ATCS summary

Axel Angel

1 Models of computation

- Recursive functions
- λ-calculus
- Turing machines

1.1 Recursive functions

Base functions

- Zero(x) $\mapsto 0$ (x is an optional argument)
- Succ(x) $\mapsto x+1$
- Family of projections $\pi_i^n(x_1, x_2, \cdots, x_n) = x_i$

Creating functions from existing ones

- Composition of functions
- Recursion (base case easy and generalisation)

Composition

$$\begin{array}{lcl} f(x_1,\cdots,x_n) & & h(y_1,\cdots,y_m) \\ g_1(y_1,\cdots,y_m) & \Rightarrow & f(g_1(y_1,\cdots,y_m), \\ g_2(y_1,\cdots,y_m) & & g_2(y_1,\cdots,y_m), \\ \vdots & & \vdots \\ g_n(y_1,\cdots,y_m) & & g_n(y_1,\cdots,y_m)) \end{array}$$

Recursion

$$\begin{cases} f(i+1,x,y,z) = g(i,f(i,x,y,z),x,y,z) \\ f(0,x,y,z) = h(x,y,z) \end{cases}$$

Given g, a recursive function of $n+1$ arguments, and h, a recursive function of $n-1$ arguments. We define f, a function of n arguments, as

$$\begin{array}{rcl} f(0,x_1,x_2,\cdots,x_n) & = & h(x_2,x_3,\cdots,x_n) \\ f(i+1,x_2,\cdots,x_n) & = & g(i,f(i,x_2,\cdots,x_n),x_2,\cdots,x_n) \end{array}$$

Example: Add defined by recursion, Add:

$$\begin{array}{rcl} f(0,y) & = & \pi_1^1(y) = y \\ f(i+1,y) & = & g(i,\underbrace{f(i,y)}_{i+y},y) \\ & = & \text{Succ}\left(\pi_2^3(i,f(i,y),y)\right) \end{array}$$

$$\begin{array}{l} \text{Add}(0,y) = y \\ \text{Add}(i+1,y) = \text{Succ}(\text{Add}(i,y)) \end{array}$$

Example: Mult

$$\begin{aligned} f(0,y) &= h(y) = \text{Zero}(y) \\ f(i+1,y) &= g(i, \underbrace{f(i,y)}_{iy}, y) \\ &= \text{Add}(\pi_2^3(i,f(i,y),y), \pi_3^3(i,f(i,y),y)) \end{aligned}$$

level: How to define level: $\begin{matrix} 0 & \text{if } x = 0 \\ 1 & \text{otherwise} \end{matrix}$; done like this:

$$\begin{aligned} \text{level}(0) &= h() = \text{Zero}() \\ \text{level}(i+1) &= g(i, \text{level}(i)) = \text{Succ}(\text{Zero}(\pi_1^2(i, \text{level}(i)))) \end{aligned}$$

guard: How to return 0 if $x = 0$ or y otherwise? $\text{guard}(x,y) = \text{Mult}(\text{level}(x), y)$

Conclusion: We still haven't the whole capabilities, by using Cantor theorem with a table with the recursive functions we defined in row and possible arguments in column. We then define $g(x) = \text{Succ}(f_x(x))$. With recursive functions as defined above, we don't capture the whole possibilities of programming.

1.2 Primitive recursive functions

$$\begin{aligned} \text{Exp}(0,y) &= y \\ \text{Exp}(i+1,y) &= \text{Mult}(y, \text{Exp}(i,y)) \\ &= y \times \text{Exp}(i,y) \\ &= y \times y \times \text{Exp}(i-1,y) \end{aligned}$$

We can continue to define increasingly big functions:

$$\begin{aligned} \text{Tower}(i+1,y) &= \text{Exp}(y, \text{Tower}(i,y)) \\ &= \text{Tower}(i,y)^y \\ &= (\text{Tower}(i-1,y)^y)^y \end{aligned}$$

We can define generally these kind of combining functions:

$$\begin{aligned} f_{n+1}(0,y) &= y \\ f_{n+1}(i+1,y) &= f_n(y, f_{n+1}(i,y)) \end{aligned}$$

These functions f_i stays primitives recursive functions. We combine the fastest growing function we have yet and create one which grow faster than the old one.

1.3 Ackermann's function

Let's define $F(n =< i,j >) = f_n(n,n)$, but F is not in our collections of $\{f_i\}$ functions. Suppose F is in the set $\{f_i\}$ then $\exists j : F = f_j, F(j+1) = f_{j+1}(j+1,j+1) \neq f_j(j+1,j+1)$. F is not a primitive recursive function. So we have a partial ordering of growing: $\{\underbrace{f_0}_{\text{Succ}}, f_1, \cdots\} < F$.

We can obviously combine these functions F the same way: $\{\underbrace{g_0}_{F}, g_1, \cdots\} < G(n) = g_n(n,n)$.

And again $g_{n+1}(i+1,y) = g_n(y, g_{n+1}(i,y))$. We can continue again and again. This is called *Grzegorczyk* hierarchy.

1.4 μ-recursion (unbounded search)

The recursive functions means are growing too slowly. We need an other mechanism of creating functions. We reserve f, g, h for *total functions*, functions that are well defined for all inputs; and ϕ, ψ for partial functions.

Given a function ψ of $n+1$ arguments, define a new function ϕ of n arguments by μ-recursion as follows:

$$\phi(x_1, \cdots, x_n) = \text{argmin } m : \psi(m, x_1, \cdots, x_n) = 0 \ \& \ \forall i, 0 \leq i \leq m, \psi(i, x_1, \cdots, x_n) \text{ is defined}$$

A function defined from Zero, Succ, P_i^j through a finite number of applications of composition, primitive recursion and (μ-recursion) are called (partial) recursive functions.

1.5 Enumeration of partial recursive functions

Let us enumerate partial recursive functions. Let attribute codes to our functions and use the pairing functions to reduce any code to one big number :

- $< 1 >$: Zero
- $< 2 >$: Succ
- $< 3, i, j >$: P_i^j
- $< 4, < h >, < g_1 >, \cdots, < g_n >>$: composition h of n arguments, $g1, \cdots, g_n$ of m arguments.
- $< 5, < h >, < g >>$: primitive recursion $< g_1 >$.
- $< 6, < \psi >>$: μ-recursion

Use pairing to get ψ. One number that codes up the partial recursive functions. By convention we will says that invalid code are undefined programs (that goes into a loop forever whatever input is). That method of enumeration is called *Gödel numbering Arithmetization.*

1.6 Programming system

We define the family of partial functions (called a *Programming system*): $\{\phi_0, \phi_1, \cdots\}$. We have a way to launch these functions: $\phi_{univ}(i, j) = \phi_i(j)$. We have the composition function: $c(i, j)$. It gives us: $\phi_i \circ \phi_j = \phi_k$ where $k = c(i, j)$.

The series $\{\phi_0, \phi_1, \cdots\}$ is called a programming system. An acceptable programming system is composed of :

- a universal function ϕ_{univ} and
- a composition function c.

Roger's isomorphism theorem: all acceptable programming systems are isomorphic (bijection and respect with operations). That is: acceptable programming systems are equivalent (a program A in system ϕ can be expressed in system ψ giving program B: same results). Note that: if $\phi_{univ} = \phi_u$ then $\psi_{f(u)} = \psi_{univ}$.

Example: If we have ϕ_i, ϕ_j in system ϕ and:

$$\phi_i \circ \phi_j = \phi_{c(i,j)} = \phi_k \Leftrightarrow \psi_{f(c(i,j))}$$
$$\phi_i \circ \phi_j \Leftrightarrow \psi_{f(i)} \circ \psi_{f(j)} = \psi_{d(f(i),f(j))}$$

If f is an isomorphism, then $f(c(i, j)) = d(f(i), f(j))$ where d is the composition in system ψ and c is the one for ϕ.

If φ_j is total (thus $\varphi_j = f$) and f is a composition function, that is $\varphi_{f(x,y)} = \varphi_x \circ \varphi_y, \forall x, y$.

s-m-n theorem in any acceptable programming system, with $m = 1$ and $n = 1$ $\exists i$, an index, such that φ_i is total, that is $\varphi_i = s$, such that $\varphi_j(x, y) = \varphi_{s(x,j)}(y)$ $\forall j, x, y$. This is a way to move variables between data (input, arguments) and in the program code (hard coded in the code).

Example given two acceptable programming systems:

$$\{\varphi_0, \cdots\}, \{\psi_0, \cdots\}$$

then $\exists i$, an index, such that φ_i is total, $\varphi_i = t$, such that $\psi_j(x) = \varphi_{t(j)}(x)$.

universality we need t, such that $\varphi_{t(i)}(x) = \psi_(i), \forall i, x$. $\{\psi_i\}$ has a universal function $\psi_{univ} \leftrightarrow \varphi_k$ the same function (but perhaps not universal in the $\{\varphi_i\}$ system. We have $t(i) = s(k, i)$:

$$\varphi_{t(i)}(x) = \varphi_{s(k,i)}(x) = \varphi_k(i, x) = \psi_{univ}(i, x) = \psi_i(x)$$

Boolean functions (or set membership) given a set S, what can we say about the question "does x belongs to S?" We can decide membership, for example, for S there exists a total recursive function c such that:

$$\begin{array}{ll} c(x) = 0 & x \notin S \\ c(x) = 1 & x \in S \end{array}$$

there are the recursive sets. If S is recursive, so is $\overline{S}$.

Halting problem If we have to answer to these questions:

- $\varphi_x(y)$ halts, we can answer "yes" but we cannot say no.
- φ_x halts on all inputs, we can't answer at all because there is an infinite number of inputs. No answer at all.
- The set $\{y|\varphi_x(y) \text{ defined}\}$ cannot be enumerated, but it is enumerated (running the program for each input, one step at a time for each input, because y is finite). This is a recursively enumerable set (r.e.).
- The set $\{x|\forall y \in \mathbb{N}, \varphi_x(y) \text{ defined}\}$ cannot be decided and cannot be enumerated. This is *not* a recursively enumerable set (r.e.).

Step function We could prove that any acceptable programming system has a "step" function: a total recursive function $\varphi_i = \text{step}$ such that:

$$\text{step}(j, k, x) : \begin{array}{ll} 0 & \text{if } \varphi_j(x) \text{ has not stopped after } k \text{ steps} \\ \varphi_j(x) + 1 & \text{otherwise} \end{array}$$

Enumerable set with step We can make a table with inputs (arguments) as columns and the number of steps as rows. By going on back diagonals we can enumerate the outputs and with step we can halts on every value <step, input>. We can print out, if $S \neq \emptyset$:

$$\pi_2 \left(\min z \left[\text{step}(x, \pi_1(z), \pi_2(z)) \neq 0\right]\right) \in S$$

definitions We have:

- Recursive sets are decidable.
- Recursively enumerable sets are semi-decidable (synonym).

and moreover if S is RE, but not recursive, then $\overline{S}$ cannot be RE. We have the following sets: primitive recursive functions $\subset$ total recursive $\subset$ partial recursive $=$ programs $\subset$ all math functions. Similarly with sets: recursive sets $\subset$ RE sets $\subset$ all sets.

basic RE, non recursive set Let's define the diagonal halting set by $K = \{x \mid \varphi_x(x) \text{ defined}\}$.

Rice's theorem If P is a predicate about the I/O behavior of programs and define $\chi_P = \{x \mid P(\varphi_x) \text{ is true}\}$ then the Rice's theorem says:

$$\text{if } \chi_P \neq \emptyset \wedge \overline{\chi}_P \neq \emptyset \to \chi_P \text{ is not recursive}$$

and the proof is a reduction from K to χ_P written $K \leq \chi_P$, but we know we can't decide membership of K (halting problem).

Other definition of RE A set is RE iff it is the domain/range of a partial recursive function, that is: given S, S is RE iff $\exists i$ such that dom $\varphi_i = S = \{x \mid \varphi_i(x) \text{ defined}\}$.

Note: Equality We have $\varphi_i = \varphi_j$ we have $\forall x$:

- $\varphi_i(x)$ undefined iff $\varphi_j(x)$ undefined, and
- $\varphi_i(x)$ defined iff $\varphi_j(x)$ defined, and
- if $defined$ then output must be equal: $\varphi_i(x) = \varphi_j(x)$.

Recursion theorem: $\forall f$ total recursive functions, $\exists i$ such that $\varphi_i = \varphi_{f(i)}$, a fixed point for f.

Algorithmic information theory: or information complexity theory (due to Kolgomorov and Chaitin). Let's define $f(n)$ to be the shortest program that prints n and hals, then f is not computable. Proof: Let's $g(x) = \min i(f(i) \geq m)$, and the length of f is fixed, some $\theta(1)$; the loop $\min i$ and test use $\theta(1)$ } + code m_i $\log_2(m)$ bits but size of $g < m$.

Probability of halting Given a random program, what is the probability that it is a halting program (total). Length $|x|$ has probability of $2^{|x|}$. If x is a halting program, then no w such that $wz = x$, $|z| \geq 1$, is a halting program. We define:

$$\Omega = \sum_{\substack{\text{all total not RE} \\ \text{program x}}} 2^{-|x|}$$

2 Complexity

Define classes of problems in terms of resources and time bounds. Given some $f(n)$, $f(n) > 0$ $\forall n$. Define class χ of all problems solvable in $f(n)$ time / space. f must be "constructible", ie: $\exists$ TM that runs on input n in exactly $f(n)$ steps. We can use a specific TM that runs other TM and check how many steps (time) or states (memory) are used.

Hierarchy theorems (for space and time)

space: $(\chi_f \subsetneqq \chi_g)$ if $\lim_{n\to\infty} \inf \frac{f(n)}{g(n)} = 0$ then there exists a problem solvable in $g(n)$ space but not in $f(n)$ space.

Proof: Let M a TM that runs in $g(n)$ space. $M(z)$: runs $\varphi_{\pi_1^3(z)}(\pi_2^3(z)$ for $\pi_3^3(z)$ steps, where $z = <$ program , input , number of steps $>$. Subject to staying on marked tape squares (we limit our space by marking some squares on the machine tape). If $M(z)$ halts, staying on marked squares, change the resulting tape contents, otherwise $M(z) = 0$. We iterate over the program/input pairs which lies on the diagonal: $(\varphi_0, 0), (\varphi_1, 1) \cdots$.

time: $(\chi_f \subsetneqq \chi_g)$ if $\lim_{n\to\infty} \inf \frac{f(n)\log(f(n))}{g(n)} = 0$ then there exists a problem solvable in $g(n)$ time, but not in $f(n)$ time. The log is here because we have to memorize the current state (done in binary for example).

Model-independent classes of complexity Our classes were dependant on the model we have but we can generalize.

$$\left.\begin{array}{l}\text{time } f(n)\\ \text{space } f(n)\end{array}\right\}\text{on machine model 1}$$
$$\downarrow \text{translation}$$
$$\left.\begin{array}{l}\text{time } O(f(n)g(n)) \sim O(f^2(n))\\ \text{space } \theta(g(n))\end{array}\right\}\text{on machine model 2}$$

If time is $f(n)$, what space? $O(f(n))$. If space is $g(n)$, what is time (for a halting computation)? $O(c^{g(n)})$, because number of configurations in TM is: $|Q|g(n)|\Sigma|^{g(n)} \leq c^{g(n)}$, where $g(n)$ is the head position.

3 Randomized algorithm

3.1 Travelling salesperson problem

We know that problems like the travelling salesperson problem are NP and so there doesn't exist solution in polynomial time. However we can use heuristic algorithm to solve this problem but we don't know if this is optimal. For example we can always choice the nearest unvisited neighbour yet.

If we have the inequality $d(i,j) \leq d(i,k) + d(k,j)$ we can solve in a special way: construct a minimal spanning tree for the cities, convert the tree to a tour. Start anywhere and visit nodes on the tree but short-circuit node already visited (go to the next node directly, not using tree edges). This solution yields a solution S that is at most twice the cost of the minimal spanning tree MST*. Because: $S \leq 2MST^*$ and MST$^* \leq SPT \leq TSP^*$ then $S \leq 2MST^* \leq 2TSP^*$.

3.2 Other problem

SAT: Pack a random truth assignment (2^n choices for n vars) and compute the value of the formula:

- if this eval to true: then answers is yes (it is satisfiable), correct
- if false: answer no, error is $\leq 1 - \frac{1}{2^n}$ (too small).

If we are getting no, we can improve our answer (in term of error) by trying again multiple times.

Generalization: we can generalize this solution to other problems: we can pick a random possible solution and test it. Lucky if we found a solution otherwise try again.

For example with matrix multiplication 0/1 vector $x_{n\times 1} \in \{0,1\}^n$. We want to calculate $Cx? = A(Bx)$. The naive method is $\theta(n^2)$ but we can use the randomized algorithm. If $Cx \neq A(Bx)$ then answer "no", correct, but if $Cx = A(Bx)$ then answer "yes" with error $\leq \frac{1}{2}$ only. As proof for this last probability we test if $Dx = 0$ where $D = AB - C$, and there is error if $D \neq 0$: $\sum_{i=1}^{k-1} x_i a_i + x_k a_k = 0$.

4 the Randomized Model

Comparison with other model It allows to yields uniformly distributed random samples. This introduces next to the non- and deterministic models (external help: correct divine guess and no help respectively), the *randomized* model (external help: random uniform guess).

Introduction: There may be bound for both answer “yes” and “no” for example:

- answers “yes”, with error $\leq \frac{1}{4}$
- answers “no”, with error $\leq \frac{1}{4}$

so we can run n times and get $n\alpha$ yes and $(1-\alpha)n$ no then we can compute the preferred answer and a probability of error. We analyse the different sides:

1-sided: error bound on the answer: 0, and the other $1-\alpha$. We have α a fixed fraction or $\alpha = \Omega(\frac{1}{O(n)})$.

2-sided: (general case) each probability of error: one side $\beta - \alpha$, the other $(1-\beta) - \alpha$.

4.1 Probabilist complexity classes

The optimal algorithm is deterministic and always gives the correct answer. These are in complexity classes: L, P, POLYL, EXP. We need new classes for probabilist algorithms: RP (random), co-RP; both two classes contains the problems for which $\exists$ one-sided (one of the answer is correct at 100%, the other has some bounded error) randomized algorithm. The RP and co-RP classes are contained in the class: BPP. The intersection of these two is the ZPP (zero probability polytime) class.

One-sided random algorithm:

- an algorithm that runs in polynomial time
- can make random choices (fair probability)
- answers yes or no; one answer is 100% correct, the other has an bounded error of $\leq \frac{1}{2}$.

The probabilist complexity classes has the one-sided property:

RP : if answer is “yes” then it is for sure. If “no”, the error is bounded.

co-RP : if answer is “no” then it is for sure. If “yes”, the error is bounded.

Parallel with deterministic classes: We have P $\subseteq$ RP and P $\subseteq$ co-RP. Because these algorithm has no error, their error on one side is bounded, it is always 0: the answers are corrects. Moreover NP is a nondeterministic class: RP $\subseteq$ NP. We could define co-NP such that: RP $\subseteq$ co-NP.

Two-sided random algorithm: The BPP class (bounded probability polytime) is two-sided and has properties:

- an algorithm that runs in polynomial time
- can make random choices (fair probability)
- answers yes or no; both answers comes with an bound error of $\lneq \frac{1}{2}$.

ZPP: We define ZPP = RP $\cap$ co-RP but this is not equal to P. Because in ZPP we may run the two one-sided algorithm which gives one answer for sure for the same problem, but what happens if the two algorithms keep saying their unsure answer (decreasing error) and never answer anything else. We have little information, and so these problems are in ZPP but not in P.

```
      PP
   /  |  \
NP   BPP  coNP
|  /    \ |
 RP      coRP
  \     /
    ZPP = RP inter coRP
     |
     P
```

ZPP ZPP Zero poly-time probability error class (small probability that it will take more time). Algorithm that's always correct but may exceed polytime (so we can stop it and give an arbitrary answer, our error is still bounded).

Two coRP algorithms If we have two algorithms A_1 and $A_2 \in$ RP $\cap$ coRP then:

- when A_1 answers "yes" then it is correct;
- when A_1 answers "no" then it as probability error $\leq 1 - \varepsilon$;
- when A_2 answers "yes" then it as probability error $\leq 1 - \varepsilon$;
- when A_2 answers "no" then it is correct;

We run A_1 and A_2 in a lockstep (in parallel), but how many loops?

- If A_1 answers "yes", or A_2 answers "no", stop and answer the same
- otherwise repeat.

The expected number of loops is constant: $\sum_{i=1}^{\infty} ip_i$ where p_i is the probability of i loops. The running time is exponential polytime.

We can always convert a one-sided probability check to a two-sided: by using a coin multiple times then based on this result we run the test and answer the same or just answer the always-correct answer (with bounded error).

4.2 Types of Proofs

Batch-style proofs: (à la NP: claim and a certificate that can be checked):

prover: very smart (lots of computational power)

checker: polytime (randomized)

Interactive proofs (involve communication):

prover: very smart

checker: polytime (randomized), more than one?

special case: zero-knowledge proofs (eg: crypto), without revealing the solution.

Example of interactive zero-knowledge proof: We claim we can solve the graph coloring problem (3-colorable is NP-complete problem) efficiently. We send an encrypted version of the coloring solution (one key per edge) and the checker will be given it without the keys. Then the checker requests the color of a pair of edges, the claimer sends the key for these and can see the color of the chosen edges. For the next round the claimer permutes the colors, send the encrypted version of it: after k rounds, the error $\leq (\frac{m-1}{m})^k$.

Graph isomorphism checking: We are given two graphs G_1 and G_2 and we want to test if they are isomorphic. That is $\exists f : V_1 \to V_2$ such that:

- f is a bijection;
- if $\{u, v\} \in E_1$, then $\{f(u), f(v)\} \in E_2$

and we can use a probabilist checker running in polytime. The prover has an unbounded power.
The interaction we would like:

- If the answer is "yes", a good prover can convince the checker with probability $\geq \frac{1}{2} + \varepsilon$
- If the answer is "no", no prover can convince the checker (the checker rejects with probability $\geq \frac{1}{2} + \varepsilon$)

and how to do it (protocol to test whether G_1 is not isomorphic to G_2): pick an index $i \in \{1, 2\}$ and send H, a relabelled version of G_i (permutation of the graph vertices indexes). Ask prover: return index j such that $H \equiv G_j$. We check if $i = j$.

- If $G_1 \equiv G_2$, then $H \equiv G_1$ and $H \equiv G_2$, so the prover cannot decide which index to return and so it has 50% chance to disagree with the checker.
- If $G_1 \not\equiv G_2$, then an honest prover can always tell the correct index.

Interactive proof system:

- A checker (BPP)
- A prover (EXP)

Protocol: a series of "rounds" during when:

- the checker asks question (polytime)
- prover sends an answer (no time)
- checker verifies the answer (polytime)

IP decision problem class We define the class IP: the class of decision problems solvable in the following sense: $\exists a$, constant and $\varepsilon > 0$, a checker program P that runs in polytime such that:

- a "yes" instance of problem is accepted by P with probability $\geq \frac{1}{2} + \varepsilon$ ($\exists$ prover that will make this happen)
- a "no" instance is rejected by P with probability $\geq \frac{1}{2} + \varepsilon$ ($\forall$ prover)

and we define IP(f) : the class of decision problems that can be solved by an IP protocol in $f(n)$ rounds. We are interested in IP$(\theta(n^{O(1)}))$. We define **MIP** a class similar to IP but uses multiple provers, and **PCP** the probabilist checkable proofs (batch proofs, without interaction).

A problem P is in PCP if there exists an algorithm that runs in polytime and costs $\varepsilon > 0$ such that.

- For any instance x that is a "yes" instance, $\exists$ proof string c_x such that the algorithm runs on x and c_x is accepted with probability $\geq \frac{1}{2} + \varepsilon$ with a limited access (bounded) to c_x.
- For any "no" instance and any proof string c_x, the algorithm rejects it with probability $\geq \frac{1}{2} + \varepsilon$.

Let's parametrize PCP(f, g) where:

- $f(n)$: the number of random bits you can use;

- $g(n)$: the number of bits that can be read from the proof

then we have:

- PCP$(0, 0)$ = P;
- PCP$(n^{O(1)}, 0)$ = BPP
- PCP$(0, n^{O(1)})$ = NP
- PCP$(\log(n), O(1))$ = NP (**PCP theorem**, big result)

(don't forget we have to answer "yes" or "no" with a bounded error on each randomised-chosen chunk of the proof)

5 Approximations

5.1 Simpler special cases in hard problems

If a problem is in NP-complete it doesn't mean there isn't case which are simpler and could be solved more efficiently. We now study some particular cases:

- specials cases:
  - graph problems: graph can be planar or can have a bounded degree;
  - SAT: 2-SAT is in P but 3-SAT in NP;
- approximation algorithms:
  - within a constant distance of optimal
  - within a constant ratio from optimal
- heuristics: uses some insight of the problem but can be fooled (no guarantees)

Example of approximation: Vertex cover: pick an uncovered edge (if any) and put both endpoint in the cover; remove covered edges and redo (returns $\leq 2\times$ optimal)

Example of heuristics: pick the highest degree and put it in the cover and remove covered edges. Empirically in practice this algorithm does better than the approximation but it can be fooled. There is no proof of that, but only a lots of tests was done and this was the case.

Example of special cases:

- SAT is NP-complete
  - 3-SAT is NP-complete
  - 2-SAT is P
- Vertex cover is NP-complete
  - VC with degree 3 is NP-complete
  - VC on planar graph is NP-complete
- Maximum cut is NP-complete
  - MC with bounded degree is NP-complete
  - MC on planar graph is P

- graph 3-color (G3C) is NP-complete
  - G3C on graph of degree 3 is NP-complete
  - G3C on planar graph is NP-complete (we can convert non-planar to planar graph with a transformation, a gadget)

Uniquely promised 3-SAT (UP3-SAT): if the formula is satisfiable, then I'm guarantee that there is only one such assignment. To check this is the case it seems difficult because we would have to try all assignments (negation problem). There are 3 categories of SAT-equation:

- The yes part with more than one assignments: original "no" instance remains "no" instances.
- The promised equations are the "yes" instances that obey the promises remains "yes".
- The rest (no true assignment) then we don't care and assume we are given not these cases.

then UP3-SAT cannot be solved in polytime unless P = NP.

Graph edge-coloring approximation: That is to find the required the number of colors (chromatic index: ch.i.) to color a graph such that so vertices with the same color are connected with an edge. We have: ch.i. $\leq$ max degree. The Vizing theorem says: if max degree is k then ch.i $leqk + 1$.

Bin-packing: Given the size function, $s : S \mapsto \mathbb{N}$, $B \in \mathbb{N}$ and $K \in N$, can we partition S into K subsets $\{S_1, \cdots, S_k\}$ such that $\forall i \sum_{x \in S_i} s(x) \leq B$. This prolbem is NP-hard even if $K = 2$!

We can use a greedy algorithm to solve it: sort the item in increasing order by size: $x_1, \cdots, x_n$ so $s(x_i) \leq s(x_{i+1})$. Find max j' such that: $\sum_{i=1}^{j'} s(x_i) \leq B$. The optimal solution $\leq j''$: find max j'' such that $\sum_{i=1}^{j''} s(x_i) \leq 2B$. Our approximation gives:

$$\begin{array}{llll}
\underbrace{x_1, x_2, \cdots, x_{j'},}_{\text{in bin 1 } : \sum \leq B} & j_{j'+1}, & \underbrace{j_{j'+2}, \cdots, x_{j''},}_{\text{in bin 2}} & \cdots, x_n \\
<- & \sum \leq 2B & \qquad\quad -> &
\end{array}$$

approximation yields $j'' - 1$ items and optimal solution is $\leq j''$ items.

Max SAT: satisfy as many clauses as possible. Claim: Max SAT cannot be approximated within a constant distance unless P = NP. Proof: by contradiction, we assume $\exists$ algorithm A and a constant $d > 0$ such that on any Max SAT instance I: $A(I) \geq I^* - d$ make $(d+1)$ "copies" of the clauses then we run A: one of the copies must be solved optimally (pigeonhole principle). That way we solved SAT in polytime but it isn't possible.

5.2 Type of approximation

- best: within a fixed distance of optimal
- proportional complexity: within arbitrary distance but it costs proportionally with it.
- ok: within a fixed ratio

5.3 Approximation complexity

We must define classes and a reduction (which preserves ratios).

APX: (APproXimation) $\exists$ polytme algorithm A and a ration ε such that $A(x)$ is within a ration ε of the x^* (optimal). e.g.: VC, Max3SAT $\in$ APX.

PTAS: (Polytime Approximation Scheme) $\exists$ family of algorithms $\{A_j\}$ such that, for any ratio ε, $\exists i$ such that $A_i(x)$ gives a solution within ratio ε of x^* and each A_i is a polytime algorithm.

5.4 MaxKSAT approximation

we develop an approximation algorithm for this problem using a notion of weight for clauses. A clause of n variables has weight 2^{-n}. The algorithm is:

- Pick any remaining unassigned variable x
- Compute the weights:

$$w_t = \sum_{\substack{\text{clauses } c \\ \text{containing } x}} 2^{-|c|} \quad ; w_f = \sum_{\substack{\text{clauses } c \\ \text{containing } \overline{x}}} 2^{-|c|} \quad ; w_{nx} = \sum_{\substack{\text{clauses } c \\ \text{not containing } x}} 2^{-|c|}$$

  where $|c|$ is the number of unassigned variable in the clause. Define $w = w_f + w_t + w_{nx}$, if $w_t \leq w_f$ then set $x \leftarrow$ True else $x \leftarrow$ False.
- Repeat until all variables are assigned.

This algorithm leaves at most w_{initial} clauses unsatisfied.

Proof By induction on the number of clauses.

- If there is only one clause: trivial (satisfied).
- Induction hypothesis (IH): Ok on all instances of $\leq m$ clauses, consider an instance with $m+1$ clauses. Initial weight w_{m+1}, pick unassigned x, define w_t, w_f, w_{nx}. Assign $x \leftarrow$ True because $w_t \leq w_f$ so:

$$w_{m+1} = w_t + w_f + w_{nx} \geq 2w_f + w_{nx}$$

  fewer clauses left then by IH, the remaining clauses have weight $w_{nx} + 2w_f$ (because we assigned True (one less unassigned variable), the false clauses have doubled weight). At most $w_{nx} + 2w_f$ clauses will be left unsatisfied. Overall the number of clauses left unsatisfied is $\leq w_{nx} + 2w_f \leq w_{m+1}$.

5.5 APX reductions

We have to distinguish mapping of instance and mapping of (approximation) solution:

Problem A		Problem B
I	$\xrightarrow{f}$	$f(I)$
$\not\downarrow A'$		$A(f(I)) \downarrow$
$\hat{I}$	$\xleftarrow{g}$	$\widehat{f(I)}$
$h(\varepsilon) = \varepsilon'$	$\xleftarrow{h}$	ε

$A(I) = g(A(f(I)))$, there is conditions on f and g, must ensure some $\varepsilon' \in \mathbb{Q}$ (ratio) given A, ε. An APX reduction is a triple (f, g, h) so $A \leq B$ where $f : A \mapsto B$ is an instance mapping ($I \in A$ then $f(I) \in B$), g is a solution mapping and h is an approximation ratio mapping (all in polytime).

We have that MAX3SAT is APX-complete.

5.6 Reductions for approximations algorithm

We need to preserve approximation ratio. We had above an approximation ratio ε mapped by h to $h(\varepsilon)$. Negative result: that way we can show that if algorithm A cannot have a better ratio than $h(\varepsilon)$ then if we can reduce A to B we have that ε cannot be better than $h^{-1}(h(\varepsilon)) = \varepsilon$, which is defined by the reduction: So $\varepsilon^* \geq h(\varepsilon) \xleftarrow{h} \varepsilon$.

PCP theorem: Any problem in NP has a probabilistically checkable proof such that:

- if x is a "yes" instance, then with $\theta(\log n)$ random bits, we can use $O(1)$ bits of the proof and accept with probability $\geq \frac{1}{2} + \varepsilon$;
- if x is a "no" instance, for any choice of the random bits, the machine rejects with probability $\geq \frac{1}{2} + \varepsilon$.

Probabilistic proof checker: It uses $c_1 \log n$ random bits, look at c_2 bits of the proof. The checker runs in deterministic polytime $p(n)$ with $n = |x|$, using as inputs x, $c \log n$ bits, c_2 bits from proof. The possible paths of execution on inputs x are determined by $c_1 \log n$ bits which implies $2^{c_1 \log n} = n^{c_1}$ paths and $\geq (\frac{1}{2} + \varepsilon)n^{c_1}$ paths accept. For a path to accept the c_2 bits have to be "right".

The reduction from NP to Max3SAT: So we write a boolean formula describing all computation paths. There is n^{c_1} terms in it, one for each path, all AND'ed together (conjunction). An accepting path depends on the c_2 bits so there is 2^{c_2} possible combinations at most we write a boolean formula for the accepting combinations. The formula will ask truth values for the all bits of the proof.

Problem $A \in$ NP $\leq$ Max3SAT, we have:

- yes: $P_{\text{accept}} \geq \frac{1}{2} + \varepsilon \longleftrightarrow \geq \left(\frac{1}{2} + \varepsilon\right) \times \alpha$ of the clauses are satisfied.
- no: $P_{\text{reject}} \geq \frac{1}{2} + \varepsilon \longleftrightarrow \leq \left(\frac{1}{2} - \varepsilon\right) \times \alpha$ of the clauses are satisfied. .

We have established **a gap** between the tow, that's what we wanted. We cannot cross this gap (get better approximation) unless P = NP.

Example of G3C: We have NAE3SAT $\leq$ G3C with:

- yes $\rightarrow$ number of necessary colors ≤ 3
- no $\rightarrow$ number of necessary colors ≥ 4

and again we have a gap.