UCT

Polynomial Time

Klaus Sutner

Carnegie Mellon University

Spring 2024

Feasible Decision Problems

The question arises whether any time complexity class is a good match for our intuitive notion of "feasible computation" or "tractable problem."

Note that whatever answer we give, we are in a similar situation as with the Church-Turing thesis: since we deal with intuitive notions there cannot be a formal proof – though one can try to amass ample evidence.

However, while there is fairly good agreement about what constitutes a computation, there is much less agreement about what constitutes a feasible computation. If one deals with very large data sets, anything beyond $n \log n$ is not really feasible.

(Deterministic) Polynomial Time

Recall our definition of polynomial time solvable problems:

$$\mathbb{P} = \mathrm{TIME}(n^{O(1)}) = \bigcup_{k \geq 0} \mathrm{TIME}(n^k).$$

Let's put up a straw man: let's declare $\mathbb{P}$ to be our stand-in for feasible computation.

But note that $\mathbb{P}$ is only about decision problems, in the world of actual algorithms there is no doubt that function problems are more important.

Types of Problems

Here is a fairly natural taxonomy of computational problems.

Decision Problems Return a Yes/No answer.

Counting Problems Determine the size of some object.

Function Problems Calculate a certain function.

Search Problems Select one particular solution.

How about all this other stuff?

Example: Vertex Cover

Problem: **Vertex Cover (decision)**
Instance: A ugraph G, a bound k.
Question: Does G have a vertex cover of size k?

Problem: **Vertex Cover (counting)**
Instance: A ugraph G.
Solution: Find the size of a minimal vertex cover of G.

Problem: **Vertex Cover (function)**
Instance: A ugraph G.
Solution: Find the lex-minimal vertex cover of G.

Problem: **Vertex Cover (search)**
Instance: A ugraph G.
Solution: Find a vertex cover of G of minimal size.

Connections

It should be clear that there are lots of logical connections between these different versions of the problem, they are all somehow related.

For example, if we can solve the counting/function/search version, then we also can handle the decision version. In other words, given a fast algorithm for counting/function/search, we can construct a fast algorithm for decision.

The opposite direction is also true, but much more involved to deal with in a general, abstract way (more later). For Vertex Cover specifically, try to figure out how to go from decision to, say, counting.

Feasible Counting/Function/Search Problems

To deal with problems that require more than a single bit of output, we need transducers as opposed to acceptors. Recall that in a transducer we attach an additional input/output tape to our Turing machines.

Definition

A function is polynomial time computable if it can be computed by a polynomial time Turing machine transducer.

The output for counting problems is just a number, but for function and search problems it can be some arbitrary combinatorial object (represented as, say, a bit-string).

Full Disclosure

Computing functions, as opposed to just solving decision problems, is beyond any doubt more important in the world of real algorithms. Interestingly, the concept of the class of polynomial time computable functions appeared a bit before modern complexity theory:

- von Neumann 1953
- Gödel 1956
- Cobham 1964
- Edmonds 1965

So why don't we bite the bullet and deal with feasible computable functions?

Because it's very hard. Decision problems make enough of a mess, so let's not worry about things that are even more complicated at this point.

Closure

We mention just one hugely important property of polynomial time computable functions: they can be composed sequentially without falling out of the class.

Lemma

Polynomial time functions are closed under composition.

Informally, if we have a fast algorithm to take A to B, and another fast algorithm to take B to C, then we also have a fast algorithm to take A to C.

Proof

Proof. The short version is: polynomials are closed under substitution.

More precisely, suppose $y = f(x)$ is computable in time at most $p(n)$ where $n = |x|$.

Then $|y| \leq p(n)$ and $z = g(y)$ is computable in time at most $q(p(n))$ for some polynomial q.

Hence we have a polynomial time bound for $z = g(f(x))$. □

In fact, if $p(n) = O(n^k)$ and $q = O(n^\ell)$, then the composition is $O(n^{k\ell})$ at most.

Polynomials are Great

Just to be clear: this result may seem trivial, but it depends crucially on our choice of polynomials as the resource bounds.

Closure fails for, say, exponential time $\mathrm{EXP}_1 = 2^{O(n)}$.

But similar claims do hold for

- linear time (easy),
- logarithmic space (hard, more later).

Two Standard Objections

The notation $T_{\mathcal{M}}(n) = O(n^k)$ hides two constants:

$$\exists\, n_0, c\, \forall\, n \geq n_0 \left(T_{\mathcal{M}}(n) \leq c \cdot n^k\right)$$

What if these constants are huge? Something like 1000^{1000}? Note that there are only around 10^{80} particles in the universe.

This would render the asymptotic bounds entirely useless for anything resembling feasible computation. It could even wreck physical computation entirely.

Rant

At this point, one really should distinguish between two kinds of algorithms:

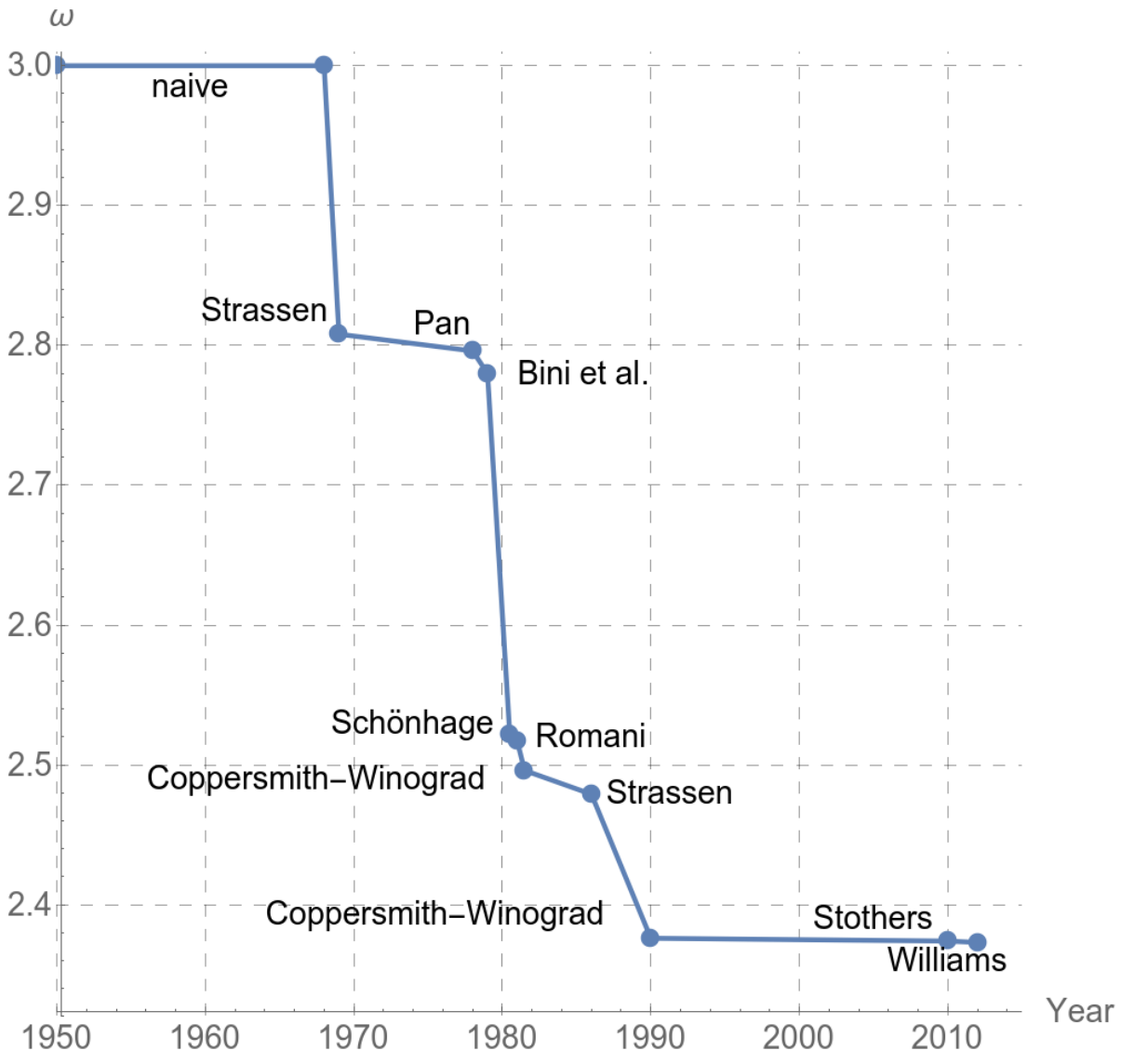
practical Developed with the intent of implementation and actual use; close to programming and data types.

theoretical Developed with no intention whatsoever to implement and use; the goal is to publish a paper.

In the second category there are, for example, algorithms that try to get matrix multiplication ever closer to n^2. The matrices need to be huge to get mileage out of the asymptotic bound.

These asymptotically fast algorithms are very interesting conceptually, but may be entirely irrelevant for feasible computation.

Matrix Multiplication



Standard Answer

This objection has merit in principle, but in the RealWorld$^{\mathsf{TM}}$ it is pointless: for practical problems it is a matter of experience that the constants are easy to determine and are (almost) always very reasonable.

In fact, we can even compute the constants by writing down the algorithms very carefully in a low-level language like C.

For practical algorithms, this is somewhat of a pain, but not really terribly difficult (as long as we are fairly relaxed about the bounds).

Example: Quadratic

Consider a code fragment like

```
// P
  for i = 1 to n do
      for j = 1 to i do
          blahblahblah
```

Suppose `blahblahblah` is constant time, a bunch of comparisons and assignments, say.

Clearly, the running time of P is quadratic, $O(n^2)$.

Assembly

But if we wanted to, we could write P in assembly,

```
0:   55                          push   ebp
1:   89 e5                       mov    ebp,esp
3:   83 e4 f0                    and    esp,0xfffffff0
6:   83 ec 10                    sub    esp,0x10
9:   c7 04 24 00 00 00 00        mov    DWORD PTR [esp],0x0
```

Now we can count the number of steps each execution of P takes, likewise for the control structures. The result might be $55n + O(1)$.

But it surely won't be $1000^{1000}n + O(1)$.

Digression: Knuth

D. Knuth (everybody bow three times in the direction of Stanford) actually thinks that pure asymptotic bounds are fairly cheesy.

We should never say "this algorithm has complexity $O(n)$", we should explicitly figure out the coefficients of the leading term. Say

$$\ldots = 7 \cdot n + O(\log n)$$

This is easy to say for Knuth, who just develops the necessary math on the fly, but quite challenging for ordinary mortals. One also needs to be a bit careful about stating the meaning of "one step" explicitly.

The Bibles

D. E. Knuth
The Art of Computer Programming (TAOCP)
Addison-Wesley, 1968–now

D. E. Knuth, O. Patashnik, R. Graham
Concrete Mathematics
Addison-Wesley, 1988

Aside

Knuth is very unhappy about people publishing algorithms without ever implementing them (the practical vs theoretical distinction above).

http://www.informit.com/articles/article.aspx

This is currently really an unsolved problem in the profession, reproducibility of algorithmic claims needs to be added to publication requirements.

Truth in Advertising

The claim that we can always figure out the multiplicative constant is somewhat of a white lie. There are a few algorithms in graph theory, based on the Robertson-Seymour theorem for graph minors, where these constants are utterly unknown.

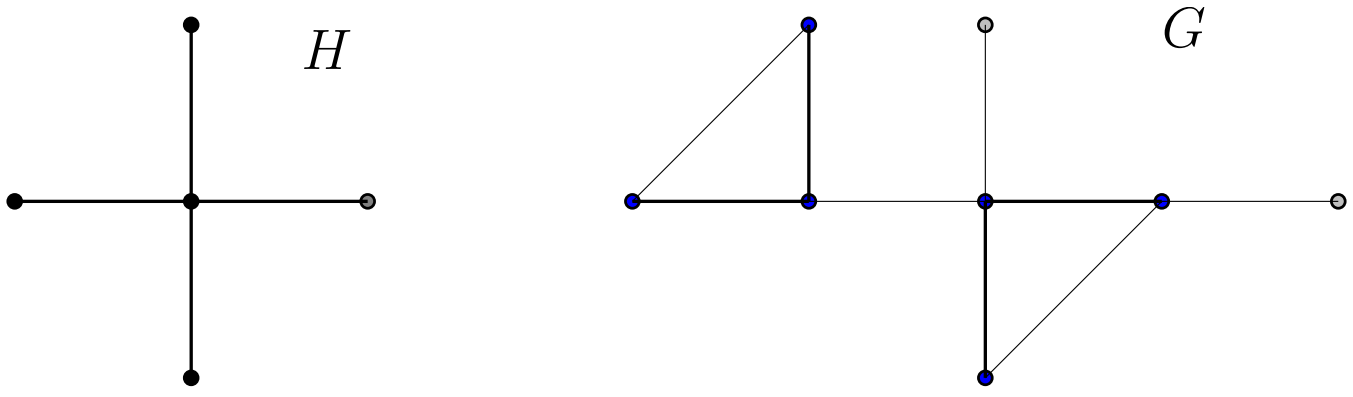
The theorem is a huge result that make it possible to prove the existence of a polynomial time algorithm **without** actually exhibiting the algorithm. Non-constructive proofs are completely standard in mathematics, but in the theory of algorithms they are a bit strange.

Graph Minors

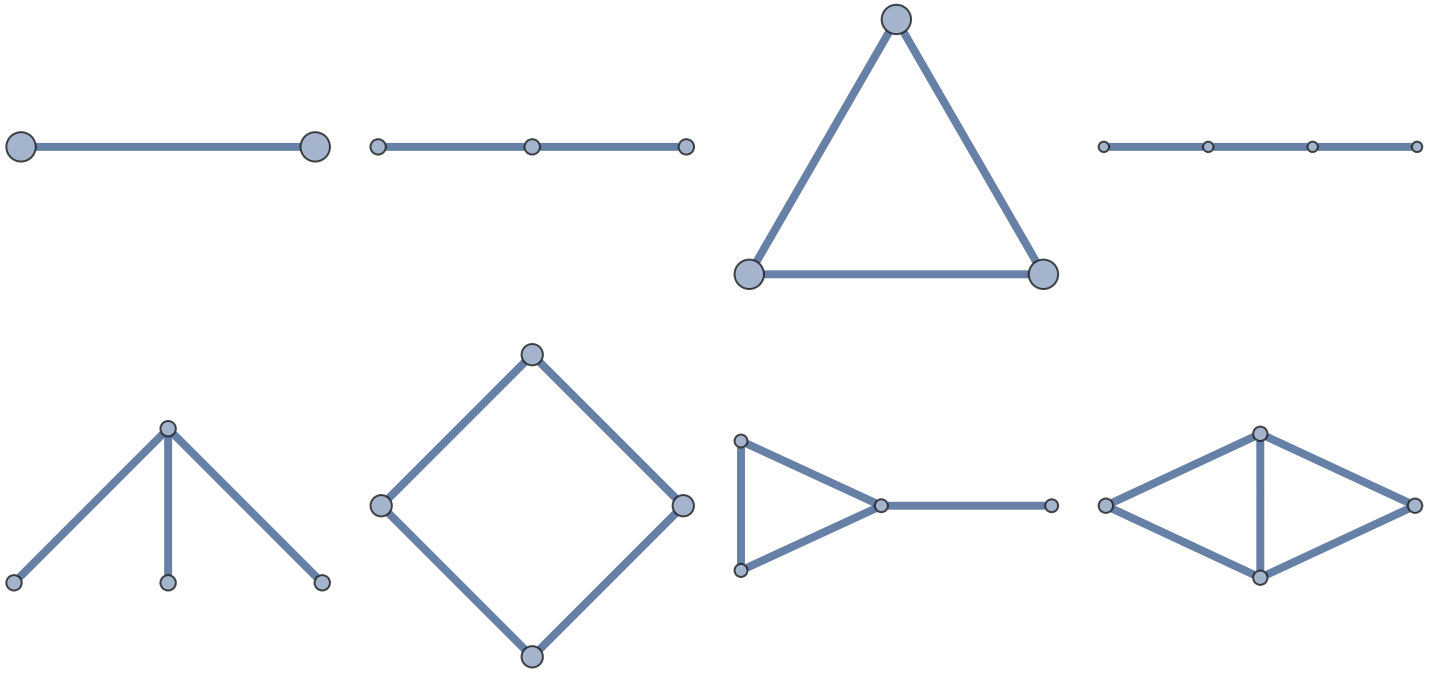
We are only dealing with undirected, simple graphs here.

Graph H is a minor of graph G if H is (isomorphic to) a graph obtained from G by

- edge deletions,
- edge contractions, and
- deletion of isolated vertices.



To first-order approximation, you can think of a minor as being a subgraph of sorts.



Some minors of the complete graph on 4 points.

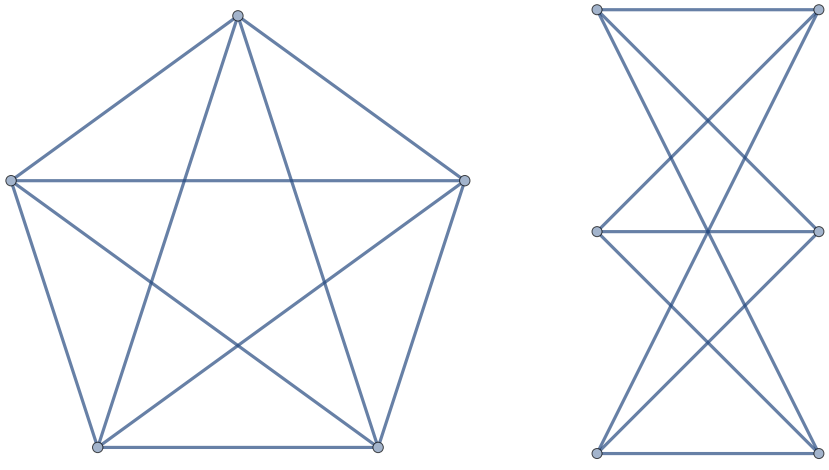
Closed Classes

A collection of finite graphs $\mathcal{G}$ is closed under minors if, for all $G \in \mathcal{G}$ and H a minor of G, we have $H \in \mathcal{G}$.

The classical example is the class of planar graphs: every minor of a planar graph is always planar. This produces a famous theorem:

Theorem (Kuratowski-Wagner)

A graph is planar if, and only if, it has no minor K_5 or $K_{3,3}$.



Wagner's Conjecture

In 1937, K. Wagner conjectured that a similar theorem holds for every class $\mathcal{C}$ of finite graphs that is closed under minors:

Conjecture

Suppose $\mathcal{C}$ is minor-closed. Then there are finitely many obstruction graphs $H_1, H_2, \ldots, H_r$ such that:
G is in $\mathcal{C}$ if, and only if, none of the H_i appear as a minor in G.

Note that this yields a decision algorithm: we just check if some given graph G has one of the finitely many forbidden minors.

In terms of order theory, the key insight is that all antichains of graphs wrt the minor order are finite[†].

[†]This is very closely connected to Higman's theorem about the lack of infinite antichains in the subsequence order.

Wagner's conjecture gave rise to the following amazing theorem:

Theorem (Robertson, Seymour)

Every minor-closed family of graphs has a finite obstruction set.

This was proven in a series of 23 papers from 1984 to 2004, an incredible tour de force. The total proof is hundreds of pages long.

The proof has been examined very carefully and is probably correct, but putting it through a theorem prover would not hurt.

The proof of the Robertson-Seymour theorem is brutally non-constructive: the finite obstruction set exists in some set-theoretic universe, but we have no way of constructing it.

Worse, often we don't even know its (finite) cardinality.

Does it have any computational content? Well, for planar graphs it certainly does: we could check planarity by looking for minors K_3 and $K_{3,3}$. Needless to say, there are much better planarity testing algorithms (linear time by Tarjan), but in principle this works just fine.

Theorem To Algorithm

Suppose that graph H is fixed.

Problem:	H-**Minor Query**
Instance:	A ugraph G.
Question:	Is H a minor of G?

There is an algorithm that tests whether H is a minor of G in $O(n^2)$ steps, $n = |G|$.

Note that H has to be fixed, otherwise we could check whether G contains a cycle of length $|G|$ in quadratic time (aka Hamiltonian cycle). As we will see, this problem is $\mathbb{NP}$-complete, so a polynomial time solution is rather unlikely.

Doom and Disaster

But then we can check all H from a finite list of obstruction graphs in quadratic time.

Hence we can check membership in any minor closed class $\mathcal{G}$ in quadratic time.

Alas, there is a glitch: the finite obstruction list is obtained non-constructively; it exists somewhere, and we can prove its existence using sufficiently strong set theory, but we cannot in general determine its elements—and, in fact, not even its cardinality. So we get a quadratic time algorithm, but we cannot bound the multiplicative constant.

Ouch.

Objection Two

What if $T_{\mathcal{M}}(n) = O(n^{1000})$?

This is polynomial time, but practically useless. By the time hierarchy theorem, we know that problems exist that can be solved in time $O(n^{1000})$ but essentially not in less time.

But note that these problems are constructed by diagonalization techniques, and are thus entirely artificial; they do not correspond to decent RealWorld$^{\text{TM}}$ problems.

Weird Empirical Fact

> If a natural problem is in $\mathbb{P}$ at all, then it can actually be solved in time $O(n^{10})$ – for some small value of 10.

Take this with ample amounts of salt, all we know is that, so far, there simply is no **natural problem** where the best known algorithm has running time $O(n^{1000})$.

Alas, no one has any idea why this low-degree principle appears to be true. Note the qualifier "natural". Everyone understands intuitively what this means, but it seems very difficult to give a formal definition.

Close Call

In 2002, Agrawal, Kayal and Saxena published a paper that shows that primality testing is in polynomial time.

Amazingly, the algorithm uses little more than high school arithmetic.

The original algorithm had time complexity $\widetilde{O}(n^{12})$, but has since been improved to $\widetilde{O}(n^6)$.

Alas, the AKS algorithm seems useless in the RealWorld$^{\mathsf{TM}}$, probabilistic algorithms are much superior[†].

[†]An indication that $\mathbb{P}$ is probably not quite the right definition of feasible computation; we need randomness.

Scaling Down

"Small" classes like $\mathbb{P}$ or EXP don't play well with our reductions from CRT. For example

$$A \in \mathrm{EXP} \qquad \text{implies} \qquad A \leq_T \emptyset$$

The problem is that the oracle TM can directly solve A, it has no need to query the oracle. And, $\leq_T$ can eviscerate much more than just EXP.

But recall: we mentioned that the power source in a reduction is supposed to be the oracle, not the auxiliary computation. We need to adjusyt our reductions in a way that limits the power of this auxiliary work.

Polynomial Time Turing Reductions

Perhaps the most obvious attempt would be to use Turing reductions, but with the additional constraint that the Turing machine must have polynomial running time. This is called a polynomial time Turing reduction.

Notation: $A \leq_T^p B$.

Note that this actually makes sense for any kind of problem, not just decision problems. Also, it really captures nicely the idea that problem A is easier than B since we could use a fast algorithm for B to construct a fast algorithm for A.

Free Lunch

Just to be clear: in a computation with oracle B we only charge for the steps taken by the ordinary part of the Turing machine. For example, it costs to write a query for the oracle on the tape.

But the answer provided by the oracle appears magically in just one step, we do not care about the internals of the oracle.

Proposition

Polynomial time Turing reducibility is a preorder (reflexive and transitive).

For transitivity, this works since polynomials are closed under substitution. Hence we can from equivalence classes as usual, the polynomial time Turing degrees: we lump together all problems that have the same level of difficulty and then try to figure out how these collections of "equivalent" problems compare to each other.

We won't pursue this idea here.

Compatibility

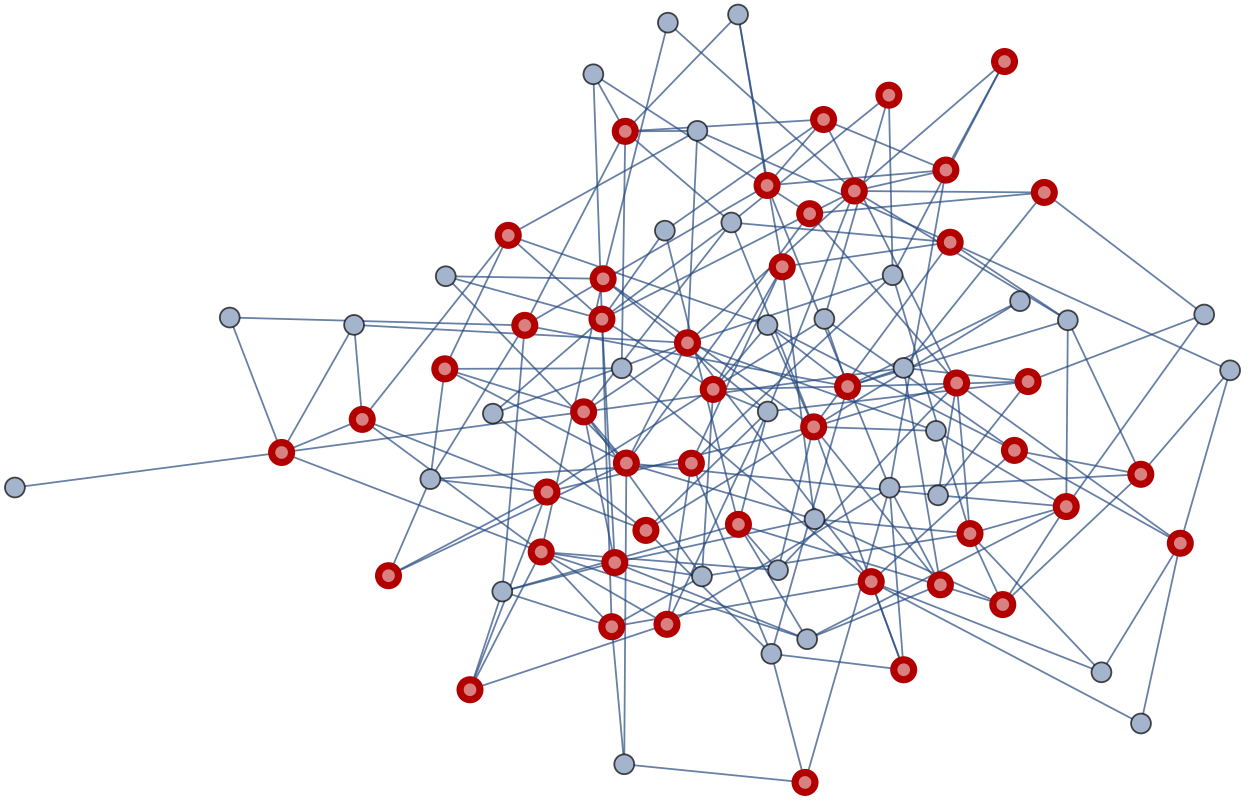
Proposition

$B \in \mathbb{P}$ and $A \leq_T^p B$ implies $A \in \mathbb{P}$.

Here we can simply replace the oracle by a real polynomial algorithm.

Since polynomials are closed under substitution, this will produce a polynomial time algorithm.

Recall: Vertex Cover



Finding minimal vertex covers is quite messy, even for moderate size graphs. In fact, even verifying a solution is hard.

Comparisons

Recall the 4 versions: decision, counting, function and search.

Intuitively, the function version is the hardest: once we have a lex-minimal cover, all other problems are trivial. Clearly, they are all polynomial Turing reducible to the function version.

The counting and decision versions are also Turing reducible to the search version.

Lastly, the decision version is Turing reducible to counting.

$$\text{Decision} \leq_T^p \text{Counting} \leq_T^p \text{Search} \leq_T^p \text{Function}$$

The Other Way

Proposition

The function version of Vertex Cover is polynomial time Turing reducible to the decision version.

This requires a little argument. Let n be the number of vertices in G, say, $V = \{v_1, \ldots, v_n\}$ in lexicographic order.

- First, do a binary search to find the size k_0 of a minimal cover, using the oracle for the decision problem.
- Then find the least vertex v such that $G - v$ has a cover of size $k - 1$. Place v into C and remove it from G.
- Recurse.

This is polynomial time, even using Turing machines.

Justification

The last proposition is important: it justifies our narrow focus on decision problems: up to a polynomial factor, they are no worse than the fancier, more applicable versions. They all live in the same polynomial time Turing degree.

Of course, this does not help much when we are trying to find super-efficient algorithms. Typical example: Tarjan's strongly connected component algorithm, a form of DFS on steroids.

But, if all we want is to make sure that things can be done in polynomial time, the decision version is often good enough. Ditto for showing that things cannot happen in polynomial time.

Still No Good

Unfortunately, while polynomial Turing reductions are compatible with $\mathbb{P}$, they are still too brutish. To wit:

$$A \in \mathbb{P} \qquad \text{implies} \qquad A \leq_T^p \emptyset$$

The problem again is that the oracle TM can directly solve A in the auxiliary computation, it has no need to actually query the oracle.

How about the other reductions (many-one in particular) that we considered in the CRT part?

Weak Reductions

Many-one reducibility is based on a computable function translating from one problem to another. We'll just have to constrain the function a bit.

Definition

$A \subseteq \Sigma^\star$ is polynomial time (many-one) reducible to $B \subseteq \Sigma^\star$ if there is a polynomial time computable function f such that

$$x \in A \iff f(x) \in B.$$

Notation: $A \leq_m^p B$.

Proposition

Polynomial time many-one reducibility is a preorder.

Proof.

Reflexivity is trivial. For transitivity, consider a polynomial time reduction f from A to B and g from B to C.

Obviously $h(x) = g(f(x))$ is a reduction from A to C.

h is still polynomial time computable since polynomials are closed under substitution.

□

Better Mousetrap?

Proposition

$\leq_m^p$ is compatible wrto $\mathbb{P}$: $B \in \mathbb{P}$ and $A \leq_m^p B$ implies $A \in \mathbb{P}$.

This is clear: we can replace the oracle B by a polynomial time algorithm.

But is $\leq_m^p$ really a useful reduction for $\mathbb{P}$?

We could use the mediating function to "reduce" a problem in $\mathrm{TIME}(n^{1000})$ to $\mathrm{TIME}(n)$.

Again: we really want the reductions to be lightweight, the oracle should be the place where the heavy lifting takes place.

Log-Space Reductions

As it turns out, a better reduction results from restricting f even more: we'll insist that f can be computed in logarithmic space. More details about space complexity later, for the time being use your intuition.

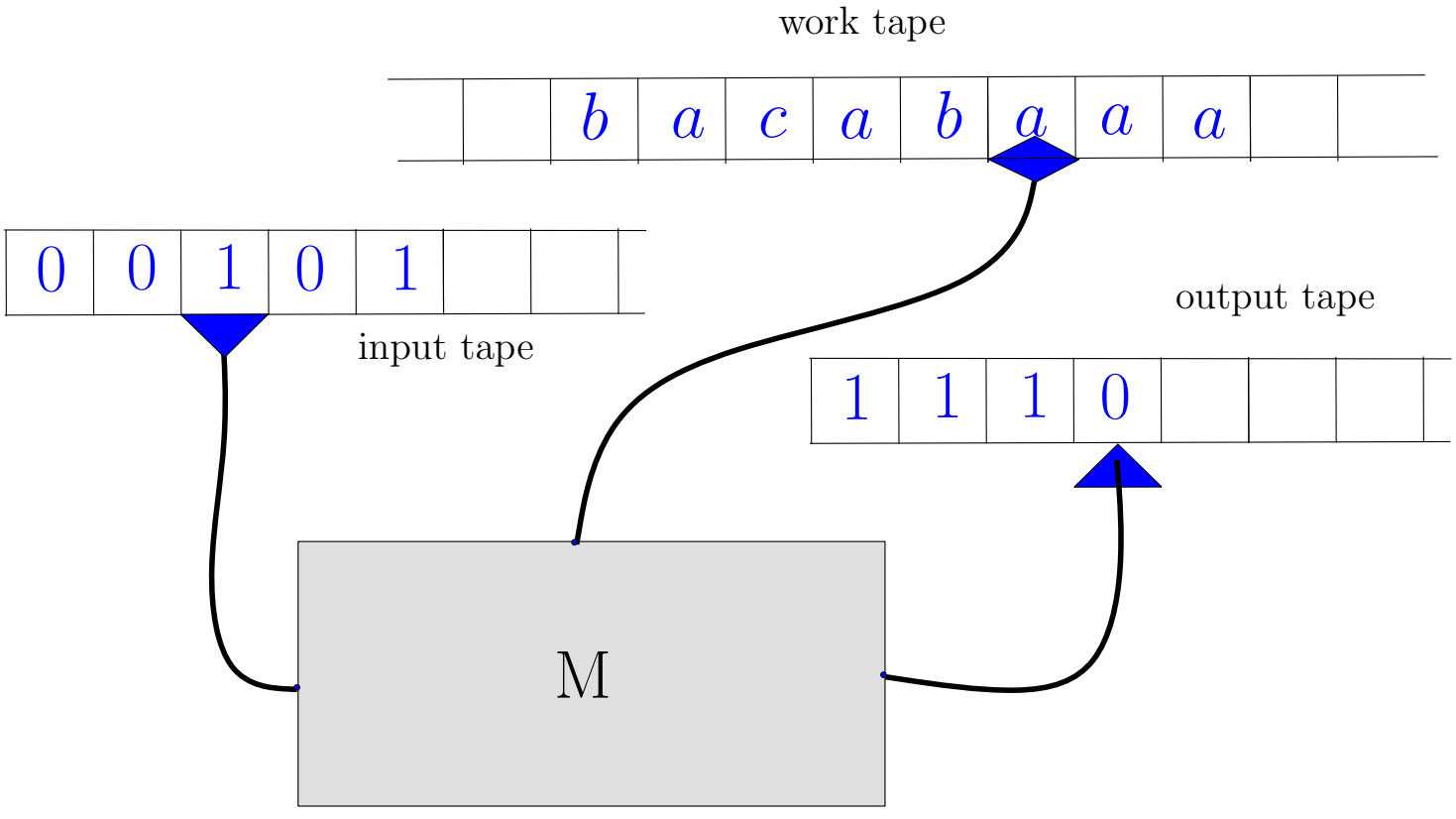
Definition

$A \subseteq \Sigma^\star$ is log-space reducible to $B \subseteq \Sigma^\star$ if there is a log-space computable function f such that

$$x \in A \iff f(x) \in B.$$

Notation: $A \leq_\ell B$.

Recall: Transducer



Only the work tape counts. We want it to be logarithmic in the size of the input.

But Why?

Clearly $A \leq_\ell B$ implies $A \leq_m^p B$.

But the mediating function f is now much more constrained. Think about graph problems where $[n]$ is the vertex set.

We cannot remember an arbitrary subset $S \subseteq [n]$, since that would require a linear amount of memory.

This constraint immediately wrecks many graph algorithms such as DFS that the transformation might try run under the cover to solve a reachability problem. For us, that's good news since it forces the oracle to do most of the work.

Example: Neighborhood Traversals

Here is a typical code fragment that appears in lots of graph algorithms: look at the neighborhoods of all vertices. For example, to compute the maximum degree of a graph, we could use a code fragment like this one:

```
foreach  v in V  do
   foreach  (v,u) in E do
            blahblahblah
   blahblahblah
```

The outer loop can be handled in logarithmic space: we only need to store/manipulate one vertex in $[n]$. Since a vertex is written in binary. this requires only logarithmic memory, say, k bits.

Neighbors

Let's assume the graph is given as an adjacency matrix, flattened out to a binary string A of length n^2. We are currently sitting at vertex v.

To find all the neighbors u of v we do the following:

- Use two k-bit binary counters to find the first bit of the block of length n in A that represents the neighbors of v.
- Traverse the next n bits, looking for ones and counting as you go along. Every time a 1 is found, the counter represents u.

A similar method would work for adjacency lists. The running time may differ, but the memory requirement is logarithmic.

Structuring Polynomial Time

Unlike polynomial-time reductions, log-space reductions can be used to study the fine-structure of $\mathbb{P}$.

Definition ($\mathbb{P}$-Completeness)

A language B is $\mathbb{P}$-hard if for every language A in $\mathbb{P}$: $A \leq_\ell B$.
It is $\mathbb{P}$-complete if it is $\mathbb{P}$-hard and in $\mathbb{P}$.

Building a $\mathbb{P}$-hard set is easy: take an enumeration $(\mathcal{M}_e)$ of all polynomial time TMs, and build the analogue to the Halting set:

$$K_p = \{\, e\#x \mid \mathcal{M}_e(x) \text{ accepts} \,\} \subseteq \Sigma^\star$$

Of course, there is no reason why K_p should be in $\mathbb{P}$: there is no fixed k that bounds the individual tests at $O(n^k)$; instead, k depends on e.

More Precisely . . .

As we have seen previously, in order to construct an enumeration $(\mathcal{M}_e)$ of polynomial time machines, we cannot start with an arbitrary effective enumeration of all Turing machines and filter out the good machines.

First, we cannot check whether the machines are total.

But even if we could, we would still need to check that $\mathcal{M}_e$ runs in polynomial time.

Needless to say, this is also undecidable. Intuitively we need to check

$$\exists\, k \,\forall\, n, x \left(|x| = n \Rightarrow T_{\mathcal{M}}(x) \leq n^k + k \right)$$

This looks completely hopeless, something like Σ_2 rather than Δ_1.

Usual Workaround

Start with an effective enumeration (N_e) of all machines that has infinitely many repetitions.

- attach a clock to N_e, and
- stop the computation after at most $n^e + e$ steps;
- return the same output as $N_e(x)$ if the computation has converged,
- otherwise return 0 (or some other default value).

As written, this is still a little wishy-washy. We really need to build a universal machine $\mathcal{U}$ that simulates $\mathcal{M}_e$ in the appropriate manner.

Done

Since $\mathcal{U}$ chops off computations that take too long and returns a default value, all the simulated machines $\mathcal{M}_e$ are total by construction.

Furthermore, if some total machine $\mathcal{M}$ runs in polynomial time, then there is an index e such that $\mathcal{M}(x) = \mathcal{M}_e(x)$ for all x. Here we are using a infinite-repetitions assumption to make sure we can get an index with a sufficiently large k.

Using this simulator, we can build K_p in style. Alas, there is no reason why this set should be in $\mathbb{P}$: we are simulating machines of arbitrarily high polynomial running time. the simulator cannot be expected to run in some particular time $O(n^k)$.

Circuit Value Problem

Problem:	**Circuit Value Problem (CVP)**
Instance:	A Boolean circuit C, input value x.
Question:	Check if C evaluates to true on input x.

Obviously CVP is solvable in polynomial time (even linear time given a halfway reasonable representation).

There are several versions of CVP, here is a particularly simple one: compute the value of the "last" variable X_m given

$$X_0 = 0, \qquad X_1 = 1$$
$$X_i = X_{L_i} \diamond_i X_{R_i}, \quad i = 2, \ldots, m$$

where $\diamond_i = \wedge, \vee$ and $L_i, R_i < i$.

Theorem (Ladner 1975)

The Circuit Value Problem is $\mathbb{P}$-complete.

Sketch of proof. [†] For hardness consider any language A accepted by a polynomial time Turing machine $\mathcal{M}$.

We can encode the computation of the Turing machine $\mathcal{M}$ on x as a polynomial size circuit (polynomial in $n = |x|$): use lots of Boolean variables to express tape contents, head position and state.

Constructing this circuit only requires "local" memory; for example we need $O(\log n)$ bits to store a position on the tape.

The circuit evaluates to true iff the machine accepts.

□

[†] We will provide much more detail in the proof of the Cook-Levin theorem.

Alternating Breadth First Search

A long list of $\mathbb{P}$-complete problems in known, though they tend to be a bit less natural than $\mathbb{NP}$-complete problems.

For example, consider an undirected graph $G = \langle V, E \rangle$ where E is partitioned as $E = E_0 \cup E_1$.

Question: Given a start vertex s and a target t, does t get visited along an edge in E_0 if the breadth-first search can only use edges in E_0 at even and E_1 at odd levels?

Theorem

ABFS is $\mathbb{P}$-complete (wrto logspace reductions).

Consider a standard algorithm problem like Shortest Path: we are given a directed graph G whith positive edge weigths, a source vertex s and a target vertex t. The goal is to find a minimum cost path from s to t.

In any standard algorithm text you will find a statement along the lines of

Proposition

Dijkstra's algorithm runs in time $O(n^2 \log n)$ where $n = |V|$.

What does that mean in our framework, if anything?

First a general idea that we will use later.

Lemma

A k-tape TM can be simulated by a one-tape TM in quadratic time.

Proof. Suppose the k-tape machine $\mathcal{M}$ has tape alphabet Σ. Let $\Sigma_1 = \Sigma \times \{\rightarrow, |, \leftarrow\}$. The idea is that $(a, \rightarrow)$ indicates that the tape head is to the right, $(a, |)$ we are at the tape head, and similarly for $(a, \leftarrow)$

The simulator $\mathcal{M}'$ uses tape alphabet $\Gamma = \Sigma_1^k \cup \{\#, b\}$ where $\#$ is an endmarker and b the blank symbol: tape inscriptions look like

$$\#\boldsymbol{a}_1\boldsymbol{a}_2 \ldots \boldsymbol{a}_n\#$$

where $\boldsymbol{a}_i = (a_{i1}, \ldots, a_{ik})$.

Then $\mathcal{M}'$ operates as follows: to simulate a single step of $\mathcal{M}$, assume the head is originally at the left endmarker.

The machine then sweeps to the right, and collects information about the current symbol in each track (there $\mathcal{M}$'s tape head is for that tape).

Next, the machine sweeps back left and performs the necessary updates to all the tracks.

The first sweep is one-way, the second one may require some small local movements for the update steps.

So the simulator runs in quadratic time.

□

We are using an exponential size alphabet, but that's allowed.

Turing Implementation for Dijkstra

To implement Dijkstra's algorithm on a Turing machine we will need some sort of *array* of integer, for the data as well as the priority queue. The key operations are read and write:

```
x := A[i];        \\ read
A[i] := x;        \\ write
```

To simplify matters, we will use three tapes on the TM to represent each single array: one for the actual array, one for the index i, and one for the value x.

This is fine, we have just seen how to ultimately get rid of extra tapes, with a modest slow-down.

TM Arrays

The array tape looks like

$$\#a_0\#a_1\#\ldots\#a_{n-1}\#$$

where the numbers a_i are written in binary, and $\#$ is a special separator symbol.

For a read operation, we use the index i to find the ith separator, searching from the left. Then we copy the following binary number to the value tape.

For a write operation, we again search for the ith block. We then copy x over from the other tape, shifting the remainder of the tape contents according to the number of bits in x (compared to the previous number of bits).

Alternatively, we could determine the number k of bits of the largest number ever stored, and reserve k bits in each block right from the start.

Done!

Of course, this wrecks the crucial property of an array, constant time read/write access. But: It only takes $O(nk)$ steps to simulate the ops.

More generally, when we implement some graph algorithm such as Dijkstra's shortest path on a Turing machine, the whole computation slows down, but only by a polynomial amount. In particular, if our pseudo-code runs in polynomial time, then the TM also runs in polynomial time.

So we actually have a proof (sketch) that the shortest path problem is in $\mathbb{P}$ in the strict technical sense.

Easy versus Hard

Again: we are losing all fine-grained information, but this is fine if one is mostly interested in separating feasible from infeasible. For example, the key accomplishment of Dijkstra's algorithm is to avoid an exponential search over all paths, the TM still reflects this.

We don't care about the algorithms per se, we want lower bounds, in particular statements like

> Problem such-and-such **cannot** be solved in polynomial time.

A polynomial increase doesn't matter in this case: we might as well think about the computation in a luxurious RAM or a C program. If the Turing machine does not run in polynomial time, these won't either.

This makes it much easier to reason about the problem.