Greedy Algorithms

Interval scheduling: Job J(k) is an interval (sk, fk), k = 1…n. Two jobs are compatible if they are disjoint. Find a maximal compatible subset S.
Greedy approach: Consider jobs in some order (in this case, earliest finish time). Add each job to a running set S as long as it's compatible.
Stay-ahead argument: E.g. for interval scheduling, show that it "stays ahead" of any other compatible set with some number of jobs. (a) Show the r-th interval in S finishes as soon as possible.
Exchange argument: Prove at each step the subset S "is promising". Use induction on L(k): After the k-th step, there exists some optimal solution O such that J(i) ∈ S if and only if J(i) ∈ O.

Greedy Algorithms on Graphs

Single source shortest path: Given directed graph G = (V, E) and cost(e), find the shortest path from one vertex, say s, to all others.
Dijkstra's algorithm: For cost(e) ≥ 0, return s-v cost for every v ∈ V.
initialize S = {s}, d(s) = 0.
while V\S is not empty:
- Loop invariant: d(u) is the min s-u distance for u ∈ S.
- Define R = {v | v ∈ V\S and exists (u, v) ∈ E with u ∈ S}
- Greedily choose v ∈ R which minimizes

$$\pi(v) = \min_{e=(u,v):u\in S} (d(u) + w_e).$$

- Set d(v) = π(v) and add v to S.

return d
Proof: By induction on k = |S| from 1 to |V| using above loop invariant.
Minimum spanning tree (MST): Given a connected undirected graph G = (V, E) with real-valued edge weights w(e), an MST is a spanning tree whose sum of edge weights is minimized. A **spanning tree** is a subgraph H = (V, T) with T ⊆ E such that (V, T) is a tree.
Cycle: A path of the form a-b-c-d-…-z-a.
Simple cycle: No edge is repeated (in either direction) and no vertex is repeated other than the two endpoints a. (So a-b-a is not simple.)
Cut set: A cut is a subset of vertices S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.
Cut property: Let S be any subset of V, and let e be a minimum cost edge with in S and the other in V – S. Then some MST of G contains e.
Cycle property: Let C be any simple cycle, and let e be a maximum cost edge belonging to C. Then some MST of G does not contain e.
Strong cycle property: If e is the unique edge of maximum cost in the simple cycle C, then no MST contains e.
Prim's algorithm: Initialize S = any node, T = {}
while V\S is not empty.
- Let D be the cutset associated with S.
- Add a min weight edge e ∈ D to T (cut property)
- Add new explored node u to S.

return (V, T)
Kruskal's algorithm: Initialize T = {}.
for each e = (u, v) ∈ E sorted by increasing weight:
- Let S be the connected component in T containing u.
- Case 1: If v is the same CC, discard e (i.e. cycle properly)
- Case 2: Insert e = (u, v) into T (i.e. cut property using S)

return (V, T).
Reverse-Delete algorithm: Let T = E. Consider edges by descending weight. Delete edge e from T unless doing so would disconnect T.
Proving correctness: Try using a loop invariant L(S): the current set of edges is a subgraph of some MST T (supergraph for reverse-delete).
Some properties of MSTs:
- Any undirected graph with distinct edge weights has a unique MST.
- The max spanning tree **can** be computed by negating edge weights.
- Adding a constant to each edge weight does **not** change the MST.
- If all weights are the same, then every spanning tree is minimal.
- Adding an edge connecting two existing vertices of G creates a cycle.
- On any cycle in G, the cheapest edge is **not** necessarily in an MST.
- The shortest path between any two nodes in an undirected graph will not necessarily lie on some MST.

Divide and Conquer

Proofs: Using induction by cases.
Master Theorem: Any function that satisfies a recurrence of the form T(n) = a T(n/b) + Θ(n^d) has solutions in time (i) T(n) = Θ(n^d), if a < b^d; (ii) T(n) = Θ(n^d log n), if a = b^d; and (iii) T(n) = Θ(n^log$_b$ a), if a > b^d.
Karatsuba multiplication: For n-bit integers x and y, break into n/2 bit integers x = x$_1$x$_0$ and y = y$_1$y$_0$. If x = $2^{n/2}$ x$_1$ + x$_0$ and y = $2^{n/2}$ y$_1$ + y$_0$ then: xy = 2^n x$_1$y$_1$ + $2^{n/2}$((x$_1$ + x$_0$)(y$_1$ + y$_0$) – x1y1 – x0y0). Runs in O(n^1.585).
Matrix multiplication C = AB: Partition A and B into n/2-by-n/2 blocks. Compute 14 n/2-by-n/2 matrices via 10 matrix additions. Conquer by multiplying 7 matrices recursively, and combine 7 products into 4 terms using 8 matrix additions. T(n) = 7T(n/2) + Θ(n^2) = O(n^2.81).
Closest Pair: Compute separation line L such that ½ of points on each side. Let d = min(closestPair(RightHalf), closestPair(RightHalf)). Delete remaining points further than d from L and sort by y coordinates. Scan points by y. Compare distance from each to next 7 neighbours. If any of these distances is less than d, update d. Finally, return d.

Dynamic Programming

Informal algorithm design guidelines:
(i) There are only a polynomial number of subproblems.
(ii) The solution to the original problem can be easily computed from the solutions to the subproblems.
(iii) There is a natural ordering on subproblems from "smallest" to "largest" together with an easy-to-compute recurrence that allows one to determine the solution to a subproblem from the solutions to some number of smaller subproblems.
Knapsack: If w < w[i] then OPT(i, w) = OPT(i – 1, w). Otherwise:
OPT(i, w) = max(OPT(i – 1, w), w[i] + OPT(i – 1, w – w[i]))
Sequence alignment: Given strings x[1..m] and y[1..n], an alignment M is a set of ordered pairs x[i]-y[j] such that each item occurs in at most one pair and there are no crossings in the pairing. The pairs x[i]-y[j] and x[i']-y[j'] *cross* if i < i' but j > j'. The cost(M) is the sum of alpha(x, y) + sum(delta for x unmatched) + sum(delta for y unmatched).
Let M[i, j] = min cost to align prefixes x[1..i] and y[1..j]. Then:
M[i, j] = min(α[x[i], y[j]] + M[i – 1, j – 1], δ + M[i – 1, j], δ + M[i, j – 1])

Dynamic Programming on Graphs

Shortest path: Need the shortest s-t path, but cost may be negative.
OPT(i, v) = cost of cheapest v-t path P using ≤ i edges.
Either (1) P uses at most i – 1 edges, or (2) P uses exactly i edges. In this case if (v, w) is first edge in P and then P selects best w-t path using at most i – 1 edges, providing cost c(w, v) + OPT(i – 1, w).

$$OPT(i,v) = \begin{cases} 0 \text{ for } v = t, \infty \text{ otherwise}, & i = 0, \\ min\{OPT(i-1,v),\ min_{(v,w)\in E}\{OPT(i-1,w) + c_{vw}\}\} \end{cases}$$

Bellman-Ford(G, s, t) Push-Based efficient implementation:
for each node v ∈ V: set M[v] ← INF and successor[v] ← NIL
M[t] = 0
for i = 1 to n – 1:
for each node w with M[w] updated in previous iteration:
for each node v such that (v, w) ∈ E
if M[v] > M[w] + cost(v, w):
Update M[v] ← M[w] + cost(v, w) and successor[v] ← w
if no M[w] value changed in iteration i, stop
If no negative cycles, then M[n – 1, s] = cost of cheapest s-t path.
Complexity: O(mn) = O(|V||E|) time, O(m + n) memory.
Correctness: Prove using loop invariant L(i): after i rounds of updates, M[i, v] is no larger than the cost of cheapest v-t path using ≤ i edges and, if M[i, v] < INF then M[i, v] is the cost of some such path.
Absence of t-connected negative cycles: If OPT(n, v) = OPT(n–1, v) for all v, then no negative cycle in G includes a vertex having a path to t.
Presence of t-connected negative cycles: If OPT(n, v) < OPT(n–1, v) for some v, then exists a v-t path P with at most n edges and cost(P) = OPT(n, v). Moreover, P contains a negative t-connected cycle W.
Note: May fail to detect negative cycle that's not t-connected. We can fix this by adding a 0-cost edge from all nodes to t.
All pairs shortest path with Floyd-Warshall algorithm in O(|V|^3):
OPT(v, w, i) = min cost of any v-w path consisting of paths of the restricted form (v, x(j1), …, x(jp), w) where each jq ∈ {1, …, i}.

$$OPT(v,w,i) = \begin{cases} c_{v,w} & \text{if } i = 0 \\ min\{OPT(v,w,i-1), \\ \quad OPT(v,x(i),i-1) + OPT(x(i),w,i-1)\} & \text{for } i > 0. \end{cases}$$

Largest independent set problem:
OPT(w) = max size of independent set for the subtree rooted at w.
Case 1: w ∈ S. The children of w cannot be in S but grandchildren can.
Case 2: w not in S. Children of w can be in S.

$$OPT(w) = \max\left[1 + \sum_{v\in c(c(w))} OPT(v),\ \sum_{u\in c(w)} OPT(u)\right]$$

Network Flow

Flow network: Defined by a directed graph G = (V, E), capacities c(e) associated with each edge e, a single source node s ∈ V, and sink node t ∈ V. Other nodes are called *internal*. No edge ends at s nor leaves t.
Flow: An s-t flow is a function f mapping each edge e to a nonnegative real number. A flow f must satisfy the following properties:
(i) (Capacity) For each e E, we have 0 ≤ f(e) ≤ c(e).
(ii) (Conservation) For each node v other than s and t, we have:

$$\sum_{e \text{ into } v} f(e) = \sum_{e \text{ out of } v} f(e)$$

The value v(f) of a flow f is the total flow generated at the source (sum of f(e) across every e out of s).
Cut: An s-t cut is a partition (A, B) of V with s ∈ A and t ∈ B.
The **capacity** of a cut (A, B) is $cap(A, B) = \sum_{e \text{ out of } A} c(e)$.
The **net flow** Nf(A) across an s-t cut (A, B) is defined as the sum of flow on edges A to B, minus the sum on edges B to A.
Flow value lemma: For any flow f and s-t cut (A, B), v(f) = Nf(A).
Pf. $v(f) = \sum_{e\in out(s)} f(e) = \sum v \in A[\sum_{e\in out(v)} f(e) - \sum_{e\in in(v)} f(e)]$
$= \sum_{e,u\in A, v\in B}[f(e) - f(e)] + \sum_{e,u\in A, v\in B} f(e) - \sum_{e,v\in A, u\in B} f(e)$
Weak duality: For any flow f and s-t cut (A, B), v(f) ≤ cap(A, B).
Residual graph: Given a flow network G = (V, E) and a flow f on G, the residual graph Gf = (V, E') consists of the same nodes V, but E' has:
- *Forward edges:* for each e = (u, v) of G on which f(e) < c(e), include the same edge e = (u, v) but with a capacity of c(e) – f(e).
- *Backward edges:* for each e = (u, v) of G on which f(e) > 0, include a back-edge e' = (v, u) with a capacity of f(e).

The capacity cf(e) of an edge in Gf is called the *residual capacity*. Any simple s-t path in the residual graph is called an **augmenting path**.

$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

Augment(f, P, Gf, E):	**Ford-Fulkerson(G=(V,E), s, t):**
b ← bottleneck(P, Gf) foreach e ∈ P: if e ∈ E: f(e) ← f(e) + b else: f(e') ← f(e) – b return f	Init Gf ← G and f(e) ← 0 for e ∈ E. while exists augmenting path P in Gf: f ← Augment(f, P, Gf, E) Update Gf based on the new f return f

Here, bottleneck(P, Gf) is the minimum residual capacity on P in Gf. Note that Augment updates f on the original graph G. The residual graph Gf does not have associated flows, only capacities that update.
Time Complexity: O(mC) with m = |E| and C = cap({s}, V\{s}). This is pseudo-poly but can be poly by choosing augmenting paths carefully.
Theorem: If f is a feasible flow, then the following are all equivalent: (i) There exists a cut (A, B) such that v(f) = cap(A, B). (ii) The fllow f is a max flow. (iii) There is no augmenting path relative to f.
Augmenting path theorem: Flow f is a max-flow iff there are no augmenting paths. Proof: Follows from equivalence of (i) and (ii).
Max-Flow Min-Cut Theorem: The max-flow is equal to the capacity of the min-cut. Proof: Follows from (i) <=> (ii) along with weak duality.
Integrality theorem: If all capacities are integers, then there exists a max flow f for which every flow value f(e) is an integer. Proof: Since algorithm terminates this follows from the invariant.
Other properties:
- In some max-flows, there may be cycles carrying positive flow.
- There always exists a max-flow without cycles carrying positive flow.
- Unique min-cut does **not** necessarily mean unique max-flow.
- Distinct capacities does **not** necessarily imply unique max-flow.
- Multiplying all edge capacities by a positive number L leaves the minimum-cut unchanged.
- Adding a positive number L to all capacities does **not** necessarily increase the max flow by multiple of L.

Network Flow Applications

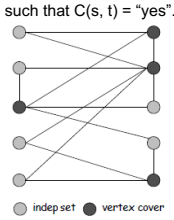
Bipartite matching: Input bipartite graph G = (L ∪ R, E) with e = (l, r). M ⊆ E is a matching if each vertex appears in at most one edge in M. Find a matching with a max number of edges.
Max flow formulation: Create directed graph G' = (L ∪ R {s, t}, E'). Direct all edges in E from L to R, and assign capacity c(e) = INF. Add unit capacity edges from s to each node in L, from each node in R to t.
Max edge-disjoint s-t paths: Assign unit capacity to each edge.
Min edge deletes to disconnect s-t: Equal to max edge-disjoint paths.
Circulation with demands: Given directed graph G = (V, E), edge capacities c(e), and node supply-demand d(v) (here, d(v) < 0 is supply, d(v) = 0 is transshipment, and d(v) > 0 is demand). A *circulation* is a function f(e) that satisfies:

For each e ∈ E: $0 \le f(e) \le c(e)$ (capacity)
For each v ∈ V: $\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ (conservation)

Circulation problem: Does there exist a circulation?
Necessary condition: D = sum of sinks d(v) = sum of sources –d(v).
Max flow formulation: Let G' = (V', E'). Add new source and sink to V. For each v with d(v) < 0, add edge (s, v) with capacity –d(v) to E'. For each v with d(v) > 0, add edge (v, t) with capacity d(v) to E'. Then, G has circulation if and only if G' has max flow value D.

Decision Problems

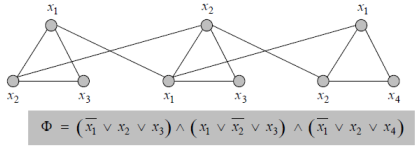
Definition: Each decision problem is represented as a set X of strings deemed to be "positives". An instance is a string s. Algorithm A "solves problem X" means: A(s) = "yes" if and only if s ∈ X.
Poly-time certifier: An poly-time algorithm C(s, t) for problem X(s) where there exists a polynomial pc(|s|) such that, for every input string s s ∈ X iff there is a certificate t with |t| ≤ pc(|s|), such that C(s, t) = "yes".
INDEPENDENT-SET, IS(G, k): Given a graph G = (V, E) and an integer k, is there a subset of vertices S ⊆ V such that |S| ≥ k and for each edge at most one of its endpoints is in S?
VERTEX-COVER, VC(G, k): Given a graph G = (V, E) and an integer k, is there a subset of vertices S ⊆ V such that |S| ≤ k, and for each edge, at least one of its endpoints is in S?



SET-COVER, SC(U, S, k): Given a set U of elements, a collection S = {S1, S2, …, Sm} of subsets of U, and an integer k, does there exist a collection of ≤ k of these sets whose union is equal to U?
SAT: Does a CNF formula Φ have a satisfying truth assignment?
3-SAT: SAT but each clause contains exactly 3 literals (each corresponding to a different variable).
CIRCUIT-SAT: Given a combinational circuit built with AND, OR, and NOT gates, can the circuit inputs be set so that the output is 1?
HAM-CYCLE: Given an undirected graph G = (V, E), does there exist a simple cycle C that visits every node?
SET-PACK SP(U, S, k): Given a set U of elements, a set S = {S1, S2, …, Sm} of subsets of U, and an integer k, does there exist a collection C ⊆ S of size ≤ k such that no two distinct elements Si, Sj ∈ C intersect?
CLIQUE(G, k): Given an undirected graph G and integer k, does there exist clique K of G with |K| ≥ k? A clique is a subset of vertices K ⊆ V such that for every pair of distinct vertices u, v ∈ K, edge (u, v) is in E.
PRIMES: X = {2, 3, 5, 7, 11, 13, 17, …}
FACTOR: Given integers x and y, does x have a non-trivial factor less than y? E.g. for integers x > 1, FACTOR(x, x) = ¬PRIMES(x). The (non-decision) problem FACTORIZE(x) returns the prime factorization of x.
Partial3DM: Does there exist a set C of triples from X×Y×Z of size ≥ k where no two distinct triples of T have any elements in common?
Perfect3DM: Partial3DM but no k is provided. Exists set |C| = n?
3-SAT ≤p Perfect3DM ≤p Partial3DM which is NP-hard & NP-complete.
Partial2DM ≤p SET-PACK. Partial2DM is in P, solved using bipartite.

Polynomial Reduction

Reductions: Can arbitrary instances of problem X be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to an "oracle" (black box) that solves problem Y? If yes, then we write X ≤p Y and read this as "X is polynomial-time reducible to Y" or "Y is at least as hard as X (with respect to polynomial time)."
Reduction by simple equivalence: E.g. Show IS(G, k) ≡p VC(G, k).
Proof: Show S is an independent set <=> V – S is a vertex cover.
Reduction from special case to general case: E.g. Show VC ≤p SC.
Proof: We wish to solve VC(G, k) by encoding the input as one or more SC(U, S, j) problems. Use solutions to SC(.) problems to solve VC(G, k)
Reduction via "gadgets": E.g. 3-SAT ≤p INDEPENDENT-SET.
Proof: Given an instance Φ of 3-SAT, construct an instance (G, k) of IS where the independent set size |S| ≥ k = |Φ| iff Φ is satisfiable.



$\Phi = (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$

Complexity Classes

P: Decision problems for which there exists a poly-time algorithm.
EXP: Decision problems for which there exists an exp-time algorithm.
NP: Decision problems for which there is a poly-time certifier.
NP-complete: A problem Y ∈ NP such that for every X ∈ NP, X ≤p Y. That is, every problem in NP reduces to Y in poly-time.
NP-hard: A problem Y such that for every X ∈ NP, X ≤p Y (e.g.: searchIndepSet(G) as is not a decision problem, CNF-TAUTOLOGY since SAT ≤p CNF-TAUTOLOGY.
Proving NP-ness of problem X:
1. Note X is a decision problem and define |s| to be the input size.
2. Define certificate t corresponding to each instance s of X, and show that some polynomial-time certifier C(s, t) = "yes" iff s ∈ X.
3. Show that certificate length is bound by a polynomial pc(|s|).
Proving NP-completeness of problem Y:
1. Show Y ∈ NP. 2. Choose some X ∈ NP-complete 3. Show X ≤p Y.

Complement of Decision Problems

Definition: The complement of a decision problem X (denoted Xbar), is the set of strings not in X. An algorithm B "solves problem Xbar" means B(s) = "yes" if and only if s ∉ X.

Poly-time disqualifier: A poly-time algorithm DQ(s, t) together with a polynomial pd(|s|) such that for every input string s, s ∉ X iff there exists a string t with |t| ≤ pd(|s|) and DQ(s, t) = "yes".

Theorem: if $A \leq_p B$ then $\overline{A} \leq_p \overline{B}$.

Proof: Suppose A reduces to B. Let x ∈ A, then we have a transformed instance of x, call it f(x) ∈ B. Note that x and f(x) are "yes" instances of their respective problems. If x were a no-instance of A, then f(x) must also be a no-instance of B, hence the reduction. To summarize:

$$x \in A \iff f(x) \in B \text{ implies that } x \in \bar{A} \iff f(x) \in \bar{B}.$$

co-NP: Decision problems X for which the complement of X is in NP. Or equivalently, decision problems X with a **poly-time disqualifier**.
E.g. CNF-TAUTOLOGY: Given a CNF formula Φ, is Φ unsatisfiable? That is, is ¬Φ a tautology? This is NP-hard, not known to be in NP.
Does P = NP? Does NP = co-NP? Consensus opinion: No for both.
Theorem: If NP ≠ co-NP, then P ≠ NP.
Proof: We will prove the contrapositive. Assume P = NP.

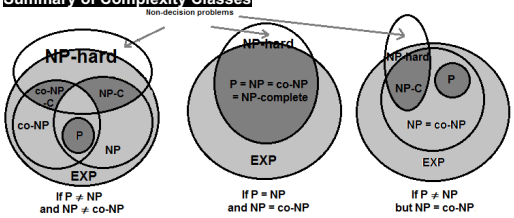
$$X \in \mathcal{NP} \Longrightarrow X \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow X \in \text{co-}\mathcal{NP}$$

and

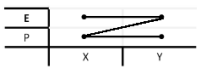
$$X \in \text{co-}\mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{NP} \Longrightarrow \overline{X} \in \mathcal{P} \Longrightarrow X \in \mathcal{P} \Longrightarrow X \in \mathcal{NP}.$$

Thus NP = co-NP. Thus NP is closed under complementation.
Some observation: PRIMES is in P (AKS primality test, 2002), NP ("Pratt certificate", 1975), and co-NP (a factor p where mod(s, p) = 0 is a disqualifier). FACTOR is in NP and co-NP, but not believed to be in P. FACTOR ≡p FACTORIZE (reduction by binary search).

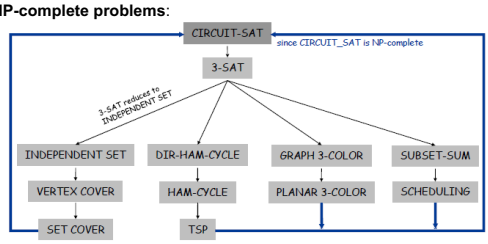
Summary of Complexity Classes



If we have shown X ≤p Y. Then the possibilities for their classifications are depicted in the following diagram. If we order a chain of problems left to right by least to most "difficult", then lines are horizontal or up-right.



NP-complete problems:



Some relationships between problems:
3-SAT ≤p INDEPENDENT-SET ≡p VERTEX-COVER ≤p SET-COVER
INDEPENDENT-SET ≡p SET-PACK ≡p CLIQUE
Proof CIRCUIT-SAT is NP-complete: Any yes/no algorithm accepting n fixed bits can be represented by a circuit K. If it runs in poly-time, then the circuit is poly-size. If X is NP, then it has poly-time certificate C(s, t) and poly length pc(|s|). View C(s, t) as an algorithm on |s| + pc(|s|) bits and convert it to a poly-sized circuit K. First |s| bits are hard-coded with s, remaining bits are t. Then K is satisfiable iff C(s, t) = yes.
Proof INDEPENDENT-SET ≡p CLIQUE: Suppose G has independent set S of size k, then for every distinct u, v ∈ S, edge (u, v) is not in G. Then, e is an edge in the complement graph Gc (graph G but with all and only edges that are not in E). Thus S is a clique. Similarly, it follows that if S is a clique in Gc, then S is an independent set in G.

Linear Programming

Defn. Consider the linear programming problem (std form):

maximize $c^T x$
subject to $A x \leq b$ and $x \geq 0$, (1)

The dual of this LP problem is the LP minimization problem:

minimize $y^T b$
subject to $y^T A \geq c^T$ and $y \geq 0$. (2)

These two LP problems are said to be duals of each other.

In (1), **x** is an n-vector, **A** is m-by-n, **b** is an m-vector, **c** is an n-vector.
In (2), y is an m-vector, **A** is m-by-n, **b** is an n-vector, **c** is an m-vector.
Alternative equivalent forms: Given an LP in standard (A, b, c) form:

- minimize –(c^T) is equivalent to maximizing (c^T) x
- Constraint (A^T)x ≤ b equivalent to –(A^T)x ≥ –b.
- Constraint (A^T)x = b iff (A^T)x ≤ b and –(A^T)x ≤ –b.
- Ax ≤ b iff Ax + z = b and z ≥ 0, with "slack variables" z1, …, zm.
- min(|e|) can be rewritten as min((e+) + (e-)) with the constraint e = (e+) + (e-) and the non-negativity constraint e+, e- ≥ 0.

Feasible set: F = { **x** | **Ax ≤ b** and **x ≥ 0** } could be empty (no solution).
Note: F is a convex polytope, i.e. a region of R^n defined by the intersection of finitely many half-spaces. F is convex, meaning that for every u, v ∈ F and s ∈ [0, 1], we have su + (1 – s)v ∈ F.
Theorem: A linear program where **c** ≠ 0, either:
1. Has no maximal solution, in which case either:
   (a) the feasible set F is empty, or
   (b) the objective (c^T)x is unbounded (and F is unbounded)
2. Has a solution **x***, taken to be a vertex of the polytope F (although there may be non-vertex solutions x ∈ F where (c^T) x = (c^T) x*).

Mutual Bound Theorem: If x is a feasible solution of LP (1) and y is a feasible solution of LP (2), then $c^T x \leq y^T A x \leq y^T b$.

Complementary slackness: Given feasible solutions x and y of LP and dual LP, then x and y are optimal iff: (sum(Aij xj for j=1..n) < bi ⇒ yi = 0) and (sum(yi Aij for i=1..m) > cj ⇒ xj = 0).
LP duality theorem: Consider a standard LP problem:
The feasible set F for (1) is not empty and $c^T x$ is bounded above for $x \in F$ iff the corresponding dual LP (2) (above) has a non-empty feasible set $G = \{ y \mid y^T A \geq c^T \text{ and } y \geq 0 \}$ and $y^T b$ is bounded below for $y \in G$.
Moreover, in this case, $\max \{ c^T x \mid x \in F \} = \min \{ y^T b \mid y \in G \}$.

Vertex of LP and dual LP:
Let $t = \{t_1, t_2, \ldots, t_m\}$ be a selection of m columns of (3), $1 \leq t_i \leq m+n$.
Define E(t) to be the m x m matrix formed from the t-columns of D, and $e^T(t)$ the (1 x m)-vector formed from the same columns of d^T.
A point $v \in \mathbb{R}^m$ is a vertex of the feasible set (3) iff there exists an t such that E(t) is nonsingular, $v^T = e^T(t)[E(t)]^{-1}$, and v satisfies (3).
Define the m+n dimensional binary valued indicator vector δ(s) where $\delta_j = 1$ if $j \in s$, and $\delta_j = 0$ otherwise. Define δ(t) similarly.

$$\delta(s) = (\alpha_1, \alpha_2, \ldots, \alpha_m, \quad \alpha_{m+1}, \ldots, \alpha_{m+n}),$$
$$\delta(t) = (\beta_1, \ldots, \beta_n, \quad \beta_{n+1}, \beta_{n+2}, \ldots, \beta_{n+m}).$$

Vertex of LP: If the j^{th} coefficient of δ(s) is one (i.e., $[\delta(s)]_j = 1$) then the j^{th} row below is an equality for vertex x:

$$Px \equiv \begin{pmatrix} A \\ -I \end{pmatrix} x \leq p \equiv \begin{pmatrix} b \\ 0 \end{pmatrix}.$$

Vertex of Dual LP: If the i^{th} coefficient of δ(t) is one (i.e., $[\delta(t)]_i = 1$) then the i^{th} column below is an equality for vertex y:

$$y^T D \equiv y^T \begin{pmatrix} A & I \end{pmatrix} \geq d^T \equiv \begin{pmatrix} c^T & 0^T \end{pmatrix}. \qquad (3)$$

Simplex method: Given LP in standard form (A, b, c). Let v = v(s) be a feasible vertex, where s = {s1, …, sn} denotes a set of n "selected" rows of Pv ≤ p, such that Q(s)v = q(s) and Q(s) is non-singular. Then,
while True (modulo non-cycling conditions):
  Consider each neighbour s' of s (i.e. s' and s differ by only 1 element)
  Choose an edge v(s) to v(s') such that the objective increases.
  If there is no such edge, v(s) is a solution. Stop.
  If there is an edge leaving v(s) on which the objective is unbounded, then there is no solution to this LP. Stop.
  Set s ← s', and v' ← v'(s').
end while
The v(s) to v(s') step is called **pivoting**. One row of Pv ≤ p is dropped from s and replaced by another row to form s'. The selection of a pivot is guaranteed to not decrease the objective function. If some care is taken to avoid cycling, then Simplex is guaranteed to converge to a solution after finitely many pivot steps. Each pivot costs O((m + n)n) ops if implemented efficiently. Worst case, may visit exponentially many vertices in contrived cases (e.g. # of choices for s, (n+m) choose n).
Initial Feasible Vertex: Given (A, b, c), consider the **startup LP** maximize –z1 – … – zm (i.e. minimize z1 + … + zm) subject to **x, z ≥ 0** and **Ax – z ≤ b**. For this, use initial guess x = 0, $z = b^-$ where $b^-_k = -b_k$ and 0 otherwise. This startup has a solution (x_0, 0) (with z = 0 and the objective function equal to 0) iff x0 is a feasible solution of the original LP. Simplex will return a feasible vertex x_0 on startup as long as original feasible set F is nonempty.
Max-flow as LP: For each edge (u, v), add an LP variable x(u, v) to represents the flow from u to v. The first |E| rows of A consists of flow capacity constraints x(e) ≤ cap(e). The next |V| rows of A consists of flow conservation constraints. The row for node v has the restriction sum(x(e) for e entering u) = sum(x(e) for e leaving u). The objective function is to maximize sum(x(e) for e entering the sink).
Integer LP: Only search for integer solutions. This is NP-hard. We can use an LP relaxation by converting the constraints x(i) ∈ {0, 1} into a pair of linear constraints 0 ≤ x(i) ≤ 1. Then, rounding is used to obtain some info. The approximation ratio is usually based on the integrality gap, defined IG = Mint / Mfrac = round minimum / real minimum.

Approximation Algorithms

We consider sacrificing optimality but attempt to preserve a guarantee.
ρ-approximation algorithm:
- Guaranteed to solve arbitrary instances of the problem in poly-time
- Finds solution within ratio ρ of the true optimum.

Let C be the computed value and C* be the optimal, then:
- For a maximization problem, C ≥ C* / ρ, with ρ ≥ 1.
- For a maximization problem, C ≤ C* / ρ, with ρ ≥ 1.

i.e. for either maximization or minimization: 1 ≤ max(C*/C, C/C*) ≤ ρ.
WEIGHTED-VERTEX-COVER: Vertex i now has weight w(i). Find min-weight subset S such that each edge is incident to some vertex in S. Can use LP + rounding or pricing method (edge e pays price p(e) >= 0 to use vertex i and edges incident to vertex i pays <= w(i) in total).
LOAD-BALANCE: Minimize the makespan = min L(i) (where L(i) = sum(ti across the jobs J(i) assigned to machine i)) of a schedule (NP-hard: SUBSET-SUM ≤p NUM-PARTITION ≤p LOAD-BALANCE). Greedy by assigning to the smallest-load machine is a 2-approximation.
Pf. Consider load L_i of bottleneck machine i.
- Let j be last job scheduled on machine i.
- When job j assigned to machine i, i had smallest load. Its load before assignment is $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ for all $1 \leq k \leq m$.
- Sum inequalities over all k and divide by m:

$$L_i - t_j \leq \frac{1}{m}\sum_k L_k = \frac{1}{m}\sum_i t_i \leq L^* \quad \leftarrow \text{Lemma 2}$$

- Now $L = L_i = (L_i - t_j) + t_j \leq 2L^*$. ($L_i - t_j \leq L^*$, by Lemma 1; $t_j \leq L^*$, by Lemma 2)

Lemma 1. optimal makespan $L^* \geq \max_j t_j$.
Lemma 2. optimal makespan $L^* \geq \frac{1}{m}\sum_j t_j$.

KNAPSACK: Given a finite set X, a target value V, nonnegative values v(i), a weight limit W, and nonnegative weights w(i), does there exist a subset S ⊆ X such that sum(v(i)) >= V and sum(w(i)) <= W?
KNAPSACK is NP-complete and FPTAS using the following algorithm:
Round up all values: $\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta$, $\hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$
v_{max} = largest value in original instance
ε = precision θ = scaling factor = ε v_{max} / n
Running time. $O(n^3 / \varepsilon)$.
Optimal sol to prob with $\bar{v}$ or $\hat{v}$ are equivalent.
Theorem. If S is solution found by our algorithm and S* is any other feasible solution then $(1+\varepsilon)\sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$

Knapsack FPTAS Proof:
Let S* be any feasible solution satisfying weight constraint.

$$\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} \bar{v}_i \leq \sum_{i \in S} \bar{v}_i \leq \sum_{i \in S} (v_i + \theta)$$

always round up; solve rounded instance optimally; never round up by more than θ

$$\leq \sum_{i \in S} v_i + n\theta \leq (1+\varepsilon)\sum_{i \in S} v_i$$

DP alg can take v_{max}

$|S| \leq n$; $n\theta = \varepsilon v_{max}$, $v_{max} \leq \Sigma_{i \in S} v_i$

Hierarchy of approximability:

- There can be no ρ-approximation algorithms (assuming P ≠ NP) (e.g. TSP on a weighted graph).
- There is a ρ(n)-approximation algorithm, where ρ(n) grows with the input size n (e.g. Set-Cover, ρ(n) = Ω(log n)).
- There is a ρ-approximation algorithm, with ρ > 1 a constant (e.g. Vertex-Cover, Weighted-Vertex-Cover).
- Polynomial-time approximation scheme (PTAS): For any ε > 0, there is a (1 + ε)-approximation algorithm (e.g. Load Balancing and Euclidean TSP). Produces an arbitrarily high-quality solution by trading accuracy for time. Runtime might grow quickly with 1/ε, e.g. O(n^(1/ε)).
- Fully polynomial-time approximation scheme (FPTAS): For ε > 0, there is a (1 + ε)-approximation algorithm whose runtime is polynomial in both size of the input, n, and 1/ε (e.g. Knapsack)

Miscellaneous Solutions to Problems

Updates to a Max Flow Problem. Suppose $G = (V, E, c)$ is a standard network flow problem where all the capacities $c(e)$ are positive integers, and f is an integer-valued maximum flow. Suppose $e_0 \in E$ is a saturated edge (i.e., $f(e_0) = c(e_0)$).

(8)
(a) **Increase the capacity on e_0 by one.** Consider $G' = (V, E, c')$ where $c'(e) = c(e)$ for all $e \in E \setminus \{e_0\}$, and $c'(e_0) = c(e_0) + 1$. Given f (as above) and the residual graph G_f, describe an efficient algorithm for updating the maximum flow f to form a maximum flow f' for G'. What is the run-time for your update algorithm? Explain why the updated flow f' is maximal for G'.

(b) **Decrease the capacity on e_0 by one.** Consider $G' = (V, E, c')$ where $c'(e) = c(e)$ for all $e \in E \setminus \{e_0\}$, and $c'(e_0) = c(e_0) - 1$. Given f and the residual graph G_f, describe an efficient algorithm for updating f to form a **feasible flow** f' for G' with value $v(f') = v(f) - 1$. Note f is not a feasible flow on G' since it $f(e_0) = c'(e_0) + 1$. What is the run-time for your update algorithm?

Note f is an integer-valued feasible flow on the updated graph G' (since we only changed G by increasing one edge capacity). Let G'_f be the corresponding residual graph. Use breadth first search (BFS) to see if there exists an augmenting path in G'_f. Note that, since there is no such path in the residual graph G_f for the original problem (because f is maximal for that graph), there will be an augmenting path P **only if** it contains the edge $e_0 = (u, v)$ in the forward direction.

If there is no such path P then f is a maximal flow for G' (by the Lemma in the proof of the Max-Flow Min-Cut Theorem).

Otherwise, e_0 is on the path P. The residual capacity of e_0 is $c'_f(e_0) = c'(e_0) - f(e_0) = 1$. Note, since we used BFS, the path is a simple path and no edge appears twice or more on P. Therefore the bottleneck for P is one (it cannot be less than one since we have integer-valued flows and capacities). We can then push one more unit of flow on this augmenting path (note that, to know we are able to do this, we are using the fact that P is a simple path). Finally, this flow f' must be maximal because: a) we know the minimum cut capacity for the graph G is $C = v(f)$; b) $v(f') = v(f) + 1$; and c) the minimum cut capacity for the graph G' can be no bigger that $C + 1$ (because we have only added +1 to a single edge capacity). But with the above augmentation we have $v(f') = C + 1$, so this must be both the min cut capacity on G' and the maximum possible flow value.

The runtime for this algorithm is dominated by the BFS step, which runs in $O(|E|)$ time. The flow augmentation runs in $O(|V|)$ time.

1. Find the dual to the following standard minimum problem. Find y_1, y_2 and y_3 to minimize $y_1 + 2y_2 + y_3$, subject to the constraints, $y_i \geq 0$ for all i, and

$$\begin{aligned} y_1 - 2y_2 + y_3 &\geq 2 \\ -y_1 + y_2 + y_3 &\geq 4 \\ 2y_1 + y_3 &\geq 6 \\ y_1 + y_2 + y_3 &\geq 2. \end{aligned}$$

2. Show $(y_1, y_2, y_3) = (2/3, 0, 14/3)$ is optimal for this problem, and that $(x_1, x_2, x_3, x_4) = (0, 1/3, 2/3, 0)$ is optimal for the dual.

2. We check that $\boldsymbol{y} = (\frac{2}{3}, 0, \frac{14}{3})$ and $\boldsymbol{x} = (0, \frac{1}{3}, \frac{2}{3}, 0)$ are feasible for their respective problems, and that they have the same value. Clearly, $\boldsymbol{y} \geq 0$ and $\boldsymbol{x} \geq 0$. Substituting $\boldsymbol{y}$ into the main constraints of Exercise 1, we find $\frac{2}{3} + \frac{14}{3} = \frac{16}{3} \geq 2$, $-\frac{2}{3} + \frac{14}{3} = 4 \geq 4$, $\frac{4}{3} + \frac{14}{3} = 6 \geq 6$, and $\frac{2}{3} + \frac{14}{3} = \frac{16}{3} \geq 2$, so $\boldsymbol{y}$ is feasible. Similarly substituting $\boldsymbol{x}$ into the main constraints of the solution of Exercise 1, we find $-\frac{1}{3} + \frac{4}{3} = 1 \leq 1$, $\frac{1}{3} \leq 2$, and $\frac{1}{3} + \frac{2}{3} = 2 \leq 2$, so $\boldsymbol{x}$ is feasible. The value of $\boldsymbol{y}$ is $\frac{2}{3} + \frac{14}{3} = \frac{16}{3}$, and the value of $\boldsymbol{x}$ is $\frac{1}{3} + \frac{2}{3} = \frac{16}{3}$. Since these are equal, both are optimal.

TutApproxQ3. Given N integers L = [a1, …, an] and positive integer C. Find a subset S of {1, …, N} such that T(S) = sum(ai, for i ∈ S) ≤ C.
(a) Show greedy by taking list values in order is not a ρ-approximation:
Proof: Let ρ > 1 be a given positive integer. Consider the input set L = [1, ρ + 1], with the upper bound C = ρ + 1. Then the Prof. Jot's algorithm returns S = {1}, for which T(S) = 1, while the optimum solution is S* = {2}, for which T(S*) = ρ + 1. Therefore

$$\frac{T(S^*)}{T(S)} = \frac{\rho + 1}{1} > \rho,$$

so this is not a ρ-approximation for any ρ, since ρ > 1 was arbitrary.
(b) Show that this modified version is a ρ-approximation?
Proof: Given that ai ≤ C for all i, there are now 2 general case:
1) sum(ai) = A ≤ C; and
2) sum(ai) = A > C.
In case 1 algorithm produces the optimum solution. In case 2, the algorithm above produces a set S such that T(S) ≥ C/2. But note for any optimal solution S*, it follows that T(S*) ≤ C. Therefore T(S) ≥ 1/2 T(S*), and so the algorithm is a 2-approximation.

```
Initialize S ← { }, T = 0
For i = 1, 2, ..., N:
    If a_i ≤ C:
        If T + a_i ≤ C: S ← S ∪ {i}
                        T ← T + a_i
    Else:
        If T < C/2: S ← {i}
        break
End for
return S
```

Reducing weighted 3-SAT to ILP: Each true clause gains weight w(k). Solution: Make the first row of A equal to [a1, a2, a3, 0] where a(i) = –1 if xi appears in the clause as xi and ai appears as ¬xi. Add another row (–a1, –a2, –a3, –*l*) where *l* is 1 if it is satisfied given y1, y2, y3 as input, 0 otherwise. Let K denote the number of literals negated in the clause and b = (–1 + K, –1 + K, 2 – K). Let c = (0, 0, 0, w1). E.g. The CNF clause (x1, x2, ¬x3) becomes the constraint y1 + y2 + (1 – y3) ≥ 0.