Greedy Method

- Backtracking - recursive brute force
- Divide and Conquer - e.g. mergesort

Interval Scheduling

input: \{I_1, ..., I_n\}, I_i = [s_i, t_i]
output: max num of disjoint intervals

for greedy to work optimally, every solution to a subproblem must be part of solution to problem constraint search space (tree)

alg: \( S = \emptyset \)
for \( i = 1 \) to \( n \)
add next interval with earliest finish time that doesn't
intercept previous ones to \( S \).

proof: exchange argument

Let \( S \) be sol° returned by alg, \( S' \) be OPT sol°.
Let \( S_{i-1} \) denote \( S \) at start of ith iteration.

hyp: \( S_{i-1} \) can be extended to OPT sol°.
Assume it is true at start of loop \( i \).

\( S_i = \begin{cases} S_{i-1} & \text{if } I_i \text{ intersects interval in } S_{i-1} \\ S_{i-1} \cup \{I_i\} & \text{o/w} \end{cases} \)

\( \Rightarrow S_i \) can be extended to OPT sol°.
Minimum Spanning Forest

MST(G): connected weighted undirected graph G \rightarrow \text{minimum spanning tree of } G

MSF(G): connected weighted undirected graph G \rightarrow \text{a MSF of } G

\[ P:\text{ Prim's alg} \quad \boxed{(MST)} \]

1. Choose an arbitrary vertex v and add it to an empty tree T
2. Grow the tree by adding the vertex connected by an edge to T, such that that edge is smallest such edge

\[ O(IV^2) \]

Greedy works be of cut property: for a given cut C of a w. graph, the smallest edge in C is in every MST of it.

Kruskal's alg \quad \boxed{(MSF)}

1. Sort all edges by weight, create a forest F, where every node is a root.
2. Grow the forest by adding the smallest weighted edge that connects 2 trees in the forest (i.e. avoid cycles)

\[ O(IV^2) \]

Note [CLRS]: Greedy method yields optimal solution on several struct., one of which is called a Matroid (e.g. job sched - cf 2420), which exhibits optimal substructure & greedy choice property.
**Dynamic programming**

**Shortest path**

**Input:** \( G = (V, E) \), weighted directed graph and \( s, t \in V \)

**Output:** a shortest path \( s \to t \) (in all shortest paths starting at source)

**Dijkstra's alg:**

\[
\begin{align*}
\text{(Greedy)} & \quad \text{for } i = 1 \text{ to } n \\
& \quad d[i] = \infty \\
& \quad p[i] = \text{NIL} \\
& \quad d[s] = 0 \\
& \quad H = \text{Build Heap } (d) \\
\text{while } H \neq \emptyset \\
& \quad v = H.\text{extractMin}() \\
& \quad \text{for } u \in \text{Edgelist } [v] \\
& \quad \quad \text{if } d[u] > d[v] + w_{uv} \text{ then} \\
& \quad \quad \quad d[u] = d[v] + w_{uv} \\
& \quad \quad \quad p[u] = v \\
& \quad H.\text{decreaseKey} (u, d[u]) \\
\end{align*}
\]

**Relaxation**

**Shorkest path alg's**

\[
\begin{align*}
& \text{Floyd - all pairs - } O(V^3) \\
& \text{Bellman-Ford - single source - } O(VE) \\
& \text{Dijkstra - single source - } O(V + V \log V) \\
\end{align*}
\]

**Longest path**

**Simple path** - no repeated vertices

**No subproblem optimality** \( \Rightarrow \) brute force \( \Rightarrow \) NP-complete

**Relaxation** (it may seem strange, that the term relaxation is used) for an operation that tightens an upper bound

**[CLRS]** The process of relaxing an edge consists of testing whether we can improve the shortest path \( s \to t \) found so far by going through, if so update. The use of the term relaxation is historical.
Bellman-Ford alg:

(Dynamic prog)
(Edges can be negative)
Checking for negative cycles

for $i = 1$ to $m$
    $d[i] = \infty$
    $d[1] = 0$
    # predecessor list
    for each $(u, v)$ in $E$
        relax $(u, v)$ # see Dijkstra
    for each $(u, v)$ in $E$
        if $d[v] > d[u] + w_{uv}$ then FAIL

All pairs shortest paths

→ single-source shortest paths: predecessor subgraph is a tree
of shortest paths from source

→ all-pairs shortest paths: is a forest:

Two DP formulations (recommends):

min weight of any path in $G$ that contains
at most $m$ vertices

$\ell_{ij}^{(m)} = \begin{cases} 0 & \text{if } m = 0 \text{ } i = j; \\
\min_{k \leq m} \left( \ell_{ik}^{(m-1)} + w_{kj} \right) & \text{if } m \geq 1 \end{cases}$

weight of a shortest path from $i$ to $j$ s.t. all
intermediate vertices $(p \neq i, j)$
are in the set $\{1, \ldots, k\}$

$\ell_{ij}^{(k)} = \begin{cases} \omega_{ij} & \text{if } k = 0 \text{ saves time} \\
\min (\ell_{ij}^{(k-1)}, \ell_{ik}^{(k-1)} + d_{kj}) & \text{if } k \geq 1 \text{ more intuitive} \end{cases}$

Floyd-Warshall alg:

for $k = 1$ to $m$
    let $D^{(k)}$ be a new naiive # $D^{(k)} = (d_{ij}^{(k)})$
    for $i = 1$ to $m$
        for $j = 1$ to $m$
            $d_{ij}^{(k)} = \min (d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
Matrix multiplies (matrix-chain parenthesization)

E.g.

2 matrices $A_{m \times n}$, $B_{n \times m}$, $A_{m \times n}$ needs $m \times n \times m$ computations

3 matricies $A_{m \times n}$, $B_{n \times m}$, $C_{m \times n}$, with $A_{m \times n}$, $B_{n \times m}$, $C_{m \times n}$

\[
\begin{array}{c}
A(B(C)) = (AB)C \\
\end{array}
\]

with $A_{m \times n}$, $B_{n \times m}$, $C_{m \times n}$

1. $O(5500000)$
2. $O(10010000)$

Input: $M_1, \ldots, M_n$, $M_i \neq M_j$

Output: min. no. of comps

Let $m_{i,n}$ be OPT sol, then $m_{i,n} = \min_{1 \leq k \leq n-1} (m_{i,k} + m_{k+1,n} + c_{i,k+1})$

Use DP: keep the 2D array of $m_{i,j}$

Alg: for $i = 1$ to $n$ - 1

for $j = i$ to $n$ - 1

for $k = i$ to $j$ - 1

if $x = m_i,k + m_{k+1,j} + c_{i,k+1,j}$

$m_{i,j} = x$

Flow Networks

Input: $G$, dir. weight con.; $s, t \in V$

Output: max flow ($\Leftrightarrow$ min cut)

Def: a flow network is a graph $G = (V, E)$

1. such that every edge has a non-negative capacity $c(e,v) \geq 0$ (as a convention $0$ is when there is no edge);
2. such that all edges are directed and there is no self loops or recurrent connections ;
3. and such that $s$ and $t$ are distinguished as source and sink resp.

A flow is a function:

$f: E \rightarrow R^+$

$s f(e) \in [0, e.w]$ (i)

$\forall v \in V, v \neq s, v \neq t$ (ii)

$\forall e \in E, f(e) \in [0, e.w]$ (iii)

flow conservation (iii)

$\forall v \in V, v \neq s, t$ (iv)

capacity constraint (i)
Maximum Flow pb: determine maximum flow \( f^* \) w/ at least source & sink

- Any directed graph s.t. (i) holds can be reduced to a flow network:
  1. add parallel edge (s, t) if there are connections both ways, add an extra node \( s' \) or \( t' \).
  2. supersource/sink if there are multiple sources/sinks, add a single one that feeds in (\( s' \) or \( t' \)).

Ford-Fulkerson Method

- Residual networks

Def: The residual net of a flow net \( G \) with flow \( f \), denoted \( G_f \), is the graph induced by the residual capacity \( c_f \) of each edge of \( G \). The residual capacity is constructed by:
  1. \( c_f(u, v) = c(u, v) - f(u, v) \) (the capacity remaining unused by \( f \) and \( f \)' edge in the other direction) to allow an edge to cancel out some existing flow.
  2. \( c_f(u, v) = f(u, v) \) if \( f(u, v) \) is 0.

Def: Once given a flow \( f \), the residual net we can augment it with:

Def: A cut \((S, T)\) in flow net \( G \) is a cut such that \( s \notin V \). A cut \((S, T)\) in the residual net we allow across edges.

The net flow across the cut is \( \sum_{(u, v) \in E} (f(u, v) - f(v, u)) \). The capacity of a cut is just \( \sum_{(u, v) \in E} c(u, v) \), upper bound given.

Thm: Max-Flow Min-Cut: the max flow in \( G \) is equal to the cut \( c^* \) of \( G, \ s.t. \ \forall \text{cut} \( c \) of \( G \), \( c^* = \min \{c(\mathcal{E}) \ | \ \mathcal{E} \in \mathcal{E}\} \). If \( f \) is max \( f^* \) if \( G_f \) has no augmenting paths.

Ford-Fulkerson Method:

- For each \((u, v) \in E\) \( f = 0 \), while \( \exists \) path \( p \) from \( s \) to \( t \) in \( G_f \) while \( c_f(p) \) \( > 0 \) for each edge of \( p \), augmenting path - finding is not given.

- For each \((u, v) \in E\) \( f = 0 \), while \( \exists \) path \( p \) from \( s \) to \( t \) in \( G_f \) while \( c_f(p) \) \( > 0 \) for each edge of \( p \), augmenting path - finding is not given.
proof: consider a shortest path from s to t in the residual net.

1. Consider a shortest path from s to t in the residual net. Let this path be p and let the edge in p with the smallest capacity be (u, v).

2. If (u, v) is in p, decrease the capacity of (u, v) by one unit, and repeat step 1.

3. If (u, v) is not in p, then (v, u) is in p. Decrease the capacity of (v, u) by one unit, and repeat step 1.

4. Continue until there are no more paths from s to t in the residual net. The resulting path is then the desired path.

Proof: This algorithm correctly finds the minimum capacity path from s to t in the residual net. Each iteration of the algorithm either decreases the capacity of an edge in the residual net by one unit or removes an edge from the residual net. Thus, the algorithm must terminate after a finite number of iterations. The path returned by the algorithm is a shortest path because it minimizes the number of times the capacity of an edge is decreased.
Linear & Integer Programming

LP + variables \((x_i) \in \mathbb{R}\)
+ objective \(\sum c_i x_i\), \(c_i \in \mathbb{R}\)
+ constraints: \(\text{min/\text{max}}\)

IP \(\rightarrow\) NP-hard

Def: Feasible region: all values of \(x_i\)'s that satisfy every constraint.

+ empty \(\rightarrow\) infeasible LP
+ unbounded \(\rightarrow\) unbounded LP
+ bounded \(\rightarrow\) normal LP

Algs . simplex
 . other alg's

Applications: network flow
- Given network \(N = (V, E)\) capacities \((c(e))\), \(e \in E\)
- Create linear program with variables \(f_e, \forall e \in E\)
  \[\text{obj.}: \quad \sum f_e \quad \text{s.t.}\]
  \[c(e) \geq f_e \quad \forall e \in E\]
  \[f_e \geq 0 \quad \forall e \in E\]
- Formulating as LP:

- Max/min obj function
- Under linear constraints
- Non-strict inequalities

- Feasible region
- 2D convex polytope = z-polytope (i.e., convex polygon)

- Simplex = convex m-polytope (i.e., the convex hull of its m1 vertices)

- Simplex alg.: some form of gaussian elimination on as simplex. If the matrix of constraints respects certain properties (e.g., positive semi-definite) then the routine is poly, but expo in general

- Feasible region:
  - Empty ⇒ infeasible LP
  - Unbounded ⇒ if obj function "closes" the polytope then unbounded LP (otherwise infeasible)

- Approximation algorithms
  - (see cs2420)
    - Approx alg.: known approx ratio
    - Approx scheme: also takes E as input as input and thus
      - FPTAS (fully polynomial approximation scheme): poly in \( \frac{1}{\varepsilon} \) head off runtime for precision
    - Hardness of approximation: NP-complete plus have different possible approximations (I guess the polytope reduction itself hides this)

- Min vertex cover X max matching: have cst approx of 2.
- Knapsack \((0,1)\): has approx of \( (1+\varepsilon) \)
- TSP: no FPTAS! (\( \text{OPT}=\text{NP} \))

- Knapsack admits a pseudo-polynomial weak comp, alg. (polytime in the size \( \text{OPT} \) as an integer, \( \text{OPT} \) = \( \text{logarithm} \) of the numeric value
  - Weak comp alg. (polytime in the size \( \text{OPT} \) as an integer, \( \text{OPT} \) = \( \text{logarithm} \) of the numeric value
  - Strongly encoded in binary of input, but not in its representation as a bit string
  - No pseudo polynomial (a way to measure the real length of the input) that uses DP.
- Vertex cover example

1) Naive approx: pick an arbitrary edge \((u,v)\) from the graph and delete all edges incident to \(u\) or \(v\), repeat until the graph has no more edges (the vertices picked out will be a VC).

\(C^0\) provides a 2-approx alg.

proof: consider all the edges picked out, call that set \(A\). Then \(A\) is a maximal matching, thus \(|A| \leq 1|OPT|\).

Since we are always collecting 2 vertices, we also know that the size of the output is \(|C| = 2|A| \leq 2|OPT|\).

\[ \frac{|C|}{|OPT|} \leq 2 \]

if you are wondering, the minimum maximal matching (global optimum) is NP-complete.

2) LP approx: set up an equivalent LP and relax it.

\[ \text{solve the relaxed LP} \]

Create a cover by \(C = \{ v \in V \mid x_i \geq \frac{1}{2} \} \)

\(C^0\) also provides a 2-approx alg.