**CS 38 Introduction to Algorithms** **Week 10 Recitation Notes**

# Final Review

**TAs:** Chenchao You (cyyou@caltech.edu), Milan Cvitkovic (mcvitkov@caltech.edu)

## 1 The Big Picture

We're covered a lot of material this term. But broadly speaking, there were two main themes:

1. **Design and Analysis of Algorithms.** This course attempts to give you a set of algorithmic tools and teach you how, and in what scenarios, to deploy them.
2. **Important Problems.** Studying algorithms reveals that a host of seemingly disparate problems are really just instances of a few important problems. This course has explored a number of such problems: packing, scheduling, path–finding, linear programming, etc.

Depending where your interests fall on the practical/theoretical spectrum, you'll probably care about one of these more than the other. But whatever your interests, we hope this course has given you a foundation for creating and understanding solutions to computational problems.

The structure of this final review will be to go over basic definitions, concepts, and examples from the majority of the topics we've covered in this course. If you are confused about any of the material below, we encourage you to go back through the lecture notes or work through some example problems from CLRS.

## 2 Dynamic programming

Dynamic programming (DP) is a method of building up solutions to problems from solutions to smaller instances of the same problem. Typically, the problems for which DP is useful are optimization problems. For DP to be applicable, the optimization problem must have two features:

1. Optimal substructure: The optimal value of the problem can easily be obtained given the optimal values of subproblems. In other words, there is a recursive algorithm for the problem, which would be fast if we could just skip the recursive steps.
2. Overlapping subproblems: The subproblems share sub–subproblems. In other words, if you actually ran that naïve recursive algorithm, it would waste a lot of time solving the same problems over and over again.

If you have a problem on your hands which has optimal substructure but not overlapping subproblems, then divide and conquer is the strategy you are looking for. (Divide and conquer just says to use the recursive algorithm — it works well!) But when your problem has both the above features, the DP method proceeds as follows:

1. Iterate through all subproblems, starting from the "smallest" and building up to the "biggest." For each one:
   (a) Find the optimal value, using the previously–computed optimal values to smaller subproblems.
   (b) Record the choices made to obtain this optimal value. (I.e. if multiple smaller subproblems were considered as candidates, record which one was chosen.)

2. At this point, we have the *value* of the optimal solution to this optimization problem (the length of the shortest edit sequence between two strings, the number of activities that can be scheduled, etc.) but we don't have the actual solution itself (the edit sequence, the set of chosen activities, etc.). Go back and use the recorded information from the 1(b) steps to reconstruct the optimal solution.

"Smallest" and "biggest" don't necessarily refer literally to problem size (though they usually do). By "smallest" subproblems we mean the base–cases of the recursion, whatever those may be. What matters is that whenever the algorithm is computing the solution to a subproblem, it should already have solved the sub–subproblems that are needed for solving that subproblem.

The basic formula for the runtime of a DP algorithm is

$$\text{Runtime} = (\text{Total number of subproblems}) \times \begin{pmatrix} \text{Time it takes to solve problems} \\ \text{given solutions to subproblems.} \end{pmatrix}$$

(Warning: sometimes, a more nuanced analysis is needed.) Some important DP algorithms from CS38:

- Two different algorithms for Knapsack, an NP-complete problem.[1]
  - The first algorithm (from Lecture 3) breaks up the Knapsack problem into $nW$ total subproblems, where subproblem $(n', W')$ is the problem of choosing maximum-value items among the first $n'$ with weight at most $W'$. Each subproblem can be solved in constant time given the solutions to smaller subproblems, leading to a runtime of $O(nW)$.
  - The second algorithm (from Problem Set 4) breaks up the Knapsack problem into $nV$ total subproblems, where subproblem $(n', V')$ is the problem of choosing minimum-weight items among the first $n'$ with value at least $V'$. Each subproblem can be solved in $O(n)$ time given the solutions to smaller subproblems, leading to a runtime of $O(n^2V)$.
- $k$-center in 1–dimension. The algorithm in Lecture 2 breaks up the problem into $O(nk)$ subproblems, with subproblem $(t, j)$ being the problem of clustering the first $t$ items using $j$ centers. Each subproblem could be solved in $O(\log n)$ time given the solutions to smaller subproblems, leading to a runtime of $O(nk \log n)$.
- Edit distance. The algorithm in Lecture 2 breaks up the problem into $O(nm)$ subproblems, with subproblem $(k, \ell)$ being the problem of determining the edit distance between the length-$k$ prefix of $X$ with the length-$\ell$ prefix of $Y$. Each subproblem can be solved in $O(1)$ time given the solutions to smaller instances, leading to a runtime of $O(nm)$.
- The CYK algorithm (the solution to Problem 1 on Problem Set 2), for determining membership in context-free languages. The algorithm from problem set 2 breaks up the problem into $O(n^2)$ subproblems, with subproblem $(A, i, j)$ being the problem of determining whether $A$ yields the substring of $u$ starting at location $i$ and ending at location $j$. Each subproblem can be solved in $O(n)$ time given the solutions to smaller instances, leading to a runtime of $O(n^3)$.

**Exercise 1.** *Given $a_0$ balls of type A, $b_0$ balls of type B, and $c_0$ balls of type C, count how many ways there are to arrange the balls so that no balls of the same type are adjacent.*

**Algorithm Idea**. Let $f(a, b, c, A)$ be the number of ways to arrange $a$ balls of type $A$, $b$ balls of type $B$, and $c$ balls of type $C$ such that that last ball is of type $A$. Define $f(a, b, c, B)$ and $f(a, b, c, C)$ similarly. We can produce the following recursive definition for $f$:

$$\begin{cases} f(a,b,c,A) &= f(a-1,b,c,B) + f(a-1,b,c,C) \\ f(a,b,c,B) &= f(a,b-1,c,A) + f(a,b-1,c,C) \\ f(a,b,c,C) &= f(a,b,c-1,A) + f(a,b,c-1,B) \end{cases} \tag{1}$$

[1] Incidentally, this is an important theme of the course: when you hit an NP-complete problem, it's not time to give up! It's just time to start looking for approximation algorithms, algorithms which are fast for your instance distribution, heuristics, etc.

Because, we know (in the case of $f(a, b, c, A)$) that the penultimate element must be a $B$ or $C$. We leave it as an exercise to calculate appropriate base cases and prove correctness.

**Complexity.** We would need to store a table of size $3a_0b_0c_0 = O(a_0b_0c_0)$. Each problem is calculable in $O(1)$ given subproblems so the total runtime is $O(a_0b_0c_0)$.[2]

# 3 The greedy method

The greedy method is a simple technique: build up a solution one piece at a time, always choosing to add the "best" available piece to the solution. This is obviously an informal statement of the method, and it can take some cleverness to figure out what the right greedy algorithm to use on a problem is. But more often, the tricky part of using the greedy strategy is understanding whether it works! (Typically, it doesn't.)

One source of problems for which the greedy method makes sense and works is problems on matroids. If you can interpret the elements of the problem you're trying to solve as a matroid, then the greedy algorithm is guaranteed to be optimal.

**Definition 3.1.** *A matroid is a pair $(U, \mathcal{F})$, where $U$ is a finite set and $\mathcal{F}$ is a collection of subsets of $U$, satisfying*

- *(Non-emptyness) There is some set $I \in \mathcal{F}$ (equiv. $F \neq \emptyset$)*
- *(Hereditary axiom) If $I \subseteq J$ and $J \in \mathcal{F}$, then $I \in \mathcal{F}$.*
- *(Exchange axiom) If $I, J \in \mathcal{F}$ and $|I| > |J|$, then there is some $x \in I \setminus J$ so that $J \cup \{x\} \in \mathcal{F}$.*

We refer to $U$ as the *universe* and to sets in $\mathcal{F}$ as *independent sets.* A *basis* of a matroid is a maximal independent set, i.e. an independent set $B$ such that if $x \notin B$, then $B \cup \{x\}$ is not independent.

Here are a some examples of a matroid to help you conceptualize them:

- The universe $U$ is a finite set of vectors. We think of a set $S \subseteq U$ as independent if the vectors in $S$ are linearly independent. A basis for this matroid is a basis (in the linear algebra sense) for the vector space spanned by $U$.
- The universe $U$ is again a finite set of vectors. But this time, we think of a set $S \subseteq U$ as independent if $\text{Span}(U \setminus S) = \text{Span}(U)$. (This is the *dual matroid* to the previous matroid. See Problem Set 3.) A basis for this matroid is a collection of vectors whose *complement* is a basis (in the linear algebra sense) for $\text{Span}(U)$.
- The universe $U$ is the set of edges in some connected, undirected graph. The independent sets $\mathcal{F}$ are sets of edges with no cycles.

A *weighted matroid* is a matroid $(U, \mathcal{F})$ together with a weight function $w : U \to \mathbb{R}$. In the minimum-weight matroid basis problem, we are given a weighted matroid, and we are asked for a basis $B$ which minimizes $\sum_{u \in B} w(u)$. For the minimum-weight matroid basis problem, the following greedy algorithm works:

[2] An additional exercise is to extend this problem to $n$ different types of balls. Formally, there are $a_1$ balls of type $A_1$, ..., $a_n$ balls of type $A_n$. A similar argument would show then there are $O(n \prod a_i)$ subproblems and each problem takes $O(n)$ time to run, yielding total complexity $O(n^2 \prod a_i)$.

**Matriod Greedy Algorithm.**

1. Initialize $B = \emptyset$.
2. Repeatedly add to $B$ the minimum-weight point $u \in U$ such that $B \cup \{u\}$ is still independent, until no such $u$ exists.
3. Return $B$.

## 3.1 Approximation algorithms

When you are faced with an NP-hard problem, you shouldn't hope to find an efficient exact algorithm. (Though if you do, call the Turing Award Committee!) But you can hope for an approximation algorithm. Often, a simple greedy strategy yields a decent approximation algorithm. Some examples:

- For the set cover problem, the simple greedy algorithm is to always pick the set covering the largest number of uncovered elements. If the smallest cover has size $k$, then the greedy algorithm finds a cover of size at most $k \log n$.
- For the metric Steiner tree problem (Lecture 6), a simple MST-based algorithm gets within a factor of two.
- You showed on Problem Set 4 that the Knapsack problem can be approximated arbitrarily well in polynomial time, by scaling down the problem instance (effectively forgetting the lower-order bits) to a regime that can be handled efficiently.

**Exercise 2** (Bin Packing)**.** *Given $n$ items with integer weights $w_1, \ldots, w_n$ and bins of integer capacity $c$, assign each item to a bin such that the total number of used bins is minimized. It may be assumed that all items have weights smaller than the bin capacity.*

This problem can be shown to be NP-hard[3]. We are going to concern ourselves with a greedy 2-factor approximation algorithm.

**Algorithm.** When processing the next item, check if it fits in the same bin as the last item. Use a new bin only if it does not.

**Complexity.** Each item requires $O(1)$ processing so total time is $O(n)$.

**Correctness.** Consider any two adjacent bins in our greedy solution. The sum of the weights of the items in these bins must be $> c$. Otherwise, we would have fit the items into 1 bin. So if our greedy algorithm used $k$ bins, and $W = \sum_i^n w_i$ is the total amount of weight, it must be that $\frac{k}{2}c < W$ for $k$ even or $\frac{k-1}{2}c < W$ for $k$ odd.

Now, if $m$ is the optimal solution (the minimum number of bins one could possibly use), then $W \leq cm$. Combining these inequalities shows that the greedy algorithm is within a factor of 2 of optimal.

## 3.2 Clustering

An important example of greedy approximation that we focused on is from clustering:

**Definition 3.2.** *A finite metric space is a pair $(V, w)$, where $V$ is a finite set (the set of "points") and $w : V \to \mathbb{R}$ is a function (the "distance function") such that*

[3] As an additional exercise, convince yourself of this!

- *(Non-negativity)* $w(x, y) \geq 0$ *with equality only if* $x = y$.
- *(Symmetry)* $w(x, y) = w(y, x)$.
- *(The triangle inequality)* $w(x, z) \leq w(x, y) + w(y, z)$.

**Definition 3.3.** *A ball centered at a point* $x$ *with radius* $r$ *is the set* $B(x, r) = \{v \in V : w(x, v) \leq r\}$. *A* $(k, r)$*-covering of the space is a set of centers* $x_1, \ldots, x_k \in V$ *such that* $V = \cup_i B(x_i, r)$ *(i.e. every vertex is within distance* $r$ *of some* $v_i$*.) A* $(k, r)$*-packing is a set of center vertices* $y_1, \ldots, y_k$ *such that the balls* $B(y_i, r)$ *are pairwise disjoint.*

In first computational problem associated with clustering (the $k$-center problem), we are given $k$ (and the metric space), and we are supposed to compute

$$R_{\text{cover}}(k) := \inf\{r : \exists (k, r)\text{-cover}\} \tag{2}$$

(The inf (short for infimum) of a set is its greatest lower bound. For finite sets it's the same as min.) It is NP-hard to solve this problem exactly. But the G/H-S algorithm (Lecture 8) gets within a factor of 2 by a simple greedy strategy.

In a *dual* computational problem, we are given $k$, we and we are asked to compute

$$R_{\text{pack}}(k) := \sup\{r : \exists (k, r)\text{-packing}\}. \tag{3}$$

(sup is supremum: the least upper bound.) The same G/H-S algorithm gets within a factor of 2 for this problem as well.

The "inverse" computational problems are similar: we are given $r$, and we are supposed to compute either

$$K_{\text{cover}} := \min\{k : \exists (k, r)\text{-cover}\} \tag{4}$$

or

$$K_{\text{pack}} := \max\{k : \exists (k, r)\text{-packing}\}. \tag{5}$$

Weak duality tells us that $K_{\text{pack}}(r) \leq K_{\text{cover}}(r)$, and $R_{\text{pack}}(k + 1) \leq R_{\text{cover}}(k)$.

# 4 Divide and Conquer

The divide and conquer strategy is essentially a simpler version of dynamic programming. Divide and conquer is applicable when your problem exhibits optimal substructure, but not overlapping subproblems. In such a case, the divide and conquer strategy is the "naïve recursive algorithm" that works poorly in the DP setting:

1. Divide your problem into smaller subproblems.
2. Recursively solve those subproblems.
3. Combine the solutions into a solution of the original problem.

When analyzing the runtime of such an algorithm, one is usually faced with a recurrence of the form $T(n) = aT(n/b) + f(n)$. Here, $n$ is the size of the original problem, $n/b$ is the size of the subproblems, $a$ is the number of immediate subproblems needed, and $f(n)$ is the amount of work done (ignoring recursive steps) to solve a problem of size $n$. The solutions to such recurrences are given by the so-called "master theorem."

**Theorem 4.1** (Master Theorem (CLRS) or Ducks and Horses Theorem (Schulman)). *If $T(n)$ satisfies the above recurrence relation, then if*

- *if $\exists\ c > 1, n_0$ such that for $n > n_0$, $af(n/b) \geq cf(n)$ then $T(n) \in \Theta(n^{\log_b a})$*
- *if $f(n) \in \Theta(n^{\log_b a})$ then $T(n) \in \Theta(n^{\log_b a} \log n)$*
- *if $\exists\ c < 1, n_0$ such that for $n > n_0$, $af(n/b) \leq cf(n)$ then $T(n) \in \Theta(f(n))$*

Important examples of divide and conquer algorithms we've covered:

- Two different sorting algorithms (Lecture 12):
  - Mergesort divides the list into the left half and the right half, sorts them recursively, and them merges the two sorted lists. The runtime is given by $T(n) \in 2T(n/2) + \Theta(n)$, which has solution $T(n) \in \Theta(n \log n)$.
  - Quicksort divides the list into items less than a pivot and items greater than a pivot, and then sorts the halves recursively. If the pivot is chosen arbitrarily, the runtime is $\Omega(n^2)$ in the worst case. If the pivot is always chosen to be the median (which can be found in linear time via another divide and conquer algorithm), the runtime is again $O(n \log n)$.
- The FFT (Lecture 13): converts a polynomial between the coefficient basis and the evaluation basis in $O(n \log n)$ time. Corollary: polynomials can be *multiplied* in $O(n \log n)$ time.
- Strassen's algorithm (Lecture 14): matrix multiplication in $O(n^{2.81})$ time.[4]

**Exercise 3** (L Tiling). *Given a $n \times n$ grid where $n := 2^k$, with a square missing, find an algorithm to cover the grid with L tiles. An L tile is a $2 \times 2$ tile with 1 square missing.*

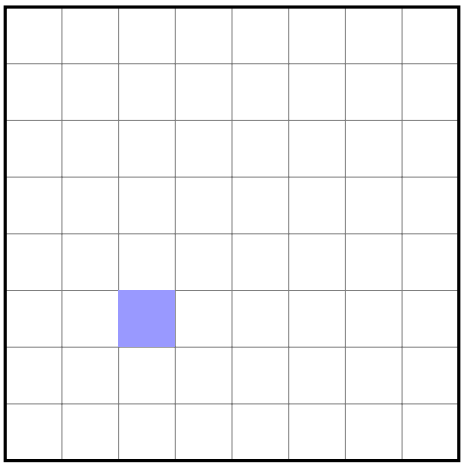

Figure 1: Case of $n = 8$, with some tile missing.

[4]Here 2.81 is a rounding of $\log_2 7$ as we can argue that the multiplication of a $2 \times 2$ matrix by another actually only requires 7 multiplications.

**Algorithm.** Divide the square into 4 $(n/2) \times (n/2)$ grids. One of the grids is missing a square. Remove a square from each of the other 3 quadrants by removing the inside corner pieces that form an L tile (see figure). Then recursively solve each of the quadrants. For a base case, if $n = 2$ then place the logical L tile.

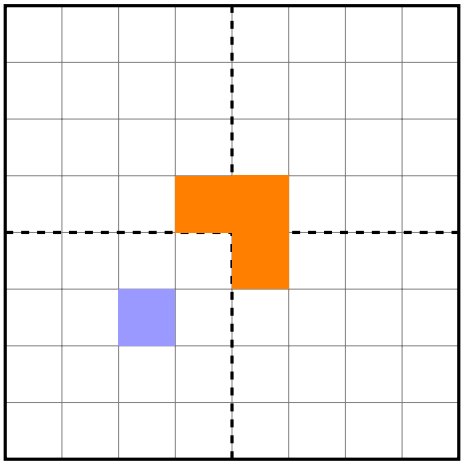

Figure 2: 1 recursion of the algorithm

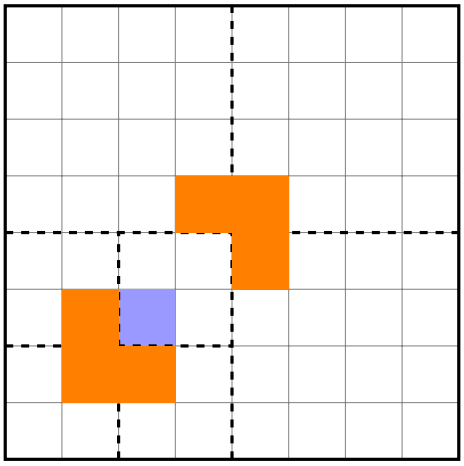

Figure 3: 2 recursions of the algorithm

**Correctness.** We will argue existence of a solution as a corollary that the algorithm's correctness. We proceed by induction. For base case, $k = 1$ $(n = 2)$. The solution here is trivial: place the L tile in the 3 available squares. For induction assume correctness up to $k - 1$. By removing the L piece as suggested in the algorithm, each of the 4 $(n/2) \times (n/2) = 2^{k-1} \times 2^{k-1}$ sub-grids has 1 square removed. By induction, we have a solution for each of the sub-cases and therefore the problem.

**Complexity.** We can argue that $T(n) = 4T(n/2) + C$ where $C$ is the constant time to remove the initial L piece and make the recursive calls. By the Master Theorem, this yields $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$. This is at the theoretic bound, as it takes $O(n^2)$ time to write out all the L shapes.

# 5 Streaming Algorithms

We only briefly touched on them in this course, but streaming algorithms are an important class of algorithm that deal with problems with huge inputs.

In the streaming model we assume the input is given to the algorithm in the form of a streamed list $x_1, ..., x_n$ with $x_i \in \Sigma$ and $n$ usually assumed to be very large. The goal of a streaming algorithm is to have small space complexity while performing some computation on the input stream.

Examples of streaming algorithms include:

- Finding a majority symbol in a stream by keeping track of a single symbol and how many times it's been seen (Lecture 14). The key is the pigeonhole–principle idea that the majority symbols in the stream can't all be paired with a non–majority symbol, and the only way the algorithm could output a non–majority symbol is if this were the case.
- Finding all symbols that occur $> n/q$ times in $x$ for some $q \geq 2 \in \mathbb{N}$. One can give an algorithm to do this using $O(q \log n)$ space (Lecture 14). One can also show that it's impossible to do this without false positives using less than $\Omega(|\Sigma| \log \frac{n}{|\Sigma|})$ space. The key to the latter is remembering that the algorithm is deterministic, and therefore the number of different values it can output is bounded by the number of different configurations of its space, which is exponential in the size of its space.
- Finding the largest sum of contiguous elements in $x$ (Problem 5, Problem Set 6).

# 6 Graph algorithms

Some important graph algorithms covered:

- BFS and DFS (breadth-first and depth-first search) are greedy strategies for exploring graphs in $O(n+m)$ time. These are versatile building blocks for other algorithms. For example, BFS gives shortest paths from a designated source vertex in an unweighted graph, and DFS gives a topological sort of a directed acyclic graph.
- Kruskal's algorithm and Prim's algorithm are both greedy algorithms for finding minimum spanning trees. The key insight that makes both of them work is: Suppose we have a partially-built MST, i.e. we have a set of edges $\mathcal{F}$ which is contained in some MST. Let $e$ be the lightest edge such that $\mathcal{F} \cup \{e\}$ is still acyclic. Then $\mathcal{F} \cup \{e\}$ is still contained in some MST. With optimizations, Prim's algorithm runs in time $O(m + n \log n)$.
- Two algorithms for the "single source shortest paths" problem, where we are given a weighted graph and a distinguished "source" vertex and asked for shortest paths to all other vertices:
  - Djikstra's algorithm is a greedy algorithm which only works if the weights are nonnegative. We keep track of a "tentative distance" for each vertex from the source. At every step, we greedily select the closest vertex from the source, and use it to update ("relax") the tentative distances of its neighbors. With optimizations, the runtime is $O(m + n \log n)$.
  - Bellman-Ford works even with negative-weight edges, and detects negative-weight cycles (which make the problem ill-defined.) The runtime is $O(nm)$.
- The Floyd-Warshall algorithm solves the "all-pairs shortest-paths" problem, where we are looking for shortest paths between all pairs of vertices in a weighted graph. It runs in time $O(n^3)$.
- Savitch's algorithm tests $s$-$t$ connectivity in a directed graph ("Is there a directed path from $s$ to $t$?") using only $O(\log^2 n)$ space, via "middle-first search." The idea is, if there's a path, then that path has a midpoint, so we can just iterate over all candidate midpoints and then recursively check whether that candidate midpoint works.

# 7 Flow

**Definition 7.1.** *A flow network (or capacitated network, or transportation network) consists of*

- *A set $V$ of vertices.*
- *A capacity function $C : V \times V \to \mathbb{R}^+$. (Note that the capacity is effectively associated with a directed edge.)*
- *Distinguished vertices $s, t \in V$; $s$ is the source and $t$ is the sink.*

**Definition 7.2.** *An $s$-$t$ flow is a function $f : V \times V \to \mathbb{R}$ (think of $f(u, v)$ as telling how much material flows from $u$ to $v$ in unit time) such that*

- *(Skew symmetry) $f(u, v) = -f(v, u)$*
- *(Flow can only be created/destroyed at the source/sink. Everywhere else, inflow = outflow.) If $u \notin \{s, t\}$, then $\sum_v f(u, v) = 0$.*
- *(Capacity constraints) $f(u, v) \leq C(u, v)$.*

The *value* of a flow is defined by $U(f) = \sum_v f(s, v)$. (This is just the amount of flow coming out of $s$, or going into $t$.) The basic computational problem associated with flow is: we are given a flow network, and we are asked to find the flow with maximum value.

An $s$-$t$ cut of the network is a subset $S \subseteq V$ such that $s \in S$ and $t \notin S$. (Effectively, this is a partition of the vertex sets into two pieces, one of which contains the source and the other of which contains the sink.) The capacity of the cut $S$, denoted $C(S, S^c := T)$, is the sum of the capacities of edges which go from the source side of the cut to the sink side of the cut.

Every cut presents a *barrier* to flow: if we want to squeeze $U$ units of flow from $s$ to $t$, we're somehow going to have to push them from $S$ to $S^c$, so we must have $U \leq C(S, S^c)$. (This is a "weak duality" statement.) This elegant idea is the basis for the *max-flow min-cut theorem*, which asserts that cuts are the *only* barriers to flow.

But the better version of max-flow min-cut is in terms of the *residual network.* For a flow $f$, we define $G_f$ to be the flow network with the same vertex set, source, and sink, but with the capacity $C_f(u, v) = C(u, v) - f(u, v)$. The residual network indicates where there's potential room for improvement of $f$ – it shows which edges can have more flow pushed across them.

An *augmenting path* for $f$ is a path from $s$ to $t$ along edges with positive capacities in the residual network. If there exists an augmenting path for $f$, then $f$ is not a max-flow (since we can send additional flow down the augmenting path.) Again, the beautiful fact is that the existence of augmenting paths is the *only* reason that a flow is not maximum:

**Theorem 7.1** (Max-Flow Min-Cut, Expanded Edition)**.** *Fix a flow network $G = (V, C, s, t)$ and a flow $f$. The following are equivalent:*

1. *$f$ is a max flow.*
2. *There is some $s$-$t$ cut $S$ such that $C(S, S^c) = U(f)$.*
3. *There is no augmenting path for $f$.*

One way of looking at this theorem is that it says that the *greedy strategy* works for finding max-flow: just keep sending more flow from $s$ to $t$ until there is no route along which we can send any more flow. This algorithm is called the Ford-Fulkerson method; it takes exponential time in the worst case. Specifically it runs in $O(E|f|)$ (if the capacities are integers). By choosing augmenting paths a little more carefully (e.g. use the *shortest* augmenting path), one obtains the Edmonds-Karp algorithm, which runs in time $O(VE^2)$.

**Exercise 4** (Bipartite Matching)**.** *You run a cupcake shop that sells $D$ different flavors of cupcake. You currently have $c_j \in \mathbb{N}$ of the $j$th flavor of cupcake in stock for $j = 1, ..., D$.*

*$N$ manic, sugar–crazed children (and their exhausted guardians) have just entered your shop. Each child is picky and will only eat certain flavors of cupcake: let $p_i \subseteq \{1, ..., D\}$ denote the subset of cupcake flavors the $i$th child will eat for $i = 1, ..., N$.*

*Cupcakes are indivisible, and each child is allowed at most 1 cupcake. Each child $i$ who does not receive a cupcake from their set $p_i$ will scream. Find an allocation of your current cupcake stock to children that will minimize the number of screaming children in your shop. You must do so in time $O(N^2D)$ time or everyone will scream.*

**Algorithm.** Construct a graph $G$ with source vertex $s$, $N$ vertices $u_i$ for $i = 1, ..., N$ representing children, $D$ vertices $v_j$ for $j = 1, ..., D$ representing cupcake types, and sink vertex $t$.

Add a capacity 1 edge from $s$ to $u_i$ for all $i$; add a capacity 1 edge from $u_i$ to $v_j$ for each $j \in p_i$ and $i = 1, ..., N$; and add a capacity $\min\{c_j, N\}$ edge from $v_j$ to $t$ for $j = 1, ..., D$.

Run Ford–Fulkerson to compute max flow $f$ on $G$. Give child $i$ a cupcake of flavor $j$ if and only if $f(u_i, v_j) = 1$.

**Correctness.** By construction, $G$ forces the cupcake assignment to satisfy all required constraints: each child can receive at most 1 cupcake since $outflow(u_i) \leq incapacity(u_i) = 1$ for all $i$; at most $c_j$ cupcakes of flavor $j$ can be given out since $inflow(v_j) = outflow(v_j) \leq outcapacity(v_j) \leq c_j$ for all $j$; and by executing Ford–Fulkerson we guarantee all values in $f$ are integers. We also note that setting the capacity of the $v_j$ to $t$ edges to be $\min\{c_j, N\}$ does not affect the solution, since at most $N$ cupcakes can be given out due to the construction of the $s$ to $u_i$ edge capacities.

The number of screaming children equals the number of $s$ to $u_i$ edges that have flow 0, since all such edges have flow either 0 or 1, and by construction of $G$ any node $u_i$ with inflow 1 implies child $i$ received a cupcake of some flavor in $p_i$. We know the number of such edges is minimized, since if a greater number of edges could have been given a flow of 1 without violating a problem constraint, doing so would increase the total value of the flow, and Ford–Fulkerson would have produced such a flow. Thus this algorithm minimizes the number of screaming children.

**Complexity.** The number of edges is $O(ND)$ and the largest capacity is $\leq N$ by construction, so Ford–Fulkerson runs in $O(N^2D)$.

# 8 Linear Programming

A linear program is a linear optimization problem. This means that the function being optimized (the "objective function") and the constraint function are linear. The traditional notation for a linear program is:

$$(\mathcal{P}) = \begin{cases} \max & c^T x \\ \text{s.t.} & Ax \leq b \\ & x \geq 0 \end{cases} \tag{6}$$

Here $x \in \mathbb{R}^n$ is a vector over the reals; $c \in \mathbb{R}^n$ defines objective function $c^T x$; $A \in \mathbb{R}^{m \times n}$ and $b \in \mathbb{R}^m$ define the constraint function $Ax \leq b$. Each row of $A$ with the corresponding entry of $b$ is considered to be one constraint. The standard form is not a loss of generality: any linear optimization problem can be written in

the standard form.

The feasible region of an LP is defined as the set of $x$ such that $Ax \leq b$ and $x \geq 0$. It is a *convex polyhedron*. A feasible LP is called *unbounded* if its objective function can be made arbitrarily large while satisfying the constraints. For an LP, a local optimum is necessarily a global optimum. This can also be paraphrased as "the greedy strategy works". Furthermore, if an LP is feasible and bounded, then there is some vertex of the polyhedron at which the optimal value is achieved.

The standard LP problem is solved. The solution is called the Simplex algorithm (although there are many other solutions as well) and it was covered in the lecture notes. We won't review it here in depth, but the overview is the following:

1. Create an auxiliary LP, to test for feasibility and to find a feasible vertex if the original LP was feasible.
2. Introduce slack variables to convert inequalities to equations.
3. Slide from vertex to vertex in such a way as to increase the objective function at each step.
4. Stop when increasing any variable would decrease the objective function.

The simplex algorithm works well in practice, though instances can be cooked up on which it takes exponential time. The simplex algorithm also lends insight into the structure of LPs, through *LP duality*. The dual of a LP problem written in the primal form as in (6) is precisely:

$$(\mathcal{D}) = \begin{cases} \min & y^T b \\ \text{s.t.} & y^T A \geq c \\ & y \geq 0 \end{cases} \tag{7}$$

There are two statements that can be made relating the primal and dual LP problems.

**Theorem 8.1** (Weak LP Duality)**.** *If both the primal and the dual LPs are feasible, then the value of the primal is at most that of the dual.*

**Theorem 8.2** (Strong LP Duality)**.** *For any primal LP in standard form, precisely one of the following situations happens:*

1. *Both the primal and the dual are infeasible, or*
2. *One is unbounded and the other is infeasible, or*
3. *Both are feasible and bounded, and they have the same optimal value.*

Strong LP Duality is (arguably) the deepest theorem in CS38, and it has lots of important consequences in many areas of mathematics. For example, max-flow min-cut is a special case of the Strong LP duality.

It's very important that you understand how to write an optimization problem as an LP (assuming it can be). Innumerable practical problems are in fact LPs, and as such there are highly optimized algorithms and programs available for solving them.

**Exercise 5** (Job Assignment Problem)**.** *There are $I$ people available to work $J$ jobs. The value of person $i$ working 1 day at job $j$ is $a_{ij}$, for $i = 1, \ldots, I$, $j = 1, \ldots, J$. Each job is completed after a total of 1 day's worth of time has been done on it, though partial completion still has value (e.g. person $i$ working half a day on job $j$ is worth $0.5a_{ij}$). The problem is to find an optimal assignment of jobs for each person for one day.*

**Possible Solution** An assignment $x$ is a choice of numbers $x_{ij}$ where $x_{ij}$ is the portion of person $i$'s time spent on job $j$. There are three constraints that $x$ must satisfy. First, no person $i$ can work more than 1 day's worth of time per day:

$$\sum_{j=1}^{J} x_{ij} \leq 1 \qquad \text{for } i = 1, \ldots, I \tag{8}$$

Second, no job $j$ can be worked past completion:

$$\sum_{i=1}^{I} x_{ij} \leq 1 \qquad \text{for } j = 1, \ldots, J \tag{9}$$

Third, we require positivity. i.e. $x_{ij} \geq 0$ for all $i, j$. This means nothing can be worked a negative ammount. The maximization function we are seeking is

$$\sum_{i=1,j=1}^{I,J} a_{ij} x_{ij} = A \bullet x \tag{10}$$

The entire LP can be succinctly expressed as

$$\begin{cases} \max & A \bullet x & \\ \text{s.t.} & \sum_{j=1}^{J} x_{ij} \leq 1 & i = 1, \ldots, I \\ & \sum_{i=1}^{I} x_{ij} \leq 1 & j = 1, \ldots, J \\ & x_{ij} \geq 0 & i = 1, \ldots, I; j = 1, \ldots, J \end{cases} \tag{11}$$

**That's all! Good luck with the final exam! We hope you've enjoyed CS38!**