Algorithms & Complexity: Lecture 1

Sam Barrett

February 12, 2021

1 Defining the Turing machine model

In order to talk about the time taken or the space used by an algorithm, we require a precise **model of computation**. There are many proposed models, we will focus on the Turing machine as defined by Arora and Barak in their book.

1.1 Arora-Barak Turing machines

1.1.1 Tapes

A Turing machine is defined as having k tapes where $k \geq 2$

- The first tape is the *input tape* and is **read-only**
- The $2..k-1$ tapes are *work* tapes and are **read-write**
- The k^{th} tape is the *output* tape.

Each tape has a leftmost cell and *potentially* infinitely many cells to the right of it. Potentially infinite meaning that at any given time, there are a finite number of cells but we can infinitely extend the tape over time.

Each tape has a **head** that sits on a cell and can move left and right.

1.1.2 Alphabet

A Turing machine also has an alphabet, denoted Γ. This is a **finite** set and it's elements are called *symbols*. There are 4 primary symbols: $\rhd, \Box, 0, 1$.

Here:

- $\{0,1\}^*$ is the set of bitstrings, the empty string is denoted with ε.
- $\rhd$ is the left-of-tape symbol and $\Box$ is the blank symbol

At any point in time, each cell of each tape contains a symbol. All but a finite number will be blank ($\Box$)

1.1.3 Initial configuration

The input tape has $\triangleright$ on the leftmost cell, then a bitstring (the **input**) and the rest of the tape is blank. The work tapes (including the output tape) have $\triangleright$ on the leftmost cell and the rest are blank. Each tape starts with it's head on it's the leftmost cell.

1.1.4 Computation step

In a single step of computation the machine:

- reads the character at each tape head
- writes a character at each work tape head
- may move each tape head to the left or to the right. **note: our tapes are not recursive, if a head on the leftmost cell moves left, it stays put**

1.1.5 Formal definition

A **Turing machine** is defined as a (6) tuple, $M = (k, \Gamma, Q, q_{\texttt{start}}, q_{\texttt{halt}}, \delta)$ consists of the following data:

- the number of tapes, k, $k \geq 2$
- the alphabet $\{0, 1, \triangleright, \Box\}, \Gamma$
- a finite set of Q states, including the start state $q_{\texttt{start}}$ and the halt state $q_{\texttt{halt}}$
- a transition function, $\delta : Q \times \Gamma^k \to Q \times \Gamma^{k-1} \times \{L, R, S\}^k$ Where:
  - the initial Q is the state at the start of transition
  - Γ^k is the set of symbols read
  - the final Q is the state at the end of transition
  - Γ^{k-1} is the set of symbols written
  - $\{L, R, S\}^k$ is the set of movement instructions where:
    - L means *move left*
    - R means *move right*
    - S means *stay*

  Note: we read k symbols but only write $k-1$ symbols as we do not write on the input tape, we also have k movement instructions as we are able to move on all k of the tapes.

1.1.6 Example transition

Say we have $k =$, and $\Gamma = \{\triangleright, \Box, 0, 1\}$ and $Q = \{4, 5, 6, 7, 8\}$ with $q_{\texttt{start}} = 4$ and $q_{\texttt{halt}} = 8$. We are currently in state 7 and the three tapes respectively say 1 (input), 1 (work) and $\Box$ (output).

Say that $\delta(7, \langle 1, 1\Box\rangle) = (5, \langle 0, \Box\rangle, \langle L, L, S\rangle)$ then we:

- transition to state 5
- overwrite the 1 on the work tape with 0
- overwrite the $\Box$ on the output tape with $\Box$ (no change)
- move left on the input tape (if possible)
- move left on the work tape (if possible)
- stay put on the output tape

We do not transition from the halt state (regardless of δ)

2 Computing with Turing machines

2.1 Computing a function

Given a function $f : \{0,1\}^* \to \{0,1\}^*$ a Turing machine $M = (k, \Gamma, Q, q_{\texttt{start}}, q_{\texttt{halt}}, \delta)$,

1. what does it mean to say that M **computes** f?

   It means that for every bitstring $x \in \{0,1\}^*$, if we start in state $q_{\texttt{start}}$ with the initial configuration showing x (meaning x appears on the input tape and and the work tapes are blank), when we run M, we eventually reach $q_{\texttt{halt}}$ with the output tape showing $\triangleright$ on the leftmost cell and then the bitstring $f(x)$ followed by all blanks.

Our initial configuration can be shown as:

Input tape:	▷	1	0	1	1	
Work tapes:	□	□	□	□	□	
Output tape: (also a work tape)	□	□	□	□	□	

Given that $f(x) = 0110111$, our required output tape will then be as follows:

▷	0	1	1	0	1	1	1	1	

If the machine, M does *this* for every bitstring x then we say it **computes** f. In the Arora-Barak definition, it does not matter what is on the work tape at the end of execution or the location of the work heads.

2.2 Computable functions

We say a function $f : \{0,1\}^* \to \{0,1\}^*$ or a function f from bitstrings to bitstrings is **computable** if there exists some Turing machine that computes it and **non-computable** if there isn't.

In the second case where there exists no Turing machine that computes a function, is there some other kind of machine that *does* compute it?

2.2.1 Church's thesis

We have only looked at one definition of Turing machines, there are many different variations that have been studied. 1 tape vs ∞ tapes, large alphabets, tapes infinite in both directions, 2D tapes, etc.

None of these variations affect our definition of computability. The same definition holds for all models that have been investigated, leading to Church's thesis which (informally) states:

Thesis 1 *"any algorithm that computes a function $\{0,1\}^* \to \{0,1\}^*$ can be converted into a Turing machine that computes the same function"*

2.3 Boolean functions and language

A language can be defined as any set of *words*, for example *all the words with an even occurrence of 1* is a language.

A **boolean function** is a function of the form: $f : \{0,1\}^* \to \{0,1\}$. Noting that the output is a single bit rather than a bitstring.

A important point about languages and boolean function is that they correspond. There is a one-to-one correspondence in fact between languages and boolean functions.

- For a given boolean function f the corresponding language is the set of bitstrings x s.t. $f(x) = 1$
- For a language L, the corresponding boolean function sends x to 1 if $x \in L$ and to 0 otherwise.

This allows us to treat boolean functions, languages and decision problems as essentially the same thing.

A decision problem is said to be **decidable** when the corresponding boolean function is **computable**. I.e. given a language L, for L to be decidable there must exist some Turing machine that will start with a bitstring x and will run continuously until it halts and upon halting there will be a 1 on the output tape if $x \in L$ or 0 if it is not in the language.

2.3.1 Example: palindromes

A **palindrome** is a bitstring that *reads* the same forwards as backwards.

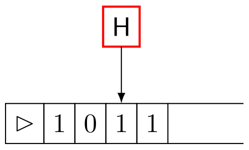
We can define our decision procedure for `PAL`, the set of all palindromes as:

1. Copy the input to the work tape
2. Move the input head to the start of the input
3. move the input head to the right while moving the output head to the left. If at any moment, the machine observes two different values, it writes 0 to the output tape and halts
4. Write 1 to the output tape and halt

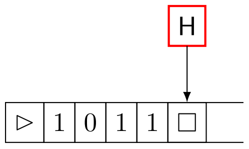
We can represent this as a Turing machine with 3 tapes and 5 states in the following example:

Step 1:

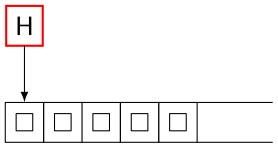
Input tape:



Work tape:

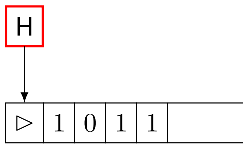


Output tape:

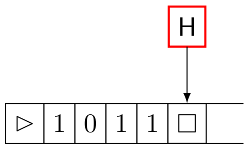


Step 2:

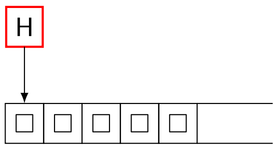
Input tape:



Work tape:

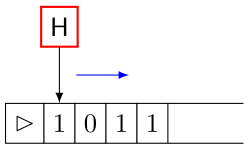


Output tape:

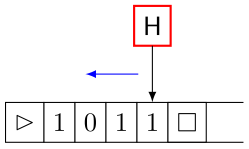


Step 3:

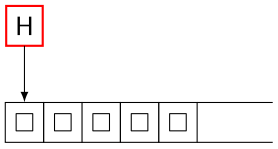
Input tape:



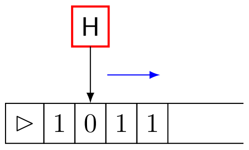
Work tape:



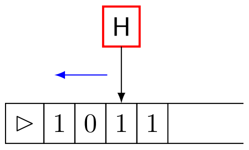
Output tape:



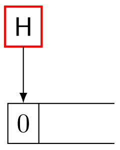
Input tape:



Work tape:



Output tape:



Note: the formal definition may require a $\triangleright$ at the beginning of the output tape, the procedure would be adjusted accordingly

2.4 Data representation

In many real-world problems our input data does not the innately take the form of a bitstring, when working with Turing machines, it must be encoded as a bitstring. Not all data can be encoded as bitstrings but many can. An example of data that cannot is any member of the set of Real numbers $\mathbb{R}$.

Knowing this, we can answer a common question: "*why don't* we consider Turing machines with more than 1 input? " the answer: We can simply encode a list of bitstrings as a single bitstring!

3 Code as data

We have just seen that many types of data can be encoded as bitstrings and processed by Turing machines. We will now focus on the case where we encode **Program code** as input to a Turing machine.

We can encode Java programs as bitstrings, we can even encode other Turing machines as bitstring input to a Turing machine as our Turing machine is essentially a 6-tuple as we mentioned earlier.

For any bitstring α we will denote the corresponding Turing machine M_α.

We will examine two consequences of this encoding:

3.1 Universal Java program

A **universal Java program**, this simply refers to a Java *interpreter* written entirely in Java. It is able to execute **any** Java program.

This is the same principal as a **universal Turing machine**, $\mathcal{U}$ being a Turing machine interpreter written in (on?) a Turing machine.

The universal Java program takes 2 parameters: α and x, encoded as a single bitstring and returns the same result as the machine M_α when applied to x

How can this be achieved?

3.1.1 Sketch of the universal Turing machine, $\mathcal{U}$

$\mathcal{U}$ can be implemented using 4 (work) tapes. It first *unpacks* the input $\langle \alpha, x \rangle$ into its two constituent parts and places them onto the first two work tapes.

The machine M_α may use an alphabet of 100 symbols and 700 work tapes, we can always simulate it using just the alphabet $\{\triangleright, \Box, 0, 1\}$ and two work tapes (inc. output tape).

We essentially simulate M_α by providing the machine defined on 1^{st} work tape (α) with our final 3 work tapes as it's input, work and output tapes respectively. An example will clarify this:

Input tape:

$\triangleright$	$\langle \alpha, x \rangle$

Work tape 1:

$\triangleright$	α

Work tape 2:

$\triangleright$	x

Work tape 3:

$\triangleright$	

Work tape 4:

$\triangleright$	

Where work tape 2,3,4 become input, work tape and output tape for the machine defined on work tape 1 (M_α)

3.2 Diagonalisation

This is our second consequence of treating code as data.

Diagonalisation is a proof method that can be used to show problems to be hard or even impossible.

Let us examine how to use it to prove the undecidability of the **halting problem**

The halting problem (HALT) can be defined as follows:

Problem 1 (The halting problem) *the set of pairs $\langle \alpha, x \rangle$ (encoded as a single bitstring) such that machine M_α, executed on input x, halts, i.e. it does not run forever.*

Turing's proof to this problem is as follows:

Proof 1 *Suppose that N is a machine that solves the halting problem.*

We can convert it into a machine N' that, given x, runs forever if $\langle x, x \rangle \in \mathtt{HALT}$ (i.e. the machine M_x executed on x halts), and halts otherwise.

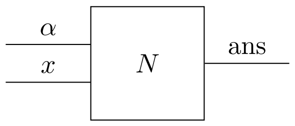
We know $N' = M_\alpha$ for some α, i.e. there exists some bitstring α that represents our new machine, as we know every machine can be represented as a bitstring.

Running N' on α halts if it runs forever and runs forever if it halts.

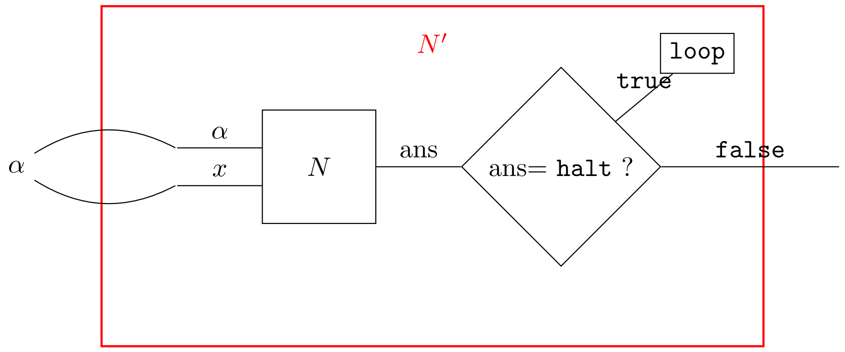
We have derived a contradiction.

This can be shown clearly diagrammatically:

Given a machine N that can solve the halting problem:



We can construct a machine N' that given a single input, x, runs forever if M_x executed on x halts and halts otherwise:



Where the outermost α is the bitstring of N'

Above you can clearly see that if N' is run on the bitstring representation of itself (α), it will halt only if it does not halt and it will hang if it halts. This cannot occur, therefore we cannot construct N and the problem is undecidable.

Algorithms & Complexity: Lecture 2, Time and space complexity

Sam Barrett

May 19, 2021

1 Upper and lower bounds

A simple set of examples for upper and lower bounds could be:

- **Upper bound:** I can clear my flat in a couple of days at most.
- **Lower bound:** It will take me at least a day to clear my flat.

1.1 Upper bound notation

Note: this notation is not only used for time complexity.

Say that we have two functions: $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$.

We say that $f(n)$ is $O(g(n))$ if f is **no bigger** than g up to a constant factor. Or more precisely, if there are numbers c and n_o such that, $\forall n, n \geq n_0$ we have $f(n) \leq c \cdot g(n)$.

Example

$$f(n) \leq 15n^3, \forall n \geq 1000$$

In this situation we can say that $f(n)$ is $O(n^3)$.

We have:

- $c = 15$
- $n_0 = 1000$
- $g(n) = n^3$

We say that $f(n)$ is $o(g(n))$ if f is not as big as g, even up to any constant factor. Or more precisely, if, for any $\epsilon > 0$, there is n_0 such that, $\forall n \geq n_0$ we have $f(n) \leq \epsilon \cdot g(n)$

We can therefore see, if $f(n)$ is $o(g(n))$ then $f(n)$ is always also $O(g(n))$ this can be proven if you take c to be 1.

1.1.1 Examples

Example 1
$5n^2 + 17n + 3$ is $O(n^2)$ and $o(n^3)$ and $O(n^3)$ but **not** $o(n^2)$.

- This is the case as we it is clearly no bigger than $O(n^2)$ (up to a constant factor) as it contains a quadratic term.
- It is *small* compared with n^3 (hence $o(n^3)$) as the highest factor again is n^2.
- It is also $O(n^3)$ as if it is no bigger than $O(n^2)$ it follows that it must also be no bigger than $O(n^3)$.
- We cannot, however, say that it is $o(n^2)$ as it cannot be smaller than n^2 due to it containing a quadratic term.

Example 2
$8n \log n$ is $O(n \log n)$ and $o(n^2)$
We can say this as:

- our term cannot be any bigger than $n \log n$ (up to a constant factor)
- It must be smaller than n^2, due to the nature of logarithms.

1.2 Lower bound notation

- We say that $f(n)$ is $\Omega(g(n))$ when $g(n)$ is $O(f(n))$

  Meaning, there c and n_0 such that, $\forall n \geq n_0$ we have $f(n) \geq c \cdot g(n)$
- We say that $f(n)$ is $\omega(g(n))$ when $g(n)$ is $o(f(n))$
- We say that $f(n)$ is $\Theta(g(n))$ when it is both $O(g(n))$ and $\Omega(g(n))$

  Informally we say this means: "$f(n)$ and $g(n)$ are the same, up to a constant factor"

2 Time complexity

2.1 Running time for a machine M

The running time of a machine M is the time taken from the input state, where x sits on the input tape and the other tapes are blank, to reach the halt state ($q_{\texttt{halt}}$).

For any number n, we define $\texttt{WT}_M(n)$ to be the **worst case** running time for an input of length n. For example,

Input	Running time
00	15
01	23
10	7
11	12

Here $\texttt{WT}_M(2) = 23$. If we were to say that $\texttt{WT}_M(n)$ is $O(n^2)$ we are saying that there are numbers n_0 and C such that, $\forall n \geq n_0$, the running time is $\leq Cn^2$.

2.2 DTIME classes

DTIME(n^2) is a **complexity class**, a complexity class can be thought of as a set of decision problems.

A decision problem, $f : \{0,1\}^* \to \{0,1\}$ is in **DTIME**(n^2) when there is some machine (of any sized alphabet or number of tapes) that decides it (f) and has worst case running time in $O(n^2)$.

2.2.1 Example: palindromes

We can again define our set `PAL` of all palindromic bitstrings with a boolean function $f : \{0,1\}^* \to \{0,1\}$.

Given we have a machine $A{-}B$ which utilises 3 tapes to decide *palindromicity* and has worst case running time $O(n)$. We can therefore say that `PAL` is in **DTIME**(n).

Can this be improved upon?

No! This can be trivially explained as any solution to palindromicity **must** at least read the input string of length n, therefore there must be **at least** n steps to the computation, leading to a best case running time in $\Omega(n)$.

2.3 Polynomial time

We can define the complexity (super) class of **polynomial time decision problems** as:

$$P \overset{\text{def}}{=} \bigcup_{k \geq 1} \mathbf{DTIME}(n^k)$$

From this definition, you can see that **any** decision problem in $\{\mathbf{DTIME}(n^k)\}_{k=0}^{\infty}$ is also in P

2.3.1 Robustness

Is this definition robust?

- Converting a large alphabet into our default alphabet ($\{\triangleright, \square, 0, 1\}$) only multiplies the running time by a constant factor

- Converting a n tape machine to a 3,2 or 1 tape machine **squares** the running time. This is more significant
- Converting a machine whose tapes are infinite in both directions to a machine whose tapes are infinite in only one direction multiplies the running time by a constant factor
- Converting a machine whose tapes are 2 dimensional to a machine whose tapes are one dimensional **squares the running time**. This is more significant.

In **all** of the cases listed above, the notion of polynomial time that you are left with is the **same**. The same class of decision problems are solvable in polynomial time.

Note: this is true for polynomial time (as defined above) but is not the case for linear or quadratic time

For example, `PAL` can be solved in $O(n)$ on a multitape Turing machine but is $\Theta(n^2)$ on a single tape machine.

2.3.2 Size of input

Another common concern is that our data may be represented as a bitstring in more than one way. However, in practical examples, the representations differ by a **constant** factor leading to polynomial time being the same.

2.4 Exponential time

We can define the complexity class of **exponential time decision problems as**:

$$\mathbf{EXP} \overset{\text{def}}{=} \bigcup_{k \geq 1} \mathbf{DTIME}(2^{n^k})$$

Therefore, any decision problem in $\mathbf{DTIME}(2^{5n^{17}})$ is in **EXP** and so on.

Also clearly $\mathbf{P} \subseteq \mathbf{EXP}$

3 Space complexity

Although we often regard time complexity as being the most important, there are many cases in which we need to worry about space complexity as well.

3.1 Space usage of a machine M

The **space usage** for an input x is the number of cells on the **work tapes** that are non-blank at some point during execution.

We ignore blank cells as at any point in computation there are infinitely many of these

For any number n we define $\texttt{WS}_M(n)$ to be the worst case space usage for an input of length n.

For example:

Input	Space usage
00	5
01	12
10	9
11	9

Here $\texttt{WS}_M(2) = 12$. Saying that $\texttt{WS}_M(n)$ is $O(n^2)$ means that there are numbers n_0 and C such that, $\forall n \geq n_0$, the space usage is $\leq Cn^2$.

Example execution

Input tape:

$\triangleright$	1	0	1	1	

A key point about the calculation of space usage is that we **do not** count the number of non-blank cells on the input tape, only on the work tapes.

Work tape 1:

$\triangleright$	1	0	1	1	1	1	

Work tape 2:

$\triangleright$	1	0	1	1	1	

The space usage above would be 11.

3.2 SPACE complexity class

SPACE(n^2) is a complexity class and a decision problem $f : \{0,1\}^* \to \{0,1\}$ is in this space when there is some machine (with any size of alphabet or number of tapes) that decides it (f) and has a worst case space usage in $O(n^2)$

3.3 L and PSPACE

We can now define logarithmic space (**L**) which defines the set of things that can be computed with a machine using a logarithmic number of cells

Note: this relies on the fact that we do not count the number of cells on the input tape. This is because if we were to count the input tape there would be at least n cells used and $n > \log n$, similarly to how we cannot have `PAL` solved in less than linear time.

$$\mathbf{L} \overset{\text{def}}{=} \mathbf{SPACE}(\log n)$$

We can also define polynomial space **PSPACE**

$$\mathbf{PSPACE} \overset{\text{def}}{=} \bigcup_{k \geq 1} \mathbf{SPACE}(n^k)$$

It is also clear that $\mathbf{L} \subseteq \mathbf{PSPACE}$

3.3.1 Robustness

Is this robust?

Yes, it is in fact more simple than with time.

- Converting a large alphabet into our default alphabet ($\{\triangleright, \Box, 0, 1\}$) only multiplies the space usage by a constant factor
- Converting a n tape machine to a 3,2 or 1 tape machine multiplies space usage by a constant factor.
- Converting a machine whose tapes are infinite in both directions to a machine whose tapes are infinite in only one direction multiplies the space usage by a constant factor
- Converting a machine whose tapes are 2 dimensional to a machine whose tapes are one dimensional multiplies space usage by a constant factor.

In all of these cases, logarithmic space does not depend on the model.

3.4 Space vs time

We can show that in all cases, space complexity is less or equal to time complexity.

We can prove by example that $\mathbf{P} \subseteq \mathbf{PSPACE}$:

- Let M be a machine with 5 work tapes that, for any input of length $n \geq 1000$, has a running time of $\leq 18n^3$ steps (a poly-time machine).
- For such an input, the space used is at most $5 + 5 \times 18n^3$

  This is true as 5 cells are non-blank initially and at most 5 more cells per step of execution ($5 \times \texttt{steps}$) $\implies 5 + (5 \times 18n^3)$

We can also show that, in all cases, time is less than or equal to exponentiated space. The following is a proof of $\mathbf{L} \subseteq \mathbf{P}$

- Let M be a machine with 5 work tapes, 74 states 13 symbols and it eventually halts, that, for any input of length $n \geq 1000$ has a space usage of $\leq 18 \log n$ cells

- For such an input, the number of **configurations** is at most

$$74 \times 13^{18 \log n} \times (18 \log n)^5 \times (n+2)$$

  Where a **configuration** tells us everything about the machine at a given point in execution

  - the state
  - what is written on each work tape
  - where the head is on each work tape
  - where the head is on the input tape

  and

  - 74 is the number of states in which the following apply
  - $13^{18 \log n}$ is the number of possible symbols in each of the maximum number of memory cells
  - $(18 \log n)^5$ represents all the possible head locations over the 5 tapes
  - On the input tape we have $n + 2$ cells in use. This is due to it containing the start symbol $\rhd$, n bits and a single blank cell.

  We can also see that this number of configurations in bounded by a polynomial as its constituent parts are bounded by polynomials (log etc.).

  The execution time cannot be greater than this because that would mean some configuration is repeated, causing an infinite loop. This is the case as if we reach the same configuration for a second time, there is nothing to prevent it from simply repeating everything it did subsequent to the last time it was in that configuration, thus looping. This cannot be the case as we have assumed our machine M to halt.

Therefore, if the space usage is logarithmic, the running time is polynomial.

The same argument can be made to show that if we have something in polynomial **space** it must be in **exponential** time. To construct this proof simply replace the $\log n$ in the above proof with a polynomial.

4 Nondeterministic time complexity

A simple definition of the complexity class **NP** is

Definition 1 *(***NP***)*
Problems for which checking a solution is easy

There are two methods for formally defining **NP**:

1. using certificates
2. using nondeterministic Turing machines.

4.1 Example: Sudoku

Let `SUD` be the set of solvable n-Sudoku puzzles, where n refers to the dimension of the grids.

Given a Sudoku puzzle x, a solution **certifies** that $x \in \texttt{SUD}$

The size of a solution is polynomial in $|x|$ (the length of x). The time taken to check a candidate solution is also polynomial in $|x|$

4.2 Defining NP using certificates

Definition 2 (**NP**) *A language L is said to be in* **NP** *if there is a polynomial-time machine for checking polynomially-sized certificates of L.*

Or, more precisely:

If there is a polynomial p, which gives the size of a candidate certificate) and a polynomial-time machine (for checking a candidate certificate) M such that, $\forall x \in \{0,1\}^*$ (where x is a bitstring representation of a Sudoku puzzle), the following are equivalent:

- $x \in L$
- *There is some bitstring u (a solution to the puzzle) of length $p|x|$ such that, $M\langle x, u\rangle = 1$.*

Here we say that u certifies the fact that $x \in L$

Note above, all text in parenthesis is not a part of the definition

4.3 Nondeterministic Turing machine

A **nondeterministic Turing machine** is similar to a Turing machine except for:

- it has 2 transition functions: δ_0 and δ_1
- besides having a halting state $q_{\texttt{halt}}$ it also has an accepting state $q_{\texttt{accept}}$

It starts in the initial state $q_{\texttt{start}}$, the same as a conventional Turing machine. At each step it *follows* either δ_0 or δ_1. Once the machine's state is $q_{\texttt{accept}}$ or $q_{\texttt{halt}}$, no further transition takes place.

When we have a nondeterministic Turing machine we need to be more careful when talking about the worst-case time complexity. For example:

Input	Running time
00	15, 7, 3, 9
01	6, **23**
10	7, 11, 5, 11, 8
11	12, 3, 4, 3, 12

Here $\texttt{WT}_M(2) = 23$ and the machine is polynomial-time if WT_M is $O(n^k)$ for some $k \geq 1$

4.4 Defining NP using nondeterministic Turing machines

A language L is in **NP** when there's a polynomial-time nondeterministic machine M such that, for $\forall x \in \{0,1\}^*$, the following are equivalent:

- $x \in L$, x is in the language L.
- When M is executed with input x, there's some sequence of choices that leads to $q_{\texttt{accept}}$

4.4.1 Example: `SUD`

In the case of n-Sudoku, given a Sudoku puzzle x, the nondeterministic Turing machine does the following:

1. begins by copying x to the work tape.
2. Then it non-deterministically fills each blank with a digit. **This stage takes time polynomial in $|x|$**
3. It goes on to check whether the completed grid is valid. **This step (and sub steps) also takes time polynomial in $|x|$**
   (a) If it is, it goes to $q_{\texttt{accept}}$
   (b) if it is not, it goes to state $q_{\texttt{halt}}$.

4.5 Equivalence of definitions

Our two definitions of **NP are** equivalent.

4.6 Is NP = P?

Clearly $\mathbf{P} \subseteq \mathbf{NP}$. This is the case trivially because any deterministic Turing machine is also a nondeterministic Turing machine by simply setting the accepting state to be the same as the halting state, and both transition functions to be the same.

It is an open problem as to whether $\mathbf{P} = \mathbf{NP}$. The currently supported hypothesis is **no**, $\mathbf{P} \neq \mathbf{NP}$. If the answer is *yes*, then there is a polynomial time algorithm for deciding whether an n-Sudoku puzzle is solvable.

It follows that there is a polynomial time algorithm that, given a solvable n-Sudoku puzzle, finds a solution. This is by testing all possible digits for each blank space.

5 Nondeterministic space complexity

Given a nondeterministic Turing machine, what does it mean to be in nondeterministic polynomial space complexity?

Let M be a nondeterministic Turing machine. It is in polynomial-space if $\texttt{WS}_M$ is $O(n^k)$ for some $k \geq 1$.

The worst case space complexity is polynomial, therefore, **NPSPACE** is the class of languages that can be decided by a polynomial-space nondeterministic Turing machine. This is the same principal as we saw in our second definition of **NP** in Section 4.4.

The same as $\mathbf{P} \subseteq \mathbf{NP}$, $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$.

5.1 Savitch's theorem

Using a special case of Savitch's theorem, we show $\mathbf{PSPACE} = \mathbf{NPSPACE}$

Suppose that M is a nondeterministic Turing machine, and for an input size n, the space usage is polynomial in n

Then the length of a configuration (as defined earlier) is also polynomial in n.

Let us say that, for $n \geq 1000$, a configuration has length at most $7n^{18}$.

Consider the configuration (directed) graph, which shows all $\leq 2^{7n^{18}}$ different configurations and the transitions between them. Each configuration has at most 2 next configurations.

With this graph, we want to, using a space-efficient algorithm, work out if there's a path from the start configuration to any accepting configuration. If such a path exists we know that the input is accepted.

How do we find this path space-efficiently?

5.1.1 Finding a path space-efficiently

To answer this question, we generalise.

Given nodes s and t and a number k, how much space do we use when deciding whether there is a path of length $\leq 2^k$ from s to t ? We will refer to this result as $D(k)$.

We will argue, by induction, that $D(k) \leq k \times 7n^{18}$.

To do this we check, for each configuration t, whether t is accepting and whether there is a path from the start configuration to t. This requires at most $7n^{18}$ bits to store t, and $D(7n^{18})$ bits to check for the path, i.e. at most $7n^{18} + D(7n^{18})^2$ bits in total. This is polynomial, as is required by *Savitch's theorem.*

All that is left is to finish the inductive proof of $D(k) \leq k \times 7n^{18}$.

Proof 1 *Base case: If $k = 0$, the problem is trivial. Just check if $s = t$.*

Inductive step: To find whether there is a path from s to t of length $\leq 2^{k+1}$, do the following:

1. *For each node z, test whether there is a path from s to z of length $\leq 2^k$ and a path from z to t of length $\leq 2^k$*
2. *By inductive hypothesis, this takes $\leq k \times 7n^{18}$ cells, plus a further*

$7n^{18}$ cells to store z. Totalling $\leq (k+1) \times 7n^{18}$.

Hence, $D(k+1) \leq (k+1) \times 7n^{18}$ therefore true $\forall k$

Algorithms & Complexity: Lecture 3, Completeness and Reductions

Sam Barrett

February 10, 2021

1 SAT and its variants

1.1 Propositional connectives

A basic reminder of Propositional logic and connectives:

- `T` (`True`) and `F` (`False`) are the propositional **constants**
- $a \wedge b$ is `True` if both a and b are `True` , otherwise `False` . **Conjunction**
- $a \vee b$ is `True` if either a or b is `True` , otherwise `False` . **Disjunction**
- $\neg a$ is `True` if a is `False` and vice versa. **Negation**
- $a \to b$ is `True` if either a is `False` or b is `True` , but `False` otherwise. **Implication**

Lemma 1 ***A propositional expression can be evaluated in linear time.*** *This is done using the shunting yard algorithm to translate into postfix notation and then evaluating using a stack.*

1.2 Conjunctive normal form

A formula is in CNF when it is a conjunction of disjunctions of variables and their negations.

For example,

$$(u_0 \vee \bar{u_1} \vee u_2) \wedge (u_1 \vee \bar{u_2} \vee u_3) \wedge \underbrace{(u_0 \vee \bar{u_2} \vee \bar{u_3})}_{\text{clause}}$$

Where in the above example $\bar{u}$ is the negation of u.

The disjunctions within the formula are called **clauses** and the variables are called **literals**

A clause can be written as $u \to (v \vee w \vee x)$ rather than $\bar{u} \vee v \vee w \vee x$

1.2.1 3CNF formulae

A CNF formula is **3CNF** when each clause has at most 3 literals

Note: any 3CNF clause can be written as an implication

1.2.2 Conversion to negation-free form

A formula is negation free when there are no occurrences of $\neg$ or $\rightarrow$. However these are still permitted in the form of negated variables which are still literals.

Every formula is equivalent to one in negation-free form. Simply push in each negation to a literal using de Morgan's laws:

$$\neg(\psi \vee \psi') = (\neg\psi) \wedge (\neg\psi')$$
$$\neg(\psi \wedge \psi') = (\neg\psi) \vee (\neg\psi')$$

Each push is $O(n)$-time, so the overall conversion is in $O(n^2)$ time.

1.2.3 Negation-free to CNF

A negation-free formula ϕ can be converted to a CNF formula ϕ' using extra free variables that are **equisatisfiable** with ϕ. This means that the new formula will be satisfiable iff ϕ is satisfiable.

This is done via induction on ϕ:

- The case where ϕ is a literal is clear, literals are already in CNF
- The case where ϕ is a conjunction is clear, it is already in CNF
- What if ϕ is a disjunction?

  For any variable c and clause ϕ, the formula $c \rightarrow \phi$ is equivalent to the clause $\bar{c} \vee \phi$.

  Therefore, any variable c and CNF formula ϕ, the formula $c \rightarrow \phi$ is equivalent to a CNF formula by the law:

  $$c \rightarrow (\psi \wedge \psi') = (c \rightarrow \psi) \wedge (c \rightarrow \psi')$$

  For any CNF formulas ϕ and ϕ', the formula $\phi \vee \phi'$ is equisatisfiable with:

  $$(c \vee c') \wedge (c \rightarrow \phi) \wedge (c' \rightarrow \phi')$$

  Which is equivalent to a CNF formula, obtained in $O(n)$ time. Thereby, we have a conversion to CNF in $O(n^2)$

1.2.4 CNF to 3CNF

In CNF, each clause is of the form:

$$a \to (b_0 \vee \cdots \vee b_{n-1} \vee c \vee c')$$

is equisatisfiable with:

$$\begin{aligned} &(a \to b_0 \vee d_0) \\ &\wedge(d_0 \to b_1 \vee d_1) \\ &\dots \\ &\wedge(d_{n-2} \to b_{n-1} \vee d_{n-1}) \\ &\wedge(d_{n-1} \to c \vee c') \end{aligned}$$

Where $d_0, \dots, d_{n-1}$ are *fresh* variables.
This gives an $O(n)$ time conversion to 3CNF

1.3 Satisfiability

Satisfiability is the process of answering questions of the form: *Over the variables* p, q, r *is the formula* $(\neg(q \to p) \wedge r) \vee (p \wedge q)$ satisfiable?

In this particular example, the answer is *yes*, in the case where $p = \texttt{F}$ and $q = r = \texttt{T}$

1.3.1 Formula-SAT

`Formula-SAT` is the set of all formulas that are satisfiable.

`Formula-SAT` is in **NP**, this is the case as given a formula ϕ, and an interpretation u,

- the length of u is no longer than that of ϕ
- it takes linear time to test whether it is a satisfying assignment by Lemma 1.

1.3.2 SAT

`SAT` is the set of CNF formulae that are satisfiable. Since `SAT` is a special case of `Formula-SAT` (which is in **NP**), it too is in **NP**

1.3.3 3SAT

`3SAT` is the set of 3CNF formulae that are satisfiable. Again, since it is a special case of `SAT`, it too is in **NP**.

2 Reductions

We often want to reduce a problem in mathematics/ Computer science to another, simpler or more understood problem. Intuitively, this can be thought of in the same way as reducing the problem of making *profiteroles* to the problem(s) of making cream-filled pastries and making chocolate sauce.

Let L and L' be languages.

A (many-to-one) **reduction** from L to L' is a function $f : \{0,1\}^* \to \{0,1\}^*$ such that for any bitstring, x we have $x \in L$ iff $f(x) \in L'$.

Or, more plainly, if we know how to decide membership of L', then the reduction enables us to decide membership of L.

2.1 Computable reductions

We write $L \leq_m L'$ when there is a reduction from L to L' that is **computable**. From this we can see that:

- If L' is decidable, then L is decidable
- If L is undecidable (*e.g. Halting problem*), then L' is undecidable .

This is a very useful property and allows us to easily prove the deciability or undecidability of problems without explicitly having to prove them. We will **not** look any closer in this module.

2.2 Polynomial time reductions

We write $L \leq_P L'$ when there is a reduction from L to L' that is **polynomial time**.

- If L' is in **P**, then L is also in **P**
- If L' is in **NP**, then L is also in **NP**

2.3 NP-Completeness

A language L is **NP**-hard if **every** language in **NP** has a polynomial-time reduction to it.

Therefore, if L is in **P** and **NP**-hard then **P** = **NP**!

If L is in **NP** and also **NP**-hard, we say that it is **NP**-complete. These are the *hardest* problems in **NP**.

2.3.1 Proving NP-completeness

To prove that a problem is **NP**-complete:

- One must show that it is in **NP**
- One must show that some other **NP**-hard problem reduces to it.

3 The Cook-Levin theorem

Theorem 1 *3SAT is* **NP***-complete*

We know that 3SAT is in **NP**. Therefore, to show that it is **NP**-complete, we must show it to also be in **NP**-hard.

For any language $L \in \mathbf{NP}$ we want to give a polytime reduction from L to 3SAT.

We will approach this in order from Formula-SAT $\to$ SAT $\to$ 3SAT

3.1 Reducing to Formula-SAT

Since L is in **NP** there must be a nondeterministic Turing machine which decides it.

Say that M is a NDTM for the language L, using an input tape, a work tape and an alphabet $\{\triangleright, \square, 0, 1\}$ with 50 states and a running time and space usage of at most n^3, where n is the size of the input.

From this, we must convert a bitstring x of length n into a propositional logic formula that is satisfiable iff $x \in L$.

The variables

- Let $a_{i,j,s}$ say that at time i, cell j of the work tape contains symbol s. Here $i, j < n^3$
- Let $b_{i,j}$ say that, at time i the input head is in position j. Here $i < n^3$ and $j < n$.
- Let $c_{i,j}$ say that, at time i, the work head is in position j. Here $i, j < n^3$
- Let $d_{i,q}$ say that, at time i the current work state is q. here $i < n^3$ and $q < 50$ (as per machine definition)

The constraints

- For any time i, each cell j contains only one symbol and there is only one current state.
- The configurations at time i and time $i+1$, and the input, are related by the transition function.

  This is stated locally, meaning if, at time i the state at time $i+1$ is determined only by adjacent states.
- At some time $i < n^3$, the current state is $q_{\texttt{accept}}$.

Putting these things together gives a formula of size $O(n^3)$, It is satisfiable iff the bitstring x is acceptable ($x \in L$).

3.2 Reduction to SAT

By converting a formula to an equisatisfiable CNF formula in $O(n^2)$ time (See Section 1.2.3), we show that Formula-SAT $\leq_P$ SAT.

3.3 Reduction to 3SAT

By converting a CNF formula to an equisatisfiable 3CNF formula in $O(n)$ time we show that SAT $\leq_P$ 3SAT

3.4 Proving NP-completeness

We previously outlined how to prove a problems is **NP**-complete in Section 2.3.1. We have shown that 3SAT reduces to **NP**-hard thus satisfying the second point.

4 Logspace reductions

We know that $\mathbf{L} \subseteq \mathbf{P}$, i.e every decision problem that can be solved in logspace can be solved in polynomial time.

4.1 Requirements

- We will write $L \leq_L L'$ when there is a logspace reduction from L to L'.
- We *want* the identity reduction to be logspace, i.e. $L \leq_L L$.
- We want a composite of logspace reductions to be logspace, so that:

$$L \leq_L L' \leq_L L'' \implies L \leq_L L''$$

- If you have two languages that are related, $L \leq_L L^{'}$ then $L' \in \mathbf{L}$ should imply $L \in \mathbf{L}$
- Every logspace reduction should be polytime, so that, $\leq_L$ implies $\leq_P$.

4.2 Logspace reduction: definition

We impose the following requirements on a function $f : \{0,1\}^* \to \{0,1\}^*$:

- f must be **polynomially bounded**, meaning, there must be a c such that, for every bitstring x, $|f(x)| \leq |x|^c$ holds true.
- We can test in logarithmic space whether a particular position in the output it within or outside of the length of $f(x)$. Formally:

  The set of pairs $\langle x, i \rangle$, s.t. $i \leq |f(x)|$ **must** be in $\mathbf{L}$
- We can test whether a particular position $\langle x, i \rangle$ gives a result of 1 or not, $f(x)_i = 1$ must be in $\mathbf{L}$. This is referred to as the **bitwise** problem for f. (This is the most important condition)

4.3 Composing logspace reductions

Suppose we have two logspace reductions f and g, then the composite function $x \mapsto g(f(x))$ is also logspace, we are going to show that this is the case:

To compute $g(f(x))_i$ (the i^{th} bit of the result) using three work tapes (A,B,C), we assume we can compute f and g using one work tape each. We assign f work tape C and g work tape A.

We cannot use tape B as an input tape for g as this would take too much space, resulting in a non-logspace computation. Instead, we use a *virtual* input tape. This means that the current input position j is stored on work tape B (using a logarithmic amount of space), and in each step we work out $f(x)_j$, using work tape C.

All of these components are logspace, meaning our composition function x is also logspace as if $L \leq_L L'$ then $L' \in \mathbf{L}$ implies $L \in \mathbf{L}$.

4.4 Logspace reduction are polytime

Let f be a logspace reduction. The bitwise problem is in **L** and therefore also in **P**.

So, for any x, the length of $f(x)$ is polynomially bounded and each bit can be computed in polynomial time, allowing us to compute $f(x)$ in polynomial time (polynomially many steps over polynomially bits).

4.5 Application: P-completeness

Just as polytime reductions give a reasonable notion of **NP**-completeness, so logspace reductions give a reasonable notion of **P**-completeness.

With **P**-completeness, we cannot simply look to the degree of the polynomial to determine how *hard* it is, as these are infinite and so **P** would be the same as **P**-complete. We instead need a different measure.

Definition 1 *(**P**-completeness) A problem is **P**-complete if it is in **P** and every problem in **P** logspace-reduces to it.*

Algorithms & Complexity: Lecture 4, Hierarchy theorems and a complexity zoo

Sam Barrett

February 10, 2021

1 Low-level conventions

1.1 Representation of Turing machines

- We will associate with every $\alpha \in \{0,1\}^*$ a Turing machine M_α s.t. for each Turing machine M, there are **infinitely many** α where $M = M_\alpha$.

  We will also fix a **bijection** between $\{00, 11\}^*$ (a fragment of all binary strings) and the set of all TMs (for every word inside this language, there is a corresponding unique TM), we will write M_β for the TM M that $\beta \in \{00, 11\}^*$ is mapped to. Here β is the canonical description of M or a *code of M* .

- We will extend our notion of M_β to $\alpha \in \{0,1\}^*$ (any binary string), we may write $\alpha = \beta\gamma$ with $\beta \in \{00, 11\}^*$ and with $\gamma \in \{0,1\}^*$ being either empty or beginning with 01 or 10. In this case we also set $M_\alpha := M_\beta$. Here α is a **description** of $M = M_\beta$.

  Unpacking this:

  If we have a bitstring α and want to find the machine that it represents, you write α in the form $\beta\gamma$ and extract the initial β section.

A very useful property of the above framework is that given any $\alpha = \beta\gamma$ we can **computably extract** low-level information about $M = M_\alpha$ such as its states, transition table, alphabet etc. Extracting this information can be done in time and space **only dependant on** β, its canonical description. We can completely ignore γ in this case, thinking of it as *padding*.

Note: we know when to stop reading β as we treat the *gadgets* "01" or "10" as blanks/ end of input markers.

1.2 Constructable functions

We shall identify $\mathbb{N}$ and $\{0,1\}^*$ by some *fixed bijective coding* . This will make more sense later in the lecture. Refer back.

1.2.1 Time-constructibility convention

All functions $t : \mathbb{N} \to \mathbb{N}$ we consider are **time-constructible** meaning:

- $t(n) \geq n$
- There is a TM M computing $1^n \mapsto t(n)$ in time $t(n)$

1.2.2 Space-constructibility convention

All functions $s : \mathbb{N} \to \mathbb{N}$ we consider are **space-constructible** meaning:

- $s(n) \geq \log n$
- There is a TM M computing $1^n \mapsto s(n)$ in space $O(s(n))$

2 Universality

We have previously seen the following:

2.1 Normal form of Turing machines

Theorem 1 *Suppose M computes $f : \{0,1\}^* \to \{0,1\}$ in time $t(n)$ and space $s(n)$. Where M can have an arbitrary alphabet and any number of tapes.*

There exists a 3-tape TM $\tilde{M}$ with alphabet $\{\triangleright, \Box, 0, 1\}$ computing f,

- *in time $O(t(n)^2)$*
- *in space $O(s(n))$*

These constraints depend on M and not its description $\tilde{\beta}$

Moreover, we can compute the canonical description $\tilde{\beta}$ of $\tilde{M}$ from the canonical description β of M or any other description,α, of M for that matter (in the form $\alpha = \beta\gamma$).

2.2 An efficient universal machine, $\mathcal{U}$

For a TM M and a bitstring $x \in \{0,1\}^*$, we shall write $M(x) \in \{0,1\}^* \cup \{\uparrow\}$ for the output of M on x, if it exists, otherwise $\uparrow$ if it diverges or does not terminate.

Theorem 2 *There exists a TM $\mathcal{U}$ s.t. $\forall x \in \{0,1\}^*$ and $\alpha \in \{0,1\}^*$, we have $\mathcal{U}(x, \alpha) = M_\alpha(x)$*

*Moreover, we can also talk about its complexity, if M_α halts on x in t steps and uses s space, then $\mathcal{U}$ halts on (x, α) within $c_M t^2$ steps and $d_M s$ space, where c_M and d_M are constants depending only on $M = M_\alpha$, **not** its description α. (It will precisely depend on the canonical description of*

M β)
*Our formulation of $\mathcal{U}$ will have **5 tapes**: 1 input and 4 work tapes.*

We will now examine how $\mathcal{U}$ operates over an input (x, α):

1. Computing the normal form

   Let $\alpha = \beta\gamma$ be as defined in Section 1.1 and recall the definition of $\tilde{\beta}$ and $\tilde{M}$ from Theorem 1

   - $\mathcal{U}$ first computes $\tilde{\beta}$ from $\alpha = \beta\gamma$ and prints it onto tape 2, it only reads up to end of β
   - The first step concludes by printing a description of the start state of M on tape 3

   This step takes time and space complexity depending on β, ignoring γ.

   Usage of tapes and space complexity

   - From this stage onward, only the initial x section of the input (x, α) will be used on tape 1.

     Where we can visualise our input tape as:

$\triangleright$	x_0	x_1	$\cdots$	x_k	,	α_1	α_2	$\cdots$	α_l	

     Where the comma can be encoded as the first occurrence of our 01 or 10 gadget and our delimiter, if we encode our x in the same way as we do for our canonical descriptions β.
   - Tape 2 will become **read-only** and is used as a *lookup* table for simulating the transitions of $\tilde{M}$. Therefore, this tape uses space $|\tilde{\beta}|$
   - Tape 3 will always store a *current state*, using only as much space as the description of a state of $\tilde{M}$ (without loss in generality our space usage is $< |\tilde{\beta}|$)
   - Tapes 4 & 5 will be used as the two work tapes of $\tilde{M}$. Therefore, these tapes use only as much space as M does on its work tapes.

2. The simulation & time complexity

   Each step of $\tilde{M}$ is simulated as follows:

   - $\mathcal{U}$ inspects tape 3 to find the current state q and reads the symbols b_1, b_4, b_5 at the head-positions of tapes 1,4 and 5. This process takes no (0) time.
   - $\mathcal{U}$ scans the transition table of $\tilde{M}$ (by inspecting tape 2) to find the transition corresponding to (q, b_1, b_4, b_5). The time of this depends only on $\tilde{\beta}$
   - $\mathcal{U}$ overwrites tape 3 with the description of the new state. The time this takes depends only on $\tilde{\beta}$.

- $\mathcal{U}$ writes the appropriate symbols at the head-positions of tapes 4 and 5 before moving the heads of tapes 1,4 and 5 in the appropriate directions. This takes a single (1) time step.

$\mathcal{U}$ will halt whenever $\tilde{M}$ does, outputting the content of tape 5.

3 Diagonalisation

Theorem 3 *(Time hierarchy theorem) There is a language $L \in \mathbf{DTIME}(t(n)^4)$ s.t. $L \notin \mathbf{DTIME}$*

*i.e. $\mathbf{DTIME}(t(n)) \subsetneq \mathbf{DTIME}(t(n)^4)$ (one is **strictly contained within the other**)*

Where t is arbitrary but time constructable as defined in Section 1.

3.1 Time-sensitive diagonalisation

To perform diagonalisation in such a way as to concern ourselves with time complexity, we define a Turing machine D that does the following:

Definition 1 *(Turing machine D)*

- *on input x (x is a binary string $x \in \{0,1\}^*$), run $\mathcal{U}$ on (x, x) for $t(|x|)^3$ steps, we use $t(|x|)^3$ as it is somewhere between the time overhead for $\mathcal{U}$ ($t(n)^2$) and our states time hierarchy constraint of $t(n)^4$.*
- *if it halts in this time and rejects (where rejecting means it outputs 0) then accept (output 1)*
- *otherwise, reject (output 0).*

We can now define a language $L \subseteq \{0,1\}^*$ as the language that is decided by D. L is just the set of descriptions of Turing machines for which when $\mathcal{U}$ runs it on itself it rejects the appropriate amount of time.

By our construction of L we can observe that $L \in \mathbf{DTIME}(t(n)^4)$ as our machine D can only run for $t(|x|)^3$ steps.

We claim therefore, that L is the **explicit** language that separates $\mathbf{DTIME}(t(n^4))$ from $\mathbf{DTIME}(t(n))$, meaning that $L \notin \mathbf{DTIME}(t(n))$. We will prove this by contradiction.

3.1.1 Proof

Assume that $L \in \mathbf{DTIME}(t(n))$, and suppose M decides L taking $ct(n)$ steps on inputs of length n.

We now use $\mathcal{U}$ to simulate M and say it does this within $c_M ct(n)^2$ steps on inputs of length n. Where c_M depends only on M and not its description.

Let us fix $n_0 \in \mathbb{N}$ s.t. if $n \geq n_0$ then $t(n)^3 > c_M ct(n)^2$.

This is a key point, it means that there is a point in $\mathbb{N}$, n_0 where whenever n is greater than n_0 we can say that $t(n)^3$ (the number of steps $\mathcal{U}$ runs for) is greater than the number of steps our machine M is purported to take.

This is where *"foo is dependant only on bar not its description"* becomes important, the trick to *breaking* this inequality and deriving a contradiction is to let α be a description of M (which has infinitely many descriptions) with $|\alpha| \geq n_0$

We will now examine what happens when we run D with the input α that I described above.

- D runs $\mathcal{U}$ on (α, α) for $t(|\alpha|)^3$ steps as per our definition of D in Definition 1
- From our fixing above, along with our definition of α, we can say that $t(|\alpha|)^3 \geq c_M ct(|\alpha|)^2$, giving us in turn:

  $$\mathcal{U}(\alpha, \alpha) = M_\alpha(\alpha) = M(\alpha)$$

  As M must halt on α **within** $ct(|\alpha|)$ steps, by our assumption of M.
- As per our definition of D, as $M(\alpha)$ halts, D must return $1 - M(\alpha)$ as it always returns the inverse.

  However, by doing so, as M was meant to decide the language described by D we have derived a contradiction by constructing a situation in which M and D disagree on an input α. Therefore, M could **not** have decided the language described by D.

By a similar proof, tracking space instead of time we can show that:

Theorem 4 *(Space hierarchy theorem) There is a language* $L \in \textbf{SPACE}(t(n)^2)$ *s.t.* $L \notin \textbf{SPACE}(t(n))$

We will not go on to prove this.

4 Consequences and the complexity *zoo*

4.1 Separations of complexity classes

Theorem 5 *We have the following:*
$\textbf{P} \subsetneq \textbf{EXP}$
$\textbf{L} \subsetneq \textbf{PSPACE}$

4.1.1 Proof

For both of the above statements, the $\subseteq$ case is *obvious*. However, for non-equality we have,

$$\mathbf{P} \subseteq \overbrace{\mathbf{DTIME}(2^n) \subsetneq \mathbf{DTIME}(2^{4n})}^{\text{Time hierarchy theorem}} \subseteq \mathbf{EXP}$$

This can be seen by the time-hierarchy theorem explored earlier.
We also have for space:

$$\mathbf{L} \subseteq \mathbf{SPACE}(n) \subsetneq \mathbf{SPACE}(n^2) \subseteq \mathbf{PSPACE}$$

which can equally be seen via the space-hierarchy theorem.
We now have a *zoo* of complexity classes:

$$\mathbf{L} \underbrace{\subseteq}_{\text{Lecture 2}} \mathbf{P} \underbrace{\subseteq}_{\text{obv}} \mathbf{NP} \subseteq \overbrace{\mathbf{PSPACE} \underbrace{\subseteq}_{\text{obv}} \mathbf{NPSPACE}}^{\leftarrow\text{Savitch's theorem}} \subseteq \mathbf{EXP}$$

This is all we know! Every other such problem remains open.

Algorithms & Complexity: Lecture 5, P vs NP & Algorithms

Sam Barrett

February 15, 2021

1 History of Computing

In the 1950s to 1960s one of computer science researcher's focuses was on general ways to solve problems.

One such problem was `SAT`, is a given logical formula ϕ satisfiable?

Many of the problems known at this time had polynomial running time. We know that polynomial time is ultimately always lower than exponential, for instance $1.000001^n > n^{10000000000}$ if n is large enough. We can prove this by taking logs of both sides.

During this time polynomial time became the accepted standard of efficiency. It has many *nice* properties including being **closed** under addition, multiplication and composition. I.e. if both p and q are polytime functions: $p(x) + q(x)$ is polytime, $p(x) \times q(x)$ is polytime and $p(q(x))$ is polytime. We define **P** as the class of problems solvable in polynomial time (wrt. the size of the input).

Later, in the 1970s research moved to focus on the problems that no efficient algorithm was known to be able to solve, the set of problems that could not be included in **P**.

Our previous example `SAT` cannot be brute forced in polynomial time. If given N variables in M clauses, we must make 2^N truth assignments, checking each of the M clauses in $O(N)$ time resulting in overall running time in $O(2^N \cdot N \cdot M)$, clearly this is in **EXP**.

Research started to focus on trying to prove that there is no solution in **P** for `SAT`.

In so doing a new class was defined, **NP**. The class of problems where we can verify a potential solution, or certificate, in polynomial time.

How much harder is solving compared with verifying?

Given that we define **P** as the class of problems with solutions in polytime, and **NP** is the class of problems for which we can verify a potential solution in polytime, clearly $\mathbf{P} \subseteq \mathbf{NP}$ as solving can be seen as a very difficult way of verifying.

But what about the opposite direction? Say we can verify potential solutions in n^{12} time, how long would it take us to **solve** the problem? If we can solve this in some polynomial amount of time n^k then we have shown that $\mathbf{P} = \mathbf{NP}$!

2 From SAT to graphs

Definition 1 *(Independent Set) Given an undirected graph $G = (V, E)$ on n vertices, find a set X of maximal size such that no pair of vertices in X form an edge*

We can reduce the independent set problem to `SAT`:

- Take an instance I of 3-`SAT` with N variables and M clauses
- Build a graph G on $3M$ vertices as follows:
  - Introduce triangle for each clause
  - Add conflicts to ensure every variable is not both `True` and `False`
- Claim: I is satisfiable iff G has an independent set of size M

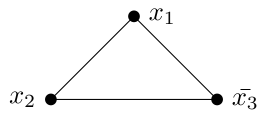
This reduction is in polytime meaning that the Independent set problem is also in **NP**

2.1 Example

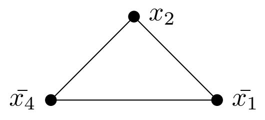
Suppose we have an $I = \underbrace{(x_1 \vee x_2 \vee \bar{x_3})}_{c_1} \wedge \underbrace{(x_2 \vee \bar{x_4} \vee \bar{x_1})}_{c_2} \wedge \underbrace{(x_3 \vee x_1 \vee \bar{x_2})}_{c_3}$

We can convert each clause into a separate triangle:

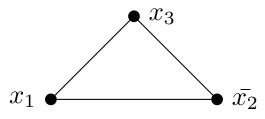
c_1:



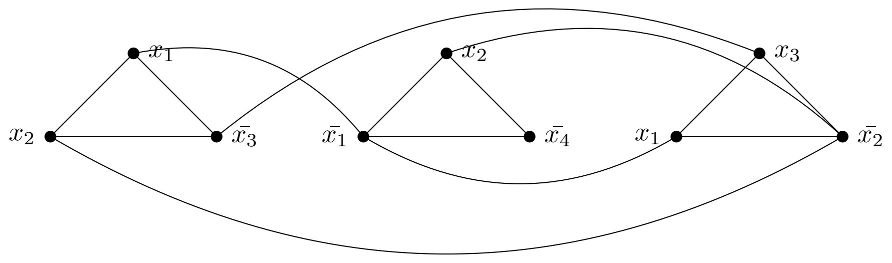
c_2:



c_3



Now we must add our conflicts. We do this by connecting conflicting variables.



3 Algorithmic paradigms to cope with NP-hardness

Assuming $\mathbf{P} \neq \mathbf{NP}$, a problem X being **NP**-hard implies that we cannot have an algorithm `ALG` for it which satisfies both of the following properties:

- `ALG` is always correct
- `ALG` runs in polytime

This has lead to the development of new **algorithmic paradigms** including:

1. Exact Exponential algorithms

   Here we are focused on producing correct results, no matter the time complexity cost.

2. Polytime approximation algorithms

   Here we want to focus on having a polynomial running time, to do this we relax our first requirement and allow our algorithm to return only approximately correct results in some cases.

3. Parameterised algorithms

   Before trying to solve a problem optimally we can fix a parameter k and using this we can now reevaluate running time with respect to both the length of the input as we always have but also our fixed parameter k.

4. Polytime randomisation algorithms

   Here our algorithm must still run in polytime, and instead of always being correct we say that it must be correct with a given probability.

3.1 Exact exponential algorithms

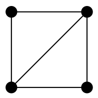
This approach essentially tries to answer the question: *"Can we do better than brute force, even if it still uses exponential time?"*

3.1.1 Vertex Cover

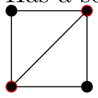
Definition 2 *Given an undirected graph $G = (V, E)$ on n vertices, find a set X of minimum size such that each edge of G has at least one endpoint in X*

The brute force approach to this problem runs in $2^n \cdot n^{O(1)}$ time. Can we design an $1.99^n \cdot n^{O(1)}$ time algorithm?

For instance the graph:



Has a solution of:



Where $X = 2$.

The brute force approach to finding this would be to enumerate all subsets of vertex set of G, V in increasing sizes and for each subset, checks in polynomial time whether it is a vertex cover. This check is performed simply by checking whether every element in E is connected to at least one of the vertices in the subset.

We can make observations which can allow us to improve upon this approach. One such observation is that there is no point in adding a vertex of degree 1 to our vertex cover, it can never connect to more points than its unique neighbour (the vertex on the other end of the edge), so we add the unique neighbour instead.

So we say that we only add vertices of degree ≥ 2 to the vertex cover. This, for every potential vertex, introduces a binary choice as to whether we add it to the vertex cover. Using this we define $T(n)$ as the time needed to solve the vertex cover on graphs with n vertices. We can therefore say that $T(n) \leq T(n-1) + T(n-3)$.

We derive this inequality by saying:

- $T(n-1)$ is *spawned* as a sub problem when we *take* v into the vertex cover, we reduce the size of V by 1
- Alternatively, if we do **not** add v to the vertex cover, we **must** add all ≥ 2 neighbours to the VC, so the number of vertices to consider in the subproblem has decreased by **at least** 3.

We can solve this inequality by setting $T(n)$ equal to x^n which gives us $x^0 - x^2 - 1 = 0$, solves to $1.47^n \cdot n^{O(1)}$.

3.2 Polynomial time approximation algorithms

Again looking at our Vertex Cover problem, can we find a vertex cover with has size at most 10 times that of the minimum vertex cover ? Can we do it in $n^{O(1)}$ time?

Do do this we must first find a maximal set M of pairwise disjoint edges. This **can** in fact be found in polynomial time!

Using this we can output a solution R which has both endpoints of each edge from M. By definition R is a vertex cover, albeit not a minimal vertex cover. We can see that if $\texttt{OPT}$ is a vertex cover of minimum size, then $|R| \leq 2 \cdot |\texttt{OPT}|$.

We have designed a 2-approximation for the Vertex Cover problem in $n^{O(1)}$ time, nothing better is known.

3.3 Parameterised algorithms

To look at this approach we have to tweak the definition of our Vertex Cover problem:

Definition 3 *(Parameterised Vertex Cover) Given an undirected graph $G = (V, E)$ on n vertices and an integer k, goes G have a vertex cover of size $\leq k$?*

This can be thought of as the question: *"Given a parameter k, how fast can we check whether there is a vertex cover of size k?* . Here we are not concerned with the size of the minimum vertex cover, rather we are concerned with being able to say whether the minimal vertex cover lies above or below k

An algorithm has been found which runs in $2^k \cdot n^{O(1)}$ time.

3.4 Polynomial time randomisation algoriths

Definition 4 *(Max Cut) Given an undirected graph $G = (V, E)$ on n vertices, find a set X which maximises the number of edges which have one endpoint in X and the other endpoint in $V \backslash X$*

We can define a 0.5-approximation algorithm. Since the goal with this problem is to maximise, the approximation ratio is usually written as < 1. This algorithm works by flipping a coin to decide which side of the partition each vertex goes to. We expect half of the edges to be in the *cut.*

This algorithm runs in $n^{O(1)}$ time.

This approach can be de-randomised by starting with any arbitrary partition, and for each vertex if it has strictly more neighbours in the same side then move it to the other side. This re-shuffling must be finite as eventually exactly half the edges are in each cut.

The best known result for this problem is a 0.86-approximation.

Algorithms & Complexity: Lecture 6, The Stable Matching Problem

Sam Barrett

February 16, 2021

1 The stable matching problem

Allocation is a fundamental task to life. We want any set of allocations we make to be *stable*. An allocation is stable if there are no unstable pairs, where an unstable pair is a pair that do not *want* to be together, and would prefer a different matching. To achieve this we must introduce a notion of *preferences* .

Allocations can be between 2 individuals from the same set or 2 individuals from different sets. For example Employees $\mapsto$ Teams and Doctors $\mapsto$ Hospitals.

1.1 Matchings within one group

Given an example with 4 people A, B, C and D with the following preferences:

- Preferences for A are $B > C > D$
- Preferences for B are $C > A > D$
- Preferences for C are $A > B > D$
- Preferences for D are $A > B > C$

There are 3 possible matchings:

1. (A, B) and (C, D)
   (B, C) is an **unstable** pair
   This is the case as B prefers C over A and C prefers B over D.
2. (A, C) and (B, D)
   (A, B) is an **unstable** pair
3. (A, D) and (B, C)
   (A, C) is an **unstable** pair

In the above example you can clearly see that no stable matching exists.

1.2 Matchings between two groups

Now we consider the case when we try to allocate between two disjoint sets. Does there always exist a stable matching or, like in the previous case, do there sometimes exist settings where stable matchings do not exist?

Given an example of allocations between hospitals and students, each of size n. Each hospital has a ranking of the n students and each student has a ranking of the n hospitals. We **assume** the list of preferences are strict and complete.

Not all matchings are stable.

- Consider two hospitals h_1, h_2 and two students s_1, s_2.
- Both hospitals prefer s_1 over s_2
- Both students prefer h_1 over h_2
- Therefore, the matching (h_1, s_2) and (h_2, s_1) is not stable as h_1 and s_1 form an unstable pair.

1.3 Definition

Definition 1 *(The* `STABLE MATCHING` *problem) The* `STABLE MATCHING` *problem asks to find a stable matching, if one exists.*

The `STABLE MATCHING` problem is in **NP**. This is the case as for n elements, there are $n^2 - n$ pairs in a candidate certificate, each of which is possibly unstable and must be checked individually. If no unstable pair is found then the matching is stable.

The brute force algorithm for `STABLE MATCHING` tries all $n!$ possible matchings, this is very slow as $n! \approx 2^{n \log n}$.

2 Gale-Shapely Algorithm

A better algorithm was proposed by Gale and Shapely in 1962 for complete lists which always finds a stable matching where all elements are allocated. Their algorithm runs in $O(n^2)$ time and puts `STABLE MATCHING` into **P**

The pseudocode for this algorithm is as follows:

Algorithm 1: The Gale-Shapley algorithm (1962)

```
Initially all hospitals and students are free
while There is a hospital which is free and hasn't made an offer to every student do
    Choose such a hospital $h$
    Let $s$ be highest ranked student to which $h$ hasn't made an offer yet
    if $s$ is free then
        $(s, h)$ are matched
    else
        $s$ is currently matched to some hospital $h'$
        if $s$ prefers $h'$ to $h$ then
            $h$ remains free
        else
            $s$ prefers $h$ to their current match $h'$
            $(s, h)$ get matched and $h'$ becomes free
        end
    end
end
```

The running time of this algorithm is clearly $O(n^2)$ as each while loop makes 1 new offer and there can only be n^2 offers made before termination.

2.1 Correctness

From the pseudocode we can see that after receiving their first offer, students always have a better offer or as good an offer *in hand* . We can also see that if a hospital is *free*, then there is a student to whom they have not yet made an offer. Therefore, upon termination, all hospitals and students are matched. But is the matching stable?

Theorem 1 *(Gale-Shapely returns a stable matching)*

- *Let us suppose that it does not, and there therefore exists an unstable pair h and s'*
- *Let (h, s) and (h', s') be allocations in the result. Then we can say that for the unstable pair h, s' to exist h must prefer s' over s and s' must prefer h over h'*
- *The last offer made by h was to s*
- *Did h make an offer to s' before making an offer to s?*
  - *If **NO** then h prefers s to s' **CONTRADICTION***
  - *If **YES** then h was rejected by s' in favour of some other hospital. By our previous point, a student's offers keep getting better*

(or stay the same) and since $h' \neq h$ it follows that s' prefers its final offer h' to h

There is a surprising property of the Gale-Shapely algorithm: it always returns the **SAME** stable matching.

To understand why we require the following definitions:

For each hospital h

- Let $\texttt{Valid}(h) = \{\forall s : S, \text{for which there is a stable matching which matches } h \text{ to } s\}$
- $\texttt{Best}(h)$ is the highest ranked (in preference of h) student from $\texttt{Valid}(h)$

The theorem can be written as "Gale-Shapely always returns the matching with which matches h to $\texttt{Best}(h)$ for each hospital h"

Similarly, we can show that for each student s, s gets their **worst** possible choice.

- Let $\texttt{Valid}(s) = \{\forall h : H, \text{for which there is a stable matching which matches } s \text{ to } h\}$
- $\texttt{Worst}(s)$ is the lowest ranked (in preference of s) hospital from $\texttt{Valid}(s)$

The Gale-Shapely algorithms always returns the matching which matches s to $\texttt{Worst}(s)$ for each student s

You can however, reverse this algorithm in the sense that you can make it so that the students make the offers to the hospitals, this procedure maintains the property that it returns the **same** matching each time but instead of the students getting their worst choices and hospitals getting their best, hospitals get their worst choices and students get their best!

3 Extensions

What about the case in which the two groups are of different size?

Project allocation in the school of Computer Science

- Say we have $5n$ students and n members of staff
- Each faculty member has a preference list over $5n$ students
- Each student has a preference list over n staff
- We want to have a stable allocation of 5 students per faculty member.

Can we extend Gale shapely? **Yes**, this is what is used in MSci project allocation.

The stability of a matching is just the start of desirable properties in a matching. We can add many more conditions, such as *no one gets allocated their n^{th} choice* etc.

Algorithms & Complexity: Lecture 7, Greedy Algorithms I

Sam Barrett

February 24, 2021

1 What are greedy algorithms?

There is no formal definition for greedy algorithms, we can actually design multiple different greedy algorithms to solve the same problem. They are categorised as algorithms that make local decisions to improve a solution, working step-by-step with no concern for what they have done previously or might do later. Often this short-sighted approach does not help in finding an optimal solution, however, some can still approximate an optimal solution.

There are advantages and disadvantages of greedy algorithms.

Advantages:

- They are intuitive
- This leads to them being easy to explain and implement
- Most heuristics are based on *greedy* choices
- They can be shown to sometimes be approximately correct

Disadvantages:

- There are different notions of greedy, no formal definition
- Local correct steps do not guarantee a globally correct approach
- Often do not result in optimal solutions

Our 2-approximation for the vertex cover problem discussed in the previous lecture is an example of a greedy algorithm, it ran in polynomial time.

2 Dijkstra's Algorithm

This is an algorithm that finds the shortest paths in a directed graph from a fixed vertex s.

The running time of this algorithm is $O(mn)$ where n, m are the number of vertices and edges respectively. This is the case as the while loop runs at most n times and each iteration takes m steps.

Algorithm 1: Dijkstra's Algorithm

```
1 Let $S$ be the set of explored vertices
2 For each $u \in S$ we store the distance of the shortest $s \to u$ path in
    $\texttt{dist}(u)$
3 Initialise $S = \{s\}$ and $\texttt{dist}(s) = 0$
4 while $S \neq V$ do
5     Select a vertex $v \notin S$ which minimises
        $\texttt{temp-dist}(v) = \min_{(u,v) \in E, u \in S} (\texttt{dist}(u) + \texttt{length}(u, v))$
6     Add $v$ to $S$ and set $\texttt{dist}(v) = \texttt{temp-dist}(v)$
7 end
```

2.1 Correctness of Dijkstra's algorithm

Theorem 1 *Consider the set S at any point in the running of the algorithm. For each $u \in S$ the quantity $\texttt{dist}(u)$ stores the value of the shortest $s \to u$ path.*

We can prove this by induction:

- Base case: $|S| = 1$ and $\texttt{dist}(s) = 0$
- Inductive hypothesis: Suppose that the theorem holds for $|S| = k$
- Inductive step: Suppose we now grow S by one more vertex by adding some vertex v.

  By line 5 we know: $\texttt{temp-dist}(v) = \texttt{dist}(u) + \texttt{length}(u, v)$ for some u already in S

  Suppose that there is a path P $s \to v$ that is shorter than $\texttt{dist}(v)$

  Let P leave S via an edge (x, y) for some $x \in S, y \notin S$

  Contradiction

3 Prim's algorithm

Prim's algorithm is an algorithm for finding the minimum spanning tree of a given graph G.

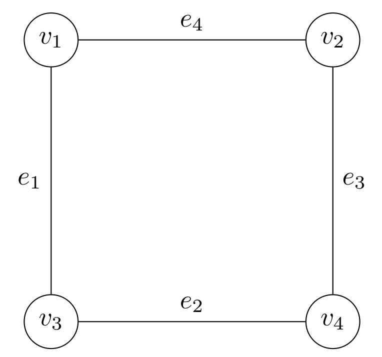
3.1 Minimum spanning trees

Let G be a undirected, connected graph $G = (V, E)$ with n vertices.

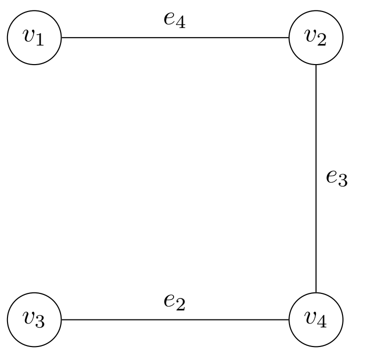
A subgraph $T = (V', E')$ of G is said to be a **spanning tree** if:

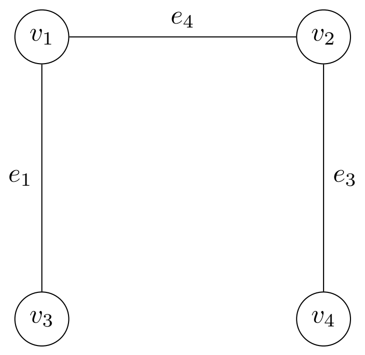
- spanning $\to V' = V$
- tree $\to |E'| = n - 1$

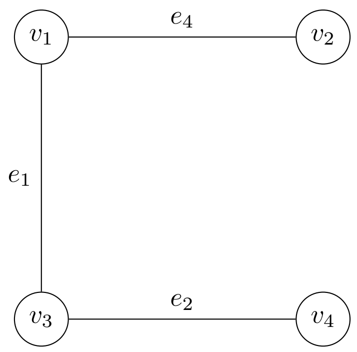
For instance, the graph:



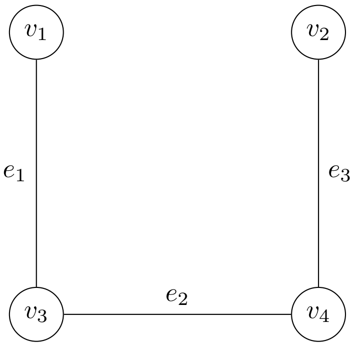
Has 4 different spanning trees:

1. 

2. 



3.



4.

Problem 1 *(Minimum Spanning Tree (MST) problem)*
Given an undirected, connected graph $G = (V, E)$ with edge costs given by $\texttt{cost} : E \to \mathbb{R}^+$, find a spanning tree $T = (V, E')$ such that $\sum_{e \in E'} \texttt{cost}(e)$ is minimised

Algorithm 2: Finding MST - Prim's algorithm

```
1 Let $S$ be the set of explored vertices
2 Initialise $S = \{s\}$ where $s$ is any vertex
3 Initialise $E' = \emptyset$
4 while $S \neq V$ do
5     Select a vertex $v \notin S$ which minimises $\min_{e=u-v, u \in S} \texttt{cost}(e)$
6     Add $v$ to $S$ and $e$ to $E'$
7 end
```

Running time of this algorithm is $O(mn)$ where n, m are the number of vertices and edges respectively. The while loop runs at most n times with each loop requiring m time.

3.2 Correctness of Prim's algorithm

We first make an assumption that all edge costs are **distinct**.

Theorem 2 *For any $S \subset V$, let e be the edge of minimum cost having one end point in S and one end point in $V \backslash S$. Then every MST contains the edge e*

First suppose that there is a MST T which does not contain this edge e whose endpoints are $v \in S$ and $w \notin S$. We will now find an edge e' in T s.t. $\texttt{cost}(e') > \texttt{cost}(e)$. Therefore, replacing e' with e gives a spanning tree of lower cost, thus deriving a contradiction.

4 Kruskal's algorithm

Algorithm 3: Kruskal's algorithm

```
1  Order the edges of $E$ as $e_1, e_2, \dots, e_m$ in order of cost (increasing)
2  Initialise $E' = \emptyset$ and $i = 1$
3  while $i \leq m$ do
4      if adding $e_i$ to $E'$ does not create a cycle then
5          Add $e_i$ to $E'$
6      else
7          Do not add $e_i$ to $E'$
8      end
9      $i++$
10 end
```

This algorithms also runs in $O(mn)$ time where n, m are the number of vertices and edges respectively.

The notable difference with this algorithm is that we do **not** consider all edges

4.1 Correctness of Kruskal's algorithm

We again make the assumption that all edge-costs are distinct.

Theorem 3 *(Same as for Prim's) For any $S \subset V$, let e be the edge of minimum cost having one end point in S and one end point in $V \backslash S$. Then every MST contains the edge e*

We consider the algorithm at some arbitrary step.

Suppose Kruskal's algorithm adds the edge $v - w$ at this step.

Let S be the set of all vertices to which v had a path to before this step. We can see that $w \notin S$ as otherwise we would have a cycle, breaking the algorithm at line 4.

By the definition of S no edges from S to $V \backslash S$ have been added before this stage.

Since $v \in S$ and $w \notin S$ and as Kruskal's algorithm adds edges in increasing order of cost, it follows that $v - w$ is the *cheapest* edge with one endpoint in S and the other in $V \backslash S$

5 Reverse-delete algorithm for finding MSTs

We can also approach this problem from the opposite direction. Instead of adding edges until we cannot avoid creating a cycle, we start and remove (the most expensive) edges until no cycles are left.

Algorithms & Complexity: Lecture 9, Dynamic Programming

Sam Barrett

March 1, 2021

Dynamic programming is very different from greedy algorithms, greedy algorithms follow a rule *blindly* whereas DP algorithms are more careful.

The general way a dynamic programming algorithm works is by building up a final solution from the solutions of multiple sub-problems. But how do we know which sub-problems to consider? And how do they contribute to the final solution?

1 Fibonacci Numbers

We can define the set of fibonacci numbers recursively as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2}, \forall n \geq 2 \end{aligned}$$

This recursive definition can easily be interpreted into an algorithm:

Algorithm 1: RecursiveFibonacci

```
if $n = 0$ then
    return 0
if $n = 1$ then
    return 1
else
    return RecursiveFibonacci($n - 1$) + RecursiveFibonacci($n - 2$)
end
```

Intuitively, you can see that the path of this algorithm will form a (recursive) tree with the final solution being the root.

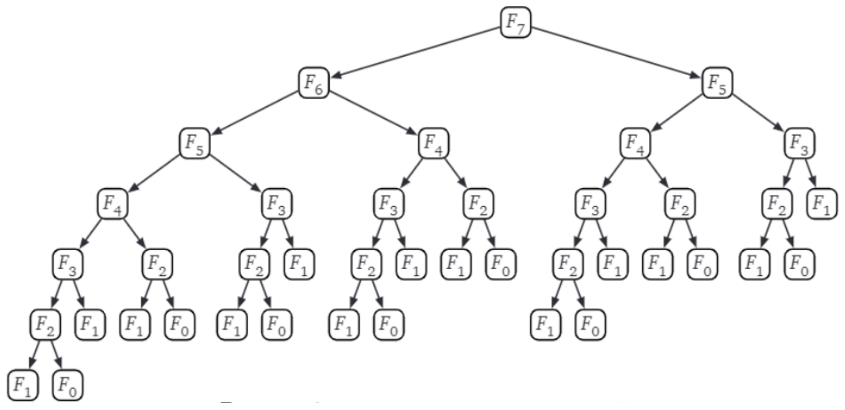


Figure 1: Recursive tree to compute F_7

1.1 Memo(r)isation of recursion

This method looks at the previous algorithm and asks whether it would be more efficient to store values on the first time they are computed so that they may be retrieved from a lookup table on subsequent recursions, for instance you can see that F_2 is recalculated in every sub-tree in Figure 1, If we were to store it we could remove the need for this calculation to be repeated.

Algorithm 2: MemoisationFibonacci

```
1 **if** $n = 0$ **then**
2     return 0
3 **if** $n = 1$ **then**
4     return 1
5 **if** $F[n]$ *is undefined* **then**
6     $F[n] \leftarrow$ MemoisationFibonacci(n-1) + MemoisationFibonacci(n-2)
7 **end**
```

Using this algorithm we can trim or *prune* the tree that needs to be generated.

1.2 Iterative approach

At this point we are already maintaining an array, so why do we not fill it up iteratively? Recursion acts as a layer of abstraction, making our process closer to the formal (recursive) definition of the set. We can remove this and instead write our algorithm as:

What is the running time of such an algorithm?

- we store n items in our array F

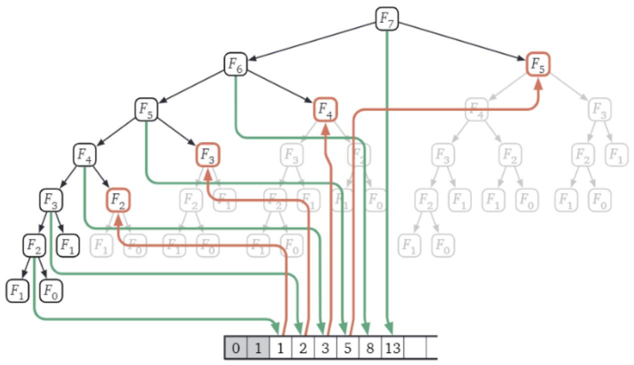


Figure 2: Computing F_7 via memoisation

Algorithm 3: IterativeFibonacci(n)

```
1 $F[0] \leftarrow 0$
2 $F[1] \leftarrow 1$
3 for $i \leftarrow 2$ to $n$ do
4     $F[i] \leftarrow F[i-1] + F[i-2]$
5 end
```

- Computing each new entry needs 2 lookups and one addition.

  Therefore, the total running time to compute F_n is $O(n)$

2 Interval Scheduling Problem

2.1 Without weights (lecture 8 recap)

- We are given a set of n requests R

  $R = \{\texttt{Req}(1), \texttt{Req}(2), \dots, \texttt{Req}(i), \dots, \texttt{Req}(n)\}$

- $\texttt{Req}(i)$ has a start time of $\texttt{Start}(i)$ and a finish time of $\texttt{Finish}(i)$

- There is a machine which can handle one request at a time

- Two requests **conflict** if they overlap

The interval scheduling problem asks:

Problem 1 *(Interval Scheduling) Select a set $C \subseteq R$ of requests such that $|C|$ is maximised and no two requests from C conflict.*

2.2 Weighted

- We are given a set of n requests$R$
  $R = \{\texttt{Req}(1), \texttt{Req}(2), \ldots, \texttt{Req}(i), \ldots, \texttt{Req}(n)\}$
- $\texttt{Req}(i)$ has a start time of $\texttt{Start}(i)$ and a finish time of $\texttt{Finish}(i)$
- **Additionally, each request $\texttt{Req}(i)$ has a weight given by $\texttt{Weight}(i)$**
- There is a machine which can handle one request at a time
- Two requests **conflict** if they overlap

The weighted interval scheduling problem asks:

Problem 2 *(Weighted interval scheduling problem) Select a set $C \subseteq R$ of requests such that $\sum_{i \in C} \texttt{Weight}(i)$ is maximised and no two requests from C conflict.*

I.e. maximise the weight of all chosen requests.
The algorithm seen for unweighted ISP does not hold. (Algorithm)

Algorithm 4: Select requests by increasing order to finish times

```
1 Let $R = \{\texttt{Req}(1), \texttt{Req}(2), \ldots, \texttt{Req}(i), \ldots, \texttt{Req}(n)\}$ be the set of all
    requests
2 Let $C$ denote the set of requests that we select, initialise it as $C = \emptyset$
3 while $R \neq \emptyset$ do
4     Find the request $\texttt{Req}(i) \in R$ which has the smallest finish time.
5     Add $\texttt{Req}(i)$ to $C$
6     Delete from $R$ all requests that conflict with $\texttt{Req}(i)$
7 end
```

Remember: This is an example of a greedy algorithm. It does not take the newly added weights into account, therefore often finds a sub-optimal solution.
Instead to solve the weighted interval scheduling problem we:

Algorithm 5: Weighted interval scheduling algorithm

```
1 Begin by ordering requests in increasing order of finishing time.
2 $M[0] = 0$
3 for each request $j \in 1..n$ do
4     $M[j] = \max\{\texttt{Weight}(j) + M[\texttt{Last}(j)], M[j-1]\}$
5 end
```

Where $\texttt{Last}(i)$ is given by:

$$Last(i) = \begin{cases} i & \text{the largest index } i \text{ s.t. } i \text{ is disjoint from } j \\ 0 & \text{if there is no request } i < j \text{ that is disjoint from } j \end{cases}$$

and is the last compatible request with i.

In this algorithm, $M[j]$ stores the value of the set of requests of maximum weight which can be chosen for the sub-instance containing requests $\{1, 2, \ldots, j\}$. Our goal is then to compute $M[n]$ from the entries $M[1], M[2], \ldots, M[n-1]$ (**Note** these have all been computed previously)

Our recurrence of $\max\{\texttt{Weight}(j) + M[\texttt{Last}(j)], M[j-1]\}$ is basically deciding whether request j is worth including in the final solution, if we had a higher total weight previously without j then we can ignore it (keeping our previous value), but if not, our result is the weight of j along with the result of the sub-instance concerned with the last compatible request with j ($\texttt{Last}(j)$).

2.2.1 Correctness

We can prove the correctness of this algorithm by induction on j. Our base case will be $j = 0$ and our inductive step essentially argues the correctness of our recurrence.

2.2.2 Running time

The running time of this algorithm can be broken down into:

- Sorting the requests in increasing order of finishing time, this can be done in $O(n \log n)$ time.
- We can now find $\texttt{Last}(j)$ for $1 \leq j \leq n$ in $O(n)$ time.
- Filling $M[j]$ requires a comparison between two existing entries from M along with an addition and a max operation. This can be done in $O(1)$ time.

  Therefore filling the entire array M can be done in $O(n)$ time

3 Bellman-Ford algorithm for finding shortest paths

We have previously looked at Dijkstra's algorithm for finding shortest paths. However, this algorithm has one major flaw: it does not work when we have negative edge-lengths.

One might think a simple solution to this problem is to add a constant of the largest negative length to all edges in a graph. I.e. if the lowest edge value in a graph G is -2, by adding 2 to all edges there are no more negative edge lengths. This does not work in practise as it affects what paths are the shortest, preferring paths with fewer edges. (Change in path cost is equal to number of edges multiplied by the constant added).

An alternative algorithm that deals with this is the **Bellman-Ford** algorithm.

This algorithm assumes that there are no **negative cycles**. As this would imply that the optimal route is infinite in number of edges!

By assuming no negative cycles, we can say that for any two vertices s and t, there is a shortest $s \to t$ path which has **at most** $n-1$ edges. We can show this to be correct:

- Let P be a shortest $s \to t$ path with the fewest number of edges.
- If a vertex x repeats on P, then delete the $x \to x$ cycle from P
  - Therefore the number of edges in P decreases
  - The length of P cannot increase as there as no negative edges.

3.1 Defining our algorithm

We can say, for each vertex $v \in V$ and each $0 \leq i \leq n-1$, let $\texttt{OPT}[i, v]$ denote the shortest $s \to v$ path having at most i edges.

We can now set up our recurrence:

1. If the shortest $s \to v$ path actually uses at most $i-1$ edges out of the original i then the value is $\texttt{OPT}[i-1, v]$
2. Otherwise, the last (i^{th}) edge has to be (w, v) for some $w \in V$ as our shortest $s \to v$ path uses all i edges.
   - The cost of this path is $\texttt{OPT}[i-1, w] + \texttt{cost}(w, v)$
   - We need to minimise this over all w s.t. (w, v) is an edge in G

We must take a minimum of these two choices:

$$\texttt{OPT}[i, v] = \min \left\{ \texttt{OPT}[i-1, v], \min_{w \in V} \left(\texttt{length}(w, v) + \texttt{OPT}[i-1, w] \right) \right\}$$

We can now formulate our algorithm: (See Algorithm 6)

3.1.1 Running time

- $\texttt{OPT}$ has $O(n^2)$ entries
- Computing each entry needs to lookup (and compute)$O(n)$ entries.
  - Computing $\texttt{OPT}[i, v]$ needs knowledge of $\texttt{OPT}[i-1, x], \forall x \in V$
  - Needs to perform $O(n)$ min operations and $O(n)$ additions
- Total running time $O(n^3)$

Algorithm 6: Shortest path from a vertex s to all other vertices

```
1 We maintain a $n \times n$ table indexed by $0 \leq i \leq (n-1)$ and $v \in V$ where
  $\texttt{OPT}[i, v]$ stores the length of the shortest $s \to v$ path which uses at
  most $i$ edges.
2 Define $\texttt{OPT}[0, s] = 0$ and $\texttt{OPT}[0, v] = \infty$ for each $v \in V, v \neq s$
3 for $i = 1..n$ do
4     for $v \in V$ do
5         $\texttt{OPT}[i, v] =$
            $\min \left\{ \texttt{OPT}[i-1, v], \min_{w \in V} (\texttt{length}(w, v) + \texttt{OPT}[i-1, w]) \right\}$
6     end
7 end
8 return $\texttt{OPT}[n-1, v]$ for each $v \in V$
```

4 Subset Sum

The subset sum problem is defined as:

Problem 3 *(Subset sum problem) Given n items $\{1, 2, \ldots, n\}$, where each item i has a non-negative integral weight given by $\texttt{Weight}(i)$ and a number W as an* ***upper bound****.*

Find a set S of items such that $\sum_{i \in S} \texttt{Weight}(i)$ is maximised subject to the constraint $\sum_{i \in S} \texttt{Weight}(i) \leq W$

We can show that the greedy algorithm which always picks the item with the heaviest weight (while sum is at most W) fails.

We can show this by counterexample:

- Item 1 has weight 15
- Item 2 has weight 25
- Item 3 has weight 15

Suppose we are given $W = 30$.

Our greedy algorithm picks item 2 as it has the highest weight (while sum is at most W). But now it cannot pick either item 1 or 3 as then the sum would exceed W. However, we can see that the optimal solution is to pick items 1 and 3 to total 30.

In fact, we know that **no** greedy algorithm is known which can solve this problem optimally.

4.1 A DP algorithm

We can attempt to construct a dynamic algorithm to solve this problem:

Let $\texttt{OPT}[i]$ denote the max weight of choosing items from $\{1, 2, \ldots, i\}$ such that sum of weights is maximised subject to being at most W.

Our final answer would therefore be stored in $\texttt{OPT}[n]$.

Similarly to Algorithm 6, if an item i is not chosen to be in our result set, $\texttt{OPT}[i] = \texttt{OPT}[i - i]$, i.e. we skip it and continue with our previous value.

However, if i is chosen to be in the result set (at this point), then $\texttt{OPT}[i]$ contains $\texttt{Weight}(i)$ plus the optimal selection of items from $\{1, 2, \ldots, i - 1\}$ with weight at most $W - \texttt{Weight}(i)$.

Note however, we do not store this value in $\texttt{OPT}[i - i]$, in this location we store the max weight of choosing items from $\{1, 2, \ldots, i\}$ subject to W.

We therefore require a table to store this information.

For each $1 \leq i \leq n$ and $1 \leq X \leq W$, let $\texttt{OPT}[i, X]$ be the max weight of choosing items from $\{1, 2, \ldots, i\}$ subject to weight totalling at most X.

Recurrence We can define our recurrence:

Let $\texttt{OPT}[i, X]$ be the max weight of choosing items from $\{1, 2, \ldots, i\}$ subject to a max weight of X,

1. If i is **not** chosen,

   $\texttt{OPT}[i, X] = \texttt{OPT}[i - 1, X]$, this occurs if $\texttt{Weight}(i) > X$

2. If i is chosen,

   $\texttt{OPT}[i, X] = \texttt{Weight}(i) + \texttt{OPT}[i - 1, X - \texttt{Weight}(i)]$

We can therefore construct the recurrence:

$$\texttt{OPT}[i, X] = \max\{\texttt{OPT}[i - 1, X]; \texttt{Weight}(i) + \texttt{OPT}[i - 1, X - \texttt{Weight}(i)]\}$$

This allows us to construct the following algorithm:

Algorithm 7: Subset Sum(n, W)

```
1 We maintain an $(n+1) \times (W+1)$ table indexed by $0 \leq 0 \leq n$ and
   $0 \leq X \leq W$ where $\texttt{OPT}[i, X]$ stores the max weight of choosing items
   from $\{1, 2, \ldots, i\}$ subject to weight being at most $X$
2 We initialise $\texttt{OPT}[0, Y] = 0$ for each $0 \leq Y \leq W$
3 for $i = 1, 2, \ldots, W$ do
4     for each $Z = 0, 1, 2, \ldots, W$ do
5         $\texttt{OPT}[i, Z] = \max\{\texttt{OPT}[i-1, Z]; \texttt{Weight}(i) + \texttt{OPT}[i-1, Z - \texttt{Weight}(i)]\}$
6     end
7 end
8 return OPT [n,W]
```

4.1.1 Running time

The table has $O(n \cdot W)$ entries. Filling a new entry in the table needs to look-up two existing entries from the table and perform one addition and one max operation. This can be done in constant time $O(1)$

Therefore, the time required to fill all entries is $O(n \cdot W)$.

Algorithms & Complexity: Lecture 11: Divide and Conquer

Sam Barrett

March 17, 2021

Divide and conquer is a very natural and useful algorithmic technique.

It breaks a problem down in to smaller subproblems which can be solved recursively and recombined into a final solution.

Abstractly they can be thought as following the steps:

1. Divide

   decompose the problem into smaller subproblems

2. Conquer

   Solve the smaller subproblems, usually done recursively

3. Combine

   Recombine the solutions to the subproblems into the final solution we return.

1 MergeSort algorithm

Problem 1 *(MergeSort) Given a list of n numbers, we want to sort them in ascending order.*

As specified earlier, we must first divide the problem into smaller problems. We will start by dividing in two (as it is the most trivial way)

$$\begin{tikzcd}
& 6,8,1,4,9,2,3 \arrow[dl] \arrow[dr] & \\
6,8,1 \arrow[d] & & 4,9,2,3 \arrow[d] \\
1,6,8 \arrow[dr] & & 2,3,4,9 \arrow[dl] \\
& 1,2,3,4,6,8,9 &
\end{tikzcd}$$

Above you can see a very naive approach to merge sort. We split the initial problem in half, sort the halves and recombine.

Recombination in this example is done as follows:

1. Initialise pointers p_1, p_2 at the start of both sub-arrays
2. If $A_1[p_1] < A_2[p_2]$ then append $A_1[p_1]$ to the return array and move the p_1 to the right, otherwise add $A_2[p_2]$ and move p_2 to the right
3. repeat until both sub-arrays are empty (pointers cannot move to the right)

A more formal definition for merge sort is:

Algorithm 1: MergeSort(A)

```
1  Divide the given list $A$ on $n$ numbers into two lists $B$ and $C$ of equal
     size
2  Let $B' =$ MERGESORT(B) and $B' = \{b_1 < b_2 < \ldots < b_{n/2}\}$
3  Let $C' =$ MERGESORT(C) and $C' = \{c_1 < c_2 < \ldots < c_{n/2}\}$
4  Initialise $i = 1, j = 1, k = 1$
5  Initialise $D$ to be an empty array
6  while $i \neq n/2 \cap j \neq n/2$ do
7      if $b_i < c_j$ then
8          Set $d_k = b_i$
9          $k, i++$
10     else
11         Set $d_k = c_j$
12         $k, j++$
13     end
14 end
15 If one of the lists becomes empty, append the remainder of the other
     list to $D$
16 return $D$
```

Clearly this algorithm does not simply divide the array once, it will divide recursively until the arrays are a single element in length, relying on the recombination process to actually sort the array.

The time needed to recombine B' and C' to obtain D is $O(n)$.

Therefore, the recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n)$.

We can show (but won't) that $T(n) = O(n \log n)$ satisfied this recurrence.

2 Solving recurrences

At the end of the merge sort section we saw that we formulated a recurrence to describe the complexity of our divide and conquer algorithms.

There are two main methods for solving these recurrences.

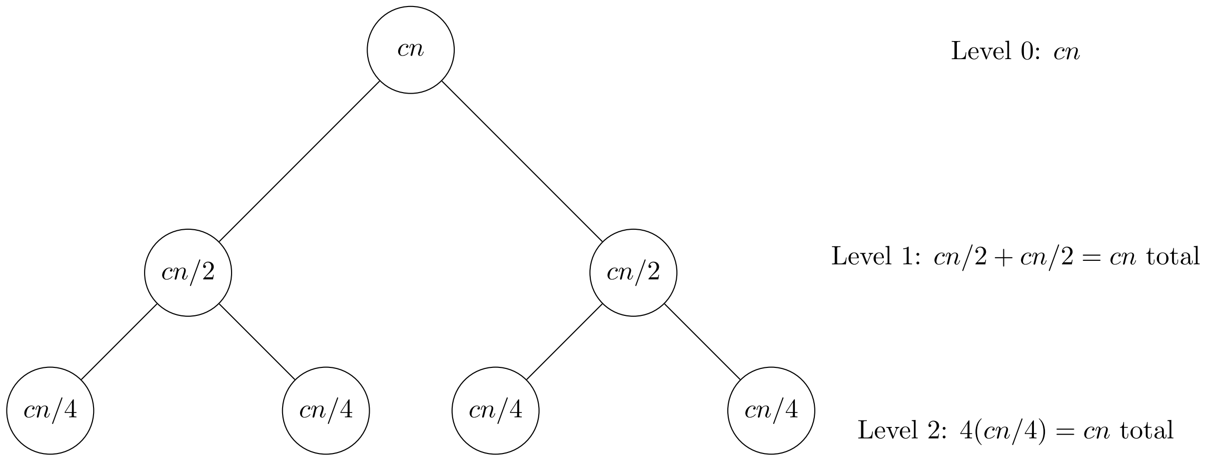
2.1 Method 1: *Unrolling* the recurrence

In this method we *open up* the recurrence step-by-step. It does not require any knowledge of the final result of the process.

We start by writing our recurrence for some constant value c:

$$T(n) \leq 2 \cdot T(n/2) + cn, \forall n > 2$$

Base Case: $T(2) \leq c$


cn
Level 0: cn
$cn/2$
$cn/2$
Level 1: $cn/2 + cn/2 = cn$ total
$cn/4$
$cn/4$
$cn/4$
$cn/4$
Level 2: $4(cn/4) = cn$ total

Note that at each step/level we have exactly cn running time. This is our pattern, and would continue until we reach a level in which each node is of size 2 in which case we can use our base case.

We must now sum over all levels of the recursion. We will have $\log_2 n$ levels as our base case is that $n = 2$. It will take $\log_2 n$ splits until we reach sub-arrays of size 2.

We can therefore say that as we have $\log_2 n$ levels and each level has cn running time, that our overall running time is $cn \cdot \log_2 n$.

We can formalise this process as follows:

1. Write the recurrence explicitly for some constant c

   $T(n) \leq 2 \cdot T(n/2) + cn$

2. Construct our base case

   $T(2) \leq c$

3. Analyse the first few levels

   Level 0 takes cn time, and delegates two recursive calls of size $n/2$

   Level 1 takes $c(n/2)+c(n/2)$ time and delegates four recursive calls of size $n/4$

4. Identify a pattern

   The number of subproblems doubles at each level

   The size of each subproblem halves at each level

   Level j has 2^j subproblems each having size $n/2^j$

   So level j takes a total of $2^j \cdot (c \cdot (n/2^j)) = cn$ time and delegates some tasks to the next level

5. Sum up over all levels of the recursion

   There are $O(\log_2 n)$ levels of the recursion

   Each level requires a running time of cn

   $T(n) = cn \cdot \log_2(n) = O(n \cdot log_2 n)$

2.2 Verifying by substitution in the recurrence

This works well if you already have a *guess* for the running time. You can then check by induction whether your guess satisfies the recurrence.

- **Base Case:** $n = 2$

  $T(2) \leq c \leq cn \log n$

- **Inductive Hypothesis**

  For each $2 \leq m \leq n$ we have $T(m) \leq cm \cdot \log m$

- **Inductive step**

$$\begin{aligned} T(n) &\leq 2T(n/2) + cn && \text{(by the recurrence)} \\ &\leq 2c \cdot (n/2) \cdot \log(n/2) + cn && \text{(by inductive hypothesis since } n/2 < n) \\ &= cn(\log n - 1) + cn && \text{(since } \log(n/2) = \log n - 1) \\ &= cn \cdot \log n \end{aligned}$$

NOTE: we have here verified that $T(n) = cn \cdot \log n$ is <u>one</u> possible solution for this recurrence, unlike the unrolling method, this method may lead to verifying that a much higher running time is correct.

Algorithms & Complexity: Lecture 11: Divide and Conquer II

Sam Barrett

March 17, 2021

1 Counting number of inversions

Problem 1 *(Inversion)*
Given an array of n pairwise disjoint (unique) numbers $a_1, a_2, \ldots, a_n$
Two numbers a_i and a_j form an inversion if $i < j$ but $a_i > a_j$

The number of inversions is a measure of the *sortedness* of an array.

The naive approach to solving this requires $O(n^2)$ time, as it checks if each of the $\binom{n}{2}$ pairs is an inversion or not.

We can instead construct an algorithm similar to MergeSort:

1. Divide

   Divide the list L into two equal sections L_1 and L_2

2. Conquer

   Count the number of inversions within L_1 and L_2

3. Combine

   Count the number of inversions where one number is from L_1 and the other is from L_2

Analysis:

Let $T(n)$ be the time needed to count the number of inversions from a given list of n numbers.

- The divide step requires $O(1)$ time
- The Conquer step requires $2 \cdot T(n/2)$ steps
- The combine step requires $O(n^2)$ steps as $|L_1| = n/2 = |L_2|$
- Our recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n^2)$

This approach doesn't appear to be any better than our naive $O(n^2)$ approach. Can we improve the combination step?

If we can get the combination step down to $O(n)$ time the our entire procedure can be reduced to the complexity of MergeSort $(T(n) = O(n \cdot \log n))$

1. Divide

   Divide the list L into two equal sections L_1 and L_2

2. Conquer

   Sort and count the number of inversions within L_1 and L_2

3. Combine

   We can assume that both lists are **sorted** Count the number of inversions where one number is from L_1 and the other is from L_2

Analysis:

- The divide step requires $O(1)$ time
- The Conquer step requires $2 \cdot T(n/2)$ steps
- The combine step:

  It can be shown that this step now only needs $O(n)$ time as both lists are sorted

Algorithm 1: Combine(L_1, L_2)

```
Input: Two sorted lists $L_1, L_2$ of length $n/2$ each
Let $L_1 = \{b_1 < b_2 < \ldots < b_{n/2}\}$
Let $L_2 = \{c_1 < c_2 < \ldots < c_{n/2}\}$
Initialise $i, j = 1$
Initialise $c = 0$ ;      // counter to maintain number of inversions
while $i \neq n/2 \cap j \neq n/2$ do
    ;             // Even if one list becomes empty, we can stop
    if $b_i > c_j$ then
        $j{+}{+}\ c{+}{=}(n/2) - i + 1$ ;    // All numbers in $L_1$ after $a_i$ are $> b_j$
    else
        ;                          // In this case we have $b_i < c_j$
        $i{+}{+}$
    end
end
return $c$
```

Running time:

- In each step, either i or j increases

    - The while loop ends when one of the lists becomes empty
    - Hence, the total running time is $|L_1| + |L_2| = O(n)$

- Our recurrence is $T(n) \leq 2 \cdot T(n/2) + O(n)$

  Same as for MergeSort and we have solved this recurrence to $T(n) = O(n \log n)$

We can now construct an algorithm `Sort-and-Count` which takes a list L and returns the number of inversions in L and the sorted version of L.

Algorithm 2: `Sort-and-Count`(L)

```
1 Divide $L$ into two lists $L_1$ and $L_2$
2 Let $(r_1, L_1') =$ Sort-and-Count$(L_1)$
3 Let $(r_2, L_2') =$ Sort-and-Count$(L_2)$
4 Let $(c, L')$ be the output of Combine$(L_1', L_2')$
5 return $c + r_1 + r_2$
6 return $L'$
```

2 Faster integer multiplication

We will make two assumptions:

1. Multiplying 2 bits can be done in constant time
2. Adding 2 bits can be done in constant time

When we consider the process of long multiplication, we will split the problem of, for example, 12×13 into 12×10 and 12×3 and combine these results. Intuitively we can see that this process is similar to that which we have been employing to create divide and conquer algorithms.

In this formulation (for multiplying two binary strings of length n) we need to add $O(n)$ binary strings where each binary string takes $O(n)$ time to compute. This leads to a total running time of $O(n^2)$.

2.1 Attempt 1

Let x_1, x_0 be the first and last $n/2$ bits respectively.

We can therefore say that: $x = x_0 + 2^{n/2} \cdot x_1$, where we also know that both $x_{0,1}$ have length $n/2$

Let y_1, y_0 be the first and last $n/2$ bits respectively.

We can therefore say that: $y = y_0 + 2^{n/2} \cdot y_1$, where we also know that both $y_{0,1}$ have length $n/2$

We can then also see that:

$$\begin{aligned} x \cdot y &= (x_0 + 2^{n/2} \cdot x_1) \cdot (y_0 + 2^{n/2} \cdot y_1) \\ &= x_0 y_0 + 2^{n/2} \cdot (x_1 y_0 + x_0 y_1) + 2^n \cdot x_1 y_1 \end{aligned}$$

We therefore need to solve the following four instances of multiplication of binary strings of length $n/2$ each:

- x_0 and y_0
- x_1 and y_1
- x_0 and y_1
- x_1 and y_0

Giving us the recurrence: $T(n) \leq 4 \cdot T(n/2) + O(n)$. We know this recurrence solves to a running time of $O(n^2)$, no better than our naive algorithm

2.2 Attempt 2

We will require the following three quantities:

1. $x_0 y_0$
2. $x_1 y_1$
3. $x_0 y_1 + x_1 y_0$

We have previously shown that these can be obtained from solving four instances of size $n/2$, but can we do better?

Observe:

$$x_0 y_1 + x_1 y_0 = (x_1 + x_0)(y_1 + y_0) - x_1 y_1 - x_0 y_0$$

We therefore only need to solve **three** instances of multiplications of binary strings of length $n/2$:

1. x_0 and y_0
2. x_1 and y_1
3. $(x_0 + x_1)$ and $(y_0 + y_1)$

This gives us the recurrence: $T(n) = 3 \cdot T(n/2) + O(n)$ which can be solved to a time of $O(n^{\log_2 3}) \equiv O(n^{1.59})$. This is an improvement over our naive and first approach.

Algorithms & Complexity: Lecture 13: Coping with **NP**-hardness

Sam Barrett

March 23, 2021

If we assume that $\mathbf{P} \neq \mathbf{NP}$, then it follows that **no** **NP**-hard problem has a polytime solution. But we still want to be able to solve these problems in the most efficient way possible.

We will now look at designing exact exponential time algorithms for **NP**-hard problems as these are still preferable to brute force. (**note** here *exact* simply means that the algorithm solves the problem *exactly*).

1 Algorithms via branching

1.1 Example: Vertex Cover

Problem 1 *(Vertex Cover) Given an undirected graph $G = (V, E)$ on n vertices and m edges, find a set X of minimum size s.t. each edge of G has at least one endpoint in X*

The brute force approach to this problem runs in $2^n \cdot n^{O(1)}$ time and works by:

- Enumerating all 2^n subsets of vertex set V
- For each set $X \subseteq V$, check in $O(|X|)$ time if each of the m edges has at least one endpoint in X

We now ask ourselves: Can we design an *"Can we design an $(2-\epsilon)^n \cdot n^{O(1)}$"* time algorithms for some $\epsilon > 0$?

Note we focus on minimising the 2 in $2^n \cdot n^{O(1)}$ as:

$$\lim_{n\to\infty} \left(2^n > \forall k \in \mathbb{N}.n^k\right)$$

Or as n gets increasingly larger, the 2^n gets *exponentially* larger.

Our algorithm relies on the observation that there is no point adding any vertex of degree 1 to the vertex cover. I.e. a vertex only connects to one other vertex. In this case we lose nothing by just adding the vertex it is connected to. This second vertex has the possibility of being as good **or better** than the degree-1 vertex.

Now we can assume that there will be **no vertices of degrees < 2**. We can then say that for each vertex v of degree ≥ 2

- If we pick v then the number of vertices (remaining to be *covered*) reduces by 1
- If we do not pick v then the number of vertices reduces by at least $2+1 = 3$. This is as by not picking v **all** of its neighbours must now be picked and v has at least 2 neighbours. (2 refers to the (at least) 2 neighbours of v and the 1 refers to v being implicitly covered by selecting its neighbours)

If we define $T(n)$ as the time needed to solve the `VertexCover` problem for a graph with n vertices, we can construct the following recurrence:

$$T(n) \leq T(n-1) + T(n-3)$$

Where we solve both instances: where we pick v and where we do not and take the minimum of the two results. This gives us a total running time of $T(n-1) + T(n-3)$

Base case:

$T(2) = 1$ as for the case where the graph is two vertices connected by a single edge we select one node and have the minimum vertex cover. This requires a single operation.

Our recurrence is what is known as a **linear homogenous recurrence**

- *linear* - terms on the RHS are raised to the power 1
- *homogenous* - there are no constants on the RHS

The standard method for solving such a recurrence is to set $T(n) = x^n$ and solve for x:

- $T(n) \leq T(n-1) + T(n-3) \mapsto x^n \leq x^{n-1} + x^{n-3}$
- Taking x^{n-3} as a common factor gives: $x^3 \leq x^2 + 1$
- This is satisfied by $\forall x \leq 1.46557$
- Therefore, we can say that $1.47^n \cdot n^{O(1)}$ is an upper bound for our branching algorithm.

2 Algorithms via Dynamic Programming

2.1 Example: Travelling salesman problem (TPS)

Problem 2 *(Travelling Salesman Problem)*

- *Given a set C of n cites $c_1, c_2, \ldots, c_n$*
- *A distance function $\texttt{dist} : C \times C \to \mathbb{R}^{\geq 0}$ which gives the distance between every pair of cities*
- *We want to start at c_1 and end at c_1 after visiting* ***all*** *cities on the way*
- *What order should we visit each city to minimise the total distance we have travelled?*

The brute force approach tries all $n!$ orderings. And for each computes its cost by summing all n values. Hence, the running time of this approach is $n! \approx \left(\frac{n}{e}\right)^n$. (Where e is the Euler constant equal to around 2.718)

2.1.1 Deriving our recurrence relation

For each $S \subseteq \{c_2, c_3, \ldots, c_n\}$ and each $c_i \in S$, let $\texttt{OPT}[S, c_i]$ be the minimum length of a tour that starts at c_1, visits all cities in S and ends at c_i

Base Case: when $|S| = 1$; for each $i \geq 2$ we have $\texttt{OPT}[\{c_i\}, c_i] = \texttt{dist}(c_1, c_i)$

Now suppose that $|S| > 1$ and we want to compute $\texttt{OPT}[S, c_i]$ for some $c_i \in S$.

- Let c_j be the last city visited before ending at c_i
- c_j **must** be in $S \backslash \{c_i\}$
- Looking at all possibilities gives the following recurrence:

$$\texttt{OPT}[S, c_i] = \min_{c_j \in (S \backslash \{c_i\})} \texttt{OPT}[S \backslash \{c_i\}, c_j] + \texttt{dist}(c_j, c_i)$$

Using this recurrence we can construct the algorithm **??** and analyse its running time:

The number of entries in our table $\texttt{OPT}$ is $O(2^n \cdot n)$ as we maintain an entry $\texttt{OPT}[S, c_i]$ for each $S \subseteq \{c_2, \ldots, c_n\}$ and each $c_i \in S$

To compute $\texttt{OPT}[S, c_i]$ we take the minimum of $|S|$ numbers. To do this we look up $|S| - 1$ entries and do $|S| - 1$ addition operations. Giving an overall running time on $O(n)$ since $|S| \leq n - 1$

Algorithm 1: Travelling Salesman Problem (TSP)

```
Input: A set $C = \{c_1, c_2, \ldots, c_n\}$ of $n$ cities and a distance function
       $\texttt{dist} : C \times C \to \mathbb{R}^{\geq 0}$
Output: The value/distance of a tour which minimises the total
        distance travelled when starting and ending at $c_1$, while
        visiting all cities from $C$
1  for $i = 2 : n$ do
2      $\texttt{OPT}[\{c_i\}, c_i] = \texttt{dist}(c_1, c_i)$
3  end
4  for $k = 2 : n$ do
5      for $S \subseteq \{c_2, c_3, \ldots, c_{n-1}, c_n\}$ with $|S| = k$ do
6          $\texttt{OPT}[S, c_i] = \min_{c_j in (S \backslash \{c_i\})} \texttt{OPT}[S \backslash \{c_i\}, c_j] + \texttt{dist}(c_j, c_i)$
7      end
8      k++
9  end
10 return $\min_{2 \leq i \leq n} \texttt{OPT}[\{c_2, c_3, \ldots, c_{n--1}, c_n\}, c_i] + dist(c_i, c_1)$
```

2.2 Set Cover problem

Problem 3 *(Set Cover problem) Given:*

- *A set $U = \{u_1, u_2, \ldots, u_N\}$ of N elements*
- *A set $\mathcal{S} = \{S_1, S_2, \ldots, S_M\}$ of non-empty subsets of U s.t. $|\mathcal{S}| = M$*

Find a collection $\mathcal{S}'$ from $\mathcal{S}$ of minimum size s.t. the unions of sets in $\mathcal{S}'$ covers all elements of U. We assume that the union of all sets in $\mathcal{S}$ covers all elements in U

The brute force approach to this problem takes $O(2^M \cdot M \cdot N)$ time as it:

- Tries all possible 2^M subsets of $\mathcal{S}$
- On each subset we can check in $O(M \cdot N)$ time if the union of all sets in the subset covers all elements in U.

  Each of the $\leq M$ sets in our subset $\mathcal{S}'$ can have at most N elements each.

Can we do better than this?

2.2.1 Setting up our recurrence

For each non-empty subset $X \subseteq U$ and each $1 \leq j \leq M$, let $\texttt{OPT}[X, j]$ be the size of the minimum cardinality subset of $\{S_1, S_2, \ldots, S_j\}$ that covers all elements from X

Base Case: $j = 1$

TODO: complete this along with formative assignment (6?)

Algorithms & Complexity: Lecture 14: Coping with **NP**-hardness II

Sam Barrett

March 23, 2021

We have previously seen that `IndependentSet` $\leq_P$ `VertexCover` as if S is an independent set, $V \backslash S$ is a vertex cover. I.e. to convert an independent set to a vertex cover, take its compliment and vice vira.

We can therefore say that the time complexity of `IndependentSet` and `VertexCover` is the **same** . Any $f(n)$ time algorithm for one problem also works for the other.

We know that `VertexCover` is a **NP**-hard problem but what if we are not concerned with finding the **minimum** sized vertex cover and instead a cover of some arbitrary size 25?

1 Fixed-parameter tractable (FPT) algorithms

We can define the parameterised vertex cover problem to be the same as the standard vertex cover problem but instead of attempting to find the minimum sized cover we attempt to find a vertex cover of size $\leq k$

We say the paramterised vertex cover problem is fixed-parameter tractable if it can be sovled in $f(k) \cdot n^{O(1)}$ time.

Problem 1 *(The k-VC problem) Given an undirected graph $G = (V, E)$ on n vertices and a parameter k, is there a set $X \subseteq V$ of size $\leq k$ s.t. each edge of G has at least one endpoint in X?*

The standard `VertexCover` problem can then be solved using a series of $O(n)$ calls to the k-VC problem.

If we assume $\mathbf{P} \neq \mathbf{NP}$, we cannot then expect to find an algorithm for k-VC which runs in time polynomial in **both** k and n.

We can construct an algorithm for k-VC which uses bounded-depth search trees.

This algorithm is best visualised as:

If at least one of the leaf nodes at the bottom of the graph has no edges remaining then there exists a vertex cover of size $\leq k$, otherwise one does not exist.

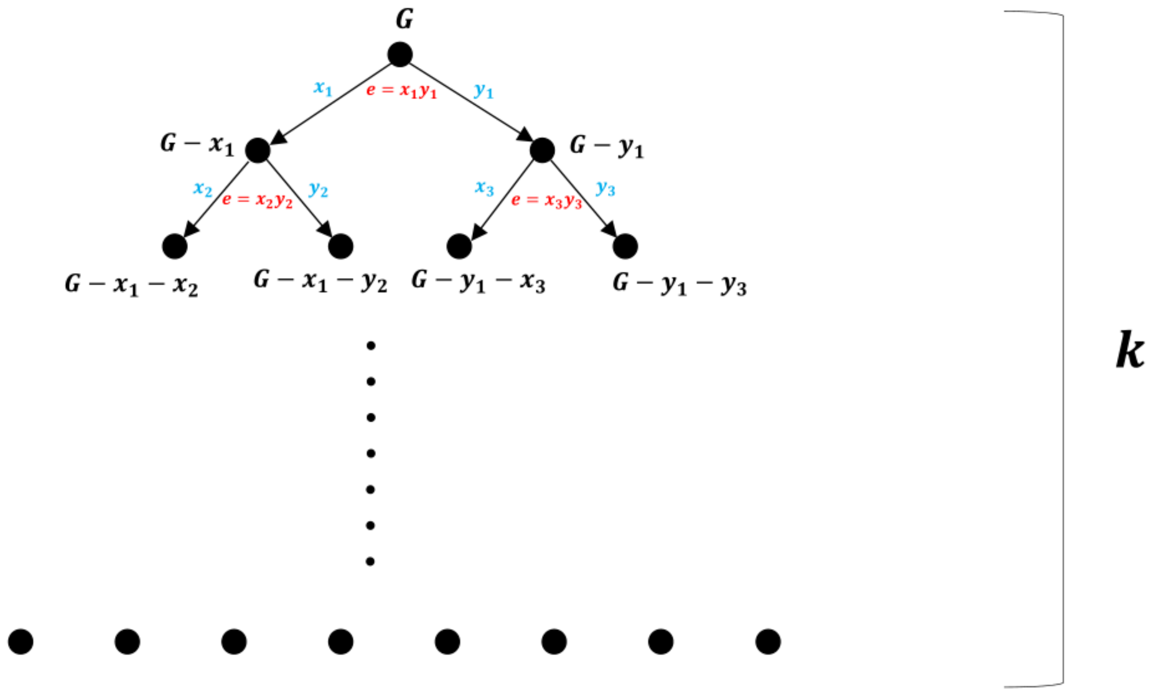


Figure 1: FPT k-VC algorithm

This algorithm can be seen to have a running time in $2^k \cdot n^{O(1)}$. As there are 2^k leaf nodes for a tree of height k and we must spend $n^{O(1)}$ time at each node to determine if any edges remain.

0.1 Introduction

0.1.1 Non-uniform models

The difference between **uniform** and **non-uniform** models of computation is that for **uniform** models, we will have a single program which can handle inputs of any length. Whereas, for non-uniform models, we have a different program for each input length $n \in \mathbb{N}$

0.1.2 CNFs

A CNF formula is a logical formula which fits the form:

$$\bigwedge_i \bigvee_j l_{ij}$$

Where each l_{ij} is a literal, i.e. either x or $\neg x$ for some variable x.

We define computation by CNF as follows:

Definition 1 *(Computation by CNF) Given a Boolean function, $f : \{0,1\}^n \to \{0,1\}$, we say that f is computed by a CNF, $A(x_1, \ldots, x_n)$ if, for any $\vec{b} \in \{0,1\}^n$, we have:*

$$A(b_1, \ldots, b_n) = f(b_1, \ldots, b_n)$$

I.e. for any assignment of A, f produces the same result.

Theorem 1 *We can show that for **every** Boolean function, $f : 0, 1^n \to 0, 1$ there is a CNF A_f which computes it.*

We show this as follows:

Proof 1 *Let $F := \{\vec{b} \in \{0,1\}^n : f(\vec{b}) = 0\}$ For $b \in \{0,1\}$ write:*

$$b \star x := \begin{cases} x & b = 0 \\ \neg x & b = 1 \end{cases}$$

and define:

$$A_f := \bigwedge_{\vec{b} \in F} \bigvee_{i=1}^{n} b_i \star x_i$$

Recall that we link Boolean functions, $f : 0, 1^n \to 0, 1$ with languages $\subseteq \{0,1\}^n$.

We say that a language $L \subseteq \{0,1\}^*$ is computed by a **family** $\{A_n\}_{n=1}^{\infty}$ of CNFs if, for each $n \in \mathbb{N}$, A_n is a CNF computing $L \cap \{0,1\}^n$.

With this we can define our first *non-uniform* class:

Definition 2 *The size of a CNF A, $|A|$ is the number of literals occurrences it contains.*

***CNF**^{poly} is the class of languages computed by polynomially sized families of CNFs*

Undecidability

Proposition 1 **CNF**^{poly} *contains undecidable languages*

Proof 2 *Let $H \subseteq \mathbb{N}$ be the set of natural numbers coding for halting Turing Machines (excluding inputs). Define the following sequence of CNFs:*

$$A_n := \begin{cases} x_1 \lor \neg x_1 & n \in H \\ x_1 \land \neg x_1 & n \notin H \end{cases}$$

A_n evaluates to 1 on an input $\in \{0,1\}^n$ iff the n^{th} TM halts.

The induced language $\subseteq \{0,1\}^*$ is therefore clearly undecidable by reduction to H.

This is the case as if $H \subseteq \mathbb{N}$, $H \subseteq *$. This shows that on the set of all Languages, and therefore all Boolean functions in **CNF**$^{\text{poly}}$, there exist undecidable languages.

0.1.3 Boolean Circuits: Basics

A Boolean circuit is similar to a formula, but we can reuse *sub-formulas*. They are represented as a *well-founded* diagram built from Boolean gates.

Boolean formulae which we have seen earlier can be though of as a special class of circuits whose underlying graphs are trees.

Definition 3 *(Boolean Circuit) An n-input Boolean circuit C is a* ***directed acylic graph*** *with:*

- *n sources, $x_1, \ldots, x_n$*

  These are visualised as vertices with no incoming edges (leaves in a tree)

- *one sink s*

  A vertex with no outgoing edges (root in a tree)

- *A labelling from nonsource vertices to the set of gates, by default, $\{\neg, \land, \lor\}$*

- *Vertices labelled $\neg$ are said to have* ***fan-in*** *1, i.e. 1 incoming edges*

- *Vertices labelled $\lor$ or $\land$ have* ***fan-in*** *2*

The size of C is written $|C|$, and is the number of vertices.

Computation of Boolean Circuits Given a dag C and an assignment $b : \{x_1, \dots, x_n\} \to \{0, 1\}$ we can define a value $b(v)$ at a node v by induction over the structure of C:

- $b(x_i)$, is already defined as these are the input nodes
- any nonsource node labelled $\neg$ with an incoming edge from node u can be said to have a value $b(v) := 1 - b(u)$
- any nonsource node labelled with $\vee$, with incoming edges from u_0 and u_1 can be said to have a value $b(v) := \max(b(u_0), b(u_1))$
- Nonsource nodes labelled $\wedge$ with inputs from nodes u_0, u_1 has a value $b(v) := \min(b(u_0), b(u_1))$

With the above definitions, we can say that the value of $b(C)$ is the same as $b(s)$.

We can show that this definition of outputs allows us to construct a polytime algorithm for evaluating circuits.

Proposition 2 *(We can evaluate a boolean circuit C in polynomial time)*
The set of pairs (C, b) where

- *C is an n input circuit and*
- *$b : 0, 1^n \to 0, 1$ s.t. $b(C) = 1$*

is in **P**

Depth of a Circuit The depth of a circuit is C is the length of the longest path, i.e. the maximum number of edges from a source node to s, in its underlying graph.

The depth of a node is the longest path from a source node to that node.

The inductive definition of circuit depth is the same as for the structural induction seen earlier.

Proposition 3 *If $L \subseteq \{0, 1\}^n$ is computed by a circuit C of depth d, then it is also computed by a formula of size $< 2^d$.*

Proof 3 *For a node v of C, define the formula $F(v)$ by induction on the depth of v.*

- *$F(x_i)$ is just the formula x_i, this has size 1 ($1 < 2 = 2^1$)*
- *If v of depth e is labelled with $\neg$ with an incoming edge from node u (of depth $< e$)m then $F(v) := \neg F(u)$ (of size $< 2^{e-1} + 1, \therefore < 2^e$)*

- *If v of depth e is labelled $\star \in \{\wedge, \vee\}$ with incoming edges from u_0, u_1 (both of depth $< e$), then $F(v) := (F(u_0) \star F(u_1))$ which has size $< 2^{e-1} + 2^{e-1} + 1$ which is $< 2^e$*

We have now constructed $F(s)$ which is a formula computing L in size $< 2^d$

We say that a language $L \subseteq \{0,1\}^*$ is computed by a circuit family $\{C_n\}_{n=1}^{\infty}$ if, for each $n \in \mathbb{N}$, C_n is an n-input circuit computing $L \cap \{0,1\}^n$

For $f : \mathbb{N} \to \mathbb{N}$ we define:

- **SIZE**$(f(n))$ as the set of languages $L \subseteq \{0,1\}^*$ computed by circuits of size $\leq f(n)$
- **DEPTH**$(f(n))$ as the set of languages $L \subseteq \{0,1\}^*$ computed by circuits of depth $\leq f(n)$

The class $\mathbf{P}\backslash poly$ is the class of languages computed by polynomial-size circuit families, i.e.:

$$\mathbf{P}\backslash poly := \bigcup_{c=1}^{\infty} \mathbf{SIZE}(n^c)$$

1 EXACT functions

We define the set of functions $\texttt{EXACT}^n_k : 0, 1^n \to 0, 1$ as:

$$\texttt{EXACT}^n_k(x_1, \ldots, x_n) = 1 \iff \text{precisely } k \text{ of } x_1, \ldots, x_n \text{ are } 1$$

These functions also satisfy the following recurrence:

$$\texttt{EXACT}^1_k(x_1) = \begin{cases} \neg x_1 & k = 0 \\ x_1 & k = 1 \\ x_1 \wedge \neg x_1 & \text{otherwise} \end{cases}$$

With an inductive case defined as follows:

$$\texttt{EXACT}^{n+1}_k(\vec{x}, x_{n+1}) = (x_{n+1} \wedge \texttt{EXACT}^n_{k-1}(\vec{x}) \vee (\neg x_{n+1} \wedge \texttt{EXACT}^n_k(\vec{x})))$$

Here we essentially do a case analysis on a newly introduced variable x_{n+1}, the LHS of the disjunction states: if x_{n+1} is 1, then to satisfy $\texttt{EXACT}^{n+1}_k$, we must have $k - 1$ 1s (true values) in the remaining variables $\vec{x}$. Alternatively on the RHS of the disjunction, we say if x_{n+1} is false, there must be exactly k 1s in $\vec{x}$ in order to satisfy $\texttt{EXACT}^{n+1}_k$.

We can show that circuits computing $\texttt{EXACT}^n_k$ ($\texttt{EX}^n_k$) are polynomial in n and k:

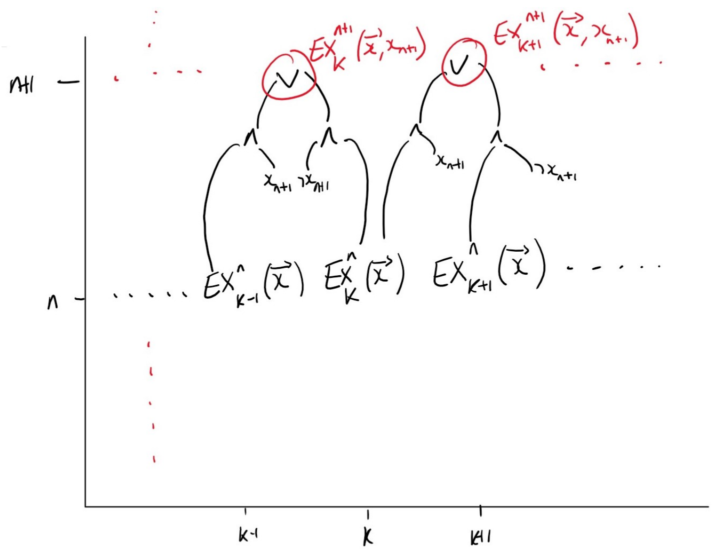


Figure 1: Polynomial-size circuits for $\texttt{EXACT}^n_k$

In Figure 1 you can see that we can construct our inductive case for any n and k. We must be careful to construct all circuits for k before attempting to construct the next layer of n.

Because we only need to consider values of k from $-n$ to n we add a constant number of nodes each time we increase n, we can say that this construction is polynomial in size wrt n and k.

2 Inequality

Inequality is known to be non-symmetric. Symmetric boolean functions are boolean functions whose results depend only on the number of 1s in the input.

We define the $\leq$ relation on natural numbers as $\texttt{LEQ}^n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ by:

$$\texttt{LEQ}^n = 1 \iff \text{number coded by } \vec{x} \leq \text{ number coded by } \vec{y}$$

$$\texttt{LEQ}^1(x, y) = \neg x \vee y$$

$$\texttt{LEQ}^{n+1}(\vec{x}x_{n+1}, \vec{y}y_{n+1}) = \begin{cases} \texttt{LEQ}^1(\vec{x}, \vec{y}) \wedge \texttt{NEQ}^n(\vec{x}, \vec{y}) & x_{n+1} \wedge \neg y_{n+1} \\ \texttt{LEQ}^n(\vec{x}, \vec{y}) & \text{otherwise} \end{cases}$$

Where $\texttt{NEQ}^n(\vec{x}, \vec{y}) = 1, iff \vec{x} \neq \vec{y}$. We also know that $\texttt{NEQ}^n$ can be computed by linear-size circuits.

We can again demonstrate the polynomial size of this circuit family computing the $\leq$ relation:

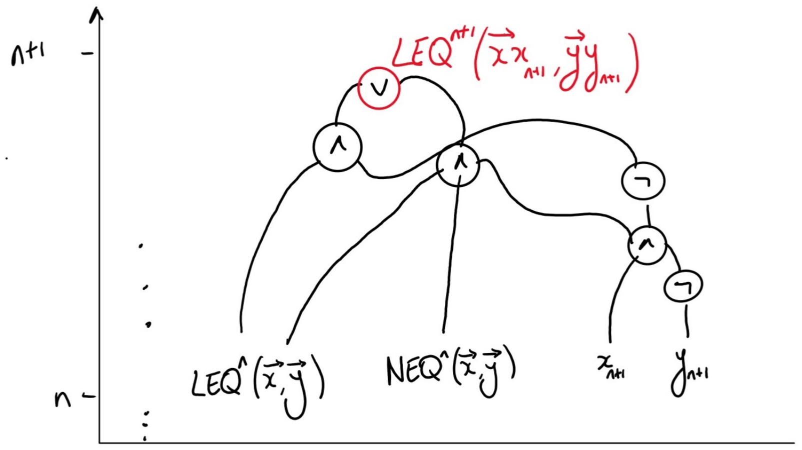


Figure 2: Polynomial size circuits for inequality

3 Composing Constructions

We can compose the previous constructions through the use of **Boolean Connectives** in order to define new circuit families:

For instance the set $\texttt{ALMOST}_k^n \subseteq \{0,1\}^n$ is the set of binary strings with **at most** k 1s. It can be computed as follows using our `EXACT` circuit family:

$$\texttt{EXACT}_0^n(\vec{x}) \lor \texttt{EXACT}_1^n(\vec{x}) \lor \ldots \lor \texttt{EXACT}_k^n(\vec{x})$$

3.1 Undecidability

In the last lecture we saw how we can construct the set $\mathbf{P}\backslash poly$.

We constructed this set in relation to the length of an input to a function.

We said that if the length of the input codes a halting Turing machine, it is in $\mathbf{P}\backslash poly$. And more generally defined it as:

$$\mathbf{P}\backslash poly := \bigcup_{c=1}^{\infty} \mathbf{SIZE}(n^c)$$

We can therefore say that any unary language is in $\mathbf{P}\backslash poly$ as in this case $n = 1$. Meaning Any unary language has the possibility of being undecidable.

It can also be shown that $\mathbf{P} \subseteq \mathbf{P}\backslash poly$. This allows us to recover polynomial sized circuit families for a wide variety of problems.

This also allows us to consider Turing machines as *high level* programs which can be *compiled* to *low level* circuitry with low complexity overhead.