Optimizations and Heuristics

Vidit Jain, A Kishore Kumar

December 1, 2021

Contents

Computational Problems

To study & analyze algorithms, we must have a solid foundational theory for what algorithms are. We often define algorithms as a series of steps that must be followed to solve a problem. This raises a few basic questions that we must answer concretely.

- What are **computational** problems?
- Once *a* solution is obtained, how do we know that it is indeed correct?
- Say we have 2 solutions `a` & `b`. How can we compare these 2 solutions? On what basis can we say one solution is better than the other?
- Once we can do the above, how can we find the lower bound, i.e., the most optimal solution to a problem? and more importantly, can we prove that it is indeed optimal?

What are computational problems?

There are many kinds of problems in the world. Math problems, world peace & hunger problems, problems arising from eating a bat, etc. to name a few. We wish to concern ourselves with problems of a particular class, **computational** problems. However, there are many difficulties associated with defining such a class. Consider the following challenges:

- **There may be infinite ways of posing the same problem**

  Consider the following problems.

  1. *What is the smallest prime number?*
  2. *What is the smallest even positive integer?*
  3. *What is the GCD of the set of even positive integers?*

  Notice that the output (solution) for all the above problems is two. From a computational standpoint, the above problems are all the same. But there are infinite ways to pose the same problem.

- **How do we pose a question without solving it?**

  Consider the age-old problem of sorting. The problem is usually phrased as follows,

  > *Given a sequence of* `n` *integers, among all possible permutations of this sequence, output such a sequence that it is in ascending order. That is,* $a_i < a_{i+1} \quad \forall \quad 1 \leq i < n$

  Notice that in the above statement, we provide the solution to the question itself. Phrased differently, the problem is essentially telling us to iterate over all possible $n!$ permutations and pick the permutation such that the sequence is in ascending order. While granted, this isn't a *good* solution, it is a solution. We must come up with a way to pose problems that we do not have a solution to yet. Or maybe even problems for which there does *not* exist any solution.

Defining a computational problem

When defining a computational problem, we make the following assumption about the world we live in.

The input is digitized. We live in a noisy environment. Whenever there is noise, if 2 symbols are closer than some threshold, we say they are in the same equivalence class and are one and the same. This ensures that the total number of symbols we must deal with becomes finite in a finite space. This ensures that we're able to digitize the input and output.

Further, assume that the output has multiple but a finite number of bits. We can then model each bit as a separate problem. This enables us to model all problems with finite output as decision problems. A decision problem is simply a problem with **1-bit** output. "*Is this true or false?*"

This allows us to pose problems as membership queries in *"languages."*

We can reduce the question *"X is my input and I am looking for the output Y"* to "*Does my input X belong to the set of all inputs which give output one?*"

Languages Each decision problem is characterized by a subset of the set of all possible inputs. (,i.e., subset of say, {0, 1}*)

$L = \{x \mid s \in \{0, 1\}^*\}$

For example: Consider the problem of checking if a sequence is sorted or not.

Let us encode our strings as numbers in binary separated by some terminator which splits each number in the sequence. Our question now reduces to, *"Given a string encoded this form, does it belong to the language L_{sorted}?"* The string that encodes the sequence {1, 2, 3} would belong to this set. But the string which encodes the sequence {3, 2, 5} would not. Our encoding must be able to represent all different inputs in a unique manner. Notice that this has allowed us to reduce the problem to a simple decision problem of querying membership in our language L_{sorted}

This is essentially how all problems in complexity theory are formalized.

This formalization allows us to deal with most of the challenges aforementioned. Multiple ways to pose the same question no longer matter as the problems are characterized by the language. If the set of all possible inputs for which the output is 1 is the same for 2 different problems, then they are one and the same. Further, we can now pose problems without providing a solution as it is possible for us to define sets without having to enumerate them.

Now that we have formulated what a computational problem is, a natural follow-up question is asking, *"Are there computational problems that we **cannot** solve?"*

The answer is depressing, but sadly, we must acknowledge it.

Notice that if we are able to prove that there are **uncountable many** computational problems and only **countably many** computer programs. Then this would imply that there must exist uncountable many problems for which, **no computational program solution exists**.

Countable sets An infinite set is countable if there is a bijection $f : N \to S$ from natural numbers to S **Uncountable sets** An infinite set is countable if it is not possible to construct a bijection $f : N \to S$ from natural numbers to S. A common proof method is cantor's diagonalization which first assumes that it is possible to construct such a bijection and then proves that for every such bijection, we can always create a new element in the set that was not mapped before. Thus disproving that any such bijection can be created.

Proving that the set of all programs is countable

Now, notice that every single program that we write, must be encoded to some subset in the set of all finite-length bit strings, i.e., some subset of $\{0, 1\}^*$.

All the axioms, rules, formulas, etc. followed by a program can be encoded via a scheme such as Gödel Numbering to map to a unique natural number. Once such an encoding is established, because each rule and symbol is mapped to a unique natural number, we can make the argument that this set is countable.

Proving that the set of all computational problems is uncountably infinite

Let us prove that $P(\{0, 1\}^*)$, i.e., the power set of all finite-length bit strings is uncountable. Notice that every problem is modeled as a decision problem. And every decision problem is characterized by a set. Or it's "language." Therefore, every possible subset of the set of all finite-length binary strings actually represents a problem. Each subset is a unique language and each of them characterizes unique problems.

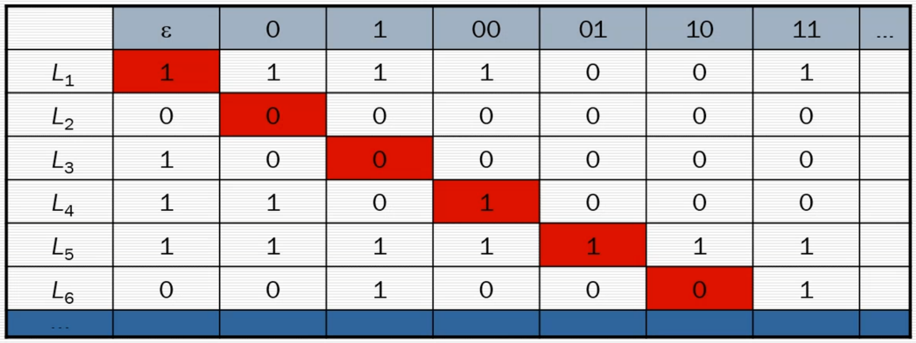
Therefore, counting the total number of computational problems essentially reduces to calculating the cardinality of the power set $P(\{0, 1\}^*)$

Consider the following function $f : \{0, 1\}^* \to \{0, 1\}$ which maps the set of all finite-length binary strings to a subset. Let us pick some subset $S \subset \{0, 1\}^*$. Then the function is defined as follows:

f(x)=

$$\begin{cases} 1 \; \forall \; x \in S \\ 0 \; \forall \; x \notin S \end{cases}$$

Now let us calculate $f(x)$ for every such language and write it in the form of a table



	ε	0	1	00	01	10	11	...
L_1	1	1	1	1	0	0	1	
L_2	0	0	0	0	0	0	0	
L_3	1	0	0	0	0	0	0	
L_4	1	1	0	1	0	0	0	
L_5	1	1	1	1	1	1	1	
L_6	0	0	1	0	0	0	1	
...								

Let us assume that we have enumerated an infinite number of such languages. Now we will use diagonalization to prove that there will always exist some language L_x that does not belong to our set.

We construct L_x as follows. We move along the diagonal and flip the value of L_i for each element i of the set.

$L_x(\epsilon) = 0, L_x(0) = 1, L_x(1) = 1, L_x(00) = 0, L_x(01) = 0 \ \ldots$

We notice that such a language L_x does not belong to the set as it differs from each L_i belonging to our bijection at the i^{th} element. This means we have successfully proved the existence of a language that does not belong to our bijection. No matter how many times we repeat the process of finding such a language and adding it to the bijection, we will always be able to prove the existence of such a new language that does not belong to the bijection. Hence we have proved that the power set $P(\{0,1\}^*)$ is indeed, uncountably infinite.

This implies that the cardinality of the set of all computational problems is greater than the set of all possible computer programs. This in turn implies that there are uncountably many computational problems that we cannot find computational solutions for.

However, it **is** true that there is often inherently *"algorithmic"* about what we as humans pose as problems. So most of the time, we are usually able to construct algorithms to solve our problems. That said, there are definitely many examples of problems which we would love to know the solution for but are sadly, unsolvable. Some examples are the "Halting problem", "Program equivalence program", etc.

Introduction to Complexity Theory

In most algorithms courses, students are taught a plethora of algorithms that are capable of solving many interesting problems. It often tends to internally suggest to the student that most problems have solutions. Solutions that are feasible to compute on their machines should they need to. On the contrary, most problems are unsolvable and even fewer are computable in any feasible amount of time.

Computation complexity theory is a field of study where we attempt to classify "computational" problems according to their resource usage and relate these classes to one another. We begin by defining a few to classify algorithms based on running time.

1. P
2. EXP
3. NP
4. R

P or PTIME

This is a fundamental complexity class in the field of complexity theory. We define P as the set of **all** decision problems that can be solved by a **deterministic** Turing machine in polynomial time.

A more formal definition is given below: A language L is said to be in P $\iff$ there exists a **deterministic** Turing machine M such that: 1. M runs for polynomial time on **all** inputs 2. $\forall l \in L$, M outputs 1 3. $\forall l \in L$, M outputs 0

When we talk about computational problems, we like problems in P. These problems are feasible for computers to compute in a reasonable amount of time.

EXP or EXPTIME

This is the class of **all** decision problems that can be solved by a **deterministic** Turing machine in exponential time. Similar to how we gave a formal definition for P, it is easy to see that we can modify the same formal definition to fit EXP as well.

R or RTIME

The R here stands for "recursive." Back when complexity theory was being developed, there was a different idea of what the word 'recursive' meant. But in essence, R is simply the set of all decision problems that can be solved by a deterministic Turing machine in some finite amount of time.

It might seem as though all problems are solvable by a Turing machine in some finite time and hence unnecessary to have a class dedicated to it. But this is not true.

Undecidable problems

An undecidable problem is a decision problem for which it has been proven that it is impossible to develop an algorithm that always leads to a valid yes-or-no answer. To prove that there exist undecidable problems, it suffices to provide even just one example of an undecidable problem.

The Halting Problem

One of the most famous examples of undecidable problems is the halting problem, put forth by Alan Turing himself. Using this, Turing proved that there do indeed exist undecidable problems. But this isn't just the only reason why the halting problem is "special."

The halting problem poses the following question: *"Given the description of an arbitrary program and a finite input, decide whether the program finishes running or will run forever."*

In fact, if the halting problem were decidable, we would be able to know a LOT more than what we do today. Proving conjectures would be a LOT easier and we might have made a lot of progress in many fields.

Solve Goldbach's conjecture?

Consider Goldbach's conjecture. It states that *every even whole number greater than 2 is the sum of two prime numbers.*

Using computers, we have tested the conjecture for a large range of numbers. This conjecture is "probably" true, but till today, we have **no** proof for this statement. Simply checking for large ranges is simply not enough. Finding even just one counter-example, even if this counterexample is 10s of digits long is enough to prove the conjecture **false**.

Let's say we constructed a Turing machine M that executes the below algorithm (given in pseudocode).

[] iterate from i : 0 -> \infty: iterate from j : 0 -> i: if j is prime and (i-j) is prime: move to next even i if none of its summation were both prime: output i halt # We have disproved Goldbach's conjecture!

This definition of our Turing machine is capable of disproving Goldbach's conjecture. But the question is, how long do we let it run for? If the number is not small, it might take years to find this number. Maybe millions of years. We do not know. And even worse, if the conjecture is indeed true, then this machine will **never** halt. It will keep running forever.

However, what if the halting problem was decidable?

What if, we could construct another such Turing machine M_1 this time which solves the halting problem? We can feed it M as input, and let M_1 solve the halting problem.

If M_1 outputs "halt" then there **must** be some input for which Goldbach's conjecture fails. We have disproved it.

If M_1 outputs “run forever” then Goldbach’s conjecture **must** be true. It is no longer a conjecture, we have managed to prove it!

Being able to solve the halting problem would help us solve so many such conjectures. Take the twin primes conjecture, for example, we would be able to solve it. We would be so much more powerful and armed in terms of the knowledge available to us. However, sadly, Alan Turing proved that the halting problem is undecidable. And the proof is quite fascinating to describe

The proof

We will prove that the halting problem is undecidable using contradiction. Therefore, we begin by assuming that there exists some Turing machine M that is capable of solving the Halting problem.

More formally, there exists some deterministic Turing machine Mwhich accepts some other Turing machine A and A’s input X as input and outputs 1 or “Halt” if A will half on that input and 0 or “Run forever” if A will not halt on that input.

Now, let’s construct another Turing machine “Vader” which does something quite interesting. Given some Turing machine A and its input X, Vader first runs Mon the input. If M returns “halt”, Vader will run forever. And if M returns “run forever”, Vader will halt.

This is still fine, but the masterstroke that Turing came up with was to give Vader, itself as input!

In the above explanation, we make $A = Vader$ and $X = Vader$. For Vader to work, it will first run Mon this input. For simplicity, we will call the input program Vader as iVader. There can only be two possible outputs,

1. $**M$ returns “Halt”**

   This means that M thinks that iVader will halt when run on itself. *However*, when M returns “halt”, Vader will run forever. Remember that Vader is given itself as input. The input iVader and the program Vader are identical. M predicts that iVader will halt but we know that Vader will run forever. We have a contradiction.

2. $**M$ returns “Run forever”**

   Again, we have ourselves a contradiction. Just like before, M thinks that iVader will run forever, but we know that Vader will halt. The emphasis here is that iVader and Vader here are the same Turing machines that run on the same input.

Therefore, neither of the cases can be true. The fact that we have a contradiction here arises from the fact that our assumption is wrong. There can exist no such Turing machine M which can solve the Halting problem.

A fun exercise

Given a set of problems, let us see if we're able to match the below problems to the fastest complexity class they're solvable in.

1. GCD
2. $n \times n$ Chess
3. Computing the product of two d degree polynomials
4. Tetris
5. All pairs shortest paths on a graph

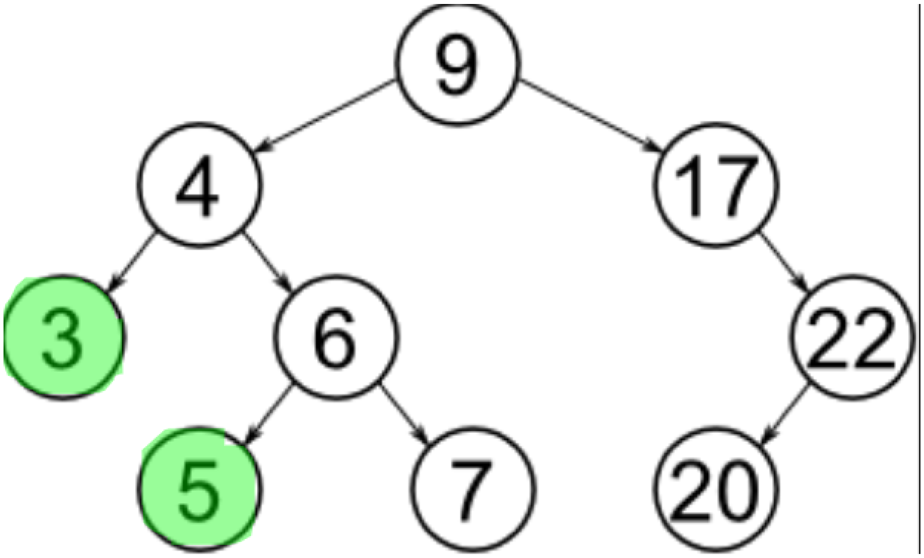
If you've tried matching the above problems yourself once, then go ahead and read further below.

GCD is solvable in polynomial time using Euclid's algorithm. $n \times n$ Chess can be solved in exponential time by playing out every possible sequence. It is possible to prove that this problem cannot be solved in polynomial time. Computing the product of d degree polynomials can be done in polytime using FFT. All pairs shortest paths can also be solved using Floyd Warshall in polytime. However, Tetris has a known solution which lies in the EXP class. We do **not** know if Tetris lies in P or not.

NP or NP-TIME

There are a couple of different definitions used to define the class NP.

One of these definitions says, NP is the set of problems that can be solved in polynomial time by a **nondeterministic** Turing machine. Notice that the keyword here is **nondeterministic.** What this essentially means that at every "step" in the computation, the machine *always* picks the right path. Let's say a Turing machine had states similar to the below picture. A non-deterministic machine would accept any input string that has **at least one accepting run** in its model. It is "lucky" in the sense that it is always capable of picking the right choice and moving to the right state which guarantees ending at a **YES** result as long as such a run exists in its model.



The second definition for NP calls it the set of decision problems for which the problem instances, where the answer is "yes", have proofs verifiable in polynomial time by a deterministic Turing machine. To understand this, we must understand verification vs decision.

Verification vs Decision

We covered what it means to solve what a decision problem is, verification is on the other hand is something you can send along with a solution. In most intuitive terms, let's say someone claims that they are very good at the game of Tetris and can win the game for some specified input. Here we consider a modified version of Tetris where all the next pieces are known in advance. How does this person **prove** to you that they can indeed win the game? By playing it out of course! It might be very difficult to figure out the strategy to win, but given the proof (the sequence of moves), implementing the rules of Tetris and playing it out to check if the person is correct can be done easily.

Essentially, to be in NP, our machine can take an arbitrary amount of time to come up with proof for its solution for all possible inputs, but this proof must be *verifiable* in polynomial time.

We'll attempt to explain further via means of an example. Consider the clique problem.

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is an undirected graph with a k-clique}\}$$

How would a *verifier* verify this answer? Let's say the input to the verifier is given in the form $\langle\langle G, k\rangle, c\rangle$ where c is the answer to our problem defined by G and k.

1. First, check if the answer c contains exactly k unique nodes $\in G$ or not. If no, the answer can be trivially rejected. This can be done in $O(V)$ time.
2. Next, check if there exists an edge between **every** pair of nodes in c. This is done in $O(V + E)$ time. If no, reject the answer.
3. If both the above checks passed, accept the answer!

Hence we can say that the clique problem is in NP because we've demonstrated that it is indeed possible to write a verifier that can check the "correctness" of an answer. In the field of complexity theory, we call such 'solution paths' or 'proofs' or 'witnesses' a **certificate** of computation.

NP-Complete

For a problem p to be $\in NP - Complete$ it must fit 2 criteria.

1. p must be $\in NP$
2. *Every* problem $\in NP$ must be *reducible* to p

We cover reductions in depth later, but essentially, if we can come up with a polynomial-time algorithm(s) to 'reduce' the inputs and outputs $\langle I, O \rangle$ given to some machine s to new inputs/outputs $\langle I', O' \rangle$ such that when applied to another machine t, $O' = O$. If this can be done, we say that we have reduced the problem solved by s to t.

P vs NP

This is one of the most famous unsolved questions in computer science. I mean, seriously, the clay math institute offers a reward of a **million** dollars to the first person that is able to solve this problem. https://www.claymath.org/millennium-problems/p-vs-np-problem

Why? What's so special about this problem and what even *is* the problem?

Let's begin by defining the problem. The problem asks, is $P = NP$? That is, is the set of *all* the problems in NP the same as the set of all the problems in P? A more intuitive way to phrase this question would be asking, "Are all problems that can be *verified* in polynomial time, also be *solved* in polynomial time?"

But why is this one of the most famous unsolved problems in computer science? What are the implications of such a result? Why is this even a question, do we even have *any* reason to believe that P *might* equal NP?

Here are a few, *interesting* answers to these questions.

1. If P *did* equal NP, it would mean that simply being able to *check* if a solution is correct, would be **no harder** than solving the problem itself. Optimization problems like transport routing, production of goods, circuit design, etc. are **all** NP problems. We would be able to get optimal answers to these solutions *much* faster than we are able to today. The economy could be made so much more efficient. Protein folding is an NP problem.

If we could make protein folding a problem in P then we would be able to make huge breakthroughs in biology. We would be able to cure cancer! One of my favorite quotes describing the implications of $P = NP$ is from an MIT researcher,

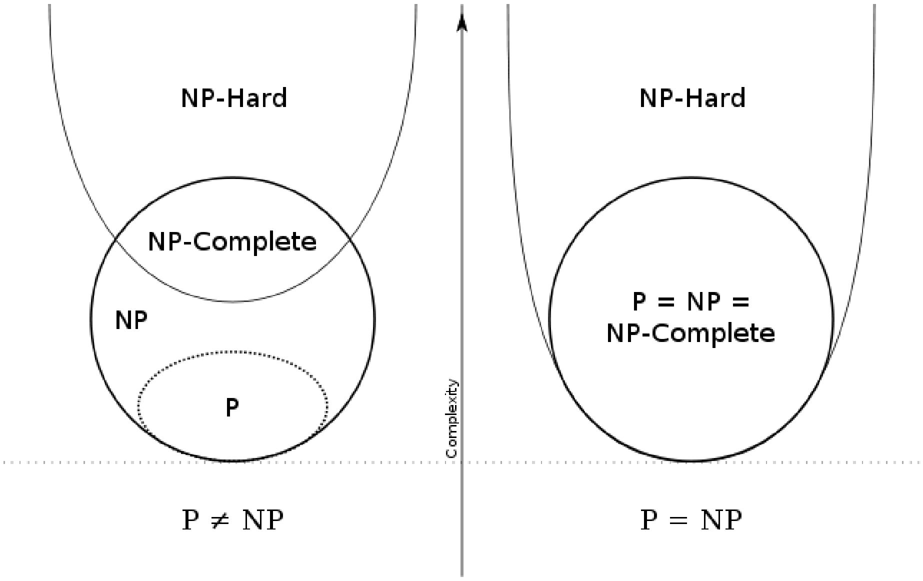
> *"If P=NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in "creative leaps," no fundamental gap between solving a problem and recognizing the solution once it's found. Everyone who could appreciate a symphony would be Mozart; 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 Aaronson

One small downside is that RSA is also a NP problem. If $P = NP$, all known security measures encryptions would breakdown and none of our passwords would be safe :)

2. But the truth is, most computer science researchers do **not** believe that $P = NP$. Consider the first definition of NP problems that we gave. We are essentially relying on non-determinism in our Turing machine. We are relying on the fact that the Turing machine is somehow able to "magically" or "luckily" *always* pick the right path of traversal. Luck or magic is not something we can model in a deterministic Turing machine. However, despite all this, no one has been able to prove $P \neq NP$.

3. Finally, problems in NP have indeed been shown to be in P. Consider sorting an array by going through all its different permutations, such an algorithm would take $O(n!n)$ time. It is not in P. However, after we cleverly came up with a better algorithm such as bubble sort or merge sort, we managed to reduce this problem to be in P by coming up with an $O(nlogn)$ algorithm for solving it. Similarly, problems we once thought to be in NP have been shown to be in P after someone managed to come up with a clever algorithm to solve the problem faster. But just because some problems we thought to be in NP were later found to be in P , does not mean that the two classes are equal. In fact, that the question $P = NP$? is really asking is if $P = NP - Complete$. Recall that $NP - Complete$ problems are the hardest problems in NP. Every single problem that belongs in NP, including the $NP - Complete$ problems are reducible to an $NP - Complete$ problem. This means that if we could somehow reduce even **one** problem belonging to the $NP - Complete$ class to P, we would be able to prove $P = NP$. So far, problems in NP were found to be reducible to P, but never an $NP - Complete$ problem. As mentioned on the Clay institute website, *"However, this apparent difficulty may only reflect the lack of ingenuity of your programmer."* Someday, someone just might be able to come up with a radical new algorithm to reduce one of the $NP - Complete$ problems to P. There is a possibility, even if highly unlikely.

This is a view of the complexity classes as we know it, depending on the result of the P vs NP problem.



[Courtesy: https://brilliant.org/wiki/complexity-classes/]

NP - Hard

These, on the surface, appear to be problems that are at least as hard as $NP-Complete$ problems. It's worth noting that $NP-Hard$ problems don't have to be in NP.

The precise definition is that a problem X is $NP-Hard$ if there is an $NP-Complete$ problem Y that is polynomial-time reducible to X .

However, because each NP-Complete problem in polynomial time may be reduced to any other NP-complete problem, all NP-complete problems can be reduced to an NP-hard problem in polynomial time.

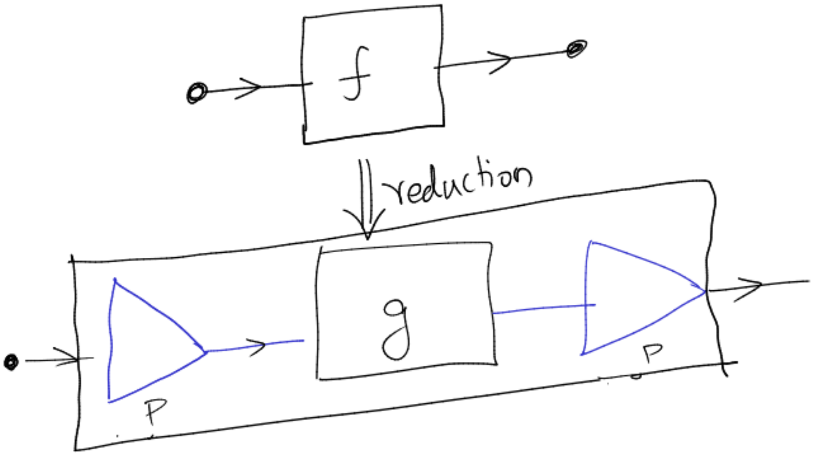
Over the course of our book, we will attempt to introduce how we humans have come up with cool ideas and algorithms to "solve" NP problems. For problems in P, we know solutions that run in a feasible amount of time. However, there are many important problems, for example, traffic routing, that do not fall in P. Just because we cannot find optimal solutions fast does not mean that we can ignore these problems. We might still be able to come up with answers that are *very close* to the optimal answer via heuristics and other methods that help us run and solve these problems faster!

Reductions

In computability theory and computational complexity theory, a reduction is an algorithm for transforming one problem into another problem. A sufficiently efficient reduction from one problem to another may be used to show that the second problem is at least as difficult as the first.

Intuitively, what does this mean?

Let's say we have two problems f and g. Let's suppose that problem g has a known solution. Then the following is a reduction from $f \to g$.



The **"Reduction"** is basically finding those two blue boxes, which convert the input and output from that of problem f to equivalent input for problem g. Now we can simply compute the solution for problem g and then use the reverse of our reduction algorithm to transform the output to that required by f.

If we can find two such blue triangles which can transform the input & output in such a way then we can effectively say that problem f has been reduced to solving problem g. This is because solving g implies being able to solve f.

What's more interesting is if these blue triangles are **polynomial-time** algorithms. If we can find poly-time algorithms which can perform this transformation of the input and output, then we have an *efficient* reduction.

We have effectively managed to solve f using the solution of g, along with some (hopefully efficient) pre and post-processing.

Motivation

Why study optimization algorithms?

Optimization problems are often among the most difficult problems in computer science. That is, most optimization problems are in the class of $NP-Complete$ problems. This makes them almost impossible to optimally solve for large input.

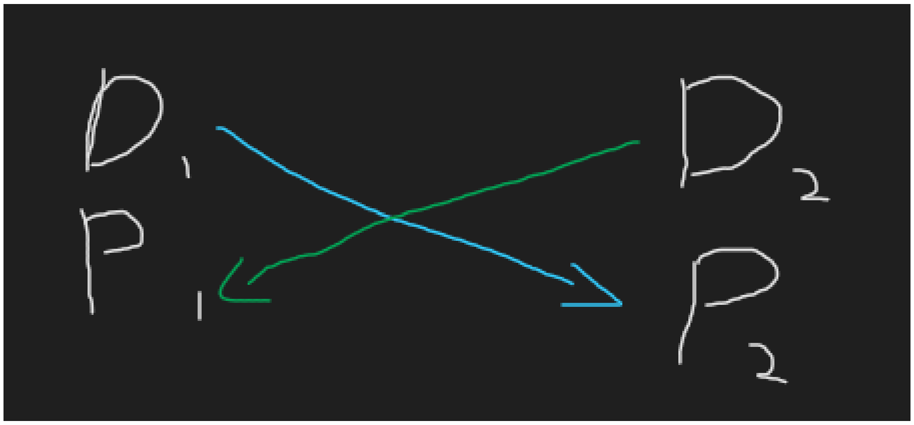
Yet, these problems show up in a **lot** of places. Some of the most important problems are optimization problems. Consider the tasks of supply chains & export infrastructure, sport scheduling logistics, electrical power systems, disaster management, kidney exchange, traffic routing, etc. These are all $NP-Complete$ problems. There exists no simple solution to solve these problems. But just because it's not easy doesn't mean we can give up on these problems. Solving them is clearly very important to us as a society.

Some examples

Kidney Exchanges

In India, around 1.5 lakh people die of kidney failure every year. Humans can survive on one kidney. When one person has kidney failure, usually a close relative such as their parent or spouse are willing to donate them a kidney. However, it is not always the case that the donated kidney is compatible with the victim. It is imperative that the kidneys are compatible, else the procedure will fail.

However, consider the situation where donor 1 (D_1) is compatible with patient 2 (P_2) and vice versa. In this scenario we can agree on a mutual exchange between $D_1 - P_2$ and $D_2 - P_1$.



If we model this as a graph problem we reduce our problem to finding the largest disjoint cycle cover of the graph. This is a $NP - Complete$ problem.

Export

Consider export at harbors. This is an important part of trade and economy. Scheduling the exports so that the harbor is left empty for the shortest period, ensuring that we have the most optimal scheduling of trucks so that they're always on the move transporting goods, dispatching goods from the harbor optimally using the limited resources present, etc. are also optimization problems.

By now, we know that these problems are important. But what we need are solutions.

Greedy

Greedy Algorithms

As discussed previously, greedy algorithms are an amazing choice when we can prove that they do indeed give the optimal solution to some given problem. This signifies the importance of being able to prove the optimality of greedy solutions. In general, if the following two conditions hold, we can certainly say that a greedy strategy will give us the globally optimal answer.

Locally optimum choice

Given some problems, can we focus on its local state and solve for a solution that produces the most locally optimal solution at that state? In other words, we should be able to take *one step* towards the optimum solution.

Optimum substructure property

Once this step is taken, even after the change in state after taking that step, are we able to restate the problem such that the new problem is the same as the original problem, albeit for a smaller input?

Notice that if the answer to the above two questions is **yes**, then it is possible to prove that repeatedly taking the locally optimal choice will indeed give us the optimal solution. This is easily proven via induction.

Take the optimal step at any given step i, now restate the problem as a smaller version of the original problem, and again take the locally optimal step at $i + 1$. We can inductively repeat till this is the final state where we can again take the optimal choice. Since we can solve each subproblem independently simply by taking the best choice at each step, the solution **must** be optimal.

Activity Selection

Consider the famous activity selection problem. The problem is as follows,

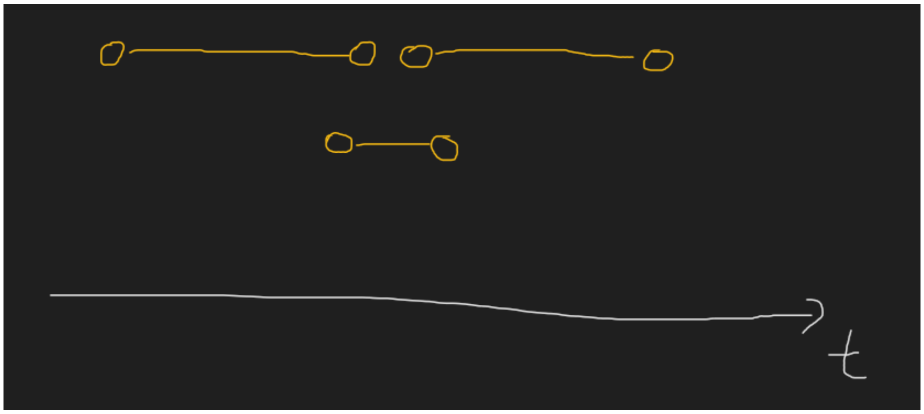
Given a set of activities $S = \{a_1, a_2, \dots, a_n\}$, where activity a_i takes time $[s_i, f_i)$ to finish. Find the maximum number of activities that can be picked such that *there are zero overlaps, i.e., pick the subset of the maximum size where all activities are disjoint.*

The naïve solution would be to brute force over all $n!$ different permutations in linear time to find the optimal answer. This is obviously far too slow to be of much use to us. So, how can we do better? Would a **greedy** solution work?

Greedy #1

Sort the intervals by duration $|f_i - s_i|$ and greedily pick the shortest ones

Does this satisfy our two properties? The answer is... no. Notice that by picking the shortest interval activity, we cannot restate the problem for a smaller input the same way. We do not have optimum substructure. Consider the below case.

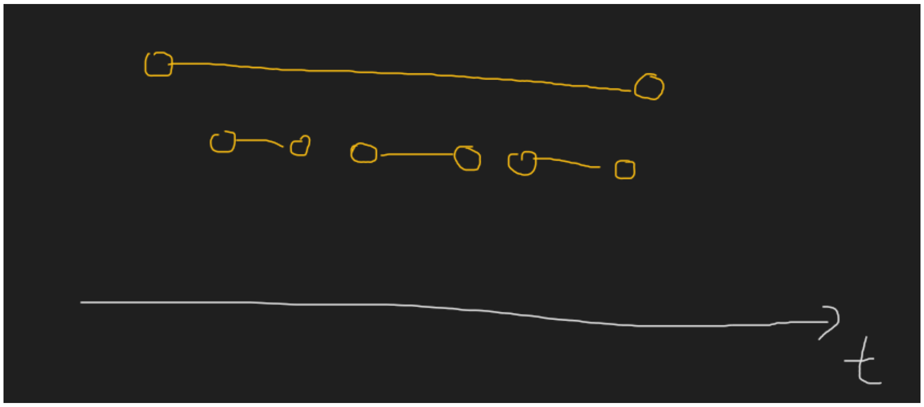


Greedily we would pick the middle activity, but this removes two activities for the next step. This problem has no optimum substructure. The optimal solution would be to pick both the large intervals.

Greedy #2

Greedily pick the activities that start the earliest

That approach follows neither property. Consider this case,



We are neither picking a locally optimum choice nor maintaining an optimum substructure. The greedy solution gives 1 whereas the answer is clearly, 3.

Greedy #3

Greedily pick the activities that end the earliest

Does this approach satisfy both criteria? The answer is… yes.

Let us pick the activity that ends the earliest. If this is not part of the optimal solution and the activity it overlaps with is part of the optimal solution, notice that because the activity we picked ends earlier, our activity cannot have any other overlap. Both contribute +1 to the solution and hence our activity is locally optimal. Further, since we have picked the earliest ending activity (which is optimal) we can cross off overlaps and restate the problem for smaller input. This approach maintains both properties! It **must** be right.

A more formal proof

Let us suppose that we know the answer. Let the answer be A. Let us sort A by finish time such that $\forall a_{i<n} \in A,\ f_i < f_{i+1}$

Now, let our optimal choice activity be x. By virtue of our greedy choice, we know that

$f_x \leq f_{a_i} \forall a_i \in A$

Consider f_{a_0}. If $x = a_0$, we are done. But if $x \neq a_0$, notice that $f_x \leq f_{a_0}$. This means that x cannot overlap with any more activities in the set A than a_0. And the set A is disjoint by definition. Our solution can be written as

$$B = A - \{x\} \cup \{a_0\}$$

Notice that x cannot overlap with any element in A. This is because they're the first choice to be picked, there is no overlap on the left. And $f_x \leq f_{a_0}$ implies there is no overlap on the right and both provide a +1 to the final answer. Hence x **must** be an optimal choice.

This solution is **much better** than our $O(n!)$ solution and can find the optimal answer in just $O(nlogn)$. The $nlogn$ comes from the sorting requirement.

Huffman Encoding

The compression problem

Let's think about how computers store text. A lot of the text on machines is stored in ASCII. ASCII is a character encoding used by our computers to represent the alphabet, punctuations, numbers, escape sequence characters, etc. Each and every ASCII character takes up *exactly* one byte or 8 bits. The encoding chart can be found here

Oftentimes, computers need to communicate with one another, and sending large volumes of text is not an uncommon occurrence. Communication over networks, however, have their own cost and speed disadvantages that make sending smaller chunks of data a *very* favorable option. This is one of the times when ranking an algorithm by **space** is preferred over ranking algorithms by **time**. As our metric for comparison between algorithms changes, so does our problem statement.

"What is the most optimal way to losslessly compress data such that it takes up minimum space?"

Notice that unlike video or audio compression, ASCII text compression must be **lossless**. If we lose *any* data, we have also lost the character. This means we can no longer figure out what the original ASCII character was. These requirements give us a few basic requirements that our algorithm **must** meet.

Prefix-free property

The idea of compression is to reduce the size of the data being compressed. But ASCII requires 8 bytes. This means that we must try to encode data in fewer than 8 bytes based on the frequency of occurrence. This will allow us to dedicate fewer bits for more commonly occurring characters and more bits for characters that occur almost never, thus helping us compress our data. However, this implies that we need some form of **variable-length** encoding for our characters. One variable-length encoding that might work is the binary system.

However, notice that the following assignment will fail.

$$Space \rightarrow 0e \rightarrow 1t \rightarrow 00\ldots$$

When we encounter the encoding 00 in the compressed data, we no longer know whether it is "two spaces" or one "t" character. We have lost information in our attempt to compress data. This implies that our algorithm **must** fulfill the prefix-free property. That is while reading the compressed data, based on the prefix, we must be able to **uniquely** identify the character that it is representing. If this is not possible then we will not have an injective mapping and data will be lost.

A little detour to information theory

Back in the world of information theory, Shannon laid out the 4 key axioms regarding information.

1. **Information $I(x)$ and probability $P(x)$ are inversely related to each other**

   Consider the following thought experiment.

   1. The kings of the Middle East are rich
   2. The man washing cars here is a rich man

   The second sentence conveys a lot more information than the first. The first statement is highly probable and hence does not convey as much information as the second.

2. $**I(x) \geq 0**$

   Observing an event never causes a loss in information

3. $P(x) = 1 \implies I(x) = 0$

   If an event is 100% certain to occur then there is no information to be gained from it

4. $P(x \cap y) = P(x).P(y) \implies I(x \cap y) = I(x) + I(y)$

   Two independent events if observed separately, give information equal to the sum of observing each one individually

It can be proven that the only set of functions that satisfy the above criteria are

$$I(x) = log_b(\frac{1}{P(x)}) = -log_b P(x)$$

He then went on to define a term called Information Entropy. It is a quantity that aims to model how "unpredictable" a distribution is. It is defined as the weighted average of the self-information of each event.

$$H(x) = \sum_{i=1}^{n} P(x_i).I(x_i) = \sum_{i=1}^{n} -P(x_i).log_2 P(x_i)$$

An intuitive way to think of it is as follows. If an event that has a high self-information value has a high frequency, then this will increase the entropy. This makes sense as we are essentially saying that there is some event that is hard to predict which occurs frequently. Vice versa, if low self-information (something predictable) has a high frequency then the entropy of the distribution is lesser.

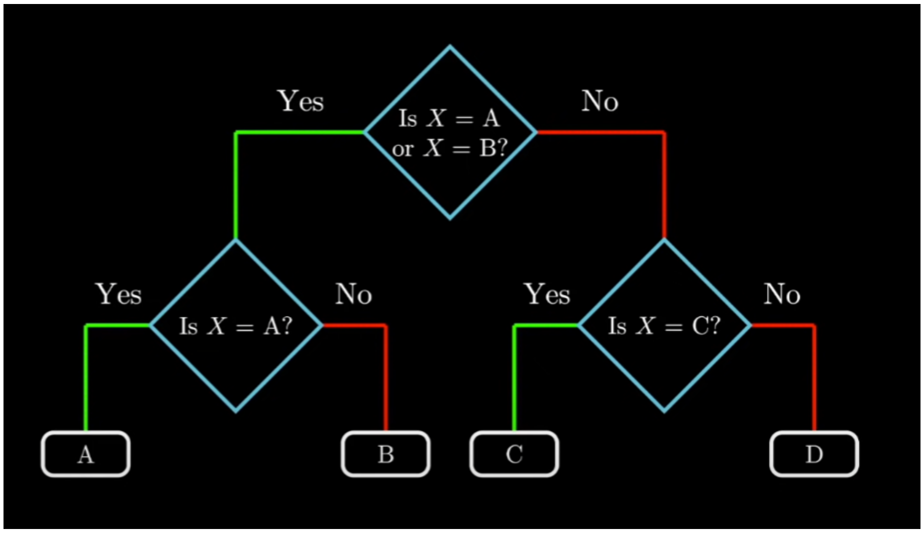
An interesting fact to note behind the coining of the term “Entropy” in information theory. Shannon initially planned on calling it “uncertainty.” But after an encounter with John von Neumann who told him “No one knows what entropy really is, so in a debate, you’ll always have the advantage.” he changed the term to “Entropy”

Back to algorithms!

Let’s say we have some encoding E for our data D. We can measure the compression of our data by the “Average expected length per symbol.” This quantity is essentially just the weighted average of the lengths of each symbol in D in our encoding E. Let’s call the average length per symbol L.

Shannon discovered that the fundamental lower bound on L is given as $L \geq H(x)$. No matter what we do, we cannot compress the data to an average length lower than the information entropy of each data point occurrence.

Consider the case where the letters `A`, `B`, `C`, `D` occur in our data with a frequency of 0.25 each. We can divide the decoding process into a simple decision tree as follows,



Representing the encoding as binary trees

In the above image, if we replace every **left** branch with 1 and every **right** branch with 0, we get a very interesting encoding. We get a **prefix-free** encoding that maps every character to a unique encoding. Given some bit string, all we have to do is start at the node and follow the bit string along the tree till we reach a leaf node. Every path to a leaf node in a tree is unique and hence our encoding is unique. Further, since it is on a tree and we stop only after reaching the leaf node, there can be **no ambiguity**. This means that the encoding is prefix-free!

In fact, for the above data, we can do no better than the encoding above. However, when we get to work with varying probabilities, things change. Shannon and Fano came up with an encoding that used the same concept of representing the encoding on binary trees to ensure they maintain the uniqueness and prefix-free requirements.

Their algorithm began by sorting the frequency of every event and then splitting the tree into two halves such that the prefix and suffix sum on either side of our division was as close to each other as possible. This had the intended effect of relegating lesser used symbols to the bottom (greater depth and hence longer encoding) and more frequently used symbols to shorter encodings. This was a big achievement and was known as the Shannon-Fano encoding for a long period of time. It was a good heuristic and performed well but it was **not** optimal.

Notice that with this greedy strategy, we **cannot** prove that it is taking the most optimal choice at the local level. This algorithm is **not** optimal.

At the same time, the Shannon-Fano encoding achieved both a unique representation of our data and more importantly, a prefix-free encoding that performed really well. Perhaps we can build upon their idea to obtain a prefix-free encoding with optimal compression.

Enter Huffman

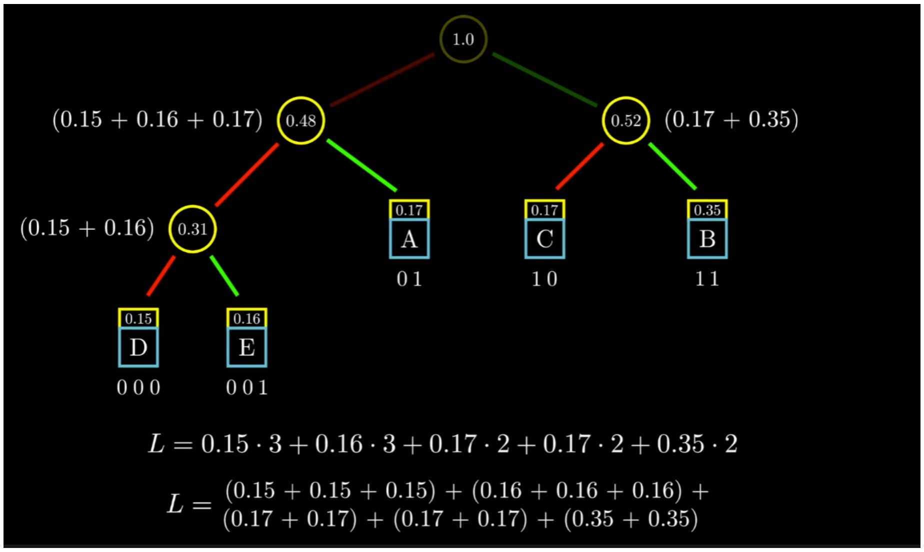
Contrasting the top-down approach used by Shannon and Fano, Huffman viewed the problem with a *slight* change in perspective. Instead of trying to start at the root, he claimed that if we picked the least two probable events, then they **must** be at the bottom of the tree.

Locally optimal choice

We want lesser used symbols to have longer encodings. If the above was not true, then that would imply that there is a symbol with a higher frequency of occurrence that is now given a longer encoding. This increases the size of the compression and is hence not an optimal choice. We now know for a fact that the least used symbols must belong to the bottom of the tree.

Optimum Substructure

We can write $L = \sum_{i=1}^{n} p_i.d_i$ where d_i is the depth of the ith node in the tree. Note that this quantity L is actually the same as the sum of the probabilities of every node except the root node in our tree. Consider the following example, notice that in the expanded view, the probability of each symbol gets included as many times as its depth in the tree.



$$L = 0.15 \cdot 3 + 0.16 \cdot 3 + 0.17 \cdot 2 + 0.17 \cdot 2 + 0.35 \cdot 2$$

$$L = \begin{array}{l}(0.15 + 0.15 + 0.15) + (0.16 + 0.16 + 0.16) + \\ (0.17 + 0.17) + (0.17 + 0.17) + (0.35 + 0.35)\end{array}$$

Remember that our goal is to minimize L. Let our symbols have probabilities/frequency $p_1, p_2, \ldots, p_k$ each and let us assume $p_1 \leq p_2 \leq \cdots \leq p_k$. Using our optimal greedy choice, we can choose the bottommost nodes as $p_1 + p_2$ and then restate the equation as follows.

$$L(p_1, \ldots, p_k) = p_1 + p_2 + L((p_1 + p_2), p_3, \ldots, p_k)$$

That is, we have managed to express the next problem as a smaller version of the original problem for which we realize that again, the greedy choice holds. We have managed to obtain the optimum substructure in our problem.

This implies that therefore, our greedy algorithm is indeed correct. **This** is the Huffman encoding.

Given some text data in the form of $(data, frequency/probability)$ tuples we can build the Huffman tree by using the greedy logic described above. Always greedily pick the smallest two probabilities to form the leaf node, then repeat. This is guaranteed to give us the optimal solution.

It is interesting to note its similarity to Shannon-Fano encoding, sometimes, all you need is the slightest shift in perspective to solve some of the world's unsolved problems :)

Huffman was able to **prove** that his encoding gives us the most optimal solution for encoding any set of $(data, probability)$ pairs as given. But… *can we do even better?* Theoretically no, but there are algorithms that can reduce the size of our data even more. The primary idea used by these algorithms is to chunk data into multiple byte chunks and then applying Huffman encoding. Note that while we mostly referred to ASCII text, Huffman encoding can be used to losslessly compress any form of binary data.

The following video was referenced while making this diary and is the source of some of the illustrations above, highly recommend watching this video.

Set Cover

Sometimes, greedy strategy doesn't work as the local optimum doesn't align with the global optimum. However, in some problems such as Set Cover, which is NP-hard, we still use the greedy approach as it gives a good approximation of an answer.

Input is a set of elements B, and a family of subsets $S_1, S_2, \ldots, S_m \subseteq B$

Output; A selection of the S_i whose union is B.

Cost: Number of sets picked

Greedy Strategy

Repeat until all elements of B are covered

Pick the set S_i with the largest number of uncovered elements

Example

$S = \{\text{arid, dash, drain, heard, lost, nose, shun, slate, snare, thread, lid, roast}\}$

Using each word as a set of letters, we must pick the minimum number of words to cover the set

$B = \{a, d, e, h, i, l, n, o, r, s, t, u\}$

First pick

We see that the word **length** covers the most uncovered elements

After picking the word **length,** the uncovered elements left are $\{i, l, n, o, s, u\}$

Second pick

We pick the word **lost**

We are left with uncovered elements $\{i, n, u\}$

Third pick

We pick the word **drain**

You're left with the sole element $\{u\}$

Fourth pick

You pick the word **shun** to cover the last uncovered element left

In this example, we find that greedy gives us the optimum answer which is 4, however, it's just by sheer luck, and it's not always the case we'll get the correct answer.

We can see that the **optimum substructure property** is held in this problem, However, we do not have the **greedy choice property.** If we did have a way to pick a local optimum that was aligned with the global optimum, the greedy approach would work. However, this is not the case.

Counter-example to Greedy approach

We have to cover set $B = \{1, 2, 3, 4, 5, 6\}$

we have subsets $S = \{\{1, 2, 3, 4\}, \{1, 3, 5\}, \{2, 4, 6\}\}$

If we used the greedy approach, we would pick the first set, which would force us to pick the other two sets as well.

However, it's clearly evident that the optimum solution is picking the 2nd and 3rd set. Therefore, we've proven that greedy approach is not optimum

Approximation

If we have a set B of n elements, and the optimal cover consists of k sets. Then the greedy algorithm will use at most $k \cdot \ln n$ sets.

Proof of approximation

Let n_t be the number of elements left uncovered after the t^{th} iteration.

We know that all elements are left uncovered before starting, therefore $n_0 = n$

We know we are done with iterations when $n_t = 0$, and t will be the number of sets we used in our answer

Say at the t^{th} iteration, we have n_t elements left, we know for a fact that the optimal answer k can definitely cover the elements left. Hence, there is a set that has to cover at least n_t/k elements by pigeonhole principle.

Hence, if we pick that element, the upper bound on the number of uncovered elements left in the $(t+1)^{th}$ iteration is given by

$$n_{t+1} \leq n_t - \frac{n_t}{k} = n_t \cdot \left(1 - \frac{1}{k}\right)$$

Therefore, for the general case n_t, the upper bound is given by

$$n_t \leq n_0(1 - 1/k)^t$$

We know that $1 - x \leq e^{-x}$, equality only occurs when $x = 0$

Hence, we can rewrite the upper bound for n_t as

$$n_t \leq n_0 \cdot \left(1 - \frac{1}{k}\right)^t < n_0(e^{-1/k})^t = ne^{-t/k}$$

When we plug $t = k \ \ln \ n$, then we get

$$n_t < ne^{-t/k} = ne^{-(k \ \ln \ n)/k} = ne^{-\ln \ n} = ne^{\log_e \ \frac{1}{n}} = n \cdot \frac{1}{n} = 1$$

Hence, after t iterations, we get $n_t < 1$, and we know that n_t has to be a non-negative integer, it implies that n_t is 0, which is our terminating condition

Which means that after k ln n iterations, we are guaranteed to have covered all the sets, where k is the optimal number of solutions.

Knapsack and Dynamic Programming

The Knapsack problem

The Knapsack is a very famous problem. It asks the following question, *"Given a list of n elements, each of which have some value v_i and weight w_i associated with them, what is the maximum value of elements I can fit into my Knapsack given that my Knapsack can only hold at max a weight of W capacity?"*

There are two variations of the above problem as well. The simpler one assumes that we have an infinite quantity of each element. That is, we can pick an element as many times as we wish. The harder version does not assume this. Each element can only be picked once.

A toy example

For the sake of illustration, we'll assume we are attempting to solve the Knapsack for the given inputs

We have 4 items with their respective v_i and w_i values. Our Knapsack has a maximum capacity of $W = 10$.

With repetition

Optimal solution with DP

If repetition is allowed, we can solve the problem using a very simple approach. All we need to observe is that to compute the maximum value for a bag of capacity W, we can simply brute force over all elements with a simple recurrence.

Let $F(W)$ be the maximum value obtainable for a bag of capacity W. Then,

$$F(W) = max(v_1 + F(W - w_1),\ \dots v_n + F(W - w_n))$$

In our example, this corresponds to the following computation $\implies$

$$F(10) = max(30 + F(10 - 6), 14 + F(10 - 3), 16 + F(10 - 4), 9 + F(10 - 2))$$

$$\implies F(10) = max(30 + F(4), 14 + F(7), 16 + F(6), 9 + F(8))$$

The idea behind this recurrence is as follows. At any capacity W, we are simply picking every possible element and asking what is the maximum value I can achieve **after** picking each element. It's more or less just a brute force that considers picking every element for each capacity W.

It is easy to see that we are computing the answer for W such sub-problems from $W_i = 1 \to W$. And at each sub-problem, we are iterating over n elements.

It is also important to note that we do not consider including the element in our brute force when we reach a state where $W - w_i < 0$. This is an impossible/unreachable state. The base case is when we no longer have any elements which we can fit into the bag.

1. Hence we have W sub-problems.
2. We are doing $O(n)$ computation at every node.
3. The recurrence is as described above.
4. The DAG structure is also easy to reason about. It's simply just a linear chain from state

   $W_i = 1 \to 2 \to \cdots \to W$
5. Therefore, our final algorithm will have $O(nW)$ complexity.

Further, since there are only $O(W)$ sub-problems, we only need $O(W)$ space to store the DP table.

Without repetition

Greedy Ideas

When we are working on $NP - Complete$ optimization problems such as this, it is always a good idea to **keep it simple** and come up with simple greedy heuristics first. We can always work on improving the greedy idea and, see how we perform, what counter cases and exist and then repeat. This is always a good starting point to explore before we throw heavier tools at it.

1. Value 1 - Weight 2 (x3)
2. Value 10 - Weight 5 (x2)
3. Value 7 - Weight 3 (x1)

4. Value 13 - Weight 8 (x1)

$W = 10$

To the human eye, we can quickly see that the optimal solution would be to pack both elements of type 2 into the bag to get a maximum score of 20 points. But let's try out greedy ideas to see how they work out.

Picking highest value

One natural greedy idea is to pick the element with the highest value irrespective of weight. If we do this, we notice that we can pick one element of type 4 and one element of type 1. This gives us a total score of 14.

Picking lowest weight

Another idea might be to pick items of least weight so that we can fill our knapsack with as many items as possible. This idea tells us to pick 3 items of type 1, and one item of type 3. This gives us a total score of 10 points.

Picking highest value/weight ratio

A natural combination of the above two ideas would be to pick the elements sorted by their value/weight ratio. This strategy tells us to pick one item of type 3, one item of type 2 and one item of type 1. This gives us a score of 18 points!

Conclusion?

Notice that although the 2nd greedy wasn't the best, it could've easily fared much better given inputs where some of the 2nd most valuable items also had the least weight. Similarly we can construct cases for which the greedy solutions perform optimally. The third one tries to combine both and is a pretty good heuristic for even a generic case such as the one discussed. It did decent and came within 2 points of the optimal solution! But with greedy solutions such as this, it is important to note that we can always construct a case where it performs poorly. In the realm of optimization algorithms, we try to start by solving the whole problem optimally, and when we realize we can't, we try to relax the constraints of the problem and lower the quality of the solution from "optimal" to "high quality" to slowly extract more and more performance in exchange for slightly worse solutions. However, for this problems, we can indeed write an optimal DP solution. We would have to rely on greedy heuristics only for very large input.

Dynamic programming

Highest cost

Notice that our previous DP solution will not work here. Because we cannot choose elements multiple times. However, the order of choosing the elements does not matter either. But because of this condition, notice that it is not enough to simply consider sub-problems defined by just one characteristic.

That is, a sub-problem in the previous case was simply identified by W, the size of the Knapsack. Here, this is no longer the case. A "state" or "sub-problem" has at **least** two changing variables. Both the number of elements we are including into the Knapsack **and** the weight of the Knapsack.

The new DP state

That is, we must change the definition of our DP to a 2-d DP where $DP[i][j]$ represents the state where we are considering the **first** i elements among the list of available elements and our Knapsack is of size j.

1. **Number of sub-problems**

   Since we have n possible prefixes which we will consider and W possible values for the weight, we have of the order $O(nW)$ sub-problems to compute

2. **Finding the solution at some node**

   Notice that since we changed the definition of our DP to storing the best possible answer to the problem given that our Knapsack has size W and we are only considering the first i elements, when computing $DP[i][j]$, notice that we are only trying to **include** the i^{th} element wherever it maximizes our answer.

   This has the important implication that we do not need to brute force over n elements at some state $[i, j]$. We only need to check the states $[i-1, W-w_i]$. This is $O(1)$ computation at every node.

3. **Coming up with the recurrence**

   We are essentially trying to answer the question

   "At some capacity W, when considering the i^{th} element, does including it in the Knapsack help increase the previously obtained score at capacity W when considering only $i-1$ elements?"

   Writing this recurrence formally,

   $*F(i, W) = max\{F(i-1, W), F(i-1, W-w_i)\}*$

   The first term in the max represents the previously obtained score at capacity W. The second term is the value we would get if we tried including element i when considering a bag of size W.

4. The **DAG structure** for this problem is very similar to the structure obtained when solving the Edit distance problem. It is simply a graph where each state $[i, W]$ depends on the state $[i-1, W-w_i]$.

5. We have an algorithm that requires us to perform $O(1)$ computation for each of the $O(nW)$ sub-problems. Hence the total running time will be

$O(nW)$. However, since there are nW sub-problems, we will also require $O(nW)$ space.

Can we do better?

This time, we actually can! Notice that just like how we did in the Edit distance problem, the DP state at any $[i, W]$ is **ONLY** dependent on the DP states exactly one level below it. That is, every DP value in row i is only dependent on the DP values in row $i - 1$.

This means that again, we can do the exact same thing and use **Single Row Optimization** to reduce the space complexity of our DP from $O(nW)$ to just $O(W)$. For small values of W, we might even consider this linear!

Pseudo-polynomial-time algorithms

At first glance, it is very easy to write off the Knapsack problem as belonging to the P complexity class. After all, it seems to just be quadratic right?

But this is not true. We define the complexity of algorithms based on input size n.

To be more precise: *Time complexity measures the time that an algorithm takes as a function of the **length in bits** of its input.*

However, notice that in this case, the complexity of our algorithm relies on both n and W. W is the **value** of an input. If we consider Win binary, we would require $log_2(W)$ bits to represent W. If the input is in binary, the algorithm becomes **exponential.**

Why?

We will try to explain this by means of a nice example.

1. Let's say we are trying to solve the problem for $n = 3$ and $W = 8$. Keep in mind that $W = 1000$ in binary. That is, W is **4 bits** long.

   Hence total complexity $= O(nW) \implies O(3 \times 8) = O(24)$

2. Now let's increase n to $n = 6$. We have linearly multiplied it by 2. Notice that this still gives us

   Time complexity: $O(nW) \implies O(6 \times 8) = O(48)$. It is the expected increase by 2.

3. Now let us increase W by a factor of 2. **Notice that this means we double the length of W in bits. Not the value of W itself.** This means W will now be represented by $W = 8$ bits. This means W is now equal to 10000000 in binary.

   This gives us a complexity of $O(nW) \implies O(3 \times 2^8) = O(768)$. That is, there is an exponential increase in complexity for a linear increase in W .

Knapsack is NP-Complete

The Knapsack problem is in fact, an **NP-Complete** problem. There exists no known polynomial-time algorithm for this problem. However, it is nice to know that is it often classes as *"Weakly NP-complete."*

That is, for small values of Wwe can indeed solve the optimization problem in polynomial time. If we give input W in the form of smaller integers, it is weakly NP-Complete. But if the value W is given as rational numbers, it is no longer the case.

Alternate views to solve the Knapsack problem

While we solved the Knapsack problem in the standard manner by defining $DP[i][j]$ as the maximum value achievable when considering the first i elements and a bag of capacity j, what do we do if the value of W is large, but the value of $\sum_i^n v_i$ is small?

Consider the following two problems from the AtCoder Educational DP contest.

Knapsack - 1

The first problem is simply the standard Knapsack problem.

The constraints for it were as follows,

$$1 \leq N \leq 1001 \leq W \leq 10^5 1 \leq w_i \leq W1 \leq v_i \leq 10^9$$

A $O(nW)$ solution would take around 1e7 operations which is passable.

Here's a link to my submission: https://atcoder.jp/contests/dp/submissions/19493344

Knapsack - 2

The second problem is a little different. It asks the same question, but for different constraints.

$$1 \leq N \leq 1001 \leq W \leq 10^9 1 \leq w_i \leq W1 \leq v_i \leq 10^3$$

Notice that W is now 10^9. $O(nW)$ would now take 1e11 operations. This would practically have a very slow running time in comparison to our previous ~1e7 operation solution.

We will have to think of something different.

Notice that for this problem, the values v_i are much smaller. In fact, considering $n = 100$ elements, the maximum value obtainable is just $max(v_i) \times n = 10^5$.

Now, we can exploit this by doing the same Knapsack DP, but this time, instead of storing the maximum value achievable in max capacity j when considering the first i elements, we redefine the dp as follows.

$DP[i][j]$ will now store the minimum weight required to achieve value j when considering just the first i elements. We can now simply pick the maximum j in row $i = n$ which satisfies the condition $DP[i][j] \leq W$.

This solution runs in $O(n \times \sum_i^n v_i)$ which gives us ~1e5 operations. This is much faster than the standard approach.

Here's a link to my submission: https://atcoder.jp/contests/dp/submissions/19494460

Why is Knapsack useful?

Interplanetary exploration

Consider the Mangalyan project from ISRO. Only a very limited amount of resources can be packed on such a rocket and sent to Mars. It is not easy to send supplies to Mars following this launch. Hence it is of great importance that we attempt to maximize the value of the items sent to Mars. These items can be given importance empirically by the scientists. Remember that the rocket has limited space. Does this problem sound familiar? Indeed, this is a real life example of the importance of the Knapsack optimization algorithm.

Consider storage sites

Warehouse systems, database systems, export sites, etc. Any place that profits from storing high value contents and has limited space greatly relies on solving the knapsack problem to run.

Knapsack using Branch and Bounding

When trying to solve problems in general, (especially optimization problems) it's always a good idea to formulate a mathematical definition of the problem. To formulate it in mathematical terms, we assign each item in the knapsack a *decision variable* $x_i \in \{0, 1\}$. Now, each item in the knapsack also has some weight w_i and value v_i associated with it. Let's say our knapsack capacity is denoted by W.

The decision variable x_i simply indicates whether we include item i in the knapsack or not. Using this, we can define the objective function that we wish to optimize (maximize) as:

$$V = \sum_{i=1}^{n} v_i x_i$$

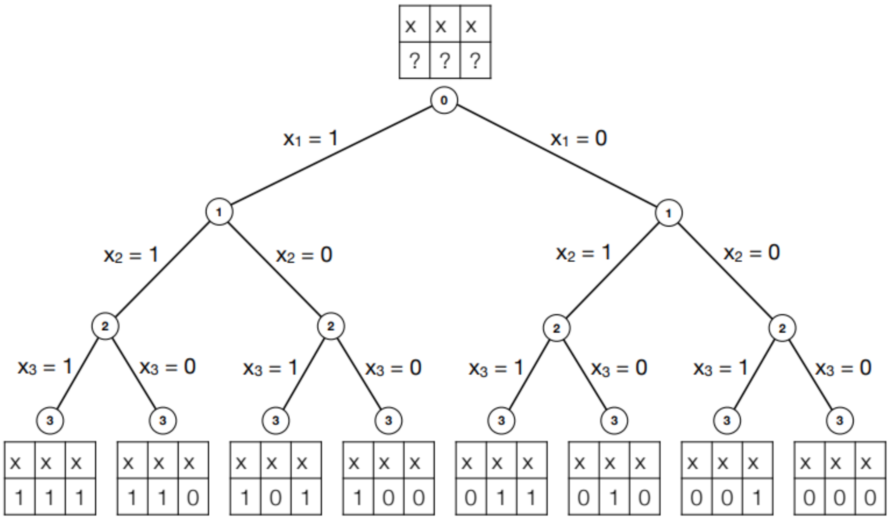
Under the constraints that

$$K = \sum_{i=1}^{n} w_i x_i \leq W$$

Now we have a formal definition of the function we want to maximize under some constraints.

Branching

Now, one way to solve the knapsack problem would be to perform an "exhaustive" search on the decision variables. This would mean checking every possible combination of values that our decision variables could take. Visualized as a tree, it would look like this:



Here, we compute the answer by **branching** over all possible combinations. This is, of course, $O(2^n)$. Increasing input size by one would literally mean doubling the amount of computation done.

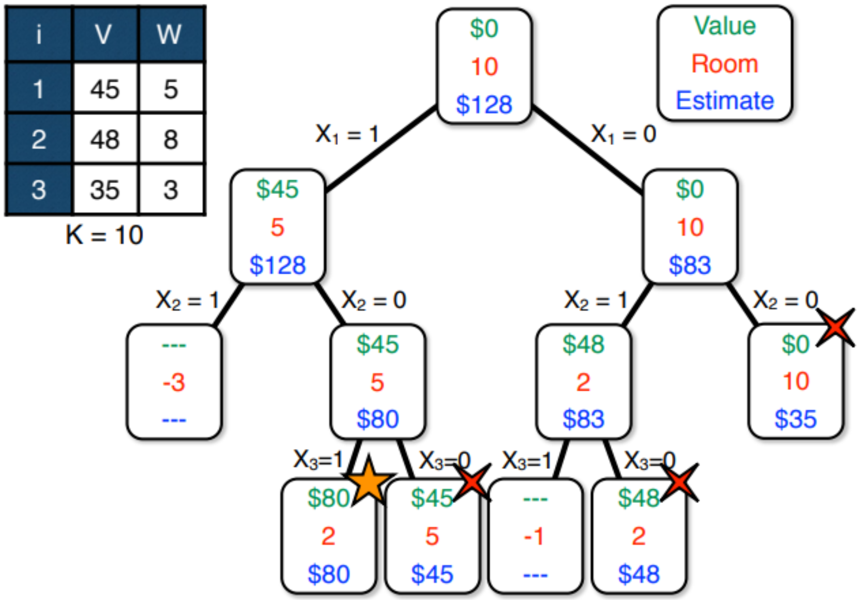
Relaxation

This is where the idea of relaxation comes in. Branching was basically us splitting the problem down into subproblems by locking or *constraining* each variable and then deciding on the constraints for other variables. Exhaustive search isn't ideal, hence we try to reduce the search space by implementing some **"bound."** Essentially, at each step of the recursion, we place an optimistic bet or estimate on just how good the solution to our subproblem can be. If at any point in the recursion, this estimate is lower than the best-found value so far, we can kill off the recursion.

As we've mentioned before, **relaxation** is the key to optimization. The original problem is **NP-Complete**. This means that until someone can prove $P = NP$, these problems have **NO** polynomial-time optimal solution. Relaxation is the art of how we deal with these problems.

Using relaxation on the exhaustive search

So, what constraints can we try to "relax" in the knapsack problem? The only constraint there is the weight of the Knapsack. So let's start by relaxing it to let us have an **infinite** knapsack. A picture is worth a thousand words, so let me just show you what the search would like with this relaxation.



i	V	W
1	45	5
2	48	8
3	35	3

K = 10

Let's try to see what we did here. First, we begin by letting root have $W = 10$ space and have $V = 0$ as it is completely unfilled. This is the $(0, 0, 0)$ state. The **estimate** here is our relaxation. Assuming infinite space, we see that the **most optimistic** value we can reach from here is $128 if we include all the items. Remember, the *relaxation* is for calculating this *estimate*.

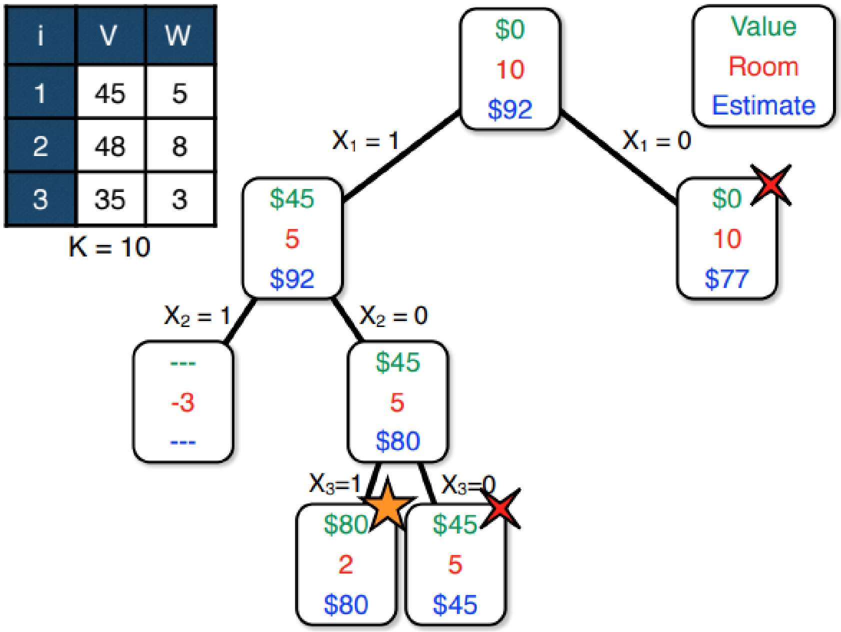
Once this is done, we're just performing an exhaustive search. But now, notice that the node on the left stops its recursion once the room in the knapsack has become negative. This is a simple base case on the original recursion.

The interesting part is the recursion that we've killed on the rightmost node. The leaves marked with crosses went till the end until they were discarded in favor of the left bottom leaf which is our optimal score of 80. Now, the rightmost node was killed even **before** it reached the leaves. This is because we had already achieved a better score (80) than the best possible estimate from this node. Hence we know for a fact that following the recursion can never give us a *better* score. Recall that our relaxation was an infinite knapsack. If we cannot do better with an infinite knapsack from that point, there is no point in searching further down that track. This is the key idea behind relaxation and how using bounds can help us optimize exhaustive search. However, this particular relaxation was not very effective and did not help much in optimizing our search. But maybe with a *better* heuristic, we can optimize the search further.

Coming up with a better heuristic

Let's think about how we would normally solve the Knapsack problem **if we were allowed to take rational amounts of an item**. That is, it is no longer a 0-1 problem where we must either take or discard an item. We can take items in parts now. This problem has a fairly straightforward and greedy solution. We simply sort items by $\frac{v_i}{w_i}$. This is essentially their "value per 1 unit room." Simply pick the element that gives the best value per weight. So the strategy is now picking the element with the highest $\frac{v_i}{w_i}$ ratio, and when we run out of space pick the last element in a fractional amount such that it fills up the entire knapsack.

This is the **optimal** solution for this version of the knapsack problem. But what about when we apply this relaxation to the original exhaustive search model instead of the infinite bag relaxation?



i	V	W
1	45	5
2	48	8
3	35	3

K = 10

Notice how much better we've managed to optimize the exhaustive search. The right child of the parent node is cut off at $77 and does not search further, because our "estimated" cost is lesser than the highest value we have found so far ($80).

Optimality

Notice that the fractional knapsack is the best-case version of the knapsack as we can optimally fill every unit of space according to a greedy strategy. Therefore if even the greedy estimation is below previously found maxima, then this quantity cannot be optimal. This means this branch and bound relaxation will still give us the **optimal** solution to the 0-1 knapsack problem.

Complexity

Analyzing the running complexity of branch and bound algorithms has proved notoriously difficult. The following blog gives some intuition as to why we find placing a bound on such techniques very difficult. https://rjlipton.wpcomstaging.com/2012/12/19/branch-and-bound-why-does-it-work/

George Nemhauser is one of the world's experts on all things having to do with large-scale optimization problems. He has received countless honors for his brilliant work, including membership in the National Academy of Engineering, the John Von Neumann Theory Prize, the Khachiyan Prize, and the Lanchester Prize. To quote him,

> *"I have always wanted to prove a lower bound about the behavior of branch and bound, but I never could."* - George Nemhauser

Putting a good bound on branching and bounding is very difficult and is an open problem. One alternative measure that is used to better estimate the efficiency of branch and bound algorithms is its **effective branching factor (EBF).**

We define EBF as the number b so that your search took the same time as searching a b-ary tree with no pruning. If you are searching a tree of depth {d}, this is well-defined as the d-th root of the total number of nodes you searched.

This is computed in practice and is used a lot in solving optimization problems as it is quite effective in practice, even if it is difficult to put bounds on theoretically. The fact that the runtime can be altered significantly by simply changing the relaxation criteria also makes it a great option to try out when coming up with relaxation ideas.

Why not just stick with Dynamic programming?

This is a natural question. DP seems to give us an approach where we can be comfortable in sticking with an $N * W$ or $N * V$ complexity solution. However, what if we modify the question ever so slightly and allow items to have fractional or real weights? This seems like a problem that might surface in the real world a fair amount. The DP table approach is no longer feasible. In such a situation the branch and bound algorithm might come in clutch. As we explore such problems and minor variations of such problems, the need to expand our toolbelt and come up with more and more optimization algorithms becomes clear.

Probabilistic Algorithms

At times we come across problems which may be hard problems (NP hard, NP complete), or problems which become infeasible to compute for a given input size due to the algorithm that is used. For example an $O(n^2)$ algorithm might be pretty good for smaller input sizes, but practically useless if we need to solve the problem for input sizes $\geq 10^6$ regularly.

However, for some problems, we can come up with algorithms which may not guarantee the correct answer, but can guarantee a very high probability of giving the right answer. These types of algorithms can provide a huge improvement over the deterministic algorithms for the problem, and can have huge applications provided the probability for failure is sufficiently low.

However, since there is a small probability of the algorithm failing, people with a malicious intent could take a look at the algorithm, and construct a case for which the algorithm would fail, and the malicious user could take advantage of this failure. This is when randomization comes into play

Randomization

By introducing randomization into a probabilistic algorithm, it becomes significantly more tough to break an algorithm, as it's hard to construct a case while predicting what the random number would generate next.

Example of the usefulness of Randomization

Say we have a list of size n where half of them are $1s$ and the other half of them to be $0s$.

If the problem we have at hand is to find the index of any 0 that occurs in the list, then we could simply try solving this problem by going through the list.

[] for i in binary_string: if i == 0: return current_index else continue

This algorithm resembles the Geometric distribution, number of failures till a success. Since the probability of getting a 0 is 50%, we can say that the expected number of iterations, and the variance will be

$$E(iterations) = 1/p = 1/0.5 = 2$$

$$Variance = \frac{1-p}{p^2} = \frac{0.5}{0.5^2} = 2$$

Hence, we can see that the algorithm we used should terminate quite quickly. However, we're making one very important assumption while calculating these values, which is that we are assuming that the input is truly random.

If we were told that the strings given will be truly random, then we can safely say that our algorithm should terminate quickly.

However, if we aren't given this guarantee, there's nothing to stop the user from entering malicious input to *break* our algorithm.

For example, the user may enter a list of size n, with the first $n/2$ bits as 1, and the next $n/2$ bits as 0. This would cause our algorithm to iterate over half of the entire list before terminating.

Even if we tried to modify the algorithm, it could still be broken based on the modifications we make. Checking the even indexes first, then the odd indexes, iterating from both the first element and the last element, all these algorithms can be broken, as the behavior of the algorithm is highly predictable given an input.

Hence, if we aren't given the guarantee that the input is random, we could make use of randomization to end up using the same properties as if the input was random.

Therefore, instead of iterating from the start and going through the entire list with some predetermined order, we could pick random indexes, making it nearly impossible to construct a malicious input to break the algorithm.

[] while true: pick a random element in list, call it i if i == 0: return curr_index else continue

By introducing randomness, we've made the probability of the algorithm running too long very less, even in the instance of a malicious user attempting to break the algorithm.

Running Time

We have an estimate on the average number of times that the algorithm would run. However, it would be more descriptive and informational to give the probability of the algorithm running in some specified time.

If we say that finding the first 0 took x attempts, we can say that the answers to our queries would've looked something like this $1111\ldots0$. Hence, the probability of getting this exact configuration is $1/2^x$.

We can also define the probability of not getting a single 0 in y attempts to be $1/2^y$. Using this, we can define the probability of finding a 0 in y tries to be $1 - 1/2^y$.

Hence if we only have 3 attempts to find a 0, then the probability of succeeding will be $7/8 = 0.875$.

The probability of finding a 0 skyrockets to ≈ 0.996 when we increase the number of attempts to 8. Hence we can see that although the algorithm doesn't have a fixed running time, it has a very high probability of running extremely quickly.

Types of Probabilistic Algorithms

There are two main types of probabilistic algorithms

1. **Monte Carlo -** Monte Carlo algorithms are algorithms are algorithms which give an output that is "probably correct". Hence, the probabilistic part of the algorithm is the correctness of it. An example of a Monte Carlo algorithm is primality testing.
2. **Las Vegas -** Algorithms which do give the correct output, but whose execution time is "probably fast". Hence the probabilistic part is its running time. An example of a Las Vegas algorithm is the Quicksort algorithm.

Is the random we use truly random?

While the random values generated by library functions in various languages seem to be quite random, computers practically cannot generate truly random values. Most languages implement their random value generating functions using a *Linear Congruential Generator,* which uses an initial seed. It can be observed that if we tried running the `rand()` function in C++, we end up getting the same values every time we execute it. Hence, it's recommended to provide the current time in seconds as the seed, by doing `srand(time(NULL))` hence, making it tougher to predict the values, and this is usually sufficient to get "random" values.

However, based on the determination and passion of the malicious user, even this may not be sufficient, and may still be breakable. Since the seed only changes every second, the malicious user may attempt to run the algorithm at a certain time at which he may have calculated the values that would be generated with such a seed.

Obviously this may not work every time as the user may not be able to get the algorithm to run at the exact time he pleases, but if the user wishes to, he could definitely break it within a few attempts, as he's bound to get an execution at the second he wishes eventually

Therefore, the most safe option for generating random numbers is using the number of milliseconds as the seed for the random number generator which can be found using `std::chrono::system_clock::now().time_since_epoch() / std::chrono::milliseconds(1);`, as predicting the millisecond count at the time of execution is nearly impossible.

Line through points

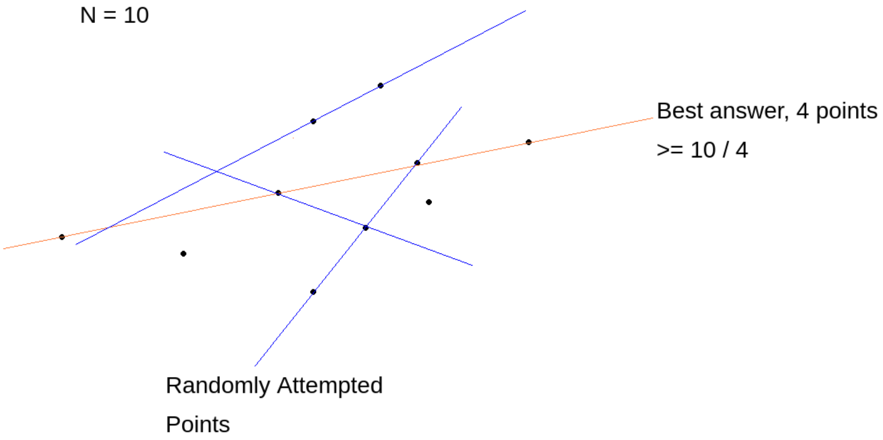
There are problems for which a fast algorithm may not exist, but under some specific constraints could allow for a much more efficient probabilistic algorithm. We'll cover an example of such a type of problem

Problem

You are given a set of N points on a 2-dimensional plane. Find a line that passes through the maximum number of points possible. It's guaranteed that the answer is at least $N/4$.

If we exclude the last constraint, that "the answer is guaranteed to be $\geq N/4$", then the solution to this question is $O(n^2)$, which can be efficient, but infeasible for $N \geq 10^5$, for which we would require a solution with better time complexity.

We can make use of the fact that the answer is $\geq N/4$ to construct a probabilistic algorithm



Probabilistic Algorithm

Say we pick two points from the set. What's the probability that both of these points are included in the answer?

$$P(\text{Both points are in answer}) = \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$$

Therefore, if we pick 2 points, the probability of the line connecting the two dots passes through the maximum number of points is 1/16. Hence, the probability of it not being the answer is 15/16.

Using this, we can pick 2 random points, and go through all the points in the set to check if they fall on this line (i.e co-linear with the 2 random points picked).

After doing this multiple times, we can pick the maximum answer we've gotten with a good amount of confidence.

Say that we picked 2 random points x times. The probability of not getting the answer will be

$$P(\text{not getting best answer}) = \left(\frac{15}{16}\right)^x$$

This may not seem very reliable, because if we pick a value $x = 10$, the probability of not getting the best answer is ≈ 0.52, which is pretty terrible. However, we need to remember that running a simulation takes $O(n)$ time, compared to the deterministic $O(n^2)$ solution, hence we can pick a larger x, such as 100, which gives probability of failure as ≈ 0.00157.

Therefore, with a sufficient value of x, we can get the right answer with a very high probability.

Need for N/4 constraint

Say that we weren't given the $\geq N/4$ constraint. Would this algorithm still work?

Take an example where $N = 10^5$, and the answer is 100.

The probability that both the points chosen randomly lie on the answer is

$$P(\text{Both points are in answer}) = \frac{100}{100000} \cdot \frac{100}{100000} = \frac{1}{10^6}$$

Hence the probability of not getting the best answer is

$$P(\text{not getting best answer}) = \left(\frac{99999}{10^6}\right)^x$$

Even if we pick $x = 100$, the probability of not getting the best answer is ≈ 0.999

Only if we pick a value like $5 \cdot 10^5$, we get the probability of failure down to ≈ 0.006, but at that point, we're basically running an $O(n^2)$ solution.

Hence, we can see the significance of the $\geq N/4$ constraint, and how it allows us to pick random points with much higher probability of them being in the final answer that we're looking for.

Freivald's Algorithm

Freivald's algorithm is a simple algorithm that beautifully demonstrates the motivation behind using probabilistic algorithms. It comes under the umbrella of **Monte Carlo** algorithms, as it outputs an answer, which is correct with a certain probability, but has a fixed running time.

Problem

In this problem, we deal with 3 $n \times n$ matrices, which we can refer to as A, B, C.

Given these three matrices, we are required to find out if

$$A \times B = C$$

Simple Method

The simplest way to verify this condition is to multiply the two matrices A, B and then verifying if their product matches with the matrix C.

- If we use the simple matrix multiplication algorithm, the time complexity of this method will be $O(n^3)$.
- However, we can make use of faster matrix multiplication algorithms such as Strassen's matrix multiplication algorithm which has time complexity $O(n^{2.81})$, reducing the time complexity. (There have been breakthroughs, and the best known matrix multiplication algorithm has a time complexity of $O(n^{2.3728})$

However, this time complexity might be too slow in certain applications, and we must figure out methods that can do the verification with much better time complexity.

Probabilistic Algorithm

Instead of ensuring the output we get is 100% correct, we end up going for an algorithm which has a much better time complexity $O(n^2)$, at the cost of the risk of the algorithm failing with a non-zero probability. Hence, we'll go over how Freivald's algorithm works

The algorithm we'll use has two main properties

- If $A \times B = C$, then the probability with which we'll say that the condition satisfies is 1
- If $A \times B \neq C$, then the probability with which we'll say that the condition satisfies is $\leq \frac{1}{2}$

Freivald's Algorithm

For demonstration, we'll make use of matrices with only binary numbers $\{0, 1\}$.

The algorithm is as follows

- First, we pick a random binary vector r

$$r = \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}$$

- The probability that $r_i = 1$1 is $\frac{1}{2}$, for all values of i. Therefore, every binary column vector of size n is equiprobable of being selected/generated.
- Next we'll use this matrix to perform a verification whether the condition $A \times B = C$ is satisfied.

  We know that if $AB = C$, this implies that $ABr = Cr$.

  Therefore, if we verify if $ABr = Cr$, we are indirectly verifying if $AB = C$ with some probability. If we find out that $ABr \neq Cr$, then we can output that the condition isn't satisfied with 100% confidence. However, if the condition $ABr = Cr$ is satisfied, it increases the probability of the initial condition being true, although it doesn't guarantee it.

  One may feel that computing ABr and Cr provides no improvement over just performing normal matrix multiplication. However, one can easily see that multiplying a $n \times n$ matrix with $1 \times n$ matrix can be done in $O(n^2)$ time.

  $$Br = \begin{bmatrix} b_{11} & b_{12} & \dots & b_{1n} \\ b_{21} & b_{22} & \dots & b_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ b_{n1} & b_{n2} & \dots & b_{nn} \end{bmatrix} \cdot \begin{bmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{bmatrix}$$

  Computing Br takes $O(n^2)$ time as it involves computing an $n \times n$ matrix by $1 \times n$ matrix. Similarly, computing $A(Br)$ also takes $O(n^2)$ time, and Cr has the same time complexity to compute too.

  Therefore, checking if $A(Br) = Cr$ takes $O(n^2)$ time.

  Hence, we end up doing 3 matrix multiplications with a column vector which take $O(n^2)$ time each, and performing an equality check of two column vectors, which takes $O(n)$ time.

Proving that if $AB \neq C$, then $Pr[ABr \neq C] \geq \frac{1}{2}$

We'll define the difference matrix D as

$$D = AB - C$$

If $D \neq 0$,(Null matrix) then we know the condition is not satisfied. Therefore, finding the probability that $AB \neq C \implies ABr \neq Cr$ is equivalent to finding the probability of $Dr = 0$

If $D \neq 0$, we know that $\exists\ i, j$ such that $d_{ij} \neq 0$.

We'll find a column vector v such that $Dv \neq 0$

$$Dv = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \\ \vdots & & & & & \\ 0 & 0 & \dots & d_{ij} & \dots & 0 \\ \vdots & & & & & \\ 0 & 0 & 0 & \dots & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ \vdots \\ v_j \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ (Dv)_i \\ \vdots \\ 0 \end{bmatrix} \neq 0$$

Now, we find a "bad" r, which essentially gives $Dr = 0$, even though $D \neq 0$.

We'll make use of v to get a "good" $r' = r + v$.

By using this r', we'll get $Dr' \neq 0$.

This can be easily proven

$$Dr' = D(r + v) = Dr + Dv = 0 + Dv \neq 0$$

We can show that the mapping between r and r' is one-to-one i.e. injective.

To show that a mapping is injective, we need to show that $f(x_1) = f(x_2) \implies x_1 = x_2$

We define the function $f(r) = r + v = r'$

If $r_1 + v = r_2 + v = r'$, we could subtract v from both sides, to show that $r_1 = r_2$

Hence, we've shown that the mapping of r to r' is one to one. Hence the set of "bad" r is injective on the set of "good" r.

Since the mapping is injective, we can say that the range of the function is $\geq$ the domain of the function.

Therefore, we know that |good r| $\geq$ |bad r| .

Hence, we can say that the number of r for which $Dr \neq 0$ is greater than the number of r for which $Dr = 0$, if it's given that $D = 0$.

Hence, we've proven that the probability that $ABr \neq Cr$ given $AB \neq Cr$ is $\geq \frac{1}{2}$.

Now, a 0.5 probability of success might not seem that great, but we can make use of this property to take multiple trials by choosing different values for r.

Say the probability of failure is 1/2. If we pick k different vectors, then the probability of failing to identify that the condition doesn't satisfy will be $\frac{1}{2^k}$.

Assuming that the trials are independent, and we pick the vectors randomly, the probability of the test failing after taking 10 different vectors is $1/2^{10} = 0.0009$, which is very low.

Therefore, we have an algorithm which instead of performing the entire matrix multiplication to verify the product of 2 matrices, uses a probabilistic method to verify the product with very low chance of failure, which can be adjusted to suit the application's need, at the cost of running time.

Primality Tests

Why do we need Primality Testing?

Primality Testing is a test which we use to determine whether a given number is prime or not. Primality testing has a major application in cryptography, where prime numbers form the base of encrypting your data safely.

Obviously, primality testing sounds like a test that's very straightforward, just check all the numbers that are less than the inputted number n and check if any of them divide it without any remainder. If there exists such a number, then we say that the number is **composite**, else it's **prime.**

A simple optimization can be made however, to reduce the time complexity from $O(n)$ to $O(\sqrt{n})$ by just checking the first $\sqrt{n}$ numbers.

An $O(\sqrt{n})$ solution can be very fast, and sufficient for many use cases as well. However, in fields such as cryptography, even an $O(\sqrt{n})$ is not sufficient, as cryptography deals with **very** large primes, beyond the range of 10^{18}, hence igniting the search for a faster solution.

Probabilistic Algorithms

Instead of only looking at algorithms which are guarantee the answer to the primality test, we can widen our search range for solutions by looking for solutions that give a very high probability of giving the correct answer. One of these solutions is the **Miller-Rabin Primality Test.**

The **Miller-Rabin primality test** is a test that does not guarantee that the answer it gives is correct. However, it is an algorithm which provides a bound on the probability of the answer being correct, and can easily be tweaked to increase the probability at the cost of efficiency, or vice versa.

Miller-Rabin Algorithm

The Miller-Rabin primality test is guaranteed to say that a number is prime if it actually is prime, and has a very high probability of saying that a composite number is composite. Hence, the error that might occur is that the algorithm may call a composite number a prime number with very low probability.

However, an adversary might try to come up with worst case input to break the algorithm, hence the primality test makes use of randomization, which makes the algorithm much harder to break due to lack of a uniform pattern.

Steps of Algorithm

Input : An integer p.

1. If $p = 2$, then accept (claim that the number is prime). Else if p is even, then reject (claim the number is composite).
2. Pick k numbers, $a_1, a_2, \ldots, a_k$ from $\mathbb{Z}_p^+$ (Numbers $0, 1, \ldots, p-2, p-1$).
3. For each i from 1 to k
    1. Compute a_i^{p-1} mod p and reject if different from 1. This is in accordance with Fermat's Little Theorem, which states that $a^{p-1} \equiv 1 \bmod p$ if p is prime.
    2. Since we are only dealing with odd value of p, since we accepted/rejected all even numbers in step 1, we can say that $p-1$ is even. Hence, we can factorize $p - 1 = 2^k \cdot s$, where s is some odd number.
    3. We then compute the sequence $a_i^{s \cdot 2^0}, a_i^{s \cdot 2^1}, \ldots, a_i^{s \cdot 2^k}$ mod p.
    4. We read from right to left, and try to find an element in this sequence that is not 1. If the first number we come across a number which is not 1, and that number isn't -1, then we reject p (claim that it's composite).
4. If we haven't rejected the number p till now, it implies that we have passed all the tests, hence we accept the number p (claim that it's prime).

Time Complexity

If we are given a number p, and we pick k numbers from $\mathbb{Z}_p^+$, then the time complexity of the test is $O(k \cdot log^3 p)$, a significant improvement over the simple $O(\sqrt{n})$ solution, provided we pick a reasonable k

Probability of Correctness

We made a claim earlier that if a number is prime, then the test will claim that the given number is prime with 100% probability. This claim is proven below

Proof a prime number is always accepted with probability = 1

If the prime number is even and 2, then we automatically accept it in the first step itself.

Else the prime number is odd.

In step 3a, we make use of Fermat's Little Theorem that states that $a^{p-1} \equiv 1 \mod p$, if p is prime. Hence, if the given input is prime, we won't ever reject it on this step as it will always satisfy this condition.

In step 3d, we need to prove that the first number we come across from right to left that's not 1 has to be -1 if the given number is prime.

Proof By Contradiction:

We can notice that the sequence that we calculate is just $a_i^{s \cdot 2^0}$ squared repeatedly. Hence an element in the sequence is just the square of the previous term (except for the first term).

Take the first term that is not 1 as b. We'll formally state that $b \not\equiv 1 \mod p$ trivially, and make the assumption $b \not\equiv -1 \mod p$ to arrive at a contradiction later.

Since b is the first term that is not equal to 1, we know the next term is.

Hence $b^2 \equiv 1 \mod p \implies b^2 - 1 \equiv 0 \mod p$. Any number that is equivalent to $0 \mod p$ implies that the number is divisible by p.

We can factorize $b^2 - 1$ as

$$b^2 - 1 \equiv (b-1)(b+1) \equiv 0 \mod p$$

Since $(b-1)(b+1) \equiv 0 \mod p$, this implies that at least one of the two, $(b-1)$ or $(b+1)$ are divisible by p, as p is prime.

If $(b-1)$ is divisible by p, it implies that $(b-1) \equiv 0 \mod p$. Hence, $b \equiv 1 \mod p$. However, this goes against our earlier assumption that $b \not\equiv 1 \mod p$.

If $(b+1)$ is divisible by p, it implies that $(b+1) \equiv 0 \mod p$. Hence, $b \equiv -1 \mod p$. However, this goes against our other assumption that $b \equiv -1 \mod p$

Hence, we arrive at a contradiction. Hence, if $b \not\equiv 1 \mod p$, then $b \equiv -1 \mod p$ when p is an odd prime number.

However, if a number is an odd composite number, than there is a chance that the number may or may not be correctly identified as composite. We prove below that with one run, the probability that it will be identified as composite is $\geq 1/2$.

Proof a Odd Composite Number will be rejected with Probability $\geq \frac{1}{2}$

There are two nontrivial witnesses, 1 and -1. There are two types of witnesses as well, those which generate a sequence of only 1s, and those which have some numbers which aren't 1. -1 falls into the second kind.

We will find the non-witness of the second kind that generates a sequence with the -1 coming at the rightmost position possible. We consider that number as h, and the position of the -1 at position j, assuming 0 based indexing.

Therefore $h^{s\cdot 2^j} \equiv -1 \mod p$.

Since p is composite, it must either be a power of a prime, or it can be represented as a product of two numbers that are relatively prime, q, r.

Using the Chinese Remainder Theorem, we know there exists a number t such that

$$t \equiv h \mod q t \equiv 1 \mod r$$

From this we can state that

$$t^{s\cdot 2^j} \equiv -1 \mod q t^{s\cdot 2^j} \equiv 1 \mod r$$

We know that $h^{s\cdot 2^j}$ can be stated in the form $a(qr) - 1$. Therefore, we can restate it as $ar(q) - 1$, hence $h^{s\cdot 2^j} \equiv -1 \mod q$ as well. Since $t \equiv h \mod q$, we can clearly conclude that $t^{s\cdot 2^j} \equiv -1 \mod q$.

The second equivalence is trivial as $t \equiv 1 \mod r$, hence raising it to any power will give 1.

Now, we'll make the claim that t is a witness.

If $t^{s\cdot 2^j} \equiv 1 \mod p$, Then we can write $t^{s\cdot 2^j} = kp + 1 = (kr)q + 1$. This implies $t^{s\cdot 2^j} \equiv 1 \mod q$, which is incorrect. Hence $t^{s\cdot 2^j} \not\equiv 1 \mod p$

If $t^{s\cdot 2^j} \equiv -1 \mod p$, Then we can write $t^{s\cdot 2^j} = kp - 1 = (kq)r - 1$. This implies that $t^{s\cdot 2^j} \equiv -1 \mod r$, which is incorrect.

Hence $t^{s\cdot 2^j} \not\equiv \pm 1 \mod p$.

However, since $t^{s\cdot 2^{j+1}} \equiv 1 \mod q$, and $t^{s\cdot 2^{j+1}} \equiv 1 \mod r$, this implies that $t^{s\cdot 2^{j+1}} - 1$ is divisible by both q and r. Since q and r are relatively prime, we know that $t^{s\cdot 2^{j+1}} \equiv 1 \mod p$

Since $t^{s\cdot 2^j} \not\equiv \pm 1 \mod p$ and $t^{s\cdot 2^{j+1}} \equiv 1 \mod p$, this implies that t is a witness.

Mapping each non-witness to a unique witness

We can prove that $dt \mod p$ is a unique witness for each unique non-witness d.

Since d is a non-witness, it implies that $d^{s\cdot 2^j} \equiv \pm 1 \mod p$. However, $t^{s\cdot 2^j} \not\equiv \pm 1 \mod p$ as t is a witness.

Hence, $(dt)^{s\cdot 2^j} \equiv d^{s\cdot 2^j} \cdot t^{s\cdot 2^j} \equiv \pm 1 \cdot t^{s\cdot 2^j} \not\equiv \pm 1 \mod p$. Hence dt is a witness as well

Proving uniqueness of witnesses. i.e $d_1 t \equiv d_2 t \mod p \implies d_1 \equiv d_2 \mod p$

If $d_1 t \equiv d_2 t \mod p$, then

$$d_1 t\cdot t^{s\cdot 2^{j+1}-1} \equiv d_2 t\cdot t^{s\cdot 2^{j+1}-1} \mod pd_1 t^{s\cdot 2^{j+1}} \equiv d_2 t^{s\cdot 2^{j+1}} \mod pd_1\cdot 1 \equiv d_2\cdot 1 \mod p$$

Hence we've proven that the witnesses are unique. Hence, we've proven that $|\text{Witnesses}| \geq |\text{Non-witnesses}|$. Hence, the probability of an odd composite number being rejected is $1/2$.

Hence, upon running the Miller-Rabin primality test with k numbers, we get an accuracy of $\frac{2^k-1}{2^k}$ of correctly identifying if the number is prime or not.

Hence, when working with the test in cryptography, the designers pick a k to balance between the speed of the algorithm, and the accuracy of it as well to suit their needs, hence providing a very fast algorithm, providing great security with very less probability of failure.

Intro to Local Search

There are many problems in this world which are computationally intractable if we wish to get the deterministic answer to it. These types are much more common than we think in the real world, and we cannot afford to solve these problems slowly, and we must find a workaround to this issue.

Fortunately, for a many of these computationally intractable problems, finding a *good enough* answer suffices as well, instead of finding the perfect answer. Therefore, we can come up with algorithms which might not always give the *perfect answer*, but can give a *good answer,* with much better time complexity.

Now, we'll attempt to define Local Search more formally, and give a generalized working of the algorithm.

Definition of Local Search

Define $X =$ set of candidate solutions to the problem

Neighborhood

Since we have a set of solutions X, we need to define the concept of neighborhoods, and define for every $x \in X$, what are its neighbourhoods $y \in X$.

As we saw in the Maximum Cut problem, we had defined a neighbor as

$$x, y \text{ are neighbors} \equiv \text{ their configuration differs by position of 1 vertex}$$

Therefore, a **neighbor** is essentially a solution that coincides with the current solution in the most ways possible while being distinct.

Defining a Generalized Local Search algorithm

1. We first pick an arbitrary initial solution, x.
2. We iterate through all the neighborhoods of the current solution, and if we find a solution y such that y is superior to the current solution, we assign y to x.
3. We terminate the search once we arrive at a solution which has no superior neighbors, and return the solution x as our answer. This solution is known as a local optimum.

Picking the initial solution for the Local Search algorithm

There are mainly two methods of starting off with a local search algorithm

1. **Starting off with local search -** This strategy is used when we have no clue on how to start off with a solution, and are fully dependent on the local search algorithm to give some solution. There is a good probability of picking a starting point which leads to giving a very poor answer, which is why we run several independent trials with different starting points to reduce the possibility of such an issue occurring.
2. **Using a greedy strategy or heuristic -** There are often many times where we have a heuristic algorithm that could help us start off with a decent initial solution. An example of this is the nearest neighbor heuristic in the Travelling Salesman Problem. After running the heuristic, we apply local search to improve upon the solution we get.

Choosing a neighbor among several superior neighbors

There will be many times in the execution of a local search algorithm that we have multiple superior neighbors to switch to. There are mainly 3 options on choosing what to do.

1. **Choosing the neighbor at random -** One may feel that if they pick a determined neighbor, their algorithm could be susceptible to input cases which are specially designed to break the algorithm designed. By introducing randomness, the probability of this is reduced.
2. **Biggest Improvement -** ****Another popular strategy is choosing to be greedy by going to the neighbor which gives the biggest improvement.
3. **Complex Heuristic -** One may feel that the biggest improvement principle may not be ideal, and that some other greedy intuitive strategy could give a better answer. Usually, complex heuristics to decide the next neighbor are application specific, based on the use case, and the nature of the input and data you have to work with.

Running Time

Say that we have a problem, with a finite optimum answer. We can be sure that the local search algorithm will terminate eventually, if we assume that every time we make a switch, we make an improvement in the score.

However, we cannot say that local search algorithms terminate quickly, although it might in some cases. Local search algorithms are "anytime" algorithms, hence in case you don't want to wait for too long, or you want an answer in a fixed amount of time, you can instead put a bound on the number of local search iterations that can be made, or a bound on the running time, after which you can just ask for the current solution that is held, even though it might not be the local optimum.

Maximum Cut Problem

In the Maximum Cut problem, we are given an undirected graph $G(V, E)$. The motive of the maximum cut problem is to find a way to partition the nodes in the graph G such that the number of edges that cross this partition is maximum.

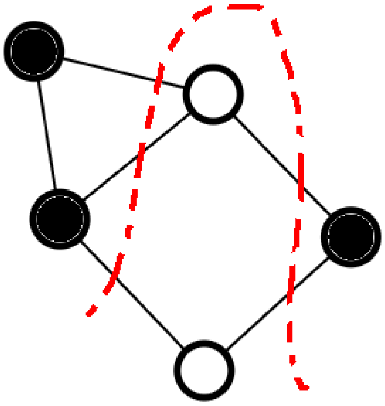


Figure 1: Partitioning the given graph into two sets with max cut

Partitioning the given graph into two sets with max cut

Unfortunately, the Maximum Cut problem is NP-Complete, hence it's computationally intractable to find the exact answer in most cases.

However, there are special cases of graphs for which this problem may be must easier to compute.

Bipartite Graphs

One of the clearest examples of types of graphs for which the maximum cut problem is much simpler computationally is the class of bipartite graphs. The definition of bipartite graphs is that we can create 2 independent sets of nodes, U and V such that every edge in the graph connects a node in U to V. Hence, if we are given a bipartite graph $G(V, E)$, then the answer to the maximum cut problem is $|E|$.

Checking if a graph is bipartite can be done using a simple Depth First Search, in an attempt to color the nodes with 2 colors such that no 2 neighbors have the same color. The time complexity of this check is equal to the time complexity of the depth first search, which is $O(|V| + |E|)$.

Example

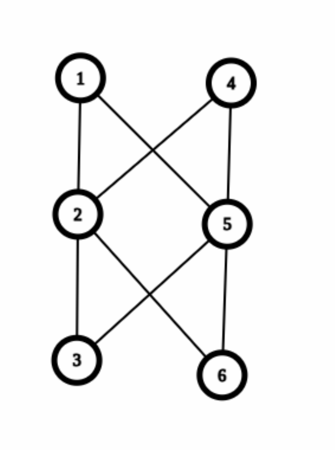


Figure 2: Example of a bipartite graph

Example of a bipartite graph

We are given a graph G with 6 vertices, and 8 edges. By applying a bipartite check which takes linear time, we observe that the graph is bipartite. Hence, we can rearrange the graph to partition the nodes into 2 independent sets such that every edge crosses the partition.

A maximum cut on the given graph

Hence, the answer to this problem is 8 edges, which is the value of $|E|$ in his graph.

Hence, if the given graph is bipartite, the time complexity goes from NP-complete to a linear solution. A very significant improvement.

However, we must still find a way to provide a good answer to the maximum cut problem for graphs which don't have any special properties to reduce the time complexity.

One of the methods that can be used to solve the maximum cut problem using optimization is the Local Search algorithm.

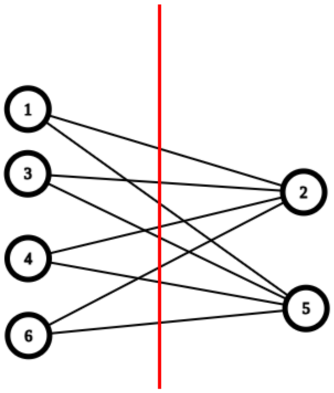


Figure 3: A maximum cut on the given graph

Using Local Search

Before we formally go over Local search, it'll be more fruitful if we go over an example of it in the maximum cut problem

We start off with an arbitrary partitioning of the nodes

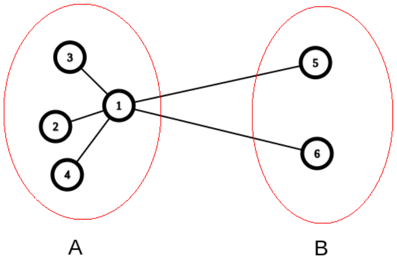


Figure 4: Arbitrary initial cut for the given graph

Arbitrary initial cut for the given graph

We can clearly see that this partitioning isn't ideal, however it doesn't matter as we just have to start the algorithm with some initial configuration, after which we continue to improve upon that configuration.

Now, we'll define some notation in this algorithm

$$c_v(A, B) = \text{Number of edges incident on } v \text{ that cross } (A, B)$$

$$d_v(A,B) = \text{Number of edges incident on } v \text{ that don't cross } (A,B)$$

If the degree of the node v is x, we can see that $c_v + d_v = x$.

The strategy we'll use, is that we'll go over every vertex, and compare their c_i and d_i values.

If we see that for a certain vertex v, $d_v > c_v$, then we can improve upon the current answer.

Why is this true though? It's actually pretty easy to see why this works visually

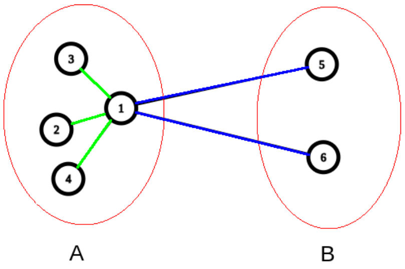


Figure 5: Crossing edges marked blue, non-crossing edges marked green

Crossing edges marked blue, non-crossing edges marked green

We can see that for vertex 1, $c_1 = 2$, $d_1 = 3$. In this configuration therefore, if we switch the partition to which the vertex 1 belongs to, we'll get an improvement of 1 more edge to the answer.

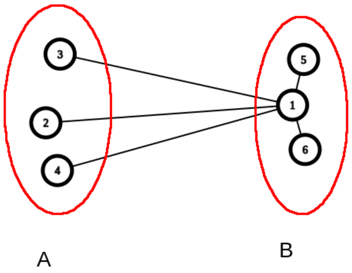


Figure 6: New configuration after switching partition of vertex 1

New configuration after switching partition of vertex 1

By switching the partition of vertex 1, we've essentially switched the values of c_1 and d_1, as the edges which didn't cross the partition, cross them now, and vice versa.

However, we still haven't reached the end of the Local Search algorithm, as there are still vertices in the graph whose $d_v > c_v$.

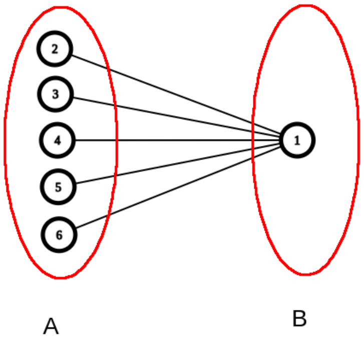


Figure 7: Final configuration, which ends up being the optimal answer as well

Final configuration, which ends up being the optimal answer as well

We can move the vertices 5 and 6 to partition A to end up with an answer 5, which ends up being the best answer as well for the given graph. Note that however, the Local Search algorithm doesn't necessarily give the best answer.

Running Time

We know that the algorithm will continue to run until it cannot find a single vertex for which $d_v > c_v$, after which it will terminate.

We know that in a graph with V vertices, there can be at max $\binom{V}{2}$ edges, assuming no multi-edges and self-loops. hence, in the worst case, we'd start off with a cut that gives 0 as the answer, and with every switch of vertices between partitions, we get an improvement of 1 edge, hence taking $\binom{V}{2}$ switches in total.

Each switch takes $O(V)$ time, as we have to iterate through all the vertices to find a vertex that satisfies the inequality, as well as updating the c_v, d_v values of all the nodes that connect to the node we're switching.

Hence, the time complexity turns out to be

$$O(V^2) \times O(V) = O(V^3)$$

Which is a major improvement over exponential running time, although it's not guaranteed to give the best answer.

Why does it not work optimally?

It might seem enticing to claim that this local search algorithm can do pretty well, and it's hard to see where it might fail. However, getting sub-optimal answers from the algorithm is more common than you think. In fact, even if you run the local search algorithm on bipartite graphs, it still might not give the optimal maximum cut! We'll show how with an example

Say we're given a bipartite graph, and we choose an arbitrary cut. This arbitrary cut may end up giving us maximum cut, and we could just terminate. In the figure given, we can see that the arbitrary cut gives us 4, which is the optimal answer for this graph.

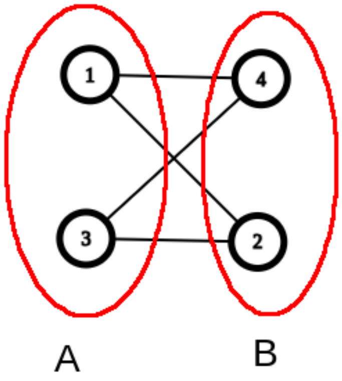


Figure 8: Sometimes we may get lucky with our arbitrary cut

Sometimes we may get lucky with our arbitrary cut

However, this may not be the case every time, and we may end up with a cut that prevents us from improving the answer any further with the local search algorithm

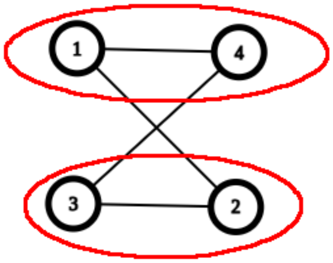


Figure 9: And sometimes, we might get unlucky

And sometimes, we might get unlucky

Here, we can see that the cut we have is clearly not the optimal. However, for every vertex in the graph $c_i = d_i$, hence we cannot switch any vertices to improve our answer. Hence, if we chose an arbitrary cut like this, we'd end up with the answer 2, instead of the optimal answer 4.

Expected Value of an arbitrary cut

One may wonder what is the expected value that we get upon using a completely random cut. This is actually quite simple to analyze.

Instead of looking from the perspective of vertices, we'll analyze the expected value by looking at the edges.

Given an edge, there are 4 cases of where the 2 vertices at the endpoints of the edge might be.

1. Vertex 1 and 2 are both in partition A
2. Vertex 1 is in partition A, while Vertex 2 is in partition B
3. Vertex 1 is in partition B, while Vertex 2 is in partition A
4. Vertex 1 and 2 are both in partition B

We can see that cases 1 and 4 won't contribute to the final answer, while cases 2 and 3 will.

Each case has a 25% chance of occurring, hence the expected value that this edge will contribute to the answer is

$$E(\text{value contributed by edge e}) = \frac{1}{4}\cdot(0) + \frac{1}{4}\cdot(1) + \frac{1}{4}\cdot(1) + \frac{1}{4}\cdot(0) = 0.5$$

Using Linearity of Expectation, the total value contributed by all the edges will be

$$|E|\cdot E(\text{value contributed by arbitrary edge e}) = |E|\cdot 0.5 = \frac{|E|}{2}$$

Hence, the expected value of an arbitrary cut is $|E|/2$.

Performance Guarantee

The Local search algorithm provides a lower bound for it's performance, and states that it'll output a cut in which the number of crossing edges is at least 50% of the optimal answer of the given graph. (Which ends up being $\geq 50\%$ of $|E|$)

Proof

Say we have a locally optimum cut (A, B), i.e. there are no more vertices to switch.

This implies that for every vertex v, $d_v(A, B) \leq c_v(A, B)$

Hence, if we sum up all the values, we get

$$\sum_{v \in V} d_v(A, B) \leq \sum_{v \in V} c_v(A, B)$$

We are double counting every edge in this inequality, therefore, the inequality can be re-written as

$$2 \cdot |\text{non-crossing edges}| \leq 2 \cdot |\text{crossing edges}|$$

Adding $2 \cdot |\text{crossing edges}|$ to both sides,

$$2 \cdot |E| \leq 4 \cdot |\text{crossing edges}|$$

Which gives us

$$|\text{crossing edges}| \geq \frac{|E|}{2}$$

Hence, we can see that on average, the random cut will give an answer $|E|/2$, while the local search algorithm will guarantee to give an answer $\geq |E|/2$

Travelling Salesman Problem

We are going to deal with a somewhat specific version of the Travelling Salesman Problem.

Problem

We are given an input of a complete, undirected weighted graph $G(V, E)$, which means there exists an undirected edge with some weight w_i between every pair of edges.

What we need to find is some tour T. A **tour** is a path which visits every vertex in the graph exactly once, and returns to the starting vertex. We need to minimize the sum of the weights of the edges taken in such a tour.

This problem is **NP-hard**. There exists a solution which takes $O(n \cdot 2^n)$ time, using dynamic programming, but this solution can be quite slow as soon as n takes a relatively large value.

Therefore, we must make use of optimization algorithms to find a *good enough* solution which runs efficiently. We'll go over a method that uses local search to come up with a good answer.

We'll first make use of a heuristic to end up with a decent solution, upon which we'll apply the local search algorithm to make some improvements on top of that.

The heuristic is quite simple, and it involves just greedily picking the closest neighbor to a vertex that we haven't visited

Nearest neighbor Heuristic

1. Pick an arbitrary vertex x
2. Mark the current vertex v as visited (visited$[v]$ = true)
3. Find the set of all vertices that are connected to the current vertex, and are unvisited. Name this set U. If the set U is empty, then go to step 8.
4. Find the vertex $u \in U$ such that $w(v,u)$ is minimal.
5. Add the weight to the total cost of the tour total cost$+= w(v,u)$
6. Go to vertex u
7. Go to step 2
8. Go back to starting vertex x, by adding weight $w(v,u)$ to total cost.
9. Terminate program

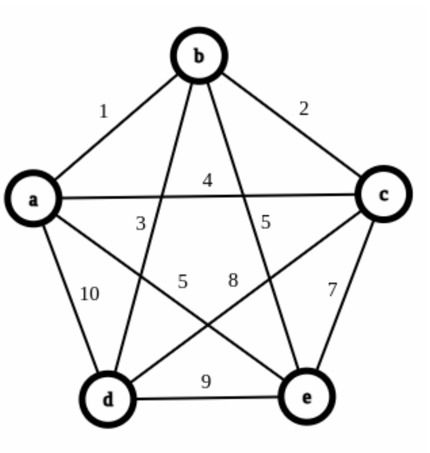


Figure 10: Using this graph as an example for TSP

Using this graph as an example for TSP

Say we need to run this heuristic on the given graph. Let's arbitrarily pick the starting vertex as a.

Then, we tour over all the vertices in the graph as follows

1. Starting vertex a
2. Go to vertex b (total cost = 1)
3. Go to vertex c (total cost = 3)
4. Go to vertex e (total cost = 10)
5. Go to vertex d (total cost = 19)
6. Go to vertex a (total cost = 29)

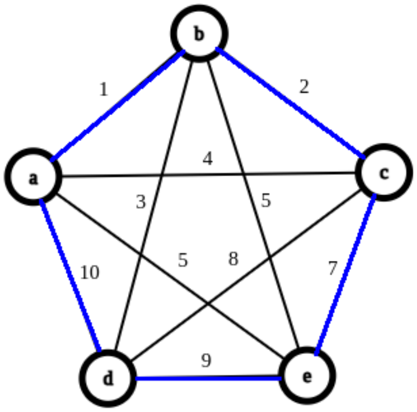


Figure 11: The tour after using the heuristic algorithm

The tour after using the heuristic algorithm

Hence we end up with a total cost of 29. We can observe that there is a tour that does end up with a lower cost, by going the route $a \Rightarrow c \Rightarrow b \Rightarrow d \Rightarrow e \Rightarrow a$, giving us a cost of 23

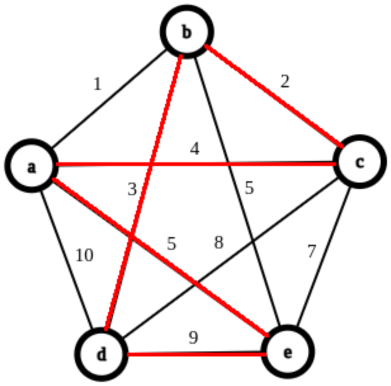


Figure 12: An optimal solution

An optimal solution

Hence, we can see why the heuristic doesn't give us the optimal answer, but it did end up giving us a pretty good solution.

However, a very glaring issue with the heuristic is the tunnel visioning it performs, looking at only the next step, which could cost dearly at the very end

Say instead of the cost of the edge between a, d was 10000 instead of 10, and we still used the same greedy heuristic. We'd end up taking the, same path as before, and as a result of that, we'd end up with a cost of 10019, instead of the optimal 23.

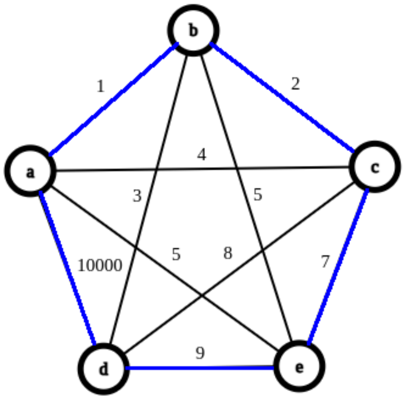


Figure 13: The heuristic has glaring issues

The heuristic has glaring issues

Hence, we can clearly see that while the greedy heuristic can provide a pretty good solution in many cases, using the heuristic can lead to disastrous answers in other cases. Therefore, we must couple this heuristic with a method to improve upon a solution we reach to avoid very poor answers. Hence, using the tour we reach by using the heuristic, we apply local search to improve upon the solution.

Local Search

Before we try to use local search on the problem at hand, we must first define what is a neighbor.

Obviously, we want a neighbor of a tour, to be a tour that shares the maximum amount of edges.

If we are given a graph with n vertices, then the maximum amount of edges that two distinct tours can share is $n - 2$.

Now you may be wondering, why not $n - 1$? If you select $n - 1$ edges of a tour, then the last edge that has to be chosen is uniquely determined, which is using the edge between start and end vertices. Hence, if we had two distinct tours with $n-1$ common edges, the n^{th} edge would be common as well.

Therefore, if we want to switch to a neighbor tour, then we must switch 2 of the edges of the current tour and add edges so that we get a new tour.

Switching to a neighbor using 2-change

1. Say we have a tour T. We remove 2 edges, (u, v), (w, x), such that the two edges do not share a common endpoint.
2. Insert two new edges (u, w), (v, x) or (u, x), (v, w) such that the solution we get is a connected tour.

The new cost of the tour will be

$$\text{new cost} = \text{original cost} - c_{(u,v)} - c_{(w,x)} + c_a + c_b$$

Where a, b is (u, w), (v, x) or (u, x), (v, w) based on which pair gives a connected tour

Therefore, we can define the decrease in cost as

$$(c_{(u,v)} + c_{(w,x)}) - (c_a + c_b)$$

Therefore, the algorithm we use to solve TSP using local search is:

- Given the graph, we pick an arbitrary starting node and use the nearest neighbor heuristic
- After the completion of the heuristic, we start applying local search by switching two edges whenever it'll give us a decrease in cost, and continue to do this step until we don't have any more neighbors which have a better cost.
- Return the tour T we come up with

Example

Making use of the earlier 5 node complete graph example, we'll show how local search will improve upon the solution we arrive at using the heuristic.

1. Apply the heuristic to get an initial configuration with cost 29.
2. We see that deleting edges (a, b) and (d, e) to switch them with (a, e) and (b, d) reduces the cost to 27.
3. We delete (b, c) and (a, d), to switch them with (d, c) and (a, b), reducing the cost to 24.
4. There are no more 2-change neighbors which provide a decrease in cost, hence we terminate with cost 24 as the answer.

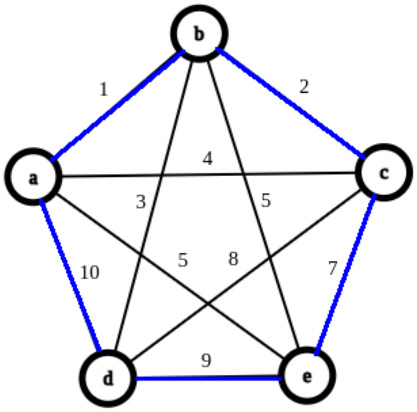


Figure 14: Tour after heuristic

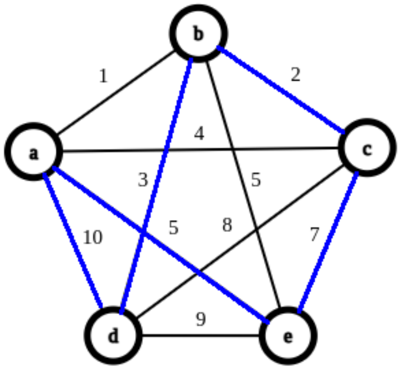


Figure 15: After one iteration of local search

Tour after heuristic

After one iteration of local search

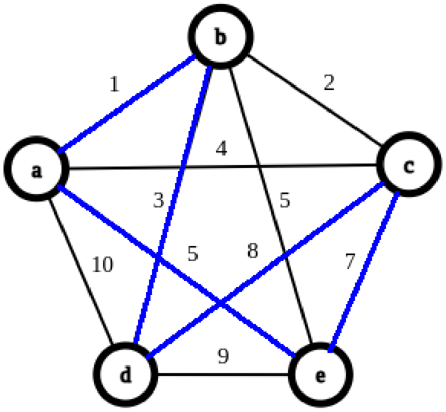


Figure 16: After the second iteration of local search

After the second iteration of local search

Running Time

The number of ways we can choose a 2-change is $O(n^2)$, hence each iteration of the local search algorithm will take $O(n^2)$ time.

The number of iterations that local search does before halting can be exponential, however it's guaranteed to halt, as with each iteration, the cost of the tour has to be strictly lesser than the previous tour, else the local search wouldn't have made the change.

However, local search very rarely ends up taking exponential number of iterations, and even if it does, the local search algorithm for TSP has the convenient property of being an “anytime algorithm”.

What's an anytime algorithm

An anytime algorithm is an algorithm which is allowed to terminate at any time. In local search, what we are attempting to do is improve upon a solution that we already have. Therefore, we could just terminate the local search after a fixed number of iterations, or a fixed amount of time in case you need the algorithm to terminate within a certain deadline.

Bound on solution quality

Unfortunately, there are no hard bounds on the quality of the solution you receive using this algorithm. However, it's been observed in practice that the solution is usually within 20% of the actual solution, and based on the application can get better or worse in terms of quality.

Graphs

Graphs are probably one of my favorite ideas in computer science. Graphs are often some of the greatest tools in modeling real-world connections and problems into computational problems. Many real-world problems like scheduling, routing, recommendations, etc. can all be modeled as graph problems. But before we tout the awesomeness of graphs, we should perhaps begin by explaining what a graph is.

What is a graph?

To define them formally, a graph can be defined by a set of vertices and a set of edges, where each edge in this set links two of the vertices of the graph to each other. However, we tend to lose a lot of the intuition behind modeling problems using graphs when restricting ourselves to the formal definition.

We take a problem and model the "object" in the problem as the vertices of the graph. Now, the relation between objects is modeled via an edge. This gives us a powerful representation to represent data to solve many computational problems.

After modeling the problem

Once the problem has been modeled well in terms of a graph, it now makes the problem open to be attacked by a plethora of algorithms that we know. BFS/DFS, Dijkstra, Bridge finding, etc. are names that come to find. But graphs are not a concept that must be used alone, below I will go over one of my favorite examples of an elegant optimization problem that can be solved via an elegant mix of data structures, DP, and graphs.

The problem is the B problem in the 2016 ACM-NCPC contest.

> The autocorrect feature on Jenny's phone works like this: the phone has an internal dictionary of words sorted by their frequency in the English language. Whenever a word is being typed, autocorrect suggests the most common word (if any) starting with all the letters typed so far. By pressing tab, the word being typed is completed with the autocorrect suggestion. Autocorrect can only be used after the first character of a word has been typed – it is not possible to press tab before having typed anything. If no dictionary word starts with the letters typed so far, pressing tab has no effect. Jenny has recently noticed that it is sometimes possible to use autocorrect to her advantage even when it is not suggesting the correct word, by deleting the end of the autocorrected word. For instance, to type

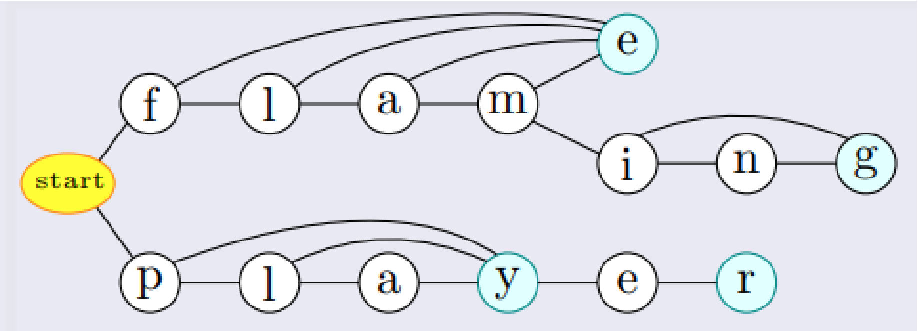
> the word "autocorrelation", Jenny starts typing "aut", which then autocorrects to "autocorrect" (because it is such a common word these days!) when pressing tab. By deleting the last two characters ("ct") and then typing the six letters "lation", the whole word can be typed using only 3 ("aut") + 1 (tab) + 2 (backspace twice) + 6 ("lation") = 12 keystrokes, 3 fewer than typing "autocorrelation" without using autocorrect. Given the dictionary, on the phone, and the words, Jenny wants to type, output the minimum number of keystrokes required to type each word. The only keys Jenny can use are the letter keys, tab, and backspace.

The problem attempts to model a very real-world idea/implementation of autocorrect and asks us to optimize the number of steps required to type any word given the configuration of the autocorrect algorithm at any point in time.

Our solution for this problem is a combination of DP, Graphs and string data structures. The string data structure we use here is called a Trie. Since this book is on algorithms and not data structures per say, we will not cover what a Trie is. The Wikipedia article does a great job explaining the Trie https://en.wikipedia.org/wiki/Trie

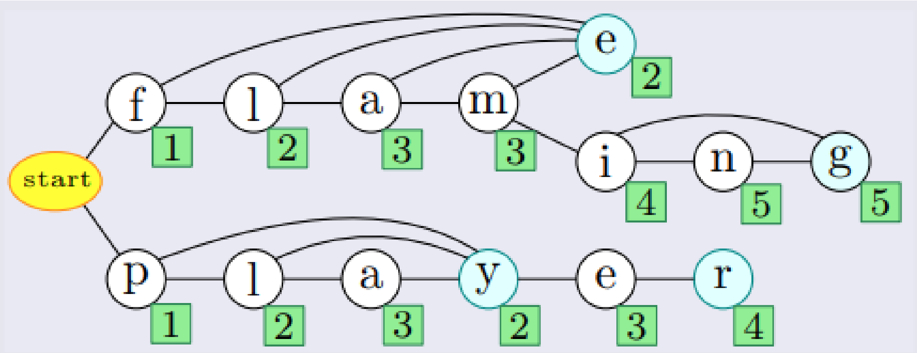
Now, to problem here is sometimes we might need to do **MULTIPLE** auto correct tab jumps to optimally type the word. Simply iterating over the Trie and performing these jumps recursively is not fast enough. It runs in O(m*n) which is too slow. However, we can model the **Trie** as a graph and perform BFS DP on it.

The idea is pretty simple, we put "jump" edges to all the letters we can reach by tab completing. This signifies that each of the last letters of each word in the trie, can be reached by **one** move from any of the letters behind it. We took the Trie data structure and added this new edge to make it the **right** graph model to work with. Once this is done, the rest is simple. Simply run a BFS or any other fast shortest path finding algorithm from the root node. It will compute the optimal distances to each letter ending. We can store these values in our DP table. The recurrence is reached in the order of the BFS.



Note: The edges from the previous letters to the ending letter (signifying the tab) is directed. The rest of the edges are undirected as they be reached by both backspace and character type operations.

Once the BFS is run, our graph looks like this:



After this, all we have to do is run the usual Trie search operation on the dictionary and the answer is the DP value stored at the position we are on the Trie. If the entire word cannot be constructed with the Trie, we search the Trie for the maximum prefix we can type, get the DP value stored at this location and then type the rest of the characters 1 by 1.

And that's it! A problem that didn't seem to have much with graphs was modelled into graphs and then we could use concepts like shortest paths and DP to solve the optimization problem optimally.

The total time complexity is now reduced to linear $O(n)$.

Network-Flow algorithms

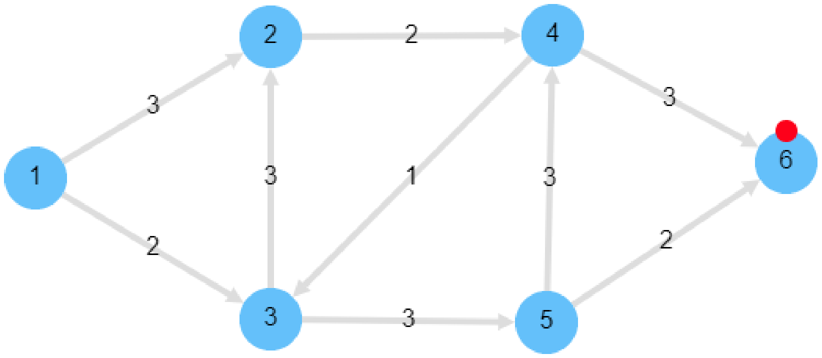
Let's learn another really cool tool that can be used to solve optimization problems, network flows!

What is the network flow graph?

A network flow graph $G = \langle V, E\rangle$ is nothing but a directed graph, with 2 distinctive features.

1. It has 2 distinct vertices S and T marked. S is the **source** vertex and T is the **sink** vertex. These vertices are distinct.
2. Every edge $e \in E$ has some capacity c_i associated with it. It is implicitly assumed that $\forall e \in E, c_i = 0$.

An example of one such graph is given below



Here, $S = 1$ and $T = 6$. We will use this same example when discussing further ideas.

The problem

The problem that network flow attempts to solve is pretty simple. It asks the questions, *"Given an infinite amount of"flow" at source S, what is the maximum amount of "flow" you can push through the network at any point of time and reach sink T?"*

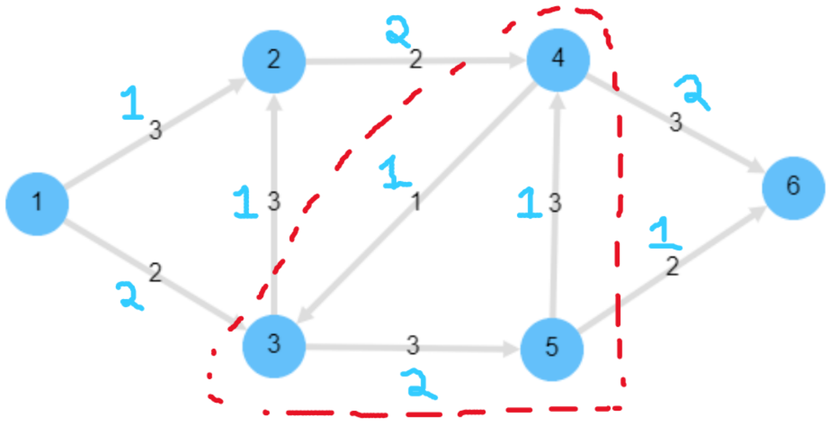
An intuitive way to think about this is to pretend that the source S is an infinite source of water and that the capacities on each edge are sort of like the maximum amount of water that can flow through each of the "pipes" or edges. If we think of edges in terms of pipes, the question basically asks how much water we can push through the pipes so that the maximum amount of water reaches sink T per unit time.

Why is this helpful? Think about traffic scheduling, for example, we could replace water with traffic and the problem would be similar to scheduling traffic through a busy set of streets. Replace it with goods flowing in a warehouse system and we begin to see how powerful this model of the optimization problem is.

To define this more formally, the only primary constraints are as follows:

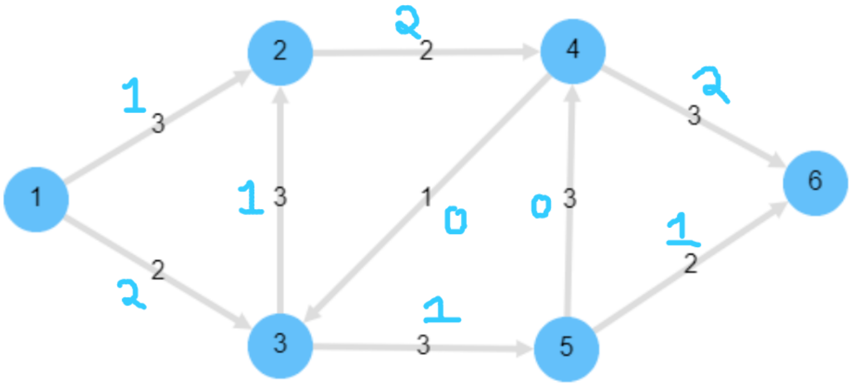
1. The flow through any edge **must** be $\leq$ the capacity of that edge.
2. The flow entering and leaving any given vertex (except S or T) must be the same. (Pretty similar to Kirchhoff's current laws.)

Here is an example of a valid network flow assignment:

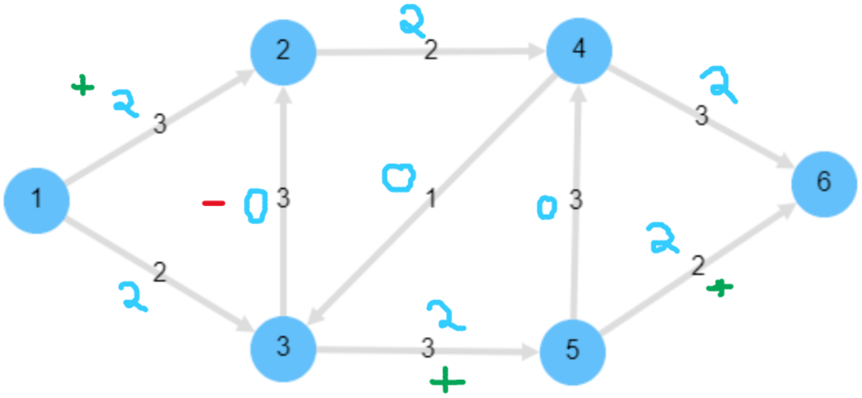


We can manually go over every vertex and ensure that the two constraints are obeyed everywhere. Further, notice that the flow of this network = 3. (Just sum up the flow going to T, i.e., the edges incident on T)

An interesting observation is that we appear to have "cyclic flow" within our graph with this particular assignment of flow. Eliminating this does not change the total flow going to T, so this is pretty much the same assignment without that cyclic flow within the network:



But what about the max flow assignment for this network? is 3 the maximum flow we can achieve? Or can we do better? After a bit of fiddling around, we can notice that we can do better by pushing more flow on the bottom half of this network instead of sending 1 flow up to the top from node 3. Fixing this ends up giving this network:



It can be proven that we cannot do better than this for this particular network. The max flow of this network is 4.

Hopefully, the above examples have managed to convey the true difficulty that flow algorithms face. Solving network flow is **not** easy, primarily because from any given state, the optimal state might not be reached by just monotonically increasing flow through edges. We might have to reduce the flow through some edges to increase the flow in others. Changing the flow amount through any one edge ends up affecting the entire network. So we need to find ways to iteratively increase the flow in our network, **BUT** it is not a monotonic increase. So we must sometimes backtrack and reduce flow in some edges. But perhaps by focusing on monotonically increasing the *max flow* of our network, we might be able to figure out a proper algorithm that incorporates this backtracking. This is the primary goal we keep in mind when trying to solve max flow.

Defining the problem formally

Some useful notation

For the remainder of this article, we will use "implicit summation" notation. All sets will be named by capital letters, and whenever we use sets in the place of elements like for example $f(s, V)$, this means the summation of flow $\sum_{v \in V} f(s, v)$. We use this notation to simplify the math we will be writing.

Formal definition

Flow: We define the *flow* of a network G as a function $f : V \times V \rightarrow R$ satisfying the following 3 constraints,

1. $\forall u, v \in V, f(u, v) \leq c(u, v)$. That is, flow through any edge must be less than the capacity of that edge.
2. $\forall u \in V - \{s, t\}, f(u, V) \implies \sum_{v \in V} f(u, v) = 0$. That is, flow entering and exiting every node except source and sink is 0. It is conserved.

3. $\forall u, v \in V, f(u,v) = -f(u,v)$. This is not the flow between two vertices, given any two vertices on the network u and v, the flow going from one vertex u to v should be the negation of the flow from v to u. This property is called *skew-symmetry.*

Defining flow

Let us denote the value of the flow through a network by $|f|$. Then we define this quantity as

$$|f| = f(s, V)$$

Intuitively, this is essentially all the flow (sum) that is going from the source node to every other vertex on the graph. It is important to note that the summation is not of all positive terms, if there is flow going from some vertex v to s, then this term would be negative (skew symmetry).

Using this, it is possible to prove that $|f| = f(s,V) = f(V,t)$. That is, it is all the flow going to vertex t and is more “intuitive” to understand as the definition of flow. But before we can prove this, let’s go over some key properties of flow-networks which we can derive from the constraints.

Properties:

1. $f(X, X) = 0, X \subset V$, this is derivable from skew-symmetry.
2. $f(X, Y) = -f(Y, X), X, Y \subset V$. Direct consequence of skew symmetry.
3. $f(X \cup Y, Z) = f(X, Z) + f(Y, Z)$ if $X \cup Y = \phi$. If the intersection of X and $Y$is null, then we can safely add the two flows separately as there is no risk of double counting.

Now that we know these properties, let’s prove it!

$$|f| = f(s, V)$$

Let’s start from the definition of our flow amount $|f|$. Using property 3, we can transform it to mean

$$f(V, V) = f(s \cup (V - s), V) = f(s, V) + f(V - s, V)$$

$$\implies |f| = f(s, V) = f(V, V) - f(V - s, V)$$

$$\implies |f| = 0 - f(V - s, V)$$

This is intuitively just saying that flow from $s \to V$ is the negative of the flow from other vertices to all vertices. This is because flow within non-source-sink vertices is 0 and they must all flow out the sink. Now, notice that we want to try to prove that $|f| = f(V,t)$. To do this, we will attempt to isolate t from the above equation using the 3rd property again.

$$f(V-s,V) = f(t \cup (V-s-t),V) = f(t,V) + f(V-s-t,V)$$

$$\implies |f| = -f(t,V) - f(V-s-t,V)$$

$$\implies |f| = f(V,t) + 0$$

$$\implies |f| = (V,t)$$

The tricky part here is understanding why $f(V-s-t,V) = 0$. This is because of flow conservation. Flipping it around, we get $f(V,V-s-t)$. By the 2nd constraint imposed on our flow network, this quantity is constrained to be 0 always. Hence we have now proved that

$$|f| = f(s,V) = f(V,t)$$

Ford-Fulkerson

Residual networks

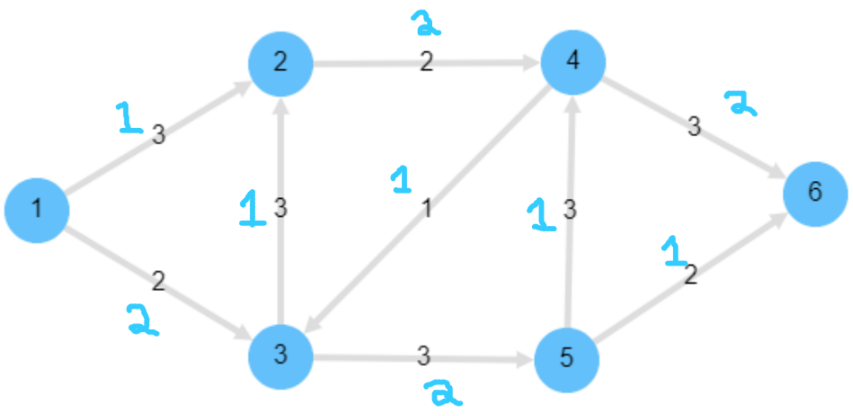
We denote the residual network of a flow network G by $G_R(V_R, E_R)$.

The only constraints on the edges are that all the edges have strictly positive residual capacities. 0 means the edge is deleted. And, if $(u,v) \notin E, c(v,u) = 0, f(v,u) = -f(v,u)$

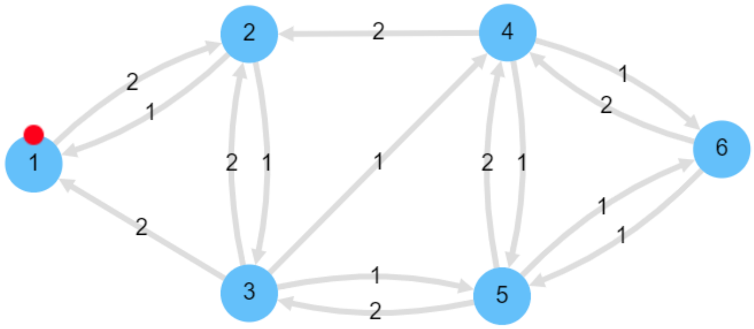
Essentially, $\forall e \in E_r, c_{Re} = c_e - f_e$. The residual edges represent edges that "could" admit more flow if required. Here c_e is the capacity of the edge in the original flow graph and f_e is the flow passing through the edge in the original flow network.

The idea behind these edges becomes more apparent when we actually construct the network.

Consider the old suboptimal max flow network we had.



We'll begin by constructing the residual graph for this network. Remember, for each edge in the network, we add an edge with capacity $c_e - f_e$ as long as this quantity is > 0. And now, to respect the last constraint, we must ensure that we add a back-edge in the opposite direction with value $= f_e$ as long $f_e > 0$. This is the **key** idea behind what the residual network hopes to accomplish. Recall back when said one of the reasons the flow problem was very difficult was because it is very difficult to account for having to *reduce* flow in some edges to increase max flow? This residual network is what helps the algorithm get around this problem. Here is the residual network:



Now, the Ford Fulkerson algorithm becomes extremely simple. It simply says, use any graph traversal algorithm such as BFS or DFS to find *an augmenting path* in this graph, and apply it to the original graph.

We formally define an augmenting path as a path from s_R to t_R in the residual graph. Recall that every edge in the residual graph **must** be a positive value. If such a path is found, then it **must** be possible to increment the value of max flow in the network by **at least** 1. This is because the residual graph is essentially an entire encoding of every possible increase/decrease in flow that we can perform on the original graph. The presence of a path with all edges > 0 implies I can increase flow from s_R to t_R by at least 1.

If this is understood, the Ford Fulkerson algorithm becomes pretty simple.

Pseudocode

```
1. Construct the residual graph for some given flow network $G$
2. While we can find an augmenting path in the residual graph:
    1. Get the min of the edges that constitute this path and increment the
       flow in the original graph by this value along the edges in the residual
       graph. If it is a direct edge, increment by min. If it is a back-edge,
       decrease flow by min.
    2. Reconstruct residual graph.
    3. Repeat. If no more augmenting paths are found, we have achieved
       max flow.
```

Proof

Why does this algorithm work optimally all the time? To prove the correctness of this algorithm, we will first prove the correctness of the Max-flow, Min-cut theorem.

Max-Flow, Min-Cut

The theorem states that the following statements are equivalent.

1. $|f| = c(S, T)$ for some cut (S, T).
2. f is the maximum flow.
3. f admits no augmenting paths.

We will prove this theorem by proving $1 \implies 2 \implies 3 \implies 1$.

Proving $1 \implies 2$

We know that $|f| \leq c(s, t)$ for any cut (s, t). Hence, if $|f| = c(s, t)$ then f must be the maximum flow through this network.

Proving $2 \implies 3$

We can prove this by contradiction. Assume that there existed some augmenting path. Then this would imply that we could increase the max flow by some amount, hence contradicting the fact that f is the maximum flow. Hence f cannot admit any augmenting paths.

Proving $3 \implies 1$

Let us assume that $|f|$ admits no augmenting paths. That means, we have no path from s to t in G_R. We now define two sets $S = \{v \in V : \text{there exists a path in } G_R \text{ from } s \to v\}$. The other set is defined as $T = V - S$. Trivially, $s \in S$ and $t \in T$, as I cannot reach t from s. Therefore, these two sets form a cut (S, T).

Now, we pick two vertices $u \in S$ and $v \in T$. Now, by definition, there is a path from s to u. But no path from $u \to v$. Otherwise $v \in S$, which is false.

Now, $c_R(u, v)$ **must** be zero. $c_R(u, v)$ is by definition always positive. Now, if $c_R(u, v) > gt0$ it would imply that $v \in T$. This is a contradiction. Therefore, $c_R(u, v) = 0$.

Now, we know that $c_R(u, v) = c(u, v) - f(u, v) \implies f(u, v) = c(u, v)$

For our arbitrary choices of $u \in S$ and $v \in T$, we arrive at the conclusion that $f(S, T) = C(S, T)$.

Since $1 \implies 2 \implies 3 \implies 1$, the Min-Cut Max Flow theorem is true.

Proving Ford-Fulkerson

Now, the Ford Fulkerson algorithm terminates when there are no longer any more augmenting paths in G_R. According to the Maxflow MinCut theorem, this is equivalent to our network reaching maximum flow. Hence we have proved the correctness of our algorithm.

Complexity

It is easy to see that for integral capacities and flow constraints, finding an augmenting path implies increasing the value of maximum flow by **at least** one. This means that the algorithm will at least increment flow in network by 1 per iteration. Hence it will terminate and we can bound the complexity to $O((V + E)U)$ where $V + E$ is the complexity of the BFS and U is max flow.

For non-integer capacities, the complexity is unbounded.

This… isn't great. Because our complexity depends on the maxflow of the graph. If we construct a graph such that at each iteration, we have worst case and the algorithm increases flow in the network by only one unit and the capacity on the edges is large, we might end up doing millions of iterations for a small graph.

Edmond-Karp Analysis

Edmond and Karp were the first to put a polynomial bound on this algorithm. They noticed that BFS implementations of Ford Fulkerson's outperformed DFS versions a lot. Upon analyzing these implementations, they were able to put a polynomial bound on the problem. Their were able to reduce it the following bound: $O(VE^2)$. **The coolest part about this is that this is true even for *irrational* capacities!**

The intuition is, that every time we find an augmenting path one of the edges becomes saturated, and the distance from the edge to s will be longer, if it appears later again in an augmenting path. And the length of a simple paths is bounded by V.

Dinics

Dinic's algorithm solves the maximum flow problem in $O(V^2E)$.

More recent research

The asymptotically fastest algorithm found in 2011 runs in $O(VElog_{\frac{E}{VlogV}}V)$ time.

And more recently, Orlins algorithm solves the problem in $O(VE)$ for $E \leq O(V^{\frac{16}{15}-\epsilon})$ while KRT (King, Rao and Tarjan)'s does it in $O(VE)$ for $E > V^{1+\epsilon}$

There's a lot of research going on in this field and we know no proven lower bound for this algorithm. Who knows, we might be able to get even faster! Techniques like push-relabel, with a greedy optimization have managed to get a lower bound of $O(V^3)$. This modification was proposed by Cheriyan and Maheshwari in 1989.

Uses

Flows are a very heavy hammer in our toolkit that we can use to solve bipartite matching! And all kinds of problems where we must pick choices to maximize some flow or any kind of routing problem which requires us to push more flow through a graph network. Baseball / Football elimination is one such example.