Counting

Klaus Sutner

Carnegie Mellon University

Spring 2024

Where Are We?

- We have a collection of space complexity classes, deterministic and nondeterministic, analogous to time complexity classes.
- Removing nondeterminism for space seems cheaper than for time.
- Particularly natural seem
  - logarithmic space
  - linear space
  - PSPACE-complete

Kuroda's Problems

Kuroda stated two open problems in his 1964 paper on linear bounded automata:

- Is nondeterminism in an LBA really needed?
- Are context-sensitive languages closed under complements?

The first problem is still open and seems to be very difficult.

But the second one has been solved, using an amazing technique.

Nondeterminism vs. Completeness

We have seen a number of PSPACE-complete problems, including:

- Quantified Boolean Formulae
- Context-Sensitive Recognition

The first has a natural deterministic algorithm in linear space.

The second one is also in linear space, but it seems to require nondeterministic choices: we don't know which reverse production to apply next, and where. And, it is utterly unclear how to eliminate nondeterminism without increasing space complexity.

Immerman-Szelepsényi

Kuroda's second problem was solved independently by two researchers in the late 1980s[†].

Theorem (Immerman-Szelepsényi, 1988)

Context-sensitive languages are closed under complement.

Just to be clear: for specific CSLs such as $\{ a^i b^i c^i \mid i \geq 1 \}$ one can often construct a CSG for the complement without too much trouble. In this case, e.g., we only have to worry about strings $a^i b^j c^k$ and make sure $i \neq j$ or $i \neq k$ or $j \neq k$.

The problem is that there appears to be no general method to convert a CSG to a CSG for the complement. And LBA don't help, either.

[†] Just like Friedberg/Muchnik and Cook/Levin.

Proof Strategy

Strangely, the way to tackle this problem is to take another look a a seemingly unrelated and obviously (?) harmless graph problem: our old friend reachability.

Problem:	**Graph Reachability**
Instance:	A digraph G, two nodes s and t.
Question:	Is there a path from s to t?

Of course, this can easily be solved in linear time using standard graph algorithms.

But we are here interested in expoiting nondeterminism to cut down the space requirement.

Recall: Prototype Reachability Algorithm

The strategy is clear, but there is a problem. We build the set $R \subseteq V$ of vertices reachable from s in G in stages.

Let's say $(u, v) \in E$ requires attention if $u \in R$ but $v \notin R$.

```
$R = \{s\}$;
while some edge $(u, v) \in E$ requires attention do
    add $v$ to $R$;
return $R$;
```

DFS and BFS are both instances of this strategy. Alas, these algorithms require linear space: we have to keep track of R. It's utterly unclear how we could reduce the space requirement.

How About Logarithmic Space?

As already mentioned, we can get down to logarithmic space if we are willing to go nondeterministic and give up on the usual true/false symmetry.

```
// path guessing $s \rightsquigarrow t$
$n = |V|$
$\ell = 0$
$x = s$
**while** $\ell < n - 1$ **do**
    **if** $x = t$ **then** return Yes
    guess an edge $(x, y)$
    $\ell$++
    $x = y$
**return** No
```

Rather strange, but perfectly correct.

Non-Reachability

Unfortunately, to deal with complements we really need the opposite problem: t is not reachable from s.

Problem:	**Graph Non-Reachability**
Instance:	A digraph G, two nodes s and t.
Question:	Is there no path from s to t?

This completely wrecks the pass guessing approch, there is nothing to guess for the negation.

It is a major surprise that Non-Reachability can also be handled in nondeterministic logarithmic space, though the logical complexity of the algorithm is substantially higher.

Counting Reachable Points

Let

$$R = \{\, x \in V \mid \text{ exists path } s \rightsquigarrow x \,\}$$

be the set of vertices reachable from s.

From the perspective of nondeterministic logarithmic space, R can be exponentially large (it may have cardinality up to n). But, it suffices to know just its cardinality to determine non-reachability.

Claim

Given the cardinality $r = |R|$, we can solve Non-Reachability in nondeterministic logarithmic space.

Deciding Non-Reachability

```
// non-reachability algorithm
$c = 0$                                // path counter
forall $x \neq t$ in $V$ do
    guess path $s \rightsquigarrow x$
    if path found
    then $c$++
return $c == r$
```

This works since $i = r$ means that all reachable vertices are different from t.

Surprising and nice, but useless unless we can actually compute r.

Onion Trick

Instead of trying to compute r directly, let

$$R_\ell = \{\, x \in V \mid \text{ exists path } s \rightsquigarrow x \text{ of length } \leq \ell \,\}$$

and set $r_\ell = |R_\ell|$.

Obviously $r_0 = 1$ and $r_{n-1} = r$.

So we only have to figure out how to compute r_ℓ, given $r_{\ell-1}$.

Counting

```
// inductive counting algorithm
// input: $\ell$, current count $\rho$ (supposedly $r_{\ell-1}$)
// output: next count $\rho'$ (supposedly $r_\ell$)

$\rho' = 0$
**forall** $x$ in $V$ **do**
    $b = 0$                        // may increment to 1
    $c = 0$                        // path counter
    **forall** $y$ in $V$ **do**
        guess path $s \rightsquigarrow y$ of length $< \ell$
        **if** path found
        **then** $c{+}{+}$
        **if** $y = x$ or $(y, x) \in E$
        **then** $b = 1$
    **assert** $c == \rho$             // all paths found
    $\rho' = \rho' + b$
**return** $\rho'$
```

Quoi?

Suppose ρ is the correct value of $r_{\ell-1}$.

Since we check that $c = \rho$, we know that all paths $s \rightsquigarrow y$ have been guessed correctly.

But then we have properly found and counted all paths of length at most ℓ: we just have to add one more edge. Hence, the output ρ' is indeed r_ℓ.

Note that the algorithm requires only storage for a constant number of vertices, so logarithmic space suffices, plus nondeterminism to guess the right path from s to y. As before, there are no false positives, but the computation may crash.

Nondeterministic Functions?

The counting algorithm is heavily nondeterministic: we have to guess multiple paths $s \rightsquigarrow y$ (which would never work out if we were to flip a coin to make nondeterministic decisions).

But note the **assert** statement: if we make a mistake, the whole computation crashes. This means in particular that no function value is produced on any of these branches.

The branches where all the guesses are correct all produce the exact same value ρ. So this is a fairly tame version of nondeterminism.

Back to Immerman-Szelepsényi

The theorem now follows easily.

For suppose $\mathcal{M}$ is some LBA (nondeterministic!) which accepts some CSL L. We would like to build another LBA that accepts $\Sigma^\star - L$.

Consider some input $x \in \Sigma^\ell$ and the usual computation graph $\mathfrak{C}(\mathcal{M}, x)$. Clearly the size of $\mathfrak{C}(\mathcal{M}, x)$ is bounded by $n = c^\ell$ for some constant $c > 1$.

Non-acceptance of $\mathcal{M}$ now translates into a non-reachability problem for $\mathfrak{C}(\mathcal{M}, x)$, and we have just seen that this task can be handled by another nondeterministic LBA; done.

Full Immerman-Szelepsényi

We can rephrase the Immerman-Szelepsényi result in terms of space complexity classes:

Theorem

For any reasonable function $s(n) \geq \log n$ *we have* $\mathrm{NSPACE}(s(n)) = \text{co-}\mathrm{NSPACE}(s(n))$.

In particular $\mathbb{NL} = \text{co-}\mathbb{NL}$ *and* $\mathrm{NSPACE}(n) = \text{co-}\mathrm{NSPACE}(n)$.

This follows easily by considering the computation graphs of the corresponding Turing machines.

YABA

Yet another Boolean algebra: the context-sensitive languages form a Boolean algebra, just like the regular and decidable ones.

But the following result is somewhat surprising:

Theorem

The Boolean algebras of regular, context-sensitive and decidable languages are all isomorphic.

So wrto Boolean operations only, there is no difference between these language classes. The proof requires a bit of lattice theory; we'll skip.

Size Matters

One can show that the following problem about estimating the size of a power automaton produced by the Rabin-Scott determinization construction is PSPACE-complete.

Problem:	**Power Automaton Size**
Instance:	An NFA $\mathcal{A}$ and a bound β.
Question:	Does pow$(\mathcal{A})$ have size at least β?

Again by Immerman-Szelepsényi, the corresponding problems $|\text{pow}(\mathcal{A})| \leq \beta$, $|\text{pow}(\mathcal{A})| = \beta$, $|\text{pow}(\mathcal{A})| \neq \beta$ can all be solved by an LBA.

Polynomially Computable Functions

We are interested in computing functions

$$f : \mathbf{2}^\star \to \mathbf{2}^\star$$

Definition

The class $\mathbb{FP}$ is the collection of all functions $f : \mathbf{2}^\star \to \mathbf{2}^\star$ that can be computed in polynomial time (by an ordinary deterministic Turing machine).

The intent here is that machine $\mathcal{M}$ on input x performs a polynomial time computation and writes the appropriate output in binary on the output tape.

Writing numbers in binary as usual, addition and multiplication are in $\mathbb{FP}$ by the standard algorithms, but exponentiation is not. Here are two examples where a combinatorial problem leads to a polynomial time solvable counting problem.

Search To Counting

Any search problem

Problem:	$\mathit{\Pi}$
Instance:	Some instances x.
Solution:	Some solution $z \in \mathsf{sol}(x)$, or NO.

can always be turned into a counting problem

Problem:	$\#\mathit{\Pi}$
Instance:	Some instances x.
Solution:	The number of solutions $\lvert\mathsf{sol}(x)\rvert$.

There is also the associated decision problem

Problem:	$D\mathit{\Pi}$
Instance:	Some instances x.
Question:	Is $\mathsf{sol}(x) \neq \emptyset$?

Connections

We write $\#\Pi(x)$ for the number of solutions.

The counting problem is always at least as hard as the associated decision problem:

$$\text{there is a solution for } x \iff \#\Pi(x) > 0$$

So if the decision problem is, say, $\mathbb{NP}$-complete we should not expect the counting problem to be in $\mathbb{FP}$.

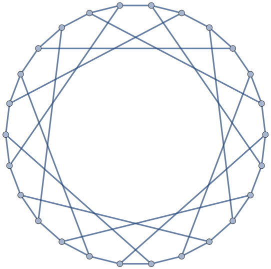
But: Even if the decision version is trivial, $\#\Pi$ may be challenging to unmanageable.

And it is not clear a priori what the relationship between Π and $\#\Pi$ is.

Spanning Trees

Here is a classical example: given an undirected connected graph $G = \langle [n], E \rangle$ consider its spanning trees. The decision problem here is quite simple: we can construct a spanning tree using DFS or BFS.

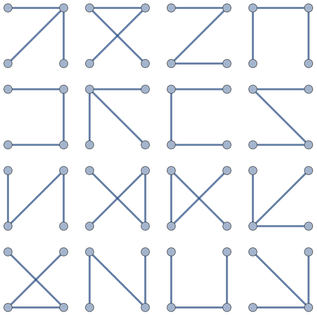
But in the associated counting problem we want to determine the number of **all** spanning trees.



Yes, the answer is $812,017,791$.

Cayley's Theorem

Of course, some cases are easy: Cayley has shown that a complete graph on n points has n^{n-2} spanning trees.



The Laplacian of a Graph

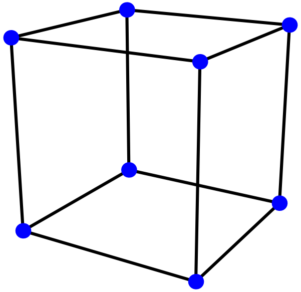
Define the Laplacian of G to be the $n \times n$ matrix L with entries

$$L(i,j) = \begin{cases} \deg(i) & \text{if } i = j, \\ -1 & \text{if } i \neq j,\ (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Thus the Laplacian is the difference $D - A$ of the degree matrix and the adjacency matrix.

This may seem like a rather peculiar construct, but it does have interesting computational properties. In general, the algebraic properties (characteristic polynomial, eigenvalues, eigenvectors) of matrices associated with graphs are quite important for various algorithms (spectral graph theory).

Example: Cube



Laplacian:

$$\begin{array}{rrrrrrrr}
3 & -1 & -1 & 0 & -1 & 0 & 0 & 0 \\
-1 & 3 & 0 & -1 & 0 & -1 & 0 & 0 \\
-1 & 0 & 3 & -1 & 0 & 0 & -1 & 0 \\
0 & -1 & -1 & 3 & 0 & 0 & 0 & -1 \\
-1 & 0 & 0 & 0 & 3 & -1 & -1 & 0 \\
0 & -1 & 0 & 0 & -1 & 3 & 0 & -1 \\
0 & 0 & -1 & 0 & -1 & 0 & 3 & -1 \\
0 & 0 & 0 & -1 & 0 & -1 & -1 & 3
\end{array}$$

Kirchhoff’s Theorem

Theorem

The number of spanning trees in an undirected connected graph $G = \langle [n], E \rangle$ *is*

$$\#(\textit{SpanTree}, G) = \lambda_1 \lambda_2 \ldots \lambda_{n-1}/n$$

where the λ_i *are the non-zero eigenvalues of the Laplacian of* G*.*

In the cube example above the eigenvalues are $6, 4, 4, 4, 2, 2, 2, 0$, so there are 384 spanning trees.

This generalizes Cayley's formula for the complete graph. For example, for $n = 5$ the Laplacian looks like

$$\begin{pmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 4 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & -1 & 4 \end{pmatrix}$$

It has nonzero eigenvalues $5, 5, 5, 5$.

Exercise

Prove that these matrices have the right eigenvalues.

Kirchhoff's Theorem, Algorithmically

The version stated above is very elegant, but not particularly attractive computationally: we can easily compute the characteristic polynomial $|A - Ix|$ but then we need to compute its roots to get at the eigenvalues (all integers).

Here is a better version: the number of spanning trees is

$$\#(\mathsf{SpanTree}, G) = \det\big((L)_{ii}\big)$$

Here L is the Laplacian, and $(L)_{ii}$ denotes a cofactor, the submatrix obtained by removing the ith row and ith column. In fact, this is the original version of the theorem.

Hence we can compute the number of spanning trees in polynomial time, $\#(\mathsf{SpanTree}, G)$ is in $\mathbb{FP}$.

Another Success Story

Around 1960 Kasteleyn discovered an amazing way to calculate the number of perfect matchings in a planar graph.

Of course, the existence of a perfect matching is well-known to be in polynomial time, but it is by no means clear that once can count all such matchings without brute-force enumeration.

As with spanning trees, the solution relies on linear algebra and determinants.

Thus, counting spanning trees and counting perfect matchings in a planar graph are in $\mathbb{FP}$.

Unfortunately, it is rather rare that counting problems associated with standard combinatorial problems are solvable in polynomial time.

Nondeterministic Counting Machines

So far we mostly talked about functions computable in deterministic polynomial time (except for counting in Immerman-Szelepsényi). Is there a more systematic way to compute functions using nondeterminism?

The problem is to deal with potentially different results on different branches in the computation tree. Here is a outside-of-the-box definition that produces good results.

Definition

A counting Turing machine (CTM) is a nondeterministic Turing machine $\mathcal{M}$, normalized to 2 choices. For any input x, we define the output of $\mathcal{M}$ on x to be the number of accepting computations of $\mathcal{M}$ on x.

$\#\mathbf{P}$ is the class of all functions that are computable by a polynomial time CTM in this sense.

Quoi?

This definition is rather surprising, we would expect the result to be written on a output tape, somehow. Instead we use a nondeterministic acceptor, and count accepting computations from the outside.

The problem is that nondeterministic transducers would produce multiple possible "results." Requiring that all results are the same ruins nondeterminism, so we would need to select one of the many results as the true answer.

One could try to take the min/max/average or some such, but none of these attempts produce a good theory.

Note that SAT is a great motivation: it is easy to build a nondeterministic machine that counts the number of satisfying truth assignments in this sense: first guess the assignment, then verify correctness.

Examples

We have reasonable closure properties: e.g., if f and g are in $\#\mathbf{P}$ then so is $f + g$.

To see this, build a machine $\mathcal{M}$ that adds one more nondeterministic step that picks either the machine $\mathcal{M}_f$ for f or the machine $\mathcal{M}_g$ for g.

Clearly $\#_{\mathcal{M}}(x) = \#_{\mathcal{M}_f}(x) + \#_{\mathcal{M}_g}(x)$.

Exercise

How would you handle the product $f \cdot g$?

Exercise

Find some other closure properties of the class of $\#\mathbf{P}$ functions.

Witness Version

Recall that $\mathbb{NP}$ can be defined either in terms of nondeterministic polynomial machines, or in terms of polynomially bounded projections on polynomial time decidable relations. The same applies here.

Definition

A polynomial time decidable relation $R(z, x)$ is polynomially balanced if there is some polynomial p such that $R(z, x)$ implies $|z| \leq p(|x|)$.

It is easy to check that $\#\mathbf{P}$ is the class of all counting problems associated with polynomially balanced relations R: we need to compute

$$f(x) = |\{\, z \mid R(z, x)\,\}|$$

Example: Satisfiability

Consider SAT, satisfiability of Boolean formulae. Here is the counting version.

Problem:	**#Satisfiability**
Instance:	A Boolean formula in CNF.
Solution:	The number of satisfying truth assignments.

It is straightforward to construct a polynomial time CTM that counts the number of satisfying truth assignments of a given formula: guess an assignment, and verify that the assignment is a model of the formula.

So if $\#$SAT is in $\mathbb{FP}$ then $\mathbb{P} = \mathbb{NP}$.

This is no big surprise, a hard decision problem will produce hard counting problems.

Let's pin down more carefully what we mean by hard counting.

Hardness

In order to obtain a notion of hardness we need to define appropriate reductions. Here we return to polynomial time Turing reductions.

Definition

A counting problem is $\#\mathbf{P}$-hard if there is a polynomial time Turing reduction from any other problem in $\#\mathbf{P}$ to it. If the problem is also in $\#\mathbf{P}$ it is said to be $\#\mathbf{P}$-complete.

Note that we are dealing with two function problems here, we want to compute $f : \mathbf{2}^\star \to \mathbf{2}^\star$ given $g : \mathbf{2}^\star \to \mathbf{2}^\star$.

A polynomial time Turing machine can evaluate $g(z)$ for various values of z and use the results to compute $f(x)$.

It follows that if any $\#\mathbf{P}$-hard problem can be solved in deterministic polynomial time the whole class can be so solved.

Parsimonious Reductions

In practice, there is another more restrictive notion of reduction that seems to apply to all interesting counting problems.

Definition

A parsimonious transformation from a problem X to a problem Y is given by a polynomial time computable function f such that $\#X(x) = \#Y(f(x))$.

In other words, a parsimonious transformation preserves the number of solutions when translating instances of one search problem into another.

Of course, a parsimonious reduction is a special case of a polynomial time Turing reduction: there is only one query and it produces the final result.

As it turns out, many of the polynomial reductions in the study of $\mathbb{NP}$ problems naturally translate into parsimonious transformations.

Example: Satisfiability

Theorem

$\#$SAT *is* $\#\mathbf{P}$*-complete.*

Proof.

This argument is similar to the Levin-Cook theorem: we can translate the computations of a polynomial time counting Turing machine into satisfying truth assignments of a suitable Boolean formula.

It is entirely natural to construct the formula in a way that makes the reduction parsimonious: the number of computations corresponds exactly to the number satisfying truth assignments.

$\square$

Counting Cycles

Let $\#\mathrm{CYCLE}$ be the problem of counting all simple cycles in a directed graph. Of course, the decision version here is trivially solvable in linear time.

Theorem

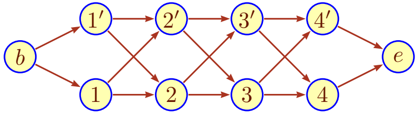
If $\#\mathrm{CYCLE}$ has a polynomial time solution then $\mathbb{P} = \mathbb{NP}$.

Proof.

We will show how to check for Hamiltonian cycles: given a ugraph G on n points we construct a digraph H such that G is Hamiltonian iff H has at least n^{n^2} simple cycles.

Proof, Contd

Replace every edge (b, e) of G by the gadget shown below:



The gadget has depth $m + 1$ where $m = \lceil n \log n \rceil$; hence there are 2^m paths from begin to exit. Hence a simple cycle of length ℓ in G yields $(2^m)^\ell$ simple cycles in H.

If G is Hamiltonian, then H has at least $(2^m)^n \geq n^{n^2}$ cycles.

If not, then this cycle count is at most $(2^m)^{n-1} \cdot n^{n-1} \leq n^{(n+1)(n-1)} < n^{n^2}$.

□

Parsimonious Problems

A closer look at the reductions we gave from 3SAT to Vertex Cover, and from there to Hamiltonian Cycle reveals that all these reductions are parsimonious: for example, a satisfying truth assignment correspond precisely to one vertex cover of a certain size.

So we have:

Theorem

$\#$VC *is* $\#\mathbf{P}$*-complete.*

Theorem

$\#$HAMCYC *is* $\#\mathbf{P}$*-complete.*

Again, parsimonious reductions are quite common; in fact it takes a bit of effort to produce natural, non-parsimonious ones. Try.

Permanents

It is unsurprising that $\mathbb{NP}$-complete problems should give rise to $\#\mathbf{P}$-hard counting problems.

However, there are examples of problems whose decision version is polynomial time, but whose counting version nonetheless is $\#\mathbf{P}$-hard.

The permanent of an n by n matrix A is defined by

$$\mathsf{perm}(A) = \sum_{\sigma} \prod_{i} A(i, \sigma(i))$$

where σ ranges over all permutations of $[n]$.

So the permanent differs from the determinant only in that the sign of the permutation is missing.

Permanents are Hard

Theorem (L. Valiant 1979)

Computation of the permanent of an integer matrix is $\#\mathbf{P}$-complete. The problem remains $\#\mathbf{P}$-complete even when the matrix is binary.

The corresponding decision problem is to check for the existence of a permutation σ such that $\prod_i A(i, \sigma(i)) = 1$, i.e., such that $A(i, \sigma(i)) = 1$ for all i. But that is equivalent to determining the existence of a perfect matching in a bipartite graph, a problem well-known to be in $\mathbb{P}$.

Also note that the rather similar problem of computing the determinant of A is easily solved in cubic time (there are asymptotically better methods).

More Problems

Valiant established the $\#\mathbf{P}$-hardness of a number of other problems.

Problem:	**#Minimal Vertex Cover**
Instance:	An undirected graph G.
Solution:	The number of minimal vertex covers.

Problem:	**#Maximal Clique**
Instance:	An undirected graph G.
Solution:	The number of maximum cliques.

The hardness of counting minimal vertex covers and maximal cliques is fairly obvious: the corresponding decision problems are $\mathbb{NP}$-hard after all.

How About These Two?

Problem:	**#Monotone 2-Satisfiability**
Instance:	A monotone formula in 2-CNF.
Solution:	The number of satisfying truth assignments.

Problem:	**#Perfect Matching**
Instance:	A bipartite graph G.
Solution:	The number of perfect matchings.

Monotone here means: no negated variables. Might seem fairly easy. Alas . . .

Surprise

Recall that a monotone Boolean formula in 2-CNF has the form

$$\varphi = (u_1 \vee v_1) \wedge (u_2 \vee v_2) \wedge \ldots \wedge (u_s \vee v_s)$$

where all the u_i and v_i are variables from some set $\{x_1, x_2, \ldots, x_r\}$. All these formulae are trivially satisfiable; it's only the counting part that is difficult.

Finding a perfect matching in a bipartite graph is also easy (though harder than Monotone 2-SAT). Again, counting is the difficult part.

Still, both these counting problems are $\#\mathbf{P}$-hard.

Example: Network Reliability

There is a large group of problems dealing with network reliability: a network is represented by a (directed or undirected) graph and one would like to understand how the network behaves under random failures. There are two basic events:

- Edge failure: an edge disappears from the graph.
- Vertex failure: a node disappears from the graph, together with all incident edges.

The problem is that these events occur at random and one would like to understand how the quality of the networks is degraded by such random events.

Measuring Reliability

It is far from clear exactly what properties of a network one should be interested in from a a reliability perspective. Here are some ideas:

- connectivity
- connectivity between s and t
- number of components

For example, given a pair of critical nodes s and t, we would like to make sure that there is at least one path from s to t after failures have occurred.

Or we would like the number of connected components after failure to be small; ideally, just one or two components.

Residual Node Connectedness Reliability

Here is one popular reliability model: edges do not fail, but nodes do. Node failures occur with uniform probability p and independently of each other.

The residual node connectedness reliability R_G of G is the likelihood that, after failure, the remaining graph is still connected.

We can write the reliability in terms of a polynomial in the failure probability p:

$$R_G(p) = \sum_{k=0}^{n} S_k(G)\, p^{n-k}\, (1-p)^k,$$

Here $S_k(G)$ is the number of connected k-node subgraphs of G. For convenience, let $S_0(G) = 0$.

Yet another counting problem: we have to determine the number of such subgraphs.

Counting Connected Subgraphs

To simplify matters a bit we show that $R_G(1/2)$ is hard to compute, even when the underlying graph is not too complicated. Let

$$S(G) = 2^n \, R_G(1/2) = \sum_{k=0}^{n} S_k(G)$$

A split graph is a graph where the vertex set is partitioned into an independent set and a clique, possibly connected by some edges.

Theorem (KS 1991)

It is $\#\mathbf{P}$-complete to compute $S(G)$ for split graphs G.

Proof

We will produce a reduction from $\#$Monotone-2-SAT, a well-known $\#\mathbf{P}$-complete problem.

Consider a monotone Boolean formula Φ in 2-CNF with variables $x_1, \ldots, x_n$ and clauses $C_1, \ldots, C_m$. We may safely assume that every variable occurs in at least one clause: this can be checked in polynomial time, and, if it fails, we simply multiply by 2^k where k is the number of useless variables.

Then $n \leq m/2$ and $m \leq \binom{n}{2}$, so we can think of m as the size of Φ.

Contd

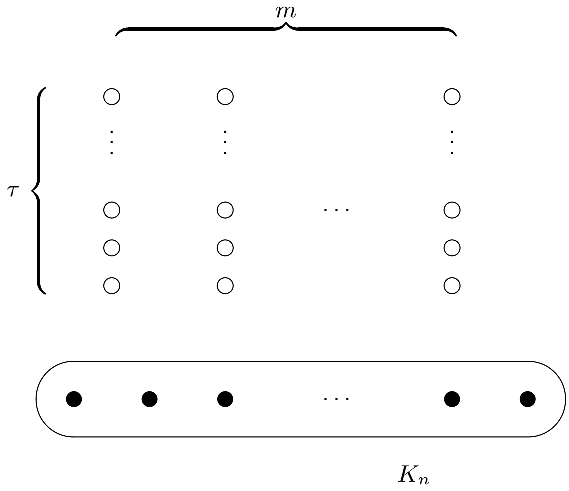
For any $\tau \geq 1$ define a graph G^τ as follows:

- truth vertices: $X = \{\, x_i \mid i \in [n] \,\}$
- clause vertices: $C = \{\, C^i_j \mid i \in [\tau], j \in [m] \,\}$,
- clause edges: x_k is adjacent to C^i_j iff variable x_k occurs in clause C_j
- clique edges: turn X into a clique.

By construction, G^τ is a split graph on $N = n + m\tau$ vertices. Moreover, we can construct G^τ in time polynomial in m and 0^τ.

Note that representing each clause multiple times is a trick similar to the proof for 3DM.

The Split Graph



Proof, Contd

What do the connected subgraphs H of G^τ look like? We can think of $H \cap X$ as a truth assignment α_H: the variables in H are set to true, all others to false.

Idea: Organize connected subgraphs according to weight.

Here the weight of a truth assignment $\alpha : X \to \mathbf{2}$ is

$$w(\alpha) = \text{ number of clauses of } \Phi \text{ satisfied by } \alpha.$$

Also let T_k be the number of satisfying truth assignments of weight k, $0 \le k \le m$.

Proof, Contd

The trivial case is weight 0: then H is a one-point subset of C (recall that all variables appear somewhere).

For weight $k \geq 1$, we claim that

$$H = (H \cap X) \cup S$$

where S is an arbitrary subset of $\{\, C_j^i \mid \alpha_H \models C_j, i \in [\tau] \,\}$

Hence there are $T_k 2^{k\tau}$ such connected subgraphs.

That's it, there are no other connected subgraphs.

Proof, Contd

Now consider the degree m polynomial

$$P(z) = m\tau + \sum_{1 \le k \le m} T_k \, z^k$$

Then $S(G^\tau) = P(2^\tau)$, and we can use polynomial interpolation to compute the coefficients by choosing $m+1$ sufficiently small values of τ. One can check that the whole computation is duly polynomial time.

But T_m is the number of satisfying truth assignments of Φ, and we are done.

$\square$