CSE347 Analysis of Algorithms

Albert Peng

May 1, 2024

SP2024 with Prof. Kunal Agrawal

Contents

1 Greedy Algorithms

1.1 Scheduling Problem

Suppose we are given a set P of project, $|P| = n$. Each requests $p_i \in P$ occupies interval $[s_i, f_i)$, where $f_i = s_i + t_i$, where t_i is time required to process a request.

The goal is to choose subset $\Pi \subseteq P$ such that 0

1. No two projects in Π have overlapping intervals.
2. The number of selected projects $|\Pi|$ is *maximized*.

1.2 Greedy Algorithm Proof Outline

1. **Greedy Choice Property:** Show that first choice is not "fatal" i.e. at least one optimal solution contains this. This often uses the *exchange argument*, stating that "if an optimal solution does not choose q, it can be turned into an equally good solution that does."
2. **Inductive Structure:** There is a *strictly smaller* subproblem of the same type, and any feasible solution to the subproblem combined with our previous choice q yield a feasible solution to P.
3. **Optimal Substructure:** If the subproblem is solved optimally, then combining this with our first choice solves the larger problem optimally.

1.2.1 Proof of Scheduling

Proof. **Greedy Choice Property:** Suppose Π^* is an optimal solution for project set P. If $q \in \Pi^*$, we are done. Otherwise, let q be the earlist finish time item and x be the first time in p, so it must be that $f_p < f_x$. Thus $\overline{\Pi^*} := \Pi^* - \{x\} + \{q\}$ is an equally good solution.

Inductive Structure: Let $P' := P - \{q\} - \{\text{projects conflicting with } q\}$, which is a strictly smaller subproblem. Now, if Π is some feasible solution to P', $\Pi + \{q\}$ is a feasible solution to P since q does not conflict with anything in Π'.

Optimal Substructure: Suppose P' is a strictly smaller subproblem and Π' is an instance of its optimal solution.

Putting it all together, proceed by induction on $|P|$. When $|P| = 1$, this is clear. Otherwise for $|P| = n$, we pick greedy choice q. Notice that for the corresponding subproblem P', $|P'| < n$. By the inductive hypothesis, EFT gives an optimal solution to P'. So by the inductive structure and optimal substructure, they provide an optimal solution to P.

■

1.3 Network Problem

Given an undirected, non-negative weighted graph $G : (V, E)$, where $|V| = n, |E| = m$, and $w(e)$ is the weight of edge e.

Our goal is to select a set of edges $T \subseteq E$ such that $\overline{G} = (V, T)$ is connected and sum of weights is minimized.

Observation: T does not contain cycles.

Solution: With Prim's algorithm, the greedy choice is to pick the minimum weight edge.

Lemma: Some optimal solution picks the minimum weight edge.

Proof. GCP: Suppose the lowest weight edge is (u, v). If an optimal solution contains (u, v), we are done. Otherwise if some optimal solution didn't contain (u, v), there must be some path connecting (u, v). If (u, v) is added, this becomes a cycle which is impossible. Thus we can remove the heaviest edge and this is still a minimum spanning tree.

Inductive Substructure: General idea is we can combine the solved vertices.

Optimal Substructure: Clear.

■

Kruskals Algorithm. First, sort edges by weight. Look at edges in order. When looking at e, add e to T if adding it doesn't generate a cycle and continue.

Borurkas

2 Divide and Conquer

Example 1. Consider the problem of multiplying two n-digit numbers. Naively, we multiply each digit of a number with another number, so the running time is $\Theta(n^2)$.

A more efficient solution for multiplication in base 2 can be written for X, Y can be written when X and Y are divided in half into X_h, X_l, Y_h, Y_l. Then,

$$XY = Z_{hh}2^n + Z_{hl}2^{n/2} + Z_{ll}$$

- $Z_{hh} = x_h y_h$
- $Z_{ll} = x_l y_l$
- $Z_{hl} = (x_h - x_l)(y_h - y_l) + Z_{hh} + Z_{ll}$

Then, the complexity becomes $T(n) = 3T(\frac{n}{2}) + \Theta(n)$.

Example 2. Suppose we want to find the minimum distance between a set of n points.

Solution 1. For preprocessing, we sort P by x to get P_x. Let Q be all points left of the midpoint, or $P_x[1, ..., \frac{n}{2}]$. Also let R be all points right of the midpoint, or $P_x[\frac{n}{2} + 1, ..., n]$.

Now for the base case, either there's one point, so closest distance is ∞ or there are two points, and we just find the distance between them.

The *divide step* is where we compute the mid points and get Q, R. The *reursive step* is where d_l =ClosestPair(Q), d_r =ClosestPair(R), and $\delta = \min\{d_l, d_r\}$.

For the *combine step*, we calculate the closet pair such that one point is on the left, one is on the right. However, this takes $\Theta(n^2)$ so the total runtime is $T(n) = 2T(n/2) + \Theta(n^2)$

Here, our "bottleneck" is on the $\Theta(n^2)$ term!

Solution 2. The intuition to this idea we create an δ-radius y-strip. Then, we only need to consider at most 4 points from the other side of the δ strip, since we don't need to consider distances greater than δ. We can write the algorithm as following, preprocessed with sorting of x of $\Theta(n \log n)$

1. Divide P_x into two halves using midpoint $\Theta(1)$
2. Recursively compute d_l and d_r, compute $\delta = \min\{d_r, d_l\}$.
3. Filter points into y-strip. $\Theta(n)$
4. Sort y-strip by y coordinate. $\Theta(n \log n)$
5. For every point p, look at y-strip in sorted order starting at this point and stop when we see a point with y coordinate greater than $p_y + \delta$.

Overall runtime: $T(n) = 2T(n/2) + \Theta(n \log n) \implies \Theta(n \log^2 n)$.

Solution 3. The key idea is that in preprocessing, we have a P_x and P_y. So when we filter points into y-strip, we can look at the points directly.

Overall runtime: $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$

2.1 Maximum Contiguous Subsequence Sum (MCSS)

Problem. Given array of integers $S = \{s_1, ..., s_n\}$, find

$$\text{argmax}_{i,j} \left\{ \sum_{k=i}^{j} s_k \;\middle|\; 1 \le i \le j \le n \right\}$$

Naive algorithm gives $O(n^3)$ time. Using DP gives $O(n^2)$ naively. For an approach using Divide and Conquer, consider the following algorithm:

Algorithm 1 Naive MCSS with Divide and Conquer

function MCSS(S, i, j)
  if $i = j$ **then**
    return $S[i]$
  else
    $mid \leftarrow \lceil \frac{i+j}{2} \rceil$
    L $\leftarrow MCSS(S, i, mid)$
    R $\leftarrow MCSS(S, mid + 1, j)$
    C $\leftarrow$ maximum continuous subsequence including mid
    return $\max\{L, R, C\}$
  end if
end function

Here, calculation of C takes $\Theta(n)$ time when we consider a $\Theta(n)$ calculation of maximum prefix and suffix. So,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n) = \Theta(n \log n)$$

Notice here that C (hence the left suffix and right prefix) can be calculated through *strengthening the recursion* to return the max suffix and prefix values. So, consider the algorithm below in algorithm 2:

Algorithm 2 Modified MCSS with Divide and Conquer

function MCSS(S, i, j)
  if $i = j$ **then**
    $m \leftarrow S[i]$
    $p \leftarrow S[i]$
    $s \leftarrow S[i]$
    $sum \leftarrow S[i]$
    return $S[i]$
  else
    $mid \leftarrow \lceil \frac{i+j}{2} \rceil$
    $(m_l, p_l, s_l, sum_l) \leftarrow MCSS(S, i, mid)$
    $(m_r, p_r, s_r, sum_r) \leftarrow MCSS(S, mid + 1, j)$
    $mcss \leftarrow \max\{m_l, m_r, s_l + p_r\}$
    $mP \leftarrow \max\{p_l, sum_l + p_r\}$
    $mS \leftarrow \max\{s_r, sum_r + s_l\}$
    sum $\leftarrow sum_l, sum_r$
    return $(mcss, mp, ms, sum)$
  end if
end function

Then, this is improved to

$$T(n) = 2T(n/2) + O(1) = O(n)$$

3 Dynamic Programming

3.1 Problem

Problem Statement. Suppose there are 2 string S, T with length n, m respectively. Find the *minimum* number of insertions or deletions to convert S to T.

3.2 General Correctness Proof Outline

Complete Choice Property. THe solution makes *one* of the choices that we consider.

Inductive Structure. Once you make *any* choice, you are left with a smaller problem of the same type. *Any* first choice and a *feasible* solution to the subproblem = feasible solution to the entire problem.

Optimal Substructure. If we optimally solve the subproblem for *particular choice* c, and combine it with c, the resulting solution is the optimal solution that makes choice c.

3.3 Problem 2

For a given set of positive numbers S, is there a subset $X \subseteq S$ such that for a positive integer K,

$$\sum_{x \in X} x = K$$

Clearly, a brute force solution by trying all subsets gives 2^n time, for $|S| = n$.

Note that our algorithmic runtime will depend on K, and it is considered *pseudo-polynomial.*

Now, consider ordering elements of S arbitrarily. Then at every time, we have recursive calls to $\texttt{SS}(S[1, ..., m-1], K)$ and $\texttt{SS}(S[1, ..., m-1], K - S[m])$. Then, make a $k \times n$ cell where $S[i, j] = true$ if we can construct a subset with the first j elements of S to form sum i.

4 Max Flow

4.1 Max Flow Problem

This problem states that for a directed and weighted graph (V, E), let there be unique source and sink nodes s, t. Suppose each edge has a capacity $c(e)$. Then, a **valid flow** assigns integer $f(e)$ to each edge such that the following are satisfied:

- **Capacity Constraint**: $0 \leq f(e) \leq c(e)$
- **Flow Conservation**: $\sum_{e \in E_{in(v)}} f(e) = \sum_{e \in E_{out}(v)} f(e)$, $\forall v \in V \setminus \{s, t\}$.

We want to compute the maximum flow, or $|F| = \sum_{e \in E_{out(s)}} f(e) = \sum_{e \in E_{in(t)}} f(e)$

4.2 Ford-Fulkerson (FF) Algorithm

The idea is the we have a residual graph G_R that stores the remaining capacity of backward flow, which is equivalent to the amount of flow. With each flow on original edges, we increase the flow by increasing the remaining capacity of backward flow. Formally:

1. Initialize the residual graph $G_R = R$
2. Find some augmentating P with capacity k
3. Push flow so that $c(e) - = k, c(\tilde{e}) + = k$, where e is original edge and $\tilde{e}$ is mirror edge. The c here denotes remaining capacity, differing from notatation above.
4. Repeat 2 and 3 until no augmenting path can be found.

Correctness Proof. We induct on augmenting paths. For the base case, $f(e)$ on all edges, so capacity constraint and flow conservation are satisfied.

Then suppose by induction there is a valid flow and we found a path P and we can push flow k onto the path. We look at capacity constraints and flow conservation.

For an edge e in P, if it is a forward edge, by construction we can have k pushed on it and there are capacity remaining. Now if e is a back/mirror edge with remaining residual capacity $\geq k$, then this means that the flow of the forward edge is greater than k, so we can reduce k and still be positive.

For flow conservation, we consider four scenarios for any vertex v on P.

1. *Both forward edges*: k is added to in and out, constraint satisfied.
2. *Both back edges*: k reduced to both in and out, constraint satisfied.
3. *Forward edge in, mirror edge out*: We actually "redirected" the flow k originally from the mirror edge into the forward edge. So the input remains the same.
4. *Mirror edge in, forward edge out*: Similar to above, we redirected the output.

Note that FF *terminates* since every time it finds an augmenting path, the total flow increases.

Runtime: For each iteration, it takes $\Theta(m + n)$ to find a path using BFS or DFS, and this is a maximum of $|F|$ times. So this becomes $\Theta(|F|(m + n))$ ■

We will use the max-flow min-cut theorem to prove the optimality of FF!

4.3 Graph Cut

Definition. A **graph cut** is a partition of vertices into 2 subsets S and T, where $s \in S, t \in T$. The **cut capacity** is defined as the sum of capacity of edges from vertex to S to vertex in T.

$$C(S) = \sum_{u \in S, v \in T} c(u, v)$$

Lemma. For all valid flows f, $|f| \leq C(S) \forall$ cut S. This means that $|f| \leq C(S^*)$, where S^* denotes cut of the smallest capacity.

Proof. All flow must go through one of the cut edges. ■

Lemma. FF prouces a flow $f = C(S^*)$.

Proof. Let $\hat{f}$ be the flow found by FF such that no more paths can be augmented in G_R. This means that there is no path from s to t with remaining capacity > 0. Now, define $\hat{S}$ as all the vertices that can be reached from s using edges with capacities > 0.

Here $t \notin \hat{S}$ because otherwise there is another path. Note that this means that all paths from $\hat{S}$ to $V \setminus \hat{S}$ have remaining capacity 0. In other words, all forward edges are saturated, and all backward edges are empty. In particular, empty backward edges means that no flow is going into $\hat{S}$.

Therefore $|\hat{f}| = C(\hat{S})$, and so $\hat{S}$ is the minimum cut, and $|\hat{f}|$ is the maximum cut. ■

4.4 Bipartite Matching

Suppose we have n classes and n rooms and we want to match classes to different rooms with different properties.

Formally, we have bipartite graph $G = (V, E)$ where vertices are either in L or R, and edges only go between vertices of different sets. For a **matching**, we find a subset $M \subseteq E$ such that each vertex has at most one edge from M incident on it, and we want the matching of the largest size. We want to *reduce* this to maximal flow problem.

Reduction Solution. Given $G = (V, E)$, we want to construct a graph $G' = (V', E')$ where

$$\text{max-flow}(G') = \text{max-matching}(G)$$

Construct s that connects to all vertices in L and t that connects with vertices in R, and the capacity on all edges is 1.

Claim. G' has a flow of k if and only if G has matching of size k¿

Correctness Proof. Suppose G has matching of size k, then we can extend the edges to s and t to create flow of size k.

For the other direction, suppose G' has flow of size k. Consider any vertex $v \in L$. Its input is at most 1, so its output is at most one. Similar for $v \in R$. ■

4.5 Lecture

Consider an image segmentation problem where for each pixel i, we denote p_i as the probability that $i \in A$, the foreground object. For every pair of adjacent pixels, $c_{i,j}$ is the cost of placing boundary between i and j. In this case, a **segmentation** would be assignment of pixel to A (foreground) or B (background). The goal is to maximize

$$A^*, B^* = \text{argmin}_{A,B} \left(\sum_{i \in A} p_i + \sum_{i \in B} (1 - p_i) - \sum_{i,j \in \text{ boundary}} c_{i,j} \right)$$

Using the fact that $\sum_{i \in A} p_i + \sum_{i \in A} (1 - p_i) + \sum_{i \in B} p_i + \sum_{i \in B} (1 - p_i) = N$, where N is the total number of pixels, this is equivalent to minimizing

$$\sum_{i,j \in \text{ boundary}} c_{i,j} + \sum_{i \in A} (1 - p_i) + \sum_{i \in B} p_i$$

We can construct a graph as following:

1. Cut graph into S, T from a connected graph of s, t, and a node for each pixel p_i.
2. The edge from s to all pixels are denoted by p_i, with edge from pixels to t denoted as $(1 - p_i)$.
3. For each pair of adjacent pixels, connect them with an edge of weight $c_{i,j}$.

$$\sum_{i \in S} (1 - p_i) + \sum_{i \in T} p_i + \sum_{i \in S, j \in T \text{ or vice versa}} c_{i,j}$$

We claim that the image segmentation cost C if and only if graph G has a cut of cost C.

Proof. ($\Longleftarrow$) : Suppose graph G has a cut of cost C. Then, put all pixels representations by vertices in S in A. If $i \in S$ in G's cut, then put pixel i in A. If $i \in T$ in G's cut, put pixel i in B. Thus this equals the cost of image segmentation.

($\Longrightarrow$) : Clear. Given image segmentation of cost C, put node i in S if $i \in A$ and put it in T if $i \in B$ ■

Thus the algorithm to solve for this problem is clear:

1. Construct correspondingn graph G.
2. Use FF to calculate max-Flow.

3. Use result to compute min-cut.
4. Reconstruct segmentation.

5 Reduction

In general, mapping from one problem to another known solvable problem which preserves solutions is called **reduction**. Formmally, problem L reduces to K ($L \leq K$, which means L is no harder than K)if there is a mapping Φ from **any** instance $l \in L$ to some instance $\Phi(l) \in K' \subset K$, such that $\Phi(l)$ can lead to a solution to l. This can be written into following steps:

1. Reduction: $l \to \Phi(l)$
2. Solve problem $\Phi(l)$
3. Convert the solution for $\Phi(l)$ to l.

5.1 Efficient Reduction

A reduction $\Phi : L \to K$ is **efficient** ($L \leq_p K$) if for any $l \in L$:

- $\Phi(l)$ is computable from l in polynomial l times.
- Solution to l is computable from solution of $\Phi(l)$ in polynomial ($|l|$) times.

We call L **poly-time reducible** to K, or L poly-time reduces to K.

Theorem. If $L \leq_p K$ and there is a polynomial time algorithm to solve K, then there is a polynomial time algorithm to solve L.

Proof. It suffices to show that $|\Phi(l)| = poly(|l|)$, this the problem reduces to $|poly(l)| + poly(\phi(l))$.

Since we can construct $\Phi(l)$ using $poly(l)$ time and at every step we can only write constant data, it has to be that $poly(|l|) = \Phi(l)$ ■

5.2 Canonical Decision Problem

Does an instance $l \in L$ of an optimization problem have a feasible solution with objective value k? For maximization, this means that our objective $\geq k$ and for minimization, $\leq k$. We denote this easier, decision version of the problem as DL.

Clearly, $DL \leq_p L$. Suppose (l, k) is an instance of DL, and we have $\Phi(l) \in L$, where $\Phi(l)$ gives objective value $v^*(l)$, and we can just check if this is greater than k. Thus, DL is clearly no harder than L.

Lemma. If $v^*(l) = O(c^{|l|})$ for constant c, then $L \leq_p DL$.

Proof. We can consider DL as a blackbox that returns true or false given (l, k). Naively, we can run linear search and go through every value, which is not efficient. Rather, we can try an exponential binary search, which is at $\log(v^*(l)) = poly(l)$ ■

Note that this means that we can use the **contrapositive** for the statement. If K is *easy*, then L is easy. If L is hard, then K is *hard*

Example. Independent Set Problem. Suppose we are given undirected graph $G = (V, E)$, we find a subset of vertices V such that no two vertices of S are connected by an edge. We believe this to be a **hard** problem and we can take this as given.

Example. Vertex Cover. Given undirected graph $G = (V, E)$, a subset $C \subseteq V$ is called a *vertex cover* if C contains at least one end point of every edge so that $\forall (u, v) \in E$, either $u \in C$ or $v \in C$. We want $VC(G, j)$ to return G is G has a vertex cover of size $\leq j$, or false otherwise.

Claim: $ISET \leq_p VC$. Note we *don't* need to show $VC \leq_p ISET$. We show that VC of size $|V| - k$ iff there exists ISET of size k.

Lecture

Proof for ISET$(G, k) \leq_p$ VC(G', q). The idea is that we want to show a construction of (G, k) to (G', q). First, need to show that if G has an ISET of size k, then G' has VC of size q. For the other direction, it suffices to prove that for the G' previously of size q (not arbitary), then G has a ISET of size k.

In particular, we related them by $q = |V| - k$. Thus for this direction, assume S is a vertex set with $|S| = k$. Also let $C = V - S$, and claim that C is a vertex cover with $|C| = |V| - k = q$. Assume by contradiction $\exists (u, v)$ not covered. Then $u, v \notin C \implies u, v \in S$, so S not ISET, a contradiction.

Otherwise, suppose $\exists$ VC of size $|V| - k$ denoted as C. Then suppose by contradiction $S = V - C$ is not an independent set, then $\exists (u, v)$ such that $u, v \in S$, $u, v \notin C$, and (v, u) not covered. So C not a VC, a contradiction.

Remark: When we say polynomial time, it usually refers to polynomial in **size** of inputs measured by bits. This means that particular emphasis should be placed on size of values within problems.

6 P/NP

Definition. **Polynomial-time Solvable**, or **P** is a class of decision problems L such that there is a polynomial-time algorithm that correctly answers yes or no for every instance $l \in L$

NP does not refer to non-polynomial. Rather, this means **non-deterministic poly-time**. In particular, let l be an instance of decision problem L that the answer happens to be "yes". A **certificate** $c(l)$ for l is a "proof" that the answer for l is true.

Example. We want to know if there is a path from s to t. The *instance* would be a $G(V, E), s, t$ for l. The *certificate* would be the path.

For a problem to be in NP, it needs to have "useful" certificates. Namely, it should be

- Not too long (poly(l))
- Easy to check

Definition. A **verifier algorithm** is one that takes instance $l \in L$ and certificate $c(l)$ and returns true if certificate prove l is a true instance and false otherwise.

V is a poly-time verificate for L if it is a verfier and runs in $poly(|l|, |k|)$ time. This runtime must be polynomial.

Thus, **NP** is a class of decision problems such that there exists certificate schema c and a verifier algorithm V such that

1. certificate is poly(l) in size.
2. V is poly(l) in size.

In other words,

- P: class of problems you can solve in polynomial time.
- **NP**: class of decision problems we can verify true instances in polynomial time given polynomial time certificate.

Claim. P $\subseteq$ NP

Proof. Let L be a problem in P, we want to show that $\exists$ polynomial size certificate with polynomial time verifier.

By assumption, there is an algorithm A which solves L in polynomial time. In this case, the certificate is empty. For verfier (l, c), we can run A on L and return the answer. ■

Definition. A **NP-hard** problem is a decision problem L where $\forall K \in NP$, $K \leq_p L$. This means that L is at least as hard as all the problems in NP Thus solving L means able to solve any problem in NP with polynomial cost.

Lemma. Let L be an NP-hard problem. If $L \in P$, then $P = NP$.

Proof. Say L has a poly-time solution, then we have a poly-time solution for all K in NP, so $NP \subseteq P$ ■

Definition. L is **NP-complete** if it is both NP-hard and $L \in NP$.

L is called **NP-optimization** if the canonical decision problem is NP complete. If any NP optimization is solvable, $NP = P$.

NP Complete Problem: Satisfiability (SAT). Suppose we have a set of Boolean variables and boolean formulas consisting of AND, OR, and NOT. Given a formula Φ, is there a setting M of variables such that Φ evaluates to True under this setting. If there is such an assignment, then Φ is satisfiable. This is shown to be NP-Complete.

In general, SAT is NP-complete as following:

- $\exists$ certificate schema and a polynomial time verifier. Namely, c satisfies the assignment and v checks that M makes Φ true.
- SAT is NP-hard. Accept this as a fact.

Then we can show a problem L to be NP-complete by:

1. Show $L \in NP$ by having a poly(l) certificate schema and verification algorithm.
2. Prove $SAT \leq_p L$, i.e., solving L solves SAT and thus solves all problems in NP in poly-time

Meanwhile, there's an easier version of SAT called **Conjugate normal form, CNF**, where the formula ϕ is an "and of ors". Each variable or its negation is called a *literal*, the literals are combined into *clausers* using `or`, and there are `and` of clauses. This is NP complete.

In fact, 3-SAT, where every clause has 3 literal, is also NP complete. There are n variables, m clauses for a total of $3m$ literals.

Example. ISET is NP-Complete.

Proof. Suppose we have a problem L.

1. ISET $\in$ NP. Certificate would be given set of k vertices, and verification checks that there are no edges that have the same vertex.
2. Show ISET is NP-hard by proving 3SAT $\leq_p$ ISET by constructing a reduction from 3SAT to ISET, and showing that ISET is harder than 3SAT.

Reduction mapping: Make a vertex for each literal, and connect the vertex within each clause. All variables are connected to all their negations. ■

6.1 Lecture

Proof outline for ISET NP-hard

For $a \in A, b \in B$, let there be a reduction satisfying

1. reduction take poly time

2. a is true $\iff$ b is true. In other words, solution ψ is satisfiable iff g has independent set of size k.

7 7

Fill in from video

7.1 Class Lecture

Scheduling Problem. Suppose there are n jobs and each job has a release time r_i, deadline d_i, and a continuous executaion time of t_i (non-preemptive.) We want to find a way to schedule all such jobs.

First, prove SCHED is in NP.

Then show that SCHED is NP-hard; in particular, reduce form $SSS \leq_P SCHED$. Formally, denote $Y \subseteq S$ such that $\sum a_i \in Y a_i = t$, where t is the target. Then let the execution time be a_i for each number, and let X be the sum of all numbers in S. Then it suffices to have original jobs and a new created job with starting time t and deadline $t + 1$ with execution time 1. Then, this suffices as a valid subset sum solution to the original problem.

Component Grouping. Let there be an undirected graph not necessarily connectd. We define a *component* as a connected subgraph. For this graph G, we want to check is there is a subset of connected components of size k.

Note that we can also naively reduce this from SSS, by making components of corresponding sizes. However, note that this reduction is NOT polynomial time and thus would not work.

8 Approximation Algorithms

Recall: NP-optimization problems are problems where the canonical DP is NP-complete. There are things we can do to avoid this:

- Solve small instances
- Hard instances are rare and we can perform average case analysis
- Consider special cases

For **approximation algorithms**, we find a "good" solution in polynomial time but may not be optimal. Question is, how do we quantify the quality of the solution?

We use **approximation ratio for an algorithm** A for an NP-optimization L. For an instance l A finds solution $c_A(l)$ when the optimal solution is $c^*(l)$. The ratio would

be either

$$\max_l \frac{c_A(l)}{c^*(l)} = \alpha \qquad \min_l \frac{c_A(l)}{c^*(l)} = \alpha$$

The greedy VC $O(|E|)$ where we go through every edge that is not covered always finds a 2-approximation.

Example. Consider a maximum cut problem with $G = (V, E)$, undirected and unweighted. Let a cut partition vertices into S and $V \setminus S$, and size of cut be number of edges that cross from S to $V \setminus S$.

We can have a LOCAL-CUT algorithm, where we start with an arbitrary cut of G. While we can move a vertex v from its partition to the other while increasing size of the cut, do that.

It turns out that this is a $1/2$ approximation with $O(|E||V|^2)$ time.

8.1 Lecture

Set Cover. Suppose there is a set X of all cards and pack j where $S_J \subseteq X$. One can either buy whole pack or not. Let the collection of sets be Y. The goal is pick a C such that $\cup_{S_i \in C} S_i = X$, and minimize $|C|$

Greedy solution. Initialize $C = \emptyset$, and while X is not covered, let U_{i-1} be the elements not covered after picking $i-1$ set. For each unpicked set S, compute a $\delta(S) = U_{i-1} \cap S$. We want to pick the S_i such that $\delta(S_i)$ is maximized.

Using harmonic sum, we have that $H_n := \sum_{i=1}^{n} \frac{1}{i} = \Theta(\log n)$. Suppose d is the number of elements in the largest S_i. We claim that GSC has an approximate ratio of H_d.

First, the proof claims that GSC is $\ln(m)$ approximation where $m = |X|$. Denote C^* as optimal solution and C as greedy solution. Let (m_i) be a decreasing sequence counting the number of remaining uncovered elements, such that m_0 is initialized as all elements of X. For a round i, the set that covers the most amount of elements must cover at least $\frac{m_{i-1}-1}{|C^*|}$ elements. Thus we have

$$m_i \leq m_{i-1}\left(1 - \frac{1}{|C^*|}\right) \implies m_i = m_0\left(1 - \frac{1}{|C^*|}\right)$$

Now suppose that after j rounds, there is only one more uncovered elements, so this can be written as

$$1 \leq m\left(1 - \frac{1}{|C^*|}\right) = m\left(\left(1 - \frac{1}{|C^*|}\right)^{|C^*|}\right)^{\frac{j}{|C^*|}} \leq me^{-\frac{|C^*|}{j}} \implies j \leq |C^*| \log m$$

Thus we can write $|C| \leq |C^*| \ln m + 1$

To further reduce this, consider the cost of our algorithm to be $|C|$ and if an element was added in step i when set S^i is picked, then we covered $\delta(S_i)$ elements by adding

one set, where the cost of each of the elements is $\frac{1}{\delta(S_i)}$. If element x was covered in step i, the then cost is $c(x) = \frac{1}{\delta(S_i)}$. The total cost of GSC is $\sum_{x \in X} C_x = \sum_{i=1}^{C} \delta(S_i) \cdot \frac{1}{\delta(S_i)} = |C|$.

Now, the claim is that for a set S, the cost paid by GSC to cover all elements of S is at most $H_{|S|}$, so that $\sum_{x \in S} c(x) = H_{|S|}$. Suppose GSC covers S in the order of $x_1, ..., x_{|S|}$. Then when x_j is covered, $x_j, ..., x_{|S|}$ are uncovered. At this point, if S was picked, it would cover $|S| - j + 1$ elements. In other words, $\delta(S^i) \geq |S| - j + 1$. Thus the cost is given by $C(x_j) = \frac{1}{\delta(S_i)} \leq \frac{1}{|S|-j+1}$. Therefore summing voer elements of S, we have

$$\sum_{i=1}^{|S|} c(x_j) = \sum_{i=1}^{|S|} \frac{1}{|S| - j + 1} = \frac{1}{|S|} + ... = H_{|S|}$$

Going back, we can write

$$|C| = \sum_{x \in X} c(x) \leq \sum_{x \in C^*} \left(\sum_{x \in S_k} c(x) \right) = \sum_{S_k \in C^*} H_{|S_k|} \leq \sum_{S_k \in C^*} H_d = H_d \sum_{S_k \in C^*} 1 = H_d \cdot |C^*|$$

9 Randomized Algorithms

The idea of hashing with chaining is that we have integers in range $U = [0, N-1]$. We want to map them into a hash table with m slots. The goal is to hash a set $S \subseteq U, |S| = n$.

When multiple values are hashed to the same slot, it is know as a *collusion* and we have a list with all elements. Thus the runtime of insert, query and delete of elements is Θ(length of chain). The *worst case* is when all elements are in the same bucket, making the cost $\Theta(n)$. We therefore want chains to be $\Theta(1)$ as long as $|S|$ is reasonably sized. We want S to hash **uniformly**.

Simple Uniform Hashing assumes that the n elements is picked uniformly from U, and we therefore see that this function works fine. Note that this is a very unrealistic assumption.

Consider an adversarial model that sees the hash generating code and generates S so that the hash table behaves badly. Thus, we use **randomized algorithms** that makes random inner choices. There are two kinds of random algorithms:

- *Monte Carlo:* Running time is fixed, may be wrong or bad quality
- *Las Vegas*: Answer is always correct, but sometimes algorithms may run slowly.

In the hasing example, we will be randoming the *choice* of hash function h from a family of hash functions H. If we randomly pick a hash function from this family, then the probability that the hash function is bad on any particular input sequence is small. Thus, the adversary cannot pick a bad input.

Suppose we have m buckets and input $x \neq y$. Ideally, we want it so that $\mathbb{P}(x, y \text{ collides}) = \frac{1}{m}$

Definition. H is a **universal family** of hash functions if $\forall x \neq y, x, y \in \{0, ..., N-1\}$,

$$\mathbb{P}_{h \in H}\left(h(x) = h(y)\right) = \frac{1}{m}$$

Lemma. Given a universal hash function family H and $S = \{a_1, ..., a_n\} \subseteq \mathbb{N}$. For any probability $0 \leq \delta \leq 1$, if $n \leq \sqrt{4m\delta}$, the chance that no two keys hash to the same slot is $\geq 1 - \delta$.

Proof. First, the idea is to compute *expected value* of collision.

For $x, y \in S$, define an indicator random variable

$$C_{x,y} = \begin{cases} 1, & \text{if } h(x) = h(y) \\ 0, & \text{otherwise} \end{cases}$$

Then, we have

$$\mathbb{E}[C_{x,y}] = 1 \cdot \frac{1}{m} + 0 = \frac{1}{m}$$

Now, define C_x as the random variable that represents the cost of inserting/searching/deleting x from hash table, and $C_x \leq$ total number of elements that collide with x. Then we have

$$C_x = \sum_{y \in S, x \neq y} C_{x,y} \implies \mathbb{E}[C_x] = \mathbb{E}\left[\sum_{y \in S, x \neq y} C_{x,y}\right] = \sum_{y \in S, x \neq y} \frac{1}{m} = \frac{n-1}{m}$$

This means that $\mathbb{E}[C_x] = \Theta(1)$ if $n = O(m)$

Suppose C is the total number collusions, so $C = \sum_{x \in S} C_x/2$ since each collusion is counted twice, and

$$\mathbb{E}[C] = \frac{1}{2}\sum_{x \in S} \mathbb{E}[C_x] = \frac{1}{2}\sum_{x \in S} \frac{n-1}{m} = \frac{n(n-1)}{m} \leq \frac{n^2}{2m}$$

Then if we want $\mathbb{E}[C] < \delta$, we can have $n = \sqrt{2m\delta}$.

Recall the markov inequality equation $\mathbb{P}[X \geq k\mathbb{E}[X]] \leq \frac{1}{k}$. Applying on C,

$$\mathbb{P}\left(C \geq \frac{1}{\delta}\mathbb{E}[C]\right) < \delta \iff \mathbb{P}\left(C \geq 1\right) < \delta \iff \mathbb{P}[C = 0] > \frac{1}{\delta}$$

■

More generally, for a universal hash function family, if the number of keys $n < \sqrt{Bm\delta}$, the probability that at most $B + 1$ keys hash to the same slot $> 1 - \delta$

Example. Consider quicksort, which is based on partitioning and assumes all elements are distinct. For a partition $A[p...r]$, it picks a pivot $A[r]$ and rearranges into $A_{low} < A[r] < A_{high}$. This rearrangement can be done in linear time, where the quicksort algorithm returns the pivot.

We assume the adversary knows our algorithm but doesn't know about our random choices. Note that it is important the pivot position cannot be determined. $T(n) = T(n-c) + \Theta(n) = \Theta(n^2)$ and $T(n) = \frac{9}{10}T(n) + \frac{1}{10}T(n) + \Theta(n) = \Theta(n \log n)$.

Therefore if we select pivot variable randomly and have k be the rank of the proof, then we can write $T(n) = T(k) + T(n-k-1) + \Theta(n)$. Then,

$$\begin{aligned}
\mathbb{E}[T(n)] &\leq \mathbb{E}[T(k) + T(n-k-1) + cn] \\
&= \mathbb{E}[T(k)] + \mathbb{E}[T(n-k-1)] + \mathbb{E}[cn] \\
&= cn + \sum_{j=0}^{n-1} \mathbb{P}[j = k]T(j) + \sum_{j=0}^{n-1} \mathbb{P}[j = n-k-1]T(j) \\
&= cn + \sum_{j=0}^{n-1} \mathbb{P}[j = k]\mathbb{E}[T(j)] + \sum_{j=0}^{n-1} \mathbb{P}[j = n-k-1]\mathbb{E}[T(j)] \\
&= cn + \frac{2}{n}\sum_{j=0}^{n-1} \mathbb{E}[T(n)]
\end{aligned}$$

This is thus at worst $\Theta(n^2)$, and we want to show that this is $\Theta(n \log n)$. We want to show that $T(n) = c'n \log n + 1$.

Base Case: When $n = 1, T(1) = 1$ and this holds true.

Inductively, if this holds true for $j < n$,

$$\begin{aligned}
T(n) = \frac{2}{n}\sum_{j=0}^{n-1} T(j) + cn &= \frac{2}{n}\sum_{j=0}^{n-1}(c'j \log j + 1) + cn \\
&= \frac{2}{n}c'\sum_{j=0}^{n-1} j \log j + \frac{2}{n}\sum_{j=0}^{n-1} 1 + cn \\
&= \frac{2c'}{n}\left(\frac{1}{2}n^2 \log n - \frac{1}{8}n^2\right) + 2 + c'n \\
&= c'n \log n - \frac{1}{4}c'n + 2 + c'n \\
&\leq c'n \log n + 1
\end{aligned}$$

where the third line equality follows from a given equation. Thus we conclude this is $\Theta(n \log n)$

10 Online Algorithm

Elevator Example. Suppose I want to go to the top floor as quickly as possible. There is a elevator that takes E time to get to the top. There are also stairs that take S time to climb, $S > E$. However, I don't know when the elevator arrives.

This is online in the sense that I am missing information. If it is offline, I would know the elevator arrives in T.

Definition. An **online algorithm** must make decisions *without* full information about the problem instance, and/or it doesn't know the future. An **office algorithm** has full information about the problem instance.

We use **competitive ratio** to quantitfy quality of online algorithms. Consider a minimization problem L and l be an instance, so that $C^*(l)$ is the optimal offline solution and A is the online algorithm, with $C_A(l)$ being its solution. Then, an online algorithm is α-competitive if $\frac{C_A(l)}{C^*(l)} \leq \alpha \, \forall l$. In other words, $\alpha = \max_l \frac{C_A(l)}{C^*(l)}$.

In the elevator problem, if stairs is always taken, the ratio could be $\frac{S}{E}$. If we always wait for elevator, T could be infinity since the elevator could break. Rather, we want to wait for R time and take stairs if it has not arrived. If we take $R = S$, the competitive ratio is 2.

Proof. The optimal solution takes the elevator if $T + E \leq S$

- If OPT takes elevator, so $T < S - E$, we also take elevator. CR = 1.
- If OPT takes stairs immediately, OPT=S, and we take at worst $2S$ time. Then, CR=2.

■

In fact, consider $R = S - E$.

- If OPT takes elevator, we also take elevator, CR=1.
- If OPT takes stairs, we wait $S - E$ and then take stairs, and $CR = 2 - \frac{E}{S}$.

Then, the competitive ratio is $2 - \frac{E}{S}$. Here, waiting for more or less time is no good.

Example. Cache. Data is organized in blocks. If CPU accesses data in cache, it is called a cache hit. If CPU accesses data not in cache, it is a cache miss. Then if a block needs to be brought into cache from main memory, and the cache is already full, another block needs to be kicked out.

If we know the order of access, simple kick out the one farthest in the future.

An online algorithm is that we can kick out the least recently used (LRU) cache. We claim that LRU is k-competitive.

Proof. Split up the sequence into subsequences such that each subsequence contains $k+1$ distinct blocks. With LRU in each subsequence, there can be at most $k+1$ misses. In particular, note that the last element must be different from prevous ones.

With OPT, there is at least 1 miss in each distinct block.

In fact, this can be improved so that LRU has at most k misses with blocks of size k. ■

Many sensible algorithms are k-compeittive. In fact, there is a lower bound where no deterministic algorithm can be better than k-competitive.

Rather, a solution we can have is to use **resource augmentation**. Let the offline algorithm have k cache lines and online algorithm have ck cache lines, $c > 1$. Then, we claim that the *competitive ratio* would be $\sim \frac{c}{c-1}$

Proof. Let LRU have cache of size ck. Divide the sequence into subsequence such that there are ck distinct pages. So there are less than ck cache misses for each subsequence. Meanwhile, OPT has cache of size k, and there are greater than $(c-1)k$ cache misses for each subsequence.

So CR$\leq \frac{c}{c-1}$. The actual competitive ratio is actually $\frac{c}{c-1+\frac{1}{k}}$, so for $c = 1$, this is k.

■

11 11

Randomized Caching Suppose there is a cache of k blocks and a sequence of accesses σ. The cost of a randomized caching algorithm is the expected number of cache misses on σ.

$$\text{Competitive Ratio} = \max_{\sigma} \frac{\mathbb{E}[\text{Cost of alg A on } \sigma]}{\text{Cost of OPT on } \sigma}$$

Randomized Marking Algorithm: We let every block in cache have a 1 bit mark.

- Initially, all blocks are unmarked, so all bits are set to 0.
- When a block is accessed, it is marked and the bit is set to 1.
- We want to always randomly pick out of the unmarked block to evict.
- Once all blocks in the cache are marked, unmarked all blocks

Claim. This randomized marking algorithm has competitive ratio $\log k$, where k is the number of cache blocks. No deterministic algorithm can be better than k.

Proof. We divide the sequence into phases, where phase starts when nothing is marked, and ends when a miss causes everything to get unmarked. We observe that every

phase has exactly k distinct block accesses. Also, all pages originally in cache are still in cache at the end.

Now, for phase j let **old pages** be blocks that were in cache at the beginning of the phase, and new blocks be ones brought into cache in phase j but are not old.

Now, we find the lower bound. Let $m^*(\sigma)$ be the number of misses, and we claim that $m^*(\sigma) \geq \frac{1}{2}\sum_j n_j$, where n_j is the number of new pages in phase j. In phase 1, $m^*(1) \geq 1$. For phase j, note that phase $j-1$ accessed k distinct pages. Then phase j has n_j new pages, so the two pages has $k + n_j$ different pages combined. So,

$$m^*(j-1) + m^*(j) \geq n_j \implies \underbrace{m^*(0)}_{=0} + 2\sum_j m^*(j) \sum_j n_j \implies m^*(\sigma) \geq \frac{1}{2}\sum_j n_j$$

Rest of the proof done in class. ■

Min-cut Example. Suppose there is an undirected graph $G = (V, E)$, find $S \subset V$ such that edges from S to $V - S$ is minimized.

Randomized Algorithm: Initialize sets $S(v) = \{v\}$ for all v. Repeat for $|V| - 2$ times:

- Randomly choose edge $e = \{u, v\}$
- Delete edges between $S(u)$ and $S(v)$
- Merge $S(u)$ and $S(v)$

When 2 sets remain, return the sets. We claim that this randomized algorithm returns minimum cut with $p > 1/\binom{|V|}{2} = 2/(|V|(|V|-1))$.

Proof. Suppose the min cut have k edges, labeled with set F. ■

11.1 Lecture

Denote **old blocks** to be blocks that weren't in cache at thte end of phase $j-1$., and **new blocks** be blocks accessed in phase j that are not old.

Now, suppose $m^*(j)$ is the number of misses by OPT in phase j, and we claim that $m(\sigma) \geq \frac{1}{2}\sum_j n_j$, where σ is the entire sequence.

Proof. Assume in phase 1 the cache starts empty, then clearly $m^*(1) \geq k$. In phase $j-1$, k unique pages are accessed. At $j+1$, it accessed n_j new pages. The two combined accesses $k + n_j$ unique pages.

In other words, from the start of phase $j-1$ to the end of j, there must be at least n_j unique misses.

$$m^*(j-1) + m^*(j) \geq n_j \implies m^*(0) + m^*(1) + \implies m^*(1) + ... \geq \sum_j n_j$$

Since $m^*(0) = 0$ and this ends with 0, $2\sum_j m^*(j) \geq \sum_j n_j \implies \sum_j m^*(j) \geq \frac{1}{2}\sum_j n_j$.

Now we claim that the upper bound is $\mathbb{E}[m(\sigma)] \leq H_k \leq \sum_j n_j$.

Recall that competitive ratio is $\leq \frac{H_k \sum_j n_j}{\frac{1}{2}\sum_j n_j} = 2H_k = O(\lg k)$.

Considering phase j, there are usually some misses due to old pages. Note that there are 1 miss per new block n_j.

Suppose that naively, within phase j, we first have n_j new blocks, look at some number O of old blocks, and meet X. Let n be the number of pages, and this claims that the probability $\mathbb{P}[X$ evicted before accessed] is $\frac{n}{k-O}$. Prof. Agrawal gives the following proof:

Write

$$\mathbb{P}[X \text{ evicted}|\, n \text{ new pages}, o \text{ old pages}, k \text{ unmarked pages}] := \mathbb{P}[X|\, n, O, k]$$

Case 1 is when 1st access evicts X.

Case 2 is when the 1st access evicts one of old pages that are accessed before X.

Case 3 is when the 1st access evicts one of the other old pages

So,

$$\mathbb{P}[X|n, O, k] = \frac{1}{k} + \frac{O}{k} \cdot \mathbb{P}[X|n, O-1, k-1] + \frac{k-O-1}{k} \cdot \mathbb{P}[X|\, n-1, O, k]$$

Prove by induction that $\mathbb{P}[X|n, O, k] = \frac{n}{k-O}$.

For base cases, consider all three variables. Inductively, assume true for values smaller than n, O, k, and we have

$$\mathbb{P}[X|n, O, k] = \frac{1}{k} + \frac{O}{k}\left(\frac{n}{k-O}\right) + \frac{k-O-1}{k}\left(\frac{n-1}{k-1-O}\right) = \frac{1}{k}\left(\frac{on + nk - on}{k-0}\right) = \frac{n}{k-O}$$

Now let χ_i be the indicator variable denoting whether the ith page was evected (1 if true and 0 otherwise), and $\mathbb{E}[\#$ cache misses due to old pages$] =$

$$\mathbb{E}\sum_{i=0}^{k-n_j} \chi_i \leq \sum_{i=1}^{k-n_j} \mathbb{E}[\chi_i] = \sum_{i=1}^{k-n_j} \frac{n_j}{k-(i-1)} = n_j\left(\frac{1}{k} + \frac{1}{k-1} + \cdots\right)$$

So the expectation of number of misses in phase j equals

$$= n_j\left(1 + \frac{1}{k} + \frac{1}{k-1} + \cdots + \frac{1}{k-(k-n_j)+1}\right) \leq n_j\left(1 + \frac{1}{2} + \cdots + \frac{1}{k}\right) = n_j H_k$$

■