Foundations

Turing machines

- A Turing machine consists of: a finite set of states Q, a finite tape alphabet Γ, a transition function $\delta: Q \times \Gamma \mapsto Q \times \Gamma \times \{L, S, R\}$, an initial state q_0, and a set $Q_{halt} \subseteq Q$ of states that cause the machine to stop when entered
  - A transition $((q_i, a), (q_j, b, m))$ means if in state q_i and read a in the current cell, then overwrite the current cell with b, and move head m (left/stay/right) and change to state q_j
- Unless stated otherwise: there is a single tape and a single head, $\Gamma = \{0, 1, \sqcup\}$, the tape extends infinitely to the right and to the left
  - However, altering any of these does not affect the computational power within polynomial factors
- Every TM can be encoded by a finite bit-string. A universal Turing-machine takes as input $<\alpha , w>$ simulates running M_α on w where M_α is the TM encoded by α

Problems

- A problem = a function f: {0, 1}* ↦ {0, 1}*, that is that its inputs and outputs are bit-strings of finite but unbounded length
- A Turing machine M computes a function f in T(n)-time iff for every x ∈ {0, 1}* initialising M with x causes M to halt after at most T(|x|) steps and it does so with f(x) written on the tape
- A Turing machine M computes a function f if there exists a function T such that M computes f in T(n)-time
- Deduce that <u>if ∃x: {0, 1}*: M does not halt on x, then M cannot be said to compute any function</u>
- A decision problem = a problem with range {0, 1} i.e. a problem where the output is a single bit
    - Deduce that a decision problem can be entirely described by the set of "yes" instances (a language) — $L_f = \{x: f(x) = 1\}$
    - A TM for a decision problem may halt with f(x) written on its tape or halt with anything on its tape but in a designated state either q_{accept} or q_{reject}

Robustness of Turing machines

- Proposition: If f is computable in time T(n) by a TM M with alphabet Γ, then f is computable in time O(T(n)log(|Γ|)) by a TM M' with alphabet {0, 1, ␣}
  Proof: Bijectively assign each character in Γ a number in {0, …, |Γ| − 1}. This leaves use |Γ| to use as a separator. At each step write the character to be written as its numerical encoding in binary followed by the separator of the binary encoding of |Γ|. As Γ is finite, only finitely many new states need to be added to keep track of which binary we have read so far for each character. We are expanding each symbol in the original alphabet into ceil(log(|Γ|) + 1) symbols so this is the number of writes (and reads and moves to return the head to the correct position) we now have to do for one step. Thus, each of the T(n) steps takes O(log |Γ|) time giving the required result.
- Proposition: A TM with k-tapes that computes a function f in T(n)-time can be simulated by a single-tape TM to compute f in $O([T(n)]^2)$ time
  Proof: AWLOG the tapes are bounded to the left (pre-process the tapes to place a marker ⊢ at the left edge). Use each ith cell of the single-tape machine as the qth cell of the rth tape of the multi-tape machine where q and r are the quotient and remainder repseicely when i is divided by k. By the previous proposition we can extend the alphabet to contain each character with and without accent $\hat{}_i$ for each head i to denote that the ith head is over it.
  To carry out a single step of the multi-tape machine: scan the tape to find each of the k heads and move through states to remember them all (we only need finitely many additional states as the tape alphabet and the number of heads are each fixed and finite). Hence, we do O(T(n)) work for each of the T(n) steps of the original machine as required.

Oblivious Turing machines

- **An oblivious TM is a TM for which the head movements depend only on the length of the input (not the content of the input)**
  - *In security we may want to avoid leaking data to an attacker who can see our memory access patterns*
- Proposition: Let M be a TM that decides a language L in time O(T(n)). Then there exists an oblivious TM that decides L in time $O((T(n))^2)$.
  Proof sketch: At every step read the entire tape (between maintained markers for the start and end of the used portion), reading (and possibly writing) the cell M would have read when you come across it. The markers can move apart at most linearly with the number of steps, and the number of steps is T(n), $\sum$ O(i) from i = 0 to i = T(n) is in $O((T(n))^2)$ as required.

Computability

HALT is undecidable

- Theorem: There exist uncomputable decision problems
  Proof: The set of all decision problems (set of all subsets of $\{0, 1\}^*$) is uncountably infinite but the set of all descriptions of TMs ($\{0, 1\}^*$) is countably infinite. Hence, there are actually uncountably many uncomputable problems.
  Corollary: The probability of a randomly chosen decision problem being decidable is exactly zero, as decision problems biject to the reals and computable problems (i.e. TMs) biject to naturals
- Lemma: Let $f(\alpha) = 0$ if $M_\alpha(\alpha) = 1$, 1 if $M_\alpha(\alpha) = 0$ or M_α does not halt on α. Then, f is not computable
  Proof: Assume for the sake of contradiction that there exists a TM M that computes f. Then, $\forall x$, M halts with f(x) on the tape. Consider x = $\underline{M}$ where $\underline{M}$ is the encoding of M.
  Case halts with 0 on the tape: By definition of f, $f(\underline{M}) = 1$ so did not have f(M) on tape?!
  Case halts with 1 on the tape: $f(\underline{M}) = 0$?!
- Theorem: Let $HALT(\langle\alpha, x\rangle) = 1$ if M_α halts on x, 0 otherwise. HALT is not computable
  Proof: We will show that if HALT was decidable then the lemma would have been false. Construct the following TM: Run $M_{HALT}(\langle\alpha, \alpha\rangle)$. By our hypothesis that HALT is decidable, M_{HALT} halts on every input (and does so with the correct output). If $M_{HALT}(\langle\alpha, \alpha\rangle) = 0$ (M_α does not halt on α) , we return 1 in accordance with f. Otherwise, M_α does halt on α and so we are safe to simulate $M_\alpha(\alpha)$ and return the negation of the result.

Oracle machines

- An oracle machine (OM) is a TM equipped with an additional tape called the oracle tape and additional states q_{ask}, q_{yes}, q_{no}. An OM is constructed with access to an oracle for a particular problem f. When the OM enters the state q_{ask}, the machine moves into either q_{yes} or q_{no} based on the answer to f for the contents of the oracle tape. The oracle is assumed to be able to compute answers to f in a single step regardless of the true complexity or even computability of f
- *There is a hierarchy of undecidability: An OM with access to an oracle for HALT can solve undecidable problems about TMs (specifically all Turing-recognizable problems (including many open problems in mathematics) become Turing-decidable; languages that are neither recognizable nor co-recognizable (e.g. ∀-ACCEPTANCE) remain undecidable) but cannot decide whether itself will halt (HALT-OM) and so still cannot decide ∀-ACCEPTANCE, an OM with an oracle for HALT-OM (can trivally extend to include HALT as well etc) can decide HALT-OM (and so ∀-ACCEPTANCE) but not HALT-OM-OM but an OM with an oracle for HALT-OM-OM can etc*

Cook reductions

- **Defn: A problem f is Cook-reducible to a problem g ($f \leq^C g$) if there exists an oracle machine M with access to an oracle g that computes f**
- Deduce that "If $f \leq^C g$ and g is computable, then f is computable" and so "If $f \leq^C g$ and f is not computable, then g is not computable"

Rice's Theorem

- **Rice's theorem:** Let R be the set of functions which map $\{0, 1\}^*$ into $\{0, 1\}^* \cup \{\perp\}$ where $\perp$ denotes divergence which obey the restriction that they can be implemented by a Turing machine. $\forall C: C \subset R$ and $C \neq \varnothing$; $\{\alpha \mid \exists f \in C: M_\alpha$ corresponds to f} is undecidable. In natural language, **the decision problem "Given a description of a Turing machine α, is that TM in a class <u>of problems</u> C" is undecidable if there exists at least one TM that is in C and at least one TM that is not in C**
  - If a class relates to the inner workings of particular TMs instead of any TM that solves a particular problem, Rice's theorem is not applicable
- Intuition: For each TM that does solve a problem, there is a TM that would do the same operations but enters an infinite loop first and so does not actually solve the problem. To be able to distinguish between these in general requires us to decide the halting problem which is known to be undecidable

Proof of Rice's Theorem

Assume for the sake of contradiction that Rice's theorem is false, we will show that then the halting problem would be decidable. Pick an arbitrary class C for which Rice's theorem applies and let M_R be a decider for the Rice's theorem decision problem for C.
Let <α, x> be an arbitrary instance of HALT. Let u(x) = a TM that disregards its input and enters an infinite loop.
Pick arbitrary $f_p \in C$, $f_n \notin C$ (using the restrictions on C).
Construct a OM M that takes an input y, simulates $M_\alpha(\underline{x})$ and disregards the output, and uses the oracle for M_R to check whether $u \in C$ to decide what to do next:
If $u \notin C$: Return the result of simulating $M_{fp}(y)$.
Otherwise $u \in C$: Return the result of simulating $M_{fn}(y)$.
Construct a OM N that simulates M(y), and uses the oracle for M_R to check whether $u \in C$ and whether $M \in C$. If $u \notin C$, have N return bool($M \in C$). Otherwise $u \in C$, have N return ¬bool($M \in C$).
Denote the one of fp or fn (based on whether $u \in C$) that we actually use f. **All that M does in terms of output is return the output of an f such that f has the opposite membership property in C to the infinite loop u, however before this it does the busy work of simulating the function application we want to check the halting behaviour of. All that N does is checks the membership property of M in C.**
If the function we are checking halts, M behaves just like f and so will have the same membership property in C as f (the opposite to u). If it does not, the M will behave just like u (as it will never reach the f section) and so will have the same membership property in C as u (the opposite to f). Hence, by comparing the membership property of M in C to that of f, we can decide our arbitrary HALT instance?!

Using Rice's Theorem

- HALTALWAYS = {α : M_α halts on all inputs} is undecidable as this corresponds to deciding whether $M_\alpha \in C$ = {f ∈ R: ∀x. f(x) ≠ ⊥}. It is obvious that **$C \subseteq R$**. **$C \neq R$** as the TM that enters an infinite loop on every input is in R but not in C. **$C \neq \varnothing$** as the TM that disregards its input and returns a constant is in C. **Thus, by Rice's theorem** HALTALWAYS is undecidable

Self-references

- **Recursion theorem: Let P be a Turing machine that computes an arbitrary function f. Then, there exists a Turing machine M that computes a function r such that ∀x. r(x) = f(<x, M>)**
  Corollary: Turing Machines can be assumed to have access to their own description
- This is useful for succinct proofs by contradiction of undecidability for problems which Rice's theorem does not fit to
- Prop: L = {<α, x> | M_α accepts x} is undecidable
  Proof: We cannot apply Rice's theorem as this problem is about machines not problems. We can however use the recursion theorem. Assume for the sake of contradiction there exists a decider D for L. Construct a TM M that takes an input x and simulates D(<$\underline{M}$, x>) where $\underline{M}$ is a description of M (accessible to M thanks to recursion theorem) and returns the negation. As M checks whether D says M will accept x and then does the opposite, whatever D says will be wrong for M and so L was not actually decidable?!

NP

P

- **$f(n) \in O(g(n))$ iff $\exists\ C, n_0 \in \mathbb{N}: \forall\ n \in \mathbb{N}_{\geq n_0}. f(n) \leq C*g(n)$ iff $\exists\ C: \mathbb{R}: \lim_{n \to \inf} f(n)/g(n) = C$**
- L ∈ P iff ∃k: There exists a decider for L that runs in time $O(n^k)$ where n is the length of the input
- **A language L is (polynomial-time) Karp-reducible to a language L′ iff there exists a (polynomial-time) computable function f such that $\forall x.\ x \in L \Leftrightarrow f(x) \in L'$**
  - **We denote X polynomial-time Karp reduces to Y as $X \leq_P Y$**
- **A Cook reduction allows multiple oracle calls and post-processing of oracle answers, whereas a Karp reduction only allows one oracle call and no post-processing**
  - A Karp reduction is a Cook reduction with additional constraints, but it believed that some Cook reductions cannot be converted to Karp reductions
- **Prop: If $X \leq_P Y$ and $Y \in P$, then $X \in P$**
  Proof:
  Let f be the polynomial time reducing function.
  Compute f(x), this takes polynomial time. Then, return bool(f(X) ∈ Y). Checking if f(x) ∈ Y takes polynomial time as Y ∈ P

NP

- **$L \in NP$ iff $\exists k, m: \exists M$: M is a decider that runs in time $O(n^k)$ and $(\forall x \in \{0, 1\}^*. x \in L \Leftrightarrow \exists w \in \{w \in \{0, 1\}^*: |w| \in O(n^m)\}: M(\langle x, w \rangle)$ accepts)**
  - As the certificate is polynomial length in the input to the problem and the verifier is polynomial time in the length of its input (the certificate) and the set of polynomials is closed under composition, the verifier is polynomial time in the length of the input to the problem
- $L \in P \Leftrightarrow \neg L \in P$, but $L \in NP$ does not necessarily mean $\neg L \in NP$ because the negation flips the quantifiers!
- Thm: $P \subseteq NP$
  Proof: Pick arbitrary $L \in P$, then there exists a poly-time decider D for L. We can use any certificate of polynomial length (e.g. a single constant symbol); the certifier will discard the certificate and simulate D.
- *Ladner's Theorem: If $P \neq NP$, then NP-INTERMEDIATE = NP\(P U NP-COMPLETE) is not empty*

PRIMES ∈ NP

- Omitted due to time constraints

NP-Complete

- **$L \in$ NP-Hard iff $\forall M \in NP.\ M \leq_P^C L$**
  - **Corollary: $(\exists N \in$ NP-HARD: $N \leq_P^C L) \Rightarrow L \in$ NP-HARD**
  - Allowing Cook reductions allows optimization problems to be included as well as decision problems
- **$L \in$ NP-Complete iff $\forall M \in NP.\ M \leq_P L$ <u>and $L \in NP$</u>**
  - Note that in this module we require the use of Karp reductions instead of taking NP-Complete = NP-Hard $\cap$ NP

Cook-Levin Theorem

Cook-Levin Theorem: SAT ∈ NP-COMPLETE

Proof sketch: It is easy to see that SAT ∈ NP but proving the universal quantifier will be non-trivial.

Deciding a particular instance of a particular NP problem is equivalent to deciding whether there exists a certificate that causes its verifier to accept. Every verifier corresponds to a (oblivious) TM. The concept of a legal path through snapshots of this TM that takes exactly i steps and ends with acceptance can be expressed by a boolean formula whose time to produce depends only on the TM (and i) not the input. The number of steps required by the verifier is polynomially bounded by the input by the definition of NP. To continue with a Karp reduction is mathematically sound but unintuitive as we don't know the size of formula required for a given input only that it will be small enough.

We will lose a touch of rigour for the sake of clarity. Use a Cook reduction to generate and check the formula for ascending values of i until a satisfiable formula is found. If the input is a yes instance, we will accept, moreover this will happen within poly-many (in the length of the input) formulas. If the input is a no instance, we will not halt but the important thing is we do not accept.

3SAT ∈ NP-COMPLETE

3-SAT ∈ NP: A satisfying assignment **is a witness. It is trivial to see that this is polynomial length and can be verified in polynomial time.**

SAT $\leq_P$ 3-SAT:

Reduction: We will use the reduction from CS262. If there exists no clause C = ($L_1 \vee \ldots L_k$) such that k > 3, we are done. Assume we are not done. **Replace C with C′ ∧ C″ where C′ = $L_1 \vee \ldots \vee L_{k-2} \vee z$ and C″ = $L_{k-1} \vee L_k \vee \neg z$ where z is a new variable**. Apply this process repeatedly until all clauses are of length 3. This is guaranteed to occur as at each step a clause C of length k < 3 is replaced by clauses C′ with length

k − 1 and C″ has length 3.

P-time: Each step is trivially poly-time. Each step reduces the length of a clause, so we reach our stopping condition in poly-steps.

⇒: Assume SAT = Yes. Then, there exists a satisfying assignment φ and in φ at least one of $L_1, \ldots, L_k$ (for any arbitrary clause C) is true. Thus, at least one of C′ and C″ is true. As z is a new variable (and distinct for each decomposed clause) it can be set freely to satisfy the decomposed clause not yet satisfied (or arbitrarily if both already satisfied)

⇐: Assume 3-SAT = Yes. Then, both C′ and C″ must be true.

Case z is true: Then, ¬z is false. So, at least one of L_{k-1} and L_k is true. So, C is true.

Case z is false: At least one of $L_1, \ldots, L_{k-2}$ is true. So, C is true.

SUBSET-SUM ∈ NP-COMPLETE

- SUBSET-SUM = Given a number W and a set S of numbers $\{w_1, \ldots, w_n\}$, does there exists $S' \subseteq S$ such that $\sum S' = W$. We assume numbers are given encoded in binary

∈ NP: A candidate S' is a witness. It is trivial to see that this is poly-length and that the membership of its elements and the value of their sum can be verified in poly-time.

3SAT $\leq_P$ SUBSET-SUM: We will map the choices about which variables to set to true to the choices about which numbers to include in the subset. We need to find a way of encoding the clauses in the target W. We will use each place value place to encode a constraint (and we will argue that no overflows occur). Pre-process the formula to remove clauses which contain both its literal and its negation. Let the variables be ordered 1, …, m and the clauses be ordered 1, …, n.

For each literal l construct a number: $\#_{i=1}^{m} \mathbb{1}$[l is an occurrence of the ith variable] # $\#_{j=1}^{n} \mathbb{1}$[l occurs in the jth clause] where # denotes concatenation. By constructing $W = \#_{i=1}^{m} 1$ # … we require that exactly one literal for each variable is chosen as required but we want to allow 1, 2, or 3 literals in each clause to be chosen. Introduce 2 dummy numbers for each clause C, $\#_{i=1}^{m} 0$ # $\#_{j=1}^{n} \mathbb{1}$[C is the jth clause] and $\#_{i=1}^{m} 0$ # $\#_{j=1}^{n} 2*\mathbb{1}$[C is the jth clause]. Now, by setting $W = \#_{i=1}^{m} 1$ # $\#_{j=1}^{n} 4$, we are done. Each variable occurs as only 2 literals so these columns add up to at most 2 so cannot overflow. Each clause contains 3 variables and we create 2 dummy numbers for so these columns add up to at most (1+1+1)+(1+2)=6 so cannot overflow. It is obvious that this transformation is polynomial time by construction.

3SAT $\leq_P$ SUBSET-SUM: Correctness

$\Leftarrow$: Let S′ be a subset with the required sum. S′ induces a legal assignment to the variables of the CNF (make each literal whose number is included in S′ true), as exactly one literal for each variable is in S′. Moreover, this must be a satisfying assignment as each clause must be satisfied as the place for each clause sums to 4 (and can only reach 3 from the dummies) and so at least one literal in every clause must be in S′.
$\Rightarrow$: Let A be a satisfying assignment. A induces a subset S″ representing the literals which are assigned true by A. Then $\exists$ S′ $\subseteq$ S: sum of S′ = W. In particular, S′ = S″ with the following (dummy) elements from S added for each clause. As S″ corresponds to a satisfying assignment, each clause's place sums to one of 1, 2, 3. For 1, include the numbers that add 2 and 1 to that clause. For 2, include the number that adds 2 to that clause. For 3, include the number that adds 1 to that clause. The property of S″ of sum 1 in all the literal places is unaffected in S″ (as all the new numbers have 0s in all the literals places and we don't remove any numbers) and now the sum is 4 in all the clause places.

- Note that when we describe a further reduction of sorts to construct a solution to one given a solution to the other, this does not necessarily need to run in poly-time (although I think this one does)

Pseudo-polynomial-time

- **Defn: An algorithm is pseudo-polynomial-time** iff it returns in polynomial time when the numbers are encoded in unary **iff is it is polynomial time in terms of the value of the numbers**
  - Usually we consider whether an algorithm is polynomial time in terms of the length of the input ($\log_k$(value of the numbers) (where k is the base) if $k > 1$)
- **Defn: A problem is strongly NP-Complete iff it is NP-Complete even when the numbers are encoded in unary**
- SUBSET-SUM is pseudo-polynomial time, so is only weakly NP-Complete
- Problems like SAT that don't use numbers are obviously strongly NP-Complete but also some problems that do use numbers such as TRAVELLING-SALESPERSON are to

Exponentiation by repeated squaring

- The naive algorithm for computing x^y mod s is only pseudo-polynomial time:

```
a = 1
for i = 1 to y
    a *= x
    a %= s
return a
```

  - This does y multiplications, but the input is log(y)
- **Exponentiation by repeated squaring is true polynomial time:**

```
POWER(x, y, s):
    if y = 0 then return 1
    if y is even then return (POWER(x, y/2, s))^2 % s // (x^(y/2))^2 = x^y
    a = POWER(x, (y - 1)/2, s)
    return (x*a^2) % s // a^2 = x^(y - 1), hence x*a^2 = x^y
```

  - Each individual multiplication (note that each squaring can be rewritten as a single multiplication) or modulo is p-time (in the length of y (log(y))
  - Each call does a constant number of operations
  - At least every other call is the even case, hence the recursion depth is O(log(y)), so linearly many calls are made

(DIRECTED-)HAMILTONIAN-CYCLE ∈ NP-COMPLETE

- (DIRECTED-)**HAMILTONIAN-CYCLE** = Given a (directed) graph G = (V, E) does there exist a cycle that **visits every vertex exactly once**

(DIRECTED-)HAMILTONIAN-CYCLE ∈ NP: A set of edges is a witness and is obviously poly-length. Easy to see can be verified to meet the constraints in poly-time.

3SAT $\leq_P$ DIRECTED-HAMILTONIAN-CYCLE: **Create a pair of vertices connected by bidirectional edges for the literals of each variable, with the positive on the left and the negative on the right. Create an edge from each literal for each variable x_i to <u>both of</u> the literals for x_{i+1}.** Add a source node with edges to the literals for the first variable and a target node with edges from the literals for the last variable. Add an edge from the target node to the source node to form a cycle. Now, every hamiltonian cycle corresponds to an assignment (and vice versa) where **the literal visited first for each vertex is assigned true** (and so second false). It remains to encode the clauses.

For each clause: Create a vertex: For each variable that occurs in the clause: Subdivide the edge between the variable's literals into 4 new (not shared between clauses) **intermediate vertices** (with bidirectional edges between them). **Leave the first and last new vertex alone.** For occurence as a positive/negative literal in the clause, add an edge from the second/third (from the left) new vertex to the clause and an edge to the third/second new vertex from the clause.

- Proposition: DIRECTED-HAMILTONIAN-CYCLE $\leq_P$ HAMILTONIAN-CYCLE and so HAMILTONIAN-CYCLE ∈ NP-COMPLETE

  Proof: Convert each vertex into two vertices, one with the incoming edges, and one with the outgoing edges, connected by a single edge. To require that the edge in between is actually taken (and so enforce the directionality of the other edges in the undirected setting) **subdivide it with another node.**

3SAT $\leq_P$ DIRECTED-HAMILTONIAN-CYCLE: Correctness

Poly-time: Easy to see

SAT ⇒ DHC: Start at the source. For each variable in turn, if it is assigned true/false, take the edge to the left-/right-most node for that variable (positive/negative literal) then move right/left until the other literal is encountered then take the edge to apply the same procedure to the next variable. Take the edges to and from clauses when they are encountered iff the clause has not been visited yet. Once all the nodes for the final variable have been visited, take the edge to t then the edge to s to complete the cycle. By construction this visits every variable node exactly once (and the construction is well-defined as the assignment is consistent). As the assignment is satisfying, every clause node will have at least one opportunity to visit it and as we only take the first opportunity will be visited exactly once. We follow the directions of all the edges. Thus, this is a directed hamiltonian cycle.

DHC ⇒ SAT: It is possible to show that the only way of leaving a clause node within a directed hamiltonian cycle is to return to the neighbour node in the same row. It is fairly easy to see that within a directed hamiltonian cycle each row can only be traversed either right-to-left or left-to-right, hence our scheme for building an assignment from the assigns exactly one truth value to each variable. By the definition of a hamilton cycle, every clause node is visited, and so by the construction of the graph every clause is satisfied by our assignment.

3-COLORABILITY ∈ NP-COMPLETE

- 3-COLORABILITY = Given an undirected graph, is it possible to assign colours from a set of 3 colours to vertices such that no adjacent vertices are assigned the same colour.

∈ NP: A coloring function is a witness. Obviously poly-length and poly-time.

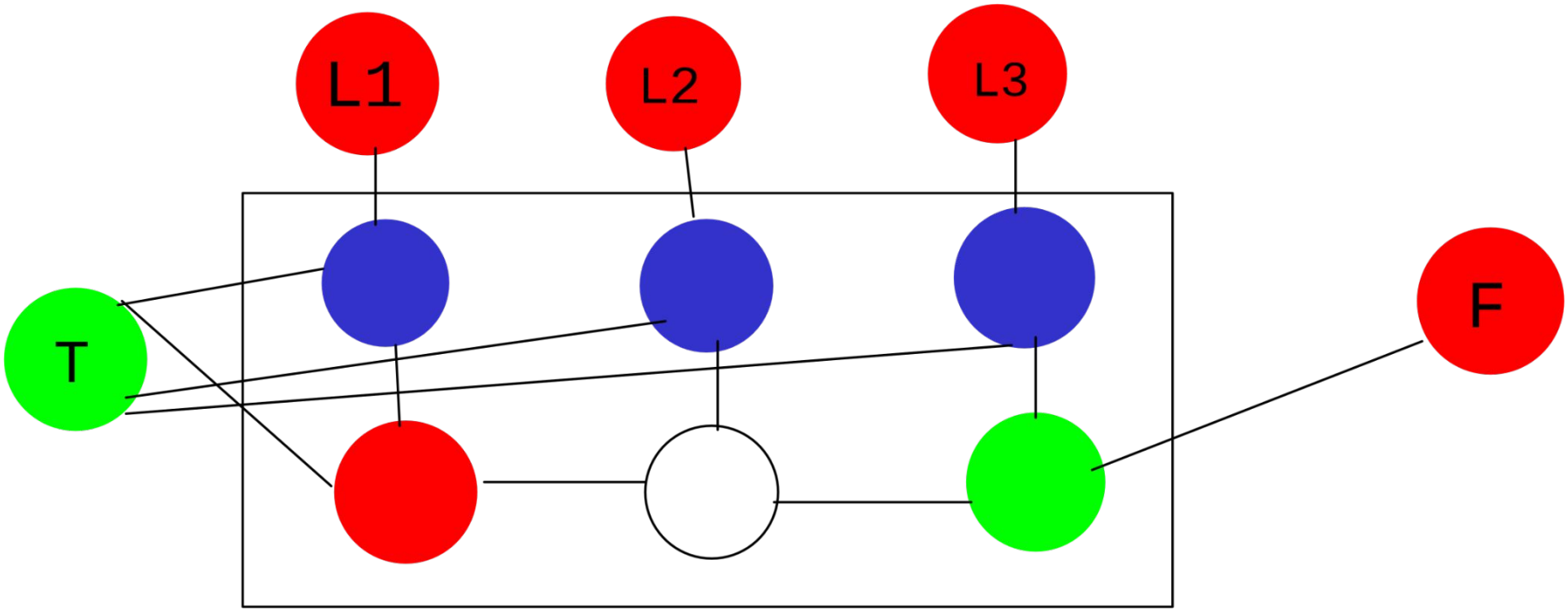
3SAT $\leq_P$ 3-COLORABILITY: **For each literal, create a node. Create 3 further nodes which we will call T, F, B and connect them in a triangle**, deduce that then the colors assigned to these (e.g. green, red, blue) are the 3 distinct colors in use. **Connect each literal to B to force there to be only 2 colours available to literals**, literals coloured with the colour of T will be assigned true and those coloured with the colour of F will be assigned false. Connect each literal to its negation so that they are assigned opposite truth values to each other (the assignment is consistent). **For each clause, create 6 vertices. For each literal in the clause, connect a clause vertex to it. Connect each clause vertex that is connected to a literal to T and to one (unique to each) of the other 3 vertices. Connect the other 3 vertices to form a simple path starting from T and ending at F.** Now every clause must be satisfied by any assignment induced by a 3-coloring.

- Proposition: For all k>3, k-COLORABILITY ∈ NP-COMPLETE
  Proof: (k-1)-COLORABILITY $\leq_P$ k-COLORABILITY by adding a new vertex that is connected to every existing vertex
- 2-COLORABILITY ∈ P (equivalent to checking if bipartite)

3-COLORABILITY ∈ NP-COMPLETE

Poly-time: Obvious
SAT ⇒ 3C: Colour the literals green and red according to whether they are assigned true or false in the satisfying assignment. For each clause: **Pick an arbitrary clause node that is connected to a green literal** (this must always exist as the assignment is satisfying) **and colour it red. Colour the remaining nodes connected to literals blue. Colour the dummy node connected to the red node blue.** If the dummy node connected to T is not yet colored, colour it red. If the dummy node connected to F is not yet coloured, colour it green. If the dummy node between those connected to T and F is not yet coloured, colour it with the remaining colour. It can be shown by case analysis that this leads to a legal coloring.
SAT ⇐ 3C: Set all literals coloured green to true and red to false. Assume for the sake of contradiction there is a clause which has not been satisfied (even though its gadget is part of our successful 3-coloring). As it has not been satisfied, all the literals it is connected to are coloured red. Thus, the dummy nodes connected to literals are all coloured blue. Thus, none of the other dummy nodes can be coloured blue. Thus, the ones connected to T and F must be coloured red and green respectively. Thus, the remaining dummy node cannot have been legally 3-colored as its neighbours have all 3 colours?!

Visualization of non-3-colorability of a falsifying assignment



Problems well known to be NP-Complete

- INDEPENDENT-SET = Is there a subset S of (at least) k vertices such that at most one endpoint of each edge is in S? — reduce from 3-SAT
- VERTEX-COVER = Is there a subset S of (at most) k vertices such that every edge has at least one endpoint in S? — the complement of a vertex cover is an independent set (and vice versa)
- SET-COVER = Let $S = \{S_1, \ldots, S_m\} \subseteq 2^U$. Does there exist $S^* \subseteq S$ such that $|S^*| \leq k$ and $\cup_{Si \in S} S_i = U$? — reduce from VERTEX-COVER
- Travelling salesperson problem (TSP) = Given a threshold D and a list of n cities and a function d that gives the distance between all pairs of cities, does there exist an ordering x1, x2, …, xn s.t. d(x1, x2) + … + d(xn-1, xn) + d(xn, x1) ≤ D? — reduce from HAMILTONIAN-CYCLE

co-NP

Introduction to complementation

- Recall that for a language L, the complement of L, ¬L = $\{w \in \Sigma^* : w \notin X\} = \Sigma^* \backslash X$. We will extend this to **the complement of a complexity class Y co-Y = {¬X ∈ Y} ≠ ¬Y**
- Deduce that P = co-P
- Let TAUTOLOGY = Is a given DNF formula a tautology? (if it was a CNF this would be trivial), SAT = is a given CNF formula satisfiable? (if it was a DNF this would be trivial), IS-CONTRADICTION = is a given CNF formula a contradiction? (if it was a DNF this would be trivial). Then, IS-CONTRADICTION = ¬SAT. Moreover, TAUTOLOGY $\leq_P$ IS-CONTRADICTION (can apply DeMorgan in poly-time to obtain a CNF for the negation of the DNF)
    - It is easy to disprove being a contradiction/tautology (give a satisfying/falsifying assignment) but seems hard to prove
    - It is easy to prove it being satisfiable (give a satisfying assignment) but seems hard to disprove

co-NP

- **$X \in coNP$ iff $\neg X \in NP$**
  - By double negation, $\neg X \in coNP$ iff $X \in NP$
- For problems in NP we have efficient checkable certificates that are proofs that a word is in the language, for problems in coNP we have efficient checkable disqualifiers that are proofs that a word is not in the language
- Thm: $P \subseteq coNP$
  Proof: $X \in P \Rightarrow \neg X \in P \Rightarrow \neg X \in NP \Rightarrow X \in coNP$

NP =? coNP

- It seems more likely that NP ≠ coNP than NP = coNP but we don't actually know
- **Proposition: It is not possible that coNP ⊃ NP**. In other words, either NP = coNP or there exists at least one problem that is in NP but not in coNP
  Proof: If P = NP, then as P is closed under complementation so is NP and so NP = coNP.
  Suppose NP ≠ coNP. Assume for the sake of contradiction that NP ⊂ coNP. Then, ∃X: X ∈ coNP\NP. As X ∈ coNP, ¬X ∈ NP. As NP ⊂ coNP, ¬X ∈ coNP. As ¬X ∈ coNP, ¬¬X ∈ NP but ¬¬X = X and so X ∈ NP but X came from coNP\NP?!
  - This does not mean that coNP ⊆ NP, it is very possible that there also exists problems that are in coNP but not in NP

Why have we been using Karp reductions?

Lemma: **NP is closed under polytime Karp reductions**

Proof: Let X be an arbitrary problem in NP with certifier C_x and Y be an arbitrary problem that polytime Karp reduces to X. Let σ be the function that maps instances of Y to instances of X. Pick arbitrary $y \in Y$ and let $x = \sigma(y)$. Then, there exists a certificate c s.t. $C_x(x, c)$ accepts ⇔ x is a yes instance of X ⇔ y is a yes instance of Y. Thus, the following TM is a certifier for Y: $C_Y(y, c)$ = Simulate $\sigma(y)$ and store the output x, then simulate $C_X(x, c)$ and return the result. It is fairly easy to see that this follows the conditions for NP. Hence, $Y \in$ NP as required.

Why have we been using Karp reductions?

Theorem: **NP is closed under polytime Cook reductions iff NP = coNP**
Proof:
NP = coNP ⇒ NP is closed under Cook reductions: Let X be an arbitrary problem in NP and Z be an arbitrary problem that polytime Cook reduces to X. If the reduction makes no oracle calls, then Z ∈ P and so certainly Z ∈ NP. Suppose the reduction does make a non-zero number of oracle calls. As polynomial time is spent on the reduction (assuming oracle calls take constant time), it is possible to construct a polynomial length string w that describes the input to and decision made by each oracle call. As X in NP and by the hypothesis NP = coNP, there exists certificates and witnesses respectively for yes and no instances of X. It is then possible to create a certifier for Z that simulates the reduction corresponding to a given overall certificate by simulating the certifier/disqualifier (as appropriate) for X on the appropriate certificate/witness to verify the decision made at each oracle call (and so runs in polynomial time).
NP = coNP ⇐ NP is closed under Cook reductions: Let X be an arbitrary problem in co-NP, then ¬X ∈ NP, Calling the oracle for ¬X then returning the opposite decision is a polytime Cook reduction from ¬¬X=X to ¬X, so by the hypothesis, X ∈ NP. As X was arbitrary from co-NP, and we have shown that X ∈ NP, coNP ⊆ NP. Finally, by a previous result, coNP cannot be a proper subset of NP, so coNP = NP as required.

co-NP-Complete

- **$L \in$ coNP-Complete iff $\forall L' \in$ coNP. $L' \leq_P L$**
- Lemma: If $X \leq_P Y$, then $\neg X \leq_P \neg Y$
  Proof: Suppose there exists a polytime computable function σ such that $\forall x \in X$; $x \in X$ iff $\sigma(x) \in Y$. Then, $x \notin X$ iff $\sigma(x) \notin Y$. Thus, $x \in \neg X$ iff $\sigma(x) \in \neg Y$ as required
- **Proposition: coNP-complete = {¬X : X ∈ NP-complete} (and NP-complete = {¬X : X ∈ coNP-complete})**
  Proof: Easily follows from the lemma
- Lemma: If $X \in$ coNP and $Y \leq_P X$, then $Y \in$ coNP
  Proof: Same as the proof for NP being closed under polytime Karp reductions, just replace the word certifier with disqualifier and certificate with witness.
- **Thm: If NP ≠ coNP, then no problem is in both NP-COMPLETE and coNP and symmetrically no problem is in both coNP-COMPLETE and NP**
  Proof: We will show the contrapositive.
  Suppose NP-complete ∩ coNP contains some problem X. Then, for every Y in NP, $Y \leq_P X$ as X is NP-complete. As X in coNP and Y p-time Karp reduces to X, Y is also in coNP. Hence, NP ⊆ coNP. By a previous theorem, NP cannot be a proper subset of coNP, so NP = coNP.

FACTOR $\in$ NP $\cap$ coNP

- FACTOR(x, y) = Does x have a non-trivial (not 1) factor which is smaller than y? = {<x, y> | $\exists$ z: 1 < z < y and z is a factor of y}
- Theorem (1975): FACTOR $\in$ NP $\cap$ coNP
  Proof:
  $\in$ NP: Certificate is a number a. Certifier verfies a > 1 and a < y and a divides x.
  $\in$ coNP: Disqualifier is a prime factorization, verifier verifies all the numbers are prime (even if we didn't know PRIMES $\in$ P, we know PRIMES $\in$ NP and we have enough space to include the certificates in this certificate) and that their product is x
- COMPOSITES = {n: n $\notin$ PRIMES}
- Proposition: COMPOSITES $\leq_P$ FACTOR
  Proof: σ(x) = <x, x> is such a reduction. This works because x has a non-trivial factor less than x is the definition of x is composite.
- *It is believed (but not known) that FACTOR $\notin$ P (there exists no polytime reduction from FACTOR to PRIMES)*
  - *PRIMES $\in$ NP $\cap$ coNP, and was eventually proven to be in P*
  - *FACTOR $\in$ NP $\cap$ coNP. If FACTOR is found to be in P, then RSA cryptography is in trouble (before quantum computers become feasible) as the search problem FACTORIZE polytime Cook reduces to binary search with FACTOR*

P ≠? NP ∩ co-NP

- P =? NP: Are problems that are easy to check yes answers to easy to decide?
- coNP =? NP: Are problems that are easy to check yes answers to easy to check no answers to?
- P =? NP ∩ coNP: Are problems that are easy to check both yes and no answers to **(we call such problems (i.e. exactly the problems in NP ∩ coNP) well-characterised)** easy to decide?
    - *There are quite a few problems (such as linear programming and PRIMES) that took a while to be shown to be in P after being shown to be well-characterised, but currently there really is a feeling that FACTOR (which is known to be well-characterised) is NP-Intermediate (and so is not in P)*
        - *PRIMES ∈ NP ∩ coNP is 1975 but PRIMES ∈ P is only 2002!*

PSPACE

Space complexity

- P(TIME) = the set of decision problems solvable in polynomial time
- PSPACE = the set of decision problems solvable in polynomial space
- **X ∈ PSPACE iff there exists k s.t. there exists a Turing machine that decides X and in doing so visits $O(n^k)$ <u>distinct</u> cells on the tape**
- Prop: **P ⊆ PSPACE**
  Proof: A TM visits at most one new cell in each step. Thus, any TM that runs for polynomial many steps visits at most polynomially many new cells.
  - *Once again, this is believed to be a proper subset (as* **<u>you can reuse space</u> but you cannot reuse time***), it feels like the time restriction should be strictly stronger), but nobody knows*
- Theorem: SAT ∈ PSPACE
  Proof: Storing and checking one assignment requires polynomial space (and time). There are 2^n assignments to check, but each one can reuse the same space
  Corollary: **NP ⊆ PSPACE**
- **L ∈ PSPACE-complete iff ∀L' ∈ PSPACE. L' $\leq_p$ L — note this requires a polynomial <u>time</u>** (a stronger condition than polynomial space) **reduction**

QSAT

- **QSAT (Quantified Satisfiability) = Given $\phi(x_1, \ldots, x_n)$ is $\exists x_1 \forall x_2 \ldots$ (continuing to alternate) $\phi(x_1, \ldots, x_n)$ true**
  - SAT was our prototypical NP(-Complete) problem, QSAT will be our prototypical PSPACE(-Complete) problem
- Theorem: **QSAT $\in$ PSPACE**
  Proof: **We can see this as a game between two players and our algorithm will explore the tree of the state space of the game.** Leaves are labelled with the output of the boolean function given the assignment of the edges on the path from the root to that leaf. Each internal node is assigned the $\vee$ ($\exists$), or the $\wedge$ ($\forall$) of its children. Thus, every node ends up labelled with whether the $\exists$ player can win from that node and the root is thus the answer to our problem. However, storing the values of all the leaves simultaneously would use exponential space. Fortunately, **it is possible to do the computations so that only the children of the node currently being expanded are stored at each point, although this is wasteful in terms of repeated computations in time it allows us to reuse the space a lot**. Now, we store a constant amount of information at each level. Thus, as the depth of the tree is linear in the size of the input, the overall space is now linear and so certainly is polynomial.

```
nodeValue(n)
    if n is a leaf then return Φ(α) where α is the induced assignment
    z_0 = nodeValue(n.left)
    z_1 = nodeValue(n.right)
    if n.quantifier == ∃ then return z_0 ∨ z_1
    else if n.quantifier == ∀ then return z_0 ∧ z_1
```

- Theorem (Stockmeyer-Meyer): **QSAT $\in$ PSPACE-complete**
  Proof: Out of scope

COMPETITIVE-FACILITY-LOCATION

- COMPETITIVE-FACILITY-LOCATION = Given an undirected graph with positive vertex weights and a number B. Two players are playing a game on this graph where they take turns to pick a vertex out of those which have not had any of their neighbours chosen (and has not been chosen itself) until there are no vertices it is possible to select. Is it possible for the first player to stop the second player from obtaining a collection of vertices that sum to at least B?

COMPETITIVE-FACILITY-LOCATION ∈ PSPACE-Complete

∈ PSPACE: Note that for the first player to win from a node they need to be able to win from some child if they control the node but from every child if the second player controls the node. Thus, by taking checking whether the first player has won as the boolean function at the leaves, this is simply a generalisation of the algorithm for QSAT to n (choosing a vertex of the input) children instead of 2 (choosing a truth value for a variable) children. Thus, this is now quadratic space (as is now linear space in the input on each level) instead of linear but that is still polynomial.
QSAT $\leq_P$ COMPETITIVE-FACILITY-LOCATION: We need to convert a boolean formula into a vertex-weighted graph. Without loss of generality, assume that the formula contains an odd number of variables. For each literal we create a vertex, connect literals of opposite parity from the same variable with an edge (so that at most one is chosen), and say that the literal that is set to true is the one that is chosen in the game. **We would like to set the weights so that the players will have no reason not to pick a literal from X_1 first, then a literal from X_2 etc. To do this, we let c = k+2 where k is the number of clauses, and give the literals for the ith variable weight c^{n-i}, and set B = $c^{n-1} + c^{n-3} + \ldots + c^2 + 1$.**
The constraints introduced by the clauses must be modelled. For each clause, add a vertex with weight 1, and connect each literal that occurs in the clause to the clause node, this means that a clause can be chosen iff none of its literals have been chosen (it is false). Because the clause nodes are worth the least, a clause will only be chosen as the last move in the game (which is by player 2 as we forced that the formula has an odd number of variables), and the construction of our B means that player 2 can win iff they can choose a clause node. Hence, player 1 loses in this game iff it possible for player 2 in the original QSAT game to make choices that result in ϕ being falsified.

EXPTIME

- **$X \in$ EXPTIME iff there exists k s.t. there exists a Turing machine that decides X in $O(2^{n^k})$**
- Thm: PSPACE $\subseteq$ EXPTIME
  Proof:
  Pick arbitrary $Y \in$ PSPACE. As QSAT $\in$ PSPACE-complete, $Y \leq_P$ QSAT. It is easy to see that QSAT $\in$ EXPTIME. Thus, $Y \in$ EXPTIME also.
  Direct proof:
  For each problem in PSPACE, there is a corresponding Turing machine that only uses a polynomial amount of the tape p(n) in the input size n and halts on every input (and does so with the correct answer to the decision problem). Let Γ be the tape alphabet, p(n) be the number of cells used, and Q be the states. Then, the number of distinct configurations is $|\Gamma|^{p(n)}p(n)|Q|$ which is exponential in n. Recall that if a TM enters the same configuration twice during its computation it will not halt. Thus, a TM must terminate in at most the number of distinct configurations, which completes the proof.
- We have shown that: **P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME**
  - It is conjectured that these are all $\subset$, but the only thing we know for sure is that P $\neq$ EXPTIME (and so at least one must be $\subset$ but we don't know which one or if more than one is)

Communication Complexity

Communication complexity

- The form of a communication complexity problem: Alice knows x, Bob knows y, together they will compute f(x, y) using a pre-arranged protocol, how many bits do they exchange?
  - For a given algorithmic problem, can we derive bounds on the number of bits exchanged under a protocol which minimises the number of bits exchanged?
  - We require that at the end of the protocol Alice and Bob both know f(x, y) but it is not necessarily required that one sends f(x, y) to the other
- **Cost of a communication protocol = number of bits exchanged in worst case**
- **Communication complexity of f = D(f) = cost of the best communication protocol for f**
- **Theorem: For any f, D(f) ≤ log |X| + log |Z|**
  Proof: The following protocol works for any f: $X \times Y \mapsto Z$. Alice can communicate which $x \in X$ she has using an encoding of length $\log_2 |X|$. Bob can then compute z = f(x, y) and communicate which element of Z this is using an encoding of length $\log_2 |Z|$. Hence the cost of f is $\log_2 |X| + \log_2 |Z|$
  - For some problems, there exists more efficient specialized protocols as the exact values of x and y are not needed to compute f(x, y) — e.g. $x_1 \vee \ldots \vee x_n \vee y_1 \vee \ldots \vee y_n \equiv (x_1 \vee \ldots \vee x_n) \vee (y_1 \vee \ldots \vee y_n)$ meaning only 2 bits are required instead of $\log_2(2^n) = n$

MEDIAN

- MEDIAN = Alice has a $x \subseteq \{2, \ldots, 2n\}$ and Bob has a $y \subseteq \{1, \ldots, 2n - 1\}$. $f = \text{median}(x \cup y) =$ the $\text{ceil}(|x \cup y|/2)^{th}$ smallest element of $x \cup y$.
- Proposition: $D(\text{MEDIAN}) \in O(n)$
  Proof: The generic protocol can use an n bit bitmask for the 2n/2=n element parent set to communicate the x, and $\log_2(2n)$ bits to send the value of the median (which can be any number between 1 and 2n). Hence, $D(f) \leq n + \log_2(2n)$
- Theorem: $D(\text{MEDIAN}) \in O(\log^2 n)$
  Proof: We will essentially binary search for the median. Let [i, j] be the interval at the start of an arbitrary round. Individually compute $k = \text{floor}((i+j)/2)$. Have Alice compute $L_x = |[i, k] \cap x|$, $R_x = |[k+1, j] \cap x|$ and Bob do the same for L_y and R_y. Now the communication: exchange L_x, R_x, L_y, R_y. Individually compute whether the median is in [i, k] or [k+1, j] based on whether $L = L_x + L_y + L'$ (where L' = L from the last round if L was chosen in the last round, 0 otherwise) (recall these are cardinalities) or R (symmetrically defined) respectively is greater and thus (roughly) half the size of the interval.
  Each round sends 4 numbers in the range 0 to n, so uses 4 log n bits. As the size of the interval asymptotically halves each time, there are O(log n) rounds. Hence, the cost is $O(\log^2 n)$ which is better than the O(n) before.

Protocol trees

- Without loss of generality assume each message is a single bit and Alice and Bob alternate sending messages
- **We can represent a communication protocol as a protocol tree where the edges show the messages sent and the leaves show the derived value of f**
  - **Then, the height of the tree is the cost of the protocol in the worst case**

Combinatorial rectangles

- We can represent a f as a matrix M_f where the rows are labeled with the possible xs and the columns are labelled with the possible ys and the matrix entries are the value of f for those inputs
- **A combinatorial rectangle** = a subset of entries of M_f of the form A cartesian product B where A is a subset of the rows and B is a subset of the columns
    - **For some reordering of the rows and columns, this forms a literal rectangle in the matrix**
- **A combinatorial rectangle is monochromatic iff all the entries in M_f which are inside it are the same**
    - **If the combinatorial rectangle corresponding to the possible values of the inputs based on the communications so far is monochromatic, then no further communication is needed** for both participants to be able to deduce the value of f(x, y) even though they might not be able to know each others exact values yet

Combinatorial rectangles and protocols

- **Each leaf of a protocol tree must correspond to a monochromatic combinatorial rectangle (corresponding to all the pairs of inputs that lead to it), together all these cover the entire matrix**
- **C(f) = the minimum number of monochromatic combinatorial rectangles needed to completely partition** (or equivalently cover, but the minimum will always correspond to a partitioning (no overlaps)) $\mathbf{M_f}$
- **C(f) ≤ number of leaves in the protocol tree for f with the fewest leaves**
  - This is not equality because every protocol corresponds to a partition but some partitions do not correspond to protocols
- **Theorem: D(f) ≥ ceil($\log_2$(C(f)))**
  Proof: Assume we have the best possible protocol tree. The worst possible case for the covering is making a seperate (trivally monochromatic) combinatorial rectangle for every leaf and not being able to merge any of them into a larger combinatorial rectangle without breaking monochromaticity. **The number of leaves is at most $2^{D(f)}$ as D(f) = height of the best protocol tree.** Thus, C(f) ≤ $2^{D(f)}$ and so we have the required result.

Fooling sets

- **A fooling set = a set of inputs $\{(x_1, y_1), \ldots, (x_k, y_k)\}$ such that $f(x_1, y_1) = \ldots = f(x_k, y_k)$ and $\forall i \neq j.\ f(x_i, y_i) \neq f(x_i, y_j) \vee f(x_i, y_i) \neq f(x_j, y_i)$**. That is that along the leading diagonal the combinatorial rectangle corresponding to $\{x_1, \ldots, x_k\} \times \{y_1, \ldots, y_k\}$ is monochromatic but for each possible size of step inside the rectangle off of each point on the diagonal it is not monochromatic as we vary one of x and y
- **Lemma: No two inputs in a fooling set have the same transcript (sequence of communications)**
  Proof: Assume for the sake of contradiction that there exists a fooling set F such that $\exists (x_i, y_i) \in F$: $\exists (x_j, y_j) \in F$: ($i \neq j$ but they have the same transcript). Deduce that a transcript corresponds to a particular path through the protocol tree. Thus, as they have the same transcript, our instances must be at the same leaf and so in the same monochromatic combinatorial rectangle. But any combinatorial rectangle that contains (x_i, y_i) and (x_j, y_j) also contains (x_i, y_j) and (x_j, y_i). Thus, as F is a fooling set this combinatorial rectangle cannot be monochromatic?!
  Corollary: Each element of the fooling set must be in a different monochromatic combinatorial rectangle to all the others. Thus, if f has a fooling set F, then $C(f) \geq |F|$.
  Corollary of corollary: **If f has a fooling set F, then $D(f) \geq \text{ceil}(\log_2(|F|))$**

EQUALITY

- EQUALITY: f(x, y) = 1 iff x and y agree in every bit; 0 otherwise
- Proposition: The naive communication protocol with cost n + 1 is the best cost possible
  Intuition: A single bitflip anywhere in one the inputs can change the answer
  Proof: Any change to exactly one of x and y in a 1 instance makes it a 0 instance, and so these $\{(x, x): x \in X\}$ are a fooling set but this set is equinumerous with the set $\{x: x \in X\}$ and there are 2^n bitstrings of length n. Hence, $D(f) \geq \text{ceil}(\log_2(2^n))$ but $\text{ceil}(\log_2(2^n)) = n$ and by the naive protocol $D(f) \leq n + 1$. Thus D(f) = n or n+1.
  - It is possible to see that we really do need n+1 combinatorial rectangles. We have already established that we need a different leaf of the protocol tree for each of the 2^n 1s on the diagonal. We need at least one more for the 0s. Thus, $D(f) \geq \text{ceil}(\log_2(2^n + 1)) > n$. Thus, $D(f) \geq n + 1$ as required.

The rank technique

- **Rank of a matrix A = the dimension of the vector space spanned by its columns (or equivalently its rows)**
- **Subadditivity property of rank: rank(A + B) ≤ rank(A) + rank(B)**
- An nxn matrix is invertible iff it has full rank (rank n)
- **Theorem: Let f be a <u>decision</u> problem defined by a matrix M_f. Then, C(f) ≥ 2 * rank(M_f) − 1**
  Proof:
  Pick an arbitrary combinatorial rectangle R. Then we can define a matrix M_R of the same shape as M_f in which the entries that are in R are 1 and all others are 0. Deduce that rank(M_R) = 1 as the columns that are all zeroes don't contribute to the rank and the columns that contain 1s are all the same (up to elementary column operations (which do not affect rank)).
  Let C be a set of optimal monochromatic combinatorial rectangles. Then, C(f) = |C|. Partition C into O = those that cover the ones and Z = those that cover the zeros. Thus, C(f) = |O| + |Z|.
  Let J be the matrix of all ones which has the same shape of M. Then, $M_f = \sum_{R \in O} M_R = J - \sum_{R \in Z} M_R$.
  Thus, by subadditivity, rank(M_f) ≤ $\sum_{R \in O}$ rank(M_R) and rank(M_f) ≤ rank(J) − $\sum_{R \in O}$ rank(M_R).
  Note that, $\sum_{R \in O}$ rank(M_R) = $\sum_{R \in O} 1$ = |O| and rank(J) − $\sum_{R \in O}$ rank(M_R) = $1 - \sum_{R \in Z} 1 = 1 + |Z|$.
  Thus, rank(M_f) + rank(M_f) ≤ |O| + 1 + |Z| so C(f) = |O| + |Z| ≥ 2*rank(M_f) − 1 as required
- For f = EQUALITY, M_f = the 2^nx2^n identity matrix. M_f has rank 2^n (as it is invertible) and thus D(f) ≥ ceil($\log_2(2*2^n - 1)$) = n + 1. This was much faster than our previous arguments

Randomised Algorithms

RP

- We can generalise Turing machines to probabilistic Turing machines for which there is a character that when written causes a 0 to be written with probability ½ and 1 to be written with probability ½
  - A probabilistic TM halts iff the expectation of running time is finite
- Alternatively (and equivalently), **randomization can be achieved with a deterministic Turing machine by supplying random bits in the input**
- **L ∈ NP iff** there exists a deterministic Turing machine M that checks a polylength certificate in polytime and has the property that **(a randomly chosen certificate has non-zero probability of being accepted iff x ∈ L)**
- There is a (at least intuitively) stricter notion, **L ∈ RP iff** there exists a deterministic TM M which checks a polylength (p(|x|)) certificate in polytime such that
  (x ∈ L ⇒ $Pr_{t \in \{0,1\}^{p(|x|)}}$[M accepts <x, t>] ≥ ½) and
  (x ∉ L ⇒ $Pr_{t \in \{0,1\}^{p(|x|)}}$[M accepts <x, t>] = 0)
  - Replacing ½ with an f(|x|) that goes to 0 no faster than $1/|x|^c$ for any c gives exactly the same class
    - Note any non-zero constant certainly obeys this, so ½ is a completely arbitrary convention

Exploring RP

- Theorem: **P ⊆ RP ⊆ NP**
  Proof: Easy to see
  - Whether P = RP is an open (and genuinely contentious) question
- co-RP = {X : ¬X ∈ RP}
- Theorem: P ⊆ coRP ⊆ coNP
  Proof: Mutatis mutandis previous theorem
  - Whether RP = coRP is also an open question
- **Probability amplification: Let M be a probabilistic TM for a language L ∈ RP. Running M k times and accepting iff <u>any</u> of the runs accepted gives error probability $1/2^k$, thus for a linear increase in running time we get an exponential decrease in the error probability**
  - Eventually the probability of the algorithm giving the wrong answer is smaller than that of other causes of system failure

BPP

- **RP and coRP allow one-sided errors (only false negatives and only false positives respectively), BPP allows two-sided errors**
- **$L \in BPP$ iff** there exists a deterministic TM M which checks a polylength ($p(|x|)$) certificate in polytime such that **$(x \in L \Rightarrow P(M \text{ accepts } \langle x, t \rangle) \geq \frac{2}{3})$ and $(x \notin L \Rightarrow P(M \text{ rejects } \langle x, t \rangle) \geq \frac{2}{3})$**
  - This gives the same class for any constant strictly greater than ½ (and strictly less than 1) — if we allowed exactly ½ the machine can just accept with probability ½ and reject with probability ½ without looking at x
- By probability amplification of (co)RP: **$P \subseteq RP \subseteq BPP$ and $P \subseteq coRP \subseteq BPP$**
  - *Although $P \subseteq (co)RP \subseteq NP$, it is not known whether $BPP \subseteq NP$ or $NP \subseteq BPP$ (or $NP = BPP$) or neither*
- For two-sided errors, probability amplification still applies, but needs to be tweaked to take a majority vote

ZPP

- **As well as Monte Carlo randomized algorithms which have deterministic runtime and probabilistic error, there are Las Vegas randomized algorithms which make no errors and have probabilistic runtime**
    - An equivalent definition of a Las Vegas algorithm is one which can detect its own errors such that the number of re-runs until an error-free run gives rise to in expectation a finite overall running time
- **BPP was the class of problems for which there exists a Monte Carlo algorithm with polynomial run time**
- **ZPP is the class of problems for which there exists a Las Vegas algorithm with polynomial run time in expectation**
- **Augment a Turing machine with a dunno terminal state as well as accept and reject. Then, $L \in$ ZPP iff** there exists a deterministic TM which checks a polylength (p(|x|)) certificate in polytime such that
  $(x \in L \Rightarrow P(M \text{ accepts or dunnos } \langle x, t\rangle) = 1)$ and $(x \notin L \Rightarrow P(M \text{ rejects or dunnos } \langle x, t\rangle) = 1)$ and
  $\forall x.\ P(M \text{ dunnos } \langle x, t\rangle] \leq ½)$
    - ½ can be any <u>constant</u> <u>strictly</u> between 0 and 1

ZPP = RP ∩ coRP

- Lemma: $P \subseteq ZPP \subseteq RP \cap coRP$
  Proof sketch: $P \subseteq ZPP$: Trivial
  $ZPP \subseteq RP \cap coRP$: Replace the dunnos with the error allowed by each class in turn (and see that dunnos are allowed infrequently enough to be an acceptable error rate for (co)RP) to see it is a subset of each of them. Thus, be able to conclude it is a subset of the intersection
- Theorem: $RP \cap coRP \subseteq ZPP$
  Proof: Pick an arbitrary $L \in RP \cap coRP$. Let M_2 be a witness for $L \in RP$ and M_3 be a witness for $L \in coRP$. As their errors are one-sided and on the different sides, running the same $\langle x, t \rangle$ on M_2 and M_3 gives the same answer iff neither has made an error. Thus, simulating both machines and returning the answer they both give if they agree and dunno otherwise ought to be sufficient. In particular, regardless of whether $x \in L$ or $x \notin L$, the probability of an error (and so a dunno) is at most ½ as required
  Corollary: **$RP \cap coRP = ZPP$**
  - *Recall that it is not known whether $NP \cap coNP \subseteq P$, whereas we have shown here that $RP \cap coRP$ corresponds to a specific model of computation*

Exploring ZPP

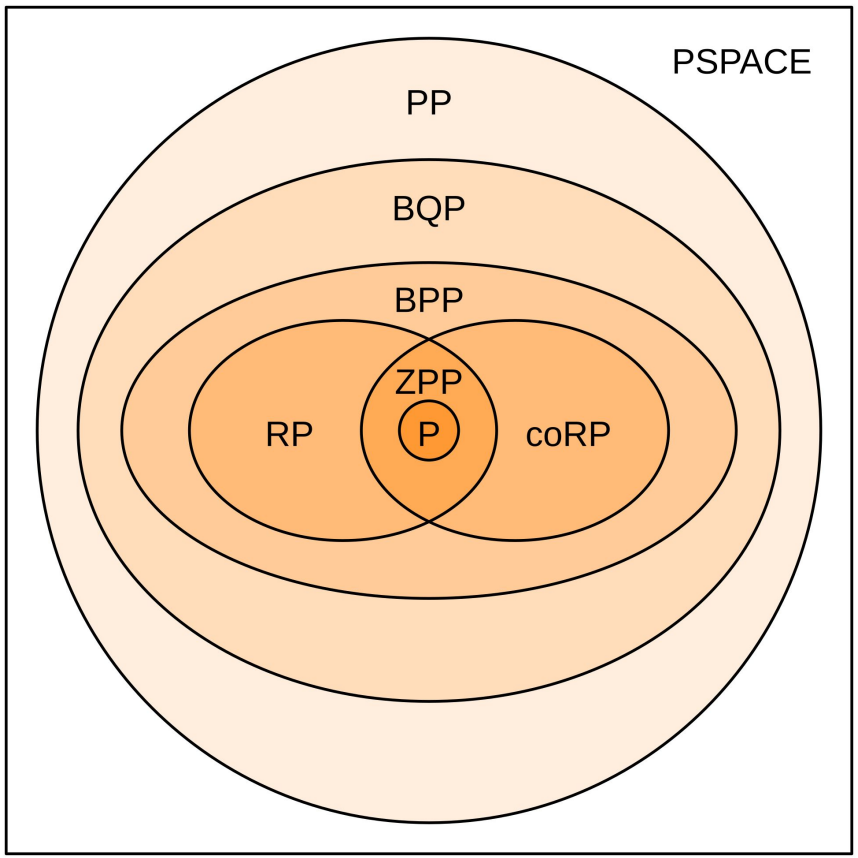
Thm: **L ∈ ZPP iff there exists a probabilistic TM M that runs in expected polynomial time and decides (i.e. only accepts and rejects and always does so correctly) L**. That is that this definition in which the output is entirely deterministic (but the runtime is now a random variable) is equivalent to our previous definition which made the parallels to (co)RP clearer

Proof:

⇒: Let M be the TM for L that satisfies our previous definition of ZPP. Construct a TM M′ that runs M until it either accepts or rejects then returns that decision. It is clear that this decides L but we need to verify that the expected running time is polynomial. The running time of M is polynomial and the expected running time of M′ is this multiplied by the expectation of the number of bernoulli trials before a success but this is upper bounded by (1/(1 − p)) (e.g. 2 taking the literal definition). Finally, the error probability p does not depend on the input (i.e. it is a constant) and so this multiplier can actually be ignored for asymptotics.

⇐: Let M be a TM for L that satisfies this new definition. We will construct an M′ that satisfies the previous definition. M′ will simulate M for a fixed period of time to reintroduce the determinism as required, and say dunno if M has not yet made a decision, the time chosen to be polynomial but have small enough probability of dunno. Specifically, Markov's inequality says that for non-negative rv X and constant c Pr[X ≥ cE[X]] ≤ 1/c, and thus running M for twice (If the probability mass beyond the expectation is greater than ½, then the expectation of a non-negative X then the expectation must be higher than claimed as this term alone gives rise to a higher expectation and the remaining mass cannot decrease it due to the non-negativity) **its expected run time gives the necessary bound of ≤ 1/2**

The known randomized complexity landscape



Approximation Algorithms

α-approximation

- **An algorithm is an α-approximation algorithm iff it is a polynomial-time algorithm which finds a solution that is within a factor α of the optimal solution**
  - E.g. a 2-approximation algorithm for independent set will find one of size at least m/2 where m is the size of a maximal independent set whereas a 2-approximation algorithm for vertex cover will find one of size at most 2m where m is the size of a minimal vertex cover
- MAXSAT = Given a CNF formula find an assignment to variables that maximizes the number of satisified clauses
- Proposition: The following very simple algorithm is a 2-approximation algorithm for MAXSAT: Consider setting all variables to true and setting all variables to false; return the best of these two options

  Proof: It is obvious this runs in polynomial time

  Let M be the answer to MAXSAT for some input and c be the number of clauses in that input. Clearly, $M \leq c$ and so for a 2-approximation it suffices to show that we return at least c/2. Let m_0 be the number of clauses satisfied when all variables are false and m_1 when true. Our algorithm returns $\max(m_0, m_1)$. Deduce that as the assignments are exactly the opposite of each other, each clause is satisfied by at least one of these two assignments (as it is a CNF, the clauses are disjunctive) and thus, $m_0 + m_1 \geq c$. Thus, $\max(m_0, m_1\} \geq c/2$ as required.

(F)PTAS

- A polynomial time approximation scheme (PTAS) for a problem is a family of approximation algorithms for that problem that allows you to get arbitrarily close to the optimal solution in polynomial time (with the time increasing as you get closer).
- Precisely **a PTAS is an infinite family A of algorithms such that $\forall \varepsilon > 0.\ \exists A_\varepsilon \in A$: A_ε is a $(1 + \varepsilon)$-approximation algorithm**
- A PTAS may not be practical if the running time increases fast as ε decreases (i.e. as $1/\varepsilon$ increases), **a fully polynomial time approximation scheme (FPTAS) has polynomial running time not only in n but also in $1/\varepsilon$**
- E.g. $O(n^{2/\varepsilon})$ is a PTAS but not an FPTAS (as it is exponential in $1/\varepsilon$) whereas $O(n^3/\varepsilon^2)$ is an FPTAS

Knapsack FPTAS

- Knapsack = Given items 1, …, n each with integer value $v_i > 0$ and integer weight $w_i > 0$ find a subset of items that maximize the total value subject to the constraint that the total weight does not exceed some given threshold W.
    - Deciding whether a Knapsack instance has a solution with total value at least V is NP-complete as it is easy to see that SUBSET-SUM reduces to it (V=W=T and $v_i = w_i = n_i$) (and that it is in NP). Thus, Knapsack is NP-Hard
- Theorem: There exists a FPTAS for Knapsack
  Proof: AWLOG we have a pre-processing step that throws out items whose individual weight exceeds the threshold.
  Let OPT(i, v) = the minimum total weight of a subset of the first i items with total value at least v
  = 0 if $v \leq 0$; +inf if $v > 0$ and $i = 0$; $\min\{ OPT(i-1, v), w_i + OPT(i-1, v-v_i) \}$ otherwise
  This has running time O(1) for each entry and so $O(n^2 v_{max})$ overall which is only pseudo-polynomial. However, the total value of the subset is the answer to our optimization problem and so is what we are allowed to approximate. Thus, we can squish the values into a small enough range to get a polynomial time approximation algorithm.
  Let $\Theta = \varepsilon v_{max}/n$. Let $v'_i = ceil(v_i/\Theta)$ for all i. Then, using our DP with the v'_is gives running time $O(n^2 v'_{max}) = O(n^2 * (v_{max}/(\varepsilon v_{max}/n))) = O(n^3/\varepsilon)$ which is fully polynomial but it remains to show that this gives a $(1+\varepsilon)$-approximation.
  Let S the solution found by the DP with the rounded input and S^* be the optimal solution (e.g. the solution found by the DP with the original input). We will show that $(1+\varepsilon) \sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$ as required.
  As we rounded the values up to obtain the v's, $\sum_{i \in S^*} v_i \leq \sum_{i \in S^*} v'_i$. As S is an optimal solution for the v's, $\sum_{i \in S^*} v'_i \leq \sum_{i \in S} v'_i$.
  As we have only rounded up the v's by adding at most Θ, $\sum_{i \in S} v'_i \leq \sum_{i \in S} (v'_i + \Theta) \leq n\Theta + \sum_{i \in S} v'_i = \varepsilon v_{max} + \sum_{i \in S} v'_i$ [defn of Θ] $\leq (1+\varepsilon) \sum_{i \in S} v'_i$ [$v_{max} \leq \sum_{i \in S} v'_i$] .
  We have built up a giant chain of inequalities and taking the very start and end gives $\sum_{i \in S^*} v_i \leq (1+\varepsilon) \sum_{i \in S} v_i$ as required.