Symmetric Computation: Lecture 1

Anuj Dawar and Gregory Wilsenach

Department of Computer Science and Technology, University of Cambridge

ESSLLI, August 2021

The Science of Abstraction

Computer Science is the *Mechanization of Abstraction*.

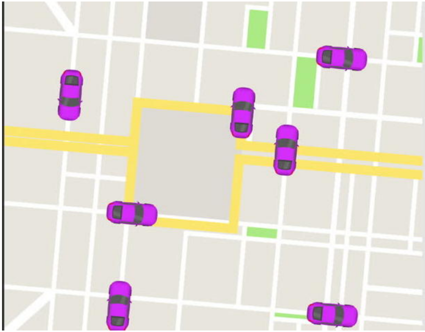
(Aho and Ullman)

The first step to solving a problem *computationally* is to strip away irrelevant concrete detail and formulate it as an *abstract* problem.

In the terms of **Aho and Ullman**, this means constructing an abstract *data model* and deciding which aspects of *concrete, messy* reality are represented there.

Example: Matching Taxis to Passengers

Example: Consider the problem of assigning a set of available taxis to a set of *waiting passengers*.



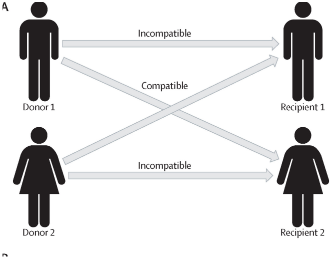
The assignment may have to minimize distance travelled by the taxis, and respond in *real-time*.

Example: Matching Kidney Patients to Donors

The English *National Health Service* runs a *kidney matching programme*.

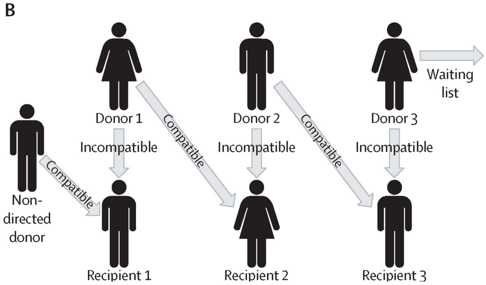
Patients needing a *kidney transplant* often have a relative who is willing to donate a kidney.
However, there may be tissue incompatibility.

If one is fortunate, a matching donor/recipient *pair* can be found and a kidney *swap* arranged:



Kidney Matching

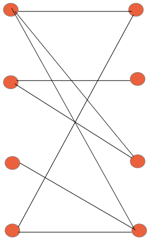
The likelihood of finding a match is greatly increased if we look for longer chains of donor/recipient matches:



Graph Matching

At the core of both is the same *algorithmic problem*.

We have a *bipartite graph* in which to find a *perfect matching*.

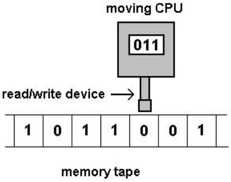


Thinking about it at this *abstract* level is what allows us *re-use* algorithmic ideas.

Note, this is *not* just about re-using *code* but also more *fundamental insights*.

Turing Machines

One of the original *abstractions* in computer science is the formalization of the notion of *algorithm* as a **Turing Machine**.



A **Turing Machine** consists of a processor (with a finite memory) and an infinite tape.

At any point, the processor can look at one symbol on the tape, and perform an action based on a finite table.

The **Church-Turing** thesis asserts that *all* models of computation are equivalent to this one.

So, why would be bother with any others?

High-level Languages and Abstraction

The reason for a wide variety of *computational models* (such as *high-level programming languages*) is that different programming applications require different *abstractions*.

A rich vein of theoretical research explores different abstractions and their formal semantic properties.

The simplicity of **Turing machines** is useful in establishing *impossibility results*.

Algorithms and Abstraction

Algorithms are usually described by operations on *abstract data* such as graphs.
The *complexity* of algorithms, and particularly *complexity classes* are defined by machine models (*e.g. Turing machines*) operating on *strings of symbols*.

The *mismatch* is generally considered harmless as data structures can be encoded as strings.

However, it does *break abstraction*.

Sometimes even *high-level* descriptions of algorithms break the level of abstraction.

Graph Matching

Given a bipartite graph $G = (A \cup B, E)$,

> *Start with an empty matching $M = \emptyset$.*
> *Choose an $a \in A$ that is currently unmatched and find an augmenting path P starting at a and ending in an unmatched $b \in B$.*
> *Set M to be $M \oplus P$.*

The *choice* of a is arbitrary and generally relies on *concrete* hidden data.

Abstract data has *symmetries* that an abstract algorithm should respect.

The Role of Symmetry

If we expect an algorithm to work *at the level of abstraction* of graphs, then it must respect *symmetries* of the graph.

If two nodes in a graph G are *indistinguishable* by properties of the graph, then they should not be distinguished in any way by the algorithm.

This opens up the question of what algorithms have the property of respecting symmetry?

Algorithms for Matching

The algorithm for finding a *maximum size matching* in a bipartite graph, based on augmenting paths goes back to **Ford-Fulkerson**.

It was in the context of algorithms for matching that **Edmonds 1965** first defined *good algorithms*, i.e. ones that run in polynomial time.

The asymptotically fastest algorithm (for general, not just bipartite graphs) is due to **Micali and Vazirani, 1980** and runs in time $O(\sqrt{|V|}|E|)$.

Can *Matching* be solved by an *efficient* and *symmetry respecting* algorithm?

For instance, it can be shown that the **Micali-Vazirani** algorithm for *graph matching* is *not* symmetry-respecting.

Symmetry from High-level Description

Algorithms that are *automatically generated* from high-level descriptions, will preserve symmetries.

This sentence says that the relation M is a matching in the graph with edge relation E.

$$\forall x \forall y [M(x,y) \to E(x,y)] \wedge \forall x \exists! y M(x,y)$$

An algorithm to search for such an M in a graph, generated from this description, would *most likely* be *exponential*.

Relational Databases

$$Cinema = \{Movies[3], Location[3], Guide[3]\}$$

Movies	Title	Director	Actor
	Magnolia	Anderson	Moore
	Magnolia	Anderson	Cruise
	Spiderman	Raimi	Maguire
	Spiderman	Raimi	Dunst
	...		
	Rocky	Avildsen	Stallone
	RockyII	Stalone	Stallone

Guide	Title	Cinema
	Rocky	Warne
	Spiderman	Picture
	...	
	Spiderman	Phoeni
	Magnolia	Picture

Location	Cinema	Address	Tel
	Picturehouse	Cambridge	504444
	Phoenix	Oxford	512526
	Warner	Cambridge	560225

Relational Algebra

In *relational algebra*, queries are built up from

Base relations: R

Singleton constant relations: $\{\langle a \rangle\}$

using

select: $\sigma_{j=a}(q)$ or $\sigma_{j=k}(q)$

project: $\pi_{j_1,\ldots,j_k}(q)$

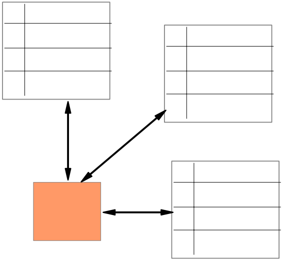
join: $q_1 \bowtie q_2$

union: $q_1 \cup q_2$

difference: $q_1 - q_2$

Relational Machines

Formal Models of algorithms that work on *abstract* structures have been well-studied in the context of *database query languages*.



Input: A relational database

Store: relational and numerical registers

Operations: *join, projection, complementation, counting*

Logic

Query languages for relational databases are often modelled in *Logic*.

The relational algebra has a natural translation into *first-order logic*.

First-order predicate logic.

> *Fix a vocabulary* σ *of relation symbols* $(R_1, \ldots, R_m)$ *and constant symbols* $c_1, \ldots, c_k$ *and*
> *a collection* X *of variables.*

The formulas are given by

$$R_i(\mathbf{t}) \mid s = t \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \exists x \varphi \mid \forall x \varphi$$

where $x \in X$; $s, t \in X \cup \{c_1, \ldots, c_k\}$ and $\mathbf{t} \in (X \cup \{c_1, \ldots, c_k\})^a$—$a$ the *arity* of R_i.

First-Order Logic

For a first-order sentence φ, we ask what is the *computational complexity* of the problem:

Given: a structure $\mathbb{A}$
Decide: if $\mathbb{A} \models \varphi$

In other words, how complex can the collection of finite models of φ be?

In order to talk of the complexity of a class of finite structures, we need to fix some way of representing finite structures as strings.

Encoding Structures

We use an alphabet $\Sigma = \{0, 1, \#\}$.
For a structure $\mathbb{A} = (A, R_1, \ldots, R_m)$, fix a linear order $<$ on $A = \{a_1, \ldots, a_n\}$.

R_i (of arity k) is encoded by a string $[R_i]_<$ of 0s and 1s of length n^k.

$$[\mathbb{A}]_< = \underbrace{1 \cdots 1}_{n} \# [R_1]_< \# \cdots \# [R_m]_<$$

The exact string obtained depends on the choice of order.

Invariance

Note that the decision problem:

Given a string $[\mathbb{A}]_<$ *decide whether* $\mathbb{A} \models \varphi$

has a natural invariance property.

It is invariant under the following equivalence relation

Write $w_1 \sim w_2$ *to denote that there is some structure* $\mathbb{A}$ *and orders* $<_1$ *and* $<_2$ *on its universe such that*

$$w_1 = [\mathbb{A}]_{<_1} \textit{ and } w_2 = [\mathbb{A}]_{<_2}$$

Note: deciding the equivalence relation $\sim$ is just the same as deciding structure isomorphism.

Naïve Algorithm

The straightforward algorithm proceeds recursively on the structure of φ:

- Atomic formulas by direct lookup.
- Boolean connectives are easy.
- If $\varphi \equiv \exists x\, \psi$ then for each $a \in \mathbb{A}$ check whether

$$(\mathbb{A}, c \mapsto a) \models \psi[c/x],$$

  where c is a new constant symbol.

This runs in time $O(ln^m)$ and $O(m \log n)$ space, where l is the length of φ and m is the nesting depth of quantifiers in φ.

$$\mathrm{Mod}(\varphi) = \{\mathbb{A} \mid \mathbb{A} \models \varphi\}$$

is in *logarithmic space* and *polynomial time*.

Second-Order Logic

There are computationally easy properties that are not definable in first-order logic.

- There is no sentence φ of first-order logic such that $\mathbb{A} \models \varphi$ if, and only if, $|A|$ is even.
- There is no formula $\varphi(E, x, y)$ that defines the transitive closure of a binary relation E.

Consider second-order logic, extending first-order logic with *relational quantifiers* — $\exists X \varphi$

Examples

Evennness

This formula is true in a structure if, and only if, the size of the domain is even.

$$
\begin{aligned}
\exists B \exists S \quad & \forall x \exists y B(x,y) \land \forall x \forall y \forall z B(x,y) \land B(x,z) \rightarrow y = z \\
& \forall x \forall y \forall z B(x,z) \land B(y,z) \rightarrow x = y \\
& \forall x \forall y S(x) \land B(x,y) \rightarrow \neg S(y) \\
& \forall x \forall y \neg S(x) \land B(x,y) \rightarrow S(y)
\end{aligned}
$$

Examples

Transitive Closure

The following formula is true of a pair of elements a, b in a structure if, and only if, there is an E-path from a to b.

$$\forall S\big(S(a) \land \forall x \forall y[S(x) \land E(x,y) \to S(y)] \to S(b)\big)$$

Matching

The following formula is true in a graph (V, E) if, and only if, the graph contains a perfect matching.

$$\exists M \; \forall x \forall y[M(x,y) \to E(x,y)] \land \forall x \exists! y M(x,y)$$

Examples

3-Colourability

The following formula is true in a graph (V, E) if, and only if, it is 3-colourable.

$$\begin{aligned} \exists R \exists B \exists G \quad & \forall x (Rx \lor Bx \lor Gx) \land \\ & \forall x (\quad \neg(Rx \land Bx) \land \neg(Bx \land Gx) \land \neg(Rx \land Gx)) \land \\ & \forall x \forall y (Exy \to (\quad \neg(Rx \land Ry) \land \\ & \qquad\qquad\qquad\qquad \neg(Bx \land By) \land \\ & \qquad\qquad\qquad\qquad \neg(Gx \land Gy))) \end{aligned}$$

Fagin's Theorem

Theorem (Fagin)
A class $\mathcal{C}$ of finite structures is definable by a sentence of *existential second-order logic* if, and only if, it is decidable by a *nondeterminisitic machine* running in polynomial time.

$$\text{ESO} = \text{NP}$$

Is there a logic for P?

The major open question in *Descriptive Complexity* (first asked by Chandra and Harel in 1982) is whether there is a logic $\mathcal{L}$ such that

> *for any class of finite structures $\mathcal{C}$, $\mathcal{C}$ is definable by a sentence of $\mathcal{L}$ if, and only if, $\mathcal{C}$ is decidable by a deterministic machine running in polynomial time.*

Formally, we require $\mathcal{L}$ to be a *recursively enumerable* set of sentences, with a computable map taking each sentence to a Turing machine M and a polynomial time bound p such that (M, p) accepts a *class of structures*.

(Gurevich 1988)

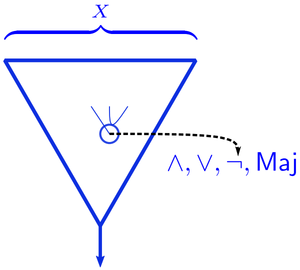
Circuits

A *language* $L \subseteq \{0,1\}^*$ can be described by a family of *Boolean functions*:

$$(f_n)_{n \in \omega} : \{0,1\}^n \to \{0,1\}.$$

Each f_n may be computed by a *circuit* C_n made up of

- Gates labeled by Boolean operators: $\wedge, \vee, \neg$,
- Boolean inputs: $x_1, \ldots, x_n$, and
- A distinguished gate determining the output.



Circuit Complexity

Circuits are just the *unfoldings* of the behaviour of an algorithm on inputs of a fixed size n into simple actions such as Boolean *AND*, *OR* and *NOT* operations.

If there is a polynomial $p(n)$ bounding the *size* of C_n, i.e. the number of gates in C_n, the language L is in the class P/poly.

If, in addition, the function $n \mapsto C_n$ is computable in *polynomial time*, L is in P.

Note: For these classes it makes no difference whether the circuits only use $\{\wedge, \vee, \neg\}$ or a richer basis with *threshold* or *majority* gates.

Circuit Lower Bounds

It is conjectured that NP $\not\subseteq$ P/poly.

Lower bound results have been obtained by putting further restrictions on the circuits:

- No *constant-depth* (unbounded fan-in), *polynomial-size* family of circuits decides *parity*. **(Furst, Saxe, Sipser 1983)**.
- No *polynomial-size* family of *monotone* circuits decides *clique*. **(Razborov 1985)**.
- No *constant-depth*, $O(n^{\frac{k}{4}})$-*size* family of circuits decides *k-clique*. **(Rossman 2008)**.

No known result separates NP from *constant-depth*, *polynomial-size* families of circuits with *majority gates*.

Circuits for Graph Properties

We want to study families of circuits that decide properties of *graphs* (or other relational structures—for simplicity of presentation we restrict ourselves to graphs).

We have a family of Boolean circuits $(C_n)_{n \in \omega}$ where there are n^2 inputs labelled $(i, j) : i, j \in [n]$, corresponding to the *potential edges*.
Each input takes value 0 or 1;

Graph properties in P are given by such families where:

- the size of C_n is bounded by a polynomial $p(n)$; and
- the family is uniform, so the function $n \mapsto C_n$ is in P (or DLogTime).

Invariant Circuits

C_n is *invariant* if, for every input graph, the output is unchanged under a permutation of the inputs induced by a permutation of $[n]$.

That is, given any input $G : [n]^2 \to \{0, 1\}$, and a permutation $\pi \in S_n$,

C_n accepts G if, and only if, C_n accepts the input πG given

$$(\pi G)(i, j) = G(\pi(i), \pi(j)).$$

Note: this is not the same as requiring that the result is invariant under *all* permutations of the input. That would only allow us to define functions of the *number* of 1s in the input. The functions we define include all *isomorphism-invariant* graph properties such as *connectivity*, *perfect matching*, *Hamiltonicity*, *3-colourability*.

Symmetric Circuits

Say C_n is *symmetric* if any permutation of $[n]$ applied to its inputs can be extended to an automorphism of C_n.

> *i.e., for each $\pi \in S_n$, there is an automorphism of C_n that takes input (i, j) to $(\pi i, \pi j)$.*

Any symmetric circuit is invariant, but *not* conversely.

> *Consider the natural circuit for deciding whether the number of edges in an n-vertex graph is even.*

Any invariant circuit can be converted to a symmetric circuit, but with potentially *exponential blow-up*.

Logic and Circuits

Any formula of φ *first-order logic* translates into a uniform family of circuits C_n

> *For each subformula $\psi(\overline{x})$ and each assignment $\overline{a}$ of values to the free variables, we have a gate.*
> *Existential quantifiers translate to big disjunctions, etc.*

The circuit C_n is:

- of *constant* depth (given by the depth of φ);
- of size at mose $c \cdot n^k$ where c is the number of subformulas of φ and k is the *maximum number of free variables* in any subformula of φ.
- *symmetric* by the action of $\pi \in S_n$ that takes $\psi[\overline{a}]$ to $\psi[\pi(\overline{a})]$.

Linear Programs for Hard problems

In the 1980s there was a great deal of excitement at the discovery that *linear programming* could be done in *polynomial time*.

This raised the possibility that linear programming techniques could be used to *efficiently* solve hard problems.

Many proposals were put forth for encoding *hard* problems (such as the *Travelling Salesman Problem*) (TSP) as linear programs.

(Yannakakis 1991) proved that *any* encoding of TSP as a linear program, satisfying natural *symmetry* conditions, must have *exponential size*.