Algorithms

or: the Unofficial Guide to the Georgia Institute of Technology's **CS6515**: *Graduate Algorithms*

George Kudrayvtsev
george.k@gatech.edu

Last Updated: March 25, 2020

> The only way to get through this course is by solving an uncountably-infinite number of practice problems while fueled by copious amounts of caffeine ☕.

WHY do we need to study algorithms, and why these specifically? The most important lesson that should come out of this course (that is unfortunately only mentioned in Chapter 8 of *Algorithms* and the 4th lecture of *Graduate Algorithms*) is that many problems can be reduced to an algorithm taught here; they are considered the **fundamental algorithms in computer science**, and if you know them inside-and-out, you can "transform" novel problems into a problem that can be solved by one of these algorithms.

For example, a complicated problem like "can you make change for v using some limited number of coins" can be *reduced* to a Knapsack problem, a quest to "find the length of the longest palindromic subsequence in a string" can be *reduced* to the finding the Longest Common Subsequence, and the arbitrage problem of finding a way to trade currencies on international exchanges to make a profit can be *reduced* to finding negative weight cycles via Shortest Paths.

Keep this in mind as you work through exercises.

List of Algorithms

PART I

Notes

BEFORE we begin to dive into all things algorithmic, some things about formatting are worth noting.

An item that is highlighted **like this** is a "term" that is cross-referenced wherever it's used. Cross-references to these vocabulary words are subtly highlighted like this and link back to their first defined usage; most mentions are available in the Index.

Linky I also sometimes include margin notes like the one here (which just links back here) that reference content sources so you can easily explore the concepts further.

Contents

Dynamic Programming

The **dynamic programming** (commonly abbreviated as DP to make undergraduates giggle during lecture) problem-solving technique is a powerful approach to creating efficient algorithms in scenarios that have a lot of repetitive data. The key to leveraging dynamic programming involves approaching problems with a particular mindset (paraphrased from *Algorithms*, pp. 158):

> *From the original problem, identify a collection of subproblems which share two key properties: (a) the subproblems have a distinct ordering in how they should be performed, and (b) subproblems should be related such that solving "earlier" or "smaller" subproblems gives the solution to a larger one.*

Keep this mindset in mind as we go through some examples.

1.1 Fibbing Our Way Along...

A classic toy example that we'll start with to demonstrate the power of dynamic programming is a series of increasingly-efficient algorithms to compute the **Fibonacci sequence**:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

In general, $F_n = F_{n-1} + F_{n-2}$, with the exceptional base-case that $F_n = n$ for $n \leq 1$. The simplest, most naïve algorithm (see algorithm 1.1) for calculating the n^{th} Fibonacci number just recurses on each F_m as needed.

Notice that each branch of the recursion tree operates independently despite them doing almost identical work: we know that to calculate F_{n-1} we need F_{n-2}, but we are also calculating F_{n-2} separately for its own sake... That's a lot of extra work that increases *exponentially* with n!

Wouldn't it be better if we kept track of the Fibonacci numbers that we generated as we went along? Enter Fib2(n), which no longer recurses down from F_n but instead

Algorithm 1.1: Fib1 (n), a naïve, recursive Fibonacci algorithm.

Input: An integer $n \geq 0$.
Result: F_n

```
if n ≤ 1 then
    return n
end
return Fib1 (n − 1) + Fib1 (n − 2)
```

builds up to it, saving intermediate Fibonnaci numbers in an array:

Algorithm 1.2: Fib2 (n), an improved, iterative Fibonacci algorithm.

Input: An integer $n \geq 0$.
Result: F_n

```
if n ≤ 1 then
    return n
end
F := {0, 1}
foreach i ∈ [2, n] do
    F[i] = F[i − 1] + F[i − 2]
end
return F[n]
```

The essence of dynamic programming lies in identifying the potential to implement two main principles:

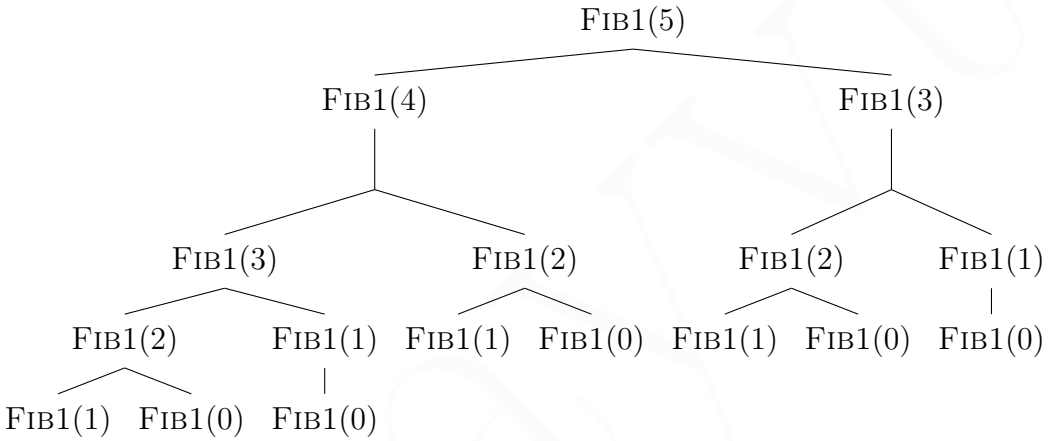
- *avoid repetitive work* and instead store it after computing it once. Identifying the overlapping subproblems (like the fact that F_{n-1} and F_{n-2} share large amounts of work) is a key part in developing a dynamic programming approach. This means you should not shy away from high memory usage when implementing a dynamic programming algorithm—the speed savings from caching repeated intermediate results outweigh the "cost" of memory.
- *avoid recursion* and instead use an iterative approach. This point is actually **not** universal when it comes to dynamic programming and pertains specifically to our course. We could have likewise made algorithm 1.2 pass around an array parameter to a recursive version; this would be an example of a **memoization** technique.

Memoization and recursion often go hand-in-hand when it comes to dynamic programming solutions, but this class shies away from them. Some of the walk-throughs may not, though, since it's (in my opinion) an arbitrary restriction that may make problems harder than they need to be.

1.1.1 Recursive Relations

Let's look at algorithm 1.1 through a different lens and actually try to map out the recursion tree as it develops.

Suppose we want to calculate F_5... our algorithm would then try to calculate F_4 and F_3 separately, which will try to calculate F_3 and F_2, and so on...



Notice the absurd duplication of work that we avoided with Fib2()... Is there a way we can represent the amount of work done when calling Fib2(n) in a compact way?

Suppose $T(n)$ represents the running time of Fib1(n). Then, our running time is similar to the algorithm itself. Since the base cases run in constant time and each recursive case takes $T(n-1)$ and $T(n-2)$, respectively, we have:

$$T(n) \leq \mathcal{O}(1) + T(n-1) + T(n-2)$$

So $T(n) \geq F_n$; that is, our *running time* takes at least as much time as the Fibonacci number itself! So F_{50} will take 12,586,269,025 steps (that's 12.5 *billion*) to calculate with our naïve formula...

The Golden Ratio

Interestingly enough, the Fibonacci numbers grow exponentially in φ, the **golden ratio**, which is an interesting mathematical phenomenon that occurs often in nature.

The golden ratio is an irrational number:

$$\varphi = \frac{1+\sqrt{5}}{2} \approx 1.6180\ldots$$

and the Fibonacci numbers increase exponentially by approximately $\frac{\varphi^n}{\sqrt{5}}$.

1.2 Longest Increasing Subsequence

A common, more-practical example to demonstrate the power of dynamic programming is finding the longest increasing subsequence in a series.

A **series** is just a list of numbers:

$$5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9, 3$$

A **subsequence** is a set of numbers from a series that is still in order (but is not necessarily consecutive):

$$4, 1, 4, 9$$

Thus, what we're looking for in a longest-increasing-subsequence (or LIS) algorithm is the longest set of numbers that are in order relative to the original series that increases from smallest to largest. For our example, that would be the subsequence:

$$-3, 1, 4, 5, 8, 9$$

1.2.1 Finding the Length

To start off a little simpler, we'll just be trying to find the length. Let's start by doing the first thing necessary for all dynamic programming problems: *identify shared work.*

Suppose our sequence was one element shorter:

$$5, 7, 4, -3, 9, 1, 10, 4, 5, 8, 9$$

Intuitively, wouldn't everything we need to do stay almost the same? The only thing that should matter is checking whether or not the new digit 3 can affect our longest sequence in some way. And wouldn't 3 only come into play for subsequences that are currently smaller than 3?

That's a good insight: at each step, we need to compare the new digit to the largest element of all previous subsequences. And since we don't need the subsequence itself, we only need to keep track of its length and its maximum value. Note that we track

all previous subsequences, because the "best" one at a given point in time will not necessarily be the best one at *every* point in time as the series grows.

What is a potential LIS' maximum value? Well, it's the increasing subsequence that ends in that value! So given a list of numbers: $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$, we want a subproblem $L(i)$ that is the longest increasing subsequence of $x_{1..i}$ that, critically, includes x_i itself:

$$L(i) = 1 + \max_{1 \leq j < i} \{L(j) \mid x_j < x_i\}$$

Algorithm 1.3: LIS1 ($\mathcal{S}$), an approach to finding the longest increasing subsequence in a series using dynamic programming.

Input: $\mathcal{S} = \{x_1, \ldots, x_n\}$, a series of numbers.
Result: L, the length of the LIS of $\mathcal{S}$.

```
/* Initialize the LIS length to a baseline.  */
```
$\mathcal{L} := 0$

foreach $x_i \in \mathcal{S}$ **do**
  $m := 1$
  foreach $j \in [1, i)$ **do** `/* find the best LIS to append to */`
    if $x_i > x_j \wedge \mathcal{L}[j] \geq m$ **then**
      $m = \mathcal{L}[j]$
    end
  end
  $\mathcal{L}[i] = m + 1$
end
return $\max(\mathcal{L})$

The running time of the algorithm is $\mathcal{O}(n^2)$ due to the double `for`-loop over (up to) n elements each.

1.3 Longest Common Subsequence

Let's move on to another dynamic programming example. Given two equal-length strings,

$$X = \{x_1, x_2, \ldots, x_n\}$$
$$Y = \{y_1, y_2, \ldots, y_n\}$$

we want to find the length ℓ of their longest common subsequence (commonly abbreviated LCS). The list of characters that appear in the same relative order (possibly with gaps) is a common subsequence. For example, given:

$$X = BCDBCDA$$
$$Y = ABECBAB$$

the LCS is $BCBA$. How can we find this algorithmically?

1.3.1 Step 1: Identify Subproblems

Dynamic programming solutions are supposed to build upon themselves. Thus, we should naturally expect our subproblems to just be increasingly-longer prefixes of the input strings. For example, suppose we're three characters in and are analyzing the 4^{th} character:

$$X = BCD \leftarrow \boldsymbol{B} = x_{i=4}$$
$$Y = ABE \leftarrow \boldsymbol{C} = y_{i=4}$$

Notice that $\ell = 1$ before and $\ell = 2$ after, since we go from B to BC as our LCS. In general, though, there are two cases to consider, right? Either $x_i = y_i$, in which case we know *for sure* that ℓ increases since we can just append both characters to the LCS, or $x_i \neq y_i$, in which case we need to think a little more.

It's possible that either the new x_i or the new y_i makes a new LCS viable. We can't integrate them into the LCS at the same time, so let's suppose we only have a new y_i:

$$X = BCD$$
$$Y = ABE \leftarrow \boldsymbol{C}$$

Wait, what if we built up X character-by-character to identify where C fits into LCSs? That is, we first compare C to $X = B$, then to $X = BC$, then to $X = BCD$, tracking the length at each point in time? We'd need a list to track the best ℓ:

	B	C	D
C	0	1	1

What about if there was already a C? Imagine we instead had $X' = BCC$. By our logic, that'd result in the length table

	B	C	C
C	0	1	2

In proper dynamic programming fashion, though, we should assume that our previous subproblems gave us accurate results. In other words, we'd know that $\ell_1 = 1$ because of the earlier $y_2 = B$, so we can assume that our table will automatically build up:

	B	C	D
C	1	2	2

How would we know that? One idea would be to compare the new character y_i to all of the previous characters in X. Then, if $x_j = y_i$, we know that the LCS will increase, so we'd increment ℓ_i:

$$\ell_i = 1 + \max_{1 \le j < i} \{\ell_j\}$$

For our example, when we're given $y_4 = C$, we see that $x_2 = C, \ell_2 = 1$, so we set $\ell_4 = 2$.

1.3.2 Step 2: Find the Recurrence

Under this mindset, what's our recurrence? We need to track ℓs for every character in our string, and each new character might increment any of the $j < i$ subsequences before it. One might envision a matrix relating the characters of one string to the other, filled out row-by-row with the LCS length at each index:

	B	C	D	B
A	0	0	0	0
B	1	1	1	1
E	1	1	1	1
C	1	2	2	2

With the introduction of each i^{th} character, we need to compare to the previous characters in the other string. We can't evaluate both x_i and y_i simultaneously (unless they're equal), and so we say $L(i, j)$ contains the LCS length for an i-length X and a j-length Y.

Then, the latest LCS length is based on whether or not the last characters match:

$$L(i,j) = \begin{cases} 1 + L(i-1, j-1) & \text{if } x_i = y_j \\ \max\left(L(i-1,j), L(i,j-1)\right) & \text{otherwise} \end{cases}$$

In other words, if the last characters match, we increment the LCS that would've resulted *without* those letters. If they don't, we just consider the best LCS if we hadn't used x_i xor y_j.

Our base case is trivially: $L(0, \cdot) = L(\cdot, 0) = 0$. That is, when finding the LCS between strings where one of them is the empty string, the length is obviously zero.

1.4 Knapsack

A popular class of problems that can be tackled with dynamic programming are known as **knapsack** problems. In general, they follow a formulation in which you

must select the optimal objects from a list that "fit" under some criteria and also maximize a value.

More formally, a knapsack problem is structured as follows:

Knapsack:

Input: A set of n objects with values and weights:
$$V = \{v_1, v_2, \ldots, v_n\}$$
$$W = \{w_1, w_2, \ldots, w_n\}$$
A total capacity B.

Output: A subset S that both:
(a) maximizes the total value: $\max_{S \in B} \sum_{i \in S} v_i$
(b) while fitting in the knapsack: $\sum_{i \in S} w_i \leq B$

There are two primary variants of the knapsack problem: in the first, there is only one copy of each object, whereas in the other case objects can be repeatedly added to the knapsack without limit.

1.4.1 Greedy Algorithm

A natural approach to solving a problem like this might be to greedily grab the highest-valued object every time. Unfortunately, this does not always maximize the total value, and even simple examples can demonstrate this.

Suppose we're given the following values:

$$V = \{v_1 = 15, v_2 = 10, v_3 = 8, v_4 = 1\}$$
$$W = \{w_1 = 15, w_2 = 12, w_3 = 10, w_4 = 5\}$$
$$B = 22$$

A greedy algorithm would sort the objects by their value-per-weight: $r_i \frac{v_i}{w_i}$. In this case (the objects are already ordered this way), it would take v_1, then v_4 (since after taking on w_1, w_4 is the only one that can still fit). However, that only adds up to 16, whereas the optimal solution picks up v_2 and v_3, adding up to 18 and coming in exactly at the weight limit.

1.4.2 Optimal Algorithm

As before, we first define a subproblem in words, then try to express it as a recurrence relation.

Attempt #1

The easiest baseline to start with is by defining the subproblem as operating on a smaller prefix of the input. Let's define $K(i)$ as being the maximum achievable using a subset of objects up to i. Now, let's try to express this as a recurrence.

What does our 1D table look like when working through the example above? Given

values:	15	10	8	1
weights:	15	12	10	5

We can work through and get $K:\ 15\ \ 15\ \ 15\ \ 15$. On the first step, obviously 15 is better than nothing. On the second step, we can't take both, so keeping 15 is preferred. On the third step, we can grab both v_3 and v_2, but notice that this requires bigger knowledge than what was available in $K(1)$ or $K(2)$. We needed to take a *sub*optimal solution in $K(2)$ that gives us enough space to properly consider object 3...

This tells us our subproblem definition was insufficient, and that we need to consider suboptimal solutions as we build up.

Solution

Let's try a subproblem that keeps the $i-1^{\text{th}}$ problem with a smaller capacity $b \leq B - w_i$. Now our subproblem tracks a 2D table $K(i, b)$ that represents the maximum value achievable using a subset of objects (up to i, as before), AND keeps their total weight $\leq b$. The solution is then at $K(n, B)$.

Our recurrence relation then faces two scenarios: either we include w_i in our knapsack if there's space for it AND it's better than excluding it, or we don't. Thus,

$$K(i,b) = \begin{cases} \max \begin{cases} v_i + K(i-1, b-w_i) \\ K(i-1,b) \end{cases} & \text{if } w_i \leq b \\ K(i-1,b) & \text{otherwise} \end{cases}$$

with the trivial base cases $K(0,b) = K(i,0) = 0$, which is when we have no objects and no knapsack, respectively. This algorithm is formalized in algorithm 1.4; its running time is very transparently $\mathcal{O}(nB)$.

1.5 Knapsack With Repetition

In this variant, we have an unlimited supply of objects: we can use an object as many times as we'd like.

Algorithm 1.4: Knapsack$(\cdot)$, the standard knapsack algorithm with no object repetition allowed, finding an optimal subset of values to fit in a capacity-limited container.

Input: List of object weights: $W = \{w_1, \ldots, w_n\}$
Input: List of object values: $V = \{v_1, \ldots, v_n\}$
Input: A capacity, B.
Result: The collection of objects resulting in the maximum total value without exceeding the knapsack capacity.

```
// For the base cases, only the first row and column are required to
   be zero, but we zero-init the whole thing for brevity.
```
$\mathbf{K}_{n+1\times B+1} \leftarrow \mathbf{0}$
foreach $i \in [1..n]$ **do**
  foreach $b \in [1..B]$ **do**
    $x = \mathbf{K}[i-1, b]$
    if $w_i \leq b$ **then**
      $\mathbf{K}[i, b] = \max[x, v_i + \mathbf{K}[i-1, b-w_i]]$
    else
      $\mathbf{K}[i, b] = x$
    end
  end
end
return $K(n, B)$

1.5.1 Simple Extension

Let's try to solve this with the same formulation as before: let's define $K(i, b)$ as being the maximum value achievable using a **multiset** (that is, a set where duplicates are allowed) of objects $\{1, \ldots, i\}$ with the weight $\leq b$. Our resulting recurrence relation is almost identical, but we don't remove the object from our pool of viabilities when considering smaller bags:

$$K(i, b) = \max \begin{cases} K(i-1, b) \\ v_i + K(i, b-w_i) \end{cases}$$

1.5.2 Optimal Solution

Do we even need i—the variable we use to track the objects available to us—anymore? Intuition says "no," and we might actually be able to formulate an even better solution because of it.

Since we're never considering the object set, we only really need to define our subproblem as an adjustment of the available capacity. Let's define $K(b)$ as the maximum value achievable within the total weight $\leq b$.

Now we need to consider all the *possibilities* of a last object, rather than restrict ourselves to the i^{th} one, and take the best one. The recurrence can then be expressed as:

$$K(b) = \max_i \{v_i + K(b - w_i) \mid 1 \leq i \leq n, w_i \leq b\}$$

This is formalized in algorithm 1.5.

Algorithm 1.5: KnapsackRepeating$(\cdot)$, the generalized knapsack algorithm in which unlimited object repetition is allowed, finding an optimal multiset of values to fit in a capacity-limited container.

Input: List of object weights: $W = \{w_1, \ldots, w_n\}$
Input: List of object values: $V = \{v_1, \ldots, v_n\}$
Input: A capacity, B.
Result: The collection of objects (possibly with duplicates) resulting in the maximum total value without exceeding the knapsack capacity.

```
K ← 0                    // defines our 1D, B-element array, K.
foreach b ∈ [1..B] do
    K(b) = 0
    foreach w_i ∈ W | w_i ≤ b do
        v ← v_i + K(b − w_i)
        if v > K(b) then
            K(b) = v
        end
    end
end
return K(B)
```

1.6 Matrix Multiplication

Multiplying any two matrices, $\mathbf{A}_{n \times m}$ and $\mathbf{B}_{m \times k}$, takes $\mathcal{O}(nmk)$ time. Anyone who has worked with matrices knows that the order in which you multiply a chain of matrices can heavily impact performance. For example, given a 2×5 matrix $\mathbf{A}$, a

5×10 matrix $\mathbf{B}$, and a 10×2 matrix $\mathbf{C}$ as such:

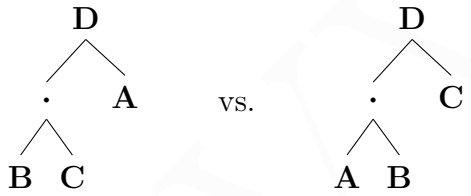
$$\mathbf{D}_{2\times 2} = \underbrace{(\mathbf{A}_{2\times 5} \cdot \mathbf{B}_{5\times 10})}_{\text{2×10 result, 100 ops}} \cdot \mathbf{C}_{10\times 2}$$

takes 140 total ops, a bit less than the alternative order which takes 150 ops:

$$\mathbf{D}_{2\times 2} = \mathbf{A}_{2\times 5} \cdot \underbrace{(\mathbf{B}_{5\times 10} \cdot \mathbf{C}_{10\times 2})}_{\text{5×2 result, 100 ops}}$$

The set of ordering possibilities grows exponentially as the number of matrices increases. Can we compute the best order efficiently?

To start off, envision a particular parenthesization as a binary tree, where the root is the final product and the branches are particular matrix pairings:



Then, if we associate a cost with each node (the product of the matrix dimensions), a cost for each possible tree bubbles up. Naturally, we want the tree with the lowest cost.[1] The dynamic programming solution comes from a key insight: **for a root node to be optimal, all of its subtrees must be optimal**.

1.6.1 Subproblem Formulation

Let's formally define our problem. If we have n matrices, $\mathbf{A}_1, \mathbf{A}_2, \ldots, \mathbf{A}_n$, we can define their sizes as being $m_{i-1} \times m_i$, since the inner dimensions of a matrix product must match. That is, the size of matrix $\mathbf{A}_2$ would be $m_1 \times m_2$. The cost of a particular product, $\mathbf{A}_i\mathbf{A}_j$, is then $c_{ij} = m_{i-1}m_im_j$.

Now we can generalize the above binary tree's structure: the subtree of the product on the left will be $\mathbf{A}_1\mathbf{A}_2 \ldots \mathbf{A}_i$ while the subtree on the right will be $\mathbf{A}_{i+1}\mathbf{A}_{i+2} \ldots \mathbf{A}_n$. Thus, an arbitrary subtree is a product defined by some range $[j, k]$: $\mathbf{A}_j\mathbf{A}_{j+1} \ldots \mathbf{A}_k$ which can be thought of as a "substring" of $\{1, 2, \ldots, n\}$.

$$\mathbf{A}_j\mathbf{A}_{j+1} \ldots \mathbf{A}_k$$
$$\mathbf{A}_j\mathbf{A}_{j+1} \ldots \mathbf{A}_m \quad \mathbf{A}_{m+1}\mathbf{A}_{m+2} \ldots \mathbf{A}_k$$

[1] It should be fairly obvious how we compute cost. Multiplying two matrices, $\mathbf{W}_{a\times b}$ and $\mathbf{Y}_{b\times c}$, takes $\mathcal{O}(abc)$ time. A single entry z_{ij} in the result is the dot product between the i^{th} row of $\mathbf{W}$ and the j^{th} column of $\mathbf{Y}$: $z_{ij} = \sum_{k=0}^{b} (w_{ik} \cdot y_{kj})$. This takes b operations and occurs ac times.

This indicates a subproblem with two indices: let $M(i, j)$ be the minimum cost for computing $\mathbf{A}_i \dots \mathbf{A}_j$. Then our solution will be at $M(1, n)$. Obviously it's a little strange if $j < i$, so we'll force $j \geq i$ and focus on building up the "upper triangle" of the 2D array.

1.6.2 Recurrence Relation

Let's formulate the base case for our $M(i, j)$. Unlike with the subproblems we've seen before, $M(1, j)$ doesn't mean anything special. However, $M(i, i)$ does: there are no ops required to find $\mathbf{A}_i$, so $M(i, i) = 0$.

To compute $M(i, j)$, we need to consider the best point at which $\mathbf{A}_i \dots \mathbf{A}_j$ can be split, so choose the k that results in the best split:

$$M(i,j) = \min_{i \leq k < j} [\underbrace{M(i,k)}_{\substack{\text{left} \\ \text{subtree}}} + \underbrace{M(k+1,j)}_{\substack{\text{right} \\ \text{subtree}}} + \underbrace{m_{i-1} m_k m_j}_{\substack{\text{cost of new} \\ \text{product}}}]$$

Notice that each new cell depends on cells both below it and to the left of it, so the array must be built from the diagonal outward for each off-diagonal (hence the weird indexing in algorithm 1.6).

Computing the best value for a single cell takes $\mathcal{O}(n)$ time in the worst case and the table has $\frac{n^2}{2}$ cells to fill, so the overall complexity is $\mathcal{O}(n^3)$ which, though large, is far better than exponential time.

Algorithm 1.6: An $\mathcal{O}(n^3)$ time algorithm for computing the cost of the best chain matrix multiplication order for a set of matrices.

Input: A set of sizes corresponding to matrix dimensions, $[m_1, m_2, \dots, m_n]$.
Result: The lowest-cost matrix multiplication possible.

$\mathbf{M}_{n \times n} \leftarrow \emptyset$
$\mathbf{M}[i, i] = 0 \qquad \forall i \in [1, n]$
foreach $d \in [1, n-1]$ **do**
  foreach $i \in [1, n-d]$ **do**
    $j \leftarrow i + d$
    $\mathbf{M}[i, j] = \min_{i \leq k < j} [\mathbf{M}[i, k] + \mathbf{M}[k+1, j] + m_{i-1} m_k m_j]$
  end
end
return M

Divide & Conquer

> *It is the rule in war, if ten times the enemy's strength, surround them; if five times, attack them; if double, be able to divide them; if equal, engage them; if fewer, defend against them; if weaker, be able to avoid them.*
>
> — Sun Tzu, *The Art of War*

Dividing a problem into 2 or more parts and solving them individually to conquer a problem is an extremely common and effective approach. If you were tasked with finding a name in a thick address book, what would you do? You'd probably divide it in half, check to see if the letter you're looking for comes earlier or later, then check midway through that half, disregarding the other half for the remainder of your search. This is exactly the motivation behind **binary search**: given a sorted list of numbers, you can halve your search space every time you compare to a value, taking $\mathcal{O}(\log n)$ time rather than the naïve $\mathcal{O}(n)$ approach of checking every number.

2.1 An Exercise in D&C: Multiplication

Consider the traditional approach to multiplication with decimal numbers that we learned in grade school: for every digit, we multiply it with every other digit.

$$
\begin{array}{cccccccc}
 & & & 1 & 2 & 3 & 4 \\
\times & & & & 1 & 2 & 3 \\
\hline
 & & & 3 & 7 & 0 & 2 \\
+ & & 2 & 4 & 6 & 8 & \\
+ & 1 & 2 & 3 & 4 & & \\
\hline
 & 1 & 5 & 1 & 7 & 8 & 2
\end{array}
$$

The process overall in $\mathcal{O}(n^2)$. Can we do better? Let's think about a simpler divide-

and-conquer approach that leverages the fact that we can avoid certain multiplications.

Quick Maffs: **Off-By-One**

This minor digression gives us insight on how we can do better than expected when multiplying two complex numbers together. The underlying assumption is that addition is cheaper than multiplication.[a]

We're given two complex numbers: $a+bi$ and $c+di$. Their product, via the FOIL approach we all learned in middle school, can be found as follows:

$$\begin{aligned} &= (a+bi)(c+di) \\ &= ac + adi + bci + bdi^2 \\ &= (ac - bd) + (ad + bc)i \end{aligned}$$

There are four independent multiplications here. The clever trick realized by Gauss is that the latter sum can be computed without computing the individual terms. The insight comes from the fact that $ad + bc$ looks like it comes from a FOIL just like the one we did: a, b on one side, d, c on the other:

$$(a+b)(d+c) = \underbrace{ad+bc}_{\text{what we want}} + \underbrace{ac+bd}_{\text{what we have}}$$

Notice that these "extra" terms are already things we've computed! Thus,

$$ad + bc = (a+b)(d+c) - ac - bd$$

Thus, we only need three multiplications: ac, bd, and $(a+b)(d+c)$. We do need more additions, but we entered this derivation operating under the assumption that this is okay. A similar insight will give us the ability to multiply n-bit integers faster, as well.

[a] This is undeniably true, even on modern processors. On an Intel processor, 32-bit multiplication takes xxx CPU cycles while adding only takes xxx.

We're working with the n-bit integers x and y, and our goal is to compute $z = xy$.[1] Divide-and-conquer often involves splitting the problem into two halves; can we do that here? We can't split xy, but we can split x and y themselves into two $n/2$-bit halves:

$$x = \boxed{x_l}\boxed{x_r}$$

[1] We're operating under the assumption that we can't perform these multiplications on a computer using the hardware operators alone. Let's suppose we're working with massive n-bit integers, where n is something like 1024 or more (like the ones common in cryptography).

$$y = \boxed{y_l \mid y_r}$$

The original x is a composition of its parts: $x = 2^{n/2}x_l + x_r$, and likewise for y. Thus,

$$\begin{aligned} xy &= \left(2^{n/2}x_l + x_r\right)\left(2^{n/2}y_l + y_r\right) \\ &= 2^n x_l y_l + 2^{n/2}(x_l y_r + x_r y_l) + x_r y_r \end{aligned}$$

This demonstrates a recursive way to calculate xy: it's composed of 4 $n/2$-bit multiplications. Of course, we're still doing this in $\mathcal{O}(n^2)$ time;[2] there's no improvement yet. Can we now apply a similar trick as the one described in the earlier aside, though, and instead solve *three* subproblems?

Beautifully, notice that $x_l y_r + x_r y_l$ can be composed from the other multplications:

$$(x_l + x_r)(y_l + y_r) = \underbrace{x_l y_r + x_r y_l}_{\text{what we want}} + \underbrace{x_l y_l + x_r y_r}_{\text{what we have}}$$

If we let $A = x_l y_l$, $B = x_r y_r$, and $C = (x_l + x_r)(y_l + y_r)$, then we only need to combine three subproblems:

$$xy = 2^n A + 2^{n/2}(C - A - B) + B$$

which is $\mathcal{O}\left(n^{\sqrt{3}\approx 1.59}\right)$, better than before! Turns out, there is an *even better* D&C approach that'll let us find the solution in $\mathcal{O}(n \log n)$ time, but this will require a long foray into polynomials, complex numbers, and roots of unity that we won't get into right now.

2.2 Another Exercise in D&C: Median-Finding

Suppose you have an unsorted list of numbers, $A = [a_1, a_2, \ldots, a_n]$. Can you find its median value without sorting it? Turns out, you can. Let's concretely define the median value as being the $\left\lceil \frac{n}{2} \right\rceil^{\text{th}}$-smallest number. It'll actually be easier to solve a "harder" problem: finding the k^{th} smallest value in A.[3]

Naïvely, we could sort A then return the k^{th} element; this would take $\mathcal{O}(n \log n)$ time, but let's do better. Let's choose a **pivot**, p, and divide A into three sub-arrays: those less than p, those greater than p, and those equal to p:

$$A_{<p}, A_{=p}, A_{>p}$$

[2] You can refer to Solving Recurrence Relations or apply the Master theorem directly: $T(n) = 4T\left(\frac{2}{n}\right) + \mathcal{O}(n^2)$.

[3] Critically, it's not the k^{th}-smallest *unique* element, so ties are treated independently. That is, the 2^{nd} smallest element of $[1, 2, 1]$ would be 1.

Now we know which sublist the element we're looking for is in: if $k < |A_{<p}|$, we know it's in there. Similarly, if $|A_{<p}| < k < |A_{<p}| + |A_{=p}|$, we know it IS p. Finally, if $k > |A_{<p}| + |A_{=p}|$, then it's in $A_{>p}$.

Let's look at a concrete example, we (randomly) choose $p = 11$ for:

$$A = [5, 2, 20, 17, 11, 13, 8, 9, 11]$$

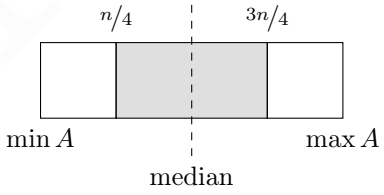
Then,

$$A_{<p} = [5, 2, 8, 9]$$
$$A_{=p} = [11, 11]$$
$$A_{>p} = [20, 17, 13]$$

If we're looking for the $(k \leq 4)^{\text{th}}$-smallest element, we know it's in the first sublist and can recurse on that. If we're looking for the $(k > 6)^{\text{th}}$ smallest element, then we want the $(k-6)^{\text{th}}$-smallest element in the last sublist (to offset the 6 elements in the two other lists we're now ignoring).

Now the question is reduced to finding a good pivot point, p. If we always chose the median, this would divide the array into two equal subarrays and an array of the median values, so $|A_{<p}| \approx |A_{>p}|$. That means we'd always *at least* halve our search space every time, but we also need to do $\mathcal{O}(n)$ work to do the array-splitting; this gives us an $\mathcal{O}(n)$ total running time, exactly what we're looking for. However, this is a bit of a chicken-and-egg problem: our original problem is searching for the median, and now we're saying if we knew the median we could efficiently find the median? Great.

What if, instead, we could find a pivot that is *close* to the median (say, within $\pm 25\%$ of it). Namely, if we look at a sorted version of A, that our pivot would lie within the shaded region:



In the worst case, then the size of our subproblem will be $3n/4$ of what it used to be, so: $T(n) = T\left(\frac{3n}{4}\right) + \mathcal{O}(n)$, which, magnificently, still solves to $\mathcal{O}(n)$. In fact, *any* valid subproblem that's smaller than the original will solve to $\mathcal{O}(n)$. We'll keep this notion of a *good* pivot as being within $1/4^{\text{th}}$ of the median, but remember that if we recurse on a subproblem that is at least one element smaller than the previous problem, we'll achieve linear time.

If we chose a pivot randomly, our probability of choosing a good pivot is 50%. The ordering within the list doesn't matter: exactly half of the elements make good pivots.

We can check whether or not a chosen pivot is "good" in $\mathcal{O}(n)$ time since that's what splitting into subarrays means, and if it's not a good p we can just choose another one. Like with flipping a coin, the **expectation** of p being good is 2 "flips"[4] (that is, it'll take two tries on average to find a good pivot). Thus, we can find a good pivot in $\mathcal{O}(n)$ time on average (two tries of n-time each), and thus we can find the n^{th}-smallest element in $\mathcal{O}(n)$ on average since we have a good pivot. Finding the median is then just specifying that we want to find the $\frac{n}{2}^{\text{th}}$-smallest element. ■

2.3 Solving Recurrence Relations

Recall the typical form of a recurrence relation for divide-and-conquer problems:

$$T(n) = 2T\left(\frac{n}{b}\right) + f(n)$$

Here, n is the size of the input problem, a is the number of subproblems in the recursion, b is the factor by which the subproblem size is reduced in each recursive call, and $f(n)$ is the complexity of any additional non-recursive processing.

Consider the (recursive) merge sort algorithm. In it, we split our n-length array into two halves and recurse into each, then perform a linear merge. Its recurrence relation is thus:

Lesson 10: DC3

$$T(n) = 2T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

It's possible to go from any recursive relationship to its big-O complexity with ease. For example, we can use the merge sort recurrence to derive the big-O of the algorithm: $\mathcal{O}(n \log n)$. Let's see how to do this in the general case.

2.3.1 Example 1: Integer Multiplication

The brute force integer multiplication algorithm's recurrence relation looks like this:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

We can similarly derive that its time complexity is $\mathcal{O}(n^2)$, but ***how?*** Let's work through it. To start off, let's drop the $\mathcal{O}(n)$ term. From the definition of big-O, we know there is some constant c that makes the following equivalent:

$$T(n) = 4T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

[4] The expectation can be defined and solved recursively. If we say flipping "heads" makes a good pivot, we obviously need at least one flip. Then, the chance of needing to flip again is 50%, and we'll again need to see the average flip-count to get heads, so $E_h = 1 + \frac{1}{2}E_h \implies E_h = 2$.

$$\leq 4T\left(\frac{n}{2}\right) + cn, \qquad \text{where } T(1) \leq c$$

Now let's substitute in our recurrence relation twice: once for $T(n/2)$, then again for the resulting $T(n/4)$.

$$\begin{aligned}
T(n) &\leq cn + 4T\left(\frac{n}{2}\right) \\
&\leq cn + 4\left[c\left(\frac{n}{2}\right) + 4T\left(\frac{n}{4}\right)\right] && \text{plug in } T\left(\tfrac{n}{2}\right) \\
&= cn\left(1 + \frac{4}{2}\right) + 4^2 T\left(\frac{n}{2^2}\right) && \text{rearrange and group} \\
&\leq cn\left(1 + \frac{4}{2}\right) + 4^2\left[4T\left(\frac{n}{2^3}\right) + c\left(\frac{n}{2^2}\right)\right] && \text{plug in } T\left(\tfrac{n}{2^2}\right) \\
&\leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2\right) + 4^3 T\left(\frac{n}{2^3}\right) && \text{rearrange and group, again}
\end{aligned}$$

Notice that we're starting to see a geometric series form in the cn term, and that its terms come from our original recurrence relation, $4T\left(\frac{n}{2}\right)$:

$$1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^n +$$

The pattern is clear after two substitions; the general form for i subtitutions is:

$$T(n) \leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \left(\frac{4}{2}\right)^3 + \ldots \left(\frac{4}{2}\right)^{i-1}\right) + 4^i T\left(\frac{n}{2^i}\right)$$

But when do we stop? Well our base case is formed at $T(1)$, so we stop when $\frac{n}{2^i} = 1$. Thus, we stop at $i = \log_2 n$, giving us this expansion at the final iteration:

$$T(n) \leq cn\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^{\log_2 n - 1}\right) + 4^{\log_2 n} \cdot T(1)$$

We can simplify this expression. From the beginning, we defined $c \geq T(1)$, so we can substitute that in accordingly and preserve the inequality:

$$T(n) \leq \underbrace{cn}_{\mathcal{O}(n)} \underbrace{\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^{\log n - 1}\right)}_{\text{increasing geometric series}} + \underbrace{4^{\log n} \cdot c}_{\mathcal{O}(n^2)}$$

Thankfully, we established in **??** that an increasing geometric series with ratio r and k steps has a complexity of $\Theta(r^k)$, meaning our series above has complexity:

$$\mathcal{O}\left(1 + \left(\frac{4}{2}\right) + \left(\frac{4}{2}\right)^2 + \ldots + \left(\frac{4}{2}\right)^{\log n - 1}\right) = \mathcal{O}\left(\left(\frac{4}{2}\right)^{\log n}\right)$$

$$
\begin{aligned}
&= \mathcal{O}\left(\frac{4^{\log n}}{2^{\log n}}\right) \\
&= \mathcal{O}\left(\frac{2^{\log_2 n} \cdot 2^{\log_2 n}}{n}\right) && \text{recall that } x^i \cdot y^i = (xy)^i \\
&= \mathcal{O}\left(\frac{n \cdot n}{n}\right) && \text{remember, } b^{\log_b n} = n \\
&= \mathcal{O}(n)
\end{aligned}
$$

Thus, we ultimately have quadratic complexity, as expected:

$$T(n) = \mathcal{O}(n) \cdot \mathcal{O}(n) + \mathcal{O}\left(n^2\right) = \boxed{\mathcal{O}\left(n^2\right)} \quad \blacksquare$$

Quick Maffs: Generalizing Complexity

One could argue that we got a little "lucky" earlier with how conveniently we could convert $4^{\log n} = n^2$, since 4 is a power of two. Can we do this generically with any base? How would we solve $3^{\log n}$, for example?

Well, we can turn the 3 into something base-2 compatible by the definition of a logarithm:

$$
\begin{aligned}
3^{\log n} &= \left(2^{\log 3}\right)^{\log n} \\
&= 2^{\log 3 \cdot \log n} && \text{power rule: } x^{a^b} = x^{ab} \\
&= 2^{\log\left(n^{\log 3}\right)} && \text{log exponents: } c \cdot \log n = \log(n^c) \\
&= n^{\log 3} && \text{d-d-d-drop the base!}
\end{aligned}
$$

This renders a generic formulation that can give us big-O complexity for any exponential:

$$b^{\log_a n} = n^{\log_a b} \tag{2.1}$$

2.3.2 Example 2: Better Integer Multiplication

Armed with our substitution and pattern-identification techniques as well as some mathematical trickery up our sleeve, solving this recurrence relation should be much faster:

$$T(n) = 3T\left(\frac{n}{2}\right) + \mathcal{O}(n)$$

As before, we substitute and generalize:

$$T(n) \leq cn + 3T\left(\frac{n}{2}\right)$$

$$\leq cn\left(1+\left(\frac{3}{2}\right)+\left(\frac{3}{2}\right)^2+\ldots+\left(\frac{3}{2}\right)^{i-1}\right)+3^iT\left(\frac{n}{2^i}\right)$$

Our last term is $i=\log_2 n$, as before, so we can again group each term by its complexity:

$$\begin{aligned}T(n) &\leq \underbrace{cn}_{\mathcal{O}(n)}\underbrace{\left(1+\left(\frac{3}{2}\right)+\left(\frac{3}{2}\right)^2+\ldots+\left(\frac{3}{2}\right)^{\log n-1}\right)}_{\text{again, increasing: } \mathcal{O}\left(\left(\frac{3}{2}\right)^{\log n}=\frac{3^{\log n}}{2^{\log n}}\approx n^{0.585}\right)}+\underbrace{3^{\log n}T(1)}_{\mathcal{O}\left(n^{\log 3\approx 1.585}\right)}\\ &\approx \mathcal{O}(n)\cdot\mathcal{O}\left(n^{0.585}\right)+\mathcal{O}\left(n^{1.585}\right)\\ &\approx \boxed{\mathcal{O}\left(n^{1.585}\right)}\end{aligned}$$

2.3.3 General Form

If we know certain things about the structure of the recurrence relation, we can actually arrive at the complexity by following a series of rules; no substitutions or derivations necessary!

Given the general form of a recurrence relation, where $a>0, b>1$:

$$T(n)=2T\left(\frac{n}{b}\right)+\mathcal{O}(n)$$

we have three general cases depending on the outcome of the geometric series after expansion:

$$T(n)=cn\underbrace{\left(1+\left(\frac{a}{b}\right)+\left(\frac{a}{b}\right)^2+\ldots+\left(\frac{a}{b}\right)^{\log_b n-1}\right)}+a^{\log_b n}\cdot T(1)$$

- If $a>b$, the series is dominated by the last term, so the overall complexity is $\mathcal{O}\left(n^{\log_b a}\right)$.
- If $a=b$, the series is just the sum of $\log_b n-1$ ones, which collapses to $\mathcal{O}(\log_b n)$, making the overall complexity $\mathcal{O}(n\log_b n)$.
- If $a<b$, the series collapses to a constant, so the overall complexity is simply $\mathcal{O}(n)$.

Of course, not every algorithm will have $\mathcal{O}(n)$ additional non-recursive work. The general form uses $\mathcal{O}\left(n^d\right)$; this is the **Master theorem** for recurrence relations:

Property 2.1. *For the general case of of the recurrence relation,*

$$T(n) = aT\left(\frac{n}{b}\right) + \mathcal{O}\left(n^d\right)$$

our case depends on the relationship between d *and* $\log_b a$*.*

$$T(n) = \begin{cases} \mathcal{O}\left(n^d\right) & \text{if } d > \log_b a \\ \mathcal{O}\left(n^d \log_2 n\right) & \text{if } d = \log_b a \\ \mathcal{O}\left(n^{\log_b a}\right) & \text{if } d < \log_b a \end{cases}$$

Memorize these rules into the deepest recesses of your mind (at least until May).

2.4 Fast Fourier Transform

I don't know enough about the **fast Fourier transform** to take good notes; this is just a quick-reference guide.

The **n^{th} roots of unity** are defined as:

$$\omega_n^0, \omega_n^1, \ldots, \omega_n^{n-1}$$

where

$$\omega_n = e^{\frac{2\pi i}{n}}$$

so, for example, the 4^{th} of the 8^{th} roots of unity is $\omega_8^4 = \left(e^{\frac{2\pi i}{8}}\right)^4 = e^{\pi i}$. Notice that $\omega_n^n = \omega_n^0 = 1$.

The FFT takes in a list of k constants and specifies the n^{th} roots of unity to use. It outputs a list of evaluations of the polynomial defined by the constants. Namely, if $\mathcal{K} = \{c_0, c_1, c_2, \ldots, c_{k-1}\}$ defines:

$$A(x) = c_0 + c_1 x + c_2 x^2 + \ldots + c_{k-1} x^{k-1}$$

Then the FFT of $\mathcal{K}$ with the n^{th} roots of unity will return n evaluations of $A(x)$:

$$(z_1, z_2, \ldots, z_n) = \text{FFT}(\mathcal{K}, \omega_n)$$

The **inverse fast Fourier transform** just uses a scaled version of the FFT with the inverse roots of unity:

$$\text{IFFT}(\mathcal{K}, \omega_n) = \frac{1}{2n}\text{FFT}(\mathcal{K}, \omega_n^{-1})$$

where ω_n^{-1} is the value that makes $\omega_n^{-1} \cdot \omega_n = 1$. This is defined as ω_n^{n-1}:

$$\omega_n^{-1} = \omega_n^{n-1} = e^{\frac{2\pi i(n-1)}{n}}$$

$$
\begin{aligned}
&= e^{\frac{2\pi i n}{n}} \cdot e^{\frac{-2\pi i}{n}} && \text{distribute and split} \\
&= e^{2\pi i} \cdot e^{-\frac{2\pi i}{n}} && \text{cancel } n\text{s} \\
&= e^{-\frac{2\pi i}{n}} && \text{Euler's identity: } e^{\pi i} = 1
\end{aligned}
$$

Since $\omega_n \cdot \omega_n^{n-1} = e^{\frac{2\pi i}{n}} \cdot e^{-\frac{2\pi i}{n}} = e^0 = 1$.

Graphs

I FUCKING love graphs.

Definitions

There are many terms when discussing graphs and they all have very specific meanings. This "glossary" serves as a convenient quick-reference.

The typical way a graph is defined is $G = (V, E)$. Sometimes, the syntax $\overrightarrow{G}$ is used to indicate a **directed graph**, where edges can only be traversed in a single direction.

- The syntax $(a, b) \in E$ indicates an edge from $a \to b$ within G.
- The syntax $w(a, b)$ indicates the **weight** (or *distance* or *length*) of the **edge** from $a \to b$.
- A **walk** through a graph can visit a vertex any number of times.
- A **path** through a graph can only visit a vertex once.
- A **cycle** is a path through a graph that starts and ends at the same node.
- A graph is **fully-connected** if each vertex is connected to every other vertex. In this case, there are $|E| = |V|^2$ edges.
- The **degree** of a vertex is the number of edges it has. For directed graphs, there is also $d_{\text{in}}(v)$ to indicate incoming edges (u, v) and $d_{\text{out}}(v)$ to indicate outgoing edges (v, u).

When discussing the big-O complexity of a graph algorithm, typically n refers to $|V|$, the number of vertices, and m refers to $|E|$, the number of edges. In a fully-connected graph, then, a $\mathcal{O}(nm)$ algorithm can actually be said to take $\mathcal{O}(n^3)$ time, for example.

3.1 Common Algorithms

The EXPLORE function described in algorithm 3.1 is important; it creates the building block for almost any traversal algorithm. It traverses the graph by digging as deep as possible of a path down a single vertex, then backtracking and traversing neighbors until all nodes are searched.

The optional PREVISIT and POSTVISIT functions described in algorithm 3.1 allow nodes to be processed as they're traversed; the latter of these will be used heavily soon.

ALGORITHM 3.1: EXPLORE(G, v), a function for visiting vertices in a graph.

Input: $G = (V, E)$, the graph itself.
Input: $v \in V$, a vertex from which to start exploring.
Input: Optionally, the functions PREVISIT(z) and POSTVISIT(z), which can do additional processing on the first and last time a vertex z is visited.

Result: The global *visited* array (of length $|V|$) is marked with visited[u] = *true* if the vertex u can be visited from v.
Result: The global *prev* array (also of length $|V|$) is marked with prev[z] = w, where w is the first vertex to explore z.

```
visited[v] = true
PREVISIT(v)                                          // optional
foreach edge (v, u) ∈ E do
    if ¬visited[u] then
        EXPLORE(G, u)
        prev[u] = v
    end
end
POSTVISIT(v)                                         // optional
```

3.1.1 Depth-First Search

Performing **depth-first search** on an undirected graph (described in algorithm 3.2) is just a simple way of leveraging EXPLORE() (see algorithm 3.1) on all nodes.

3.1.2 Breadth-First Search

While depth-first search recursively explores nodes until it reaches leaves, **breadth-first search** explores the graph layer by layer. Its output is different than that of

Algorithm 3.2: DFS(G), depth-first search labeling of connected components.

Input: $G = (V, E)$, an undirected graph to traverse.

```
cc := 0
foreach v ∈ V do
    visited[v] = false
    prev[v] = ∅
end
foreach v ∈ V do
    if ¬visited[v] then
        cc += 1
        EXPLORE(G, v)
    end
end
```

DFS, which tells us about the connectivity of a graph. Breadth-first search (starting from a vertex s) fills out a $dist[\cdot]$ array, which is the minimum number of edges from $s \to v$, for all $v \in V$ (or ∞ if there's no path). It gives us *shortest* paths. Its running time is likewise $\mathcal{O}(n + m)$.

3.2 Shortest Paths

First off, its worth reviewing **Dijkstra's algorithm** (see *Algorithms*, pp. 109–112) so that we can use it as a black box for building new graph algorithms. In summary, it performs BFS on graphs with **positively**-weighted edges, outputting the length of the shortest path from $s \rightsquigarrow v$ using a (binary) `min-heap` data structure. Its running time is $\mathcal{O}((n + m) \log n)$ with this construction.

3.2.1 From One Vertex: Bellman-Ford

In the subproblem definition, we condition on the number of edges used to build the shortest path. Define $D(i, z)$ as being the shortest path between a starting node s and a final node z using exactly i edges.

$$D(i, z) = \min_{y:(y,z)\in E} \{D(i-1, y) + w(y, z)\}$$

But what if we can get from $s \to z$ without needing exactly i edges? We'd need to find $\min_{j \le i} D(j, z)$, right? Instead, we can incorporate it into the recurrence directly:

$$D(i, z) = \min \begin{cases} D(i-1, z) \\ \min_{y:(y,z) \in E} \{D(i-1, y) + w(y, z)\} & \text{iterate over every edge that goes into } z \end{cases}$$

where we start with the base cases $D(0, s) = 0$, and $D(0, z) = \infty$ for all $z \neq s$, and the solutions for *all* of the vertices is stored in $D(n-1, \cdot)$.

This is the **Bellman-Ford algorithm**. Its complexity is $\mathcal{O}(nm)$, where n is the number of vertices and m is the number of edges (that is, $n = |V|, m = |E|$). Note that for a fully-connected graph, there are n^2 edges, so this would take $\mathcal{O}(n^3)$ time.

Negative Cycles If the graph has a negative weight cycle, then we will notice that the shortest path changes after $n-1$ iterations (when otherwise it wouldn't). So if we keep doing Bellman-Ford and the n^{th} iteration results in a different row than the $(n-1)^{\text{th}}$ iteration, there is a negative weight cycle. Namely, check if $\exists z \in V : D(n, z) < D(n-1, z)$; if there is, the cycle involves z and we can backtrack through D to identify the cycle.

3.2.2 From All Vertices: Floyd-Warshall

What if we don't just want the shortest path from $s \to z$ for all $z \in V$, but for all *pairs* of vertices (that is, from *any* start to any end). The naïve approach would be to compute Bellman-Ford for all vertices, taking $\mathcal{O}(mn^2)$ time. However, we can use the **Floyd-Warshall** algorithm to compute it in $\mathcal{O}(n^3)$ time, instead (better for dense graphs, since $m \le n^2$).

This should intuitively be possible, since the shortest path from $s \to z$ probably overlaps a lot with the shortest path from $s' \to z$ if s and s' are neighbors.

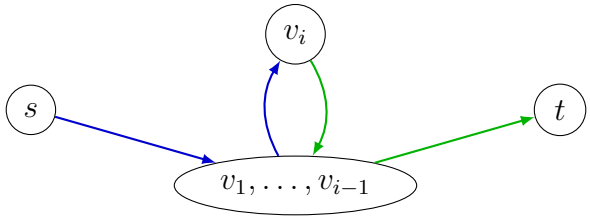
First, lets assign a number to each vertex, so $V = \{v_1, v_2, \ldots, v_n\}$. Now our subproblem is the "prefix" of the vertices, so we solve the all-pairs problem by only using some subset of vertices: $\{v_1, v_2, \ldots, v_i\}$ (and allowing $\emptyset$ as our base case). This is done for all possible start and end vertices, so $s, t \in \{v_1, v_2, \ldots, v_n\}$. Then,

Let $D(i, s, t)$ be the shortest path from $s \to t$ using a subset of the first i vertices. To start off, if we allow no intermediate vertices, only vertices that share an edge have a valid base path length; namely, the base cases are:

$$D(0, s, t) = \begin{cases} \infty & \text{if } \exists (s, t) \in E \\ w(s, t) & \text{otherwise} \end{cases}$$

Now lets look at the recurrence. There's a shortest path $\mathcal{P}$ from $s \to t$, right? Well, what if v_i is not on that path? Then we can exclude it, defering to the $(i-1)^{\text{th}}$ prefix.

And if it is? Then our path looks something like this:



(note that $\{v_1, \ldots, v_{i-1}\}$ might be empty, so in fact $s \to v_i \to t$, but that doesn't matter)

Then, our recurrence when v_i is on the path is just the shortest path to v_i and the shortest path *from* v_i! So, in total,

$$D(i, s, t) = \min \begin{cases} D(i-1, s, t) & \textit{(not on the path)} \\ D(i-1, s, i) + D(i-1, i, t) & \textit{(on the path)} \end{cases}$$

Notice that this implicitly determines whether or not $\{v_1, \ldots v_i\}$ is sufficient to form the path $s \to t$ because of the base case: we have ∞ wherever $s \nrightarrow t$, only building up the table for reachable places. Thus, $D(n, \cdot, \cdot)$ is a 2D matrix for all pairs of (start, end) points.

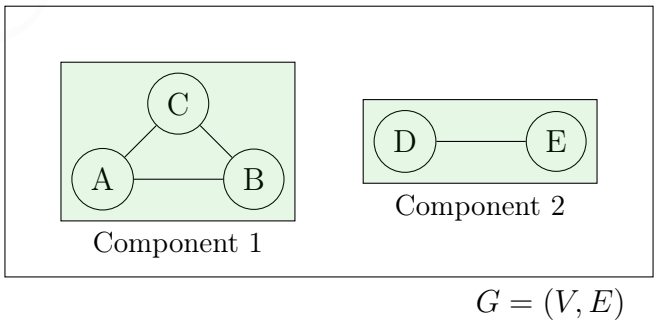
Negative Cycles If the graph has a negative weight cycle, a diagonal entry in the final matrix will be negative, so check if $\exists v \in V : D(n, v, v) < 0$.

3.3 Connected Components

Given a graph, there's a basic question we need to be able to answer:

What parts of the graph are reachable from a given vertex?

A vertex is **connected** to another if a path exists between them, and a **connected component** in a graph is a fancy way to refer to a set of vertices that are all reachable from each other.



DFS lets us find the connected components easily in $\mathcal{O}(n + m)$ time since whenever it encounters an unvisited node (post-exploration) it must be a separate component.

3.3.1 Undirected Graphs

How do we find a path between two vertices (s, e) using DFS on an undirected graph? Well, since $prev[e]$ tracks the first vertex to explore e, we can traverse this until we reach our original vertex s, so the path becomes: $\mathcal{P} = \{e, prev[e], prev[prev[e]], \ldots, s\}$. ~~Notice that the exploration order of EXPLORE() actually makes this the shortest path, since $prev[v]$ is only modified the *first* time that v is visited (and the earliest visit obviously comes from the quickest exploration).~~

(I don't think this last statement is actually true, so proceed with caution)

3.3.2 Directed Graphs

Directed graphs (also called **digraph**s) can be traversed as-is using algorithm 3.2, but finding paths is a little more challenging from an intuitive standpoint. However, its implementation is just as simple as before.

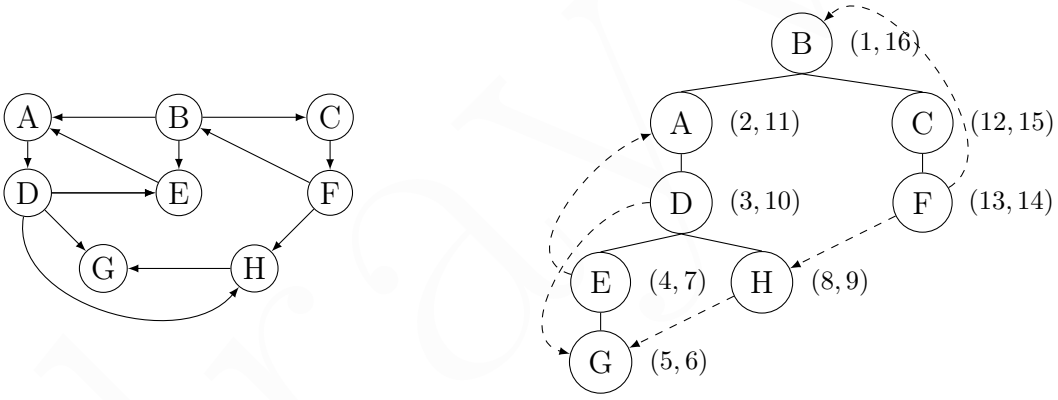


Figure 3.1: A directed graph (left) and its DFS tree with pre- and post-visit clock values at each vertex (right). The dashed edges in the tree represent edges that were ignored because the vertices were already marked "visited."

We'll create a so-called "clock" that tracks the first and last "times" that a node was visited, putting these in global $pre[\cdot]$ and $post[\cdot]$ arrays, respectively. Specifically, we'll define PREVISIT(v) to store $pre[v] =$ clock++ and POSTVISIT(v) to do likewise but store in $post[v]$.[1] An example of the values in these variables at each vertex is shown in Figure 3.1 after a DFS traversal. Note that a DFS traversal tree can start at any node; the tree in Figure 3.1 is just one of the possible traversals of its graph (namely, when starting at vertex B).

Types of Edges Given a directed edge $z \to w$, it's classified as a **tree edge** if it's one of the primary, first-visited edges. Examples from Figure 3.1 include all of the

[1] Here, the post-increment syntax is an allusion to C, so the clock is incremented *after* assigning its previous value to the *post* array.

black edges, like (B, A). The other, "redundant" edges are broken down into three categories depending on their relationship in the graph; it's called a...

- **back edge** if it's an edge between vertices going *up* the traversal tree to an **ancestor**. Examples from Figure 3.1 include the dashed edges like (F, B) and (E, A).
- **forward edge** if it's an edge going *down* the tree to a **descendant**. Examples from Figure 3.1 include the dashed edges like (D, G) and (F, H).
- **cross edge** if it's an edge going *across* the tree to a **sibling** (a vertex at the same depth). Unfortunately there are no examples in Figure 3.1, but it could be (E, H) if such an edge existed.

In all but the forward-edge case, $post[z] > post[w]$; in the latter case, though, it's the inverse relationship: $post[z] < post[w]$.

The various categorizations of (possibly-hidden) edges in the DFS traversal tree can give important insights about the graph itself. For example,

Property 3.1. *If a graph contains a cycle, the DFS tree will have a back edge.*

3.3.3 Acyclic Digraphs

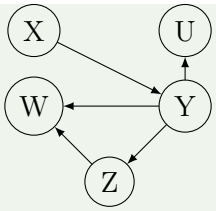
A **directed acyclic graph** (or DAG) contains no cycles (and thus no back edges, as we just saw). Given a DAG, we want to *topologically sort* it: order the vertices so that the higher nodes of the DFS tree have less children than than lower nodes (that is, the degree of a vertex is inversely-proportional to its depth in the DFS tree).

We can achieve this by running DFS and ordering the vertices by their post-order clock values; this takes linear time.[2]

EXAMPLE 3.1: **Topological Ordering**

Given the following graph, arrange the vertices by (one of their possible) topological ordering:

[2] It takes linear time since we can avoid a $\mathcal{O}(n \log n)$ sort: given a fixed number of contiguous, unique values (the post-orders go from $1 \to 2n$), we can just put the vertex into its slot in a preallocated $2n$-length array. DFS takes linear time, too, so it's $\mathcal{O}(n + m)$ overall.



Answer: $\{X, Y, Z, U, W\}$, though that's only one of the three possible answers. In all cases, X and Y must go first.

After we've transformed a graph based on its topological ordering, some vertices have important properties:

- a **source** vertex has no *incoming* edges ($d_{\text{in}}(v) = 0$)
- a **sink** vertex has no *outgoing* edges ($d_{\text{in}}(v) = 0$)

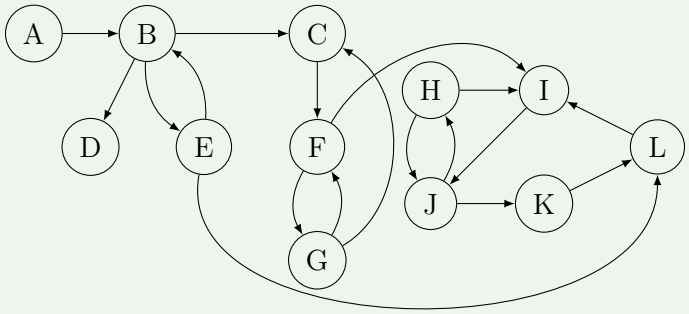
A DAG will always have at least one of each. By this rationale, we can topologically sort a DAG by repeatedly removing sink vertices until the graph is empty. The question now becomes: how do we find sink vertices?

First, we'll answer a related question: what's the analog of a connected component in generic digraphs?

3.4 Strongly-Connected Components

Vertices v and w are **strongly connected** if there is a path from $v \rightsquigarrow w$ and vice-versa, from $w \rightsquigarrow v$. Then, a **strongly-connected component** in a digraph is a maximal set of strongly-connected vertices.

Identify the strongly-connected components in this directed graph:



Answer: There are five.

The highlighted components are rendered in Figure 3.2.

Consider the meta-graph of the strongly-connected components presented in the example above: what if we collapsed each component into a single vertex? The labeled result (see Figure 3.2) is actually a DAG in itself!

This will always be the case, giving us a nice property:[3]

Property 3.2. *Every directed graph is a directed acyclic graph of its strongly-connected components.*

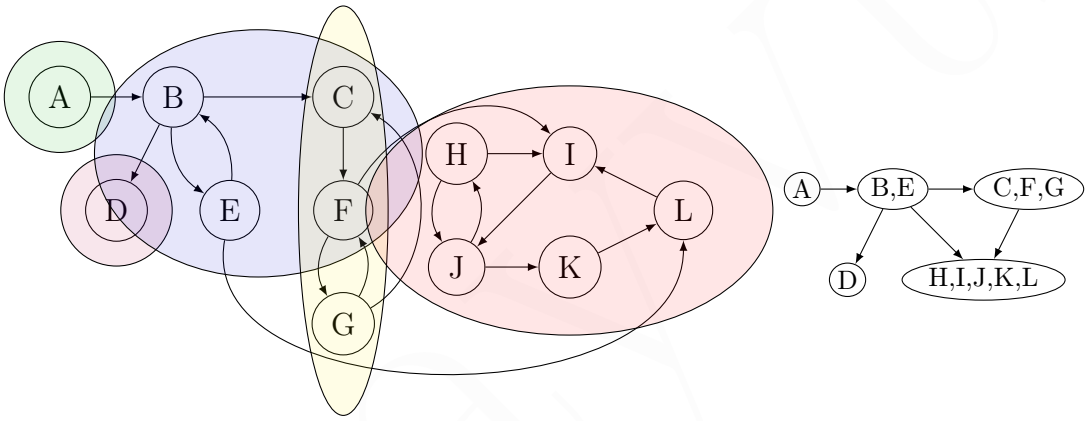


Figure 3.2: A directed graph broken up into its five strongly-connected components, and the resulting meta-graph from collapsing these components into a single "meta-vertex" (again, ignore the implication that vertex F is part of the blue component—this is just a visual artifact).

Let's discuss the algorithm that finds the strongly-connected components; in fact, the *order* in which it finds these components will be its topologically-sorted order (remember, that's in order of decreasing post-visit numbers). We'll actually achieve this in linear time with two passes of DFS.

3.4.1 Finding SCCs

Remember our basic idea for topologically sorting a graph: find a sink, rip it out, and repeat until the graph is empty. The idea for finding SCCs is similar: we're going to find a sink *component* (that is, a strongly-connected component that is a sink in its metagraph, so it only has incoming edges) and output it, repeating until the graph is empty.

[3] The proof of this is fairly straightforward: if there was a cycle in the metagraph, then there is a way to get from some vertex S to another vertex S' and back again (by definition). However, S and S' can't connected to each other, because if they were, they'd be a single strongly-connected component. Thus we have a contradiction, and the post-SCC metagraph must be a DAG. ■

That begs the question: how do we find a sink SCC?[4] More specifically, as explained in footnote 4, how do we find any vertex within a sink component?

A key property holds that will let us find such a vertex:

> **Property 3.3.** *In a general directed graph, the vertex v with the* ***highest*** *post-visit order number will always lie in a* ***source*** *strongly-connected component.*

Given this property, how do we find a vertex w in a *sink* SCC? Well, it'd be great if we could invert our terms: we can find a source SCC and want a sink SCC, but what if we could find sink SCCs and wanted to find a source SCC? Well, what if we just... reversed the graph? So now sink vertices become sources and vice-versa, and we can use the property above to find a source-in-reversed-but-sink-in-original vertex!

For a digraph $G = (V, E)$, we look at $G^R = (V, E^R)$, the reverse graph of G, where $E^R = \{(w, v) : (v, w) \in E\}$, the reverse of every edge in E. All of our previous notions hold: the source SCCs in G become the sink SCCs in G^R and vice-versa, so the topologically-sorted DAG is likewise reversed. From a high level, the algorithm looks as follows:

1. First, create the reverse graph G^R.
2. Perform DFS on it to create a traversal tree.
3. Sort the vertices decreasing by their post-order numbers to identify the vertex v to start DFS from in the *original* graph.
4. Run the standard connected-component algorithm using DFS (from algorithm 3.2) on G starting at v—increase a component count tracker cc whenever exploration of a vertex ends—to label each SCC.
5. When DFS terminates for a component, set v to the next unexplored vertex with the highest post-order number and go to Step 4.
6. The output is the metagraph DAG in reverse topological order.

3.5 Satisfiability

This is an application of our algorithms for determining (strongly-)connected components: solving **SAT** or **sat**isfiability problems.

[4] Though the same rationale of "find one and remove it" applies to source SCCs, sinks are easier to work with. If v is in a sink SCC, then Explore(v) visits all of the vertices in the SCC and nothing else; thus, all visited vertices lie in the sink. The same does not apply for source vertices because exploration does not terminate quickly.

3.5.1 Notation

First, we need to define what a Boolean formula is; we're given:

$$\begin{aligned} n \textbf{ variables}&: \quad x_1, x_2, \ldots, x_n \\ 2n \textbf{ literals}&: \quad x_1, \overline{x_1}, x_2, \overline{x_2}, \ldots, x_n, \overline{x_n} \end{aligned}$$

combined using the logical operators AND ($\wedge$) and OR ($\vee$) into **conjunctive normal form** (CNF).

A **clause** is the OR of several literals: for example, $x_3 \vee \overline{x_5} \vee \overline{x_1}$; to "satisfy" a clause, at least one of its members must be true. Finally, a **formula** in CNF is the AND of m clauses; for example:

$$f = \underbrace{(x_2)}_{\text{clause}} \wedge \underbrace{(\overline{x_3} \vee x_4)}_{\text{clause}} \wedge \underbrace{(x_3 \vee \overline{x_5} \vee \overline{x_1} \vee x_2)}_{\cdots} \wedge (\overline{x_2} \vee \overline{x_1})$$

To satisfy f, at least one **literal** in each **clause** must be true. For example, $x_1 = F$, $x_2 = T$, and $x_3 = F$ satisfies the above formula. Any formula can be converted to CNF, but its size may blow up quickly.

3.5.2 An SAT Problem

Given an input formula f in CNF with n variables and m clauses, we want to find the assignment (T or F to each variable) that **satisfies** f, if one exists (or "no" otherwise).

EXAMPLE 3.2: **SAT**

Given the following formula,

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

assign a value to each x_i to satisfy f.

ANSWER: $x_1 = F, x_2 = T, x_3 = F$ is a viable solution.

Generally-speaking, a k-SAT problem limits the clauses to having $\leq k$ elements, so the above example is 3-SAT. We'll see later than k-SAT is NP-complete for all $k \geq 3$, but there's a polynomial time algorithm for 2-SAT.

3.5.3 Solving 2-SAT Problems

Since we are limiting our clauses to have up to two elements, we can break any formula down into two categories: either all of the clauses have two elements, or at least one is a **unit clause**.

Simplifying

A unit clause is a clause with a single literal, such as (x_4). If a formula f has unit clauses, we know more about the satisfiability conditions and can employ a basic strategy:

1. Find a unit clause, say the literal a_i.
2. Satisfy it, setting $a_i = T$.
3. Remove clauses containing a_i (since they're now satisfied) and drop the literal $\overline{a_i}$ from any clauses that contain it.
4. Let f' be the resulting formula, then go to Step 1.

Obviously, f is satisfiable iff f' is satisfiable. We repeatedly remove unit clauses until either (a) the whole function is satisfied or (b) only two-literal clauses exist. We can now proceed as if we have two-literal clauses.

Graphing 2-SATs

Let's convert our n-variable, m-clause formula (where every clause has two literals) to a directed graph:

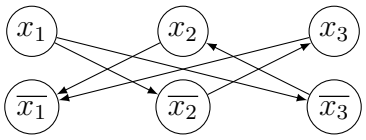
- $2n$ vertices corresponding to $x_1, \overline{x_1}, \ldots, x_n, \overline{x_n}$.
- $2m$ edges corresponding to two "implications" per clause.

In general, given we have the clause $(\alpha \vee \beta)$, the implication graph has edges $\overline{\alpha} \to \beta$ and $\overline{\beta} \to \alpha$.

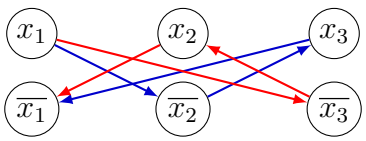
For example, given

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

the resulting graph would be:



Now, we follow the path for a particular literal. For example, here are the implication paths starting from $x_1 = T$:



Clearly, there's a path $x_1 \rightsquigarrow \overline{x_1}$, and obviously that implication is nonsense—$x_1 \implies \overline{x_1}$ is a contradiction. However, if $x_1 = F$, we have no leads (that is, $\overline{x_1}$ is a sink

vertex), so it *might* be okay?

Obviously, if $x_1 = F$ also led to an implication path resulting in a contradiction, we could conclude that f is not satisfiable. Purely from the perspective of graph theory, having a path from x_1 to $\overline{x_1}$ and vice-versa implies that they're part of a connected component.

Property 3.4. *If a literal a_i and its inversion $\overline{a_i}$ are part of the same strongly-connected component, then f is not satisfiable.*

With this property, we can prove when f is satisfiable: if, for every variable x, x_i and $\overline{x_i}$ are in *different* strongly-connected components, then f is satisfiable. Here's an algorithm that adds a little more structure to this basic intuition:

1. Find the sink SCC, S.
2. Set $S = T$; that is, satisfy all of the literals in S.
3. Since these are all tail-ends of their implications, the head will implicitly be satisfied. Thus, we can rip out S from the graph and repeat.
4. Then, $\overline{S}$ will be a *source* SCC, so that setting $\overline{S} = F$ has no effect (setting the head of an implication to false means we don't need to follow any of its outgoing edges).

This relies on an important property which deserves a proof.

Property 3.5. *If $\forall i$, the literals x_i and $\overline{x_i}$ are in different strongly-connected components, then:*

$$S \text{ is a sink SCC} \iff \overline{S} \text{ is a source SCC}$$

Given this property, we can formalize the 2-SAT algorithm a little further:

1. First, assume that the Boolean function in CNF-form, f, only has two-literal clauses (since we can force that to be the case by simplifying unit clauses).
2. Construct the graph G for f.
3. Find a sink SCC, S, using DFS and set $S = T$ and $\overline{S} = F$.
4. Remove both S and $\overline{S}$.
5. Repeat until the graph is empty.

The only complex task here is finding SCCs which we know takes linear time, so the overall running time is $\mathcal{O}(n + m)$.

3.6 Minimum Spanning Trees

The input to the **minimum spanning tree** problem (or MST) is an undirected graph $G = (V, E)$ with weights $w(e)$ for $e \in E$. The goal is to find the minimal-size, minimum weight, connected subgraph that "spans" the graph. Naturally, the overall weight of a tree is the sum of the weights of edges.

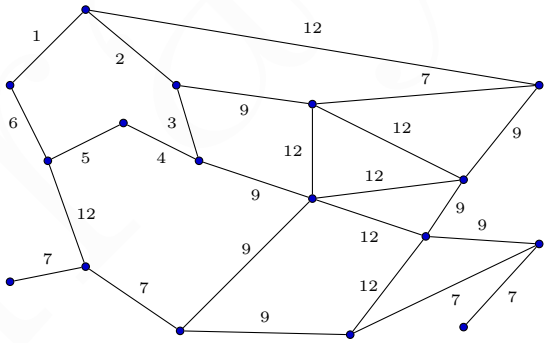
Note some basic properties about trees:

- A tree on n vertices has exactly $n - 1$ edges.
- Exactly one path exists between every pair of vertices.
- *Any* connected $G = (V, E)$ with $|E| = |V| - 1$ is a tree.

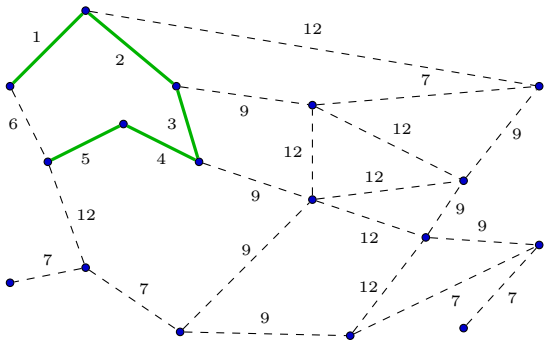
  This last point bears repeating, as it will be important in determining the minimum cut of a graph, later. The only thing necessary to prove that a graph is a tree is to show that its edge count is its vertex count minus one.

3.6.1 Greedy Approach: Kruskal's Algorithm

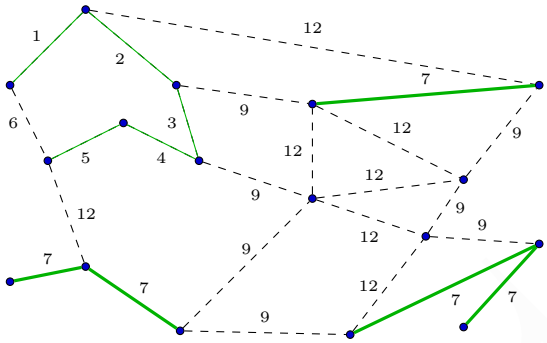
We'll first attempt to build the minimum spanning tree by consistently building up from vertices with the lowest cost without creating any cycles. Our running example graph will be:



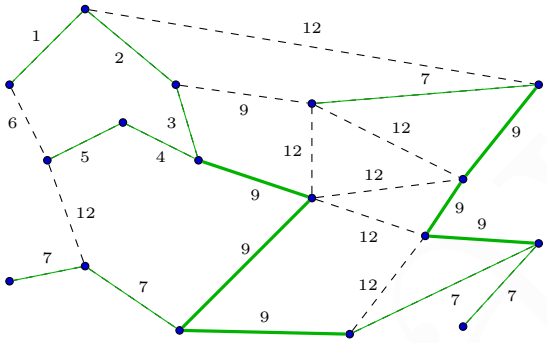
The first 5 edges are extremely easy to determine, but the 6-cost edge would cause a cycle:

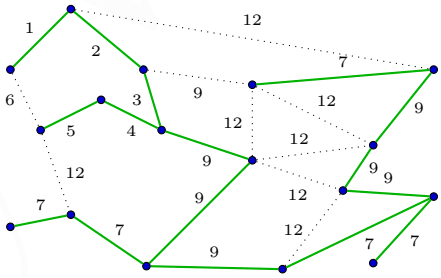


All of the 7-cost edges can be added in without trouble—notice that them not being connected is not relevant:



Finally, all but the last 9-cost edge can be added in to create a final MST for the graph. Notice that the vertices touched by the unused 9-cost edge are connected by a longer path.





This is **Kruskal's algorithm**, and its formalized in algorithm 3.3, below. Its **running time** is $\mathcal{O}(m \log n)$ overall, since sorting takes $\mathcal{O}(m \log n)$ time and checking for cycles takes $\mathcal{O}(\log n)$ time using the **union-find** data structure and operates on m items.[5] Its correctness can be proven by induction on X, the MST for a subgraph of G, as we add in new edges.

3.6.2 Graph Cuts

The proof of correctness of Kruskal's algorithm (omitted from the above) is deeply related to **graph cuts**, which are our next topic of discussion. A cut in a graph is a set of edges that partitions the graph into two subgraphs. Formally, for an undirected graph $G = (V, E)$ and the partition $V = S \cup \overline{S}$, the cut is:

$$\begin{aligned} \text{cut}\left(S, \overline{S}\right) &= \{(v, w) \in E : v \in S, w \in \overline{S}\} \\ &= \text{edges crossing } S \longleftrightarrow \overline{S} \end{aligned}$$

An cut on our running example graph is demonstrated in Figure 3.3. We'll look at determining both the minimum cut—the fewest number of edges necessary to divide a

[5] Reference *Algorithms*, pp. XX for details on the union-find data structure.

Algorithm 3.3: Kruskal$(\cdot)$, a greedy algorithm for finding the minimum spanning tree of a graph.

Input: An undirected graph, $G = (V, E)$ and its edge weights, $w(e)$.
Result: The minimum spanning tree of G.

Sort E by ascending weight (via MergeSort(E), etc.)
Let $X := \emptyset$
foreach $e = (v, w) \in E$ **do** `// in sorted order`
  if $X \cup e$ *does not create a cycle* **then**
    $X = X \cup e$
  end
end
return X

graph into two components—and the maximum cut—the cut of the largest size—later when discussing optimization problems.

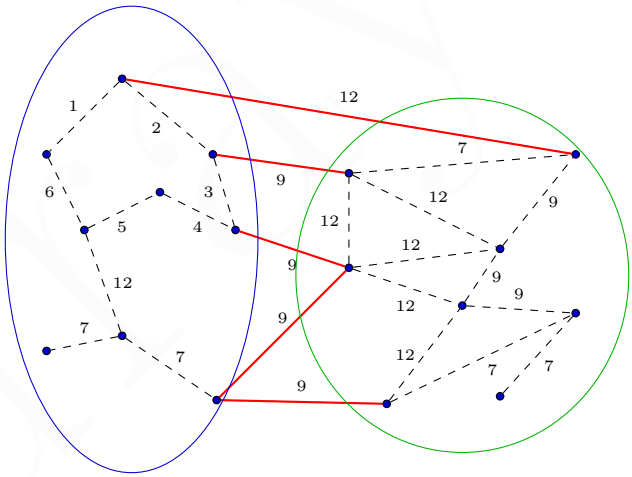


Figure 3.3: The edges in red cut the graph into the blue and green subgraphs.

Cuts come with an important property that becomes critical in proving correctness of MST algorithms. In one line, its purpose is to show that **any minimum edge across a cut will be part of an MST**.[6]

[6] Though its important to understand the proof of correctness of Property 3.6, it's left out here because the lectures cover the material to a sufficient degree of formality.

Property 3.6 (cut property)**.** *For an undirected graph, $G = (V, E)$, given:*

- *a subset of edges that is also a subset of a minimum spanning tree T of G (even if T is unknown); that is, for:*

$$X \subset E \text{ and } X \subset T$$

- *and a subset of vertices $S \subset V$ where no edge of X is in*

$$\text{cut}\left(S, \overline{S}\right)$$

any of the minimum-weight edges, e^*, in the above cut will be in a new minimum spanning tree, T'. That is:

$$X \cup e^* \subset T'$$

Note that T' after adding the e^* described in Property 3.6 and the original (potentially-unknown) T may be different MSTs, but the purpose is to find *any* MST rather than a specific MST. Critically, the weight of T' is *at most* the weight of T, so we're always improving.

3.6.3 Prim's Algorithm

This variant for creating the minimum spanning tree is reminiscent of Dijkstra's algorithm. It's still a greedy algorithm, but constrains the edges it can consider. Its purpose is to always maintain a subtree along the way to finding the MST, so it will only consider edges that are connected, choosing the cheapest one. It's similarity to Dijkstra's is reflected by its running time: $\mathcal{O}((m+n)\log n)$.

3.7 Flow

The idea of **flow** is exactly what it sounds like: given some resource and a network describing an infrastructure for that resource, a flow is a way to get said resource from one place to another. To model this more concretely as a graph problem, we have source and destination vertices and edges that correspond to "capacity" between points; a flow is a path through the graph.

Given such a network, it's not unreasonable to try to determine the **maximum flow** possible: what's the maximum amount of resource sendable from $s \rightsquigarrow t$ without exceeding any capacities?

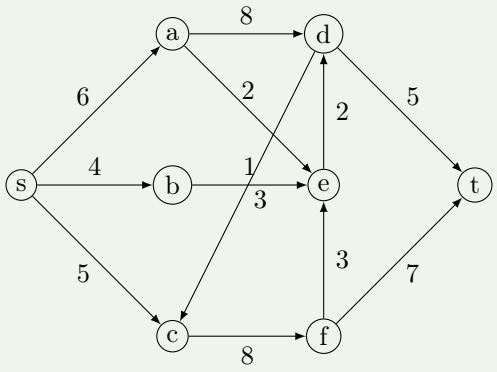
Formally, the **input** is a **flow network**: a directed graph $G = (V, E)$, a designated $s, t \in V$, and a capacity for each $e \in E : c_e > 0$. The **goal** is to *find a flow*—that is, a

set of "usages" on each edge—that does not exceed any edge's capacity and maximizes the incoming flow to t.

Notice from the example below that cycles in a flow network are not problematic: there's simply no reason to utilize an *entire* cycle when finding a flow.

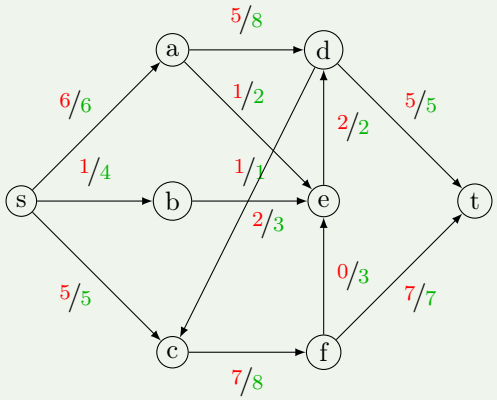
EXAMPLE 3.3: **Max-Flow**

Given the following graph:



determine a flow of maximum size that satisfies the edge constraints.

The solution is as follows:



This is obviously the maximum flow because the total capacity incoming to t is 12, so we've reached the hard cap. Namely,

$$\text{size}(f) = \underbrace{6+1+5}_{\text{outgoing}} = \underbrace{5+7}_{\text{incoming}} = \boxed{12}$$

The key construction in calculating the maximum flow network is the **residual flow network**, which effectively models the flow capacities remaining in a graph as well as the "reverse flow." Specifically, the network $G^f = (V, E^f)$ is modeled with capacities

c^f such that:

$$\begin{cases} c_{uv} - f_{uv} & \text{if } (u,v) \in E \text{ and } f_{uv} < c_{uv} \\ f_{vu} & \text{if } (u,v) \in E \text{ and } f_{vu} > 0 \end{cases}$$

3.7.1 Ford-Fulkerson Algorithm

The running time of the **Ford-Fulkerson algorithm** is **pseudo-polynomial** in $\mathcal{O}(C\,|E|)$, where C is the theoretical maximum output flow to t (the sum of its incoming vertices' capacities).

This analysis relies on a (huge) assumption that all capacities are integers; then, the algorithm guarantees that the flow increases by ≥ 1 unit per round. Since there are $\leq C$ rounds to the algorithm, and a single round takes $\mathcal{O}(n+m)$ time (dominated by path-finding), it requires $\mathcal{O}(mC)$ time in total (if we assume $|E| \geq |V|$ to simplify).

Algorithm 3.4: The Ford-Fulkerson algorithm for computing max-flow.

Input: A flow network: a digraph $G = (V, E)$, a capacity for each $e \in E$, c_e, and start/end vertices $s, t \in V$.
Result: The maximal flow graph, where f_e indicates the maximum flow along the edge $e \in E$.

$\forall e \in E : f_e = 0$
repeat
  Build the residual network, G^f for the current flow, f.
  Find *any* path $P = s \rightsquigarrow t$ in G^f.
  Let $c(P)$ be the smallest capacity along P in G^f: $c(P) = \min_{c_e}\{e \in P\}$.
  Augment f by $c(P)$ units along P.
until *no* $s \rightsquigarrow t$ *path exists in* G^f
return f

Its correctness follows from the max-flow–min-cut theorem, shown (and proven) later.

3.7.2 Edmonds-Karp Algorithm

This algorithm fundamental varies in the way it augments f. Rather than using *any* path (found via depth-first search, for example), it uses the *shortest* path found by breadth-first search. When this is the case, it can be proven that it takes at-most mn rounds to find the maximum flow; thus, the overall running time of the **Edmonds-Karp algorithm** is $\mathcal{O}(nm^2)$.

FUN FACT: **Record-Breaker**

James Orlin achieved a max-flow algorithm with a $\mathcal{O}(mn)$ run time in 2013.

3.7.3 Variant: Flow with Demands

This variant on the max-flow problem poses demands on certain edges, requiring that they have a particular capacity. We will see how to reduce this to the regular max-flow problem.

The input is a flow network, as before, as well as a demand for each edge:

$$\forall e \in E : d(e) \geq 0$$

A *feasible* flow f is thus one that fulfills both the demand and does not exceed the capacity: $d(e) \leq f(e) \leq c(e)$. If we can find the existence of a feasible flow, augmenting it to be the maximum feasible flow will be straightforward.

We want to find a new flow network—with graph $G' = (V', E')$ and edge capacities $c'(e)$—that we can plug-and-play into our existing max-flow algorithms.

Integrating the demands into the capacities is straightforward: an edge with capacity 5 and demand 3 is similar to an edge with capacity 2 and no demand. This is the first step in our reduction: $c'(e) = c(d) - d(e)$.

This is insufficient, though. We need to model the demand itself. Each vertex has some amount of incoming demand and some outgoing demand. To ensure its fulfillment, we connect every $v \in V$ to a new "master" source s' and sink t'. Namely,

1. add an edge $s' \to v$ with $c'((s', v)) = d^{\text{in}}(v)$ (where $d^{\text{in}}(v)$ is the total incoming demand to v) for every vertex *except* s.
2. add an edge $v \to t'$ with $c'((v, t)) = d^{\text{out}}(v)$ for every vertex *except* t.

Finally, to ensure that the flow out of s matches the flow into t, we create an edge $t \to s$ with $c'((t, s)) = \infty$.

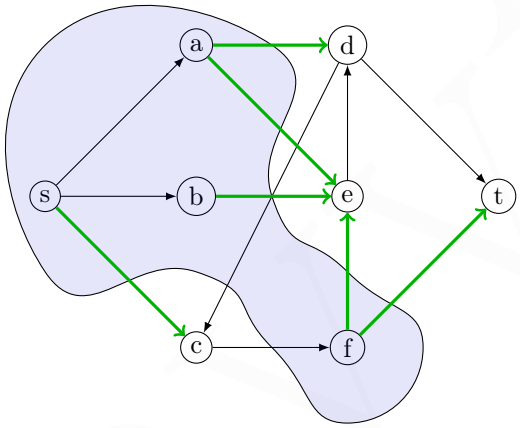
For a flow f in G', we know it cannot exceed the total demand, so $\text{size}(f') \leq D$. We call f a **saturated** flow if $\text{size}(f') = D$.

Property 3.7. *A flow network G with additional demands has a feasible flow if the constructed G' has a saturated flow.*

3.8 Minimum Cut

Remember, a cut is a way to split (or **partition**) a graph into two parts—call them "left" and "right" subgraphs—so $V = L \cup R$. In this section, we'll focus on a specific type of cut called an st-cut, which separates the vertices s and t such that $s \in L$ and $t \in R$.

This is an st-cut within our example graph from before:

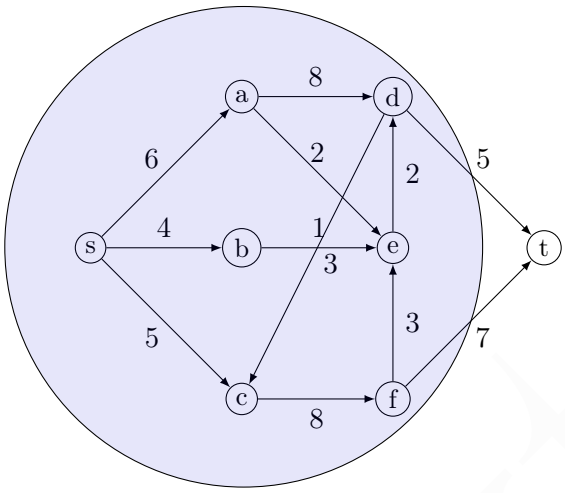


Notice that neither subgraph actually needs to be connected: $f \in L$ but is not connected to a, b, or s. We're interested in the capacity of this cut; this is the total capacity of edges from $L \rightsquigarrow R$. These are the edges that "exit" L and are highlighted in green above.

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

The capacity of the above example cut is 26 (again, simply sum up the capacities of the green edges). Given these definitions and example, we can formulate a problem.

Here's a min-st-cut of our example graph:



How do we *know* that this is a min-st-cut? Because its capacity is of size 12, and we're about to prove that the capacity of the minimum cut is equal to the capacity of the maximum flow. Obviously, 12 is the *theoretical* maximum flow on this graph, so since this cut is equal to it, we know it's optimal (remember that we actually showed that it's also the *actual* maximum flow in Example 3.3 above).

3.8.1 Max-Flow = Min-Cut Theorem

This section walks through the proof that the minimum capacity of an st-cut is equal to the size of the maximum flow for a graph.[7] This is the **max-flow–min-cut theorem**; it's a long proof, but it's broken down into two key, independently-digestible parts.

To show that $\substack{\text{size of}\\\text{max flow}} = \substack{\text{min capacity}\\\text{of } st\text{-cut}}$ we will proceed by showing

$$\substack{\text{size of}\\\text{max flow}} \leq \substack{\text{min capacity}\\\text{of } st\text{-cut}} \qquad \text{and} \qquad \substack{\text{size of}\\\text{max flow}} \geq \substack{\text{min capacity}\\\text{of } st\text{-cut}}$$

separately, which clearly implies that they are actually equal.

Forward Direction

First, we aim to prove that the size of the maximum flow is *at most* the minimum st-cut:

$$\substack{\text{size of}\\\text{max flow}} \leq \substack{\text{min capacity}\\\text{of } st\text{-cut}}$$

We can actually achieve this by proving a more general, simpler solution, proving that for *any* flow f and *any* st-cut (L, R), that the flow is at most the cut capacity:

$$\text{size}(f) \leq \text{capacity}(L, R)$$

[7] It's important to note that we're explicitly specifying that it's the minimum st-cut—which separates s and t—as opposed to simply the minimum cut. In the latter case, there's no relation to the implicit s and t in the max-flow problem.

Obviously, if it's true for any flow and any cut, it's true for the maximum flow and the minimum cut.

Claim 1: The size of a flow is the flow coming out of L sans the flow coming in to L (notice that R is not involved):

$$\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L)$$

Let's expand this definition: the out-flow is the total flow along edges leaving L and the in-flow is similarly the total flow along edges entering L.

$$f^{\text{out}}(L) - f^{\text{in}}(L) = \sum_{\substack{(v,w)\in E:\\ v\in L, w\in R}} f_{vw} - \sum_{\substack{(w,v)\in E:\\ w\in R, v\in L}} f_{wv}$$

We'll pull a trick from middle school algebra out of our hat: simultaneously add and subtract the same value so that the whole equation stays the same. That value's name? ~~Albert Einstein~~ The total flow of edges within L:

$$= \sum_{\substack{(v,w)\in E:\\ v\in L, w\in R}} f_{vw} - \sum_{\substack{(w,v)\in E:\\ w\in R, v\in L}} f_{wv} + \underbrace{\sum_{\substack{(v,w)\in E:\\ v\in L, w\in L}} f_{vw} - \sum_{\substack{(w,v)\in E:\\ w\in L, v\in L}} f_{wv}}_{=0\text{, same edges, just relabeled}}$$

Notice the first and third term: one is all edges $v \rightsquigarrow R$, while the other is all edges $v \rightsquigarrow L$. Together, this is all of the flow leading out of vertices in L. Similarly, the second and third terms are the total flows leading *in* to any v in L. Thus,

$$= \sum_{v\in L} f^{\text{out}}(v) - \sum_{v\in L} f^{\text{in}}(v)$$

Now consider the source vertex, s. Its incoming flow is obviously zero by definition:

$$= \sum_{v\in L\setminus s} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right) + f^{\text{out}}(s)$$

However, by the very definition of a valid flow, the incoming flow must equal the outgoing flow! Thus the entire summation is zero. And again, by definition, the size of a flow is the size of the source vertex's total flow, so we arrive at our original claim:

$$\begin{aligned} f^{\text{out}}(L) - f^{\text{in}}(L) &= \underbrace{\sum_{v\in L\setminus s} \left(f^{\text{out}}(v) - f^{\text{in}}(v)\right)}_{=0} + f^{\text{out}}(s) \\ &= f^{\text{out}}(s) \\ &= \text{size}(f) \quad \blacksquare \end{aligned}$$

Claim 2: Let's finish the job. We now claim (and show) our original goal:

$$\text{size}(f) \leq \text{capacity}(L, R)$$

From *Claim 1*, we have

$$\text{size}(f) = f^{\text{out}}(L) - f^{\text{in}}(L)$$

Obviously, if $a = b - c$, then $a \leq b$ alone. Thus,

$$\text{size}(f) \leq f^{\text{out}}(L)$$

Now what exactly is $f^{\text{out}}(L)$? It's the total flow leaving L. But obviously the total flow is limited to the total capacity of each edge—any edge flow exceeding its capacity would be an invalid flow. Thus,

$$\text{size}(f) \leq f^{\text{out}}(L) \leq \text{capacity}(L, R) \quad \blacksquare$$

Backward Direction

Whew, halfway there. Now we aim to prove that the maximum flow is at least the minimum cut's capacity:

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

This proof's approach is quite different than the other. We start with the optimal max-flow f^* from the Ford-Fulkerson algorithm. By definition, the algorithm terminates when there is no path $s \rightsquigarrow t$ in the residual graph G^{f^*} (refer to algorithm 3.4).

Goal Given this fact, we're going to construct an (L, R) cut such that:

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

This will prove our statement. Even if f^* is not the max-flow, it's at no bigger than the max-flow. Similarly, the capacity is at least as big as the minimum. In other words,

$$\begin{aligned} \text{Since:} \quad & \max_f \text{size}(f) \geq \underbrace{\text{size}(f^*) = \text{capacity}(L, R)} \geq \min_{(L,R)} \text{capacity}(L, R) \\ \text{then:} \quad & \max_f \text{size}(f) \geq \min_{(L,R)} \text{capacity}(L, R) \end{aligned}$$

Cut Construction Again, the underbraced statement is what we're aiming to prove. We know that there's no path $s \rightsquigarrow t$ in the residual, so let's let L be the vertices that *are* reachable from s in the residual, G^{f^*}. Then, let $R = V \setminus L$, the remaining vertices. We know $s \in L$ and $t \notin L$, so (L, R) is an st-cut by definition.

Now we need to prove that this constructed (L, R) cut has a capacity equal to the flow size.

Cut Capacity What are the properties of this cut? Again, L is the set of vertices reachable from s in the residual network.

What about the edges leading from $L \rightsquigarrow R$ in the *original* network? Well, they must be **saturated**: if an edge (v, w) wasn't, then there'd be an edge (v, w) in the residual and thus w would be reachable from L. This is a contradiction on how we constructed L.

Thus, for every $(v, w) \in E$ where $v \in L, w \in R$, we know $f^*_{vw} = c_{vw}$. Since every edge leaving $L \rightsquigarrow R$ is saturated, the total flow leaving L is their sum, which is the capacity of L by definition. Thus:

$$f^{*\text{out}}(L) = \text{capacity}(L, R)$$

Now consider the edges $R \rightsquigarrow L$. Any such edge (z, y) *incoming* to L in the original graph must not have its back-edge (y, z) appear in the residual. If it did, then there's a path $s \rightsquigarrow z$, so $z \in L$, but we just said that $z \in R$! Another contradiction. Thus, for every $(z, y) \in E$ where $z \in R, y \in L$, we know that $f^*_{zy} = 0$.

$$f^{*\text{in}}(L) = 0$$

In part of our proof in the forward direction, we showed that the size of a flow is related to the flows of L (see *Claim 1*). By simple substition, we achieve our goal and so prove the backward direction:

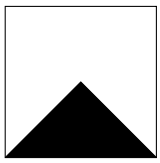
$$\begin{aligned}\text{size}(f^*) &= f^{*\text{out}}(L) - f^{*\text{in}}(L) = f^{*\text{out}}(L) \\ &= \text{capacity}(L, R) \quad \blacksquare\end{aligned}$$

We've shown that the inequality holds in both directions. This completes the proof of the theorem: the size of the maximum flow is equal to the capacity of the minimum st-cut. ■

3.8.2 Application: Image Segmentation

The above proof provided us with some beautiful corollaries. By operating under the assumption that f^* from Ford-Fulkerson is not the max flow, we still proved the theorem. This thus proves the correctness and optimality of Ford-Fulkerson: if you stop when there is no longer a path $s \rightsquigarrow t$ in the residual network, you have found the maximum flow. Furthermore, the constructed (L, R) cut from the maximum flow gives a minimum cut.

These ideas are key in image segmentation, where the goal is to separate the image into objects. We'll keep it simple, aiming to separate the foreground from the background. For example, we want to discern the triangle from the rest of the "image" on the right.



Problem Statement

We're going to interpret images as an undirected graph $G = (V, E)$: vertices are pixels and edges connect neighboring pixels. For example,

We also have the following parameters that we assume to be non-negative:

- For each pixel $i \in V$, we track its likelihood of being either in the foreground (f_i) or the background (b_i)—though it might make sense for $f_i + b_i = 1$ (probabilities), this isn't necessarily required.
- Additionally, there's a "separation penalty" p_{ij} across each edge $(i, j) \in E$ which represents the cost of segmenting into different objects along that edge.

> **Keep It Simple, Stupid**
>
> Since we're studying algorithms rather than computer vision, our examples are very simplistic and these parameters are hard-coded. However, in a realistic setting, these values would be otherwise gleaned from the image. For example, hard edges in an image are typically correlated with object separation, so edge detection might be used to initialize the separation penalties. Similarly, you might use a statistical metric like "most common pixel color" and a color difference to quantify the fore/background likelihoods.

Our goal is to partition the pixels into a foreground and a backgrond: $V = F \cup B$. We define the "quality" of a partition as follows: it's the total likelihood of the foreground plus the total likelihood of the background minus the total separation penalty.

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

(note that the edges go $F \rightsquigarrow B$, so $i \in F$, $j \in B$)

Thus, the overall goal is to find: $\max_{(F,B)} w(F, B)$. We're going to solve this by reducing to the min-cut problem. There are a few key points here from the high level: our max should be a min and our totals need to be positive (so the separation penalty total should be positive).

Reformulation: Weights

Let $L = \sum_{i \in V} (f_i + b_i)$, the sum of likelihoods for every pixel. Then, we can reformulate the weight problem as follows. Since,

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

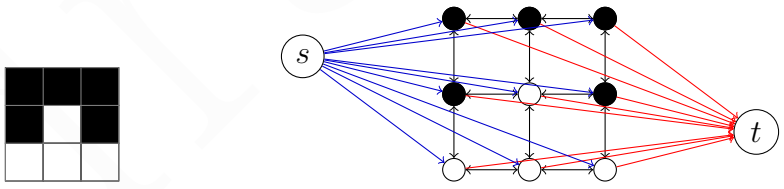
we can substitute in for $w(F, B)$:

$$\begin{aligned} w(F, B) &= \boxed{\sum_{i \in F} f_i + \sum_{j \in B} b_j} - \sum_{(i,j) \in E} p_{ij} \\ &= \underbrace{L - \sum_{i \in F} b_i + \sum_{j \in B} f_j - \sum_{(i,j) \in E} p_{ij}}_{w'(F,B)} \end{aligned}$$

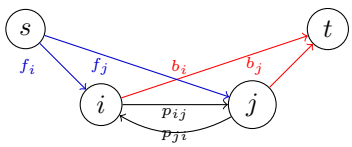
We'll call this latter value $w'(F, B)$. This converts things to a minimization problem: $\max w(F, B) = \min w'(F, B)$. Now we can find the minimum cut by further reducing it to the max-flow.

Reformulation: Flow

To formulate our pixel-graph as a flow network, we'll make some simple modifications. Our undirected graph will be a digraph with each edge being bidirectional; our capacities will be the separation costs; and our source and sink will be new vertices representing the foreground and background:



The foreground and background likelihoods are encoded as the capacities on the edges leading from s and to t, respectively. So for any pair of vertices, we have the following:



When we find the maximum flow of this new network, we will also find the minimum cut by construction (as outlined in the backward direction proof of the max-flow–min-cut theorem). We will find the max-flow f^* such that:

$$\text{size}(f^*) = \text{capacity}(F, B)$$

What's described by this st-cut (F, B)? It's the edges leaving $F \rightsquigarrow B$. For every pixel-vertex $i \in F$, we are cutting across the edge $i \to t$ which has capacity b_i. Similarly, for every vertex $j \in B$, we are cutting $s \to j$ with capacity f_j. Finally, we're also cutting every neighbor $i \to j$ with capacity p_{ij} (since we're only considering edges from F to B, we ignore the $j \to i$ edges). In total:

$$\begin{aligned}
\text{capacity}(F, B) &= \sum_{\substack{(v,w)\in E:\\ v\in F, w\in B}} c_{vw} && \text{definition} \\
&= \sum_{i\in F} b_i + \sum_{j\in B} f_i + \sum_{\substack{(i,j)\in E:\\ i\in F, j\in B}} p_{ij} && \text{substitution} \\
&= w'(F, B)
\end{aligned}$$

*(**Question:** shouldn't it be* minus *the separation costs rather than plus to match $w'(F, B)$? I'm not sure how to reconcile this...)*

Summary

This describes the application of the min-cut–max-flow theorem to image segmentation. Given an input (G, f, b, p), we define the flow network (G', c) on the transformed data as defined above. The max-flow f^* is then related to the min-cut capacity that fulfills our transformed requirement, $\min_{(F,B)} w'(F, B)$, whose cut also maximizes the $w(F, B)$ we described.

Cryptography

I'm going to try to keep this section super general so that I can copy-paste it over to my notes on *Applied Cryptography*.

In asymmetric cryptography, we typically deal with *massive* integers (think 1024-bit values, where $2^{1024} \approx 10^{308}$).

4.1 Modular Arithmetic

I'm not going to review how modular arithmetic actually works, but we've likely all seen it before: $x \bmod 2$ tells us if a number is odd or even. Generally-speaking, $x \equiv y \pmod{N}$ means that x/N and y/N have the same remainder. Generally, $x \bmod N = r$ if there's a multiple of N plus r that works out x, so $qN + r = x$.

An **equivalence class** is the set of numbers which are equivalent under a modulus. So mod 3 has 3 equivalence classes: $\{-6, -3, 0, 3, 6, \ldots\}, \{-2, 1, 4, 7, \ldots\}$, and $\{-1, 2, 5, \ldots\}$.

4.1.1 Modular Exponentiation

Equivalence in modular arithmetic works just like equality in normal arithmetic. So if $a \equiv b \pmod{N}$ and $c \equiv d \pmod{N}$ then $a + c \equiv a + d \equiv b + c \equiv b + d \pmod{N}$.

This fact makes fast modular exponentiation possible. Rather than doing $x^y \bmod N$ via $x \cdot x \cdot \ldots$ or even $((x \cdot x) \bmod N) \cdot x) \bmod N) \ldots$, we leverage repeated squaring:

$$
\begin{aligned}
&x^y \bmod N : \\
&\qquad x \bmod N = a_1 \\
&\qquad\qquad x^2 \equiv a_1^2 \pmod{N} = a_2 \\
&\qquad\qquad x^4 \equiv a_2^2 \pmod{N} = a_3 \\
&\qquad\qquad \ldots
\end{aligned}
$$

Then, we can multiply the correct powers of two to get x^y, so if $y = 69$, you would

use $x^{69} \equiv x^{64} \cdot x^4 \cdot x^1 \pmod{N}$.

Algorithm 4.1: ModExp(x, y, N), the recursive fast modular exponentiation algorithm.

Input: x, y, N, some n-bit positive integers.
Result: $x^y \bmod N$

if $y = 0$ **then**
  return *1*
end
$z = \text{ModExp}\left(x, \left\lfloor \frac{y}{2} \right\rfloor, N\right)$
if y *is even* **then**
  return $z^2 \bmod N$
end
return $xz^2 \bmod N$

4.1.2 Inverses

The **multiplicative inverse** of a number under a modulus is the value that makes their product 1. That is, x is the multiplicative inverse of z if $zx \equiv 1 \pmod{N}$. We then say $x \equiv z^{-1} \pmod{N}$.

Note that the multiplicative inverse does not always exist; if it does, it's **unique**. They exist if and *only* if their greatest common divisor is 1, so when $\gcd x, N = 1$. This is also called being **relatively prime** or **coprime**.

Quick Maffs: Unique Multiplicative Inverse

We claim that if z has a multiplicative inverse mod N, then it is unique. The proof proceeds by contradiction.

Suppose

$$z \equiv x^{-1} \quad \text{and} \quad z \equiv y^{-1} \pmod{N}$$

where $x \not\equiv y \pmod{N}$. By definition, then

$$xz \equiv yz \equiv 1 \pmod{N}$$

but then $x \equiv y \pmod{N}$, contradicting our claim. ■

Greatest Common Divisor

The greatest common divisor of a pair of numbers is the largest number that divides both of them evenly. **Euclid's rule** states that if $x \geq y > 0$, then

$$\gcd(x, y) = \gcd(x \bmod y, y)$$

This leads directly to the **Euclidean algorithm** which runs in $\mathcal{O}(n^3)$ time, where n is the number of bits needed to represent the inputs.[1]

Algorithm 4.2: $\textsc{Gcd}(x, y)$, Euclid's algorithm for finding the greatest common divisor.

Input: x, y, two integers such that $x \geq y \geq 0$.
Result: The greatest common divisor of x and y.

if $y = 0$ **then**
  return x
end
return $\textsc{Gcd}(y, x \bmod y)$

Extended Euclidean Algorithm

Bézout's identity states that if x and y are the greatest common divisors of n, then there are some integers a, b such that:

$$ax + by = n$$

These can be found using the **extended Euclidean algorithm** in $\mathcal{O}(n^3)$ time and are crucial in finding the multiplicative inverse. If we find that $\gcd(x, n) = 1$, then we want to find x^{-1}. By the above identity, this means:

$$\begin{aligned} ax + bn &= 1 \\ ax + bn &\equiv 1 \pmod{n} && \text{taking mod}n \text{ of both sides doesn't change the truth} \\ ax &\equiv 1 \pmod{n} && bn \bmod n = 0 \end{aligned}$$

Thus, finding the coefficient a will find us x^{-1}.

[1] The running time comes from the fact that taking the modulus takes $\mathcal{O}(n^2)$ time and $\textsc{Gcd}(\cdot)$ needs $\leq 2n$ rounds to complete.

Algorithm 4.3: EGCD(x, y), the extended Euclidean algorithm for finding both the greatest common divisor and multiplicative inverses.

Input: x, y, two integers such that $x \geq y \geq 0$.
Result: d, the greatest common divisor of x and y.
Result: a, b, the coefficients fulfilling Bézout's identity, so $ax + by = d$.

if $y = 0$ **then**
  return $(x, 1, 0)$
end
$d, a', b' = \text{EGCD}\,(y, x \bmod y))$
return $(d, b', a' - \lfloor \frac{x}{y} \rfloor b')$

4.1.3 Fermat's Little Theorem

This is the basis of the RSA algorithm we're about to see.

Theorem 4.1 (Fermat's little theorem)**.** *If p is a prime number, then*

$$a^{p-1} \equiv 1 \pmod{p}$$

for any number $1 \leq a \leq p - 1$.

Proof. Let S be the possible values of a in the theorem: $S = \{1, 2, \ldots, p-1\}$

Then, look at $S' = aS \bmod p$:

$$S' = \{1a \bmod p, 2a \bmod p, \ldots, (p-1)a \bmod p\}$$

In general, S' is just a permutation of S, so in reality $S = S'$[2] Then, their products are equal mod p:

$$\prod S \equiv \prod S' \pmod{p}$$
$$1 \cdot 2 \cdot 3 \cdot \ldots (p-1) \equiv 1a \cdot 2a \cdot 3a \cdot \ldots (p-1)a \pmod{p}$$
$$(p-1)! \equiv a^{p-1}(p-1)! \pmod{p}$$

If p is prime, then its greatest common divisor with any other number is 1. Thus, every number mod p has a multiplicative inverse, so we can multiply both sides by

[2] You can show this by proving that since S' is made up of distinct, non-zero elements (by contradiction), its elements are S's elements. I'm just too lazy to replicate it here.

all of the inverses. Let

$$(p-1)^{-1} = \left(-1^{-1} \cdot 2^{-1} \cdot \ldots (p-1)^{-1}\right)$$

Then, we're done:

$$\begin{aligned}(p-1)! &\equiv a^{p-1}(p-1)! \pmod{p} \\ (p-1)^{-1}(p-1)! &\equiv a^{p-1}(p-1)!(p-1)^{-1} \pmod{p} \\ 1 &\equiv a^{p-1} \pmod{p}\end{aligned}$$

■

4.1.4 Euler's Totient Function

The totient function is defined as the number of integers that are relatively prime to some value:

$$\phi(n) = |\{x : 1 \leq x \leq n, \gcd(x, n) = 1\}|$$

For prime numbers, all numbers are relatively prime, so $\phi(p) = p - 1$. From this comes **Euler's theorem** which is actually a generalization of Fermat's theorem.

Theorem 4.2 (Euler's theorem)**.** *For any N, a where $\gcd(a, N) = 1$ (that is, they are relatively prime), then*

$$a^{\phi(N)} \equiv 1 \pmod{N}$$

4.2 RSA Algorithm

Given this brief foray into number theory, we can finally derive the **RSA** encryption and decryption algorithms.

Let $N = pq$ be the product of two prime numbers. Then, $\phi(N) = (p-1)(q-1)$. Then, Euler's theorem tells us that:

$$a^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

Suppose we take two values d and e such that $de \equiv 1 \pmod{p-1}$. By the definition of modular arithmetic, this means that: $de = 1 + k(p-1)$ (i.e. some multiple k of the modulus plus a remainder of 1 adds up to de). Notice, then, that for some m:

$$\begin{aligned}m^{de} &= m \cdot m^{de-1} \\ &= m \cdot m^{k(p-1)}\end{aligned}$$

$$\begin{aligned} &\equiv m \cdot \left(m^{p-1}\right)^k \pmod{p} \\ &\equiv m \cdot 1^k \pmod{p} && \text{by Fermat's little theorem} \\ \therefore \quad m^{de} &= m \pmod{p} \end{aligned}$$

We're almost there; notice what we've derived: Take a message, m, and raise it to the power of e to "encrypt" it. Then, you can "decrypt" it and get back m by raising it to the power of d.

Unfortunately, you need to reveal p to do this, which reveals $p-1$ and lets someone derive de. We'll hide this by using $N = pq$ and Euler's theorem. The rationale is the same: take d, e such that $de \equiv 1 \pmod{(p-1)(q-1)}$. Then,

$$\begin{aligned} m^{de} &= m \cdot \left(m^{(p-1)(q-1)}\right)^k \\ &\equiv m \cdot \left(m^{(p-1)(q-1)}\right)^k \pmod{N} \\ &\equiv m \cdot 1^k \pmod{N} && \text{by Euler's theorem} \\ \therefore \quad m^{de} &\equiv m \pmod{N} \end{aligned}$$

That's it. That's RSA. *(Say what?)*

A user reveals some information to the world: their **public key**, e and their modulus, N. To send them a message, m, you send $c = m^e \bmod N$. They can find your message via:

$$\begin{aligned} &= c^d \bmod N \\ &= (m^e \bmod N)^d = m^{ed} \bmod N \\ &= m \bmod N \end{aligned}$$

This is secure because you cannot determine $(p-1)(q-1)$ from the revealed N and e without exhaustively enumerating all possibilities (i.e. factoring is hard), so if they are large enough, it's computationally infeasible to do.

4.2.1 Protocol

With the theory out of the way, here's the full protocol.

Receiver Setup To be ready to receive a message:

1. Pick two n-bit random prime numbers, p and q.
2. Then, choose an e that is relatively prime to $(p-1)(q-1)$ (that is, by ensuring that $\gcd(e, (p-1)(q-1)) = 1$. This can be done by enumerating the low primes and finding their GCD.

3. Let $N = pq$ and publish the public key (N, e).
4. Your private key is $d \equiv e^{-1} \pmod{(p-1)(q-1)}$ which we know exists and can be found with the extended Euclidean algorithm.

Sending Given an intended recipient's public key, (N, e), and a message $m \leq N$, simply compute and send $c = m^e \bmod N$. This can be calculated quickly using fast exponentiation (see algorithm 4.1).

Receiving Given a received ciphertext, c, to find the original message simply calculate $c^d \bmod N = m$ (again, use algorithm 4.1).

4.2.2 Limitations

For this to work, the message must be small: $m < N$. This is why asymmetric cryptography is typically only used to exchange a secret *symmetric* key which is then used for all other future messages.

We also need to take care to choose ms such that $\gcd(m, N) = 1$. If this isn't the case, the key identity $m^{ed} \equiv m \pmod{N}$ still holds—albeit this time by the Chinese remainder theorem rather than Euler's theorem—but now there's a fatal flaw. If $\gcd(m, N) \neq 1$, then it's either p or q. If it's p, then $\gcd(m^e, N) = p$ and N can easily be factored (and likewise if it's q).

Similarly, m can't be too small, because then it's possible to have $m^e < N$—the modulus has no effect!

Another problem comes from sending the same message multiple times via different public keys. The Chinese remainder theorem can be used to recover the plaintext from the ciphertexts. Let $e = 3$, then the three ciphertexts are:

$$\begin{aligned} c_1 &\equiv m^3 \pmod{N_1} \\ c_2 &\equiv m^3 \pmod{N_2} \\ c_3 &\equiv m^3 \pmod{N_3} \end{aligned}$$

The CRT states that $c_1 \equiv c_2 \equiv c_3 \pmod{N_1 N_2 N_3}$, but this is just m^3 "unrolled" without the modulus! Thus, $m = \sqrt[3]{m^3}$, and finding m_3 can be done quickly with the extended Euclidean algorithm.

4.3 Generating Primes

The last step we've left uncovered is Step 1: generating two random primes, p and q. This will turn out to be pretty easy: just generate random bitstrings until one of

them is prime. This works out because primes are dense: for an n-bit number, we'll find a prime every n runs on average.

Given this, how do we check for primality quickly?

4.3.1 Primality

Fermat's little theorem gives us a way to check for positive primality: if a randomly-chosen number r is prime, the theorem holds. However, checking all $r-1$ values against the theorem is not ideal: the check takes $\mathcal{O}(rn^2 \log n)$ time.[3]

It will be faster to identify a number as being **composite** (non-prime), instead. Namely, if the theorem *doesn't* hold, we should be able to find any specific z for which $z^{r-1} \not\equiv 1 \pmod{r}$. These are called a **Fermat witnesses**, and every composite number has at least one.

This "at least one" is the **trivial** Fermat witness: the one where $\gcd(z, r) > 1$. Most composite numbers have many **non-***trivial* Fermat witnesses: the ones where $\gcd(z, r) = 1$.

The composites without non-trivial Fermat witnesses called are called **Carmichael numbers** or "pseudoprimes." Thankfully, they are relatively rare compared to normal composite numbers so we can ignore them for our primality test.

> **Property 4.1.** *If a composite number r has at least one non-trivial Fermat witness, then at least half of the values in $\{1, 2, \ldots, r-1\}$ are Fermat witnesses.*

The above property inspires a simple *randomized* algorithm for primality tests that identifies prime numbers to a particular degree of certainty:

1. Choose z randomly: $z \xleftarrow{\$} \{1, 2, \ldots, r-1\}$.
2. Compute: $z^{r-1} \overset{?}{\equiv} 1 \pmod{r}$.
3. If it is, then say that r is prime. Otherwise, r is definitely composite.

Note that if r is prime, this will always confirm that. However, if r is composite (and not a Carmichael number), this algorithm is correct half of the time by the above property. To boost our chance of success and lower false positives (cases where r is composite and the algorithm says it's prime) we choose z many times. With k runs, we have a $1/2^k$ chance of a false positive.

[3] I think... We have $\log n$ rounds of fast modular exponentiation; each one takes n^2 time to mod two n-bit numbers; and we do it r times.

Property 4.2. *Given a prime number p, 1 only has the trivial square roots ± 1 under its modulus. In other words, there is no other value z such that: $z^2 \equiv 1 \pmod{p}$.*

The above property lets us identify Carmichael numbers during the fast exponentiation for $^3/_4^{\text{ths}}$ of the choices of z, which we can use in the same way as before to check primality to a particular degree of certainty.

Linear Programming

> *Linear programming can be viewed as part of a great revolutionary development which has given mankind the ability to state general goals and to lay out a path of detailed decisions to take in order to "best" achieve its goals when faced with practical situations of great complexity.*
>
> — George Dantzig, *inventor of the simplex algorithm*

Linear programming is a general technique for solving optimization problems in which the goal (called the **objective function**) and the constraints can be expressed as linear functions.

5.1 2D Walkthrough

The following is an example of a scenario that can be tackled by linear programming:

> *A company makes two types of products: A and B.*
>
> *Suppose each unit of A makes \$1 of profit, while B makes \$6 of profit. However, the factories operate under some **constraints**:*
>
> - *The machines can only produce up to 300 units of A and 200 units of B per day.*
> - *The workers can only (cumulatively) work 700 hours per day. A takes 1 hour to prepare for assembly by the machine, while B takes 3 hours.*
>
> *How many of each should they make to **maximize** their profits?*

Since our goal is to maximize profits, our variables are the quantities of each product to produce. They should be positive quantities, and shouldn't exceed our labor supply and machine demand constraints.

Objective Function: $\text{profit} = \max x_A + 6x_B$

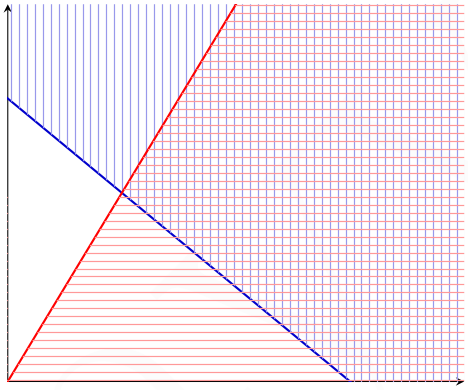
Subject to:

$$0 \leq x_A \leq 300$$
$$0 \leq x_B \leq 200$$
$$x_A + 3x_B \leq 700$$

You may recall from middle school algebra that we solved simple inequalities like:

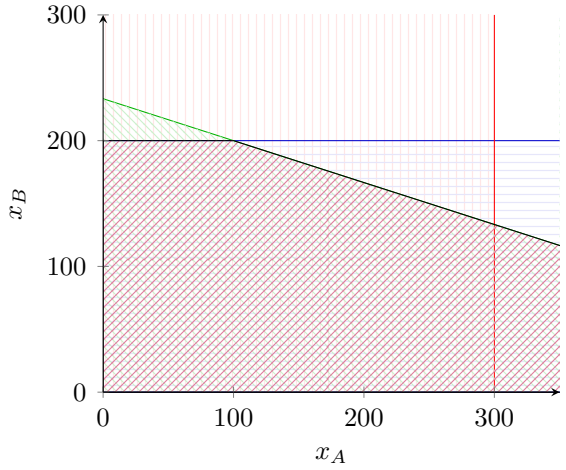
$$y \geq x + 5$$
$$2x - y \geq 0$$

by graphing them and then finding the overlap of valid regions:



This same concept will apply here with linear programming. Each of our 5 constraints represents a **half-plane** within the space of our variables. In this case, each half-plane is just a line in $x_A x_B$-coordinate plane:

$$x_A \leq 300$$
$$x_B \leq 200$$
$$x_A \geq 0$$
$$x_B \geq 0$$
$$x_A + 3x_B \leq 700$$



Clearly, the purple-shaded area is the only *feasible region* in which to search for (x_A, x_B) for our objective function. Now how do we find the optimal point within that region? Our goal is to maximize profit, so: $\max_p [x_A + 6x_B = p]$. This represents a series of lines in the plane, depending on p (see Figure 5.1). Our maximum profit,

then, is the highest line that is still within the region. In this case, that works out to $x_A + 6x_B = 1300$, with $x_A = 100$, $x_B = 200$.

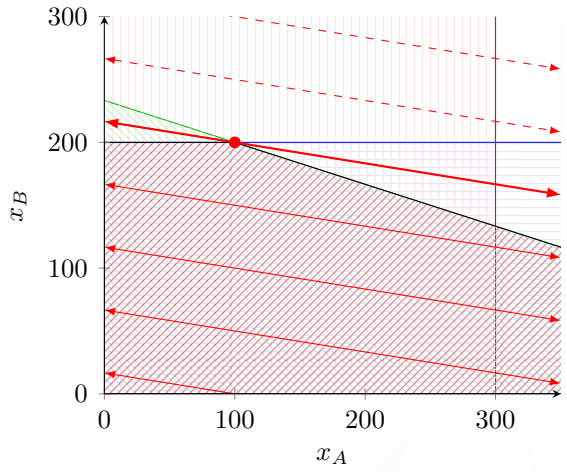


Figure 5.1: Solving the linear program described in the example by finding the highest line that intersects the feasible region.

5.1.1 Key Issues

Even with this simple example, we can identify some key points and important issues.

Integers Critically, the optimum does not have to be an integer point, despite the fact that it was in our example. We are optimizing over the *entire* feasible region; in our problem statement, though, a partial quantity of the A or B products does not make sense. In fact, we have no way to enforce a constraint that our solution be an integer.

This leads to an interesting conundrum: while linear programming is within the P **complexity class** (that is, it's solvable in polynomial time), integer linear programming (denoted ILP) is NP-complete. We'll study this in more detail in chapter 6.

Optimum Location By its very nature, the optimum will lie on one of the vertices of the convex polygon bounded by our constraints. Even if other points are optimal, they will lie along a line (or plane in 3D, or **hyperplane** in n-d) which contains one of the vertices, so that vertex will be just as optimal.

Convexity As an extension from the previous point, the feasible region will be a **convex** polygon: a line segment connected by any two points within the region will still be contained within the region. If a vertex is better than its neighboring vertices in the polygon, it's a global optimum.

5.2 Generalization

The simple example above we can work out by hand and visualize neatly, but how does this scale to 3 variables or an arbitrary n variables? Things get really hard to visualize very quickly. The last key points that we observed above will be essential in our generalization and lead to the simplex algorithm.

From a high level, it's essentially a "locally-greedy" algorithm: we continually choose better and better vertices in the convex polygon until we reach one whose neighbors are all worse, and this is a global optimum.

5.2.1 Standard Form

First, we need to standardize the way that linear programs are represented before we can devise a generic algorithm for solving them.

Given n variables, $x_1, x_2, \ldots, x_n$, the goal is to optimize the objective function given the constraints described by an $m \times n$ matrix $\mathbf{A}$ and a m-length vector $\mathbf{b}$ as follows:

Objective Function: $\max[c_1x_1 + c_2x_2 + \ldots c_nx_n]$ or simply $\max_{\mathbf{c}} \mathbf{c}^{\mathrm{T}}\mathbf{x}$

Subject to:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n &\leq b_1 \qquad x_1 \geq 0 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n &\leq b_2 \qquad x_2 \geq 0 \\ &\vdots \qquad \text{and} \qquad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n &\leq b_m \qquad x_n \geq 0 \end{aligned}$$

Or, simply $\mathbf{Ax} \leq \mathbf{b}$ and $\mathbf{x} \geq 0$.

We can transform most sets of problems to follow this form. However, problems that define strict inequalities are **not allowed** in linear programming: notice that $\max_x[x < 100]$, for example, is ill-defined since we can have infinite precision.

5.2.2 Example: Max-Flow as Linear Programming

The max-flow problem can be expressed as a linear program. Recall its formulations:

Max-Flow Definition:

Input: A directed graph, $G = (V, E)$ with capacities $c_e > 0, \forall e \in E$. Source and sink vertices $s, t \in V$.

Output: An optimal flow, f^*, such that any f_e cannot be any bigger while still maintaining a valid flow.

To express this as a linear program, our objective function wants to maximize the

flow coming out of the source vertex while ensuring each edge has a valid flow:

Objective Function: $\max \sum_{(s,v)\in E} f_{sv}$

Subject to: For every $e \in E$: $0 \leq f_e \leq c_e$.

For every $v \in V \setminus \{s,t\}$: $\sum_{(w,v)\in E} f_{wv} = \sum_{(v,z)\in E} f_{vz}$

Every term here is a linear function, so linear programming applies.

5.2.3 Algorithm Overview

With our generic problem formulation out of the way, we can work towards an algorithm for finding the global optimum. Remember, the goal is to find the vertex of the convex polygon that satisfies the constraints that is maximal relative to its neighbors.

With n variables, we're working in n dimensions, and we have $n+m$ constraints. The feasible region is the intersection of $n+m$ half-spaces and it forms a convex polyhedron. Then, the vertices of said polyhedron are the points that satisfy some n constraints with equality that also satisfy the other m constraints with $\leq$.

Bounds In the worst case, there's a *huge* number of vertices in the polyhedron: from $n+m$ constraints, we choose n; $\mathcal{O}\left(\binom{n+m}{n}\right)$ grows exponentially. Further, there are up to nm neighbors for each vertex.

Choices There are several algorithms that solve linear programs in polynomial time, including ellipsoid algorithms and interior point methods. The **simplex algorithm** has a worst-case bound of exponential time, but it's still widely used because it guarantees a global optimum and works incredibly fast on huge linear programs.

5.2.4 Simplex Algorithm

The basic high level idea of the simplex algorithm is very simple.

- We begin at $\mathbf{x} = 0$, which satisfies our baseline non-negativity constraints. If it doesn't satisfy *any* of our *other* constraints, we know that it's an infeasible LP.
- Then, we look for the a neighboring vertex with a (strictly) higher objective value. When found, we "move" there and repeat this process.
- If at any point in time *none* of the neighbors are strictly better, we've found the global optimum.

The variations of the simplex algorithm lie in the choice of vertex. We could choose the first one we see (to avoid enumerating them all), we could choose randomly (to

avoid getting stuck on multiple runs), and we could even just choose the best (though then we pay the price of enumeration). There are many different heuristics for this algorithm, each with their own tradeoffs.

5.2.5 Invalid LPs

Some linear programs will be *infeasible*, having no feasible region; this is dependent only on the constraints. Others will be *unbounded*, where the maximum objective function increases indefinitely; this is dependent only on the objective function.

Feasibility Determining feasibility can be done by introducing a new unrestricted variable z that we introduce into our constraints:

$$\begin{aligned} \text{from:} \quad & a_1x_1 + \ldots + a_nx_n \leq b \\ \text{to:} \quad & a_1x_1 + \ldots + a_nx_n + z \leq b \end{aligned}$$

Satisfying this is trivial with a small-enough z (so $z \rightsquigarrow -\infty$). However, if we could find a $z \geq 0$ that satisfies the new constraint, then the original LP is feasible. We do this by actually running maximization on a *new* LP:

$$\begin{aligned} &\textbf{Objective Function:} && \max[\mathbf{z}] \\ &\textbf{Subject to:} && \mathbf{Ax} + \mathbf{z} \leq \mathbf{b} \\ & && \mathbf{x} \geq 0 \end{aligned}$$

Checking $\mathbf{z} \geq 0$ tests feasibility *and* gives us a starting point for the simplex algorithm on the original LP.

Unbounded Determining whether or not a linear program is unbounded depends on its dual LP (discussed next). Specifically, we'll see that Corollary 5.3.2 tells us that if an LP's dual linear program is infeasible (which we can determine easily as we just showed), then the original is either unbounded or infeasible.

5.3 Duality

Suppose someone gives us a value for the objective function and claims that it's the global optimum for its LP. Can we verify this?

Consider a simple fact: the maximum objective function value is the same for any linear combination of the constraints. For example, let's work through this LP for which the known optimum is $\mathbf{x}^* = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} = \begin{bmatrix} 200 & 200 & 100 \end{bmatrix}$:

Objective Function:	$\max[x_1 + 6x_2 + 10x_3]$	
	$x_1 \leq 300$	①
	$x_2 \leq 200$	②
Subject to:	$x_1 + 3x_2 + 2x_3 \leq 1000$	③
	$x_3 + 3x_3 \leq 500$	④
	$x_1, x_2, x_3 \geq 0$	

Now let's let $\mathbf{y} = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix} = \begin{bmatrix} 0 & \frac{1}{3} & 1 & \frac{8}{3} \end{bmatrix}$ and consider the linear combination of the constraints, not worrying about how $\mathbf{y}$ was determined just yet.

$$(y_1 \cdot ①) + (y_2 \cdot ②) + (y_3 \cdot ③) + (y_4 \cdot ④) \leq 300y_1 + 200y_2 + 1000y_3 + 500y_4$$

$$\begin{aligned} x_1y_1 + x_2y_2 + x_1y_3 + 3x_2y_3 + 2x_3y_3 + x_2y_4 + 3x_3y_4 &\leq \\ x_1(y_1 + y_3) + x_2(y_2 + 3y_3 + y_4) + x_3(2y_3 + 3y_4) &\leq \end{aligned}$$

$$\begin{aligned} x_1 + 6x_2 + 10x_3 &\leq \frac{200}{3} + 1000 + \frac{8 \cdot 500}{3} \\ &\leq 2400 \end{aligned}$$

after plugging in $\mathbf{y}$

This is exactly the upper bound found by the optimum shown above: $x_1 + 6x_2 + 10x_3$ comes out to 2400 when applying $\mathbf{x}^*$.

How would we find this $\mathbf{y}$ knowing the optimum? Well, the goal was to find a $\mathbf{y}$ such that the right hand side of the equation is *at least* as big as the optimum. This is the same thing as saying we want the left-hand size to be at least as big as the objective function. The left side:

$$x_1(y_1 + y_3) + x_2(y_2 + 3y_3 + y_4) + x_3(2y_3 + 3y_4) \geq x_1 + 6x_2 + 10x_3$$

translates to a new set of constraints:

$$\begin{aligned} y_1 + y_3 &\geq 1 \\ y_2 + 3y_3 + y_4 &\geq 6 \\ 2y_3 + 3y_4 &\geq 10 \end{aligned}$$

Any $\mathbf{y}$ satisfying these constraints will be an upper bound (though not necessarily the *smallest* upper bound) on the objective function. Thus, *minimizing* the right-hand size will give us the smallest upper bound:

$$\min_{\mathbf{y}} [300y_1 + 200y_2 + 1000y_3 + 500y_4]$$

This is LP-**duality**: the maximization of the original objective function under directly corresponds to a minimization of a new objective function based on the right-hand size of the constraints.

More abstractly, when given a *primal* LP with n variables and m constraints in the canonical Standard Form:

$$\begin{aligned} &\textbf{Objective Function:} && \max \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ &\textbf{Subject to:} && \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & && \mathbf{x} \geq \mathbf{0} \end{aligned}$$

The corresponding *dual* LP results in a minimization on m variables and n constraints:

$$\begin{aligned} &\textbf{Objective Function:} && \min \mathbf{b}^{\mathrm{T}}\mathbf{y} \\ &\textbf{Subject to:} && \mathbf{A}^{\mathrm{T}}\mathbf{y} \geq \mathbf{c} \\ & && \mathbf{y} \geq \mathbf{0} \end{aligned}$$

where $\mathbf{y}$ describes the linear combination of the original constraints.

EXAMPLE 5.1: **Finding the Dual LP**

Given the following linear program:

$$\begin{aligned} &\textbf{Objective Function:} && \max[5x_1 - 7x_2 + 2x_3] \\ &\textbf{Subject to:} && x_1 + x_2 - 4x_3 \leq 1 \\ & && 2x_1 - x_2 \geq 3 x_1, x_2, x_3 \geq 0 \end{aligned}$$

What's the dual LP?

We can solve this without thinking much by forming all of the matrices and vectors necessary and simply following the pattern described above.

First, we need to transform the first constraint to follow standard form. This is trivial algebra: multiplying an inequality by a negative flips the sign. This gives us:

$$-1 \times (2x_1 - x_2) \geq -1 \times 3 \iff -2x_1 + x_2 \leq -3$$

Now, just fill in the blanks:

$$\mathbf{c} = \begin{bmatrix} 5 & -7 & 2 \end{bmatrix}$$
$$\mathbf{A} = \begin{bmatrix} 1 & 1 & -4 \\ -2 & 1 & 0 \end{bmatrix}$$
$$\mathbf{b} = \begin{bmatrix} 1 & -3 \end{bmatrix}$$

And transform them into our new LP:

$$\min \mathbf{b}^{\mathrm{T}}\mathbf{y} \implies \min[y_1 - 3y_2]$$
$$\mathbf{A}^{\mathrm{T}}\mathbf{y} \geq \mathbf{c} \implies \begin{cases} y_1 - 2y_2 & \geq 5 \\ y_1 + y_2 & \geq -7 \\ -4y_1 & \geq 2 \end{cases}$$
$$\mathbf{y} \geq \mathbf{0}$$

An immediate consequence of the way we formed the dual LP is the **weak duality** theorem.

Theorem 5.1 (Weak Duality)**.** *Any feasible* $\mathbf{x}$ *for a primal LP is upper bounded by any feasible* $\mathbf{y}$ *for the corresponding dual LP:*

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} \leq \mathbf{b}^{\mathrm{T}}\mathbf{y}$$

We used this theorem when we were proving the Max-Flow = Min-Cut Theorem. It leads to a number of useful corollaries.

Corollary 5.3.1. *If there exists a feasible* $\mathbf{x}$ *and feasible* $\mathbf{y}$ *that satisfy Theorem 5.1 more strictly:*

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{b}^{\mathrm{T}}\mathbf{y}$$

then these points are optimal for their respective LPs.

Corollary 5.3.2. *If the primal LP is* ***unbounded****, then the dual LP is* ***infeasible****. Similarly, if the dual LP is unbounded, then its corresponding primal LP is infeasible.*

This latter corollary is one-way: it's still possible to have situations in which both the primal and the dual LPs are infeasible. As we've already alluded, this corollary is critical in identifying Invalid LPs. Specifically, we determine whether or not an LP is unbounded by checking the feasability of its dual.

Now, do the optima described in 5.3.1 always exist? This is answered by the **strong duality** theorem.

Theorem 5.2 (Strong Duality)**.** *A primal LP has an optimal* $\mathbf{x}^*$ *if and* ***only*** *if its dual LP has an optimal* $\mathbf{y}^*$*. These satisfy Corollary 5.3.2:*

$$\mathbf{c}^{\mathrm{T}}\mathbf{x} = \mathbf{b}^{\mathrm{T}}\mathbf{y}$$

Secondarily, these can exist if and ***only*** *if their corresponding LP is both feasible and bounded.*

5.4 Max SAT

Recall tackling SATisfiability problems in our discussion of graphs in chapter 3.

Satisfiability:

Input: A Boolean formula f in conjunctive normal form with n variables and m clauses.

Output: An assignment satisfying f or an indication that f is not satisfiable.

We established that while 2-SAT can be solved in polynomial time with connected components, SAT is NP-complete. The Max-SAT problem is an alternative construction that does not require *full* satisfiability; instead, we try to *maximize* the number of satisfied clusters.

Max-SAT:

Input: A Boolean formula f in conjunctive normal form with n variables and m clauses.

Output: An assignment that maximizes the number of satisfied clauses in f.

This version of the problem is NP-hard: though it's at least as hard as the full SAT problem, we no longer have an easy way to check if the solution truly is the maximum. Instead, we will provide an *approximate* solution to the problem. Specifically, if m^* is the solution to Max-SAT, we're going to find an assignment that guarantees some portion of m^*.

5.4.1 Simple Scheme

For the first pass, we're going to guarantee an assignment that satisfies $\frac{1}{2}m^*$ clauses. Consider f as we've defined it with n variables $\{x_1, \ldots, x_n\}$ and m clauses $\{c_1, \ldots, c_m\}$.

The approach is incredibly simple: we assign each x_i to true or false randomly with equal probability for all n variables. Suppose W denotes the number of satisfied clauses, and it's expressed as the sum of a bunch of w_js that denote whether or not the corresponding c_j clause has been satisfied. Then, the expectation (or average) value of W is, by definition:

$$
\begin{aligned}
E[W] &= \sum_{\ell=0}^{m} \ell \cdot \Pr[W = \ell] && \text{definition of expectation} \\
&= E\left[\sum_{j=1}^{m} w_j\right] = \sum_{j=1}^{m} E[w_j] && \text{linearity of expectation} \\
&= \sum_{j=1}^{m} \Pr[w_j = 1] && \text{since } \ell = 0 \text{ when } w_j = 0 \\
&= \sum_{j=1}^{m} (1 - 2^{-k}) && \text{where } k \text{ is the number of variables in the clause, since only a single } x_i \text{ needs to be set to true} \\
&\geq \sum_{j=1}^{m} \frac{1}{2} \geq \frac{m}{2} && \text{since } k \geq 1
\end{aligned}
$$

Thus, just guessing a random value is a randomized algorithm that satisfies at least $1/2$ of the maximum satisfiable clauses.

Notice that this lower bound is not dependent on k; this is for simplicity of calculation—k varies for each clause. What if we considered Max-Ek-SAT, in which every clause is *exactly* of size k? Then we simply have a $(1 - 2^{-k})$ approximation.

It was proved that for the case of Max-3-SAT, it's NP-hard to do any better than a $7/8$ approximation (that is, when doing our simple algorithm of random assignment).

5.5 Integer Linear Programming

The structure for an integer linear program is the same as with a normal linear program, except $\mathbf{x}$, the point maximizing the objective function, is restricted to be only integral:

$$
\begin{aligned}
&\textbf{Objective Function:} && \max[\mathbf{c}^{\mathrm{T}}\mathbf{x}] \\
&\textbf{Subject to:} && \mathbf{A}\mathbf{x} \leq \mathbf{b} \\
& && \mathbf{x} \geq 0 \\
& && \boxed{\mathbf{x} \in \mathbb{Z}^n}
\end{aligned}
$$

5.5.1 ILP is NP-Hard

Many problems can be reduced to integer linear programming. While LPs are in the P complexity class, ILP is NP-hard; we'll prove this by reducing the Max-SAT from earlier to an ILP. Since Max-SAT is NP-hard, ILP must be also.

Take the input f from Max-SAT (the CNF Boolean function) and craft an ILP as follows:

- For each variable $x_i \in f$, we set $y_i = x_i$ in the ILP.
- For each clause c_j, add a z_j indicating satisfiability.

  How do we express this? Well, given some $c_j = (x_a \vee x_b \vee x_c)$, we want $z_j = 0$ if all of the x_is are 0, right? So just sum them up! In general,

  $$z_j \leq y_a + y_b + \dots$$

  However, this doesn't work for clauses with complements, like $c_j = (\overline{x_a} \vee x_b \vee x_c)$, since $z_j = 0$ despite $y_a = 1$. To alleviate this, we look at the complement as well:

  $$z_j \leq (1 - y_a) + (1 - y_b) + \dots$$

Given this translation from f to the ILP, we now want to maximize the number of satisfied clauses:

Objective Function: $\max \left[\sum_{j=1}^{m} z_j \right]$

Subject to:

For all $i = 1, \dots, n$: $0 \leq y_i \leq 1$.

For all $j = 1, \dots, m$: $0 \leq z_j \leq 1$.

Then, both the positive and complementary versions of y_i should be summed: $\sum_{i \in C_j^+} y_i + \sum_{i \in C_j^-} (1 - y_i) \geq z_j$

Finally, $y_i, z_j \in \mathbb{Z}$.

Given this reduction and given the fact that Max-SAT is known to be NP-hard, ILP must also be NP-hard.

We can relax the integer restriction to find some $\mathbf{y}^*$ and $\mathbf{z}^*$ that essentially define an upper bound on m^*, the maximum number of satisfiable clauses in f. By rounding said solutions, we can find a feasible point in the ILP that is not far off from the true optimal. In fact, we can prove that this results in a $\left(1 - \frac{1}{e}\right) m^*$ approximation (where e is the exponential, $e \approx 2.718\dots$) of Max-SAT.

Udacity '18 — the proof is omitted for brevity

This technique is a common approach in approximating something is NP-hard: reduce it to an ILP, approximate it with an LP, then run a randomized approximation algorithm that (hopefully) results in a close approximation.

The approximate LP-based approach described above scales quite well with k when applied to the exactly-k SAT problem:

k	**Simple**	**LP-Based**
1	$1/2$	1
2	$3/4$	$3/4$
3	$7/8$	$1 - (2/3)^3 \approx 0.704$
	$\dots$	
k	$1 - 2^{-k}$	$1 - \left(1 - \frac{1}{k}\right)^k$

Notice that we can always get at least a $3/4^{\text{ths}}$ approximation. This leads to a simple algorithm where we simply take the better of the simple vs. LP-based scheme to always get $\frac{3}{4}m^*$.

Computational Complexity (or, P $\stackrel{?}{=}$ NP)

> *If* P = NP*, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in 'creative leaps,' no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss; everyone who could recognize a good investment strategy would be Warren Buffett.*
>
> — Scott Anderson, *complexity researcher*

The purpose of finding a problem's computational complexity is to determine whether or not a problem is **intractable**, meaning it's unlikely to be solvable efficiently. We define "efficiently" as "within polynomial time."

We will define **NP** as being the complexity class of all search problems. Roughly, a search problem is one where we can efficiently verify solutions. Given an input problem and a proposed solution, it should be easy (i.e. doable in polynomial time) to verify whether or not that solution is correct.

This leads to the **P** complexity class, which includes only search problems that are *solvable* in polynomial time. Thus, P $\subset$ NP.

The notion of P $\stackrel{?}{=}$ NP is a way of asking whether or not solving a problem is equivalent to verifying a solution. Intuitively, it seems much more difficult to generate a proof or solution for a problem than to verify its correctness.

Terminology Obviously, P stands for *polynomial time*. Paradoxically, NP does *not* stand for non-polynomial time; instead NP specifies *non-deterministic polynomial*

idk man

time. This is the set of problems that can be solved in polynomial time on a non-deterministic machine—one that is allowed to guess at each step. The rough idea is that there is *some* choice of guesses that can lead to a polynomial solution, so if you could perform all of the guesses simultaneously, you could find the correct answer.

6.1 Search Problems

Formally, problems within NP are defined in the following way:

Search problem:

Form: Given an instance I, we can find a solution S for I if one exists, or indicate that there is no solution.

Requirement: To be a search problem, given an instance I and *any* solution S (not necessarily one given by an algorithm), it must be verifiable that S is a solution to I in time polynomial in $|I|$.

The way to show the above requirement is to provide a verifier algorithm $V(I, S)$ that provably fulfills it.

6.1.1 Example: SAT

Recall the structure of the SAT problem.

SAT:

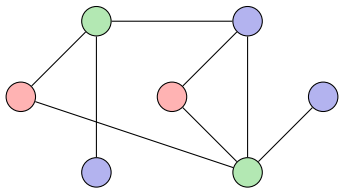
Input: A Boolean formula f in CNF with n variables and m clauses.

Output: A satisfying assignment—if one exists—or "no" otherwise.

A proposed solution is obviously trivial to verify in polynomial time: simply assign each x_i accordingly and run through each clause once to check if it's satisfied. In the worst case, this takes $\mathcal{O}(mn)$ time, since each of the m clauses needs to be checked once, and each clause has (at most) $2n$ literals. This means SAT $\in$ NP. ■

6.1.2 Example: k-Coloring Problem

The k-color problem is well known in computer science. Given a graph and a palette with k colors, you must find a way to label each node with one of the colors such that no neighboring nodes share a color. For example, the following is a valid 3-coloring of the graph:



Finding the coloring is hard, but verifying it is easy: given a graph G and a coloring, we simply need to check that each edge has different colors on each endpoint. Thus, k-coloring is $\in$ NP. ■

6.1.3 Examples: MSTs

Recall the problem of finding the minimum spanning tree:

MST:

Input: A graph $G = (V, E)$ with positive edge lengths.

Output: A tree T with minimum weight.

This problem is both within NP *and* P: this is simply because *finding* an MST takes polynomial time (recall Kruskal's algorithm). We can verify that a given T is a tree with depth-first search (no back edges), then check if T has the same weight as the output of Kruskal's (or Prim's) algorithm. This means MST $\in$ NP and MST $\in$ P.

6.1.4 Example: Knapsack

Recall the knapsack problem from dynamic programming.

Knapsack:

Input: A set of n objects with values and weights:

$$V = \{v_1, v_2, \ldots, v_n\}$$
$$W = \{w_1, w_2, \ldots, w_n\}$$

A total capacity B.

Output: A subset S that both:

(a) maximizes the total value: $\max_{S \in B} \sum_{i \in S} v_i$

(b) while fitting in the knapsack: $\sum_{i \in S} w_i \leq B$

Given an instance of the problem and a solution, checking whether or not it fits within the capacity is easy, taking $\mathcal{O}(n)$ time. What about checking its optimality? The only way to do this is by actually solving the problem and comparing the total value.

Recall, though, that unlike in the MST case above, the running time for knapsack was $\mathcal{O}(nB)$. This is not polynomial, since $|I| = (n, \log_2 B)$.

Our conclusion is thus that generic knapsack *not known* to be in NP: there *may* exist a better algorithm to find or verify the solution. Since we can't know for sure that there *isn't*, we can't definitively say if knapsack $\stackrel{?}{\in}$ NP.

Variant: Knapsack Search

We can consider a variant of knapsack in which we drop the maximization requirement and instead add a new input parameter: a *goal*, g. Then, the algorithm tries to find a subset of values that meet the goal.

Knapsack Search:

Input: A set of n objects with values and weights:
$$V = \{v_1, v_2, \ldots, v_n\}$$
$$W = \{w_1, w_2, \ldots, w_n\}$$
A total capacity B.
A goal, g.

Output: A subset S that both:
(a) meets the goal: $\sum_{i \in S} v_i \geq g$
(b) while fitting in the knapsack: $\sum_{i \in S} w_i \leq B$

This is clearly in NP: checking that it meets the goal involves summing up the values, $\mathcal{O}(n)$ time.

6.2 Differentiating Complexities

Complexity Zoo, YouTube '14

We do not know if P $\neq$ NP, but the debate comes down to a single distillable point:

> *Does being able to recognize correct answers quickly mean that there's also a quick way to find them?*

We can assume P $\neq$ NP in order to differentiate between problems within NP. The intractable problems within NP are called **NP-complete** problems. This means that if any NP-complete problem can be solved in polynomial time, then *all* problems in NP can be solved in polynomial time.

In essence, any problem that is NP-complete can be "reduced" (or, more accurately, "transformed") into any other. This means that if we can find a polynomial-time

solution for SAT, we can use it as a black box to solve *any* NP problem.

Can we show this transformation? Namely, can we show that literally any problem within NP (including problems in P) can be reduced to solving SAT? This involves a bidirectional proof:

- Suppose $f(G, k)$ is a function that transforms the input from the k-coloring problem to an SAT input.
- Also suppose that $h(S)$ is a function that transforms the output from SAT into an output for the k-coloring problem.
- Then, we need to show that if S is a solution to SAT, then $h(S)$ is a solution to k-color.
- Similarly, we need to show that there are no solutions to the SAT that do *not* map to a solution to k-color. In other words, if SAT says "no," then k-color also correctly outputs "no".

We need to do this for *every* problem within NP, which seems insurmountable. However, if we have *any* known NP-complete problem (like SAT), if we can reduce *from* said problem to the *new* problem (like, for example, the independent set problem), then the new problem is also NP-complete.

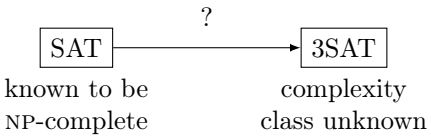
6.3 Reductions

We'll enumerate a number of reductions to prove various algorithms as NP-complete. Critically, we will take the theorem that SAT is NP-complete for granted.

Cook '71
Levin '71
Karp '72

6.3.1 3SAT to SAT

The difference between 3SAT and SAT is that in the former, each clause is limited to having at most 3 literals. To show that 3SAT is NP-complete, we first show that 3SAT $\in$ NP; then, we find a reduction from SAT to 3SAT:



Within NP Verifying the solution for 3SAT is just as easy as for SAT, in fact only taking $\mathcal{O}(m)$ time since $n \leq 3$.

Reduction Because of the similarity in the problems, forming the reduction is fairly straightforward. A given long SAT clause c can be transformed into a series of 3-literal

clauses by introducing $k-3$ new auxiliary variables that form $k-2$ new clauses:

$$\begin{aligned}(x_1 \vee x_2 \vee \ldots \vee x_k) \implies &(x_i \vee x_{i+1} \vee y_1) \wedge (\overline{y_1} \vee x_{i+3} \vee y_2) \wedge \\ &(\overline{y_2} \vee x_{i+4} \vee y_3) \wedge \ldots \wedge \\ &(\overline{y_{k-4}} \vee x_{i+4} \vee y_{k-3}) \wedge (\overline{y_{k-3}} \vee x_{k-1} \vee x_k)\end{aligned}$$

For example, the following 5-SAT is transformed as follows:

$$\begin{aligned}C &= (\overline{x_2} \vee x_3 \vee \overline{x_1} \vee \overline{x_4} \vee x_5) \\ C' &= (\overline{x_2} \vee x_3 \vee y) \wedge (\overline{y} \vee \overline{x_1} \vee \overline{z}) \wedge (\overline{z} \vee \overline{x_4} \vee x_5) \\ C &\iff C'\end{aligned}$$

The bidirectional proof of correctness can be done by contradiction.

6.3.2 Independent Sets

The subset of vertices $S \subset V$ is an **independent set** if no edges are contained in S; in other words, $\forall x, y \in S, (x, y) :\notin E$. Finding small independent sets is easy; finding large, or the larg*est* independent set is difficult.

Maximum Independent Set:

Input: An undirected graph, $G = (V, E)$.

Output: The independent set S of maximum size.

This problem is not known to be $\in$ NP: the only way to check the validity of a solution to the problem is by solving it, and no polynomial-time algorithm for solving it is known.

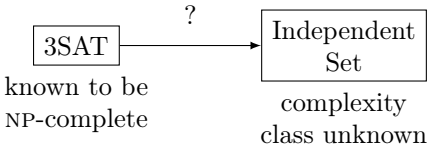
Much like we adapted knapsack to be NP-complete (see subsection 6.1.4), we can adapt the independent set problem to be NP-complete by adding a goal input g. Then, the solution's validity becomes trivial to check.

Independent Set Search:

Input: An undirected graph, $G = (V, E)$.
A goal g.

Output: The independent set S with size $|S| \geq g$, if one exists.

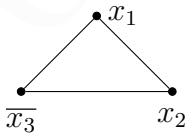
Now, we'll prove that the independent set problem is NP-complete with a reduction from 3SAT.



Construction

Consider again the 3SAT scenario: there's an input f with variables $x_1, \ldots, x_n$ and clauses $c_1, \ldots, c_m$ where each clause has at most 3 literals. We'll define our graph G with the goal matching the number of clauses: $g = m$. Then, for each each clause c_i in the graph, we'll create $|c_i|$ vertices (the number of literals).

Clause Edges We encode a particular clause as a subgraph by fully connecting its literals. So, for example, the clause $c = (x_1 \vee \overline{x_3} \vee x_2)$ is encoded as shown on the right.

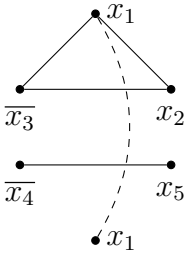


By construction, the resulting independent set S has at most 1 vertex per clause, and since $g = m$, the solution has exactly 1 vertex per clause.

Variable Edges In order to prevent the SAT assignment from creating contradictions, we need to also add edges between any variable and its complement. For example, given the following f:

$$f(x_{1..5}) = (x_1 \vee \overline{x_3} \vee x_2) \wedge (\overline{x_4} \vee x_5) \wedge (\overline{x_1})$$

we would craft the graph on the right, in which the solid edges are the clause edges we just defined and the dashed edge is the new variable edge.



Correctness

Given this formulation, we need to prove correctness in both directions. Namely, we need to show that if f has a satisfying assignment, then G has an independent set whose size meets the goal, and vice-versa.

Forward Direction We know that we start with an assignment that satisfies f. For each clause c, then, at least one vertex in its corresponding subgraph is set to true; add exactly one of the vertices to the independent set. Since there are m clauses, we get $|S| = m = g$ and fulfill our goal. ■

Notice that if the chosen vertex has a variable edge, then by definition its corresponding edge cannot be chosen (this would cause a contradiction in f's assignment). Similarly, we did not include any clause edges by definition.

Backward Direction This time, we start with an independent set of G of at least size g. Since each clause's subgraph is fully connected, the independent set must contain no more than 1 vertex per clause. And since there are at least g vertices, each clause must have a vertex. By setting the vertices' literals to be true within f, we satisfy it. ■

We've now shown the equivalence between 3SAT and IS and hence proven that the independent set problem is NP-complete.

Maximum Independent Set

Remember that we discovered that the Max-IS problem is not $\in$ NP. Furthermore, it's easy to reduce the IS problem to the Max-IS problem: if we find the maximum m^*, we just check if it's bigger than the goal, $m^* \stackrel{?}{\geq} g$. This means that we can reduce any problem in NP to Max-IS (either by reducing through IS or directly), but Max-IS itself is outside of NP.

To denote that Max-IS is at least as hard as anything in NP, but we can't say whether Max-IS is NP-complete itself, we say that it is **NP-hard**. In summary, by difficulty we say that P $\leq$ NP-complete $\leq$ NP-hard.

6.3.3 Cliques

A **clique** (pronounced "klEEk" like "freak") is a fully-connected subgraph. Specifically, for a graph $G = (V, E)$, a subset $S \subseteq V$ is a clique if $\forall x, y \in S : (x, y) \in E$.

Much like with independent sets, the difficulty lies in finding *big* cliques, since things like a single vertex are trivially cliques.

Clique Search:

Input: A graph $G = (V, E)$ and a goal, g.

Output: A subset of vertices, $S \subseteq V$ where S is a clique whose size meets the goal, so $|S| \geq g$.

Checking that S is fully-connected takes $\mathcal{O}(n^2)$ time, and checking that $|S| \geq g$ takes $\mathcal{O}(n)$ time, so clique problem is NP. It's also NP-complete: we can prove that this by a reduction from the independent set problem.

The key idea behind the reduction is that these problems are basically polar opposites: in a clique, there's a full set of edges within S; in an independent set, there are no edges within S.

The "opposite" of a graph $G = (V, E)$ is a graph such that all of its edges are edges that weren't in G. In other words, $\overline{G} = (V, \overline{E})$ where $\overline{E} = \{(x, y) : (x, y) \notin E$. By definition, then, S is a clique in $\overline{G}$ if and **only** if S is an independent set in G.

Given an input $G = (V, E)$ and a goal g for the IS problem, we form $\overline{G}$ as above and leave g as inputs to the clique problem. If S is a solution for the clique problem, return it as the solution for the IS problem.

6.3.4 Vertex Cover

A subset of vertices is a **vertex cover** if it covers every edge. Specifically, for $G = (V, E)$, $S \subseteq V$ is a vertex cover if $\forall (u, v) \in E : v \in S \vee u \in S$.

6.4 Undecidability

A problem that is **undecidable** is one that is impossible to construct an algorithm for. One such problem is Alan Turing's **halting problem**. Turing '36

Halting Problem:

Input: A program P (in any language) with an input I.

Output: "Yes" if $P(I)$ ever terminates, and "no" otherwise.

The proof is essentially done by contradiction: if such a program exists, can it determine when *it* would terminate? Suppose Terminator(P, I) is an algorithm that solves the halting problem. Now, let's define a new program JohnConnor(J) that performs a very simple sequence of operations: if Terminator(J, J) returns "yes," loop again and terminate otherwise.

There are two cases, then:

- If Terminator(J, J) returns "yes," that means $J(J)$ terminates. In this case, JohnConnor$(\cdot)$ will continue to execute.
- If it doesn't, then we know that $J(J)$ never terminates, so JohnConnor$(\cdot)$ stops running.

So what happens when we call JohnConnor$(JohnConnor)$?

- If Terminator$(JohnConnor, JohnConnor)$ returns "yes," that means JohnConnor$(JohnConnor)$ terminates. But then the very function we called—JohnConnor$(JohnConnor)$—will continue to execute *by definition*. This contradicts the output of Terminator().
- If Terminator$(JohnConnor, JohnConnor)$ returns "no," then that means JohnConnor$(JohnConnor)$ never terminates, but then it immediately does...

In both cases, we end up with a paradoxical contradiction. Therefore, the only valid explanation is that the function Terminator() cannot exist. The halting problem is undecidable. ■

Index of Terms