CS 6515 Introduction to Graduate Algorithm Cheat Sheets

Jie Wu, Jack

Fall 2023

1 Dynamic Programming: FIB - LIS - LCS

1.1 Longest Increasing Subsequence

input: n numbers $a_1, a_2, \ldots, a_n$

goal: find the longest increasing subsequence in $a_1, a_2, \ldots, a_n$

define the sub-problem in words: let $L[i]$ be the length of LIS on $a_1, a_2, \ldots, a_i$ that ends at $a_i \Rightarrow$ indexed by prefix with ending restriction

state the recursive solution: if $a_j \geq a_i$ then a_i cannot be added to any increasing sequence ending at a_j, otherwise it can be added thus we have

$$L[i] = 1 + \max_{1 \leq j < i} \{L[j] : a_j < a_i\}$$

Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^2)$.

1.2 Longest Common Subsequence

input: two strings $X = x_1 x_2 \cdots x_n$ and $Y = y_1 y_2 \cdots y_n$ (for simplicity we assume they have the same length)

goal: find the length of the longest string which is a subsequence of both X and Y

define the sub-problem in words: let $L[i, j]$ be the length of LCS in $x_1 x_2 \cdots x_i$, $y_1 y_2 \cdots y_j \Rightarrow$ indexed by double-prefix

state the recursive solution: first the base cases: $L[i, 0] = 0$ and $L[0, j] = 0$
if $x_i \neq y_j$ then $L[i, j]$ excludes x_i or y_i or both, we don't need to consider dropping both because we can drop one by one, hence we are left with only two cases: dropping x_i or dropping y_j
if $x_i = y_j$ then $L[i, j]$ includes x_i or y_i or both, actually it can be simplified to one case, because any LCS we obtain by matching x_i/y_j to some earlier y/x element can be obtained by matching x_i/y_j to y_j/x_i thus we just need to consider including both
to summarize

$$L[i, j] = \begin{cases} \max\{L[i-1, j], L[i, j-1]\} & x_i \neq y_j \\ 1 + L[i-1, j-1] & x_i = y_j \end{cases}$$

Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^2)$.

2 Dynamic Programming: Knapsack - Chain Multiply

2.1 Knapsack Problem

input: n objects with integer weights $w_1, w_2, \ldots, w_n$ & integer values $v_1, v_2, \ldots, v_n$, total capacity B

goal: find a subset S of the n object such that

- the total weight is within capacity (i.e. fit in the backpack) i.e. $\sum_{i \in S} w_i \leq B$
- the total value is maximized i.e. $\max \sum_{i \in S} v_i$

2.1.1 Knapsack without repetition

define the sub-problem in words: let $K[i, b]$ be the maximum value obtainable from a subset of the first i objects with total weight $\leq b \Rightarrow$ indexed by available objects and capacity

state the recursive solution: if $w_i \leq b$ then we can add the ith object and $K[i, b] = \max\{v_i + K[i-1, b-w_i], K[i-1, b]\}$; if $w_i > b$ then we cannot add the ith object and $K[i, b] = K[i-1, b]$

Since we have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(nB)$.

2.1.2 Knapsack with repetition

define the sub-problem in words: let $K[b]$ be the maximum value obtainable from all n objects with total weight $\leq b \Rightarrow$ indexed by available capacity

state the recursive solution: we will try all the possibilities for the last object to add, i.e. $K[b] = \max_i\{v_i + K[b - w_i] : w_i \leq b\}$

Because we still have a nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is still $O(nB)$.

2.2 Chain Matrix Multiply

input: n matrices $A_1, A_2, \ldots, A_n$ where A_i is of size $m_{i-1} \times m_i$

goal: find the minimum cost of computing their product $A_1 \times A_2 \times \cdots \times A_n$

define the sub-problem in words: let $C[i, j]$ be the minimum cost of computing $A_i \times A_{i+1} \times \cdots \times A_j$ where $1 \leq i \leq j \leq n \Rightarrow$ indexed by substring

state the recursive solution: the base case is $C_{ii} = 0$
we try all possible l splits of $A_i \times A_{i+1} \times \cdots \times A_j$ where $1 \leq i \leq j \leq n$ and take the minimum, i.e. $C[i, j] = \min_l\{C[i, j] + C[l+1, j] + m_{i-1}m_l m_j : i \leq l \leq j-1\}$ where $m_{i-1}m_l m_j$ is the cost of multiplying $A_i \times A_{i+1} \times \cdots \times A_l$ and $A_{l+1} \times A_{l+2} \times \cdots \times A_j$

Since we have a 3-nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^3)$.

3 Dynamic Programming: Shortest Paths

3.1 Shortest Paths

3.1.1 Single-source shortest paths

input: a directed graph $\vec{G}$ with edge weights $w(e)$, a start vertex $s \in V$, assuming there are no negative weight cycles

goal: find the shortest path $\mathcal{P}$ from s to z visiting every vertex at most once thus $|\mathcal{P}| \leq$ edges

define the sub-problem in words: let $D[i, z]$ be the length of the shortest path from s to z using $\leq i$ edges, the final solution is $D[|V| - 1, z]$ $\Rightarrow$ indexed by the max number of edges

state the recursive solution: base case is $D[0, s] = 0$ while $D[0, z] = \infty \; \forall z \neq s$
use the shortest path from s to the penultimate vertex y with $i - 1$ edges, and consider $\widetilde{D}[i, z] = \min_{y:\vec{yz}\in E}\{\widetilde{D}[i-1, y] + w(y, z)\}$ which is the length of the shortest path from s to z using exactly i edges, then we have $D[i, z] = \min\{D[i-1, z], \widetilde{D}[i, z]\} = \min\{D[i-1, z] + \min_{y:\vec{yz}\in E}\{D[i-1, y] + w(y, z)\}\}$ because the shortest path from s to z uses either at most $i - 1$ edges or exactly i edges

Since the nested for-loop for $z \in V$ and for all $\vec{yz} \in E$ combined go over all edges of the graph exactly once, the total running time of this Bellman-Ford algorithm is $O(|V||E|)$.

3.1.2 All-pairs shortest path

input: a directed graph $\vec{G}$ with edge weights $w(e)$

goal: find the length of the shortest path $\text{dist}(y, z)$ from y to z for all $y, z \in V$

define the sub-problem in words: let $D[i, s, t]$ be the length of the shortest path from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, the final solution is $D[n, s, t]$ $\Rightarrow$ indexed by the available intermediate vertices

state the recursive solution: base case is $D[0, s, t] = w(s, t)$ if $\vec{st} \in E$ and ∞ otherwise
if $i \notin$ the shortest path $\mathcal{P}$ from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, then $D[i, s, t] = D[i - 1, s, t]$
if $i \in$ the shortest path $\mathcal{P}$ from s to t using a subset of $\{1, 2, \ldots, i\}$ as intermediate vertices, then $D[i, s, t] = D[i - 1, s, i] + D[i - 1, i, t]$, because $\mathcal{P}$ must start from s, go through i, and end at t and all the intermediate vertices that can be used is from $\{1, 2, \ldots, i - 1\}$

$$s \xrightarrow{\text{subset of } \{1,2,\ldots,i-1\}} i \xrightarrow{\text{subset of } \{1,2,\ldots,i-1\}} t$$

Since we have a 3-nested for-loop and the if-else statement within the nested loop takes $O(1)$ time, the running time is $O(n^3)$.

4 Divide and Conquer: Fast Integer Multiplication

4.1 Fast Integer Multiplication

The standard divide-and-conquer approach is to divide both X and Y into halves—the first $n/2$ bits and the last $n/2$ bits: $X = 2^{n/2}X_L + X_R$ and $Y = 2^{n/2}Y_L + Y_R$. Therefore XY can be expressed as

$$XY = \left(2^{n/2}X_L + X_R\right)\left(2^{n/2}Y_L + Y_R\right) = 2^n X_L Y_L + 2^{n/2}\left(X_L Y_R + X_R Y_L\right) + X_R Y_R$$

We then follow Gauss's idea to compute three instead of four multiplications: $A = X_L Y_L$, $B = X_R Y_R$, and $C = (X_L + X_R)(Y_L + Y_R)$ and compute Z as $Z = 2^n A + 2^{n/2}(C - A - B) + B$. The running time satisfies the recurrence $T(n) = 3T(n/2) + O(n)$ which solves to $O(n^{\log_2 3})$ by the master theorem.

5 Divide and Conquer: Linear-Time Median

5.1 Linear-Time Median

The standard divide-and-conquer approach is to partition the unsorted list A into $A_{<p}$, $A_{=p}$, and $A_{>p}$ sublists based on a pivot p, and then recursively search in one of the three sublists. To have a linear $T(n) = O(n)$ running time the recurrence should be of the form $T(n) = T(cn) + O(n)$ with a constant $c < 1$. We select the median of the medians of $n/5$ groups of A to be p, which guarantees $|A_{<p}| \leq (1 - (3/10))n = 0.7n$ and $|A_{>p}| \leq (1 - (3/10))n = 0.7n$. This extra sample median step introduces $T(n/5)$ running time to the recurrence which becomes $T(n) = T(0.7n) + T(0.2n) + O(n)$. The revised recurrence solves to $T(n) = O(n)$ by the master theorem.

6 Divide and Conquer: Solving Recurrences

6.1 Solving Recurrences

For the general form of recurrence: $T(n) = aT(n/b) + O(n)$ with constant $a > 0$ and $b > 1$. Following the same expansion as above we have

$$T(n) \leq cn\left(1 + \left(\frac{a}{b}\right) + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^{\log_b n - 1}\right) + a^{\log_b n} T(1)$$

where

$$cn\left(1 + \left(\frac{a}{b}\right) + \left(\frac{a}{b}\right)^2 + \cdots + \left(\frac{a}{b}\right)^{\log_b n - 1}\right) = \begin{cases} O\left(n^{\log_b a}\right) & \text{if } a > b \\ O(n \log n) & \text{if } a = b \\ O(n) & \text{if } a < b \end{cases}$$

Of course we can look at more general forms of recurrence with $O(n^d)$ for some constant d. The solution bears similar form:

$$T(n) = \begin{cases} O\left(n^{\log_b a}\right) & \text{if } \log_b a > d \\ O(n^d \log n) & \text{if } \log_b a = d \\ O(n^d) & \text{if } \log_b a < d \end{cases}$$

7 FFT: Part 1

7.1 Polynomial Multiplication

The problem is:

input: two coefficient vectors $a = (a_0, a_1, a_2, \ldots, a_{n-1})$ and $b = (b_0, b_1, b_2, \ldots, b_{n-1})$ defining two polynomials

goal: compute the coefficient vector $c = a * b = (c_0, c_1, c_2, \ldots, c_{2n-2})$ for their product polynomial in $O(n \log n)$ time, where $c = a * b$ is known as the **convolution** of a and b

The algorithm uses the following two representations of a polynomial $A(x)$

coefficients: $a = (a_0, a_1, \ldots, a_{n-1})$

values: $A(x_1)$, $A(x_2)$, $\ldots$, $A(x_n)$

and FFT which converts between the coefficient representation and the value representation is:

$$\text{coefficients} \xrightarrow{O(n\log n)\text{ FFT}} \text{values} \implies O(n) \text{ value multiplication} \implies \text{values} \xrightarrow{O(n\log n)\text{ FFT}} \text{coefficients}$$

8 FFT: Part 2

8.1 FFT

The algorithm of FFT to evaluate a polynomial $A(x)$ of degree $\leq n-1$ at n points which are the nth roots of unity (assume $n = 2^k$) is:

- define $A_{\text{even}}(y)$ and $A_{\text{odd}}(y)$ of degree $\leq n/2-1$ which takes $O(n)$ time, and recursively evaluate them at the square of the nth roots of unity which are the $n/2$th roots of unity
- take $O(n)$ time to get $A(x)$ at the nth roots of unity by applying the property $x_{n+i} = -x_i$ via

$$A(x_i) = A_{\text{even}}(x_i^2) + x_i A_{\text{odd}}(x_i^2)$$
$$A(x_{n+i}) = A_{\text{even}}(x_i^2) - x_i A_{\text{odd}}(x_i^2)$$

Let $T(n)$ be the running time of input size n. We have two subproblems of half the size, and it takes $O(n)$ time to partition into A_{even} and A_{odd} and merge the solutions together. Hence $T(n)$ satisfies the recurrence $T(n) = 2T(n/2) + O(n)$, which solves to $O(n\log n)$ by the master theorem.

8.2 Inverse FFT

The FFT conversion from coefficient vector to evaluation at the nth roots of unity for a polynomial can be represented as a matrix-vector product $A = M_n(\omega_n)a = \text{FFT}(a, \omega_n)$, where ω_n is a primitive nth root of unity. To convert the evaluation back to the coefficient vector we multiply the evaluation vector with the inverse of $M_n(\omega_n)$, which turns out to be $M_n(\omega_n^{-1})/n$ thus it can also be represented as a FFT: $na = M_n(\omega_n^{-1})A = \text{FFT}(A, \omega_n^{n-1})$. Consolidating everything we have the following $O(n\log(n))$ polynomial multiplication algorithm:

$$\left.\begin{array}{l}\text{FFT}(a,\omega_{2n}) = (r_0, r_1, \ldots, r_{2n-1}) \\ \text{FFT}(b,\omega_{2n}) = (s_0, s_1, \ldots, s_{2n-1})\end{array}\right\} \implies t_j = r_j \times s_j \implies \frac{1}{2n}\text{FFT}(t, \omega_{2n}^{2n-1}) = (c_0, c_1, \ldots, c_{2n-1})$$

Since the FFT takes $O(n\log n)$ time, dominating over the $2n$ value multiplications which takes $O(n)$ time, the running time of the algorithm is $O(n\log n)$.

9 Graph Algorithms: Strongly Connected Components

9.1 DFS on Undirected Graphs

The explore() subroutine is:

```
Explore(z)
    cc number(z) = cc
    visited(z) = true
    for all (z, w) ∈ E do
        if not visited(w) then
            Explore(w)
```

The running time is $O(n+m)$ where $n = |V|$ and $m = |E|$.

9.2 DFS on Directed Graphs

The differences from DFS on undirected graphs are highlighted in bold:

```
Explore(z)
    preorder(z) = clock
    clock++
    visited(z) = true
    for all (z, w) ∈ E do
        if not visited(w) then
            Explore(w)
            prev(w) = z
    postorder(z) = clock
    clock++
```

There are four types of edges:

tree edge: go down one depth of the DFS tree, postorder of the head > postorder of the tail

back edge: go from a descendant to an ancestor, postorder of the head < postorder of the tail

forward edge: go down multiple depths of the DFS tree, postorder of the head > postorder of the tail

cross edge: head and tail have no ancestor-descendant relation to each other, postorder of the head > postorder of the tail

Out of the four only back edge has increasing postorder number. A directed graph G has a cycle if and only if its DFS tree has a back edge.

9.3 Directed Acyclic Graphs

A **directed acyclic graph** or DAG has no cycle thus no back edge. We will topologically sort a DAG i.e. order vertices so that all edges go from lower order number vertices to higher order number vertices. To do so we run DFS on the DAG and sort the vertices by decreasing postorder number. Since all postorder numbers fall within the range from 1 to $2n$ where $n = |V|$, it takes $O(n)$ time to sort the vertices by decreasing postorder number and $O(n + m)$ time to run DFS, hence the total algorithm takes $O(n + m)$ time.
There are two types of vertices in a topological order:

source vertex: has no incoming edge, a DAG has at least one source—the first vertex

sink vertex: has no outgoing edge, a DAG has at least one sink—the last vertex

There might be multiple sources and/or sinks because there might be multiple topological orders. This motivates the following alternative topological sorting algorithm, which is not useful to DAG but useful to general directed graphs:

1. find a sink, output it, and delete it from the graph
2. repeat step 1 until the graph is empty

9.4 Strongly Connected Component

The directed graph analog of connected component in undirected graph is **strongly connected component** or SCC. Vertices v and w are strongly connected if there is a path $v \rightsquigarrow w$ and a path $w \rightsquigarrow v$. Similar to undirected graph where a connected component is a maximal set of connected vertices, a SCC is a maximal set of strongly connected vertices. A source in a DAG is a SCC by itself because no other vertices can reach it, so is a sink in a DAG. Take a representative vertex from each SCC we form a metagraph of SCCs. Every metagraph on SCCs of a directed graph is a DAG. Hence we can sort its SCCs into topological order. We can find the SCCs and their topological order with two runs of DFS. We will find a sink SCC, output it, delete it from the metagraph, and then repeat.
In a DAG the vertex with the highest postorder number is a source. Similarly for general directed graphs the vertex with the highest postorder number is guaranteed to lie in a source SCC. We can use this property to identify a sink SCC, because with all edges flipped a source SCC becomes a sink SCC. Therefore the algorithm of finding and topologically sorting SCCs in a directed graph G consists of the following steps:

1. construct the reverse graph G^R
2. run DFS on G^R to determine the postorder numbers of V
3. order V by decreasing postorder number
4. run explore() on G from the vertex with the highest postorder number to identify a SCC of G
5. mark off the vertices of the SCC and repeat step 4

With this algorithm the SCCs of G are output in reverse topological order, and the *cc* number produced by the undirected version DFS gives the topological order of the SCCs.
To prove the fact that the vertex with the highest postorder number must lie in a source SCC, we first prove the simpler claim that if there exists a edge that goes from $v \in S$ to $w \in S'$, then the maximum postorder number in SCC $S >$ the maximum postorder number in SCC S'. This claim provides a way to topologically sort SCCs—sort them by the maximum postorder number in them in decreasing order. Since we can find SCCs and their topological order with two runs of DFS, the SCC topological sorting algorithm takes $O(n + m)$ time.

9.5 BFS and Dijstra's Algorithm

The input to BFS is similar to that for DFS except for an additional input parameter—start vertex s. With unweighted G $\text{dist}_s(v)$ is the minimum number of edges from s to v for all $v \in V$. Like DFS BFS is linear time, so the running time is $O(n + m)$.
Dijkstra's algorithm is a more sophisticated version of BFS. It solves a weighted version of the graph and has an additional input—a length function $l : e \mapsto l(e)$ assigning a positive length to all edges. Dijkstra's algorithm uses the BFS framework with the min-heap/priority queue data structure. Each operation in the min-heap data structure takes $O(\log n)$ time, hence the total running time of Dijkstra's algorithm is $O((n + m) \log n)$.

10 Graph Algorithms: Satisfiability

10.1 SAT Problem

One application of strongly connected component is the **satisfiability problem** or the SAT problem. Consider n Boolean variables $x_1, x_2, \ldots, x_n$. There are $2n$ literals: $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n$. Use $\wedge$ to

denote logical AND and $\vee$ for logical OR. We look at Boolean formulas in **conjunctive normal form** or CNF. A CNF is an AND of m clauses. Each clause is the OR of several literals, e.g. $x_3 \vee \bar{x}_5 \vee \bar{x}_1 \vee x_2$. Any Boolean formula can be converted into a CNF form, but the size of the formula may blow up. The SAT problem is defined as follows:

input: a Boolean formula f in CNF with n Boolean variables $x_1, x_2, \ldots, x_n$ and m clauses

output: assignment (assigning True or False to each variable) that satisfies f (f = True) if one exists, and no if none exists

We will look at a restrictive form of the SAT problem called k**-SAT**, where the sizes of the clauses—the number of literals in it—are at most k. While k-SAT is NP-complete for all $k \geq 3$, there exists a polynomial-time algorithm for 2-SAT which utilizes SCC.

10.2 2-SAT

An input f for 2-SAT can be simplified by removing unit clauses i.e. clauses with only one literal. The basic procedure for eliminating unit clauses is:

1. take a unit clause say literal a_i and satisfy it i.e. set a_i = True
2. remove clauses containing a_i and drop $\bar{a}_i$ from all other clauses
3. the resulted f' is satisfiable if and only if f is satisfiable
4. repeat the above steps until f' is empty which is satisfied or f' with all clauses of size exactly two

Hence we can assume that the input to a 2-SAT problem is constituted by clauses of size exactly two. We can then encode any formula with a directed graph as follows:

- the $2n$ vertices correspond to the $2n$ literals $x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n$
- the $2m$ edges correspond to 2 implications per clause, e.g. to satisfy a clause $\bar{x}_1 \vee \bar{x}_2$ if x_1 = True then x_2 = False so we have an edge from x_1 to $\bar{x}_2$; if x_2 = True then x_1 = False so we have an edge from x_2 to $\bar{x}_1$, in general for a clause $\alpha \vee \beta$ we have an edge $\bar{\alpha} \to \beta$ and an edge $\bar{\beta} \to \alpha$

Translated into the terminology of graph, the polynomial-time algorithm for 2-SAT is:

- if for some i, x_i and $\bar{x}_i$ are in the same SCC, then f is not satisfiable
- if for any i, x_i and $\bar{x}_i$ are in different SCCs, then f is satisfiable

The former is obvious: if there is a path $x_i \rightsquigarrow \bar{x}_i$ and a path from $\bar{x}_i \rightsquigarrow x_i$, then the former requires x_i to be False whereas the latter requires x_i to be True at the same time. It implies that the formula is not satisfiable. For the latter we explicitly construct a satisfying assignment by modifying the SCC topological order algorithm based on the key fact that S is a sink SCC if and only if $\bar{S}$ is a source SCC: $\Rightarrow$ SCC correspondence claim $\Rightarrow$ 2-SAT

2SAT(f)

1. construct directed graph G for f and run the SCC topological order algorithm on G
2. take a sink SCC S and set it to be True, which sets $\bar{S}$ to be False
3. remove S and $\bar{S}$ and repeat until we empty the graph

The running time is dominated by the SCC construction step, which takes $O(n+m)$ time.
To prove the key fact that S is a sink SCC if and only if $\bar{S}$ is a source SCC, we need a simpler claim—there is a path $\alpha \rightsquigarrow \beta$ if and only if there is a path $\bar{\beta} \rightsquigarrow \bar{\alpha}$. With this claim the key fact can be proved as follows: $\Rightarrow$ reverse path claim $\Rightarrow$ SCC correspondence claim

- take a pair of vertices α and β in sink SCC S, since they belong to the same SCC there exist paths $\alpha \rightsquigarrow \beta$ and $\beta \rightsquigarrow \alpha$ which correspond to clauses $(\bar{\alpha} \vee \beta)$ and $\left(\alpha \vee \bar{\beta}\right)$
- this implies that there exist paths $\bar{\beta} \rightsquigarrow \bar{\alpha}$ and $\bar{\alpha} \rightsquigarrow \bar{\beta}$, which implies that $\bar{\alpha}$ and $\bar{\beta}$ are in the same SCC denoted by $\bar{S}$, hence S is a SCC if and only if $\bar{S}$ is a SCC
- since S is a sink SCC, for any $\alpha \in S$ there is no outgoing edge $\alpha \to \gamma$, by the claim above this implies that there is no incoming edge $\bar{\gamma} \to \bar{\alpha}$ for any $\bar{\alpha} \in \bar{S}$, hence $\bar{S}$ is a source SCC, reversing the argument we can prove the key fact

To prove the simpler claim let $\alpha \to \beta$ go along $\alpha = \gamma_0 \to \gamma_1 \to \cdots \to \gamma_{l-1} \to \gamma_l = \beta$, where the edge $\gamma_i \to \gamma_{i+1}$ comes from the clause $(\bar{\gamma}_i \vee \gamma_{i+1})$ which implies the other edge $\bar{\gamma}_{i+1} \to \bar{\gamma}_i$. This argument applies to $i = 0, 1, 2, \ldots, l-1$, which form a path $\bar{\beta} = \bar{\gamma}_l \to \bar{\gamma}_{l-1} \to \cdots \to \bar{\gamma}_1 \to \bar{\gamma}_0 = \bar{\alpha}$.

11 Graph Algorithms: Minimum Spanning Tree

11.1 The MST Problem

Greedy algorithm takes local optimal move. It does not always lead to global optimum as in knapsack problem, but it does for the MST problem:

input: undirected graph $G = (V, E)$ with weight $w(e)$ for $e \in E$

goal: find the minimum size of the connected subgraphs, since the connected subgraph of minimal size is a spanning tree, it is equivalent to finding the minimum-weight spanning tree

The basic properties of a **tree** (connected + acyclic) include

- an n-vertex tree has $n - 1$ edges
- there is exactly one (connected + acyclic) path between every pair of vertices
- any connected graph $G = (V, E)$ with $|E| = |V| - 1$ is a tree

The third property follows from combining the first property and second property.
Kruskal's algorithm is a greedy algorithm for MST, which consists of the following steps:

1. sort E by increasing weight
2. set X to be the edges that we have inserted into our MST and initialize it to $\emptyset$
3. for $e = (v, w) \in E$ in order, if $X \cup e$ doesn't have a cycle then add e to X

The first step takes $O(m \log n)$ time where $m = |E|$ and $n = |V|$. For the third step we can add e into X if they belong to different connected components. Checking whether v and w are in the same connected component can be done using the union-find data structure, which takes $O(\log n)$ time because:

find: takes $O(\log n)$ time to check the component containing v and that containing w

union: merging the two components after adding e takes $O(\log n)$ time

Hence the third step also takes $O(m \log n)$ time, and so is Kruskal's algorithm.

11.2 Cut Property

We prove Kruskal's algorithm via the following cut property: for an undirected graph $G = (V, E)$ take subgraph $X \subset T$ where T is a MST. Take any $S \subset V$ where no edge of X is in the cut $(S, \bar{S})$. We take e^* to be the edge of minimum weight among all edges in the cut$(S, \bar{S})$. Such an edge always exists because we assume the graph is connected. Then $X \cup e^* \subset T'$ where T' is a MST (T' need not be the same as T).
The proof of the cut property is as follows:

1. fix G, X, T, S so that $X \subset T$ where T is a MST and there is no edge of X crossing S and $\bar{S}$
2. choose any edge of minimum weight across this cut $w(e^*) \leq w(e_1), \ldots, w(e_i)$
3. construct MST T' where $X \cup e^* \subset T'$, if $e^* \in T$ then let $T' = T$ and we are done, otherwise set $T' = T \cup e^* - e'$ where $e' \in T$ crosses S and it always exists because T is a MST thus by definition must connect S and $\bar{S}$

T' is a tree because T'

has exactly $n-1$ edges: trivial because T has exactly $n-1$ edges

is connected: for any $y, z \in V$ let $\mathcal{P}$ be a path from y to z in T, let $e' = c \to d$, let C be the cycle in $T \cup e^*$, let $\mathcal{P}' = C - e'$ be a path from c to d in T', then to go from y to z in T' we modify $\mathcal{P}$ by replacing e' with $\mathcal{P}'$

T' is a MST because $T' = T \cup e^* - e'$ thus $w(T') = w(T) + w(e^*) - w(e')$. By construction $w(e^*) \leq w(e')$ thus $w(T') \leq w(T)$. Because $w(T)$ is minimum since it is a MST, $w(T')$ is minimum as well.
To apply the cut property to Kruskal's algorithm, define S to be the set of vertices connected through the edges in $X \cup v$ of the edge $e = (v, w)$. Then since adding e to X doesn't create a cycle, the other endpoint w of e must belong to $\bar{S}$, thus by sorting e is the minimum-weight edge across the cut$(S, \bar{S})$. It then follows from the cut property that $X \cup e$ belongs to a MST.

12 Max-Flow: Fork-Fulkerson Algorithm

12.1 Max-Flow

input: a directed graph $G = (V, E)$ and designated $s, t \in V$, a capacity $c_e > 0$ for each $e \in E$

goal: maximize the flow from s to t without exceeding capacities, specified by flow f_e for each $e \in E$

There are two constraints associated with f_e

capacity constraint: $0 \leq f_e \leq c_e$ for all $e \in E$

conservation of flow: inflow to v = outflow from v for all $v \in V - \{s, t\}$

and the flow f from s to t can be measured by either the outflow from s or the inflow to t.

12.2 Residual Network

Define a **residual network** $G^f = (V, E^f)$ for a flow network $G = (V, E)$ with capacity c_e and flow f_e for $e \in E$, where the edge dependence on current flow f is:

forward edge addition: if $\overrightarrow{vw} \in E$ and $f_{vw} < c_{vw}$ then add $\overrightarrow{vw}$ to G^f with capacity $c_{vw} - f_{vw}$

backward edge addition: if $\overrightarrow{vw} \in E$ and $f_{vw} > 0$ then add $\overrightarrow{wv}$ to G^f with capacity f_{vw}

12.3 Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm works on residual network as follows:

1. set $f_e = 0$ for all $e \in E$
2. build the residual network G^f for current flow f
3. use BFS/DFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f
4. given $\mathcal{P}$ let $c(\mathcal{P})$ be the minimum capacity along $\mathcal{P}$ in G^f
5. augment f by $c(\mathcal{P})$ along $\mathcal{P}$—increasing the flow by $c(\mathcal{P})$ along every forward edge while decreasing the flow by $c(\mathcal{P})$ along every backward edge
6. repeat step 2 to 5 until there is no such st-path, and current flow f is output in step 3

The time per round breakdown of the Ford-Fulkerson algorithm is as follows (repeated step 2 to 5):

2. the path $\mathcal{P}$ is of length at most $n-1$ edges thus it takes $O(n)$ time to update residual network
3. the BFS/DFS takes $O(n+m)$ time, assume the number of edges is at least $n-1$ then the time is bounded by $O(m)$
4. the path $\mathcal{P}$ is of length at most $n-1$ edges thus it takes $O(n)$ time to find the minimum capacity
5. the path $\mathcal{P}$ is of length at most $n-1$ edges thus it takes $O(n)$ time to augment current flow

Combined it takes $O(m)$ time per round to run the Ford-Fulkerson algorithm.
To analyze the running time of the Ford-Fulkerson algorithm we need to assume that all capacities are integers in order to bound the number of rounds the algorithm will run. Because if all the capacities in the residual network are integers, then the flow will be increased by at least 1 per round for every forward edge. Let C be the size of the max-flow. Then the algorithm will run for at most C rounds. Therefore the overall running time is $O(Cm)$. Since the running time depends on the input we say that it is pseudo-polynomial. The Edmonds-Karp algorithm overcomes this constraint by using the shortest st-path via BFS instead of DFS, shortest in the sense of minimum number of edges without caring about the capacity and flow.

13 Max-Flow: Max-Flow = Min-Cut

13.1 Min-Cut Problem

A **st-cut** is a cut with $s \in L$ and $t \in R$. Define the **capacity** of a st-cut to be the total capacity of all edges from L to R:

$$\text{capacity}(L, R) = \sum_{\overrightarrow{vw} \in E, v \in L, w \in R} c_{vw}$$

The min-cut problem is to find the st-cut with minimum capacity.

13.2 Max-Flow = Min-Cut

The max-flow = min-cut theorem states that for any flow network the size of the max-flow equals the minimum capacity of an st-cut. We will prove the equivalence by showing that

max-flow $\leq$ min st-cut: $\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L)$

$$\begin{aligned} f^{\text{out}}(L) - f^{\text{in}}(L) &= \sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f_{vw} - \sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f_{wv} \\ &= \left(\sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f_{vw} + \sum_{\overrightarrow{vw} \in E, v \in L, w \in L} f_{vw} \right) - \left(\sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f_{wv} + \sum_{\overrightarrow{wv} \in E, w \in L, v \in L} f_{wv} \right) \\ &= \sum_{v \in L} f^{\text{out}}(v) - \sum_{v \in L} f^{\text{in}}(v) \\ &= f^{\text{out}}(s) = \text{size}(f) \end{aligned}$$

it follows that $\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L) \leq f^{\text{out}}(L) \leq \text{capacity}(L, R)$, which holds for any flow f and any st-cut (L, R), we thus can take the maximum of the left-hand side and the minimum of the right-hand side to arrive at $\max_f \text{size}(f) \leq \min_{(L,R)} \text{capacity}(L, R)$

max-flow $\geq$ min st-cut: take the flow f^* output from the Ford-Fulkerson algorithm to construct a st-cut—define L = vertices reachable from s in G^{f^*} and $R = V - L$, then $s \in L$ while $t \in R$ because having no st-path in G^{f^*} means that t is not reachable from s in G^{f^*}
this st-cut has the following desired properties:

- for $\overrightarrow{vw} \in E$, $v \in L$, $w \in R$, $f^*_{vw} = c_{vw}$, because there is forward edge in the residual network for each not fully capacitated edge and that forward edge will connect s to w, contradicting with $w \in R$, contradicting with $w \in R$, it then follows that $\sum_{\overrightarrow{vw} \in E, v \in L, w \in R} f^*_{vw} = f^{*\text{out}}(L) = \text{capacity}(L, R)$
- for $\overrightarrow{wv} \in E$, $w \in R$, $v \in L$, $f^*_{wv} = 0$, because there is backward edge in the residual network for each edge with nonzero flow and that backward edge will connect s to w, contradicting with $w \in R$, contradicting with $w \in R$, it then follows that $\sum_{\overrightarrow{wv} \in E, w \in R, v \in L} f^*_{wv} = f^{*\text{in}}(L) = 0$

combined we have $\text{size}(f^*) = f^{*\text{out}}(L) - f^{*\text{in}}(L) = \text{capacity}(L, R)$ and thus

$$\max_f \text{size}(f) \geq \text{size}(f^*) = \text{capacity}(L, R) \geq \min_{(L,R)} \text{capacity}(L, R)$$

13.3 Proof of Ford-Fulkerson Algorithm

Since max-flow $\leq$ min st-cut, the only way we can achieve equality $\text{size}(f^*) = \text{capacity}(L, R)$ is to have both sides optimal, i.e. f^* is a max-flow and (L, R) is a min st-cut. This proves the Ford-Fulkerson algorithm. The stopping condition takes $O(n+m)$ time to check whether a particular flow f is maximal or not, because it consists of building a residual network from current flow in $O(n+m)$ time and running DFS in $O(n + m)$ time to look for a st-path in it.

14 Max-Flow: Image Segmentation

14.1 Image Segmentation

Model an image as lying on a undirected graph with vertices being the pixels of the image and edges connecting neighboring pixels.

input: undirected graph $G = (V, E)$, for each $i \in V$, let

- f_i = likelihood/weight that i is in the foreground, $f_i \geq 0$
- b_i = likelihood/weight that i is in the background, $b_i \geq 0$

for each $(i, j) \in E$, let P_{ij} = separation penalty or the cost of separating i and j into different objects, $P_{ij} \geq 0$

goal: partition V into $V = F \cup B$ (F = foreground, B = background), for a partition (F, B) define its weight $w(F, B)$ as

$$w(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

find a partition (F, B) with maximum weight

To convert this max-cut problem into a min st-cut problem, we first need to reformulate the weight definition: let $L = \sum_{i \in V} (f_i + b_i)$ thus

$$\sum_{i \in F} f_i + \sum_{j \in B} b_j = L - \sum_{i \in B} f_i - \sum_{j \in F} b_j \implies$$

$$w(F, B) = \sum_{i \in F} f_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij} = L - \sum_{i \in B} f_i - \sum_{j \in F} b_j - \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

$$= L - w'(F, B)$$

where the new weight $w'(F, B)$ is defined as

$$w'(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$$

Since $w(F, B) = L - w'(F, B)$, maximizing $w(F, B)$ is equivalent to minimizing $w'(F, B)$. In addition unlike $w(F, B)$ all terms in $w'(F, B)$ are positive.

14.2 Flow Network

Next we need to define a directed graph $G' = (V', E')$ based on the undirected graph $G = (V, E)$ input to the image segmentation problem:

- for $(i, j) \in E$, add $i \to j$ and $j \to i$ with capacity both equal to P_{ij}
- add s corresponding to the foreground pixels, for $i \in V$ add $s \to i$ of capacity f_i
- add t corresponding to the background pixels, for $i \in V$ add $i \to t$ of capacity b_i

For a cut (F, B) its capacity turns out to be exactly $w'(F, B)$ because

- for $i \in B$, we get $s \to i$ of capacity f_i
- for $j \in F$, we get $j \to t$ of capacity b_j
- for $(i, j) \in E$ with $i \in F$ and $j \in B$, we get $i \to j$ of capacity P_{ij}

capacity(F, B) sums them up and exactly equals to $w'(F, B) = \sum_{i \in B} f_i + \sum_{j \in F} b_j + \sum_{\substack{(i,j) \in E \\ i \in F, j \in B}} P_{ij}$.

In summary given input (G, f, b, P) for image segmentation, we can define a flow network (G', c) and get a max-flow f^* on this network, where size$(f^*) = \min_{(F,B)} w'(F, B)$. It follows that $\max_{(F,B)} w(F, B) = L - \min_{(F,B)} w'(F, B) = L - \text{size}(f^*)$. $\Rightarrow$ maximization $\Rightarrow$ minimization $\Rightarrow$ min-cut $\Rightarrow$ max-flow

15 Max-Flow: Edmonds-Karp Algorithm

15.1 Edmonds-Karp Algorithm

Ford-Fulkerson: assume integral capacities, the main step is to find augmenting paths using DFS/BFS, the running time is $O(mC)$ where C = size of max-flow

Edmonds-Karp: no assumption on capacities, the main step is to find augmenting paths using BFS, the running time is $O(m^2n)$

The only distinction between Edmonds-Karp and Ford-Fulkerson lies in step 3, where

Ford-Fulkerson: use BFS/DFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f

Edmonds-Karp: use BFS to look for a st-path $\mathcal{P}$ in G^f, if there is no such path then output f

To show the running time of the Edmonds-Karp algorithm is $O(m^2n)$, we bound the number of rounds it runs by mn. Then since at each round we run BFS and BFS takes linear time, which is $O(m)$ assuming the number of edges is at least the number of vertices, the total running time is $O(m) \times O(mn)$.
To bound the number of rounds by mn, we show that we can delete and re-insert an edge in residual network for at most $n/2$ times. Since at least one edge is deleted in every round (the one with minimum capacity $c(\mathcal{P})$ along the augmenting path), it follows that there are at most mn rounds in the Edmonds-Karp algorithm where m is the number of edges.
To show we can delete and re-insert an edge in residual network for at most $n/2$ times, we show

level(v) **never decreases:** because it might decrease only if we add an edge $\overrightarrow{wv}$ to give a shorter path from s to z, however adding $\overrightarrow{wv}$ implies $\overrightarrow{vw} \in \mathcal{P}$, which in turn implies level(w) = level$(v) + 1 = i+1$, hence the added edge goes from a higher level to a lower level thus will not decrease level(z)

deletion + re-insertion increases level(v) **by 2:** because removing $\overrightarrow{vw}$ from G^f implies $\overrightarrow{vw} \in \mathcal{P}$ and hence level(w) = level$(v) + 1 = i+1$, and re-inserting $\overrightarrow{vw}$ implies $\overrightarrow{wv} \in \mathcal{P}$ and hence level(v) = level$(w) + 1 \geq i + 2$

Because the minimum level is 0 (for source s) while the maximum level is n, we can delete and re-insert an edge for at most $n/2$ times.

16 Max-Flow: Generalization

16.1 Max-Flow with Demand

input: a directed graph $G = (V, E)$ and designated $s, t \in V$, a capacity $c(e) > 0$ for $e \in E$, a demand $d(e) > 0$ for $e \in E$

goal: whether a feasible flow from s to t exists, and if yes maximize it, a flow is **feasible** if $d(e) \leq f(e) \leq c(e)$ for $e \in E$

To reduce this feasible flow problem to a max-flow problem, we construct a max-flow input $(G', c'(e))$ from the feasible flow input $(G, c(e), d(e))$ as follows:

- for $e \in E$, add e to G' with $c'(e) = c(e) - d(e)$
- for $v \in V$, add $s' \to v$ with $c'(\overrightarrow{s'v}) = d^{\text{in}}(v)$, add $v \to t'$ with $c'(\overrightarrow{vt}) = d^{\text{out}}(v)$
- add $t \to s$ with $c'(\overrightarrow{ts}) = \infty$

Let $D = \sum_{e \in E} d(e)$. $D = \sum_{v \in V} d^{\text{in}}(v) = \sum_{v \in V} d^{\text{out}}(v)$ since every edge has exactly one tail and exactly one head. $c'^{\text{out}}(s') = \sum_{v \in V} d^{\text{in}}(v) = D$ hence $\text{size}(f') \leq D$. We say f' is **saturating** if $\text{size}(f') = D$. We will show that G has a feasible flow if and only if G' has a saturating flow.

saturating $\Rightarrow$ feasible: let $f(e) = f'(e) + d(e)$ for $e \in E$, f is

- feasible because $f'(e) \geq 0$ thus $f(e) \geq d(e)$
- valid because $f'(e) \leq c'(e)$ thus $f(e) \leq c'(e) + d(e) = c(e)$ and

$$f'^{\text{in}}(v) = f'(\overrightarrow{s'v}) + \sum_{w \in V} f'(\overrightarrow{wv}) = d^{\text{in}}(v) + \sum_{w \in V} (f(\overrightarrow{wv}) - d(\overrightarrow{wv}))$$
$$= d^{\text{in}}(v) + \sum_{w \in V} f(\overrightarrow{wv}) - d^{\text{in}}(v) = f^{\text{in}}(v)$$

$$f'^{\text{out}}(v) = f'(\overrightarrow{vt'}) + \sum_{w \in V} f'(\overrightarrow{vw}) = d^{\text{out}}(v) + \sum_{w \in V} (f(\overrightarrow{vw}) - d(\overrightarrow{vw}))$$
$$= d^{\text{out}}(v) + \sum_{w \in V} f(\overrightarrow{vw}) - d^{\text{out}}(v) = f^{\text{out}}(v)$$
$$\Longrightarrow \quad f^{\text{in}}(v) = f'^{\text{in}}(v) = f'^{\text{out}}(v) = f^{\text{out}}(v)$$

feasible $\Rightarrow$ saturating: let $f'(e) = f(e) - d(e)$ for $e \in E$, add to f'

- $f'(\overrightarrow{s'v}) = d^{\text{in}}(v)$ and $f'(\overrightarrow{vt'}) = d^{\text{out}}(v)$ for $v \in V \Rightarrow$ saturating condition
- $f'(\overrightarrow{ts}) = \text{size}(f)$

f is valid because $d(e) \leq f(e) \leq c(e)$ implies $0 \leq f'(e) \leq c'(e) = c(e) - d(e)$ and by the algebra above $f'^{\text{in}}(v) = f^{\text{in}}(v) = f^{\text{out}}(v) = f'^{\text{out}}(v)$.

Hence to solve the feasible flow problem in G we construct G' and run the max-flow algorithm. Then we check whether the size of the resulted max-flow is saturating. If yes then we transform the max-flow in G' back to a feasible flow in G as described above.
Once we obtain a feasible flow we can augment it to a maximum-sized feasible flow in the same manner as in the Ford-Fulkerson or Edmonds-Karp algorithm, but with the following modifications:

- start from the found feasible flow f instead of zero flow
- the capacity of the residual graph G^f is

$$c^f(\overrightarrow{vw}) = \begin{cases} c(\overrightarrow{vw}) - f(\overrightarrow{vw}) & \text{if } \overrightarrow{vw} \in E \\ f(\overrightarrow{wv}) - d(\overrightarrow{wv}) & \text{if } \overrightarrow{wv} \in E \\ 0 & \text{otherwise} \end{cases}$$

17 Randomized Algorithms: Modular Arithmetic

17.1 Modular Arithmetic

x and y are **congruent** mod N if $x \bmod N = y \bmod N$, denoted by $x \equiv y \bmod N$. Fast modular exponentiation uses repeated squaring, i.e. $x \bmod N = a_1$, $x^2 \bmod N = a_1^2 \bmod N = a_2$, $x^4 \bmod N$

$= a_2^2 \bmod N = a_4$ and so on. Then for even y we compute $x^y = \left(x^{y/2}\right)^2$ and for odd y we compute $x^y = x\left(x^{\lfloor y/2 \rfloor}\right)^2$.

```
Mod-Exp(x, y, N)
    if y = 0 then
        return 1
    z = Mod-Exp(x, ⌊y/2⌋, N)
    if y is even then
        return z^2 mod N
    else
        return xz^2 mod N
```

Since it takes $O(n^2)$ time to multiply or divide two n-bit integers and there are $\log y \leq n$ rounds of multiplication, the running time is $O(n^3)$.

17.2 Multiplicative Inverse

x is the **multiplicative inverse** of z mod N if $xz \equiv 1 \bmod N$, denoted by $x \equiv z^{-1} \bmod N$. $x^{-1} \bmod N$ exists if and only if $\gcd(x, N) = 1$, i.e. when x and N are **relatively prime**, because

- if $\gcd(x, N) > 1$ then $x^{-1} \bmod N$ does not exist, suppose it exists and equals to z, let $\gcd(x, N)$ be r, then xz is certainly a multiple of r so is qN for any integer q, but then $xz \neq qN + 1$ since the left-hand side is divisible by r while the right-hand side is not, a contradiction
- if $\gcd(x, N) = 1$ then $x^{-1} \bmod N$ exists, we find it by extended Euclid algorithm

If $x^{-1} \bmod N$ exists, then it must be unique.

17.3 Extended Euclid's Algorithm

Euclid's rule states that for integers x, y where $x \geq y > 0$, $\gcd(x, y) = \gcd(x \bmod y, y)$. This follows from the fact that $\gcd(x, y) = \gcd(x - y, y)$, which is true because

- if d divides x and y, then d divides $x - y$
- if d divides $x - y$ and y, then d divides x

Utilizing Euclid's rule we devise **Euclid's GCD algorithm**

```
Euclid(x, y) where x ≥ y ≥ 0
    if y = 0 then
        return x
    else
        return Euclid(y, x mod y)
```

where the base case $\gcd(x, 0) = x$ can be shown by using Euclid's rule: $\gcd(x, 0) = \gcd(kx, x) = x$.
To compute the running time of Euclid's GCD algorithm, notice that the only nontrivial step in each round is computing $x \bmod y$ which takes $O(n^2)$ time for n-bit integers x and y, and the number of rounds the algorithm is recursively called is bounded by $2n$, because the input size is reduced by at least a factor of 2 after two rounds of recursive call

$$(x, y) \longrightarrow (y, x \bmod y) \longrightarrow (x \bmod y, y \bmod (x \bmod y)) \longrightarrow \cdots$$

Therefore the running time of Euclid's GCD algorithm is $O(n^3)$.
To compute multiplicative inverse we will use the following extended version of Euclid's GCD algorithm, which outputs $d = \gcd(x, y) = x\alpha + y\beta$ for some integer α, β

ExtendedEuclid(x, y) where $x \geq y \geq 0$
  if $y = 0$ then
    return $(x, 1, 0)$
  else
    $(d, \alpha', \beta') =$ ExtendedEuclid$(y, x \bmod y)$
    return $(d, \beta', \alpha' - \lfloor x/y \rfloor \beta')$

The running time is the same as Euclid's GCD algorithm, as each round still takes $O(n^2)$ time and the number of rounds is bounded by $n/2$.
With extended Euclid's algorithm, if $d = \gcd(x, N) = 1$ then we have $1 = x\alpha + N\beta$ for some integer α and β. Taking mod N on both sides we get $1 \equiv x\alpha \bmod N$, which means $\alpha = x^{-1} \bmod N$. $\Rightarrow$ use Euclid to check whether inverse exists, use extended Euclid to find the inverse if exists

18 Randomized Algorithms: RSA

18.1 Fermat's Little Theorem

Fermat's little theorem states that, if p is prime then for every $1 \leq z \leq p-1$ we have $z^{p-1} \equiv 1 \bmod p$. Note that if p is prime the condition $1 \leq z \leq p-1$ can be replaced with $\gcd(z, p) = 1$, which motivates its generalization to any integer N in Euler's theorem.

18.2 Euler's Theorem

We will actually use a generalization of Fermat's little theorem called **Euler's theorem**, which states that for any integer N and z where $\gcd(z, N) = 1$, $z^{\phi(N)} \equiv 1 \bmod N$ where $\phi(N) =$ the number of integers between 1 and N which are relatively prime to N. This $\phi(N)$ is called *Euler's totient function*. For prime p, $\phi(p) = p-1$. Hence Euler's theorem reduces to Fermat's little theorem for primes. We will apply Euler's theorem for $N = pq$ where both p and q are primes. $\phi(N) = pq - p - q + 1 = (p-1)(q-1)$ where we subtract q multiples of p and p multiples of p and $+1$ to correct the double subtraction of pq. Thus for any z where $\gcd(z, pq) = 1$ we have $z^{(p-1)(q-1)} \equiv 1 \bmod pq$.

18.3 RSA

RSA consists of the following steps

1. randomly choose primes p and q and let $N = pq$
2. find e where $\gcd(e, (p-1)(q-1)) = 1$
3. compute $d \equiv e^{-1} \bmod (p-1)(q-1)$ using extended Euclid's algorithm
4. publish public key (N, e)
5. encrypt m by $y \equiv m^e \bmod N$
6. decrypt y by $m \equiv y^d \bmod N$ using fast modular exponentiation

so that $z^{de} \equiv z \times \left(z^{(p-1)(q-1)}\right)^k \bmod pq = z \bmod pq$, where $z \mapsto z^e$ is encryption and $z^e \mapsto (z^e)^d \equiv z$ is decryption. The RSA algorithm is as hard as factorizing N into p and q.
There are in fact some pitfalls associated with the above RSA protocol.

gcd$(m, N) > 1$: if $\gcd(m, N) > 1$ then $\gcd(m, N) = p$, since m is divisible by p so is $y \equiv m^e \bmod N$ thus $\gcd(y, N) = p$, this gives a way for the eavesdropper to learn p and factorize N into p and q

m **not too large:** need $m < N$, can break huge input into n-bit segments thus $m < 2^n$

m **not too small:** common practice is to use $e = 3$, but if $m^3 < N$ then $y = m^3 \bmod N = m^3$, thus taking cubic root would decrypt it $m = y^{1/3}$

send the same message multiple times: if the eavesdropper learns multiple encryption of the same message $y_1 \equiv m^3 \bmod N_1$, $y_2 \equiv m^3 \bmod N_2$, $y_3 \equiv m^3 \bmod N_3$, then he can use Chinese remainder theorem to decrypt m

18.4 Primality Testing

To randomly choose p and q, let r be a random n-bit number, check if r is prime. If yes then output r; if no then repeat the random n-bit number generation. To perform the primality test we use Fermat's little theorem: if r is prime then $z^{r-1} \equiv 1 \bmod r$ for all z in $\{1, 2, \ldots, r-1\}$; if there is z where $z^{r-1} \not\equiv 1 \bmod r$ then r is composite. We call this z a **Fermat witness**.
Every composite r has at least two Fermat witnesses. Simply take z to be a divisor of r, then $\gcd(z, r) = z > 1$ which implies that z does not have multiplicative inverse by Euclid's GCD algorithm. $z^{r-1} \not\equiv 1 \bmod r$ because otherwise $z^{r-1} \equiv z^{r-2}z \equiv 1 \bmod r$ would imply z^{r-2} is the multiplicative inverse of z which is a contradiction. This z is called *trivial Fermat witness* because $\gcd(z, r) > 1$. There also exists *nontrivial Fermat witness* z with $\gcd(z, r) = 1$. Some composites have no nontrivial Fermat witness which are called *Carmichael numbers* or *pseudoprimes*. It is inefficient to use Fermat's primality test for them.
If r has at least one nontrivial Fermat witness, then at least half of $z \in \{1, 2, \ldots, r-1\}$ are Fermat witnesses (ref. book for proof). Based on this fact and ignoring Carmichael numbers we can devise the following primality test: for an n-bit r

1. choose $z_1, z_2, \ldots, z_k$ randomly from $\{1, 2, \ldots, r-1\}$
2. compute $z_i^{r-1} \bmod r$ for $i = 1$ to k
3. if $z_i^{r-1} \equiv 1 \bmod r$ for all i then output r is prime; else output r is composite

For prime r the probability that this test outputs correctly r is prime is 1, whereas for composite and non-Carmichael r the probability this test outputs incorrectly that r is prime is bounded by $(1/2)^k$.

19 NP: Definitions

19.1 NP Problems

Define NP to be the class of all search problems, and P the class of search problems that are solvable in polynomial time. Of course if we can generate a solution in polynomial time, we can also verify it in polynomial time. Hence P is a subset of NP, P $\subseteq$ NP, and P = NP means that it is as difficult to solve a problem or generate a proof as to verify its solution or check the proof.
Formally a search problem is a problem of the following form:

form: given instance I, find a solution S for I if one exists, or output no if I has no solution

requirement: given an instance I and a solution S, we can verify that S is a solution to I in polynomial time, that is the running time is a polynomial of $|I|$

Some examples of NP are

SAT: defined as

> **input:** Boolean formula f in CNF with n variables and m clauses
>
> **output:** satisfying assignment if one exists and no otherwise
>
> it belongs to NP because given f and an assignment to x_1, x_2, ..., x_n it takes $O(n)$ time to check that one clause is satisfied and there are m clauses, hence the total time to verify is $O(nm)$

k-coloring: defined as

> **input:** undirected $G = (V, E)$ and integer $k > 0$ where k is the number of colors available
>
> **output:** assignment to each vertex a color in $\{1, 2, \ldots, k\}$ so that adjacent vertices get different colors if one exists and no otherwise
>
> it belongs to NP because given G and a coloring scheme it takes $O(m)$ time to check that for every edge $(v, w) \in E$ color of $v \neq$ color of w

MST: defined as

> **input:** $G = (V, E)$ with positive edge lengths
>
> **output:** tree T with minimum weight
>
> it belongs to NP because the problem is formulated such that there always is a solution, more importantly given G and a tree T we can run
>
> - $O(m + n)$ time BFS/DFS to check whether it is a tree (connected + acyclic)
> - $O(m \log n)$ time Kruskal's/Prim's algorithm to check whether it has the minimal weight (not necessarily the same tree but the minimal weight is known)
>
> in fact MST is also in P as it can be solved in polynomial time using Kruskal's/Prim's algorithm

knapsack: defined as

> **input:** n objects with integer weights $\omega_1, \omega_2, \ldots, \omega_n$ and integer values $v_1, v_2, \ldots, v_n$, a capacity B
>
> **output:** a subset S of objects with total weight $\sum_{i \in S} w_i \leq B$ and maximal total value $\max\{\sum_{i \in S} v_i\}$
>
> it is not known to be in NP because we don't know a polynomial-time algorithm that can check whether the total value is maximal (recall that the dynamical programming solution is $O(nB)$ not poly$(n, \log B)$), it is not known to be in P either because we don't know a polynomial-time algorithm that can generate the maximal value subset, this argument applies to both the with repetition and without repetition version

knapsack-search: variant of knapsack that is a search problem, defined as

> **input:** n objects with integer weights $\omega_1, \omega_2, \ldots, \omega_n$ and integer values $v_1, v_2, \ldots, v_n$, a capacity B, a goal g
>
> **output:** a subset S of objects with total weight $\sum_{i \in S} w_i \leq B$ and total value $\sum_{i \in S} v_i \geq g$ and no otherwise

it belongs to NP because given subset S it takes $O(n)$ time to check whether $\sum_{i \in S} w_i \leq B$ and $\sum_{i \in S} v_i \geq g$ (more precisely $O(n \log W)$ and $O(n \log V)$ where $\log W$ and $\log V$ are input size) note that if we can solve this knapsack search version in polynomial time, then we can solve the original optimization version in polynomial time, because we can do binary search over g to find the maximal g that has a solution

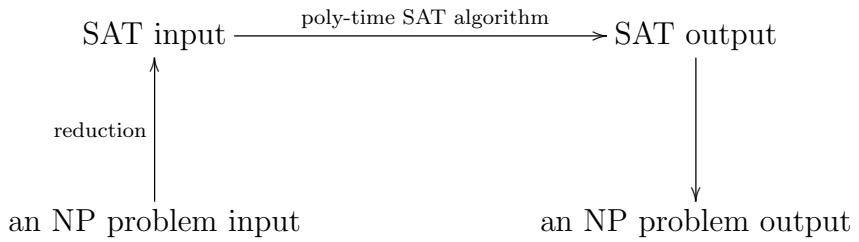
19.2 NP Completeness

NP stands for nondeterministic polynomial time instead of not polynomial time. It is a class of problems that can be solved in polynomial time on a nondeterministic machine. A **nondeterministic machine** is a machine that is allowed to guess at each step, and there is a series of choices of branching which lead to an accepting state.
NP-complete problems are the hardest in the class NP in the sense that

- if P $\neq$ NP then all NP-complete problems are not in P
- if an NP-complete problem can be solved in polynomial time, then all NP problems can be solved in polynomial time i.e. P = NP

SAT is NP-complete means that

- SAT $\in$ NP
- if we can solve SAT in polynomial time then we can solve every NP problem in polynomial time (hardest), this requires every NP problem to have a reduction to SAT



Thus if P $\neq$ NP then SAT $\notin$ P.

19.3 Reduction

A **reduction** from Problem A to Problem B means that if we can solve Problem B in polynomial time, then we can use that algorithm to solve Problem A in polynomial time:

$$\begin{array}{l} \text{poly-time} \\ \text{algorithm for A} \end{array} : I \xrightarrow{f} f(I) \longrightarrow \begin{array}{lllll} \text{poly-time} & \longrightarrow & S & \xrightarrow{h} & h(S) \\ \text{algorithm for B} & \longrightarrow & \text{no} & \longrightarrow & \text{no} \end{array}$$

To show such a reduction exists we need to define

input transformation f**:** input for Problem A $\mapsto$ input for Problem B

output transformation h**:** solution for Problem B $\mapsto$ solution for Problem A

and show that S is a solution for Problem B if and only if $h(S)$ is a solution for Problem A. To show the independent set problem or IS for short is a NP-complete problem, we can show that (1) IS $\in$ NP (2) a known NP-complete problem such as SAT $\rightarrow$ IS. $\Rightarrow$ note the direction of reduction

20 NP: 3-SAT

20.1 Reduction: Input

The 3-SAT problem is defined as follows:

input: a Boolean formula f in CNF with n variables and m clauses where each clause has ≤ 3 literals

output: satisfying assignment if one exists and no otherwise

3-SAT belongs to NP because given Boolean formula f and an True/False assignment σ for $x_1, x_2, \ldots, x_n$, it takes $O(1)$ time to check that at least one literal is satisfied in each clause $C \in f$ since each of them has at most three literals. Hence overall it takes $O(m)$ time to verify the assignment satisfies f.
To show SAT $\rightarrow$ 3-SAT we create input f' for 3-SAT from input f for SAT as follows: for a size-k clause $C = (a_1 \vee a_2 \vee \cdots \vee a_k)$ where $a_1, a_2, \ldots, a_k$ are literals, create $k-3$ new variables $y_1, y_2, \ldots, y_{k-3}$ and replace C with the following $k-2$ clauses:

$$C' = (a_1 \vee a_2 \vee y) \wedge (\bar{y}_1 \vee a_3 \vee y_2) \wedge (\bar{y}_2 \vee a_4 \vee y_3) \wedge \cdots \wedge (\bar{y}_{k-4} \vee a_{k-2} \vee y_{k-3}) \wedge (\bar{y}_{k-3} \vee a_{k-1} \vee a_k)$$

We show that C is satisfiable if and only if C' is satisfiable:

$C \Rightarrow C'$**:** take assignment σ to $a_1, a_2, \ldots, a_k$ that satisfies C, let a_i be the minimal i where a_i is satisfied, since a_i = True the $(i-1)$th clause of C' is satisfied, then we can set $y_1 = y_2 = \cdots = y_{i-2}$ = True to satisfy the clauses before the $(i-1)$th clause and $y_{i-1} = y_i = \cdots = y_{k-2}$ = False to satisfy the clauses after the $(i-1)$th clause

$C' \Rightarrow C$**:** take assignment σ' to $a_1, a_2, \ldots, a_k, y_1, y_2, \ldots, y_{k-3}$ that satisfies C, it suffices to show at least one of $a_1, a_2, \ldots, a_k$ is True, suppose no then from the first clause we have y_1 = True, which in turn implies y_2 = True as a_3 = False, continue this inference until the penultimate clause we have $y_3 = y_4 = \cdots = y_{k-3}$ = True, but then we have $\bar{y}_{k-3} = a_{k-1} = a_k$ = False thus the last clause and hence C' is not satisfied, which is a contradiction, therefore at least one of $a_1, a_2, \ldots, a_k$ is True and thus C is satisfied

Therefore f is satisfiable if and only if f' is satisfiable.

20.2 Reduction: Output

If the polynomial-time 3-SAT algorithm gives us a satisfying assignment σ' for f', then we can simply ignore the auxiliary variables to get a satisfying assignment σ for f.

21 NP: Graph Problems

21.1 Independent Set

For an undirected graph $G = (V, E)$ a subset $S \subset V$ is an **independent set** if no edges are contained in S. The maximal independent set problem is known to be not in NP, because there is no known way to verify a given independent set is maximal in polynomial time. Nonetheless the search version of the independent set problem (cf. knapsack) defined as follows is NP-complete:

input: an undirected graph $G = (V, E)$ and a goal g

output: independent set S with size $|S| \geq g$ if one exists and no otherwise

IS is in NP because given input (G, g) and a solution S

- we can check $\forall x, y \in S$, $(x, y) \notin E$ in $O(n^2)$ time
- we can check $|S| \geq g$ in $O(n)$ time

IS is NP-complete because 3-SAT reduces to IS. Consider a 3-SAT input f with variable $x_1, x_2, \ldots, x_n$ and clause $C_1, C_2, \ldots, C_m$. We create $|C_i|$ vertices for each clause C_i and add the following edges:

clause edge: to encode a clause we add edges between all pairs of the vertices (fully connected) thus

- an independent set S will have at most one vertex per clause in this graph
- since $g = m$ a solution S will have exactly one vertex per clause in this graph, which ensures that we have one satisfied literal in every clause

variable edge: $\forall x_i$ add an edge between x_i (from one clause) and $\bar{x}_i$ (from another clause) to ensure that any independent set contains x_1 or $\bar{x}_1$ or neither thus corresponds to a valid assignment

A 3-SAT input f has a satisfying assignment if and only if the corresponding undirected graph G has an independent set of size $\geq g$, because

$\Rightarrow$: take a satisfying assignment σ for f, for each clause C take one of the satisfied literals and add the corresponding vertex to S thus $|S| = m = g$, moreover S is an independent set because

- S contains exactly one vertex per clause thus it contains no clause edge
- S never contains both x_i and $\bar{x}_i$ thus it contains no variable edge

$\Leftarrow$: take an independent set S of size $\geq g$, S has exactly one vertex per clause because $|S| \geq g = m$, set the corresponding literal to be True so as to satisfy the clause that contains it, since every clause is satisfied the formula f is satisfied, it is a valid assignment because S contains no contradictory literals x_i and $\bar{x}_i$ constrained by the variable edges

The search version of the independent set problem, IS, reduces to the maximal independent set problem, max-IS, because we only need to check whether the maximal size $\geq g$ or not. Since IS is NP-complete, every NP problem has a reduction to max-IS. Hence if we know max-IS is in NP then it is NP-complete. We say such a problem is **NP-hard**. The difference between NP-hard and NP-complete is that NP-complete requires both NP-hard and the fact that the problem is in NP.

21.2 Clique

For an undirected graph $G = (V, E)$, a subset $S \subset V$ is a **clique** if $\forall x, y \in S$, $(x, y) \in E$ i.e. S forms a fully connected subgraph of G. The (search version of) clique problem or Clique for short is defined as follows:

input: an undirected graph $G = (V, E)$ and a goal g

output: $S \subset V$ where S is a clique of size $|S| \geq g$ if one exists and no otherwise

Clique is in NP because

- it takes $O(n^3)$ time ($O(n)$ time for each pair and $O(n^2)$ pairs) to check $\forall x, y \in S$, $(x, y) \in E$ (in fact we can do it in $O(n^2)$ time but $O(n^3)$ is also polynomial time)
- it takes $O(n)$ time to check that $|S| \geq g$

Clique is NP-complete because IS reduces to it. For $G = (V, E)$ we define its opposite graph $\bar{G} = (V, \bar{E})$ where $\bar{E} = \{(x, y) : (x, y) \notin E\}$ so that S is a clique in $\bar{G}$ if and only if S is an independent set in G. Then given input $G = (V, E)$ and goal g for IS, let $\bar{G}$ and g be an input to Clique. If we get a solution S for Clique then we just return S as a solution for IS.

21.3 Vertex Cover

For an undirected graph $G = (V, E)$, a subset $S \subset V$ is a **vertex cover** if it covers every edge, i.e. $\forall (x, y) \in E$, either $x \in S$ and/or $y \in S$. The vertex cover problem or VC for short is defined as:

input: undirected graph $G = (V, E)$ and budget b

output: vertex cover S of size $|S| \leq b$ if one exists and no otherwise

VC is in NP because given input (G, b) and solution S

- it takes $O(n + m)$ time to check for every edge $(x, y) \in E$ at least one of x and y are in S
- it takes $O(n)$ time to check whether $|S| \leq b$

VC is NP-complete because IS reduces to it. The idea is that the complement of a vertex cover form an independent set, that is S is a vertex cover if and only if $\bar{S}$ is an independent set. Then given input $G = (V, E)$ and goal g for IS, let $b = n - g$ and run VC on G and b. If we get a solution S for VC then we return $\bar{S}$ as a solution for IS.
In summary there are roughly two flavors of NP-completeness reductions:

proof by generalization: show that the new problem is more general than a known problem

proof by gadget: modify input by introducing auxiliary variables to the formula or additional structure to the graph

22 Linear Programming: Introduction

22.1 Definition

Linear programming handles any problem that can be formulated as an optimization over a set of variables with a goal known as the **objective function** subject to **constraints** that can be expressed as linear functions of the variables. A problem is a LP problem if

- the objective function is a max or a min of a linear function of the variables
- the constraints are linear functions of the variables

22.2 Geometric View

The key points are

ILP is NP-complete: Linear programming optimizes over a polygon intersection thus is polynomial-time solvable, hence LP $\in$ P. However the corresponding integer-valued problem, that is we only want integer points, also called **integer linear programming** or ILP is NP-complete. As such if the optimum is not an integer value, then we can try to round it to an integer point.

optimum at corner: The optimum always lies at a vertex of the polygon intersection. Suppose no i.e. the optimal line intersects at a point z which is not a corner. But then either one of the two endpoints of the edge z lies in must be better than z, or the optimal line intersects with the entire edge in which case all the points on the edge is optimal. In either case the optimal point lies at a vertex of the polygon intersection.

feasible region is convex: The line connecting any two points in the region is entirely contained in the region. Therefore if a point is better than its neighbors then it is optimal, because the feasible region is below the lines connecting it to its neighbors. Hence the optimum always lies at a vertex of the feasible region. This is the basis for the simplex algorithm.

22.3 Standard Form

The standard form for linear programs has n variables $x_1, x_2, \ldots, x_n$, and

objective function: $\max\{c^T x\}$ where $x = (x_1, x_2, \ldots, x_n)^T$ and $c = (c_1, c_2, \ldots, c_n)^T$

constraint: $x \geq 0$ and $Ax \leq b$ where

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ & & \vdots & \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \quad \text{and} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

The non-negativity constraint is important, because if the feasible region is non-empty then we know that the zero vector is a feasible point. Hence we can trivially find a feasible point, or determine that the feasible region is empty and therefore the LP is infeasible.
To convert an arbitrary linear program into this standard form, we can use the following equivalences

- $\min\{c^T x\} \Leftrightarrow \max\{-c^T x\}$
- $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \geq b \Leftrightarrow -a_1 x_1 - a_2 x_2 - \cdots - a_n x_n \leq -b$
- $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = b \Leftrightarrow a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \leq b$ and $a_1 x_1 + a_2 x_2 + \cdots + a_n x_n \geq b$
- unconstrained $x \Leftrightarrow$ replace x with $x = x^+ - x^-$ where $x^+, x^- \geq 0$

We are in an n-dimensional space because we have n variables. We have $m + n$ constraints including n non-negativity constraints, each of which corresponds to a half-space in the n-dimensional space. The feasible region is the intersection of these $n + m$ half-spaces, which corresponds to a convex polyhedron in the n-dimensional space. The vertices of this polyhedron are the points that satisfy n constraints with equality as well as the remaining m constraints. The number of choices of these n constraints is $\binom{n+m}{n}$, therefore the upper bound of the number of vertices is exponential in n. In addition a neighboring vertex corresponds to swapping out one of the n constraints with equality and swapping in a different constraint. Since we have n choices for which constraint to swap out and m choices for which constraint to swap in, the number of neighbors for a given vertex is bounded by nm.
Strict inequalities are not allowed in linear programming, because for strict inequalities the points on the boundary of the polyhedron do not lie in the feasible region.

22.4 LP Algorithms

The basic idea of the simplex algorithm is to start at some feasible point e.g. the zero vector $x = 0$. Then we do a local search around the feasible point to try to find a neighboring vertex with strictly higher objective value. If we managed to find one then we move there and repeat the local search. If all neighbors are smaller than the current feasible point, by convexity all the feasible region is smaller than it. Hence the current feasible point must be optimal.

23 Linear Programming: Geometry

23.1 Geometry

The optimum of LP is achieved at a vertex of the feasible region except if

the LP is infeasible: the feasible region is empty, resulted from the half-spaces defined by two constraints do not intersect

the LP is unbounded: the optimum is arbitrarily large, resulted from the optimum being a function of the objective

While whether an LP is feasible or infeasible depends only on the constraints, whether an LP is bounded or unbounded depends on the objective function as well.

23.2 Feasibility Check

To find out whether there is some x that can satisfy the constraints $Ax \leq b$ and $x \geq 0$, we introduce a new variable z and consider the following LP problem: $\max\{z\}$ subject to $Ax+z \leq b$ and $x \geq 0$ leaving z unconstrained. To check whether the original LP is feasible, we can run the new LP to check whether the optimal value of z is non-negative. If yes then the point x satisfying the constraints involving z gives a feasible point to the original LP.

24 Linear Programming: Duality

24.1 Duality

To apply the duality we need the primal LP in the canonical form:

objective function: $\max\{c^T x\}$

constraints: $Ax \leq b$ and $x \geq 0$

which gives the following dual LP in canonical form:

objective function: $\min\{b^T y\}$

constraints: $A^T y \geq c$ and $y \geq 0$

While the original LP is maximization the new LP is minimization. While the primal LP has n variables and m constraints besides the non-negativity constraints the dual LP has m variables and n constraints besides the non-negativity constraints.

24.2 Unbounded Check

The **weak duality theorem** states the following: take a feasible point x for the primal LP the objective value of which is $c^T x$; take a feasible point y for the dual LP the objective value of which is $b^T y$. Since any feasible y gives an upper bound on the objective function of the primal LP, we have $c^T x \leq b^T y$. There are two corollaries of this theorem.

corollary #1: if we find a feasible point x for the primal LP and a feasible point y for the dual LP such that $c^T x = b^T y$, then x and y are both optimal

corollary #2: if the primal LP is unbounded then the dual LP is infeasible as nothing can be larger than infinity, similarly if the dual LP is unbounded then the primal LP is infeasible as nothing can be smaller than negative infinity
note that this is not an equivalence, if the dual LP is infeasible then the primal LP can be either unbounded or infeasible

The second corollary above provides a method to check whether an LP is unbounded—if the dual LP is infeasible and the primal LP is feasible (as we know how to check whether an LP is feasible), then the primal LP must be unbounded. The existence of the optimal x and y in the first corollary is mandated by the **strong duality theorem**, which states that the primal LP is feasible and bounded if and only if the dual LP is feasible and bounded, or equivalently the primal LP has an optimal point x^* if and only if the dual LP has an optimal point y^*. By the first corollary of the weak duality theorem we have $c^T x^* = b^T y^*$.

25 Linear Programming: Max-SAT Approximation

25.1 Max-SAT

Max-SAT is the optimization version of the search problem SAT. It is defined as

input: a Boolean formula f in CNF with n Boolean variables $x_1, x_2, \ldots, x_n$ and m clauses

output: the assignment that maximizes the number of satisfied clauses

Max-SAT is NP-hard because it is straightforward to reduce SAT to max-SAT. It is not NP-complete because it is not a search problem thus not in NP. In fact we have no way to verify that the number of satisfied clauses is maximal.

25.2 Approximate Max-SAT

For a Boolean formula f with m clauses, let $m^* = m^*(f)$ be the maximal number of satisfied clauses. Clearly $m^* \leq m$. We will construct an algorithm on input f which outputs l—the number of satisfied clauses (actually an assignment that satisfies l clauses of f), where $l \geq m^*/2$. If this holds for every Boolean formula f, then we call it a $\frac{1}{2}$**-approximation algorithm**.

25.3 Random Approximation Algorithm

Consider an input f in CNF with n Boolean variables $x_1, x_2, \ldots, x_n$ and m clauses $C_1, C_2, \ldots, C_m$. We implement the following random assignment: set $x_i = \text{True}$ with $1/2$ probability and False with $1/2$ probability for all $i = 1, 2, \ldots, n$. Let W be the number of clauses satisfied by this random assignment. Since the assignment is random, so is W. Define a random variable W_j for each clause C_i, which takes value 1 if C_j is satisfied and 0 otherwise. W_j is related to W by $W = \sum_{j=1}^{m} W_j$. Hence the expectation of W can be expressed as

$$\mathbb{E}[W] = \mathbb{E}\left[\sum_{j=1}^{m} W_j\right] = \sum_{j=1}^{m} \mathbb{E}[W_j]$$

where the last equality uses the linearity of expectation, and

$$\mathbb{E}[W_j] = 1 \times \Pr(W_j = 1) + 0 \times \Pr(W_j = 0) = \Pr(W_j = 1) = 1 - \frac{1}{2^k} \geq \frac{1}{2}$$

where k is the number of literals contained in C_j. Because there are 2^k possible assignments to these k literals, and only one of which makes C_j False (i.e. False $\vee$ False $\vee \cdots \vee$ False = False). Plug this into the expectation of W we have

$$\mathbb{E}[W] = \sum_{j=1}^{m} \mathbb{E}[W_j] \geq \frac{m}{2} \geq \frac{m^*}{2}$$

Hence the randomized algorithm achieves 1/2-approximation in expectation.
An important special case is so-called Ek-SAT, which stands for exactly-k-SAT, i.e. every clause has a size of exactly k. For max-Ek-SAT the randomized algorithm achieves a $(1-2^{-k})$-approximation. In particular it achieves a 7/8-approximation for max-E3-SAT. In fact Hovstad proved that this 7/8-approximation is the best possible for max-E3-SAT, because it is NP-hard to do any better than the 7/8-approximation for max-E3-SAT.

25.4 Integer Linear Programming

The canonical form of integer linear programming or ILP differs from that of LP by only one additional constraint—$x \in \mathbb{Z}^n$ or each x_i is an integer. LP has a nice property that there always is a vertex of the feasible region which is an optimal point. Due to the additional integral constraint ILP no longer has that property. Moreover while LP is in P, ILP is NP-hard. To show this we will show that max-SAT reduces to ILP:

1. take input f for max-SAT, for each variable x_i add integral variable y_i to ILP and for each clause C_j add integral variable z_j to ILP so that ILP has $n+m$ integral variables
2. add constraints $0 \leq y_i \leq 1$ and $0 \leq z_j \leq 1$, since $y_i, z_j \in \mathbb{Z}$ they take value 0 or 1, intuitively $y_i = 1$ corresponds to $x_i =$ True and $z_j = 1$ corresponds to C_j being satisfied
3. for each clause C_j let C_j^+ be the positive literals in C_j and C_j^- be the negative literals in C_j, for a Boolean formula f in CNF define an ILP $\max \sum_{j=1}^m z_j$ subject to $0 \leq y_i \leq 1 \ \forall i = 1, 2, \ldots, n$ and $0 \leq z_j \leq 1 \ \forall j = 1, 2, \ldots, m$ and
$$\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j$$
so that
   - if all literals in C_j are unsatisfied, then z_j is forced to take value 0 i.e. the clause is unsatisfied
   - if at least one literal is satisfied, then z_j will take value 1 because $0 \leq z_j \leq 1$, $z_j \in \mathbb{Z}$, and ILP maximizes z_j

25.5 LP Relaxation

Take the ILP max-SAT reduces to and let y^*, z^* be its optimal point. Then m^*, the maximum number of satisfied clauses in f, is given by $m^* = z_1^* + z_2^* + \cdots + z_m^*$. Dropping the integral constraints $y_i, z_j \in \mathbb{Z}$ we convert the ILP to an LP, which can be solved in polynomial time. Let $\widehat{y}^*, \widehat{z}^*$ be the optimal point of the resulted LP. We use **randomized rounding** to convert it into a feasible point for the ILP: since $0 \leq \widehat{y}^* \leq 1$ we can think of it as a probability and set $y_i = 1$ with probability $\widehat{y}^*$ and 0 with probability $1 - \widehat{y}^*$. Consider a clause $C_j = x_1 \vee x_2 \vee \cdots \vee x_k$. The LP constraint for this clause is that $\widehat{y}_1^* + \widehat{y}_2^* + \cdots + \widehat{y}_k^* \geq \widehat{z}_j^*$ which forces the clause to be False when all literals are False. We then apply the arithmetic mean-geometric mean inequality (let $w_i = 1 - \widehat{y}_i^*$)
$$\frac{1}{k} \sum_{i=1}^k w_i \geq \left(\prod_{i=1}^k w_i \right)^{1/k} \qquad \text{for } w_1, w_2, \ldots, w_k \geq 0$$
and the calculus $f(\alpha) = 1 - (1 - \alpha/k)^k \geq \left[1 - (1 - 1/k)^k\right] \alpha$ in $[0, 1]$ to show that the expectation of W is bounded by
$$\mathbb{E}[W] = \sum_{j=1}^m \mathbb{E}[W_j] = \sum_{j=1}^m \Pr(C_j \text{ is satisfied}) \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^m \widehat{z}_j^* \geq \left(1 - \frac{1}{e}\right) \sum_{j=1}^m z_j^* = \left(1 - \frac{1}{e}\right) m^*$$

In summary we can approximately solve an NP-hard problem by

1. reduce it to ILP
2. relax the resulted ILP problem to an LP problem by dropping the integral constraint
3. apply randomized rounding to the optimal LP solution to obtain a feasible point of the ILP

26 NP: Knapsack

26.1 Subset-Sum

The subset-sum problem is defined as

input: positive integers $a_1, a_2, \ldots, a_n$ and integer t

output: output subset S of $\{1, 2, \ldots, n\}$ where $\sum_{i \in S} a_i = t$ if such a subset exists and no otherwise

It is in NP instead because given input $a_1, a_2, \ldots, a_n$ and t and a solution S, it takes $O(n \log t)$ to check that $\sum_{i \in S} a_i = t$ since a_i has at most $\log t$ bits.
To show subset-sum is NP-complete we will show that 3-SAT reduces to it. The input to subset-sum consists of $2n + 2m + 1$ variables $v_1, v_1', v_2, v_2', \ldots, v_n, v_n', s_1, s_1', s_2, s_2', \ldots, s_m, s_m'$ and t. All are $\leq n + m$ digits long, and are base 10 so that there is no carry between digits thus all digits behave independently of each other. Moreover

- v_i corresponds to x_i, and $v_i \in S \iff x_i = \text{True}$
- v_i' corresponds to $\bar{x}_i$, and $v_i' \in S \iff x_i = \text{False}$

thus we need to ensure that exactly one of v_i or v_i' is in S. To do so we can put 1 in the ith digit of v_i, v_i' and t and 0 in the ith digit of all other numbers.
For the remaining digits let the $(n + j)$th digit corresponds to clause C_j so that

- if $x_i \in C_j$ then put 1 in the $(n + j)$th digit for v_i
- if $\bar{x}_i \in C_j$ then put 1 in the $(n + j)$th digit for v_i'

To ensure that at least one of the literals in C_j is satisfied, we put 3 in the $(n + j)$th digit of t and use s_j, s_j' as buffers—put 1 in the $(n + j)$th digit of s_j, s_j' and 0 in the $(n + j)$th digit of all other numbers.
We show that the subset-sum has a solution if and only if the input to 3-SAT is satisfiable

$\Rightarrow$: take a solution S to subset-sum, for the first n digits to get 1 in the ith digit of t we need to include v_i or v_i' but not both, and which one to include depends on the assignment to x_i, hence we get an assignment
for the remaining m digits to get 3 in the $(n+j)$th digit of t, we need to include at least one literal of C_j and use s_j, s_j' if necessary, because S is a solution there must be at least one satisfied literal in C_j, thus C_j is satisfied and hence all clauses are satisfied and we get a satisfying assignment

$\Leftarrow$: take a satisfying assignment for f, if $x_i = \text{True}$ then add v_i to S, otherwise add v_i' to S, hence the ith digit of t is correct
because it is a satisfying assignment each clause C_j has at least one satisfied literal, using this satisfied literal together with the buffers s_j, s_j' we get 3 in the $(n + j)$th digit of t, and therefore we have a solution to subset-sum

27 NP: Halting Problem

27.1 Undecidable

NP-complete describes computationally difficult problems. If P $\neq$ NP then there is no algorithm which takes polynomial time on every input.
Undecidable describes computationally impossible problems. If a problem is undecidable then there is no algorithm that can solve it on every input even given unlimited time and space. Turing proved in 1936 that the halting problem is undecidable on a Turing Machine.