Dealing with NP-hard problems
COMS20010 (Algorithms II)

John Lapinskas, University of Bristol

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away...

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Option 1: Prove P = NP.

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Option 1: Prove P = NP.
Problem: Not going to happen.

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Option 1: Prove P = NP.
Problem: Not going to happen.

Option 2: Quit your job and become a computer science lecturer, so you never have to write code again.

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Option 1: Prove P = NP.
Problem: Not going to happen.

Option 2: Quit your job and become a computer science lecturer, so you never have to write code again.
Problem: Writing proofs is harder than writing code...

Dealing with NP-hard problems

Sadly, knowing that a problem is NP-hard doesn't make it go away... So what are your options from that point?

Option 1: Prove P = NP.
Problem: Not going to happen.

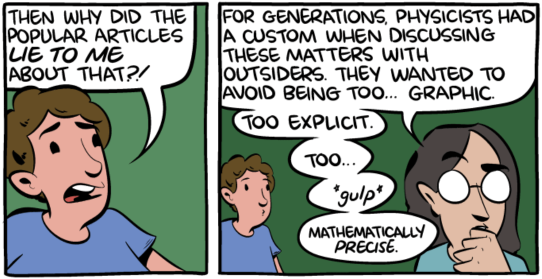
Option 2: Quit your job and become a computer science lecturer, so you never have to write code again.
Problem: Writing proofs is harder than writing code...

Option 3: Wait for a quantum computer and use that.

Option 3: Wait for a quantum computer and use that.

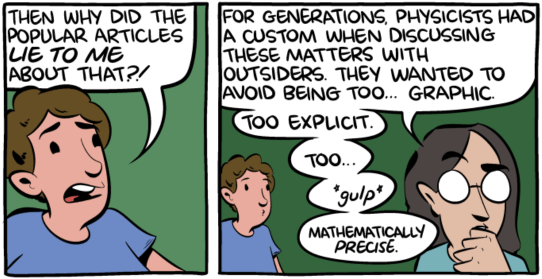
Option 3: Wait for a quantum computer and use that.

Problem: The media has lied to you about quantum computers! They can't just "check all possible witnesses in parallel".



Option 3: Wait for a quantum computer and use that.

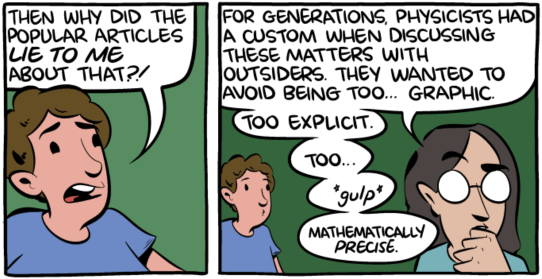
Problem: The media has lied to you about quantum computers! They can't just "check all possible witnesses in parallel".



If we can build quantum computers, they will be good for two things:

Option 3: Wait for a quantum computer and use that.

Problem: The media has lied to you about quantum computers! They can't just "check all possible witnesses in parallel".

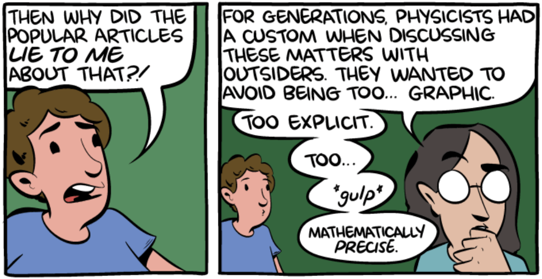


If we can build quantum computers, they will be good for two things:

- Breaking public-key cryptography with e.g. fast prime factorisation;

Option 3: Wait for a quantum computer and use that.

Problem: The media has lied to you about quantum computers! They can't just "check all possible witnesses in parallel".

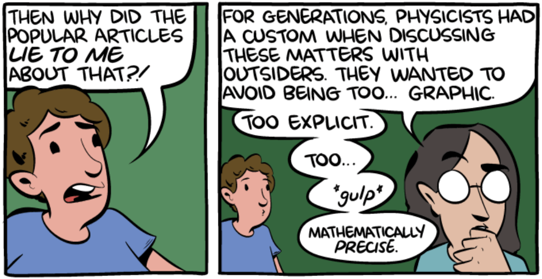


If we can build quantum computers, they will be good for two things:

- Breaking public-key cryptography with e.g. fast prime factorisation;
- Simulating quantum systems.

Option 3: Wait for a quantum computer and use that.

Problem: The media has lied to you about quantum computers! They can't just "check all possible witnesses in parallel".



If we can build quantum computers, they will be good for two things:

- Breaking public-key cryptography with e.g. fast prime factorisation;
- Simulating quantum systems.

That's still very useful for e.g. drug discovery! But it's not solving SAT. There's no reason to think your home PC will ever need a "quantum chip".

The problem is: suppose you form a superposition of every possible witness, and you run the verifier on all of them at once.

The problem is: suppose you form a superposition of every possible witness, and you run the verifier on all of them at once.

Suppose there are 2^n possible witnesses, and only one actual witness.

The problem is: suppose you form a superposition of every possible witness, and you run the verifier on all of them at once.

Suppose there are 2^n possible witnesses, and only one actual witness.

Then your output will be a quantum state which, when you measure it, gives `No` with probability $1 - 2^{-n}$ and `Yes` with probability 2^{-n}.
And measuring the state destroys it.

There's **provably** no way to distinguish this state from a pure "`No`" with any reasonable probability.

(Maths with CS students: see the quantum courses in year 4!)

The problem is: suppose you form a superposition of every possible witness, and you run the verifier on all of them at once.

Suppose there are 2^n possible witnesses, and only one actual witness.

Then your output will be a quantum state which, when you measure it, gives `No` with probability $1 - 2^{-n}$ and `Yes` with probability 2^{-n}.
And measuring the state destroys it.

There's **provably** no way to distinguish this state from a pure "`No`" with any reasonable probability.

(Maths with CS students: see the quantum courses in year 4!)

Most quantum computing researchers are pretty sure that quantum computers **can't** solve NP-complete problems in polynomial time.

The problem is: suppose you form a superposition of every possible witness, and you run the verifier on all of them at once.

Suppose there are 2^n possible witnesses, and only one actual witness.

Then your output will be a quantum state which, when you measure it, gives `No` with probability $1 - 2^{-n}$ and `Yes` with probability 2^{-n}.
And measuring the state destroys it.

There's **provably** no way to distinguish this state from a pure "`No`" with any reasonable probability.

(Maths with CS students: see the quantum courses in year 4!)

Most quantum computing researchers are pretty sure that quantum computers **can't** solve NP-complete problems in polynomial time.

The best evidence for quantum computers being more powerful than classical computers is that they can factor large numbers quickly.

But the evidence this is a hard problem is much weaker than e.g. $\mathrm{P} \neq \mathrm{NP}$.
It's in $\mathrm{NP} \cap$ Co-NP, and some people do think this is contained in P...

Option 4: Try to solve the problem **approximately**.

Option 4: Try to solve the problem **approximately**.

For example, it's NP-hard to find a **minimum** vertex cover, but we already set out an algorithm that finds a vertex cover which is minimum to within a factor of 2.

(See Advanced Algorithms next year for much more on these!)

Option 4: Try to solve the problem **approximately**.

For example, it's NP-hard to find a **minimum** vertex cover, but we already set out an algorithm that finds a vertex cover which is minimum to within a factor of 2.

(See Advanced Algorithms next year for much more on these!)

Problem: There might not be a fast approximation algorithm either.

For example, it's NP-hard to approximate vertex cover to within a factor of 1.36, and it's "UGC-hard" to approximate it to within a factor of $2 - \varepsilon$ for any $\varepsilon > 0$. (So our algorithm is probably best possible.)

Option 4: Try to solve the problem **approximately**.

For example, it's NP-hard to find a **minimum** vertex cover, but we already set out an algorithm that finds a vertex cover which is minimum to within a factor of 2.

(See Advanced Algorithms next year for much more on these!)

Problem: There might not be a fast approximation algorithm either.

For example, it's NP-hard to approximate vertex cover to within a factor of 1.36, and it's "UGC-hard" to approximate it to within a factor of $2 - \varepsilon$ for any $\varepsilon > 0$. (So our algorithm is probably best possible.)

And it's NP-hard to approximate independent set even to within a factor of $n^{0.9999}$ on an n-vertex graph.

Option 4: Try to solve the problem **approximately**.

For example, it's NP-hard to find a **minimum** vertex cover, but we already set out an algorithm that finds a vertex cover which is minimum to within a factor of 2.

(See Advanced Algorithms next year for much more on these!)

Problem: There might not be a fast approximation algorithm either.

For example, it's NP-hard to approximate vertex cover to within a factor of 1.36, and it's "UGC-hard" to approximate it to within a factor of $2 - \varepsilon$ for any $\varepsilon > 0$. (So our algorithm is probably best possible.)

And it's NP-hard to approximate independent set even to within a factor of $n^{0.9999}$ on an n-vertex graph. (You can approximate it to within a factor of n by outputting a single vertex, so that's not great...)

Option 5: Look for exploitable properties of your specific input.

Option 5: Look for exploitable properties of your specific input.

For example: It's NP-hard to find a size-k independent set in a graph.

Even if k is a constant, you (almost certainly) need $n^{\Omega(k)}$ time where n is the number of vertices in the input graph.

Option 5: Look for exploitable properties of your specific input.

For example: It's NP-hard to find a size-k independent set in a graph.

Even if k is a constant, you (almost certainly) need $n^{\Omega(k)}$ time where n is the number of vertices in the input graph.

But if k is constant **and** your input graphs all have maximum degree at most 100 (say), then you can do it in $O(n)$ time!

Option 5: Look for exploitable properties of your specific input.

For example: It's NP-hard to find a size-k independent set in a graph.

Even if k is a constant, you (almost certainly) need $n^{\Omega(k)}$ time where n is the number of vertices in the input graph.

But if k is constant **and** your input graphs all have maximum degree at most 100 (say), then you can do it in $O(n)$ time!

Sample questions to ask yourself for graph problems:

- Do my input graphs have few edges?
- Do my input graphs have low maximum degree?
- Are my input graphs close to trees?
- Can I assume my input graphs follow some random distribution?

Option 5: Look for exploitable properties of your specific input.

For example: It's NP-hard to find a size-k independent set in a graph.

Even if k is a constant, you (almost certainly) need $n^{\Omega(k)}$ time where n is the number of vertices in the input graph.

But if k is constant **and** your input graphs all have maximum degree at most 100 (say), then you can do it in $O(n)$ time!

Sample questions to ask yourself for graph problems:

- Do my input graphs have few edges?
- Do my input graphs have low maximum degree?
- Are my input graphs close to trees?
- Can I assume my input graphs follow some random distribution?

One very useful concept that can often be exploited is **low treewidth**. The definition is too complicated for this course, but if your graph can be split into two by removing only a few vertices, and then you can continue the process recursively, then it probably has low treewidth.

A lot of useful restrictions to the input are of the form “this parameter is low”, such as low maximum degree, low average degree, few cycles, or low treewidth. These are called **parameterised problems**.

A lot of useful restrictions to the input are of the form “this parameter is low”, such as low maximum degree, low average degree, few cycles, or low treewidth. These are called **parameterised problems**.

How should we define efficiency for them? If our input is an n-vertex graph with maximum degree Δ, it’s not very useful if our algorithm takes time n^{Δ}, even though that’s polynomial for fixed Δ.

A lot of useful restrictions to the input are of the form "this parameter is low", such as low maximum degree, low average degree, few cycles, or low treewidth. These are called **parameterised problems**.

How should we define efficiency for them? If our input is an n-vertex graph with maximum degree Δ, it's not very useful if our algorithm takes time n^{Δ}, even though that's polynomial for fixed Δ.

Idea: We want there to be some constant C such that the running time is $O(n^C)$ for **all** fixed Δ. The leading constant will get worse as Δ increases, but not the exponent C.

A lot of useful restrictions to the input are of the form "this parameter is low", such as low maximum degree, low average degree, few cycles, or low treewidth. These are called **parameterised problems**.

How should we define efficiency for them? If our input is an n-vertex graph with maximum degree Δ, it's not very useful if our algorithm takes time n^{Δ}, even though that's polynomial for fixed Δ.

Idea: We want there to be some constant C such that the running time is $O(n^C)$ for **all** fixed Δ. The leading constant will get worse as Δ increases, but not the exponent C.

Equivalently: We require the running time to be at most $g(\Delta) \cdot n^C$ for some computable function g. (Ideally we want $g(\Delta) = 2^{O(\Delta)}$ or $\Delta^{O(\Delta)}$.)

A lot of useful restrictions to the input are of the form “this parameter is low”, such as low maximum degree, low average degree, few cycles, or low treewidth. These are called **parameterised problems**.

How should we define efficiency for them? If our input is an n-vertex graph with maximum degree Δ, it's not very useful if our algorithm takes time n^{Δ}, even though that's polynomial for fixed Δ.

Idea: We want there to be some constant C such that the running time is $O(n^C)$ for **all** fixed Δ. The leading constant will get worse as Δ increases, but not the exponent C.

Equivalently: We require the running time to be at most $g(\Delta) \cdot n^C$ for some computable function g. (Ideally we want $g(\Delta) = 2^{O(\Delta)}$ or $\Delta^{O(\Delta)}$.)

We call problems with algorithms like this **fixed-parameter tractable** (**FPT**). The analogue of NP-hardness in this setting is **W[1]-hardness**.

These can be useful search terms when looking for practical algorithms!

Option 6: Try something slower than polynomial time.

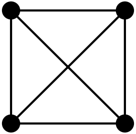
Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

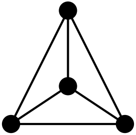
Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

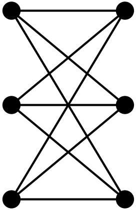
A graph is **planar** if it can be drawn without any two edges overlapping.

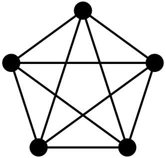


This graph is planar...



because it can be drawn like this.



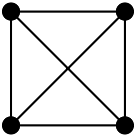


But these graphs are **not** planar.

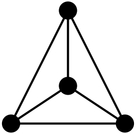
Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

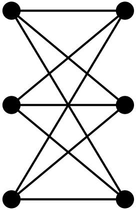
A graph is **planar** if it can be drawn without any two edges overlapping.

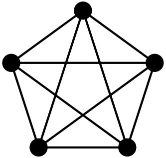


This graph is planar...



because it can be drawn like this.



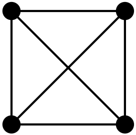


But these graphs are **not** planar. (In some sense they're the only ones that aren't...)

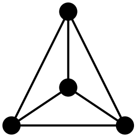
Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

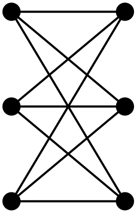
A graph is **planar** if it can be drawn without any two edges overlapping.

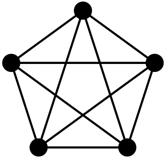


This graph is planar...



because it can be drawn like this.





But these graphs are **not** planar. (In some sense they're the only ones that aren't...)

Planar graphs are often used to model physical locations, with edges between neighbouring locations.

Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

A graph is **planar** if it can be drawn without any two edges overlapping.

Planar graphs are often used to model physical locations, with edges between neighbouring locations.

Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

A graph is **planar** if it can be drawn without any two edges overlapping.

Planar graphs are often used to model physical locations, with edges between neighbouring locations.

Many problems, including independent set, are still NP-complete on planar input graphs. But they have $2^{O(\sqrt{n})}$-time algorithms!

(This is because n-vertex planar graphs have $O(\sqrt{n})$ treewidth.)

Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

A graph is **planar** if it can be drawn without any two edges overlapping.

Planar graphs are often used to model physical locations, with edges between neighbouring locations.

Many problems, including independent set, are still NP-complete on planar input graphs. But they have $2^{O(\sqrt{n})}$-time algorithms!

(This is because n-vertex planar graphs have $O(\sqrt{n})$ treewidth.)

Problem: Again, this is often impossible. The **Exponential Time Hypothesis** (**ETH**) says that an n-variable instance of 3-SAT can't be solved in $2^{o(n)}$ time. There's no subexponential-time algorithm for independent set on arbitrary graphs unless ETH is false.

Option 6: Try something slower than polynomial time.

Many NP-complete problems have subexponential-time algorithms!

A graph is **planar** if it can be drawn without any two edges overlapping.

Planar graphs are often used to model physical locations, with edges between neighbouring locations.

Many problems, including independent set, are still NP-complete on planar input graphs. But they have $2^{O(\sqrt{n})}$-time algorithms!

(This is because n-vertex planar graphs have $O(\sqrt{n})$ treewidth.)

Problem: Again, this is often impossible. The **Exponential Time Hypothesis** (**ETH**) says that an n-variable instance of 3-SAT can't be solved in $2^{o(n)}$ time. There's no subexponential-time algorithm for independent set on arbitrary graphs unless ETH is false.

Of course, if your instances are of size 15–20 you can just use brute force...

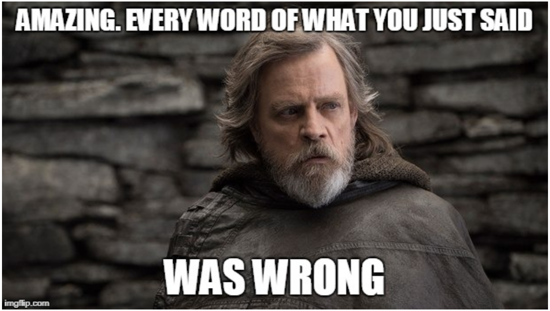
Option 6b: Try a modern SAT-solver.

Option 6b: Try a modern SAT-solver.

Under ETH, this will take $2^{\Omega(n)}$ time to solve an instance with n variables. So it's not going to be practical on e.g. a 200-variable instance.

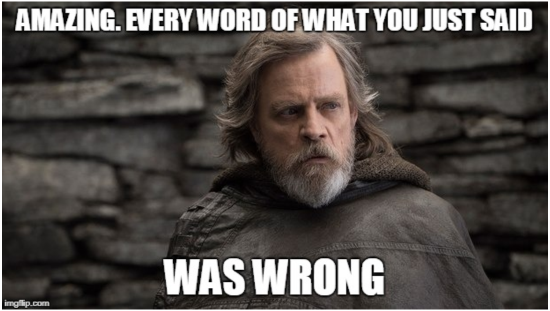
Option 6b: Try a modern SAT-solver.

Under ETH, this will take $2^{\Omega(n)}$ time to solve an instance with n variables. So it's not going to be practical on e.g. a 200-variable instance.



Option 6b: Try a modern SAT-solver.

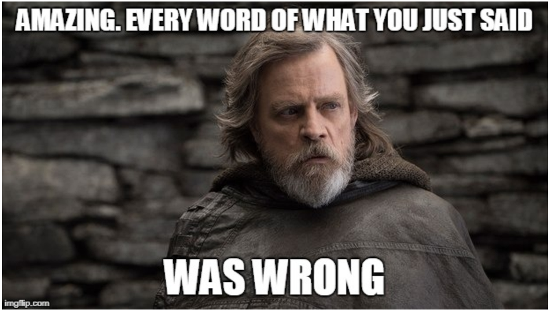
Under ETH, this will take $2^{\Omega(n)}$ time to solve an instance with n variables. So it's not going to be practical on e.g. a 200-variable instance.



Modern SAT-solvers can often handle **tens of thousands of variables**!

Option 6b: Try a modern SAT-solver.

Under ETH, this will take $2^{\Omega(n)}$ time to solve an instance with n variables. So it's not going to be practical on e.g. a 200-variable instance.



Modern SAT-solvers can often handle **tens of thousands of variables**!

Clearly "real-world SAT instances" often have some nice properties that solvers are exploiting... but we can't nail down what those properties are. This is a huge open problem right now.

So just give it a try! It might not work, but it's worth checking.