Taking Stock

Stuff We Learned

- Dynamic Programming (15.[1,3])
- Greedy Algorithms (16.[1,2])
- Graphs and Search (22.*)
- Spanning Trees (23.*)
- (Single Source) Shortest Paths (24.[1,2,3])
- (All Pairs) Shortest Paths (25.[1,2])
- Max Flow (26.[1,2,3])

Dynamic Programming

Dynamic Programming in a Nutshell

- Characterize the structure of an optimal solution
- Recursively define the value of an optimal solution
- Compute the value of an optimal solution “from the bottom up”
- Construct optimal solution (if required)

Examples

- Assembly Line Balancing
- Lot Sizing
Assembly Line Balancing

- Let \( f_i(j) \) be the fastest time to get through \( S_{ij} \) \( \forall i = 1, 2, \forall j = 1, 2, \ldots, n \)

\[
    f^* = \min(f_1(n) + x_1, f_2(n) + x_2) \\
    f_1(1) = e_1 + a_{11} \\
    f_2(1) = e_2 + a_{21} \\
    f_1(j) = \min(f_1(j - 1) + a_{1j}, f_2(j - 1) + t_{2,j-1} + a_{1j}) \\
    f_2(j) = \min(f_2(j - 1) + a_{2j}, f_1(j - 1) + t_{1,j-1} + a_{2j})
\]

Lot Sizing

- Let \( f_t(s) \): be the minimum cost of meeting demands from \( t, t + 1, \ldots, T \) (\( t \) until the end) if \( s \) units are in inventory at the beginning of period \( t \)

\[
    f_t(s) = \min_{x \in 0, 1, 2, \ldots} \{ c_t(x) + h_t(s + x - d_t) + f_{t+1}(s + x - d_t) \}.
\]

- \( c_{ij} \) be the size of a maximum-sized subset of mutually compatible jobs in \( S_{ij} \).
- If \( S_{ij} = \emptyset \), then \( c_{ij} = 0 \)
- If \( S_{ij} \neq \emptyset \), then \( c_{ij} = c_{ik} + 1 + c_{kj} \) for some \( k \in S_{ij} \). We pick the \( k \in S_{ij} \) that maximizes the number of jobs:

\[
    c_{ij} = \begin{cases} 
        0 & \text{if } S_{ij} = \emptyset \\
        \max_{k \in S_{ij}} c_{ik} + c_{kj} + 1 & \text{if } S_{ij} \neq \emptyset
    \end{cases}
\]

- Note we need only check \( i < k < j \)

To Solve \( S_{ij} \)

1. Choose \( m \in S_{ij} \) with the earliest finish time. The Greedy Choice
2. Then solve problem on jobs \( S_{mj} \)

Greedy

- Greedy is not always optimal!
- But it sometimes works:

Activity Selection

- Let \( S_{ij} \subseteq A \) be the set of activities that start after activity \( i \) needs to finish and before activity \( j \) needs to start:

\[
    S_{ij} \overset{\text{def}}{=} \{ k \in S \mid f_i \leq s_k, f_k \leq s_j \}
\]

- Let’s assume that we have sorted the activities such that

\[
    f_1 \leq f_2 \leq \cdots \leq f_n
\]

- Schedule jobs in \( S_{0,n+1} \)

Graphs!

- Adjacency List, Adjacency Matrix
- Breadth First Search
- Depth First Search

BFS

- Input: Graph \( G = (V, E) \), source node \( s \in V \)
- Output: \( d(v) \), distance (smallest # of edges) from \( s \) to \( v \) \( \forall v \in V \)
- Output: \( \pi(v) \), predecessor of \( v \) on the shortest path from \( s \) to \( v \)
**BFS**

BFS($V, E, s$)

1. for each $u$ in $V \setminus \{s\}$
2. do $d(u) \leftarrow \infty$
3. $\pi(u) \leftarrow$ NIL
4. $d[s] \leftarrow 0$
5. $Q \leftarrow \emptyset$
6. add($Q, s$)
7. while $Q \neq \emptyset$
8. do $u \leftarrow$ poll($Q$)
9. for each $v$ in $\text{Adj}[u]$
10. do if $d[v] = \infty$
11. then $d[v] \leftarrow d[u] + 1$
12. $\pi[v] = u$
13. add($Q, v$)

**DFS**

**DFS**

- **Input:** Graph $G = (V, E)$
- **Output:** Two timestamps for each node $d(v), f(v)$,
- **Output:** $\pi(v)$, predecessor of $v$
- not on shortest path necessarily

DFS($V, E$)

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

**DFS (Visit Node—Recursive)**

DFS-Visit($u$)

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

**Classifying Edges in the DFS Tree**

Given a DFS Tree $G_\pi$, there are four types of edges $(u, v)$

- **Tree Edges:** Edges in $E_\pi$. These are found by exploring $(u, v)$ in the DFS procedure
- **Back Edges:** Connect $u$ to an ancestor $v$ in a DFS tree
- **Forward Edges:** Connect $u$ to a descendent $v$ in a DFS tree
- **Cross Edges:** All other edges. They can be edges in the same DFS tree, or can cross trees in the DFS forest $G_\pi$
Modifying DFS to Classify Edges

- DFS can be modified to classify edges as it encounters them...
- Classify $e = (u, v)$ based on the color of $v$ when $e$ is first explored...
- GREEN: Indicates Tree Edge
- YELLOW: Indicates Back Edge
- RED: Indicates Forward or Cross Edge

Stuff You Can Do with DFS

Topological Sort: The Whole Algorithm
1. DFS search the graph
2. List vertices in order of decreasing finishing time

Strongly Connected Components
1. Call DFS($G$) to topologically sort $G$
2. Compute $G^T$
3. Call DFS($G^T$) but consider vertices in topologically sorted order (from $G$)
4. Vertices in each tree of depth-first forest for SCC

Spanning Tree

Kruskal’s Algorithm

KRUSKAL($V, E, w$)
1. $A \leftarrow \emptyset$
2. for each $v$ in $V$
3. do MAKE-SET($v$)
4. SORT($E, w$)
5. for each $(u, v)$ in (sorted) $E$
6. do if FIND-SET($u$) $\neq$ FIND-SET($v$)
7. then $A \leftarrow A \cup \{(u, v)\}$
8. UNION($u, v$) return $A$

Prim’s Algorithm

- Builds one tree, so $A$ is always a tree
- Let $V_A$ be the set of vertices on which $A$ is incident
- Start from an arbitrary root $r$
- At each step find a light edge crossing the cut $(V_A, V \setminus V_A)$
Pseudocode for Prim

**Prim**(*V*, *E*, *w*, *r*)
1.  
2.  for each \( u \in V \)
3.      do 
4.          key[\( u \)] ← \( \infty \)
5.          \( \pi[\( u \)] \leftarrow \text{NIL.Insert}(Q, u) \)
6.  key[\( r \)] = 0
7.  while \( Q \neq \emptyset \)
8.      do \( u \leftarrow \text{Extract-Min}(Q) \)
9.          for each \( v \in \text{Adj}[\( u \)] \)
10.         do if \( v \in Q \) and \( w_{uv} < \text{key}[\( v \)] \)
11.              then \( \pi[\( v \)] \leftarrow \( u \) \)
12.                  key[\( v \)] = \( w_{uv} \)

Shortest Paths

- (Single Source) shortest-path algorithms produce a label: \( d[\( v \)] = \delta(s, v) \).
- Initially \( d[\( v \)] = \infty \), reduces as the algorithm goes, so always \( d[\( v \)] \geq \delta(s, v) \).
- Also produce labels \( \pi[\( v \)] \), predecessor of \( v \) on a shortest path from \( s \).

Relax!

- The algorithms work by improving (lowering) the shortest path estimate \( d[\( v \)] \).
- This operation is called **relaxing** an edge \((u, v)\).
- Can we **improve** the shortest-path estimate for \( v \) by going through \( u \) and taking \((u, v)\)?

**Relax**(*u*, *v*, *w*)
1.  if \( d[\( v \)] > d[\( u \)] + w_{uv} \)
2.      then \( d[\( v \)] \leftarrow d[\( u \)] + w_{uv} \)
3.          \( \pi[\( v \)] \leftarrow \( u \) \)

Lemma, Lemma, Lemma

Path Relaxation Property

Let \( P = \{v_0, v_1, \ldots, v_k\} \) be a shortest path from \( s = v_0 \) to \( v_k \). If the edges \((v_0, v_1), (v_1, v_2), (v_{k-1}, v_k)\) are relaxed in that order, (there can be other relaxations in-between), then \( d[\( v_k \)] = \delta(s, v_k) \).
Bellman-Ford Algorithm

- Works with Negative-Weight Edges
- Returns \texttt{true} if there are no negative-weight cycles reachable from \( s \), \texttt{false} otherwise

\texttt{Bellman-Ford(V, E, w, s)}

1. \texttt{Init-Single-Source(V, s)}
2. for \( i \leftarrow 1 \) to \(|V| - 1\)
   3. do for each \((u, v)\) in \( E \)
      4. do \texttt{Relax}(u, v, w)
     5. for each \((u, v)\) in \( E \)
       6. do if \( d[v] > d[u] + w_{uv} \)
       7. then return \texttt{False}
9. return \texttt{True}

SSSP Dijkstra

DAG-Shortest-Paths (\( V, E, s, w \))

1. \texttt{Init-Single-Source(V, s)}
2. topologically sort the vertices
3. for each \( u \) in topologically sorted \( V \)
4. do for each \( v \in Adj[u] \)
5. do \texttt{RELAX}(u, v, w)

All Pairs Shortest Paths

- The output of an all pairs shortest path algorithm is a matrix \( D = (d)_{ij} \), where \( d_{ij} = \delta(i, j) \)
- DP: \( \ell^{(m)}_{ij} \) be the shortest path from \( i \in V \) to \( j \in V \) that uses \( \leq m \) edges
  \[ \ell^{(m)}_{ij} = \min_{1 \leq k \leq n} \left( \ell^{(m-1)}_{ik} + w_{kj} \right) \]

\texttt{All-Pairs-Shortest-Paths(V, E, w)}

1. create \((n \times n)\) matrix \( L' \)
2. for \( i \leftarrow 1 \) to \( n \)
3. do for \( j \leftarrow 1 \) to \( n \)
4. do \( \ell'_{ij} \leftarrow \infty \)
5. for \( k \leftarrow 1 \) to \( n \)
6. do \( \ell'_{ij} \leftarrow \min(\ell'_{ij}, \ell_{ik} + w_{kj}) \)

- Dijkstra's Algorithm Runs in \( O(E \lg V) \), with a binary heap implementation.
- We can speed this up.
Faster All-Pairs-Shortest-Paths

APSP2(W)
1 \( L^{(1)} = W \)
2 \( m \leftarrow 1 \)
3 while \( m \leq n - 1 \)
4 do \( L^{(2m)} = \text{Extend}(L^m, L^m) \)
5 \( m \leftarrow 2m \)
6 return \( L^{(m)} \)

- OK to “overshoot” \( n - 1 \), since shortest path labels don’t change after \( m = n - 1 \) (since no negative cycles)
- “Repeated squaring” is a technique used to improve the efficiency of lots of other algorithms
- Analysis:

Floyd-Warshall

\[ d_{ij}^{(k)} = \begin{cases} 
  w_{ij} & k = 0 \\
  \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & k \geq 1 
\end{cases} \]

- We look for \( D^{(n)} = (d_{ij}^{(n)}) \)

Flows

- A net flow is a function \( f : V \times V \rightarrow \mathbb{R}^{|V| \times |V|} \) that satisfies three conditions:
- Capacity Constraints:
  \[ 0 \leq f(u, v) \leq c(u, v) \]
- Skew Symmetry:
  \[ f(u, v) = -f(v, u) \quad \forall u, v \in V \]
- Flow Conservation:
  \[ \sum_{v \in V} f(u, v) = 0 \quad \forall u \in V \setminus \{s, t\} \]

**The Maximum Flow Problem**

Given \( G = (V, E) \), source node \( s \in V \), sink node \( t \in V \), edge capacities \( c \). Find a flow whose value is maximum.
Flow Phacts

- For any cut \((S, T)\), \(f(S, T) = |f|\)
- Residual capacity of arcs given flow:
  \[c_f(u, v) \overset{\text{def}}{=} c(u, v) - f(u, v) \geq 0.\]
- Give flow \(f\), we can create a residual network from the flow.
  \(G_f = (V, E_f)\), with
  \[E_f \overset{\text{def}}{=} \{(u, v) \in V \times V \mid c_f(u, v) > 0\},\]
  so that each edge in the residual network can admit a positive flow.

Max-Flow Min-Cut Theorem

The following statements are equivalent
1. \(f\) is a maximum flow
2. \(f\) admits no augmenting path. (No \((s, t)\) path in residual network)
3. \(|f| = c(S, T)\) for some cut \((S, T)\)

Ford-Fulkerson (\(V, E, c, s, t\))
1. for \(i \leftarrow 1\) to \(n\)
2. do \(f[u, v] \leftarrow f[v, u] \leftarrow 0\)
3. while \(\exists\) augmenting path \(P\) in \(G_f\)
4. do augment \(f\) by \(c_f(P)\)

Analysis of this? Do better algorithms exist?

Maximum Bipartite Matching

- A graph \(G = (V, E)\) is bipartite if we can partition the vertices into \(V = L \cup R\) such that all edges in \(E\) go between \(L\) and \(R\)
- A matching is a subset of edges \(M \subseteq E\) such that for all \(v \in V, \leq 1\) edge of \(M\) is incident upon it.

Stuff To Know: EVERYTHING!

DP and Greedy
- Develop (and potentially solve small) problems via DP
- Activity Selection (or related problems): Greedy Works

Graphs
- BFS, DFS, and Analysis.
- Classifying edges in directed and undirected graphs
- Topological Sorting
- Finding Strongly Connected Components

Spanning Trees
- Kruskal’s Algorithm (and analysis)
- Prim’s Algorithm (and analysis)
More Stuff To Know...

**Single Source Shortest Paths**
- Distance Labels and Relax
- Path Relaxation Property
- Bellman-Ford Algorithm
  - How to do it
  - When (Why?) it works
  - Analysis
- SSSP Dag
  - How to do it
  - When (Why?) it works
  - Analysis
- Dijkstra’s Algorithm
  - How to do it
  - When (Why?) it works
  - Analysis

Even More Stuff To Know...

**All Pairs Shortest Paths**
- Analogue to Matrix Multiplication
- Floyd-Warshall
  - How to do it?
  - When (Why?) it works?
  - Analysis

**Flows**
- What is a flow?
- What is a cut?
- What is MFMC Theorem?
- How to create residual graph \( G_f \)?
- How to do Augmenting Paths algorithm (Ford Fulkerson/Edmonds Karp)
- Analysis

Next Time

- **Quiz!** April 4
- No Class: Friday April 6. Have a nice holiday! We start numerical methods on Monday