$ in depth-first tree.
  - By the parent-theorem, $f[u] = \hat{f}(C) > \hat{f}(C')$.

- **Case 2:** $d(C) > d(C')$
  - Let $u$ be the first vertex discovered in $C$.
  - At time $d(u)$, all vertices in $C$ and $C'$ are white. Thus, there exist white paths from $u$ to each vertex in $C$.
  - All vertices in $C$ become descendants of $u$. Again, $f[u] = \hat{f}(C)$.
  - By earlier lemma, since there is an edge $(u, v)$, we cannot have a path from $C'$ to $C$.
  - So no vertex in $C$ is reachable from $v$.
  - Therefore, at time $f[y]$, all vertices in $C$ are still white.
  - Therefore, for all $w \in C, f[w] > f[y]$, which implies that $f(C') > f(C)$.

**Corollary 1**

Let $C$ and $C'$ be distinct SCCs in $G = (V, E)$. Suppose there is an edge $(u, v) \in E$ such that $u \in C$ and $v \in C'$. Then $f(C') > f(C)$.

**Proof:**

- $(u, v) \in E' \Rightarrow (v, u) \in E$.
- Since SCCs of $G$ and $G'$ are the same, $f(C') > f(C)$, by Lemma.
Correctness of SCC

- The next root chosen in the second DFS is in SCC $C'$ such that $f(C')$ is maximum over all SCC's other than $C$.
  - DFS visits all vertices in $C'$, but the only edges out of $C'$ go to $C$, which we've already visited.
  - Therefore, the only tree edges will be to vertices in $C$.
- We can continue the process.
- Each time we choose a root for the second DFS, it can reach only
  - vertices in its SCC—get tree edges to these,
  - vertices in SCC's already visited in second DFS—get no tree edges to these.