Discrete Structures for Computer Science

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU CSci 341

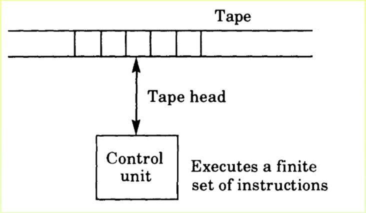
Subsection 1

Turing Machines and the Church-Turing Thesis

The Idea of a Turing Machine

- A **Turing machine** consists of a tape and a control unit.
  - The **tape** is a sequence of cells that extends to infinity in both directions.

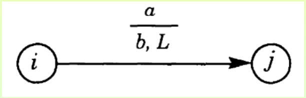
    Each cell contains a symbol from a finite alphabet Γ.



    A tape head reads from a cell and writes into the same cell.
  - The **control unit** contains a finite set of instructions.
    Each **instruction** causes the tape head to:
    - read the symbol from a cell;
    - write a symbol into the same cell;
    - either move the tape head to an adjacent cell or leave it at the same cell.

Instructions of a Turing Machine

- Each instruction of a Turing machine can be represented as a 5-tuple consisting of the following five parts:
  - The current machine state;
  - A tape symbol read from the current tape cell;
  - A tape symbol to write into the current tape cell;
  - A direction for the tape head to move (*L*eft, *S*tay, *R*ight);
  - The next machine state.
- Example: Suppose we have the instruction (i, a, b, L, j). The instruction is interpreted as follows:


$$\begin{tikzcd} \boxed{i} \arrow[rr, "\frac{a}{b,\,L}"] & & \boxed{j} \end{tikzcd}$$

  If the current state of the machine is i, and if the symbol in the current tape cell is a, then:
  - write b into the current tape cell;
  - move left one cell;
  - go to state j.

Formal Definition of Turing Machines

- A **Turing machine** $M = \langle A, \Gamma, Q, q_0, h, \delta \rangle$ consists of:
  - An **input alphabet** A, consisting of the symbols that are allowed in the initial tape contents;
  - A **tape alphabet** Γ, which is a finite set of symbols, such that:
    - $\Lambda \in \Gamma$, called the **blank symbol**;
    - $A \subseteq \Gamma - \{\Lambda\}$;
  - A finite nonempty set of **states** Q;
  - $q_0 \in S$ is the **initial state**;
  - $h \in S$ is the **final** or **halt state**;
  - $\delta \subseteq (Q - \{h\}) \times \Gamma \times \Gamma \times \{L, S, R\} \times Q$ is the **transition relation**.
- If the transition relation contains two instructions with the same current state and current tape symbol, then the machine is **nondeterministic**.
- Otherwise it is **deterministic**.

How the Turing Machine Computes

- An input string is represented on the tape by placing the letters of the string in contiguous tape cells.
- All other cells of the tape contain the blank symbol Λ.
- The tape head is positioned at the leftmost cell of the input string unless specified otherwise.
- The computation starts at the start state q_0.
- The execution of a Turing machine stops when:
    - it enters the halt state h; or
    - it enters a state for which there is no valid move.
- Example: If a Turing machine enters state i and reads a in the current cell, but there is no instruction of the form $(i, a, \ldots)$, then the machine stops in state i.

The Language Accepted by a Turing Machine

- Let M be a Turing machine.
- We say that an input string is **accepted** by M if the machine enters the halt state h.
- Otherwise, the input string is **rejected**.
- There are two ways to reject an input string:
    - The machine stops by entering a state other than h from which there is no move;
    - The machine runs forever.
- The **language** of M, denoted $L(M)$, is the set of all input strings accepted by M.
- Note that Turing machines can solve all the problems that PDAs can solve because a stack can be maintained on some portion of the tape.

Example

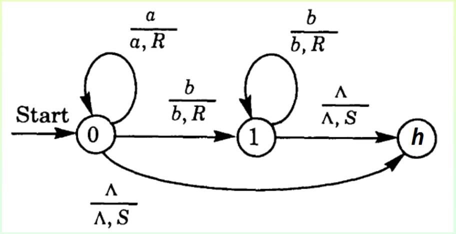
- Construct a Turing machine for the regular language $\{a^n b^m : n, m \in \mathbb{N}\}$.

  The machine starts at the start state (0) scanning the first symbol of the input string.

  It continues scanning the tape to the right, looking for the empty symbol (end of the input string), making sure that no a's are scanned after any occurrence of b.

- The instructions are

  $(0, \Lambda, \Lambda, S, h)$ (Accept Λ or only a's)
  $(0, a, a, R, 0)$ (Scan a's)
  $(0, b, b, R, 1)$
  $(1, b, b, R, 1)$ (Scan b's)
  $(1, \Lambda, \Lambda, S, h)$.



Example

- Construct a Turing machine for the non-context-free language $\{a^n b^n c^n : n \geq 0\}$.

  First we devise an algorithm for the task:
  - If the current cell is empty, then halt with success;
  - If the current cell contains an a, then write an X in the cell and scan right; looking for a corresponding b to the right of any a's;
    Replace the b by Y;
  - Then continue scanning to the right, looking for a corresponding c to the right of any b's;
    Replace the c by Z;
  - Now scan left to the X and see whether there is an a to its right;
    - If so, then start the process again.
    - If there are no a's, then scan right, making sure there are no b's and no c's.

Example (Instructions)

- If Λ is found, then halt.

  If a is found, then write X and scan right.

  If Y is found, then scan over Y's and Z's to find the right end of the string.

$(0, a, X, R, 1)$	Replace a by X and scan right
$(0, Y, Y, R, 0)$	Scan right
$(0, Z, Z, R, 0)$	Go make the final check
$(0, \Lambda, \Lambda, S, h)$	Success

- Scan right, looking for b.

  If found, replace it by Y.

$(1, a, a, R, 1)$	Scan right
$(1, b, Y, R, 2)$	Replace b by Y and scan right
$(1, Y, Y, R, 1)$	Scan right

Example (Instructions Cont'd)

- Scan right, looking for c.

  If found, replace it by Z.

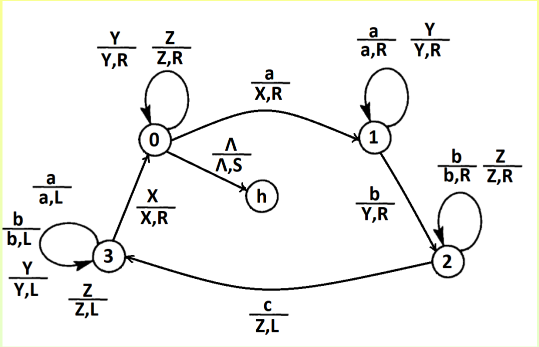
$(2, c, Z, L, 3)$	Replace c by Z and scan left
$(2, b, b, R, 2)$	Scan right
$(2, Z, Z, R, 2)$	Scan right

- Scan left until an X is found.

  Then move right one cell, and repeat the process.

$(3, a, a, L, 3)$	Scan left
$(3, b, b, L, 3)$	Scan left
$(3, X, X, R, 0)$	Found X; Move right one cell
$(\langle 3, Y, Y, L, 3)$	Scan left
$(3, Z, Z, L, 3)$	Scan left

Example (Diagram)



0 : *aabbcc*	3 : *XaYbZc*	2 : *XXYYZc*	0 : *XXYYZZ*
1 : *Xabbcc*	3 : *XaYbZc*	2 : *XXYYZc*	0 : *XXYYZZ*
1 : *Xabbcc*	3 : *XaYbZc*	3 : *XXYYZZ*	0 : *XXYYZZ*
2 : *XaYbcc*	0 : *XaYbZc*	3 : *XXYYZZ*	0 : *XXYYZZ*
2 : *XaYbcc*	1 : *XXYbZc*	3 : *XXYYZZ*	0 : *XXYYZZ*Λ
3 : *XaYbZc*	1 : *XXYbZc*	3 : *XXYYZZ*	*h* : *XXYYZZ*Λ

Equivalence with Other Models

- Some alternative models are the following:
  - Multihead Turing machines;
  - Multitape Turing machines;
  - Nondeterministic Turing machines.

    ⋮
- It turns out that all these models are **equivalent** in the sense that, given a Turing machine of a certain kind, there is a Turing machine of any other kind that accepts the same language as the given one.

Universal Turing Machine

- Up to now, we saw "special purpose" Turing machines built to accept a specific language.
- A **universal Turing machine** U is a Turing machine that can take as input:
  - an arbitrary Turing machine M (a program);
  - an arbitrary input for M (input to the program)

  and then perform the execution of M on its input.
- A universal Turing machine acts like a "general purpose" computer that:
  - stores a program;
  - stores data for the program

  and then executes the program using the data.

Description of a Universal Turing Machine

- Since U can have only a finite number of instructions and a finite alphabet of tape cell symbols, we must represent any Turing machine in terms of the fixed symbols of U.
- We select:
  - A fixed infinite set of states, say $\mathbb{N}$;
  - A fixed infinite set of tape cell symbols, say $L = \{a_i : i \in \mathbb{N}\}$.
- We require that every Turing machine must use states from the set $\mathbb{N}$ and tape cell symbols from L.

  This is easy to do by simply renaming the symbols used in any Turing machine.
- We select a fixed finite alphabet A for the machine U.
- We fix a way to encode, using strings over A:
  - Any Turing machine (i.e., the instructions for any Turing machine);
  - Any input string for a Turing machine.
- Now that we have the two strings over A, one for the Turing machine and one for its input, we can describe the action of machine U.

Description of a Universal Turing Machine

- We describe U as a three-tape Turing machine:
- This does no harm since any k-tape machine can be simulated by a one-tape machine.
- We initialize U by placing on:

Tape 1: The representation for a Turing machine M;
Tape 2: The representation of an input string w.
Tape 3: The start state of M.

- U repeatedly performs the following actions:
  - If the state on tape 3 is the halt state, then halt.
  - Otherwise, get the current state from tape 3 and the current input symbol from tape 2.
  - With this information, find the proper instruction on tape 1.
  - Write the next state at the beginning of tape 3, and then perform the indicated write and move operations on tape 2.

The Church-Turing Thesis

- We say that something, e.g. a language or a problem, is **intuitively computable** if there is a formal description (algorithm) of a process (execution) that accepts the language or decides the problem.
- A formalization of the notion of computation is called a **model** of computation. Examples of models are:
  - The derivation process based on grammars;
  - The evaluation process based on functions;
  - The state transition process based on machines;
  - The execution process based on programs written in some programming language.
- We saw that there are models that differ in power, such as finite automata, pushdown automata and Turing machines.
- The **Church-Turing Thesis** postulates that there exists a most powerful model:

  Anything that is intuitively computable can be computed by a Turing machine.

Subsection 2

Computability

Effective Enumerations

- An **effective enumeration** of a set is a listing of its elements by an algorithm.
- There is no requirement that:
  - the elements be listed in any particular order, or
  - the elements not be repeated.
- Our goal is to effectively enumerate all instances of a particular computational model.
- Since any instance of a computational model can be thought of as a string of symbols, we associate each natural number with an appropriate string of symbols.

Effective Enumeration of Models

- We apply the following steps:
  - First, let $b(n)$ denote the binary representation of a natural number n.
  - Next, partition $b(n)$ into seven-bit blocks by starting at the right end of the string.

    If necessary, we can add leading zeros to the left end of the string to make sure that all blocks contain seven bits.
  - Let $a(b(n))$ denote the string of ASCII characters represented by the partitioning of $b(n)$ into seven-bit blocks.
    - If the string $a(b(n))$ represents a syntactically correct definition for an instance of the model, then we use it as the nth instance of the model.
    - If $a(b(n))$ does not make any sense, we set the nth instance of the model to be some specifically chosen instance.

Example: Enumeration of Turing Machines

- For each natural number n, let T_n denote the Turing machine defined as follows:
  - If $a(b(n))$ represents a string of valid Turing machine instructions, then let $T_n = a(b(n))$.
  - Otherwise, let T_n be the simple machine $T_n =$ "$(0, a, a, S, h)$".

  So we can effectively enumerate all the Turing machines

$$T_0, T_1, T_2, T_3, \ldots.$$

Example: Enumeration of Computable Functions

- We can effectively enumerate all Turing machines.
- By the Church-Turing Thesis, we can effectively enumerate all possible computable functions (including partially defined ones).
- We assume that we have an effective enumeration of all the computable functions as follows:

$$f_0, f_1, f_2, f_3, \ldots$$

- We can also effectively enumerate all possible computable functions with a single argument.

Decision Problems and Decidability

- A **decision problem** is a problem that asks a question that has a YES or NO answer.
- A decision problem is **decidable** if there is an algorithm that, given any arbitrary instance of the problem, halts with the correct answer.
- If no such algorithm exists, then the problem is **undecidable**.
- A decision problem is **partially decidable** if there is an algorithm that, given any arbitrary instance of the problem:
    - Answers YES for those instances of the problem that have YES answers;
    - May run forever for those instances of the problem whose answers are NO.

The Halting Problem

- **The Halting Problem**:
  Is there an algorithm that can decide, given an arbitrary program and an arbitrary input, whether the execution of the program halts on the given input?
- The Halting Problem is undecidable.
  Assume, to the contrary the the Halting Problem is decidable.
  Consider an effective enumeration of all computable (partial) functions of a single argument $f_0, f_1, f_2, f_3, \ldots$.
  Define the halt function $h(x)$ as follows:

  $$h(x) = \text{if } f_x(x) \text{ halts then 1 else 0.}$$

  By our hypothesis, $h(x)$ is computable.
  Define the function $g(x)$ as follows:

  $$g(x) = \text{if } h(x) = 1 \text{ then loop forever else 0.}$$

  Since $h(x)$ is computable, then $g(x)$ is also computable.

The Halting Problem (Cont'd)

- We defined

$$\begin{array}{rcl} h(x) & = & \text{if } f_x(x) \text{ halts then 1 else 0} \\ g(x) & = & \text{if } h(x) = 1 \text{ then loop forever else 0} \end{array}$$

and reasoned that $g(x)$ is computable.
Since $f_0, f_1, f_2, f_3, \ldots$ is an effective enumeration of all computable functions, there exists $n \in \mathbb{N}$, such that $g(x) = f_n(x)$.
Now we compute $f_n(n)$:

$$\begin{array}{rcl} f_n(n) & = & g(n) \\ & = & \text{if } h(n) = 1 \text{ then loop forever else 0} \\ & = & \text{if } f_n(n) \text{ halts then loop forever else 0.} \end{array}$$

This gives an immediate contradiction:
  - If $f_n(n)$ halts, then $f_n(n)$ loops forever;
  - If $f_n(n)$ does not halt, then $f_n(n)$ halts with value 0.

We conclude that the Halting Problem is undecidable.

Total Functions

- A **total function** is one that is defined (or produces an output) for any arbitrary input.
- There is no effective enumeration of all total computable functions.

  We prove the statement for the case of natural number functions having a single variable.

  Suppose, by way of contradiction, that we have an effective enumeration of all the total computable functions:

  $$h_0, h_1, h_2, h_3, \ldots.$$

  Now define a new function $H(n)$ by

  $$H(n) = h_n(n) + 1.$$

  Since each h_n is total, it follows that H is total.

Total Functions

- We defined the total function

$$H(n) = h_n(n) + 1.$$

Since $h_0, h_1, h_2, h_3, \ldots$ is an effective enumeration, there is an algorithm that, given n, produces h_n.

Therefore, $h_n(n) + 1$ is computable.

Thus, H is a total computable function.

It follows that there exists a $k \in \mathbb{N}$, such that $H(n) = h_k(n)$.

But then $h_k(k) = H(k) = h_k(k) + 1$.

This gives a contradiction, since $h_k(k)$ is a natural number.

The Total Problem

- **The Total Problem**:

  Is there an algorithm to tell whether an arbitrary computable function is total?

- The Total Problem is undecidable.

  Suppose, to the contrary, that the Total Problem is decidable.

  Consider an effective enumeration of all computable functions

  $$f_0, f_1, f_2, f_3, \ldots.$$

  By hypothesis, there exists a computable function computing the condition, "f_x is a total function".

  We obtain a contradiction by exhibiting an effective enumeration of all total computable functions, which is impossible.

The Total Problem (Cont'd)

- We construct an effective enumeration of all total computable functions as follows:

  We define the function g as follows:

$$\begin{array}{rcl} g(0) & = & \text{the smallest index } k \text{ such that } f_k \text{ is total} \\ g(n+1) & = & \text{the smallest index } k > g(n) \text{ such that } f_k \text{ is total} \end{array}$$

  Since the condition "f_k is total" is computable, it follows that g is computable.

  Therefore, we have the following effective enumeration of all the total computable functions.

$$f_{g(0)}, f_{g(1)}, f_{g(2)}, f_{g(3)}, \ldots.$$

Other Undecidable Problems

- **The Equivalence Problem**: Does there exist an algorithm that can decide whether two arbitrary computable functions produce the same output?
- **Posts Correspondence Problem**: Given a finite sequence of pairs of strings $(s_1, t_1), \ldots, (s_n, t_n)$, is there a sequence of indices $i_1, \ldots, i_k$, with repetitions allowed, such that $s_{i_1} \ldots s_{i_k} = t_{i_1} \ldots t_{i_k}$?
- **Hilbert's Tenth Problem**: Does a polynomial equation $p(x_1, \ldots, x_n) = 0$ with integer coefficients have a solution consisting of integers?
- **Turing Machine Problems**: Given a Turing machine M:
  - Does M halt when started on the empty tape?
  - Is there an input string for which M halts?
  - Does M halt on every input string?

Partially Decidable Problems

- Whenever we can search for a YES answer to a decision problem and are sure that it takes a finite amount of time, then the problem is partially decidable.
- Example: The halting problem is partially decidable because, for any computable function f_n and any input x, we can evaluate the expression $f_n(x)$.
  - If the evaluation halts with a value, then we output YES.
  - We do not care what happens if $f_n(x)$ is undefined or its evaluation never halts.
- Example: Post's correspondence problem is partially decidable. We can check for a solution by systematically looking at all sequences of length 1, then length 2, and so on.
  - If there is a sequence that gives two matching strings, we eventually find it and output YES.
  - Otherwise, we do not care what happens.
- Example: The Total Problem is not even partially decidable.

Subsection 3

Complexity Classes

The Traveling Salesman Problem

- Recall that a **decision problem** is one that admits a YES or NO answer.
- Even general computational problems can often be rephrased as decision problems without altering their essence.
- Example:
  The Traveling Salesman Problem

  > Find the shortest tour of a set of cities that starts and ends at the same city.

  Traveling Salesman Problem (TSP) (Decision Version)

  > Given a set of cities $\{c_1, \ldots, c_n\}$, a set of distances $d(c_i, c_j) > 0$, for $i \neq j$, and a bound $B > 0$, does there exist a tour of the n cities that starts and ends at the same city, such that the total distance traveled is less than or equal to B?
- We restrict our discussion to decision problems.

Instances and their Length

- An **instance** of a decision problem is a specific example of the input to the problem.
- Example: An instance I of TSP can be represented as follows:

$$\begin{aligned} I \;=\; & \{\{c_1, c_2, c_3, c_4\}, \\ & B = 27, \\ & d(c_1, c_2) = 10,\; d(c_1, c_3) = 5,\; d(c_1, c_4) = 9, \\ & d(c_2, c_3) = 6,\; d(c_2, c_4) = 9,\; d(c_3, c_4) = 3\}. \end{aligned}$$

- The **length** of an instance is an indication of the space required to represent the instance.
- Example: The length of the preceding instance I might be the number of characters that occur between the two braces $\{$ and $\}$.

  Or the length might be some other measure, like the number of bits required to represent the instance as an ASCII string.

Approximation of the Length

- We often approximate the length of an instance.
- Example: An instance I of TSP with n cities contains $\frac{n(n-1)}{2}$ distances and one bounding relation.

  We can assume that each of these entities takes no more than some constant amount of space.

  If c is this constant, then the length of I is no more than

$$c\left[n+1+\frac{n(n-1)}{2}\right]=c\left(\frac{1}{2}n^2+\frac{1}{2}n+1\right)=O(n^2).$$

  So we can assume that the length of I is $O(n^2)$, where n is the number of cities.

Solutions and YES Instances

- Sometimes we want more than just a YES or NO answer to a decision problem.
- A **solution** for an instance of a decision problem is a structure that yields a YES answer to the problem.
- If an instance has a solution, then the instance is called a **YES instance**.
- Otherwise, the instance is a **NO-instance**.
- Example: Consider again

$$\begin{array}{rcl} I & = & \{\{c_1, c_2, c_3, c_4\}, \\ & & B = 27, \\ & & d(c_1, c_2) = 10,\ d(c_1, c_3) = 5,\ d(c_1, c_4) = 9, \\ & & d(c_2, c_3) = 6,\ d(c_2, c_4) = 9,\ d(c_3, c_4) = 3\}. \end{array}$$

The tour (c_1, c_2, c_4, c_3) is a solution for the instance I because its total distance is 27. So I is a YES instance of TSP.

The Class P

- A **deterministic algorithm** is one whose steps during a computation are uniquely determined.
- We say that a deterministic algorithm **solves** a decision problem if, for each instance of the problem, the algorithm halts with the correct answer, YES or NO.
- The class P consists of those decision problems that can be solved by deterministic algorithms with worst case running times bounded by a polynomial in the size of the input.
- More formally, a decision problem is in the class P if:
  - There is a deterministic algorithm A that solves the problem;
  - There is a polynomial p such that

$$W_A(n) = \max\{\text{time}_A(I) : \text{I an input of size } n\} \leq p(n).$$

- The class P consists of those decision problems that can be solved by deterministic algorithms of order $O(p(n))$ for some polynomial p.

Example

- Consider the problem determining whether an item can be found in an n-element list.
- A simple search that compares the item to each element of the list takes at most n comparisons.
- If we assume that the size of the input is n, then the algorithm solves the problem in time $O(n)$.
- Since $p(n) = n$ is a polynomial, the problem is in P.

Tractable and Intractable Problems

- A problem is said to be **tractable** if it is in P
- A problem is **intractable** if it is not in P.
- A problem is intractable if it has a lower bound worst case complexity greater than any polynomial.

Non-Deterministic Algorithms

- A **nondeterministic algorithm** for an instance I of a decision problem has two distinct stages:

Guessing Guess a possible solution S for instance I.

Checking A deterministic algorithm starts up to check whether the guess S from the guessing stage is a solution to instance I.
This checking algorithm will halt with the answer YES if and only if S is a solution of I.
But it may or may not halt if S is not a solution of I.

- In theory:
  - The guess at a possible solution S could be made out of thin air;
  - S could be a structure of infinite length so that the guessing stage would never halt;
  - The checking stage may not even consider S.

The Class NP

- We say that a nondeterministic algorithm **solves a decision problem in polynomial time** if there is a polynomial p such that:
    - For each YES instance I there is a solution S that, when guessed in the guessing stage, will lead the checking stage to halt with YES answer;
    - The time for the checking stage is less than or equal to $p(n)$, where $n = \text{length}(I)$.
- The class NP consists of those decision problems that can be solved by nondeterministic algorithms in polynomial time.

P versus NP

- P is a subset of NP: $\mathrm{P} \subseteq \mathrm{NP}$.

  Suppose that a decision problem π is in P.

  Then, by definition, there is a deterministic algorithm A that solves any instance of π in polynomial time.

  Construct a nondeterministic polynomial time algorithm to solve π as follows:

  - Let I be an instance of π;
  - The guessing stage makes a guess S;
  - The checking stage ignores S and runs algorithm A on I.
    This stage will halt with YES or NO, depending on whether I is a YES instance or not.

  Clearly this nondeterministic algorithm solves π in polynomial time.

  Therefore, π is in NP.

  We conclude $\mathrm{P} \subseteq \mathrm{NP}$.

The $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$ Problem

- We showed that $\mathrm{P} \subseteq \mathrm{NP}$.
- It is not known whether this inclusion is proper, i.e., whether P is a proper subset of NP or whether the two are equal.
- In other words:
  - No one has been able to find an NP problem that is not in P, which would prove that $\mathrm{P} \neq \mathrm{NP}$;
  - No one has been able to prove that all NP problems are in P, which would prove that $\mathrm{P} = \mathrm{NP}$.
- This problem, known as the $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, is one of the foremost open questions of Mathematics and Computer Science.

Example: The Traveling Salesman Problem

- TSP is an NP problem.

  Consider an instance of TSP with n cities, a bound B, and a distance function d.

  Suppose that a guess is made that $(c_1, \ldots, c_n, c_1)$ is a solution.

  The checking stage can check whether

  $$d(c_1, c_2) + \cdots + d(c_{n1}, c_n) + d(c_n, c_1) \leq B.$$

  This check takes $n - 1$ additions and one comparison.

  Assuming that each of these operations takes a constant amount of time, a guess can be checked in time $O(n)$.

  So the checking stage can be done in polynomial time.

  Therefore, TSP is in NP.

Example: TSP (Cont'd)

- It is not known whether TSP is in P.

  It appears that any deterministic algorithm to solve the problem might have to check all possible tours of n cities.

  Since each tour begins and ends at the same city, there are $(n-1)!$ possible tours to check.

  But $(n-1)!$ has higher order than any polynomial in n.

Example: The Clique Problem

- A **clique** is a set of vertices in a graph that are pairwise adjacent (i.e., connected by an edge).
- **Clique Problem** (**CLIQUE**)

  Given graph G with n vertices and a natural number $k \leq n$, decide whether G has a clique with k vertices.

- CLIQUE is in NP.

  Consider a graph G with n vertices and a natural number $k \leq n$.

  Suppose a guess is made that a set of vertices $\{v_1, \ldots, v_k\}$ forms a clique.

  In this case, it takes at most $\frac{k(k-1)}{2}$ comparisons to check whether the k vertices are connected to each other by edges.

  Since $k \leq n$, the comparisons for any instance can be checked in time $O(n^2)$.

Example: The Clique Problem (Cont'd)

- As with TSP, it is not known whether CLIQUE is in P.
- A brute force solution searches through all the subsets of k vertices looking for one consisting of pairwise adjacent vertices.
- There are $\binom{n}{k}$ subsets of k vertices to check.
- Each of those subsets has at most $\frac{k(k-1)}{2}$ possible edges to check.
- So, the total number of comparisons could be as large as

$$\binom{n}{k}\frac{k(k-1)}{2}.$$

- This has exponential order when k is close to $\frac{n}{2}$.

Example: The Hamiltonian Cycle Problem

- A **Hamiltonian cycle** in a graph is a cycle that visits all the vertices of the graph.
- **The Hamiltonian cycle problem (HCP)**

  Given a graph with at least three vertices, does the graph have a Hamiltonian cycle?
- HCP is in NP.

  An instance I of HCP is a graph G with n vertices.

  Suppose a guess is made that a sequence of its vertices $\langle v_1, v_2, \ldots, v_n, v_1\rangle$ forms a Hamiltonian cycle.

  Since G has n vertices it has a maximum of $\frac{n(n-1)}{2}$ edges.

  To see whether the sequence is a cycle, we need to check if there is an edge connecting each of the n adjacent pairs in the sequence.

  So, we must perform $n \times \frac{n(n-1)}{2} = O(n^3)$ comparisons.

Example: The Hamiltonian Cycle Problem (Cont'd)

- As with TSP and CLIQUE, it is not known whether HCP is in P.
- A brute force solution looks for a cycle by examining different sequences of n vertices.
- There are $n!$ such sequences.
- Moreover, $n!$ has higher order than any polynomial in n.

The Class PSPACE

- The class PSPACE is the set of decision problems that can be solved by deterministic algorithms that use no more memory cells than a polynomial in the length of an instance.
- More formally, a problem is in PSPACE if:
  - There is a deterministic algorithm that solves it;
  - There is a polynomial p such that the algorithm uses no more than $p(n)$ memory cells, where n is the length of an instance.

NP versus $\mathrm{PS{\scriptsize PACE}}$

- $\mathrm{NP} \subseteq \mathrm{PSpace}$.

  For any problem π in NP, there is a nondeterministic algorithm A and a polynomial p such that A takes at most $p(n)$ steps to check a solution for a YES instance I of length n.

  Any step of A can access at most a fixed number k of memory cells.

  So A uses at most $kp(n)$ memory cells to check a solution for I.

  Since p is a polynomial, kp is also a polynomial.

  So the checking stage uses polynomial space.

  If S is a solution for I, then the part of S used by the checking stage fits within $kp(n)$ memory cells.

  So we can assume that S is a string of length at most $kp(n)$ over a finite alphabet of symbols - one symbol per memory cell.

NP versus $\mathrm{PS}\textsc{pace}$ (Cont'd)

- Define a deterministic algorithm B to solve π:
    - For an instance of length n, B generates and checks - one at a time - all possible strings of length at most $kp(n)$.
    - The checking is done by the checking stage of A modified to stop after $p(n)$ steps if it has not stopped yet.
    - If a solution is found, then B stops with a YES answer.
    - Otherwise, B stops with a NO answer after generating and checking all possible strings of length at most $kp(n)$.

  The generating stage uses polynomial space because it generates a string of length at most $kp(n)$.

  The checking stage uses polynomial space because it is the checking stage of A modified by adding a clock.

  The finite alphabet and other local variables use a constant amount of space.

  So B uses polynomial space.

  Therefore, $\mathrm{NP} \subseteq \mathrm{PS}\textsc{pace}$.

Relations Between Classes

- We have the following relations between complexity classes:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PS{\scriptsize PACE}}.$$

- We mentioned that it is not known whether

$$\mathrm{P} \stackrel{?}{=} \mathrm{NP}.$$

- Similarly, it is not known whether

$$\mathrm{NP} \stackrel{?}{=} \mathrm{PS{\scriptsize PACE}}.$$

Quantified Boolean Formulae

- A **quantified Boolean formula** is a logical expression of the form

$$Q_1 x_1 Q_2 x_2 \cdots Q_n x_n E, \quad n \geq 1,$$

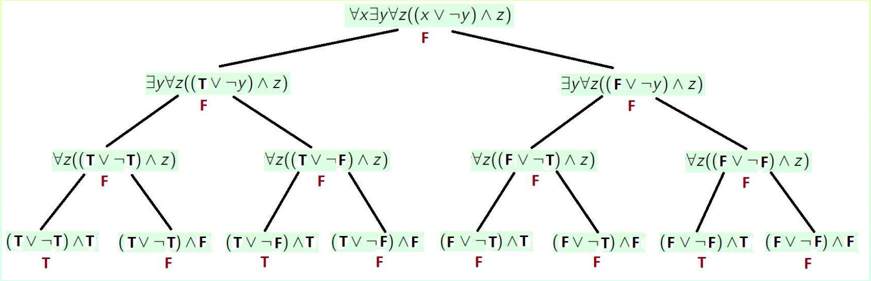
  where:
  - each Q_i is either $\forall$ or $\exists$;
  - each x_i is distinct;
  - E is a formula of the propositional calculus that is restricted to using the variables $x_1, \ldots, x_n$; the operations $\neg, \wedge$ and $\vee$ and parentheses.
- Example: The following formulas are quantified Boolean formulas:

$$\begin{array}{l} \exists x\ x \\ \forall x \exists y(\neg x \vee y) \\ \forall x \exists y \forall z((x \vee \neg y) \wedge z) \end{array}$$

Truth Value of a Quantified Boolean Formula

- A quantified Boolean formula is **true** if its value is T over the domain $\{T, F\}$. Otherwise, it is **false**.
- Example: We have the following truth values:

Formula	Truth Value
$\exists x\ x$	T
$\forall x \exists y (\neg x \vee y)$	T
$\forall x \exists y \forall z ((x \vee \neg y) \wedge z)$	F



$\forall x \exists y \forall z ((x \vee \neg y) \wedge z)$
F
$\exists y \forall z ((\mathbf{T} \vee \neg y) \wedge z)$
F
$\exists y \forall z ((\mathbf{F} \vee \neg y) \wedge z)$
F
$\forall z ((\mathbf{T} \vee \neg \mathbf{T}) \wedge z)$
F
$\forall z ((\mathbf{T} \vee \neg \mathbf{F}) \wedge z)$
F
$\forall z ((\mathbf{F} \vee \neg \mathbf{T}) \wedge z)$
F
$\forall z ((\mathbf{F} \vee \neg \mathbf{F}) \wedge z)$
F
$(\mathbf{T} \vee \neg \mathbf{T}) \wedge \mathbf{T}$
T
$(\mathbf{T} \vee \neg \mathbf{T}) \wedge \mathbf{F}$
F
$(\mathbf{T} \vee \neg \mathbf{F}) \wedge \mathbf{T}$
T
$(\mathbf{T} \vee \neg \mathbf{F}) \wedge \mathbf{F}$
F
$(\mathbf{F} \vee \neg \mathbf{T}) \wedge \mathbf{T}$
F
$(\mathbf{F} \vee \neg \mathbf{T}) \wedge \mathbf{F}$
F
$(\mathbf{F} \vee \neg \mathbf{F}) \wedge \mathbf{T}$
T
$(\mathbf{F} \vee \neg \mathbf{F}) \wedge \mathbf{F}$
F

The Function val

- We define the function val that computes the truth value of a quantified Boolean formula by induction:

Basis: $\text{val}(T) = T$ and $\text{val}(F) = F$;

Induction: We have multiple cases depending on the main connective:

  - $\text{val}(\neg A) = \neg\text{val}(A)$;
  - $\text{val}(A \wedge B) = \text{val}(A) \wedge \text{val}(B)$;
  - $\text{val}(A \vee B) = \text{val}(A) \vee \text{val}(B)$;
  - $\text{val}(\forall x E) = \text{val}(E(x/T)) \wedge \text{val}(E(x/F))$;
  - $\text{val}(\exists x E) = \text{val}(E(x/T)) \vee \text{val}(E(x/F))$.

- Example: We compute $\text{val}(\forall x \exists y(\neg x \vee y))$.

$$\begin{aligned}
&\text{val}(\forall x \exists y(\neg x \vee y)) \\
&= \text{val}(\exists y(\neg T \vee y)) \wedge \text{val}(\exists y(\neg F \vee y)) \\
&= (\text{val}(\neg T \vee T) \vee \text{val}(\neg T \vee F)) \wedge (\text{val}(\neg F \vee T) \vee \text{val}(\neg F \vee F)) \\
&= ((\neg T \vee T) \vee (\neg T \vee F)) \wedge ((\neg F \vee T) \vee (\neg F \vee F)) \\
&= ((F \vee T) \vee (F \vee F)) \wedge ((T \vee T) \vee (T \vee F)) \\
&= (T \vee F) \wedge (T \vee T) = T \wedge T = T.
\end{aligned}$$

The Quantified Boolean Formula Problem

- **The Quantified Boolean Formula Problem (QBF)**
  Given a quantified Boolean formula, is it true?
- QBF is in $\mathrm{PS}\textsc{pace}$.
- The number of operations ($\neg$, $\wedge$, $\vee$, $\forall$ and $\exists$) and distinct variables in a formula is at most the length k of the formula.

  Each operation requires at most two recursive calls in its evaluation.

  So the time required by the algorithm is $O(2^k)$.
- The depth of recursion is proportional to the number of connectives, i.e., $O(k)$.

  The space required for each recursive call is $O(k)$, because space can be reused.

  So the space used by the algorithm is $O(k^2)$.

  Therefore, QBF is in $\mathrm{PS}\textsc{pace}$.
- It is not known whether QBF is in NP.

Subsection 4

Reductions and Completeness

Polynomial Time Reductions

- A problem A is **polynomial time reducible** to a problem B if there is a polynomial time computable function f that maps instances of A to instances of B such that:

$$I \text{ is a YES instance of } A \text{ iff } f(I) \text{ is a YES instance of } B.$$

- This property of f says two things:
  - YES instances of A get mapped to YES instances of B;
  - NO instances of A get mapped to NO instances of B.

Example

- **The Hamiltonian Cycle Problem (HCP)**

  Given a graph with n vertices, $n \geq 3$, does it have a Hamiltonian, cycle?
- **Traveling Salesman Problem (TSP)**

  Given a set of cities $\{c_1, \ldots, c_n\}$, a set of distances $d(c_i, c_j) > 0$, for $i \neq j$, and a bound $B > 0$, does there exist a tour of the n cities that starts and ends at the same city, such that the total distance traveled is less than or equal to B?
- Let I be an instance of HCP.

  For each pair of distinct vertices v and w in the graph, we set

  $$d(v, w) = \text{if } vw \text{ is an edge then } 1 \text{ else } 2.$$

  Define $f(I)$ to be the instance of TSP where:
  - The cities are the vertices in the graph;
  - The bound is the number of vertices $B = n$;
  - The distance between cities is given by d.

Example (Cont'd)

- **Time needed for the reduction from I to $f(I)$**:

  If the graph has n vertices, then there are $\frac{n(n-1)}{2}$ values of the form $d(v,w)$ that must be computed.

  For each value of d, there are at most $\frac{n(n-1)}{2}$ edges to check.

  So f is polynomial time computable because the number of comparisons to compute $f(I)$ is $O(n^4)$.
- **Condition for the reduction**:

  An instance I is a YES instance of HCP

  if and only if the sequence of vertices forms a cycle of length n

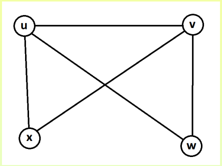
  if and only if the sequence is a tour of the n cities (which are the vertices)

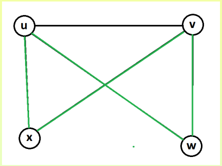
  if and only if $f(I)$ is a YES instance of TSP.
- So we have a polynomial time reduction from HCP to TSP.

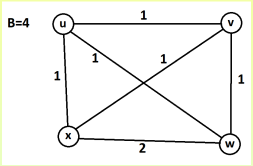
Example (Illustration)

- Consider the following instance of HCP:

$$I = \{\{u, v, w, x\}, \{uv, uw, ux, vw, vx\}\}.$$







- It is a YES instance, since (u, x, v, w, u) is a Hamiltonian cycle.
- Applying f, we obtain the following instance $f(I)$ of TSP:

$$\begin{aligned} f(I) \quad = \quad & \{\{u, v, w, x\}, \\ & B = 4, \\ & d(u, v) = 1,\ d(u, w) = 1,\ d(u, x) = 1, \\ & d(v, w) = 1,\ d(v, x) = 1,\ d(w, x) = 2\}. \end{aligned}$$

- $f(I)$ is a YES instance of TSP since tour (u, x, v, w, u) has length 4.

Polynomial Time Reductions and Efficient Algorithms

- Suppose the following:
  - There is a polynomial time reduction from problem A to problem B via the polynomial time computable function f;
  - There is a polynomial time algorithm M to solve B.

  Then there exists a polynomial time algorithm that solves A.
- The algorithm can be described as follows:
  1. Given an arbitrary instance I of problem A.
  2. Construct the instance $f(I)$ of problem B.
  3. Run algorithm M on the instance $f(I)$.
  4. If M finds that $f(I)$ is a YES instance of B, then I is a YES instance of A.
  5. If M finds that $f(I)$ is a NO instance of B, then I is a NO instance of A.
- In short:

  Reducing, via a polynomial time reduction, a problem A to an efficiently solvable problem shows that A is also efficiently solvable.

NP-Hard and NP-Complete Problems

- A decision problem is said to be NP-**hard** if every decision problem in NP is polynomial time reducible to it.
- A decision problem is said to be NP-**complete** if:
  - It is in the class NP;
  - It is NP-hard.
- **Significance of NP-Completeness**: Suppose we find an NP-complete problem B (i.e., a problem in NP such that every other problem in NP is polynomial time reduced to B).

  If we ever found a deterministic polynomial time algorithm for B, then, by the previous slide, every problem in NP would have a deterministic polynomial time algorithm.
- Success in this quest would have dramatic consequences:
  - It would provide us with efficient solutions to all NP problems among which are many well known ones;
  - It would also solve the $\mathrm{P} \overset{?}{=} \mathrm{NP}$ question to the affirmative.

Conjunctive Normal Form

- A **literal** is a variable or the negation of a variable.
- Example: x and $\neg z$ are literals, but $\neg(x \wedge y)$ is not a literal.
- A **clause** is a disjunction of one or more literals.
- Example: $x \vee y \vee \neg z$ and $\neg x$ are clauses.
  $\neg(x \wedge y)$ is not a clause.
- A Boolean formula, built up from variables $x, y, z, x_1, x_2, \ldots$ and the connectives $\neg, \wedge$ and $\vee$ (negation, conjunction (and) and disjunction (or), respectively) is in **conjunctive normal form** (**CNF**) if it is written as a conjunction of clauses (conjunction of disjunctions).
- Example: The formulae $(x \vee y \vee \neg z) \wedge (x \vee z)$ and $(x \vee y) \wedge (x \vee \neg y) \wedge \neg x$ are in conjunctive normal form.
  The formula $\neg(x \wedge \neg y)$ is not in CNF.
- A formula is said to be in **3-conjunctive normal form** (**3CNF**) if it is in CNF and each clause is a disjunction of at most three literals.

CNF-Satisfiability Problem

- A Boolean formula φ is **satisfiable** if there exists an assignment of truth values, T or F, to its variables, such that the value of φ is T.
- Example: Consider the formula $(x \vee y \vee \neg z) \wedge (x \vee z)$.

  Let $\mathrm{val}(x) = T$, $\mathrm{val}(y) = F$ and $\mathrm{val}(z) = F$.

  Then we have

  $$\begin{array}{l} \mathrm{val}((x \vee y \vee \neg z) \wedge (x \vee z)) \\ = (T \vee F \vee \neg F) \wedge (T \vee F) = T \wedge T = T. \end{array}$$

  Therefore, the formula is satisfiable.
- Example: Consider the formula $(x \vee y) \wedge (x \vee \neg y) \wedge \neg x$.

  By constructing the truth table, we can see that this formula is not satisfiable (is **unsatisfiable**) since all assignments of truth values to its variables result in the formula assuming the value F.
- **CNF-Satisfiability Problem** (**SAT**)

  Given a Boolean formula in CNF, is it satisfiable?

SAT is in NP

- SAT is in NP.

  Let the *length* of a formula be the total number of literals that appear in it. If n is the length of a formula, then the number of distinct variables in the formula is at most n.

  E.g., the length of $(x \vee y \vee \neg z) \wedge (x \vee z)$ is 5, and it has 3 variables.

  - The guessing stage of a nondeterministic algorithm can produce some assignment of truth values for the variables of the formula.
  - The checking stage must check to see whether each clause of the formula is true for the guessed assignment.

    This involves checking that at least one literal in the clause takes the value true. Since there are n literals in the formula, there are at most n literals to check.

    So the checking stage can be done in $O(n)$ time.

  Therefore, SAT is in NP.

Cook's Theorem

- Cook proved that SAT is NP-complete by showing that any NP problem can be polynomial time reduced to SAT:

Cook's Theorem

The CNF-satisfiability problem (SAT) is NP-complete.

Proving NP-Completeness

- Once we have an NP-complete problem (e.g., SAT), to show that some other problem A is NP-complete, we only have to show that:
  - A is in NP;
  - some known NP-complete problem (e.g., SAT) is polynomial time reducible to A.
- **Algorithm to Show NP-Completeness**
  - Let A be an NP problem;
  - Find, if possible, an NP-complete problem B and a polynomial time reduction from B to A;
  - Then A is also NP-complete.
- The reason this algorithm works is:
  - By B's NP-completeness, any problem π in NP is reducible to B.
  - But B is reducible to A;
  - So, any problem π in NP is reducible to A;
  - As A is also in NP, it is NP-complete.

3-Satisfiability

- The 3-satisfiability problem (3-SAT) is NP-complete.

  First, 3-SAT is in NP because it is just a restricted form of SAT, which we know is in NP.

  To show that 3-SAT is NP-complete, we show that SAT can be polynomial time-reduced to it.

  The basic idea is to transform each clause that has four or more literals into a conjunctive normal form where each clause has three literals with the property that, for some assignment of truth values to variables, the original clause is true if and only if the replacement conjunctive normal form is true.

3-Satisfiability (Reduction)

- Suppose we have the following clause that contains k literals, where $k \geq 4$: $(\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$.
  We transform it into the following conjunctive normal form, where $x_1, x_2, \ldots, x_{k-3}$ are new variables (i.e., not appearing in the original formula):
  $$\begin{aligned}(\ell_1 \vee \ell_2 \vee x_1) \wedge (\ell_3 \vee \neg x_1 \vee x_2) \wedge (\ell_4 \vee \neg x_2 \vee x_3) \wedge \cdots \\ \wedge (\ell_{k-2} \vee \neg x_{k-4} \vee x_{k-3}) \wedge (\ell_{k-1} \vee \ell_k \vee \neg x_{k-3}).\end{aligned}$$
  This transformation can be applied to each clause (containing four or more literals) of a conjunctive normal form, resulting in a conjunctive normal form where each clause has three or fewer literals.
- Example: The clause $(u \vee \neg w \vee x \vee \neg y \vee z)$ is transformed into
  $$(u \vee \neg w \vee x_1) \wedge (x \vee \neg x_1 \vee x_2) \wedge (\neg y \vee z \vee \neg x_2),$$
  where x_1 and x_2 are new variables.

3-Satisfiability (Why It Works)

- We must show that there is some assignment to the new variables such that the original clause is true (i.e., one of its literals is true) if and only if the new expression is true.
  - If $\ell_i = T$, then set x_j to T, for $j \leq i-2$ and set x_j to F, for $j > i-2$. This will make the new expression true.

    $$(\ell_1 \vee \ell_2 \vee x_1) \wedge (\ell_3 \vee \neg x_1 \vee x_2) \wedge (\ell_4 \vee \neg x_2 \vee x_3) \wedge (\ell_5 \vee \ell_6 \vee \neg x_3)$$

  - Conversely, suppose there is some truth assignment to the variables x_j, that makes the new expression true.
    Then some literal in the original fundamental disjunction must be true. Otherwise, we can see that both x_{k-3} and $\neg x_{k-3}$ must be true, which is a contradiction.

    $$(\ell_1 \vee \ell_2 \vee x_1) \wedge (\ell_3 \vee \neg x_1 \vee x_2) \wedge (\ell_4 \vee \neg x_2 \vee x_3) \wedge (\ell_5 \vee \ell_6 \vee \neg x_3)$$

    Since at least one of the ℓ_i's must be true, the original clause must be true.

3-Satisfiability (Time)

- We need to show that the transformation can be done in polynomial time.

  A straightforward algorithm to accomplish the transformation applies the definition to each clause that contains four or more literals.

  If an input formula has length n (i.e., n literals), then there are at most $\frac{n}{4}$ clauses of length four or more.

  Each of these clauses is transformed into a conjunctive normal form containing at most $3(n-2)$ literals.

  Therefore, the algorithm constructs at most $\frac{3n(n-2)}{4}$ literals.

  Since each new literal can be constructed in a constant amount of time, the algorithm will run in time $O(n^2)$.

  Therefore, SAT can be polynomial time-reduced to 3-SAT.

The CLIQUE Problem (Reduction)

- The Clique Problem (CLIQUE) is NP-complete.

  The usual way to show that CLIQUE is NP-complete is to show that 3-SAT is polynomial time reducible to CLIQUE.

  Let I be an instance of 3-SAT that consists of a formula of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_k$, where each C_i is a clause with at most 3 literals.

  Construct an instance $f(I)$ of CLIQUE that consists of a graph and the number k, where the vertices of the graph are all pairs of the form (ℓ, C), where ℓ is a literal in clause C.

  Construct an edge between two vertices (ℓ_i, C_i) and (ℓ_j, C_j) if $C_i \neq C_j$ and the literals ℓ_i, and ℓ_j are not negations of each other.

The CLIQUE Problem (Why it Works)

- Now we show that an instance I of 3-SAT is a YES instance iff the corresponding instance $f(I)$ of CLIQUE is a YES instance:

  I is a YES instance of 3-SAT
  iff $C_1 \wedge C_2 \wedge \cdots \wedge C_k$ is satisfiable
  iff for each i there is a literal ℓ_i in C_i
  that is assigned the truth value T
  iff for each $i \neq j$, the literals ℓ_i and ℓ_j
  are not negations of each other
  iff for each $i \neq j$, there is an edge connecting (ℓ_i, C_i) to (ℓ_j, C_j)
  iff the vertices (ℓ_i, C_i) form a k-clique
  iff $f(I)$ is a YES instance of CLIQUE.

The CLIQUE Problem (Time)

- Because the formula in I has k clauses and each clause has at most $3k$ literals, it follows that $f(I)$ constructs a graph with at most $3k$ vertices and at most $\frac{3k(3k1)}{2}$ edges.

  So, $f(I)$ can be constructed in time $O(k^2)$.

  Therefore, f is polynomial time-computable.

  We conclude that 3-SAT is polynomial time reducible to CLIQUE.

  Since:

  - CLIQUE is in NP;
  - the NP-complete problem 3-SAT is polynomial time reducible to CLIQUE,

  it follows that CLIQUE is also NP-complete.