Algorithms & Complexity: Lecture 5, P vs NP & Algorithms

Sam Barrett

March 4, 2021

1 History of Computing

In the 1950s to 1960s one of computer science researcher's focuses was on general ways to solve problems.

One such problem was `SAT`, is a given logical formula ϕ satisfiable?

Many of the problems known at this time had polynomial running time. We know that polynomial time is ultimately always lower than exponential, for instance $1.000001^n > n^{10000000000}$ if n is large enough. We can prove this by taking logs of both sides.

During this time polynomial time became the accepted standard of efficiency. It has many *nice* properties including being **closed** under addition, multiplication and composition. I.e. if both p and q are polytime functions: $p(x) + q(x)$ is polytime, $p(x) \times q(x)$ is polytime and $p(q(x))$ is polytime. We define **P** as the class of problems solvable in polynomial time (wrt. the size of the input).

Later, in the 1970s research moved to focus on the problems that no efficient algorithm was known to be able to solve, the set of problems that could not be included in **P**.

Our previous example `SAT` cannot be brute forced in polynomial time. If given N variables in M clauses, we must make 2^N truth assignments, checking each of the M clauses in $O(N)$ time resulting in overall running time in $O(2^N \cdot N \cdot M)$, clearly this is in **EXP**.

Research started to focus on trying to prove that there is no solution in **P** for `SAT`.

In so doing a new class was defined, **NP**. The class of problems where we can verify a potential solution, or certificate, in polynomial time.

How much harder is solving compared with verifying?

Given that we define **P** as the class of problems with solutions in polytime, and **NP** is the class of problems for which we can verify a potential solution in polytime, clearly $\mathbf{P} \subseteq \mathbf{NP}$ as solving can be seen as a very difficult way of verifying.

But what about the opposite direction? Say we can verify potential solutions in n^{12} time, how long would it take us to **solve** the problem? If we can solve this in some polynomial amount of time n^k then we have shown that $\mathbf{P} = \mathbf{NP}$!

2 From SAT to graphs

Definition 1 *(Independent Set) Given an undirected graph $G = (V, E)$ on n vertices, find a set X of maximal size such that no pair of vertices in X form an edge*

We can reduce the independent set problem to SAT:

- Take an instance I of 3-SAT with N variables and M clauses
- Build a graph G on $3M$ vertices as follows:
  - Introduce triangle for each clause
  - Add conflicts to ensure every variable is not both `True` and `False`
- Claim: I is satisfiable iff G has an independent set of size M

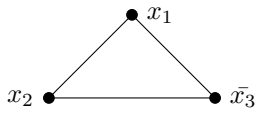
This reduction is in polytime meaning that the Independent set problem is also in **NP**

2.1 Example

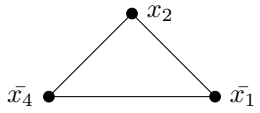
Suppose we have an $I = \underbrace{(x_1 \vee x_2 \vee \bar{x_3})}_{c_1} \wedge \underbrace{(x_2 \vee \bar{x_4} \vee \bar{x_1})}_{c_2} \wedge \underbrace{(x_3 \vee x_1 \vee \bar{x_2})}_{c_3}$

We can convert each clause into a separate triangle:

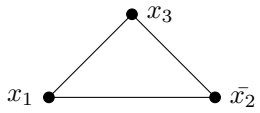
c_1:



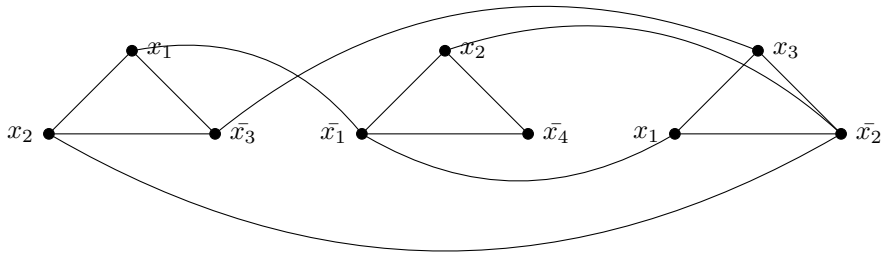
c_2:



c_3



Now we must add our conflicts. We do this by connecting conflicting variables.



3 Algorithmic paradigms to cope with NP-hardness

Assuming $\mathbf{P} \neq \mathbf{NP}$, a problem X being **NP**-hard implies that we cannot have an algorithm `ALG` for it which satisfies both of the following properties:

- `ALG` is always correct
- `ALG` runs in polytime

This has lead to the development of new **algorithmic paradigms** including:

1. Exact Exponential algorithms

   Here we are focused on producing correct results, no matter the time complexity cost.

2. Polytime approximation algorithms

   Here we want to focus on having a polynomial running time, to do this we relax our first requirement and allow our algorithm to return only approximately correct results in some cases.

3. Parameterised algorithms

   Before trying to solve a problem optimally we can fix a parameter k and using this we can now reevaluate running time with respect to both the length of the input as we always have but also our fixed parameter k.

4. Polytime randomisation algorithms

   Here our algorithm must still run in polytime, and instead of always being correct we say that it must be correct with a given probability.

3.1 Exact exponential algorithms

This approach essentially tries to answer the question: *"Can we do better than brute force, even if it still uses exponential time?"*

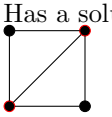
3.1.1 Vertex Cover

Definition 2 *Given an undirected graph $G = (V, E)$ on n vertices, find a set X of minimum size such that each edge of G has at least one endpoint in X*

The brute force approach to this problem runs in $2^n \cdot n^{O(1)}$ time. Can we design an $1.99^n \cdot n^{O(1)}$ time algorithm?

For instance the graph:

Has a solution of:



Where $X = 2$.

The brute force approach to finding this would be to enumerate all subsets of vertex set of G, V in increasing sizes and for each subset, checks in polynomial time whether it is a vertex cover. This check is performed simply by checking whether every element in E is connected to at least one of the vertices in the subset.

We can make observations which can allow us to improve upon this approach. One such observation is that there is no point in adding a vertex of degree 1 to our vertex cover, it can never connect to more points than its unique neighbour (the vertex on the other end of the edge), so we add the unique neighbour instead.

So we say that we only add vertices of degree ≥ 2 to the vertex cover. This, for every potential vertex, introduces a binary choice as to whether we add it to the vertex cover. Using this we define $T(n)$ as the time needed to solve the vertex cover on graphs with n vertices. We can therefore say that $T(n) \leq T(n-1) + T(n-3)$.

We derive this inequality by saying:

- $T(n-1)$ is *spawned* as a sub problem when we *take* v into the vertex cover, we reduce the size of V by 1
- Alternatively, if we do **not** add v to the vertex cover, we **must** add all ≥ 2 neighbours to the VC, so the number of vertices to consider in the subproblem has decreased by **at least** 3.

We can solve this inequality by setting $T(n)$ equal to x^n which gives us $x^0 - x^2 - 1 = 0$, solves to $1.47^n \cdot n^{O(1)}$.

3.2 Polynomial time approximation algorithms

Again looking at our Vertex Cover problem, can we find a vertex cover with has size at most 10 times that of the minimum vertex cover ? Can we do it in $n^{O(1)}$ time?

Do do this we must first find a maximal set M of pairwise disjoint edges. This **can** in fact be found in polynomial time!

Using this we can output a solution R which has both endpoints of each edge from M. By definition R is a vertex cover, albeit not a minimal vertex cover. We can see that if `OPT` is a vertex cover of minimum size, then $|R| \leq 2 \cdot |\texttt{OPT}|$.

We have designed a 2-approximation for the Vertex Cover problem in $n^{O(1)}$ time, nothing better is known.

3.3 Parameterised algorithms

To look at this approach we have to tweak the definition of our Vertex Cover problem:

> **Definition 3** *(Parameterised Vertex Cover) Given an undirected graph $G = (V, E)$ on n vertices and an integer k, goes G have a vertex cover of size $\leq k$?*

This can be thought of as the question: *"Given a parameter k, how fast can we check whether there is a vertex cover of size k?* . Here we are not concerned with the size of the minimum vertex cover, rather we are concerned with being able to say whether the minimal vertex cover lies above or below k

An algorithm has been found which runs in $2^k \cdot n^{O(1)}$ time.

3.4 Polynomial time randomisation algoriths

> **Definition 4** *(Max Cut) Given an undirected graph $G = (V, E)$ on n vertices, find a set X which maximises the number of edges which have one endpoint in X and the other endpoint in $V \backslash X$*

We can define a 0.5-approximation algorithm. Since the goal with this problem is to maximise, the approximation ratio is usually written as < 1. This algorithm works by flipping a coin to decide which side of the partition each vertex goes to. We expect half of the edges to be in the *cut*.

This algorithm runs in $n^{O(1)}$ time.

This approach can be de-randomised by starting with any arbitrary partition, and for each vertex if it has strictly more neighbours in the same side then move it to the other side. This re-shuffling must be finite as eventually exactly half the edges are in each cut.

The best known result for this problem is a 0.86-approximation.